Drawing Transforms: A Unifying Interaction Primitive to Procedurally Manipulate Graphics across Style, Space, and Time

Sonia Hashim
University of California
Santa Barbara, USA

Tobias Höllerer
University of California
Santa Barbara, USA

Jennifer Jacobs
University of California
Santa Barbara, USA

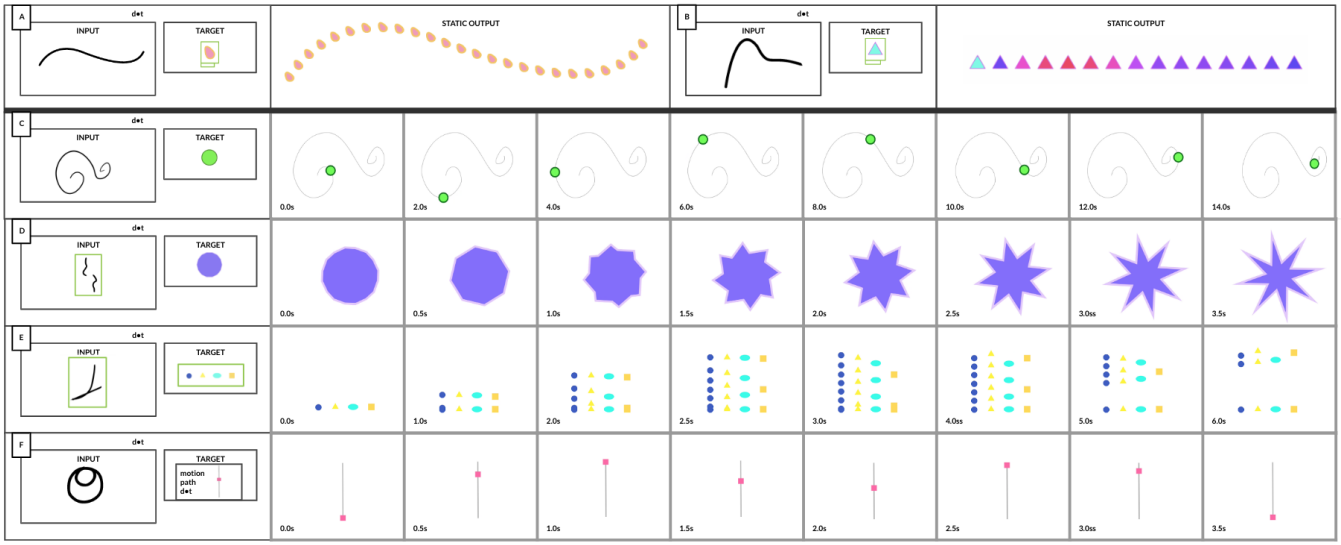

Figure 1: A drawing transform (DT) is a flexible primitive that uses vector geometry to define a procedural transformation. Here we show the breadth of ways DTs can transform artwork: (a) position artwork, (b) update a hue distribution, (c) translate a shape, (d), animate vertex geometry, (e) interpolate duplication rates, and (f) loop and modify the easing of an animation.

ABSTRACT

Procedural functionality enables visual creators to rapidly edit, explore alternatives, and fine-tune artwork in many domains including illustration, motion graphics, and interactive animation. Symbolic procedural tools, such as textual programming languages, are highly expressive but often limit directly manipulating concrete artwork; whereas direct manipulation tools support some procedural expression but limit creators to pre-defined behaviors and inputs. Inspired by visions of using geometric input to create procedural relationships, we identify an opportunity to use vector geometry from artwork to specify expressive user-defined procedural functions. We present Drawing Transforms (DTs), a technique that enables the use of any drawing to procedurally transform the stylistic, spatial, and temporal properties of target artwork. We apply DTs in a prototype motion graphics system to author continuous and discrete transformations, modify multiple elements in a composition simultaneously, create animations, and control fine-grained procedural instantiation. We discuss how DTs can unify procedural authoring through direct manipulation across visual media domains.

CCS CONCEPTS

• **Human-centered computing** → **Human computer interaction (HCI)**; **Graphical user interfaces**; • **Applied computing** → **Fine arts**; **Media arts**.

KEYWORDS

creativity support tools, direct manipulation, procedural art

ACM Reference Format:
Sonia Hashim, Tobias Höllerer, and Jennifer Jacobs. 2023. Drawing Transforms: A Unifying Interaction Primitive to Procedurally Manipulate Graphics across Style, Space, and Time. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3544548.3580642

CHI '23, Hamburg, Germany,

ACM ISBN 978-1-4503-9421-5/23/04.
https://doi.org/10.1145/3544548.3580642

1 INTRODUCTION

Procedural authoring—describing a visual work as a series of instructions or relationships executed by a computer—enables creators to

create generative and interactive works, automate tasks, and manage complex compositions [35]. Creators procedurally define and parameterize many types of artwork: data visualizations [16], models in computer-aided design and fabrication [39], and generative illustrations and animations [33].

To work procedurally, creators often rely on symbolic tools [35]. Symbolic tools for generating visual output consist of textual [43] or visual [10, 12, 17] programming languages. These tools are extremely computationally expressive because creators can use low-level primitives and abstractions to create custom procedural behaviors. The representational nature of symbolic tools can present barriers for some visual creators who are accustomed to working with graphic representations and direct manipulation. Direct manipulation software lets creators access the benefits of digital authoring and interact through concrete representations with continuous and immediate visual feedback on the results of their actions [45]. Moreover, visual thinkers can use direct manipulation for problem solving [53] because they can develop mental models of abstract problems through concrete visual interaction [24].

To integrate the opportunities of symbolic programming and direct manipulation, researchers and software designers have developed *procedural direct manipulation systems*: tools where creators control procedural relationships through the direct manipulation of graphical elements [13, 20, 25, 55, 57, 58]. These systems are powerful because they support describing constraints, mappings, and other procedural effects through direct selection and sketching in the drawing canvas. Current procedural direct manipulation systems are often limited in comparison to symbolic tools in two ways. First, they may rely on predefined mapping behaviors [20, 28, 58], limiting creators to accessing procedural functionality that is encapsulated in fixed, high-level data types which constrain the range of outcomes. Second, different systems rely on different types of mappings to achieve similar outcomes [55, 58]. As a result, specific interaction techniques developed in one system often only apply to one target effect. For example, a creator might use object constraints to control the layout of multiple elements [20] and kinetic textures to control the animation of multiple elements [26]. While this may not be a significant restriction for individual applications, it presents a fundamental limitation towards developing interoperable procedural direct manipulation paradigms [1]–for example in cases where creators seek to integrate procedural layout *and* animation.

We are inspired by the opportunities of domain-specific procedural direct manipulation systems. Our objective is to contribute a generalizable procedural-graphical interaction approach. Specifically, we seek to develop a flexible mechanism for procedural authoring wherein creators use geometry to describe processes for transforming graphical elements. Our objective is centered around three intersecting design objectives:

- **Flexible input**: any geometry should be interpretable as procedure.
- **Procedural expressiveness**: geometry should determine the behavior of low-level procedural relationships.
- **Breadth of application**: our method should generalize to effects affording procedural control of stylistic, spatial, and temporal qualities of visual output, as well as behaviors that integrate these properties.

We introduce *Drawing Transforms* (DTs), a novel interaction primitive that extracts geometry from *any manually-drawn input* to specify expressive procedural transformations of *target artwork*. DTs enable authoring low-level transformations of target artwork through a flexible parameterization of arbitrary vector graphic input; creators can use hand-drawn input to describe continuous and discrete transformations, evaluate conditionals, and map transformations to single or multiple targets. DTs also support abstraction and reuse by enabling creators to create different transformations by manipulating the geometry context.

To illustrate how a person would use DTs within a sample system, we describe an example artist, Lily, who is creating a digital birthday invitation. Lily has drawn a balloon and wants to change its color. She taps on it to select its color and then draws a vector curve in an upward arc. The system uses the starting point of the curve to map to the default base color of the balloon and interprets the curve to change the color of the balloon to red. Lily selects the arc and clicks on a *play* icon in the system interface that allows her to see that instead of instantly changing the color of the balloon she can play the change over time as an animation. She creates many copies of the balloon and applies the same animation. She draws another upward arc to test out a different color change from blue to purple. She tries to group both arcs and use them together to transform the color of the balloons. They change from blue to a range of colors between purple and red. Then she realizes she can also animate how the balloons are moving by selecting their position and drawing a few winding motion paths from the middle to the top of the page. Instead of having them move at a steady rate, she taps on the motion paths and draws how she wants the balloons to move: a straight line for a steady speed, then an arc curved upwards so they increase in speed. For good measure, she moves her pen in a squiggly line up and down so the balloons bob up and down near the top of the card. Lily has created a custom, generative animation of a bunch of balloons by hand.

DTs builds on prior approaches in procedural direct manipulation to use properties of artwork to manipulate artwork [20]. We contribute an interaction primitive that enables controlling distributions across space, style, and time for multiple elements. In contrast to higher-level procedural direct manipulation tools, DTs are akin to a lower-level programming language: more procedurally expressive and requiring additional authoring effort to express complex outcomes.

We informed the design and implementation of DTs through expert interviews with visual motion graphics artists and designers who work across symbolic programming and direct manipulation. We used these interviews to identify strategies for managing distributions of visual elements across space and time. We also analyzed the interaction techniques and applications of prominent procedural direct manipulation tools for visual art, design, animation, and interactivity to identify factors that shape authoring expressiveness across multiple visual media domains.

