UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI

Corso di Laurea Magistrale in Informatica

VISUALIZATION OF UV-MAPS FOR POLYGONAL MESHES AS A 3D MORPHING

Relatore: Prof. Marco Tarini

Tesi di Laurea di:
Giorgio Sepe
Matr. Nr. 02786A

ANNO ACCADEMICO 2023-2024

To my family
and to myself

Abstract

In Computer Graphics, 3D *uv*-mapped meshes are the most common format to describe detailed 3D objects, and are ubiquitously used in contexts such as videogames, virtual reality, scientific visualization, etc. They consist of a polygonized approximation of the surface enriched with a *uv*-map, which describes a mapping of the surface into a planar domain. The construction of a good *uv*-map for a given mesh is an important and difficult task in the creation of these datasets, performed either by trained digital artists or by automatic Geometry Processing algorithms. The objective of this thesis is the design and development of a tool aimed at providing an intuitive data visualization of this mapping, given one instance of a *uv*-mapped mesh (regardless of its origin). This tool consists in automatically creating and displaying (in real time) the mapping in the form of a physically plausible morphing [1] between the original 3D shape and its 2D counterpart. The intended uses range from a visual qualitative assessment of the quality of a given *uv*-map (in addition to the many numerical assessments commonly employed in literature), to illustration purpose, including in didactic contexts.

In Computer Grafica, l'utilizzo di mesh 3D dotate di *uv*-map è il modo più comune per descrivere i dettagli visivi sugli oggetti 3D e sono utilizzate ovunque in contesti come videogiochi, realtà virtuale, visualizzazione scientifica, ecc. Esse consistono in una superficie poligonale approssimata arricchita da una mappatura *uv*, che descrive una corrispondenza tra la superficie della mesh 3D e un dominio planare. La creazione di una buona *uv*-map per una determinata mesh è un compito importante, svolto sia da artisti digitali specializzati che da algoritmi automatici di Elaborazione Geometrica. L'obiettivo di questa tesi è la progettazione e lo sviluppo di uno strumento volto a fornire una visualizzazione intuitiva dei dati di questa mappatura, data un'istanza di una mesh dotata di *uv*-map (indipendentemente dalla sua origine). Questo strumento consiste nella creazione e visualizzazione automatica (in tempo reale) della mappatura sotto forma di un morphing fisicamente plausibile [1] tra la forma 3D originale e la sua controparte 2D. Gli usi previsti spaziano da una valutazione qualitativa di una determinata *uv*-map (in aggiunta alle numerose valutazioni numeriche comunemente impiegate in letteratura) a scopi illustrativi, compresi contesti didattici.

Acknowledgments

I would like to express my deepest gratitude to my professor Marco Tarini, for his invaluable guidance, support, and encouragement throughout the entire process of this work. Thank you for the countless hours spent with me in the lab, not only for providing technical insight but also for sharing your passion for computer graphics and engaging in fun, lighthearted activities that made the journey all the more enjoyable.
To my family, thank you for believing in me and always supporting my choices, despite the physical distance between us. To my dad, your presence remains in my heart, constantly motivating me to keep moving forward.
To my friends, thank you for the moments of joy that have supported me along the way.
To Sara, thank you for sharing almost every day with me over the last few years. I eagerly anticipate all the beautiful adventures that await us in the future together.

Giorgio Sepe

Contents

Abstract **V**

Acknowledgments **VII**

1 Introduction **1**
1.1 Context 1
1.2 Challenges 1
1.3 Our Proposal 2
1.4 Objective 2
1.5 Structure of this Work 2

2 Background **5**
2.1 3D Meshes 5
2.1.1 Introduction to 3D Meshes 5
2.1.2 Components of a 3D Mesh 5
2.1.3 Mesh Construction and Representation 6
2.1.4 Importance of Meshes in 3D Graphics 6
2.1.5 *uv*-mapping and Its Role in Meshes 7
2.1.6 Challenges in Mesh and *uv*-map Construction 7
2.1.7 Summary 8
2.2 *uv*-mapping 8
2.2.1 The *uv*-mapping Process 8
2.2.2 Applications of *uv*-mapping 9
2.2.3 Challenges 10
2.2.4 Methods and Techniques 10
2.3 Dual Quaternion Skinning 11
2.3.1 Dual Quaternion 12
2.3.2 Advantages 12
2.3.3 Applications 13
2.3.4 Challenges 13

3 State of the Art **15**
3.1 Automatic uv-map Generation 15
3.1.1 Challenges in *uv*-map Generation 15
3.1.2 Early Approaches to *uv*-mapping 16
3.1.3 Least-Squares Conformal Mapping (LSCM) 16
3.1.4 ABF++: Angle-Based Flattening 16
3.1.5 Spectral Conformal Parameterization 17
3.1.6 Recent Advances in Machine Learning for *uv*-mapping . . 17
3.2 Existing *uv*-map Visualization Tools 17
3.2.1 Real-Time UV Overlay 18
3.2.2 Distortion Visualization 18
3.2.3 Seam Visualization 18
3.2.4 Texel Density Visualization 19
3.2.5 UV Island Visualization and Management 19
3.2.6 Symmetry and Mirroring Visualization 19
3.2.7 UV Space Utilization Visualization 20
3.2.8 Multi-Channel UV Visualization 20
3.2.9 Existing Tools for uv-mapping and Visualization 20
3.2.10 Our Approach 22
3.3 3D Morphing Techniques 23
3.3.1 Local-Global Optimization Framework 25
3.3.2 Visual Assessment of of Functional Maps 25
3.3.3 Our Approach 26

4 Methodology **27**
4.1 Overview 28
4.2 Morphing 29
4.2.1 Linear Morphing 29
4.2.2 Dual-Quaternion Based Morphing 30
4.2.3 Initial Transformation Optimization 34
4.3 Surface Continuity Enforcement 35
4.3.1 Local Gluing 36
4.3.2 Global Isometry Maximization 36
4.3.3 Iterative optimization 38
4.4 Interpolation Timing Differentiation 38
4.5 Baking The Animations 39

5 Implementation **41**
5.1 Used Tools 41
5.1.1 C++ and Javascript Interaction 42
5.1.2 Rendering with Three.js 42

5.2 Implementation Details 44
5.2.1 Initialization 44
5.2.2 Interpolation Algorithm for Mesh Transformation 45
5.2.3 UI and user interaction 47
5.2.4 Meshes Dataset 47
5.2.5 Libraries 49

6 Results and Conclusions **51**
6.1 Results 51
6.1.1 Performace 55
6.2 Conclusions 55
6.2.1 Future Work 56

Bibliography **58**

Chapter 1

Introduction

1.1 Context

In Computer Graphics, accurately representing 3D objects is crucial in fields such as videogames, virtual reality, animation, and scientific visualization. A common way to define these models is through a mesh, which is a polygon-based approximation of an object's surface, combined with a *uv*-map. *uv*-mapping is a technique that projects a 3D surface onto a 2D plane, creating a link between the object's 3D surface coordinates and a flat image (usually a texture). This method allows artists and algorithms to easily apply textures and other surface details to 3D models. However, creating high-quality *uv*-maps is a notoriously challenging task that plays a key role in producing detailed and realistic 3D objects.

1.2 Challenges

Creating an effective *uv*-map, however, is a highly complex task. Despite its ubiquity in computer graphics, generating *uv*-map that are efficient, low-distortion, and visually appealing remains one of the most difficult aspects of 3D modeling. Digital artists, who rely on manual *uv*-mapping techniques, often spend hours fine-tuning their mappings to ensure textures are applied seamlessly across a 3D object's surface. In contrast, automatic *uv*-mapping tools in geometry processing have improved significantly, but even the most advanced algorithms often fail to produce perfect results. Issues such as texture stretching, seam visibility, and poor space utilization are common, necessitating both technical precision and artistic skill.

1.3 Our Proposal

Despite these challenges in *uv*-map generation, once a *uv*-mapped mesh is produced whether through manual effort or automated algorithms the need arises to evaluate its quality. Common approaches for assessing *uv*-maps focuses on quantitative metrics such as distortion or texture efficiency. These measures, while useful, do not always provide an intuitive understanding of how the 3D surface relates to its 2D projection. In many contexts, particularly for visual artists or in didactic settings, data visualization becomes essential in assessing *uv*-maps, offering a clear, qualitative representation of how the 3D surface transforms into a 2D layout.

This thesis aims to address this gap by designing and developing a tool that provides a visual, intuitive description of the *uv*-mapping process. The core of the proposed tool is the creation of a real-time data visualization tool that morphs between the 3D surface and its 2D *uv*-mapped counterpart. By representing the transition between these two domains, the tool allows users to visually assess the quality of the mapping, enhancing their understanding of the *uv*-mapping process. This approach can serve as a supplement to existing numerical assessments, offering a more direct and illustrative way to evaluate *uv*-maps.

1.4 Objective

The primary objective of this thesis is to design a system capable of visualizing any *uv*-mapped mesh, regardless of its origin, whether manually created by an artist or generated through an automated process. By focusing on the data visualization of the mapping itself, this tool aims to be widely applicable, offering value to professionals in the entertainment industry, researchers in geometry processing, and educators in computer graphics. The ultimate goal is to bridge the gap between quantitative evaluation and visual intuition, providing a tool that can support artists, researchers, and educators in better understanding and improving *uv*-maps.

1.5 Structure of this Work

The following chapters of this thesis will discuss the technical background of 3D meshes, *uv*-mapping and the process of deforming 3D surfaces (in Chapter 2), then will be survey the current state-of-the-art methods in *uv*-map generation, morphing and others existing applications in Chapter 3. Chapter 4 outlines the approach and techniques used to enable the transformation between a 3D mesh and its UV map. Key methods discussed include linear interpolation

and dual-quaternion-based morphing. The chapter also covers surface continuity enforcement and optimization techniques like local-global interpolation (ARAP-like). Chapter 5 covers the technical aspects of building the visualization tool and Chapter 6 presents the qualitative results and suggests future work.

Chapter 2

Background

This chapter provides context for the research topics relevant to the thesis and prepares the reader for the upcoming chapters. Furthermore, we describe the state-of-the-art regarding *uv*-mapping visualization techniques, morphing, automated mesh unwrapping and what are the key features that make a mesh unwrapping good.

2.1 3D Meshes

2.1.1 Introduction to 3D Meshes

In the field of computer graphics, a 3D mesh is a representation of a three-dimensional object, consisting of vertices, edges, and faces. These components work together to approximate the surface of the object in a polygonal form, typically through the use of triangles or quadrilaterals. Meshes serve as the backbone of 3D modeling and are fundamental to the creation, manipulation, and visualization of complex geometric shapes in various applications such as video games, virtual reality, scientific visualization, and animation.

2.1.2 Components of a 3D Mesh

A 3D mesh is defined by several key elements:

Vertices: These are points in 3D space, each defined by a set of coordinates (x, y, z). Vertices serve as the corner points of the mesh and are the basic building blocks from which edges and faces are constructed.

Edges: An edge is a straight line connecting two vertices. Edges define the boundaries of the faces within the mesh and contribute to the overall shape of the 3D object.

Faces: A face is a flat surface bounded by edges, typically a triangle or quadrilateral. The collection of faces defines the visible surface of the mesh. In most cases, triangular faces are preferred due to their mathematical simplicity and stability in rendering.

Normals: Normals are vectors perpendicular to the faces of the mesh, representing the orientation of the surface at each vertex or face. Normals are critical for lighting calculations and rendering, as they influence how light interacts with the surface.

Texture Coordinates (*uv*-mapping): Texture coordinates, or *uv*-mapping, are a set of 2D coordinates that map each vertex of the 3D mesh to a corresponding point on a 2D texture. This process enables the application of detailed textures to the surface of the 3D object, enhancing visual realism.

2.1.3 Mesh Construction and Representation

Meshes are typically constructed through various methods, including manual modeling, procedural generation, or 3D scanning. In manual modeling, artists use specialized software to create and manipulate meshes, often starting from simple primitives (e.g., cubes, spheres) and refining them into detailed objects. Procedural generation involves the use of algorithms to automatically generate mesh structures based on predefined rules or parameters. 3D scanning, on the other hand, captures real-world objects and converts them into digital mesh representations.

The representation of a mesh in computer memory involves storing the vertices, edges, and faces in a structured format. Common file formats for storing meshes include OBJ, STL, PLY, and FBX, each with its own advantages depending on the application.

2.1.4 Importance of Meshes in 3D Graphics

Meshes are crucial in 3D graphics due to their ability to represent complex shapes with a high level of detail while remaining computationally efficient. Unlike other representations, such as voxel grids or implicit surfaces, polygonal meshes strike a balance between accuracy and performance, making them the preferred choice for real-time rendering applications like video games and virtual reality.

Moreover, meshes are versatile, supporting various levels of detail (LOD) techniques, where the complexity of the mesh can be adjusted dynamically based on factors such as the distance from the camera or the performance requirements of the rendering system. This adaptability is essential in managing the computational load during real-time rendering, ensuring smooth performance without compromising visual fidelity.

2.1.5 *uv*-mapping and Its Role in Meshes

One of the key challenges in using meshes is the application of textures, which adds visual detail to the object. *uv*-mapping is the process by which a 2D image (texture) is mapped onto the 3D surface of the mesh. The term "UV" refers to the 2D coordinate system used in this mapping, where "U" and "V" denote the axes of the texture space, analogous to the "X" and "Y" axes in Cartesian coordinates.

The creation of a *uv*-map involves "unwrapping" the 3D mesh into a 2D plane, similar to peeling the surface off an orange and laying it flat. This process is non-trivial and requires careful consideration to minimize distortions, seams, and overlaps. A well-constructed *uv*-map ensures that the texture is applied smoothly and accurately across the surface of the mesh, preserving the visual integrity of the model.

uv-mapping is especially important in contexts where high visual quality is required, such as in character modeling, where skin textures, clothing, and other surface details need to align perfectly with the underlying geometry. The effectiveness of *uv*-mapping directly influences the realism and aesthetic quality of the final rendered image.

