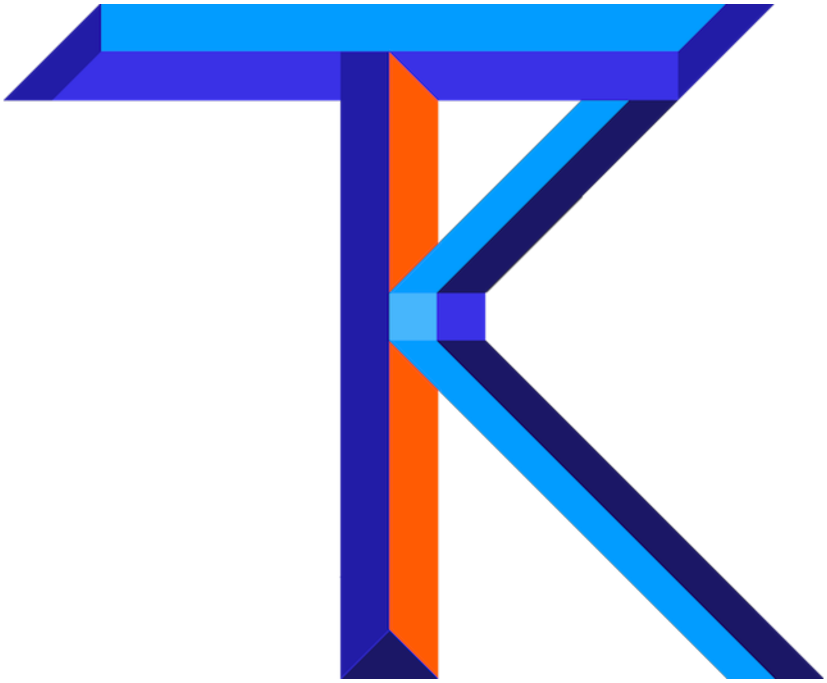
DrawTalking: Building Interactive Worlds by Sketching and Speaking

by

Karl Toby Rosenberg

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
January, 2024

Dr. Kenneth Perlin


TK

Dedication

Dedicated to those who dream and create beautiful things, little or big, or both.

Also, in honor of Rosie the English Springer Spaniel, who was a cartoon character in real-life.

Acknowledgments

I thank my advisor, Dr. Ken Perlin, for his support and imagination.

I also thank my external mentors and collaborators:

Rubaiat Habib Kazi, Li-Yi Wei, and Professor Haijun Xia

for their continued support.

Additional thanks to all of my friends and peers.

Abstract

This thesis introduces the design and implementation of an interaction concept called DrawTalking. Through simple combinations of sketching and speaking, the user can improvisationally build an interactive world of graphics, animations, diagrams, and dynamic mechanisms with behavior and rules, as if by narrating a story or explaining a concept to an audience. The interface demonstrates a possible step towards designing future interfaces more closely in-tune with how we naturally communicate and think.

For context, sketching while speaking has played a major part in innovation across disciplines. The combination of visuals and spoken language enables us to make-believe: think about, describe, communicate, and interact with anything that we can think of, including things that do not or cannot exist in the real world. Evolving technology creates opportunities to move beyond sketching and speech alone. Human-computer interactions of the future, drawing inspiration from our process of make-believe, can add interactive computation to the combination of sketching and speech, allowing us to work with explorable worlds, simulations, and mechanics. By enabling such interactions, we might think, learn, design, play, and tell stories in increasingly expressive ways.

Towards this idea, what makes for a good interface for computation-mediated sketching and speaking? This touches upon several fundamental questions in interaction design, human-AI interaction, and human-centered interfaces, chiefly among them, how to balance human control and machine automation?

Inspired by real-world speaking and sketching interactions, and seminal works in dynamic sketching, interactive visual programming, and language interfaces, we designed interaction techniques that draw on the way people describe objects and phenomena when telling stories and explaining processes at a whiteboard.

How does it work? the user speaks to label hand-drawn sketches with names and properties, and to define rules for how their world should behave. This communicates semantic intent to the computer, while giving the user the flexibility to choose how to represent and change their drawings. Now the user can interact with a simulated world simply by narrating stories or describing mechanics, which dynamically creates running interactive programs from built-in primitives and user-customized rules.

To gauge understanding of the mechanics of DrawTalking and to derive use cases, we invited participants to an open-ended one-on-one user-study session with the researcher to discover and explore the features in DrawTalking. Each user improvised and prototyped interactive sketch-based animations and gameplay scenarios by collaborating with the researcher. The resulting artifacts and discussion were oriented around each participant's specific experiences and background.

Feedback suggests that our approach is promising and intuitive: it prioritizes user control; it is flexible and supports improvisation; the workflow is fluid; the features are extensible and adaptable to other application domains and contexts beyond sketching; the design demonstrates how multiple applications can use similar language-based interaction techniques and behaviors predictably alongside other language-based technologies; it enables programming-like capability without code.

Through the research and design process of DrawTalking, we learned that it could represent an approach to designing complex interoperating systems for human-AI collaboration. We hope it can serve as a useful example for research and design of future machine-mediated interfaces, interactions, and computer systems.

Contents

List of Figures

List of Tables

List of Appendices

1 Introduction

Sketching while speaking has played a major part in innovation, thinking, and communication across disciplines [Fan et al. 2023; Tversky 2011], facilitating a wide range of activities across storytelling, animation, games, education, presentation, iterative design, spatial problem-solving, and many others [Subramonyam et al. 2020; Snyder 2013; Agrawala et al. 2011; Sturdee and Lindley 2019; Walny et al. 2011; Novick et al. 2011; Victor 2013; Chandrasegaran et al. 2018].

We can think of the combination of sketching and speech as a means to play make-believe; When we sketch while speaking, we create concrete representations and describe them to allow for a shared understanding of ideas that emerge from our imaginations [Turner 2016]. Sketching provides a concrete, visual form or symbol for an idea, and language adds semantics to those sketches. The combination helps people to visualize and think about the mechanics of how ideas behave and interact.

Sketching while speaking allows us to act out scenarios and to simulate outcomes. In doing so, we tell stories, iterate on designs, and think things through by improvisation or spontaneity. By making-believe through abstract sketches and spoken language, we metaphorically create a running and explorable simulation of an interactive process.

The ability to combine visual and verbal information is fundamental to communication, thinking, and creativity, and perhaps is miraculous: The combination of visuals and spoken language enables us to think about, describe, and share anything that we can think of, including things that do not or cannot exist in the real world.

Evolving technology creates opportunities to move beyond sketching and speech alone. Human-computer interactions of the future, drawing inspiration from our process of make-believe, can add interactive computation to the combination of sketching and speech, allowing us to work with explorable worlds, simulations, and mechanics. By enabling such interactions, we might think, learn, design, play, and tell stories in increasingly expressive ways [Victor 2014].

What makes for a good interface for computation-mediated speaking and sketching? Introducing computation and machine intelligence raises a number of research questions regarding design choices, functionality and intended audience, including:

- What are the essential mechanics of such an interface? i.e. how does the machine receive and interact with user intent? What is the balance between user interaction and machine automation? How should the workflow be designed?
- What are the technical trade-offs in a concrete implementation of this kind of interface?
- How and for what purposes is such an interface useful, as an extension of our existing language and sketching ability? What are the use cases from the perspective of potential users?

Related to all of these, common problems in interface design revolve around figuring out how to balance user-machine control, how to give agency to the user, and how to capture the user's intent.

To address these questions, we begin with the assumption that a good speaking and sketching interface should transparently "blend-in" with how we already speak and sketch, serving as an extension of the user's abilities. Furthermore, in the spirit of make-believe, we assume that the user will want to be able to explore and improvise in a live environment, which means that we will need to give them creative freedom and flexibility, and the ability to create new things without prior preparation. Above all, we wish to understand how best to extend the creative *process*.

1.1 Research Motivation

To help define the scope of our research, we start by examining a broad array of existing interactive interfaces that contain elements of sketching and animation, language-based interaction, graphical programming and game play to see what choices were made.

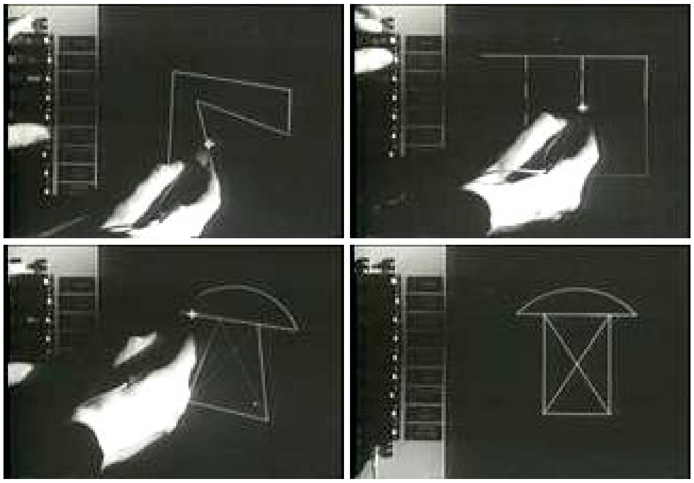
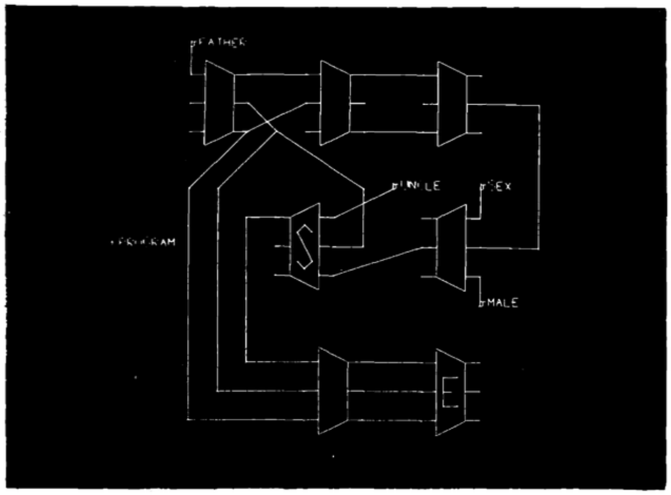


Figure 1.1: First GUI, Interactive Sketching — Ivan Sutherland's "SketchPad"



Graphical Associative Programming Language
Figure 6.1

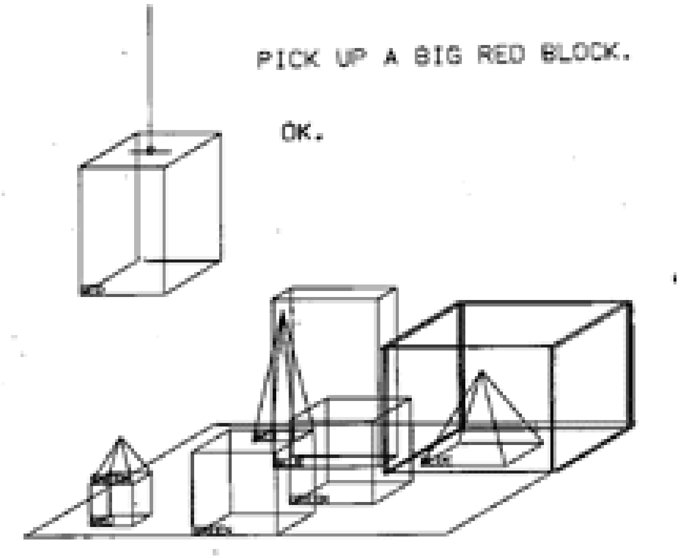
Figure 1.2: (Figure from) William Sutherland's Thesis on Visual Program Specification

Ivan Sutherland pioneered the graphical user interface (GUI) with SketchPad, whose machine-mediated capabilities aided in computational illustration [Sutherland 1963]1.1. In this work the machine interpreted the user's rough drawings to create more precise lines and shapes. William Sutherland then introduced visual programming via nodes and connections between the nodes, while incorporating the user's drawings as a way to specify elements of the visual program [Sutherland 1966]1.2.

In both cases, the user's intent was predicted by the machine according to known examples.

Both works inspired subsequent interfaces to incorporate increasingly dynamic and programmatic illustrations, empowered with relational models, logic, and procedural animation.

Meanwhile, natural language interfaces such as Terry Winograd's seminal SHRDLU have enabled users to communicate with machines in more natural ways [Winograd 1972]1.3.



```
PICK UP A BIG RED BLOCK.

OK.
```

Figure 1.3: Seminal Language-Interface — Terry Winograd's "SHRDLU"

SHRDLU was a machine agent that received natural English text as keyboard input and appeared to respond intelligently to users' directives. Its task was to edit and manipulate a scene of 3D objects according to the user's intent. In response to the user's typed directives, SHRDLU moved objects in a logical manner and could even respond with natural language to justify why it performed intermediate steps. This was one of the earliest examples of an AI interactively working together with the user in a simulated environment, using only language. Here the user's intent is telegraphed purely from language, and the user is communicating with the machine agent as though it were human.

Similarly, in Richard Bolt's "Put That There" demo, the user addressed the machine directly via a combination of speech and pointing as though it were a human assistant. It was an early and influential example of multimodal speech and gesture input [Bolt 1980].

In general all categories above combined have supported a broad range of interfaces such as 3D scene creation, editing of animation and video, and conjuring of graphical content based on language context. Many works likely derive from SHRDLU or similar ideas. e.g. [Cohen et al. 1997; Coyne and Sproat 2001] for language-based command and scene editing, [Subramonyam et al. 2018; Xia 2020] for editing of animation and video, or [Huang et al. 2020; Liu et al. 2023] for conjuring-up of graphical content based on language context.

A large variety of previous research across dynamic illustration, visual programming, and language-based or language-inspired interfaces borrow and mix ideas from these forerunners in interface research, for example, by integrating relational models, logic, and procedural animation. These have run the gamut of use cases across interactive animation and illustration[Kazi et al. 2014a; Suzuki et al. 2020; Subramonyam et al. 2018; Saquib 2020], live presentation and storytelling[Perlin et al. 2018b; Saquib et al. 2019], and UX/UI design and systems prototyping [Landay 1996]. In some sense, these all converge towards common goals with similar approaches: enabling increasingly natural interfaces through a mix of creative user input, language and/or machine intelligence, and some form of programming.

We observe that in many cases:

1. The machine is treated as an anthropomorphic machine assistant or agent.

2. The machine is responsible for interpreting the user's input to produce an intended output.

The system assumes a given representation is the correct one.

3. The goal of many interfaces is to help create a specific desired output.

4. Often the user is expected to know precisely in advance what their intended outcome should be, and must register graphical elements before using the system [Liao et al. 2022; Saquib et al. 2019].

5. Many interfaces support very explicit editing with feature-rich UI.

6. Tools incorporating programming elements tend to require explicit programming knowledge.

These are often useful qualities, but our research explores different directions that do not always align.

1. Treating the machine as a human assistant simulates collaboration with another person or other social situations and can create positive impressions on the user. Many interfaces can successfully engage the user this way. However, this human-machine relationship might create the feeling that the capability is in the assistant, not the person. In our research we are interested in exploring one of the alternative approaches, in which computational capability might seem linked directly to the person's natural sketching and speaking ability. For example, when narrating a story or explaining (out-loud to themselves or to an audience), a person would normally not address a third-party with imperative commands. Rather, they would use narrative and description. To an audience, the speaker might appear to cause something by their own abilities. With a machine agent approach, the speaker might appear to address a limitation in their abilities; asking for help or giving direct commands to an external agent explicitly delegates a task. It also directs attention away from a potential audience. We think that fully-developed interfaces might benefit from supporting all modes of addressing the machine in different social contexts and use cases, e.g. addressing

an agent, addressing an element within the interface, narration-style, or others. To scope our research, we choose to focus on enabling a narration style because it fits well within existing storytelling, explanation, and teaching contexts. It also works towards the idea of extending the user's abilities. We'd like to move towards instilling the capabilities in the user, or creating the appearance that this is the case.

2. Systems might assume that graphical elements should have a realistic representation, or rather, their algorithms assume a representation given rules or evidence from data, but this limits the user's individual agency and creative options. If we look at many sketch-based and speech-based interfaces, the machine is often assigned the task of inferring what a sketch represents or what the user intends (e.g. through sketch recognition and speech recognition). That is, the machine assumes to the best of its ability what the semantics of the user's content should be, based on the user's input. However, this means that the machine is prescriptive in that the user must provide certain inputs or sketch a certain way to get best results. Especially for rough sketching, we should hope that the user is in-control of how their content looks, and that the machine need not assume a particular representation of the content.

3. Many interfaces, e.g. generative ones, try reading the user's intent to create a specific output as a singular goal. We are interested in supporting interactivity and matching the user's imagination and creative process. In this case, there is no correct or best outcome. Rather, there is exploration and iteration. We can take advantage of tools for creating specific outputs at given moments over the course of our process, but we treat this as a complementary direction. This is one reason why we are interested in supporting open-ended interactive simulation and "programming," as these represent more complicated open-ended results and interactive play, rather than a single desired outcome.

4. Within a creative process, the user often cannot know what they want until they have

worked through their own creative process [Compton and Mateas 2015]. The flexibility to change is necessary.

5. Many interactive visual programming environments and games (for example) give users expressive capability and control. They were designed with playful exploration in mind, encouraging users to tinker with worlds [Solomon et al. 1986; Resnick et al. 2009; Maloney et al. 2010; Dietz et al. 2021; 2023; Little Big Planet 2008; Little Big Planet 2 2011; Dreams 2020]. These arguably prioritize a playful "feel" fitting for an imagination sandbox and come closest to our vision in that sense. However, they often rely on complicated editing or explicit programming interfaces. This comes at the expense of the fluidity and flexibility one might expect in real-time scenarios.

6. We'd like to move to a point at which people, including beginners, can create simulations and interactive elements that behave like programs, but do not require traditional programming expertise.

In summary, our initial assumptions are that we are looking for an interface without an explicit AI-agent, which supports a creative exploration process with interactive capability, but without imposing too many assumptions about the user's intent or content. The focus should be on the process, not just on the artifact. It should not use an explicit UI or require programming knowledge. Above all, the user should have control.

1.2 Formative Motivation

Motivated by these initial assumptions, we first focused on exploring playful sketch-based interactions more deeply in several different contexts. The goal was to better understand what forms of content people made, as well as how they behaved while speaking and sketching.

