# Expressive Skinning Methods for 3D Character Animation

By

NICHOLAS TOOTHMAN
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

---

Michael Neff, Chair

---

Nina Amenta

---

Bernd Hamann

Committee in Charge

2020

I would especially like to thank my advisor, Professor Michael Neff, for his wellspring of knowledge, patience, and support. I once heard that it is best to choose an advisor based more on their compassion rather than solely on their academic prowess. I am most fortunate that mine has stellar performance on both of these measures. The same can be said for my committee. Professor Nina Amenta graciously provided time for brainstorming sessions, paper reviews, and a grant proposal that preceded this research. Professor Bernd Hamann saved me from a world of bureaucratic pain during my qualifying exam, and he has always shared ample enthusiasm and expertise with me and many other researchers. Thank you to all of you.

To the ModLab and its members, thank you for the best cross-disciplinary collaborations one could hope for in graduate school. It has been a joy working with you all. Professors Gina Bloom and Colin Milburn elevated the efforts of myself, Evan Buswell, Sawyer Kemp, and many others that followed to levels I could not reach alone.

To Gail Bimson and the physical therapy team on campus for helping me stay in the game when I felt broken and unable to continue.

Thank you to my family and friends, for helping me find my way here. To Lance Simons, for the camaraderie, proofreads, game jams and 2THS1M projects, myriad errands and adventures on and off campus, and the mutual commiseration when code is broken.

To Bea, for all of your love, the long walks and snack runs, the moves for jobs and internships, and the sheer volume of animation we've shared together. This is just a small sample from the population of reasons I am eternally grateful, but you already know the rest.

# Contents

Expressive Skinning Methods for 3D Character Animation

## Abstract

Real-time three-dimensional (3D) character animation has benefited greatly from advancements in skinning and surface deformation. While both fields offer tremendous techniques for controlling character shape over time, few methods address the needs of both. I propose such a method to enable greater artistic control and expressive potential. First, I introduce a skinning algorithm to address common artifacts and offer configurable shape behavior around joints. Second, I discuss embedding surface deformations in the skinning process. Both of these operate by adjusting *scale vectors* to provide the character's final shape. These vectors run from attachment points located within the mesh volume to vertex positions. Smooth skinning is achievable by combining a rigid transformation on the attachment point and a blended transformation on the scale vector. Further adjustments to scale vectors can produce surface deformation. This representation permits a variety of surface deformation methods. To demonstrate the effectiveness of the approach, I have written a graphics processing unit (GPU)-accelerated implementation of the deformation method, a rendering pipeline built to support and compare various skinning methods, and an input system well-suited for authoring character animations. Future work includes furthering shape control in the skinning method, developing more expressive deformation tools, and incorporating virtual reality (VR) support to provide an embodied interface for 3D animation using my deformation methods.

# 1 Introduction

To animate is to impart something with movement over time to represent physical interactions or create the appearance of sentience. In this usage, animation is a tangible practice where we directly manipulate objects with our bodies. In more concise terms, animation brings something to life. Humans do this using toys and props for entertainment, storytelling, and play. For a short time, the objects we animate become living characters, enabling us to have experiences vicariously through them. The choices of movement and timing can produce myriad emotions, behaviors, and actions that resemble what we perceive in reality from living things. In a sense, we practice how to animate ourselves by using non-living things as avatars in low-risk environments, where ludic behavior is welcome and encouraged. In modern usage, animation more often refers to visual media that exhibits change over time, but these qualities are still immediately relevant, and the objective remains to make things come alive. Across usages, the elements of animation are space and time. Returning to a technical definition, animation is the act of repeatedly defining what space an object occupies at what time.

Traditional animation is the phenomenon of "moving pictures" made possible by the persistence of vision. Due to the brain's ability to briefly retain an object's appearance after it is no longer visible, this optical illusion enables humans to make sense of rapidly-changing images (frames) and comprehend them as movement over time. Critically, for this phenomenon to occur, the difference between consecutive frames must be sufficiently large to recognize the change, but not so large that the connection between them is lost and the illusion breaks. This is a fundamental concept throughout animation known as believability, and achieving it requires considerate effort.

In traditional animation, each frame is drawn by hand, then sequenced together as film to create motion. This optical illusion is called beta movement, as described by Max Wertheimer in 1912 [61], and achieving it requires a minimum frame rate somewhere between 10 and 12 frames per second (FPS). At the film and animation standard of 24 FPS, a 30-second scene requires 720 frames. While many frames can likely be reused to hold or repeat a pose, the

work required is still significant. To manage the scale of hand-drawn animation, the labor is divided by hierarchy. After developing a storyboard to outline the general scene timeline, principal animators draw the key frames, when the characters change directions or stop, and provide in-between charts indicating how and when the pose should change between keys. In-between animators then use the key frames and character references to draw the movement "in between." It follows that the key frame's quality drastically impacts how well the animation looks.

Over the last century, the rapid co-development of animation techniques and technologies has led to lower cost of labor and greater artistic possibilities, enabling the emergence of animation powerhouses such as Disney and Studio Ghibli. While the style and appearance of traditional animation remains incredibly popular, computer animation has steadily risen as the dominating art form due to the indisputable convenience and speed advantages. Specialty tools and equipment, such as the multiplane camera for parallax and depth effects and splines for drafting shapes with smooth curvature, have been replicated in software, made user-friendly, and widely distributed. Characters can be posed interactively to make key frames, and because they are represented with geometric positions and orientations, in-between frames can be automatically computed through interpolation and refined with animation curves.

Despite inherent differences between the media, the principles established for traditional, hand-drawn animation apply well to 3D animation. Without them, movement looks rigid and artificial in both media. Principles such as: squash and stretch, weight, anatomy, silhouette, depth, volume, and secondary and overlapping movement, help turn sequential frame changes into believable character animation. Most of all, these concepts highlight the importance of shape control over time. The essential goal of my research is to make the freedom of shape change from hand-drawn animation possible for 3D character animation. At this point, it will be helpful to introduce some terms and concepts from animation and 3D computer graphics.

## 1.1 Primer on 3D characters

Traditional animation depends on character references. Made by a character artist, these portray the character's general appearance from a range of angles and serve as guidelines to keep the character's appearance consistent between animators as scenes and outfits change. Digital characters also have reference sheets for the same reasons, but they also help guide the modeling process to create the character's shape. Moreover, rather than remodeling the same character for each frame, once constructed, a digital character can be deformed repeatedly to various poses. To deform meshes of arbitrarily high geometry counts, animators typically control the character using handles with relatively few degrees of freedom compared to the surface - a skeleton made of a few joints, or a freeform deformation lattice. A *character rig* consists of the geometry required to define its surface, a set of materials and textures that describe how the surface appears, and a skeleton that is used to control the character's pose. The following is a brief coverage of the terms and concepts used for characters in computer animation. Although there are both 2D and 3D digital character representations, the research presented here mainly focuses on the 3D case.

- Vertex: a vector in 3D representing a surface point
- Edge: connectivity between two vertices
- Face: a polygon between 3 (triangle) or 4 (quadrilateral or quad) vertices. When modeling a character, it is convenient to build the surface using quads, but rendering pipelines in graphics hardware typically expect triangles. Two faces may share an edge between two vertices.
- Indices: Integer values describing an offset into a block of sequential memory used to store geometry data. For example, faces are stored as indices into a vertex array.
- Normal: a unit-length vector in 3D representing the direction perpendicular to a surface, used when shading to determine how much light is absorbed and reflected.

- Texture: a function used to compute a surface's appearance. Usually this is implemented as an image file to store color, normal, or deformation data.

- UV: a parameter tuple in texture space ($[0, 0], [1, 1]$) representing input coordinates for a texture.

- Texture mapping: a process to convert a 3D surface into a 2D representation that fits in texture space . To appreciate the challenge, consider the act of peeling an orange and flattening the pieces to fill a square.

- Mesh: a set of vertices, edges, faces, indices, normals, UVs, and textures used to discretely define the character's surfaces.

- Transform: an affine transformation for deformation, usually restricted to translation, rotation, and scale.

- Joint: a transform approximating an anatomical joint, used as a control handle for the mesh. A joint may be connected to a parent and children.

- Skeleton: a hierarchy of joints whereby a root joint placed at the mesh's center of gravity determines the overall pose and descending joints more finely articulate it. Typically the pelvis joint serves as the root, although it may have a parent joint placed between the feet to aid in floor positioning.

- Skin weights: a set of index-scalar values for every vertex to indicate the degree of influence the indexed joint has on the vertex. Automatic methods for skin weight computation give an initial mapping, but usually manual adjustment is required to fix erroneous deformation.

- Binding: the process of computing skin weights that relate the mesh to the skeleton. It is standard practice to model a character and its skeleton so that they are standing straight up with their arms to the side at 90 degrees (T-pose) or 45 degrees (A-pose). This pose provides ample spacing around limbs to improve binding quality. Typically,

binding is somewhere between an automatic and manual process, requiring iterations to obtain satisfying skin weights.

- Key: a transform value that is specified to be an exact value at a specific time. A collection of different key values that occur at the same time is collectively referred to as a keyframe.
- Interpolation: the act of computing intermediate values between two keys.
- Animation curve: a linear function that defines how the value between two keys should change for interpolation.
- Framerate (FPS): how many frames of animation should be processed per time interval. A standard FPS for animation is 24, and 60 for games.

## 1.2 Motivation

With this representation, animation software can pose and render a character rig in real-time. This is made possible by the graphics pipeline, which is responsible for deforming the mesh vertices based on the skeleton pose ("skinning"), culling non-visible geometry, and shading the visible geometry based on the given textures and lighting. Skinning usually occurs in the vertex shader stage of the graphics pipeline, so for real-time (24+ FPS) animation, the graphics hardware and skinning algorithm must be sufficiently fast, and the mesh geometry sufficiently low, especially if multiple meshes are being deformed per frame. The straightforward computation in most skinning shaders helps maintain high performance at the cost of certain deformation artifacts. In addition, because skinning deformations are principally anatomical, there is little avenue for freedom of shape change.

One of the strengths of 2D animation is the free shape change that it permits, but achieving the same freedom of shape in 3D animation is not as straightforward. While having a skeleton makes interactive posing and refinement quite easy for 3D animation, it also limits the character's range of deformations. Conversely, while freeform deformation

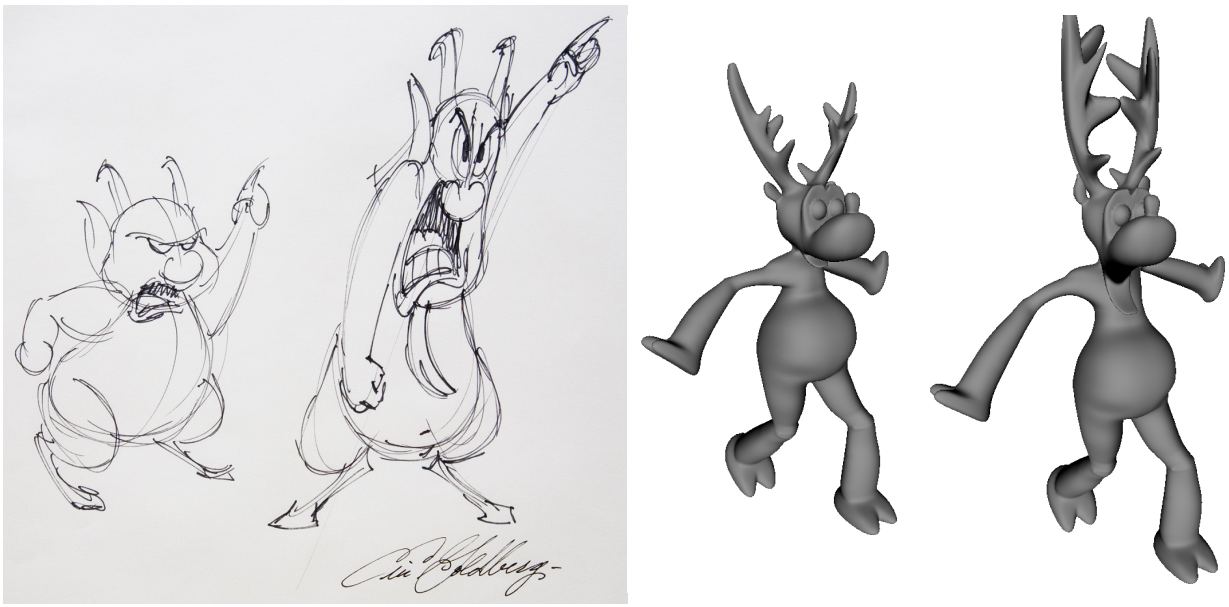


Figure 1: Exaggeration for expressive control in 2D and 3D spaces.

techniques may preserve surface features, they do not adhere to anatomical constraints. To approximate hand-drawn animation shape changes, the digital animator needs to use both skeletal skinning and surface deformers, making explicit the order of operation, influence size, and constraints required to achieve a desired pose. A major drawback of this approach is the lack of cohesion between anatomical and surface deformation; the artist is responsible for using both in such a way that the effects complement one another. Ideally, the artist would like to have the convenience and consistency of a digital character rig with the shape control capabilities of hand-drawn characters. A valuable tool for artists would then offer the speed, interactivity, and convenience of 3D animation with the freedom and expressive potential of drawing by hand. Animating characters with such a tool necessitates a deformation method more flexible and expressively capable than skeleton-based methods or surface editing techniques alone. It would also require a framework for real-time use, offering features to support rapid, penalty-free exploration of ideas and interactive animation.

### 1.2.1 Real-time animation

Modern games and applications offer a substantial amount of content customization. Avatar creation is a meta-game of adjusting dozens of sliders until the appearance is satisfactory. The sheer number of options creates an impressive subspace of possible character shapes, but still restricts appearance. The skinning and deformation techniques discussed later could loosen this restriction by offering easy, interactive editing of character appearance. A growing number of titles use in-game cinematics (or *cutscenes*) rendered in real-time using the game engine, as opposed to playing a pre-rendered video, which allows custom player avatars to naturally fit into the experience. In addition, it offers great narrative potential, wherein the player might actively participate in the storytelling through various methods, including: camera and player control, choosing characters to play certain roles, and even custom animations. The potential offered by in-engine cutscenes motivates deformation methods that perform well in real-time.

## 1.3 Approaches for shape control

Having discussed the differences in freedom of shape change between 2D and 3D animation, I introduce the most practical methods, and their limitations, for shape control when dealing with 3D character rigs next.

### 1.3.1 Binding

Skeletons provide an intuitive control handle for character posing; the artist simply rotates joints to pose the character as desired. Before a skeleton can deform a mesh, the *binding* process first defines skin weights that bind vertices to one or more bones. Binding only needs to occur once, or whenever joints need to be added to or removed from the skeleton graph. Automatic binding methods can skin weights that produce smooth skeletal deformations, but they may require manual touch-ups using painting tools before the animation quality reaches an acceptable level.

### 1.3.2 Skinning

During animation, a *skinning algorithm* uses the skin weights to compute new vertex positions based on each bone's transformation from the pose used during binding. With few exceptions, the quality of many skinning methods depends primarily on joint placement and skin weight quality. Writing a skinning algorithm essentially means deciding how changes in the skeleton will influence vertex positions. Having a skinning method that is less sensitive to skin weight variation helps attenuate the aforementioned issues, but some artifacts are inherent to the algorithm used and can only be addressed by changing the algorithm. Even with occasional artifacts, skinning algorithms are valuable for helping maintain basic anatomical shape during deformation. In addition to providing a logical, hierarchical method of control over meshes of arbitrary resolution, skinning algorithms tend to map well to graphics hardware, making them suitable for real-time animation. Animating a skeleton over a series of frames is drastically faster, requires less storage, and is easier to revise than attempting to reproduce the same animation with surface-only deformations.

Assuming skin weights of sufficient quality, skinning algorithms provide smooth mesh deformation around joints. Even when using high-quality weights, the skinning method in use can produce artifacts in certain skeletal poses that compromise the final shape quality. We see these artifacts in linear blend skinning (LBS) with joint collapse on large twists (see Figure 2), and in dual quaternion skinning (DQS) with spherical bulging, Figure 3). Common approaches for resolving these include: weighted blending between LBS and DQS, defining non-functional "support" or "twist" joints to exhibit greater influence over such vertices, constraining certain degrees of freedom on joints (such as confining elbow and knee rotations to a single axis between a minimum and maximum angle), and, of course, developing new skinning methods that intrinsically avoid the same artifacts, occasionally swapping them for more subtle ones.

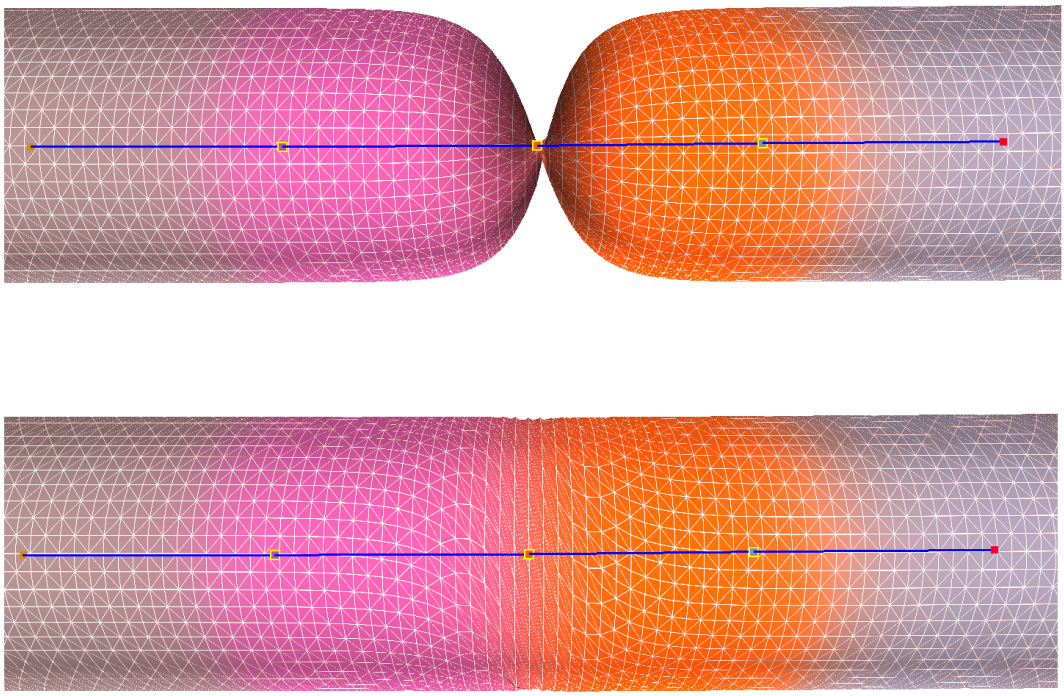


Figure 2: 180 degree twist transform on LBS (top) and DQS (bottom). The transform produces volume collapse on LBS. DQS avoids collapse, but tightly packs the twist within a small region.

### 1.3.3 Surface deformers

Despite their performance and usability advantages, skinning algorithms tend to limit character shape control to the skeleton's possible joint angles. Some may permit bones to stretch or bend, but for additional shape change, 3D animators turn to other tools for deformation: freeform deformation lattices, blend shapes, and displacement maps, to name a few. More involved techniques that are generally unfit for real-time use include soft-body physical simulation and finite element methods.

## 1.4 Statement of problem

The problem to solve is how to easily provide a comparable range of expressive shape control for 3D characters as there is for hand-drawn characters, Figure 1. With the deformation

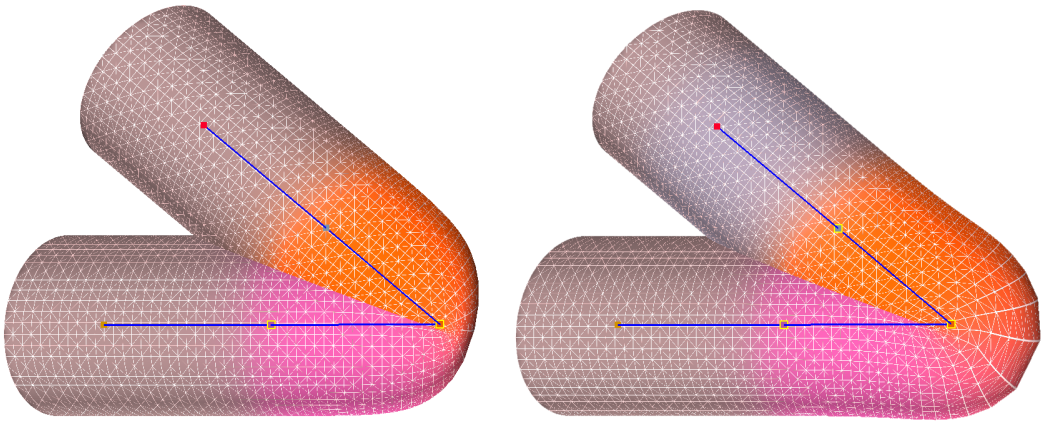


Figure 3: 135 degree bend transform on LBS (left) and DQS (right). More volume loss around the bent joint with LBS. DQS prevents volume loss, but introduces bulge around the joint.

method I propose, skeletal and surface deformations may occur in tandem without explicit awareness of one another. Given that both deformations are likely to vary over time, it is vital to define the skinning algorithm such that it behaves well in the presence of either deformation type. I should support:

- **Commutative skeleton and surface deformations:** applying one before the other should not result in different shapes.
- **Non-bind deformations:** requiring the skeleton and mesh to match their input bind values before authoring new deformations is cumbersome and inconvenient for the artist. She should be able to deform the surface in any skeletal pose.
- **Shape-aware deformation controls:** ideally, the artist has some degree of control over how the mesh behaves should it come into contact with itself. For example, muscles around a joint might come into contact and bulge laterally, proportional to the amount of contact. The method should be able to detect and appropriately handle self-intersection cases.

I address shape control in two manners. First, I provide a novel skinning algorithm that circumvents existing skinning artifacts and offers configurable shape behavior around joints. This approach to skinning decomposes mesh vertex positions into two values. An interior *attachment point* that lies somewhere along the skeleton is chosen for each vertex to serve as a relative origin. The difference between a vertex and its attachment point defines a *scale vector*. With this decomposition, anatomical and surface deformations are created by recomputing attachment points and scale vectors with optimization algorithms and constraints to achieve a target shape. By accounting for both surface-level and skeletal deformations in the optimization, the skinning algorithm enables a greater range of shape control than traditional skinning algorithms. After shape control, the accompanying challenge in this thesis concerns providing the interface and tools necessary to make use of new techniques. To provide these tools and interface, I desire software that is sufficiently simple for novices to use, but powerful enough for advanced tasks. A VR interface provides the artist with tools to control the rig's deformation using intuitive interactions, such as drawing on the surface, defining paths and timing with gesture, hands-on skeletal posing, and natural camera operations.

# 2 Prior research

## 2.1 Skinning

### 2.1.1 Linear blend skinning and dual quaternion skinning

Together, skinning algorithms and skin weights allow a small set of joint angles to control the shape of a high-resolution character mesh, providing an intuitive control abstraction. By associating vertices with one or more bones in the skeleton, the mesh deforms along with the bones to which they are bound. Linear blend skinning (LBS), also known as skeletal subspace deformation (SSD) and smooth skinning, introduces blended transforms as a sknning technique [36]. LBS associates vertices with multiple bones using skin weights; given a mesh $M = (V, E)$ of vertices $v_i \in V$ and edges $(v_i, v_j) \in E$, and a hierarchy of affine transformations $B$, there is a function mapping each $v_i \in V$ and $b_j \in B$ to a scalar value $w_{ij}$, usually $\in [0, 1]$. To compute smooth linear vertex transformations, the sum of influences for $v_i$ should equal one: $\sum_{j \in B} w_{ij} = 1$. This constraint, called partition of unity, is essential for LBS. Each vertex's skinned position $v_i'$ is found by transforming the initial (or bind) position $v_i$ by a weighted average of the bone's transforms:

$$v_i' = (\sum_{j \in B} w_{ij} T_j) v_i. \tag{1}$$

Partition of unity ensures that the sum yields a valid transformation for $v_i$. It also informs the computation of skin weights ($w_{ij}$) for smooth deformation. In the construction of 3D rigs, the rigging and binding phases involves building the mesh's skeleton and defining skin weights that produce the desired mesh shape as the skeleton moves. Automatic binding methods compute values for $w_{ij}$ with only a few tuning parameters, but the resulting set typically requires manual corrections using artist tools to adjust bone influences.

Prior research in automatic binding improves the quality of skin weights in various admirable ways. Baran and Popović model bone influence as heat diffusion across the surface [4]. Other research improves on this by incorporating visibility checks between bones

and surfaces [60]. Jacobson et al. compute skin weights by minimizing the mesh's Laplacian energy subject to constraints, which requires a discrete tetrahedral volumization of the mesh interior [19]. For volumization to be successful in tools like TetGen [45], the mesh must meet certain requirements: no degenerate geometry, no holes in the surface, etc. Even with professionally made models, these properties can be difficult to guarantee and tedious to fix. Dionne and de Lasa leverage graphics hardware to quickly compute orthogonal, axis-aligned mesh slices, then generate interior voxels for geodesic distance tests between surfaces and bones [11]. This technique handles meshes that would not be suitable for TetGen.