2.1.6 Challenges in Mesh and *uv*-map Construction

Constructing high-quality meshes and *uv*-maps is a complex task that involves several challenges. One key issue is managing polygon count. Balancing the level of detail with performance is crucial; high-polygon meshes offer greater detail but require more computational resources, while low-polygon meshes are faster to render but may lack the necessary visual quality.

Another challenge is maintaining topological consistency. It is essential that the mesh has a consistent topology with no non-manifold edges or degenerate faces. This ensures accurate rendering and facilitates further processing, such as *uv*-mapping and subdivision.

uv-mapping itself presents difficulties, particularly in avoiding distortions. For complex or organic shapes, artists and algorithms must carefully manage issues like stretching, compression, and seams to ensure that the texture quality is maintained. Seams, where the 3D surface is split to lay flat in 2D space, are often necessary in *uv*-maps, but managing them to avoid visible discontinuities in the texture is a significant challenge.

Finally, the choice between automatic and manual *uv*-mapping comes with its own trade-offs. While automatic *uv*-mapping algorithms can save time, they may not always produce optimal results, especially for complex models. In contrast, manual *uv*-mapping allows for greater control but can be a time-consuming

process.

2.1.7 Summary

In summary, 3D meshes are foundational to the creation and visualization of digital 3D objects. They offer a versatile and efficient means of representing complex surfaces and are integral to the process of texture application through *uv*-mapping. The construction of high-quality meshes and *uv*-maps, though challenging, is essential for achieving realistic and visually appealing results in computer graphics. Understanding these concepts is critical for the development of tools that can assist in the evaluation and visualization of *uv*-mappings, as explored in this thesis.

2.2 *uv*-mapping

uv-mapping is a fundamental technique in computer graphics that involves projecting a 2D texture onto a 3D model. The process of *uv*-mapping creates a relationship between the vertices of a 3D mesh and corresponding points on a 2D image, allowing the texture to be wrapped around the 3D object in a way that aligns with its surface geome try. The term "UV" refers to the axes of the 2D texture coordinate space, with "U" representing the horizontal axis and "V" the vertical axis, analogous to the "X" and "Y" axes in Cartesian coordinates.

uv-mapping is critical in achieving realistic surface detail on 3D models, enabling artists and developers to apply detailed textures—such as colors, patterns, and even material properties—directly onto the surfaces of 3D objects. This technique is widely used in video games, movies, virtual reality, and any application that requires the rendering of detailed, lifelike 3D models.

2.2.1 The *uv*-mapping Process

The process of *uv*-mapping involves several key steps:

- Unwrapping the 3D Mesh: The first step in *uv*-mapping is to "unwrap" the 3D mesh into a 2D plane. This is akin to peeling the surface off a three-dimensional object, like an orange, and laying it flat. The goal is to create a 2D representation of the 3D surface without introducing significant distortions. This step often requires the creation of seams—cuts along the edges of the 3D model where the surface is split to facilitate the unwrapping process.

- Defining UV Coordinates: Once the mesh is unwrapped, UV coordinates are assigned to each vertex of the mesh. These coordinates correspond to specific points on the 2D texture. For example, a vertex at the corner of a face might be assigned UV coordinates (0.1, 0.2), indicating its position on the texture map.

- Texture Mapping: After UV coordinates are defined, the 2D texture is "mapped" onto the 3D model. This involves taking the 2D image and wrapping it around the 3D surface according to the UV coordinates, allowing the texture to follow the contours of the model. This step can reveal issues such as stretching or compression of the texture, which may require adjustments to the *uv*-map.

- Optimization: The final step in *uv*-mapping involves optimizing the UV layout to minimize distortions, reduce visible seams, and ensure efficient use of texture space. Artists often tweak the UV coordinates manually, adjusting the placement of seams and the distribution of texture space to achieve the best visual result.

2.2.2 Applications of *uv*-mapping

uv-mapping is used across various domains in computer graphics:

- Video Games: In video game development, *uv*-mapping is crucial for applying textures to characters, environments, and objects. Detailed textures can include everything from skin and clothing on characters to surfaces of buildings and landscapes. The use of *uv*-maps enables the creation of highly detailed and visually rich game worlds while maintaining efficient performance.

- Film and Animation: In the film industry, *uv*-mapping is used extensively in creating visual effects and animated characters. High-resolution textures are mapped onto 3D models to bring digital characters and environments to life with photorealistic detail.

- Virtual Reality (VR) and Augmented Reality (AR): In VR and AR applications, *uv*-mapping ensures that textures appear correctly on 3D models within immersive environments. This is critical for maintaining the illusion of realism and depth in these experiences.

- Product Design and Visualization: *uv*-mapping is also used in product design and visualization to apply textures to digital prototypes. This allows

designers to visualize how materials and patterns will look on a physical product before it is manufactured.

- Scientific Visualization: In scientific visualization, *uv*-mapping can be used to apply data-driven textures to 3D models, such as applying temperature maps to geological models or visualizing anatomical details in medical imaging.

2.2.3 Challenges

Despite its importance, *uv*-mapping presents several challenges that can impact the quality and efficiency of the resulting 3D model:

Distortion: One of the most significant challenges in *uv*-mapping is managing distortion. When a 3D surface is unwrapped onto a 2D plane, some level of distortion is inevitable. This distortion can manifest as stretching, compression, or warping of the texture, which can negatively affect the appearance of the model. Minimizing distortion while maintaining a usable *uv*-map is a key goal in the *uv*-mapping process.

Seams: Seams are necessary for unwrapping the 3D model into a 2D plane, but they can create visible discontinuities in the texture if not managed carefully. Artists must strategically place seams in less noticeable areas of the model or use techniques such as texture painting to hide them.

Manual Labor: While automated *uv*-mapping tools exist, creating a high-quality *uv*-map often requires manual intervention. Artists need to adjust UV coordinates, reposition seams, and optimize the layout to achieve the best visual results. This process can be time-consuming and requires a high level of skill.

Resolution and Texture Space: Efficient use of texture space is another challenge in *uv*-mapping. The goal is to maximize the resolution of the texture in areas where detail is most needed while minimizing wasted space. Poorly optimized *uv*-maps can lead to textures that appear blurry or pixelated on the 3D model.

Complex Geometries: For complex or highly detailed models, creating an effective *uv*-map can be particularly challenging. Organic shapes, such as characters or creatures, often require more intricate UV layouts to avoid distortion and ensure that textures align correctly with the model's geometry.

2.2.4 Methods and Techniques

Over the years, several methods and techniques have been developed to improve the *uv*-mapping process:

- Automatic Unwrapping Algorithms: Algorithms such as angle-based flattening (ABF) and least squares conformal mapping (LSCM) have been developed to automate the unwrapping process. These algorithms aim to minimize distortion and create a *uv*-map that accurately represents the 3D surface. However, the results often still require manual adjustment to achieve the best quality.

- Seam Placement Strategies: Effective seam placement is crucial for reducing visible discontinuities in the texture. Techniques such as hiding seams along natural edges or within folds of the model can help minimize their impact. Some tools also offer seam-painting features, where seams are placed interactively by painting directly on the model's surface [2].

- Texture Atlases: For models with multiple UV islands (separate sections of the *uv*-map), texture atlases are used to organize these islands into a single texture space efficiently. This approach is particularly useful for complex models, where different parts of the model require separate UV layouts.

- Multi-Resolution Texturing: Multi-resolution texturing involves using different texture resolutions for different parts of the model. This technique ensures that high-detail areas receive more texture space, while less important areas use lower-resolution textures, optimizing both quality and performance.

- Interactive UV Editing Tools: Modern 3D modeling software provides interactive tools for UV editing, allowing artists to manipulate UV coordinates, adjust seam placement, and visualize distortion in real time. These tools are essential for fine-tuning *uv*-maps to achieve the desired results.

2.3 Dual Quaternion Skinning

In 3D animation, "skinning" refers to the process of deforming a 3D mesh based on the movement of an underlying skeleton [3]. This technique is crucial for animating characters, enabling a mesh to move naturally and smoothly in response to the movements of bones and joints. Skinning is fundamental to character animation in video games, movies, and virtual reality, where it ensures that the character's surface deforms in a way that mimics real-world skin, muscles, and joints.

Traditional methods of skinning, such as linear blend skinning (LBS), are widely used due to their simplicity and efficiency. However, LBS can suffer from artifacts like "collapsing joints" and "candy-wrapper effects," where the mesh

deforms unnaturally, especially around joints like elbows, knees, and shoulders. To address these issues, more advanced techniques like dual quaternion skinning have been developed, providing better results in maintaining volume and smooth deformation.

2.3.1 Dual Quaternion

Dual quaternion skinning is a modern approach to character skinning that leverages the mathematical concept of dual quaternions. Quaternions are a type of hypercomplex number that extend the concept of rotation in three dimensions, without suffering from other rotation representation problem, like gimbal lock, a common problem with Euler angles. Dual quaternions extend this concept further by combining quaternions with dual numbers, allowing for the representation of both rotation and translation in a single, compact form.

A dual quaternion is represented as:

$$\mathbf{D} = \mathbf{q_r} + \epsilon \mathbf{q_d}$$

where

- $\mathbf{q_r}$ is the real part, representing rotation.
- $\mathbf{q_d}$ is the dual part, representing translation.
- ϵ is a dual unit with the property $\epsilon^2 = 0$

Dual quaternions allow for smooth and continuous transformations, making them ideal for interpolating between different poses in character animation.

2.3.2 Advantages

Dual quaternion skinning (DQS) offers several notable advantages over traditional skinning methods. One of the most significant benefits of DQS is its ability to preserve the volume of the mesh during deformations. This is particularly crucial in areas such as elbows, knees, and shoulders, where linear blend skinning (LBS) often causes the mesh to collapse, leading to unrealistic results.

In addition to volume preservation, DQS provides smooth and continuous rotations, effectively avoiding the "candy-wrapper" effect often seen in LBS, where the mesh twists unnaturally around a joint. This smoothness is achieved through the use of quaternions, which handle rotations more gracefully than matrices, ensuring more natural deformations.

Another advantage of DQS is its ability to blend both rotational and translational transformations in a more natural way, thanks to the dual quaternion representation. This capability allows for more complex and nuanced animations, where both rotation and movement are interpolated seamlessly, enhancing the realism of the animation.

While DQS is more computationally intensive than LBS, it remains efficient enough for real-time applications, making it suitable for use in video games and interactive simulations. Furthermore, by utilizing dual quaternions, DQS minimizes common skinning artifacts, resulting in cleaner and more aesthetically pleasing animations overall.

2.3.3 Applications

Dual quaternion skinning (DQS) is widely applied across several domains in computer graphics, offering significant advantages in various contexts.

In the realm of character animation within video games, DQS is increasingly favored for creating more realistic animations. As the demand for realism in games rises, DQS provides a means to achieve lifelike deformations without compromising performance. Its ability to handle complex movements while preserving mesh volume makes it particularly valuable in enhancing the visual fidelity of animated characters.

The film industry also leverages DQS, especially in animating digital characters and creatures for high-quality visual effects. The level of detail and realism required in cinematic productions necessitates techniques like DQS, which can maintain mesh integrity during intricate animations. This capability is crucial in delivering the level of visual effects expected in modern films.

In virtual reality (VR) and augmented reality (AR) applications, immersion and realism are paramount. DQS plays a critical role in ensuring that characters and objects behave in a believable manner, thereby enhancing the user's sense of presence within the virtual environment. This technology helps create more engaging and convincing experiences in VR and AR settings.

Additionally, DQS finds applications in simulation and scientific visualization, where accurate movement and deformation are essential. For example, in medical simulations, DQS is employed to model the movement of muscles and joints, closely mimicking real anatomy. This level of precision is vital for realistic and educational simulations.

2.3.4 Challenges

However, dual quaternion skinning is not without its challenges and considerations. Implementing DQS is more complex compared to traditional methods like

linear blend skinning (LBS). It requires a deeper understanding of quaternions and dual numbers, as well as careful management of blending operations. Despite its efficiency, DQS still incurs a higher computational cost than LBS, which can impact performance in real-time applications like video games, especially when animating a large number of characters simultaneously.

Interoperability is another challenge, as integrating DQS into existing pipelines that heavily rely on LBS or matrix-based transformations can be difficult. Ensuring compatibility and smooth transitions between different skinning methods may require additional effort. Moreover, setting skin weights correctly is crucial to achieving good results with DQS. Even with its advanced capabilities, improper weighting can lead to unwanted deformations, undermining the benefits of dual quaternion skinning.

Chapter 3

State of the Art

3.1 Automatic uv-map Generation

uv-mapping is a fundamental technique in computer graphics, enabling the application of textures, materials, and other surface properties to 3D models by linking the 3D surface coordinates to a 2D plane. While manual *uv*-mapping remains a common practice among digital artists, automatic *uv*-map generation has seen significant advancements over the past decades. Automated tools have emerged to simplify and speed up the process of *uv*-map creation, reducing the workload for artists and designers and making 3D modeling more accessible. However, the task is still complex due to inherent challenges like minimizing distortion, seam placement, and optimizing texture space utilization. This chapter reviews the current state-of-the-art methods for automatic *uv*-map generation, highlighting their strengths, limitations, and impact on modern 3D workflows.

3.1.1 Challenges in *uv*-map Generation

Flattening a curved 3D surface onto a 2D plane inevitably introduces distortions like stretching and compression, especially in regions of high curvature. The primary challenges in *uv*-mapping include:

- Distortion Minimization: Preserving the geometric properties of the 3D surface, such as angles and areas, is critical for maintaining texture quality.
- Seam Placement: Seam placement directly impacts the visibility of texture discontinuities on the final model.
- Texture Space Utilization: Efficiently using the available 2D space while maximizing resolution for detailed areas is key for high-quality texture mapping.