We did an informal survey of sketch content, both live and previously-produced (such as classroom talks and educational videos). Next, we ran design exercises in which 6 participants narrated their own personal stories while drawing.

- *Found content*: We saw playful stories with abstract figures, playful and stylized animations with simple movements, and an overall rough sketch aesthetic. Even in previously-produced content, people find the rough aspect to be endearing and do not require complicated content.
- *Observed behavior*: During sketch+speech interactions, we observed that people aimed for flexibility and control, continually revising and reworking their drawings, moving things around, and iteratively updating objects. People would add visual hints such as arrows or text labels to describe objects, or add props or set-pieces into the sketched scene. We concluded that sketching+speaking in these live contexts is a live process of working things out. These observations confirmed our hypothesis that people need to be able to improvise during live sketching.

A key insight from our observations was the way in which people conveyed information by using drawings, speech, and text together. Simply, people draw or display sketches while verbally narrating and/or annotating with text. This spoken narration and text serve a dual-purpose: 1) engaging with an audience (as in explaining or telling stories) or thinking out-loud; 2) describing: i.e. leaving a visual or auditory artifact indicating *what* a sketch represents, *how* it behaves, and *how* the world behaves – naming objects, rules, and phenomena out-loud or by text, referring by deixis (words like this/that/those) [Stapleton 2017], and touching or pointing to objects.

We observed that 1) represents the user expressing their creative intent, and 2) is exactly what a computational system would need in order to understand that intent. So if an interface could tap into the information expressed by the user in these situations, then the result would be a positive experience for the user. Secondly this could be a channel for the computer to receive the user's intent without interrupting the user or imposing creative assumptions such as a required visual representation. In other words, our goals could be a form of human-AI collaboration: The user's natural output is their language, which becomes input to the computer that communicates semantic intent, which then becomes the interface for the user to create and control an interactive world.

Such an interface might work as follows: If users can name and define properties on their drawings within the digital medium, the computer can listen to this information and know the user's intent regarding what is in the world and how it behaves. This way, the user decides the representation and behavior of their sketches, just as they might during narration at a whiteboard or when playing make-believe to make the "abstract" concrete. The computer is therefore not required to guess what a sketch represents. Once the computer knows what entities are in the world, the user can describe events in a story or explanation while referencing objects in plain spoken language (English), and the computer, having received the user's intent, can then apply animation and simulation.

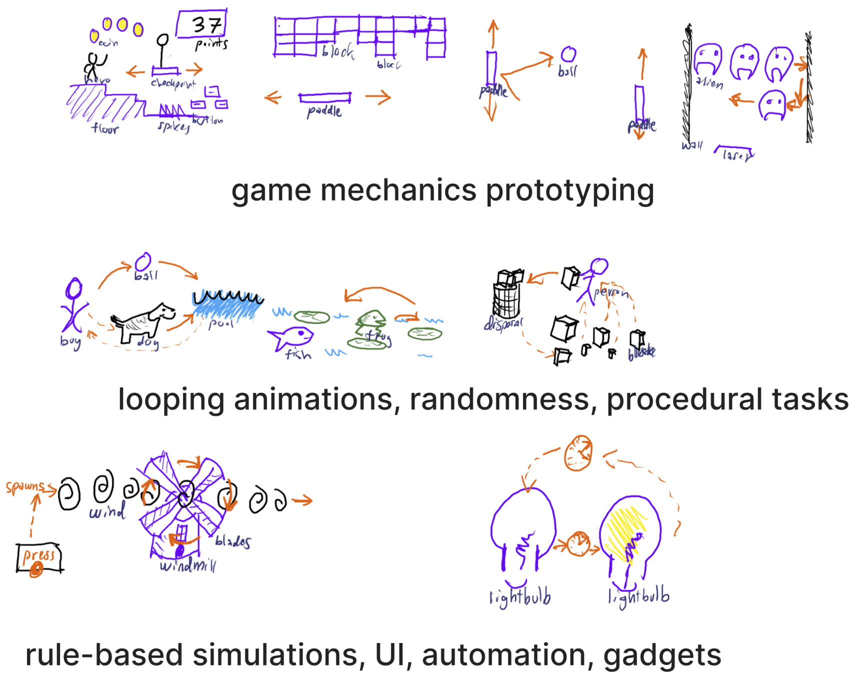
The question remains of *when* things should happen during this process. Should the computer decide when an operation should happen? We hypothesized that the user should decide.

We further hypothesized that an effective interface for extending speaking and sketching with computation should incorporate all of the above: Flexibility and fluidity, a focus on improvisation and play, and interaction techniques based on treating the user's natural behavior as semantic labeling and input to computer software that then "makes things happen."

These were the essential components that led to prototyping a proof-of-concept.

1.3 Project

We introduce *DrawTalking*[Rosenberg et al. 2024], an interactive software system whose users speak while freehand sketching to create, control, and iterate on interactive visual mechanisms, simulations, and animations 1.4. Through speech and direct manipulation, the user names their



game mechanics prototyping

looping animations, randomness, procedural tasks

rule-based simulations, UI, automation, gadgets

Figure 1.4: A subset of demo categories in DrawTalking

sketches to provide semantic information to the system. Given this information, DrawTalking offers users computational capability for interactive animation and simulation within worlds of their own creation. This results in interactions that are similar to explaining, discussing, or storytelling while pointing at a whiteboard in everyday life, but with added capabilities of interactive computation which provide some of the power of programming. The workflow is a type of human-computer collaboration in which the user and computer help each other with minimal effort, and in a way that feels natural to the user. The result is a step toward the development of future mature interfaces that provide computational support for sketching while speaking.

We implemented the DrawTalking prototype as a multitouch application on the iPad that runs as a continuous simulation in which the user is free to build-up and explorable their own worlds through freehand sketching and speech. Controlling the system entails the user alternating between (1) directing sketches to perform actions and (2) defining rules and relationships that dictate how sketches should interact and behave in the future.

DrawTalking synthesizes approaches from prior interfaces, while addressing the limitations of those prior interfaces to enable more creative and exploratory use cases:

- It **prioritizes user control** because the user defines what their drawings are named and what they do, not the computer. This puts the responsibility into the user's hands and assumes little about their intentions and the content they are making. By giving the user more control over how their abstract sketches should be named, the system does not need to perform inferencing on the user's world on its own, thereby reducing the number of possibilities for error and promoting greater user agency.
- The workflow is **flexible and supports improvisation** because users can change their minds during any point of their creative process to revise the look, behavior and properties of their drawings. At any step they can narrate via full spoken or typed sentences to modulate sketch behavior, either by giving commands or by defining persistent rules. Through this approach, users can quickly iterate and try out their ideas for interactive mechanics and worlds within a sandbox of their own design that enables interactions for gameplay mechanics, animations, and stories, all through simple combinations of sketching and talking, with no preparation of content required.
- The workflow is **fluid** because all operations can be done consistently and with easy access using just pen, touch, and speech, with low complexity in the interface and high spatial proximity of all elements.
- The system is **extendable and adaptable.** It implements several language primitives that can be parameterized interactively based on sketches' labels. These primitives can be composed together into more complex processes at runtime without needing to code entirely new-behaviors from scratch. DrawTalking's interactions are general and can be feasibly adapted to or extended by other domain-specific applications.

- DrawTalking's interactions give users **programming-like capability without requiring them to write code.**
- DrawTalking is implemented with **reproducible research, portability, and interoperability with other language-based applications** in-mind. It **demonstrates how we can convert natural language input into a generic format for broader use.** We chose to integrate built-in primitives to enable a deterministic and stable target that is easier to understand and introspect. By compiling external language processing output to an intermediary format that is generic, deterministic, and interpretable, we can develop interactive language-enabled systems independently from natural language technology, and benefit from better technologies as they improve.

To test our approach, we studied how users responded to DrawTalking. Do people find it useful? Are its mechanics understandable? Can people derive their own use cases and possible re-applications?

We chose to use a qualitative experimental setup, not only because there is no baseline system to compare against, but also because at this stage, we care more about user's personal feedback without imposing a specific use case. The original spirit of the project was to work toward capturing a spirit of creativity and make-believe, so we likewise aimed to capture the open-endedness of a creative process.

In our user study, we invited participants to the features in the system through an open-ended discovery session one-on-one with the researcher, who served as a guide and collaborator. The researcher would ask a participant to draw a few objects of their own, and would then go on to introduce more advanced features, improvised around the participant's own creations. Each user improvised sketch-based animations and gameplay scenarios. The final results comprised the artifacts from the improvised exploration and collaboration process with the researcher, and the deeper discussions and comments surrounding the exploration session. We chose this approach as an effective way to observe a participant being creative and collaborative, while introducing the complexities of DrawTalking within a short time-frame. During this process, we aimed to elicit meaningful discussion around the use cases and perceptions of the tool in comparison to the participants' personal experiences. The goal was to learn not only whether people found our approach intuitive, but if so, why and in what ways, and how did they understand the interface in relation to their life and work? What was their definition for why DrawTalking might be useful? Additionally, this study gauged paths for future research and improvements to our approach.

The collective feedback and observations from the sessions shows that DrawTalking has potential as a form of "digital make-believe" for multidisciplinary applications, and as a step toward designing future computationally-enhanced systems that support sketching while talking – and more generally – human-machine collaboration in a similar vein. We learned that users define DrawTalking as an intuitive and fluid approach to programming using language, with a wide range of use cases across game prototyping, design, education, and presentation. Our approach can generalize and has the potential to be adapted to or complemented in many possible contexts and systems.

In summary, our contributions are:

- ***DrawTalking***: A new approach within the space of live sketching+speech interaction that balances user-agency with machine automation. Through simple combinations of sketching and speech, the user constructs and programs interactive simulations by naming objects, defining rules, and directing the machine.

- ***DrawTalking* Research Prototype**: A digital sketching interface (for the iPad) that

  demonstrates how sketching and speaking can be used together to build, program, and control a simulated world of hand-drawn objects interactively. The implementation is an instance of our approach that showcases the interaction techniques in the context of 2D animation and playful game prototyping. The ideas generalize to other domains, and the implementation is designed to support future extensions and adaptations in other contexts.

- **User Feedback, Artifacts, Discussion, Future Directions**: Results from user studies in which users discovered features, created artifacts, and discussed the utility and impact of the tool and the ideas behind it. The results demonstrate the utility and potential of DrawTalking across several use cases for creative visual expression, prototyping, and design.

This thesis contributes a new approach to the sketching+speaking space, as well as a prototype that demonstrates this approach using existing technology. We hope for this work to inspire many positive future directions in human-centered interface design.

2 Prior Explorations: Part 1

Sketching as a Glyphic Language

2.1 In the abstract

Prior to "sketching+speech," we tried treating sketching as a form of context-sensitive visual language. The research platform, "Chalktalk" served as the testing ground for this idea. Chalktalk was an interactive canvas supporting procedural sketch objects for illustrating known concepts through animation and graphics. These objects were predefined and instantiated by drawing a known glyph. To command these objects, the user had to memorize which directional stroke to draw on an object to invoke the equivalent of a behavior. But a limitation of this approach was that these sketch-gestures were all the same across all sketches. However, we wanted to see how it might feel to extend the metaphor of sketching-to-bring-to-life beyond just creating objects. By drawing glyphs *on* sketches, we could create glyphic language to control sketches with visually-meaningful correspondence between the command being performed (the verb) and the object being commanded (the noun). We created a selection of glyphs the user could draw on top of other interactive sketches. These sketches would interpret the glyphs as commands. We picked the topic of computer science data structures education, as often when teaching algorithms, we'll describe gesture-like traversals of data (e.g. tree traversal.) In the rest of the chapter, we introduce Chalktalk and show examples of the "control by glyphs" approach [Perlin et al. 2018a].

2.2 Introduction

Especially in the context of a classroom, presenters must supplement verbal communication with visuals to illustrate concepts – specifically those whose behavior and representations change variably or over time. Such visualizations are crucial for effective conveyance of ideas in areas such as physics, computer science, and animation, wherein ideas revolve around dynamic, interactive entities. However, traditional media such as blackboards, slide shows, and (more recently) electronic smart boards allow only for static drawings and text/image sequence, or in the best cases, fixed animations made in advance[Nunes and Perlin 2017; Perlin et al. 2018b].

Chalktalk is an open-source presentation and communication tool in which the user creates and manipulates interactive, animatable objects – called "sketches" – in real-time to demonstrate ideas[Perlin et al. 2018b]. Chalktalk contains a growing library of programmable sketches based on concepts from areas such as physics, mathematics, audio, computer graphics, procedural animation, and others. These can be controlled via mouse gestures and linked to form increasingly complex systems and to adapt to the changing flow of a presentation – particularly classroom lessons. Chalktalk presents the opportunity to improve our visualization of key concepts in computer science: especially data structures, whose data and form change over time due to the variety of interactions within a computer system. Like the popular Scratch[Resnick et al. 2009; Maloney et al. 2010] language, Chalktalk can help new programmers focus on higher level logic and concepts rather than on the syntactic peculiarities of a particular language. In addition, Chalktalk's links can transmit any data (including function callbacks) between Sketches, allowing for a more flexible system than one built for a specific domain (e.g. Max/MSP's audio modules [Nunes and Perlin 2017]). These characteristics, combined with the ability to interact and build with sketches in real time, make Chalktalk a promising environment in which to learn and explore computer science. Thus, we propose a computer science-centric Chalktalk library and contribute prototype sketches based on fundamental data structures – the binary search tree (BST) and stack. We will use these prototypes as the basis for further investigation of alternative visualizations and interactions for use in computer science education. Here we provide an overview of Chalktalk's system and use cases, then discuss our prototype data structure sketch implementations in detail. We conclude with notes on ongoing and future research using the Chalktalk platform.

2.3 Presenting with Chalktalk

2.3.1 Example Use Case

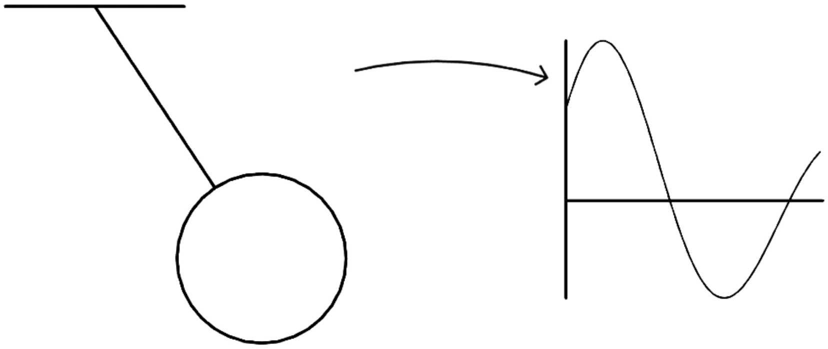


Figure 2.1: Pendulum Example: The user has dragged the mouse to swing the pendulum (left), which outputs its angle as numerical data for the graph (right) to display as a curve. In this case, the curve represents the pendulum's displacement. However, the graph can interpret all numerical data it receives, which means any sketch that outputs numbers can interact with the graph sketch.

A simple example use case[Nunes and Perlin 2017] is as follows:

Suppose a physics teacher wishes to illustrate the motion of a pendulum. The teacher should be able to draw the pendulum and swing it to demonstrate its physical properties. In addition, the teacher should be able to draw a graph to which she can link the pendulum to show the mathematical curve representing its displacement from equilibrium (see figure 2.1). Furthermore, this pendulum sketch should be reusable in a network of other sketches to allow for further experimentation – perhaps as a controller for the movement of other objects such as a fan (see

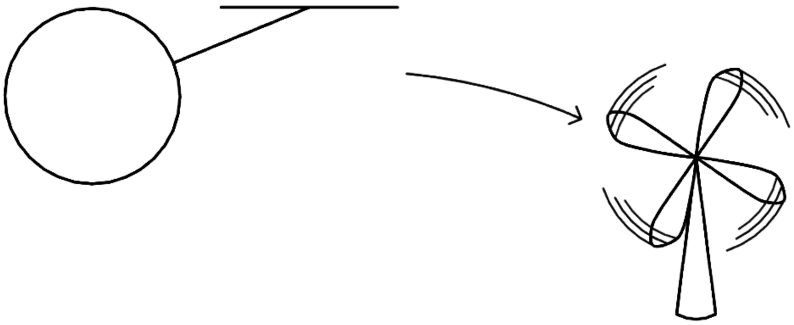


Figure 2.2: Pendulum Linked with Fan: The same pendulum from 2.1 is linked to a fan sketch (right), which uses the data from the pendulum to set its own angle, causing it to rotate. Neither sketch is aware of each other's types. Each merely sends and receives data that are interpreted independently.

figure 2.2) or for the configuration of a matrix to rotate 3D geometry (see figure 2.3).

2.3.2 User and Programmer Interface

2.3.2.1 Recognition

To instantiate a sketch, the user first free-hand draws a glyph, composed of a specific number and ordering of strokes. That glyph is compared against a library of glyphs (see figure 2.4) defined in Chalktalk, and the closest match is selected for recognition. Finally, the user clicks the recognized glyph to instantiate it as a sketch.

2.3.2.2 User Input

Sketches are able to recognize swipe motions, clicks, and drags as input events. Callback functions defined in a sketch can be used to trigger events in response to mouse gestures. There is also a general set of "command" gestures recognized by all sketches, which can be used to modify the scale, position, rotation, and other properties of the sketch itself. These command gestures always begin with a click around the periphery of the sketch.

2.3.2.3 Links and Data Transfer

Links transfer data from one sketch to another. Connecting a link from and to a sketch is a matter of wiring the link visually using a "command drag." This is achieved by clicking on the source sketch's left periphery, and then clicking and dragging an arrow from the source to the destination sketch. In the internal JavaScript code, an output procedure returns data for receipt by any number of other linked sketches, and each sketch may also directly access data sent to it.

