

Squidgets: Sketch-based Widget Design for Scene Manipulation

Joonho Kim
University of Toronto
Toronto, Canada
joonho@dgp.toronto.edu

Fanny Chevalier
University of Toronto
Toronto, Canada
fanny@dgp.toronto.edu

Karan Singh
University of Toronto
Toronto, Canada
karan@dgp.toronto.edu

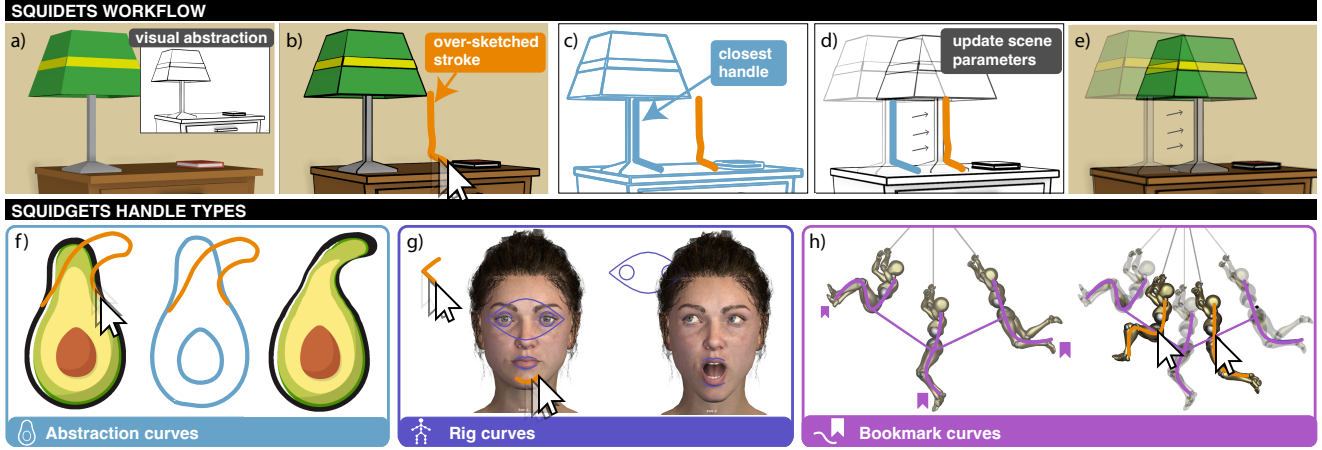


Figure 1: Humans naturally communicate desired scene changes by over-sketching (a,b). These sketched strokes, when aligned with visual abstraction curves of a scene (c), can effectively express user intended changes to scene attributes (d,e). Squidgets or ‘sketched widgets’ enable interactive manipulation of scene attributes via scene curves (f-h), such as the avocado shape attributes deformed to match its silhouette abstraction curve (f). All scene curves can homogeneously function as squidgets including pre-defined rig curves like the gaze/jaw control curves on a 3D facial rig (g), and user drawn bookmark curves that capture attribute configurations of objects, like the line-of-action curves drawn to pose an animated character (h).

Abstract

People naturally sketch strokes over graphical scenes to convey scene changes. We propose automatically interpreting these strokes to execute scene changes with squidgets (*sketch-widgets*), a novel sketch-based UI framework for direct scene manipulation. Squidgets are motivated by the observation that curves resulting from visually abstracting scene elements provide natural handles for the direct manipulation of scene parameters. Additional curves can be defined by users to author custom handles associated with scene attributes. Users manipulate a scene by simply drawing strokes, that are partially matched against scene curves to select a squidget and interactively control scene parameters associated with the squidget. We present an implementation of squidgets within the 3D animation system *Maya*, showing 2D/3D stroke input to manipulate 2D/3D scenes. We report on a controlled experiment evaluating squidgets on 2D object translation and deformation tasks, and a broader informal study on squidget creation and manipulation.

CCS Concepts

• Human-centered computing → Interaction design.

Keywords

sketching, graphics, 3D scenes, interactive design

1 Introduction

Sketching is a traditionally established tool for visual communication (Figure 2). Digital sketching is now also ubiquitous in interactive design and graphical content creation. User control of scene objects and attributes/parameters in such applications however is typically a disparate combination of traditional UI components, pre-defined widgets, and stroke gestures [43]. While a number of seminal sketch-based systems for content creation span over half a century [4, 23, 45, 53, 54], their operational interface has largely relied on pre-defined stroke gestures and traditional UI elements.

Pen and touch stroke-based interfaces are also increasingly used to control general computing applications. A vein of research has thus adapted the interaction of UI menus/buttons [1], sliders [50], and 3D widgets [43], from a point-and-click to a stroke-friendly design. Inspired by recent research [41] using sketchy-renderings of objects for interactive visualization and exploration, we aim to exploit such renderings for stroke-based scene interaction.

We thus present **squidgets** (*sketch-widgets*) as a novel stroke-based UI framework for direct scene manipulation (Figure 1). Our work uses the insight that graphical scenes inherently possess implicit, in-situ handles for scene manipulation. People naturally sketch a variety of strokes (Figure 2) over perceived scene curves to visually indicate desired scene changes (Figure 1(a-e)). Practically, scene curves are either visually imagined abstraction curves (like the silhouette of the avocado in Figure 1(f)) or explicitly modeled

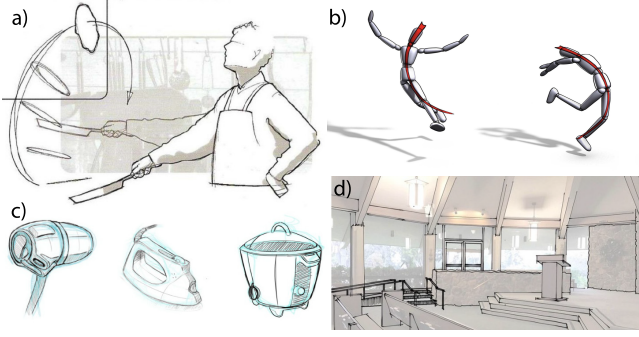


Figure 2: Traditional sketch stroke usage to convey aspects of animation: motion paths (©K. Eissen, R. Steur, BIS Publishers) [15] **(a), line-of-action** (© M. Guay) [19] **(b), design (over-sketching ideation** ©R. Arora) [3] **(c), spatial overlays** (©E. Mikiten) [36] **(d).**

scene objects (like the gaze-controller rig curve in Figure 1(g) or the line-of-action bookmark curves in Figure 1(h)). We aim to capture this natural interaction with scene curves using squidgets. Conceptually, this is an inverse rendering problem: *compute a minimal change to the current scene attribute values that will result in a manipulated scene, in which some scene curve (real or imagined) visually matches the over-sketched stroke* (Figure 1(a-f)).

The benefits of solving such a problem are twofold. First, over-sketching is natural, direct, and in-situ for 2D (Figure 1(f)) or 3D (Figure 1(g,h)) applications. Second, no explicit gestures, widgets or UI components for the scene need to be defined or learned by the user. There are however, three non-trivial and potentially ambiguous aspects to solving this ambitious problem: (1) inferring the user-imagined abstraction curves of a graphical scene; (2) discerning what part of this curve abstraction to associate with the over-sketched stroke; and (3) computing changes to the values of a set of scene attributes that result in a manipulated scene whose associated visual abstraction best matches the over-sketched stroke.

The squidgets framework allows each of these problems to be explored, constrained, and addressed independently. (1) Visual abstraction of scenes as curves have been extensively studied in Non-Photorealistic Rendering (NPR) literature [18]. Such abstractions can be defined using occluding and suggestive contours in a scene [22] (Section 3), or using image-based approaches to differential rendering [31]. (2) We define a novel curve similarity metric (Section 4) that is able to match the stroke to (partial) curves using a perceptual mix of corner, spatial, and shape proximity. A user stroke can thus be matched to a curve segment in the scene (whether explicitly modeled as a scene object, or inferred as a visual scene abstraction). (3) Scene attributes that deform the associated curve segment can then be sampled around their present values to find a resulting curve edit that best matches the user stroke.

Increasingly, artists are hand-crafting custom UI layouts using rig curves for in-situ scene manipulation [29] (Figure 1(g)). Such curves being explicit scene objects are homogeneously handled by our framework. Inspired by artistic constructs like the line-of-action for posing animated characters [19] (Figure 2(b)), we further support the rapid creation of bookmark curves that are associated with a given configuration of scene attributes (Figure 1(h)) within our

interaction framework. Note that abstraction/rig/bookmark curves are simply a categorization of all scene curves (real and imagined) that can serve as interaction handles in our framework as persistent scene objects (rig curves and bookmark curves) explicitly define both a visual scene abstraction (problem 1), and the set of scene attributes associated with the curves (problem 2). Further, since these curves and their scene attributes, are related by simple invertible (often linear) functions, finding attribute values to optimally match an over-sketched stroke is simple (problem 3).

The squidgets framework thus addresses the stroke-based manipulation of scene attributes via **all visually perceived curves** in the scene (imagined abstraction curves, or explicitly modeled scene curves: rig curves and bookmark curves).

We provide an overview of related work (Section 2), followed by details of our squidgets framework (Section 3). Section 4 presents our approach to stroke matching, attribute inference and other implementation details of a squidgets interaction prototype built within the modeling and animation system *Maya*. Section 5 presents several squidgets usage scenarios. We discuss the outcomes of a user study (Section 6) followed by a discussion of overall insights, limitations, and avenues for future work on squidgets (Section 7).

2 Related Work

The squidgets framework touches upon many areas of graphics and HCI research that we roughly classify into four themes as follows.

