Visualization of Movement in Multiscale

Brien Colwell

EECS / UC Berkeley

Figure 1: A key-framed movement through multiscale, enhanced with the visual encodings described in this paper

ABSTRACT

This paper will explore a number of visual encodings that can be layered over (x, y, scale, rotation) movement through multiscale. The visual encodings are designed to enhance viewers’ understanding of movement in between key frames. Since none of the visual encodings in this paper conveys all aspects of movement, to give the viewer a complete sense of movement, multiple visual encodings must be shown in concert. This paper will explore a rule-based method for selecting which visual encodings to show.

Keywords: visualization, multiscale, movement, keyframe

1 INTRODUCTION

With the evolution of multiscale worlds, people may want a way to share their experiences of these worlds with others. Two example multiscale worlds are multiscale poetry and fractal images. Pad++ showed that the way in which words move across the screen in a multiscale poem enhances their interpretation. In the poem Searching, the word “searching” meanders, in a squiggly path, across the screen. Given a multiscale poem, people may want to define their own unique way to move the words, in a creative surge somewhat like a doodle on a piece of paper. Flying through a fractal is both visually exciting and like a story, and people may want to tell their own fractal stories.

This is typically done through code, which is difficult to mentally map to any movement through multiscale. The goal of the software described in this paper is to:

- Not require people to code
Make movement perceptible. That is, bring it out of a just a mental construct and visually put it on the screen

Making movement perceptible is more than just playing it back, as is done in most animation systems. It is about creating a visual vocabulary that highlights the salient features of movement, allowing people to both:

- Form overviews of movement
- See how changes to specifications affect movement
- (Important for children: collaborate on the creation of movement through visual analogies – e.g. “two loops” instead of “2 · π”)

Although the second point seems somewhat paradoxical – how can the designer not know what effects his changes will have?! – consider that most movement (including the movement produced by this software) is created using key frame animation, in which, by turning knobs that correspond to a general feeling of the movement rather than a strict specification, the designer is able to only indirectly control of the movement itself.

This paper will explore one such vocabulary, a set of visual encodings that can be layered over \((x, y, \text{scale}, \text{rotation}, \text{time})\) movement through multiscale. These visual encodings in this paper are designed to complement each other. That is, instead of using one visual encoding to display all aspects of movement, one visual encoding is used to display a small set of aspects of interest. This follows from a general rule that the more information and encoding attempts to display, the less effective or more cluttered it will be. Each encoding is selectively displayed in an attempt to convey the most information possible without overloading the image.

Because the visual encodings in this paper are not critical to understanding movement (after all, one can always revert to playback), one of the core design philosophies used to create them was: Too little is better than too much.

## 2 Related Work

There has been some related work on visualizing movement, most notably the motion paths in Maya and the physics/car paths in Squeak [2][1]. Both of these use a centerline to convey position and some sort of bubble plot to represent speed (or, different name descriptors). Both of these use a centerline to convey position and some sort of bubble plot to represent speed (or, different name descriptors). However, this was not researched as thoroughly as it should have been by the author, and consequently did not (directly) influence the software described in this paper.

### 2.1 Rhapson

Rhapson) are common place. However, there are two topics that have been further explored because they are critical to the visual encodings. They are multiscale length and how to specify time, and provide a key framing system (including the ability to save/load created movements)

Although the all the details of these libraries will not be discussed here, a few details that pertain to the visual encodings are:

- Each curve is stored in a triple, \((\text{curve}, (\text{transform}_0, \ldots, \text{transform}_n), (\text{pacer}_0, \ldots, \text{pacer}_n))\)

where ...

\[
\text{curve} \in \{ \text{cubic spline}, \text{linear} \}
\]

\[
\text{transform}_n \in \{ \text{orthogonal extension} \}
\]

\[
\text{pacer}_n \in \{ \text{slow in slow out, accelerated out} \}
\]

Each of cubic spline, orthogonal extension, slow in slow out, etc. is a called a descriptor, and new descriptors can easily be added to give new expressiveness to movements.

A curve is created by the following process: first, insemination (the \(\text{curve}\) descriptor); then transformation to variants of the original curve (the \(\text{transform}_n\) descriptors); and finally time distortion to achieve various artistic effects (the \(\text{pacer}_n\) descriptors).