These challenges are well-documented in various review articles, such as the comprehensive survey by Hormann et al. (2007) [4], which provides an overview of different UV parametrization techniques and the trade-offs involved in minimizing distortion.

3.1.2 Early Approaches to *uv*-mapping

Early approaches to automatic *uv*-mapping sought to reduce distortion by relying on simple heuristics, such as planar and cylindrical projections. These methods were efficient but often resulted in significant texture distortion, particularly when dealing with complex geometries. Blender's "Unwrap" function and Autodesk Maya's UV tools offer these traditional projection methods, which remain in widespread use due to their simplicity. However, they require substantial manual adjustment by artists, particularly in regions with high curvature, where texture stretching or compression is most evident. Although not fully automated, these early tools paved the way for more advanced methods by highlighting the need for optimization.

3.1.3 Least-Squares Conformal Mapping (LSCM)

One of the significant advances in automatic UV generation came with the development of Least-Squares Conformal Mapping (LSCM), introduced by Lévy et al. in 2002 [5] . LSCM aims to minimize angular distortion by preserving local angles between edges of the mesh. This method translates *uv*-mapping into a constrained optimization problem, in which the goal is to find a mapping that minimizes distortion while ensuring a one-to-one correspondence between the 3D surface and the 2D map.

LSCM operates by fixing two vertices of the mesh and solving a linear system to generate a UV layout. The main advantage of this method is its ability to handle complex surfaces, producing relatively low-distortion mappings without the need for extensive manual intervention. However, LSCM is not perfect, as it can still introduce area distortion, which leads to stretching in some areas of the texture.

3.1.4 ABF++: Angle-Based Flattening

Another important contribution is the Angle-Based Flattening (ABF++) algorithm, proposed by Sheffer et al. in 2005 [6]. ABF++ improves upon LSCM by directly controlling both angle and area distortions. The algorithm works by formulating the *uv*-mapping problem as a nonlinear optimization process that preserves the angles between edges more effectively.

ABF++ builds on the idea of flattening a 3D surface while keeping the sum of angles around each vertex as close as possible to their values in 3D space. This method is particularly effective at reducing angular distortion, making it one of the most widely used techniques in *uv*-mapping. Its non-linear nature, however, requires longer computational time compared to LSCM, which makes it less suited for real-time applications.

3.1.5 Spectral Conformal Parameterization

An additional notable approach to *uv*-mapping is Spectral Conformal Parameterization (SCP), as introduced by Mullen et al. [7] . This method leverages spectral theory, which uses eigenfunctions of the Laplacian operator to compute parameterizations. SCP attempts to preserve conformal properties while being computationally more efficient than previous methods. The spectral approach has gained attention for its ability to handle highly detailed and complex geometries effectively, making it suitable for use in scientific visualization and virtual reality applications where detail and performance are critical.

3.1.6 Recent Advances in Machine Learning for *uv*-mapping

In recent years, machine learning techniques have emerged as promising tools for *uv*-map generation. Neural networks and deep learning models can be trained to learn complex patterns in 3D data and predict optimal *uv*-mappings with minimal distortion. Researchers such as Chen et al. (2020) [8] have explored using neural networks to automate *uv*-map generation, producing results comparable to traditional methods while requiring less computation and manual effort.

These machine learning methods are still in their infancy, but they hold potential for further automating the *uv*-mapping process, especially in scenarios where real-time performance is required. Neural networks could potentially adapt to different kinds of 3D meshes and learn to generate more context-aware *uv*-maps, thus reducing the need for human intervention and post-processing.

3.2 Existing *uv*-map Visualization Tools

uv-map visualization tools are essential for artists and developers working with 3D models, providing insights into how textures are applied to surfaces and identifying potential issues. These tools typically offer a range of features that help users assess and refine uv-maps, ensuring that textures are accurately and efficiently mapped onto 3D objects. This section outlines the key features that are

commonly found in *uv*-map visualization tools and an overview of some of the available tools will be provided at the end.

3.2.1 Real-Time UV Overlay

One of the most critical features of *uv*-map visualization tools is the ability to overlay the *uv*-map directly onto the 3D model in real-time. This overlay allows users to see how the 2D texture is mapped onto the 3D surface, providing immediate visual feedback on the alignment and fit of the texture. Real-time visualization offers interactive feedback, enabling users to adjust the *uv*-map and observe the effects on the model instantly. This interactivity is essential for fine-tuning UV layouts and ensuring that textures are applied correctly. Additionally, by visualizing the *uv*-map on the model, users can assess how well the texture aligns with the underlying geometry. This feature is particularly valuable for identifying issues such as misalignment, stretching, or compression, which can distort the texture's appearance.

3.2.2 Distortion Visualization

Distortion visualization is a crucial feature that enables users to identify areas where the *uv*-map does not accurately reflect the 3D model's geometry. Distortion can result in stretched, compressed, or skewed textures, adversely affecting the model's visual quality.

One approach to visualizing distortion is the use of heat maps, where different colors represent varying degrees of stretching or compression. This method provides a visual guide, allowing users to quickly identify and correct problematic areas within the UV layout.

Another common technique involves the application of a checkerboard or grid pattern to the *uv*-map. When the grid appears non-uniform on the 3D model, it highlights areas where the *uv*-map requires adjustment, making distortion evident and easier to address.

3.2.3 Seam Visualization

Seam visualization plays a crucial role in understanding the segmentation of a *uv*-map into islands. Seams, which are the "cuts" made in a 3D model to flatten it into a 2D *uv*-map, have a significant impact on the final appearance of the texture. To aid in this process, many tools offer seam highlighting, which directly displays seams on the 3D model, making them easier to identify and assess. This feature helps users ensure that seams are strategically placed in less visible areas or where they will not disrupt the texture. Additionally, some tools provide interactive

seam adjustment, allowing users to modify seam placement and immediately observe the effects on the *uv*-map. This capability is essential for minimizing visible seams and optimizing texture mapping.

3.2.4 Texel Density Visualization

Texel density refers to the uniformity of texture resolution across a 3D model's surface. Maintaining consistent texel density is crucial to avoid areas of low or excessively high resolution, which can result in uneven texture quality.

Visualization tools often employ density heat maps to illustrate variations in texel density across the model. These heat maps use different colors to indicate areas of high or low density, assisting users in balancing texture resolution throughout the *uv*-map. Additionally, some visualization tools offer scaling features that enable users to interactively adjust the scaling of UV islands. This functionality helps achieve consistent texel density, ensuring that texture details are evenly distributed across the model.

3.2.5 UV Island Visualization and Management

UV islands are distinct regions on a *uv*-map that correspond to specific parts of a 3D model. Proper management and visualization of these islands are crucial for creating an efficient and distortion-free UV layout. Tools often include features to easily identify and select individual UV islands, highlighting them both in the 2D *uv*-map and on the 3D model. This allows users to focus on specific areas that may need adjustment. Overlapping UV islands can cause texture conflicts, as different parts of the model might share the same texture space. To address this, visualization tools often feature overlap detection, highlighting these problematic areas so users can correct them. Additionally, efficient packing of UV islands within the UV space is important for maximizing texture resolution and minimizing wasted space. Visualization tools may offer visual guides or automatic packing suggestions to help users optimize their UV layouts.

3.2.6 Symmetry and Mirroring Visualization

Symmetry and mirroring features play a crucial role in 3D modeling by ensuring that symmetrical parts of a model share consistent uv-maps. This approach significantly reduces the need for duplicate texture work and optimizes the use of UV space. One of the key aspects of these tools is symmetry detection, which is often included in visualization tools. This feature automatically detects symmetrical parts of the model and applies mirrored uv-maps, ensuring that textures are applied consistently across all symmetrical components. Additionally, mirroring

visualization is another valuable feature provided by these tools. It visually indicates which UV islands are mirrored, helping users to verify that textures are correctly aligned on mirrored sections of the model. This is particularly useful for ensuring that symmetrical details are perfectly matched, leading to a more polished and accurate final model.

3.2.7 UV Space Utilization Visualization

Efficient use of UV space is essential for maximizing texture resolution and applying textures with the highest level of detail. Visualization tools are designed to help assess and improve UV space utilization through various features. One key feature is the use of space utilization metrics, which indicate the percentage of UV space occupied by UV islands. These metrics enable users to evaluate the efficiency of their UV packing and identify areas where optimization is needed.

Additionally, visualization tools often include visual guides or grids that assist users in aligning and distributing UV islands more evenly within the UV space. By using these guides, users can minimize wasted areas and ensure optimal texture coverage across their models.

3.2.8 Multi-Channel UV Visualization

In complex models, multiple UV channels are often utilized to apply various textures, such as diffuse, normal, and specular maps. Multi-channel UV visualization enables users to manage and assess these channels individually, ensuring that each texture is accurately mapped onto the model.

Channel switching tools allow users to toggle between different UV channels, providing a clear view of how each texture is applied. This is crucial for verifying that all textures align correctly and that each *uv*-map is optimized. Additionally, some tools offer channel synchronization features, allowing adjustments made to one *uv*-map to be automatically reflected in others. This ensures consistency across different texture maps, maintaining a cohesive appearance across the model.

3.2.9 Existing Tools for uv-mapping and Visualization

These tools vary widely in features and use cases, catering to different needs within the industry.

Blender is an open-source 3D modeling software that offers comprehensive uv-mapping and visualization tools. It features a dedicated UV editing workspace where users can unwrap, edit, and visualize uv-maps in real time. Blender supports both manual and automatic unwrapping methods, allowing for flexible

workflows. Its advanced packing features optimize the use of UV space, making it a versatile tool for professionals and hobbyists alike.

Autodesk Maya is a professional-grade software widely used in the film, television, and gaming industries. It offers robust uv-mapping tools that handle complex models and detailed textures. Maya's UV Editor provides a powerful environment for layout, packing, and minimizing distortion, and its integration with broader modeling and texturing workflows makes it particularly suitable for professional environments requiring high-quality uv-mapping.

RizomUV is a specialized tool focused entirely on uv-mapping, renowned for its precision and efficiency. It offers advanced automated unwrapping algorithms and real-time updates, allowing users to make immediate adjustments. RizomUV's sophisticated packing algorithms and tools for minimizing UV distortion make it ideal for professionals needing high-quality uv-maps quickly and efficiently.

Substance Painter is primarily a 3D texturing tool, but it includes essential UV visualization features. It allows users to preview uv-maps with textures applied, helping to identify potential issues like stretching and seams. While not primarily a uv-mapping tool, its UV-related features are valuable in ensuring that textures align correctly on the model, making it a useful complement to dedicated uv-mapping software.

Marmoset Toolbag is a real-time rendering tool that also includes *uv*-map visualization features. It provides real-time feedback on uv-maps, helping users identify and correct issues before final rendering. Marmoset Toolbag's capabilities in texture baking and visualization make it a useful tool for verifying and refining uv-maps as part of the final rendering process.

Houdini is a high-end 3D software package known for its procedural approach to modeling and effects. It includes advanced uv-mapping tools that are integrated into its broader effects and simulation workflows. Houdini's procedural uv-mapping capabilities, along with its sophisticated packing options, make it a powerful choice for complex visual effects pipelines.

In summary, the choice of uv-mapping and visualization tool depends on the specific needs of the project. Blender and Maya offer broad, integrated solutions, while RizomUV excels in specialized, high-precision uv-mapping. Substance Painter and Marmoset Toolbag are valuable in ensuring correct texture application, and Houdini provides advanced procedural tools for complex effects-driven workflows. Each tool has its strengths, making them suitable for different aspects of the uv-mapping and texturing process.

3.2.10 Our Approach

In the domain of UV-mapping, several tools and software solutions have been developed to assist in the creation, editing, and analysis of UV-mapped meshes. These tools generally fall into two categories: those focused on the creation and manipulation of uv-maps (e.g., software like Blender, Maya, and 3ds Max) and those designed for analyzing the quality of uv-maps, such as TexTools and UV Layout. While these tools are powerful, they typically rely on numerical metrics or visual inspection of the *uv*-map superimposed on the 3D model, which can sometimes be challenging to interpret, especially for non-experts.

The tool developed in this thesis differs fundamentally from existing tools in its approach to *uv*-map visualization and assessment. Rather than relying solely on traditional visual overlays or numerical measures (like stretching or distortion heatmaps), our tool introduces a novel method by generating a physically plausible morphing between the 3D mesh and its 2D UV counterpart. This morphing process is designed to be intuitive and visually informative, providing users with a more direct understanding of how the *uv*-map corresponds to the 3D geometry.

These are the key differences:

- Physically Plausible Morphing: The core innovation of our tool lies in the use of a physically plausible morphing algorithm that smoothly transitions between the 3D mesh and its 2D *uv*-map. This approach contrasts with traditional tools that may only provide static views or simple overlays, offering a more intuitive understanding of the *uv*-map's fidelity.

- Qualitative Assessment: While existing tools often emphasize quantitative metrics (e.g., stretch ratios, area distortion), our tool complements these by providing a qualitative visual assessment. The morphing visualization can highlight subtle issues in the uv-mapping that might not be immediately apparent from numerical data alone, such as unnatural stretching or compression in specific regions of the mesh.

- Educational and Illustrative Utility: Beyond practical applications in uv-mapping workflows, the tool developed in this thesis has significant potential for educational use. Its ability to visually demonstrate the relationship between 3D surfaces and their 2D uv-maps can serve as an effective teaching aid, helping students and professionals alike to better understand the complexities of uv-mapping.

- Web-based Accessibility: A significant advantage of the tool developed in this thesis is its web-based deployment. Unlike many existing UV-mapping tools that require installation on local machines and often demand high computational resources, our tool is accessible directly through a web browser.

3.3 3D Morphing Techniques

3D morphing [9] is a complex process used to seamlessly transform one three-dimensional shape into another through a series of smooth, intermediate stages. This technique plays a critical role in computer graphics, animation, and scientific visualization, enabling the visualization of transitions between different shapes in a visually compelling way. In the context of UV-mapped meshes, morphing can facilitate the transformation between a 3D model and its corresponding 2D UV layout, providing an intuitive and dynamic understanding of the *uv*-mapping.

