FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master’s Thesis

A system for editing triangle mesh sequences with time-varying connectivity

Zuzana Káčereková

PILSEN, CZECH REPUBLIC

2023

FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master’s Thesis

A system for editing triangle mesh sequences with time-varying connectivity

Bc. Zuzana Káčereková

Thesis advisor
Doc. Ing. Libor Váša, Ph.D.

PILSEN, CZECH REPUBLIC **2023**

Citation in the bibliography/reference list:
KÁČEREKOVÁ, Zuzana. *A system for editing triangle mesh sequences with time-varying connectivity*. Pilsen, Czech Republic, 2023. Master's Thesis. University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering. Thesis advisor Doc. Ing. Libor Váša, Ph.D.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Zuzana KÁČEREKOVÁ**
Osobní číslo:	**A20N0059P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Počítačová grafika**
Téma práce:	**Systém pro editaci sekvencí trojúhelníkových sítí s časově proměnlivou konektivitou**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se se systémem pro sledování objemových elementů vyvíjeným na KIV ZČU.
2. Implementujte na základě vytvořených dat o sledování systém pro editaci sekvencí trojúhelníkových sítí založený na následujících technikách:
   - uživatelský vstup pohybu sledovaného objemového centra (efektoru) v GUI v libovolném snímku
   - distribuce pohybu na další centra v daném snímku podle Gaussovského rozložení v okolí efektoru
   - distribuce pohybu do dalších snímků přenesením lokálního souřadného systému do dalších snímků nalezením optimální afinní transformace efektoru a jemu blízkých center
   - deformace povrchu ve všech snímcích sekvence distribucí translačních vektorů dle vzdálenosti k nejbližším centrům
3. Analyzujte vlastnosti vytvořeného postupu a navrhněte zlepšení jednotlivých kroků tak, aby bylo dosaženo přirozenějších výsledků editace.
4. Práci důkladně otestujte a zdokumentujte.

Rozsah diplomové práce:	**doporuč. 50 s. původního textu**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování diplomové práce:	**tištěná/elektronická**
Jazyk zpracování:	**Angličtina**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce:	**Doc. Ing. Libor Váša, Ph.D.** Katedra informatiky a výpočetní techniky
Datum zadání diplomové práce:	**9. září 2022**
Termín odevzdání diplomové práce:	**18. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D. děkan	**Doc. Ing. Přemysl Brada, MSc., Ph.D.** vedoucí katedry

V Plzni dne 11. října 2022

Declaration

In Pilsen, on 18 May 2023

. . . .
Zuzana Káčereková

Abstract

Time-varying mesh (TVM) sequences are a common product of modern 3D scanning methods, which are used to represent animated 3D models. Processing TVM sequences can be challenging due to a lack of temporal correspondence between consecutive frames, which is required by many algorithms.

Using an existing system for tracking volume elements, a method for editing TVM sequences was designed and implemented as an interactive application using virtual reality.

In this work, the theoretical background required to develop the editing system is presented and its properties are analyzed. Based on this analysis, future improvements to the editing algorithm are proposed. Technical documentation of the implementation is also provided.

Abstrakt

Časově proměnné sekvence trojúhelníkových sítí (TVM sekvence) jsou častým produktem metod pro 3D skenování, které jsou využívány k reprezentování animovaných 3D modelů. Zpracování TVM sekvencí může být obtížné vzhledem k chybějící časové korespondenci mezi jejich snímky, kterou mnohé algoritmy vyžadují.

S použitím existujícího systému pro sledování objemových prvků byla navržena metoda pro editování TVM sekvencí a implementována v interaktivní aplikaci využívající virtuální realitu.

V rámci této práce jsou představeny teoretické podklady potřebné pro vyvinutí editačního systému a jeho vlastnosti jsou analyzovány. Na základě analýzy jsou pak navržena možná zlepšení použitého editačního algoritmu. Je poskytnuta také technická dokumentace implementace.

Keywords

mesh editing • time-varying mesh sequences • mesh processing • virtual reality

Acknowledgement

I would like to sincerely thank my supervisor, doc. Ing. Libor Váša, Ph.D., for his support and guidance provided throughout the course of this work. Additionally, I want to acknowledge Ing. Jan Dvořák and Ing. Filip Hácha, who provided the tracking data used in the experiments.

Contents

Introduction 1

Representing and manipulating 3D scenes has been a central focus of computer graphics since its beginning. In the early years of representing 3D models with triangle meshes, the models had to be compact and minimalistic. As computing power grew, so increased the capability of hardware to render and process increasing numbers of triangles, and with it the demand for realism.

Over time, creating realistic 3D models and animations has become extremely time-consuming, requiring trained experts who have dedicated thousands of hours to learning modeling and animation tools. This has made the possibility of capturing real-world objects along with their motion and textures attractive, due to the promise of an accelerated creative pipeline in both 3D modeling and 3D animation. This process is also called 3D scanning.

Typically, 3D models are designed as a polygon mesh, which is then deformed in order to represent motion. This produces mesh sequences with a connectivity that does not change between frames, called dynamic mesh sequences. Most existing methods for processing mesh sequences rely on this temporal connectivity correspondence property.

3D scanning, however, cannot produce such data, as in each frame the captured surface may be represented with a different number of vertices that do not correspond to vertices in the previous frames in any obvious way. As a result, 3D scanned sequence data is typically also immense in size. Since editing each frame of such data manually is not feasible and would defeat the original purpose of speeding up the creative pipeline, and existing mesh sequence editing methods typically rely on temporal correspondence, there is a demand for new TVM sequence editing methods.

In this thesis, I present the theoretical background to a recently developed method for volume element tracking, which was developed at the University of West Bohemia. Based on this method and its output, I design and develop a method for editing TVM sequences and implement an interactive editing application. Furthermore, I evaluate the properties of this implementation, suggest modifications that would improve the editing process, and document the work.

2 An introduction to mesh editing

This chapter introduces the reader to concepts in computer graphics regarding digitized representations of shapes as 3D models and the methods used to manipulate them. The text is based on information found in lecture notes on mesh processing [Váš20] and computer animation [Roh21].

Various types of 3D model representations are used across many areas with differences in demands, such as engineering, medicine, or entertainment. This chapter introduces the properties of the representations and existing editing methods. Based on these properties, the use case scenarios for the editing system are identified.

Further chapters then use the introduced concepts to describe the editing pipeline in detail, including necessary considerations regarding input data. Methods used in other areas of mesh editing are also used to achieve the desired properties of the editing system.

2.1 Representing 3D models

In order to understand the capabilities and limitations of the designed method, one must first understand how applications represent 3D models and the applications' demands for the information stored and provided by these models.

Ideally, we would like to represent all shapes with infinite precision. For most shapes, this is impossible. While we cannot definitively determine whether reality is discrete or continuous, real-life objects appear continuous up to a greater precision than we can usually afford to represent in a computer model.

Close to precise representation can be achieved by using mathematical formulations of shapes and operations on these shapes, such as boolean operations. This is a common representation in ***constructive solid geometry*** (CSG), which is typically used in engineering applications, such as component manufacturing. The exact precision is then determined by whether a finite representation of floating point numbers is used in both the model itself and subsequent mathematical operations. An example model is shown in fig. 2.1.

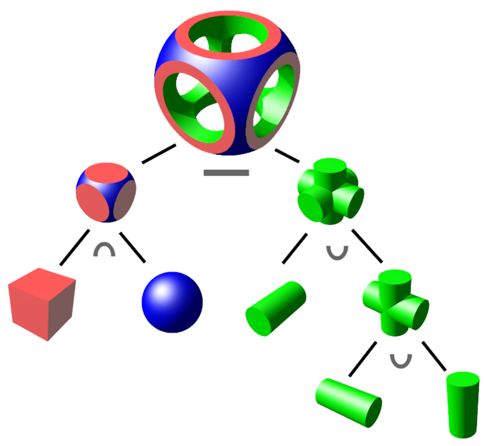


Figure 2.1: An example of an object modeled with constructive solid geometry. The object is created using two union operations, an intersection operation and a subtraction. The figure depicts a CSG tree, which is processed in a post-order traversal. Source: [Zot]

Depending on the application, a mathematical formulation of the 3D model is not always possible or even desirable. Applications that rely on capturing real-world data capture data points up to a certain precision, with the values between the captured data points remaining undefined. In such a case, it would be difficult to obtain a mathematical formulation and even then, it could only be an interpolation or approximation of the data.

While interpolations and approximations of measured data points are also used, calculating them is resource intensive. For example, using ***radial basis functions*** (RBFs) leads to a time complexity of $O(n^3)$. RBFs are commonly used to interpolate and approximate scattered data such as data produced by 3D scanning, which is characterized by having no natural ordering.

Captured data points by themselves form a ***point cloud***. However, since the points in a point cloud are not connected in any way, rendering the surface of the model is not possible without further processing. The points can be connected to form a surface by ***tessellation***. Tessellations form polygon meshes, which will be described in more detail in section 2.1.1.

Applications can also be differentiated by whether they use information about the internal structure of the represented object, or whether they use a surface representation. Applications that use the internal structure of objects can use regular or adaptive grids to store models. ***Computed tomography*** (CT) scans, which capture the internal structure of the body, are an example of a 3D model that stores data in a regular grid. Slices of the grid are typically viewed as grayscale images.

Other applications, however, may not care about the internal structure of the model. This includes 3D models created for visualization, for example in advertising, architectural rendering, or entertainment. These applications use surface models to which textures and materials are applied to achieve realistic or stylized visualizations. In all of these cases, motion is either added separately by an animator, determined by a physics system or a procedural algorithm, captured in a motion capture setup that transfers real-world movement to virtual models, or captured along with the model by 3D scanning.

In applications where realism is a goal, the process of simulating motion could benefit from having data about the internal structure of objects, especially while using physics-based motion. However, it is usually difficult or impossible to obtain such data, and if it were available, using it would require additional processing power. Since some of these applications simulate motion at runtime, rendering speed may outweigh the benefits of simulating the motion more accurately.

Capturing motion by 3D scanning bypasses concerns about motion realism since the surface is captured already in motion with high accuracy. This is done at the cost of a loss of customizability, as there is no single canonical variant of the captured scenes and models that could be further edited or animated.

There are several approaches to representing surfaces:

- implicit
- parametric
- piecewise parametric

Signed distance functions (SDFs) are an example of an implicit surface representation, as seen in fig. 2.3 in a two-dimensional case, in which the zero level set of the SDF is a curve. SDFs define distance from the surface, which is negative while inside the object and positive while outside of it. In general, an implicit surface is given by the mapping $f(x) : \mathbb{R}^m \to \mathbb{R}$.

Parametric surfaces are mappings $s : \mathbb{R}^n \to \mathbb{R}^m$. A special case of a parametric surface is a polygon mesh, which is piecewise parametric. Polygon meshes (fig. 2.4a) approximate the shape of the represented object using connected polygons.

A special case of a polygon mesh is a triangle mesh (fig. 2.4b), which is a common 3D model representation used in most 3D graphics software, including the software and algorithms described in this thesis.

2.1.1 Triangle meshes

Similarly to polygon meshes, triangle meshes consist of vertices connected with edges, which form the faces of the mesh. While polygon meshes allow any number of vertices and edges to form a face, triangle meshes only allow triangular faces.

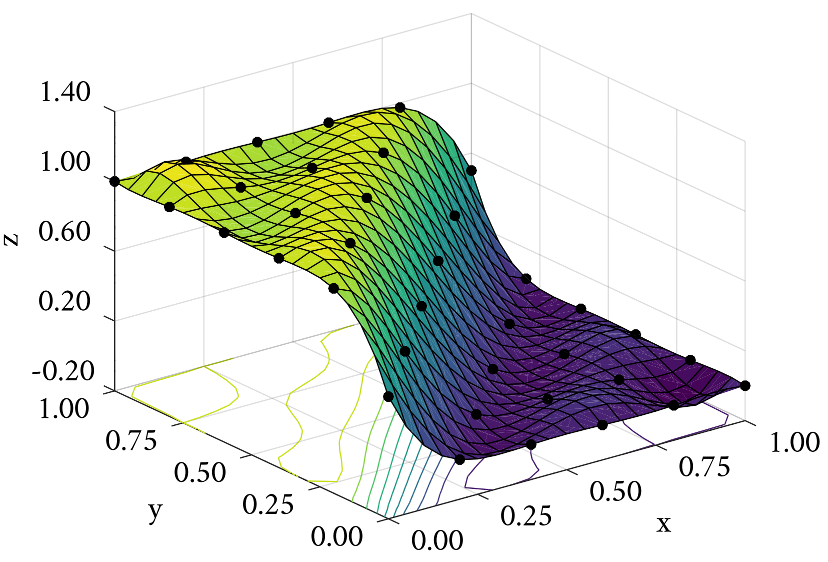


Figure 2.2: An RBF interpolated surface. Due to intentionally suboptimal parameter selection, the shape of the radial kernel functions is visible in the interpolation.

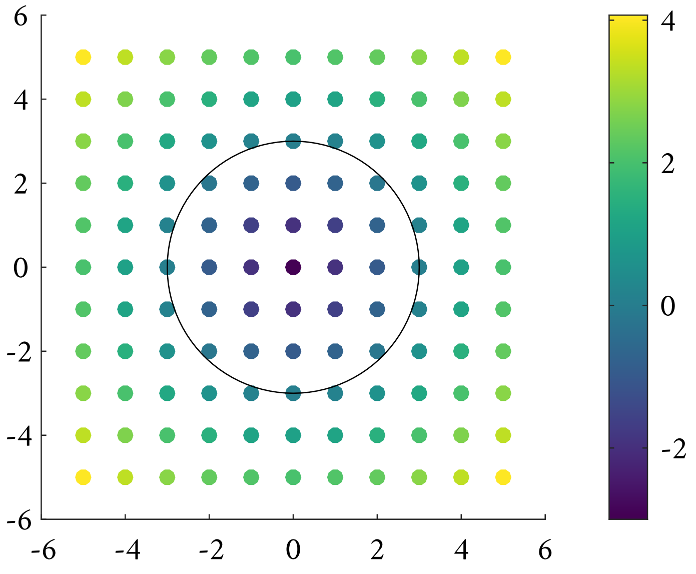


Figure 2.3: An example where distance given by the signed distance function is mapped to color. The curve represents the zero level defined by the SDF. When extended to three dimensions, SDFs can represent surfaces.

In a finite precision representation, triangular faces have the advantage of always being planar, since three different points always form a plane. When more vertices with coordinates specified with finite precision are added, they are almost certain to lie above or below the plane, making the outcome of mathematical operations potentially inconsistent.

Manifold triangle meshes are a subset of meshes in which the neighborhood of each point of the surface is isomorphic to a disc or a half-disc. This condition means that vertices are allowed to be incident with only one fan of triangles and edges are allowed to be incident with exactly two faces. Examples of meshes that do not fulfill these conditions are illustrated in fig. 2.5. A non-manifold vertex is shown in fig. 2.5a and a non-manifold edge in fig. 2.5b. The volume tracking algorithm requires meshes to be manifold.

By allowing half-disc neighborhoods, meshes with a ***boundary*** become admissible. A boundary (fig. 2.6) is formed by a closed loop of edges in which each edge is incident with only one triangle. In ***watertight*** meshes, all edges are incident with two triangles.

While watertightness is easy to achieve with meshes modeled by hand, for meshes obtained by 3D scanning, being watertight is a strong demand, especially since some parts of surfaces are likely to be obstructed at any given moment. However, an algorithm for tracking the volume occupied by a model must be able to decide whether any given point lies inside or outside of the volume. This issue is resolved in the used volume tracking software by using a generalized winding number calculation. Ideally, the input data also shouldn't contain self-intersections, which indicate scanning artifacts.

2.1.2 Mesh sequences

A mesh sequence is an ordered sequence of meshes that captures an object's motion over time. Each mesh in the sequence can be referred to as a ***frame*** of the sequence. An important parameter of the sequence is its framerate, which defines the time interval between two frames.

Mesh sequences can either be ***dynamic***, in which the vertices of the mesh change position with time but the connectivity remains constant, or ***time-varying***, in which the number of vertices can change between frames and the connectivity is typically different in each frame.

Mesh sequences can also be differentiated by the method of their creation. While TVM sequences are usually obtained by 3D scanning, dynamic mesh sequences are easy to create by deforming a canonical mesh of the represented object in a rest pose.

When this deformation is defined procedurally, for example by a set of rules and a physics system or using skinning, frames can be obtained at arbitrary intervals. Even in dynamic mesh sequences defined frame-by-frame, intermediate frames can be generated by interpolation. This is not possible for TVM sequences, since the frames have been captured at a set framerate in real-time and the lack of a temporal connectivity correspondence makes interpolating between successive frames non-trivial.

Notably, representing motion as a deformation of a rest pose mesh leads to much lower storage space requirements than when motion is represented by a sequence of static meshes. Mesh sequences can be compressed efficiently, but deformations typically require even less storage space.

In each case, additional processing is required at runtime, either decoding the compressed mesh sequence or deforming the rest pose mesh. While the deformation process can be too slow for real-time applications, it is the preferable approach to storing mesh animation, especially since it can be edited further.

2.2 Mesh editing

Mesh editing typically refers to actions that deform the vertices of the mesh, but maintain the same connectivity. Animating a rest pose mesh can be referred to as mesh editing.

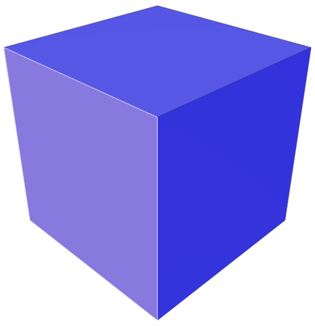
Techniques falling under mesh editing aim to modify an animation by

- modifying the animated shape over time,
- adjusting a given frame
- or modifying the rest pose.

Modifying the animated shape over time means adding a motion which extends throughout multiple frames. Adjusting a given frame then means making changes to a specific frame that make no impact on other frames of the already existing animation. Lastly, modifying the rest pose means deforming it in a way that propagates the deformations into the entire already animated sequence without making it necessary to create the animation again.

An additional demand on mesh editing techniques is that they should be easy for an animator to use, while providing results that they would intuitively expect. Editing should also preserve the details of the mesh while allowing large areas to be deformed in one action.

One way to implement an intuitive editing operation is to define ***effectors*** that are parts of the mesh, which should be moved to the position specified by the an-


(a) A quad mesh.

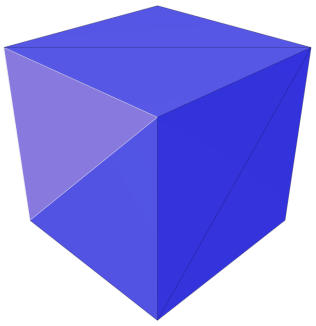
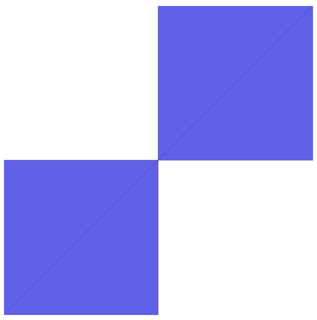

(b) A triangle mesh.

Figure 2.4: A quad mesh and a triangle mesh, each with one face highlighted.


(a) A vertex with two incident triangle fans.

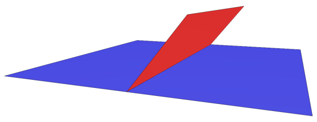

(b) An edge with more than two incident faces.

Figure 2.5: Examples of non-manifold geometries.

