Simplified Velocity Skinning

I. Introduction

In order to make characters in 2D more expressive and lively, animators use a number of standard animation principles such as squash, stretch and follow through. However, this effects are hard to reproduce in 3D.

The article [1] introduces a novel method - Velocity Skinning - which boosts character animation with secondary motion control. Unlike Linear Blend Skinning (LBS) and Dual Quaternions Skinning (DQS) which derive skin deformation from the current, static configuration of skeletal joints, Velocity Skinning considers also the influences of translational and rotational velocities along the skeletal structure. This insight allows to achieve squash & stretch and drag-like exaggeration of shapes.

This project is inspired by the article [1], intends to replicate its main ideas and test them on a simple cylinder shape.

II. Contribution

Our contribution to this method can be considered the method we introduce to calculate the velocity skinning weights. In this technique, we use a tree traversal algorithm to efficiently compute upward and downward propagated LBS weights. You can find the description of this method in the Algorithm2.

III. Velocity Skinning

In the following, we present a summary of the method described in [1].
Linear Blend Skinning can be summarized in the relation:

$$p^u = (\sum_{i \in bones} a_i^u T_i) r^u, \tag{1}$$

where we give the position of the vertex u based on the weight associated to the bone i, a_i^u and the current transform of the joint i, T_i. In this relation r^u is the rest pose of the vertex u.

Another way to write this equation would be to directly use the multiplication of T_i and r^u as p_i^u:

$$p^u = \sum_{i \in bones} a_i^u p_i^u. \tag{2}$$

Now, to induce extra movements as a result of velocity, we shall make the assumption that a vertex velocity can be described as a linear combination of a set of component velocities. These component velocities are the component velocities of the joints, and their influence on the velocity of a vertex can be controlled via using certain weights. Then we would be able to use this induced velocity to add an extra movement to the position of a vertex. More formally, instead of considering only the position induced by static pose, we add another term which is a linear combination of deformer functions of bones

$$d^u = \sum_i b_i^u \psi_i(b_i^u), \tag{3}$$

where d^u are velocity-based position displacements which are added through linear combination to the final procedural mesh deformation, b_i^u are the bone weights defined per vertex and ψ_i are customizable deformer functions which takes into account the geometry of bone i.

The LBS position p^u from 1 is displaced by d^u from 2, obtaining an additional mesh deformation.

As showed in [1], weights b_i^u can be computed from the existing skinning weights a_i^u.

In the following, we will explain the answers to the following three questions:

1) how to derive the weights associated to deformer functions of different bones with respect to a certain vertex?
2) how to obtain the velocity of joints given their frame's transform?
3) how to set the deformer function based on the desired effect?

A. Velocity Skinning Weights

The goal is to find velocity skinning weights that would be coherent with LBS weights. To calculate these weights, we shall make the following assumptions (while remembering that the bones are stores in a hierarchical order and that each bone has only one parent, but one bone may have several children) :

1) the relative velocity of a bone with respect to its immediate parent is transmitted to all its descendents.
2) the vertex "u inherits from the bone i the velocity that is transmitted to it by the same coefficient as LBS.

With the above two conditions, we can write the following relation:

$$v^u = \sum_{i \in bones} a_i^u \sum_{j \in A(i)} v_j^u \tag{4}$$

where $A(i)$ is the set of all the ancestors of the bone i and v_j^u is the relative velocity of the bone j with respect to its immediate parent induced on the vertex u.
We can simplify the equation 1 by noticing that in this relation v_j^u is counted with a coefficient a_i^u if and only if j is i or one of the ancestors of the bone i. Equivalently, v_j^u is counted for all its descendents with their corresponding LBS weights (a_i^u if $i \in D(j)$). This brings us to rewrite (1) as:

$$v^u = \sum_{i \in bones} v_i^u \sum_{j \in D(i)} a_j^u \tag{5}$$

We can write the second sum in the above equation as a new parameter:

$$a_i^{\leftarrow u} = \sum_{j \in D(i)} a_j^u$$

and hence:

$$v^u = \sum_{i \in bones} a_i^{\angle u} v_i^u \tag{6}$$

This relative velocity can be decomposed in rotational and transitional components, which would be useful for constructing the deformer function later. In fact, the transitional part of v_j^u is the same for all the vertices, and so we may neglect the u in the notation and write v_j. This however is not the case for the rotational part, and it depends on the position of the vertex u. More precisely, we have the following relation:

$$v_i^{R,u} = \omega_i \times (p^u - p_i)$$

where ω_i is the rotational velocity of the bone i and p_i is the position of the joint associated to this point.