Visual scene abstraction: The fundamental insight behind squidgets is that imagined curves in a scene provide natural manipulation handles. While such curves are often explicitly evident in 2D graphics (Figure 1(f)), they need to be algorithmically inferred in 3D scenes (Figures 1(a-e),3). Understanding and computing such a collection of curves that comprise a visual scene abstraction is addressed by research in non-photorealistic rendering [18] and perceptual psychology [49]. We rely on this body of work to automatically compute a set of curves that define a visual scene abstraction comprising silhouette, ridge/ valley, shading contrast and border curves [5], as seen in Figure 3.

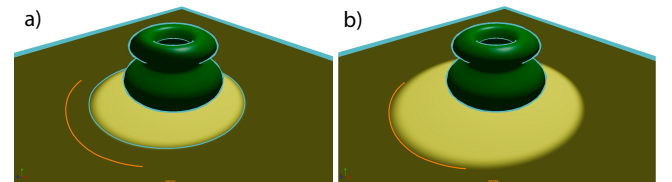


Figure 3: NPR andtoon shading is used to compute abstraction curves automatically for a 3D scene (a). User strokes can then manipulate scene attributes associated with these curves, like the cone angle of a scene spotlight (b).

Scene object transformation and composition: A large body of research spanning 50 years [45] has specifically addressed the transformation (e.g. translation, rotation, scaling) of objects in 3D scenes. We refer readers to a recent survey paper [34] tracing this research from desktop to immersive devices. Techniques designed for constrained tasks like docking [9] have also been compared across various input modalities [6]. Object manipulation techniques

interact with a 3D spatial context that can be physical (such as a tangible 3D proxy), or virtual (like mouse controlled 3D widgets) [10]. Squidget curves provide our spatial context, controlled by 2D (Figures 1) or 3D (Figure 9) stroke input.

We note that transforming 3D objects with squidgets (Figure 1) is but one example of our ability to homogeneously manipulate arbitrary scene object attributes using squidget strokes.

Sketch-based gestures and interfaces: The recognition and use of stroke-based gestures can be categorized into gestures that are hard-coded or visually matched [25]. Hard-coded gestures tend to be expressive but context-specific and difficult to customize. Visual matching, such as the \$1 Gesture Recognizer [51], geometrically matches a stroke to a user-given set of gesture templates. These gestures are typically scene-agnostic and can be mapped to general directives (e.g. invoking undo using a scratch gesture). Similar to visually-matched gestures, squidgets match user-drawn strokes to the set of squidget curves in a graphical scene.

User strokes, scribbles, and sketch gestures have been applied to interfaces in a variety of domains [7, 40] including ideation [3], modeling [4, 12, 23], illustration [27, 28, 48], and animation [19, 21, 47]. Techniques to aid the execution of sketch strokes and to aid the drawing process have also been explored [16]. While our work is focused on an application-agnostic framework for general stroke-based scene manipulation, we draw inspiration from this body of work for compelling examples, such as an animated line-of-action [19] in Figure 2(b), 1(h), to showcase squidgets interaction.

Scene proxies and custom widgets: UI widgets are visual 2D/3D elements designed to provide an in-situ interface to manipulating objects and aspects of a virtual scene [8]. Sketching such UI sliders and axes for scene manipulation have also been explored [30, 50]. Widgets are typically hand-designed to capture the form/function of scene attributes they control, and can range from simple spatial transform widgets, to custom curve controllers used to illustrate mechanical assemblies [37] or specific to the deformation parameters of complex objects such as the human face in Figure 1(g). Point-click-and-drag interaction of such widgets can be improved using stroke-based techniques [43]. Our squidgets framework is able to homogeneously interact using over-sketching with such curve-based widgets (Figure 1(h)).

Curves have also been used as visual proxies and manipulation handles for deformable objects [39, 44]. Squidgets generalize this control beyond deformable objects (e.g. the avocado in Figure 1(f)) to any scene elements whose visual appearance can be controlled by some set of attributes (e.g. the cone-angle of a spotlight in Figure 3). Finally, sketched strokes have been anchored to objects in Augmented Reality scenes to provide dynamic visualizations of changing scene attributes [46], complementary to our problem where the scene attributes respond to the sketched strokes.

Direct in-situ manipulation: In-situ visualization and control of object attributes in a scene can greatly streamline a sketch-based workflow [52]. Squidgets take such a design further, allowing a user to directly sketch the visual change they expect as a result of changes in object attribute values.

Such direct visual control requires an inverse mapping from the visual abstraction of objects and other scene elements to their attribute values. Inverse computation for direct control is popular in a number of domains on Computer Graphics, such as skeletal kinematics (IK) [2, 19], facial expressions [29, 32], CAD modeling [35], and animation [11, 24]. Squidgets are inspired by these solutions from specific graphical contexts to build a general framework for manipulating sparse sets of scene attributes using sketch-strokes.

3 The Squidgets Framework

We introduce a stroke-based UI framework which develops the idea that any visible or perceived curves in a scene can serve as visual proxies and manipulation handles to scene attributes. Squidgets exploit these curves for rapid referencing and direct visual adjustment of scene attributes, illustrated in Figure 4: with the user workflow (top row) and underlying building blocks (bottom row). Within our framework, a squidget is a scene curve C whose appearance is related to the values of an associated set of scene attributes A .

We naturally perceive a visual scene like Figure 4(a) as an abstracted collection of curves \mathcal{C} like Figure 4(b). To make a change to the scene, users first draw a selection stroke S_s in-situ to visually specify a curve $C \in \mathcal{C}$. For instance, the orange stroke in Figure 4c visually corresponds to the corner of the dice. Formally, given a selection stroke S_s and an ensemble of scene curves \mathcal{C} , we need to compute a (partial) curve $C \in \mathcal{C}$ that best matches S_s in a perceptual sense (Section 4.1). Addressing the added complexity of partial curve matching is necessary: motivational examples, prior art [39], and our experiments, all confirm that users naturally draw only as much of a scene curve as needed to convey a desired scene edit. For instance, only the part of the avocado silhouette that changes is drawn to convey a local deformation in Figure 1(f).

Once C is selected, users can draw a manipulation stroke S_m to specify how C should appear as a result of manipulating its associated scene attributes. In other words, the squidget curve C serves as an interaction handle for its associated scene attributes. Formally, let A be the set of scene attributes and f the render function for a curve C , such that $C = f(A)$. The silhouette curve of the dice in Figure 4(b) for example, is a function of the dice shape, transform, and camera parameters.

Now, given a user drawn manipulation stroke S_m , we need to find changed attribute values A' , such that the edited squidget curve ($C' = f(A') \approx S_m$). For instance, modifying the translation attributes of the dice from A to A' in Figure 4(f) results in a curve C' where the corner of the moved dice closely approximates the user stroke S_m . A' thus represents the new set of attribute values which are applied to the scene, to result in the desired visual change to C (Figure 4(g)). Note, that a single user stroke can functionally serve as both S_s and S_m (Section 3.4).

3.1 Design Guidelines

We distilled the following guidelines from multiple unstructured discussions with artists, observations of user scene mark-up, previous experience with in-situ manipulation workflows of scenes, pilot-testing, and feedback over the iterations of our framework.

Curve Design: All user-perceived curves in a scene are potential handles for squidgets interaction. These are either imagined visual

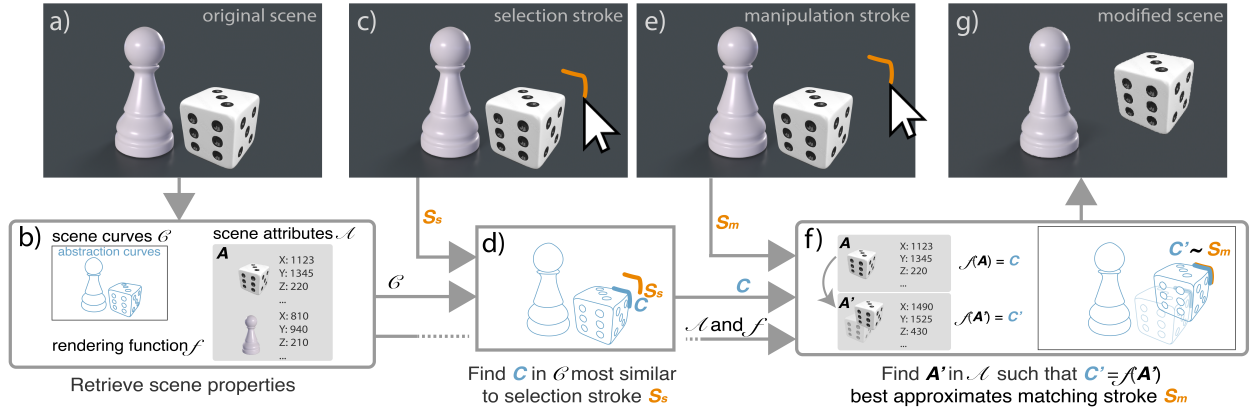


Figure 4: Squidgets enable manipulation of attributes that affect the visual appearance of a scene (a,b) through user selection S_s and manipulation S_m strokes (a single stroke can serve as both S_s, S_m). We infer the scene attributes A a user wants to edit by finding a curve C from among a set of scene curves that best match the stroke S_s (c,d: select). We then infer the change of values from A to A' needed so the resulting curve C' best fits the stroke S_m (e,f,g: manipulate).

abstraction curves or curves explicitly modeled and rendered as scene objects. The explicitly modeled rig curves in a scene typically capture the construction history of objects, often meticulously hand-crafted by artists to provide in-situ control over scene object attributes. These artists expressed the need for an overall interface where such rig curves could operate homogeneously with abstraction curves, but where such curves (bookmark curves) could be incrementally and efficiently created within the interface itself.

Stroke semantics: Users tend to draw simple and short strokes whenever possible to interact with the scene. Users often draw partial strokes with just enough context to unambiguously select a scene curve (e.g. only the corner of the eye shape in Figure 1(g)), or to clearly convey the desired manipulation (e.g. only deformed part of the avocado silhouette in Figure 1(f)). For small incremental edits, interaction can be streamlined by allowing a single stroke to serve as both the selection and manipulation stroke.