Despite the high quality of these skin weight computations, LBS contains artifacts: unwanted or undesirable deformations. To understand how these occur, we look at the bone transforms $T_j$. This is a bone's transformation from its bind pose to its current pose. Generally, these are represented as 4x4 transformation matrices, using homogeneous coordinates to represent any combination of any affine transformations, namely translation, rotation, and scale for posing purposes. Bone transforms like $T_j$ are perfectly fine when used for rigid deformation, where the same transformation is applied to all vertices in a mesh. But when computing the weighted average in Equation 1, some problems can arise. A minor change to Equation 1 shows $v_i'$ as the weighted average of each bone $j$'s rigid transformation of $v_i$:

$$v_i' = \sum_{j \in B} w_{ij}(T_j v_i). \tag{2}$$

Thus, if a vertex is influenced equally by two adjacent bones, and one of them is twisted 180° around, then $v_i'$ will be the midpoint between $v_i$ and $R(180)v_i$, lying on or near the joint between these bones. As nearby vertices experience varying degrees of this effect, it causes a pinching artifact and loss of volume around the bone being twisted, collectively forming the "candy wrapper" artifact seen in Figure 2, top. If the bone rotates in a swing, similar volume loss occurs near the joint's outer bend to create the "macaroni elbow" artifact (Figure 3, top). The explanation for this behavior is due to the effects of linear interpolation on transformation matrices, particularly the rotation component. The bone's 3D orientation is a member of the 3D rotation group SO(3). This group can be viewed as a unit circle,

where each point on its boundary represents a valid rotation between 0 and 360° on some axis. Rotation matrices cover this loop once; a 0° and 360° rotation share the same point on the circle. When two rotations are linearly interpolated, the result occurs somewhere within the circle's area, not on its boundary. For small differences between rotations, it's possible to normalize the interpolated result, effectively pushing the interior point to the closest boundary point, to make it a valid rotation. But if two rotations are separated by 180°, the interpolated result between them is degenerate and cannot be normalized (Figure 4, left).

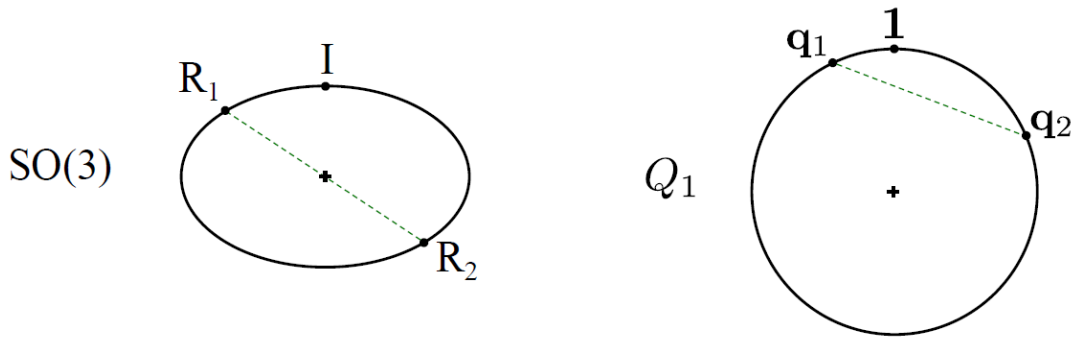



Figure 4: SO(3) rotations and averages between matrices (left) and dual quaternions (right). Image from SIGGRAPH 2014 Course *Skinning: Real-time Shape Deformation* [20].

Although this explains why some rotations drop to zero with LBS, it does not mean 3D rotations are fully incompatible with linear blending. Instead of using matrices, quaternions allow a rotation representation with double coverage of SO(3). That is, for quaternions, 0° and 720° rotations share the same point, and a 360° rotation is halfway around the unit circle (Figure 4, right). With this expanded range, the interpolation between 180° rotations can be normalized with acceptable precision. With this insight, the method of Kavan et al. represents bones as screw transforms using dual quaternions [27]. Like LBS, dual quaternion skinning (DQS) computes a weighted average transform for each vertex, but the blended dual quaternion more closely preserves distances between vertices and their centers of rotation. In the process, it introduces a new artifact known as spherical bulge. The method is so effective at preserving distance that it produces a recognizable spherical

shape around large joint swing rotations (Figure 3, bottom). DQS avoids the candy wrapper and macaroni elbow artifacts for joint twists, but introduces bulge on swing rotations and densely distributes twist rotations between bones, see Figure 2, bottom.

### 2.1.2 Artistic controls

To minimize artifacts, character riggers have developed various techniques that highlight the roles of both art and science in the discipline, particularly when manually adjusting skin weights. Surface painting tools with soft, configurable brushes are especially powerful, enabling skin weight refinement using the same movements as sketching or coloring on paper. For areas that need additional help, adding "support bones" to the skeleton hierarchy is a standard technique to give the artist finer control over the skinning deformation. These bones exist near bendable regions of the skeleton to provide influence on the region when computing skin weights. In doing so, they remove influence from the true bones and thus reduce the artifact's presence. For example, it is common to see a shoulder joint have two children at the same position: *elbow*, the bone which has children to continue the skeleton, and *elbow_twist*, a childless bone that exists purely to reduce the degree of volume collapse when the elbow twists. Support bones also often help produce secondary animation effects and correct undesired deformations in certain poses. Bones may have children closer to the surface to create tissue movement, such as for a large belly during a walk cycle. Because LBS and DQS are quite easy and inexpensive to compute, the mesh deformer can execute both and choose a final value that's linearly interpolated between the two results.

### 2.1.3 Decompositions

While the techniques described above go quite far in managing artifacts, other can issues with LBS and DQS arise under certain conditions. Both methods require bone lengths to remain fixed, limiting the methods to rotation-only skeletal deformations. Changes in bone length introduce undesired growth or shrinking around the nearby mesh. Issues like these serve as motivation to change the skinning formula. Stretchable, twistable bones (STB)

addresses artifacts caused by changing bone lengths and allows twist to be spread along a bone [23]. An extension of LBS, STB separates joint transforms into separate channels, namely bone length (scale), joint swing and twist (rotations), and position (translation), then uses these to compute the vertex transform in a more decomposed manner. The method relies on traditional skinning weights as well as *endpoint weights*, which map each vertex to a parameterized position along the length of a bone. These weights determine the degree of stretch and twist to apply to a vertex based on its position along a bone. STB makes improvements to LBS and DQS and has impressive resistance to artifacts, but the skinning results are still sensitive to skeleton, skin weight, and endpoint quality. The concept of vertex parameterization seen here proves to be essential for our approach to mesh binding in Section 3, as well as in other research that operates by attaching the surface and skeleton [29].

Another "modified LBS" approach resolves artifacts by pre-computing optimal centers of rotation for each vertex [31]. The intuition is that vertices can be clustered by their skin weight similarity, and each cluster's general bind shape should be preserved as much as possible when the skeleton changes pose. Each vertex has an optimal center of rotation to produce this effect, and Equation 1 is decomposed to apply the blended transform rotation from the optimized center. As a result, the method avoids creating volume loss effects. The notion of optimized centers for each vertex is intriguing, but the results found are optimized for an LBS-based skinning technique, and, as I show in Section 3, unsuitable for my own vertex-skeleton attachment needs.

### 2.1.4 Implicit and elastic surfaces

One approach to skinning with surface awareness uses implicit surfaces to simulate contact modeling [56, 58]. Here, the mesh undergoes segmentation based on skinning weights. Poisson disk sampling on each segment provides centers for Hermite radial basis functions (HRBFs), which compose an approximation of the segment as an implicit surface. Vertices track their offset from the $f = 0.5$ iso surface in bind pose. Traditional LBS-style skinning is performed on both the mesh vertices as an initial guess and on the implicit surfaces.

Composition operations blend separate surfaces into one, then each vertex marches to return to its offset. The marching steps interleave with tangential relaxation to improve the final result. The initial technique produced artifacts during large joint bends, which causes some vertices to project onto the incorrect side of the scalar field's medial axis [56]. The revised technique omits traditional LBS entirely, but still uses skin weights to compute per-vertex rotation matrices for an as-rigid-as-possible relaxation step [58]. To speed up convergence, the current frame uses the previous frame's results as an initial guess. Doing so, however, makes the animation output dependent on its history.

Other research has proposed methods to provide skeletal control over characters while supporting elastic deformations. Capell et al.'s method achieves skeletal control and secondary motion from physical forces by use of a volumetric mesh with bones confined to edges in the control lattice [8]. This method aims to strike a balance between skeleton-driven and secondary animation. This method depends on physical simulation, although the use of a skeleton helps to localize and reduce the computation cost. Other approaches calculate a deformation energy over the mesh that controls how much it can bend [63, 7, 6, 44]. While powerful, physics-based approaches do not accommodate the largely unrealistic, free shape deformations often used in more cartoony animations, as favored by our method.

Vector radial offsets have been used with spline-based skeletal animation with impressive results [13]. Here, the authors apply spline-aligned deformation to circumvent LBS artifacts [46] and attach high-resolution, radial FFD grids to each joint, which allows for deformation styles – surface deformations decomposed into frontal, lateral, and radial scaling values. Skeletal animation with deformation styles is possible, but the deformation style itself is static.

The technique most similar to our own is projective skinning, which also employs dynamics (projective rather than position-based), does not depend on skin weights, and connects the surface and skeleton [29]. Here, the authors volumize the skeleton and resolve contacts with the surface, while we project the surface onto the skeleton. Their approach to deformation involves least-squares energy minimization to manage elastic strain, skin stretch, and colli-

sion. Our approach achieves energy minimization by explicitly simulating springs between vertices and the skeleton to find a relaxed shape that conforms to the rig's parameters.

## 2.2 Deformation

### 2.2.1 Average-based methods

In contrast to skeleton-based deformation, mesh deformation schemes offer superior control over arbitrary mesh regions. Volumetric or cage-based methods aim to support character posing and elastic deformation by processes similar to skinning, namely computing $v_i'$ as a weighted average based on $v_i$ and some control handles. Free-form deformation (FFD) lattices are well established, simple to create, and powerful tools for deforming space [43, 35]. Keyframing lattice control points can generate surface animation, but using even small lattices in 3D ($4 \times 4 \times 4$) quickly raises the number of controls the animator must maintain. Lattices also lack the ability to preserve local geometry over a mesh, so features like wrinkles may get lost if the deformation is extreme. Other deformers can provide multiple control handle types, such as cages, bones, and points. Harmonic coordinates rig a mesh to a bounded cage volume for deformation using generalized barycentric coordinates [24]. Related approaches use mean-value coordinates [25] or higher order barycentric coordinates [30]. With these techniques, mesh editing is limited by the cage resolution, and achieving a particular mesh pose can require awkward cage transformations. Such techniques also lack the higher-level convenience of skeletons. Performing edits like adding muscle bulges or creases becomes easier, but bending an elbow or posing fingers is more difficult. While it is possible to hierarchically assign FFD lattices to bones for simultaneous mesh deformation and skeletal animation, it complicates the deformer graph and may require considerable adjusting to perform correctly. This makes them ideal for "hands-off" deformations, where the lattice's animation is defined, constrained, and driven exclusively by other handles, such as skeleton joints.

Overall, with enough time and practice, the shape control for this family of deformers

is quite capable, but they may not account for the mesh's topology or implied anatomy. Prior research has taken various approaches to addressing this. One approach is pose-space deformation, where an artist creates multiple example poses from the same mesh, which can then be interpolated or used with inverse kinematics to change shape [33, 49]. Implicit surfaces have also been used for surface authoring: volumetric extrusions of defined muscle surfaces create smooth, plausible surface deformations, while position-based dynamics add physical effects to the surfaces due to skeletal movement [42].

As a general rule of smooth surface deformers, a vertex $v_i$ should experience changes similar to its immediate neighbors. The blend-based skinning methods described in Section 2.1 meet this expectation by mapping similar weight sets to vertices with similar positions. To abide this rule for surface editing, we consider LSE and ARAP.

### 2.2.2 Laplacian surface editing

Laplacian surface editing (LSE) is a popular mesh deformation technique best known for its ability to preserve surface details, such as skin wrinkles or scars, while performing a smooth transform [48]. The input to LSE is a set of static vertices that define boundaries for the edited region, along with a set of handle vertices that lie within this region of interest (ROI). The artist transforms the handles to approximate a desired shape change, and LSE computes vertex positions for the ROI that accommodate the edit while maintaining the positions of static and handle vertices, see Figure 5. LSE can preserve surface details because as the region deforms, the solver minimizes changes to each vertex's position differential $\delta_i$, which is the difference between a vertex and the average of its immediate one-ring neighborhood:

$$\delta_i = v_i - \frac{1}{d_i} \sum_{j \in N_i} v_j, \tag{3}$$

where $N_i$ is all the vertices $v_j | (v_i, v_j) \in E$, and $d_i$ is $|N_i|$. It is possible to compute all $\delta_i$ individually, but linear algebra makes the same process achievable with a few operations. Given the sparse matrix for mesh adjacency $A_{n \times n} (A(i,j) = 1$ if $(v_i, v_j) \in E, 0$ otherwise) and the diagonal matrix for neighborhood cardinality $D_{n \times n} (D(i,i) = d_i)$, the Laplacian matrix

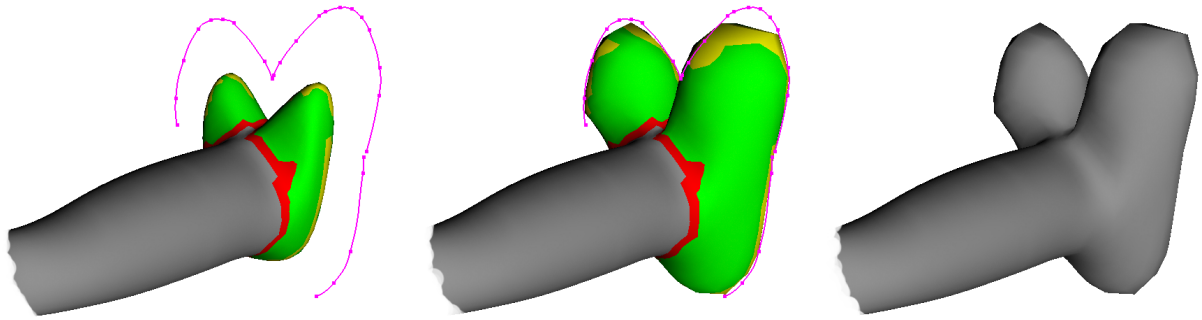


Figure 5: Example of LSE on a reindeer hoof. The anchors binding the region of interest are colored red, and the vertices to be deformed are colored green. The handles lay on the character's silhouette and are colored yellow.

of the mesh is found as $L = D - A$. For LSE, the value $L = I - D^{-1}A$ is used instead. Given the dense matrix of vertex positions $V_{n\times 3}$, the entire set of $\delta_i$ can be found as $\Delta_{n\times 3} = LV$. Computing $LV$ effectively fills $\Delta$'s rows with the result of Equation 3.

An important thing to note is $L$ has rank $n-1$, which means we can recover $V$ from $\Delta$ as long as we supply at least one fixed vertex position and solve a sparse linear system. To find the edited mesh $V'$, the objective is to find and apply the optimal transform $T_i$ for each $v_i$ such that changes to $\delta_i$ from its bind value are minimized, with $T_i$ minimized as

$$\min_{T_i} \left( ||T_i v_i - v_i'||^2 + \sum_{j \in N_i} ||T_i v_j - v_j'||^2 \right). \tag{4}$$

$T_i$ can be approximated by a matrix derived from three unknown vectors corresponding to isotropic scale $s_i$, rotation $h_i$, and translation $t_i$. If $A_{i(d_i\times 3)}$ holds the positions of $v_i$ and its neighbors, and $b_i$ contains the positions of $v_i'$ and its neighbors, the objective is to minimize $||A_i(s_i, h_i, t_i)^T - b_i||^2$, a linear linear least-squares problem solved as $(s_i, h_i, t_i)^T = (A_i^T A_i)^{-1} A_i^T b_i$.

The resulting $T_i$ are only approximations of rotation and isotropic scale transformations, and larger handle edits introduce greater error manifesting as drastic volume changes and

self-intersecting geometry. To partially address this issue, the technique can incorporate both $V$ and $V'$ when finding $T_i$, and then solve the system again. An optional follow-up procedure takes surface normals into account for further improvement in vertex placement and detail preservation. For very large angles of rotation, there's an approximate reconstruction method that can be applied first, then refined with LSE [34].

Ultimately, minimizing changes to differential coordinates will preserve local features, such as wrinkles in skin or clothing, while smoothly varying vertex positions to accommodate the transform. This is fine for lightweight deformations that alter a character's appearance in minor ways, but larger handle transformations introduce distortions. The ROI's vertex classification is typically static, so more drastic surface edits may necessitate larger ROIs to minimize distortion. Nevertheless, LSE is sufficiently fast for interactive editing, fit for making "little touches" to impact animation (see Figure 6), and as a reasonable first approximation for other surface deformation methods.

Figure 6: Bicep flex without surface deformation (left) and with (right).

### 2.2.3 As-rigid-as-possible surface modeling

Like LSE, As-Rigid-As-Possible Surface Modeling (ARAP) computes vertex positions that minimize an energy formulation [47]. This research provides two methods: one for computing the per-vertex rotations that minimize a deformation energy across the mesh, and another for computing the vertex positions. Interleaving these methods leads to improved deformation

quality. A cell $C_i$ contains the vertex $i$ as well as $i$'s one-ring neighborhood $j \in N_i$. Rigidly transforming $C_i$ to produce $C_i'$ means there exists a rotation matrix $R_i$ such that:

$$v_i' - v_j' = R_i(v_i - v_j), \forall j \in N(i). \tag{5}$$

To create a smooth transformation instead of a rigid one, the authors attempt to find $R_i$ for each $v_i$ that minimize the energy formulation. The sum of Equation 6 for all $v_i$ yields the overall energy of the deformed mesh $M'$.

$$E(C_i, C_i') = \sum_{j \in N(i)} w_{ij} ||(v_i' - v_j') - R_i(v_i - v_j)||^2. \tag{6}$$

The intended outcome is a surface that deforms gradually by helping cells maintain their shapes. $w_{ij}$ is the cotangent weight computed as

$$w_{ij} = \frac{1}{2}(\cot \alpha_{ij} + \cot \beta_{ij}), \tag{7}$$

where $\alpha_{ij}$ and $\beta_{ij}$ are the opposite angles from the two triangles (one for boundary edges) sharing the edge $(i, j)$ [41, 37].

With ARAP, there are two unknown variable sets: the new vertex positions $v_i'$ and the rotation matrices $R_i$. To account for both, the technique alternates between solving for each set, then uses the results as input for the next iteration. Some skinning techniques that use ARAP-like energy terms simply use LBS to compute $R_i$ directly for each vertex, leaving only $v_i'$ to solve for [58]. Otherwise, reformulating Equation 6 to determine the optimal $R_i$ for a given set of $v_i'$ involves finding the singular value decomposition of a covariance matrix built from $N_i$. This is the energy formulation to minimize to find $v_i'$:

$$E(M') = \sum_{i=1}^{n} w_i E(C_i, C_i') = \sum_{i=1}^{n} w_i \sum_{j \in N(i)} w_{ij} ||(v_i' - v_j') - R_i(v_i - v_j)||^2. \tag{8}$$

The value of Equation 6 is an integrated quantity, which means the cell $C_i$'s energy is proportional to its area, so is suitable to use $w_i = 1$. ARAP could also use $w_i = A_i$, the

Voronoi area of $C_i$, and $w'_{ij} = \frac{1}{A_i} w_{ij}$, but the cell area cancels out in the sum. Taking the partial derivative of $E(M')$ with respect to $v'_i$ computes the new vertex positions $v'_i$:

$$\begin{aligned}
\frac{\partial E(M')}{\partial v'_i} &= \frac{\partial}{\partial v'_i}\Big( \sum_{j \in N(i)} w_{ij} ||(v'_i - v'_j) - R_i(v_i - v_j)||^2 + \\
&\qquad \sum_{j \in N(i)} w_{ji} ||(v'_j - v'_i) - R_j(v_j - v_i)||^2 \Big) \\
&= \sum_{j \in N(i)} 2w_{ij}((v'_i - v'_j) - R_i(v_i - v_j)) + \\
&\qquad \sum_{j \in N(i)} -2w_{ji}((v'_j - v'_i) - R_j(v_j - v_i)) \\
&= \sum_{j \in N(i)} 4w_{ij}\big((v'_i - v'_j) - \frac{1}{2}(R_i + R_j)(v_i - v_j)\big).
\end{aligned} \tag{9}$$

The last line of that Equation 9 is possible because $w_{ij} = w_{ji}$. Setting $\frac{\partial E(M')}{\partial v'_i} = 0$ produces a sparse linear system of equations:

$$\sum_{j \in N(i)} w_{ij}(v'_i - v'_j) = \sum_{j \in N(i)} \frac{w_{ij}}{2}(R_i + R_j)(v_i - v_j). \tag{10}$$

The left-hand side of Equation 10 is $LV'$, where $L$ is the same discrete Laplace-Beltrami operator of $M$ used for LSE, and $V'$ are the new vertex positions. The right-hand side is computed for each vertex $v_i$ given $R_i$ from the previous stage and stored in a vector $b$, resulting in the equation

$$LV' = b. \tag{11}$$

To enforce ROI constraints, we must specify the indices of constrained vertices as $k \in F$. Then we erase each $k$ row and column from $L$, and set the $k$-th entries of $b$ to the desired position. $L$ is symmetric positive definite, and the original authors use a Sparse Cholesky factorization with fill-reducing reordering to perform the solve. In my own implementation, I used Eigen's SPQR solver, which uses QR factorization accelerated with the SuiteSparse library [17] [10], before giving both up and switching to the convenience and stability of LibIGL [22].

ARAP's results are considerably more stable than LSE for large handle deformations, and the minimization problems and solutions read more intuitively. Implementations like LibIGL even provide a parameter controlling Young's modulus to adjust material stiffness. The appeal of ARAP is also its drawback: reasonable surface posing without a skeleton. Achieving skeleton-like poses involves defining more handles and manual posing to fine-tune the shape. Due to their speed and ease of use, I explore using LSE and ARAP for free surface editing, but ultimately their lack of anatomical constraint motivates the construction of a spring-driven surface deformer in Chapter 5.

## 2.3 Interface

Ivan Sutherland's *Sketchpad* demonstrated the incredible potential of touch devices and sketch-based interfaces [50]. Modern hardware may expose more input parameters, such as input pressure and pen angle, but the core concepts persist and develop as more people use devices with touch support. In computer graphics and animation, sketch-based methods have matured considerably in recent years, as part of a growing interest in sketch as a flexible input modality [18, 26, 55, 38, 32, 15]. Using sketch interfaces is appealing for a number of reasons. Aside from providing a natural interface for user input, sketch systems excel in situations where coarse input and speed are favored more than precision. Kho and Garland controlled free-form mesh deformations by sketching over a region and then sketching a new position for the region [28]. More recently, implicit surfaces and sketch-based interactions have been used together to generate and edit natural-looking shapes using composition operators [2].

Other research has explored methods to automatically specify ROIs for LSE with minimal effort by the user [64]. Here, the authors detect silhouette vertices using image space filters, form polylines from distinct silhouette segments, and create a similarity mapping between polylines and strokes from user input. The silhouette vertices on the most similar polyline are translated to the stroke line to act as ROI handles and to help identify the ROI's static bounds according to the input stroke size. Finally, LSE executes to smoothly deform the ROI with respect to the silhouette handles and static bounds. This offers instinctive, natural

deformations, but it constricts surface editing to features detectable as silhouette lines. On a cube mesh, for example, only vertices on the cube's edges could be edited. This motivates the need for more direct surface editing tools that I describe in Section 7.

The line of action concept from traditional hand-drawn animation uses a single stroke to define a character's overall pose. The technique has been explored in recent research for 3D character posing [16, 39]. When used for 3D applications, inferring depth values for 2D input can be ambiguous. Avoiding this ambiguity has proven to be a challenging problem with approaches varying depending on the problem domain, from assisted methods that predict and suggest options as the user sketches [51, 55] to computational methods that rely on constraints to find a suitable answer [9]. In this research, I employ two stroke-based interfaces for different platforms. For the desktop, I offer an interface for both skeletal posing and surface deformation with the intent of being accessible for novices and sufficiently precise for experts. This provides a common set of tools and actions for both kinds of deformations and permits rapid exploration of ideas with minimal effort, encouraging artists to remain in "flow", rather than interrupt the creative process with rote keyboard shortcuts and menu navigations. In prior research, we explored the relationship between body movement and creativity support for digital tasks and found tendential evidence favoring interfaces that require more full-body movement than traditional desktop devices need [52]. In part, I acknowledge the benefits of regular movement for the body's circulation and ergonomic concerns, but flow also seems to emerge as a consequence of lowering resistance to task completion. The user spends more time doing and less time figuring out how, technically speaking. Together, good interface design and ample practice both contribute heavily to the user's ability to reach flow. With this in mind, to solve the depth problem and pursue creativity support, I also offer an interface for VR, where the user possesses a headset and hand controllers with 6DOF tracking. The implementations and details of these interfaces are featured in Sections 6 and 7.