B. Implementation

In the implementation, we should compute the weight $a_i^{\angle u}$ for each bone i and each vertex u. To this end, we should know:

1) what are the descendents of the bone i.
2) if a certain descendent of the bone i is connected to the vertex u in the rig structure.

To handle this task, we construct a tree structure G of the bones from the information stored in parentindex in skeleton. We define a node class for vertices in this graph that has the fields:

1) list of the children associated with the node
2) the only parent of the child

And we define the graph as struct of a buffer of node type, which has the size of number of bones, and the index of the node corresponding to the root of the tree.

We use the algorithm 1 to obtain this graph from the parentindex in $skeleton\_animation\_structure$ struct.

Algorithm 1: Constructing Bones Graph

```
N_joint = number of joints (bones)
Initialize buffer<node> Nodes of size N_joint

for each bone b do
    parent_b = parent_index[b]
    Nodes[b].parent = parent_b
    if parent_b is not −1 then
        Nodes[parent_b].children.add(b)
    end
end
root = 0
```

We use the algorithm 2 to calculate the velocity skinning weights.

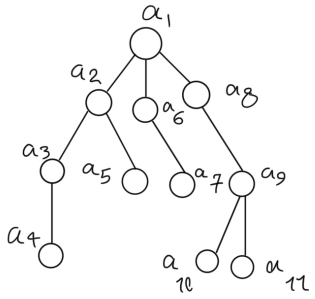


Fig. 1. Example of bone structure, with LBS weights corresponding to each bone, for a fixed vertex

Algorithm 2: Velocity Skinning Weights

```
N_vertex = number of vertices
N_joint = number of joints (bones)

Initialize v_skinning_weights[N_vertex][N_joint] = 0
for each vertex u do
    copy_graph = graph.copy() for each bone b do
        v_skinning_weights[u][b] += LBS[u][b]
    end
    Nodes = graph.Nodes
    root = Nodes[graph.root]
    current_node = root
    Ending = False
    while !Ending do
        if size(current_node.children) != 0 then
            current_node =
            Nodes[current_node.children.pop()]
        end
        else
            if current_node == root then
                Ending = true
            end
            else
                v_skinning_weights[u][current_node.parent]
                +=
                v_skinning_weights[u][current_node]
                current_node = current_node.parent
            end
        end
    end
end
```

1) initialize all the bones with their corresponding skinning weights (for the bone i, initialize it with a_i^u).
2) when moving from a child j to his parent i, add a_j^u to the node i.

You can see in the Figure 2 the weightings that we obtain after traversing the left-most branch of the graph.

C. Calculating the downward propagated weights

Downward propagated weights are defined for each pair of joint and vertex. Their expression for the joint j and the node

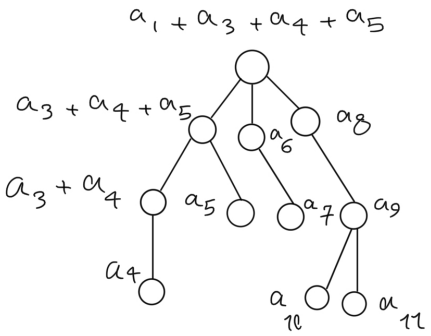


Fig. 2. Illustration of algorithm 2 on the left-most branch of the graph

u is:

$$a_i^{\rightarrow u} = \sum_{j \in D(i)} a_j^u$$

We calculate the downward propagated weights similarly to ho we computed the velocity skinning weights; we start by exploring the graph of bones from the root and at each step of type *parent to child*, we add the weights of the parent to the child.

D. Calculating bone velocity components

One of the main questions we treated in this project was how to obtain the relative velocity of bones with respect to their immediate parents, knowing only the frame's transform at each step.