Interactive manipulation: Oversketches strokes typically convey rough edits. Finer control can be enabled by repeated oversketching, or by interactively dragging the stroke spatially.

Attribute redundancy: Complex scenes often have a large number of scene object attributes that can be manipulated to satisfy a desired manipulation stroke for abstraction curves. The oversketches stroke in Figure 1(b) for example, could be manifested by moving the lamp on the table as in Figure 1(d), but also by moving the scene camera in the opposite direction, or by non-uniformly scaling the lamp to have a much thicker base. It is thus important to limit the set of attributes considered for abstraction curve manipulation, or have these associations explicitly defined with rig curves and bookmark curves.

3.2 Inferring Scene Curves and Attributes

Given a 2D/3D scene, existing NPR and toon rendering techniques [5] allow us to efficiently and automatically create a collection of outline, silhouette, shading and feature poly-line abstraction curves of scene objects (Figures 1, 3). These curves comprise a scene abstraction \mathcal{C} . The attributes A for any curve $C \in \mathcal{C}$ are typically

those of its corresponding scene object. The rendering function f specifies how these attributes A define the visual appearance of C . For example, the control vertex positions of the avocado’s outline curve and the spatial 2D transform attributes of the avocado object define its shape and location in the scene in Figure 1(f). For shading discontinuities like the drop shadow in Figure 3, the attribute set expands from the objects casting the shadow to include the light attributes like the cone angle of a scene spotlight. Note that the set A associated with any curve C in our implementation is automatic and illustratively minimal.

3.3 Stroke to Select a Squidget Curve

User strokes perform two functions: selecting a squidget curve (segment) based on similarity to a stroke; and changing scene attributes to make the selected squidget curve approximate a stroke.

The similarity measure between a selection stroke S_s and a target curve $C \in \mathcal{C}$ is highly dependent on scene context and application domain. For instance, in scenes where the curve shapes are distinct (e.g. rounded chess pawns vs. corners of a cubical dice in Figure 5(a), middle), a similarity measure based on shape alone can suffice. A shape-only metric however, performs poorly in cluttered scenes with many curve shapes being good matches for an input stroke (Figure 5(b,c)). For example, in a scene with multiple chess pawns, a similarity measure that favors spatial proximity between C and S_s may be necessary to disambiguate selection (Figure 5(a), bottom). We present a novel perceptual curve similarity metric based on corner, spatial, and shape proximity in Section 4.1. Note that such an algorithm can be further customized to add other matching criteria like stroke thickness and color, and additional heuristics constraining the attribute values search space.

The user’s input stroke is matched for similarity against all curves in the scene, and the best matching curve is selected. Optionally, nothing can be selected if all matches are worse than a given threshold. Other mechanisms could also be leveraged for selected curve disambiguation, such as reducing the size of the curve set

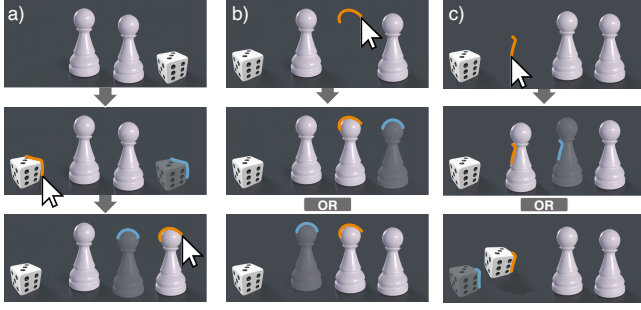


Figure 5: Disambiguation. Sketching unique shape features allows to unambiguously select and manipulate the dice in a one stroke interaction, even from a distance (a, middle). When multiple objects match the stroke, proximity is used for disambiguation (a, bottom). Ambiguities arise when two objects are an equally good match (b, c). In such cases, a two-stroke select-then-manipulate interaction is more suitable.

\mathcal{C} , e.g. using layer groupings of scene objects, or pre-selecting a specific scene object.

Note that a single point-and-click interaction is simply a degenerate case in our framework where the user selection stroke S_s is a single point. Thus, a user can select the gaze-controlling rig curve in Figure 1(g) with a point click (degenerate stroke) on or near any point of the eye-shaped curve, and then move the eye-shaped curve to a desired location by a subsequent click. Sketch strokes provide greater spatial and shape context for both curve selection and scene manipulation, than the spatial context provided by a single point. For example, the $<$ stroke drawn in different orientations can be used to both move and rotate the rig curve in Figure 1(g). Sketch strokes further enable the selection and manipulation of parts of the scene curves (Figure 1(f)).

3.4 Stroke as a Squidgets Manipulation Handle

Given a selected squidget curve, we assume the manipulation stroke S_m is representative of its appearance as a result of associated scene attributes being changed from their current values of A to A' (in other words $C' = f(A') \approx S_m$). This can be solved as an optimization where A' is close to A in value, and $f(A')$ is close to S_m . For the examples shown in our implementation, we are able to solve the above optimization as a best linear least squares minimization of transformation attributes and control point positions.

For small or incremental changes to a squidget’s handle curve C , a single user stroke can specify both selection (S_s) and manipulation (S_m) (Figure 4). For stroke manipulation/deformation in complex scenes (e.g. Figure 5(b,c)), selection and manipulation steps may necessitate two strokes: an over-traced stroke to unambiguously select a desired squidget; and a subsequent manipulation stroke.

3.5 Rig and Bookmark Squidget Curves

Artists increasingly hand-craft rig controllers: *in-situ* curve configurations pre-authored to control scene object attributes (Figure 1(g)). Such rig curves are selected and manipulated homogeneously using strokes within our squidgets interaction framework.

Scene manipulation and setup often entails interactive exploration, where key configurations of scene object attributes are incrementally discovered and need to be bookmarked for future use. We support this workflow using bookmark curves, drawn and explicitly associated with configurations of scene object attributes within the squidgets framework itself (Figure 1(h)). Specifically, a user-drawn stroke can be associated with the current values of a set of scene attributes to define a discrete bookmark curve. Selecting this curve using a squidgets framework at any point sets the associated scene attributes to their bookmarked values.

A number of such discrete bookmark curves can also be chained to define a continuous piece-wise interpolation of the curves and their scene attribute values. Two discrete bookmark squidgets C_0, A_0 and C_1, A_1 for example, can be combined into a weight $w \in [0, 1]$ interpolated, continuous squidget curve $interp(C_0, C_1, w)$ with similarly interpolated attributes $interp(A_0, A_1, w)$ for a set A' that could be the union or intersection of attribute sets A_0, A_1 . Continuous bookmark curves conceptually enable in-situ attribute keyframing (Figure 2(a)), visualized by a piece-wise linear path connecting a sequence of discrete bookmark curves (Figure 1(h)).

Any object selection and curve modeling tool can be used to create rig and bookmark curves with associated scene object attributes in 2D/3D. Our implementation specially facilitates the drawing of bookmark curves on planar canvases in 3D (Figure 8), where each canvas additionally provides a grouping for bookmark curves and their associate scene attributes.

4 Implementation

The squidget framework supports a range of implementations for a variety of applications and workflows. Here, we present details for an implementation¹ within the 3D animation system *Maya* 2024. User stroke input is provided by a mouse, track-pad, or Wacom Cintiq 24HD Touch tablet.

4.1 Perceptual Curve Segment Matching

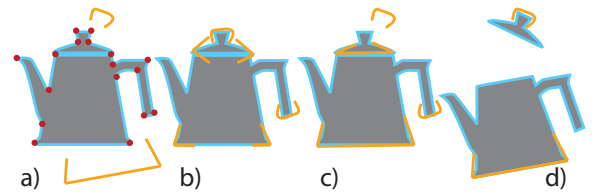


Figure 6: Strokes (orange) being matched to curves (cyan) with corners (red) (a). Strokes can be partially matched and aligned by ICP, and corner matching (b). Curves segments can be parametrically matched (c) and the best matched curves used to manipulate scene attributes (d).

A general similarity metric for curve matching is an ill-posed problem, depending on its use case and assumptions on the geometric properties of the curves. Curve matching for most interactive graphics applications are based on *corner*, *shape*, and *spatial*

¹<https://github.com/ohnooj/squidgets>

similarity [17]. Perceptually, we tend to align curves at corresponding sharp *corners* with matching smooth curve segments *shapes*. Depending on where the matching curves are drawn, *spatial* transforms may be needed to better align corresponding shape segments. We distill these observations into a curve matching algorithm with weights that favor different criteria, making the algorithm easy to adapt to specific use cases (see details in supplemental), or fit to training curve data (Figure 6). We assume P is an open curve for now and adapt our solution to closed curves later.

Terminology

Curve: we represent a curve P as a poly-line sequence of points $P = \langle p_1, \dots, p_n \rangle$.

Curve arc-length: is computed, for a curve P , as the sum of poly-line segments $al(P, k) = \sum_{i=2}^k \|p_i - p_{i-1}\|$.

Curve corners: for a curve P , let $C(P) = \langle c_1, \dots, c_m \rangle$, be indices for internal corner points p_{c_i} in sequence. The corners thus induce smooth curve segments $P_l = \langle p_{c_l}, \dots, p_{c_{l+1}} \rangle$ (for $l \in \{1, m-1\}$), the end curve segments being $P_0 = \langle p_1, \dots, p_{c_1} \rangle$ and $P_m = \langle p_{c_m}, \dots, p_n \rangle$. Segment arc-length for an internal curve segments P_l is thus $sal(P_l) = al(P, c_{l+1}) - al(P, c_l)$. At the extremes $sal(P_0) = al(P, c_1) - al(P, 1)$, and $sal(P_m) = al(P, n) - al(P, c_m)$.

