User-Guided Synthesis of Interactive Diagrams

John Sarracino
UC San Diego
jsarraci@cs.ucsd.edu

Odaris Barrios-Arciga
Scripps College
obarrios1585@scrippscollege.edu

Jasmine Zhu
Harvey Mudd
jizhu@hmc.edu

Noah Marcus
Harvey Mudd
nmarcus@cs.hmc.edu

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

Ben Wiedermann
Harvey Mudd
benw@cs.hmc.edu

ABSTRACT

Interactive diagrams are expensive to build, requiring significant programming experience. The cost of building such diagrams often prevents novice programmers or non-programmers from doing so. In this paper, we present user-guided techniques that transform a static diagram into an interactive one without requiring the user to write code. We also present a tool called EDDIE that prototypes these techniques. We evaluate EDDIE through: (1) a case study in which we use EDDIE to implement existing real-world diagrams from the literature and (2) a usability session with target users in which subjects build several diagrams in EDDIE and provide feedback on EDDIE's user experience. Our experiments demonstrate that EDDIE is usable and expressive, and that EDDIE enables real-world diagrams to be implemented without requiring programming expertise.

Author Keywords

Interactive Diagrams; Program Synthesis

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous; I.2.2. Automatic Programming: Program synthesis

INTRODUCTION

Interactive diagrams are animated diagrams that users can interact with using a computational device such as a computer or tablet. For example, an interactive physics diagram with pulleys and weights might allow the user to move the pulleys and vary the weights, while observing the physical simulation that ensues.

Developing an interactive diagram requires programming expertise with technologies like JavaScript, HTML5 and server-side databases, and diagram authors might not have the level of programming ability required to make an interactive diagram.

CHI 2017, May 06 - 11, 2017, Denver, CO, USA

ISBN 978-1-4503-4655-9/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3025453.3025467

In this paper, we present USER-GUIDED INTERACTION SYNTHESIS, a technique that bridges this gap by enabling users to build good interactive diagrams without requiring any programming knowledge. Instead of writing code directly to implement an interactive diagram, users first make a static version of the diagram and then our technique *synthesizes* (i.e., adds) interactivity automatically. This technique also allows the author to visualize alternative interactivity models, so the author can quickly explore the space of possible interactive diagrams, and select and/or refine the ones that are the most appropriate for the specific goals.

The main challenges in making program synthesis-based techniques work are twofold: (1) usability/expressiveness and (2) computational tractability. For the former, the synthesis langauage must be expressive and general enough to implement a variety of interaction models, while succinct enough to be amenable to synthesis and usable by non-programmers. For the latter, even with a succinct intermediate representation the search space is extremely large and underconstrained as the number of possible programs that implement interactive diagrams is unbounded and metrics for suitable interactions vary among diagrams.

To address these problems, USER-GUIDED INTERACTION SYNTHESIS makes use of the following two key ideas:

- **Constraint-based formalism**: This technique expresses and runs interactive diagrams using a constraint-based formalism with dynamically-adjusted constraints. This additional structure more effectively explores the search space of programs that implement interactive diagrams: instead of looking at all possible *programs*, the search looks at all possible *constraints* over a set of clearly defined variables. Furthermore, dynamic constraints offer a succinct and natural way of capturing the variation in interactivity across diagrams.

- **User-guided preview-based synthesis**: This technique employs a user-guided and interactive synthesis technique for programs in the constraint-based formalism. It does not rely on programming knowledge. Instead it uses a *preview-based* approach where the diagram author is shown previews of various interaction modalities, from which the author can pick the best one. The preview-based technique drastically reduces the user's manual effort while also narrowing down the search space.

Using the above ideas, we built a diagram editor called EDDIE (screenshot in Figure 1a) that implements USER-GUIDED INTERACTION SYNTHESIS. To demonstrate feasibility on a real-world application of interactive diagrams, we picked the domain of physics education. As benchmarks we used the interactive diagrams from the Physics Education Technology project (PhET) [42]. We performed our evaluation along three dimensions, *expressiveness*, *utility*, and *usability*.

For expressiveness and utility, we performed a case study in which we used EDDIE to re-implement a variety of existing PhET diagrams. Of the 11 existing PhET diagrams on gravity, springs, and pendulum motion, EDDIE was able to generate analogous versions for 9 of these 11 diagrams. Making these diagrams with EDDIE took a few minutes of human effort each. By comparison, these 11 benchmarks as implemented in PhET each required an average of roughly 5000 lines of expertly-written code.

For usability, we recruited a group of science teachers to author two diagrams using EDDIE and provide feedback on their experience. All of the participants were able to complete the task within an hour, and none of the participants required direct aid from the authors.

Our experiments demonstrate that EDDIE is usable and expressive, and that USER-GUIDED INTERACTION SYNTHESIS allows real-world diagrams to be authored with much less effort than the original diagrams.

In summary, our contributions are as follows:

- USER-GUIDED INTERACTION SYNTHESIS, a technique that transforms a static diagram into an interactive one without requiring the user to write code. The technique is summarized in the **Overview** section.
- A constraint-based formalism for interactive diagrams, summarized in the **Interactive Diagrams using Constraints** section.
- A user-guided, preview-based synthesis engine for interactive diagrams in the constraint-based formalism language, discussed in the **User-Guided Diagram Synthesis** section.
- An implementation of USER-GUIDED INTERACTION SYNTHESIS in a diagram editor named EDDIE, discussed and evaluated in the **Prototyping and Evaluation** section.
- An evaluation of EDDIE on a case study of PhET interactive physics diagrams, discussed in the **Case Study** subsection.
- An evaluation of EDDIE on a user insight survey of science educators, discussed in the **Usability: Teacher Usage and Insights** subsection.

RELATED WORK

Visual Editors. There is a long and rich line of work on visual editors for diagrams, for example [29, 10, 26, 17, 55, 31, 52, 32, 51, 50]. To understand how our work fits into this broad line of research, it's important to first note that the vast majority of prior editors (including all those cited above) support either *static* or *animated* diagrams, but not authoring of *interactivity* for the viewer, which is our main goal.

Much fewer editors support authoring of interactivity – Kitty [30] and Apparatus [1] are the main examples in this space. Our work is different from these editors in two ways. First, our approach of encoding diagrams using constraints and synthesizing the constraints automatically is novel. Second, our tool requires much less expertise to use than prior approaches. Consider for example Apparatus [1], which is a direct-manipulation editor for authoring interactive dataflow diagrams. While Apparatus is a powerful and sophisticated system, to use Apparatus the author must first become familiar with the tool's dataflow programming paradigm. Consider as another example Kitty [30], which is sketch-based project for professional authoring of interactive animations. While Kitty enables the diagram author to add end-user interactivity to an animation, the author in Kitty must correctly express a global interactivity modality as the composition of pairwise element functional relationships.

In contrast to prior work, in our tool, the author must only: (1) draw a static, non-animated diagram, in a style that is familiar to many computer users (2) select between self-animating previews. This is a substantially lower cognitive burden: in our tool, the author has only two concepts to cognitively process (static diagrams and previews of dynamic diagrams) whereas in prior tools, the author must also process the language/mechanism for expressing interactivity (note that we have such a mechanism—the constraint language—but we *generate* the constraints automatically). We validate in our experiments that users with no prior knowledge can quickly generate interactive diagrams after just seeing a 15 minute demo.