- A movement is a collection of curves, \((\text{curve}_0, \ldots, \text{curve}_n)(u), \text{for example} \,(x, y, \text{scale}, \text{rotation})(u)\). New curves can be added to the movement. For example, as we will see later, \(f r a m e s(u)\) was added to give timing information.

- The translation of a specific type of movement, a spatial path \((x, y, \text{scale})(u)\), to a visual representation is handled automatically, using an efficient sampling method

Although this paper is at a higher level than these details, these details enable complex visual encodings to be easily specified and rendered. Additionally, the regularity of the curve triples allows movements to be uniformly saved, which opens the doors to exporting them into other programs.

Who are the Key Frame Kids? Us. Let’s rock it.

## 4 Mathematical Foundation

Most of the math used by the software (various integration – e.g. Newton-Cotes – and root finding techniques – e.g. Newton-Rhapson) are common place. However, there are two topics that have been further explored because they are critical to the visual encodings. They are multiscale length and how to specify time, and they will be discussed in the following sections.

### 4.1 Multiscale Length

Even though multiscale has been around since the early 90s, multiscale is new. For example, the first stab at parameterizing general multiscale curves for smooth playback was made in 2004 [4]. It is still open how to approach this problem (and might always be) because one parameterization may be optimal under its metric, but the metric may not be valid for all requirements. This section will explore the parameterization used by this software. It should be noted that the mathematics here were developed for one reason: (in the words of Jeff) “to engender a positive user experience”. No claims to optimality are made. The mathematics here are used because they give decent results for our requirements, whereas the scant number of alternatives do not.

\[1\]

More time is needed with the van Wijk and Nuij reparameterization, but it seems to address different requirements than those of this program.
The reparameterization here is an arc length reparameterization using a new definition of length in multiscale, which is a sum of separate pan and zoom parts. The length defined here has the property that one unit contains the same amount of visual information, based on the metric in [3].

Why is length important? It is needed to:

- Visually sample multiscale curves
- Determine playback speed
- Reparameterize multiscale curves

Given a coordinate system of \((\text{baseX}, \text{baseY}, \text{scale})\), where \((\text{baseX}, \text{baseY})\) is \((x, y)\) on the plane where \(\text{scale} = 1^2\), we can attempt to apply usual definitions of length, namely 2D and 2D arc length.

To help us see why they break down, we can define the focus scale \((\text{scale}_{\text{focus}})\) to be the scale at which the camera is currently focused. For normal viewing, \(\text{scale}_{\text{focus}} = 1\). By zooming in, the viewer decreases the current focus scale, and vice-versa for zooming out.

2D arc length fails because the length is not dependent on the focus scale. For example, if \(\text{scale}_{\text{focus}} = 32\) and we have a line that is visually 128 pixels long — say, from pixel \((0, 0)\) to \((128, 0)\) on the screen — it is actually 128 \cdot 32 pixels long on the base plane — from coordinate \((0, 0)\) to \((128, 32)\). Thus the 2D arc does not tell us how long the line appears when in focus.

Although 2D arc length with corrected coordinates, defined as the 2D arc length of the curve \((x(v)/\text{scale}(v), y(v)/\text{scale}(v))\), closely approximates the perceived length when in focus at \(\text{scale} = 32\), it severely overestimates the perceived length when in focus at \(\text{scale} = 1/32\). To see this, take one additive term in the derivative, \(x'(v)/\text{scale}(v) \neq x(v)/\text{scale}(v)/\text{scale}(v)\). Note that this term diverges as \(\text{scale} \rightarrow 0\), which implies the length diverges as well.

There is an obvious need for another definition of multiscale length, given our (so far implicitly stated) requirement that:

Two curves should be the same length if they look the same length when in focus.

This is loosely defined (for example, one may wonder when a multiscale curve is "in focus", since it is by definition a composite of scales), but it is intended to imply one property:

The length of a curve \((\text{baseX}, \text{baseY}, \text{scale})\) should not change under the transform \((\text{baseX}', \text{baseY}', \text{scale}') = \alpha (\text{baseX}, \text{baseY}, \text{scale})\).

That is, the length of a curve should not change by pushing it to a higher scale plane, or down to a lower scale plane, because it will be perceived the same length when the camera is focused on it. Beyond this requirement, scaling the length to a meaningful unit is another consideration.