2.3.2.4 Sketch Design and Implementation

Because Chalktalk is open-source, users may choose to use existing sketches or to design their own, either for personal use or to contribute to the growing repository. This means that users (e.g. teachers and students) need not be programmers. Nevertheless, one of Chalktalk's strengths lies in its programmability. Chalktalk runs in the browser, so sketches are written as JavaScript files in which swipe, drag, output, render loop, and other methods attached to that sketch type are defined. Designing the appearance of a sketch as well as its glyph (used for recognition) involves calling draw functions that specify curves, colors, and other attributes or applying matrix transformations. These functions take the form of mCurve(..), mLine(..), mOval(..), m.translate(..), m.scale(..), color(..) and others. The drawing API may be comparable to Processing's[Reas and Fry 2014], as it allows the programmer to think in terms of curves and shapes rather than low-level draw calls. As a user becomes more proficient in programming practices, all JavaScript language constructs (conditionals, loops, variables) may be used to adjust the appearance and behavior of a sketch. Methods such as this.output() may be defined in a sketch to send data across links, and within the Chalktalk interface, the user may also access a sketch instance's code for viewing or editing at run-time (see figure 2.5).

2.3.3 Narrative and Performance

Chalktalk is also a performative medium in which "programming" is not only the literal programming of sketches, but also the narrative that unfolds when combining and juxtaposing sketches in real-time. For example, a Chalktalk-based computer science library would include sketches that might simulate pieces of a program. A teacher could begin the conversation with her students by using sketches to introduce fundamental programming constructs such as the loop. Based on student engagement, she could then arrange, link, and compose the sketches into increasingly specialized systems of data structures and logic, mirroring the way in which traditional lessons build on previous concepts. The teacher may also invite students to design their own experiments based on the day's lesson. This sense of engagement and conversation is key.

2.4 prototype: computer science data structure sketches

We believe Chalktalk to be an ecosystem particularly fitting for the visualization of computer science concepts such as data structures, which may take on multiple representations and behaviors depending on use and composition. To show the potential of a computer science sketch library, we present two work-in-progress data structures sketches – a binary search tree (BST) and stack – that illustrate the interoperability and interactivity of sketches.

2.4.1 Binary Search Tree

The internal code of the BST sketch (figure 2.6) contains a reference-based tree implementation that the user modifies via mouse interactions with the sketch, which map to operations such as insert, remove, and preorder-, in-order-, post-order-, and breadth-first traversal. Operations may also be undone using a leftward swipe. Insertion and removal are initiated by creating a numerical sketch (built into Chalktalk) and dragging and dropping it onto the tree. This starts the recursive process of searching for the proper node, whereby the in-memory tree performs the algorithm while issuing draw commands to highlight visited nodes in the Chalktalk interface. Removal is implemented in a similar manner, but accounts for the additional cases in the removal algorithm by providing additional animations. For example, when the node to remove has two child nodes, the algorithm must find the predecessor or successor node to substitute (see figure 2.7).

When demonstrating a tree traversal, one way of appealing to visual learners might be to represent the sequence of parent/child node visits with a mouse gesture. The presenter would draw a special curve (see figure 2.8 for the design and figure 2.9 for an example) to trigger its corresponding traversal. This idea is implemented into the BST sketch by reusing the functionality of Chalktalk's sketch recognition system for the recognition of line strokes drawn atop the BST.

All interactive operations require timed pauses and interpolated movements across multiple frames, so functions responsible for modifying the tree make use of JavaScript's "yield" keyword and generators to save state when an operation is in progress (e.g. tree traversals). To supplement the raw language features and provide higher level interface, a small host of utility functions for timing and pausing was written. To run and animate an operation, the BST sketch enqueues the generator function specific to a given operation, runs the generator until a pause in the animation is necessary (a yield), renders the updated sketch, and checks the queue on the next frame to resume execution of the function. This process repeats until the current operation has been completed. Objects modeled after debug breakpoints can be inserted into the code path of a generator to allow the user to pause deliberately.

2.4.2 Stack

The LIFO stack sketch uses a JavaScript array internally as well as generators to save intermediary state over multiple frames, just as the BST does. To push a value, the user draws, for example, a numerical sketch, and drags-and-drops it atop the stack. To pop, the user does a downward swipe gesture with the mouse (see figure 2.10).

If a sketch is linked to the stack, additional logic will be used to decide whether it is necessary to push or pop, as the link might send the same data repeatedly. Duplicate pushes and pops of that same data should be ignored.

2.4.3 Final Expected Behavior

The final, completed versions of the BST and stack sketches will exhibit new behavior when linked: during a traversal, the BST will output information about the most recently visited or exited node, and the stack will interpret this data as stack frames to push and pop. As a result, the linked tree and stack will show the relationship between recursion and a call-stack. The current in-progress implementations interact differently for now: the tree outputs a record of the most recently performed operation insertion or removal, and the stack displays the history of these records (see figure 2.11).

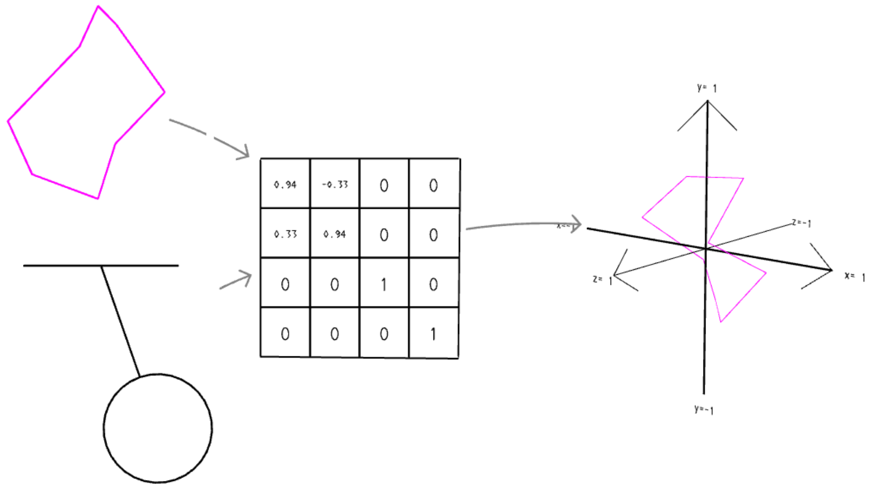
Thus, the BST and stack have new behaviors when linked, just like the pendulum and graph. Once implemented, arrays, FIFO queues, graphs, and other data structures could could be used to interact in a larger high level simulation of a program or with other compatible sketches such as matrices and vectors.

2.5 Ongoing and Future Research

One of the main strengths of Chalktalk's system is that it gives the user the freedom to build and use sketches in a variety of ways. In-development Chalktalk features such as improved code hot-loading and property editing will likely impact the way in which sketches are designed, and as a result, will lead to changes in the proposed data structures library as we experiment with different implementations. Chalktalk has, in fact, been evolving. It lies at the center of research that includes the development of a type system for Chalktalk's links[Nunes and Perlin 2017] and the creation of interactive augmented reality (AR) and virtual reality (VR) environments [Perlin et al. 2018b]. In addition, Chalktalk is being used in classrooms to teach computer graphics, animation, sound processing and other subjects [Perlin 2016]. In the near future, we also wish to conduct case studies on a larger scale.

2.6 Conclusion

We have shown how to extend Chalktalk's recognition system's usefulness beyond recognizing object primitives (nouns) to recognizing (verbs) via a form of context-sensitive visual language. Sketch recognition is useful not only as a way to instantiate visual content, but also as a means to express commands via a learnable domain-specific visual language. It can enable the user to control visualizations without an explicit GUI, in favor of staying in the flow of sketching.



0.94	-0.33	0	0
0.33	0.94	0	0
0	0	1	0
0	0	0	1

Figure 2.3: Data Flow of Sketches for Linear Algebra: In this case, the matrix and coordinate system, too, process only data transferred between sketches – not the sketches themselves. This data-oriented design allows for the flexible combination of and interaction between sketches. Top-left: a 3D shape whose output is mesh data; Bottom-left: another pendulum; middle: a transformation matrix set to rotate on z; Right: a 3D coordinate system – The matrix receives an angle from the swinging pendulum, which it uses to set the rotation. The matrix processes, transforms, and outputs the shape data, which the coordinate system displays.

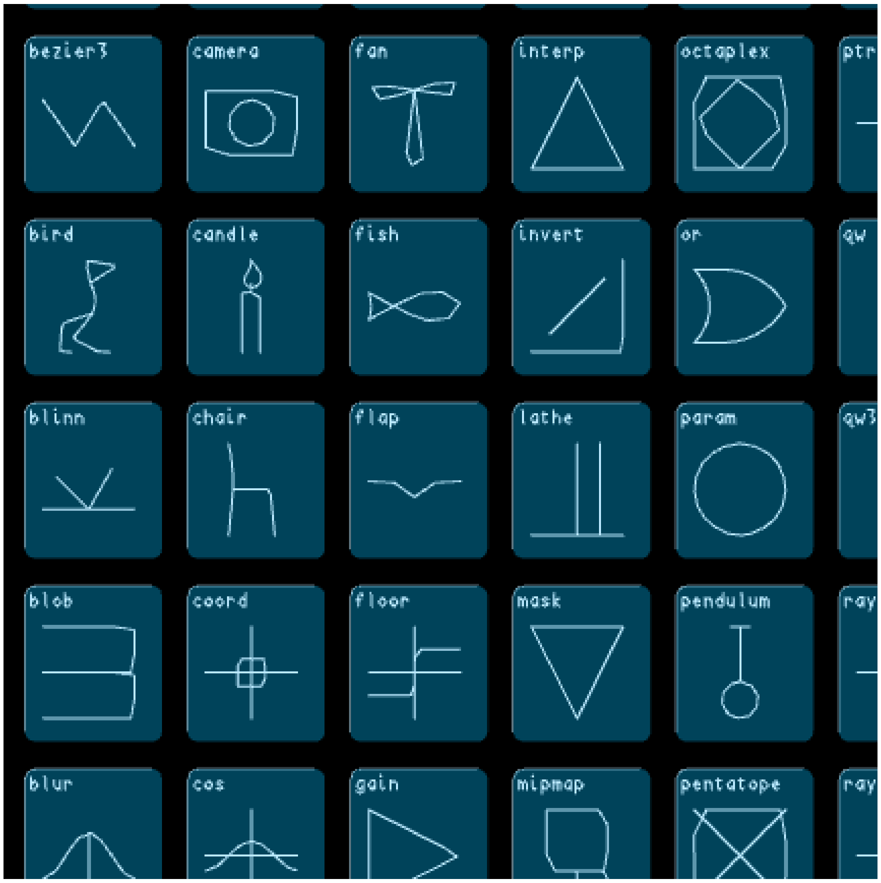
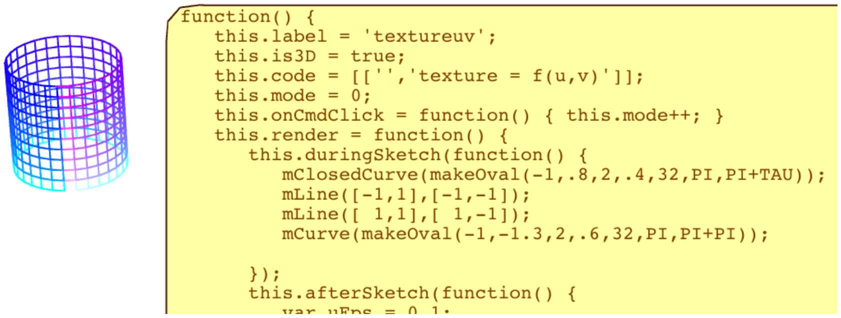


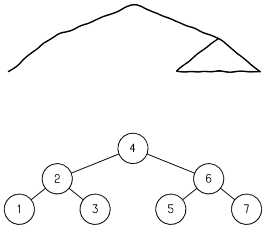
Figure 2.4: Chalktalk Sketch Library: A cross-section of Chalktalk's library, containing mathematical functions, creatures, geometry, and other entities and concepts



```
function() {
   this.label = 'textureuv';
   this.is3D = true;
   this.code = [['','texture = f(u,v)']];
   this.mode = 0;
   this.onCmdClick = function() { this.mode++; }
   this.render = function() {
      this.duringSketch(function() {
         mClosedCurve(makeOval(-1,.8,2,.4,32,PI,PI+TAU));
         mLine([-1,1],[-1,-1]);
         mLine([ 1,1],[ 1,-1]);
         mCurve(makeOval(-1,-1.3,2,.6,32,PI,PI+PI));

      });
      this.afterSketch(function() {
         var uFps = 0.1;
```

Figure 2.5: Chalktalk Live Coding: All sketch codes can be exposed in the Chalktalk interface. On the left is a 3D cylindrical wire-frame mesh. On the right is a fragment of its code, which can be edited live.



a The glyph that is recognized as a BST sketch

b The BST sketch in its initial state, pre-populated with nodes

Figure 2.6: Binary Search Tree Sketch Recognition

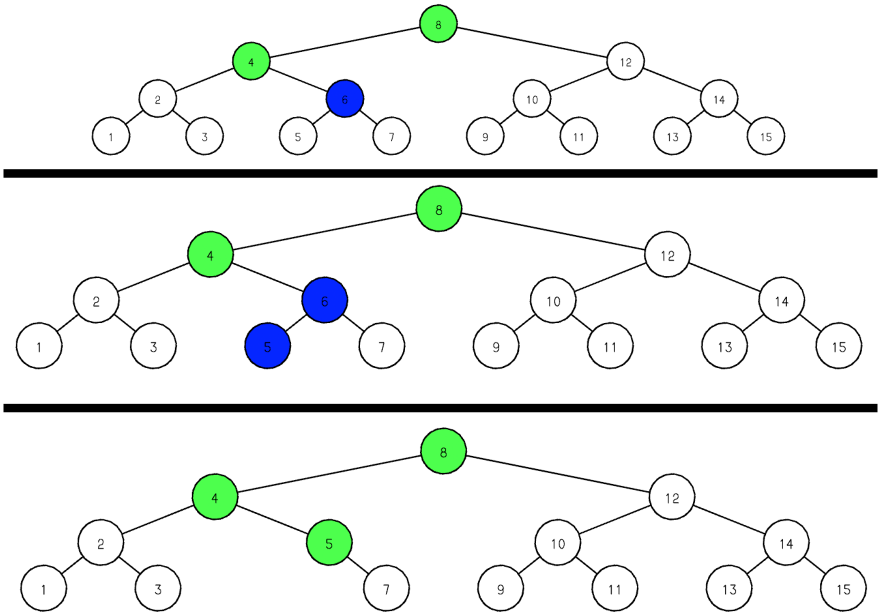


Figure 2.7: Binary Search Tree Sketch: The node removal visualization for the BST sketch – top: because the user has selected the now-blue node for removal, the algorithm has recursed down to that node; middle: since the node has two children, the algorithm searches for the predecessor node for a replacement; bottom: the blue node has been replaced with its predecessor (note that this recursion sequence is animated)

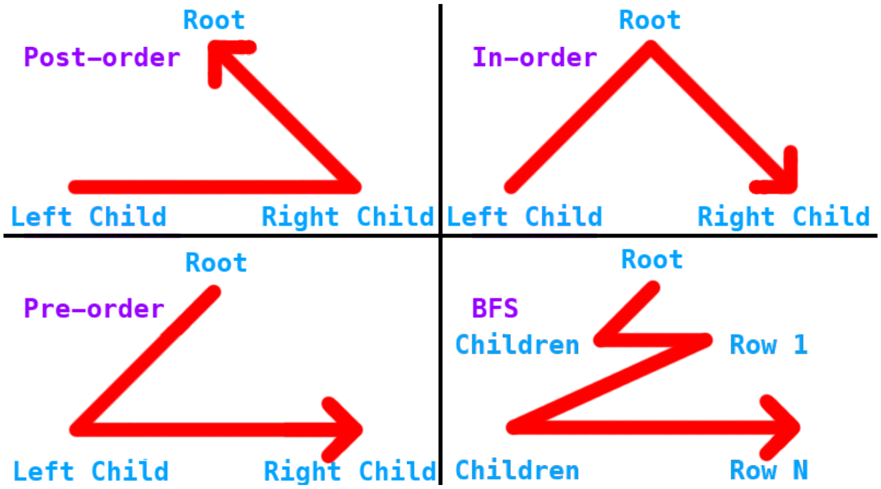


Figure 2.8: BST-specific Sketch Gestures: Seen here are the four BST-specific gestures that map to the different traversals the sketch supports. They are meant to serve as (experimental) visual mnemonic devices for learning purposes. For example, in post-order traversal, all children are visited first, so this is represented as an arrow moving from left child to right child, and then to the root. (For in-order traversal, the arrow moves from the left child to the root node, then to the right child. In Pre-order traversal the arrow moves from the root to the left child, and then to the right child.) Breadth-first search (BFS) traverses the tree in layers, so a zig zag from root to child represents this process. We are interested in pursuing research related to these sorts of mnemonic devices.