# 3 Attachment binding

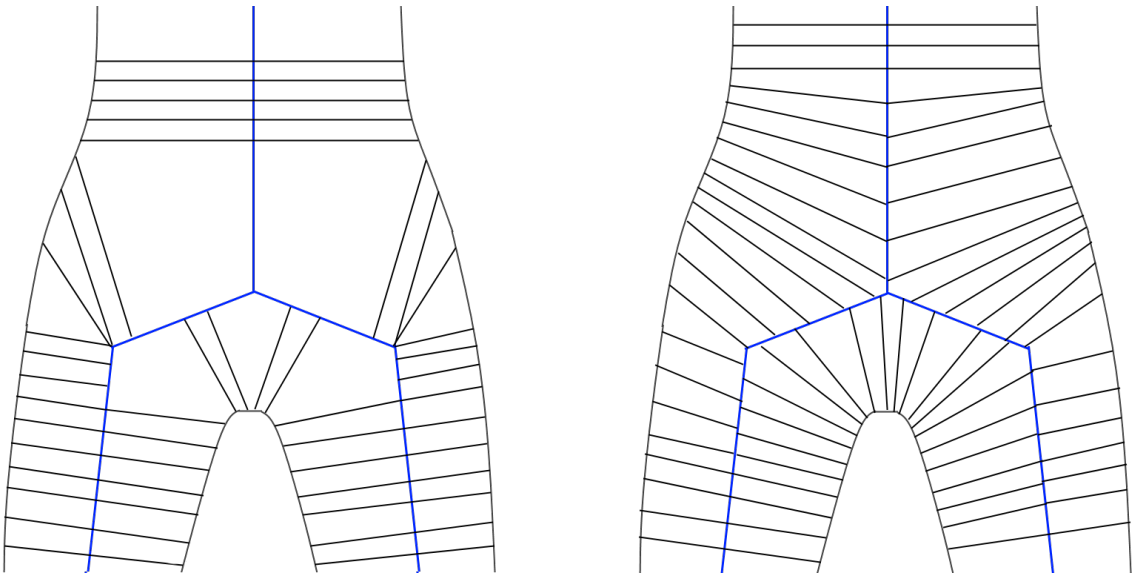


Figure 7: Orthographic cross-section of character mesh showing attachment binding on edge vertices. **A** (left): direct projection produces gaps that causes discontinuity when joints rotate. **B** (right): 6 iterations of smoothing and reprojection resolve most of the gaps seen.

With blend-based skinning, the objective of binding is to determine the influence (or weight) of each bone $b \in$ skeleton $S$ on each vertex position $v_i \in$ mesh $M$. The result is a function $W(v_i, b) \forall v_i \in M, b \in S$. There is a significant amount of research in skinning on how to define $W$ for ideal bone transform blending, subject to certain constraints. While these constraints vary across methods, a common assumption for automatic binding methods is the partition of unity, whereby $\sum_b^S W(v_i, b) = 1 \forall v_i \in M$. This is essential for blend-based skinning, because each vertex $i$ is deformed by a transform $T_i = \sum_b^S (W(v_i, b) T_b$, and if the weights for $i$ are not equal to one, the skinning transforms $T_i$ may not produce a smoothly-skinned surface.

Instead of defining weighted influences, our skinning method operates by directly connecting each vertex to a bone (Fig. 12A), resulting in an *attachment point* on the skeleton and a *scale vector* running from the attachment point to the vertex position. Formally, for a vertex $v_i \in M$, the attachment point $a_i = p_i + t_i(c - p)$ is a position along the bone between

parent and child joints $p$ and $c \in S$ parameterized by $t_i \in [0, 1]$. The scale vector $s_i = v_i - a_i$ defines the vertex position relative to its attachment point. For our purposes, the objective of binding is then to find suitable values of $t_i, p$, and $c$ for each $v_i$, since $a_i$ is computable from these values. Because of the connection to a single bone, attachment points are almost like rigid skinning, where $W(v_i, b) = 1$ for one bone per vertex.

The attachment points serve as the centers of deformation for skinning. To produce $C^0$ and $C^1$ continuous deformations, attachment points should be chosen to minimize the length of $s_i$, and vertices adjacent to $v_i$ should have similar attachment points. These objectives can sometimes be at odds; for example, the attachment point minimizing $||s_i||$ may be distant from its neighbors. The following is an overview of various methods to automatically compute satisfactory attachment points across a variety of mesh rigs.

## 3.1 Direct binding

A reasonable starting point is to compute attachment points using direct projection. For every candidate bone defined by the joints $(p, c) \in S$, we compute the following and choose the bone that yields the smallest value of $||s_i||$:

$$t_i = \max(0, \min(\frac{(v_i - p) \cdot (c - p)}{||(c - p)||}, 1)). \tag{12}$$

A direct projection test onto each bone will produce results that satisfy the first requirement, but in many cases, will also create large gaps on the skeleton between points (see Figure 9). Such cases include regions with joints that have more than 2 children and the inner side of joint bends. This scheme works best for idealized regions: collinear joints, each with a single child, surrounded by uniformly-distributed mesh vertices. Due to the large gaps occasionally seen between vertices, the direct projection approach alone does not yield attachment points sufficient for deformation (for reasons that become clear in Chapter 4).

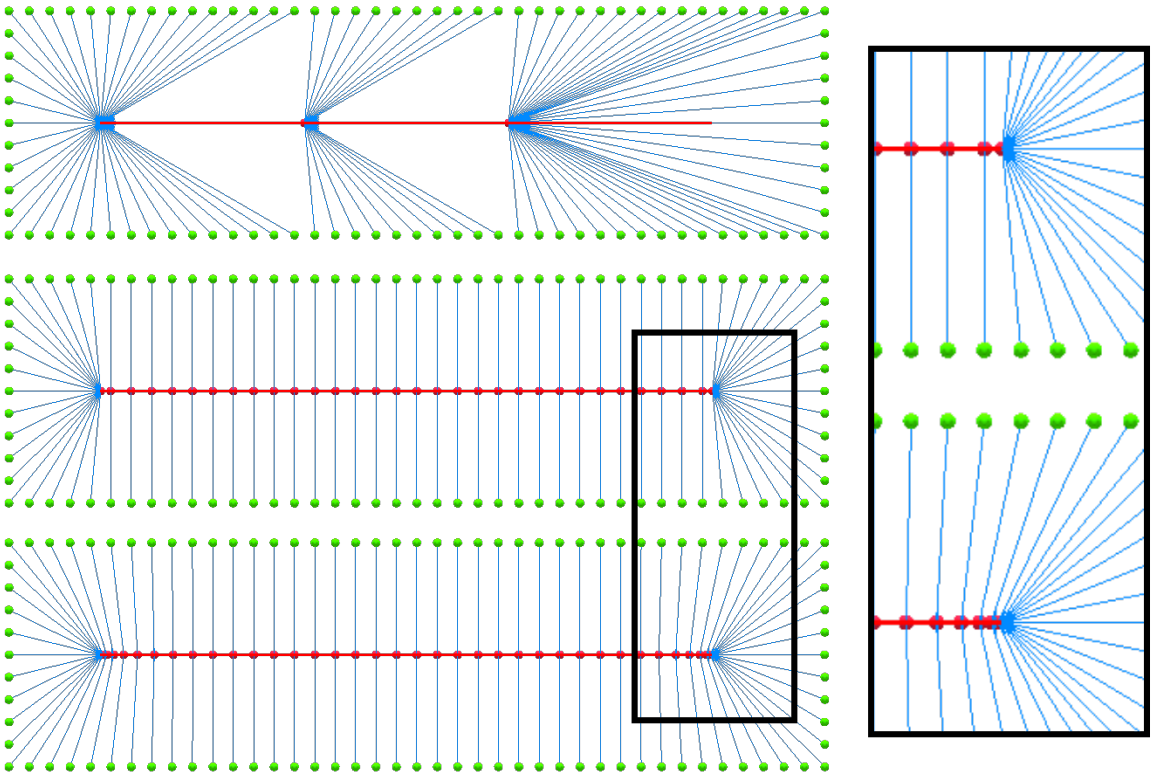


Figure 8: Three schemes for computing attachment points (red) from mesh vertices (green), producing scale vectors between them (blue). Top: closest bone. Middle: direct projection. Bottom: Alternating direct projection and smoothing. Right: close-up comparison of direct projection and alternating attachment computations.

## 3.2 Planar binding

To improve on the quality of direct projection, one approach I describe uses vertex partitioning. In this method, rather than compute bone mappings by closest projection, I insert planes at each joint to explicitly partition vertices into distinct bone mappings. Examples of this approach are shown in Figure 9. To avoid the gaps between adjacent $a_i$ values caused direct projection, $t_i$ is parameterized by the distance between $v_i$ and its two closest planes:

$$t_i = \frac{||v_i - P_p(v_i)||}{||v_i - P_p(v_i)|| + ||v_i - P_c(v_i)||}, \tag{13}$$

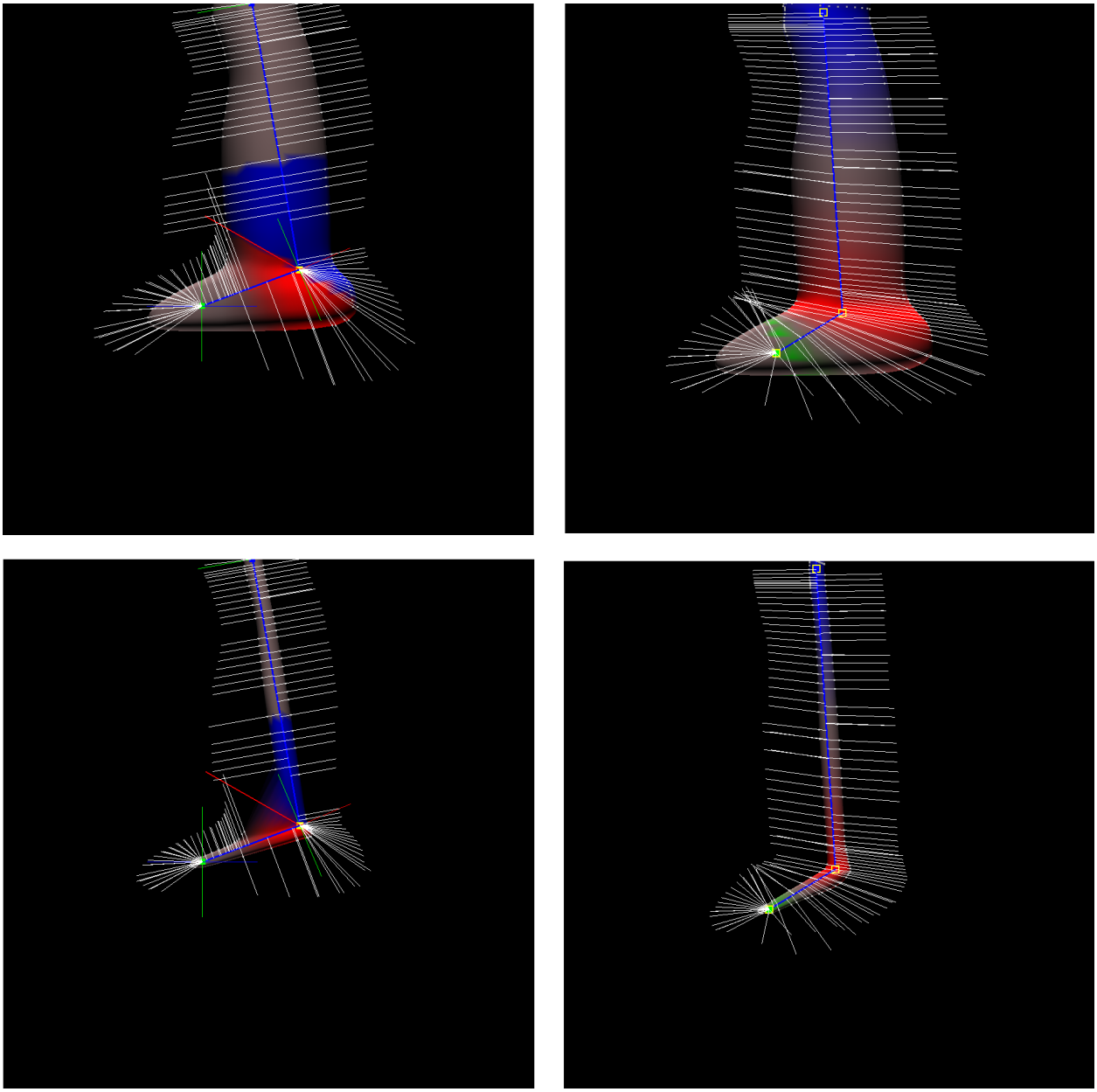


Figure 9: Simple projection-based mapping on a leg (left column) introduces discontinuities in the set of attachment points and results in stretching artifacts when scale vector lengths are altered (bottom row). Binding with joint dividers (right) and making small adjustments to the ankle's divider plane fixes the discontinuities and provides a more continuous attachment point mapping, resulting in higher quality scale vectors.

where $P_p(v_i)$ and $P_c(v_i)$ are the projections of $v_i$ onto the planes defined at $p$ and $c$, respectively. The result are a more uniform distribution of $t_i$ values across bone lengths with fewer, if any gaps. A partition plane is initialized to divide the region between each joint and its siblings, as well as its parent, see Figure 10. Each plane is initialized such that the half-vector and cross product between two adjacent bones are coplanar. However, for a joint with multiple children (e.g. the pelvis), this technique requires considerable adjustments and cleanup. Despite the automatic initialization, the drawback of this approach is the manual tuning required to better orient planes for partitioning.

As previously stated, computing attachment points is almost like binding for rigid skinning. The primary difference is that with rigid skinning, attachment points would always be coincident with the parent joint of the closest bone, while our skinning algorithm operates best when attachment points are more evenly distributed across the skeleton. Having said that, it is straightforward to use any automatic skin weight computation algorithm to initialize the computation. In other words, when the rig has skinning weights, these can be used to speed up attachment point computation. This allows the use of any modern and robust skin weight computation method, including heat diffusion [5], bounded bi-harmonic weights [19], and geodesic voxel binding [12].

For a rig with skin weights, $v_i$ is projected onto each of the bones immediately attached to the joint with the largest skin weight, and the closest projection is chosen for $a_i$. Using existing skin weights to define a binding has several key advantages. Assuming optimal skin weights, this approach is completely automatic, requires no traversal across mesh triangles or intersection tests, and makes partitioning a matter of choosing two adjacent joint indices with the largest skin weights. This only addresses the search time per vertex by culling the bones used for projection, so the gaps could still occur if the skin weights were computed with a method that does not account for volumetric distance, visibility, or skeleton hierarchy.

To address cases with attachment point gaps, I perform alternating steps of Laplacian smoothing on the solved attachment points, followed by re-projection onto the nearest bone. These steps are repeated a number of times until the projection gaps are resolved. This

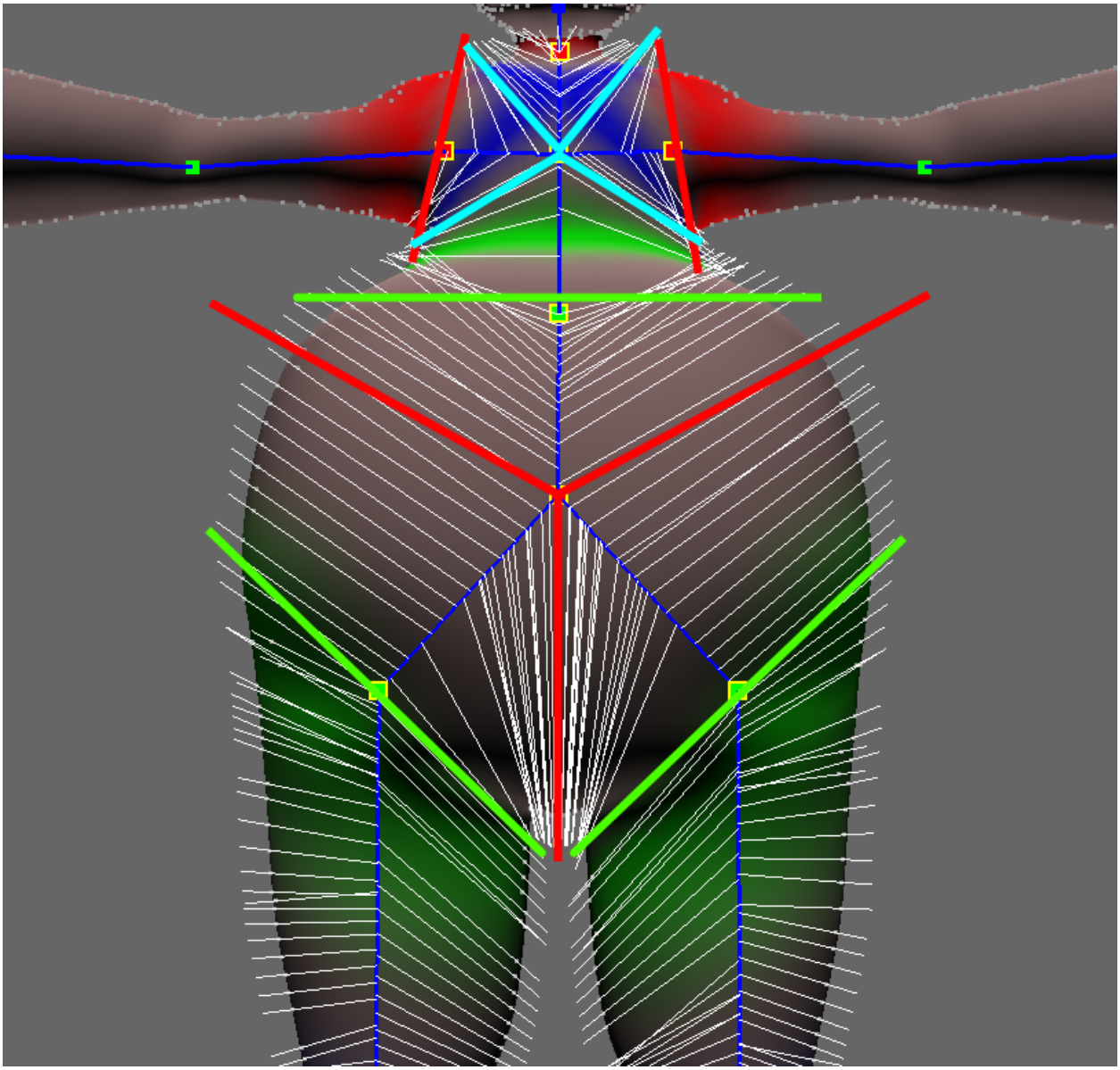


Figure 10: Explicit partitioning using planes and signed distance tests. To create a good binding for skinning, joints with multiple children automatically compute and place dividers between adjacent siblings. Here, the pelvis (red) and chest (teal) dividers split the local mesh into distinct cuts to define binding regions for each bone. Depending on the skeleton, joints may also need to place dividers between its parent and adjacent siblings.

is highly successful at closing gaps between vertices along interior bends in the mesh, but as the method does not have any constraints, the number of iterations should be kept low.

Otherwise, excessive smoothing can draw attachment points near leaf bones (joints without children) too close together on long bone chains. In practice, less than 10 repetitions is suitable for most rigs. The results of smoothing are shown in Figures 7 and 8.

## 3.3 Alternate methods

Another approach is to contract the mesh until it converges onto the skeleton [3]. This method relies on attractive and contractive forces between adjacent vertices and the skeleton to satisfy attachment requirements. In the original work, the sparse linear system includes attractive constraints to maintain distances between adjacent vertices, while the contractive constraints work to minimize mesh volume. Together, this produces a contracted mesh whose shape approximates an interior skeleton, which makes this an appealing approach to solving attachment points. The main drawback is the construction of the sparse system. In comparison, the alternating smoothing and reprojection steps provide comparable results much faster. For meshes without skeletons or skin weights, this would be a suitable technique to approximate an interior skeleton and perform binding at the same time. Because my research prioritizes compatibility with established character rig formats and features, there is less emphasis on this technique.

Related work uses skin weights for binding by clustering them accordingly to similarity. Instead of attaching to locations along bones, they compute centers of rotation for each cluster that minimize changes to their cross-section dimensions [31]. By decomposing $LBS$ to use these optimized centers of rotation, this method avoids the expected artifacts while retaining the speed of blend-based skinning. However, this approach is not immediately usable for our skinning method, as the centers of rotation may not minimize $||s_i||$ or distance between adjacent attachment points (Figure 11). For most of the vertices, the optimized centers of rotation will be located in close to the joint position with the largest skin weight, rather than evenly distributed along the bone length. Nevertheless, the observation that skinning results can be improved by changing the center of rotation is apparent.

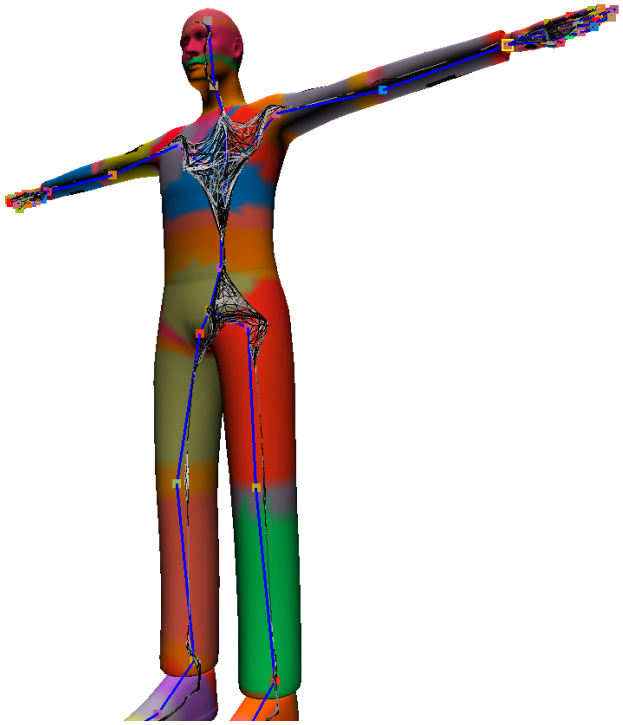


Figure 11: Optimized centers of rotation computed for decomposed LBS. Mesh is color-coded by skin weight clustering. For limbs, the centers lie chiefly on skeleton bones, while in joints with multiple children, the optimal centers are more mesh-like.

# 4 Skinning

The goal of this skinning algorithm is to minimize differences in angles and length between adjacent scale vectors. Because the scale vectors run between attachment points on the skeleton and mesh vertices, this objective is critical for unifying skeleton and surface deformations. While there are additional objectives to include for the complete solution, this is a good introduction to the algorithm's general function, and it demonstrates the limitations that motivated additional objectives to complete the solution.

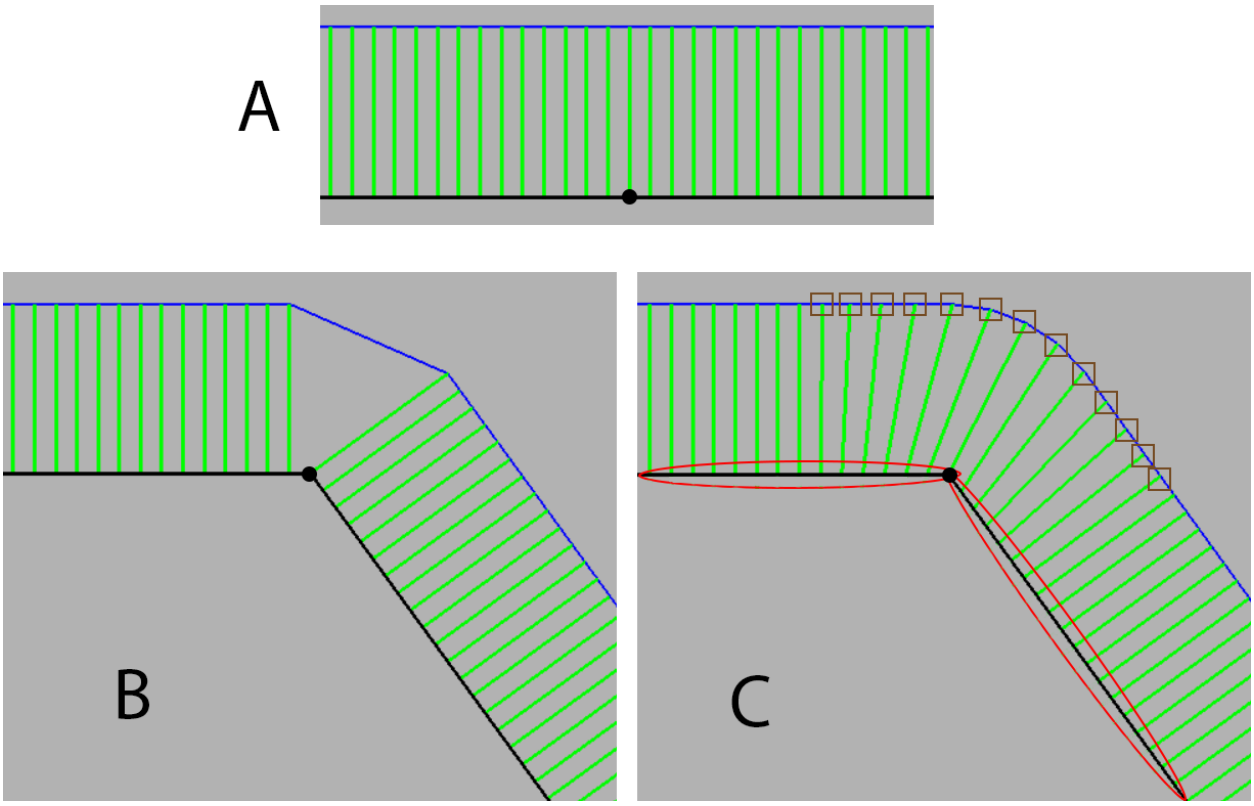



Figure 12: Cross-section of the top half of a cylinder. Black lines are bones, blue the surface of the mesh, green the scale direction vector $s_i$ and the black circle is a joint. A) shows the bind pose, B) shows a rigid deformation, C) shows an incremental rotation skinning result. The attachment points, circled in red, are the same as for the rigid deformation, but the scale vectors marked with squares on their ends are incrementally rotated to create a continuous mesh deformation.