Curve corner position+orientation: we define the position $o(P, i) = p_{c_i}$ for a corner point i in the set of corners $C(P)$ of curve P . A coarse tangent-like orientation at this point is given by the vector v between its adjacent corner points $t(P, i) = v(p_{c_{i+1}}, p_{c_{i-1}})$, where $v(a, b) = (a - b) / \|a - b\|$. For the extreme curve segments we use p_n as $p_{c_{m+1}}$ and p_1 as p_{c_0} .

Corner correspondence algorithm

Given two curves P and Q , we first find an optimal correspondence between the set of corners $C(P)$ and $C(Q)$, that act as sparse anchors in perceptual shape matching [17]. We use a dynamic programming formulation that minimizes the matching energy of corresponding corners on P and Q .

Let $M(P, i, Q, j)$ be the *corner matching energy* for the i^{th} corner in $C(P)$ to correspond to the j^{th} corner in $C(Q)$, defined as a weighted sum of a spatial alignment energy $A(i, j)$ needed to move and coarsely align the corners, and a coarse shape energy measured by a local disparity in curve arc-length of their adjacent curve segments. In other words:

$$M(P, i, Q, j) = w_t * A(i, j) + w_s * (|sal(P_{i-1}) - sal(Q_{j-1})| + |sal(P_i) - sal(Q_j)|).$$

The corner alignment energy $A(i, j)$ above is a weighted sum of a translation $Tr(i, j) = o(P, i) - o(Q, j)$, and a rotation $Ro(i, j)$ that rotates vector $t(P, i)$ to $t(Q, j)$.

We need to adjust the above computation of coarse shape energy and corner alignment at end curve segments, to allow partial curve matching. For example, when $i = m$ let parameter $s = \min(sal(P_m), sal(Q_j))$, and we use the point q on Q at parameter $al(Q, c_j) + s$ as corresponding to end point p_n of P . The shape energy term above $|sal(P_i) - sal(Q_j)|$ is then replaced by $|sal(P_i) - s|$, and the corner alignment $A(i, j)$ uses a translation $Tr(i, j) = o(P, i) - q$, and a rotation $Ro(i, j)$ that rotates vector $t(P, i)$ to $v(q, q_{c_j})$. The case for the beginning segment P_0 , and the two end segments of Q are treated similarly.

Now let $E_C(P, e, f, Q, g, h)$ be the energy to match a sequence of corners from e, \dots, f in $C(P)$ to corners g, \dots, h in $C(Q)$. We can define this energy using Dynamic Programming as:

$$E_C(P, e, f, Q, g, h) = \min_{i \in e, f, j \in g, h} [E_C(P, e, i-1, Q, g, j-1) + E_C(P, i+1, f, Q, j+1, h) + M(P, i, Q, j)].$$

Curve segment matching algorithm

The matching corners provide anchors for a parametric correspondence between poly-line curve segments P_l (for $l \in \{1, m\}$) and their matching segments in curve Q (WLOG $|C(P)| \leq |C(Q)|$). In the event that P has no corners, we find a best-fit rigid transform that minimizes the distance between points on P and the curve Q using the iterative closest point ICP algorithm [33]. We then find the closest pair of points on ICP aligned P and Q and treat them as corners on P and Q respectively to provide an anchor for parametric correspondence. We also note that for the end segments P_0 and P_m , we use an arc-length parameterization to define a correspondence truncated by the segment of P or Q with a shorter arc-length. We now have two sets of parametrically matched curve segments in P, Q induced by P_l (for $l \in \{0, m\}$). Note again, for the case where P has no corners we re-combine the closest-point-pair induced segments after parameter correspondence, so they are treated as a single smooth curve segment. For each corresponding pair of segments P_l and Q_l , we resample the sparser poly-line to match the point count d of the denser segment. We now compare these segments simply using a matching set of corresponding points where $P_l = \{pl_1, \dots, pl_d\}$ and $Q_l = \{ql_1, \dots, ql_d\}$. The matching energy M_l can be combined as a single best-fit rigid transform A_l to align the point sets, and an average of residual distance-squared between corresponding points after the alignment transform. In other words,

$$M_l = w_a * \|A_l\| + (w_p/d) * (\sum_{i=1}^d \|pl_i - ql_i\|^2).$$

The overall curve matching energy $E(P, Q)$ between curves P and Q is a sum of the matching energies of their corresponding curve segments $\sum_{l=0}^m (M_l)$.

For a given application, the best match between P and a set of curves Q^1, \dots, Q^k is then simply $\min_{i \in \{1..k\}} (E(P, Q^i))$, subject to setting the weight parameters to provide control over the impact of corners, spatial translation, rotation, and the shapes of the curves being matched.

For closed curves, we first perform a rigid ICP transformation to roughly align them. We then run the stroke matching above with a high spatial weighting to only allow small local movements of curve segments to align corners and curve segments.

4.2 Squidget Curve Creation

While rig curves are user pre-authored 3D scene curves, abstraction and bookmark curves need to be created or set-up by our system.

Abstraction Curves

We use NPR toon-rendering in *Maya* to create imagined curves from scene objects (Figure 7(b)). Arbitrary object attributes can visually change these toon curves. Our implementation restricts these attributes to object transforms and object vertex positions for illustration. We further note that toon-rendering is view-dependent and toon curves can change discontinuously upon object or camera transformation. We thus create 3D curves \mathcal{C} from 2D toon-rendered outlines given the current scene and camera parameters, as if they

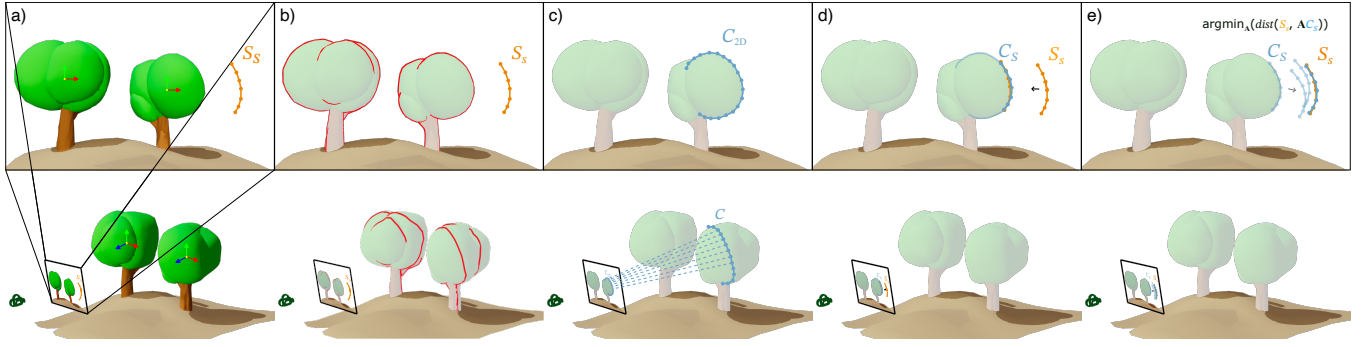


Figure 7: A user-drawn selection stroke is shown in screen space (top row, bottom row shows a world space view) (a). Maya toon render outlines are converted to abstraction curves in world space (b). Each abstraction curve C projected to screen space C_{2D} (c) is aligned with S (d), transforming C_S to S (e). The object corresponding to the best matching curve is selected for manipulation.

were pasted on their corresponding 3D objects, before any interactive scene manipulation begins.

Bookmark curves

Like rig curves, bookmark curves can be arbitrarily-created 2D/3D curves in *Maya*. Once created, they can be associated with any set of scene attributes within our framework.

Bookmark canvases: We provide support for the streamlined creation and scene attribute association of bookmark curve groups using canvases. A canvas is a plane or surface that is positioned in 3D and associated with a set of scene attributes. Users draw a screen stroke projected onto the closest canvas to create an in-situ 3D bookmark-curve and automatically map the current values of the scene attributes to the curve. Multiple squidgets can inherit the same scene attributes from the parent canvas but with different values such that groups of bookmark curves can be managed via different canvases and spatially re-arranged in the scene by the user to reduce visual clutter. Visibility of canvases can also be toggled manually or automatically (e.g. based on proximity to a hovering pen position, or on how oblique a canvas is to the current scene view). We experimented with other attribute selection schemes such as automatically selecting recently changed attributes, but found our proposed canvas-centric approach to provide a good balance between efficiency and flexibility in bookmark curve creation.

Discrete/Continuous Bookmark Curves: Bookmark curves can be linked in sequence by drawing a stroke that crosses discrete bookmark curves. This continuously interpolated bookmark is visually indicated by a poly-line path through the mid-points of discrete bookmark curves in sequence. The path further acts like a UI slider that provides linearly interpolated control between the scene attribute values associated with adjacent discrete bookmark curves.

4.3 Squidget Curve Selection

The curve matching algorithm (Section 4.1) matches the selection stroke S_s to the best matching 3D squidget (abstraction/ rig/ bookmark) curve C projected into screen space Figure 7. Note that for 3D stroke input in AR/VR the matching takes place in 3D itself.

Corner, Spatial, and Shape Weights: Our algorithm in Section 4.1 allows us to differentially weight the importance of corners, a

spatial transform, and the shape of curve segments in curve matching. A number of factors influence the weight settings, including: input device (trackpads and mice have greater noise than pen/tablet and thus a lower corner weight); separate select and manipulation strokes (the spatial weight for a dedicated selection stroke is low as we expect users to closely trace over the desired curve); nature of curves (for largely straight line segments in architectural applications the shape weight can be set as low, for known smooth curves the corner weight can be set low). In a number of scenarios, assumptions about the scene and/or interactions allow to simplify aspects of the algorithm, as described in the supplemental materials.

We hand-tuned weights for application and stroke workflow (e.g. a 2-stroke workflow ($S_s \neq S_m$) expects an overtraced selection stroke and thus weights spatial proximity heavily). In scenes with many squidget curves, spatial proximity of the user strokes to the desired select/manipulate curves provides the most reliable behavior, given the corner and shape sensitivity of strokes to sketching inaccuracy. Weights can also be learned using training examples of squidgets scenes and user strokes, or interactively set by a user.