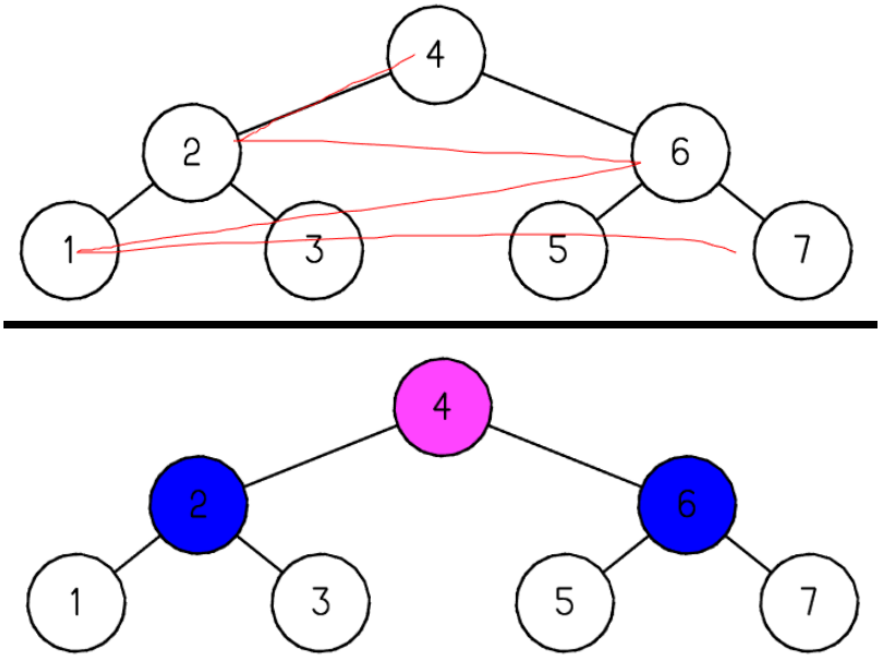


Figure 2.9: BST-specific Sketch Gesture in-progress for BFS: top: the user has drawn a zig-zag curve atop the BST to start a breadth-first search traversal; **bottom:** the traversal is underway – the algorithm is now visiting the first row of child nodes (in blue)

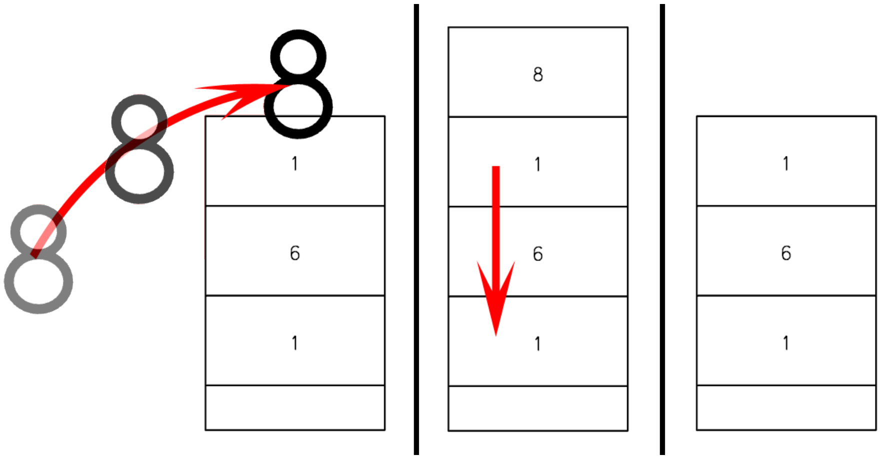


Figure 2.10: Stack Sketch: left: the user drags and drops an "8" sketch onto a stack sketch containing the values 1, 6, 1; middle: the 8 has been pushed to the top of the stack, the user now clicks and drags downwards to pop the stack; **right:** the stack has been returned to its previous state

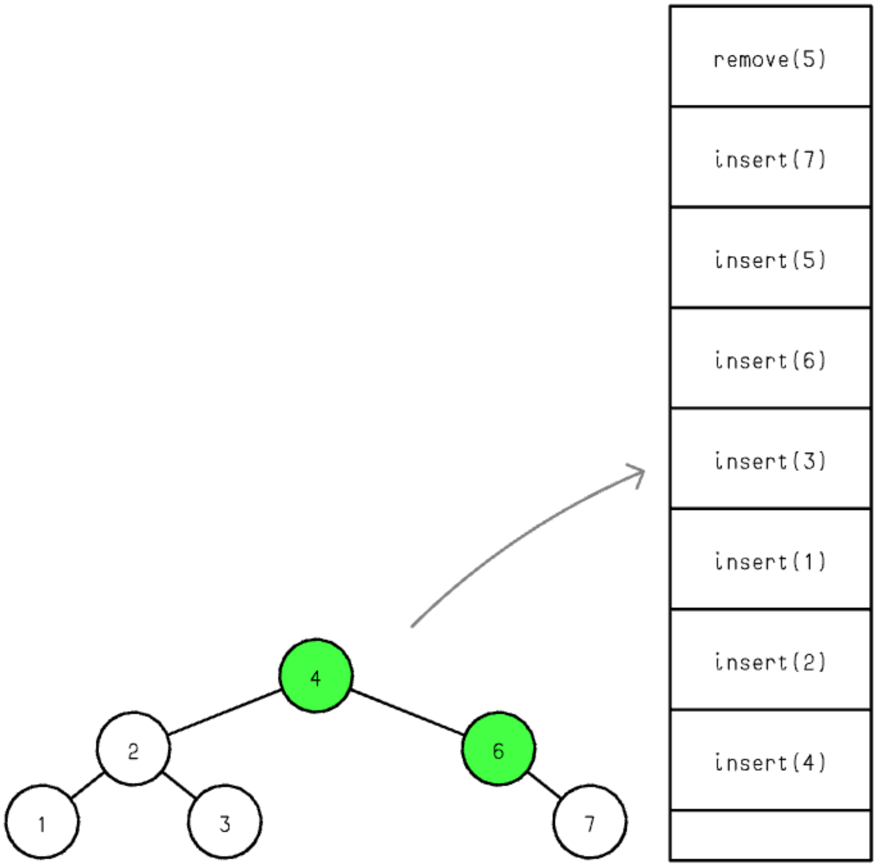


Figure 2.11: BST and Stack Sketch Interop: The tree currently outputs a record of the operations it performs. Because the stack can accept any form of numerical or string data, it can display these records as a history. It can also interact with any sketch in the Chalktalk world that sends compatible data. At the top of the stack seen here, a remove(5) record indicates that the node with value 5 has just been removed. When complete, the BST will output information that the stack can use to simulate a call stack during a recursive algorithm.

3 | Prior Explorations Part 2: Exploring Configuration of Mixed Reality Drawing Surfaces for Communication; Enabling Surface Sketching for 3D Spatial (VR) Environments

3.1 In the Abstract

Here we propose interactions in virtual reality (spatial/immersive environments) in which we enable seamless switching between kinds of 2D drawing surfaces in 3D space, support movement of 2D surface drawings into 3D space, and facilitate mutual gaze awareness in collaborative contexts around these drawing surfaces[He et al. 2019](figure 3.1). We continue to use Chalktalk as the research and content platform.

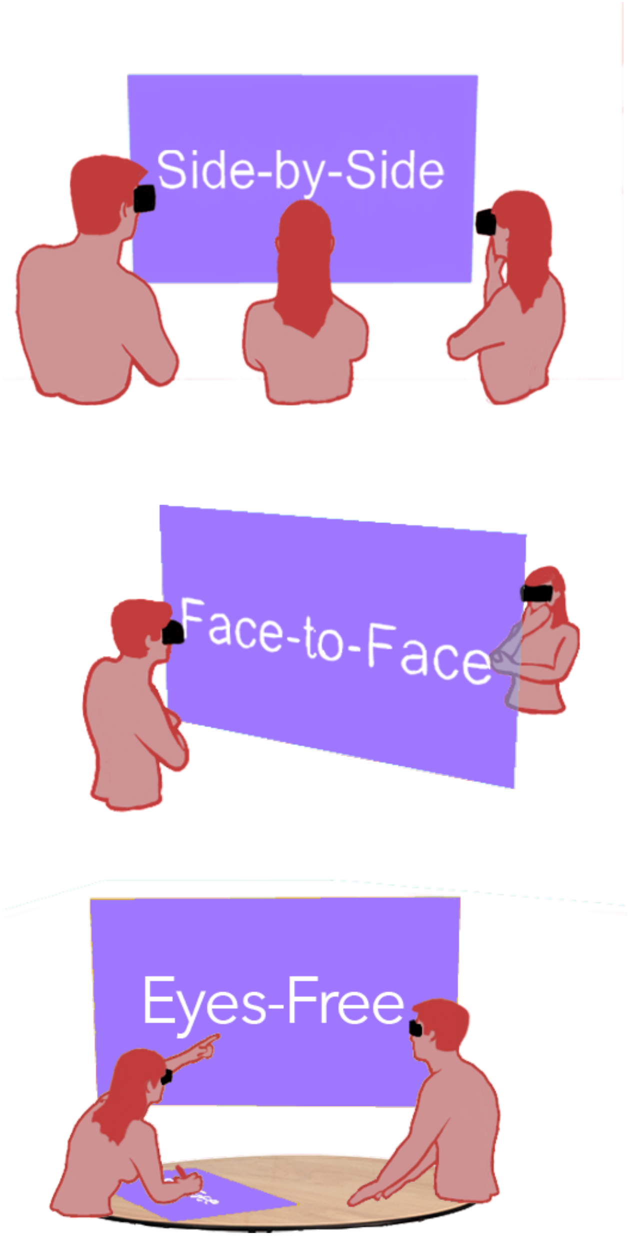


Figure 3.1: Three Configurations for Mixed-Reality (MR) Communication:
Inspiration for configurations:
1) side-by-side whiteboard brainstorming,
2) daily face-to-face conversations and
3) drafting boards

Table 3.1: Terminology Table

Term	Definition
Content Creation Server	The content creation server is an external server which takes raw drawing point data as the input, recognizes the drawing, and creates interactive objects.
Content Board	The content board is the transparent area on which content is displayed. The user's input will be projected to the content board and sent to the content creation server for processing. Afterwards, the result will be displayed in 3D in the transparent area.
Configuration	The configuration refers to the placement and orientation of users and the content board with respect to each other.

3.2 Introduction

Virtual Reality and Mixed Reality (VR, MR) are being explored increasingly, spurred by the availability of high quality consumer headsets in recent years. VR and MR enable rich design spaces in HCI by providing 3D input and immersive experiences. Decades ago, the "Office of the future" was proposed to allow remotely located people to feel as though they were together in a shared office space [Raskar et al. 1998], via a hybrid of modalities including telepresence, large panoramic displays and shared manipulation of 3D objects. The core idea was that VR/MR had the potential to enhance communication among groups of people. Since then, significant progress has been made in exploring techniques for communication [Ishii et al. 1993; Otsuka 2016]. However, less studied is the configuration (see our full definition in table 3.1) of people and shared manipulable objects in the environment, which may lead to different communication experiences.

In daily life while speaking to others, we commonly use gestures or visual aids to help present ideas, either subconsciously or purposefully. Visual aids can be drawn on paper, a whiteboard, or a screen via video chat. A key task for collaborators is the shifting of focus between spoken words, gestures and visual aids such as notes and drawings. Smooth transitions in conversations have been found to be important for collaboration [Buxton 1992]. Prior work addressed this with alternative configurations of spaces for communication. Tan et al. built a face-to-face presentation system for remote audiences [Tan et al. 2010]. ClearBoard [Ishii et al. 1993] created a shared workspace in which two users collaborate remotely without losing all the advantages of in-person face-to-face interactions. One such advantage relates to learning. When a teacher looks away from the audience, Lanir et al. [Lanir et al. 2008] observed that audiences in a classroom would not focus on the presenter, which might "create a learning environment in which there is no interpersonal engagement between the presenter and the audience, thus reducing learning outcomes." We find that personal engagement, such as one-to-one interaction and eye contact [Insa et al. 2016], in addition to focus shift [Buxton 1992], is therefore relevant to evaluating communication experiences.

Still, it is unclear how the configuration of users and content in an MR space affects communication for co-located and distant people. We have implemented a multi-user MR workstation to explore how configuration impacts communication by providing three different configurations: 1) side-by-side, 2) mirrored face-to-face and 3) eyes-free. We conducted a preliminary user-study of the mirrored face-to-face configuration. Afterwards, we proceeded to user interviews in which participants spoke about the face-to-face experience, one-to-one interaction, focus shift and eye contact.

Chalktalk's content ("sketches") is composed of graphical elements such as point data. 1) Sketch data are serialized on the Chalktalk client side every frame and 2) sent as a data array to the Chalktalk server, 3) to the Holojam relay, and then 4) to all Unity clients, where 5) the data are rendered on content board(s). 6) MR user input is sent back through the pipeline to the Chalktalk client and 7) translated into HTML canvas mouse events. 8) Avatar synchronization data are also sent between clients using the Holojam relay.

Some previous work contributed to communication in VR/MR too. ClearBoard allows a pair of users to shift easily between interpersonal space and a shared workspace [Ishii et al. 1993]. The key metaphor of ClearBoard is “talking through and drawing on a big transparent glass

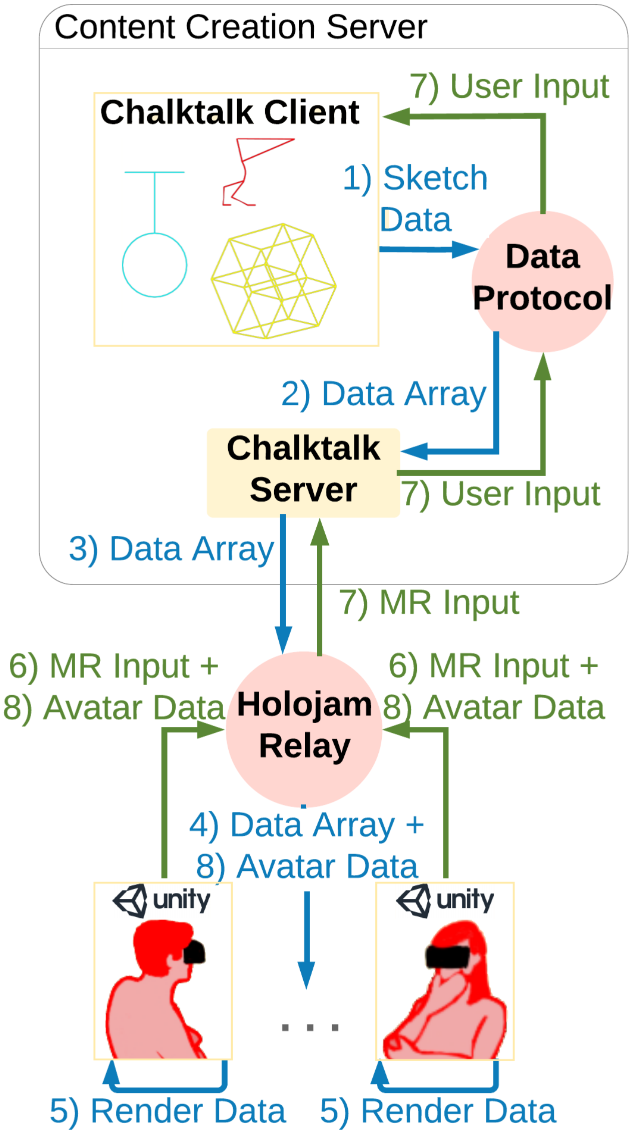


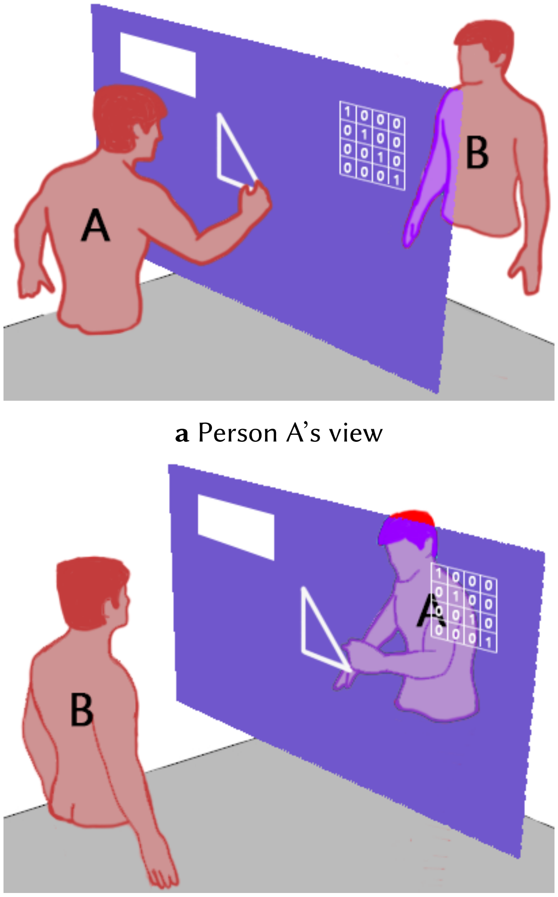
Figure 3.2: Data Pipeline

board." No gaze or eye contact information is lost while working on the content. ShareVR enables communication between an HMD user and a non-HMD user [Gugenheimer et al. 2017]. By using floor projection and mobile displays to visualize the virtual world, the non-HMD user is able to interact with the HMD user and become part of the VR experience. The work discusses how people with different devices communicate with each other. MMSpace allows face-to-face social interactions and telepresence in the context of small group remote conferences [Otsuka 2016]. It uses custom-built mechanical displays on which images of remote participants are projected, and which move in response to users' movements. Pairs of participants can maintain eye contact with each other and remain aware of each other's focus. Instead of designing a configuration to fit one specific communication use case, we provide three different configurations for general-purpose communication in VR/MR.

3.3 Detailed design and implementation

Three Configuration Designs. We implemented three *configurations* for communication in our system: 1) side-by-side, 2) mirrored face-to-face and 3) eyes-free (see figure 3.1). For 1), our implementation has users facing a *content board* (see full definition in table 3.1) from the same side.

For 2), the users are face-to-face in the virtual environment with the content board placed between them, so that each sees the other on the opposite side of the content board, left-right reversed as if reflected in a mirror. The challenge is ensuring all the content is consistent for everyone on each side of the board. Mirror reversal allows, for example, text to be readable and asymmetric objects to appear correct for each participant. ClearBoard [Ishii et al. 1993] implemented "mirror reversal" via video capture and projection techniques to solve a similar problem for 2D displays. Inspired by that, we implemented a 3D immersive mirror reversal for our MR configuration. We place the users physically on the same side of the content board and mirror all other users (from one user's perspective) to the other side. This way, information such as gaze and gesture direction is preserved, so participants can know where each other is looking and pointing (see figure 3.3).



a Person A's view

b Person B's view

Figure 3.3: Mirrored Face-to-face Configuration Implementation. Person A is drawing a triangle with right hand from his perspective, whereas person B sees person A drawing with the left hand. The content of the drawing appears the same to both.