## 4.1 Incremental rotation skinning (IRS)

The initial form of the skinning algorithm was a geometric solution; each vertex is skinned to minimize adjacent angles in one execution. This form is called incremental rotation skinning (IRS), and it was presented in poster form as "attachment-based character deformation" (ABCD) [54]. IRS consists of two stages, as illustrated in Figure 12. In the first stage, recomputing attachment points and applying the joint transformation to all scale vectors attached to a bone causes the mesh to track bone movement and appropriately deform with

changing bone lengths, but rigid deformations still create discontinuities around the bend. Rather than dealing with this by blending transformations, as is done in other skinning techniques, in the second stage, we rotate the scale vectors towards the joint (or away on the inside of the bend) proportional to their proximity in order to create a smooth deformation. Vertices closer to a joint receive more of its rotation. The length and direction of the scale vectors can be varied additionally in order to control the mesh shape around the bend or provide custom surface deformation.

Formally, for a vertex $v_i$, the attachment point $a_i = p + t_i(c - p)$ is a position along the bone between parent and child joints $p$ and $c$ parameterized by $t_i \in [0, 1]$. The scale vector $s_i = v_i - a_i$ defines the vertex relative to its attachment point. Computing the skinned attachment point as $a'_i = p' + t_i(c' - p')$ incorporates changes in bone position and scale. With this, the rigid skinned vertex position is found for the first stage as: $v'_i = a'_i + R(p)(s_i) = a'_i + s'_i$, where $R(p)$ is the parent joint's rotation transform composed of angle $p_\theta$ and axis $p_{\vec{x}}$.

In the second stage, vertices near joints are subject to additional rotation around their attachment points to smooth away discontinuities from rigid skinning. The parent and child joints of a bone, $p$ and $c$, have user-set influence values $p_i, c_i \in [0, 1]$ (both default 0.5) that denote the percent of the bone length along which the vectors $s_i$ are increasingly rotated. Due to the support range, it is possible for a vertex to receive partial influence from both the parent and child joints on a bone. Because $p_i, c_i$, and $t_i$ are parameterized by bone length, computing the incremental rotation weight is straightforward:

$$p_w = \max(0, \min(1 - \frac{t_i}{p_i}, 1)), c_w = \max(0, \min(\frac{t_i - c_i}{1 - c_i}, 1)), \tag{14}$$

After finding the weights, computing the incremental rotation is also straightforward. Intuitively, vertices close to joints experience at most half of the joint's rotation. If a joint is bent at 180 degrees, the vertices around the joint rotate up to 90 degrees around on either side of the bend. As long as nearby vertices on the joint's opposite side receive half of its rotation, but in the opposite direction, the skinning appears smooth after the incremental rotation stage. Using an angle-axis rotation matrix constructor $\text{Rot}(\theta, \vec{x})$, for each joint, the

potential incremental rotation matrices $\Delta_p, \Delta_c$ are formed as:

$$\Delta_p = \text{Rot}(-\frac{p_w p_\theta}{2}, p_{\vec{x}}) \tag{15}$$

$$\Delta_c = \text{Rot}(\frac{c_w c_\theta}{2}, c_{\vec{x}}) \tag{16}$$

When both weights are zero, no incremental rotation occurs. In general, vertices near the child or parent will be affected by only that joint, while vertices in the middle of the bone may be rigidly bound. Users may decide to overlap the influence regions extending out from the parent and child joints, in which a blend will occur. These cases are covered in the computation of a final incremental rotation matrix $\Delta$:

$$\Delta = \begin{cases} I & \text{if } p_w = c_w = 0 \\ \Delta_p & \text{if } p_w > 0, c_w = 0 \\ \Delta_c & \text{if } c_w > 0, p_w = 0 \\ \Delta_p \frac{p_w}{p_w + c_w} + \Delta_c \frac{c_w}{p_w + c_w} & \text{if } p_w > 0, c_w > 0 \end{cases} \tag{17}$$

After $\Delta$ is known, the result of this stage is:

$$v_i'' = a_i' + \Delta s_i' = a_i' + s_i'' \tag{18}$$

Although skinning is completed at this point, it may be prudent to alter the length of $s_i''$ based on the final position of $v_i''$. Without length adjustment, increasing a joint's rotation will cause the mesh to develop a sharp point (see Figure 13). To avoid this, the length for $s_i''$ can be found by re-projecting $v_i''$ onto the skeleton to find $v_s$ and comparing $||v_i'' - v_s||$ against $||s_i||$. Then $v_i''' = v_p + (\frac{||s_i||}{||v_i'' - v_s||}(v_i'' - v_p))$ and $s_i''' = v_i''' - a_i'$. This computation helps maintain each vertex's distance from the skeleton at bind.

The decomposition of skinning into rigid and localized stages is a critical step towards enabling anatomical and surface-level deformations in the same computation space. This representation introduces opportunities to modify configure how the mesh deforms around

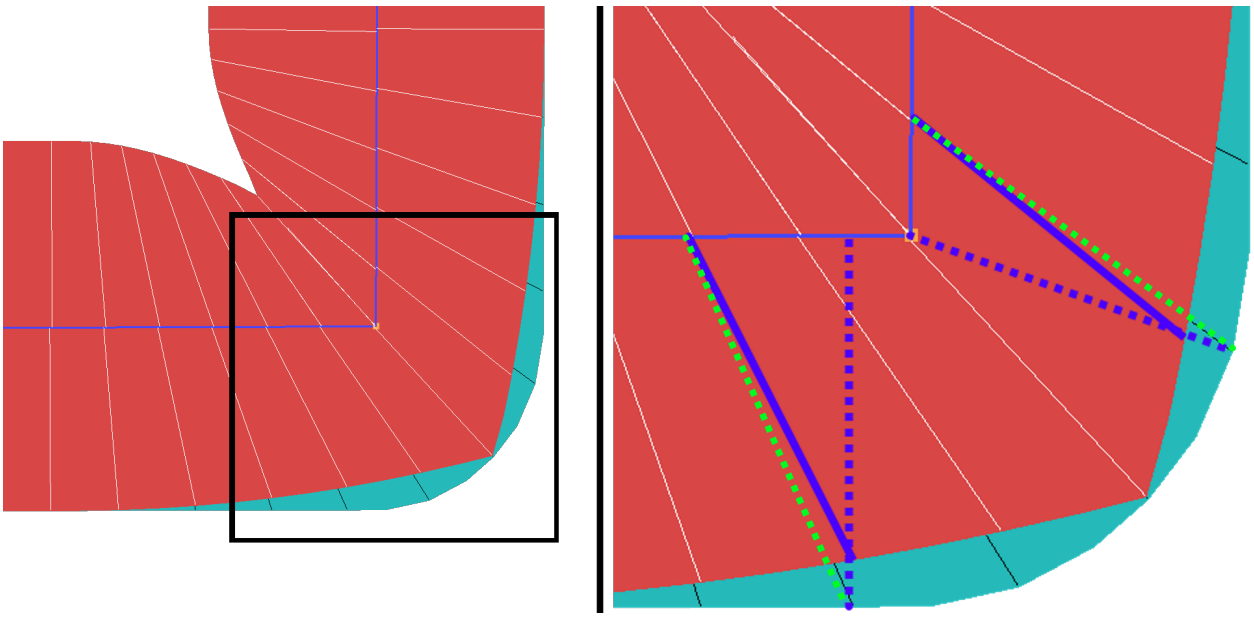


Figure 13: An artifact of IRS (red) that occurs when scale vectors lengths are left unchanged. As an optional step, IRS can project skinned vertex positions back onto the skeleton and determine a new position that preserves the vertex's distance from the skeleton at bind (teal).

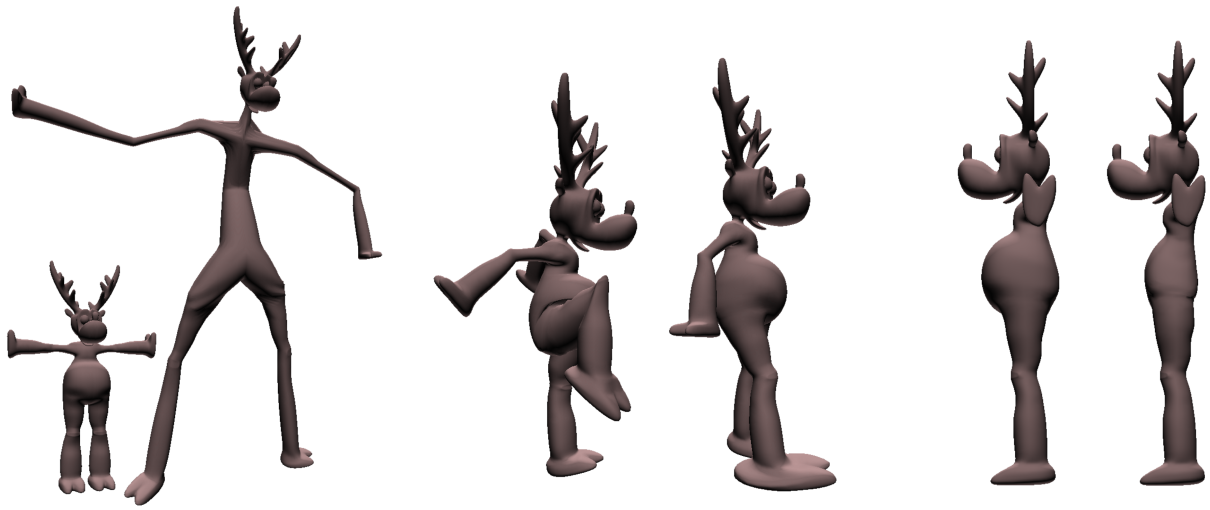


Figure 14: A series of deformations made with IRS: changes in bone length (left), squash effects (middle), and scale vector length adjustment (right)

individual joints. The length of $s'_i$ can be adjusted to make smooth rotations around joints without the volume loss of LBS or spherical bulge of DQS, and surface editing techniques can even change the mesh shape before skinning without requiring rebinding or skin weight

recomputation. Figure 14 contains examples of deformations combining anatomical and surface-level changes.

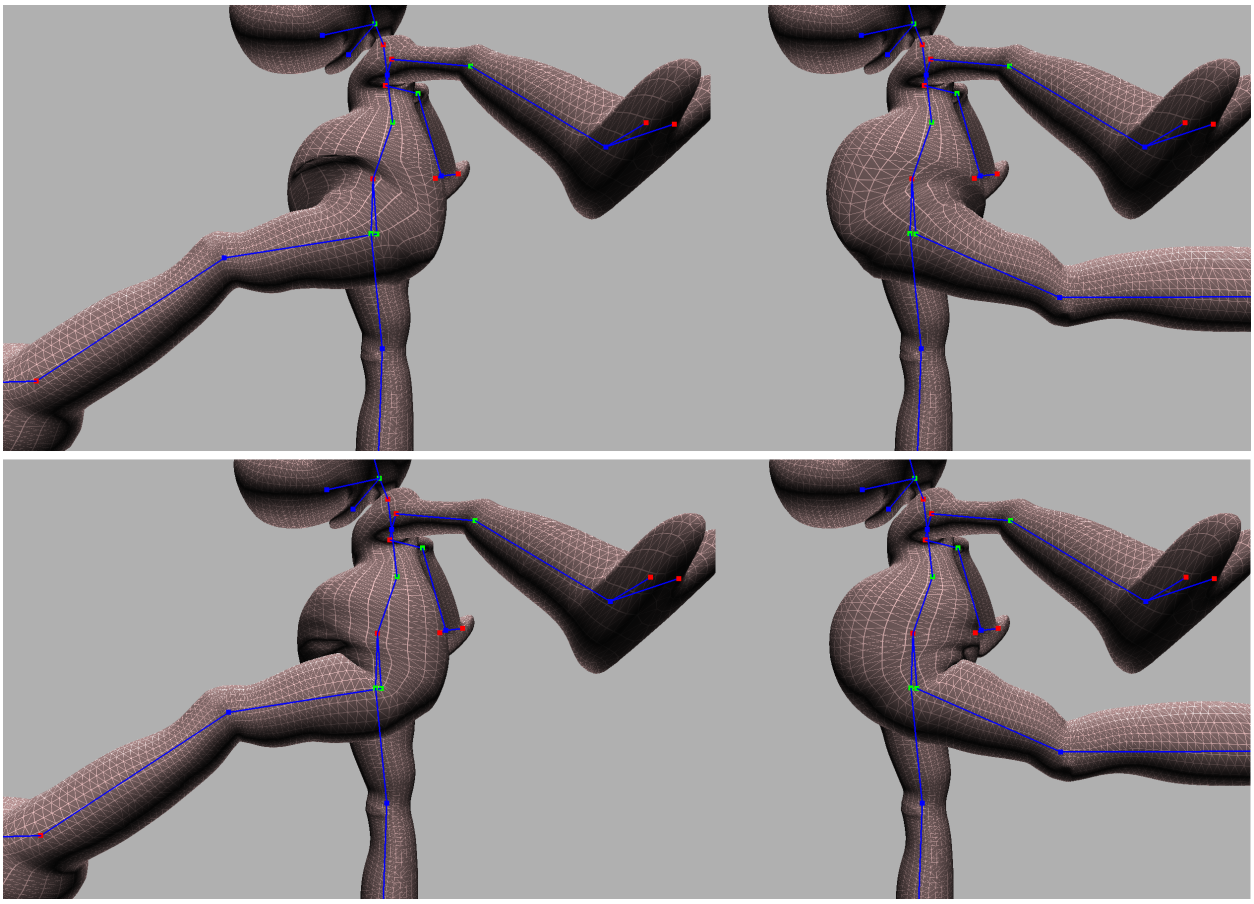


Figure 15: Joint divider binding and subsequent skinning computation result in more dramatic discontinuities compared with LBS (bottom)

Compared with blend-based skinning methods, execution speed is on the same order of magnitude on graphics hardware, but implementation necessitates more complex shader programs with numerous conditional statements to account for every case and possible joint configuration. Although this method does not rely on normal skin weights for skinning, there is a possibility of blending two joint transforms to deform a vertex. For mesh regions with multiple bones, this is not always sufficient for producing smooth skinning, as seen in Figure 15. The features and limitations of IRS served as motivation for a more robust skinning method that generalizes the concept of incremental rotations to a wider variety of

mesh and character rigs.

### 4.1.1 As-rigid-as-possible surface modeling with scale vectors

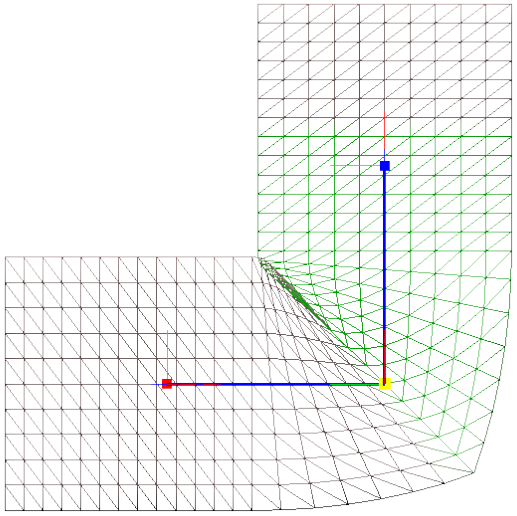


Figure 16: A 90° joint bend achievable using IRS and ARAP with scale vectors.

The as-rigid-as-possible (ARAP) surface deformation technique is known to produce consistently more stable results than LSE with comparable requirements [47]. Since it works by minimizing differences between adjacent vertices (see Section 2.2.3), I was motivated to compare it with IRS. By replacing the unknown positions with scale vectors, Equation 10 becomes

$$\sum_{j \in N_i} w_{ij}(s'_i - s'_j) = \sum_{j \in N_i} \frac{w_{ij}}{2}(R_i + R_j)(s_i - s_j), \tag{19}$$

creating a sparse linear system for solving scale vectors deformed in an as-rigid-as-possible manner:

$$LS' = b. \tag{20}$$

For this approach, the rotation matrices $R$ in the terms are assumed to be the rotation

transform of the bone to which each vertex is attached. Essentially, rigid skinning is provided as an initial guess, then the optimization improves on the fit from there. As a simple demonstration, the bone parameters in IRS can be tuned to produce deformations visually identical to ARAP, see Figure 16. In practice, using ARAP on scale vectors does not address length correction around joint bends, and it is too slow compared to IRS for real-time use. However, the similarity between results does support the viability of IRS as a real-time deformation method.

## 4.2 Spring deformers

To overcome the limitations in IRS, I proposed a method that iteratively deforms the mesh until it finds an equilibrium point between forces from the surface and skeleton with optional guidance from the artist. The forces that drive mesh deformation are spring-derived, so we refer to this family of techniques as *spring-based skinning*. This research appeared in the proceedings of *Motion, Interaction, and Games 2019* under the title "Spring Rigs for Skinning" [53]. Spring-based skinning deformers construct and simulate a number of springs between the vertices and skeleton to produce shapes on par with smooth skinning. These fall into two categories: surface-based, linear springs between adjacent vertices positions, and attachment-based, torsional springs that adjust the angle of a scale vector with its neighbors and attachment bone. Since these forces often oppose one another, the deformer iterates until the net force on each vertex converges to zero. Using forces derived from different measures together enables a range of surface deformation controls while adhering to anatomical constraints.

In the first stage, a geometric skinning algorithm deforms the mesh according to the skeleton pose. Rigid skinning, LBS/DQS, STB, CoR, and more methods are all supported. It is important to note that despite the use of an optimization method for the second stage, it is not guaranteed that the spring system will completely fix any artifacts specific to the skinning method used for the first stage. For example, a twist using LBS may collapse a vertex onto the skeleton, i.e. $v_i = a_i$, and because $v_i = a_i + s_i$, $|s_i| = 0$. Thus, I introduce

this method using rigid skinning since it is simple, avoids producing hard-to-fix artifacts, and provides anatomical deformation with no further assumptions about the final shape. The attachment binding between the mesh and skeleton provides all of the information needed to perform rigid skinning with support for bone length changes, as introduced in Section 4.1, without requiring skin weights for the actual deformation. As the name suggests, rigid skinning will introduce discontinuities around joints (see Figure 17, left). These are resolved in the second stage through the use of spring forces to minimize differences between vertices and eliminate discontinuities, resulting in a smooth deformation (Fig. 17, right). Using spring forces and iterations instead of explicit transforms per-vertex allows us to smoothly deform mesh regions with multiple influencing bones nearby without resorting to convoluted, conditional deformation rules used for IRS.

Spring forces are computed according to Hooke's law, $F = kx$, where $x$ is the displacement from rest and k is a stiffness coefficient. In this case, the springs used to relax the mesh act between neighboring vertices, so the net force computed for $v_i$ is a sum of forces between vertex $i$ and its one-ring neighbors $j \in N(i)$. Thus, the displacement $x$ is the difference between a measure for the current values of $v_i$ and $v_j$ and the bind values of $v_i$ and $v_j$. Generally, each of the forces used in this solver take the form:

$$F(i) = \sum_{j \in N(i)} k(i,j)(\Delta(i,j)_{\text{current}} - \Delta(i,j)_{\text{bind}}) \tag{21}$$

,

where $\Delta(i,j)$ is a relative difference and $k(i,j)$ is a stiffness coefficient between vertices $i$ and $j$. The solver uses four spring forces, each of which utilizes a different measure. Linear springs are used to maintain surface relationships through force $F_s$ and define $\Delta(i,j)$ as the edge length between $v_i$ and $v_j$. These forces help to resolve overstretch and compression between adjacent vertices. Offsets from the bone are maintained with force $F_l$, using linear springs that define $\Delta(i,j)$ as the difference in length between $s_i$ and $s_j$. Bone springs $F_b$ works to maintain the bind angle between $s_i$ and the bone run $c-p$. Without this constraint, the torsion springs may skew the entire mesh until it loses its anatomical shape from the

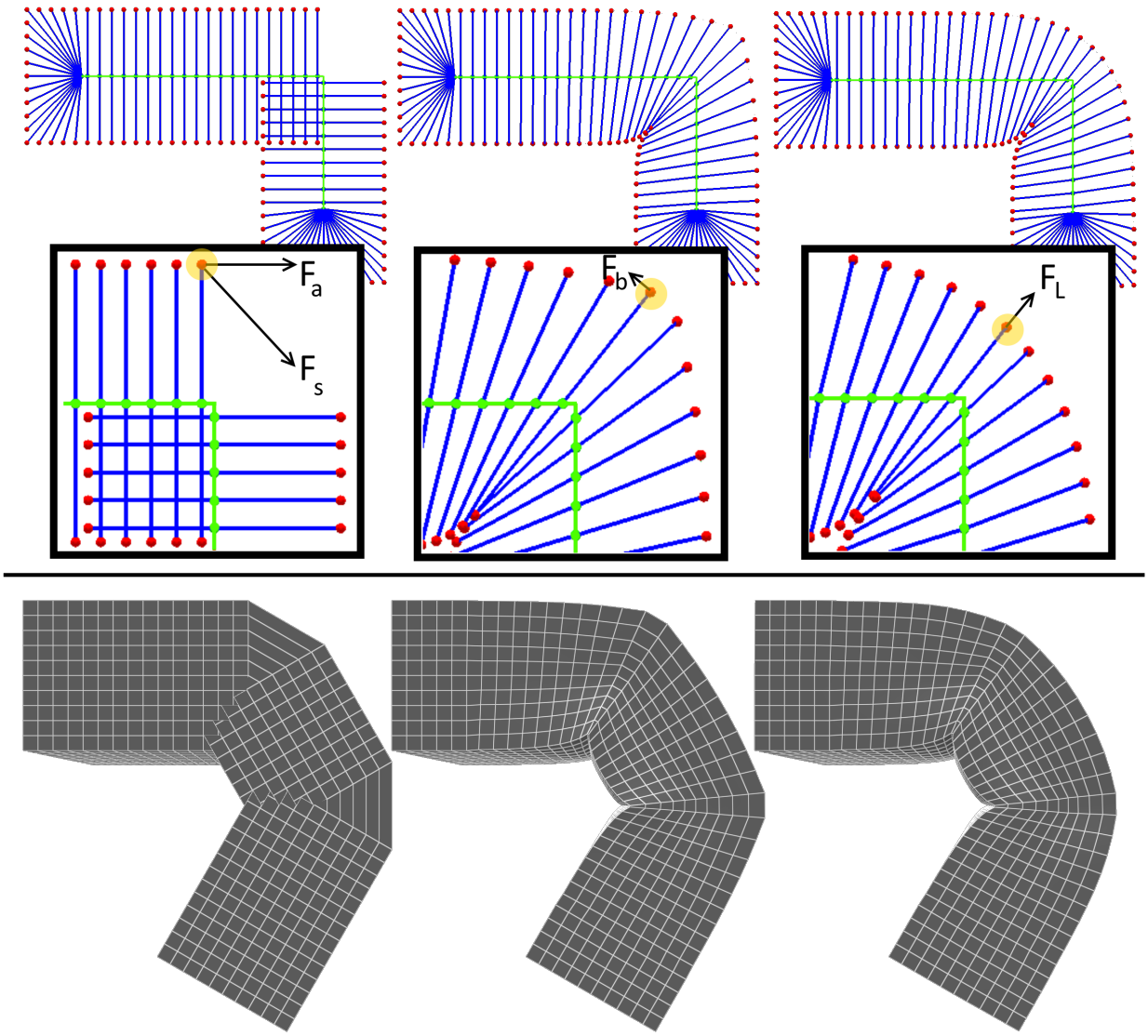



Figure 17: Top: Applying spring forces to 2D cross-section. Surface vertices are red, bones and attachment points are green, and scale vectors are blue. Spring forces are illustrated for the vertex highlighted in yellow. Left column: bone torque $F_b$ has no effect on rigidly-skinned meshes, but attachment torque $F_a$ and surface edge $F_s$ are computed to close the gap between a vertex and its neighbors. Middle: $F_b, F_a$, and $F_s$ resolve gaps, but leave a pointed outer bend. Right: increasing $F_s$ and decreasing $F_l$ help the vertex reach equilibrium without collapsing too close to its attachment point on the skeleton.

skeleton. Finally, torsion springs produce force $F_t$ and define $\Delta(i, j)$ as the angle between $s_i$ and $s_j$. This force works to close gaps around outer joint bends by rotating adjacent vertices closer together relative to their attachment points. In addition, $F_t$ is essential for resolving self-collisions on the inside of joint bends, which the other forces cannot address, see Figure 18. The net force for vertex $i$ is

$$F_{\text{net}}(i) = F_t(i) + F_s(i) + F_l(i) + F_b(i). \tag{22}$$

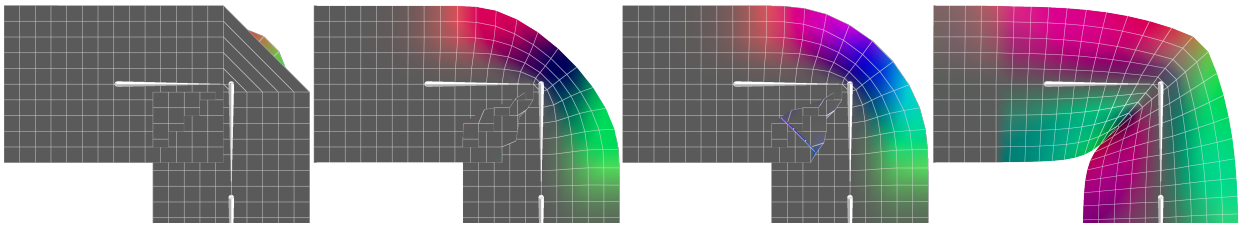


Figure 18: Torsion springs resolve artifacts on the inner and outer side of a joint bend. From left to right in accumulation: rigid skinning, surface springs, length springs, and torsion springs. Vertex color and brightness represent net force direction and magnitude.