We evaluated the procedural expressiveness of our approach by implementing the DT primitive in an example animation and

motion graphics system for animating multiple elements simultaneously (AMES). We used this system to recreate and extend animation and motion graphics work created with prominent procedural direct manipulation and symbolic programming systems. Our examples demonstrate the range of procedural control and expression that is possible with our approach. We demonstrate how DTs can: 1) recreate procedural distributions from prior direct manipulation systems while also augmenting the output with animated effects, 2) reproduce complex motion graphics work that was originally developed with the Java programming language, and 3) generate and control a variety of particle-system effects from the ground up rather than rely on pre-defined particle behaviors. We draw from these example applications to describe the potential applications of DTs to other domains of visual expression.

2 RELATED WORK

Working through concrete, visual, and geometric depictions has many advantages. Manual and digital sketching tools enable artists to create sophisticated outcomes through manual skill [38]. Sketching can also play a role in cognition. Many people solve problems by sketching graphic elements, and drawings can encapsulate information more efficiently than symbols [11]. Graphic depiction is also fast [22] and the speed of drawing allows creators to make decisions and act on them while working [3]. Efforts to integrate sketching, geometric representation, and computational expression have been underway since the advent of modern computing [11, 14]. Notable examples include *SketchPad*, which supports object-oriented relationships in geometry [46] and *GRAIL* which enables creators to quickly specify procedures by graphically drawing flowcharts [7]. In the remainder of this section, we describe recent efforts to blend procedural authoring and direct manipulation.

2.1 Integrating Symbolic and Direct Interaction

Researchers and software developers have augmented direct manipulation design tools through the addition of symbolic programming languages. Commercial design technologies for motion graphics [17], VR development [48], and CAD [10] feature visual programming languages that can read input from the direct manipulation environment and produce visual output. These languages extend direct manipulation by enabling automation and non-destructive edits, however; they enforce a strong separation between editing procedures and directly interacting with the artwork [51].

HCI researchers have explored visual programming interfaces and language design aimed at lowering barriers to generating procedural behaviors for visual design applications. *Interstate* enables the creation of interactive behaviors through a visual notation that graphically depicts state-constraint program structure [41]. *Dynamic Brushes* enables visual artists to author manual-procedural drawing behaviors by using stylus input as a first-class datatype [19] and by visualizing program state on the drawing canvas [34]. Ma et al. developed a tool for stylized animations that propagates high-level edits to animation behavior in a timeline sequencer using a node-graph-based programming language that supports custom behaviors [37]. These works demonstrate how low-level procedural descriptions support idiosyncratic visual expression. We aim to support similar degrees of expressiveness while avoiding the separation imposed by combining symbolic language and direct editing of geometry. Researchers have also developed systems for bi-directional control between symbolic programming and direct manipulation. *Leogo* supports learners by integrating programming by demonstration, manipulating graphic UI elements and textual scripting [6] and *Sketch-n-Sketch* supports the authoring of textual programs for vector graphics creation by exposing intermediate execution products that enable users to specify program functionality as they draw [13]. Our objective is aligned with the spirit of *Sketch-n-Sketch* in that we seek to enable flexible procedural behaviors; however, we eschew symbolic languages for a geometry-oriented primitive that generalizes to spatial and temporal effects.

2.2 Procedural-Direct Manipulation

Enabling people to create visual procedural output without a symbolic programming language is a major focus within HCI. Researchers have developed procedural-direct manipulation systems across a wide range of domains. For illustration and graphic design, *Many-Spector* [15] and *Para* [20] enable the creation of parametric constraints between graphic elements which are automatically maintained as the artist edits their artwork. Recursive Drawing supports self-similar generative illustrations by embedding one drawing canvas within another [44]. In animation, *Skuid* [28], *Draco* [26], and *Energy Brushes* [59] enable animators to control predefined animation effects and particle systems with hand-drawn strokes. *Megafauna* [4] and *Kitty* [25] enable creators to manipulate animation effects by sketching mapping functions on a control graph. Our research aims to support procedural direct manipulation applications across illustration, graphic design, and animation while avoiding the use of high-level pre-defined procedural functionality. Our technique supports a greater range of outcomes with the tradeoff of requiring more operations by the creator to define a procedural effect.

Researchers have also explored procedural direct manipulation as a means to reduce challenges in data visualization. *Data Illustrator* [36], *Struct Graphics* [49] and *CAST* [8] enable creating static and animated data visualizations through a direct manipulation UI rather than a symbolic programming language. *Data Ink* [58] and *Data-Driven Guides* [29] integrate manual illustration and data visualization by enabling creators to constrain stylistic properties of hand-drawn illustrations to datasets. Unlike systems that necessarily rely on using numerical datasets as input to drive procedural effects, DTs enables creators to use geometry as input. Victor has also conducted extensive work in procedural direct manipulation by presenting example techniques for animation [54], data visualization [55], and game development [52]. In describing these systems, Victor notes the absence of a general-purpose tool or conceptual framework for procedural authoring through drawing [51]. Our work targets this exact challenge.

Procedural direct manipulation is also powerful for designing for the physical world. Computer-aided design (CAD) systems such as *Sketch-it*, *Make-it* [21] and *Cuttle* [18] enable the parametric design of precise and complex cut patterns through automatic constraints on hand-drawn geometry and pre-defined parametric modifiers respectively. Dream Sketch integrates generative design for additive fabrication by constraining high-level generative growth patterns

within hand-drawn constraints [27]. *Reality Sketch* enables creators to sketch dynamic graphics that respond to real-world interactions [47], and *ChalkTalk* uses sketching to create instructional animated diagrams [42]. These systems depend on predefined constraint behaviors, heuristics, or automatic sketch recognition to determine the procedural intent of a creator's sketches. In contrast, our approach enables the creator to define how input geometry is interpreted by specifying mapping context.

Finally, researchers have applied procedural direct manipulation toward the development of new user interface mechanisms. *Object-Oriented Drawing* [57] and *StickyLines* [5] support non-WIMP interaction paradigms by reifying graphical object editing functionality and layout guidelines through visually-represented constraints and object-oriented relationships. *Sketch-sliders* [50] enables creators to explore data visualizations through hand-sketched interface elements. *Attribute Objects* enables creators in VR to simultaneously edit the visual and animated properties of multiple 3D objects by grouping selected properties in a 3D graph space, and manually adjusting property parameters [31]. Our approach can also be used to create automated mechanisms for layout or animation; however, unlike techniques that focus exclusively on procedural manipulation of visual properties, our approach also supports the creation of dynamic animation.

3 DESIGN SPACE

We informed the design and implementation of DTs by conducting expert interviews and analyzing the interaction techniques of prominent procedural direct manipulation tools.

3.1 Informational Interviews

We interviewed three professional motion graphics artists and designers who work with both direct and symbolic tools. We received IRB approval and participant permission to participants' identities. Miwa Matreyek[1] is a performance artist who creates animations with AfterEffects. James Paterson[2] is an experimental animator who uses symbolic tools animate hand-drawn animations. Kurt Kaminski[3] is a media artist who uses programming languages to build particle systems and real-time AR animation. Each interview lasted 1.5 to 2 hours. We developed custom interview frameworks for each participant by analyzing samples of their artwork (see fig. 2 for examples) and developing questions targeting specific effects for object animation, layout, and content generation. Across all interviews, we focused on each artist's use of manual and computational tools and methods to organize animations spatially and visually. We recorded each interview and discussed initial observations and impressions after each interview. We analyzed the recording transcripts to conceptualize themes on distribution strategies, timing manipulation, and manual input. Our approach centered on reflexive analysis with a focus on emergent themes. We also distilled workflow descriptions for specific effects to explore the range of methods in procedural and manual digital visual art.

3.1.1 Interview Theme #1: Arrangement-Level Visual Design Practices. All three artists create and manage representations of groups of animations. Matreyek creates groupings of related graphic elements and then animates each group in the direct manipulation tool AfterEffects. She manually adjusts the animation effects of individual graphics to achieve variation. Kaminski uses symbolic programming languages to author custom particle behaviors that integrate fluid simulations with audio synthesis. He described how he envisions a collection of particles as a single entity with properties that can manifest individually. Paterson works across manual illustration and symbolic programming and he also develops his own direct manipulation animation tools. He described the differences between working on manually created artwork and "controlling visual arrangements with code." He emphasized how he works back and forth between symbolically describing "arrangement patterns" and iterating on manually created forms. Paterson, Matreyek, and Kaminski's workflows demonstrate how artists work at an *arrangement-level* to develop coordinated visual effects.

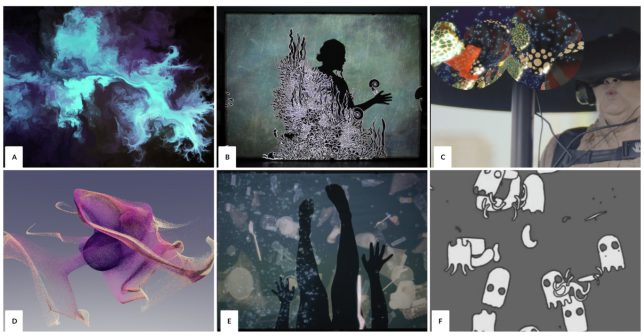

Figure 2: Artist work showing different approaches. Left: Kurt Kaminski– a. *Melange (2017)* , d. *Dust (2019).* Middle: Miwa Matreyek– b. *This World Made Itself (2013),* e. *Infinitely Yours (2020).* Right: James Paterson– c. *Colossal Wave (2017)* f. *A 25 Minute Silent Ambient Animation Compilation (2012)*

3.1.2 Interview Theme #2: Precise Manipulation of Timing across Manual and Procedural Methods. Each artist relied on precise timing control regardless of their method. Paterson uses *woven loops*– variable length loops of animation where the last frame flows seamlessly back to the first– to create phasing effects. He also creates visual density by staggering the starting points of visually related animations, like a cloud of ghosts, across time and space. Matreyek uses manual methods to manipulate loops of rotating images. She manually adjusts their position to correspond with other animated elements like a wave. Kaminski enacts precise timing control over his particle effects to develop interactive visuals for augmented reality. He fine-tunes the timing to generate cascading effects across multiple particles. Collectively all artists increase the visual complexity of their animations by manipulating a group of animations across time and fine-tuning their temporal behaviors collectively, either through manual or procedural methods.