The study of 3D morphing techniques has evolved significantly, with researchers exploring multiple approaches to ensure realistic and efficient transformations. Early methods focused on vertex interpolation, where the vertices of one shape are moved directly to the corresponding vertices of another. This approach, while computationally simple, often produces artifacts or unnatural deformations when there is a significant difference in the geometry or topology of the two shapes. As a result, more sophisticated methods have been developed to address these issues and improve the smoothness and realism of the morphing process.

One of the more advanced techniques is Radial Basis Function (RBF) interpolation, which treats the transformation as a spatial problem and allows for smoother transitions across the surface of the mesh. By defining key points on the source and target shapes, the morphing process can interpolate across the entire shape using mathematical functions that account for the relative positions of all points on the surface. This approach is widely used in applications where a high degree of surface continuity and smoothness is required, such as in character animation or medical visualization.

As-Rigid-As-Possible (ARAP) morphing is another method that has gained popularity in recent years. ARAP morphing preserves the local geometric properties of the mesh during the transition, minimizing distortions and maintaining the volume of the object. This technique is particularly useful for applications where the preservation of shape detail is critical, such as in 3D modeling, where fine surface details must be maintained throughout the morphing process. Researchers have applied ARAP morphing in various fields, including medical imaging, where it is used to transform and visualize anatomical structures while minimizing distortion to critical regions.

Space deformation techniques are another important category in the study of 3D morphing. Instead of manipulating the vertices of the mesh directly, these techniques modify the surrounding space in which the mesh is embedded. Free-Form Deformation (FFD) and Cage-Based Deformation are prominent examples. In FFD, a lattice of control points is placed around the object, and moving these control points deforms the space, resulting in the transformation of the

mesh. Cage-based techniques work similarly but use a simpler "cage" structure surrounding the object, allowing for complex deformations while maintaining control over the morphing process. These methods are particularly useful in interactive design and animation tools, where artists need precise control over deformations in real-time.

Energy minimization approaches have also been studied extensively in the field of 3D morphing. These methods rely on the principle of minimizing an energy function that encodes physical or geometric properties during the transformation. For example, minimizing surface tension or bending energy results in smooth, natural-looking deformations. These methods are highly valuable in physically-based simulations, where realistic transformations are critical, such as in the simulation of soft tissues or fluid flows.

Blend shapes and shape interpolation are commonly applied in character animation, particularly in facial animations and body movement. This technique involves creating a set of predefined key shapes or "blend shapes," which can be interpolated to produce a smooth transformation. Researchers have optimized this process for real-time applications, especially in video games and virtual reality, where rapid and smooth transitions between expressions or poses are crucial for maintaining immersion.

In terms of state-of-the-art applications, 3D morphing techniques have been applied in a variety of fields. In the entertainment industry, morphing is used to create realistic transitions between characters, facial expressions, and even complex shape-shifting transformations in films and video games. In scientific visualization, morphing allows researchers to compare different datasets, such as anatomical models or geological formations, by gradually transitioning between them. In medical imaging, 3D morphing has been employed to visualize changes in patient data over time, such as tumor growth or organ deformation, enabling clinicians to understand the progression of diseases.

Recent advances in the field have achieved significant improvements in the speed and accuracy of morphing algorithms. Machine learning techniques are increasingly being integrated into morphing workflows, enabling automatic correspondence detection between shapes, which has traditionally been a challenging problem. These advancements have allowed for more complex morphing operations to be performed in real-time, a crucial development for interactive applications like virtual reality and augmented reality.

In the context of UV-mapped meshes, modern morphing techniques such as ARAP and RBF interpolation have proven highly effective in minimizing distortions when transforming a 3D mesh into its corresponding 2D UV layout. By applying these methods, the morphing process ensures that the UV-map remains coherent and visually intuitive, enabling artists, animators, and researchers to

better assess the quality and accuracy of the *uv*-mapping. This, in turn, facilitates better optimization of textures in video games, films, and other visual media, where high fidelity and minimal distortion are essential.

3.3.1 Local-Global Optimization Framework

The paper "As-Rigid-As-Possible Surface Modeling" by Olga Sorkine and Marc Alexa [1] presents a novel surface deformation technique that balances local rigidity preservation with global flexibility. The approach is grounded in a local-global optimization framework designed to minimize distortion during surface deformations, which is crucial in maintaining the geometric integrity of the model.

$$E(S') = \sum_{i=1}^{n} w_i \sum_{j \in N(i)} w_{ij} \left\| (\mathbf{p}'_i - \mathbf{p}'_j) - \mathbf{R}_i(\mathbf{p}_i - \mathbf{p}_j) \right\|^2 \tag{a}$$

In the local Step (Local Rigidity Preservation) the method seeks an optimal local rotation $\mathbf{R}_i$ that approximates a rigid transformation. The goal is to preserve the shape of the 1-ring neighborhood around $\mathbf{p}_i$. Then the method globally optimizes the positions $\mathbf{p}'_i$ by solving a least-squares problem using $\mathbf{R}_i$. Then further minimization is performed by re-computing local rotations and using them to define a new right hand-side for the linear system, and so on. The method offers several advantages and has diverse applications. One key benefit is its ability to preserve the local shape of surface patches, making it especially effective in situations where maintaining local geometric details is crucial. Additionally, the method is computationally efficient, as it results in a sparse linear system that can be solved quickly, making it well-suited for real-time applications. Furthermore, the framework is versatile, as it can be applied to various surface modeling tasks, such as interactive shape editing, character animation, and morphing.

3.3.2 Visual Assessment of of Functional Maps

Shape matching is a crucial problem in geometry processing, with applications ranging from shape registration to interpolation and modeling. A prominent approach to shape matching is the functional maps framework, which establishes correspondences between shapes by mapping functions instead of points. However, assessing the quality of functional maps remains challenging, especially in capturing detailed surface correspondences.

Traditional visual assessment methods, such as smooth color gradients and texture transfer, have limitations. Smooth colors fail to represent high-frequency surface details and rely heavily on color differentiation, while texture transfer

struggles with distortion and cannot fully depict the mapping between shapes. These methods often fail to reveal subtle errors in shape matching.

Melzi et al. (2019) [10] propose a new approach for visually assessing functional maps by transferring object-space normals from one surface to another and visualizing the resulting lighting. This method highlights inaccuracies in high-frequency areas and provides a more intuitive and detailed view of the mapping quality. Compared to traditional techniques, the normal-based method improves the visibility of errors and allows for clearer comparison of different functional map solutions, making it a valuable tool for evaluating shape correspondences.

Both works address the challenge of evaluating complex geometric mappings between surfaces—whether it is a *uv*-map connecting a 3D model to a 2D layout or a functional map linking two 3D shapes. In both cases, the goal is to develop more effective, intuitive tools for visualizing the quality of these mappings, beyond purely numerical assessments. Our focus is on how standard methods might obscure critical details in UV-mapping and both works propose alternative visual techniques to address these shortcomings: our explores a morphing approach to intuitively illustrate the mapping between 3D and 2D, while Melzi et al. introduce normal-based lighting to reveal high-frequency errors in shape correspondences.

3.3.3 Our Approach

The ARAP method is primarily concerned with surface deformation. Specifically, it addresses the problem of how to deform a 3D surface in a way that minimizes local distortions, particularly preserving the shape of small features as much as possible, even under large deformations. The problem solved by ARAP is essential in applications such as character animation, where artists need to create smooth, realistic deformations of a model's surface as it moves. In particular, in ARAP, the user insert as inpunt a set of vertices that have to be moved to a particular position and a set of vertices that have to be still, ARAP achieves this by solving an optimization problem that adjusts vertex positions to minimize deformation energy.

In contrast, the problem addressed in this thesis is not about deforming surfaces, but about visualizing the relationship between a 3D surface and its corresponding 2D *uv*-map. The uv-mapping process involves flattening a 3D surface into a 2D plane, which inherently introduces distortions. The goal is to make this distortion as physically plausible as possible in order to show a as clear and visually straightforward possible the relation between 3D mesh and 2D *uv*-map.

Chapter 4

Methodology

The primary objective of our work is to provide users with a tool to enable them to intuitively and interactively assess the quality of *uv*-mappings through real-time, physically plausible morphing between the 3D mesh and its 2D UV counterpart (Figure 4.1).

In this chapter, we outline the approach used to achieve the real-time transformation between a 3D mesh and its *uv*-map. The first section provides an overview of the system, describing its user interface and customizable settings that allow for flexible morphing between 3D and 2D representations. The second section details the algorithms implemented for this transformation, focusing on the different interpolation methods, including linear transformations, dual quaternion blending, and more advanced techniques like local-global interpolation.

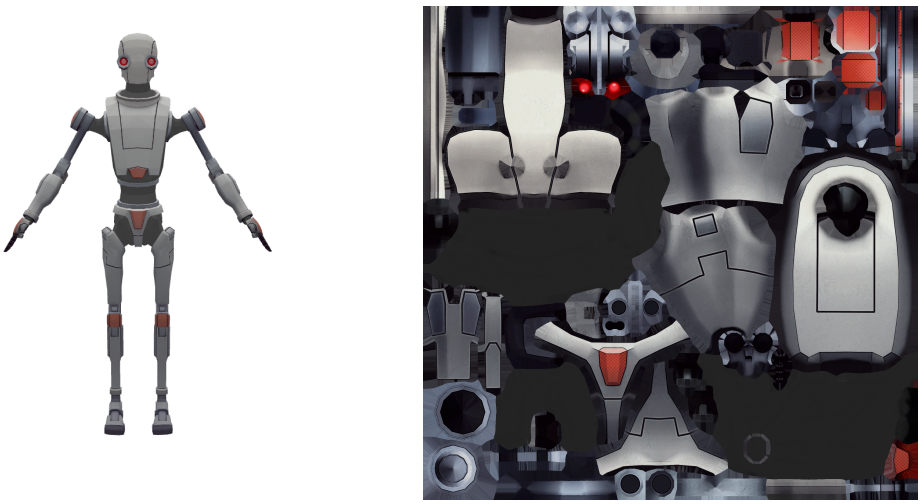


Figure 4.1: An example of a 3d Mesh (left) and its *uv*-map (right)

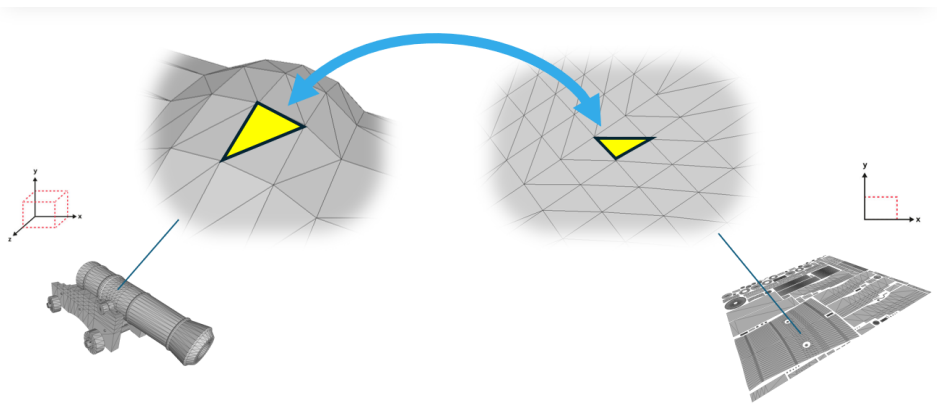


Figure 4.2: the mapping of a triangular section of a 3D mesh onto its corresponding 2D UV map.

4.1 Overview

The main objective of this system is to provide an intuitive, real-time visualization of the transformation between a 3D mesh and its 2D *uv*-map. This is achieved by allowing the user to interactively control the transition between these two representations via a slider, which controls a variable "t". The value of "t" ranges from 0 to 1.0, where:

- t = 0: The fully rendered 3D mesh is displayed.
- t = 1: The mesh is entirely transformed into its 2D *uv*-map.

Our tool allows to compute intermediate states between these two extremes (we show an axample in Figure 4.2, generating plausible and recognizable 3D embeddings for each value of "t" in a way that visually morphs the mesh into its *uv*-map. These intermediate positions must maintain the integrity of the 3D model and its *uv*-map, ensuring that the transformation is smooth, gradual, and consistent with the model's surface properties. To achieve this, the system provides several settings and customization options through a user interface (UI), allowing the user to control how the morphing process is performed. These settings influence both the mathematical interpolation techniques used and the visual representation of the transformation. The key features include:

- Linear Morphing: A basic option where the transformation between the 3D mesh and the *uv*-map is computed using simple linear interpolation between the vertices of the mesh and their corresponding positions on the *uv*-map. This approach is computationally efficient but may not always produce the most visually plausible transitions for complex geometries.

- Dual Quaternion Morphing: This option uses dual quaternion blending to perform smooth, rotationally consistent transformations. Dual quaternions are well-suited for handling rotations and translations simultaneously, ensuring that the mesh's surface characteristics are preserved during the morphing process. This method typically results in more "rigid" transformations compared to linear interpolation.

- Gluing or Separated Trajectories for Triangles: The system allows the user to choose whether the triangles of the mesh remain glued together during the transformation or if they can take independent trajectories. When this option is disabled we have potential loss of continuity between triangles.

- Local-Global Interpolation (ARAP-like): Inspired by the As-Rigid-As-Possible (ARAP) deformation technique, this setting provides a more sophisticated interpolation method. This method balances deformation across the entire mesh, resulting in smooth transitions that preserve the original surface details as much as possible.

- *uv*-map Island Timing: The system provides the ability to manipulate the timing of different *uv*-map islands (disconnected regions of the *uv*-map). This feature allows users to control when different sections of the mesh transition into 2D, this results in less compenetration.