Now we have to find a metric with this property. Here is one, which will be explained below:

\[
\begin{align*}
\text{dl}_{\text{multiscale}} &= \text{dl}_{\text{pan}} + \text{dl}_{\text{zoom}} \\
&= \frac{f_{\text{base}}(v) \cdot w(v)}{\text{scale}(v)} dv + A \cdot Q \left[ \frac{\text{scale}(v)}{\text{scale}(v)} \right] dv \\
\end{align*}
\]

\[
\begin{align*}
w(v) &= \int_{0}^{v} \text{scale}(x) dx \\
&= \int_{0}^{v} \text{scale}(x) dx
\end{align*}
\]

In this case, \(A = 1/\ln(2)\).

Some notation:

- \(v \in [0, 1]\) and is the parameter used by all curves. Under this definition, it is the percentage of pan length (that is, \(\text{length}_{\text{pan}}(v) = \alpha \cdot v\)).
- \(w \in [0, 1]\) and is the percentage of 2D arc length (that is, \(\text{length}_{\text{base}}(w) = \beta \cdot w\)). Note: this parameter is typically denoted \(s\) (for arc length), but not in this paper to be consistent with the code.

- \(u \in [0, 1]\) is the raw parameter in which only \(\text{baseX}\) and \(\text{baseY}\) are specified. Since the curves are free to distribute the parameter however they like, \(u\) does not have any nice interpretation.

Note: The final parameterization takes \(v\) to \(u\) via a composition of \(u(w)\) and \(w(v)\) — that is, \(u(w(v))\). \(u(w)\) is the inverse of the 2D arc length, which is currently computed with a linearly interpolated LUT.

To derive the two equations above, three definitions were made:

\[
\begin{align*}
\text{l}_{\text{multiscale}}(v) &:= \text{l}_{\text{pan}}(v) + \text{l}_{\text{zoom}}(v) \\
\text{l}_{\text{zoom}}(v) &:= 1D\text{arc\_length}[Q \cdot \log_2(\text{scale}(v)), 0, v] \\
\text{dl}_{\text{pan}} &:= \frac{\text{dl}_{\text{base}}}{\text{scale}(v)}
\end{align*}
\]

The definition of \(\text{length}_{\text{zoom}}(v)\) follows from the intuition that zooming has a logarithmic distance [3], whereas panning has a linear distance. \(Q\) can be chosen to equate the units of pan [pixels] and scale length [scale magnitudes]. That is, \(Q\) relates how much information one magnitude of scale contains compared to one pixel of pan. A first-guess at \(Q\) is 3, although this can be refined from user tests similar to those in [4].

The definition of \(\text{dl}_{\text{pan}}\) is perhaps most interesting. It follows from the question:

If the view window focuses on each point of a multiscale curve as it follows the curve to its end, what length will the viewer perceive?

The length in question is entirely pan (translation). The scenario considers a curve as an infinite sequence of small (position) displacements, each on a scale plane, and asks, how long does each displacement appear to the viewer when in focus. Given our coordinate system, the answer is \(\text{length}_{\text{pan}}\).

The final piece is to require the parameters to match up at the ends, \(\int_{0}^{\text{l}_{\text{zoom}}} dv = 1, w(0) = 0, \text{and } w(1) = 1\).

Curious users can switch between definitions of length (and \(v\)) in the software by pressing ENTER, then typing one of the following commands (followed by another ENTER): \(\text{p\_pan}\) (described here), \(\text{p\_total}\), \(\text{p\_vwn}\) (van Wijk and Nuij), or \(\text{p\_raw}\).

### 4.2 How To Specify Time

One of the goals of the software is to allow easy manipulation of the motion. It is suspected that most users care more about the time.
taken to travel between two visual elements than the time taken to travel between two key frames. That is, is is suspected that most users would say the time between two key frames is a function of length, not absolute. This section will explore a definition of time along these lines. This is important because, if we are to visualize time, we need to know time!

Note: from here on, we will talk about time in terms on number of frames (or simply “frames”), since the two are related by a playback constant as \( \text{time}(v) = \text{time_per_frame} \cdot \text{frames}(v) \).