To compute Equation 22, it is necessary to convert $F_t$ and $F_b$ from a torsion force to a linear force. This is realized by computing a linear force magnitude $m = ||F_t||/||s_i||$, then applying it in the direction orthogonal to $s_i$ towards the neighboring $j$, or in the opposite direction of the angular displacement in the case of $F_b$. On each solver iteration, the vertex position is updated as

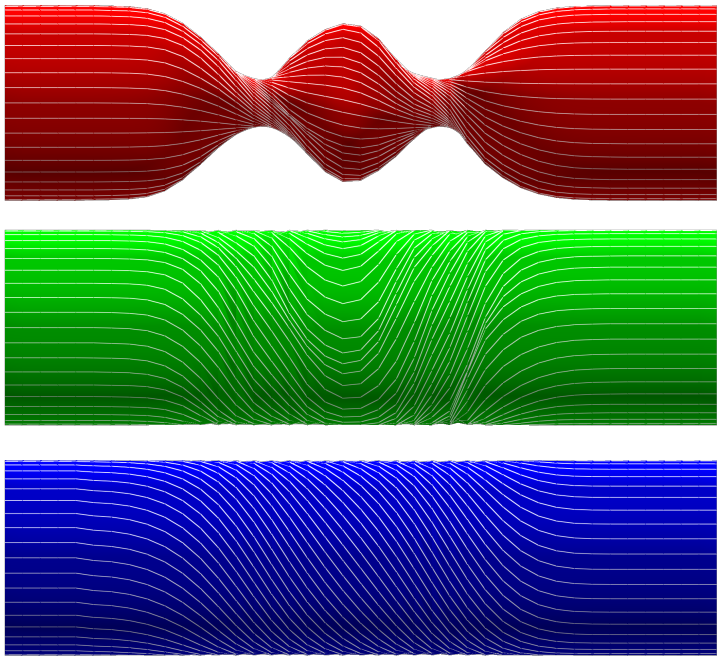
$$v_i'' = v_i' + dt \times F_{\text{net}}(i), \tag{23}$$

where $dt$ is a time step parameter. Equation 23 behaves in the same manner as used by Wilhelms and Van Gelder [62], in that the force value computed is not used for full physical simulation, but instead to help vertex positions converge to a rest state. In implementation, however, it is possible to use semi-implicit Euler integration with the forces instead, computing changes to acceleration, velocity, and position on each frame, and damping velocity

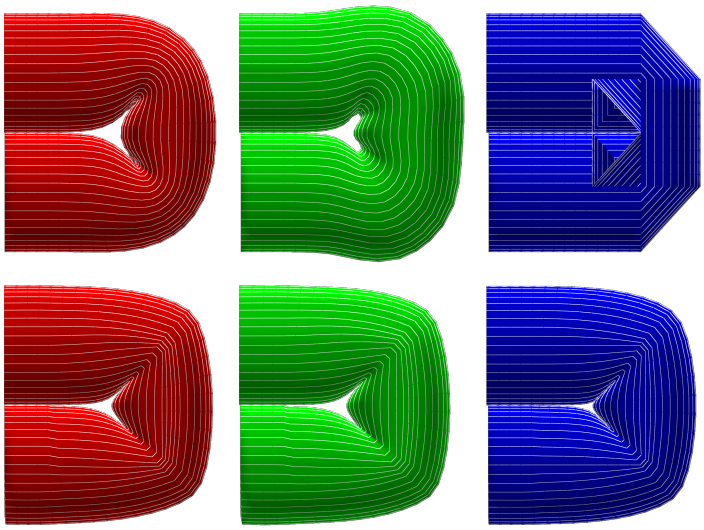
to help bring the mesh to rest. The solver can generate secondary effects such as wave propagation a conservation of energy along the surface. These behaviors may be desirable for interactive surface editing at a later point, but strictly concerning skinning, the approach in Equation 23 is still preferred for stability and memory requirements.

### 4.2.1 Features

Spring deformers handle twist accumulation along joints with smooth rotation distribution and without collapse (Figure 19a). For swing rotations, the spring deformers are rather consistent regardless of whether the initial pose is found using rigid or blend-based skinning (Figure 19b). This is encouraging to see, because it indicates that there is a consistent point of equilibrium for a given skeleton pose, and approaching it is not especially sensitive to the initial solution. As mentioned before, it is also possible to use a more advanced blend-based method, such as STB ([23]) or CoR ([31]). Since these methods handle twist rotations more gracefully, these choices may cut down on the iterations required for the spring deformers to converge. They could also provide a solution that the deformer observes to be already in equilibrium, thus losing the shape control provided by using spring deformers in the first place. For character rigs, the spring deformers can emphasize features on the mesh and skeleton that are not present with blend-based skinning, see Figure 20. With that in mind, using rigid skinning to initialize the spring deformer is reliable, inexpensive, and independent on skin weight quality.

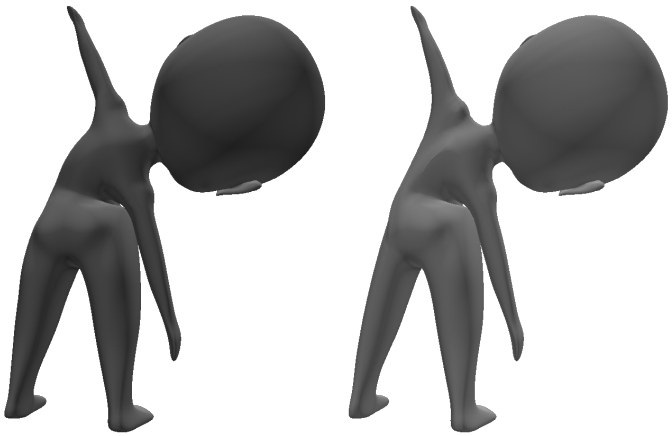


(a) 135° twist rotations on 2 bones in a cylinder with LBS (red), DQS (green) and springs (blue). Where LBS begins to collapse on large twists, DQS better preserves vertex-skeleton distance, but can change twist direction as the twist increases. Spring forces avoid collapse and diffuse the twist across the mesh length.

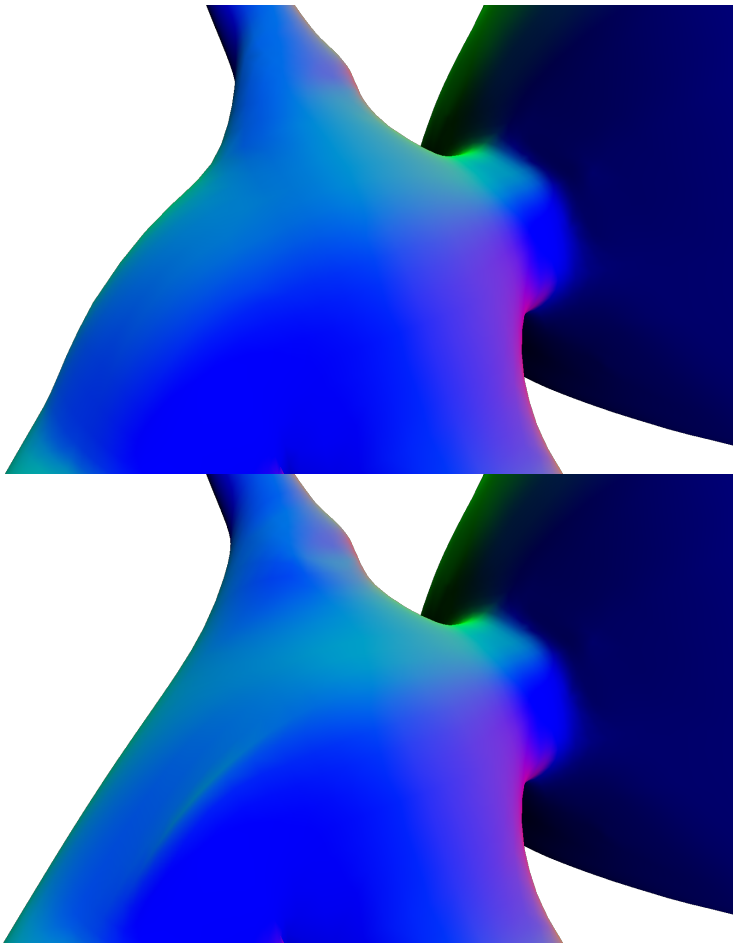


(b) 90° swing rotations on 2 bones in a cylinder. The top row shows LBS (red), DQS (green), and rigid skinning (blue). The bottom row shows the results of applying spring forces on the skinned mesh above it. The spring force results are relatively consistent regardless of initial skinning choice.

Figure 19: Comparing skinning methods for joint rotations.



(a) Chibi rig in a stretching pose. LBS on the left looks as expected, but the spring deformers on the right highlight the shoulder tissue and reveal a fading crease along the spine.



(b) A closer look at the surface normals for the stretching pose. Top: LBS. Bottom: spring deformers.

Figure 20: Comparing skinning methods for joint rotations.

### 4.2.2 Constraints

For LSE, ARAP, and spring-based skinning, explicitly marking and posing a set of vertices as boundary constraints allows the artist to reshape the mesh with minimal effort, similar to how skeletal skinning reduces control over a large set of mesh vertices to a small set of joints. In addition, without boundary constraints or specifications for convergence, these methods may continue to deform beyond the desired state. For surface editing, these boundary constraints are often provided as regions of interest (ROI), which specify a set of connected vertices with some marked as kinematic (position is set by the user), others marked as static (position is unchanged, representing the boundary of the surface edit), and the in-between vertices marked as passive (position is updated by the solver). By fixing or severely limiting the deformation of kinematic vertices, the spring system applies forces to its neighbors that, over repetitions, cause the ROI to deform locally. For LSE and ARAP, if the ROI includes the entire mesh (i.e., all vertices are either kinematic or passive), it is likely that the deformation will be degenerative. For example, if only one vertex is kinematic and the rest are passive, transforming the kinematic vertex will essentially result in a rigid transformation of the entire mesh.

To some degree, the system's timestep and convergence epsilon variables can influence the solver's behavior and serve as behavioral constraints. The variations in Figure 21 are found by varying spring stiffness and iteration counts. We find it most appropriate to keep the timestep fixed at $\frac{1}{30}$ seconds. Increasing this may cause larger discontinuities to resolve in fewer iterations, but it can also introduce oscillations before convergence. Our system's default configuration is set to repeat execution until a maximum count of 50 has been reached, or when the largest vertex position change between solves is below a threshold initially set to $AABB_{min} \times 1e{-}4$, where $AABB_{min}$ is the smallest dimension of the model's axis-aligned bounding box. The solver's flexibility permits interactive tuning, and in practice, the timestep and threshold variables may require adjusting at initialization, but then may be left intact.

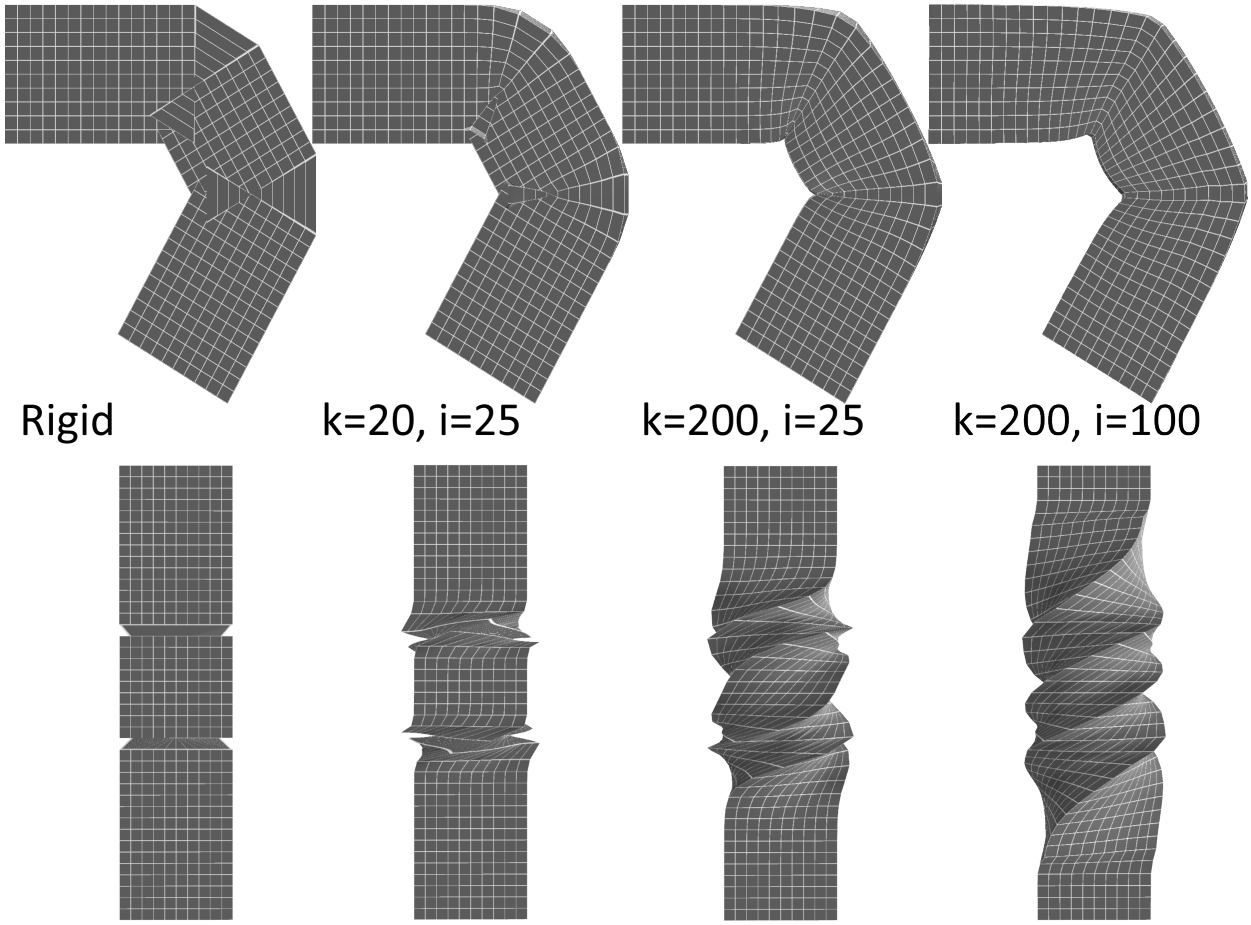



Figure 21: Results of swing (top) and twist (bottom) deformations using various spring coefficients $k$ and iteration counts $i$.

### 4.2.3 Execution

Spring-based skinning's execution model is iterative, but it differs somewhat from the optimization approaches in LSE and ARAP. With LSE, a sparse linear system is constructed using boundary constraints and solved for to find the deformed vertex positions. The objective in LSE is to minimize changes to the differential coordinates, which are invariant under translation, but not rotation, revealing why LSE exhibits artifacts under large orientation changes. The system must be rebuilt and solved whenever the boundary constraints change. To improve solve times, the deformed vertex positions of a prior LSE operation can be used

when building the next solver as an initial guess. ARAP also solves a sparse linear system for vertex positions, but also alternates and solves for cell rotations that maximize rigidity using singular value decomposition. Similarly, ARAP's solvers must be rebuilt whenever the boundary constraints change. The key difference is that ARAP runs as alternating iterations until a maximum iteration count is reached. In comparison with these methods, spring deformers iterate a single solver until convergence is reached. While the configurations provided in Section 4.2.2 usually provide results of ample quality, the deformer supports a set of convergence schemes, including maximum iteration count, thresholds for force magnitude or position change, or halting on local minima for total spring energy. For the technical artist, it may be desirable to have the choice of scheme for different character rigs. Although there is no sparse linear system solver, there is an initialization step that involves recording edge lengths and scale vector angles between adjacent vertices while the mesh is in bind pose. These lengths and angles represent the mesh's rest state, and the spring system refers to these bind values to help determine force magnitudes in each iteration.

# 5 Surface deformation

The process of skinning essentially determines how a mesh surface should move according to bone movement. As the skeleton is constructed to be a reasonable representation of a body's natural range of motion, the emphasis is on anatomical conformity. More generally, I consider skinning a member in the larger family of *surface deformation* methods, which, for the focus of this thesis, includes every technique for changing a mesh from one configuration to another without altering its topology or geometry count. This is an important distinction: high-quality character meshes are the result of tremendous artistic and technical effort, so there is inherent motivation to prevent surface deformers from adding or removing vertices, edges, or faces. In addition, changing the geometry count in real-time can become costly, and strategies for efficiently resolving changes to the geometry sets (such as filling holes, updating indices in memory, and adaptive texture mapping) are outside the scope of this work. Focusing on techniques for deforming existing geometry make possible a clear comparison between anatomical and free surface deformations, and a more optimal approach for resolving their differences in the same animation framework.

Skinning and other surface deformers are rather similar in behavior and prerequisites. Both rely on a small set of control handles, which are comparatively easy to handle and animate, to deform a larger set of geometry, which is too time-consuming for the artist to edit or computationally impractical to directly animate. Skeleton joints constitute these handles for skinning, while for a lattice-based surface deformer, its grid points serve as handles. Both require a function that defines the handles' influences on surface vertices: skin weights for skinning, and distance-based weights for the lattice. In theory, this function is defined for every handle-vertex pairing, but in practice, many vertices will receive 0 influence from a handle, and the influence functions may be discretized, culled, and normalized to the largest $n$ handles, memory space and computation time permitting. Finally, both tend to operate by applying the affine transformations between the handles' bind and current configurations to vertices based on their influence weights.

A crucial difference between traditional skinning and more general surface deformers is the

consideration of how vertices influence one another. In the classic LBS formula (Equation 1), the degree of inter-vertex influence depends solely on the method chosen for skin weight computation. A Euclidean distance-based weight computation does not account for vertex connectivity, but nearby vertices will implicitly have similar skin weights and transforms. Computing weights using heat maps explicitly uses mesh edge information to model diffusion and provide higher-quality weights [4]. In either case, topology attributes such as vertex adjacency and edge flow are kept out of blend-based skinning for a couple reasons. First, the performance demands are simple to estimate. LBS is essentially of complexity $O(|V| \times |M|)$, bound by the number of vertices and control handles, although a maximum of two or four skin weights per vertex makes this closer to $O(|V|)$. Next, it supports parallel implementations since each vertex's information can be accessed and computed independently. On graphics hardware with hundreds of cores, LBS and its derivations are closer to $O(1)$. GPUs excel at this kind of work, and consequently, introducing any degree of communication between vertices in the skinning process is non-trivial, tedious, and likely to impact performance. Vertex adjacency tends to be sparse and inconsistent across the mesh, making its storage and retrieval on graphics memory less obvious, requiring flattening and indexing. More importantly, vertex shaders typically do not permit sharing data between invocations and iterations. This prohibits deformers from accessing neighboring vertex data, or taking more than one execution to complete. For these reasons, the initial exploration in real-time surface deformation aimed to operate within the constraints for GPU skinning.

## 5.1 Radial deformations

This approach was based on the IRS formulation introduced in Section 4.1. As a direct skinning method, any kind of surface deformation must be completed in one pass for all vertices. The decomposition of mesh vertices into attachment points and scale vectors explicitly defines local frames of reference for deformation. This revealed an opportunity for making surface edits that behave consistently with skinning. Like IRS, radial deformations operate by recalculating the vertex position $v_i$ relative to its attachment point $a_i$ on a nearby bone.

Given the scale vector $s_i = v_i - a_i$ and the relationship between these values, the method worked by changing $s_i$ to produce a new value for $v_i$. Because vertex-bone attachment is explicit, a scale vector can be decomposed into its magnitude and the angle $\theta$ it forms with its attachment bone $B$. Substituting these in for $s_i$ yields

$$v_i = a_i + s_i = a_i + [(R(B) \times R(\theta) \times X(B)) \times |s_i|], \tag{24}$$

where $X(B)$ is the normalized direction of $B$ at bind, $R(\theta)$ is the rotation that aligns $X(B)$ with $s_i$, and $R(B)$ is the rotation transform for $B$. This exposes two parameters for applying surface deformations: $s_i$'s magnitude and angle with $B$. For demonstration, a surface effect in the vertex shader could multiply $|s_i|$ by a value derived from $v_i$'s parameterization along $B$, $t \in [0, 1]$. Multiplying $s_i$ by the result of $-(0.5 - t)^2 + 1.25$ produces a parabolic bulge along the length of the bone that adds volume and terminates evenly at both ends. Other functions can be seen in Figure 22.

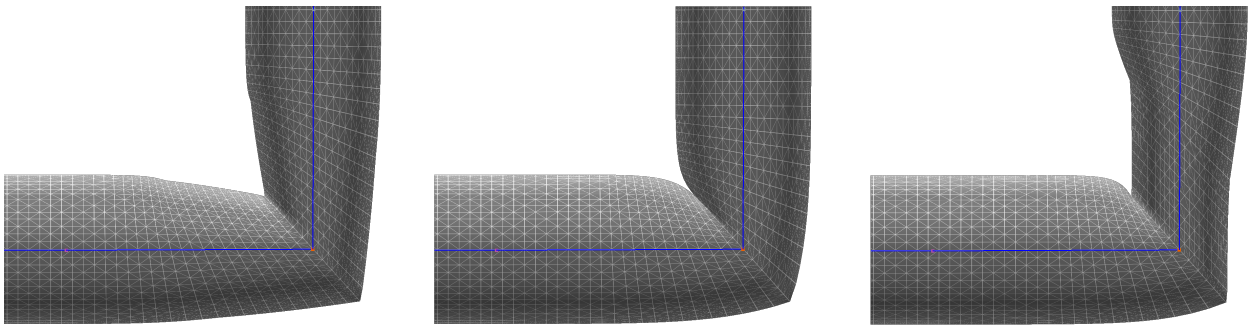


Figure 22: Using easing functions commonly found in web browser animations to change scale vector lengths. Applying across the length of a bone causes the surface silhouette to change. Left: linear. Middle: cubicIn. Right: quarticOut.

To support more deliberate surface edits made by an artist, this implementation used LSE for the actual deformation. Since this is a CPU operation, the skinning pipeline must be configured to save current vertex positions to a transform feedback buffer accessible by the CPU. LSE uses the buffer's positions as input, and once the artist has defined the ROI and deforms the surface, the vertex positions in the ROI are encoded as magnitudes and

rotations relative to their attached bones. These encodings and ROI classifications are saved to the GPU as vertex attributes. During the next execution of IRS, vertices in the ROI use the edited scale vector data instead of the default bind values. For non-overlapping ROIs and simple poses, radial deformations handle small edits quite well, see Figures 6 and 23.

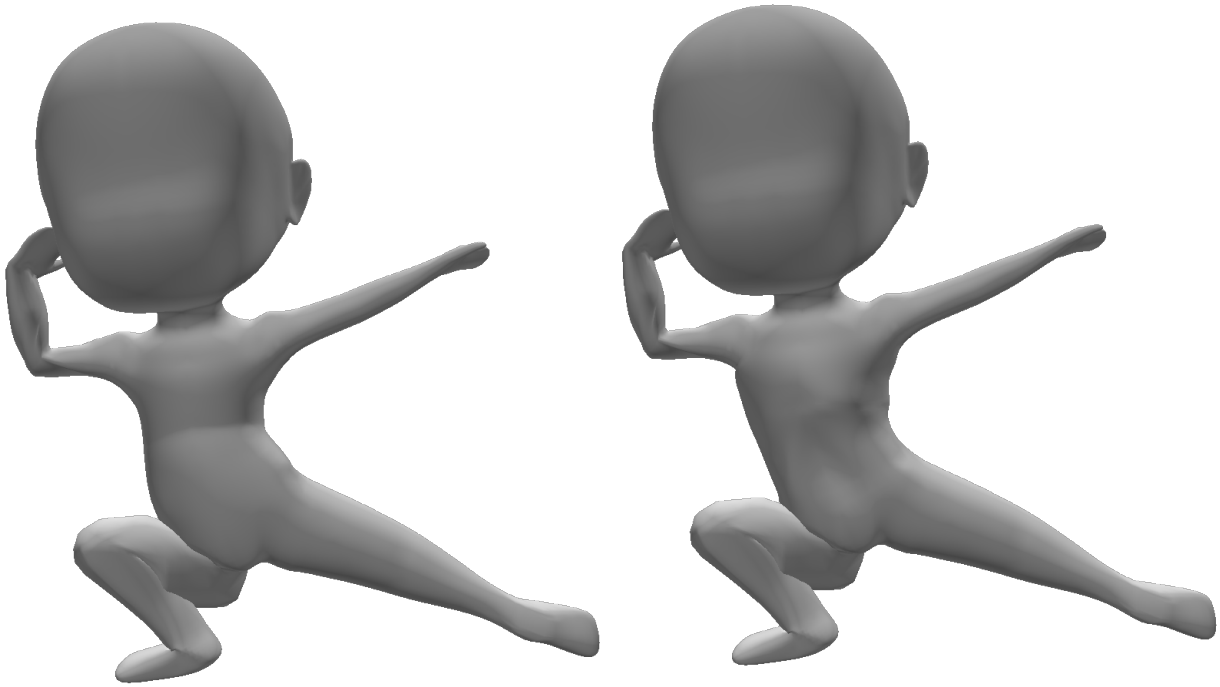


Figure 23: Deforming a posed mesh to create keyframes for breathing animation.