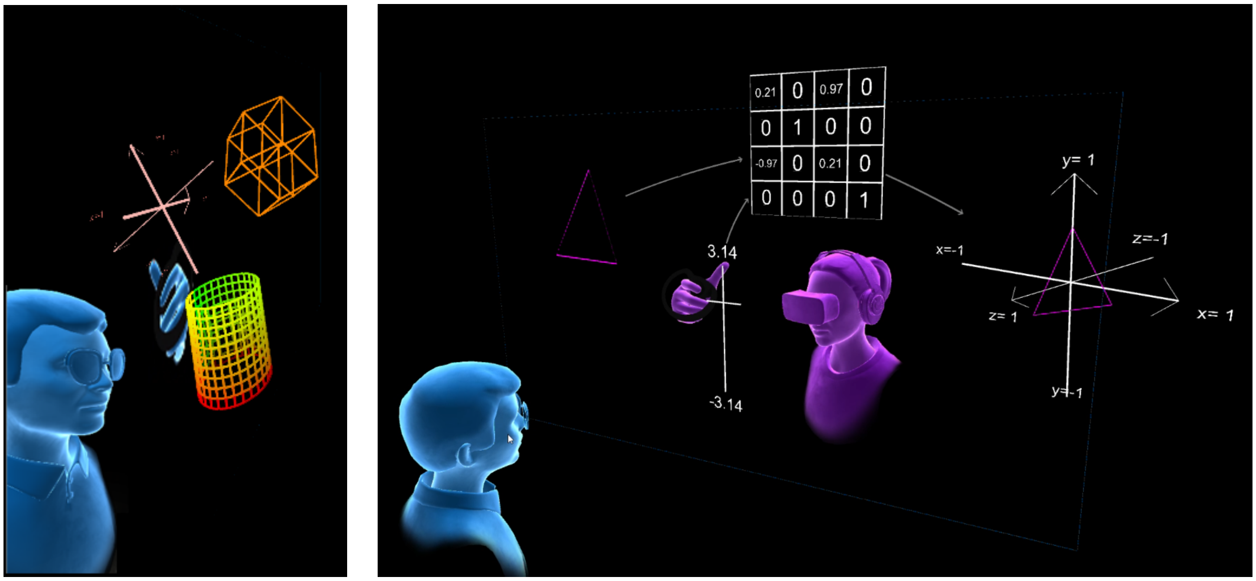
For 3), instead of writing or drawing in mid-air, users can create content in MR while resting their arms atop a horizontal surface in the real world (e.g. table), thereby avoiding the potential fatigue of drawing in mid-air for long periods of time. For this configuration, we now have two boards in the MR world: a horizontal drafting board used for writing and drawing, and a vertical board that displays all information to be seen by all people in a group. All content is duplicated onto the vertical board. A cursor on the vertical board corresponds to the position of the user's hand on the horizontal board. This way, the person using the horizontal board need not look downward, remaining free to look at the vertical content board and other people. This allows users to pay more attention to the environment and each other in a shared experience. The configuration can extend 1) and 2) since we can choose where to place the vertical duplicate board.

The MR System. Our system comprises 1) a content creation server, 2) an internal network framework and 3) VR/MR clients (see figure 3.2 for a detailed description). 1) To enable interactive content during communication, we chose Chalktalk as our content creation server (see figure 3.4). Chalktalk is a web browser-based 3D presentation and communication tool in which the user draws interactive "sketches" for presentation. We designed a generic data serialization protocol to connect and decouple the content creation server and the VR/MR clients, so alternative content could easily be plugged in. 2) To ensure communication between different devices, we used Holojam [Perlin 2016], a shared space network framework designed at our lab. It synchronizes data across devices and supports custom data formats. 3) We implemented the VR/MR clients with Unity to support multiple VR/MR devices.

3.4 Preliminary Experiment

We completed a preliminary user study on our mirrored face-to-face configuration, conducted with 8 participants (F=4) between the ages of 22 and 26 (M=23.71, SD=1.50), recruited via email and word-of-mouth.

Participants were required to have taken a linear algebra class and to have prior experience with VR/MR. First, we introduced Chalktalk to participants who were unfamiliar with it to reduce the novelty effect. Then we gave a presentation on matrix transformations (a concept in the computer graphics curriculum, see figure 3.4, subsection 3.4.1 using our system in the mirrored



0.21	0	0.97	0
0	1	0	0
-0.97	0	0.21	0
0	0	0	1

Figure 3.4: Example Content Board Scenes:
Left: The content board. **Right**: Screenshot from experiment with matrix presentation.

face-to-face configuration:

For each session, we had 1 presenter and 2 study participants in the audience, all physically remote. The presentation was repeated 4 times for 8 participants in all.

We ran the experience with Oculus Rift headsets. Upon completion of all sessions, participants answered a questionnaire that gauged their opinions on one-to-one interaction [Lanir et al. 2008], eye contact [Insa et al. 2016] and focus shift [Buxton 1992] during the experience. Participants then joined a semi-structured exit interview. All factors were evaluated using the 7-point Likert scale. The study was recorded with users' permission.

3.4.1 Matrix Lecture for Experiment

To create a realistic presentation, we invited a computer graphics professor to present a lesson on matrices. He permitted us to use his presentation as the basis for our experiment. We chose the topic of matrix transformations so participants could see and interact with 3D moving content. For content within the 3D immersive environment, we chose visualization of the way a matrix translates and rotates geometry. During the lecture the presenter demonstrated matrix transformations, including translation and rotation, by using 3D interactive visualizations from the content server. Then she showed that matrix operations are non-commutative. The following lists the steps in the presentation:

1. The presenter creates a 4x4 rotation matrix object and links geometry to it.
2. Modifying the matrix values rotates the geometry in 3D space. Students are invited to walk around the MR environment to observe from any angle.
3. The presenter creates a second 4x4 matrix for translation, which she composes with the rotation matrix.
4. To demonstrate that matrices are non-commutative, she shows that by changing the order in which the translation and rotation matrices are applied, the geometry's position and rotation change visibly with the same matrix values.

3.5 Results and Findings

The three main discussion topics during the post-test interviews were "the feeling of one-to-one," "ability to shift focus smoothly" and "eye contact" (see questions in Table 3.2 and results in Figure 3.5).

One-to-one Experience Most participants felt the face-to-face experience created a feeling of one-to-one interaction with the presenter (6/8 agree more than moderately). P1(F): "*It felt like it was a one-on-one lesson even though there was more than one person there. It felt like the person [the presenter] was right in front of me.*" P1(F) contrasted the experience with a college lecture format in which the lecturer stands to the side, which she considered "*more distant.*" Similarly, P4(F) thought it "*really felt like a private lesson*" and that "*It was one-to-one–like we were together in this.*"

Focus Shift Participants responded differently to the question (Q4) concerning how often they shifted focus between content and the presenter (2/8 very rarely, 2/8 moderately rarely, 3/8 slightly frequently and 1/8 moderately frequently). Those who shifted focus least often (P2,M and P4,F) thought the presenter was always in the field of view and felt they did not need to shift focus while looking at the content. Most reported that they could follow both content(Q3) (5/8 strongly agree) and the presenter(Q2) (6/8 more than moderately agree) well. This suggested that the focus shift between content and the presenter was smooth to some extent in the mirrored face-to-face configuration.

Eye Contact Some participants did not always want eye contact. P5(M): "*you don't want the presenter to be always looking at you.*" Others like P4(F) felt it was necessary: "*I look at the professor the whole time. That's the only way I can pay attention.*" Those who preferred less eye contact (6/8 reported eye contact less than slightly rarely) admitted they felt the presenter was looking at them frequently, but chose to look at the content instead of the presenter. Participants who preferred to have eye contact reported having it more often (2/8 more than half the time). This suggests that our mirrored face-to-face configuration supports eye contact well and that the users chose whether to make eye contact or not.

Findings Participants also thought the face-to-face format helped them concentrate during the presentation. Referring to the front-and-center presence of the presenter's avatar, P3(M) said "*I don't think I'd be able to concentrate if I was just listening to somebody. I'd need someone to actually be there.*" P6(F) also found it easier to concentrate: "*It felt one-to-one, so you won't be distracted by other people.*" P3(M), P4(F) and P5(M) noted the format also made the experience feel interactive, "*unlike a video.*"

Table 3.2: Questionnaire

Q1: To what degree do you feel it is a one-on-one lecture?
Q2: Is it easy to follow the presenter?
Q3: Is it easy to follow the content?
Q4: How often did you switch your focus between the presenter and the content?
Q5: How often did you have eye contact with the presenter?

3.6 Conclusions and Future Work

We have presented our ongoing work, a multi-user MR system for communication. We designed three configurations for MR communication and evaluated the mirrored face-to-face configuration with respect to one-to-one interaction, smooth focus shifts and eye contact.

The preliminary study suggests that the face-to-face configuration facilitates a feeling of one-to-one collaboration. Participants responded positively to being given attention and feeling as though they were working directly with the lecturer. Positioning the presenter in-front helped some participants concentrate.

In the near future, we plan to conduct multiple user studies in which we will compare side-by-side, mirrored face-to-face and eyes-free configurations. We will investigate which configurations are best suited to communication experiences involving presentations, group discussions and collaborative tasks. Additional questions remain to be explored. For example, how does the feeling of one-to-one interaction change with the configuration of the users and content?

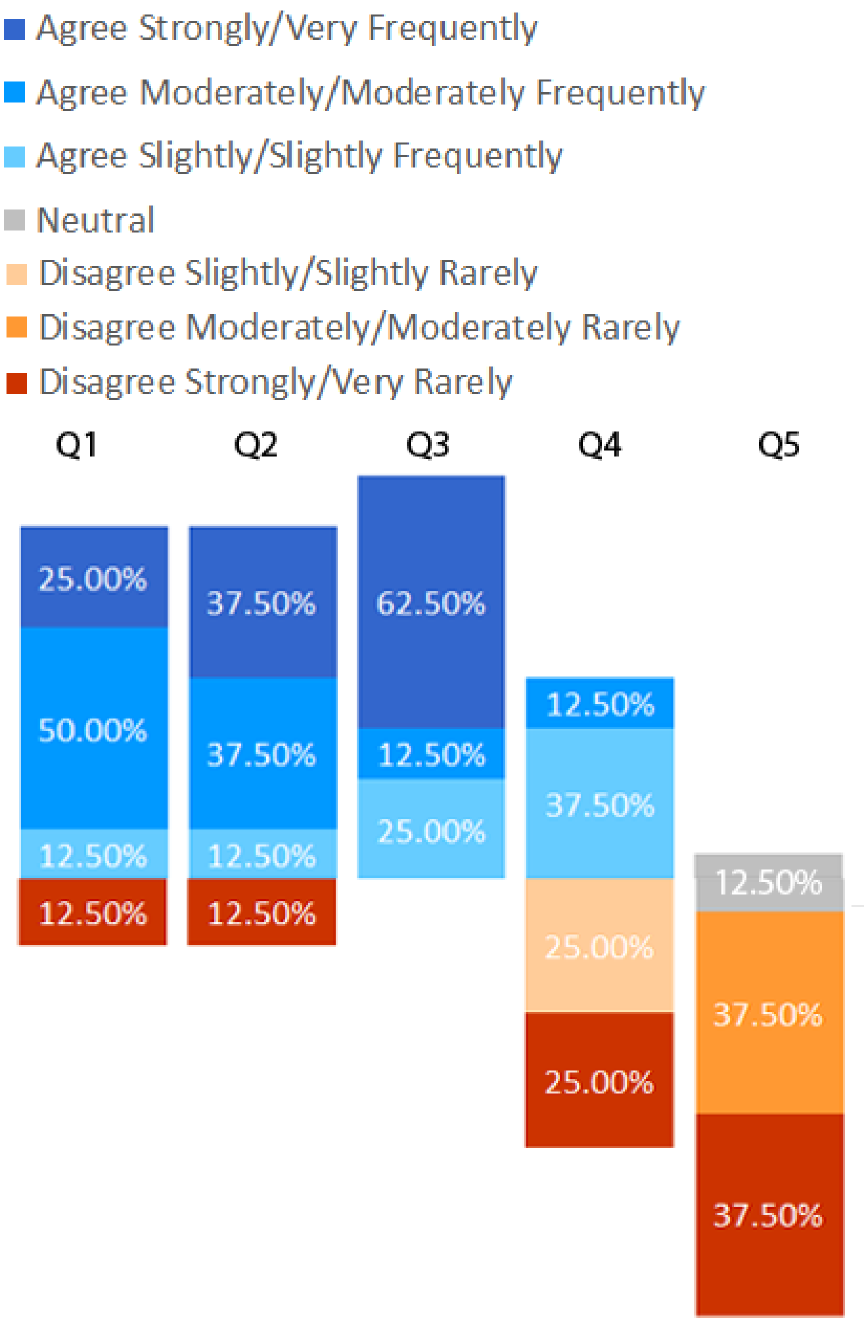


Figure 3.5: Results for the 7-point Likert Scale Questionnaire

4 | Moving towards Verb-based Primitives and Sketching+Speaking

In the initial explorations of sketch-based interaction, we found ways to streamline commands and manipulate content in 3D that used exclusively sketching and direct manipulation. This was a good start. However, Chalktalk had a number of limitations (discussed prior) similar to other interfaces. Namely: 1) all objects, and all sketch recognition glyphs are predefined – which means the user cannot choose the visual representation for their content. Rather, they must learn the visual language to produce a prescribed visual representation for an object; 2) All behaviors are predefined on specific objects – meaning only an object of a given type can perform a specific operation or animation, coded specifically for that object. This also limits the ease with which multiple objects can interact with each other, as the user needs to memorize how each object can connect to other objects to pass data through explicit wires. This was partly remedied with the introduction of a type system in experimental versions of Chalktalk[Nunes and Perlin 2017], but an explicit type system starts requiring programming knowledge. Furthermore, the user needs to memorize : 3) complex behavior starts looking like an explicit programming interface. Creating relationships and information flows between objects requires that the user create node/wire-like diagrams between sketches, and prescribes that these diagrams be ever-present and part of the presentation. This, combined with the need to program using a real programming language in order to create new behaviors, meant that a "power user" would still need to be a programmer.

Overall, an expert user would need to be a programmer to build new Chalktalk functionality. The use case for Chalktalk is at its best when the user knows exactly what they need for a presentation in-advance, and can program their own library of interrelated drawings. But creating new behavior becomes infeasible during live presentation.

So Chalktalk, along with interface of a similar nature, helped inspire a different direction. If, let's say, Chalktalk relies on "noun-based" primitives that have a predefined interface for behavior, what if we increased flexibility and reduced predefined behavior by moving to primitives at the "verb-based" level? *Any* object can do anything, and if that's the case, this should remove the requirement to memorize the behaviors only a certain object can do. By decoupling the visual representation from the behavior, we could enable all sorts of greater flexibility in manipulating digital objects. Additionally, in exploring glyph-based commands instead of commands invoked through radial drags (at fixed locations around a sketch), we realized that we could still do better than having users memorize strokes to do commands. By using speech, we can simply *say* what we want without memorizing a sequence of strokes or button positions. Speech allows us to refer to objects wherever they are. This greater ease of expression was a promising direction.

Overall, we moved towards speaking+sketching through the process of making such observations when playing with visual-programming-esque systems like [Little Big Planet 2 2011] and Chalktalk.

In the next chapter, we give an overview of several more adjacent-works to help contextualize DrawTalking.

5 | Related Work

Our research is based on a confluence of advances in interactive computational linguistics / natural language, dynamic sketching, and multimodal input interfaces using a combination of speech, sketching, and / or gesture or touch control. The following sections cover a selection of work in these areas. (Note: there is overlap between NLP and sketching interfaces.)

5.1 Natural Language-Adjacent Interfaces

Systems such as SHRDLU [Winograd 1972]1.3 and Put That There [Bolt 1980] pioneered the vision of employing natural language to communicate with computers. Thanks to recent advances in speech recognition and natural language understanding, the popularity of this interaction modality has exploded, and has been used in a wide range of domains. For example, VoiceCut [Kim et al. 2019] and PixelTone [Laput et al. 2013] allow users to speak short phrases or sentences to perform desired operations in image editing applications. DataTone [Gao et al. 2015] and Orko [Srinivasan and Stasko 2018] enable users to explore data visualizations using natural language queries. These systems extract corresponding commands and parameters from users' imperative natural language expressions and perform the desired interface operations to find information. [Siddiqui et al. 2021] also provides a multi-modal system for query visualization search, via sketching, natural language, and/or a visual regular expression language. However, these systems operate on specific commands directed to the machine. [Fraser et al. 2020] extends the idea of using speech as a selection or query mechanism to help find and sort information, the former for a data exploration use case, and the latter for searching for tutorial material fluidly with little to no diversion from the task. We similarly use elements of language semantics to form queries, but the query is given as a subset of regular English, and we specialize it to help the user perform commands on a large number of simulated (and changing) objects and interface elements in-parallel. These queries also provide context-sensitive information to commands to trigger a wider range of behavior. So since an English sentence can encode an unbounded number of operations, the result is not only a find/selection operation or single operation, but several, which comprise an entire impromptu script on an unbounded number of objects. In short, we extend the metaphor of a query-select to create running parallelized programs for interactive simulations and animations from spoken language. Along with these graphical programs, the user can interact with the system to play with the behavior of the world as it runs. This means the interface also offloads the task of controlling many elements at once to the machine while still giving the user control at-will.

In contrast to the command-centric natural language expressions, prior works have explored leveraging descriptive natural language expressions for content generation and interface manipulation, which is closer to what this thesis envisions. WordsEye, for example, maps word semantics to the spatial and graphical properties of 3D models to create 3D scenes and animations from text, but the scenes are static and for the most-part, non-interactive [Coyne and Sproat 2001], whereas the thesis is concerned about making interactively controllable elements rather than generating all scene content. CrossPower enables users to navigate, select, and compose linguistic and organizational structures in the text to generate corresponding graphics layouts and animations [Xia 2020]. CrossCast employs heuristic-based algorithms to identify key entities in the transcripts and retrieve corresponding images and maps to augment audio travel podcasts [Xia et al. 2020]. Lyons et al. explored utilizing conversational speech among users as dual-purpose speech input to the system, where words can trigger corresponding interface actions in the background [Lyons et al. 2004].

Other works including Drawmatic AR [Chang 2020] use NLP to create playful interactive story-writing experiences. StoryCoder explores teaching programming by incorporating a machine assistant [Dietz et al. 2021], and a later variant uses a blocks-based language [Dietz et al. 2023]. DrawTalking avoids anthropomorphisation of the machine in favor of letting the user narrate while sketching, with the machine operating as an interface listening in the background for user input. We also do not use an explicit programming language, and rather give the user have programming-like control through narration of a subset of English alone. The experience is different from those involving communication with an assistant. In DrawTalking, the user causes effects in the system using the computational power of the machine. This contrasted with speaking with a third-party agent to delegate or collaborate on work.. We wanted DrawTalking to feel like it was a tool extending the user's individual language.