Notation: Let the animation times be like: $t_1, t_2, t_3, ...$ and affine local transformation for the joint j at time t be $T_{t,j}$.

To illustrate the way we calculate the relative translational and rotational velocities of a joint with respect to his parent, let us consider the example of finding these two components at time t for the joint j.

$$T'_{t,j} = T_{t,j} T_{t-1,j}^{-1}$$

This latter is the transformation of the joint j at time t, from which we have to extract the translation and rotation. To obtain the translational velocity, we extract the translation from this transform and then divide it by Δt.

The calculation is however a bit more tedious in the case of rotational velocity; we extract the rotation_transform from the affine transformations $T_{t-1,j}$ and $T_{t,j}$, convert it to quaternions $Q_{t-1,j}$ and $Q_{t,j}$, perform $Q'_{t,j} = Q_{t,j} Q_{t-1,j}^{-1}$. We then extract from $Q'_{t,j}$ the corresponding axis and angle. The angular speed is then formulated as axis$*$angle$/\Delta t$.
In the implementation, we neglect the $/\Delta t$ because we may obtain diverging values.

E. Velocity-driven deformations

In this section, we describe how to compute the deformer function ψ_i. To increase the expressiveness of the method we choose a version of ψ_i that affects the components $v_i^{R,u}$ and $v_i^{T,u}$ of velocity differently. Furthermore, we want function ψ_i to combine various effects (squashy and floppy deformations), for this reason we consider a number of different deformer functions and the resulting deformer is

$$\psi(v_i^{R,u}, v_i^{T,u}) = \sum_{deform} \psi_{deform}(v_i^{R,u}, v_i^{T,u}). \tag{7}$$

After ψ_i are set, it remains for us to define the action of each deformer from their constituent velocity terms to finally get to compute d^u which is done in the next section.

IV. Squashy and Floppy Effects

Following the paper [1], in this project we concentrate on two well-known types of expressive effects: the squashy and the floppy ones. We define both effects using their translational and rotational components, ψ_T and ψ_R, through sums of the corresponding terms

$$\begin{aligned} \psi_{squashy}(v_i^{R,u}, v_i^{T,u}) &= \psi^R_{squashy}(v_i^{R,u}) + \psi^T_{squashy}(v_i^{T,u}), \\ \psi_{floppy}(v_i^{R,u}, v_i^{T,u}) &= \psi^R_{floppy}(v_i^{R,u}) + \psi^T_{floppy}(v_i^{T,u}). \end{aligned} \tag{8}$$

We are also able to change the scales of these effects using "squashiness" and "floppiness" coefficients, $k_{squashy}$ and k_{floppy}.

A. Squashy deformations

In this section, we introduce the squash effect which is inspired by the well known animation principle squash&stretch. Our effect aims to deform an object to produce a local elongation in the direction of motion. It is also essential for this deformation to preserve volume.

The squash effect is defined through controlled scalings.

For linear bone motions, the scaling is centered in c_i - the barycenter of the vertices in the bone's region of influence

$$\psi^T_{squashy}(v_i^{T,u}) = (RSR^T - Id)(p^u - c_i), \tag{9}$$

where R is a rotation matrix which maps the x-axis to the direction of the bone velocity v_i (it can be computer using Rodrigues' rotation formula - see appendix A), and S is the following anisotropic, volume-preserving scaling matrix:

$$\begin{bmatrix} 1+s & 0 & 0 \\ 0 & 1/\sqrt{(1+s)} & 0 \\ 0 & 0 & 1/\sqrt{(1+s)} \end{bmatrix}$$

where $s = k_{squashy} ||v_i^{T,u}||$.

For rotating bone motions, the scaling "resembles the effect of a centrifugal force generated by a spin". For each bone i, we calculate a medial axis as the axis connecting c_i with p_i. We also define Pr to be the operator projecting a point into this medial axis. As we want the squash effect not to produce

any elongation or shrinking along this axis, thus the formula for the displacement is

$$\psi^R_{squashy}(v_i^{R,u}) = (RSR^T - Id)(p^u - Pr(p^u)), \qquad (10)$$

where R is the rotation that maps the y-axis parallel to the medial axis, and maps the z-axis as close as possible to ω_i When ω_i is parallel to the medial axis, then R is undefined, and $\psi_{squashy}$ is a zero.