- Computation Baking: For certain scenarios where real-time computation may be too intensive, the system provides an option to bake the entire interpolation process ahead of time. This pre-computation ensures fluid and seamless morphing, even on complex meshes

4.2 Morphing

4.2.1 Linear Morphing

One of the key interpolation methods implemented in the system is linear interpolation, which computes the transformation between a 3D mesh and its *uv*-map map using a straightforward blending of the two configurations.

$$\mathrm{p}(t) = (1 - t)\boldsymbol{a} + t\boldsymbol{b} \tag{b}$$

Where $\boldsymbol{a}$ and $\boldsymbol{b}$ are the starting and the ending points, while t is the interpolation parameter. This form of interpolation is simple and widely used for blending or transitioning between values. However, applying this formula independently to each vertex can lead to undesirable effects such as collapsing and clutter.

The reason for this is that linear interpolation does not account for the inherent geometric relationships between vertices. To address these issues, the goal of this thesis is to improve the interpolation process by using dual quaternion methods, which offer a more robust approach.

4.2.2 Dual-Quaternion Based Morphing

Instead of interpolating all the vertices in an independent way, we decided to compute a 4x4 affine transformation matrix $\mathbf{M}$ for each triangle of the mesh, where $\mathbf{M}$ is the only affine matrix that bring a particular triangle from its starting point to its ending point. Then $\mathbf{M}$ will be interpolated and applied to three vertices of each triangle:

$$\mathbf{M}(t) = (1-t)\mathbf{M} + t\mathbf{I} \tag{c}$$

where $\mathbf{M}$ is the transformation matrix that converts from the 3D mesh coordinates to its 2D *uv*-map and $\mathbf{I}$ is the identity:

$$(1-t)\boldsymbol{a} + t\boldsymbol{b} = ((1-t)\mathbf{M} + t\mathbf{I})\boldsymbol{a} \tag{d}$$

The equation d shows the equality between directly interpolating two points, $\boldsymbol{a}$ and $\boldsymbol{b}$ and transforming the point $\boldsymbol{a}$ using an interpolated matrix between $\mathbf{M}$ and $\mathbf{I}$.

The matrix $\mathbf{M}$ is constructed by taking the positions of three corresponding vertices from both the 3D mesh and the 2D *uv*-map and then will be used to morph the corresponding triangle. To create the transformation matrix the following steps are executed:

- Calculate Barycenters (Centroids): The barycenter (or centroid) of both the 3D and 2D triangles is calculated by averaging the coordinates of the three input vertices ($b3$ and $b2$). This step ensures that both triangles are centered around the origin for subsequent transformations.

- Shift Vertices to the Origin: After computing the barycenters, the vertices of both the 3D and 2D triangles are shifted so that the triangles are centered at the origin. This makes it easier to compute the rotation and scaling between them.

- Construct Basis Vectors for Each Triangle: For the 2D triangle, the method creates a 3D representation by treating it as a flat triangle in the xz-plane. For both the 3D and 2D triangles, a local coordinate system (basis) is constructed using the first two edges of the triangle and the normal vector. This results in two 3×3 transformation matrices ($\mathbf{T}_2$ for the 2D triangle and $\mathbf{T}_3$ for the 3D triangle) that describe the triangles in their local space.

- Compute the Rotation Matrix: The rotation matrix R is computed by multiplying the transformation matrix $\mathbf{T}_2$ of the 2D triangle with the inverse of the 3D triangle's 3×3 matrix $\mathbf{T}_3$. This effectively calculates the rotation and scaling needed to align the two triangles.

$$\mathbf{R} = \mathbf{T}_2 \cdot \mathbf{T}_3^{-1} \tag{e}$$

  However, the matrix R may not always result in a pure rotation matrix. A valid rotation matrix should have the following properties:

  - Orthogonality: The matrix's rows and columns must be orthogonal vectors (i.e., their dot products are zero).
  - Determinant of 1: The determinant of a valid rotation matrix must be +1 to preserve orientation (no reflection or scaling).

  Singular Value Decomposition (SVD) [11] is a method that decomposes the matrix into orthogonal components and corrects any imperfections by projecting the matrix onto the space of valid rotation matrices. So, this method must be applied to $\mathbf{R}$.

- Calculate the Translation Vector: A translation vector $\boldsymbol{t}$ is calculated to shift the transformed 3D triangle into the same position as the 2D triangle. The translation is based on the difference between the 2D barycenter($b2$) and the rotated 3D barycenter($b3$).

$$\mathbf{T} = b2 - inverse(\mathbf{R}) * b3 \tag{f}$$

- Construct the Final 4x4 Transformation Matrix: Finally, M is build as 4x4 transformation matrix by embedding the 3x3 rotation matrix and the translation vector into a 4x4 matrix. One of the seminal papers discussing the use of transformation matrices in computer graphics is the classic by James Foley and Andries van Dam in their book "Fundamentals of Interactive Computer Graphics" (1982) [12].

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{g}$$

At this point we built a transformation matrix for each triangle that will produce the same results as a direct interpolation between points, like shown in equation d.
The next step is to compute the dual quaternion starting from R and T that will result in a rigid transformation. A dual quaternion consists of two quaternions: a real part (q_r) representing rotation and a dual part (q_t) representing translation. Rotation matrix R can be converted to a unit quaternion as follows:

$$q_r = (w, x, y, z)$$

$$w = \frac{1}{2}\sqrt{1 + r_{11} + r_{22} + r_{33}}$$

$$x = \frac{r_{32} - r_{23}}{4w}, \quad y = \frac{r_{13} - r_{31}}{4w}, \quad z = \frac{r_{21} - r_{12}}{4w}$$

The translation vector $T = (t_x t_y t_z)$ can be embedded into the dual part as follow:

$$q_t = \frac{1}{2}\,(0, t_x, t_y, t_z)\, q_r$$

The dual quaternion $\hat{q}$ is composed of the real part (q_r) and the dual part (q_t).

$$\hat{q} = q_r + \epsilon q_t$$

Interpolation of dual quaternions at a given time t is typically done using spherical linear interpolation (SLERP) [12], which provides smooth interpolation of rotations and translations. To determine the interpolation path, you first calculate the dot product between the real parts of the dual quaternions, this part will be explored in the section 4.2.2. The SLERP interpolation is computes on the real part of the two dual quaternions.
SLERP formula :

$$q_r(t) = \frac{\sin((1-t)\theta)}{\sin(\theta)} q_{r1} + \frac{\sin(t\theta)}{\sin(\theta)} q_{r2}$$

To apply a dual quaternion to a point in 3D space, we perform a rigid transformation that includes both rotation and translation, it consists in applying to every vertex the dual quaternion transformation associated with its face.

$$p(i)' = q(j) \cdot p(i) \cdot \overline{q(j)} \tag{h}$$

Where:

- $p(i)$ is the i-th vertex of our mesh
- $q(j)$ is the j-th triangle face of our mesh (the one that contains the i-th vertex)
- $\overline{q(j)}$ is the conjugate of $q(j)$

Residual Linear Transformation

In our method, in addition to the dual quaternion transformation, that we will call the *main transformation*, we apply an additional transformation that we will call *residual transformation*. the residual transformation is an additional correction applied after the main transformation to ensure that the final result of the morphing process accurately matches the desired target. When only a dual quaternion transformation is applied, it may not fully align the 3D triangle with the 2D triangle if non-rigid deformations are present. This is where the residual transformation comes into play. The residual transformation accounts for any remaining differences after the dual quaternion transformation—specifically, the scaling and shearing components that the dual quaternion cannot capture. By computing and applying this residual linear transformation, we ensure that the final transformation precisely maps each vertex of the 3D triangle to its corresponding vertex in the 2D uv-map. Consider a scenario where a triangle on the 3D mesh needs to be morphed onto a significantly scaled version of itself in the uv-map. The dual quaternion will correctly rotate and translate the triangle but will not scale it. The residual transformation computes the necessary scaling so that, after both transformations are applied, the triangle from the 3D mesh perfectly overlays onto its UV counterpart. In some cases, when non-rigid transformations are more pronounced, applying the residual transformation first may produce a more gradual and visually clear morph as the shape conforms to its new size or sheared structure before rotation is applied.

Shortest Path Determination

Dual quaternions have an important property: two dual quaternions can represent the same roto-translation; in particular, q and $-q$ always represent the same transformation. For smooth interpolation of transformations across the entire mesh, the signs of the dual quaternions must be consistent. Changing the sign of the dual quaternion means to change the path that our triangle will take, it means that during the interpolation a triangle could take the short or the long path, to end in the same exact position (at $t = 1$). In particular, we don't want that two

adjacent triangles will end up in completely different intermediate position that which will make them impossible to enforce e surface continuity.

In our context, there is no a general best solution to chose, so we offered to the user the possibility to change the way the interpolation will look. The best result depends on the mesh and its *uv*-map characteristics. Ideally, in a mesh with small *uv* islands, we would want all the triangles that are in the same island will take the same path that will result in the shortest, but since we interpolate every triangle in an independent way we need a "democratic" way to ensure that each triangle will take a similar path to the other triangles of the island. In addition, when a certain *uv* island is big enough we could want to split it and that each part will take different paths.

To ensures that the path verse is consistent within each UV island we need to performs a voting mechanism where each island's faces vote on the path direction. The majority vote determines the path for the entire island, ensuring consistency across the island. This approach works well because it takes into account the natural segmentation of the *uv*-map (islands), allowing each island to decide on its path collectively, rather than each face making an independent decision.

To enable to interpolate in coherent way we need to ensure that neighboring faces have consistent quaternion signs. If the dot product between two neighboring faces' dual quaternions is negative, it indicates that one of the faces has a dual quaternion with the opposite sign. In such cases, the method flips the sign of the neighboring face's dual quaternion to ensure they are consistent. We use a breadth-first search (BFS) approach to propagate the sign consistency across the entire mesh, starting from an unprocessed face and continuing to its neighbors. It ensures that every face is processed once and that the sign consistency is maintained throughout the mesh. If there is a sign conflict that cannot be resolved (e.g., due to cyclic dependencies in the graph of neighboring faces), the method returns a fail state. However, in most cases, this method ensures that the signs are consistent, preventing flipping artifacts during the transformation.

4.2.3 Initial Transformation Optimization

Since the 3D vertices and the *uv*-map are often generated independently, without any correspondence in terms of orientation or scaling there is no direct relationship between the initial 3D mesh configuration and the *uv*-map. This can lead to situations in which the vertices have to be morphed in unnecessary complicated way, especially when the vertices are initially placed very far away from their correspondent position on the *uv*-map.

To mitigate this misalignment, we perform an initial optimization that computes a global transformation for the mesh. This optimization leverages the concept of area-weighted quaternion averaging, which allows us to find a single,

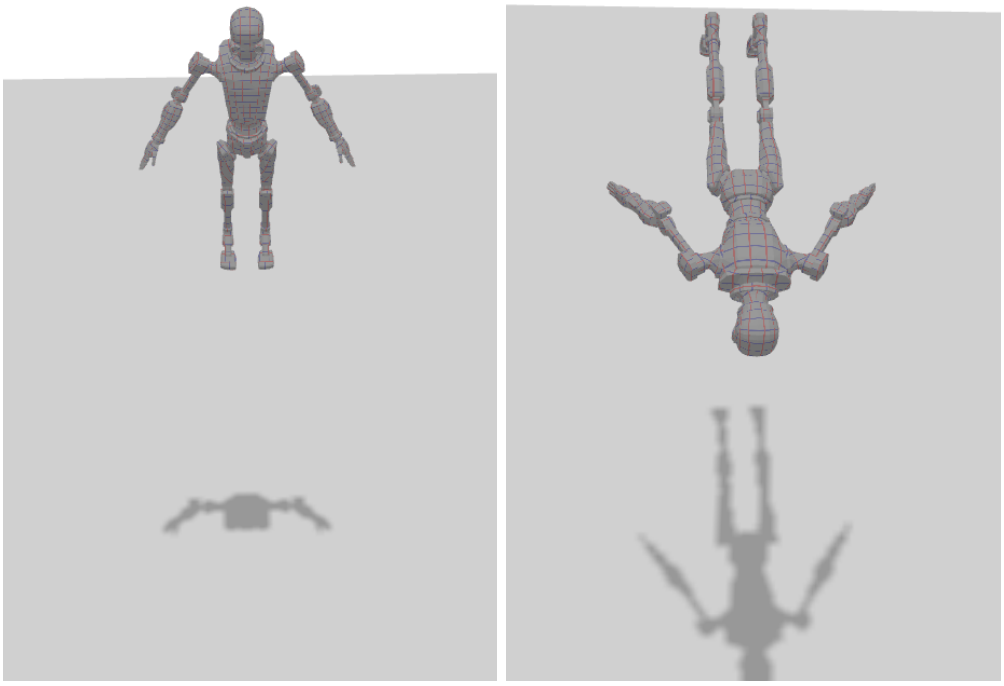


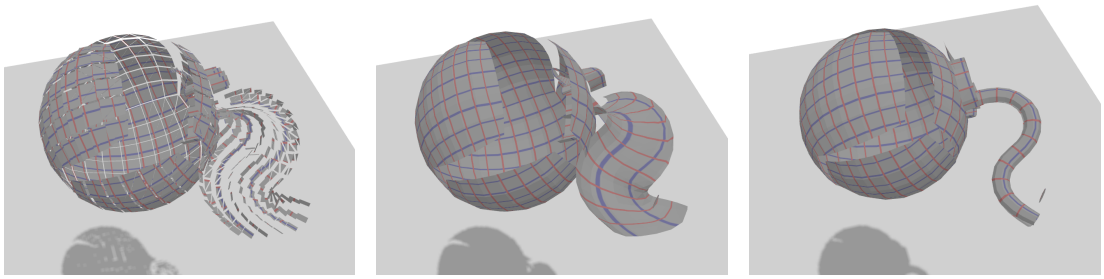
Figure 4.3: Initial Transformation optimization Off (left), On (right)

averaged rigid transformation (rotation and translation) that brings the 3D mesh into a more suitable alignment for subsequent operations.