3.1.3 Interview Theme #3: Value of Manual Input. All three artists valued manual input for enacting control and engaging with their artwork regardless of whether or not they used symbolic languages. Kaminski desired easier mechanisms to leverage physical inputs like drawing curves as input. He described the improvisational

[1]Miwa Matreyek: http://www.semihemisphere.com/
[2]James Paterson: https://presstube.com/hello/
[3]Kurt Kaminski: https://www.kurtkaminski.com/

opportunities of designing animations that respond to gestural inputs. Matreyek described using her hands as input to create manual variation in the timing of particle-based animation behaviors. She also described the importance of manually manipulating easing functions to produce an effect that is "not just the same mechanical movement." Paterson developed a custom VR animation tool centered around manual sketching. He emphasized how using drawing as the primary interaction modality made the tool "as physical as possible…leaving that spell of flow and expression unbroken." Overall using manual inputs created expressive opportunities and facilitated continuous workflows.

3.2 Analysis of Existing Procedural Authoring Techniques and Design Objectives

We compared the results of our expert interviews to our review of existing procedural direct manipulation systems. We identified four qualities in the design of existing procedural direct manipulation systems that were likely to impact key forms of expressiveness from our interview participants. We focused on capturing the range of interface and interaction design methods that shape editing flexibility, visual expression, and application domain. We analyzed how different approaches in existing systems contrasted or aligned with the practices of our interview participants. Our goal was to illuminate key approaches for a representation that could work across existing procedural direct manipulation technologies and support a range of visual art production.

Editing mechanism: Mechanisms with low flexibility map a specific geometric input in a fixed way. Strongly typed geometric inputs enable a system to infer author intent; however, they restrict outcomes. Integrating symbolic notation supports expressive outcomes but prevents graphic manipulation.

Procedural expression: Artists either relied on symbolic procedural tools or extensive manual effort to create coordinated animations consisting of multiple elements. When using predefined or custom-built procedural behaviors, artists used manual input to fine-tune the effect. Existing systems support various degrees of procedural expression for the control of multiple objects including selecting predefined behaviors, modifying parameters of behaviors, designing new behaviors from fixed data types, and enabling user-defined behaviors. Procedural tools that enable authoring low-level abstractions, like constraints, enable users to create their own procedural relationships to integrate the behavior of multiple elements.

Timing support: Fine-grained control of temporal effects is critical for producing stylistically distinct animated works. In existing systems, timing support can be static (no representation of time), fixed (such as movement at a fixed rate), manipulable (varying how property changes over time), or (re)definable (changing how timing is interpreted).

Target application: Our interview participants worked across interactive, 2D, and 3D domains. Kaminski and Paterson also blended software development with animation production. Existing techniques in procedural direct manipulation primarily focus on authoring a specific effect or target a single application area. Application-based research is important but can create strong boundaries between tools and high learning thresholds [2]. An alternative is to develop techniques that act as substrates across different media [30].

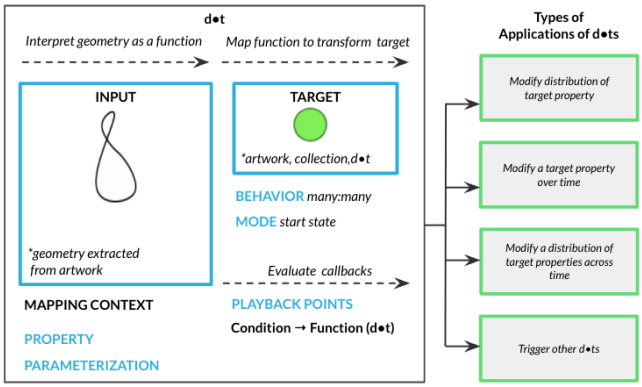

Figure 3: DT: geometry defines a procedural function. A DT evaluates geometry based on inputs (*cyan*) to describe multiple types of procedural behavior (*green*).

We drew from our analysis to define intersecting design objectives for the DTs interaction primitive. These were:

- **Flexible input**: Our approach should enable interpreting any 2D geometry as input. Stylistic differences in the structure of the geometry should produce different procedural outcomes.
- **Procedural expressiveness**: Creators should be able to use geometric input to describe low-level procedural relationships that can be combined to produce different effects. We aim to support modifying attributes of abstract data types, to create interoperability. Creators should also be able to modify data structures containing artwork.
- **Breadth of application**: Our approach should function for both static and animated output. Creators should be able to integrate control of spatial, stylistic, and temporal properties of artwork.

4 DRAWING TRANSFORMS

We contribute DTs a direct-manipulation primitive that uses drawing to author low-level transformations of target artwork through a parameterization of arbitrary vector paths. Compared to existing procedural direct manipulation the DT primitive is akin to a low-level programming representation: more expressive and more complex with a greater range of parameters. The process of working with the DTs primitive is shaped, in part, by how it is implemented within a specific procedural direct manipulation system. We describe the functional properties of the DTs primitive and then list several applications of DTs to author different types of procedural behavior. We provide pseudocode for the DTs parameterization and transformation in appendix A.

4.1 Structure of the DT Interaction Primitive

A **drawing transform** (**DT**) is a procedural transformation function that interprets input geometry to transform the spatial, stylistic, and timing properties of a target. **Input geometry** consists of one

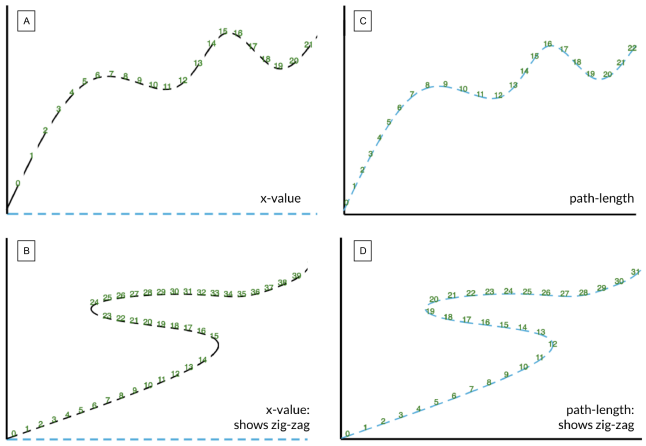

Figure 4: Parameterization specifies how vector strokes are interpreted. This enables input artwork to drive function because segment indices can be mapped to either time or to an index in a collection.

or more continuous, closed, or open vector-graphic paths with a start and end point. This can include but is not limited to lines, polylines, arcs, Bezier curves, ellipses, and irregular polygons. A DT procedurally transforms a **target**: a graphical element. In practice, targets are individual vector graphics or ordered collections of vector graphics. We visually summarize the DT primitive in Figure 3. Any ordered collection can serve as either input or target geometry for a DT. This could include existing vector-graphic collection representations like Illustrator groups–where the order is implicitly assigned, drawing order, or lists in procedural direct manipulation tools like Para [20]–where the order is explicitly set by the artist. Finally, because DTs themselves contain vector graphic input geometry they can also function as targets and be transformed by other DTs.

With DTs, drawing is akin to authoring a function. The DT transformation function evaluates input geometry through five parameters: 1) geometry parameterization, which calculates numerical values from input geometry, 2) property mappings, which control how target artwork is transformed, 3) behaviors, which determine how property mappings function for targets comprising multiple elements, 4) mode which specifies if a transformation is relative or absolute, and 5) playback points, which are used to trigger discrete events when a DT is executed. We detail each parameter below.

4.1.1 Geometry Parameterization. The geometry parameterization segments input geometry at even intervals to output a sequence of numerical values. Segmentation depends on the drawing direction or the start and end of the vector path. Each DT segments geometry in one of two ways. The first segmentation approach is *x-value* where paths are split as even segments along the x-width. *X-value* is like using a straight edge for measurement (see fig. 4-a). The second segmentation approach is *path-length* where the path is split as even segments along its length (see fig. 4-c). *Path-length* is like using a measuring tape that winds along the entire length of the path.

DTs support path-length parameterization for two reasons. First, path-length parameterization is not dependent on traversing an axis from left to right. Instead, the artist can use their direction to describe procedural functionality. Second, path-length parameterization also simplifies describing periodic structure with hand-drawn forms. For example, say the artist seeks to create a seamless looping animation of a bouncing ball. They can do so using an x-value segmentation of a waveform function (see fig 5-a) wherein x-value corresponds to time and y-value is mapped to the ball y-position. However to ensure a seamless loop they must draw a precise waveform. The artist can create an identical transformation using path-length parameterization on a circle wherein path length corresponds to time and y-value corresponds to the ball y-position (see fig 5-b). If the DT is played and looped, the circle serves as a literal representation of a seamless loop.