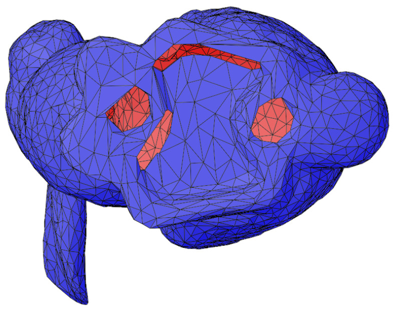


Figure 2.6: A mesh with a visible boundary. The outside of the mesh is colored blue. The inside is colored red. Model source: [TL]

imator, and ***constraints***, which are parts of the mesh that shouldn't be affected by the operation.

Any other parts of the mesh should move reasonably, in a way that is intuitive to the animator and preserves the details of the mesh. Enforcing additional conditions for the movement of the remaining parts of the mesh can create a variety of editing tools for the animator.

2.2.1 Space deformation

Editing operations can deform vertex positions either by deforming the space in which the mesh is embedded, or the surface of the model itself. The vertices of a mesh can also be referred to as the ***geometry*** of the mesh.

One example of a space deforming operation is ***free-form deformation*** (FFD). By enclosing all vertices of the mesh in a grid structure and deforming the grid, a transformation can be applied to all of the vertices simultaneously, as demonstrated in fig. 2.7. A similar approach is ***cage-based deformation***.

Alternatively, all vertices could be pulled towards a single point, or away from it, inflating or deflating space in a given area. A falloff function can also be applied to the transformation, causing nearby vertices to be pulled stronger than far away ones, making the editing operation more localized. This operation intentionally changes the volume of the edited object. If the goal is to maintain model volume, the shape can instead be deformed by a ***divergence-free vector field***. In fact, all space deformations can be modeled using vector fields.

Space deformations can be applied to most model representations, since they do not require a surface to exist, allowing point clouds to be edited without prior tessellation and volume data to be edited without surface extraction.

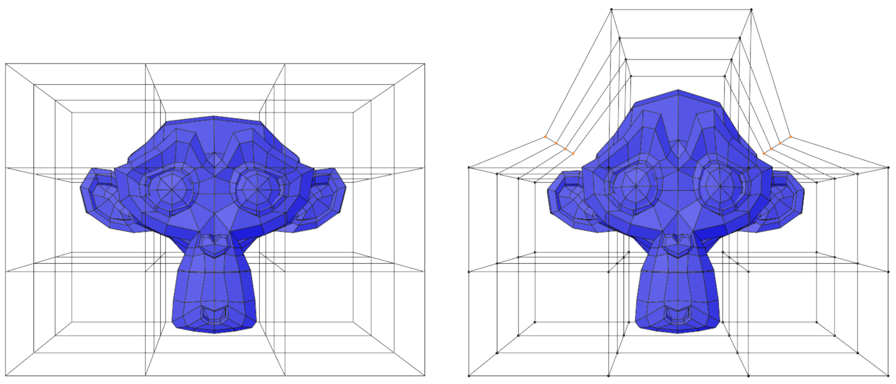


Figure 2.7: An example of free-form deformation created with Blender, which uses a 3D lattice.

2.2.2 Surface deformation

Surface based deformations simulate the behavior of elastic surfaces. The deformation is usually formulated as an optimization problem, which uses an energy function to find an optimal state of the surface (at a minimum or a maximum energy, depending on the application). Unlike space deformations, they can be formulated in a way that respects existing properties of the surface, such as its curvature.

2.2.3 Editing methods

2.2.3.1 Laplacian editing

One of the many approaches to surface deformation is ***Laplacian editing*** [Sor+04; Lip+04], which uses the coordinate Laplacian to build a system of linear equations using the constrained vertices and the existing details of the mesh.

The ***Laplace operator*** is a differential operator defined in eq. 2.1 as the divergence of the gradient of a function $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$.

$$\Delta f = div\nabla f = \frac{\partial^2 f}{\partial x^6} + \frac{\partial^2 f}{\partial y^6} + \frac{\partial^2 f}{\partial z^6} \tag{2.1}$$

Specifically when used with manifolds, the ***Laplace-Beltrami operator*** is used as per eq. 2.2, where $f : M \longrightarrow \mathbb{R}$ and $\Delta f : M \longrightarrow \mathbb{R}$.

$$\Delta_M f = div_M \nabla_M f \tag{2.2}$$

If the vector function $\mathbf{p}$ returns the Cartesian coordinates, then eq. 2.3 applies, where H is the mean curvature and $\mathbf{n}$ is the normal vector.

$$\Delta_M \mathbf{p} = div_M \nabla_M \mathbf{p} = -2H\mathbf{n} \tag{2.3}$$

In a discrete environment such as triangle meshes, the Laplacian of the coordinate function $\mathbf{p}$ can be approximated by eq. 2.4, where $N(i)$ represents the neighborhood of the vertex $\mathbf{x}_i$.

$$\Delta \mathbf{p}(\mathbf{x}_i) = \frac{1}{\|N(i)\|} \left(\sum_{j \in N(i)} \mathbf{x}_j \right) - \mathbf{x}_i \tag{2.4}$$

In Laplacian editing, a detail vector describes the deviation of a vertex from the centroid of its neighboring vertices. The constrained vertices, which should not move, as well as the constraints given by the final positions of the effector vertices, are used along with the original mesh details to determine the new positions for the remaining vertices. Fig. 2.8 shows an example of an edited mesh.

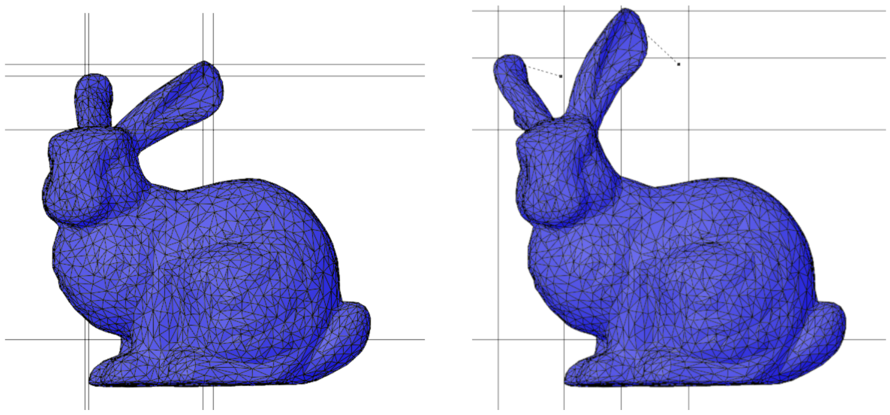


Figure 2.8: An example of Laplacian editing created with Blender. The effector movement is indicated by dashed lines.

As per eq. 2.5 and 2.6, the algorithm first calculates the details $\mathbf{d}$ of the original matrix using a Laplacian matrix $\mathbf{L}$ of the original mesh and the vertex positions $\mathbf{x}$. Then, it subtracts the constrained vertex positions $\mathbf{c}$ from the detail matrix and solves the system to obtain the new positions $\hat{\mathbf{x}}$.

$$\begin{aligned} \mathbf{d} &= \mathbf{L}\mathbf{x} & (2.5) \\ \mathbf{L}\hat{\mathbf{x}} &= \mathbf{d} - \mathbf{c} & (2.6) \end{aligned}$$

The elements of matrix $\mathbf{L}$ are given by the following schema 2.7:

$$\mathbf{L}_{i,j} = \begin{cases} -1 & i = j \\ \frac{1}{\|N(i)\|} & j \in N(i) \\ 0 & otherwise \end{cases} \tag{2.7}$$

This approach struggles with preserving details of the surface under large rotations, since it attempts to preserve the global direction of normals. The authors of *Differential coordinates for interactive mesh editing* [Lip+04] mitigate this issue by estimating local rotations and rotating the differential coordinates prior to solving the system. A better approach may be to instead preserve the Euclidean distance between the vertices of the surface. Transformations which preserve this distance are called ***rigid***.

2.2.3.2 ARAP

As Rigid As Possible (ARAP) [SA07] is a method which attempts to preserve local rigidity of transformations during deformation. While complete rigidity is not possible, since the surface does have to deform to meet the constraints, the overall non-rigidity for each vertex and its neighborhood can be minimized by this method.

The energy to be minimized by the algorithm is formulated in eq. 2.8, where $N(i) \cup i$ is the neighborhood of vertex i, including the i-th vertex itself, $\mathbf{x}_j$ is position of the j-th vertex in the original mesh, $\mathbf{R}_i$ is the optimal transformation of the i-th neighborhood and $\mathbf{x}_j$ is the transformed position of the j-th vertex. The optimal transformation $\mathbf{R}_i$ is composed of only translation and rotation with no shear or scaling. Such a transformation can be obtained using the Kabsch algorithm, which is further discussed in section 5.2.1.

$$E = \sum_{i} \sum_{j \in N(i) \cup i} ||\hat{\mathbf{x}}_j - \mathbf{R}_i \mathbf{x}_j||^2 \tag{2.8}$$

ARAP is an iterative method which first estimates rigid transformations at a local level, transforms the local details and then solves the Laplacian linear equation system for a global solution, which becomes the input to the next iteration.

While the algorithm can converge slowly for larger meshes, the results become viable after only a few iterations. The results do however depend on the mesh tessellation, achieving suitable deformation easier in more densely tessellated areas.

2.2.3.3 Mesh skinning

Humanoid and animal models are a very common type of models to be animated, and their motion can be approximated by the motion of their bone structure. As it turns out, introducing the concept of virtual bones even to the process of animating inanimate objects can efficiently describe the transformations they undergo as well.

The concept of a bone simply represents a coordinate system which can be rotated and translated along with the vertices assigned to it. In nature, skeletons are hierarchical structures - if the femur is rotated in the hip, the lower leg will follow the motion of the thigh automatically. This hierarchical structure is used in skeletal animation, allowing for easy and intuitive motion design.

The animator usually designs the skeleton manually, although automated methods for common skeletal structures exist [Au+08; Che+11]. This process is called ***rigging***. Since rotation and translation can be represented by matrix multiplication, which is fast and highly optimized, the approach can be used in real-time applications. The use of rigged 3D models is widespread, most notably in game development.

The skeleton is typically designed for the rest pose mesh. The vertices of the mesh are then assigned to the bones by weight painting. Each vertex of a mesh is assigned a weight for each bone of the skeleton. The weights represent the extent to which each bone should affect the position of a given vertex. A vertex can be influenced by multiple bones, especially in joint areas which should only deform partially. Each bone represents one separate transformation.

The application of these transformations is called ***skinning***. The transformations are executed in hierarchical order given by the order of the bones in the skeleton. However, for vertices in areas affected by multiple bones, a blending method for the transformations is required.

A simple blending method is linear blend skinning, which simply applies all of the transformations as a weighed sum, as per eq. 2.9, where $\mathbf{w}_i$ refers to the weight assigned to bone i and $\mathbf{M}_i$ to the transformation matrix of the bone i. The point $\mathbf{x}$ is from the rest pose, while $\hat{\mathbf{x}}$ is the transformed point. Unfortunately, the resulting matrix is not a rotation matrix in itself, and artifacts are introduced to the animation. This is called the ***candywrapper effect***, which can cause animated joints to twist and contract.

$$\hat{\mathbf{x}} = \left(\sum_i w_i \mathbf{M}_i \right) \mathbf{x} \tag{2.9}$$

The effect can be avoided by using ***dual quaternion skinning*** [Kav+07], which represents all transformations using dual quaternions, which can be blended more easily.

2.2.3.4 Embedded deformations

Embedded deformations [SSP07] is a method similar to skinning, in that it uses a rest pose mesh to which it applies transformations that change in every frame. However, unlike skinning, it doesn't use a bone structure. Instead, transformations influence nearby vertices. As a type of space deformation, embedded deformations can be applied to many types of model representations.

The method introduces the idea of ***deformation graphs***, which are formed by a subset of the mesh vertices. This subset can contain a fraction of the original vertices - one in twenty or even one in every hundred vertices, depending on the tessellation. The vertices are chosen at regular intervals, avoiding selecting vertices too close to each other.

The model is manipulated using the deformation graph's vertices as effectors. Using the edited positions of the effectors, the method determines the transformation of the rest of the deformation graph by optimization, which minimizes the deviation of the graph's transformation from a rotation. Rotation is desirable, since it preserves detail, unlike shearing and stretching.

Each vertex in the graph associated with a matrix $\mathbf{R}$ and translation $\mathbf{t}$ which affects the surrounding areas of the mesh, with the transformation centered at the vertex. Areas in which influence of nearby deformation graph vertices overlaps are connected with edges and their transformations are blended linearly. This can theoretically lead to artifacts, but since nearby transformations are similar, they are typically not noticeable.

The weights are assigned based on distance. For each vertex of the mesh, k nearest deformation graph vertices are found, and the weights are assigned as per eq. 2.10, where w_i and d_i are the weight and the distance of deformation graph vertex. The distance of the farthest neighboring deformation graph vertex is denoted d_k. The weights must be normalized in an additional step as per eq. 2.11, so that $\sum_i \bar{w}_i = 1$.

$$w_i = 1 - d_i/d_k \tag{2.10}$$
$$\bar{w}_i = \frac{w_i}{\sum_i w_i} \tag{2.11}$$

The method can produce undesirable deformation in overlapping areas of influence, such as when moving the arm of a humanoid model and deforming the torso as well. Ideally, geodesic distance should be used to assign nearby areas to the deformation graph vertices, however, it can be complicated to calculate. The authors instead suggest to implement tools for the user to remove undesirable areas of influence from the editing action.

2.3 Editing TVM sequences

The mesh editing methods which have been presented in this chapter can be used to define the criteria that a mesh editing system should fulfill. In this section, the criteria are summarized and used to set design goals.

From the user's point of view, the system must primarily be easy and intuitive to use. The methods achieve this by establishing some form of effectors, which act as handles, enabling easy manipulation of the mesh. By transforming the effectors, typically by specifying a translation to a new location, larger areas of the mesh should be affected at once, streamlining the editing process. In terms of implementation, the main requirement is that the editing operations preserve detail locally, preferentially composing transformations using translations and rotations and avoiding stretching and shearing.

All of the methods above are typically applied to a single rest pose mesh, which is deformed to produce frames of the sequence at arbitrary framerates. For example, when using skinning to animate a model, the user would specify a final position of a bone and the time at which the bone should reach that position. The motion would then be executed at a framerate also given by the user. Frames in which transformations are explicitly given rather than interpolated are called keyframes.

Modern software, such as Blender, also allows the user to choose whether to interpolate between the initial and final transformation linearly or using an easing function. Easing functions allow the movement to speed up or slow down between keyframes, which contributes to the look-and-feel of the animation, making it appear more natural.

Dynamic mesh sequences have additional processing requirements, seeing as that instead of a single rest pose mesh with associated deformations, they are composed of a sequence of meshes. However, the same methods can still be successfully applied to dynamic mesh sequences.

For example, a skeletal structure can be extracted from dynamic mesh sequences. This is explored in *Smooth skinning decomposition with rigid bones* [Le12]. This method uses a provided set of example poses, a rest pose and a number of bones to find a non-hierarchical set of bones (called ***proxy bones*** by the authors), a bone-weight map which assigns weights per bone to each vertex, and a set of transformations. In the case of dynamic mesh sequences, the first frame can be used as a rest pose while keyframes of the animation can be used as example poses. The number of bones is somewhat arbitrary and could be determined by trial and error as the least number of bones which provides visually satisfactory results. The problem is formulated as a constrained least-squares optimization. Other methods for extracting skeletons from dynamic mesh sequences include *Fast and Efficient Skinning of Animated Meshes* [KSO10] or *Skinning Mesh Animations* [JT05].

In contrast to dynamic mesh sequences, time-varying mesh sequences present a challenge. Throughout the sequence, the vertex counts and vertex positions change. Necessarily, this means that the connectivity also changes from frame to frame. Due to this lack of temporal correspondence between the frames, it is difficult to track the path of points on the surface throughout the animation.

These properties make processing and storing TVM sequences difficult. In the case of animation by mesh deformation, it is sufficient to store a single rest pose mesh along with the deformations that describe the motion. In the case of dynamic meshes, the first mesh and a series of vertex position differences could be stored efficiently. TVM sequences change completely in each frame, so each frame must be stored in full.

To illustrate the storage demands, storing compressed geometry requires around 10 bits per vertex (bpv), although this strongly depends on the amount of distortion considered acceptable, while storing compressed connectivity requires an average of 1 - 2.5 bpv when encoded by valence coding. These values quickly add up when temporal correspondence cannot be exploited, especially considering that 3D scanned data tends to be captured at a high resolution.

Massive savings could potentially be achieved by extracting a rest pose from TVM sequences and approximating the motion using some form of deformation. However, extracting a rest pose mesh from a TVM sequence is a non-trivial problem.

One part of this problem arises due to 3D scanning limitations, as parts of the

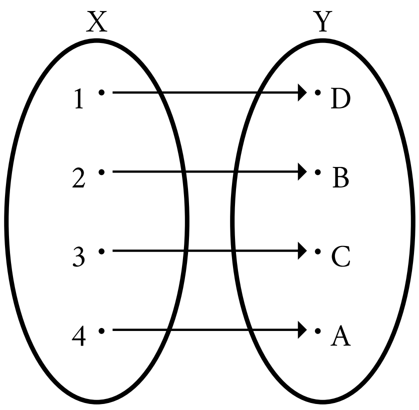


Figure 2.9: An example of a bijective or one-to-one mapping. All elements of the set $\mathbb{X}$ have exactly one image in set $\mathbb{Y}$ and all elements of set $\mathbb{Y}$ have exactly one preimage in set $\mathbb{X}$. Source: [Sch]

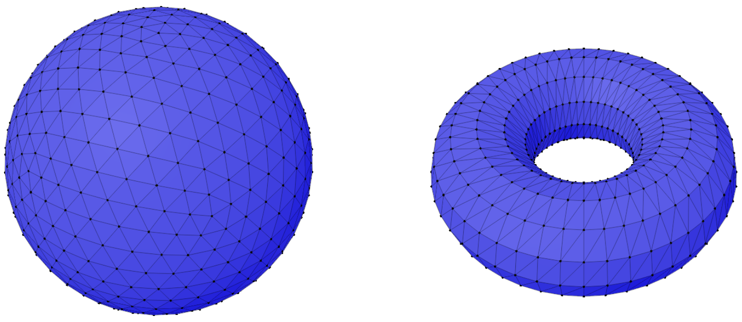
scanned surface may not be visible at all times throughout the sequence. In fact, some parts of the surface may not be captured by any frame of the sequence.

A ***bijective mapping***, shown in fig. 2.9, is a mapping of set $\mathbb{X}$ to set $\mathbb{Y}$ in such a way that elements have a one-to-one correspondence. This means that each element from set $\mathbb{X}$ maps to exactly one element from set $\mathbb{Y}$, and vice versa. The sets must therefore have the same number of elements. An attractive approach to obtaining bijective mapping between successive frames would be by adding the missing areas of the surface to all frames of the sequence.

However, determining the precise positioning of the missing surface in all frames, including frames in which the surface is occluded due to self-contact, is a difficult problem. After adding the missing surface areas, the exact bijective mapping between successive frames would still have to be established, either by re-sampling the surface or by matching vertex pairs, which could lead to further issues in surface areas which should correspond between successive frames but have been sampled with different amounts of vertices. An exact solution to this problem is not obvious.