There is however a key tradeoff here: while our approach reduces the cognitive burden required to build interactive diagrams, we don't provide the same expressiveness as prior approaches (as we will discuss in the Limitations and Future Work section). As such, our work complements prior work: we provide good automation for a subset of all possible diagrams, and authors can go to more complex techniques/approaches for the rest.

Constraints in User Applications. Constraints have been used for visual layout for many years, dating back to SketchPad [47]. More recent work uses constraints for GUI builders [39, 38, 49], for user interactions [48], and as programming paradigms [40, 18]. The DeltaBlue project [19] coined the notion of an *incremental* constraint solver, which can dynamically resolve a system when constraints are added or removed. Our work presents an application of constraint solving to the domain of interactive diagram synthesis: indeed, our implementation uses Cassowary [4, 12], an incremental constraint solver also used in Mac OS X.

Inference of Layout Constraints. Closely related to our work is research on layout editors that use constraints in the back-end, for example the work on Programming by Manipulation for Layout [25] and The Auckland Layout Editor for GUIs [54]. Our work is different in that, instead of just inferring layout constraints, our work also infers (with the help of a human) the constraints for *interactivity*: what happens when control points in a diagram are dragged by a user.

Program Synthesis. Program Synthesis infers a program from a partial program or a specification [21]. Synthesis is hard because the search space is large, which is typically addressed by limiting the domain language [22, 23, 46, 28, 41], cleverly enumerating programs [33, 53], or asking the human for help [45]. The primary difference in our work is that we apply synthesis-inspired techniques to the domain of interactive diagrams. We narrow the search space by (1) considering a constraint-based formalism whose structure we can exploit and (2) asking the human to make key decisions.

Programming by Demonstration. Programming by Demonstration (PbD), also know as Programming by Example, builds programs from example input-output pairs [15, 37]. PbD is used in a variety of settings, such as generating web scripts [34], building visual tutorials [20], and creating multi-touch interactions [35]. In contrast to PbD, our approach does not use example program input-output pairs.

Interactive Technical Diagrams. There has been a lot of recent interest in digitizing figures and diagrams for STEM education [24, 3]. A large and successful project integrating computers with education is the Physics Education Technology (PhET) project [42], which builds and evaluates interactive diagrams for K-12 [8, 36, 43]. PhET's diagrams require significant programming expertise to build. The project employs 4 full-time software developers and states that each diagram typically involves a software developer, a scientist, and an educator [6]. Our contribution is to enable users with no programming expertise to directly build interactive diagrams.

Interactive Diagram Toolkits. Kitty [30] is a sketch-based project for professional authoring of interactive animations. Kitty's model of interaction is that of *functional relationships* with one parameter, e.g., variable A as a function of variable B. The functional relationship model fits well within the domain of sketch-based animation authoring; however, many useful diagrammatic relationships involve more than two variables and so can't be expressed with Kitty's functional relationships. In addition, users of Kitty must specify the functional relationship between each diagram element and the remaining diagram elements, which leads to a quadratic growth in the number of user-provided relationships. In contrast, our use of constraints enables the technique to not only encode many useful relationships between more than two variables, but to do so very succinctly, without having to explicitly state pair-wise relationships between all diagram elements.

Sketch-based Simulation Inference. Sketching is a body of work focused on recognizing a simulation or animation from a user's diagrammatic sketch. By relying on properties of particular domains, sketching has been applied effectively to mechanical systems [10, 11], vector spaces [14], and fluid simulations [55]. In addition, more general sketching systems have also been developed [17, 27]. With the exception of Kitty [30], sketching work builds non-interactive diagrams: in contrast, we use program synthesis techniques to build interactive diagrams.

OVERVIEW

We begin with an overview of how EDDIE works through an example. There are two kinds of people who interact with EDDIE: users who *build* interactive diagram, which we will refer to as *authors* from now on; and users who *interact* with interactive diagrams, which we will refer to as *viewers* from now on. For a single diagram, EDDIE displays both perspectives simultaneously (screenshot in Figures 1a) using two panes, a left pane for the editing perspective of the author and a right pane for previewing the perspective of the viewer. The left author pane is akin to a traditional editor perspective, whereas the right pane provides a live interactive preview that is continuously updated.

For our running example, suppose that an author wants to build an interactive physics diagram depicting a platform with a weight on top of a spring, as shown in Figures 1a and 1b. The author places a weight (using an image link), a spring, a rectangle for the platform, and a rectangle for the base in the diagram by clicking the respective buttons in the "Shapes" and "Physics" dropdown menus. Once the shapes are in the diagram, the author uses the left pane of EDDIE to place the shapes in the desired configuration (as shown in Figures 1a and 1b). As the shapes are moved in the left author pane, the shapes in the right viewer pane follow.

The author next adds viewer interactivity to the diagram. EDDIE enables viewer interactivity through draggable points called *drag points*. Drag points are visible in the viewer's perspective (the right pane) and are draggable by the viewer. For this diagram, the author desires two viewer interactions through two drag points. First, when the viewer drags the *middle of the platform*, the following should happen: (1) the platform translates, but only in the Y dimension (2) the base remains in place (3) the weight translates and (4) the spring compresses/extends. Second, when the viewer drags the *middle of the base*, the entire diagram should translate in both dimensions.

To add these interactions, the author must first specify where drag points are located. This is done by clicking "Interaction", which displays all candidate drag points in the author pane. The author can then enable the desired drag points by clicking on them (clicking again disables the drag point). In the running example, the author selects the two desired drag points, in the middle of the weight and in the middle of the base, as shown in Figure 1b.

For each enabled drag point in the author pane, EDDIE automatically generates a tentative interaction modality. At this point, the author can interact with the diagram in the viewer pane and see what interaction modalities EDDIE has picked. Interacting with the diagram, the author notices that the drag point on the base translates the entire diagram, exactly as the author intended. The drag point on the weight moves the weight and platform and stretches the spring in both the X and Y dimensions, which—although a valid interaction—is *not* what the author intended. Instead, the author intended the weight and platform to ignore viewer input in the X dimension, i.e., to only translate and stretch in the Y dimension.

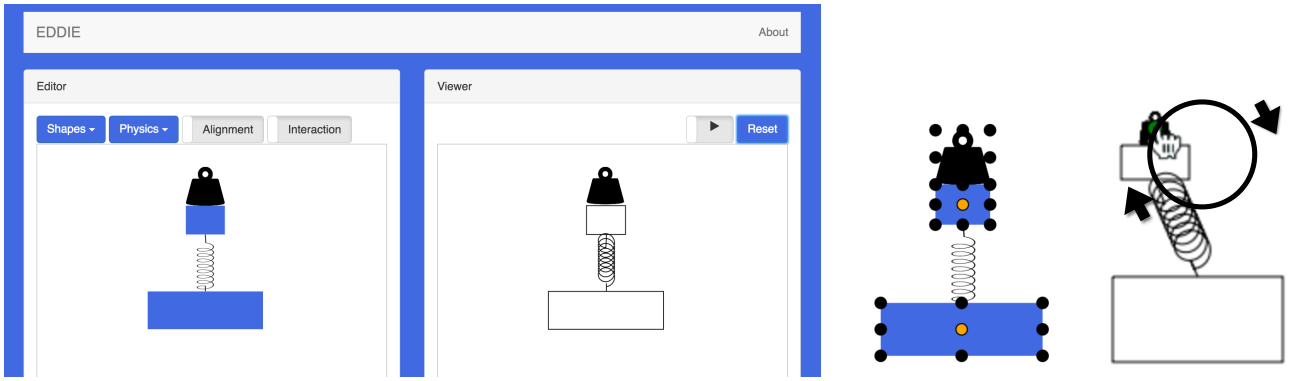


Figure 1: Screenshots of EDDIE: (a) the main view, with left editing pane and right preview pane (b) drag point selection view, with selected points in orange and unselected ones in black (c) animated preview pane—this particular interaction adjusts the spring's height and width to match the movement of the platform.