4.4 Squidget Manipulation

Once a squidget curve is selected, we need to compute the change to its associated scene attribute values, such that the resulting squidget curve best matches the manipulation stroke S_m . This search for optimal attribute values depends on the associated attributes. For example, for shape attributes like object vertex positions, the vertices can closely approximate the manipulation stroke by directly conforming object vertices to the shape of the manipulation stroke. For object transform attributes, there is an analytic best-fit transform that conforms a squidget curve to a manipulation stroke [38]. Discrete bookmark curves simply snap attributes to the bookmark curve closest to the manipulation stroke and continuous bookmarks are interpolated based on where the manipulation stroke intersects its poly-line path. Abstraction and rig curves can have complex relationships between their associated scene attributes and the curve, requiring a neighborhood search of attribute values, to find the curve that best matches the manipulation stroke.

For interactive control of squidget curves, users can employ a 2-stroke technique where S_s and S_m are separate strokes, a 1-stroke

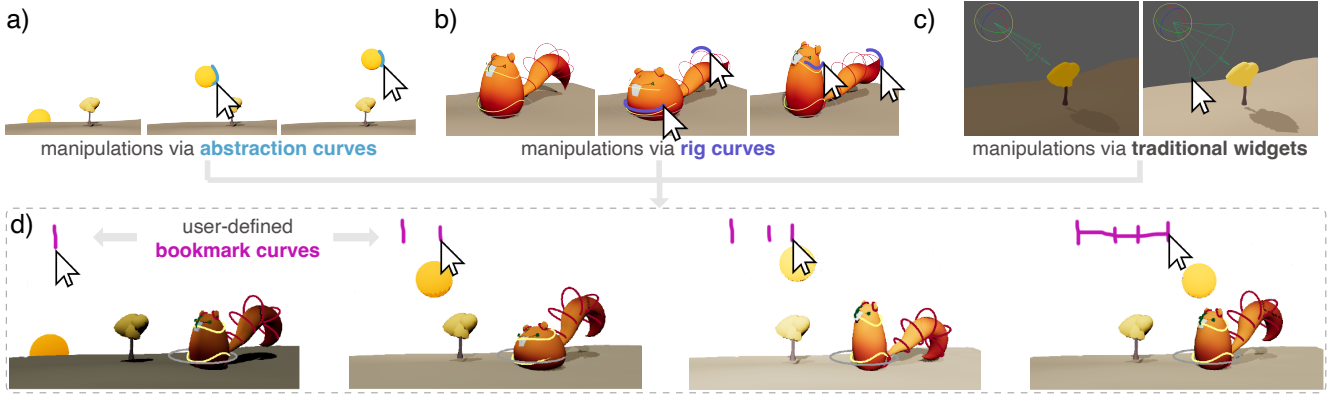


Figure 8: Top: To modify a scene, visual artists can freely choose between manipulating scene objects using abstraction curves (a), rig curves (b), or traditional widgets (c). Bottom: The artist can save the scene (and hence all the scene parameters) by sketching a bookmark curve (d, left). Iteratively modifying (a-c) and bookmarking scene configurations (d) allows the artist to create multiple versions, which can later be referred to, or interpolated by linking the bookmark curves (d, right).



Figure 9: Within a VR environment, strokes are drawn to control a wood-chopper by moving the axe they hold (a-b) and adjusting the body pose (c-d).

technique where $S_s = S_m$, or a hold-and-drag technique to interactively refine the attribute values after drawing S_m , like a virtual slider that incrementally changes scene attributes or interactively changes the object transform.

It is important to note that when squidget curves are matched in screen space, all manipulations happen orthogonal to the camera.

5 Applications

Most **2D design applications** already support sketch input, and can thus seamlessly incorporate squidgets for quick editing of scene parameters using abstraction curves, as illustrated in Figure 1(f). In **3D design applications**, in-situ control through squidgets can be a powerful approach compared to traditional widgets. For instance, in a *VR environment* sketching strokes that match the scene’s abstraction curves can control characters, e.g. a stroke indicating where an axe should move Figure 9(a,b). Character and face rigs often have many rig curve handles for each part of the body, and posing often requires manipulating multiple handles together. Squidgets can facilitate *character and face posing* by allowing artists to directly work with face (Figure 1(g)) or character rig curves (Figure 8(b)), by drawing the shape of the rig handles. Using bookmark curves, artists can also author their own “rig” curves, as illustrated in Figure 10.

Combining squidgets enables **rich authoring workflows**. Figure 8 illustrates how an artist (a) manipulates the sun’s location by drawing arc strokes to match the sun’s abstraction curve, (b) poses the squirrel by sketching strokes that match rig curves of its tail,



Figure 10: User-authored bookmark curves can be associated with discrete configurations of face attributes to capture a smile (left) and a laugh (right); and further combined into a continuous bookmark curve squidget, where the curves and related face attributes can be interpolated (center).

head and body, and (c) changes the scene lighting using traditional widgets. When satisfied with the scene, (d, left) the artist creates a bookmark curve, i.e. a simple tick mark for easy reference to this configuration. Then, (a-c) they continue to modify the scene, (d) adding a bookmark curve for each new configuration. Through this workflow, the artist authors key frames of a short animation where the sun rises and the squirrel bounces, which they can interpolate by creating a continuous bookmark squidget (d, right).

6 Evaluation

We evaluate our framework through a controlled experiment (§6.1), and an informal qualitative evaluation with visual artists (§6.2). In the interest of space, we provide a summary; a thorough description and report can be found in the supplemental materials. The study was approved by our institutional ethics board.

6.1 Controlled Experiment: Manipulate Objects

We ran a within-subject experiment ($n=12$) to evaluate whether people understand the squidglets concept and the usability of using sketch input for manipulating scene parameters. We recruited 12 participants who were familiar with pen input but with little experience using 3D design software through our institutional channels and word of mouth. The study lasted ~ 45 min; participants received CAD\$30. The study ran on a computer laptop plugged into a Wacom Cintiq 24HD screen tablet; and was implemented with the Maya software with most features stripped out from the view to best approximate a generic, traditional vector graphics editor.

6.1.1 Study Design. Participants completed two sets of scene manipulation Tasks: *translate* and *deform* a graphical object to match a target. The Techniques included a *baseline* with traditional graphical manipulation tools as found in vector graphics software (*Maya* selection/translation tool for translation tasks and single-vertex soft-selection with various deformation falloffs for deformation tasks), and different squidglet variations: *1-stroke* (selection and manipulation are performed within a single input stroke), *select→drag* (selection with an input stroke, upon which the object sticks to the cursor to be further dragged around), and *select→manipulate* (an initial stroke is used for selection; subsequent drawn strokes perform manipulations to the selected object).

For the translation task, we varied difficulty according to the ambiguity that additional distractors would introduce for selection or manipulation: *SpatialDifficulty* refers to how far each distractor image is from the target axis location; *ShapeDifficulty* determines how visually similar in shape the distractor is to the task image. For the deform task, *DeformDifficulty* modeled task difficulty as a function of type of applied deformations. See supplemental material for details on the study design and procedure.

6.1.2 Study Procedure. (1) The experimenter first obtained consent, then explained to participants that they were tasked with helping design a summer drink poster by moving or changing the shape of fruit stickers, using different techniques. (2) Because the study did not evaluate discovery but rather proficiency with the tool, the participants were allowed to practice for as long as desired with each Technique \times Task. (3) Trial repetitions within each Task \times Technique block were presented in a randomized order. For each trial, Time, Operation, and Undo were collected. After each block, participants were instructed to complete a NASA-TLX questionnaire [20]. (4) Finally, participants were invited to fill out a post-study evaluation asking them to rank the interactive techniques for each task, along with open comments.

6.1.3 Results. We report the 95% bootstrapped confidence intervals (CI) on means to assess effects [13] along with qualitative data. We focus on key insights; see supplemental for a detailed report.

Performance: squidglets are comparable to the baseline most of the time. Figure 11 shows the effect of Technique and task Difficulty on performance. Overall, the baseline was generally comparable to squidglets for both completion time and operation counts, across tasks, with a few exceptions, discussed below.

For translations, distractors challenge 1-stroke. As we anticipated, we find strong evidence that *1-stroke* presents notable challenges for move tasks with a *hard* *SpatialDifficulty*. The

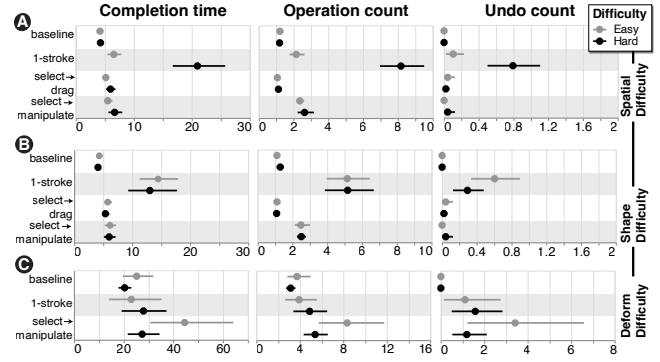


Figure 11: Effect of Technique and Difficulty on measures for the translation (A-B) and deform (C) tasks.



Figure 12: Move Task: P7's sketch strokes for *1-stroke*, *select→drag*, and *select→manipulate*. More in supplemental.

strokes the participants drew (Figure 12) are mostly arc-shaped strokes that match a small portion of the objects' silhouette. While this strategy largely works when strokes serve one of two functions (selection or manipulation), for *1-stroke* there is increased challenge in the presence of distractors. When distractors were spatially close to the target, participants drew "incremental" simple strokes to move the object progressively (Figure 12, bottom row). They did not systematically attempt to fine-tune the stroke to incorporate distinguishable features for disambiguation. Participants rated *1-stroke* as more mentally and physically demanding, more effortful, more frustrating, and less performing than the other techniques for moving tasks. However, two participants rated it as the most preferred, because "being able to do things in one stroke was overall much smoother compared to selecting." (P3).