Parameterized segments are numbered. They start at index 0 (the start point) and continue as the artist draws. For closed shapes, segmentation start and end points depend on the geometry implementation of a given system. For our testbed, we use the Paper.js [23] vector geometry implementation wherein the start and end points of a closed shape are set at the lowest point of the shape on the y-axis. The total segment count is determined by a predefined parameter. In practice we set this parameter dynamically with respect to the mapping in our testbed implementation; however, depending on implementation, the artist could directly specify this value for greater control. The segment index is used to drive the function in an unambiguous fashion. For example, say an artist draws a vertical zig-zag path where there are multiple y-values for every x-value on a standard x-y graph. Because segments depend on drawing direction, the system calculates the y value for each segment with respect to path length and x-intersection. Figures 4-b and 4-d show examples of x-value and path-length parameterization of zig-zag paths respectively.

Each segment outputs four numerical values: segment index, x-value, y-value, and path-length. Artists can use the same input path to specify a function across time or across a collection. Depending on the mapping, the DT will assign values to the target with the segment index corresponding to either time or collection index. If multiple paths are used as input, for every segment index the DT will calculate an array of x-value, y-value, and path-length values to update the target. We provide further detail on interpreting multiple paths in section 4.1.3.

4.1.2 Property Mappings. The property mapping controls how the target artwork is updated in response to the values generated by the geometry parameterization. Default property mappings consist of lower and upper bounds and set functions which linearly map a numerical value to a graphical property (e.g. `property[min, max] = f(time or index, x, y, path length)`). For example, consider a DT that uses the hue property to map output y-values from the geometry to a range of hue property values from 0 to 360. The property mapping sets the hue property of the target to that mapped value.

The min and max ranges can be set to the default ranges for the given property. In our testbed, we include four categories of property mappings specific to visual design applications, listed in Table 1. Property mappings could be expanded or modified to

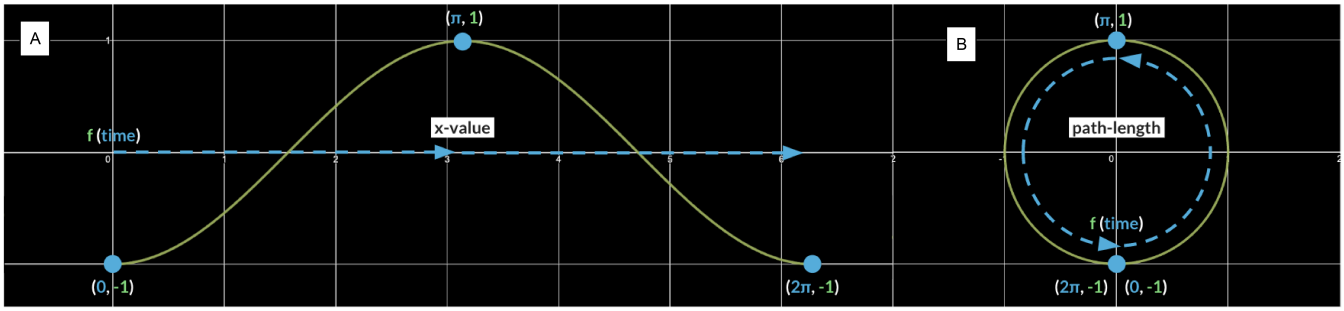

Figure 5: To modify the property of an element from [-1, 1] an artist can either draw a sine wave or a circle. The second representation, using path-length parameterization, enables creating a perfect looping animation, where the end flows seamlessly to the start state.

support specific application domains. For instance, in our testbed, we use a property mapping to duplicate new elements where the property *duplicates* can be used to create or remove copies of an element. Together, parameterizations and property mappings produce different outcomes. For example, when a DT with a position property and path-length parameterization is applied or played, the input geometry is interpreted as a set of position values for multiple graphic elements in the target artwork, or a motion path (see fig. 1-a and c).

We reify how the property and parameterization are used to interpret geometry through a graphical representation, which we call the *mapping context*. The mapping context maps the x or y-axis of the bounding box of input geometry to the property mapping range and the segmentation basis (x-width or path-length) to time or an indexed list. In our testbed, mapping contexts can be directly manipulated through operations like dragging the y-min and y-max values.

4.1.3 Behaviors. Behaviors determine how a DT samples and interpolates mapped property values by index to transform a target collection of vector graphics (e.g. `Targets[i] = alternate(i, f[inputs])`). In our testbed, we implement three behaviors listed in Table 1: interpolate, alternate, and random.

Behaviors also determine how values are calculated with respect to the geometry of multiple paths. Figure 6 demonstrates sample motion-path transformations for a DT that uses two vertical curved paths as input to control the motion path of a collection of nine circles. In this example, interpolation produces nine different motion paths that correspond with an interpolation of the points across the original two input paths. Alternate produces two motion paths that correspond with the input paths, with odd-indexed circles traversing a path that corresponds with the left-most input path, and even-indexed circles traversing a path that corresponds with the right-most input. Random samples a random interpolated path using the input geometry paths with indices closest to the relative randomized index from the artwork in the target collection. The interpolated path is then used to calculate the value update of the target artwork.

4.1.4 Modes. Modes enable describing two types of transformations for a property mapping: absolute or relative (see table 1). The mode determines if a property is set with or without respect to its initial value. The transformation begins by either resetting or preserving the original property value for the target based on the mode. In our testbed, artists select a mode with a dropdown.

Table 1: Example DT Property Mappings, Behaviors, and Modes.

Mappings	
Spatial	E.g. position distribution, motion path, scale animation.
Stylistic	E.g. hue distribution.
Geometric	E.g. polygon number of sides distribution, vertex animation.
Instancing	E.g. duplicate each.
Behavior	
Interpolate	Generate a Lagrange polynomial using input, sample with target index / total input paths.
Alternate	Cycle through input geometry using the target index % the number of input paths.
Random	Generate a Lagrange polynomial using input, sample using a randomized index.
Modes	
Absolute	Set the target state to the start state given by the input.
Relative	Transform with respect to the original target state.

4.1.5 Playback points. Playback points determine how DTs elicit discrete events during geometry evaluation to trigger other functions. They consist of trigger-value pairs of pre-defined geometric conditionals like slope change and other functions, including DTs. Each type of playback point, such as slope change, can be calculated based on the outputs from the parameterization (e.g. segment index, x, y, path length) and evaluated as the DT executes. Granularity affects accuracy, so the system may evaluate finer-grained segments to assist with playback points if a threshold is detected. In our testbed, artists can select playback point triggers through a drop-down and then indicate the triggered function by drawing a link to the artwork used as input to a specific DT.

4.2 Applications of Drawing Transforms

Artists can author different types of procedural behavior with DTs including the following:

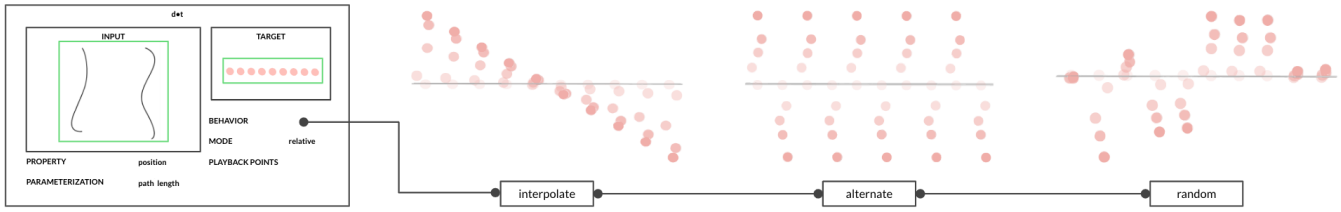

Figure 6: DTs support using multiple input paths which enables artists to author and edit a small number of input paths to rapidly and precisely control a large number of elements. These motion path animations were created using two paths drawn in opposite directions. The grey line indicates the starting point of the circles that are onion-skinned. Each application uses different interpolation behaviors: (a) interpolate, (b) alternate, and (c) random.

4.2.1 Modify a Property Across Multiple Elements. DTs can create a range of property values across a collection of artwork. We use an x-value parameterization and hue property to specify a range of hues in a sequence of triangles (see fig. 1-b). The artist can use random or evenly-spaced values from the parameterization to update or set hue across a collection by using different behaviors.

4.2.2 Transform A Property in Time. Artists can use DTs to change properties over time, such as animating artwork along a motion path (see fig. 1-c). Because DTs use artwork as input and output, they support chaining behaviors. For instance, we play two DTs: a DT with a circle as input that defines a motion path for a triangle and a DT that uses another circle to scale the first circle. The triangle traverses an expanding and contracting motion path, creating an effect where the triangle spirals outwards and then inwards.

4.2.3 Transform Multiple Elements Across Time. DTs also enable generative many-to-many animations, for example by using two motion paths to control the change in position over time for multiple target elements. This allows artists to generate and vary multiple animations procedurally and drive variation across procedural functions through dynamic instantiation. In Figure 1-e, we use input geometry paths to interpolate different rates at which particles should be duplicated across a target collection of artwork. This creates an effect where new instances of artwork move upwards at a constant rate for a fixed duration before disappearing.

4.2.4 Transform DTs with Other DTs. DTs can procedurally transform properties of other DTs. For instance, a *control DT* (a DT that modifies a DT) can modify how a target DT's segments are mapped to time. We linearly map the total number of segments from the control DT to the total number of segments in the target DT. Then, when the control DT is played, its outputs (the segment index, time, and path length) re-parameterize the target DT's input geometry. That portion of geometry, calculated with the linear mapping described above, is then mapped to the time specified by the controlling DT's output. In Figure 1-f, we use a control DT to modify a target DT that encodes a motion path. When the motion path DT is played, the pink squares traverse the motion path at a constant rate. When the control DT is played, the square accelerates up the path, slows down and descends halfway, slows down, and finishes moving upwards before moving back down along the motion path.