In 4.3 Figure is shown the difference when this optimization is activated or deactivated, when it is activated the mesh result to be in a transformation where the projected mesh area to the *uv*-map area is maximized.

4.3 Surface Continuity Enforcement

When interpolating a 3D mesh to its 2D *uv*-map, each triangle in the mesh follows an independent transformation. As the 3D mesh morphs into the 2D UV layout, the individual triangles move according to their corresponding UV positions, but the triangles will separate or deform in unintended ways. This can lead to visible gaps, distortions, or misalignment between the triangles like in 4.4a figure.



(a) Morphing using Dual Quaternions

(b) Morphing using Dual Quaternions plus simple gluing of the detached vertices

(c) Morphing using dual quaternion ensuring surface continuity solving a linear system

4.3.1 Local Gluing

Gluing triangles is the process of ensuring that the vertices of adjacent triangles stay connected and maintain smooth transitions between them, even though each triangle may be moving independently during the interpolation. Gluing is especially necessary when applying transformations where adjacent triangles might not be synchronized, such as when using independent transformations or more advanced techniques like dual quaternion interpolation. We firstly implement a simple weighted averaging approach to glue triangles, where for each vertex in the mesh we computes a weighted sum of the vertex positions where the weight is the average of the vertex's 3D and 2D areas. This weight ensures that larger triangles (in both 3D and UV space) have a proportionally larger influence on the final glued position of the vertex.

The simple weighted averaging approach used in this function works well for basic scenarios, but there are more advanced techniques that can glue triangles while preserving additional properties, such as minimizing deformation or preserving the geometric characteristics of the mesh. As you can see in the figure 4.4b the mesh result quite deformed.

4.3.2 Global Isometry Maximization

To maximize the isometry of the triangles during transformations, we implemented a method that formulate our system as a linear system of equations. This method is applied on top of the previous one.

The vertices' positions after deformation must satisfy two types of constraints:

1. Edge Length Preservation: For each triangle edge (i,j), the system enforces the condition that the difference in vertex positions after deformation should

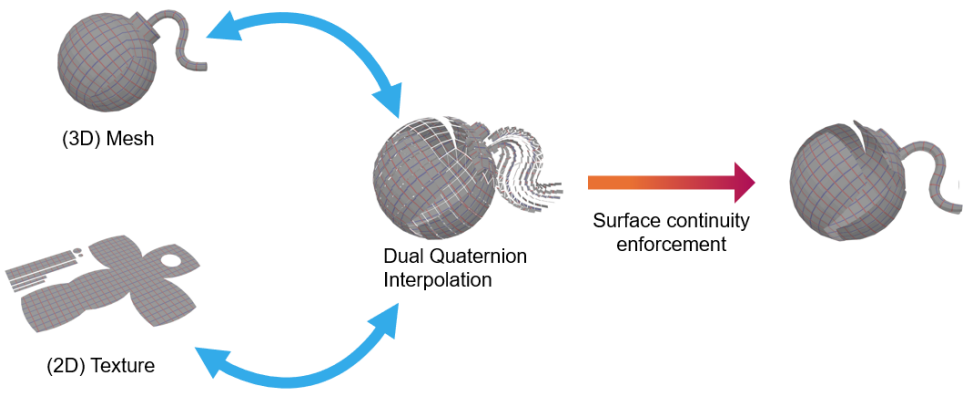


Figure 4.4: Interpolation and surface continuity enforcement scheme

approximate the original edge length. This is expressed as:

$$\mathbf{p}_i - \mathbf{p}_j \approx R(\mathbf{q}_i - \mathbf{q}_j)$$

To capture these constraints, a sparse matrix $\mathbf{A}$ and a right-hand-side vector $\mathbf{b}$ are constructed. Each row of $\mathbf{A}$ encodes the relationship between adjacent vertices, while $\mathbf{b}$ contains the positional differences between vertices after deformation.

2. Positional Constraints for Glued Vertices: In the *uv*-map, certain vertices may appear multiple times due to seams or discontinuities in the UV parameterization. These vertices are constrained to maintain consistency in the 3D space. The positions of these "glued" vertices are averaged using an area-weighted approach before deformation. The gluing constraint is added to the system by introducing additional rows in $\mathbf{A}$ and corresponding terms in $\mathbf{b}$, ensuring that these vertices maintain their continuity across the mesh. This is a soft constraint (weighted with a very low float number, e.g. 0.0001), it's like suggesting where the points should go, without this constraints the system may be singular.

This method is designed to maintain local isometry, meaning that the local shapes and angles of triangles are preserved as much as possible during deformation. At the same time, global surface continuity is enforced by ensuring that the triangles remain connected at their shared edges. Solving the system one time bring a consistent visual output but to ensure that the deformation converges and the results are as accurate as possible it should be iterate. A result of this method is shown in 4.4c figure and the complete process is shown in Figure 4.4.

4.3.3 Iterative optimization

Another method we propose consists in a local-global algorithm that leverages dual quaternion transformations to minimize distortion and preserve surface continuity. It combines dual quaternion transformations with a weighted gluing procedure. It consists in two main step:

- Local Step (Transformation Application and Gluing): The method begins by applying a dual quaternion transformation to each triangle of the mesh. After applying the initial dual quaternion transformation, the vertices are glued together using a weighted averaging scheme based on their geometric properties (e.g., triangle area). This ensure that neighboring vertices remain aligned and the mesh continuity is preserved.

- Compute Rigid Transformation: a new dual quaternion rigid transformation is computed. This time we use the vertex coordinates of the original 3D mesh as starting coordinates and the vertex coordinates of the interpolates 3D mesh after the gluing as target. This new roto-translation is applied to the original triangles.

- Iterate: The process is iterated by repeating the local and global steps until the solution converges.

This approach extends the local-global optimization framework by using dual quaternion transformations and a weighted gluing process to maintain both local and global surface consistency.

4.4 Interpolation Timing Differentiation

We introduce a method to further enhance the visual understanding of the transformation process, ensuring that the morphing trajectory is both more intuitive and informative. This method consist in differentiating the interpolation timing based on some heuristics, this means that the morphing process starts first for some part of the mesh and later for others. We propose two main heuristichs to determine this with options to uniform the timing per face or per island. These are the two heuristics:

- timing based on uv direction: the morphing will follow the uv-map position of the vertices (from o to 1 or viceversa, following the u or the v)

- timing based on 3D vertex position related to the distance from the barycenter: e.g. the system will first morph the more far away vertices, than the closer.

We also implement a parameter that control how fast is the morphing process for each time differentiated part of the mesh, this parameter offer the possibility to tweak the interpolation to see how each part of the mesh interpolate independently.

Here a detailed explanation of the timing based on uv direction: Let $\mathbf{dir}_{UV} \in \mathbb{R}^2$ be the normalized UV direction vector:

$$\mathbf{dir}_{UV} = \frac{\mathbf{dir}_{UV}}{|\mathbf{dir}_{UV}|}$$

Define the minimum and maximum values d_{min} and d_{max} along the direction $\mathbf{dir}_{UV}$, where $\mathbf{v_i}$ represents the UV coordinates of vertex i:

$$d_i = \mathbf{v_i} \cdot \mathbf{dir}_{UV}$$

$$d_{\text{min}} = \min_i(d_i)$$

$$d_{\text{max}} = \max_i(d_i)$$

For each vertex i, compute a normalized distance d'_i:

$$d'_i = \frac{d_i - d_{\text{min}}}{d_{\text{max}} - d_{\text{min}}}$$

Next, define the start time t_{start} and end time t_{end} for each vertex based on the flight time T_{flight}:

$$t_{\text{start}} = (1 - T_{\text{flight}}) \cdot d'_i$$

$$t_{\text{end}} = t_{\text{start}} + T_{\text{flight}}$$

Values of T_{flight} represent varying durations of the morphing

In results we will have less self-intersection or spatial compenetration during the morph. This leads to a more coherent and physically plausible transformation.

4.5 Baking The Animations

Morphing animations, especially those using linear systems to maintain smooth transformations, require complex computations at runtime. Each frame in the morph requires solving equations or applying transformations like dual quaternion blending or as-rigid-as-possible (ARAP) deformations. Once the morphing animation is baked, it can be reused multiple times without requiring recalculation. This improves the performance especially in resource-constrained environments like a web application.

Chapter 5

Implementation

The choice of technologies is crucial in ensuring the efficiency, performance, and user experience of the tool, the following chapter describes the implementation details, focusing on technologies we used and techniques used to handle complex mesh-related operations. Moreover, it will be explained how and why this particular technological stack was used and which libraries were employed.

5.1 Used Tools

We built the tool using a combination of C++ for backend geometry processing and JavaScript with Three.js for frontend rendering. The decision to use C++ is driven by its efficiency in handling complex geometry processing tasks, which are computationally intensive and benefit from the performance optimizations available in C++. Three.js is chosen for its capability to render 3D graphics in the browser using WebGL, providing an interactive and visually rich experience. Emscripten, a compiler toolchain, is employed to compile the C++ code into WebAssembly, allowing it to run efficiently in the web environment while still being accessible from JavaScript.

C++ and Emscripten for Geometry Processing

Geometry processing tasks, such as the morphing algorithm require high computational efficiency and precision. C++ was chosen for these tasks due to its performance advantages and mature ecosystem for geometry processing. Emscripten, a toolchain that compiles C++ code to WebAssembly, was used to integrate these tasks into the web-based environment.

The decision to use C++ and Emscripten was driven by several key considerations. First, C++ is a compiled language known for its performance and low-level

control over system resources. Allowing the tool to handle complex meshes and perform real-time calculations with minimal lag.

In addition to its performance benefits, C++ offers a rich ecosystem of libraries for geometry processing, such as CGAL (Computational Geometry Algorithms Library) and Eigen for linear algebra operations. These libraries provide robust, tested, and efficient implementations of essential algorithms and data structures. Emscripten also facilitates seamless integration between the geometry processing backend and the Three.js visualization frontend by providing bindings that allow communication between C++ code and JavaScript.

5.1.1 C++ and Javascript Interaction

A critical aspect of the implementation is the handling of mesh instances defined in C++ but used within the JavaScript execution flow. The mesh is initially instantiated and processed in C++, where operations such geometry manipulation are performed. Once processed, the mesh data needs to be made accessible to JavaScript for rendering. Emscripten's bindings allow the C++ mesh instance to be wrapped in a WebAssembly object that can be manipulated in JavaScript as if it were a native JavaScript object. This seamless integration ensures that the mesh instance, while originating in C++, can be used in the JavaScript execution flow for tasks such as updating the Three.js scene, responding to user interactions, or triggering further processing. In particular, most of the function are called on the mesh instance in Javascript code, as the instance was a classic Javascript class instance, but during the construction of the object a part of memory in the heap is allocated and initializated in the Javascript code. This memory contains the vertices data. The pointer to this area memory is used every time the mesh is modified in the C++ code, overwriting the mesh vertices, then the threejs object need to be notified when the data are changed and perform a new rendering using the new ones.

5.1.2 Rendering with Three.js

Three.js is a popular and powerful JavaScript library that simplifies the creation of 3D graphics in web applications. It is built on top of WebGL, which allows for high-performance rendering directly in the web browser. Three.js is used to handle the rendering of 3D models and their *uv*-maps in a web environment. It is a widely adopted JavaScript library that simplifies WebGL, providing an abstraction layer that allows for easier development of 3D content in the browser. Several factors influenced the decision to use Three.js for the visualization component of the tool:

- Ease of Integration with Web Technologies: Since the tool is designed to be web-based, Three.js offers seamless integration with standard web technologies like HTML5, CSS3, and JavaScript. This allows for a smooth development workflow and easy customization of the user interface.

- High-Level Abstractions: Three.js provides high-level abstractions for common 3D graphics tasks, such as creating and managing scenes, cameras, and lights. These abstractions simplify the development process and reduce the complexity involved in rendering 3D objects, allowing the focus to remain on implementing and refining the core functionalities of the tool.

- Performance and Real-Time Rendering: Three.js is optimized for performance, leveraging WebGL to utilize the GPU for real-time rendering. This is essential for the tool, which needs to render the morphing process interactively and smoothly, even with complex 3D meshes. Three.js also supports a wide range of optimizations, such as frustum culling, level of detail (LOD) management, and efficient memory usage.

- Community Support and Documentation: Three.js has a large and active community, along with extensive documentation and numerous examples. This support network is invaluable for troubleshooting issues, learning best practices, and accelerating development.

In the context of the current implementation using Three.js, a multi-pass rendering technique is employed to perform custom texture processing and debugging. Initially, textures are loaded through Three.js, which fetches the image data to be applied to 3D models in subsequent stages. To visualize debugging information on the texture, a custom shader is applied. This shader renders the texture onto a 2D quadrilateral (quad), facilitating the addition of debugging elements, such as a grid for *uv*-mapping or color adjustments.

The rendering process for this pass occurs off-screen, utilizing a render target or framebuffer object. This approach allows for the temporary modification of the texture without directly affecting the scene being displayed. During this off-screen pass, the custom shader modifies the texture by integrating visual debugging tools, such as overlaying a grid or altering color values.

Once this initial pass is completed, the contents of the off-screen buffer, which now hold the modified texture, can be read and accessed. This enables the application of the adjusted texture in further stages of the rendering process. By separating this operation into distinct passes, the texture can be processed independently before its application to the final 3D model.

The use of multi-pass rendering in this context serves to fully exploit the built-in rendering pipeline of Three.js, including advanced features such as lighting

models, shadow mapping, and reflections. By performing texture modifications in an isolated pass, the modified texture can be seamlessly integrated into the scene's lighting and shading systems in subsequent passes. As a result, the final rendered scene benefits from the real-time realism provided by Three.js's pipeline while incorporating the desired debugging tools and texture modifications.