To change the interaction of the drag point on the weight, the author right-clicks the drag point in the author pane. EDDIE generates a list of valid interaction modalities for this drag point and displays this list in a preview window. The preview window shows an *animated preview* of the currently selected modality, along with "left" and "right" arrows to navigate through the list of animated previews. Each animated preview is *self-animating*, in that it shows what would happen if a viewer were to drag the drag point in a circular pattern. Figure 1c shows a static snapshot of such a self-animating preview. The circle and arrows are not part of the preview – they are used in Figure 1c to explain what path the drag point takes during the animation. These self-animating previews enable the author to quickly flip through many possible viewer interactions. In the running example, the author clicks "right" several times until the correct version is shown. The author then clicks "accept" to use this interaction modality.

The author now interacts again with the diagram in the viewer pane, and sees that the drag point on the base has the desired viewer interactivity. Furthermore, the author also tries the simulation component of the diagram in the viewer pane (the spring contains its own physical properties), and makes sure that the diagram works as intended. The diagram is now complete.

Overview of USER-GUIDED INTERACTION SYNTHESIS. So far, this overview focused on a description of EDDIE from the perspective of the author and omitted technical details of the technique underlying EDDIE. EDDIE uses a technique we call USER-GUIDED INTERACTION SYNTHESIS, which broadly speaking works as follows. First, interactive diagrams are represented in an intermediate language in two parts: a symbolic description of the shapes and a constraint-based formulation of interaction modalities. Second, given a static diagram, which contains shape descriptions and locations of drag points (with no interactivity), a synthesis algorithm generates an ordered list of interaction modalities for each drag point, in which each interaction modality is represented by a set of constraints governing that point's motion. Third, the author is able to preview all the interaction modalities by right-clicking on a drag point, and selecting from a list of animated previews. The rest of this paper explains these technical details in more depth. The next section (Interactive Diagrams Using Constraints) shows how to encode interactive diagrams using constraints, and the following section (User-Guided Diagram Synthesis) describes our synthesis algorithm and previewing approach.

INTERACTIVE DIAGRAMS USING CONSTRAINTS

USER-GUIDED INTERACTION SYNTHESIS uses a constraint formalism to encode interactive diagrams. We first present background material on constraint systems and then we show how to use constraints to encode a given interactive diagram.

Background on Linear Constraint Systems

A *linear constraint* is a linear equality over variables, in this case floating-point variables. For example, $y = 3x$ and $t = 2$ are linear constraints, while $x^2 + y^2 = 3$ is not. A *linear constraint solver* takes a set of linear constraints and attempts to produce a valuation of constraint variables that satisfies the input constraints. If the input constraints are not solvable, e.g., $x = 2$ and $x = 3$, the constraint solver produces a variable valuation that minimizes an error metric, typically a weighted sum of violated constraints. Linear constraint solving is an example of linear programming, which is typically solved using Dantzig's classic simplex algorithm [16] or a variant thereof.

This work uses a specific kind of constraint solver, namely an *incremental constraint solver* [19]. Such constraint solvers are designed specifically for interactive domains, including graphical applications, and are typically used in the following way. First, the layout of graphical objects on the screen is encoded using a set of constraints. Then, each time the viewer moves an object on the screen, the constraints are solved again to re-compute the layout of objects. A prominent example of such a constraint solver is the Cassowary [12] constraint solver, which we use in our implementation of EDDIE (and which is also used in the layout engine for Mac OS X).

Interactive constraint solvers have two features specifically designed for interactive graphical applications: *stay* constraints and *edit* constraints.

A *stay* constraint is a constraint of the form *Stay*(v), where v is a variable. This constraint tells the constraint solver that the next time it is invoked to solve the constraints, the variable v should remain the same as its old value. This is useful because in most cases there are many solutions to the constraints, and the *stay* constraints enable an algorithm to state that for certain variables, all else being equal, they should remain unchanged.

An *edit* constraint is a constraint of the form *Edit*$(v = c)$ where v is a variable and c is a numerical constant. This constraint acts exactly like the regular constraint $v = c$, except that after the solver is done solving the constraints, it *deletes* the edit constraint. Edit constraints are used to encode changes due to viewer input. For example, if the viewer moves an object horizontally on the screen to a new position, one would create an edit constraint to temporarily set the X variable for that object to be the new position. The solver then solves the constraints (including the edit constraint) to find a new layout, after which the edit constraint is automatically removed. At this point the viewer's input has been incorporated into the current values of the variables, and so there is no need to keep the edit constraint around.

Finally, constraint solvers also have a way of *prioritizing constraints*. This is typically done by assigning constraints weights, and the constraint solver tries to minimize the sum of the weights of violated constraints. This prioritization feature is a side effect of the fact that these solvers typically use Simplex, which similarly solves constraints by minimizing the weighted sum of violated constraints. Despite being an implementation detail of the solver's algorithm, prioritization can be useful for interactive user applications.

Interactive Diagrams using Constraints

Now that we have covered background material on constraint systems, we show how to encode a *particular interactive diagram* using constraints. At this point, we are not yet concerned with synthesis of interactive diagrams (i.e., synthesis of the constraints)—we are simply showing how to encode a single interactive diagram using constraints. In the **Synthesis** section, we will show how to *synthesize* an interactive diagram by exploring a set of interactive diagrams and selecting the best one using feedback from the author.

In our technique, interactive diagrams have two components: (1) a shape description section, which states what shapes are in the diagram, and (2) a constraint section, which describes relationships between shapes as linear constraints. Figure 2 shows some examples of the shape primitives used in the shape description section, along with their graphical representation. The parameters to these shape primitives are constraint variables, which we call *control variables*; the relationships between these control variables are stated by equations in the constraint section.

We now describe the three different kinds of constraints and how they are used.