4.2.5 Define Responsive Event-Driven Sequences. We can author persistent procedural sequences using DT playback points. For instance, the DT shown in Figure 1-e, uses a playback point that is called when new instances are generated. This playback point activates the motion path mapping. Another playback point for the motion path uses the end of the transformation to trigger removing the transformed artwork. With this structure, we can edit artwork that is being duplicated, geometry that describes the duplication rate, and the motion path. When we play the duplication DT, new instances will traverse the motion path and disappear.

5 DEMONSTRATIVE EVALUATION

We implemented a version of DTs by developing a testbed motion graphics system, *Animating Multiple Elements Simultaneously* (AMES) [4] with Paper.js, a JavaScript vector-graphic scripting library [23]. We evaluate our approach by using the DTs implementation in AMES to recreate and extend work from prominent procedural layout and motion graphics systems. First, we describe a sample workflow with the AMES system. Then, we show how DTs can support a range of procedural behaviors possible through both direct manipulation and symbolic tools. We apply DTs to 1) extend direct-manipulation constraints to procedurally generate shapes and animations, 2) recreate motion graphics artwork made in a symbolic programming language, and 3) author distinct behaviors for particle systems solely through direct manipulation.

5.1 Sample AMES Workflow

Figure 7 shows an artist creating a DT in the AMES UI to animate goldfish. After she draws a goldfish, she uses the collection tool to select the goldfish and create a collection. She then drags the count value (shown in UI on the canvas in a green box) to create additional copies. She draws two paths and adds them to a second collection. Using the DT button, she creates a new DT and corresponding DT editor. She draws links from the input field in the DT editor to the path collection and the target field to the goldfish collection. In AMES, we made a design decision to combine selecting property and parameterization. In our examples, static transformations combine a property with x-value parameterization, and transformations in time combine a property with a path-length parameterization. From a dropdown, the artist selects *motion path* (path-length parameterization and position property mapping). She

[4]The source code for the AMES implementation of DTs can be found at https://github.com/SoniaHashim/ames-playground. The repository includes a link to a web-based executable demonstration of the AMES system

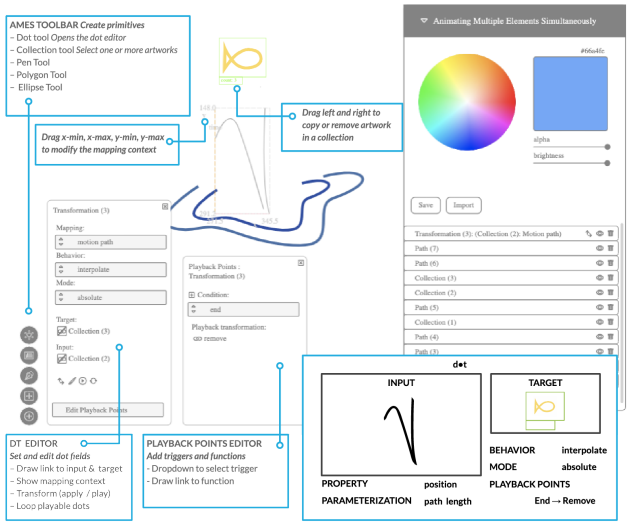

Figure 7: The AMES UI (*cyan* annotations) enables creating DTs. We graphically summarize the DT that is being edited.

adds a playback point using the playback point editor. We use this basic interaction structure in the examples we describe below.

5.2 Generate Shapes and Animations

DTs enables using manually drawn paths to author different types of procedural behavior, while also supporting the reuse of manually created geometry in multiple ways. In our first example (fig. 8), an artist creates a twinkling starfield by animating procedurally generated artwork. This example recreates and extends the approach used in the Para software [20].

First, four DTs are used to procedurally generate and arrange artwork. The first DT controls procedural shape generation. This DT uses two short, vertical paths, one drawn up and one down, as the input geometry to modify the vertex position of a regular polygon (fig. 8-a). By applying this animation with alternate and relative behavior, each input path transforms every other vertex inwards or outwards based on its starting position relative to the polygon path. The DT creates a star shape. Moreover, it encodes a procedural function to create a star shape on any regular polygon with any number of sides. The artist can modify the number of sides of the polygon and experiment with different star shapes. The DT uses the drawing direction to specify meaningful information about how the geometry is interpreted. Next, two DTs that use the same input geometry are used to vary the hue and scale across a collection of 15 stars (fig. 8-b). The artist can edit the mapping context of the scale property mapping to modify the range of scale values that the DT uses to map output values that set the scale of each star. Because the same geometry is used to generate output values for both functions, the relative relationships between the hue and scale of the stars will remain the same. The last DT that we use to arrange artwork positions the stars at random locations between two hand-drawn lines by using random behavior to map the outputs of the DTs to generate random values constrained by the input geometry (fig. 8-c). This interaction enables a random position distribution defined by hand-drawn geometry.

We use two additional DTs to animate the stars– one to specify vertex animation and one to control animation rate. The animation transformation DT reuses the same input geometry used in the first DT to specify a vertex animation. Instead of playing this animation once, which would deform the target shapes, we use a circle as input geometry to a second control DT. By playing the control DT, the animation progresses from its start to end state and then returns to its original state, 8. The control DT also modifies the easing of vertex geometry animation: the stars expand slowly at first, then increase and decrease in speed. They contract with the same easing effect, returning to their original state. Repeating the execution of the control DT loops the animation seamlessly. In the final result, shown in fig. 8-e, the stars twinkle perpetually.

Overall this example shows how DTs enable the same parametric artwork generation and layout features as Para, with greater flexibility. We can control both the geometry and the layout with the same underlying representation. Furthermore, unlike Para, this representation can also support the animation of the resulting artwork.

5.3 Recreate Artwork made with Textual Code

DTs can recreate event-driven sequences originally created in symbolic textual programming tools. In our second example (see fig. 9), we use DTs to recreate an animation made with a textual programming language [43] by artist Dave Whyte[5] [56]. We refer to this artwork as the N-Gon. In the comparison of Whyte's work and our recreation, we show the animation behavior of the N-Gon (fig.9-h, i): circles traverse as nested regular polygons with different numbers of sides. As each circle changes direction on the corner of a polygon, it is duplicated. The duplicated copy increases in scale and disappears. The circle hue matches the hue of its motion path.

To create the N-Gon with our approach, we first use two DTs to create the nested polygon structure. They both transform a collection of six triangles. The first DT uses a hand-drawn line as input geometry to scale the triangles, creating a set of nested shapes (fig. 9-a). The second DT uses another hand-drawn line as input geometry and uses a property mapping to set the number of sides of each shape (fig. 9-b). Both input geometry paths reify the parameterization of the N-Gon geometry based on the scale and the number of sides respectively. Each one can be individually manipulated to transform a key visual attribute of the N-Gon.

Next, we re-create the procedural event-driven animation where circles traverse the nested geometry as motion paths and scale up at the corners. We use three DTs to do this. The first DT uses a short hand-drawn curve as input geometry to define a scale animation (fig. 9-e). We use a playback point on this DT to remove the circle at the end of the scale animation. The second DT uses the same hand-drawn path to control the duplication of circles (fig. 9-d). We use a playback point on this DT where new instances trigger the scale animation. The third DT uses the nested polygon paths, (the procedurally generated artwork described above), as the input geometry to specify motion paths (fig. 9-c). We add a playback point

[5]Dave Whyte: https://beesandbombs.com/

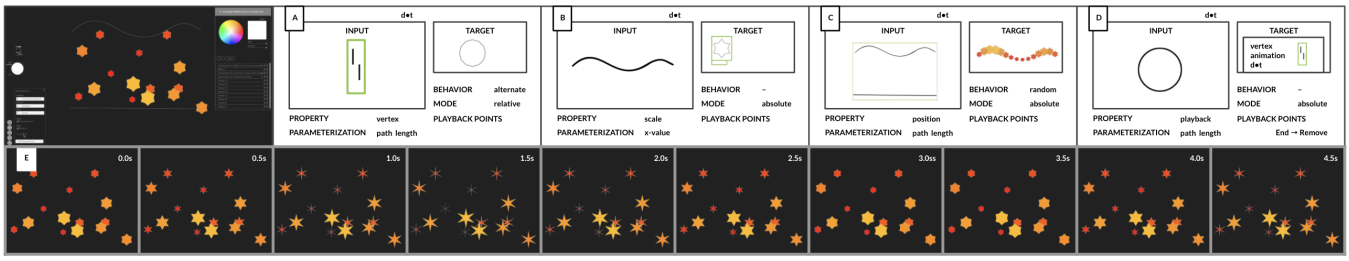

Figure 8: Artists can use DTs to define, arrange and animate artwork. Example 1: Twinkling Starfield: (a) transform vertex geometry to procedurally generate shapes and animate artwork, (b) use a DT to vary scale and color, (c) randomly position stars, (e) animate vertex geometry, (c) modify animation playback to create twinkling stars, (e) the final result.

so that the slope change that occurs at the corners of each polygon triggers the duplication DT.

Together, these three DTs enable recreating the N-Gon animation through graphical representation rather than symbolic code. When we play the motion path DT, each circle is animated around a polygon path. When the slope of its trajectory changes, the circle is duplicated. The newly instantiated circle is scaled up before it disappears. Additionally, because DTs represent procedural relationships as input geometry, an artist can create variations of the DT by editing vector graphics and visualizing geometric relationships (fig. 9-g), as opposed to editing numeric values or using algebraically-defined functions. DTs support reusing geometry in multiple ways which enables defining meaningful procedural relationships directly on the canvas. Here, the nested polygons are used on the canvas as artwork, as motion paths, and to evaluate conditional events.