The number of frames is specified by a continuous frame density along the curve, \( \rho_{\text{frame}}(v) \). This is specified at the keys and interpolated (by default using a spline to smooth the time transitions) just like other values, such as \( x \), \( y \), rotation, and scale.

Given an appropriate hang length (the be defined later) function \( h_{\text{hang}}(v) \), the number of frames can be retrieved from the density as:

\[
\text{frames}(v) := \int_0^v \rho_{\text{frame}}(v) \cdot h_{\text{hang}}(x) \, dx
\]

What is hang length? Our normal definition of multiscale length has the property that \( \ell(v) \) is largest in the places where the movement is fastest – that is, the places where the least number of frames are placed. This obviously doesn’t match with the above equation for \( \text{frames}(v) \), which leads us to define the derivative of hang length, \( \ell_{\text{hang}}(v) \), as being largest where the most frames are placed (the opposite behavior of \( \ell(v) \)).

One definition of hang length follows from a simple linear transform \( h_{\text{hang}}(v) = \beta - \alpha \cdot \ell(v) \) under the following three constraints.

\[
\begin{align*}
\ell_{\text{hang}}(v) & \geq \gamma \\
h_{\text{hang}}(0) & = 0 \\
h_{\text{hang}}(1) & \geq \phi
\end{align*}
\]

The details of the derivation won’t be covered here, but by making some reasonable assumptions (and tightening the inequalities), we end up with the following values for \( \alpha \) and \( \beta \).

\[
\begin{align*}
\gamma &= \min \ell(v) \\
\phi &= \ell(1) \\
c_0 &= \gamma + \max \ell(v) \\
c_1 &= \phi \\
\alpha &= \frac{\phi - \gamma}{c_0 - c_1} \\
\beta &= \frac{\phi \cdot c_0 - \gamma \cdot c_1}{c_0 - c_1}
\end{align*}
\]

Because we assumed two tight linear equations in this derivation, which can sometimes be singular (meaning \( h_{\text{hang}}(v) \) may dip negative), an overestimate of hang length that is guaranteed to be monotonic can be generated as:

\[
h_{\text{hang, mono}}(v) := \int_0^v \ell_{\text{hang}}(x) \, dx
\]

5 Visual Encodings: Tying It Together

This section will give an overview on the visual encodings are tied together. The following section, “Visual Encodings”, will explore the visual encodings themselves.

The visual encodings in this paper are intended to overlay the HALCYON GLAZE key frame system, which employs its own visual encodings, namely a spine curve and various width and color distortions. Indeed they are good encodings; however, as with all encodings, they have degenerate cases and cases where more information could be displayed. For example, when two key frames overlap each other (meaning their centers are very close), the width and color encodings on the spine curve are lost. Further, there are aspects of the movement that the key frame system makes no attempt to convey, namely rotation and time.

Each visual encoding in this paper is (hopefully you agree) effective at conveying some aspect of the movement, under some situation. Just as with the key frame encodings above, there are two problems:

- They all have degenerate cases
- They cannot all be shown at once

To approach this, the software uses a rule engine to selectively switch on and off appropriate visual encodings. Just like viewer must visually inspect the image, the rule engine needs senses to inspect the movements to decide which encodings to show, and we give it three: rotation swing\(^3\), log(scale) swing, and pan length (which is \( \alpha \), as defined in the multiscale length section).

At some recurring time step \( T \), for each movement segment (which is the movement between each pair of consecutive key frames), the rule engine inspects its senses, which together form a sense point, and determines which encodings to show. It makes the decision by consulting sense maps like those shown in Figure 2.

If an encoding surrounds the current sense point in all maps, then it is shown. Note that encodings that are not shown on a sense map are assumed to surround the entire map.

Here’s an example of how to read Figure 2. If you want to know when the winding encoding is shown, you would first locate the (two) off-yellow stripes that correspond to winding. On the top map, the stripe covers all sections above \( 2 \cdot \pi \) rotation swing. On the bottom map, it covers all sections above 128 pan length. Thus, winding will be shown only when the rotation swing is above \( 2 \cdot \pi \) and the pan length is above 128.