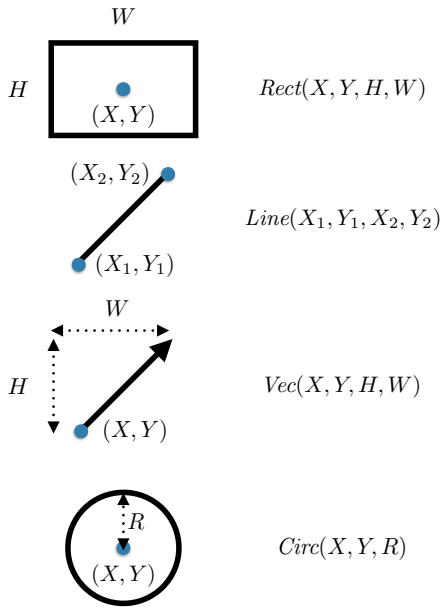


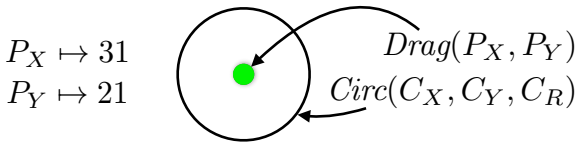
Figure 2: Example primitives for the shape description section. EDDIE also supports Images and Springs, which are analogous to Rectangles and Vectors respectively.

Layout Constraints. Layout constraints are linear equations over control variables which encode layout invariants that must remain true as the diagram is re-configured based on viewer actions. For example, one could state that two rectangles are always centered at the same location by equating their X and Y control variables; one could state that a spring is connected to the corner of a rectangle by equating the spring's end-point to an expression that corresponds to the rectangle's corner.

Taken in isolation, the layout constraints are usually *under-constrained*, meaning that there are many possible solutions. We address this in the standard way by adding *stay* constraints over all control variables. Although staying *all* control variables seems like it might pin the diagram and prevent it from moving, we will next describe how interactive changes happen to the diagram.

Interactivity Constraints. Whereas layout constraints capture layout invariants that must always be met, interactivity constraints are *edit* and *stay* constraints that capture interaction modalities. A viewer can interact with a diagram through *drag points*, which we capture using a primitive *Drag(X,Y)*, where X and Y are constraint variables. Intuitively a drag point *Drag(X,Y)* is simply a point that captures mouse events and sets X and Y to the mouse's position using edit constraints.

In prior work, when edit constraints were added to the system, all existing stay constraints would remain in effect. Although this might seem to lead to a system with no solution (since some points will be constrained to stay and to move at the same time), the edit constraints can be *prioritized* higher than the stay constraints, with the intention of having edit constraints over-ride stay constraints. However, to get this to work in



	Initial	Drag update	Result
Layout	$P_X = C_X$ $P_Y = C_Y$	$P_X = C_X$ $P_Y = C_Y$	$P_X = C_X$ $P_Y = C_Y$
Interactivity	*Stay*(P_X) *Stay*(P_Y) *Stay*(C_X) *Stay*(C_Y) *Stay*(C_R)	*Stay*(C_R) *Edit*$(P_X = 31)$ *Edit*$(P_Y = 21)$	*Stay*(P_X) *Stay*(P_Y) *Stay*(C_X) *Stay*(C_Y) *Stay*(C_R)
Solver State	$P_X = 30$ $P_Y = 20$ $C_X = 30$ $C_Y = 20$ $C_R = 10$	$P_X = 30$ $P_Y = 20$ $C_X = 30$ $C_Y = 20$ $C_R = 10$	$P_X = 31$ $P_Y = 21$ $C_X = 31$ $C_Y = 21$ $C_R = 10$

Figure 3: Solver state before, as a result of, and after a drag update of (P_X, P_Y) to $(31, 21)$.

practice requires a very careful selection of weights to get the right prioritization.

Instead, we take a different approach, in which we do not depend on any prioritization. In particular, we explicitly temporarily "unstay" the variables that we want to allow to move. Every drag point has a *free* set, which is a set of variables whose stay constraints will be temporarily removed when the drag point is moved.

Figure 3 shows an example of how this works. This example has a circle C and a drag point P at the middle of the circle. The free set for the drag point is $\{P_X, P_Y, C_X, C_Y\}$. The example shows what happens when the drag point is dragged from (30,20) to (31,21). The table at the bottom of Figure 3 shows the solver state (layout constraints, interactivity constraints, and current values of all variables) at three different points in time: (1) "Initial": before the drag happens (2) "Drag Update": after the drag happens and after the constraints are updated to reflect the drag, but before the solver has solved the new constraints (3) "Result": after the solver has solved the constraints to reflect the drag. Note how the stay constraints for the free set of P, namely $\{P_X, P_Y, C_X, C_Y\}$ are temporarily removed, and then added back in.

Note that in this example if the drag point's free set had instead been $\{P_X, C_X\}$, then the stay constraints for P_Y and C_Y would have remained in effect throughout. Since this approach prioritizes stay constraints *over* edit constraints, this would make the Y-position of the circle remain the same, while still allowing the X-position to change to 31. In essence, the final result would be that the circle would only be allowed to move in the X-direction, even if the edit constraint (i.e., the viewer moving the mouse) would try to move the circle in the Y-direction. Note also that the prioritization of stay constraints over edit constraints is precisely the opposite of what prior work has done. This can be done without problems because this approach explicitly removes stay constraints for the variables that should be allowed to be free.

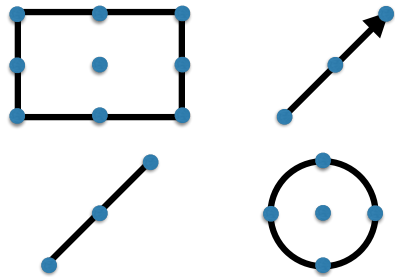


Figure 4: Snap points for our shape primitives. Images and Springs are analogous to Rectangles and Vectors.

USER-GUIDED DIAGRAM SYNTHESIS

Now that we have described how to encode a *single* interactive diagram using constraints, we can now present how interactivity synthesis works. The key insight in this synthesis approach is that it is *user-guided*: during synthesis this approach will ask the author for help with certain decisions that the author is best equipped to do. This collaboration between human and computer enables USER-GUIDED INTERACTION SYNTHESIS to avoid expensive work and incorrect results, while also reducing the work that the author needs to do.

Generating Layout Constraints

After the author draws the static diagram, USER-GUIDED INTERACTION SYNTHESIS first generates layout constraints. Layout constraints capture adjacency relationships, invariants such as "a spring connects P and Q", or "X lies right next to Z". To do so, we first define *snap points* for each of our shape primitives and express the snap point coordinates in terms of shape control variables. Recall that shape control variables are the constraint variables passed as parameters to shape primitives like *Rect*. Figure 4 shows the snap points for several of our shape primitives. For example, given a vector rooted at (X, Y) and with height H and width W, an expression for the midpoint of the vector is $(X + W/2, Y + H/2)$. This expression calculates the midpoint regardless of updates to any of the underlying variables. We call such expressions *snap point expressions*.

Now that we have defined snap points, this approach starts with the *static* diagram that is provided by the author. For each snap point the algorithm now has two things: (1) as mentioned above, it has a snap point expression stating the snap point's coordinates in terms of control variables, for example $(X + W/2, Y + H/2)$ for the mid-point of a vector; (2) it also has the snap point's concrete coordinates in the static diagram, for example $(10, 20)$.