For translate tasks, different techniques will suit different people. The best technique for move tasks depends on personal preferences. The most popular, *select→drag* (7/12 rated as most preferred), was liked for its efficiency — "The fastest and the most intuitive, and also had very good control in where you want to move the object" (P2), and continuous interaction — "You get to see real-time where the object is." (P10). Those who preferred the baseline (5/12) also mentioned high precision and control. Participants appreciated both techniques "were very simple and easy to understand right away." (P9). They were more mixed about *select→manipulate*, pointing to downsides such as "having the 2 stroke was very redundant compared to the other options" (P8). While separating the select and

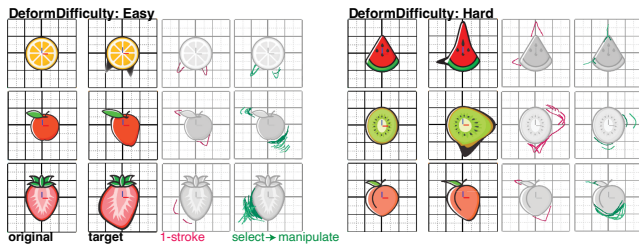


Figure 13: Deform Task: P7's sketched strokes for *1-stroke*, and *select→manipulate*. See supplemental for more.

manipulation strokes offers advantages conceptually, participants did not recognize these benefits during the controlled experiment.

For deform tasks, a tool (baseline) is used iteratively while sketching the target (squidgets) is hit-or-miss. The *baseline* resulted in very few (if any) undo operations, but required about as many operations as *1-stroke* and *select→manipulate*. This indicates that, with the *baseline*, participants opted for *subsequent* manipulations instead of reversing the last operation. For squidgets, undo operations were more common, suggesting that participants imagined that the target deformation could be achieved in one go: “The deform-by-stroke tool feels more natural because that’s how I anticipate deformation in my head” (P10). However, due to the challenges in drawing curves which perfectly match the intent, this required multiple attempts, especially for lay users with no particular expertise with digital drawing.

Deformations types do not correlate with task difficulty. DeformDifficulty had no effect, or even a countereffect on performance measures, suggesting that our metric is a poor proxy for anticipated difficulty. We found that participants struggled most with trials for which they had a hard time identifying what was deformed and how (i.e. peach & apple, Figure 13). Note that this is an artificially introduced and extraneous step which was inevitable for a controlled experiment but would not occur in a real-world scenario, where people know what target shape they want.

For deform tasks, the sense of control is affected by the predictability of the technique and the dexterity with input. While some participants found that the *baseline* offers more control and predictability, others felt the opposite was true because vertex-based distortion was constraining: “[the baseline] is my least favorite because you have to evaluate which width to use.” (P10), and having poor predictability: “it’s hard to map a larger deformation of the image to just a single vertex movement in my head.” (P11). Similarly, some found *1-stroke* offers control and is highly predictable, whereas others found it did not behave as expected, e.g. “it moved parts that I did not necessarily want moved” (P8). Opinions were also split for *select→manipulate*, with several finding that it gave “finer control” (P10), but a few finding it was difficult to understand.

For deform tasks, *select→manipulate* has strong potential. *select→manipulate* was a popular choice for deform tasks. One of its main advantages is fine control over the portion of the stroke concerned with the manipulation, which several participants noted was powerful. P11 articulated it best: “I like it best because I feel like it provided the most amount of control - I can control the length of the curve that I want to deform as well as how much I want to deform it.”

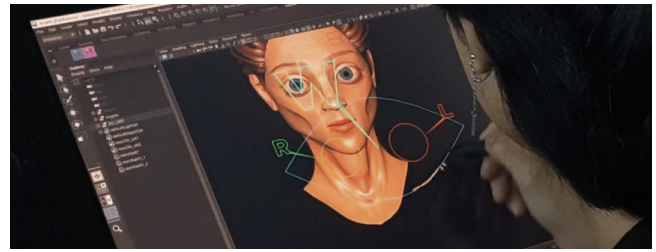


Figure 14: An animator uses rig curves to control a character.

6.2 Impressions From Visual Artists

We invited two animators for interactive sessions to gather their impressions of the squidget framework on 3D scenes (Figure 14) in a formal 1-hour session where they explored all variations of squidgets while thinking aloud, followed by a semi-structured interview to discuss their experiences. A1 is a professional animator with ~25 years of experience in the commercial and visual special effects industry, and whose primary tool is Maya. A2 is a researcher in graphics with 2 years of experience doing facial animation on Faceware² and Maya.³

Conceptually, squidgets show promise. Animators found that squidgets were well suited for creative workflows, qualifying the techniques as “very useful”, “intuitive” and “very neat”. They found sketch input *in-situ* is a powerful mechanism that “brings everything to the center of the scene” (A2), which helps maintain the creative flow. This contrasts with traditional interfaces: “as soon as you go into the interface, you’re breaking what you’re doing” (A1). Animators found it to be “very satisfying” when the system correctly interpreted their intent from “just putting a stroke down” (A2). They showed excitement for the conceptual approach: “a software that captures your strokes, and eventually learn to predict what you’re trying to do [...] That’s a perfect piece of software” (A1), albeit currently, the implementation needs improvements (see below).

Projecting a 2D stroke in a 3D scene. In our controlled experiment, we opted for 2D scenes due to the difficulties of 3D navigation, especially for lay users. Animators worked with 3D scenes, allowing us to gain insight into challenges and mitigating approaches. Our implementation interprets 3D transformations with respect to the camera view, which animators found “mentally confusing” (A1) and misaligned with their expectations. They suggested alternative constraints to guide the transformations, such as axis constraints (x,y,z planar directions), or physics-based heuristics that result in “plausible movements” (A2) like moving along a wall or undulating on a water body. A1 indicated that “even the stuff that is a little bit extraordinary is based on real world physics”. Other scene objects were also indicated as useful guides to sketch over, for instance for assembly models, where sketching over objects will allow things to fit perfectly together “like a Tetris” (A1).

Disambiguation. Animators commented on disambiguation challenges exacerbated by near-by or too similarly shaped squidgets. They strove to balance the effort required to draw enough details or unique features for disambiguation. *select→manipulate* and *select→drag* achieved this goal successfully. One strategy we

²<https://facewaretech.com/>

³Four other expert animators also freely explored squidgets in earlier informal sessions, which allowed us to refine our framework.

observed aimed to optimize for selection, then use minimal strokes for translations: *“I tried to draw over the object; and then, it would just be like dashes. Very quick ones. [...] I like how even if you were lazy and don’t want to sketch the full curve, you can also get there”* (A2). Animators also strategically picked parts that are simple to replicate to achieve precision in a two-stroke workflow. The squidget paradigm acts *“differently than how you’ve selected for, you know, 30 years”* (A1), which can hinder adoptability. But animators indicated that sketch selection would work well to select distant objects when these are visually distinct.

Squidglets workflow #1: Coarse and granular changes. A1 discussed combining squidglets for coarse and granular changes, noting that *1-stroke* and *select*→*manipulate* are efficient for large movements but lack granular precision, while *select*→*drag* affords precision but is laborious for large manipulations. They described an envisioned workflow where *1-stroke* or *select*→*manipulate* interaction would first *“snap the object in position”*, followed by refinements by dragging while swapping between translation and rotation to *“just slightly move it exactly where it needs to be”*.

Squidglets workflow #2: Pose and bookmark in-situ. A1 and A2 quickly picked up on the potential of squidglets for animation. They experimented with posing a character [19] or a face using abstract or rig curves, keying that pose with a bookmark curve, repeating the operation multiple times with a new pose and key. Then, they linked the bookmark curves and scrubbed through the interpolated animation using *select*→*drag*. Within this workflow, the ease with which they could “re-key” or add a new pose without having to switch between the scene and a timeline would *“save artists hundreds of hours”* (A1). A2 appreciated how the connected bookmark curves *“visualizes the timing in the spatial domain”* akin to notations that artists use to indicate durations between frames; also reminiscent of timelines based on motion flow [14, 26, 42].

Bookmark curves as a personalized language. Animators quickly appropriated bookmark curves as a personally-defined visual language. When bookmarking different character poses A1 and A2 drew the lines of action [19] *“following the spine and the legs of the character”* (A1), but later adopted minimal lines, as if the pose of the character was not necessary to fully “encode”. Indeed, *“making curves simpler will make it easier to query [...] I think just having the position is enough for me to specify which frame I want”* (A2). They also mentioned that custom markers would be useful for objects whose absolute position does not change, like faces.

Scenarios of usage. Concrete scenarios where animators see squidglets as a valuable alternative to their current practice include modularizing keyframe animation by creating sequences for smaller modules and controlling each squidget module with a meta-squidget; and the creation of bookmark curves for intangible, difficult to reach scene parameters (e.g. lighting). A2 also felt that *select*→*manipulate* interaction with abstraction curves would be particularly suitable for exaggeration effects, like squashing and squeezing.

Areas for improvement. Squidglets have the potential to cause visual clutter if too many are created. Finally, because multiple different squidglets can be coupled to the same scene attributes, changing one squidget may affect the states of the others, causing concerns that such cases could yield undesirable results.

7 Discussion

Our prototype implementation and user studies affirm the feasibility of squidglets interaction and that the concept of squidglets, as a powerful and natural mechanism to express scene edits, was understood and appreciated by both experts and lay users. The studies confirmed our design guidelines and provided further insight into the squidget paradigm: (i) input device fidelity, drawing skill, and the choice of strokes for selection and manipulation, all affect the ability to select and perform desired complex scene edits; (ii) oversketching naturally aligns with 1-stroke interaction, but is only effective in simple scenes where disambiguating selection is easy, or when the desired edit is small (i.e. the selection and manipulation strokes are very similar); (iii) the *select*→*drag* workflow was appreciated for allowing a quick, coarse stroke for selection and manipulation that can be interactively refined by dragging; (iv) an important difference between novice and expert users lied in their ability to draw uniquely identifying parts of strokes that aided selection and manipulation of scene objects. Experts also liked the gestural nature of squidglets interaction for quick scene exploration and bookmarking, and saw it as complementary to existing tools.