5.4 Author Particle System Behaviors

Prior procedural direct manipulation tools like Draco [26] and Kitty [25] rely on predefined particle behaviors to create animated textures. We demonstrate how DTs can be used to author particle system behaviors from the ground up by generating three distinct stylized particle system effects (fig. 10).

5.4.1 Particle Effects: Fireworks. DTs enable 1) automating the generation of new elements using user-defined events and 2) using new instances to trigger animations represented by expressive, hand-drawn artwork. In Figure 10-a, we combine these two affordances to create an animation of a firework.

This example uses four DTs. The first DT describes a motion path and uses hand-drawn paths in a firework shape to animate particles. The second DT is a control DT that changes the easing of the motion path animation. Instead of the animation playing at a constant rate, the particles accelerate as they move outward. The third DT duplicates the particles. We use a playback point to connect the end of the second DT to the third DT so that after the firework explodes, new particles are generated. The fourth DT specifies a scale animation, and uses a playback point to remove artwork at the end of the animation. New instances trigger the scale animation.

The result is a firework-like particle system: as particles explode outward new particles are created that pulse briefly and disappear. We can loop the execution of the control DT to loop the particle system behavior.

5.4.2 Particle Effects: Rain. Because DTs enable defining geometric procedural relationships and using geometry as flexible input, an artist can use DTs to interpret geometry to define many kinds of procedural functionality including defining a clock. In Figure 10-b, we create a rain effect that consists of procedurally instantiated droplets at a rate determined by a motion path that acts like a clock. Rain is distributed evenly across motion paths that are positioned randomly between two horizontal lines to create uneven, but structured, animation behavior. The first DT transforms the position of vertical paths across a space given by two horizontal lines. The second DT uses the vertical paths as motion paths to transform raindrops. Raindrops are procedurally instantiated using a DT that maps a hand-drawn curve to a duplication rate. A playback point uses new instances to trigger a motion transformation. Another playback point triggers removing the raindrop at the end of the motion path animation. The last DT creates a motion path-based animation of a small circle that traverses a large circle. This DT provides the timing functionality; at the end of each iteration, it triggers the duplication DT to create more rain. The timing DT determines the frequency and intensity of the rain.

Both the firework and rain example use the same number of DTs. Because DTs support many types of procedural control, the artist can create an entirely different effect using low-level procedural functions as building blocks.

5.4.3 Particle Effects: Smoke. DTs enable using a few input geometry paths to interpolate transformations of many target elements. We demonstrate how this affordance enables the approximation of the fluid movement of multiple particles. In Figure 10-c, we use DTs to create a particle system that resembles smoke. This example uses four DTs. The first modifies the hue and scale of a collection of translucent circles. The remaining three specify the animation and use interpolate behavior to smoothly interpolate transformation functions across multiple elements. The second and third DTs use two hand-drawn paths that have spirals like smoke as input geometry. The second DT defines a motion path transformation of the particles, which are removed at the end of the animation with a playback point. The third DT specifies the rates of procedural instantiation of the smoke particles. Because one path is shorter than the other, unequal numbers of copies are made. Fewer large

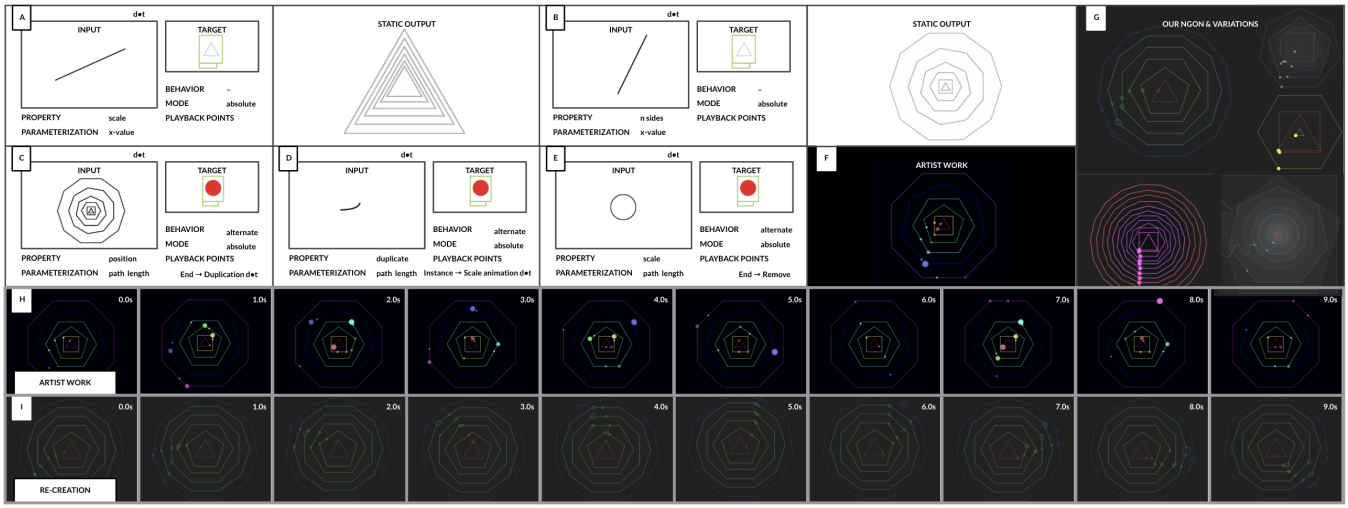

Figure 9: DTs enable artists to create work made in symbolic programming tools through direct graphic representation. Example 2. N-Gon: Reproducing animation made by artist Dave Whyte. We show the authoring process (a-e), comparisons of the original and recreated artwork (f, h, i), and exploratory variations of our N-Gon (g).

purple particles are duplicated in comparison to the small green particles. The fourth DT defines the motion path animation of the smoke particles moving up a chimney. At the conclusion of the motion path animation, the DT triggers the duplication DT to create smoke particles.

The final effect is a smooth animation where smoke particles are emitted continuously. They swirl upwards in circular patterns. This example shows how manually drawn motion paths can serve as a structure to a generative effect in DTs.

Collectively these examples suggest that DTs could offer a means to provide greater expressive control by creators in direct manipulation by enabling them to edit existing behaviors or create their own without resorting to a symbolic representation. When we asked one of the artists from our formative interviews to provide their impressions of a full demo of DTs and its potential application to their work, Kurt Kaminski shared *"I wish more content creation applications had interfaces like this. I use Houdini and the interface is not geared toward gestural input. I would love to see DTs integrated into Houdini, or even more so in Photoshop or AfterEffects which lag in both gestural and procedural tools."*.

6 LIMITATIONS

We focus on evaluating the computational expressiveness of DTs through demonstrative examples which is a common method in HCI toolkit research [32]. In particular, we showcase the "expressive match" [40] enabled by using drawing to enact multiple forms of procedural control akin to other procedural tools. Our starfield extends an example from Para [20]: we manipulate a collection and also procedurally generate shapes and animations. The N-Gon recreates work made in Processing [43]: DTs enable describing user-defined procedural sequences through direct manipulation. Our particle systems show effects comparable to those in Kitty [25]: DTs let artists describe this functionality from the ground up versus relying on predefined effects.

Studies with external participants would provide valuable further insights. We omit a study from this work because our goal is to present the abstraction without a prescriptive implementation of that abstraction. For instance, in AMES, artists can use a drop-down to select a property. A voice command or radial in-canvas menu may be a more usable mechanism to set this parameter depending on the use-case. Additionally, our abstraction can apply across applications such as VR animation tools that use VR controllers to author input artwork. We hope our contribution will enable others to apply and evaluate this primitive across different applications. Our focus is on determining the expressive range of an entirely graphical procedural specification with respect to existing standards within the field of procedural direct manipulation.

Lastly, while DTs enables artists to directly edit their artwork to change procedural behavior, DTs does not support bidirectional editing. In other words, artists can interactively edit input geometry to modify the result in a continuous way, but they can not modify the result directly. We believe this is still a valuable form of direct manipulation as it provides a means to enact procedural control through drawing and editing drawings directly. Enable bidirectional editing is a promising direction for future work.

7 DISCUSSION

In developing DTs, we sought to create a primitive that supports flexible input, high procedural expressiveness, and breadth of application. In our discussion, we examine how DTs fulfills these objectives by analyzing how DTs applies to domains of visual creation, the trade-offs of geometric authoring of low-level procedural functionality, and how DTs supports manual drawing expression.