Shadows play a crucial role in enhancing the visual understanding of the morphing process, especially when transitioning between complex 3D shapes and their 2D *uv*-maps. By casting shadows on the surface of the mesh, viewers can better perceive depth, curvature, and surface continuity, which are essential for interpreting how different parts of the object deform and transform. Shadows highlight subtle geometric changes, such as folds, creases, or distortions, that may otherwise be difficult to observe, making it easier to track the structural integrity of the mesh during the morphing sequence. This added visual cue significantly improves the user's ability to assess the quality and accuracy of the transformation.

5.2 Implementation Details

5.2.1 Initialization

The Javascript code is responsible to load the 3D model from the file system, this is convenient because of the many available loader provided by the Three js library. After this, the C++ class "Mesh" is responsible for initializing the mesh data structures, verifying the integrity of the input data, and setting up various internal states required for further mesh processing. During the mesh setup the pointers to the correct memory location used by the JavaScript part are set and this will be used for the communication between the two systems. To store the vertices data the "polygon soup" approach was used since it enables a more convenient processing of the mesh deformation. The most important part of the initialization is the computation of the transformations related to each face of the mesh, this aspect will be better described in the 4.2.1 and 4.2.2 sections. During the initialization we also calculate the initial rotation of the mesh, discussed in 4.2.3 and shown in Figure 6.1 and 6.2, this is done by iterating over faces and calculating an average rotation based on the mesh's geometry. By weighting the rotation according to the area of each face, the method ensures that larger faces, which contribute more to the overall shape of the mesh, have a greater influence on the computed average rotation. This bring the mesh in a position where the following transformation will be easier and more visually understandable.

UV Island Determination

Another key-step of the initialization is determination of the island of the vertices based on the island of the *uv*-map. An "island" on a *uv*-map refers to a distinct, contiguous region of a 2D UV space that represents a connected part of a 3D model's surface. To achieve this a union-find algorithm is used. In this case, even with a "polygon soup" configuration, we use another pre computed data structure that describes which vertex is a copy of another, then these copies and the vertices connected by an edge are passed to the union part of the union-find algorithm and then the ID of the island is found using the "find" part of the algorithm. The island ID is used in other part of the interpolation algorithm, to determining interpolation timing and in the using of the dual quaternion to determine to take the shortest path or not.

5.2.2 Interpolation Algorithm for Mesh Transformation

In this context, interpolation refers to the process of calculating intermediate vertex positions between the mesh's original state and its 2D *uv*-map over time. Depending on the t input, the algorithm computes where each vertex should be, based on its start and end positions (position on the 3D mesh and position on the 2D *uv*-map). This involves a mix between initial (in this case the Identity) and final transformations using either linear or non-linear interpolation methods.

```
Data: t: time progression, splitResidual: boolean flag, linear: boolean
      flag
Result: result: vector of interpolated vertex positions
begin
    Initialize result as an empty vector of glm::vec3;
    Reserve space in result for f.size() * 3 elements;
    foreach face f_i in f do
        for each vertex index j in face f_i do
            Compute interpolationValue as follows:

                interpolationValue ← (t − v[f_i.vi[j]].tStart) / (v[f_i.vi[j]].tEnd − v[f_i.vi[j]].tStart)

            Clamp interpolationValue between 0.0 and 1.0;
            Apply sigmoid function to interpolationValue;
            if linear then
                T ← mixLinear(I, f_i.three2two, interpolationValue);
            end
            else
                T ←
                 mix(I, f_i.three2two, interpolationValue, splitResidual,
                 f_i.pathVerse);
            end
            Set originV to the position of the vertex v[f_i.vi[j]];
            Compute resultV as T.apply(originV);
            Append resultV to result;
        end
    end
    return result;
end
```

Algorithm 1: Interpolate

This is the main function that is called every time each a different vertices interpolation at a different time t is requested. This function use pre-computed data from the mesh like the "three2two" transformation (associated to each face) that is the transformation that bring 3D points to the 2D *uv*-map, "tStart" and "tEnd" (for each vertex) that are used to calculate the actual interpolation time (described in 4.4) and two boolean values: "splitResidual" and "pathVerse" used in the dual quaternion interpolation technique (described in 4.2.2).

5.2.3 UI and user interaction

The interaction between the user and the tool is managed entirely within the JavaScript environment, with Three.js handling the visualization. When a user interacts with the tool, such as modifying the *uv*-map or triggering a morph, these inputs are processed in JavaScript and then passed to the underlying C++ code through the Emscripten bindings.

The C++ backend processes these inputs, updates the mesh data, and write the updated data to a designated area memory allocated by JavaScript code. Three.js then renders the updated mesh, providing real-time feedback to the user. This tight integration between C++ and JavaScript ensures that the tool remains responsive and interactive, even when performing complex geometry processing tasks.

In order to experiment with the many alternatives we implemented of we equipped ourselves with a lot of options, this UI is visible in 5.2 Figure. To the end user the expected interface is much simpler and consists three simple sliders: one Interpolation slider and Two style sliders. In the left up of Figure 5.1 the interpolation slider is settled to 0 (3D mesh) and the style slider are settled to 0.5, the first slider permit to visualize a mix the the color of the textures and white, the second one draw a grid on the mesh, useful to help detect deformations. In the left low is shown the interpolation is at 0, the grid is deactivated (slider at 1.0), and the colors of the original texture (slider at 0.0). In the right figures, other combinations are illustrated.

We draw the user interface using HTML and CSS.

5.2.4 Meshes Dataset

To thoroughly evaluate the effectiveness and performance of the tool, a variety of test meshes were used, encompassing both low-poly and high-poly models. These meshes were chosen to represent a range of complexity and detail, simulating the types of 3D models commonly found in video games and other digital content creation fields. All the test meshes were pre-unwrapped, meaning they already had established *uv*-maps, allowing the tool's visualization and analysis capabilities to be effectively tested.

We also created basic low-poly meshes, like cubes, cylinders, etc. using the MeshLab [13] software.

We used meshes from differents repositories and asset stores([14], [15], [16]).

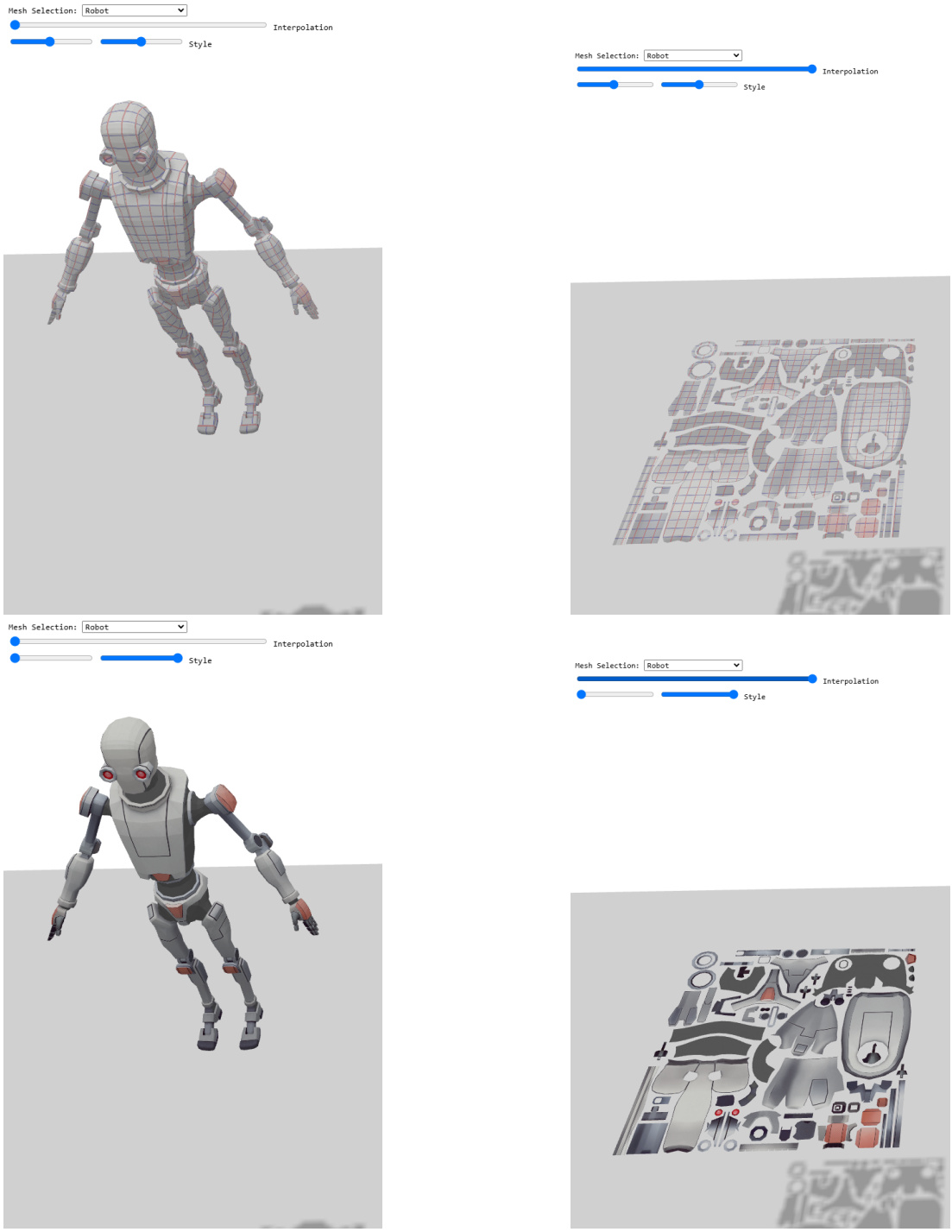


Figure 5.1: Effects of the sliders, in the y axis the style axis are moved, in the x axis the interpolation slider is moved fom 0 to 1

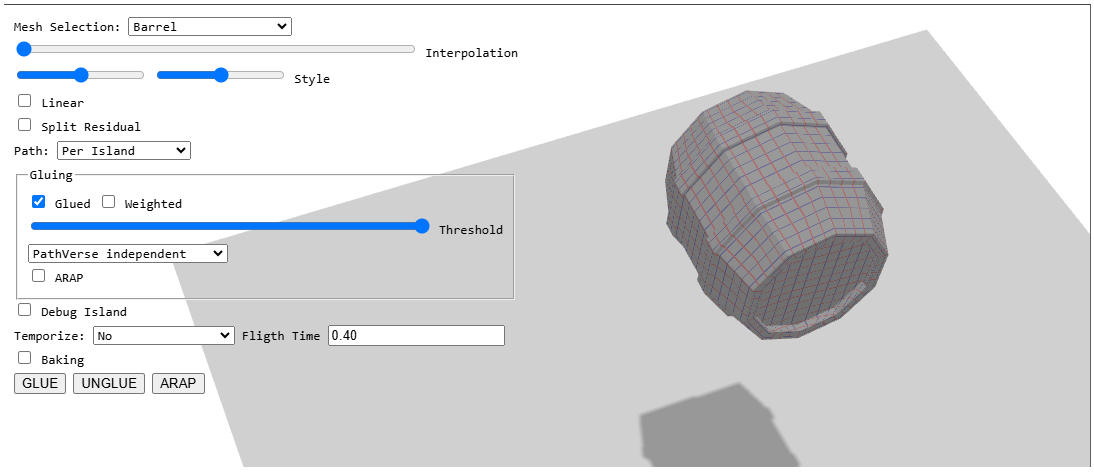


Figure 5.2: User Interface

5.2.5 Libraries

The development of our tool relies on two key libraries for mathematical operations and 3D transformations: Eigen and GLM (OpenGL Mathematics). These libraries were chosen for their efficiency, ease of integration, and widespread use in both academic research and industry applications.

Eigen

Eigen was integral to this project for several reasons:

- Matrix and Vector Operations: Eigen provides an intuitive interface for handling operations on matrices and vectors, which are essential in 3D transformations and *uv*-mapping. Throughout the implementation, Eigen was used to perform transformations between 3D mesh coordinates and their corresponding 2D UV coordinates, as well as to compute centroids, normals, and other geometric properties.

- Performance: Eigen is highly optimized for performance, supporting expression templates that minimize the number of temporary objects created during operations. This was particularly useful when dealing with real-time morphing and large datasets, where computational efficiency is critical.

- Decomposition: The library's built-in support for Singular Value Decomposition (SVD) and other matrix factorization allowed for accurate handling of non-rigid transformations, particularly when ensuring that rotation matrices were valid during the morphing process.

GLM

GLM, short for OpenGL Mathematics, is a header-only C++ library that provides mathematics functionalities commonly used in computer graphics, especially when working with OpenGL. GLM was chosen for this project due to its straightforward handling of 3D transformations, which aligned closely with our needs for morphing and *uv*-map visualization.

Chapter 6

Results and Conclusions

6.1 Results

In our research, we focus primarily on providing a qualitative understanding of *uv*-mapping rather than quantitative results. Unlike many traditional approaches that rely on numerical metrics such as distortion values or texture efficiency, our work emphasizes visual data representation. To achieve this, we present a series of detailed figures as the primary form of results, illustrating the morphing process between 3D meshes and their 2D *uv*-maps. These figures visually demonstrate the effectiveness and fidelity of the *uv*-mapping, allowing for an intuitive assessment of how well the surface is preserved during the transformation. By relying on visual results, we aim to provide a more accessible and direct evaluation of the mapping quality.

Figure 6.1 demonstrates the morphing animation using dual quaternion blending, in combination with solving a linear system to enforce surface continuity and maintain the rigidity of the faces. Additionally, the optimization process discussed in Section 4.2.3 is enabled. In comparison, figure 6.2 shows the same morphing sequence without the initial optimization. As illustrated, the latter exhibits a more complex and visually distorted transformation, underscoring the significance of the optimization step.