S is the anisotropic, volume-preserving scaling matrix defined as

$$\begin{bmatrix} 1+s & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1/(1+s) \end{bmatrix}$$

where $s = k_{squashy}||v_i^{R,u}||$.

B. Floppy deformations

As described in the paper [1], the floppy deformer is inspired by an effect described by classic animators as "The loose flesh [...] will move at a slower speed than the skeletal parts. This trailing behind in an action is sometimes called 'drag', and it gives a looseness and a solidity to the figure that is vital to the feeling of life". We observe this effect if think of how skin vertices appear to move with a delay with respects to the initiating bone movement.

For linear bone motions, we obtaine this effect by displacing the vertex in the opposite direction of the linear velocity induced by bone i:

$$\psi^T_{floppy}(v_i^{T,u}) = -k_{floppy} v_i^{T,u}. \qquad (11)$$

For rotating bone motions, the delay produces a bending around the current axis rotation of bone i in the opposite direction

$$\psi^R_{floppy}(v_i^{R,u}) = (R - Id)(p^u - Pr(p^u)), \qquad (12)$$

where Pr is the operator projecting p_u on the rotation axis, and R is the rotation matrix around the current axis rotation with angle $\theta = -k_{floppy}||v_i^{R,u}||$.

V. Results and Conclusions

In this project, we implimented Velocity Skining and tested it on a cylinder shape. The floppy implementation works as it should and does not need the mesh to be provided in order to compute the added deformation. However, our implementation of squashy deformation faces a divergence problem: after calculating the Voronoi areas of vertices, we try to calculate the position of centroids, but they are NaN. We tried to use different ideas to fix it, such as supposing the equality between areas and multiplying by different scaling numbers; however, unfortunately, we did not obtain results on cylinder for this deformation.

References

[1] D. Rohmer, M. Tarini, N. Kalyanasundaram, F. Moshfeghifar, M.-P. Cani, and V. Zordan, "Velocity skinning for real-time stylized skeletal animation," in *Computer Graphics Forum*, vol. 40, pp. 549–561, Wiley Online Library, 2021.

Appendix A
Rodrigues' rotation formula

The rotation matrix which maps the vector u to the direction of vector v is given by

$$R = Id + sin\theta K + (1 - cos\theta)K^2, \qquad (13)$$

where $\theta = arccos\left(\frac{u \cdot v}{||u||*||v||}\right)$, K is the following matrix

$$\begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix}$$

and vector k is equal to $\frac{u \times v}{||u \times v||}$.

The matrix R can also be interpreted as the rotation matrix through an angle θ counterclockwise about the k-axis.

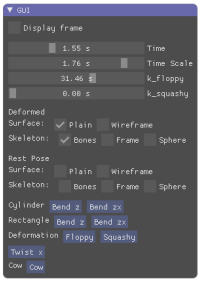
Appendix B
Centroid of Joints

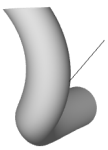
In the squashy displacement, we use the centroid of joints that are calculated as follows for the joint j:

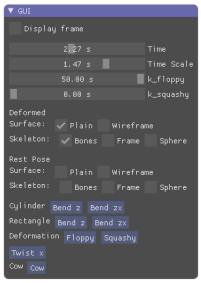
$$c_i = \sum_u a_i^{\rightharpoonup u} m_u p^u \Big/ \sum_u a_i^{\rightharpoonup u} m_u$$

For the calculation of the area of Voronoi cells, we use the provided mesh (not for cylinder) and the associated triangles and the Heron's formula.

Appendix C
Some visualisations






GUI
Display frame
2.27 s
Time
1.47 s
Time Scale
50.80 s
k_floppy
0.00 s
k_squashy
Deformed
Surface:
Plain
Wireframe
Skeleton:
Bones
Frame
Sphere
Rest Pose
Surface:
Plain
Wireframe
Skeleton:
Bones
Frame
Sphere
Cylinder
Bend z
Bend zx
Rectangle
Bend z
Bend zx
Deformation
Floppy
Squashy
Twist x
Cow
Cow