[Subramonyam et al. 2020] is a digital tool supporting the user in active diagramming: the idea of formulating and drawing while reading to develop connections and understanding of the text through iterative diagramming. (The system uses pen+touch for drawing and manipulation, and natural language understanding to provide suggestions.) Like this work, we explore a connection between language and visuals, but whereas texSketch specifically supports one known effective textual-visual annotation task for learning, we use speech + speech (semantics) and visual/language connections to introduce new interaction techniques, in support of several general creative open-ended activities for sketching, design, game-prototyping, and interactive simulation. We also more broadly introduce a way for users to express their intent to the machine transparently using language semantics and spoken word, within general interactive interfaces connected to natural language, AI, and computational capabilities. So the visual-linguistic connections act as a translation layer between user interactivity and computation.

We share a similar and complementary philosophy for design with [Xia et al. 2023]. We focus heavily on user-first and user-control-based design. The semantic context of the user's input and interaction can be used to drive the application and inform the machine of the user's intent to provide for a more fluid and natural experience. This can help the machine give the user suggestions for how to complete operations. Crosstalk specifically explores the space of semantically-enriched documents and video conferencing for live communication, and proposes an architecture for more user-aware systems and filesharing. We focus more on the interactivity loop within live sketching and creation of simulations where the domain isn't known in advance. Our interaction techniques are a proof-of-concept for how a user-in-the-loop interactive architecture could support live ideation and creation of multiple simulations and interactions through a form of programming via speech semantics. In a sense, our project could represent a methodology for prototyping domain-specific Crosstalk-like interactive systems through improvisational sketching and speech as one might do while narrating or explaining concepts and ideas at a whiteboard.

5.2 Dynamic Sketching Interfaces

HCI researchers have extensively explored sketching interfaces for dynamic and interactive visualizations ever since the first graphical user interface (GUI) SketchPad[Sutherland 1963]1.1 and the forerunner of the visual programming language [Sutherland 1966]1.2. Most of these works have thoroughly explored direct manipulation and sketching techniques for illustrated animation, UI, and visual-oriented programs [Davis et al. 2008; Kazi et al. 2014a,b; Landay 1996; Saquib et al. 2021], programming-by-demonstration [Leiva et al. 2021], and visual programming-meets games [Resnick et al. 2009] and drawing tools [Jacobs et al. 2018] as interaction methods to craft interactive behaviors. "Rapid Design of Articulated Objects" [Lee et al. 2022] contributes an intuitive set of multi-modal interactions to prototype rough sketched models and key-frame animations of 3D articulated objects (joints, hinges, etc.) using 2D multi-touch and pen techniques. The process is not meant to be for real-time use, however. It's for expert artists wishing to make an artifact.

Researchers also explored supporting development of pre-programmed simulations and domain specific behaviors to craft interactive diagrams. In Chalktalk, for example, the system uses sketch recognition to map users' hand drawn sketches into corresponding dynamic, pre-programmed behaviors/visualizations [Perlin et al. 2018b]. More domain-specific tools like MathPad2 [LaViola and Zeleznik 2004], Eddie [Sarracino et al. 2017], PhysInk [Scott and Davis 2013] and SketchStory [Lee et al. 2013] use hand-drawn sketches and direct manipulation interactions to create interactive simulations in physics, math, and data visualization.

5.3 Programming-Like Interfaces

Although plenty of speech and NLP-based systems exist that sometimes incorporate an element of sketching, none quite captures the idea of embedding information directly in our drawings. For NLP interfaces, most assume the machine-as-an-agent model or specific commands specialized to a particular domain. The closest among these works is [Lyons et al. 2004] since it processes speech indirectly. For sketching, you see either a focus on the drawing aspect or high level controls of specific features using speech. The project proposes to take some of the existing ideas, but integrate them with the concept of a natural interface that doesn't necessarily know what content the user wants to create ahead of time (i.e. supports spontaneity and iteration using narrative), and doesn't take the role of a third-party agent. Furthermore, the customizability and flexibility of such a more general interface implies a need for programmability. We have also looked into real-time world simulation systems and programmable environments such as the SmallTalk programming language[Goldberg and Robson 1983], Scratch [Resnick et al. 2009; Maloney et al. 2010], Improv [Perlin and Goldberg 1996], ChalkTalk[Perlin et al. 2018b], and creative world-building games like the Little Big Planet series [Little Big Planet 2008; Little Big Planet 2 2011; Ross et al. 2012] and Dreams [Dreams 2020]. These encourage interactive building of scenes, games and stories. They combine with elements of interactive visual programming and drawing/sculpting with many types of content (2D, 2.5D, 3D, images). But all use explicit interfaces for programming or programming-like functionality (nodes, wires, text).

The final project, DrawTalking, departs from explicit UI for sketching+programming-like capability. Instead, it largely replaces many programming-like functionality with the use of language. Our direction explores the use of verbal, descriptive story narration together with other input modalities (i.e., touch and pen input) to create animated and interactive graphics through sketching.

6 | Formative Steps

Our first step was to arrive at a set of design goals to inform the development of interaction techniques and a prototype interface. To scope our project, we decided that this interface would be for 2D sketching on a tablet akin to digital illustration and whiteboarding, so as not to distance ourselves too much from what people were used to.

To do this, our process followed three tracks: 1) an informal search for examples of live and pre-made sketch-based content to get a general sense of how people speak and sketch; 2) running formative sketching + speaking design exercises to learn how people behaved and what content they made when improvising stories over live sketches; 3) identifying technical requirements and limitations we'd need to account for.

6.1 Sketching + Speech Content Exploration

To get a sense of the relationships between speech, text, and drawings, and to find examples of content that people created, we conducted an informal (non-exhaustive) search for example content online. We looked for examples that involved creating (or showing) rough sketches while speaking, e.g. from popular video channels and educational course recordings. We looked at live sketching (e.g. whiteboarding style) and produced/prepared videos that had a rough-sketching aesthetic. In general, we looked at popular educational videos that featured content with the appearance (actually or scripted) of improvised sketching. (6.1, 6.2 show samples.)

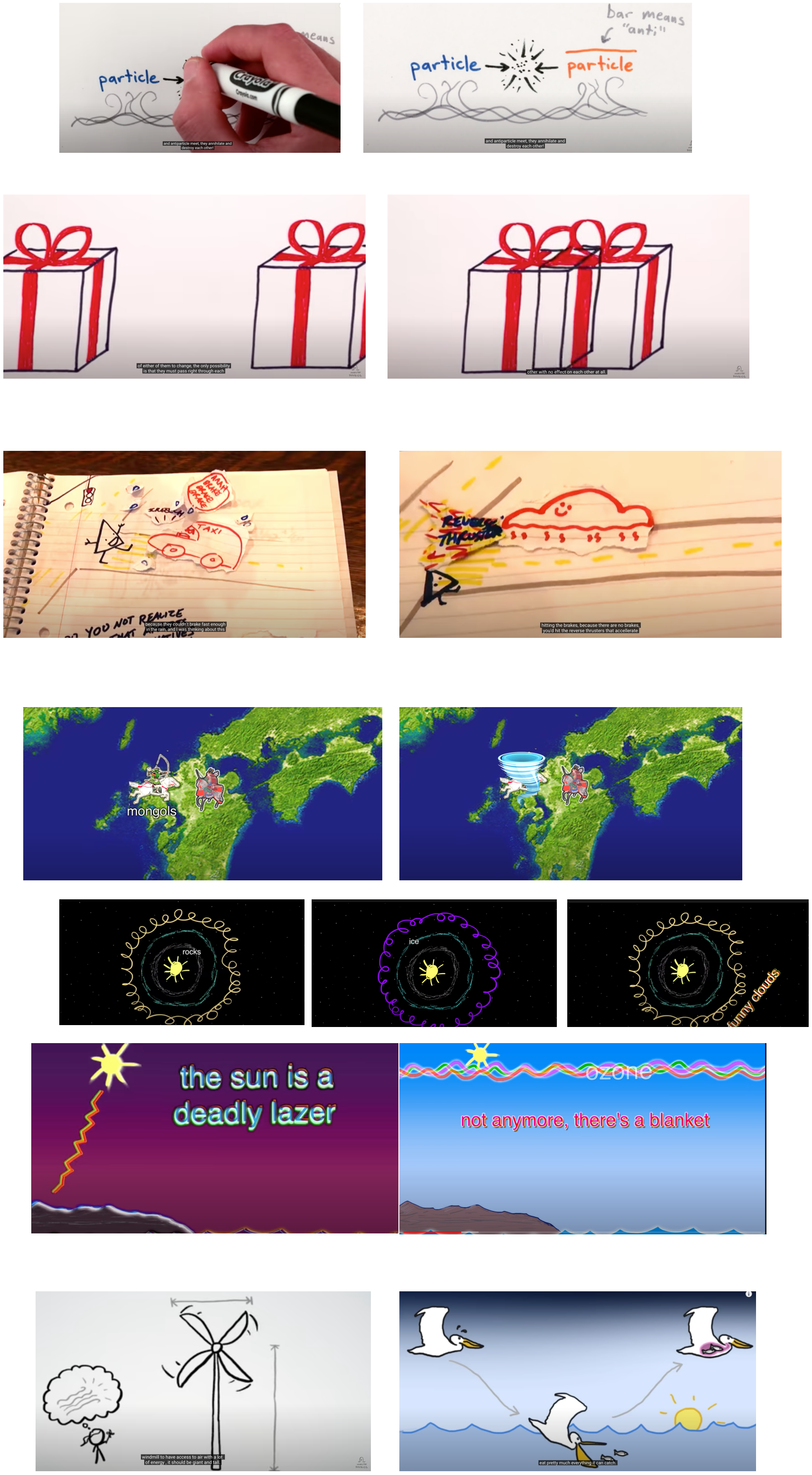


Figure 6.1: A subset of existing prepared sketch-related content we explored for inspiration

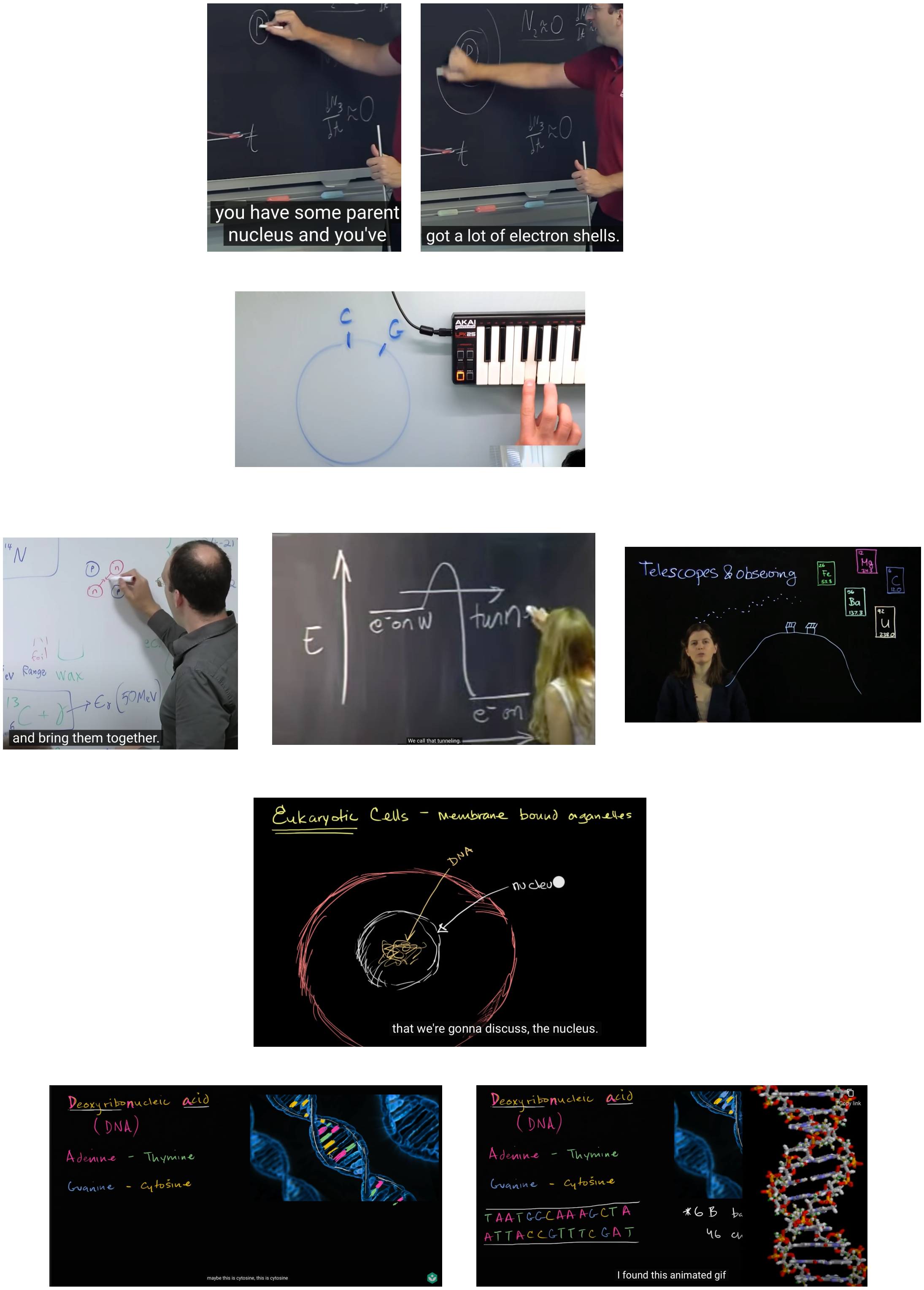

you have some parent nucleus and you've
got a lot of electron shells.
C
G
and bring them together.
Range
wax
E
e⁻ on w
tunn
e⁻ on
We call that tunneling.
Telescopes & observing
Fe
Mg
C
Ba
U
Eukaryotic Cells - membrane bound organelles
DNA
nucleus
that we're gonna discuss, the nucleus.
Deoxyribonucleic acid
(DNA)
Adenine - Thymine
Guanine - Cytosine
maybe this is cytosine, this is cytosine
TAATGCCAAAGCTA
ATTACGGTTTCGAT
I found this animated gif

Figure 6.2: A subset of existing live-performed (bottom) sketch-related content we explored for inspiration

We observed 1) the use of textual annotations placed near sketched-objects they should refer to (for persistent identification or description of behavior). These were semantic labels that defined the names and properties of the sketches. 2) narrators tended to point to objects roughly in the order they mentioned the objects (in cases when the cursor or speaker's hands were visible).

As for the type of content created, almost every sample we observed had unique representations of concrete objects depending on the style and taste of the author, regardless of whether the sketched elements were abstract and cartoon-like or more commonly-used. Sans common diagrammatic elements and symbols, **key point: the author (the one who sketches or narrates) appears to control the representation of the objects.**

As for what narrators described, to reiterate, when referring to objects on-screen, narrators would name objects, refer to existing objects to draw attention to them, and define hierarchies and relationships between objects. In short a **key point: sketched presentations and content implicitly encoded a *hierarchical object model* describing entities and the relationships between them, as well as ways to refer back to them.**

As for what was *not* described, although narration was used to explain or clarify the content on-screen, but not all content (such as additional animations in the case of produced-content or additional details) and not all narration mapped to each other. In other words, there is content that doesn't necessarily have a relationship with speech/narration, and there is content that the speaker doesn't consider necessary to represent, sometimes unpredictably. **Key point: That means that the combination of language and visual content is important, not just one modality or the other, as we can only represent the complete picture with both intertwined.**

6.2 Drawing+Talking Informal Exercises

We conducted informal exercises with 6 participants to observe casual sketching + speaking process with little to no preparation. This would inform design goals. We also wanted to source some inspiration for content, and observe people's use of existing drawing tools to try uncovering pitfalls. This resulted in design goals and some technical requirements.

We invited 6 participants $P1_{init}$-$P6_{init}$, each with some experience with sketching-out ideas and designs for school or work. e.g. concepts, storyboarding, project/presentation sketching for game design.

Each participant was asked to think of 1 personal topic (or 2 based on availability) they would feel comfortable narrating while drawing with freehand sketches. We wanted participants to be casual and conversational for these exercises. Allowing participants to choose personal topics was ideal to achieve this since it did not require them to rehearse or care about adapt another person's work accurately with visuals.

Note: we discovered it was easier to ask for personal stories only through experiment. We asked $P1_{init}$ to try sketching-out an existing content while narrating: first, a fable , and second, the water cycle process. We found the participant was distracted trying to follow the story and match the visuals, or illustrate precisely. This made their process unnatural. After $P1_{init}$, we only asked for personal stories.

We suggested general topics for sketching-out: e.g. personal project, hobby, or recent event. Participants used their choice of tools during the session (eg, MS Paint, Notability, basic tools in Photoshop), but we only allowed basic color selection and transformations to keep participants' use of their tools roughly at the same level of functionality.

6.2.1 Results / Observations

Participants had their own personal styles of rough sketching, but generally followed a simple-stylized approach with playful characters and props.

The scenes drawn were:

P1$_{init}$ a bird-watching trip. (The most complex, shown here: 6.3)

P2$_{init}$ (1) caring for a pet cat (2) a game design project;

P3$_{init}$ (1) old hobbies in school (2) research project;

P4$_{init}$ (1) learning about a hearing issue (2) finding a room-mate

P5$_{init}$ (2) dancing experience, (2) UX project

P6$_{init}$ (1,2) research projects.