This technique now searches the static diagram for *contact points*, which are two snap points that have the same concrete coordinates in the static diagram—in other words two snap points that are located at the same exact position. This algorithm assumes that a pair of overlapping snap points is not

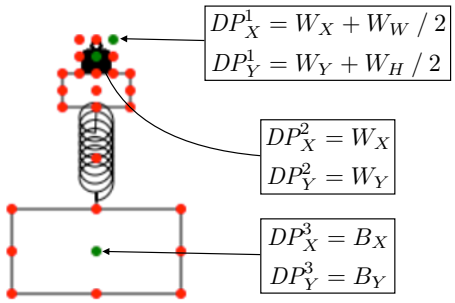


Figure 5: Drag points and their generated linear constraints.

accidental, but rather indicates that the two shapes should be connected at that point. So, for each such pair of overlapping snap points in the static diagram, the point is considered to be a *contact point*, and this technique adds a constraint to keep the two snap points co-located by equating their (X, Y) snap point expressions.

Selecting Drag Points

After generating layout constraints, the next step is to select drag points, which are the points that viewers can drag around on the screen. A drag point is a snap point (as defined previously) which the author has decided to make interactive (recall that each shape has many snap points, as shown for example in Figure 4).

One approach for selecting drag points is to exhaustively search through all subsets of snap points, make them interactive drag points, prune non-sensical results, and then have the author go through these results. However, this approach is both expensive and likely to generate an overwhelming number of options, when in fact the author in most cases already knows precisely which points should be interactive.

Instead, we have decided to let the author make this decision upfront by stating directly which snap points to turn into drag points. The author can click on "Interactions", which displays all candidate points that can be made into drag points. The author can then click on the points that should be interactive. Figure 5 shows this interface after the author has selected three drag points, which have become green to indicate they are selected. At any point later in the editing, the author can go back to this view and select additional drag points, or disable previously selected drag points.

USER-GUIDED INTERACTION SYNTHESIS now needs to generate layout constraints for the newly created drag points. For each snap point P that the author has selected to become a drag point, the technique creates control variables for the drag point's X and Y coordinates, and creates constraints equating those control variables to P's snap point expressions. These constraints have two purposes. (1) When the underlying shape moves/resizes because of other changes in the diagram, the drag point also moves—consider for example a drag point in the corner of a box for resizing, which should move with the shape. (2) When the drag point is moved, it will have an effect on the shape itself.

Generating Interactivity Constraints

Now that the author has chosen drag points, what remains is to generate the constraints that govern the interactivity of these drag points. Recall that interactivity of drag points is governed by the free set of each drag point. This free set captures those variables which will be allowed to change when a given drag point is dragged. Thus, this algorithm must generate a free-set map M from drag points to sets of variables. Our general approach will again leverage human interaction: the engine generates free-set maps and orders them heuristically, and finally displays animated previews to the author showing what interactions look like under these different free-set maps. The author will then be able to quickly find their desired interactivity.

Pruning by Validity. First, we show how to significantly narrow down the search space of free-set maps before even displaying them to the author. We define a notion of validity for the free-set map, so that the search only needs to look at valid maps, instead of all maps. To understand where this notion of validity comes from, let's return to constraint solvers.

There are two situations where constraint solvers give results that are difficult to predict. The first is when the constraints are *underspecified*, in which case there are many solutions, and the constraint solver just picks one. The second is when the constraints are *overspecified*, in which case the constraint solver uses weights to figure out which constraints to violate. In both of these situations the results are hard to predict and are very sensitive to small changes in constraint weights. Instead, the synthesis algorithm will keep constraints *exactly specified*, where there is a single solution. The notion of validity for a free-set map M will *guarantee* that the constraints are always exactly specified, so that the solution depends only on the constraints, and not on internal details of the solver or brittle weights.

At first, one might think that there is a simple way to define validity, which is to require that the number of variables is equal to the number of constraints. Although in linear systems this guarantees no more than one solution (i.e., it avoids being underspecified), it doesn't guarantee a solution (i.e., it *does not* avoid being overspecified).

Instead, we define an algorithmic definition of validity. In particular, for each drag point and the free set in M for the drag point, the algorithm constructs the edit constraints that would be built at runtime if the drag point were dragged, and then uses a simple traversal through the constraints starting at the edited variables to make sure that all variables can be solved for exactly. To begin, the algorithm initially marks as *determined* all the variables that have changed (i.e., mentioned in edit constraints) and all other variables that are constrained to stay unchanged (i.e., mentioned in a stay constraint). Next, each time it sees a linear constraint where all but one of the variables is determined, the last variable is marked as determined (which works because linear constraints provide a unique solution for the last undetermined variable in an equation). We repeat this process until no more variables can be marked as determined. If during this process each variable is marked as determined *exactly once*, this drag point and its as-

sociate free set leads to a single unique solution when dragged. If this process succeeds for all drag points and their associated free sets in M, we say that M is valid.

Narrowing down to only valid maps is really important: for a drag point in our most complex benchmark, there are 28 shape control variables for roughly 268 million possible free-set maps but only 3 of these maps are valid. The above definition of validity can be applied directly using a naïve approach that enumerates all possible free sets for each drag point, and then applies the validity definition to only maintain valid maps. While this approach works, it is also very expensive and indeed, intractible in practice. We have developed an optimized dynamic-programming computation which builds the valid free sets bottom-up efficiently.

Preview Ranking and Visualization. Empirically the number of valid maps for a given drag point is relatively small (on average 14.5). As a result, for each drag point the algorithm generates all valid free sets and ranks the results with several heuristic aesthetics functions.

This ranking function encodes numerically the following four aesthetic observations: (1) changes in response to viewer interaction tend to involve only a handful of shapes, and so we favor interactions with fewer number of shapes; (2) shapes tend to respond to changes the same way in both dimensions, e.g., resizing in one dimension but translating (not resizing) in another is very unusual; (3) drag points on the corners of shapes tend to control stretching, while drag points in the middle of shapes tend to control translation; (4) drag points that move in one dimension are much less common than points that move in multiple dimensions.

Once the free sets are ranked, the top-ranking free set for each drag point becomes the default for that drag point. The author can now interact with the diagram in the viewer pane. If any drag point does not interact in the intended way, the author can right-click on the drag point and see an ordered list of all available interaction modalities. Each modality is previewed automatically through a self-animating diagram, showing the author what would happen if the drag point were moved in a circular motion. The author can flip through the different previews, and pick the one that captures the correct interactivity. Figure 1c in the Overview section depicts an automatically-animated preview. The preview adds an image of a hand to indicate exactly which part of the diagram is being dragged. There is also an "Accept" button (not shown) that allows the author to accept the current preview, and selection arrows (not shown) that allow the author to move between different previews.

By ordering the interactivity modalities heuristically, using the top-ranked modality as the default, and allowing the author to change modalities through previews, this technique leverages the computer's ability to quickly narrow down millions of modalities into a short ordered list, and also leverages the human's ability to quickly pick among a list of self-animating previews.

PROTOTYPING AND EVALUATION

To evaluate USER-GUIDED INTERACTION SYNTHESIS, we built a diagram editor named EDDIE that implements our approach. We wanted EDDIE to be easily accessible and so implemented EDDIE as a client-side HTML5/JavaScript web application. For the static diagram editor, we slightly modified the FabricJS [2] library.

To demonstrate feasibility on a real-world application of interactive diagrams, we picked the domain of physics education. We evaluate EDDIE on diagrams in the Physics Education Technology (PhET) project, a library of interactive educational technical diagrams [42]. PhET has received many awards and accolades and has provided empirical evidence for its educational effectiveness [6, 36, 13, 9]. In addition to being interactive, PhET's diagrams include a simulation modelling the technical topic. As a proof of concept we extended EDDIE with a naïve physics engine to model a subset of PhET's physical diagrams.