A good set of sense maps should show encodings in the situations in which they are most effective (or more effective than the rest). The maps in Figure 2 are those used by the software, and their goodness can be evaluated by the curious reader (run the demo!). Sense maps do not include feedback loops to evaluate the effectiveness of the current set of visual encodings. Although the speed has not been tested with an evaluation feedback, the lack of feedback is done to increase the speed of the engine, which is critical to maintain interactivity.

6 Visual Encodings

Although usually each encoding is strongest in conveying one aspect, each cannot be categorized as giving information on just one aspect of the movement\(^4\). Consider, for example, the jitter twirl (preview enumeration). Although it is strongest in conveying rotation, it also gives some sense of scale and time. Please see the “Future Work” section for a planned survey at rating the effectiveness of each encoding across a variety of categories.

This section will detail seven visual encodings. For each, the desired effect, construction (mathematical and algorithmic), and

\(^3\)Swing is defined (as in circuits) to be the max and min value along the entire curve. Note that the extrema do not have to occur at the keys, although this will be the case for monotonic interpolating functions.

\(^4\)Note that position is not considered an aspect in need of encoding. Since the position on the screen corresponds to the position of the frame, the spine does an adequate job.
strengths and weaknesses will be discussed. All sections will assume a movement \((baseX, baseY, scale, rotation, frames)[v]\). Additionally, all sections assume a definition of the radius of the view window as \(r_{view} = (width_{view}^2 + height_{view}^2)^{1/2}\).

Before diving into our seven visual encodings, it should be noted that the ZUI in which they sit imposes its own set of encodings, which mix into the final image. These are all defined on a quantity \(closeness = k \cdot scale_e / scale_{focus}\), which is a measure of the closeness of some visual element at scale \(scale_e\) to the focus scale.

- The width of lines is proportional to \(closeness\)
- The brightness of color is inversely proportional to \(closeness\)

### 6.1 Voral Naz

Voral Naz is intended to show how the magnitude of scale changes, or roughly how many times the scale doubles from the start of the movement to the end. It works by simply enumerating frames at evenly spaced intervals of scale magnitude.

To construct Voral Naz, mark the movement whenever the quantity \(q = a \cdot \log_{a}(scale(v)) / \log_{a}(scale(0))\) (currently \(a = 1\)) crosses an integer boundary. That is, whenever \(floor(q) = ceil(q)\). At each mark, render a frame.

**STRENGTHS**

- When the frames are composed in each other, the magnitude of scale change is easily perceived

**WEAKNESSES**

- When the frames are not composed, the sense of magnitude of the scale change is not as strong

- However, when the frames are composed, they sometimes occlude each other
- The frames are not spaced well (a better choice of \(a\) would be a start). That is, dragging up a key frame does not alter the number of frames, or give any indication of scale increase, until an integer boundary (from the construction) is crossed

### 6.2 Envelope

Envelope is intended to show roughly what image will be in the frame as it scales and translates. With the movement remains at a constant scale, it is easier (although perhaps still not trivial) to mentally fill in the envelope – the approximate maximum boundary – of the frame. However, when the movement moves up and down in scale, even following just a linear scale, it becomes more difficult to fill in the envelope. For more complex scales, it becomes near impossible.

To construct the envelope, render the curve \((baseX, baseY, scale)\):

\[
\begin{align*}
\text{baseX}(v) + r_{view} \cdot scale(v) \cdot \cos(\theta), \\
\text{baseY}(v) + r_{view} \cdot scale(v) \cdot \sin(\theta), \\
scale(v)
\end{align*}
\]

\(\theta\) is chosen as the angle between the start and end of the movement, shifted by \(\pi/2\).

**STRENGTHS**

- Since it is just a line, there is little visual intrusion; however, the effect is noticable and helpful as a first-estimate (to the author). Typically a preview (press \(P\) in the software), when combined with the envelope, is an effective "focus plus context"

**WEAKNESSES**

- Envelope may include image that is not actually under the frame, which is misleading, especially with rotation
6.3 Spacing

Spacing is intended to show the parameter spacing of the movement, which is a strong indication of the time spent at each point of the movement. This encoding can be seen in a number of forms elsewhere, from the motion path in Maya to the physics plots in Squeak.

The construction is simple. Take \( n = N(length) \) position samples, evenly spaced, along the movement. If the position is non-uniform, due to pacing, it will be evident from the distribution of the position samples. At every position sample, place a dot.