The following discuss categories of observations across the content search and the exercises: *Content Style* Sketches were drawn according to the way users think they should look, not how they might look in reality. **So users are in control of the representation of their content and how it should be named.**

Minimal precise temporal synchronization of speech and drawing: In the content search and exercises, people do not tend to synchronize their speech exactly when drawing specific objects (or if they do, the delay between actions is unreliable. In fact, we created an early technical demo in which the researcher tried labeling objects with text spoken during the act of drawing the objects. However, the experience was stressful and inaccurate. Even the researcher felt distracted by worrying about timing their speech, and / or failed to label objects reliably due to the de-synchronization of their drawing and narration, however slight it was. This served as a sanity check not to rely on exact timing to label objects or do operations.); however the order in which people mention sketches verbally usually will correspond to pointing and sketching actions[Oviatt 1999]. **This means we can use this ordering property with respect to speech**

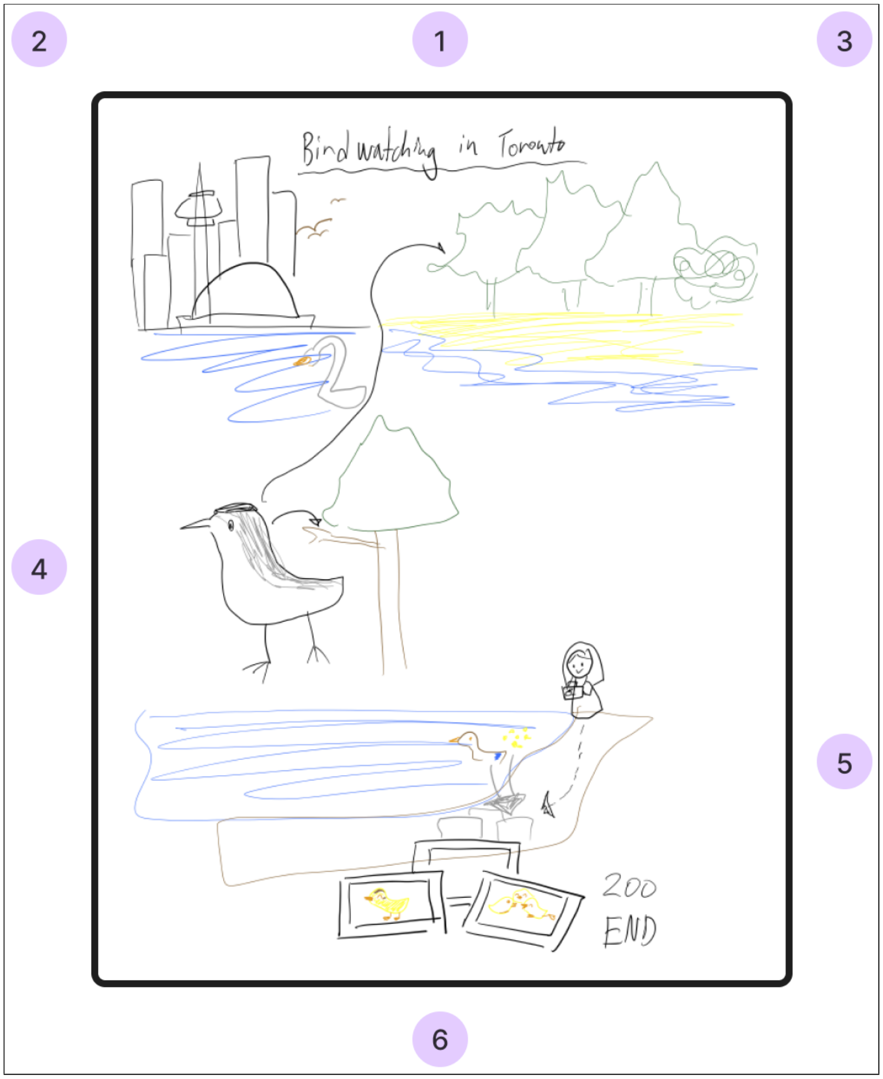


Figure 6.3: Formative Design Session P1 Final Image This result inspired the feel of the project most among them all for its use of visual "scene setting" within a single canvas and several abstract elements. 1) Inline text establishing the name of the "story", 2) drawing of Toronto (city) to set the scene, 3) section of forest to indicate movement in the story to another scene in nature, 4) bird with arrows indicating abstract location information (on a tree, in the forest) — note the spatial relation is not physically accurate, 5) final detailed location by a pond where P1 describes themself watching ducklings, 6) 200 photos took by P1

to let the user map semantics to objects in sequence, but temporal synchronization is unreliable.

Drawn versus non-drawn spoken content: (For both content exploration and the exercises) Participants did not draw all of the elements they described–only selectively. These tended to be objects representing characters or core ideas and objects. **This means we can't assume it's desirable to visualize everything someone says.**

Speed and Flow: (For both content exploration and the exercises) People operate at many different speeds when drawing and talking, and especially more slowly when for more detailed drawings in real-time. People will unpredictably move back and forth between sketches as they're iterated on. **This means that we can't force people to use a specific speed or cadence.**

Modifications to content in the scene: For the exercises, a participant would usually multitask and refine their sketches over time after initially mentioning the entity. They would add details and annotations as they became relevant to the next part of their explanations or story, or as time permitted as they were thinking about what to do next. For example, $\mathrm{P1}_{init}$ added colors to their gray bird and pond sketches as they described their experience at the scene. Because they weren't preoccupied with drawing the next thing, they went back to add more detail to the existing elements they didn't think to add before. Additionally, the participant might make verbal corrections (e.g. $\mathrm{P1}_{init}$ initially called a "pond" a "lake," but self-corrected verbally. This self-corrected was also common for live improvised talks. In other words, it is natural to misspeak sometimes. **This means that 1) speech errors are natural in real-life. We should assume users will make errors in their speech, so we ought to provide ways to correct it. (Errors are not just an artifact of speech recognition [Suhm et al. 2001].). 2) Users' intent might change frequently. They should be able to update their content with new or changed information as they iterate or elaborate their ideas.**

Spatial Relationships Are Abstract: Usually participants used their drawing canvas as an infinite space without regard for precise physical distances, positions, or spatial coherence. For example, $\mathrm{P1}_{init}$'s drawing comprised different sub-sketches representing different locations and story beats flowing from one to another, as if in a picture book. $\mathrm{P2}_{init}$ drew a world map on which they annotated locations they visited over time in their telling of their story, but this map was not-at-all to-scale. Furthermore, most elements in the sketching exercises had no spatial relationships between other objects (e.g. P3$_{init}$ drew independent objects representing different hobbies and P4$_{init}$ drew different scenes without relating them spatially to each other). Sketched objects often represent little scenes and vignettes corresponding to the narrative, rather than a precise spatially-consistent scene. **So spatial relationships between sketches are abstract and based on the semantics or the user's narrative, rather than related to physical locations.**

Use of in-line text and semantic annotations: **Perhaps most critical**, like in the content search, participants would sometimes label their objects with names or use in-line text to describe elements in the screen for them to reference later. Pointing or proximity to sketches while using deictics also acts as a way to refer to objects (as observed in content recorded with cursors or speakers' hands.) In the content exploration, where the purposes of the talks and videos were primarily educational, this was especially the case. This is also consistent with the way sketching in-design uses labels to mark what elements mean [Agrawala et al. 2011]. In short, semantics are associated with sketches in many ways, including through narration, or explicit text labels. **The natural association between spoken language, text, and sketches inspired our our main interaction technique.**

Interface Complexity Reduces Fluidity Participants used a wide variety of tools with ranging complexity (MS paint, Adobe Photoshop). To counteract this, we instructed participants to use only basic pen/pencil, erase, and transform tools. However, most participants were casual users, not experts, so we had a secondary opportunity to observe what workflow issues they might encounter. There was one main pain-point that seemed to impact all participants: forgetting *where* a UI element was to change program state (e.g. switch erase and draw tools), or finding it *too slow to find* a UI element due to it being nested in the interface's hierarchy. This would lead to mistakes or interruptions in flow. **This means our interface should have a simple design that tries to maximize spatial locality of UI elements with little nesting. A "perfect design" is not a main contribution, but we wanted to be careful not to overshadow any learning curve of our interactive techniques with avoidable pitfalls in the UI. So we took**

care to keep most UI buttons and panels in our eventual interface accessible at the top level to reduce the learning curve of finding them later during testing.

6.3 Design Goals

We derived design goals below:

D1 User Choice, User-Specified Intent & Minimal System Assumptions

Users should be in control. For general abstract, freehand sketching, their imagination and preferences determine sketch names, properties, behaviors, and representations. The machine should assume as little as possible and should not infer. Rather, it should defer to the user and receive the user's intent as input to give the user greater capability through the medium.

D2 User-Controlled Association between Sketches and Semantics People associate semantics with sketches via verbal narration or textual labels. Interactions should be designed around letting the user take control of this process.

D3 Flexibility & Mutability Users should be able to try out many different ideas and be able to modify their sketches at any time. This is because users do not always have a clear outcome in mind. Their intent and ideas change as they figure out what they want to say and do in an iterative process. Do not assume the user's narration has a correspondence with all of the visuals.

D4 Fluidity of Operations Operations should be able to happen in almost any order, without requiring precise timing or synchronization of speech and direct manipulation, to support many different speeds, preferences, and ordering of operations.

D5 Error and Ambiguity Tolerance Users might misspeak or change their minds, meaning we should provide multiple ways and opportunities for users to self-correct, refine input, or resolve ambiguity, regardless of whether it's the user's error or the machine's. The system should support multiple ways to correct language input if speech fails. For example, keyboard input, or manipulation of visual UI for alternatives or fallbacks.

D6 Multiple Content Types People use freehand sketches, glyphs, text, and labels. Support all of these representations.

D7 System Transparency Users should easily be able to know what state the system is in and what the system understands as the user's intent.

D8 Complementary Input Channels Language and direct manipulation inputs should have clear and complementary roles in the interface.

D9 Sketches Are Semantic Worlds Sketches are objects with semantic properties within the user's own world. Language encodes structured information defining names, behaviors, and relationships. Using semantic structures goes beyond simple keywords. We want to explore several ways to use the semantic structure creatively.

D9.5 Semantic Structure Encodes Spatially-Independent Selection and Search We observe that language can refer to objects that aren't visible or which do not exist. That makes language a powerful spatially-independent query system for selecting and searching for objects.

D10 Fuzzy Spatial Relationships Provide a way to encode relationships independent of position in the world. Sketches are often positioned on the canvas arbitrarily, irrespective of spatial relationships such as "inside of," sometimes due to constraints in the drawing space or due to choice.

6.4 Additional Content Scoping

In this project, we wanted to focus on the interactions with simple freehand drawings. However, the subject-matter was important. We decided a goal was not to force domain-specific knowledge on the user of our eventual tool. So rather than take a specific topic from our content search (e.g. science), we focused on the general look and feel of the content and the actions illustrated. To complement the existing material we found and the results from exercises, we also looked at storybook illustrations e.g. Aesop fables [Aesop and Winters 2023]. These traditional stories oftentimes involve dialogues or descriptions of anthropomorphic characters in a handful of simple setpieces. Although the quality of storybook illustrations was higher than something that might be drawn live, they were similar to video educational or story content. We also observed that while sketching and speaking, the content that participants drew would oftentimes result in a grouping of objects, vignettes, and scenes similar to what one might find in a storybook, so it made sense to look at these as well. Overall, across our content search we see that people generally stick to simple freehand sketching, and where there is animation, simple non-physically-based movements such as translation, rotation, scaling, and teleportation suffice to get ideas across – even in content focused on physics education. **This means that in sketch-based interaction, users seem to prioritize getting rough ideas across with simple visuals and animations coupled with more descriptive narration, rather than visual fidelity.**

6.5 Additional Inspiration from Games and Visual Programming-Like Systems, and Technical Motivation

The end-goal of this project, in-part, is to support a form of natural world-building, which has connections with our existing world-building and programming interfaces.. How do these relate to the use of semantics and sketching? At a high-level, game engines and visual editors facilitate a form of sketching and blocking-out of spaces, interactions, and mechanics. Although these are not immediately similar to 2D freehand sketching, many of these tools are built with non-programmers in mind – artists and general audiences – specifically for trying ideas-out very quickly without complete knowledge of programming. It's useful to understand the designs of these systems because their goals are complementary. A common interface paradigm is the node diagram flow in the spirit of [Sutherland 1966]. One can find variations of this style of visual programming in production tools as well as programming interfaces for wide audiences, including children. For example, have adopted these diagrammatic visual scripting systems to enable rapid iteration of effects, animations, and designs by non-programmers within a running game simulation environment. Game-like programming interfaces like [Resnick et al. 2009] are particularly relevant to us because – arguably – they're focused not only on creating an artifact or product, but also on the user's progress towards learning programming. They're environments for learning through play. Scratch, in this case, mixed a canvas designed for 2D sketches and images with a an explicit programming interface. It took the ideas behind the programming language SmallTalk [Goldberg and Robson 1983] and made them visual to support sketching of simulations via programming. Similarly, user-generated content games like [Little Big Planet 2008; Little Big Planet 2 2011] followed the same Smalltalk-like tradition (down even to the use of message-passing-based object-oriented design) to turn programming into the exercise of making a simulation, story, or game. (However, LBP used circuit diagrams and wires rather than block programming). A couple of observations: 1) Programming in these interfaces (when you dive into how they work) involve creating textual labels to name objects and their attributes, or specify behavior upon events such as collisions between objects of specific types. The blocks or node-based scripting represent functions that define the behavior of these objects, and within these interfaces, the user can also label scripts to invoke them later. Wires between nodes Scratch and LBP (and Unreal, and Unity) naturally require the user to assign some form of semantics to the entities in the game. 2) Most of these interfaces give the user a running simulation to play with in a flexible way, either in safe editing modes or during play modes in which all objects are alive during a physics or animation simulation.

We begin to see that these interfaces already have useful constructs for thinking about what interactions between language and simulations might look like. We see the use of language semantics, as well as hints of sketching. However, since we want a natural interface that could extend our language, a major problem here is that the interfaces are explicit. The programming-specific elements (as opposed to the content being manipulated) take space, are time-consuming to use (due to the navigation of menus), and still amount to programming with explicit textual (or diagrammatic) structure. Using an explicit structure means the user needs to *find* elements in the UI or their program within space, which takes time.

If we want to extend our language using simulations, then, we need to start thinking about interfaces that could feasibly be transparent and work anywhere regardless of location. i.e. it shouldn't take space away from the content, and the user should always be able to access elements of the interface or program from anywhere. This contrasted with textual programs and node diagrams or deeply-nested UI (which have visual structure).

One way to solve the problem would be to compile a more natural input *into* these diagrammatic representations behind-the-scenes to achieve similar effects i.e. defining entities, entity relationships, rules, behaviors, and running programs.

In this way, it also makes sense for us to use speech (or text) as a natural input, as language can encode complex logical, relational, and hierarchical structure through simple sentences, and it doesn't require a visual representation at all to store that encoding. Language, however, can be visualized as text, meaning speech input is spatially-independent, yet flexible enough to be transformed into other more structured representations. In other words, we're providing an alternative framing for our motivation: this is a UI-input and visualization simplification problem.

Hypothesis (from a programming language perspective) People can in theory learn to program sketched-based simulations using just speech and direct manipulation with-

out perceiving a reduction in capability. Rather, they should perceive a simplification of the user experience which could still be comparable to existing interfaces in terms of capability. Because these forms of inputs correspond to how we narrate and explain things in real life, we hope for this to be an example of a natural interface that people believe will be useful in the future.

6.6 Iterative Design & Development Methodology

Following our formative steps, the design, development, and testing of DrawTalking underwent many iterative steps. We took a continuous feedback approach in which we frequently asked for feedback on aesthetics, mechanics, and use cases of the system during informal conversations. Among the audiences we contacted for feedback were HCI practitioners, computer science professors, art, design, and computer science students (undergraduate- and graduate-level), software engineers, and friends and acquaintances. We also conducted pilot test-plays of our system. As necessary for clarification in the following sections, we report any feedback that led to significant changes or insights. (We choose this format since feedback occurred over a long period of time and couldn't reasonably be traced to one specific version of our system.)

7 DrawTalking - Design and Interaction

7.1 Workflow

In DrawTalking, the user sketches freehand on a digital canvas, and simultaneously speaks to narrate. Speaking can serve the dual-purpose of explaining concepts or telling stories to an audience, and as a way for the user to tell the system the semantics of their drawings – what their names, properties, and behaviors are. By narrating, the user controls the semantics that they want embedded into sketches as labels. Because the user does this labeling of sketched objects with names (nouns) and properties (adjectives), the system knows what the objects are. Then, the system can use the full semantic structure of the user's speech (looking at nouns, verbs, adjectives, adverbs, etc.) to understand what the user's intent is, which objects to find and select, and which commands or animations to run on sketches or the interface. The end visual result is a simulation of the user's narration in the form of animations, relationships, and automated events between sketches. This comes from the semantic structure and logic in the user's spoken[1] input. The user fully-controls what part of their speech is used for control of interactive elements, and what is pure narration. As a result, a recording of an expert using DrawTalk will look like a whiteboard presentation using narration and drawing, but animation and simulation appearing

[1] or if preferred, typed

to come from the user's words. The user, meanwhile, creates and manages complex scenes via a fluid workflow, and with speech commands that tell the machine how to automate the behavior of many elements at once. By enabling the user to express intent in simple terms though language and primitives, our approach achieves several of the criteria for expressive interactive systems (i.e. the interface achieves expressive reach, power through recombination [Olsen 2007]). It has the potential to empower wider audiences to create, play, and express themselves through simple combinations of rough-sketched animation and simulation with speech.

The DrawTalking environment couples language and drawing input as a way of facilitating a fluid and open-ended experience. It tries to combine everyday sketching and narration, while transparently augmenting our creative and thinking process with many of the capabilities of live programming.

For this project, we focused on achieving playful exploratory scenarios that the user sketches in real-time while using multimodal controls. The user can speak full sentences using simple predefined verbs that can be composed, sequenced, timed, repeated, parameterized with adjectives on nouns or adverbs, or infinitely looped as part of ongoing interactive simulations. (see Appendix for supported vocabulary). Alternatively, the user can create rules that are invoked when events occur in the future e.g. collisions between types of objects. The workflow is flexible. One possible workflow might be to (1: define) draw and label all objects (using either deixis ("this" / "that" / "these" / "those") or direct linking between words and objects), and then describe rules to describe their behavior, (2: control) speak a story to direct the sketches to perform animations and behaviors, and also influence, movement, and transformations of sketches deliberately with direct manipulation, (3: observe) watch events unfold as sketches interact with each other due to the prior labels and rules. The user can do any of these steps in any order to change labels, rules, and content, and refine their interactive scene. So although it's possible to stage all content ahead of interacting with the scene, the user can choose to experiment with different outcomes without knowing what exactly they want in advance. Insofar as the user has labeled sketches with the desired semantics, they can modify anything at any time. Whatever semantics and rules are currently set influence the simulation at-present.