7.1 Supporting Drawing Expression

Drawing is a highly expressive medium. DTs supports manual drawing expressiveness by enabling creators to define procedural

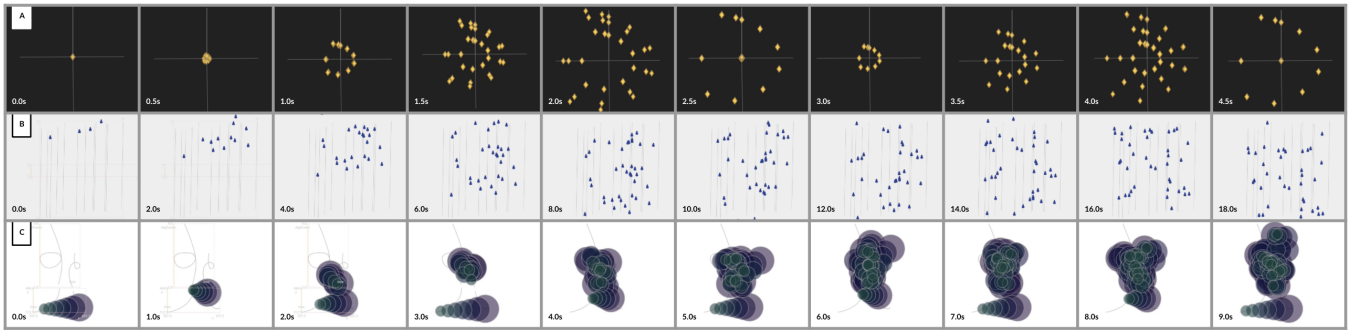

Figure 10: DTs enable authoring distinct particle system behaviors. Example 3. Particle Effects: (a) firework, (b) rain, (c) smoke.

behaviors through the quality of their hand-drawn lines. DTs' path length parametrization allows artists to draw in any direction and leverage directionality in drawing to encode information about a transformation. This parameterization approach also allows artists to encode periodic structures through hand-drawn loops instead of drawing precise, repeating waveforms. The ability to use the same vector artwork as input for multiple DTs with different mapping contexts allows artists to reuse hand-drawn inputs to describe different procedural behaviors. This structure creates the opportunity to develop expressive behavior by using contextual information to modify how hand-drawn geometry is interpreted for specific procedural functions. Lastly, playback points tie event-driven functionality to properties of the drawn geometry. This combination of approaches can allow artists to develop their own visual structures for creating sequences through drawing that best serve the visual design task at hand.

7.2 Geometric Authoring of Low-level Procedural Functionality

DTs are aligned with the stored-program concept: a principle from computer architecture of using the same substrate to represent data and programs to operate on that data [9]. Stored-program architecture can expand access to who can define procedural functionality and increase the types of procedural routines that can be developed by using the same representation as data to define operations on it. DTs apply this idea to vector graphics. Through DTs, artists use artwork to represent both *data*– the inputs and outputs of their compositions and *programs*– the procedural routines that shape their compositions. Beyond using artwork to control artwork in fixed ways, artists can author low-level forms of procedural control, because they can directly manipulate the artwork as data itself. While the ability to author low-level procedural relationships may increase the range of outcomes that are enabled through DTs, it can also place a greater burden on the artist to define detailed mappings. Artists value efficiency and may prefer forms of automation that keep them "in the loop" [35]. While DTs support "in the loop" interaction through low-level procedural control, we could envision cases where creators might also value workflows that mix low-level procedural control mechanisms with pre-defined procedural behaviors. We see future opportunities to explore how DTs can support layered procedural direct manipulation systems that enable creators to move between low-level authoring and adjusting high-level parameters without resorting to a symbolic programming language.

7.3 DTs as a General Visual Creation Primitive

We build from our examples to discuss how DTs could apply to data visualization, CAD, and interactivity.

7.3.1 Data Visualization. Although we did not implement data bindings for this work, we see opportunities for DTs to control how a data vector maps to a specific property of artwork, across a collection or across time. For instance, a designer could use visual input geometry that represents the data values of deforestation across a collection of countries as input for a DT that sets the hue across a collection of illustrated tree graphics.

7.3.2 Parametric CAD. Parametric CAD and direct modeling enable visual designers to construct models of objects based on constraints and direct manipulation of 3D geometry models. In such cases, designers often rely on blueprints to refer to numeric values to establish constraints for models. Instead of designing numeric constraints, artists could use DTs to directly encode procedural relationships through geometry. A designer could use DTs to directly map the length of a line in a diagram to the geometric features of an input model. As a result, in addition to having procedural relationships update 3d geometric models based on changes to the model, updates to a blueprint could map directly to model edits.

7.3.3 Interactive Illustration. Existing tools for interactive illustration allow visual creators to define dynamic relationships between illustrated entities. These are often encoded through data types that represent specific inputs and types of effects. DTs could aid in reifying relationships that might be useful in designing interactive relationships such as the distance from one object to another. Distance could be represented through a geometric object that serves as the input geometry to a DT that may control the scale of the target artwork. Embedded sketching tools and procedural authoring tools for AR and VR contexts could also use real-world inputs such as tracked objects. Instead of using pre-defined mappings to pipe these inputs into specific procedural functions, DTs could enable creators to author custom procedural functionality that uses geometric inputs to author a variety of responsive behaviors.

8 CONCLUSION

We present drawing transforms (DTs), an interaction primitive that expressively interprets input geometry and allows visual artists to use drawing and artwork to author procedural behaviors to manage distributions, modify one or more pieces of artwork, and control animations and instancing across time. We motivated DTs by identifying a significant design barrier that limits procedural support in visual art and design. We demonstrate several concrete examples of applying DTs in practice to procedural art and motion graphics, and we also discuss how artists can leverage DTs to author procedural functionality in other domains.

ACKNOWLEDGMENTS

We extend special thanks to Miwa Matreyek, Kurt Kaminski, James Paterson, and Dave Whyte for granting us permission to use their work as examples or inspiration for our research. This research was funded in part by the NSF IIS Human-Centered Computing Program (Award: 2007094) and a generous gift from Adobe Research.

REFERENCES

[1] Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (The Hague, The Netherlands) *(CHI '00)*. Association for Computing Machinery, New York, NY, USA, 446–453. https://doi.org/10.1145/332040.332473

[2] Michel Beaudouin-Lafon. 2019. *UIST 2019 Visions - A World Without Apps.* https://www.youtube.com/watch?v=ntaudUum06E

[3] John Berger. 2008. Drawing. In *Selected Essays of John Berger*, Geoff Dyer (Ed.). Knopf Doubleday Publishing Group.

[4] Samuelle Bourgault and Jennifer Jacobs. 2021. Preserving Hand-Drawn Qualities in Audiovisual Performance Through Sketch-Based Interaction. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576315

[5] Marianela Ciolfi Felice, Nolwenn Maudet, Wendy E. Mackay, and Michel Beaudouin-Lafon. 2016. Beyond Snapping: Persistent, Tweakable Alignment and Distribution with StickyLines. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2984511.2984577

[6] Andy Cockburn and Andrew Bryant. 2001. Leogo: An Equal Opportunity User Interface for Programming. *Journal of Visual Languages & Computing* 8 (01 2001). https://doi.org/10.1006/jvlc.1997.0152

[7] Thomas O Ellis, John F Heafner, and William L Sibley. 1969. *The GRAIL Project: An experiment in man-machine communications.* Technical Report. RAND CORP SANTA MONICA CA.

[8] Tong Ge, Bongshin Lee, and Yunhai Wang. 2021. CAST: Authoring Data-Driven Chart Animations. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 24, 15 pages. https://doi.org/10.1145/3411764.3445452

[9] Stanley Gill. 2003. *Stored Program Concept.* John Wiley and Sons Ltd., GBR, 1691–1693.

[10] Grasshopper 2007. Grasshopper. http://www.grasshopper3d.com.

[11] M. D. Gross. 2009. Visual Languages and Visual Thinking: Sketch Based Interaction and Modeling. In *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling* (New Orleans, Louisiana) *(SBIM '09)*. Association for Computing Machinery, New York, NY, USA, 7–11. https://doi.org/10.1145/1572741.1572743

[12] Experimental Media Research Group. 2004. *NodeBox.* http://www.nodebox.net.

[13] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925

[14] Christopher F Herot. 1976. Graphical input through machine recognition of sketches. In *Proceedings of the 3rd annual conference on Computer graphics and interactive techniques*. 97–102.

[15] Raphaël Hoarau and Stéphane Conversy. 2012. Augmenting the Scope of Interactions with Implicit and Explicit Graphical Structures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 1937–1946. https://doi.org/10.1145/2207676.2208337

[16] Enamul Hoque and Maneesh Agrawala. 2020. Searching the Visual Style and Structure of D3 Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2020), 1236–1245. https://doi.org/10.1109/TVCG.2019.2934431

[17] Houdini 2022. Houdini. https://www.sidefx.com/products/houdini/.

[18] Cuttle Labs Inc. 2022. Cuttle: A design tool for digital cutting machines. https://cuttle.xyz/

[19] Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. 2018. Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3173574.3174164

[20] Jennifer Jacobs, Sumit Gogia, Radomír Měch, and Joel R. Brandt. 2017. Supporting Expressive Procedural Art Creation Through Direct Manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. ACM, New York, NY, USA, 6330–6341. https://doi.org/10.1145/3025453.3025927

[21] Gabe Johnson, Mark Gross, Ellen Yi-Luen Do, and Jason Hong. 2012. Sketch It, Make It: Sketching Precise Drawings for Laser Cutting. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI EA '12)*. Association for Computing Machinery, New York, NY, USA, 1079–1082. https://doi.org/10.1145/2212776.2212390

[22] Gabe Johnson, Mark Gross, Jason Hong, and Ellen Do. 2009. Computational Support for Sketching in Design: A Review. *Foundations and Trends in Human-Computer Interaction* 2 (01 2009), 1–93. https://doi.org/10.1561/1100000013

[23] Jonathan Puckey Jürg Lehni. 2021. *Paper.js, The Swiss Army Knife of Vector Graphics Scripting.* Retrieved April 7, 2022 from http://paperjs.org/

[24] Alan C. Kay. 1990. *User Interface: A Personal View.* Addison-Wesley.

[25] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: Sketching Dynamic and Interactive Illustrations. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. ACM, New York, NY, USA, 11 pages.