Figure 6.1: Sequence of a Morphing using dual quaternion and solving a linear system

Figure 6.2: Sequenc of a Morphing using dual quaternion and solving a linear system Without initial roto-translation optimization

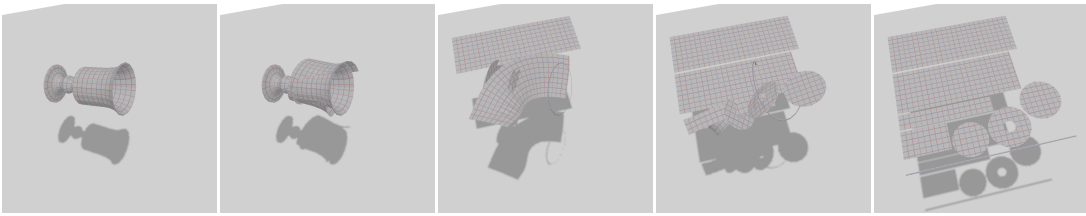


Figure 6.3: Sequence of a Morphing using dual quaternion and solving a linear system using a different mesh

In Figure 4.4 using time differentiation (as discussed in Section 4.4), we observe that different UV islands of the mesh morph asynchronously, making the transition smoother and simpler to follow.

Figure 6.5 compares between linear morphing (left) and dual quaternion morphing (right). The linear morphing introduces significant deformation, while the dual quaternion-based morphing preserves the integrity of the mesh, maintaining a more natural transformation.

Figure 6.6d illustrates the effectiveness of dual quaternion-based morphing combined with a linear system solver. The optimization enforces surface continuity, preventing unwanted deformations during the morphing process. The method maintains rigidity and smoothness throughout the transition, especially when handling more complex geometry.

Figure 6.6a illustrates the linear morphing process where each vertex of the 3D mesh transitions linearly towards its corresponding position on the *uv*-map. While this method is computationally efficient, it introduces noticeable distortions, particularly in areas with complex curvature. The vertices move independently, leading to a lack of coherence between adjacent triangles, resulting in potential overlaps and loss of geometric consistency. This method, though straightforward, fails to maintain the structural integrity of the mesh, especially during significant deformations, making it unsuitable for high-fidelity applications.

Figure 6.6b shows middle frames of the morphing using dual quaternion without enforce surface continuity and 6.6c show how applying a simple algorithm

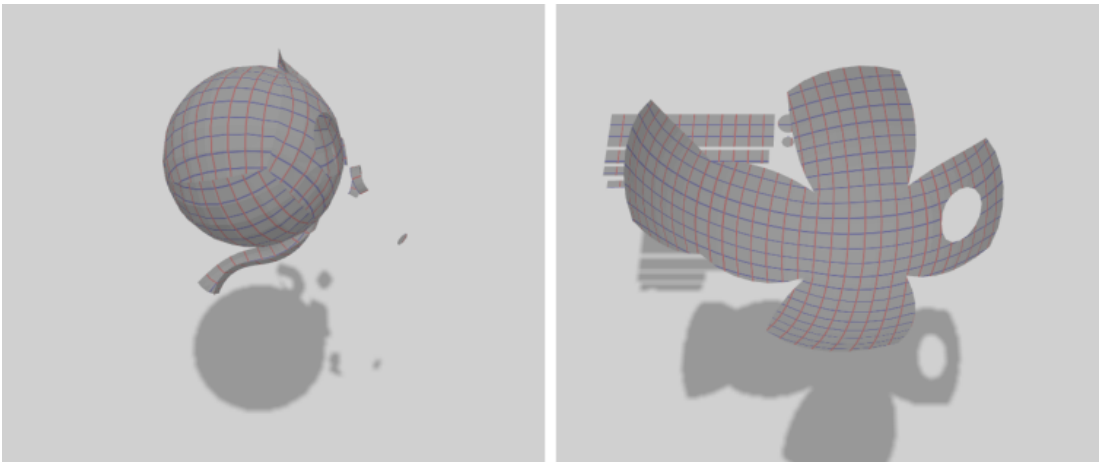


Figure 6.4: Interpolation using timing differentiation

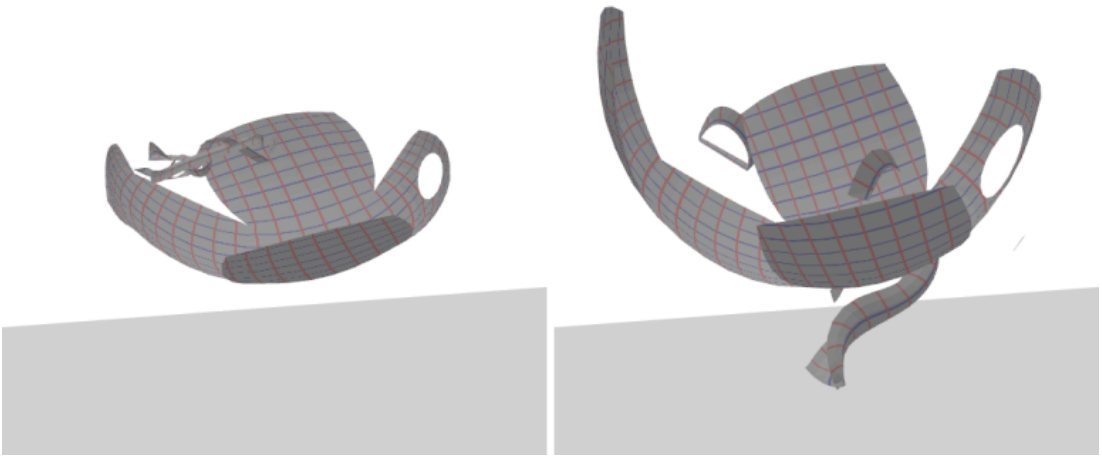
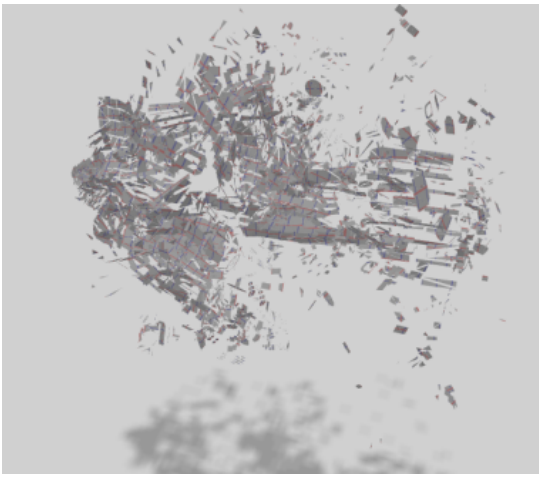
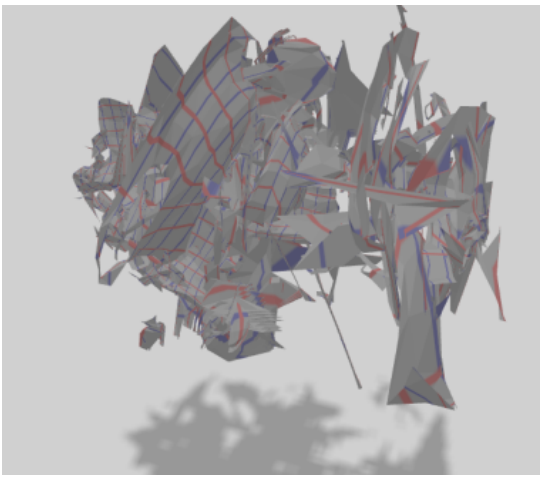


Figure 6.5: Middle morphing frame using Linear Morphing (left) and dual quaternion morphing (right)

(a) Linear Morphing



(b) Dual Quaternion Morphing



(c) Dual Quaternion plus surface continuity

(d) Dual Quaternion plus surface continuity based on linear systems

Figure 6.6: Different results using different morphing techniques

discussed in section 4.3 bring to an unclear and unusable result.

Some of the Figures show only the middle frame of the morphing process, while these still frames provide a snapshot of the transformation, the full morphing sequence offers a much richer and more comprehensive understanding of how the surface deforms and transitions between domains. To fully appreciate the effectiveness of our methods and the quality of the results, the rider is postponed to explore the tool directly or view videos that demonstrate the real-time morphing process in action.

6.1.1 Performace

We tested the tool and produced the results on a laptop equipped with an Intel Core i7-10710U CPU and an Nvidia GeForce GTX 1650 Max-Q Design.

Our tool is designed to deliver real-time performance for most of the methods implemented, ensuring smooth and interactive visualization of the morphing process between 3D meshes and their 2D *uv*-maps. Specifically, methods such as linear morphing and dual quaternion morphing perform efficiently in real-time, even for complex meshes, allowing users to manipulate and observe the transitions seamlessly.

However, there is one exception: the method that involves solving a linear system to enforce surface continuity and minimize distortion. While this method provides highly accurate results, its performance does not scale well for larger meshes. For meshes with fewer than 50K vertices, the system can still maintain real-time responsiveness. But as the vertex count increases beyond 50K, the computation required to solve the linear system becomes significantly more demanding, causing delays and reducing interactivity. In these cases, the tool is unable to sustain real-time performance, making it more suitable for precomputed or static visualization when dealing with very high-resolution meshes.

6.2 Conclusions

In this thesis, we designed and developed a tool that offers a novel approach to *uv*-map visualization through the use of real-time, physically plausible morphing between a 3D mesh and its 2D UV counterpart. The visualization of *uv*-maps is a critical aspect of 3D modeling, providing insight into how textures are applied to 3D meshes. Visualizing the relationship between a 3D model and its *uv*-map is challenging due to the difference between 3D geometry and its 2D representation. Flattening a complex surface often distorts proportions, making it hard to relate 3D forms to their *uv*-map. Managing seams is another issue, as the "cuts" in the model can disrupt texture continuity, and it's not always intuitive how the UV

pieces fit together. Additionally, complex models with many small parts make interpretation difficult, while predicting texture stretching and distortion adds further complexity to the process. Our system aims to bridge the gap between quantitative *uv*-map evaluations and qualitative, visual assessments, offering a more intuitive understanding of the mapping process.

The tool we created allows users to interactively explore the relationship between the 3D surface and its *uv*-map by gradually morphing between these two domains. By focusing on both linear interpolation and dual-quaternion-based morphing, we ensured that the visualization was smooth and physically accurate. Additionally, we introduced several innovations, including methods to optimize surface continuity, manage UV islands, and bake the morphing process for more complex meshes.

Our approach offers significant advantages over traditional *uv*-mapping visualization tools. By providing a real-time, interactive morphing process, we enable artists, researchers, and educators to better assess the quality of *uv*-maps in an intuitive, visual way. This tool supplements existing methods, making it easier to detect issues such as texture distortion, stretching, and seam placement, which are crucial for high-quality 3D rendering.

6.2.1 Future Work

In future work, it will be essential to measure the effectiveness and user experience of the visualization tool. This can be achieved by conducting user surveys that gather feedback on the clarity of the 3D mesh to 2D UV-map transformation, ease of interaction, and overall usefulness in different practical applications. These surveys should target various user groups such as digital artists, researchers in geometry processing, and educators in computer graphics, in order to gather diverse perspectives on the tool's performance.

The user study would consist of two main phases: a task-based evaluation and a feedback survey:

- Phase 1: Task-based Evaluation, participants would be given a set of predefined tasks to complete using our tool. These tasks would simulate real-world scenarios, such as: Identifying distortions, Assessing seam placement, Comparing *uv*-maps.
- Phase 2: Feedback Survey, After completing the tasks, participants would fill out a detailed survey focusing on: ease of use, effectiveness, comparison to other tools, visual clarity

Participants would rank these aspects on a Likert scale (e.g., 1-5), and we would also collect qualitative feedback through open-ended questions about their experience.

Bibliography

[1] Marc Alexa, Daniel Cohen-Or, and David Levin. As-rigid-as-possible shape interpolation. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 157–164, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[2] A. Sheffer and J.C. Hart. Seamster: inconspicuous low-distortion texture seam layout. In *IEEE Visualization, 2002. VIS 2002.*, pages 291–298, 2002.

[3] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, page 39–46, New York, NY, USA, 2007. Association for Computing Machinery.

[4] Kai Hormann, Bruno Lévy, and Alla Sheffer. Mesh parameterization: theory and practice video files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 1–es, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Bruno Levy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. *Least squares conformal maps for automatic texture atlas generation.* Association for Computing Machinery, New York, NY, USA, 1 edition, 2023.

[6] Alla Sheffer, Bruno Lévy, Maxim Mogilnitsky, and Alexander Bogomyakov. Abf++: fast and robust angle based flattening. *ACM Trans. Graph.*, 24(2):311–330, apr 2005.

[7] Patrick Mullen, Yiying Tong, Pierre Alliez, and Mathieu Desbrun. Spectral Conformal Parameterization. *Computer Graphics Forum*, 27(5):1487–1494, 2008.

[8] Zhiqin Chen, Kangxue Yin, and Sanja Fidler. Auv-net: Learning aligned uv maps for texture transfer and synthesis. In *The Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[9] Francis Lazarus and Anne Verroust. Three-dimensional metamorphosis: A survey. *The Visual Computer*, 14:373–389, 12 1998.

[10] S. Melzi, R. Marin, P. Musoni, U. Castellani, and M. Tarini. Visual Assessments of Functional Maps. In David Bommes and Hui Huang, editors, *Symposium on Geometry Processing 2019- Posters*. The Eurographics Association, 2019.

[11] K. Somani Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(5):698–700, 1987.

[12] James D. Foley and Andries Van Dam. *Fundamentals of interactive computer graphics*. Addison-Wesley Longman Publishing Co., Inc., USA, 1982.

[13] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Ugo Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.

[14] Unity Technologies. Unity asset store. `https://assetstore.unity.com/`.

[15] The Base Mesh. The base mesh - 3d models and resources. `https://www.thebasemesh.com/`.

[16] Sketchfab. Sketchfab - share and discover 3d models. `https://sketchfab.com/`.