This process for encoding and decoding surface poses makes it possible to deform the skinned mesh, save encoded poses as keys, and interpolate between them, quite similar to animating with blend shapes. Considering that the surface deformations are supplied to the skinning shader, in both cases the surface deforms occur before skeletal skinning. The main difference is that radial deformations were designed to make surface changes executable in tandem with skinning, rather than as an initial stage. Blend shapes typically must be configured and constructed before rigging a mesh to its skeleton; having the ability to dynamically define and animate surfaces on a posed mesh is appealing.

## 5.2 Spring-driven surface editing

Incorporating radial deformations with anatomical poses works reasonably well for minor edits, but authoring larger deformations quickly reveals cases with unsatisfying results. Like with skinning, the core issue is that the deformed surface is defined relative to the mesh in a static shape, so as the mesh moves out of that shape, the surface edit starts to lose quality. A relatively small surface edit may grow undesirably large as the region deforms along the outer side of a joint bend. It is problematic attempting to find a balance between anatomical and surface deformations that can be solved directly, for any combination of skeleton pose and surface edit, that produces a believable deformation for the duration of movement. Switching from direct IRS to iterative spring deformers in Section 4.2 alleviated many of the former skinning issues, and it also introduces new opportunities for supporting surface deformations.

Spring-based skinning functions by measuring and applying various forces to each vertex over several iterations. The most straightforward approach should account for surface edits in Equation 22. Like with radial deformations, when a deformation is made, a edited handle vertex position is encoded as magnitude and rotation changes to scale vector $s'_i$ relative to skinned attachment point $a'_i$ such that their sum recreates the edited surface position $v'_i = a'_i + s'_i$. This time, however, the spring deformer does not directly set vertex positions to $v'_i$, but instead computes a new force $F_e(i) = v'_i - v_i$ intended to attract handle vertices to the edited positions from their current positions. In turn, neighboring spring forces cause the surrounding region to deform to compensate for the edit and minimize distances, i.e.,

$$F_{\text{net}}(i) = F_t(i) + F_s(i) + F_l(i) + F_b(i) + F_e(i). \tag{22}$$

Like the other forces, $F_e(i)$ has a stiffness coefficient and is updated on every iteration, so its magnitude decreases as the vertex approaches its predicted position. This arrangement solves several concerns with radial deformations. First, the resulting surface deformations integrate more effectively with anatomical deformations. In this approach, surface edits are more of a prioritized suggestion that the solver tries to accommodate rather than an absolute

instruction to change the shape. The iterative nature of the solver helps significantly by allowing surface edits to reach natural balances with skinning forces. As a result, when a surface deformation is near a bending joint, the other skinning forces help maintain relative shape integrity; the surface edit still changes in size, but this is more in scale with the anatomical changes than seen with radial deformations. Figure 24 demonstrates the effect.

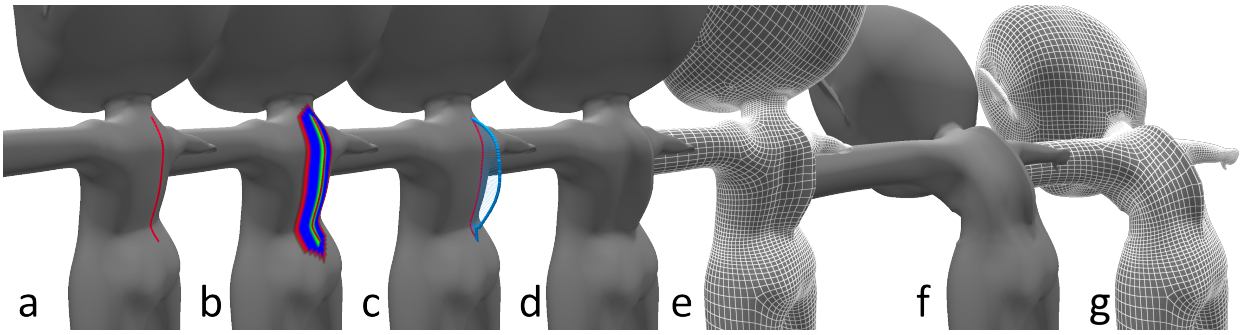



Figure 24: Surface editing by drawing directly on the mesh (a), which generates a region of interest (b) and provides an initial offset stroke to visualize vertex displacement (c). The spring deformer applies the surface edit (d, e) along with other deformers, such as skeletal poses (f, g).

This also addresses the issue of neighboring and overlapping edits. With radial deformations, the final surface edit must be known for all vertices before executing IRS. If $v_i$ and $v_j$ are neighbors and handle vertices for separate ROIs, there's a high possibility that applying both ROIs will cause a discontinuity between $v_i$ and $v_j$. Spring deformers can restore smoothness between them, even if $v_i$ and $v_j$ are initially discontinuous as shown in Figure 25.

If two or more edits affect the same vertex, they must be aggregated for radial deformations, for example as a weighted average. Say edit one includes $|s_i| = 2$ and edit two has $|s_i| = 3$. Weighed evenly ($w_1 = w_2 = 0.5$), the final edit becomes $|s_i| = 2.5$, when the artist might have preferred to see $|s_i| = 5$. Changing either the weights ($w_1 = w_2 = 1.0$) or the values (edit two: $|s_i| = 8$) can accomplish this, but either choice could have unintended consequences on other ROI vertices due to the direct, single-pass nature of IRS. For smooth blending results, it is safest to treat the edits as global blend shapes and find the weighted

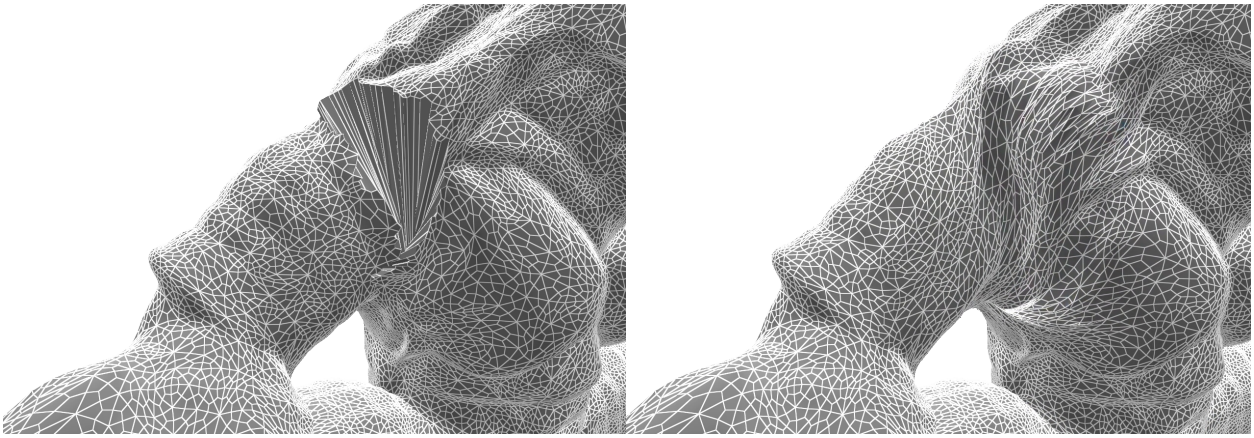


Figure 25: Left: artifacts from rigid skinning. Right: after applying spring deformers.

average, rather than try to compute and apply edits separately. This is less expensive than aggregating separate edits into a single ROI and executing LSE, but it shows less detail about how the overlapping edits interact. Spring deformers address the problem by adding $F_e(i)$ from all ROIs affecting $v_i$ to $F_{net}(i)$, treating each ROI's stiffness coefficient as a form of weight. The artist is still responsible for balancing edits by their location and stiffness, but it enables multiple surface effects to occur simultaneously.

Spring-driven surface edits can also produce desirable side effects, such as secondary animation, propagation of motion, and a sense of weight. These are only possible with radial deformations by making manual mesh edits to represent the effects, which involves much more keyframing and work for the artist. Animating a ripple moving linearly along the surface is possible with spring deformers, and requires defining a large ROI for a thin line of control vertices. To drive animation, the handle vertices receive forces that make them resemble a moving sine wave. As the wave travels, the control vertices' neighbors receive some of the passing energy and deform in its wake, see Figure 26. The stroke techniques presented in Chapter 7 make this possible with distance parameterization. These effects can apply during skeletal animation, providing extra features to accentuate the overall movement.

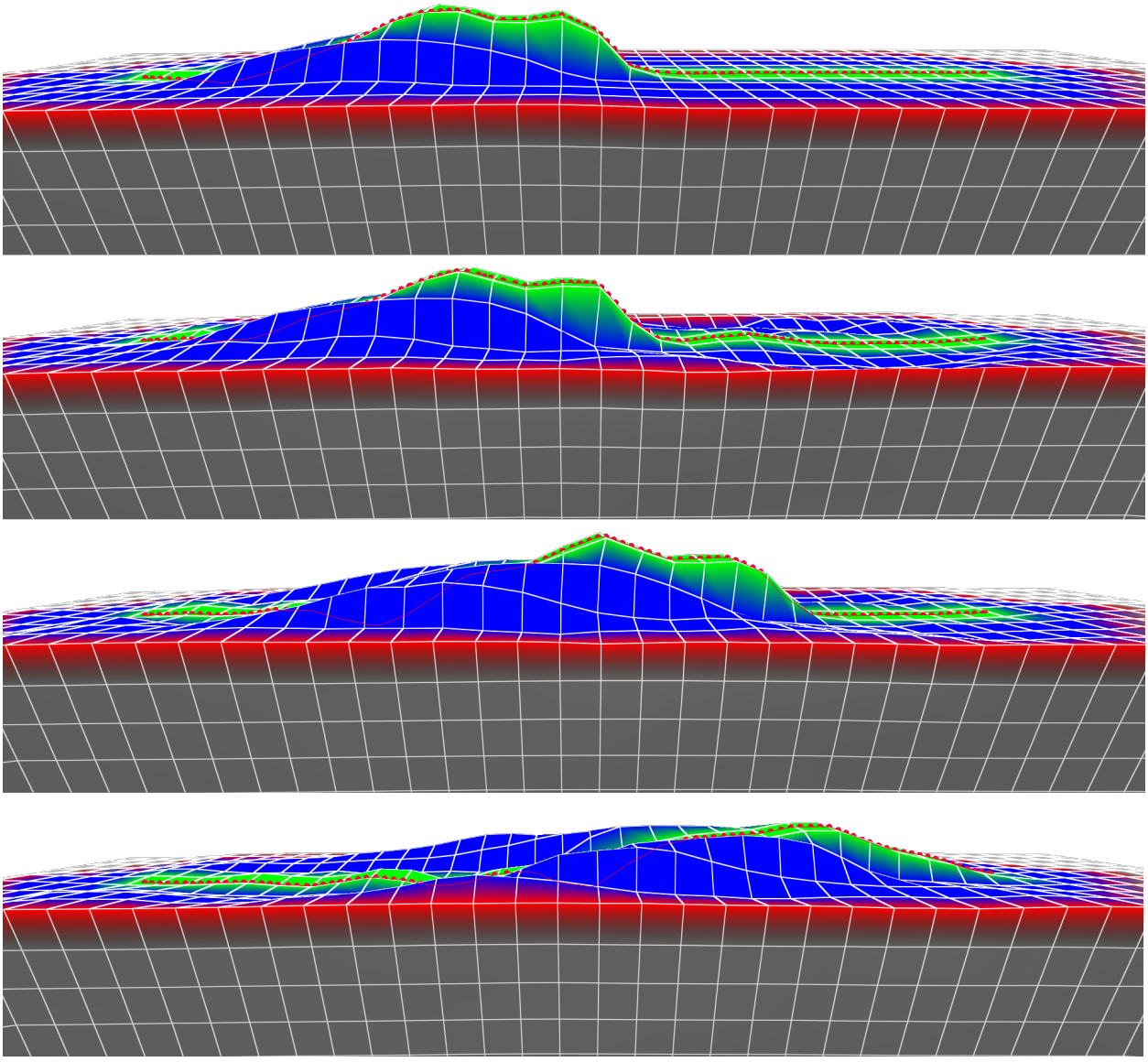


Figure 26: Secondary surface effects of a wave moving from left to right over several frames.

# 6 Implementation

## 6.1 Deformer graphs

As Chapters 4 and 5 show, combining skinning and surface deformers is not trivial. Does skinning happen before or after surface editing, or do they alternate on each frame? Can deformer results be computed in parallel and blended together with weights? More generally, for each deformer: where do its input vertices come from, and where should they go after? Can surface normals be transformed during skinning, or do they need to be recomputed to account for surface edits? The answers to these questions help form the *deformer graph* for the mesh, where each node represents a deformation operation, and the directed edges between them represent the execution order and input/output mappings.

The deformer graph is responsible for efficiently deforming the mesh before it is accessed for dependent operations like rendering, silhouette detection, and surface drawing. The deformer graph runs once per frame by starting with a generic skinning deformer that supports LBS, DQS, and rigid attachment-based skinning, followed by a spring deformer node. For a deformer to be *frame-independent*, its final results must be computed without using a previous frame's output as input. For example, an LBS deformer is frame-independent because it can compute the total mesh transform from its bind state on every time using joint transforms. In contrast, a real-time soft-body deformer is *frame-dependent* because it depends on vertex attributes (e.g. position, velocity) from a previous frame to deform the mesh. Not all skinning is frame-independent; recent work on implicit surface skinning uses a frame-dependent model for performance gains and more refined surface contact modeling [57, 59]. Our spring deformer is configurable for both modes of execution. In either case, it is responsible for iterating on the mesh data to approach a state of rest, and it does so with a double buffer setup. For a frame-independent execution, the spring deformer's input buffer is set to the skinning deformer's output, and the spring deformer saves its results to its own output buffer. On the next iteration, it swaps the input and output buffers and repeats the solve. This continues until the user-specified convergence scheme is met, and one

application frame may include numerous deformer iterations. For a frame-dependent execution, the skinning deformer must switch to using *differential transforms*, which describe the joint's transform between the last frame and the current frame, so when it is stationary, it is equal to the identity transform. It must also use the spring deformer's output as its input. This ensures that skinning only changes vertices relative to where they were on the previous frame, rather than from their bind position. The spring deformer is unchanged for this mode, so the performance gain comes largely from recycling the previous frame's output and applying very small transforms for skinning. This keeps the vertices closer together and saves the spring forces some work in restoring balance. The cost of the performance gain is independence, making it harder to distribute work on a frame-by-frame basis to a render farm-like environment.

In constructing the deformer graph, each deformer's implementation platform is taken into consideration. If all deformers are implemented on the same side of hardware (as functions executed by the CPU, for example, or as shader programs on the GPU), then memory access in between deformer stages is usually fast enough to prevent a bottleneck. Otherwise, if one deformer is on the CPU and the next is on the GPU, memory retrieval can negatively impact performance. For example, consider a deformer graph that runs IRS on the GPU followed by LSE on the CPU. This is the deformer graph that makes radial deformations in Chapter 5.1. This requires the GPU to store the skinned mesh somewhere in graphics memory, rather than pass it along the programmable pipeline to be rendered and discarded. Consequently, the rendering pipeline has to change to account for the interruption. In OpenGL, the use of transform feedbacks enable vertex shaders to save skinned vertices to a graphics buffer. Afterward, the CPU side can map to the buffer, retrieve the skinned mesh geometry, and use it to initialize and execute an LSE solver for a pre-defined ROI. Direct memory access (DMA) mitigates the retrieval cost by accessing it in place, but performing this on every frame still introduces a bottleneck. This could be eliminated using one of three options: move skinning to the CPU, run LSE and copy the results back to the GPU buffer, or move LSE to the GPU. The first option is straightforward and comes at an increase

in deformation time, which could easily outweigh memory access time for dense, higher-resolution meshes. The second option is a reasonable choice with, but in both cases, the overhead is considerably higher than a purely GPU approach. The third option is faster but much more challenging to orchestrate, as GPUs are better suited for dense linear system problems than sparse systems. As the goal remains to efficiently skin, deform, and render mesh surfaces in real-time, I execute all of the mesh's deformers on the GPU and use the CPU to primarily manage execution order and memory bindings. Before presenting the system, it is helpful to introduce the concept that makes it possible to build: the compute shader.

## 6.2 Compute shaders

The complete mesh deformation system is implemented using shaders, programs written for execution on the GPU, written in OpenGL shading language (GLSL). These are most often seen in the typical rendering pipeline, which, while programmable, expects shaders that perform the following sequence:[1]

1. skeletal skinning and world-to-screen space transformation in the *vertex* shader
2. (optional) adaptive geometry subdivision in the *tessellation evaluation* and *control* shaders
3. (optional) additional primitive edits in the *geometry* shader
4. surface shading and pixel definition in the *fragment* shader

This pipeline is designed for real-time performance, which adds limitations to the shaders used in each stage. First, attributes are commonly limited to 128 bytes per vertex, usually split into 8 16-byte values. For skinned and textured characters, the system will utilize a minimum of five: bind position, vertex normal, skinning weights with joint indices, and at

[1]There are additional stages in the pipeline that handle clipping, face culling, and rasterization, and while these are not fully programmable, they support various parameters to change their output.

least one set of texture coordinates. Using lower floating point precision can reduce memory cost or allow more attributes, but the pipeline is not built for dense or varying data sizes per vertex. In addition, these attributes are read-only, prohibits free slots from being used to store results for future use. Next, the pipeline does not allow communication between threads, so building a deformer that accounts for a vertex's neighbors is not trivial. Finally, the shader thread count is entirely dependent on the geometry bound for rendering, and the execution model automatically moves between stages until rendering is complete. When a draw command like `glDrawElements()` is executed, the vertex shader is invoked with $|V|$ threads for a mesh $M = (V, E)$, and the pipeline passes between stages automatically. It is important to recognize that mesh deformation and rendering are two distinct problems that this pipeline is optimized to solve in serial for a frame-independent deformer graph. Partitioning this process into separate solutions enables each one to better utilize the GPU while maintaining compatibility. To solve the deformation portion, we use compute shaders.

Because compute shaders are not part of the programmable pipeline, they do not have as many limitations, but in turn, greater caution must be taken to maintain mesh integrity. Given a buffer, any thread in a compute shader can perform read/write operations at any location, so typically the thread will use its invocation ID as an array index for safe, unmanaged access. Cross-thread data retrieval can be managed with calls to `barrier()` in GLSL, which acts as a blocking point until all threads have reached the same call. Finally, programs can be invoked with an arbitrarily high thread count, so it is possible to dispatch separate types compute shaders that bind to the same buffers. Unlike the programmable pipeline, a compute shader runs asynchronously, so for synchronization, calling `glMemoryBarrier()` after `glDispatchCompute()` on the CPU prevents continuation until the compute shader finishes. Separating tasks into distinct shaders and orchestrating their execution with these calls yields the complete deformation and rendering implementation shown in Figure 27. The specific stages are described in the following sections.

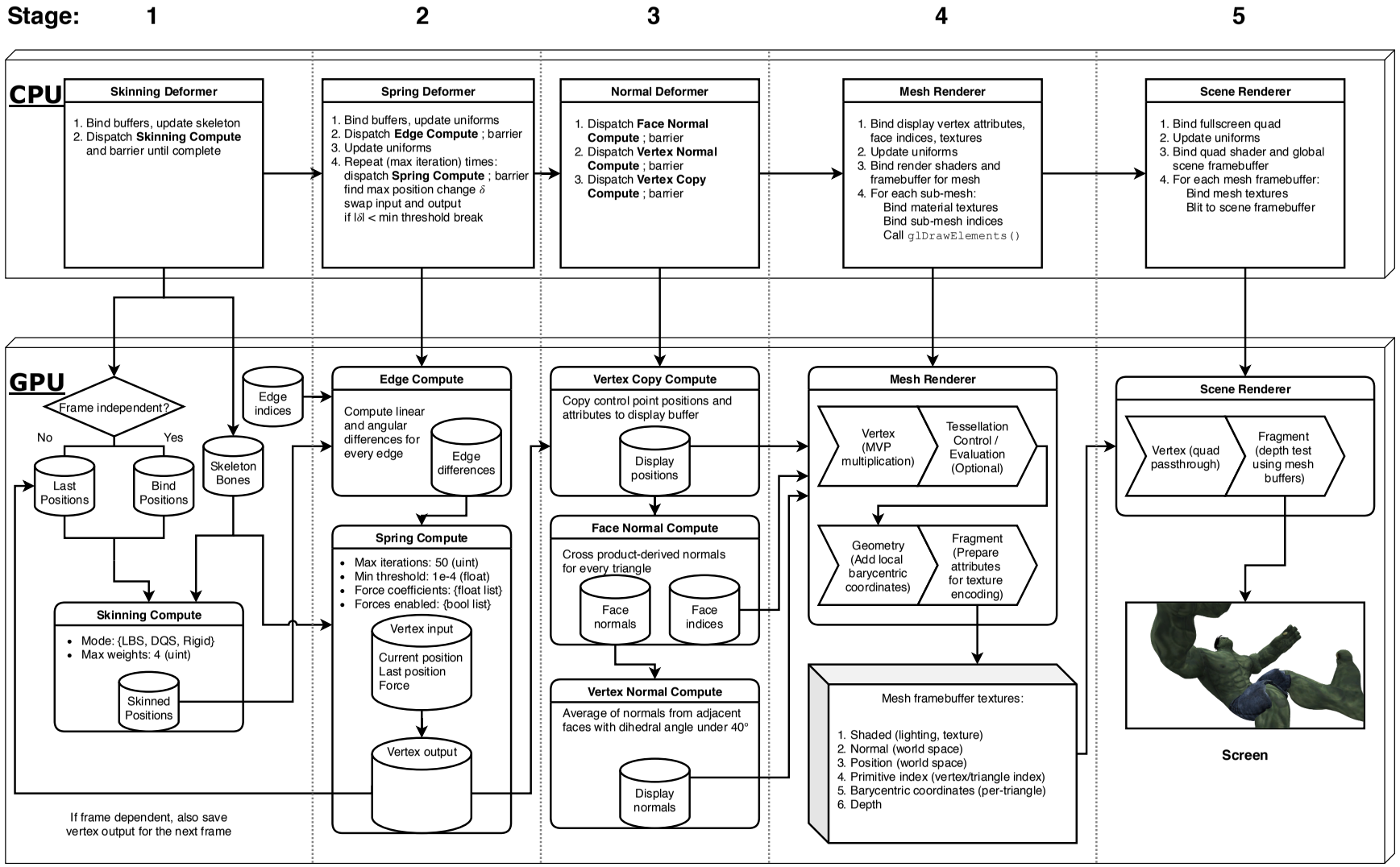



Figure 27: Diagram of the mesh deformation and rendering system. Execution flows from left to right, starting with the CPU and ending with the GPU in each stage. Vertical dotted lines indicate synchronization barriers between stages. Cylinders represent buffer allocations in GPU memory.

## 6.3 Deformation

The first stage is a classic skinning deformer with a compute shader capable of attachment-based rigid skinning and various flavors of LBS. Access to mesh data is provided through shader storage buffer objects (SSBOs), which, in the context of compute shaders, allow load and store operations indiscriminately. Dense mesh data, such as bind positions, attachment points, edge indices, and face indices, are copied to separate SSBOs before deformation begins. If the deformer graph is frame-independent, then the **Skinning Compute** shader performs a form of rigid or blend-based skinning using the mesh's bind vertex positions and the current skeleton pose. If the deformer graph is frame-dependent, the shader instead will use the last frame's computed vertex positions and the skeleton's differential transforms. While rigid skinning operates as expected with differential skeleton transforms, LBS tends to start off well and accumulate error over time, likely due to precision errors when blending transforms very close to $I$. In either case, the skinned positions are saved to an SSBO for the next stage.

Stage 2 in Figure 27 begins with a preprocessing step that stores $v_i - v_j$ and the signed angle between $s_i$ and $s_j$ for every edge $(i, j) \in E$. Consider an edge $(i, j)$: finding $i$'s net force requires evaluating the displacement with neighbor $j$, and vice versa for $j$'s net force. Considering that spring force evaluation is of complexity $O(|V| \times |E|)$ (although most 3D meshes not being fully connected), it is advantageous to compute the displacements once for each $(i, j)$ and retrieve them to compute force. In the case of $(j, i)$, the displacement sign is flipped. To access sparse data, such as edge indices per vertex, a flattening method generates a 1D array of size $\frac{|E|}{2}$ to store indices for all edges $(i, j) \in E, i < j$, followed by a 1D array of size $|V|$ containing offset-count tuples. This allows vertex $i$ to retrieve its edge count and index traversing the indices of all neighbors $j$.

The **Spring Compute** shader algorithm is given in Algorithm 1. Its purpose is to accumulate a net force $f_i$ to update each vertex $i$ based on the skeleton pose, skinned positions, shape control parameters, and surface edits. Since the spring deformer runs on

iterations and memory is cheap, it allocates separate SSBOs for input and output[2]. Forces from neighboring vertices are computed first. Assuming that $i$ is acting as a control handle for a surface edit, the net force from neighbors $j$ can be reduced so that over several iterations, $j$ receives greater influence from $i$. Bone torque is computed last, then the next position is found using the net force. This completes one iteration, and the CPU side keeps count and track maximum force magnitudes to determine if it should continue.