[26] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014. Draco: Bringing Life to Illustrations with Kinetic Textures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/2556288.2556987

[27] Rubaiat Habib Kazi, Tovi Grossman, Hyunmin Cheong, Ali Hashemi, and George Fitzmaurice. 2017. DreamSketch: Early Stage 3D Design Explorations with Sketching and Generative Design. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 401–414. https://doi.org/10.1145/3126594.3126662

[28] Rubaiat Habib Kazi, Tovi Grossman, Nobuyuki Umetani, and George Fitzmaurice. 2016. SKUID: Sketching Dynamic Drawings Using the Principles of 2D Animation. In *ACM SIGGRAPH 2016 Talks* (Anaheim, California) *(SIGGRAPH '16)*. ACM, New York, NY, USA, Article 84, 1 pages. https://doi.org/10.1145/2897839.2927410

[29] Nam Wook Kim, Eston Schweickart, Zhicheng Liu, Mira Dontcheva, Wilmot Li, Jovan Popovic, and Hanspeter Pfister. 2017. Data-Driven Guides: Supporting Expressive Design for Information Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 491–500. https://doi.org/10.1109/TVCG.2016.2598620

[30] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 280–290. https://doi.org/10.1145/2807442.2807446

[31] Cheryl Lao, Haijun Xia, Daniel Wigdor, and Fanny Chevalier. 2021. Attribute Spaces: Supporting Design Space Exploration in Virtual Reality. In *Symposium on Spatial User Interaction* (Virtual Event, USA) *(SUI '21)*. Association for Computing Machinery, New York, NY, USA, Article 11, 11 pages. https://doi.org/10.1145/3485279.3485290

[32] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3173574.3173610

[33] G. Levin and T. Brain. 2021. *Code as Creative Medium: A Handbook for Computational Art and Design.* MIT Press. https://books.google.com/books?id=hs_tDwAAQBAJ

[34] Jingyi Li, Joel Brandt, Radomír Mech, Maneesh Agrawala, and Jennifer Jacobs. 2020. Supporting Visual Artists in Programming through Direct Inspection and Control of Program Execution. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376765

[35] Jingyi Li, Sonia Hashim, and Jennifer Jacobs. 2021. What We Can Learn From Visual Artists About Software Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, 14 pages. https://doi.org/10.1145/3411764.3445682

[36] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John Stasko. 2018. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3173574.3173697

[37] Jiaju Ma, Li-Yi Wei, and Rubaiat Habib Kazi. 2022. To Appear: A Layered Authoring Tool for Creating Stylized 3D Animations. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA.

[38] M. McCullough. 1996. *Abstracting Craft: The Practiced Digital Hand.* The MIT Press, Cambridge, Massachusetts.

[39] Lora Oehlberg, Wesley Willett, and Wendy E. Mackay. 2015. Patterns of Physical Design Remixing in Online Maker Communities. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 639–648. https://doi.org/10.1145/2702123.2702175

[40] Dan R. Olsen. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

[41] Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: A Language and Environment for Expressing Interface Behavior. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. Association for Computing Machinery, New York, NY, USA, 263–272. https://doi.org/10.1145/2642918.2647358

[42] Ken Perlin. 2017. Introduction to Chalktalk. https://github.com/kenperlin/chalktalk

[43] C. Reas and B. Fry. 2004. *Processing*. http://processing.org.

[44] Toby Schachman. 2012. Alternative Programming Interfaces for Alternative Programmers. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Tucson, Arizona, USA) *(Onward! 2012)*. ACM, New York, NY, USA.

[45] Ben Shneiderman. 1997. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Proceedings of the 2nd international conference on Intelligent user interfaces*. 33–39.

[46] Ivan E. Sutherland. 1964. Sketchpad a Man-Machine Graphical Communication System. *Transactions of the Society for Computer Simulation* 2, 5 (1964), R–3–R–20. https://doi.org/10.1177/003754976400200514

[47] Ryo Suzuki, Rubaiat Habib Kazi, Li-Yi Wei, Stephen DiVerdi, Wilmot Li, and Daniel Leithinger. 2021. RealitySketch: Augmented Reality Sketching for Real-Time Embedded and Responsive Visualizations. In *SIGGRAPH Asia 2021 Real-Time Live!* (Tokyo, Japan) *(SA '21)*. Association for Computing Machinery, New York, NY, USA, Article 5, 1 pages. https://doi.org/10.1145/3478511.3491313

[48] Unity Technologies. 2022. *Develop Gameplay Mechanics with Visual Scripting in Unity*. https://unity.com/products/unity-visual-scripting.

[49] Theophanis Tsandilas. 2021. StructGraphics: Flexible Visualization Design through Data-Agnostic and Reusable Graphical Structures. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 315–325. https://doi.org/10.1109/TVCG.2020.3030476

[50] Theophanis Tsandilas, Anastasia Bezerianos, and Thibaut Jacob. 2015. SketchSliders: Sketching Widgets for Visual Exploration on Wall Displays. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 3255–3264. https://doi.org/10.1145/2702123.2702129

[51] B. Victor. 2011. Dynamic Pictures. http://worrydream.com/DynamicPicturesMotivation.

[52] B. Victor. 2012. Inventing on Principle. In *Proc. of the Canadian University Software Engineering Conference*.

[53] B. Victor. 2012. Learnable Programing: Designing a programming system for understanding programs. (2012). http://http://worrydream.com/LearnableProgramming/

[54] B. Victor. 2012. Stop Drawing Dead Fish. In *ACM SIGGRAPH 2012 Talks (SIGGRAPH '12)*.

[55] B. Victor. 2013. Drawing Dynamic Data Visualizations (Talk). http://vimeo.com/66085662.

[56] Dave Whyte. 2017. Retrieved April 7, 2022 from https://www.instagram.com/p/BVAb1LuhNBz/

[57] Haijun Xia, Bruno Araujo, Tovi Grossman, and Daniel Wigdor. 2016. Object-Oriented Drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (Santa Clara, California, USA) *(CHI '16)*. ACM, New York, NY, USA, 4610–4621. https://doi.org/10.1145/2858036.2858075

[58] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, Bruno De Araujo, and Daniel Wigdor. 2018. DataInk: Direct and Creative Data-Oriented Drawing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3173574.3173797

[59] Jun Xing, Rubaiat Habib Kazi, Tovi Grossman, Li-Yi Wei, Jos Stam, and George Fitzmaurice. 2016. Energy-Brushes: Interactive Tools for Illustrating Stylized Elemental Dynamics. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 755–766. https://doi.org/10.1145/2984511.2984585

A PSEUDOCODE FOR DTS

*// **Drawing Transform**: A class that provides a mechanism to*
// transform any attribute of target artwork, stylistic, spatial,
// or temporal by interpreting vector geometry as input for the
// creation of flexible user-defined, procedural mapping functions
// Parameters: Setters & getters omitted

```
input              // input shape or collection
target             // target shape, collection or transformation
mode               // enumerator for modes: relative, absolute
segmentation       // enumerator for parameterization:
                   // x-value, path-length
                   // determines state of DT (seg 0 = idx 0)
behavior           // enumerator for mapping behaviors:
                   // e.g. random, alternate, interpolate
tf_space           // struct (mapping context): describes x, y range
                   // for property mapping and x.y screen coords
                   // and axis mappings (for linear mapping)
property           // property function of target
is_dynamic         // describes if DT is static or dynamic
playback_points    // list of key, value pairs: (conditions, functions)
x; y; v            // arrays for each path in the input to track
                   // execution state
loops; max_loops   // array for each input path & max loop count
```

// Applies the DT by updating the target statically or dynamically
transform() {

For every element in the target…

... If the mode is absolute, call get_transform_value to get the value of this DT at the start state (segment index = 0) and call update_target

... If this DT is static, call get_transform_value based on the at the state mapped to the target index and call update_target

... If the transformation is dynamic (temporal), call playback_helper

}

// Recursive function that activates different states in the DT by
// cycling through segments on the input artwork
playback_helper(target_idx, curr_state_idx, next_state_idx, stop_state_idx, bool reverse) {

A base case evaluates the stopping and looping conditions for the DT: IF (!reverse && state_idx >= stop_state_idx) || (reverse && state_idx <= stop_state_idx) IF loops[target_idx] < max_loops call transform ELSE return

Call get_transform_value based on the curr_state_idx and next_state_idx and pass the output values to update_target

Call playback_helper for the next state

}

// Segments, indexes, and maps input values to calculate
// property values; returns a tuple
get_transform_value(target_idx, curr_state_idx, next_state_idx, axis_mapping) {

If the DT is static, call calculate_state on curr_state_idx

If the DT is temporal, calculate difference between the outputs from calling calculate_state on curr_state_idx and next_state_idx

// Exact sampling is determined by behavior or if the input is a collection; details omitted
Sample the input: call calculate_state on one or more input paths using an index given by the behavior (e.g.alternate), then interpolate or select across those values according to the behavior

Return calculated tuple (dx, dy, dv) values based on x value, y value, and path length

}

// Gets input segment values, maps to property values, and
// returns a point
calculate_state(s_idx, in_artwork) {

If the parameterization is path-length return point on the input at the segment s_idx

If the parameterization is x-value...

... If the path is non-looping calculate the intersection point of the x-axis at the segment s_idx and the input artwork path

... If the path is looping use the point of the input path at segment s_idx to calculate the nearest segment on the x-axis to calculate the nearest intersection point

Linearly map the point to the property range given in tf_space and return the new point

}

// Updates the target property and triggers playback points
update_target(dx, dy, dv) {

Update the execution state and check if any playback point conditions have been met based on the execution state. If so, call the playback point values (functions)

Call the property function of the target passing in dx, dy, or dv

}