We present squidglets as a conceptual framework for interaction. Our implementation of squidglets within *Maya* is but one instance of a squidglets workflow. As our results show, there is potential within the framework to fine-tune the use of multiple strokes to disambiguate selection, vary the associated scene attributes, and the resolution and precision with which they are edited.

7.1 Limitations

Ambiguous inverse control: Conceptually, squidglets capture an ill-posed and ambiguous inverse control problem. Despite this, constraining the number of squidget curves and their mapping to scene attributes can make interacting with squidglets predictable and satisfying. There may be inherent redundancy in some scene attributes. For example, a stroke indicating a larger sphere in *Maya* can be realized by a change to the sphere’s radius, its scale transform, or an infinite combination of the two. Such redundancy can be mitigated by regularization or authoring explicit bookmark curves that clarify the attributes to be controlled by the squidget. Our prototype implementation also makes a number of simplifying assumptions such as only controlling transform and shape attributes via abstraction curve squidglets. Given recent research on data-driven inverse rendering, we hope to solve general scene attribute manipulation for abstraction curve squidglets as an optimization learned from forward rendering simulations.

Confidence and predictability: The curve matching interaction may leave users with little transparency regarding output prediction and can leave users guessing what will happen. Users commit to drawing an entire stroke before receiving feedback on their actions, and if the result is not what a user wants, there may be little guidance on how to draw a more accurate stroke for their desired task. Several approaches can be explored for increasing confidence and predictability as follows.

For squidglets selection: The *1-stroke* workflow is best suited (1) when distinct stroke gestures are unambiguous, i.e., there are few, visually distinct squidget curves, or (2) for incremental changes, i.e., the difference between the selected and manipulated curve is

small. While more efficient, this workflow can be unpredictable, especially in scenes with many squidgets. A correction step can be considered where the user picks among the top few visualized matches. The *select*→*manipulate* workflow is very predictable if the selection stroke traces over even a small portion of the desired curve (akin to click-to-select). Visualizing the best curve match to the partially drawn stroke in real time can increase user confidence.

For squidget manipulation: Dynamically rendering a ghost preview of the manipulated curve that best matches the user stroke can communicate how squidgets interprets input in terms of scene attribute changes. *select*→*drag* allows interactive scene refinement through spatial dragging; it does not allow refining the stroke shape though, something which could be considered using oversketching before committing a scene edit. Finally, a squidgets interaction history can also be used to reinforce repeated selection and manipulation behaviors, and penalize those that are undone.

2D input: While our framework supports 2D/3D scene curve manipulation, users interact with 2D strokes drawn in screen space (with the exception of mid-air strokes drawn in AR/VR). As a result, scene attributes are edited to produce view-depth preserving edits of the selected scene curve. The onus is thus on a user to pick good views to draw strokes, avoiding views where desired attribute edits significantly impact the scene depth.

Deformation difficulty: We chose simple bell-shaped deformation tasks for fair comparison between the proportional modification baseline tool in Maya and our squidget implementation. We acknowledge though that squidgets allow more expressive shapes with any number of waves and swirls, than does our baseline. Future work should investigate alternative measures that more closely align with difficulty prediction.

7.2 Future Work

Multi-stroke input: Squidgets, like single-stroke gestures, currently use a single input stroke for selection/manipulation. Given the ability to draw multiple strokes before they are processed for selection/manipulation can greatly increase the expressive power of squidget interaction. For example, a pair of parallel strokes could better select thin tubular objects (a wine glass stem), or three intersecting strokes could indicate a 3D frame of reference providing better control over attributes that change scene depth.

Stroke attributes: Our experiment only focused on geometric shape matching for scene curve selection and manipulation. Sketch strokes themselves have a rich set of attributes like weight, color and style, that could be exploited to further control the selection and manipulation of scene curves and associated attributes.

Bookmark curve extensions: Many of the animators in Section 6.2 saw the value of bookmark curves to bootstrap and improve in-situ graphical interfaces. Bookmark curves, currently used to represent discrete scene configurations, or interpolated in sequence can be embedded in-situ on manifolds to allow a richer exploration space of scene attributes [3]. Bookmark curve templates that transfer curves and associated scene attributes across objects is also exciting.

3D Navigation: A crucial element of graphics workflow is camera navigation. Though we did not explicitly implement navigation

methods (and left users to use default camera controls), squidgets lays a foundation to build upon stroke-based 3D camera navigation. Rig curves and bookmark curves can be pre-authored like typical 3D navigation widgets, to allow camera navigation control, or for constrained 3D navigation to key 3D configurations or along transformation trajectories. Abstraction curves automatically inferred from a 3D scene-view are particularly effective for view-dependent on-screen navigation; curves can provide useful alignment and docking constraints in a scene to further aid rapid 3D navigation. Further exploration would be required to understand user tendencies to explicitly convey camera actions with strokes.

8 Conclusion

Squidgets present a grand unified vision for stroke-based interaction that leverages human perception of images as a collection of real or imagined curves. Our prototype only implements a small subset of squidget interactions in 2D/3D scenes with 2D/3D stroke input.

While our studies do also point out challenges in squidget interaction for scene attribute control, we believe these can be addressed by better constraining curve selection and attribute manipulation to the task context and application domain. Squidget do place expectations on user drawing skill, but we believe these can be reduced by simple and unique squidget curve shapes, stroke beautification techniques, incremental refinement via multiple strokes, and visual feedback that guides and snaps users towards better strokes.

Squidgets are a novel interaction framework that couples our natural tendency to indicate changes to a visual scene by drawing strokes over it. We have shown using our implementation and studies that such an approach is both viable and promising and hope that this work will fuel further work in direct, in-situ, sketch-based scene interaction.

Acknowledgments

We would like to thank: our user-study participants for providing insightful feedback on our tool; Jenny Oh for help organizing our user study; Damien Masson for helping create data figures; Zhecheng Wang for helping with the curve matching algorithm.

Park Environment in Figure 11 and *Cartoon Bedroom Set* furniture in Figure 1 were modeled by @anim_matt. *Valley Girl* characters in Figure 1 and Figure 10 and *Angela* in Figure 14 were modeled by ©Chris Landreth. *Squirrels Rig* in Figure 11: Rig or Material used with permission (©Animation Mentor 2022). No endorsement or sponsorship by Animation Mentor. Downloaded at www.animationmentor.com/free-maya-rig/. VR environments were rendered in ©Gravity Sketch. *FRUIT ICONS* for the userstudy and in Figure 1 were designed by NicoDigitalStore. This research was supported by NSERC.

References

- [1] Georg Apitz and François Guimbretière. 2004. CrossY: a crossing-based drawing application. In *Proc. UIST '04*, 3–12. doi:10.1145/1029632.1029635
- [2] Andreas Aristidou, Joan Lasenby, Yiorgos Chrysanthou, and Ariel Shamir. 2018. Inverse Kinematics Techniques in Computer Graphics: A Survey. *Comp. Graph. Forum* 37 (2018), 35–58. doi:10.1111/cgf.13310
- [3] Rahul Arora, Ishan Darolia, Vinay P Namboodiri, Karan Singh, and Adrien Bousseau. 2017. Sketchsoup: Exploratory ideation using design sketches. In *Comp. Graph. Forum*, Vol. 36. Wiley Online Library, 302–312. doi:10.1111/cgf.13081