Algorithm 1: Pseudocode for spring deformer's compute shader

```
int i   = int(gl_GlobalInvocationID.x);
vec4 fi = vec4(0.);

for(int e = 0; i < indices[id].count; e++) {
    int j = edges[indices[id].offset + e];
    if (angularForceEnabled)   fi += attachmentForce(i, j);
    if (linearForceEnabled)    fi += linearForce(i, j);
    if (lengthForceEnabled)    fi += lengthForce(i, j);
}

if (surfaceEditActive(i)) {
    fi *= controlHandleWeight;
    fi += surfaceForce(i);
}

if (boneForceEnabled)          fi += boneForce(i);

output[i].force = fi;
output[i].position = input[i].position + fi * timestep;
```

[2]For a mesh with 10,000 vertices, at 128 bytes per vertex, two buffers would occupy 2.44MB. This could be reduced to a single buffer with some redesign and careful use of `barrier()` calls

## 6.4 Surface normals

After the spring deformer is finished, some additional work is required in Stage 3 before the results can be rendered. There are two issues to address: the difference between the number of control points and display points, and the validity of surface normals after spring deformation. The first issue is demonstrated with a cube mesh, which contains eight control points. Rendering it properly requires 24 vertices: three at each control point to account for the different surface normals of adjacent faces. LBS on the GPU is especially efficient, so it is often acceptable to duplicate vertex data as needed for correct rendering when skinning. However, this practice is not ideal for spring deformers because the execution is higher due to vertex neighbor traversal. Instead, the spring deformer iterates on the control points until convergence, then **Vertex Copy Compute** runs with $|V|\times$ three threads, one for every display point, and copies positions from the control buffer to the display buffer at their respective indices[3], which provides the means to render surface triangles with the right normals, but they still need to be computed in the next step.

As spring deformers can form non-linear deformations, it is difficult to compute a transform that correctly rotates the bind vertex normal as usually done in LBS. Alternately, recomputing normals with the GPU on every frame is entirely tractable. After the display positions have been copied, **Face Normal Compute** computes the cross product between each display triangle's vertex positions, then **Vertex Normal Compute** finds the weighted average cross product of a display vertex's adjacent faces using flattened adjacency data (prepared and accessed using the same offset-index technique for vertex edge traversal in Stage 2). To preserve sharp creases and corners, the shader checks for a dihedral angle threshold to omit adjacent faces that exceed it. This is the same technique employed in demos from LibIGL, but implemented on the GPU for optimal performance [21]. Despite the use of three separate compute shaders, this approach is consistently fast and capable of providing high-quality normals for any deformation occurring. Although this system is more complex and costly than traditional GPU skinning, the execution times shown in Table 1

[3]In the simplest form, a control point at index $i$ has display indices at $i, i*2, i*3$.

are quite suitable for real-time.

| Mesh | Bones | Vertices | Triangles | $\overline{I=1}$ | I=25 | I=50 | I=100 | Max I |
|---|---|---|---|---|---|---|---|---|
| Chibi | 36 | 16314 | 32624 | 0.02 | 0.53 | 0.91 | 1.78 | 524 |
| Box | 4 | 1802 | 3600 | 0.01 | 0.28 | 0.54 | 1.10 | 909 |
| Plus | 17 | 5634 | 11264 | 0.01 | 0.34 | 0.59 | 1.27 | 787 |
| Human | 85 | 10774 | 21204 | 0.02 | 0.42 | 0.84 | 1.50 | 617 |
| Armadillo | 19 | 90000 | 180000 | 0.03 | 0.77 | 1.59 | 3.04 | 323 |

Table 1: Compute times in milliseconds for different iteration counts across various input meshes, including the mean single execution time $\overline{I=1}$, and the maximum iteration count possible to maintain 60 FPS (estimated as $\frac{10 \text{ ms}}{\overline{I=1}}$, reserving 6 ms for rendering and other frame updates). With automatic convergence threshold scaling using the mesh's bounding box, the default choice of 50 iterations is suitable. However, should the artist wish to use smaller timesteps and more iterations, there is reasonable support for costlier configurations. For reference, a compute shader implementation of LBS completes in $< 0.01$ ms for all meshes listed.

## 6.5 Deferred rendering

Once deformation and normal recomputation are finished, the mesh geometry is ready to render. The **Mesh Renderer** in Stage 4 runs the more familiar OpenGL programmable pipeline, starting with a vertex shader that simply transform display positions to screen space using the model, view, and projection matrices. Since the deformed mesh data is already stored in SSBOs on the GPU, this stage only needs to bind vertex attributes and textures to the right buffer locations. Vertex attributes such as normals and texture coordinate are then passed along to the next stage. If tessellation is turned on, additional vertices will be generated using Phong tessellation [1]. The geometry shader is configured to process individual triangles and add a new barycentric attribute to its vertices. This is useful, as the rasterizer

will automatically interpolate these coordinates across each triangle, which, in combination with the triangle's index, provide pose-independent coordinates for any visible mesh pixel. This output can be used to solve interface-related surface editing problems introduced in Chapter 7. Finally, the fragment shader performs typical lighting and shading, but instead of displaying results to the screen, it saves various fragment values to separate image textures reserved in GPU memory on a framebuffer. This deferred rendering pipeline grants access to different representations of the mesh, such as the surface appearance, positions, normals, depth, and nearest primitive IDs, as images for further processing. Figure 28 illustrates the renderer's output for one mesh.

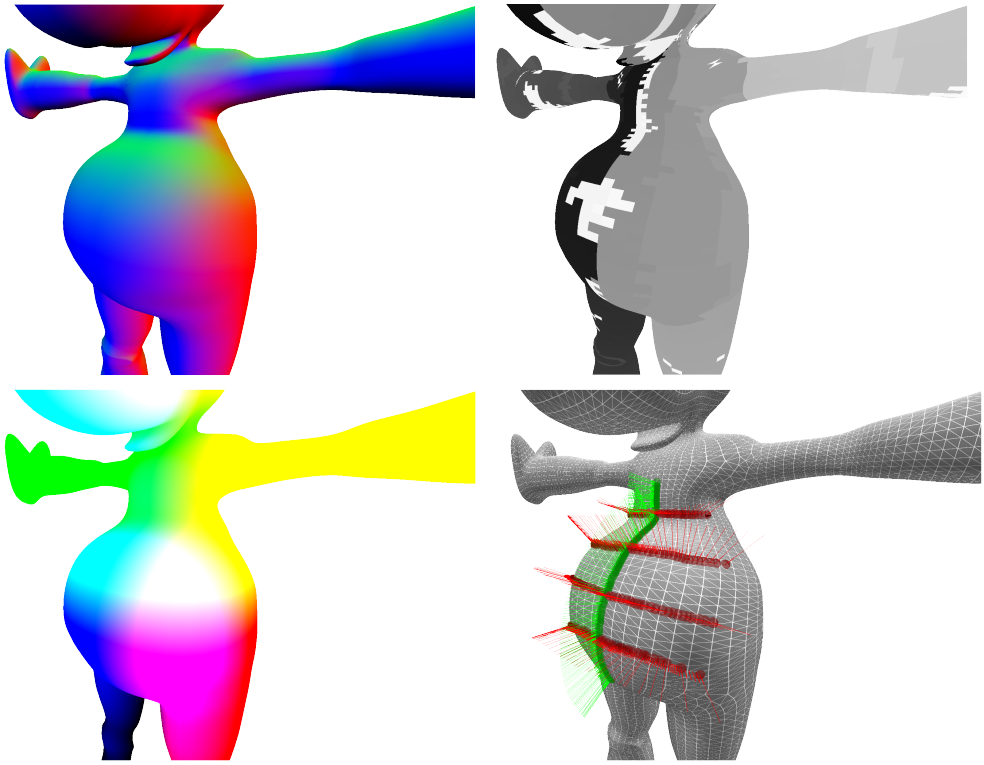


Figure 28: Deferred rendering output, separating mesh data such as surface normals (top left), positions (bottom left), and primitive IDs (top right) to their own textures, which can be used for other tasks such as surface drawing (bottom right).

The final task is to see the results by compositing mesh framebuffer textures onto the screen in Stage 5. The **Scene Renderer** clears the screen, then iterates over the meshes,

binding to each one's textures. If there is only one mesh, the following sequence will effectively copy (or *blit*) the mesh's texture to the screen. The vertex shader draws a single quad, or two triangles, to perfectly fit the visible window. This forces the rasterizer to cover the whole screen. Then, for each pixel, the fragment shader compares the mesh's depth texture at its coordinate to the screen's. If the mesh's depth value is closer, the shaded texture value will be written to the screen. Separating deformation and rendering disrupts the use of the Z-buffer technique between meshes to determine which faces are ultimately visible, so this method restores it. While the typical Z-buffer technique is bound by the number of faces to render, this is bound by the number of meshes. Each mesh has already invoked the Z-buffer test for its own geometry in Stage 4, leaving only a lightweight composition task for the final stage. When the renderer finishes, the current frame is finally visible.

# 7 Interface

The real-time interface for this software has a mantra at its core: sketch-based tools to support creative flow. The discussion in Section 2.3 is more in-depth, but the general idea is to encourage natural motions and rapid exploration of ideas. In this pursuit, the interface aims to provide similar controls for both anatomical and geometric actions. Following the prior work on building embodied interfaces for creativity support [52], I offer two interface modes: a traditional desktop mode with keyboard, mouse, and stylus controls, and a virtual reality mode with 6DOF tracking.

## 7.1 Input strokes

With a sketch-based interface, the artist uses an input device —a mouse, stylus, touch screen, motion tracked controller —to draw lines in 2D or 3D space that drive actions in the application. The process of recording and processing these *input strokes* is critical for all actions that depend on them. First, the device's coordinates and current time must be recorded for the duration of input. This data set is typically noisy and redundant, so to prepare it for interface tasks, I perform the Ramer–Douglas–Peucker algorithm on the input positions to downsample them into a manageable set of points. This operates by defining a line segment $l$ between the first and last point in the series, $a$ and $b$, then finding the farthest point $p$ from the segment. Assuming that $Distance(p, l)$ is below a threshold $\epsilon$, all of the interior points between $a$ and $b$ can be discarded. If the distance is larger than $\epsilon$, then the algorithm will repeat recursively on the two line segments $a$ and $p$, and $p$ and $b$. This works nicely for both 2D and 3D inputs, as long as $\epsilon$ is set appropriately for the input domain. For 2D strokes on a 1920×1080 window, $\epsilon = 8$ pixels. For a 3D stroke in a $1\text{m}^3$ bounding volume, $\epsilon = 0.01\text{m}$.

The remaining points are used to construct a composite Bézier curve, such that curvature at the end of one segment coincides with the next. At this stage, the user's input has been transformed into a parameterized stroke ready for various tasks. Given a value $t \in [0, 1]$,

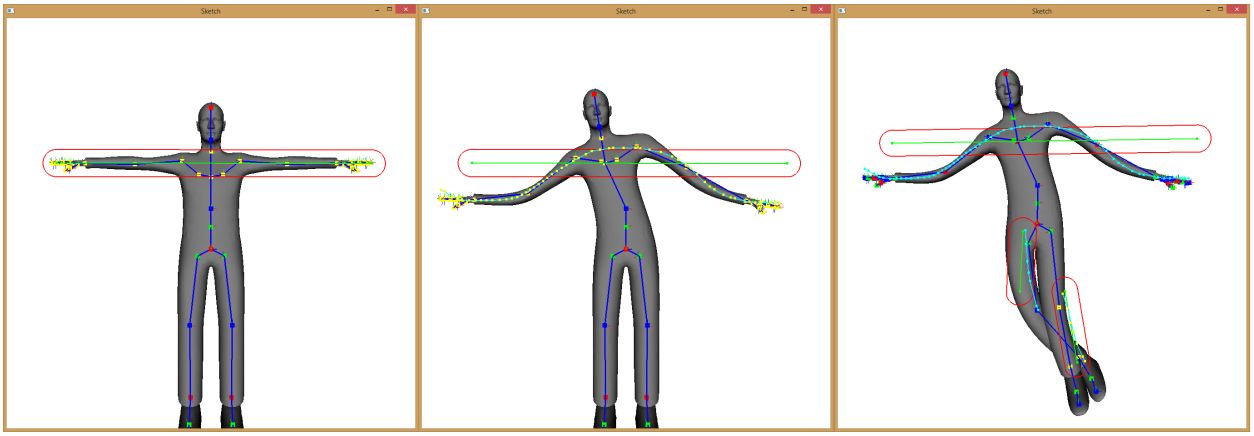
we can compute the closest position, timestamp, and Frenet frame (tangent, normal, and binormal) to $t$ on the stroke, each of which is useful for the posing and editing tasks described in the next section. Before they can be used, any strokes made by 2D input devices need to be unprojected from window coordinates to 3D world coordinates. Different approaches are used for skeleton and surface tasks.

## 7.2 Skeleton posing

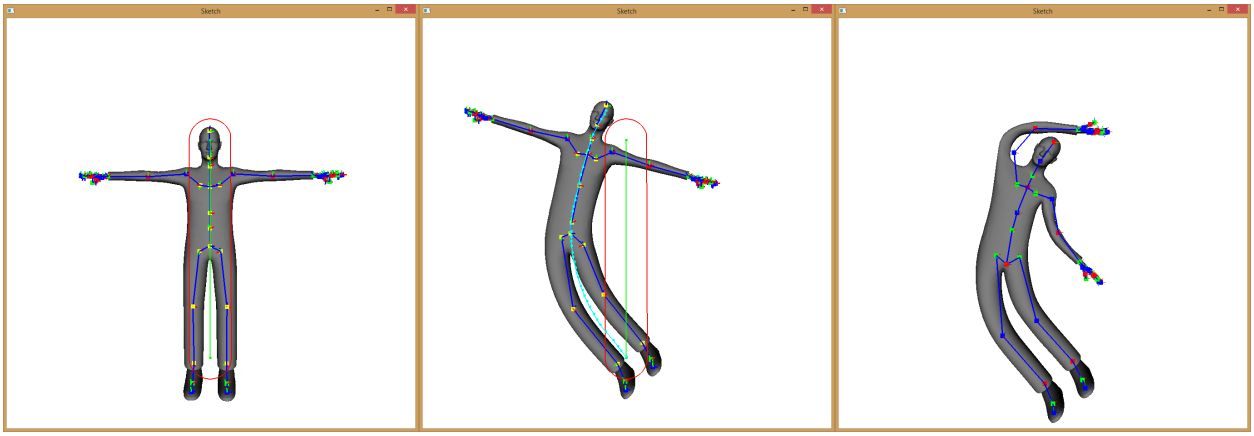
For 2D input devices, the saved coordinates are mapped onto a plane set by the user. By default, the plane is centered on the skeleton root with its normal locked to point toward the camera position, and the user can adjust it with the translate, rotate, and scale controls typical to 3D modeling programs. To find the mapping between points on the screen and plane, we need to construct a world-space ray corresponding to the screen point. The ray's origin is simply the camera position. To find the ray's direction, we first transform the screen point into normalized device coordinates (NDCs) such that $x, y \in [-1, 1], z = -1$. We multiply the NDC by the camera's inverse projection matrices to transform the point to eye space. Finally, we multiply by the camera's inverse view matrix to find the ray's direction in world-space. Once the rays are found, we perform simple ray-plane intersection tests and save the world-space coordinates for any hits. For 3D input devices, the device's coordinates are saved during the stroke drawing action, then transformed into world-space using the device's inverse transformation matrix. The saved positions are passed on for reduction and stroke construction.

Posing begins with the user drawing a baseline stroke $b$ along the desired joints. After drawing the baseline, joints are selected based on stroke proximity akin to brush width for digital painting tools (Figure 29a). To help with posing, each selected joint $j$ is parameterized to $t_j \in [0, 1]$ such that $b(t_j)$ is the closest point on the stroke to $j$. The displacement $d_j = j - b(t_j)$ is also saved to maintain relative differences during posing. Next, the user draws an offset stroke $o$ that defines how the selected joints should be transformed relative to $b$ (Figure 29b). The Frenet frames at $t_j$ are found on both $b$ and $o$ as $F(b, t_j)$ and $F(o, t_j)$.

Because a Frenet frame is an orthonormal basis for a 3D rotation matrix, the difference in orientation $R_j$ between the baseline and offset strokes is $F(o, t_j) \times F(b, t_j)^{-1}$. With this known, the final transformed position of $j$ is found as $o(t_j) + (R_j \times d_j)$.



(a) Prototype of joint selection and posing with visible selection radius. Spline control knots are highlighted along the strokes.



(b) Line of action posing: drawing a baseline to select the character in bind, then drawing an offset for the overall character pose to match.

Figure 29: Stroke-based skeletal posing using a mouse for 2D input.

Since the skeleton is represented by hierarchical transforms, where each joint is a local transform applied to its parent's transform, some additional work is needed to properly represent the pose changes in local bone spaces. This is handled by saving the new pose positions for each selected joint, then applying the changes sequentially starting with the joint closest to the skeleton root. When a joint gets posed, any changes made to its parents

are already applied. The overall selection and posing process is sufficiently instantaneous that the artist is free to undo and redo offset strokes to find an ideal pose. Alternately, they can select control knots on a stroke and adjust them manually to watch the character deform in real time. Figures 30 and 31 shows the process of drawing strokes to pose the character skeleton. This approach is similar to the line-of-action-motivated sketch interfaces seen in prior work, but without explicitly snapping the baseline stroke to lie collinearly along bones [28] [40].

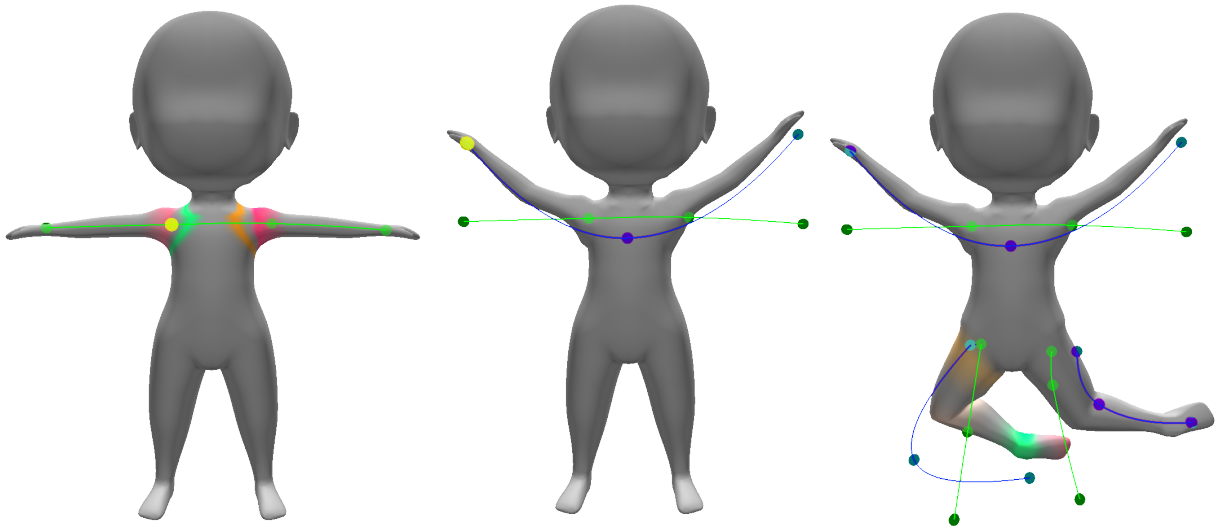


Figure 30: Posing the character with input strokes. The baselines (green) select joints and serve as references for the offset curves (blue). The process can be repeated and layered as needed.

## 7.3 Surface editing

Surface deformation requires a region of influence and the desired transformations for a subset of points within the region. We have experimented with several techniques for interacting with the 3D surface, starting with prior work in the area. Our first sketch-based approach was an implementation of SilSketch, which automates detecting and deforming silhouette vertices in screen-space, then uses LSE to deform the surrounding area [64]. To use this

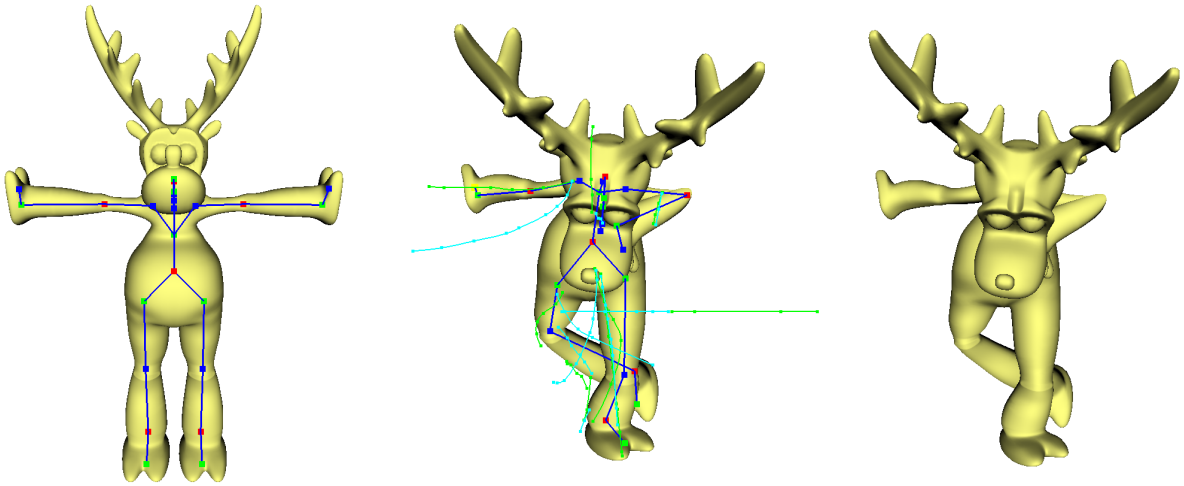


Figure 31: Posing with baseline (green) and offset (teal) strokes.

technique, the system must identify all mesh vertices that lie on or near a silhouette from the camera's perspective. We achieve this by running a Laplacian edge filter in a fragment shader, using one of the renderer's framebuffer textures as input; by default, we use the surface normal texture. The filter's output is saved to texture, then used as a mask on the renderer's vertex index framebuffer texture. Any pixel with nonzero values in both textures represents and identifies a silhouette vertex by its index in the mesh, see Figure 32. The computation is repeated any time the camera changes during surface editing, but having most of the work completed in shaders prevents it from being a bottleneck.

With edge detection solved, the system takes a shortcut to allow immediate ROI definition and surface editing with one stroke. The artist draws an accent stroke near a silhouette in the desired shape. Rather than perform polyline construction and similarity comparisons as used in SilSketch, we select the two vertices closest to the beginning and end of the stroke. These act as anchors to select the remaining silhouette vertices in between the two, forming a baseline stroke. When the silhouette vertices have been identified, a BFS algorithm selects vertices up to $n$ edge counts ($n = 4$ by default) away to serve as the passive set of ROI vertices. The artist can optionally choose an alternate selection scheme, such as all vertices within an adjustable distance, measured either as Euclidean or as a sum of edge lengths. Finally, the

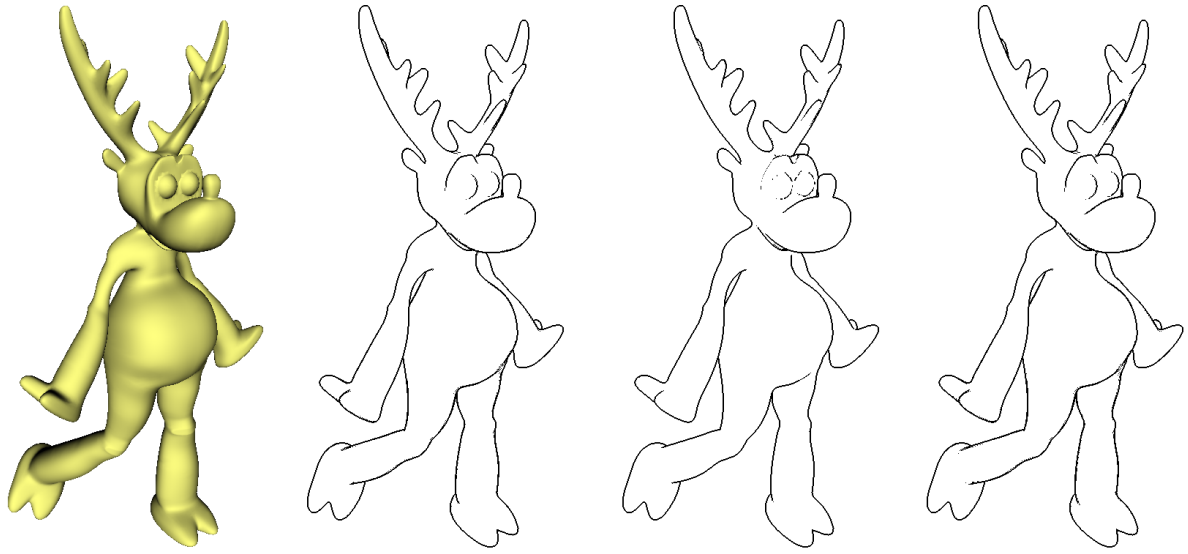


Figure 32: Detecting silhouette edges using Laplacian filters. From left to right: base mesh, then filter output using the depth, normal, and position framebuffer textures.

outermost immediate neighbors of the passive set are selected and marked static to designate a boundary for the ROI. After constructing the ROI, we map where the silhouette vertices should be moved onto the input stroke using parameterized positions, similar to how joint positions are mapped, then provide the edit data and ROI to an LSE solver, which returns positions for the passive vertices that constitute a smooth deformation.