An even more prohibitive issue with establishing a mapping between frames is the possible change of genus. In topology, ***genus*** (fig. 2.10) is a property of surfaces that determines the number of holes in a surface. Holes should not be confused with boundaries. While a boundary is a manifestation of a scanning artifact, which produces a surface edge that is impossible in physical reality, a hole is a physical property of the represented object. An example of an object with a genus of zero would be a simple sphere (fig. 2.10a), while a torus (the "donut shape", fig. 2.10b) has a single hole and therefore a genus of one.

Only deformations of surfaces with the same genus can be represented using a triangle mesh with constant connectivity. While temporal correspondence between the vertices of successive frames could be established in a scenario in which self-



(a) A sphere. (b) A torus.

Figure 2.10: A sphere is a genus zero surface, since it has no holes. A torus, which has one hole, is a genus one surface.

contact during 3D scanning changes the surface genus, changes in connectivity would be necessary, once again limiting the potential to exploit this correspondence. This issue is not negligible, since changes in the genus of 3D scanned surfaces are not uncommon.

These issues can be mitigated by taking an alternative approach that instead of tracking elements of the surface tracks the volume occupied by the model [DVV21; Dvo+22]. This assumes that the input model only changes in volume negligibly under deformation, however, most practical inputs fulfill this condition.

If the volume occupied by the model is divided into volume elements which are then tracked throughout the sequence, temporal correspondence can be established between successive frames. This correspondence can then be used to determine which areas of the surface move coherently. Therefore, motion throughout the sequence can be tracked.

For the method to be applied, it is necessary to determine the space occupied by the volume, which suggests issues with input meshes containing boundaries. However, this issue can be mitigated using the generalized winding number calculation described in section 3.2.1.1.

The tracking data obtained by a volume element tracking method are suitable for organic shape representation commonly used by the game and entertainment industries, advertising, and similar areas, rather than engineering applications, which require high precision and rigorous model definitions. This is ideal, since the areas already primarily use triangle meshes for model representation.

Volume tracking data has been used in mesh compression [Dvo+23] and is suitable for use in other fields, such as remeshing, mesh simplification or mesh editing. Specifically, it would be well suited for organic editing operations with a focus on artistic intention and providing simple and intuitive tools for animators, rather than precise vertex positioning. Editing operations modeled by volume repositioning would fall in the space deformation category.

A system for tracking volume elements has recently been developed at the University of West Bohemia (UWB) and has been made available for the purposes of this work. The editing system developed in this work, therefore, uses the tracking data produced by the tracking system as an additional input to the input TVM sequences. In the following chapter, the volume tracking algorithm is described in detail.

Volume tracking for TVM sequences 3

In this chapter, the algorithm developed in *Towards Understanding Time Varying Triangle Meshes* [DVV21] and *As-rigid-as-possible volume tracking for time-varying surfaces* [Dvo+22] is first introduced and then described in detail.

3.1 Overview

As has been mentioned in the previous text, the tracking system exploits the correspondence of ***volume elements*** throughout a TVM sequence. For this to be possible, the input must fulfill several conditions.

Most importantly, the volume must remain constant throughout the sequence or change negligibly, since disappearing and reappearing volume elements would necessarily have no corresponding elements in successive or previous frames. The input data must also be able to provide volume occupancy information, either by indicating a simple binary response of "occupied" or "not occupied", or a continuous response representing the possibility that the given area is occupied. The ability to obtain a continuous answer extends the range of possible inputs by TVM sequences containing boundaries, since ordinarily only watertight and manifold meshes could provide a binary answer to the volume occupancy query.

Since 3D scanning technology captures the surface of objects rather than directly the occupied volume, the surface representation data must first be transformed into a volumetric representation using the volume occupancy query in a step that can be referred to as ***voxelization***. A voxel is the equivalent of a pixel in a 3D environment, representing the smallest element of a grid.

In the next step, the volume is subdivided into volume elements, called ***centers***, which are represented by the location of the volume centroids. The centers should be distributed over the occupied volume uniformly, forming elements roughly similar in size. The number of centers is constant throughout the sequence, allowing temporal correspondence to be established and movement of the centers to be tracked between frames.

This structure represents the shape in a manner similar to a deformation graph or a skeleton, however, unlike the hierarchical structure of a skeleton or the edges of the deformation graph, there is no relationship between the centers at this step. Relationships between the centers are established using ***center affinity***, which the authors formulate. Roughly, center affinity describes which centers move together. Affinity is strong in neighborhoods, however, not all neighboring centers belong to the same part of the overall shape, and may not therefore move uniformly. This possibility must be captured by the center affinity as well.

Once all frames have been converted to a volume representation and the centers have been distributed throughout the volume in the first frame, the algorithm extrapolates center positions for the next frame linearly as an initial guess. The center positions then undergo an optimization process based on energy minimization using center affinity. The formulation of this energy will be discussed in detail in section 3.2.3.

This process continues, extrapolating positions from each previous frame into each following frame. After the center positions in a given frame have been finalized, center affinity is updated using new observation of the centers' movement, becoming theoretically more accurate as the sequence progresses.

The entire algorithm can therefore be divided into four steps, which are described in the remainder of the chapter. The steps are the following:

- voxelization,
- uniform volume sampling,
- optimization,
- and center affinity update.

3.2 Algorithm

In this section, each step of the algorithm is described in detail.

3.2.1 Voxelization

In the first step of the algorithm, the input TVM sequence data is converted frame-by-frame into volume data. This is a separate processing step which can be fully completed for all frames of the sequence before progressing to volume sampling and optimization.

Since the remainder of the algorithm takes a number of parameters and is likely to be ran several times while searching for an appropriate combination of the parameters, it is reasonable to fully complete voxelization and save the voxelized data

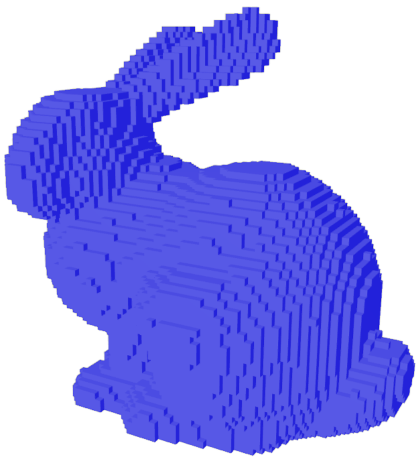


Figure 3.1: An example of a voxelized mesh generated in Blender using a resolution of 2^6. At this render size and the resolution of 2^9 voxels per side, which is used by the volume tracking system, the voxels are nearly indistinguishable from a smooth surface.

separately, so that it can be reused in the following steps without repeated processing.

Each frame is voxelized independently. The authors use a regular lattice with a resolution of 512 voxels along the longest axis, which divides the model's axis-aligned bounding box. An example of a voxelized mesh can be seen in fig. 3.1. This grid contains indicator function values. An indicator function is a positive-valued function at points in which a property is indicated and zero valued in points where the property is not present. In this case, the indicated property is whether a given voxel is a part of the model volume.

The algorithm then proceeds one of two ways, depending on whether the model is watertight or contains boundaries. For watertight models, entire columns of the grid are processed simultaneously using an axis-aligned ray cast. The intersections of the model and the ray are determined, and the indicator function value is filled in appropriately for the whole column. A ray is cast for each column of the grid, which can be done in parallel, increasing the voxelization speed.

For models with a boundary, the ray-cast method cannot be applied as some parts of the surface are missing and no intersection with the ray would be found. The authors instead use the ***generalized winding number*** to determine volume occupancy.

3.2.1.1 Generalized winding number

The generalized winding number w_g [JKS13] is defined as per eq. 3.1, where $\mathbf{q}$ is the queried point for which we would like to obtain the volume occupancy information,

F is the set of all the faces in the mesh, t is the currently evaluated triangle and $\Omega_t(\mathbf{q})$ is the signed solid angle between triangle t and the point $\mathbf{q}$.

$$w_g(\mathbf{q}) = \frac{1}{4\pi} \sum_{t \in \mathbf{F}} \Omega_t(\mathbf{q}) \tag{3.1}$$

The volume tracking system uses a more recent variant of the calculation introduced in *Fast Winding Numbers for Soups and Clouds* [Bar+18] as the ***fast winding number*** w_f. The authors of the tracking system evaluate the fast winding number in each voxel of the grid to determine the value of the indicator function *IF*, which is then given by the schema 3.2.

$$IF(\mathbf{q}) = \begin{cases} 1 & w_f(\mathbf{q}) > 0.5 \\ 0 & otherwise \end{cases} \tag{3.2}$$

3.2.2 Uniform volume sampling

In this next step, the algorithm processes the grid containing the indicator function values and attempts to distribute centers throughout the occupied volume uniformly. This section describes the process for the first frame, since further frames use an extrapolation of previous frames as an initial guess and bypass the processing necessary to initialize the first frame.

Each of the centers is associated with a set of voxels, for which this center is the nearest center. This set of voxels becomes associated with this center and its movement. The size of this set decreases with a growing number of centers used to track the volume, and the greater precision is achieved in tracking the volume elements. However, additional centers lead to increased processing time and memory requirements. The authors of the method choose to use 1000 centers as a value that has proven suitable in the experiments, maintaining a useful resolution while keeping the processing demands reasonable.

As an initial guess, the centers are distributed randomly within the volume, that is, within voxels where $IF(\mathbf{x}) = 1$, $\mathbf{x}$ being the voxel center points. Then, in a single pass over the entire volume, voxels with a positive indicator function are assigned to the center nearest to them, forming each center's set of nearby affected voxels. A kd-tree is used for finding nearest neighbors efficiently. Kd-trees are discussed further in section 5.2.2. From this set of nearby voxels, centroid positions are calculated by averaging the voxel positions, and centers are shifted towards the obtained centroid positions. This process is repeated, finding the updated nearest centers and updating centroid positions until convergence, which is achieved either by completing a sufficient number of iterations, or reaching a point at which no centroid position moves by a distance greater than a set threshold, thus achieving sufficient precision.

This positioning is then used as the initial uniform volume sampling from which further frames will be derived. This makes the result of the entire algorithm strongly dependent on initial center positioning. Since the first step of the sampling process is random and results for the first frame differ based on the initial random center positions, either the random seed or the initial guess for the first frame must be saved to achieve replicability in the later stages of the algorithm.

The approach described here is reminiscent of Lloyd's iterative algorithm for creating centroidal Voronoi tessellations, however, this algorithm differs in that only occupied space is included in the calculation and therefore the cells of voxels belonging to each center can be non-convex and even non-continuous. For Voronoi cells V_i associated with the i-th center, deviation from uniformness D_U can be quantified as per eq. 3.3.

$$D_U = \sum_i \int_{\mathbf{x} \in \mathbf{V}_i} \|\mathbf{c}_i - \mathbf{x}\|^2 \tag{3.3}$$

The authors formulate deviation from uniformness E_U as per eq. 3.4 with respect to the irregular cell shapes. Here, $\mathbf{C}$ is the set of all centers, $\mathbf{c}_i$ denotes the i-th center and $\bar{\mathbf{x}}_i$ denotes the center of mass of the voxels associated with center $\mathbf{c}_i$. The value E_U represents an uniformness energy to be minimized by the algorithm in the next step.

$$E_U = \frac{1}{2} \sum_{\mathbf{c}_i \in \mathbf{C}} \|\mathbf{c}_i - \bar{\mathbf{x}}_i\|^2 \tag{3.4}$$

3.2.3 Optimization

This section formulates the energy which is used in the iterative optimization process, which finds center positions in each frame of the sequence. In addition to the uniformness energy E_U, a smoothness energy E_S is formulated. Both energies are then considered by the optimization process, forming an overall energy.

3.2.3.1 Smoothness energy

While the positions of the centers in the first frame have been initialized using a degree of randomness, the centers must move smoothly and coherently throughout the rest of the sequence in order to produce the desired volume element tracking behavior. This necessitates the formulation of a way to propagate the centers throughout the sequence in a manner which would respect the underlying transformations of the modeled volume.

Neighboring elements of the volume are connected and should therefore travel together. In reality, parts of the volume will not always be connected rigidly - stretching is necessary especially in joint areas, otherwise objects would not deform throughout the sequence but rather only be rotated and translated, which is an uninteresting class of motion at a global level. Locally, however, transformations should be rigid, which can be exploited.

Using the optimal rigid transformation $\mathcal{A}_i$ corresponding to the i-th center $\mathbf{c}_i$, which transforms the set of centers affine with $\mathbf{c}_i$ optimally, we can define the propagated position $\mathbf{p}_i$ of the i-th center $\hat{\mathbf{c}}_i$ from a previous frame towards the current frame. Eq. 3.5 describes this transformation which uses a rotation matrix $\mathbf{R}_i$ and a translation vector $\mathbf{t}_i$.

$$\mathbf{p}_i = \mathcal{A}_i(\hat{\mathbf{c}}_i) = \mathbf{R}_i\hat{\mathbf{c}}_i + \mathbf{t}_i \tag{3.5}$$

Given this transformed position $\mathbf{p}_i$, the smoothness energy E_S can be formulated as per eq. 3.6.

$$E_S = \frac{1}{2}\sum_{\mathbf{c}_i \in \mathbf{C}} ||\mathbf{c}_i - \mathbf{p}_i||^2 \tag{3.6}$$

The transformation $\mathcal{A}_i$ can be obtained using a weighed variant of the Kabsch algorithm, which is further discussed in section 5.2.1. The transformation is calculated as per eq. 3.7, where w_{ij} represents the affinity of the centers and μ represents an affinity threshold, below which centers with minimal affinity are not considered. The authors set this threshold at a value of $\mu = 0.001$. SO(3) denotes the group of rotations in $\mathbb{R}^3$.

$$(\mathbf{R}_i, \mathbf{t}_i) = \underset{\mathbf{R}\in \mathrm{SO}(3), \mathbf{t}\in\mathbb{R}^3}{\arg\min} \sum_{w_{ij}\geq\mu} w_{ij}\, ||\mathbf{c}_j - (\mathbf{R}\hat{\mathbf{c}}_j + \mathbf{t})||^2 \tag{3.7}$$

The initialization and update of center affinities w_{ij} is discussed in 3.2.4.

3.2.3.2 Overall energy

Formulating the overall energy to be minimized is as simple as $E = E_S + \beta E_U$, adding the terms and using a weighing constant β, which controls which of the parameters has the greater impact on the outcome. The authors use a default value of $\beta = 1$, not prioritizing either term.

The authors of the method estimate the gradient of the uniformness energy and the smoothness energy as eq. 3.8 and 3.9 respectively.

$$\frac{\partial E_U}{\partial \mathbf{c}_i} \approx \mathbf{c}_i - \bar{\mathbf{x}}_i \tag{3.8}$$

$$\frac{\partial E_S}{\partial \mathbf{c}_i} \approx \mathbf{c}_i - \mathbf{p}_i \tag{3.9}$$

From these gradients they derive eq. 3.10, a closed-form solution for the location of the center $\mathbf{c}_i$.

$$\mathbf{c}_i = \frac{\mathbf{p}_i + \beta \bar{\mathbf{x}}_i}{1 + \beta} \tag{3.10}$$

However, since the position of centers $\mathbf{c}_i$ affects the centers of mass $\bar{\mathbf{x}}_i$ and the transformed center locations $\mathbf{p}_i$, an iterative solution is still necessary. Center positions are updated until a given number of iterations is reached, or until both of the gradient values are small enough for every center $\mathbf{c}_i$.

3.2.4 Updating center affinity

In the previous step, the existence of weights w_{ij}, which capture the affinity between centers i and j, was assumed. However, at the start of sequence processing, there is no such information available and must be initialized.

Since no motion data is available for the first frame, center affinity is approximated based on the distance of centers, assuming that centers which lie close to each other are probably physically connected and move together, although this is not fully correct. For example, in the case of an animation of a walking human, the legs move in opposing directions throughout the sequence. A first frame in which the legs would be just in the middle of passing each other by would incorrectly determine that the centers on the inner sides of the legs should move uniformly. This should quickly be corrected for successive frames.

In the first frame, the Gaussian function is used to assign a proximity-based affinity a^p_{ij} to neighboring centers, which starts strong and falls off with distance, as defined in eq. 3.11.

$$a^p_{ij} = \exp\left(-\sigma_p \cdot \|\hat{\mathbf{c}}_i - \hat{\mathbf{c}}_j\|^2\right) \tag{3.11}$$

In following frames the affinity can be further informed by the dissimilarity of the rigid transformations from the previous frame, $d_i(\hat{\mathcal{A}}_i, \hat{\mathcal{A}}_j)$. This defines a new affinity for centers in motion, described by eq. 3.12.

$$a^m_{ij} = \exp\left(-\sigma_m \cdot d_i(\hat{\mathcal{A}}_i, \hat{\mathcal{A}}_j)^2\right) \tag{3.12}$$

The falloff rate σ is defined using parameters ρ_p and ρ_m for convenience. The parameters represent the distance at which the Gaussian function reaches values 0.5 and 0.01 respectively. The falloff rate is then calculated as $\sigma_p = -\ln(0.5)/\rho_p^2$ and $\sigma_m = -\ln(0.01)/\rho_m^2$.

The authors evaluate the dissimilarity d_i of transformations using the set of voxels $\mathbf{V}_i$ which forms the volume cell corresponding to center $\mathbf{c}_i$. If each voxel is denoted $\mathbf{v}_k$, then d_i can be written as

$$d_i(\mathcal{A}, \mathcal{B})^2 = \frac{1}{|\mathbf{V}_i|} \sum_{\mathbf{v}_k \in \mathbf{V}_i} \|\mathcal{A}(\mathbf{v}_k) - \mathcal{B}(\mathbf{v}_k)\|^2 . \tag{3.13}$$

As shown in *On evaluating consensus in RANSAC surface registration* [HDV19], the computational complexity of evaluating transformation dissimilarity does not depend on the number of voxels. the authors of the tracking system exploit this in their implementation.

The last feature of center affinity is that it should be maintained through time rather than completely redefined in each frame. Towards this goal an infinite impulse response (IIR) filter can be used, which combines past values with the new. The filtered affinity values $\tilde{a}_{ij}$ can then be defined as

$$\begin{aligned} \tilde{a}_{ij}^{0} &= a_{ij}^{p} , \\ \tilde{a}_{ij}^{f \neq 0} &= \alpha a_{ij}^{m} + (1 - \alpha) \tilde{a}_{ij}^{f-1} . \end{aligned} \tag{3.14}$$

Here the parameter $\alpha \in \langle 0, 1 \rangle$ represents the strength of the IIR filtering. In the experiments, the authors use $\alpha = 0.01$. The superscript refers to the frame number f. Finally, the weights used in the optimization step are computed as follows:

$$\begin{aligned} w_{ij}^{0} &= a_{ij}^{p} , \\ w_{ij}^{f \neq 0} &= a_{ij}^{p} \tilde{a}_{ij}^{f} . \end{aligned} \tag{3.15}$$

Proposed method 4

This chapter proposes a four-step pipeline for editing TVM sequence data, typically captured via 3D scanning. The steps of this pipeline are designed to be executed independently and when needed, their specific algorithms should be replaceable with other, more advanced implementations. Such improvements are proposed later in the work in section 6.4 based on the results of the analysis.

The input sequences must be processed by the previously described volume tracking algorithm, which produces tracking data that is then also provided as an input to the editing system. For the volume element tracking to work, the TVM sequences must meet requirements which have also been described earlier in this work. Primarily, the sequences must consist of manifold triangle meshes, in which a boundary is permissible. Due to the dependency on the volume tracking algorithm, the editing system inherits this limitation.