We evaluate EDDIE along three dimensions: (1) *Expressiveness*, which looks at what kinds of interactive diagrams EDDIE can build (2) *Cost Savings*, which looks at how good EDDIE is at reducing a user's effort of authoring interactive diagrams and (3) *Usability*, which looks at how usable the tool is in practice. The first two dimensions (expressiveness and costs savings) are evaluated through a case study in which we ourselves implement several real-world diagrams. The third dimension (usability) is evaluated by asking thirteen educators to use EDDIE to build two PhET diagrams.

Case Study. In our case study, we looked at all interactive PhET physics diagrams for the following three topics: (1) simple spring mechanics, (2) gravity and planetary motion, and (3) pendulum motion. We discuss the physical and interactive properties of diagrams within each of these topics in turn.

- **Simple Spring Mechanics:** these diagrams study the dynamics of idealized, massless springs with one end fixed and the other subject to an applied force, for example a weight or another spring. The drag point is connected to the free end and controls the displacement of the spring. Springs can be connected end-to-end, in *series*, or side-by-side, in *parallel*. This category contains five diagrams: 1 spring, 2 springs, 2 series springs, 2 parallel springs, and a weight on a spring.
- **Gravity and Planetary Motion:** these diagrams study the force of gravity and its application to planetary motion. These diagrams have a number of spheres representing planets, as well as overlaid vectors representing velocity. PhET uses drag points in two ways: to determine the placement of a planet, and to determine the value of a velocity vector. This category contains four diagrams: a gravity lab, 2 planets, 3 planets, and 4 planets.
- **Pendulum Motion:** this diagram studies the motion of a simple pendulum subject to gravity and air resistence. The shapes consist of spheres representing the pendulum base and weight and a line between the spheres. A drag point controls the position (and by extension, angular displacement) of the pendulum weight. This category contains two

diagrams, one with two pendulums and one with a single pendulum.

Across the three categories above, there are a total of 11 interactive diagrams. EDDIE is able to generate nine of these diagrams with very little human interaction, requiring at most five minutes of effort by the authors. Before we show the benefits that EDDIE provides in these nine diagrams compared to the traditional way of building these diagrams, we first explain the two diagrams that EDDIE cannot generate.

The first unsupported diagram is a series spring simulation in which the drag point simultaneously translates and compresses one of the springs. Our system limits updates to a single variable and so does not support this interaction. To support such interactions, we could extend EDDIE to use ranking functions on stay-constraints.

The second unsupported diagram is a spring simulation in which the viewer can drag-and-drop weights onto a platform attached to a spring. This action consists of two different layout configurations, one in which the weight is free-floating and one in which the weight is attached to the spring. Our framework currently assumes the layout constraints always hold for a given diagram and so can't support this functionality. To support such interactions, EDDIE would need to add support for conditional layout constraints and drag-and-drop features.

Cost Savings. We now evaluate the manual effort required by the author to make diagrams in EDDIE compared to the traditional approach. The baseline we use is the PhET implementation of the nine benchmarks we also implemented in EDDIE. In general, PhET benchmarks require a large amount of effort to build, and the builder needs to have a lot of programming experience. At the time of this writing, the project employs four full-time software developers [6]. Each diagram requires between 3800 and 5700 lines of handwritten JavaScript and Flash. Figure 6 shows the different benchmarks, along with the lines of codes required to implement each benchmark in PhET (column "PhET LoC").

In EDDIE, authoring a diagram requires two efforts. First, the author generates a static representation of the diagram. We have not formally quantified this effort, but most static diagrams have only a handful of shapes, and creating and aligning those shapes takes on the order of minutes.

Second, the author specifies drag points and drag point interactions in the diagram. Figure 6 shows the effort required for this in three columns: "P" shows the number of drag points that the author must select; "V" shows the total number of previews (that the author must view across all drag points for that benchmark (note that the V count includes the view that the author accepts for each drag point, meaning that V cannot be smaller than P); "VPP" (which stands for *views per point)* shows the total number of previews that the author must view per drag point, in other words VPP = V/P (VPP cannot be smaller than 1).

The VPP metric captures the effectiveness of this technique's heuristic for ordering interactivity models (through ranking of free-set maps). A VPP of 1 corresponds to EDDIE always choosing the right interactivity model first, while a high VPP corresponds to showing many undesired interactivity models before the author finds the right one.

Category	Phet LoC	Name	P	V	VPP
Springs	5278	1 spring	1	3	3
		2 springs	2	6	3
		parallel	1	8	8
Orbits/Gravity	5628	lab	2	4	2
		2 body	4	4	1
		3 body	6	6	1
		4 body	8	8	1
Pendulum	3821	1 mass	1	1	1
		2 masses	2	2	1

Figure 6: Comparison of programming effort between PhET and EDDIE for a variety of PhET physics diagrams. "P" is the number of points present in the diagram, "V" is the number of candidate interactions viewed by the user, and "VPP" is our metric of "Views-Per-Point", "V"/"P". EDDIE significantly reduces the effort of authoring diagrams, requiring one or two views per point for most diagrams.

The VPP results demonstrate that EDDIE ranks interaction modalities well: in most cases, the correct interaction modality is in the top 3. Only one benchmark required 8 views per point, but even in this case, because the previews are self-animating, the viewer can very quickly browse through 8 previews, usually in less than one minute. As a whole, this case study demonstrates that EDDIE can express a variety of real-world diagrams and further, requires little effort by a power user (i.e., one of the authors of this paper).

Usability: Teacher Usage and Insights. To collect feedback and evaluate usability on the target audience of nonprogrammers, we introduced EDDIE during a higher-education computer science pedagogy class for K-12 teachers. We recruited eleven science teachers (three female) aged 25-55 and two male science professors aged 45-60. Participants were first given a brief 15-20 minute presentation on interactive diagrams and the features of EDDIE.

Next, the participants were given an instructional page about EDDIE and two interactive PhET diagrams to replicate within EDDIE. For each diagram, the organizer demonstrated the desired functionality of the specific diagram. The organizer did not directly aid the participants after participants started using EDDIE.

All participants replicated the two diagrams within an hour. The first diagram, consisting of a weight on the end of a spring [7], took the participants roughly 35-45 minutes to replicate. The second diagram was simpler, consisting of two pendulums hanging from a common pivot [5] and took roughly 5-20 minutes to replicate.

After completing both diagrams, the participants answered some qualitative questions about their experience using EDDIE. We asked participants for both positive and negative feedback about the tool, in a free form manner. We did not prompt the participants to discuss any particular feature of EDDIE, and we did not tell them what part of EDDIE was novel.

On the positive side, 8 out of 13 participants mentioned that EDDIE was easy to use and one explicitly mentioned that the side-by-side panes were useful for simultaneously displaying the author's and viewer's perspectives. Finally, 5 out of 13 participants mentioned that they liked EDDIE's static editing capabilities, which are mostly inherited from the existing FabricJS framework.