**STRENGTHS**

- The dots do not clutter the spine or interfere with the other encodings, with the possible exception of the worm
- Reading the dots is strangely intuitive (see below)

**WEAKNESSES**

- Although intuitive, if the viewer has to describe why it works, it gets more abstract, in that it requires the viewer to have some concept of acceleration. This point has not been tested and may not be valid. Additionally, the viewer may be able to readily deal with acceleration in this form anyway (this is what happens with physics in Squeak – children get it)

6.4 Jitter Twirl (Preview Enumeration)

This was the favorite in an informal survey conducted with various people. Asked how they would represent a rotation between two points, they simply enumerated the frames, being careful to not overlap them along the spine. This encoding attempts to automate that process, minimizing occlusion as much as possible.

The current construction defines this as a recursive minimization problem, with the penalty function given below. The recursive part is that the movement is broken in half, and the minimum penalty is found in the neighborhood of the break point. The two halves are then recursively broken. At each break point, a frame is displayed.

\[
\text{penalty}(v) := a \cdot \max \left( \text{overlap} \left( \text{projection interval}(v), I \right) \right) + b \cdot \text{multiple} \left( \text{rotation}(v), rm \right) + c \cdot \text{multiple} \left( \text{log} \left( \text{swing}(v)/\text{swing}(0) \right), sm \right)
\]

The overlap function is larger as two intervals overlap more (currently defined to be a triangle, where the peak is full overlap). The projection interval function finds the interval on the movement that the frame at \( v \) covers. The set \( I \) is the set of projection intervals of all the frames that have been selected for display so far in the recursive algorithm. The multiple\((\cdot, k)\) function finds the minimum distance of the argument to an integer multiple of \( k \).

\((a, b, c)\) can be chosen to give precedence to any of the three penalty terms. The software is currently set to give precedence to non-occlusion, and secondary precedence to snapping rotation to multiples of a fundamental rotation, with \((a, b, c) = (8, 4, 0)\). Currently, \( rm = \pi/6 \) and \( sm = 1 \), but these are more personal tastes than anything scientific.

Note: this is implemented in code in PreviewEnumerationPlayground.java for the reader’s inspection.

**STRENGTHS**

- Frames shown at constant rotations, which means their orientations stay constant as the path is manipulated
- Does a decent job of minimizing occlusion

**WEAKNESSES**

- Computationally expensive (the most computing cost of any encoding in this paper)
- Periodicity of enumeration a problem. That is, when the frame rotates by over \( 2\pi \), it is more difficult to tell what is happening
- When the pan length is short (that is, when the centers of the keys are close), no frames will be shown due to occlusion with the keys
6.5 Corner

The corner path is intended to both give a sense of rotation above one period and a sense of the envelope. The idea is to just trace a single corner of the frame across the movement. The corner, by definition, rolls and scales with the movement of the frame. Previous implementations attempted to choose the corner to minimize occlusion with the key frame, which constantly switched the corner depending on the orientations of the key frames. Although the responsiveness of this setup was neat, it was found that switching the corner is somewhat disorienting. The current implementation always uses the top-left corner.

To construct, simply render the curve \((baseX, baseY, scale)\):

\[
\begin{align*}
(baseX(v) + r\cdot\cos(rotation(v)), \\
(baseY(v) + r\cdot\sin(rotation(v)), \\
scale(v))
\end{align*}
\]

**STRENGTHS**

- Once used to the corner, viewers can readily pick out one full period of rotation by looking for space in between loops
- Reduces to a spiral when the pan length is small, which acts as a nice in-place envelope

**WEAKNESSES**

- Mapping a corner to its corresponding frame requires mentally finding the center point of the frame on the spine, which is (somewhat) difficult. This task may be aided by the knowledge that the frame (sometimes) moves at a constant rate along the spine

6.6 Winding

Winding is intended to show key frame rotation above one period. The idea is take a virtual straw, attach it to the center of the key frame, and wind it about the center into a spiral.

To construct it, simply render the curve \((baseX, baseY, scale)\):

\[
\begin{align*}
(baseX + r\cdot\cos(rotation\cdot v), \\
(baseY + r\cdot\sin(rotation\cdot v), \\
\scale)
\end{align*}
\]

Since winding occupies much visual space inside the key frame, the width is diminished near the center of the frame. Additionally, the color is made lighter near the center.