The system should attempt to adhere to the principles which have been presented along with the existing editing methods, that is, it should attempt to create an easy to use and intuitive editing environment with predictable behavior.

The steps of the pipeline are the following

- introducing motion,
- distributing motion to other centers,
- distributing motion to other frames,
- and deforming the surface.

In the first step, the animator executes an editing operation by manipulating an effector. In a simple case, this effector can simply be one center which is dragged to a new location by the animator.

This action launches a cascade of events, in which the translation of one center is distributed to other centers within the same frame. Here, the manner of distributing the translation will affect which centers will move along with the effector and to what extent.

Next, the motion of centers executed within one frame is distributed to all of the previous and following frames. The center positions are now changed in every frame of the sequence.

In a final step, the surface is deformed in every frame based on the changes to the tracking data, which deform the space surrounding the tracked centers. At this stage, the sequence is deformed in every frame.

Through this pipeline, interesting effects should be achievable, for example such as slimming or fattening a model throughout the sequence, or changing some of their physical features - for example growing a hump back or enlarging the nose of a model.

4.1 Introducing motion

In the most trivial case, a motion $\mathbf{t}_i$ can be introduced into the sequence by changing the position of a single center $\mathbf{c}_i$. Theoretically, multiple centers could also be moved in this step, but for simplicity, the method should initially expect one center to move at a time. This center becomes the effector.

Less trivial editing operations could be implemented in further work, such as the ability to move multiple centers at a time, an operation for shrinking or growing the volume of the cell associated with a given center, or an editing operation which could allow for manual painting of the weights which determine how nearby centers react to the translation of the effector. This could allow for locking some centers in place when their movement would be undesirable. A rotation editing operation could be also implemented in future work, rotating the neighborhood of the effector.

4.2 Distributing motion to other centers

The motion $\mathbf{t}_i$ of the i-th center must next be distributed to other centers within the same frame of the TVM sequence. A simple way to achieve this is to assign the same motion vector $\mathbf{t}_i$ to all other centers in the frame, multiplied by a factor w_j calculated using the Gaussian function using user-specified parameters. The scale of this function should be defined using a σ parameter, or by specifying an effective radius, similarly to the method described in section 3.2.4.

The Gaussian function creates an area of editing effect. The effect decreases with the distance from the original moved center. For effective editing, the user must be able to control the falloff effect of the σ parameter of the function, either directly or indirectly by setting the distance at which the weights w_j should become negligible. In eq. 4.1 for determining the editing weights, $\mathbf{c}_i$ refers to the effector center and $\mathbf{c}_j$ refers to the center for which the weight w_j is being determined.

$$w_j = \exp\left(-\sigma \cdot \|\mathbf{c}_i - \mathbf{c}_j\|^2\right) \tag{4.1}$$

The motion of center $\mathbf{c}_j$ should therefore be described as $\hat{\mathbf{c}}_j = \mathbf{c}_j + w_j\mathbf{t}_i$, where $\hat{\mathbf{c}}_j$ is the new position of the center $\mathbf{c}_j$ after editing.

As that the Gaussian function can be applied at any distance from the center, the behavior is well-defined for the entire domain, creating no sudden jumps at the edge of a given editing radius.

However, the user might conceivably prefer for distant centers not to move at all. To achieve this, a threshold parameter could be introduced. This would create a slight discontinuity at the edge of the domain, causing some centers to be affected by editing, even if minimally, and their neighbors not to be affected at all. However, even this discontinuity could be exploited by a creative user, forming a different class of deformations with a hard edge.

The impact of editing could also be restricted to nearby centers using compactly supported functions. Such functions have nonzero values only on the $\langle 0, 1\rangle$ interval. Wendland's functions [Fas07] are an example of a family of compactly supported functions.

For the purposes of the analysis, implementing the basic variant of falloff with no thresholding is most important, since this variant is more likely to preserve local detail, while hard-edge editing operations are in a certain sense more destructive. This work aims mostly at achieving smooth deformations, which preserve surface properties, rather than operations, which could be intentionally destructive.

4.3 Distributing motion to other frames

Once the motion has been distributed to all of the affected centers within one frame, their motion should be distributed to other frames. Several approaches to frame-to-frame motion distribution are possible:

- distribution, which transfers motion between two successive frames by finding the nearest neighbors of the effector in the current frame and transferring the motion to the corresponding neighborhood in the next frame (frame $A \longrightarrow B$, $B \longrightarrow C$),
- deformation, which finds the neighborhood of the effector in the edited frame and transfers the motion to the same neighborhood in each next frame (frame $A \longrightarrow B$, $B \longrightarrow C$),
- or a deformation, which finds the neighborhood of the effector in the edited frame and transfers the motion from the edited frame to all other frames of the sequence (frame $A \longrightarrow B$, $A \longrightarrow C$).

The approaches differ slightly in their expected behavior. Since centers are expected to travel throughout the sequence in a unified fashion, the first approach should be similar to the remaining two approaches. However, neighborhoods do change in practice, and an implementation using this approach could pick up the motion of nearby centers which had not been edited in the originally edited frame.

The remaining two approaches differ minimally. If the neighboring centers were transformed fully rigidly throughout the sequence, the approaches would be equivalent. However, this is not the case, and transformations between neighboring frames are likely to be closer to rigid than transformations between distant frames. Therefore, the last approach was chosen for this method and the motion is distributed frame-by-frame, using the affected frame as an origin point from which motion is distributed backwards through the sequence, where the current frame is the source frame and the previous frame is the target frame, while simultaneously also distributing the motion forward through the sequence, where the current frame is the source and the next frame is the target.

In this step, the center temporal correspondence provided by the volume tracking is vital, as it establishes the natural movement of the volume between the frames. For each center, a neighborhood of n centers can be found. By comparing their current locations with their locations in the following frame and extracting an optimal rotation $\mathbf{R}^f$, which would achieve this natural movement of centers between the frames, the natural movement can be extracted. The index f refers to the target frame index.

The translation vector $\mathbf{t}_i$ of the edited center can then be transformed using this naturally occurring rotation, in principle maintaining the natural motion but extending it by features added by the animator. In the target frame, the neighborhood of the edited center can once again be used to determine the next optimal transformation and repeatedly transform the translation vector $\mathbf{t}_i$, until it has been carried to the beginning and the end of the sequence. The transformed vector $\mathbf{t}_i$ can be denoted $\mathbf{t}_i^f$, signifying that it had been transformed from the coordinate system of its neighborhood in frame $f - 1$ to the coordinate system of the frame f. In each frame, the vector $\mathbf{t}_i^f$ can then be used to determine center deformation using the same method that had been used to determine center deformation in the frame that had originally been edited. The recursive process of calculating the vector $\mathbf{t}_i^f$ in each frame is described in eq. 4.2 and shown in the diagram in fig. 4.1.

$$\mathbf{t}_i^f = \mathbf{R}^f \mathbf{t}_i^{f-1} \tag{4.2}$$

The optimal rigid transformation can be calculated using the Kabsch algorithm, which will be further discussed in section 5.2.1 on the implementation of this method. Nearest neighboring centers can be found using a kd-tree, which is a data

$$\begin{tikzcd}
\boxed{\mathbf{t}_i^0} \arrow[r, phantom, "\cdots"] & \boxed{\mathbf{t}_i^{f-1}} & \boxed{\mathbf{t}_i^{f}} \arrow[l, "\mathbf{R}^{f-1}"'] \arrow[r, "\mathbf{R}^{f+1}"] & \boxed{\mathbf{t}_i^{f+1}} \arrow[r, phantom, "\cdots"] & \boxed{\mathbf{t}_i^{n}}
\end{tikzcd}$$

Figure 4.1: A diagram of the motion distribution process. Optimal rotation $\mathbf{R}^f$ is used to progressively deform the original translation vector $\mathbf{t}_i$.

structure that allows for space partitioning suitable for speeding up neighborhood queries. It will be described in more detail in section 5.2.2.

Notably, this part of the pipeline in principle does not care about how motion is distributed from the center which had been edited originally. It simply finds the rigid transformation, which places the edited point into the coordinate system of the next frame. This means that if the editing action can be defined by the starting position and rotation of the edited center and its final position and rotation, it is fully defined for the target frame as well, regardless of the specific algorithm for motion distribution within the frame. This is important for future implementation of functions such as rotation editing, which may not need to replace this component of the pipeline, although that depends on the specifics of the editing action.

Other editing actions, such as actions which could shrink or increase the volume around a point, would likely have to reimplement this pipeline step as well, since distributing translation vectors throughout the sequence would not be relevant. In this specific case, distributing the editing action would likely be very simple, since the volume corresponding to each center is well defined in each frame and a volume scaling parameter would remain constant throughout the sequence, requiring no special calculation to distribute it.

4.4 Deforming the surface

Finally, the motion of the volume elements throughout the entire sequence can be used to deform the triangle meshes in each frame of the sequence. At this stage, the volume elements have been successfully deformed in every frame and their repositioning should translate into changes of the surface.

In order to deform the surface itself, nearest neighboring centers to each vertex of the mesh can be discovered, and a translation can be determined by weighing the motion of these centers from their unedited positions to their edited positions. The weights can be determined in a manner similar to eq. 2.10 and 2.11 in embedded deformations, weighing the influence of each center based on its distance from the vertex. By moving every vertex on the surface in this manner, the entire surface is deformed.

The deformed position $\hat{\mathbf{v}}_i^f$ of a vertex $\mathbf{v}_i^f$, the i-th vertex in frame f, can be calculated as per eq. 4.3. Here k denotes the number of nearest neighboring centers to the vertex used by the method, $\bar{w}_j$ is the normalized weight of the edited center $\hat{\mathbf{c}}_j^f$ as per eq. 2.11 and $\mathbf{c}_j^f$ is the position of the same center prior to the sequence deformation.

$$\hat{\mathbf{v}}_i^f = \mathbf{v}_i^f + \sum_{j=0}^{k-1} \bar{w}_j (\hat{\mathbf{c}}_j^f - \mathbf{c}_j^f) \tag{4.3}$$

However, this step introduces potential problems. An initial issue is the integer number of neighbors which affect the deformation, which is not only tracking-dependent, but also somewhat arbitrary, requiring manual parameter tuning. The experiments should test the impact of the choice of the neighborhood size and improvements should be suggested in the analysis, with possible focus on the mitigating the difficulty of choosing the parameter.

Once again, the simplest variant of the algorithm should be implemented and the above listed issues should be considered during analysis. Since this part of the pipeline does not depend on the specific choice of algorithms used to modify the volume elements, observations of its effects on the surface should apply regardless of changes to the preceding pipeline steps. While modifications to the previous parts of the pipeline could lead to better volume element distribution and placement, the quality of the final product, i.e., the resulting edited time-varying mesh sequence could be improved independently.

By improving this part of the pipeline in further work, the edited center positions could be re-processed to produce potentially higher quality results. Similarly, any future software implementing this editing method could introduce new variants of this pipeline step with the possibility to apply it to existing animation. This is especially desirable in the commercial sphere. Applied to products, such as games or animated movies, remastered versions of the media could be created simply by updating the existing animation and rendering the product, requiring minimal manual intervention and artist involvement.

Implementation 5

The application was implemented using the Unity engine version 2021.3.16f. The Unity engine provides tools for advanced graphics rendering as well as user interface (UI) creation. The engine also supports virtual reality (VR) integration. Additional information about the implementation is available in the attached user documentation in appendix A and programmer documentation in appendix B.

Initially, the application was developed as a desktop application with 3D graphics. Screen captures from the original version of the application can be seen in fig. 5.1 and 5.2. The model was placed in a scene which could be manipulated using a camera that could freely fly through the scene and rotate. The camera could approach the model from any angle, however, some of the centers would be hard to view and manipulate.

In this version of the application, centers could only be translated within the camera view plane. This is sometimes exploited in other modeling applications, since aligning the view plane with an axis can lead to easy base model creation with precise vertex positioning. In an application focused on organic editing, this is less desirable, since the flattened shape of the 3D model on a 2D screen is more difficult for the animator to manipulate. The animator has to keep a mental model of the shape of the surface and attempt to find the correct plane with which to align the camera in order to make the desired changes. This often leads to unsatisfactory results once the model is viewed from other angles.

In applications such as Blender, sculpting surfaces is possible and emulates the real life process of clay sculpting. An inexperienced Blender user with experience in clay sculpting can, however, easily struggle with the provided tools, since unlike in real life, they are constrained to using them only in the view plane. These limitations only become more apparent once the internal structure of an object can be edited, rather than just the surface. A user reaching into a volume and manipulating it would have to maintain a mental model of the positions of all centers within the volume, at best assisted by implied depth by lighting and depth-based coloring. Some editing actions may even be impossible if the view of a given center is obstructed in the desired editing plane and the user cannot set a depth at which the cursor should

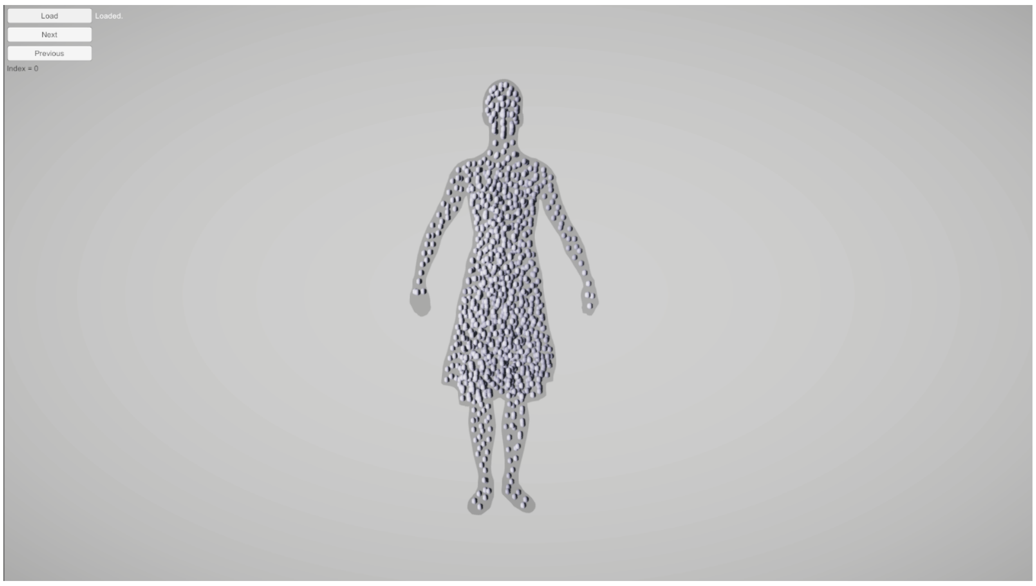


Figure 5.1: A screen capture of the original application concept which was abandoned in favor of VR integration.

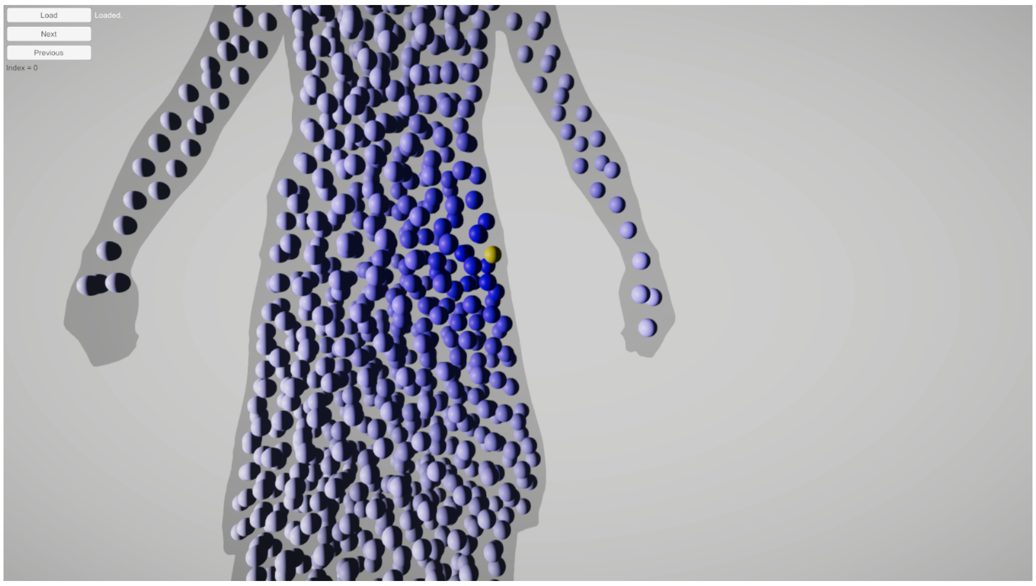


Figure 5.2: Centers could be translated only in the camera view plane in the legacy application. The influence of the editing action was shown in blue, visibly falling off with distance, while the selected vertex was shown in yellow.

operate. At that point, slicing of the model would have to be introduced to allow the user to view and manipulate the center. The animator would have to use a slicing control along with the pointer interchangeably.

These inconveniences lead to the original application concept being abandoned in favor of a VR implementation, to which the original application has been converted. One downside of such an implementation is that VR applications are often deployed on the headset without access to the processing power and storage space of a desktop computer. Specifically in this case, the Oculus Rift headset was used during development, to which applications are typically deployed as Android `.apk` packages.

However, the Unity engine allows for VR applications to be developed and previewed within the editor, while using the host computer to process the application logic and store data. Desktop targeted builds of the applications also use the host machine for processing and storage. This requires the headset to remain connected to the computer, limiting the mobility of the user and the tracking options. When connected to the computer, seated editing is most likely, rather than approaching and manipulating the model as one would a physical object in reality.

In the future, a client-server architecture could be implemented, using the VR client to display the model and submit editing actions and UI interactions to a server, which would then use the resources of the host machine to calculate the deformation and return the final center and vertex positions back to the user. Since TVM sequences have high storage requirements as well, storing the full sequence on the headset would likely be impossible. This would require some form of loading system for the data, which would keep only a part of the sequence available to the user and load in the remaining parts on-demand. Implementing this system was attempted, but was scrapped due to the complexity of the communication between the client and server components, which would not be possible to complete within the scope of this work.

In the following text, the implemented VR application is presented. First, the structure of the application is described along with the used resources. Later, the chapter describes how editing actions are processed in practice, which components contain the processing logic, and how they communicate.

5.1 Scene structure

The Unity engine organizes applications into ***scenes***. Scenes contain ***game objects***, the basic building block of Unity applications. Game objects are then further organized hierarchically within a scene. A scene defines the root of a coordinate system, in which the game objects are contained, each having their transformation - their own position, location and scale. This transformation defines a local coordinate system, in which child game objects can be contained. Scripts attached to game objects are called ***components***.

The editing system is composed of only one scene. This scene contains several top level game objects. Their functions will be described in detail along with the functions of some of their child objects and components. The notable top level objects are the following:

- *XR Origin*
- *Environment*
- *Level*
- *Controller*
- *Brush*

The *XR Origin* object is one of the most important objects in the scene, as it represents the VR headset along with the controllers. The application uses the Unity *XR Interaction Toolkit*, which is a library for integrating virtual reality into Unity applications. Many options for integrating VR into Unity are available, however the *XR Interaction Toolkit* is relatively mature and a wide range of learning materials which utilize it is available. It also offers cross-platform compatibility, enabling support for the application to be extended to other headsets in the future. While implementing the VR interaction system, I loosely followed tutorials by Valem Tutorials [Val22]. The tutorials also provide an animated variant of the Oculus hand models, which I have used in the application.