On the negative side, EDDIE performs a lot of work for the user and as a consequence several participants experienced a steep learning curve—3 out of 13 participants were at times surprised by the output of their actions. However, all three participants overcame the learning curve by referring to the help pages and all three completed both exercises. Furthermore, 3 out of 13 participants were at first confused by the functionality of the left/right buttons in the interface that enables users to select interaction modality. This confusion was resolved after viewing the help page and all three participants completed the diagrams. EDDIE's visualization and presentation of the list is tangential to our research contribution and was picked for ease of implementation. There are other mechanisms to make this interface clearer, for example displaying a swiping animation when previews are switched, replacing the left/right buttons with a single "next" button, or replacing the left/right buttons with a touch/swipe interface.

Finally, the participants used their own machines, which resulted in a wide variety of platforms (e.g., tablets, netbooks, and laptops) and web browsers (e.g., Firefox, Chrome, Safari, Internet Explorer, Edge). A significant number of these environments experienced some performance problems, for example lagging and jittery animated previews. Unfortunately, we did not discover these problems before the study because we had developed and tested EDDIE on a higher-end laptop, on which performance was not an issue. Still, despite these performance problems, EDDIE was ultimately usable and all participants finished their tasks quickly. Furthermore, we have not yet done any significant performance optimizations or extensive cross-browser testing. With further tuning and testing, we believe we can bring good performance to EDDIE across a wide variety of environments.

Insights and Lessons Learned. Through our usage and evaluation of EDDIE, we found that EDDIE's design works well for a variety of reasons.

First, the side-by-side view is useful because it gives immediate feedback on how generated diagrams work. Task-switching is well-known to require measurable cognitive effort [44]. EDDIE's side-by-side panes likely incur less cognitive load than alternating between two different views, because the panes in EDDIE are always visible at the same position.

Second, the self-animated previews are effective at quickly allowing authors to see different interaction modalities, but without having to do any actual manual interaction. This makes the selection process for interaction modalities very quick and easy to use.

Third, the approach of reducing the search space (by pruning underconstrained modalities) and then ordering the top candidates is effective because it narrows the author's attention to only the most promising interaction modalities. While developing EDDIE, prior to adding the ordering heuristics, we experienced frustration at having to navigate many incorrect interactions. After adding the aesthetic heuristics, the experience was much improved.

Limitations and Future Work. There are clear opportunities for future work on expanding our expressiveness and automation to get closer to the expressiveness of more manual tools like Kitty [30] or Apparatus [1], including: continuous locations for drag points, user-defined adjacency relationships, conditional relationships, and nonlinear relationships. Still, despite these limitations on expressiveness, our formalism of shapes and drag points connected by linear adjacency relationships can capture about 67% of the interactivity present in PhET diagrams, a real-world set of interactive diagrams that domain experts actually wanted to build.

In addition, our physics engine is relatively simplistic and as a consequence, there are many diagrams in PhET for which EDDIE can support the interactive portion but not the domain-specific chemical, mathematical, or physical components. To fully support these diagrams, an expert programmer would have to extend EDDIE's implementation by adding domain-specific primitives for these uncovered topics. Once this is done, diagram authors could then use the newly added primitives to reproduce these (currently unsupported) diagrams, without writing any additional code.

Summary. Our case study demonstrates that EDDIE can express and implement a broad variety of real-world diagrams. EDDIE only requires a handful of clicks to generate a complete interactive diagram and does not require any coding experience. In contrast, the original benchmarks require a large amount of programming expertise, which prevents many content experts from directly authoring diagrams. In addition, we demonstrated that EDDIE is usable by non-programming users and gathered some valuable target audience feedback about diagram construction using EDDIE.

CONCLUSION

We presented USER-GUIDED INTERACTION SYNTHESIS, a technique that transforms a static diagram into an interactive one without requiring any code to be written. We also presented an implementation in a tool called EDDIE, which we show is expressive and usable. By drastically reducing the cost of making interactive diagrams, this line of research opens up the possibility for experts who have domain knowledge (e.g., teachers who know about STEM) to build animated diagrams that they would otherwise not be able to build. This provides an exciting avenue not only for future research, but also for eventual impact on the adoption of interactive diagrams.

Funding: This work was funded by NSF grant 1423517.

REFERENCES

1. 2016. Apparatus: a hybrid graphics editor and programming environment for creating interactive diagrams. Website. (2016). Accessed on 2016-04-12 from http://aprt.us.
2. 2016. FabricJS JavaScript Canvas Library. Website. (2016). Accessed on 2016-09-19 from http://fabricjs.com/.
3. 2016. Interactive Mathematics: Learn math while you play with it. Website. (2016). Accessed on 2016-04-08 from http://www.intmath.com.
4. 2016. Overconstrained: Cassowary projects and its community. Website. (2016). Accessed on 2016-04-11 from http://overconstrained.io.
5. 2016. Pendulum Lab – Motion, Pendulum, Simple Harmonic Motion – PhET. Website. (2016). Accessed on 2016-09-19 from https://phet.colorado.edu/en/simulation/legacy/pendulum-lab.
6. 2016. PhET: Interactive Simulations for Science and Math. Website. (2016). Accessed on 2016-04-11 from https://PhET.colorado.edu.
7. 2016. Resonance – Resonance, Harmonic Motion, Oscillator – PhET. Website. (2016). Accessed on 2016-09-19 from https://phet.colorado.edu/en/simulation/legacy/resonance.
8. Wendy K. Adams, Sam Reid, Ron LeMaster, Sarah McKagan, Katherine Perkins, Michael Dubson, and Carl E. Wieman. 2008a. A Study of Educational Simulations Part II Interface Design. *Journal of Interactive Learning Research* 19, 4 (October 2008), 551–577. https://www.learntechlib.org/p/24364
9. Wendy K. Adams, Sam Reid, Ron LeMaster, Sarah B. McKagan, Katherine K. Perkins, Michael Dubson, and Carl E. Wieman. 2008b. A Study of Educational Simulations Part I - Engagement and Learning. *Journal of Interactive Learning Research* 19, 3 (July 2008), 397–419. https://www.learntechlib.org/p/24230
10. Christine Alvarado and Randall Davis. 2004. SketchREAD: A Multi-domain Sketch Recognition Engine. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM, New York, NY, USA, 23–32. DOI: http://dx.doi.org/10.1145/1029632.1029637
11. Christine Alvarado and Randall Davis. 2006. Resolving ambiguities to create a natural computer-based sketching environment. In *ACM SIGGRAPH 2006 Courses*. ACM, 24.
12. Greg J. Badros, Alan Borning, and Peter J. Stuckey. 2001. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction* 8, 4 (dec 2001), 267–306. DOI: http://dx.doi.org/10.1145/504704.504705
13. Julia M. Chamberlain, Kelly Lancaster, Robert Parson, and Katherine K. Perkins. 2014. How guidance affects student engagement with an interactive simulation. *Chem. Educ. Res. Pract.* 15 (2014), 628–638. Issue 4. DOI: http://dx.doi.org/10.1039/C4RP00009A
14. Salman Cheema and Joseph J LaViola Jr. 2010. Applying mathematical sketching to sketch-based physics tutoring software. In *International Symposium on Smart Graphics*. Springer, 13–24.
15. Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.
16. George B Dantzig, Alex Orden, Philip Wolfe, and others. 1955. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific J. Math.* 5, 2 (1955), 183–195.
17. Richard C. Davis, Brien Colwell, and James A. Landay. 2008. K-sketch. In *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08*. ACM Press, New York, New York, USA, 413. DOI:http://dx.doi.org/10.1145/1357054.1357122
18. Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. 2014. *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Babelsberg/JS, 411–436. DOI: http://dx.doi.org/10.1007/978-3-662-44202-9_17
19. Bjorn N Freeman-Benson, John Maloney, and Alan Borning. 1990. An incremental constraint solver. *Commun. ACM* 33, 1 (1990), 54–63.
20. Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating Photo Manipulation Tutorials by Demonstration. In *ACM SIGGRAPH 2009 Papers (SIGGRAPH '09)*. ACM, New York, NY, USA, Article 66, 9 pages. DOI: http://dx.doi.org/10.1145/1576246.1531372
21. Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. ACM, 13–24.
22. Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011a. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*. ACM Press, New York, New York, USA, 62. DOI: http://dx.doi.org/10.1145/1993498.1993506
23. Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011b. Synthesizing geometry constructions. *ACM SIGPLAN Notices* 46, 6 (jun 2011), 50. DOI: http://dx.doi.org/10.1145/1993316.1993505
24. Markus Hohenwarter and Karl Fuchs. 2005. Combination of dynamic geometry , algebra and calculus in the