- [4] Seok-Hyung Bae, Ravin Balakrishnan, and Karan Singh. 2008. ILoveSketch: as-natural-as-possible sketching system for creating 3d curve models. In *Proc. UIST '08*. 151–160. doi:10.1145/1449715.144974
- [5] Pierre B  nard and Aaron Hertzmann. 2019. Line Drawings from 3D Models: A Tutorial. *Found. Trends. Comput. Graph. Vis.* 11, 1–2 (sep 2019), 1–159. doi:10.1561/06000000075
- [6] Lonni Besan  on, Paul Issartel, Mehdi Ammi, and Tobias Isenberg. 2017. Mouse, Tactile, and Tangible Input for 3D Manipulation. In *Proc. CHI '17*. 4727–4740. doi:10.1145/3025453.3025863
- [7] Sukanya Bhattacharjee and Parag Chaudhuri. 2020. A Survey on Sketch Based Content Creation: from the Desktop to Virtual and Augmented Reality. In *Comp. Graph. Forum*, Vol. 39. Wiley Online Library, 757–780.
- [8] Eric Allan Bier. 1986. Skitters and Jacks: Interactive 3D Positioning Tools. In *Proc. Workshop on Interactive 3D Graphics '86*. 183–196.
- [9] Eric Allan Bier. 1990. Snap-Dragging in Three Dimensions. In *Proc. Workshop on Interactive 3D Graphics '90*. 193–204.
- [10] Doug A. Bowman, Ernst Kruijff, Joseph J. LaViola, and Ivan Poupyrev. 2004. *3D User Interfaces: Theory and Practice*. Addison Wesley Longman Publishing Co., Inc., USA.
- [11] Byungkuk Choi, Roger Blanco i Ribera, J. P. Lewis, Yeongho Seol, Seokpyo Hong, Haegwang Eom, Sunjin Jung, and Junyong Noh. 2016. SketchiMo: Sketch-Based Motion Editing for Articulated Characters. *ACM Trans. Graph.* 35, 4, Article 146 (2016). doi:10.1145/2897824.2925970
- [12] Julie Dorsey, Songhua Xu, Gabe Smedresman, Holly Rushmeier, and Leonard McMillan. 2007. The Mental Canvas: A Tool for Conceptual Architectural Design and Analysis. In *Proc. of Pacific Conference on Computer Graphics and Applications*. 201–210. doi:10.1109/PG.2007.64
- [13] Pierre Dragicevic. 2016. Fair statistical communication in HCI. *Modern statistical methods for HCI* (2016), 291–330.
- [14] Pierre Dragicevic, Gonzalo Ramos, Jacobo Bibliowicz, Derek Nowrouzezahrai, Ravin Balakrishnan, and Karan Singh. 2008. Video browsing by direct manipulation. In *Proc. CHI '08*. 237–246. doi:10.1145/1357054.135709
- [15] Koos Eissen and Roselien Steur. 2008. *Sketching: Drawing Techniques for Product Designers*. Bis Publishers.
- [16] Jennifer Fernquist, Tovi Grossman, and George Fitzmaurice. 2011. Sketch-sketch revolution: an engaging tutorial system for guided sketching and application learning. In *Proc. UIST '11*. 373–382. doi:10.1145/2047196.2047245
- [17] Martin A. Fischler and Helen C. Wolf. 1994. Locating Perceptually Salient Points on Planar Curves. *IEEE Trans. Pattern Anal. Mach. Intell.* 16, 2 (1994), 113–129. doi:10.1109/34.273737
- [18] Bruce Gooch and Amy Gooch. 2001. *Non-photorealistic rendering*. CRC Press.
- [19] Martin Guay, Marie-Paule Cani, and R  mi Ronfard. 2013. The Line of Action: An Intuitive Interface for Expressive Character Posing. *ACM Trans. Graph.* 32, 6, Article 205 (2013). doi:10.1145/2508363.2508397
- [20] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Human mental workload* 1, 3 (1988), 139–183.
- [21] Sonia Hashim, Tobias H  llerer, and Jennifer Jacobs. 2023. Drawing Transforms: A Unifying Interaction Primitive to Procedurally Manipulate Graphics across Style, Space, and Time. In *Proc. CHI '23*. 1–15. doi:10.1145/3544548.3580642
- [22] Aaron Hertzmann. 1999. Introduction to 3d non-photorealistic rendering: Silhouettes and outlines. *Non-Photorealistic Rendering, SIGGRAPH '99*, 1 (1999).
- [23] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. 1999. Teddy: A Sketching Interface for 3D Freeform Design. In *Proc. SIGGRAPH '99*. 409–416. doi:10.1145/311535.311602
- [24] Takeo Igarashi, Tomer Moscovich, and John F Hughes. 2007. Spatial Keyframing for Performance-Driven Animation. In *ACM SIGGRAPH 2007 Courses*. doi:10.1145/1281500.1281536
- [25] 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 (2009), 1–93. doi:10.1561/1100000013
- [26] Thorsten Karrer, Malte Weiss, Eric Lee, and Jan Borchers. 2008. Dragon: a direct manipulation interface for frame-accurate in-scene video navigation. In *Proc. CHI '08*. 247–250. doi:10.1145/1357054.1357097
- [27] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: sketching dynamic and interactive illustrations. In *Proc. UIST '14*. 395–405. doi:10.1145/2642918.2647375
- [28] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014. Draco: bringing life to illustrations with kinetic textures. In *Proc. CHI '14*. 351–360. doi:10.1145/2556288.2556987
- [29] Joonho Kim and Karan Singh. 2021. Optimizing UI Layouts for Deformable Face-Rig Manipulation. *ACM Trans. Graph. (TOG)* 40, 4, Article 172 (2021). doi:10.1145/3450626.3459842
- [30] Bum Chul Kwon, Hannah Kim, Emily Wall, Jaegul Choo, Haesun Park, and Alex Endert. 2016. Axisketcher: Interactive nonlinear axis mapping of visualizations through user drawings. *IEEE Trans. on Vis. and Comp. Graph.* 23, 1 (2016), 221–230. doi:10.1109/TVCG.2016.2598446
- [31] Yunjin Lee, Lee Markosian, Seungyong Lee, and John F. Hughes. 2007. Line drawings via abstracted shading. *ACM Trans. Graph. (TOG)* 26, 3 (2007), 18. doi:10.1145/1276377.1276400
- [32] John P Lewis and Ken-ichi Anjyo. 2010. Direct manipulation blendshapes. *IEEE Comp. Graph. and App.* 30, 4 (2010), 42–50. doi:10.1109/MCG.2010.41
- [33] Kok-Lim Low. 2004. Linear least-squares optimization for point-to-plane icp surface registration. *Chapel Hill, University of North Carolina* 4, 10 (2004), 1–3.
- [34] Daniel Mendes, Fabio Marco Caputo, Andrea Giachetti, Alfredo Ferreira, and Joaquim Jorge. 2019. A Survey on 3D Virtual Object Manipulation: From the Desktop to Immersive Virtual Environments. *Comp. Graph. Forum* 38, 1 (2019), 21–45. doi:10.1111/cgf.13390
- [35] Elie Michel and Tamy Boubekeur. 2021. DAG Amendment for Inverse Control of Parametric Shapes. *ACM Trans. Graph. (TOG)* 40, 4 (2021), 173:1–173:14. doi:10.1145/3450626.3459823
- [36] Erick Mikiten. 2022. The Layers of Architectural Design. In *Concept App*. <https://concepts.app/en/stories/the-layers-of-architectural-design/>
- [37] Niloy J. Mitra, Yong-Liang Yang, Dong-Ming Yan, Wilmot Li, and Maneesh Agrawala. 2013. Illustrating how mechanical assemblies work. *Commun. ACM* 56, 1 (2013), 106–114. doi:10.1145/2398356.2398379
- [38] Matthias M  ller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. 2005. Meshless deformations based on shape matching. *ACM Trans. Graph. (TOG)* 24, 3 (2005), 471–478. doi:10.1145/1073204.107321
- [39] Andrew Nealen, Olga Sorkine, Marc Alexa, and Daniel Cohen-Or. 2005. A Sketch-Based Interface for Detail-Preserving Mesh Editing. *ACM Trans. Graph. (TOG)* 24, 3 (2005), 1142–1147. doi:10.1145/1073204.1073324
- [40] Luke Olsen, Faramarz F Samavati, Mario Costa Sousa, and Joaquim A Jorge. 2008. A Taxonomy of Modeling Techniques using Sketch-Based Interfaces.. In *Eurographics (State of the Art Reports)*. 39–57. doi:10.2312/egst.20081044
- [41] Karran Pandey, Fanny Chevalier, and Karan Singh. 2023. Juxtaform: Interactive Visual Summarization for Exploratory Shape Design. *ACM Trans. Graph. (TOG)* 42, 4, Article 52 (2023). doi:10.1145/3592436
- [42] Stephanie Santosa, Fanny Chevalier, Ravin Balakrishnan, and Karan Singh. 2013. Direct space-time trajectory control for visual media editing. In *Proc. CHI '13*. 1149–1158. doi:10.1145/2470654.2466148
- [43] Ryan Schmidt, Karan Singh, and Ravin Balakrishnan. 2008. Sketching and Composing Widgets for 3D Manipulation. *Comp. Graph. Forum* 27, 2 (2008), 301–310. doi:10.1111/j.1467-8659.2008.01127.x
- [44] Karan Singh and Eugene Fiume. 1998. Wires: A Geometric Deformation Technique. In *Proc. SIGGRAPH '98*. 405–414. doi:10.1145/280814.280946
- [45] Ivan E. Sutherland. 1964. Sketch Pad a Man-machine Graphical Communication System. In *Proc. of the SHARE Design Automation Workshop*. 329–346. doi:10.1145/800265.810742
- [46] Ryo Suzuki, Rubaiat Habib Kazi, Li-yi Wei, Stephen DiVerdi, Wilmot Li, and Daniel Leithinger. 2020. RealitySketch: Embedding Responsive Graphics and Visualizations in AR through Dynamic Sketching. In *Proc. UIST '20*. 166–181. doi:10.1145/3379337.3415892
- [47] Daniel S  kora, Jan Buri  nek, and Ji  r   Z  ra. 2005. Sketching Cartoons by Example. In *Proc. of Eurographics Workshop on Sketch-Based Interfaces and Modeling*. 27–34.
- [48] Daniel S  kora, John Dingliana, and Steven Collins. 2009. LazyBrush: Flexible Painting Tool for Hand-drawn Cartoons. *Comp. Graph. Forum* 28, 2 (2009), 599–608.
- [49] James T Todd. 2004. The visual perception of 3D shape. *Trends in cognitive sciences* 8, 3 (2004), 115–121. doi:10.1016/j.tics.2004.01.006
- [50] Theophanis Tsandilas, Anastasia Bezerianos, and Thibaut Jacob. 2015. Sketch-sliders: Sketching widgets for visual exploration on wall displays. In *Proc. CHI '15*. 3255–3264. doi:10.1145/2702123.2702129
- [51] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. 2007. Gestures without Libraries, Toolkits or Training: A \$1 Recognizer for User Interface Prototypes. In *Proc. UIST '07*. 159–168. doi:10.1145/1294211.1294238
- [52] Haijun Xia, Bruno Araujo, Tovi Grossman, and Daniel Wigdor. 2016. Object-Oriented Drawing. In *Proc. CHI '16*. 4610–4621. doi:10.1145/2858036.2858075
- [53] Baoxuan Xu, William Chang, Alla Sheffer, Adrien Bousseau, James McCrae, and Karan Singh. 2014. True2Form: 3D curve networks from 2D sketches via selective regularization. *ACM Trans. Graph. (TOG)* 33, 4 (2014). doi:10.1145/2601097.2601128
- [54] Robert C Zeleznik, Kenneth P Herndon, and John F Hughes. 1996. Sketch: An Interface for Sketching 3D Scenes. In *Proc. SIGGRAPH '96*. 163–170. doi:10.1145/237170.237238

Received 9 April 2025; revised 14 July 2025; accepted 30 July 2025