Most of the interaction logic is contained in the *XR Interaction Toolkit* scripts attached to the *XR Origin* object and its children. The *Left Hand* and *Right Hand* child objects contain the *XR Controller* component, representing the VR controllers and their Input System action bindings. Further among their child objects, *XR Poke Interactor* components enable poke-type interactions with the user interface.

The *Left Hand* and *Right Hand* objects each have a *Canvas* component attached to one of their child objects. This object represents a menu, which the user can access by turning the inside of their wrist towards the headset. The right hand menu contains general settings and playback controls, while the left hand menu allows for setting editing parameters. This interaction between the direction of the user's sight and the menus is enabled by *XR Gaze Interactor* and *XR Gaze Interactable* components.

Lastly, a teleportation function is implemented to enable easy navigation. Teleportation is made possible by the *XR Ray Interactor* component. *Teleportation Area* and *Teleportation Anchor* objects then mark objects in the scene as areas which can be teleported to.

The *Environment* game object is mainly cosmetic, replacing the default skybox with an outer space visual. Since the application may serve as promotional material at the UWB, it is desirable for the application to be visually attractive. Therefore, it was designed to resemble a space ship or a space station hangar with a view of space and futuristic neon visuals. The sky box was designed following a tutorial by Paradyme Games [Par20] and uses textures provided by the creator. The VR environment can be seen in fig. 5.3 and 5.4. An example of editing is shown in fig. 5.5.

The *Level* game object contains all of the components of the level, which have been built using the *3D Free Modular Kit* [Bar23] available at the Unity Asset Store. It also contains the interaction UI for selecting input sequences. The *Sequence* object is a child of the *Level* object as well. The *Sequence* object is discussed further in the next section, as it contains some of the sequence loading and editing logic.

Controller is a simple game object with one script attached. This script loads the settings file from the current directory, which should contain the path to the directory containing the input data. In the archive enclosed with this work, the build of the application points to the included input sequences.

Brush is the game object that contains the majority of the logic components. It will therefore be described thoroughly in the next section, which shows how all of the above listed components contribute to the application.

Along with the already mentioned resources, I have also used Google Icons and Google Fonts to create the application UI. The application uses the *Major Mono Display* font.

5.2 Method implementation

The sequence is loaded via the *Sequence UI* component, which is represented by a world-space canvas near the location at which the user enters the application. For each available sequence, the canvas contains a button which loads this sequence. The button calls to the *Sequence* component attached to the game object of the same name, calling its *Load()* method.

The method loads frames of the sequence by reading the centers files, mesh files and a settings file containing the playback framerate in frames per second (FPS). The method is ran asynchronously, as loading the sequence is resource-intensive and would otherwise cause the main thread to wait for the sequence files to be read. From the user's point of view, the application would become unresponsive and stuck on the last rendered frame. In virtual reality, this can be an uncomfortable experience and should therefore be avoided whenever possible.

Once the files have been loaded, the method initializes the *Center Pool* to the locations of the centers in the first frame of the sequence. The *Center Pool* is a component

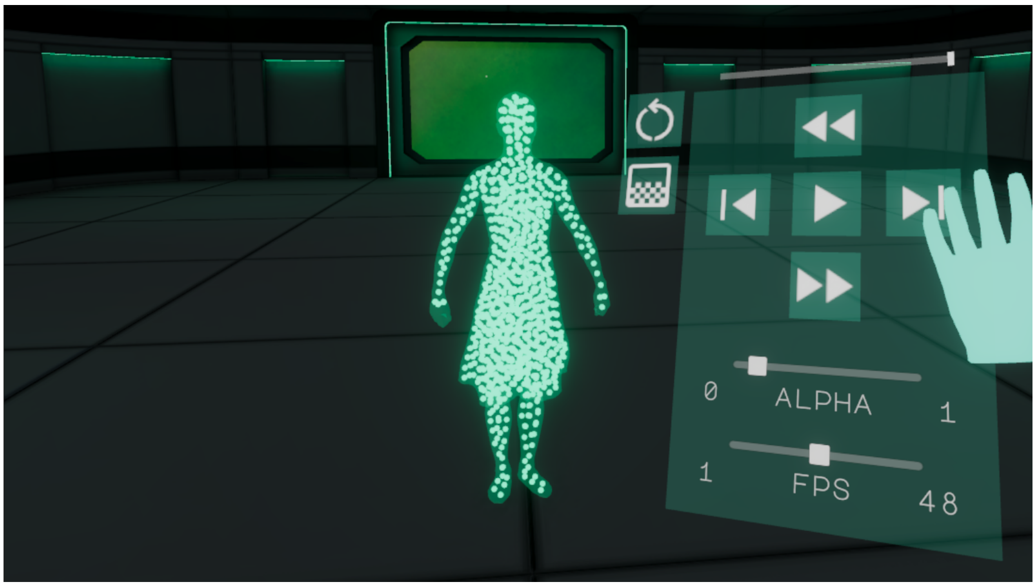


Figure 5.3: A screen capture of a frame of the TVM sequence centers displayed in VR. The mesh surface can be made transparent, making the centers easy to view and interact with. The playback menu is also shown in the image.

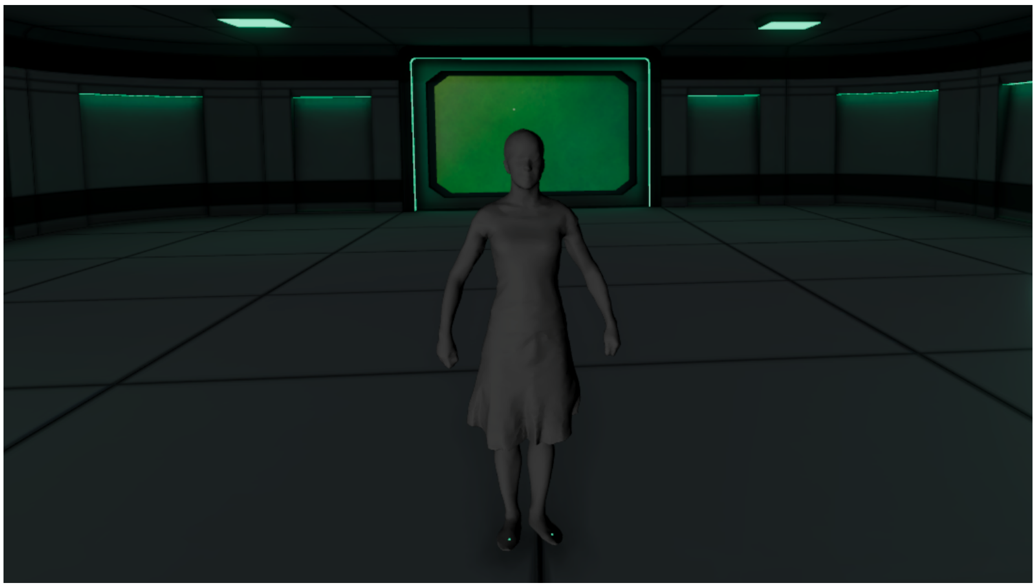


Figure 5.4: A screen capture of a frame of the TVM sequence displayed in VR. The surface material is set to an opaque lit material, making the surface easier to inspect, but hiding the centers. Some centers may be visible due to their diameter not fitting into the volume they represent. They may also be displaced from the volume by editing.

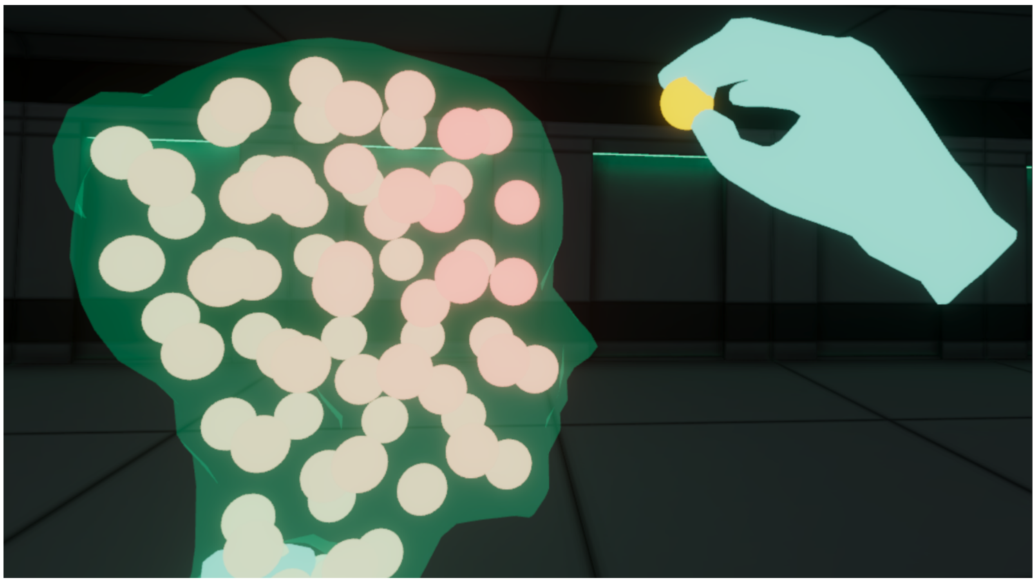


Figure 5.5: A screen capture of the editing process in the VR environment. The influence of editing is shown in red with a visible falloff, while the edited center is highlighted in yellow.

which manages the *Center UI* objects in the scene. It initializes a pool of centers of the size required by the sequence. Only centers for one frame are initialized, since no more than one frame will be shown at one time, and center counts are constant in all frames. This means that all frames of the sequence can be displayed simply by moving the center object positions. Similarly, each frame of the mesh is represented by one object, for which vertex coordinates and connectivity are updated as the sequence is played back.

The actual center position values are kept separate from the coordinates of their UI representations. This allows the centers to be edited in the UI without losing their true locations. In fact, after each editing action is committed and calculated, UI positions are fully reset using the updated model data. This is a principle from the ***Model-View-Controller*** (MVC) software design paradigm, in which the UI components are kept independent from the model representation. The user interacts with a controller, in this case the *Sequence* component, in order to commit an editing action. The controller propagates this action to the model, which is then modified. The modifications of the model are then propagated to the view components.

The *CommitEdit()* method is also executed asynchronously, since it can be very time consuming, especially for longer or larger sequences. In fact, in the current implementation, the method is nowhere near capable of running in real time, which is discussed later in section 6.2. Most of the time is spent on surface deformation.

While the method is running, the handheld controller game objects are disabled and a prompt indicating that the application is waiting to finish processing a task is shown. Hiding the controllers prevents the user from interacting with the environment in an unpredictable manner during the editing, however, their input actions still activate if buttons are pressed. This means that the user is free to continue moving throughout the room by using the teleport function or the joystick walk, but will not trigger events on the UIs which are highly sequence-dependent.

Editing activates the *Commit()* method on the *Brush* component. A *Brush* is simply a collection of components which define each of the remaining three steps of the pipeline:

- *Center Deformation*,
- *Sequence Deformation*,
- and *Surface Deformation*.

These classes are abstract. They are implemented by the classes *Gaussian Center Deformation*, *Kabsch Sequence Deformation*, and *Neighborhood Surface Deformation*. Each of these components contains a method which executes their step in the pipeline. The components' methods are called from the *Brush* class, which therefore defines the flow of data between them and the order of processing.

The *Brush* class never explicitly calls the center deformation component. Instead, it calls the sequence deformation component and hands over the center deformation component as a parameter of the call. The sequence deformation component's task is to propagate center deformation from the initial frame throughout the entire sequence, therefore it needs to have access to the center deformation algorithm. Instead of processing only the remaining frames of the sequence, it is reasonable to include the deformation of the initially edited frame in its scope.

Once the centers have been deformed in each frame by the sequence deformation algorithm, control is handed over back to the *Brush* class, which has retained original center positions and received the deformed positions. The original and deformed positions are passed over to the surface deformation component and used to calculate the translation vectors by which each vertex of the surface should be moved. The deformed vertex positions are passed back to the *Brush* class, which updates the model. At this point, the *Commit()* method is completed and the frame is redrawn in the UI. Simplified code of the algorithm can be found in the listing 5.1.

Source code 5.1: Committing edits

```
void Commit(int centerIndex, int frameIndex, Vector3
    transformedCenter, Frame[] frames)
{
     var center = frames[frameIndex].centers[centerIndex];
    var translation = transformedCenter - center;

    // Deforming the centers in each frame of the sequence
    var deformedCenters = sequenceDeformation.DeformSequence(
  centerIndex, frameIndex, translation, frames,
  centerDeformation);

    for (int i = 0; i < frames.Length; i++)
    {
        // Deforming the surface in frame i using the both
        // the unedited and the deformed center positions
        var deformedSurface = surfaceDeformation.
  DeformSurface(frames[i].centers, deformedCenters[i],
  frames[i].vertices);

        // Updating the model
        frames[i].centers = deformedCenters[i];
        frames[i].vertices = deformedSurface;
    }
}
```

The *Gaussian Center Deformation* component is trivial, in that it simply takes each center and calculates the weight by which the translation of the effector should be scaled when applied to it. A more interesting algorithm can be found in the *Kabsch Sequence Deformation* class.

The algorithm first finds the nearest neighbors of the effector in the current frame. This is the only time that the nearest neighbor query is executed by the sequence deformation algorithm. The *Carry()* function is then used along with a direction parameter which should be set to either 1 or -1. The parameter defines the direction from the current frame in which the effector translation should be distributed. The method determines the source and the target frame and finds the optimal rotation by which to rotate the translation vector. It uses the Kabsch algorithm described in section 5.2.1. It then carries this vector into the next frame in the same direction using the same neighboring vertices. Lastly, it calculates the center deformations for the current frame using the transformed vector. This process is repeated until the beginning or the end of the sequence is reached.

The assumption that the same neighborhood can be used throughout the entire sequence may lead to some inaccuracy in cases where the initial neighborhood did not consist of centers which travel together for the entire duration of the sequence. However, the method provides visually good results and in case of editing artifacts, changes to the method would be trivial. Simplified code of the sequence deformation and the carry method can be found in listings 5.2 and 5.3 respectively.

The *Neighborhood Surface Deformation* component is once again trivial in its implementation. Its notable feature is the use of a kd-tree, which is initialized once per frame and used to search for the nearest neighbors of each vertex in the frame. The kd-tree data structure is described in section 5.2.2.

Source code 5.2: Sequence deformation

```
Vector3[][] DeformSequence(int centerIndex, int frameIndex,
    Vector3 translation, Vector3[] centersBefore, Vector3[][]
    allCenters, CenterDeformation centerDeformation)
     {
         int frameCount = allCenters.Length;
         Vector3[][] deformedFrames = new Vector3[frameCount
    ][];

         // Finding the nearest neighbors to the edited vertex
         var nearestCenters = GetNearestNeighbors(allCenters[
    frameIndex][centerIndex], centersBefore);

         // Carrying the edit towards the beginning of the
    sequence
         Carry(centerIndex, frameIndex, -1, frameCount,
    nearestCenters, translation, allCenters, deformedFrames,
    centerDeformation);

         // Carrying the edit towards the end of the sequence
         Carry(centerIndex, frameIndex, +1, frameCount,
    nearestCenters, translation, allCenters, deformedFrames,
    centerDeformation);

         // Deforming the centers in the current frame
         deformedFrames[frameIndex] = centerDeformation.
    DeformCenters(centerIndex, translation, allCenters[
    frameIndex]);

         return deformedFrames;
     }
```

Source code 5.3: The Carry method

```
void Carry(int centerIndex, int frameIndex, int
    frameDirection, int frameCount, int[] nearestCenters,
    Vector3 translation, Vector3[][] allCenters, Vector3[][]
    deformedFrames, CenterDeformation centerDeformation)
     {
         int nextFrameIndex = frameIndex + frameDirection;
         bool isBeforeSequence = nextFrameIndex < 0;
         bool isAfterSequence = nextFrameIndex >= frameCount;

         // Recursion stop condition
```

```
        if (isBeforeSequence || isAfterSequence)
            return;

        // Finding the rotation matrix R using the Kabsch 
    algorithm
        var P = Kabsch.MatrixFrom(centerIndex, nearestCenters
    , allCenters[frameIndex]);
        var Q = Kabsch.MatrixFrom(centerIndex, nearestCenters
    , allCenters[nextFrameIndex]);

        var avgP = Kabsch.Avg(P);
        var avgQ = Kabsch.Avg(Q);

        Kabsch.Subtract(P, avgP);
        Kabsch.Subtract(Q, avgQ);

        var R = Kabsch.GetRotation(P, Q);
        var RD = R * translation;

        // Carrying the edit towards the next frame
        var rotatedTranslation = new Vector3(RD[0], RD[1], RD
    [2]);
        Carry(centerIndex, nextFrameIndex, frameDirection,
    frameCount, nearestCenters, rotatedTranslation, allCenters
    , deformedFrames, centerDeformation);

        // Deforming the centers in the current frame
        deformedFrames[nextFrameIndex] = centerDeformation.
    DeformCenters(centerIndex, rotatedTranslation, allCenters[
    nextFrameIndex]);
    }
```

5.2.1 The Kabsch algorithm

The Kabsch algorithm was first introduced in *A solution for the best rotation to relate two sets of vectors* [Kab76; Wik23a] in 1976 by Wolfgang Kabsch. The method is used to calculate an optimal rotation matrix which aligns the points in two sets by minimizing the root mean squared deviation (RMSD) [Wik23c] between point pairs. RMSD is calculated as per eq. 5.1, where $\hat{x}_i$ is the expected position of a point, x_i is the actual position of the point and N is the size of the point set. In the point alignment problem, x_i and $\hat{x}_i$ correspond to the coordinates of the points in the source and the target frame.

$$RMSD = \sqrt{\frac{\sum_{i=1}^{N}(\hat{x}_i - x_i)^2}{N}} \tag{5.1}$$

This method is used very frequently in computer graphics, since aligning two sets of points is a common problem. For example, the problem also appears in point cloud registration, which is one part of the 3D scanning process. Since depth images only capture the scanned object from one point of view, multiple images have to be merged to obtain a full view of the scanned surface. As depth images produce point clouds, the problem of merging two images translates directly to finding the optimal transformation which would align the two point sets.

Along with finding the optimal rotation, optimal translation is sometimes required. The problem of finding both the optimal rotation and translation is called the partial Procrustes superimposition [Wik23b]. It differs from full Procrustes superimposition by omitting the scaling of the point set.

Given a point set $\mathbb{A}$ and a point set $\mathbb{B}$ which should be aligned by optimal rotation, the Kabsch algorithm begins by building matrix $\mathbf{P}$ as an $(N \times 3)$ matrix created by placing the coordinate vectors of the points from set $\mathbb{A}$ as rows of the matrix. Analogically, the matrix $\mathbf{Q}$ is created using the points from set $\mathbb{B}$.

An average of each point set is then found. This is the position of the centroid of each of the point sets. The algorithm works by translating both point sets so that their centroids lie in the origin of the coordinate system. Therefore, the centroid position of the point set $\mathbb{A}$ must be subtracted from each row of the matrix P, and the centroid of set $\mathbb{B}$ must be subtracted from the rows of matrix $\mathbf{Q}$.

If the point sets contained identical points that had merely been rotated and translated, it is easy to see that at this step, the point sets could be precisely aligned simply by rotating one of them until alignment. However, the point sets are typically distorted and do not align perfectly.