software system GeoGebra. *Computer algebra systems and dynamic geometry systems in mathematics teaching conference 2004* 2002, July (2005), 1–6. **http://www.geogebratube.org/material/show/id/747**

25. Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14*. ACM Press, New York, New York, USA, 231–241. DOI: **http://dx.doi.org/10.1145/2642918.2647378**

26. Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. 2007. Teddy: a sketching interface for 3D freeform design. In *Acm siggraph 2007 courses*. ACM, 21.

27. Joaquim Jorge and Faramarz Samavati. 2010. *Sketch-based interfaces and modeling*. Springer Science & Business Media.

28. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 3363–3372. DOI: **http://dx.doi.org/10.1145/1978942.1979444**

29. Alan Kay and Adele Goldberg. 1977. Personal dynamic media. *Computer* 10, 3 (1977), 31–41.

30. Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014a. Kitty: sketching dynamic and interactive illustrations. *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14* (2014), 395–405. DOI: **http://dx.doi.org/10.1145/2642918.2647375**

31. Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014b. Draco: Bringing Life to Illustrations with Kinetic Textures. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2014), 351–360. DOI: **http://dx.doi.org/10.1145/2556288.2556987**

32. Rubaiat Habib Kazi, Tovi Grossman, Nobuyuki Umetani, and George Fitzmaurice. 2016. Skuid: Sketching Dynamic Illustrations Using the Principles of 2D Animation. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 4599–4609.

33. Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services - MobiSys '13*. ACM Press, New York, New York, USA, 193. DOI: **http://dx.doi.org/10.1145/2462456.2464443**

34. Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1719–1728.

35. Hao Lü and Yang Li. 2012. Gesture Coder: A Tool for Programming Multi-touch Gestures by Demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 2875–2884. DOI: **http://dx.doi.org/10.1145/2207676.2208693**

36. Emily B. Moore, Timothy A. Herzog, and Katherine K. Perkins. 2013. Interactive simulations as implicit support for guided-inquiry. *Chem. Educ. Res. Pract.* 14 (2013), 257–268. Issue 3. DOI: **http://dx.doi.org/10.1039/C3RP20157K**

37. B. A. Myers. 1986. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*. ACM, New York, NY, USA, 59–66. DOI: **http://dx.doi.org/10.1145/22627.22349**

38. Brad A. Myers. 1990. Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints. *ACM Trans. Program. Lang. Syst.* 12, 2 (April 1990), 143–177. DOI: **http://dx.doi.org/10.1145/78942.78943**

39. Dan R. Olsen, Jr. and Kirk Allan. 1990. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '90)*. ACM, New York, NY, USA, 102–107. DOI: **http://dx.doi.org/10.1145/97924.97936**

40. Stephen Oney, Brad Myers, and Joel Brandt. 2012. ConstraintJS: programming interactive behaviors for the web by integrating constraints and states. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 229–238.

41. Eleanor O'Rourke, Erik Andersen, Sumit Gulwani, and Zoran Popović. 2015. A Framework for Automatically Generating Interactive Instructional Scaffolding. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 1545–1554. DOI: **http://dx.doi.org/10.1145/2702123.2702580**

42. Katherine Perkins, Wendy Adams, Michael Dubson, Noah Finkelstein, Sam Reid, Carl Wieman, and Ron LeMaster. 2006. PhET: Interactive Simulations for Teaching and Learning Physics. *The Physics Teacher* 44, 1 (dec 2006), 18. DOI: **http://dx.doi.org/10.1119/1.2150754**

43. Noah S. Podolefsky, Katherine K. Perkins, and Wendy K. Adams. 2009. Computer simulations to classrooms: tools for change. *AIP Conference Proceedings* 1179, 1 (2009), 233–236. DOI: **http://dx.doi.org/10.1063/1.3266723**

44. Robert D Rogers and Stephen Monsell. 1995. Costs of a predictible switch between simple cognitive tasks. *Journal of experimental psychology: General* 124, 2 (1995), 207.

45. Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-guided device driver synthesis. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 661–676.

46. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 404–415. DOI: **http://dx.doi.org/10.1145/1168917.1168907**

47. Ivan E Sutherland. 1964. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*. ACM, 6–329.

48. Brad Vander Zanden. 1992. Languages for Developing User Interfaces. A. K. Peters, Ltd., Natick, MA, USA, Chapter An Active-value&Mdash;Spreadsheet Model for Interactive Languages, 183–209. **http://dl.acm.org/citation.cfm?id=131302.131313**

49. Bradley T. Vander Zanden, Richard Halterman, Brad A. Myers, Rob Miller, Pedro Szekely, Dario A. Giuse, David Kosbie, and Rich McDaniel. 2005. Lessons Learned from Programmers' Experiences with One-way Constraints: Research Articles. *Softw. Pract. Exper.* 35, 13 (Nov. 2005), 1275–1298. DOI: **http://dx.doi.org/10.1002/spe.v35:13**

50. 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*. ACM, 4610–4621.

51. 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*. ACM, 755–766.

52. Jun Xing, Li-Yi Wei, Takaaki Shiratori, and Koji Yatani. 2015. Autocomplete hand-drawn animations. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 169.

53. Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Colorful Approach to Text Processing by Example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 495–504. DOI: **http://dx.doi.org/10.1145/2501988.2502040**

54. Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. 2013. The Auckland Layout Editor: An Improved GUI Layout Specification Process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 343–352. DOI: **http://dx.doi.org/10.1145/2501988.2502007**

55. Bo Zhu, Michiaki Iwata, Ryo Haraguchi, Takashi Ashihara, Nobuyuki Umetani, Takeo Igarashi, and Kazuo Nakazawa. 2011. Sketch-based Dynamic Illustration of Fluid Systems. *ACM Transactions on Graphics* 30, 6 (dec 2011), 1. DOI: **http://dx.doi.org/10.1145/2070781.2024168**