**STRENGTHS**

- Becomes intuitive with animation. That is, when the viewer watches it wind up with the rotation of the frame

**WEAKNESSES**

- Visually dense in the center of the key frame. Even though the width variation attempts to counter this, winding still occludes objects under the key frame.

6.7 Worm

Worm is intended to show how much time is spent near each moment of the curve. The idea is to keep the total area in an amorphous membrane constant, proportional to the total time of the movement, and deform it at each moment so that its width is proportional to the time spent near that moment.

To construct the worm, takes \(frames(v)\) (from the mathematics section above; note \(frames\) is proportional to time), and compute \(frames'(v)\) (the derivative). Note the total area under \(frames'(v)\)
is proportional to the total time. Vary the width as \( \text{width}(v) = k \cdot \text{framed}(v) \) for an appropriate \( k \) (currently 0.85/16).

**STRENGTHS**

- Fun to deform. It gives some strange sense of satisfaction seeing blue pixels flow around in response to changing pacing parameters.
- The width variation gives a realistic Venturi-like effect (where fast movement is “sucked in” toward the center), which may help viewers draw analogies

**WEAKNESSES**

- The variation in width due to scale (i.e. the visual encodings imposed by the ZUI) may be misinterpreted as variation in time
- Visually dense

Figure 9: Worm showing two movement segments with slow in slow out pacing

**7 USER FEEDBACK**

Limited user feedback, mostly from roommates, has been collected over the development of this software. Positive comments were “interaction is good”, “easy to learn”, and “fun to use”. However, not much feedback was collected on encodings themselves, although – gauged by exclamations of “cool” – the worm seems to be the most sensational, followed closely by the corner.

**8 DISCUSSION**

This paper presented a number of visual encodings and the mathematics behind them. Even though each encoding has its weaknesses, it is believed each offers a positive net gain towards understanding multiscale movement. These encodings are aimed at general curves in multiscale, which offer more interesting possibilities than linear curves, but also more challenges. The HALCYON GLAZE libraries were used to approach these challenges in a general, reusable way.

Even though the software used in this paper uses a custom point-based ZUI for efficiency, its aim is not to replace any of the current ZUI toolkits, specifically (the very nice) Piccolo from UMD. Instead, it aims to complement Piccolo by allowing people to design movements and then import them into Piccolo (using a custom \( PActivity \) object). Although the integration is rather primitive right now, the author plans to continue to improve it, both in the playback and design modes.

**9 FUTURE WORK**

The biggest work for the future is getting feedback from people and refining the software to be more useful to them. A multi-resolution fractal video maker has been developed using these visual encodings, and it is hoped the content (fractal images) will draw people into the software and allow us to collect useful feedback.

One planned survey is to rank each encoding across three categories: shows scale well, shows rotation well, shows time well. The format of these questions still needs work, and the way in which the survey is presented still needs some consideration.

There are also a number of performance enhancements and features to add to the software (and HALCYON GLAZE), but no major changes to the basic structure are planned. One nice feature would be a way for the viewer to override the rule map with their own preferences. Another would be to get translucency working. Although it is supported by Java2D, it is slow for this software. Further investigation is needed. Previous versions used translucency to fade in and out visual encodings, which was very nice.

**10 CONCLUSION**

A demo of this software and the HALCYON GLAZE libraries are available for download at [http://www.ocf.berkeley.edu/~bcolwell/halcyon_glaze](http://www.ocf.berkeley.edu/~bcolwell/halcyon_glaze) (BSD license). The author plans to continue to expand and improve the code.

Enormous Thanks to Jeff Heer for all the great ideas, innumerous suggestions, and guidance. I’ll keep working on this beast!

Also, thanks to Prince and The Revolution; 8-bit musicians Lo-Bat, Twilight Electric, and Mesu Kasumai; Boards of Canada, the Beatles, and the rest of the musicians who rock out of my stereo into the night.

Figure 10: Nostalgia: One of the first composite curves generated and rendered with HALCYON GLAZE, using a then-advanced “gap skipping” technique to not draw lines between the letters

**REFERENCES**


5Note: Jeff should be listed as a co-author on any future version this paper.