In the next step, the Kabsch algorithm calculates the covariance matrix $\mathbf{H}$ as per eq. 5.2. The desired rotation matrix $\mathbf{R}$ can then be computed using Singular Value Decomposition (SVD). The SVD of matrix $\mathbf{H}$ can be seen in eq. 5.3. Finally, based on the sign of the determinant of $\mathbf{VU}^\mathsf{T}$ from eq. 5.4, the matrix $\mathbf{R}$ can be computed using eq. 5.5. The implementation of the Kabsch algorithm in this work uses the *Math.NET Numerics* [Mat23] library for linear algebra as well as the SVD algorithm implementation.

$$\mathbf{H} = \mathbf{P}^\mathsf{T}\mathbf{Q} \tag{5.2}$$

$$\mathbf{H} = \mathbf{U\Sigma V}^\mathsf{T} \tag{5.3}$$

$$d = \operatorname{sgn}\det\left(\mathbf{VU}^\mathsf{T}\right) \tag{5.4}$$

$$\mathbf{R} = \mathbf{V}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & d \end{bmatrix}\mathbf{U}^\mathsf{T} \tag{5.5}$$

5.2.2 Kd-tree

A ***k-dimensional tree*** [Ben75], also written as kd-tree or k-d tree, is a data structure used in space partitioning point sets, particularly in applications which require quick nearest neighbor or range searches. Range searches are a type of search that looks for all the points which lie within a specified distance from a query point.

A kd-tree can be constructed by splitting the point set along each axis recursively. In each step, the point with the median coordinate along the currently processed dimension is selected and used to divide the point set in two parts. Each of these parts is then processed along the next dimension. Given a 3-dimensional case, this means that the point set would first be split by a plane along the x axis, forming two point sets which would each be split along the y axis, forming four different point sets, which would then be split along the z axis. The algorithm continues to subdivide the point set, cycling back to the x axis and repeating the steps until all nodes contain sufficiently small point sets.

This has the benefit of defining very small neighborhoods. Initially, nearest neighbors can be searched for locally, within one node. If the distance to the nearest neighbor is smaller than the distance to the nearest dimension divider, the point must be the true nearest neighbor, since no closer point can lie beyond the divider. If the nearest neighbor within the node does not fulfill the condition, the neighboring node must be searched as well. The traversal of the tree continues up the tree hierarchy until the condition of the nearest neighbor being closer than the nearest divider from an unsearched area is fulfilled.

Kd-trees provide the most benefit when they can be used repeatedly for many searches over the same point set. The complexity of kd-trees construction is shown to be $O(n \log(n))$ in *On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)* [WH06]. This is the case of the proposed surface deformation method, which searches for nearest neighboring centers of each vertex in the mesh.

The input point set is the set of center positions before the editing action. Typically, the set of centers will be much smaller than the set of all vertices in each frame. Since the n in the complexity function refers to the size of the input point set, the cost of building and searching the tree is decoupled from the size of the input mesh data. At the same time, the tree has to be searched once for each vertex in each frame. The complexity of kd-tree nearest neighbor searches is $O(\log(n))$ on average and $O(n)$ in the worst case. For finer tessellations, this means that kd-tree generation and searches can remain fast as long as the center count stays the same.

Regardless of this, surface deformation remains the slowest part of the algorithm. This is expected, since vertices in each frame can number in the tens of thousands, while the number of centers, which are processed in the previous steps, is an order of magnitude smaller.

The implementation of a kd-tree used in this work was provided along with the volume tracking algorithm by the UWB. The method used to retrieve nearest neighbors from the tree searches the tree up to a given distance from the input point, returning all neighbors in the radius. This can be an insufficient number of neighbors, necessitating a second search with an increased range parameter. The parameter is doubled each time the search fails. A kd-tree which could continue the search until a sufficient number of neighbors was found could improve the efficiency of the implemented surface deformation method. Additionally, the range parameter can work well if the required number of neighbors can usually be found at first or second search. However, the optimal value of the parameter depends on the volume sampling as well as the scale of the model, complicating its selection.

Testing and analysis 6

This chapter introduces the data used in the experiments and the data formats used by the application. The data is then used in editing experiments, which are then evaluated. Based on experiment results, modifications and future directions for the work are suggested.

6.1 Input data

The sequences used in this work have been published by the authors of *Articulated Mesh Animation from Multi-View Silhouettes* [Vla+08a] and are available online [Vla+08b]. The dataset consists of dynamic mesh sequences, however, the editing system does not make use of the temporal correspondence between sequence frames and effectively treats them as TVM sequences.

6.1.1 Input data format

The input directory is specified in the *settings.xml* file. This directory is searched when the application is started and all of its subdirectories are assumed to contain input sequences of the same name. The names are listed in the UI. Upon selecting one of the sequences, the application attempts to load the sequence data. In the data folder of a sequence, the subfolders *centers* and *meshes* must be present. Optionally, a *settings.xml* file can also be present, which is used to set the default sequence FPS. Examples of *settings.xml* files are included in the attached data archive. The location of the files is listed in appendix C.

The *meshes* directory is expected to contain files in the `.obj` format. An example of an `.obj` file is shown in listing 6.1.

File 6.1: Mesh file format.

```
v 0.49353 0.0415381 0.0984336
v 0.514614 0.0468725 0.0316093
v 0.49535 0.0479066 0.0146358
```

```
...
f 1545 1538 1564
f 1539 1523 1569
f 1569 1523 1553
...
```

The *centers* directory is expected to contain files with either the `.xyz` or `.bin` extension. The specific extension is important, since it determines which file loading algorithm will be used. An example of an `.xyz` file is shown in listing 6.2. The structure of the `.bin` file is similar. It begins with a single integer indicating the number of centers stored in the file. The remaining data consists of float numbers, where every three consecutive floats represent the x, y and z coordinates of a center, respectively.

File 6.2: Centers .xyz file format.

```
0.34939298 0.89302266 -0.14726524
0.36640382 0.5401483 -0.022185493
0.37571722 0.6321302 0.074741736
...
```

6.2 Experiments

6.2.1 Sequence: samba

6.2.1.1 Enlarging the stomach

The first experiment focused on enlarging the stomach area of a model. The desired shape was easily modeled by selecting a suitable falloff parameter for the Gaussian function. The surrounding centers of the effector moved in the expected way, creating a bulging stomach. Several editing actions were required to adjust the shape to look more natural. The editing actions carried over well to other frames and extended well to the surface as well. A comparison image of the surface before and after editing can be seen in fig. 6.1.

One downside of the process was the long processing time for surface deformation. The average time in seconds spent on each stage of processing per editing action can be found in table 6.1. As the table shows, the algorithm spends the vast majority of the processing time in the surface deformation stage. In an effort to decrease the time spent in this stage, using only two nearest neighbors in the surface deformation stage was attempted and the editing process was repeated. The results can be seen in fig. 6.2. When using two nearest neighbors, effectively only one neighbor is used, since the most distant neighbor is always assigned a weight of zero. As can be seen from the figure, the surface becomes distorted when using only one neighbor to influence the deformation of the geometry. The change also did not lead to a significantly lower processing time. This can be related to the search radius of the kd-tree, which may on average be returning a greater number of centers than necessary, leading to some parts of the algorithm processing the same amount of centers, despite the fact that only two centers would be used.

6.2.1.2 Bending the arm

In the next experiment, the arm of a model was bent upwards. A larger falloff parameter had to be chosen, since the influence of the editing action would have spread to the torso of the model otherwise. The forearm was then edited by first moving a center near the middle and then near the tip of the hand. The deformation did not have the expected results, excessively deforming the forearm in a way that was not recoverable by further editing. The result of deformation in the edited frame can be seen in fig. 6.3. The deformation also did not carry through the sequence well, the forearm deforming excessively in the final frames of the sequence. This can be related to the neighborhood of the edited center, since the hand of the model touches the torso in the last frames, which causes centers to be redistributed in the volume near the contact point. A comparison of the last frame of the sequence before and after editing is shown in fig. 6.4.

6.2.2 Sequence: squat2

6.2.2.1 Mohawk

In the *squat2* sequence, the model's hairstyle is quite well defined and should be possible to deform to simulate longer hair. The sequence was tracked using 4000 centers, leading to most of the volume being densely packed with centers. However, in the head and hair area, fewer centers are available, likely due to the smaller volume of these areas. The centers inside the hair volume were rather few and far apart. By using a large falloff parameter, it was possible to move them almost separately from the remaining centers. A comparison of the surfaces before and after editing is shown in fig. 6.5

The radial symmetry of the falloff pattern is undesirable in this case, as the hairstyle extends in one direction and choosing effectors in a way that would combine translations favorably is rather unintuitive. For an animator, it would be much easier to have a tool with a different falloff pattern available.

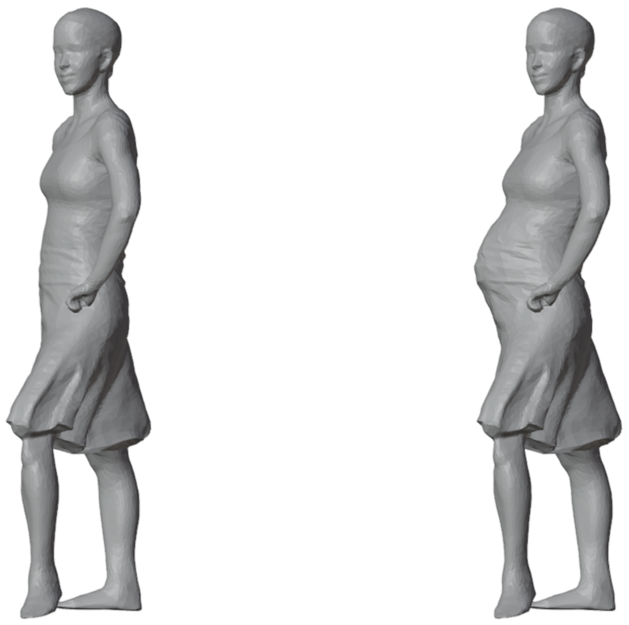


Figure 6.1: A side-by-side comparison of a frame of the sequence before and after editing.

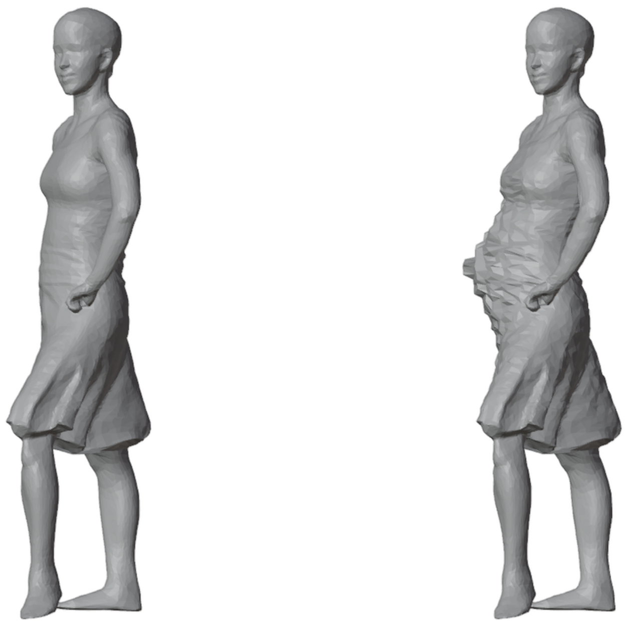


Figure 6.2: The surface is distorted when only one neighbor influences the surface deformation.

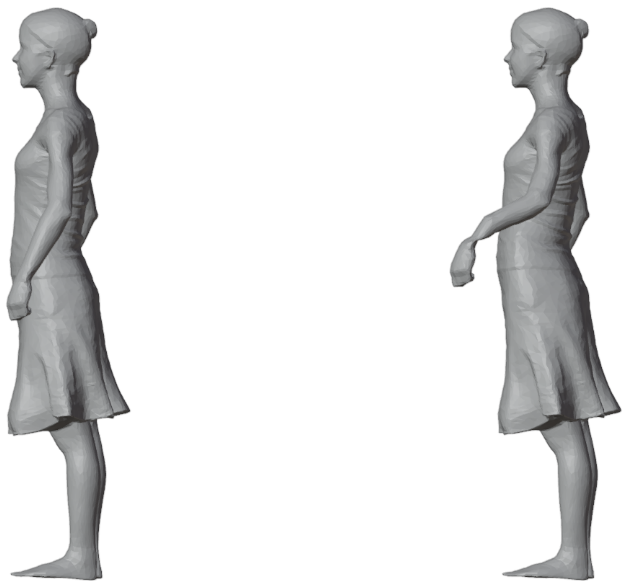


Figure 6.3: Bending the arm did not result in the desired outcome.

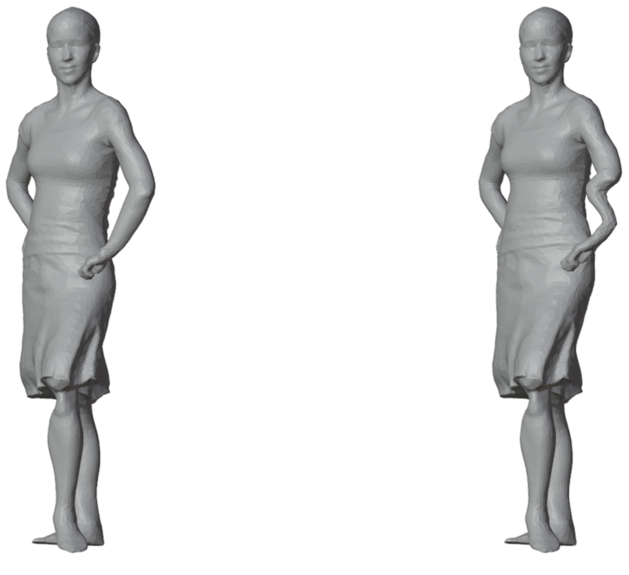


Figure 6.4: The edit did not carry through the sequence well.

6.2.2.2 Nose job

Interesting applications of the method could be found in facial deformation, since the currently implemented brush shape is well suited to making rounded, bulge-shaped edits to the surface. Since much of the human face is rounded, applications could include deforming the cheeks, cheekbones, nose, lips or eyebrow shape, as well as the overall head shape. Such deformations could even change the appearance of a person sufficiently to appear as a different sequence model. In applications where crowds of people with some variety of appearance is required, this could be beneficial.

Unfortunately, the face of the model does not contain as many centers as other areas of the body, making fine and smooth edits affecting only very small volumes difficult. As a proof of concept, the nose shape of the model was modified. This modification can be seen in fig. 6.6. The modification carried well throughout the sequence, deforming the model's face believably and maintaining the surface details.

6.2.2.3 Slimming

The model in the *squat2* sequence is wearing loose clothes which can be quite prominent in the stomach and leg areas. A slimming deformation was therefore attempted with the goal of making the clothes appear tighter. The deformation carried well throughout the sequence, although since a rather small falloff was used, large areas were affected all at once, bending the model's knees slightly. By using a larger falloff and editing more centers along the legs, a tighter fit of the pants could have been achieved without the effect spreading to the rest of the sequence. The results of the experiment can be seen in fig. 6.7 and 6.8.

6.2.3 Sequence: handstand

6.2.3.1 Wide stance

In this experiment, widening the final stance of the model in the handstand position was attempted. The results are shown in fig. 6.9 and 6.10. While unlike the *samba* sequence deformation, it was possible to spread the arms wider, this resulted in widening the model since a smaller falloff parameter was used. The model could have been edited in the final stance instead, allowing the deformation to propagate backwards to the starting pose. However, in the last frame, the shoulders of the model would have had to be moved apart as well, which could not be done without deforming the face of the model as well.

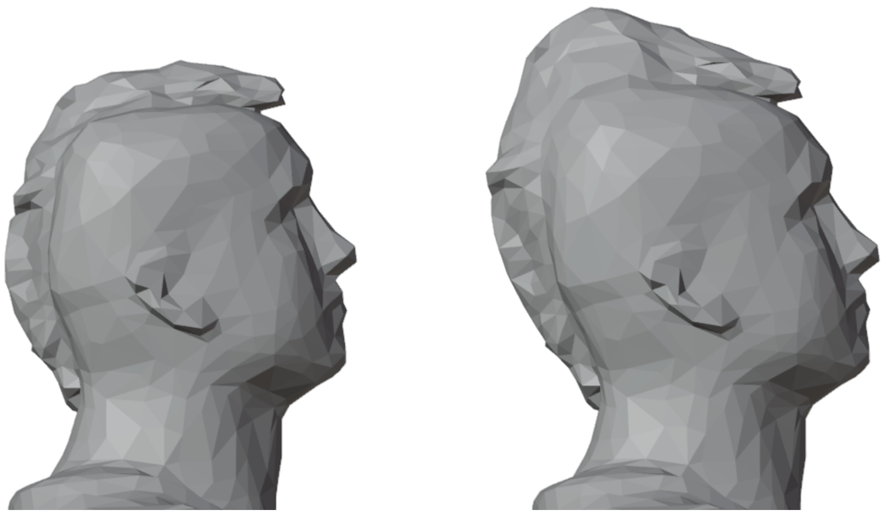


Figure 6.5: It was possible to lengthen the model's hairstyle.

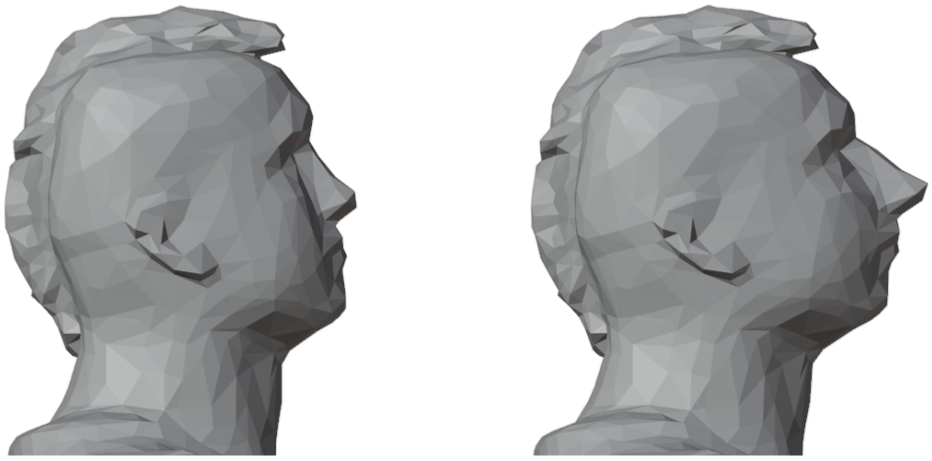


Figure 6.6: The model's facial features are changed while still resembling plausible human anatomy.

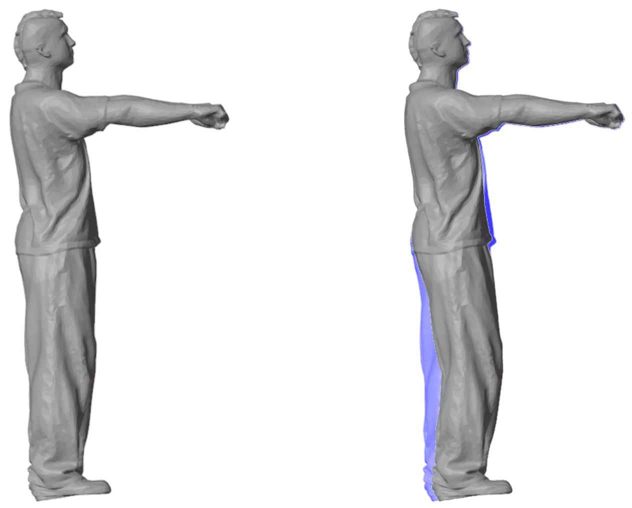


Figure 6.7: The edit in the standing position. The blue areas are the silhouette of the model prior to editing.