This is quite a powerful technique, but has some drawbacks in expressive power. The SilSketch method is limited to vertices clearly detectable on silhouette edges. In principle, this is often adequate, as clear character silhouettes are extremely helpful in animation, and this technique allows the artist to focus solely on important segments. Adjusting either the camera or the mesh transform is typically enough to bring a specific feature into editing mode, except for selecting flat regions between more distinct edges. For example, it is impossible select anything but one of the 90° edges visible by camera on a right rectangular prism, see Figure 33. We wished to support non-silhouette edits by permitting the user to create ROIs on arbitrary mesh surfaces. This required a new method for vertex selection,

which is drastically accelerated by rendering output as described in Section 6.

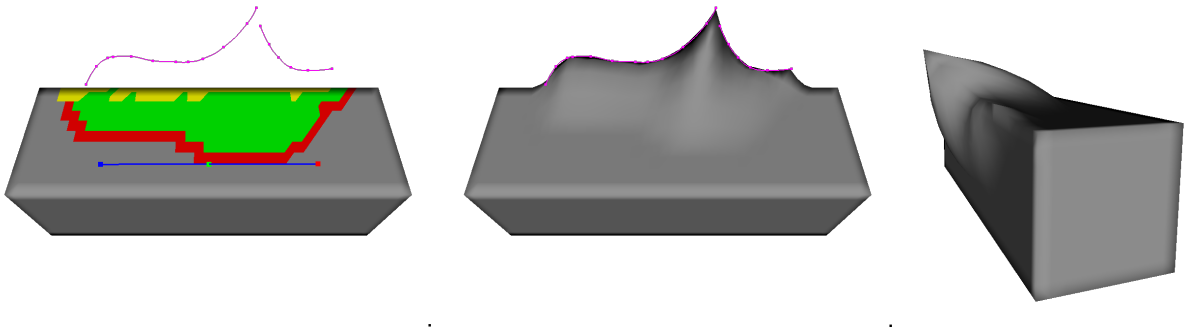


Figure 33: Silhouette-based surface editing.

## 7.4 Surface querying

Surface querying is a frequent problem in games and other 3D applications where information is needed about the scene at the input device's coordinates. Game engines like Unity will construct a ray using the camera and mouse position, then cast it into the scene to find the nearest intersecting object, often using the physics engine for optimization through bounds checks and object culling. This practice is quite optimized for primitives such as planes, spheres, and boxes, but for polygonal meshes, the cost increases with the scene's triangle count. Constructs such as octrees or k-d trees help accelerate the intersection test, but if the geometry is dynamic rather than static, these must be reconstructed if the object moves before raycasting. Instead of using rays, it is possible to use the deferred renderer's framebuffer output to drastically improve performance for such interactions. By configuring the renderer to output primitive IDs to a separate texture, it is trivial to convert input device positions to texture coordinates and simply look up values under the device's location. Since the renderer is already handling geometry culling, fragment sorting, and rasterization, it makes sense to take advantage of the output rather than resort to physics. This approach is not completely new, but it's more often used to identify objects for selection rather than to perform detailed surface work.

To support freely-drawn ROIs, we start by saving the input device's coordinates as usual.

Instead of mapping them onto a plane as done for skeleton posing, I use the screen-space positions to represent texture coordinates and retrieve surface data from the meshes' framebuffers, including position, normal, barycentric coordinates, and geometry IDs. After reducing the retrieved positions using line reduction, I construct a stroke that lies precisely on a mesh surface. To maintain placement when the mesh deforms, the stroke's control points are mapped to the surface using their source triangle IDs and barycentric coordinates. These are sufficient to compute their current positions and surface normals in world-space. This configuration also enables vertex selection and offset tracking, similar to the skeleton strokes. This also allows the artist to choose vertices for regions of interest.

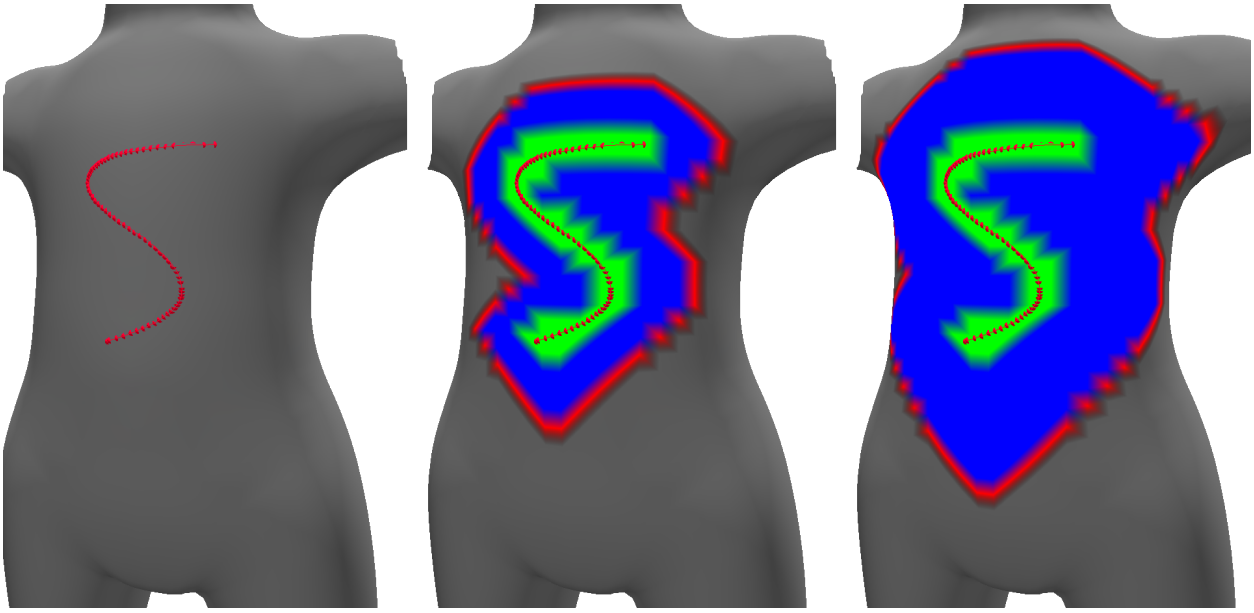


Figure 34: Creating a region of interest for deformation. Left: the artist draws a stroke to indicate the area of control. Middle: the controlled region (green) acts as a seed to grow the passively-controlled region (blue) up to 4 vertices away, with the outermost region (red) marked as boundary constraints. Right: the same stroke producing a region with up to 8 neighbors instead of 4.

Defining a ROI is done with a single stroke drawn freely on the surface. Any vertices attached to the faces intersected by the stroke are designated as kinematic, meaning their movement is controlled by artistic intent rather than by spring forces. The rest of the ROI is

found by visiting all vertices within $n = 4$ edges from the kinematic vertices. After the ROI is constructed, the stroke can be manipulated to approximate the desired shape. At this point, the data could be passed to an LSE solver on the CPU, but to unify the deformation spaces, our system encodes the kinematic vertex positions as scale vector angles and lengths relative to their attachment points. These are passed to the spring deformer to find and apply the forces needed to deform the region. This approach is preferable to CPU-based LSE because it integrates surface edit forces directly with the other spring forces on the mesh, resulting in greater compromise between anatomical and surface deformations. The use of strokes also makes it convenient to drive deformations programmatically. For example, the parabolic bulge formula provided in Chapter 5 can be easily applied to compute kinematic vertex offsets using stroke position parameterization and can be seen in Figure 24. In this edit, drawing a stroke to select handle vertices automatically defines offsets for each vertex.

Hardware-accelerated surface querying is not perfect. The process relies on rasterization, and the number of pixels per triangle decreases with object scale. Accuracy is dependent on the scene renderer's resolution and camera placement, which the artist can manage easily enough. Another drawback is the limitation to querying the nearest surface. This can become an issue for layered meshes, which may have separate geometry for skin, clothes, and hair, since only the outermost mesh will receive the edit. With this in mind, the method is best suited for single-mesh rigs. To edit layered meshes, the system can take advantage of the deferred renderer's granularity. Since each mesh is initially rendered to its own framebuffer textures, the input coordinates could be used to define strokes on every surface regardless of visibility. However, differences in mesh density, normal direction, and camera distance highly suggest that despite receiving the same input stroke, layered meshes may still deform quite inconsistently. To continue exploiting graphics hardware, one possible approach is to run the intersection test in a geometry shader and test against all mesh triangles, saving the intersections to buffer objects for later retrieval. As geometry shaders are notorious for negatively impacting the rendering pipeline on non-Intel graphics hardware, I recommend the framebuffer-based approach when applicable.

## 7.5 Virtual reality

As VR hardware and software become increasingly available to professionals and consumers, we are witnessing an explosion of experimentation and exploration in human-computer interaction across many fields, including health care, occupational training, and social networks. In 3D graphics, prominent software like Autodesk Maya and Blender both support plugins that allow the artist to perform modeling and animation tasks in an immersive, hands-on VR environment. There is still much to be done before VR interfaces can rival the functionality of the mouse and keyboard, but the tracked headset alone provides a remarkably useful stereoscopic visualization of the scene, allowing the artist to check features and shape with natural head movement instead of tedious, repetitive camera manipulation in the desktop interface.

We support skeleton and surface editing in VR with the same stroke controls used in the desktop interface. For skeleton posing, the process is actually a bit more straightforward. The artist can draw baseline and offset strokes directly in the 3D space without any need for unprojection, and they experience the same low-cost benefits that enable interactive posing and an undo-friendly workflow. The bigger challenge is with surface editing, namely with drawing directly on the mesh. This was possible in the desktop interface because there is only one camera, and the renderer's framebuffer output is instantly compatible with a 2D input device. In VR, there is one camera for each eye, so the choice of which to use is no longer obvious. Worse yet, even with configuring the renderer to save framebuffer output for texture lookups, the data is only valid from the headset perspective. This means the artist would have to essentially draw surface edits by using their nose as a virtual pointer. Prior work shows this is possible, but ideally we would prefer to use the hands for drawing [14]. Casting rays from the hand controllers on the CPU is not a desirable option, as this would introduce additional challenges to real-time performance, such as the need to copy deformed vertices from GPU memory and construct an octree every time the mesh changes shape. Without some space partitioning-based technique to accelerate ray intersection tests, ray casting would be too expensive for real-time. This approach is fine for static meshes, but

grows prohibitively expensive with animated meshes.

Our solution to this problem is simple and compatible with the existing system. I implement renderers using a class hierarchy such that the base class handles framebuffer allocation and access for each meshes in the scene. This allows useful functionality to be shared in both desktop and virtual reality mode with minimal coding. Consequently, this design also makes it trivial to instantiate a separate, lightweight, low-resolution[4] renderer with a camera that tracks a controller's pose, similar to how the main renderer's camera tracks the headset. The only purpose of this new renderer is to activate when the artist is drawing a stroke and record visible surface data — position, normal, primitive IDs — from the center of its framebuffer. When the artist finishes drawing, the saved data are converted into a surface stroke in the same manner as desktop-drawn inputs. Recall from Chapter 6 that while deformation is expensive, rendering is cheap. The renderer only has to perform model-view-projection multiplications on geometry which is already deformed and accessible on the GPU. As a nice side effect, this makes viewing and drawing independent tasks. In the desktop interface, the artist can only draw on surfaces visible in the application window. However, since the VR controller has its own camera, the artist can draw on surfaces not immediately visible in the headset. Conceptually, they are free to move completely around a mesh to draw a continuous stroke around its circumference. The same act on the desktop would need several disjoint strokes as the artist stops to rotate the camera.

[4] ($16 \times 16$ by default, but $1 \times 1$ is as low as I could manage)

# 8 Conclusion

## 8.1 Contributions

The restrictions in typical real-time skinning systems prompted us to work on a more flexible solution. Our first goal was to provide an animation system that could accommodate anatomy and free surface deformations in a unified structure. I achieved this in part with IRS in Section 4.1, which is compatible with existing programmable graphics pipelines, but limited in how well it integrates the two deformation types. I overcame this by migrating to an iterative system and modeling spring relationships between the mesh and its skeleton in Section 4.2. Because the iterative solver was not compatible with the programmable pipeline, we developed an alternate pipeline using compute shaders arranged into a deformer graph. This new pipeline is highly modular, optimized for high performance, and fully located on the graphics hardware, so rendering the results is immediate and low-cost. I also described how to arrange the rendering portion of the pipeline to perform high-precision surface drawing and editing with negligible performance impacts. To enable these, I implemented sketch-based interfaces for both desktop and virtual reality environments. Both of these leverage rendering output and user input to pose skeletons and edit mesh surfaces.

The core components of this work — spring-based skinning, the deformer pipeline, and the render-driven interface — were developed in lockstep, the needs of one influencing the construction of another. For this to be a real-time system, there was no simpler way. Although in this form the components are rather cohesive, they are coupled loosely enough to account for a range of artistic and technical needs, and the insights gained from building each one are quite useful on their own.

## 8.2 Future research

The interfaces built for these tasks was motivated by results found in prior work [52]. With the rapid growth of cross-reality hardware and software, I would like to continue research in the area of creativity support, especially for animation tasks such as camera navigation,

tactile posing and surface editing, and natural motion retargeting.

The success and customization of spring deformers shows promise for the future feasibility of real-time, non-linear shape control. While the spring deformer default settings are fairly stable, it quite easy to modify any of the system parameters — timestep, stiffness coefficients, threshold, max iterations — to a point of instability and degenerate solutions. Having worked with these springs long enough, I am highly interested in developing automated safe parameter range settings. This would make available more time to explore stylistic differences and cut down on fine-tuning.

In the current system, spring deformers are unique to each mesh, and they do not internally or externally register intersections or collisions. External collisions with other meshes are not a major concern for this work, which focuses more on skinning a rig than handling global physics, and self-collision events are largely mitigated by the acting forces without explicit measurement and response. For more direct contact modeling, it could be worth investigating a hybrid implicit skinning system with spring deformers acting on a surface's control points. Given the framework here, I believe a global spring deformer with discrete collision detection is worth exploring first. Granted that such a system would increase the runtime complexity, it would be worthwhile to consider optimizations such as LOD or point sampling. Along these lines, it seems worthwhile to apply springs using an alternate attachment scheme. Given the system's potential scalability, I would like to see how well spring-based skinning works with a discrete mesh volume representation.

Maslow's ~~hammer~~ spring states, "I suppose it is tempting, if the only tool you have is a Hookean system solver, to treat everything as if it were a spring." Springs happen to be incredibly useful in dealing with real-time mesh deformation problems, but they have limitations in the types of material they can represent. Unless, of course, the solution to that problem turns out to be, "add more springs," in which case, I have already solved everything and just need to buy more GPUs.

# References


[1] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2018.

[2] Baptiste Angles, Marco Tarini, Brian Wyvill, Loïc Barthe, and Andrea Tagliasacchi. Sketch-based implicit blending. *ACM Transactions on Graphics (TOG)*, 36(6):181, 2017.

[3] Oscar Kin-Chung Au, Chiew-Lan Tai, Hung-Kuo Chu, Daniel Cohen-Or, and Tong-Yee Lee. Skeleton extraction by mesh contraction. In *ACM transactions on graphics (TOG)*, volume 27 (3), page 44. ACM, 2008.

[4] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. In *ACM Transactions on graphics (TOG)*, volume 26, page 72. ACM, 2007.

[5] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[6] Mario Botsch, Mark Pauly, Markus Gross, and Leif Kobbelt. PriMo: Coupled Prisms for Intuitive Surface Modeling . In *Symposium on Geometry Processing*, pages 11–20, 2006.

[7] Mario Botsch, Mark Pauly, Martin Wicke, and Markus Gross. Adaptive space deformations based on rigid cells. *Computer Graphics Forum*, 26(3):339–347, 2007.

[8] Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović. Interactive skeleton-driven dynamic deformations. *ACM Trans. Graph.*, 21(3):586–593, July 2002.

[9] James Davis, Maneesh Agrawala, Erika Chuang, Zoran Popović, and David Salesin. A sketching interface for articulated figure animation. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 320–328, 2003.

[10] Tim Davis, WW Hager, and IS Duff. Suitesparse. *URL: faculty. cse. tamu. edu/davis/-suitesparse. html*, 2014.

[11] Olivier Dionne and Martin de Lasa. Geodesic voxel binding for production character meshes. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 173–180. ACM, 2013.

[12] Olivier Dionne and Martin de Lasa. Geodesic voxel binding for production character meshes. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '13, pages 173–180, New York, NY, USA, 2013. ACM.

[13] Sven Forstmann, Jun Ohya, Artus Krohn-Grimberghe, and Ryan McDougall. Deformation styles for spline-based skeletal animation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, pages 141–150, 2007.

[14] Dmitry O Gorodnichy, Shahzad Malik, and Gerhard Roth. Nouse 'use your nose as a mouse'-a new technology for hands-free games and interfaces. In *Proc. Intern. Conf. on Vision Interface (VI'2002)*, pages 354–361, 2002.

[15] Cindy Grimm and Pushkar Joshi. Just drawit: A 3d sketching system. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling*, SBIM '12, pages 121–130, 2012.

[16] Martin Guay, Marie-Paule Cani, and Rémi Ronfard. The line of action: An intuitive interface for expressive character posing. *ACM Trans. Graph.*, 32(6):205:1–205:8, November 2013.

[17] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[18] Takeo Igarashi and John F. Hughes. A suggestive interface for 3d drawing. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 173–181, 2001.

[19] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.*, 30(4):78:1–78:8, July 2011.

[20] Alec Jacobson, Zhigang Deng, Ladislav Kavan, and John P Lewis. Skinning: real-time shape deformation (full text not available). In *ACM SIGGRAPH 2014 Courses*, page 24. ACM, 2014.

[21] Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2016. http://libigl.github.io/libigl/.

[22] Alec Jacobson, Daniele Panozzo, C Schüller, O Diamanti, Q Zhou, N Pietroni, et al. libigl: A simple c++ geometry processing library, 2016, 2016.

[23] Alec Jacobson and Olga Sorkine. Stretchable and twistable bones for skeletal shape deformation. *ACM Trans. Graph.*, 30(6):165:1–165:8, December 2011.

[24] Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. Harmonic coordinates for character articulation. *ACM Trans. Graph.*, 26(3), July 2007.

[25] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. *ACM Transactions on Graphics*, 24(3):561–566, July 2005.

[26] Olga A. Karpenko and John F. Hughes. Smoothsketch: 3d free-form shapes from complex sketches. *ACM Trans. Graph.*, 25(3):589–598, July 2006.

[27] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics (TOG)*, 27(4):105, 2008.

[28] Youngihn Kho and Michael Garland. Sketching mesh deformations. *ACM Trans. Graph.*, 24(3):934–934, July 2005.

[29] Martin Komaritzan and Mario Botsch. Projective skinning. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(1):12, 2018.

[30] Torsten Langer and Hans-Peter Seidel. Higher order barycentric coordinates. In *Computer Graphics Forum*, volume 27, pages 459–466. Wiley Online Library, 2008.

[31] Binh Huy Le and Jessica K. Hodgins. Real-time skeletal skinning with optimized centers of rotation. *ACM Trans. Graph.*, 35(4):37:1–37:10, July 2016.

[32] Yong Jae Lee, C Lawrence Zitnick, and Michael F Cohen. Shadowdraw: real-time user guidance for freehand drawing. *ACM Transactions on Graphics (TOG)*, 30(4):1–10, 2011.

[33] John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172, 2000.

[34] Yaron Lipman, Olga Sorkine, Daniel Cohen-Or, David Levin, Christian Rossi, and Hans-Peter Seidel. Differential coordinates for interactive mesh editing. In *Proceedings Shape Modeling Applications, 2004.*, pages 181–190. IEEE, 2004.

[35] Ron MacCracken and Kenneth I. Joy. Free-form deformations with lattices of arbitrary topology. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 181–188, 1996.

[36] Nadia Magnenat-Thalmann, Richard Laperrire, and Daniel Thalmann. Joint-dependent local deformations for hand animation and object grasping. In *In Proceedings on Graphics interface'88*. Citeseer, 1988.

[37] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and mathematics III*, pages 35–57. Springer, 2003.

[38] Luke Olsen, Faramarz F. Samavati, Mario Costa Sousa, and Joaquim A. Jorge. Sketch-based modeling: A survey. *Computers & Graphics*, 33(1):85–103, February 2009.

[39] A Cengiz Öztireli, Ilya Baran, Tiberiu Popa, Boris Dalstein, Robert W Sumner, and Markus Gross. Differential blending for expressive sketch-based posing. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 155–164, 2013.

[40] A. Cengiz Öztireli, Ilya Baran, Tiberiu Popa, Boris Dalstein, Robert W. Sumner, and Markus Gross. Differential blending for expressive sketch-based posing. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '13, pages 155–164, 2013.

[41] Ulrich Pinkall and Konrad Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental mathematics*, 2(1):15–36, 1993.

[42] Valentin Roussellet, Nadine Abu Rumman, Florian Canezin, Nicolas Mellado, Ladislav Kavan, and Loïc Barthe. Dynamic implicit muscles for character skinning. *Computers & Graphics*, 77:227–239, 2018.

[43] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph.*, 20(4):151–160, August 1986.

[44] Xiaohan Shi, Kun Zhou, Yiying Tong, Mathieu Desbrun, Hujun Bao, and Baining Guo. Mesh puppetry: Cascading optimization of mesh deformation with inverse kinematics. *ACM Transactions on Graphics (TOG)*, 26(3):81, 2007.

[45] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2):11, 2015.

[46] Karan Singh and Eugene Fiume. Wires: A geometric deformation technique. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 405–414. ACM, 1998.

[47] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, SGP '07, pages 109–116, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[48] Olga Sorkine, Daniel Cohen-Or, Yaron Lipman, Marc Alexa, Christian Rössl, and H-P Seidel. Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry processing*, SGP '04, pages 175–184, 2004.

[49] Robert W. Sumner, Matthias Zwicker, Craig Gotsman, and Jovan Popović. Mesh-based inverse kinematics. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 488–495, 2005.

[50] Ivan E Sutherland. Sketchpad a man-machine graphical communication system. *Simulation*, 2(5):R–3, 1964.

[51] Matthew Thorne, David Burke, and Michiel van de Panne. Motion doodles: An interface for sketching character motion. *ACM Trans. Graph.*, 23(3):424–431, August 2004.

[52] Nicholas Toothman, Tyler Martin, and Michael Neff. Embodying digital creativity: Designing computer tools to support spontaneity and creative work in the digital arts. In Nicolás Salazar Sutil and Sita Popat, editors, *Digital Movement: Essays in Motion Technology and Performance*, chapter 14, pages 221–235. Palgrave Macmillan, 2014.

[53] Nicholas Toothman and Michael Neff. Spring rigs for skinning. In *Motion, Interaction and Games*, page 23. ACM, 2019.

[54] Nick Toothman and Michael Neff. Attachment-based character deformation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, page 18. ACM, 2017.

[55] Steve Tsang, Ravin Balakrishnan, Karan Singh, and Abhishek Ranjan. A suggestive interface for image guided 3d sketching. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 591–598, 2004.

[56] Rodolphe Vaillant, Loïc Barthe, Gaël Guennebaud, Marie-Paule Cani, Damien Rohmer, Brian Wyvill, Olivier Gourmel, and Mathias Paulin. Implicit skinning: Real-time skin deformation with contact modeling. *ACM Trans. Graph.*, 32(4):125:1–125:12, July 2013.

[57] Rodolphe Vaillant, Loïc Barthe, Gaël Guennebaud, Marie-Paule Cani, Damien Rohmer, Brian Wyvill, Olivier Gourmel, and Mathias Paulin. Implicit skinning: real-time skin deformation with contact modeling. *ACM Transactions on Graphics (TOG)*, 32(4):125, 2013.

[58] Rodolphe Vaillant, Gäel Guennebaud, Loïc Barthe, Brian Wyvill, and Marie-Paule Cani. Robust iso-surface tracking for interactive character skinning. *ACM Trans. Graph.*, 33(6):189:1–189:11, November 2014.

[59] Rodolphe Vaillant, Gäel Guennebaud, Loïc Barthe, Brian Wyvill, and Marie-Paule Cani. Robust iso-surface tracking for interactive character skinning. *ACM Transactions on Graphics (TOG)*, 33(6):189, 2014.

[60] Rich Wareham and Joan Lasenby. Bone glow: An improved method for the assignment of weights for mesh deformation. In *International Conference on Articulated Motion and Deformable Objects*, pages 63–71. Springer, 2008.

[61] Max Wertheimer. Experimentelle studien über das sehen von bewegung. *Zeitschrift fur Psychologie*, 61, 1912.

[62] Jane Wilhelms and Allen Van Gelder. Anatomically based modeling. In *SIGGRAPH*, volume 97, pages 173–180. Citeseer, 1997.

[63] Weiwei Xu, Jun Wang, KangKang Yin, Kun Zhou, Michiel Van De Panne, Falai Chen, and Baining Guo. Joint-aware manipulation of deformable models. *ACM Transactions on Graphics (TOG)*, 28(3):35, 2009.

[64] Johannes Zimmermann, Andrew Nealen, and Marc Alexa. Silsketch: Automated sketch-based editing of surface meshes. In *Proceedings of the 4th Eurographics Workshop on Sketch-based Interfaces and Modeling*, SBIM '07, pages 23–30, 2007.