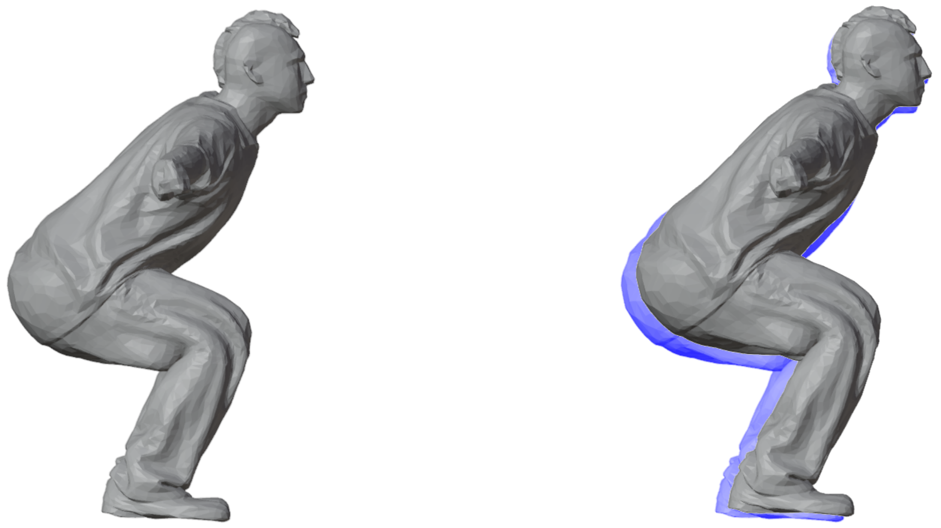


Figure 6.8: The edit in the squat position.

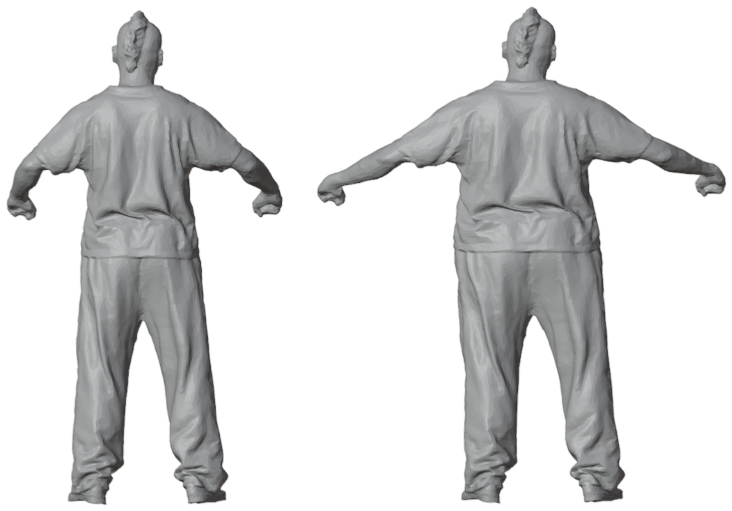


Figure 6.9: The arms of the model have been opened wider with the goal to achieve a wider stance in the handstand. Editing the arms also affected the torso, making the model wider.

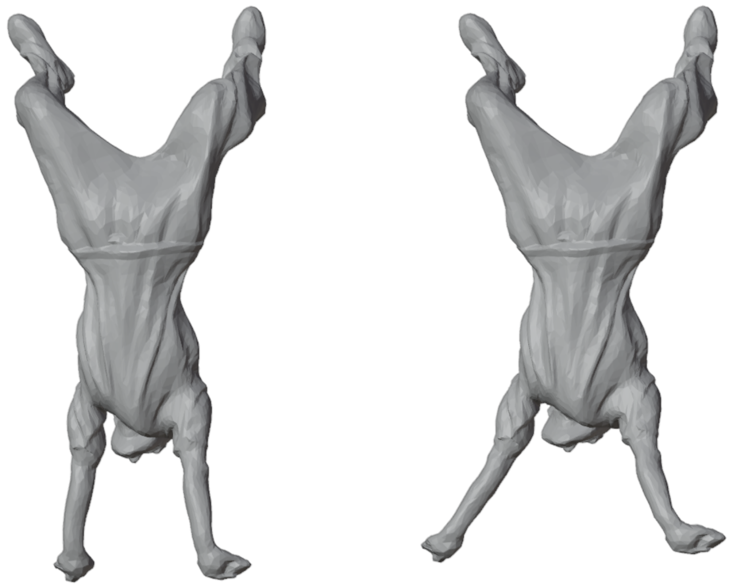


Figure 6.10: The deformation did carry through the sequence, but resulted in an awkward stance.

6.2.3.2 Feet apart

In the next experiment, a similar widening of the model's stance was attempted. However, it was very difficult to attempt and spread the feet wider apart due to the influence of the editing on the other leg. By editing only the tips of the feet, the spread of influence was limited, but resulted instead in an unnatural shape of the legs. The deformations also did not carry well through the sequence. The results are shown in fig. 6.11 and 6.12.

6.2.3.3 Hump back

In the final experiment, a similar deformation to the *samba* belly deformation was attempted. The back of the model in the *handstand* sequence moves very dynamically. A hump was added to the back successfully and carried through the sequence, however, the motion at playback is unconvincing, as the hump does not give an impression of being a part of the model's body and instead appears to move with the model's clothes. The results can be seen in fig. 6.13 and 6.14.

6.3 Analysis

The editing method in its current state has been tested and shown to be well suited for some types of intended deformation. The shape of the effector's area of effect makes the method well suited for adding rounded features to models and making small localized corrections, such as adjusting the shape of an area of a model's body or changing their facial features.

Attempts at deforming larger areas of sequences, such as attempts at changing arm or leg positions, did not lead to good results, which is expected, since the area of effect could not be properly captured. Low quality of deformation can also be caused by choosing an insufficient number of neighboring centers to be used in surface deformation.

One drawback of using the method in its current implementation is the execution time per editing action. Ideally, animators should receive instant feedback. However, the surface deformation part of the pipeline currently takes minutes to execute per each editing action. The average time per one action for each experiment is shown in table 6.1. The time spent on center deformation per method call was almost always below one millisecond. Compared to surface deformation, sequence deformation required an insignificant amount of time as well.

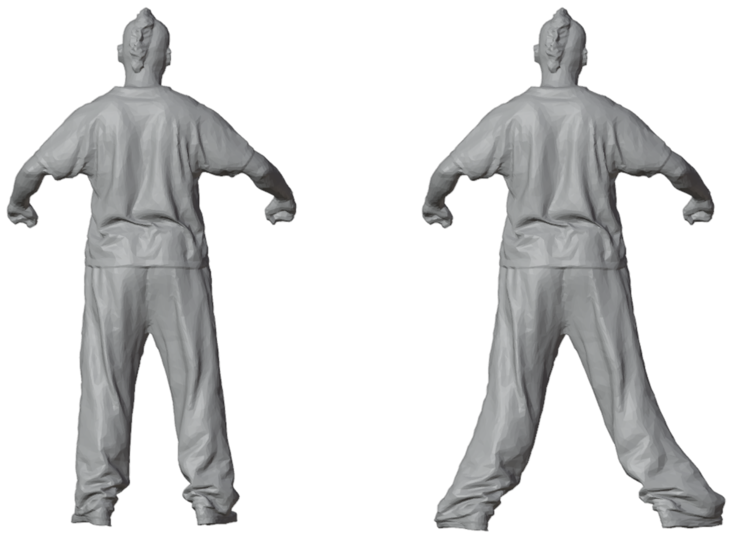


Figure 6.11: Widening the stance of the model resulted in unnatural leg bending.

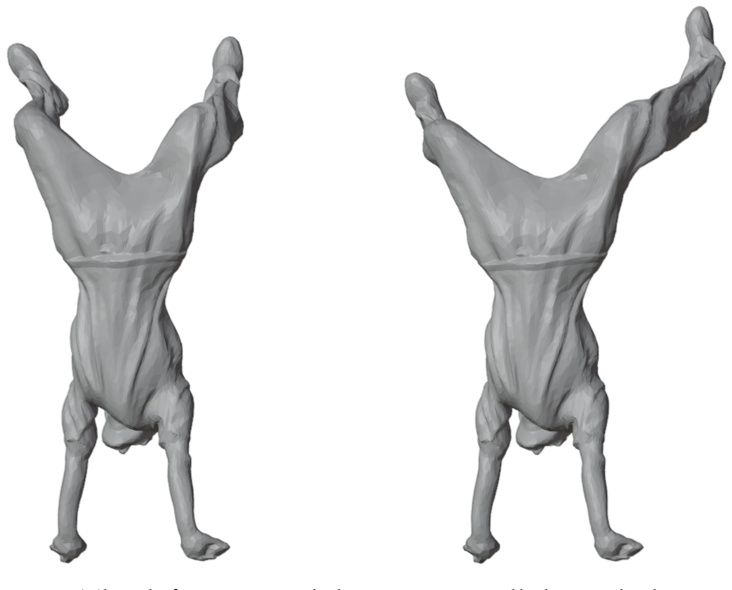


Figure 6.12: The deformation did not carry well through the sequence.

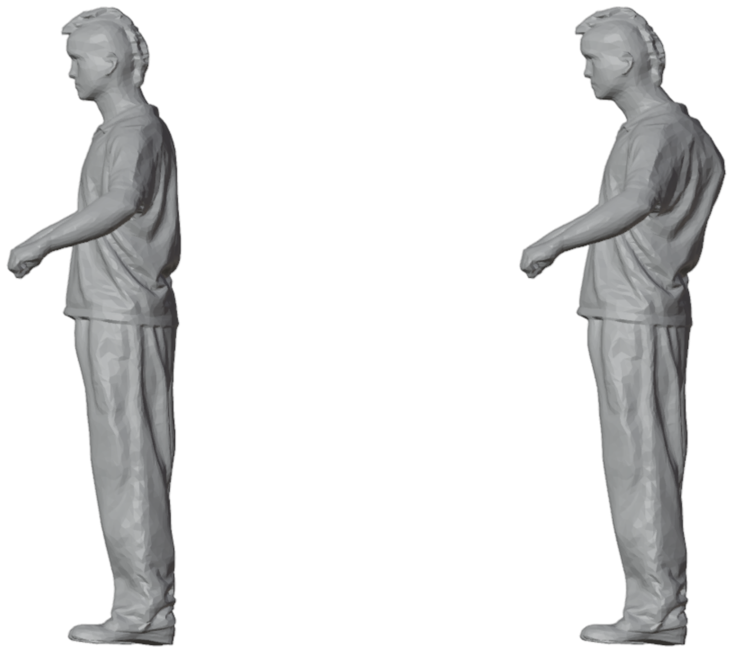


Figure 6.13: A hump was added to the model's back.

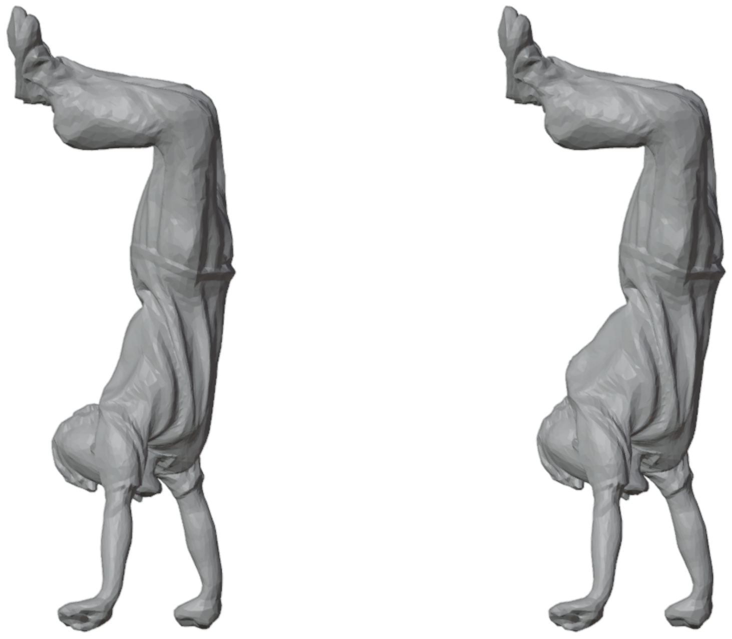


Figure 6.14: The hump was maintained throughout the sequence, but inherited the motion of the model's clothing, thus appearing unnatural in motion.

Table 6.1: Average editing times. Center and vertex counts are listed per frame.

Experiment	Frames	Centers	Vertices	Deformation time [s]		
				Centers	Sequence	Surface
belly	175	1000	9971	0.00	0.03	49.20
belly 2				0.00	0.04	48.78
arm				0.00	0.03	49.35
mohawk	250	4000	10002	0.00	0.13	197.85
nose job				0.07	0.21	199.47
slimming				0.00	0.15	208.39
wide stance	175	1000	10002	0.00	0.03	32.95
feet apart				0.00	0.03	30.90
hump back				0.00	0.03	33.43

6.4 Proposed improvements

In the experiments, issues were frequently caused by the editing area of effect extending to centers which should not be deformed, or not extending to the edited centers with a sufficient deformation intensity due to the effort not to deform other areas of the frame. This clearly indicates a need for a blocking function, which would constrain the positions of centers which should not be deformed. Similarly, the ability to assign deformation intensity ahead of committing the deformation would greatly improve the usability of the system. A system similar to weight painting in mesh skinning could be implemented and deformations could be committed on demand, rather than by center translation alone.

Deformations such as performed in the *mohawk* experiment could benefit from introducing new shapes of the editing falloff function. The shapes could be configurable similarly to brush shapes in digital painting software. The user could then be given a choice between committing edits immediately upon translating a center, in which case the deformation would be controlled by the brush shape directly, or using the weight painting system, which could be compatible with custom brush shapes as well, but would differ in the requirement to commit the editing action explicitly.

Other limitations of the system were encountered when larger scale deformations were attempted, such as bending the arm of a model. This specific use case would benefit from a rotation-based editing operation, which could use a point in space as a pivot along which centers would be rotated. Which centers should be affected by the rotation could once again be determined by weight painting or by blocking off the areas which should not be affected.

An interesting approach to modeling rotation could be inspired by mesh skinning. The user could define a bone-like effector at runtime by specifying two points in space to which nearby centers would be assigned. The user could then deform the sequence by manipulating this temporary bone. This could lead to a faster workflow than specifying a pivot and blocking off undesirable areas.

A significant quality-of-life improvement could also be achieved by optimizing the surface deformation part of the pipeline. Currently, it is not possible to edit sequences in real time. The user has to wait minutes for each editing action to complete the surface deformation.

One way to mitigate this issue (besides parallelization of the current solution) would be to enable the user to edit only the sequence of centers first and execute the surface deformation as a last step. This way, the user could make the required modifications uninterrupted, triggering surface deformation on demand at their own convenience. In this case, the user would lose the ability to verify that their actions are having the desired effect on the surface. However, during the final surface deformation, intermediate stages after each action could be saved, allowing the user to return to a point at which an action made undesirable changes without losing their previous work.

Alternatively, there is space for optimization in the nearest neighbor search algorithm. Using a kd-tree without a search range parameter, which could potentially cause the search to repeat, could speed up the calculation. The number of neighbors could possibly also be set automatically as a quality of life improvement. By scaling the models to a predefined size and finding the volume that each center represents, neighborhood size could be determined from the size of the volume which should affect each vertex. A suitable volume of effect could be determined experimentally.

A massive factor in surface deformation is also the number of affected vertices. At this time, all centers are affected by each editing action due to the Gaussian falloff parameter. If only some centers of the sequence were modified, it would naturally follow that only the vertices near those centers could be affected by the deformation. This could dramatically decrease the number of vertices to be deformed. As it is, many vertices undergo minimal deformation which does not contribute to the overall look of the animation. This approach is dependent on being able to determine which vertices should be affected by the edited centers. However, centers already define a volume to which they correspond - vertices near this volume should therefore be affected.

In fact, modeling deformation as displacement of the cells represented by each center could be beneficial. In the tracking, centers are evenly distributed throughout the volume. This property is quickly lost during editing, since centers can fully exit the volume or become clumped together. Any approach which could either maintain the volume of the model during editing and redistribute centers appropriately could possibly sustain a longer chain of deformations without the model becoming too distorted. For deformations for which maintaining the volume is too restrictive, explicit volume editing functions could be added.

Other possible improvements could arise by using the data generated during the tracking stage to identify which centers move together in the sequence and should therefore be deformed together. Such data could even be used to train a neural network which could then attempt to extend the motion of the model, thus prolonging the sequence. Although the applications of AI animation might be limited in this case, the current popularity of similar artificial intelligence applications could bring exposure to this method.

Conclusion 7

This work focused on implementing a system for editing time-varying mesh sequences. In its first part, the work introduced the theoretical prerequisites to 3D model representation and mesh editing, ensuring that the reader is familiar with concepts relevant to the implementation of the method as well as basic concepts in computer graphics.

The implemented method made use of a volume tracking system which had recently been developed at the UWB. The system was also described in detail. Using tracking data generated by the tracking system, a method using the Gaussian function for distributing deformations between deformed volume elements was developed. The volume element deformations were then further distributed to other frames of the time-varying mesh sequences. Finally, based on the deformation of the volume elements throughout the sequence, a method for deforming the mesh surfaces was developed.

The results of the implemented method were analyzed. Based on this analysis, future directions for the development of the system were proposed. Overall, the work represents a first attempt at implementing a new approach to time-varying mesh sequence editing with many possible directions for future improvement that could lead to new workflows in the area of computer animation being developed.

User documentation A

The app can be launched by double-clicking the *TVM VR Editor.exe* file located in the *Application_and_libraries/Build* directory or from the Unity engine project in *Application_and_libraries/Project*. A virtual reality headset must be connected to the computer, along with two VR controllers. The recommended headset is the Oculus Rift, as it was used throughout the development. If the user wishes to use their own data, they should change the path in the *settings.xml* file prior to starting the application. The path should not contain spaces.

Upon entering the virtual environment, the user will appear in a room, which they can navigate either using teleportation or using the VR controller thumbsticks. The left controller thumbstick controls movement, while the right thumbstick controls turning.

Teleportation is possible by lightly pressing the left trigger button, which will first highlight the target location, and upon being fully pressed, will teleport the user. The teleportation beam is shown in fig. A.1.

At start, the user should appear near the control station. The user can return

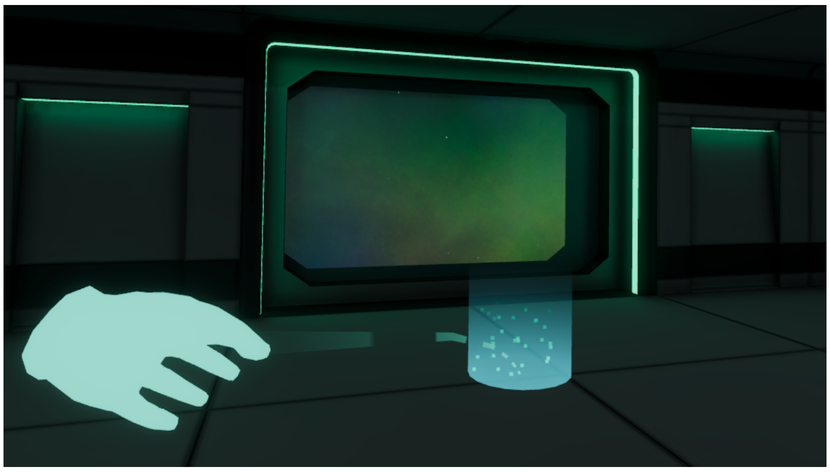


Figure A.1: By pressing the left trigger button lightly, the user can select an position to be teleported to.

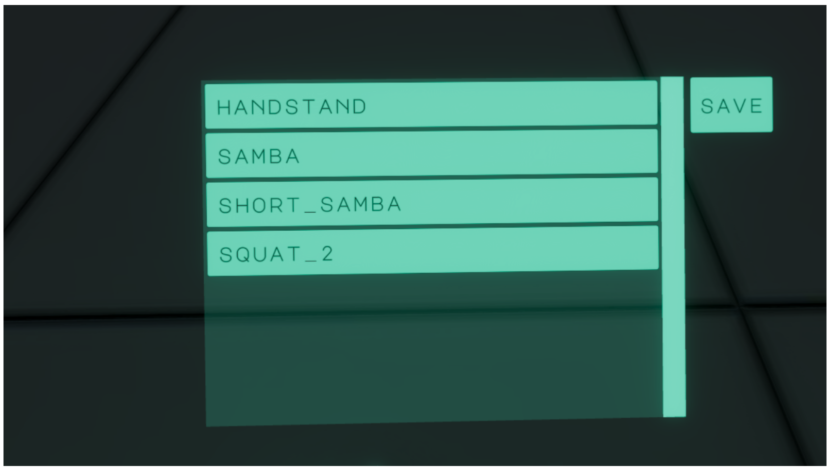


Figure A.2: The sequence menu displays the available sequences. The Save button saves the current state of the edited sequence.

to the control station by teleporting to a marked area on the floor. The control station is the user interface enabling the selection of a sequence to be displayed, as well as enabling sequences to be saved using the Save button. The UI is shown in fig. A.2. The UI can be controlled by either hand using the tip of the index finger shown in place of the controller. The list of available sequences is initialized at the start of the application and will not refresh if changes are made to the file system. Saved sequences are saved to the folder they have originally been loaded from. A new directory is created which uses the original sequence name with an appended timestamp.

Once the user has loaded a sequence, they can move towards it either by using teleportation or using the left thumbstick walk. Once they are in range of the sequence, they can use the right trigger button to grab the displayed centers an move them, thus editing the sequence. The centers to be edited light up, the intensity of their coloring changing with the strength of the editing action. If the model is difficult to access, the user can use the right grab button to drag the model into a better position. This does not affect the model data.

By facing the wrist side of the right controller, the playback menu is displayed. Out of the five playback buttons, the top button skips to the beginning of the sequence, the bottom button skips to the end, the left button moves to the previous frame, the right button moves to the next frame (both options looping if they reach a limit of the sequence) and the button in the center plays or pauses the sequence. The sequence cannot be edited while it is being played. On the left side of the menu, the upper button resets the position of the sequence, if it had been moved by using the right hand grab. The second button on the left of the menu switches the sequence mesh material. When the material is transparent, an alpha slider is available to the

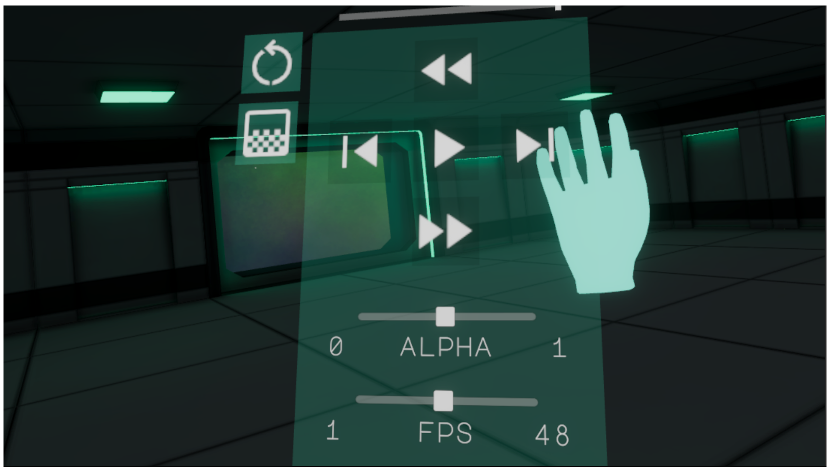


Figure A.3: The right hand menu controls playback and visualization.

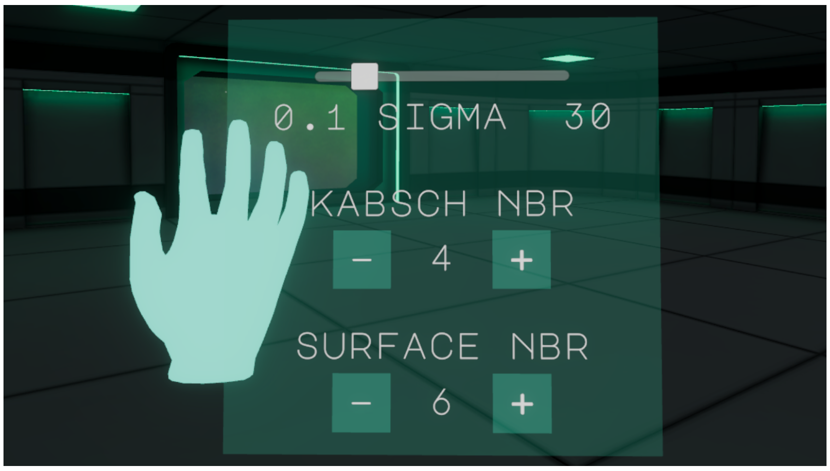


Figure A.4: The left hand menu controls the settings of the brush.

user, allowing them to to control the visibility of the surface. Lastly, an FPS slider is available, which slows down or speeds up the playback. The bar above the menu displays the playback progress. The right hand menu is shown in fig. A.3.

A similar menu attached to the left controller contains brush settings. The sigma parameter controls the falloff, where smaller numbers indicate slower falloff and therefore a larger editing area, Kabsch neighbors refer to the number of neighboring centers used to distribute center deformation between frames, and surface neighbors refer to the number of neighboring centers used to deform the surface.

B Programmer documentation

In the Unity engine, project resources as well as scripts are typically organized in the Assets folder. The structure of the Assets folder in this project is the following:

- *Plugins*
- *Resources*
- *Samples*
- *Scenes*
- *Scripts*
- *Settings*
- *TextMesh Pro*
- *XR*
- *XRI*

The *Plugins* folder contains the *MathNet.Numerics* library *.dll*. The library is used to implement the Kabsch algorithm.

The *Resources* folder contains data used to build the scene. It is further divided into the following subfolders:

- *3rd party*
- *Fonts*
- *Materials*
- *Textures*
- *Prefabs*

The *3rd party* folder contains third party resources, such as the modular environment asset, icons used in the user interface, textures used to create the galaxy particle system and the animated hand models. The *Fonts* folder contains the font used by the application as well as the generated SDF font asset. The *Materials* folder contains the materials used in the scene. The *Prefabs* folder contains the prefab objects used as templates when instantiating game objects for the scene, such as the model of sequence centers and a template for the sequence selection button. Lastly, the *Textures* contains the used textures.

The *Samples* folder is a folder created by importing the *XR Interaction Toolkit* samples library. The folder contains components provided by the library to enable a simple implementation of VR interactions.

The *Scenes* folder contains the single scene of the application. The *Settings* folder contains Unity pipeline presets. The *TextMesh Pro* folder contains a font rendering library imported by Unity. Folders *XR* and *XRI* contain XR and XR interaction settings.

The *Scripts* folder is the most interesting part of the project, as it contains the original source code of the application. It contains the following subfolders:

- *Creative*
- *Data Structures*
- *Interaction*
- *Logic*
- *UI*
- *Util*
- *Util/IO*
- *Util/Settings*

The folder *Creative* contains scripts *Sphere Wave* and *Sphere Wave Settings*. The *Sphere Wave* script contains the code which animates the centers shown prior to any sequence being loaded. The *Sphere Wave Settings* script then contains the presets for this animation.

The *Data Structures* folder contains the *Face* and *Frame* classes, which are simple containers used to organize mesh vertices into faces and sequence data into frames. It also contains the *KdTree* class, which was provided by the UWB.

The *Interaction* folder contains scripts *Activate Teleportation Ray*, *Animate Hand On Input* and *Grab To Move*. The first two scripts were created following tutorials by Valem Tutorials [Val22]. All of the scripts are used to react to input from the VR controllers.

The *Logic* folder contains the core of the application. The *Controller* script looks for input data on startup, while the remaining scripts form parts of the editing pipeline. *Center Deformation*, *Sequence Deformation* and *Surface Deformation* are abstract classes that define the input and output of each step of the editing pipeline. Classes *Gaussian Center Deformation*, *Kabsch Sequence Deformation* and *Neighborhood Surface Deformation* inherit from the abstract classes and provide implementations to the pipeline steps. The *Brush* class is responsible for calling the classes and executing the pipeline.

The *UI* folder contains mainly simple scripts for processing UI events. Notable classes include the *Center IO* class, which notifies registered listeners when a center is hovered or selected, the *Center Pool*, which maintains a pool of center game objects, and the *Sequence* class, which is responsible for loading and saving sequences, as well as triggering pipeline execution when a center is edited.

The *Util* folder contains classes *Mesh IO* and *Centers IO* used for loading the input files and class *Serialization* used to serialize classes to `.xml` file. It also contains settings container classes and mathematical helper classes including the *Kabsch* class, which provides methods implementing the Kabsch algorithm.

Readme.txt C

```
The work contains the following attachments:

Aplication_and_libraries
- Project/TVM VR Editor
  The Unity engine project folder, including the source code.
  A project settings file is located at
   Aplication_and_libraries/Build/settings.xml.
  Switch the input folder from Input_data to Results in the
   file.

- Build
  The built Unity application.

Input_data
- handstand
- samba
- short_samba
- squat2

  Input sequences used in the experiments. Each folder
   contains the centers and meshes directories as well as a
   settings.xml file, which specifies the sequence framerate.

Poster
- Kacerekova_Zuzana_2023.pub
- Kacerekova_Zuzana_2023.pdf

  Poster source file and generated PDF.

Results
- handstand_feet_apart
- handstand_hump_back
- handstand_wide_stance
- samba_arm
- samba_belly
- samba_belly2
- squat_2_mohawk
- squat_2_nose_job
- squat_2_slimming

  Experiment results.

Text_thesis
- A system for editing triangle mesh sequences with time-
    varying connectivity.zip
- A_system_for_editing_triangle_mesh_sequences_with_time-
    varying_connectivity.pdf

  Thesis source files and generated PDF.
```

Bibliography

[Au+08] AU, Oscar Kin-Chung; TAI, Chiew-Lan; CHU, Hung-Kuo; COHEN-OR, Daniel; LEE, Tong-Yee. Skeleton Extraction by Mesh Contraction. In: *ACM SIGGRAPH 2008 Papers*. Los Angeles, California: Association for Computing Machinery, 2008. SIGGRAPH '08. ISBN 9781450301121. Available from DOI: `10.1145/1399504.1360643`.

[Bar+18] BARILL, Gavin; DICKSON, Neil; SCHMIDT, Ryan; LEVIN, David I.W.; JACOBSON, Alec. Fast Winding Numbers for Soups and Clouds. *ACM Transactions on Graphics*. 2018.

[Bar23] BARKING DOG. *3D Free Modular Kit* [`https://assetstore.unity.com/packages/3d/environments/3d-free-modular-kit-85732`]. 2023. [Online; accessed 15-May-2023].

[Ben75] BENTLEY, Jon Louis. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*. 1975, vol. 18, no. 9, pp. 509–517. ISSN 0001-0782. Available from DOI: `10.1145/361002.361007`.

[Che+11] CHENG, Wang; CHENG, Ren; XIAOYONG, Lei; SHULING, Dai. Automatic skeleton generation and character skinning. In: *2011 IEEE International Symposium on VR Innovation*. 2011, pp. 299–304. Available from DOI: `10.1109/ISVRI.2011.5759655`.

[Dvo+22] DVOŘÁK, Jan; KÁČEREKOVÁ, Zuzana; VANĚČEK, Petr; HRUDA, Lukáš; VÁŠA, Libor. As-rigid-as-possible volume tracking for time-varying surfaces. *Computers & Graphics*. 2022, vol. 102, pp. 329–338. ISSN 0097-8493. Available from DOI: `https://doi.org/10.1016/j.cag.2021.10.015`.

[Dvo+23] DVOŘÁK, Jan; KÁČEREKOVÁ, Zuzana; VANĚČEK, Petr; VÁŠA, Libor. Priority-based encoding of triangle mesh connectivity for a known geometry. *Computer Graphics Forum*. 2023, vol. 42, no. 1, pp. 60–71. Available from DOI: `https://doi.org/10.1111/cgf.14719`.

[DVV21] DVOŘÁK, Jan; VANĚČEK, Petr; VÁŠA, Libor. Towards Understanding Time Varying Triangle Meshes. In: PASZYNSKI, Maciej; KRANZLMÜLLER, Dieter; KRZHIZHANOVSKAYA, Valeria V.; DONGARRA, Jack J.; SLOOT, Peter M.A. (eds.). *Computational Science – ICCS 2021*. Cham: Springer International Publishing, 2021, pp. 45–58. ISBN 978-3-030-77977-1.

[Fas07] FASSHAUER, Gregory E. *Meshfree approximation methods with MATLAB*. Singapore: World Scientific Publishing, 2007. Interdisciplinary mathematical sciences. ISBN 9789812706331.

[HDV19] HRUDA, Lukas; DVOŘÁK, Jan; VÁŠA, Libor. On evaluating consensus in RANSAC surface registration. *Computer Graphics Forum*. 2019, vol. 38, pp. 175–186. Available from DOI: 10.1111/cgf.13798.

[JKS13] JACOBSON, Alec; KAVAN, Ladislav; SORKINE-HORNUNG, Olga. Robust Inside-Outside Segmentation Using Generalized Winding Numbers. *ACM Transactions on Graphics (TOG)*. 2013, vol. 32. Available from DOI: 10.1145/2461912.2461916.

[JT05] JAMES, Doug L.; TWIGG, Christopher D. Skinning Mesh Animations. *ACM Trans. Graph.* 2005, vol. 24, no. 3, pp. 399–407. ISSN 0730-0301. Available from DOI: 10.1145/1073204.1073206.

[Kab76] KABSCH, W. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A*. 1976, vol. 32, no. 5, pp. 922–923. Available from DOI: 10.1107/S0567739476001873.

[KSO10] KAVAN, L.; SLOAN, P.-P.; O'SULLIVAN, C. Fast and Efficient Skinning of Animated Meshes. *Computer Graphics Forum*. 2010, vol. 29, no. 2, pp. 327–336. Available from DOI: https://doi.org/10.1111/j.1467-8659.2009.01602.x.

[Kav+07] KAVAN, Ladislav; COLLINS, Steven; ZARA, Jiri; O'SULLIVAN, Carol. Skinning with dual quaternions. In: 2007, pp. 39–46. Available from DOI: 10.1145/1230100.1230107.

[Le12] LE, Binh. Smooth skinning decomposition with rigid bones. *ACM Transactions on Graphics (TOG)*. 2012, vol. 31. Available from DOI: 10.1145/2366145.2366218.

[Lip+04] LIPMAN, Yaron et al. Differential coordinates for interactive mesh editing. In: 2004, pp. 181–190. ISBN 0-7695-2075-8. Available from DOI: 10.1109/SMI.2004.1314505.

[Mat23] MATH.NET PROJECT. *Math.NET Numerics* [https://www.mathdotnet.com]. 2023. [Online; accessed 15-May-2023].

[Par20] PARADYME GAMES. *Nebula Particle System and Skybox in Unity* [https://www.youtube.com/watch?v=r6ssghmcqu4]. 2020. [Online; accessed 15-May-2023].

[Roh21] ROHMER, Damien. *INF585/2020 - Computer Animation* [YouTube]. 2021. [visited on 2022-05-01]. Available from: https://www.youtube.com/playlist?list=PLkGB0Y1UEJxuwYFq7t2XBtIAzs95P7wR6.

[Sch] SCHAPEL. *Bijection* [online]. Wikimedia.org. [visited on 2022-05-07]. Available from: https://commons.wikimedia.org/wiki/%5C%5CFile:Bijection.svg. This file has been released into the public domain.

[SA07] SORKINE, Olga; ALEXA, Marc. As-Rigid-As-Possible Surface Modeling. In: 2007, pp. 109–116. Available from DOI: 10.1145/1281991.1282006.

[Sor+04] SORKINE, Olga et al. Laplacian Surface Editing. In: 2004, vol. 71, pp. 179–188. Available from DOI: 10.1145/1057432.1057456.

[SSP07] SUMNER, Robert W.; SCHMID, Johannes; PAULY, Mark. Embedded deformation for shape manipulation. *ACM Transactions on Graphics*. 2007, vol. 26, no. 3, p. 80. Available from DOI: 10.1145/1276377.1276478.

[TL] TURK, Greg; LEVOY, Marc. *Stanford bunny* [online]. Stanford University. [visited on 2022-05-01]. Available from: https://graphics.stanford.edu/~mdfisher/Data/Meshes/bunny.obj.

[Val22] VALEM TUTORIALS. *How to make a VR game - Unity XR Toolkit 2022* [https://www.youtube.com/playlist?\list=PLpEoiloH-4eP-OKItF8XNJ8y8e1asOJud]. 2022. [Online; accessed 15-May-2023].

[Váš20] VÁŠA, Libor. *Zpracování polygonálních sítí* [Courseware]. 2020. [visited on 2020-12-31]. Available from: https://courseware.zcu.cz/portal/studium/courseware/kiv/zpos/prednasky.html.

[Vla+08a] VLASIC, Daniel; BARAN, Ilya; MATUSIK, Wojciech; POPOVIĆ, Jovan. Articulated Mesh Animation from Multi-View Silhouettes. *ACM Trans. Graph.* 2008, vol. 27, no. 3, pp. 1–9. ISSN 0730-0301. Available from DOI: 10.1145/1360612.1360696.

[Vla+08b] VLASIC, Daniel; BARAN, Ilya; MATUSIK, Wojciech; POPOVIĆ, Jovan. *Articulated Mesh Animation from Multi-View Silhouettes dynamic mesh dataset* [https://people.csail.mit.edu/drdaniel/mesh_animation/]. 2008. [Online; accessed 15-May-2023].

[WH06] WALD, Ingo; HAVRAN, Vlastimil. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, pp. 61–69. Available from DOI: 10.1109/RT.2006.280216.

[Wik23a] WIKIPEDIA. *Kabsch algorithm — Wikipedia, The Free Encyclopedia* [http://en.wikipedia.org/w/index.php?title=Kabsch%20algorithm&oldid=1152183115]. 2023. [Online; accessed 15-May-2023].

[Wik23b] WIKIPEDIA. *Procrustes analysis — Wikipedia, The Free Encyclopedia* [http://en.wikipedia.org/w/index.php?title=Procrustes%20analysis&oldid=1151503166]. 2023. [Online; accessed 15-May-2023].

[Wik23c] WIKIPEDIA. *Root-mean-square deviation — Wikipedia, The Free Encyclopedia* [http://en.wikipedia.org/w/index.php?title=Root-mean-square%20deviation&oldid=1145735789]. 2023. [Online; accessed 15-May-2023].

[Zot] ZOTTIE. *CSG tree* [online]. Wikimedia.org. [visited on 2022-05-01]. Available from: https://commons.wikimedia.org/w/index.php?curid=263170. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/deed.en.

List of Figures

List of Tables

List of Listings