Another possible workflow might be to draw everything with no labels. Using speech commands or the transcript, the user can easily label them in-post.

This way, we facilitate a creative process over the course of using DrawTalking. The output is dynamic and responsive to the user's particular exploration of the vocabulary and the combinations of objects and properties in the scene. This results in an environment in which immediate visual feedback and planned-out interactions are both possible within the same interface – which means the user can be exploratory as well as decisive, and the user has a choice in affecting output directly or letting the machine control some interactions automatically (6.3 D1).

Scenarios can be wide-ranging. We will show a variety of examples to showcase our approach for creating interactive visual and computational mechanisms. All use rough sketching and language controls to create interactive animations, simulations, and game prototypes from scratch using the primitives in our interface.

7.2 Interface and General Controls

DrawTalking (7.1, 7.2) consists of multiple interactive regions and associated buttons. The components together enable the user's workflow. 1) a *drawing canvas*, 2) a pen toolbar, 3) a *transcript view* (a scrolling live speech/text transcript), and 4) an independent scrollable picture-in-picture (PIP) for a *semantic diagram view* – for displaying the machine's understanding of a command alongside selected objects –for transparency and editability. The user first confirms (stages) a command based on the text in the transcript (using the speech confirm button), which displays the semantic diagram view, and then the user confirms again. A key point is that the user can choose to cancel the operation if the diagram is somehow wrong, or change the command to select other objects. The same goes for the text transcript, which lets the user sub-select text to

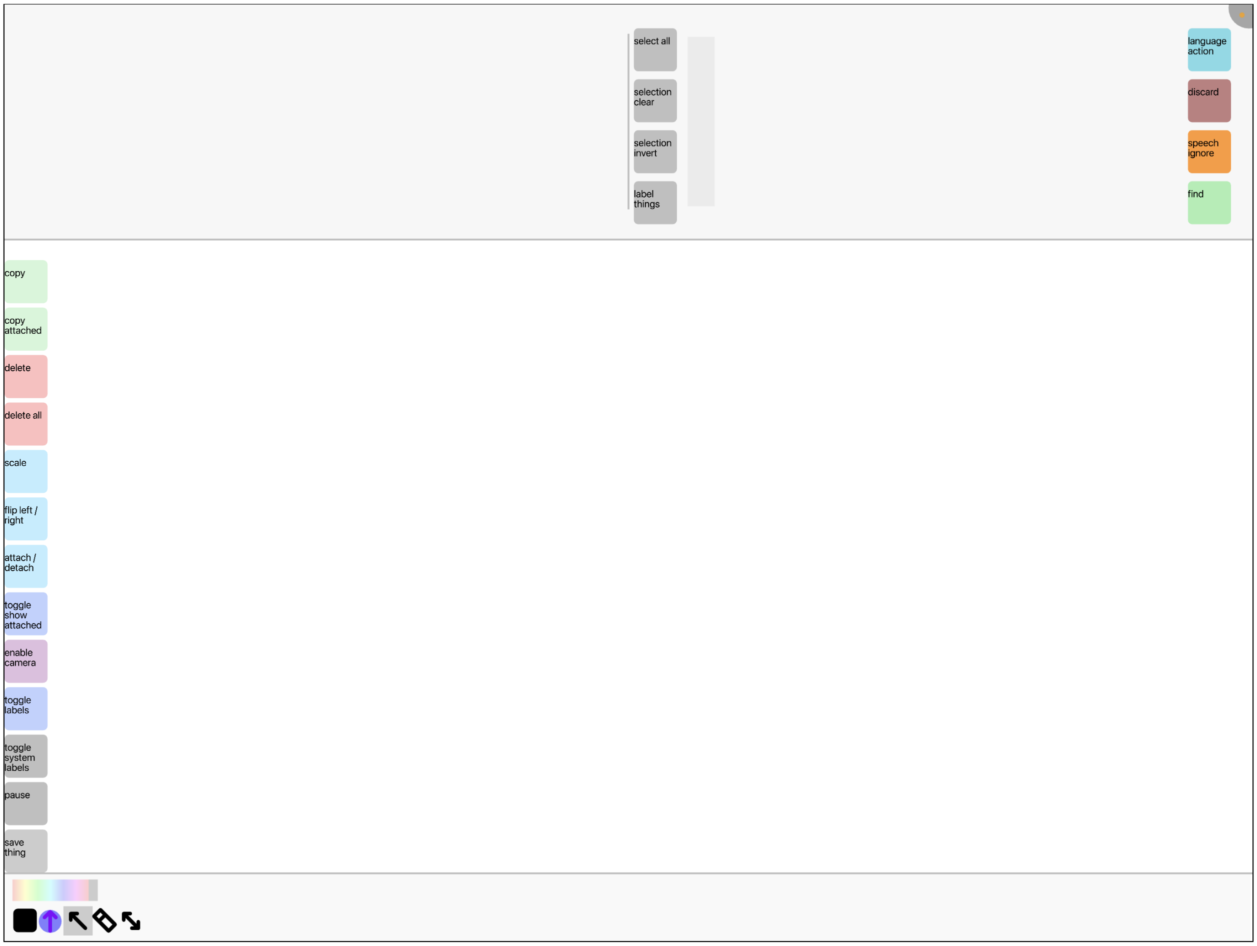
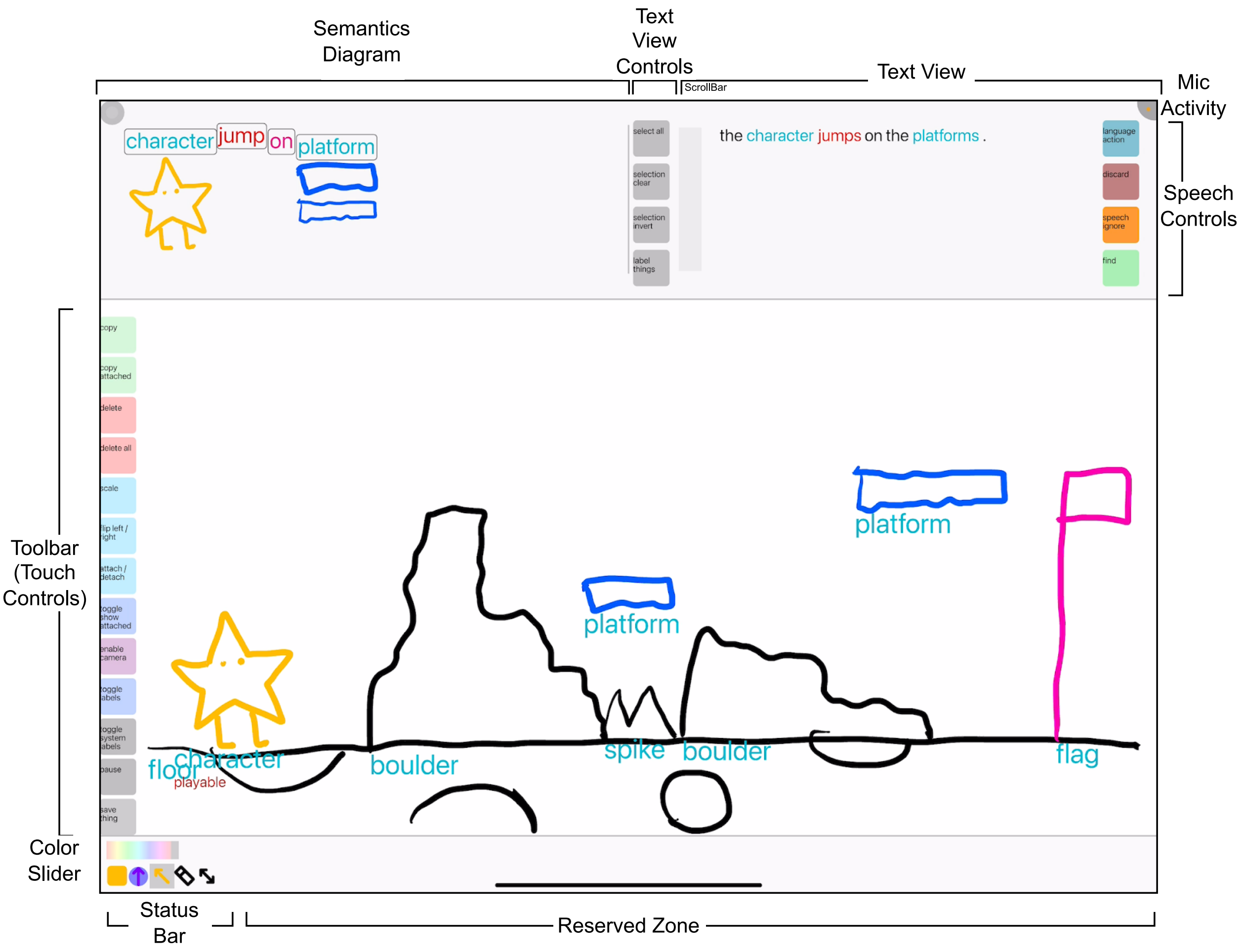


Figure 7.1: The initial state of the canvas

create different commands, or discard text if the user does not wish to use their speech so far.

The overarching workflow might be to draw and name objects while speaking, and to confirm commands using the *transcript view* and diagram interface elements in-between. The combination of multitouch and stylus inputs helps achieve fluid controls.

We choose to give the user the full control over a result before committing. The behaviors in DrawTalking can have immediate lasting side-effects that would be disruptive if performed by the machine automatically, or if the user wanted to be more careful and deliberate. However, the user can choose to ignore these steps and run their command. The diagram will telegraph if there is a critical error (not being able to find objects) and in these cases, the user simply must try



the character jumps on the platforms .

Figure 7.2: Interface with an example of a staged speech command. "The character jumps on the platforms" selects the sketch labeled "character" and all sketches labeled "platform" in the scene. The user will confirm this command with the language action button (top-right), which will cause the character to jump between all of the platforms.

again. In-practice, the less ambiguous the user is in their speech – i.e. more precisely referring to specific objects, the more they can be confident enough to execute operations immediately.

Please see the essential overview of each region in figure 7.2.

The user coordinates freehand drawing and narration via pen and multitouch.

All operations are performed via a sequence of 1 or more unimodal or multimodal pen and / or touch inputs . Holding an object while tapping another with the pen generally means to use the object for context, and the pen selection will invoke a unique operation. In general, touch manipulates, pen draws, and pen+touch combinations do context-sensitive operations.

7.3 Sketching

2D sketching in DrawTalking occurs on a simple interactive 2D canvas with standard multi-touch and stylus controls for direct manipulation (e.g. [Hinckley et al. 2010]). The canvas is infinite in size[2] and pannable, rotatable, and zoomable via multi touch(See figure 7.1).

7.3.1 Sketch Object Creation and Manipulation

A freehand sketch is represented a collection of one or more strokes[3], which together represent one semantic entity, or *thing*, that occupies part of the *world* with its own collision boundary (6.3 D9). Treating sketches as entities lets the user select, transform (move, rotate, scale) or attach sketches to other sketches as part of a scene hierarchy – via direct manipulation. It also lets the system treat individual sketches as uniquely identifiable objects with their own semantic properties – i.e. names, attributes, collision event triggers and responses.

7.3.1.1 Other sketch types

It is useful to support inline text and numbers within our drawings (6.3 **D6**), so the interface supports these these. They behave the same as freehand sketches, but lack strokes. Instead, text sketches are simply modifiable text, and numbers represent values on which we can perform math operations through speech commands, direct manipulation, and interactions and events between objects. Number objects can be treated as counter variables and checked for inequalities.

[2] within computational limits

[3] geometry, not rasterized in our case

7.3.2 Sketch Selection and Editing

When a user first draws on the canvas with no sketches selected, they create a new sketch (with its own unique identifier) with a bounding collider. To select a sketch and make it *active*, the user simply taps on it.[4] This also shows a highlighted bounding collider around the active sketch. Subsequent pen operations add or remove strokes local to the active sketch, depending on the currently-selected tool. (If no sketches are selected, the eraser deletes the full sketch including all of its strokes if tapped by the pen.) [5] To edit another sketch, the user must de-select the current sketch by tapping the background (or reserved zone) and then tap a new sketch. The user can select multiple sketches at-once (one-per finger) to move, rotate, and scale them individually. Only the active sketch (with a dark/thicker bounding box) is editable with the pen, however. Lastly, if a user wants to create several single-stroke objects in-succession: a shortcut is to hold the currently-active object with a finger, and to draw somewhere else on the canvas. This will create a new sketch and make it active even if an existing sketch has not been de-selected.

7.3.3 Semantic Labels

Each sketch is also a collection of semantic labels, embedded by the user's speech interactively. Labeling of sketches is key to specifying the correspondence between the visuals and entities referenced in the user's speech. These labels provide information about what the user intends objects to be, without requiring the objects to have a specific visual representation. The system uses them identify *what* objects to select and *how* to perform operations or drive behavior based on the objects' labels. The labels function as dynamic type information [6], meaning the user can define rules and behavior on any sketch at all that shares specific labels. The user can add or

[4] if multiple sketches overlap at the selection, the sketch with the smallest size will always be chosen (because we assume the user could have selected a larger sketch outside the smaller area.

[5] We found through pilot-testing that if we used the common method of adding strokes based on what the pen overlapped, the user would accidentally create new objects they intended to be part of the same semantic thing. The final method keeps the same object selected until explicitly de-selected, which we found was less error-prone.

[6] as in, programming language type system types

remove labels at any time via speech or direct manipulation: 1 or more nouns (defining names for the sketch) and 1 or more adjectives (with optional adverbs) defining attributes and attribute values. This mapping only needs to be done once per sketch, and thereafter, the system can operate on these sketches when processing the user's narration.

By default, all labels are displayed by a sketch (as often is the case on a whiteboard), but this can be disabled.

For flexibility and fluidity (6.3 D3, D4)), at any time the user has 2 main user-initiated[7] ways to perform this mapping between sketches and semantic labels (as discussed in D2): either through 1) verbal deixis combined with touching or pointing to 1 or more sketches or 2) linking sketches with textual labels 7.3. (For further flexibility, it's possible to assign labels to sketches up-front to reduce multi-tasking later, or defer until later if the user refines the sketch.) Note that there is no constraint on which labels can be assigned to which sketches (D3). Labels appear underneath their associated sketches. Labels' visibility is toggleable, and to remove a label, the user long-presses it with touch. Removing a label essentially changes the identity of the sketch and gives the user flexibility to try things-out (**D3**) or quickly correct themselves or the system (**D5**).

7.3.4 Labeling (Mapping) via Spoken Deixis

People often speak about or introduce objects by simultaneously pointing/touching them and referring to them by name or deictics (**D2**). We emulate this process to enable quick labeling as the user speaks. The user only needs to select objects in the order they're referenced in a sentence to label them. For example, we can tell the story about a scene in natural language, and can transparently and immediately label multiple objects by tapping or drawing them as we refer to to them in-order. (Only the order matters, not the timing, as per D4)

{ "**This** ***is*** a **beach** where **this** ***boy*** often goes, and **that** ***is*** a ***lighthouse*** that he visited a couple of times with his dog. **These** ***are*** **sailboats**." **}**

[7] Later sections introduce ways to label sketches programmatically based on user-created rules.

Using **"this"** or **"that,"** we indicate that only one object is expected. **"These"** or **"those"** indicate that we want to label multiple objects with the name name. It's possible to narrate naturally because the system looks at the language structure. This is the quickest and most fluid approach to labeling objects in DrawTalking because it interleaves with the desired narration and storytelling. In all pilots, we observed people used this mode of labeling the most.

7.3.4.1 via the Verbs "transform," "become"

. Additionally, selecting an object with the pen is also valid, e.g. if the user wants to name an object upon the first stroke in a sketch.

7.3.5 Labeling (Mapping) via Direct Link with Text

Sometimes, it might be more intuitive to narrate and refer to objects without using deictics. For example, during a talk or story when it's obvious from context what an object is, deictics might be awkward or tiring. some who piloted DrawTalking noted that Commonly when iterating on a design or scene, the user might not know immediately what they intend a sketch to be or whether to visualize it. For user-control, flexibility, and fluidity (**D1-D4**) as well as error-tolerance **D5**, the user can directly link a sketch with words in the *transcript view* using a (○, ⟶) operation to add labels. The interface displays an arc between the sketch and word for visual feedback (**D7**). Re-linking removes the label and the associated arc. With this labeling method, the user can speak naturally or even without regard for sentence structure, context. The labeling can be performed anytime, as only the individual words matter. This aids in a design process with more unknowns up-front, or when the user is *not* engaging in full narration or storytelling and wishes only for the convenience of speaking individual words. In contrast with deixis-labeling, this method is more flexible when speech is less desirable or the design process is early-on. It's also more error-tolerant because it has no assumptions about the order in which users refer to sketches (**D1, D5**) It requires more multitasking between the pen and multitouch along with speech to achieve fluidity, as in a performance or professional talk. For example, in the example sentence from the previous section , some words such as "dog" were not referenced via deictics, but the user might have labeled the dog with a direct link. Perhaps later, the user will choose to add a sun sketch or scenery like mountains and cliffs

Ideally, the user will use the full-range of deixis and direct linking as-necessary. It is up to the user and their preferred workflow (**D4**).

7.3.5.1 via Text Object Sketches

A $(\circ, \longrightarrow)$ operation between the canvas and a word (or number) in the *transcript view* creates a text / number sketch. Between the canvas and the *transcript view* itself creates a text object from the selected text.

Note: A direct link via $(\circ, \longrightarrow)$ between a freehand sketch and a text sketch will label the freehand sketch with the text contents.

7.3.6 Unlabeling

To unlabel via deixis, the user simply says that an object is *not* a *label.* To unlabel via direct manipulation, either long-press the label with a finger, or $(\circ, \longrightarrow)$ between the object and the word in the *transcript view.*

7.3.7 Status Bar

7.3.7.1 Color

1) the current color - tap with the pen while holding a sketch to change to this color 2) scrub with touch or pen to change the current color

7.3.7.2 Compass

Points towards "up" to help the user track the orientation of the canvas

7.3.7.3 Pen Modes

1) draw 2) erase - delete a sketch by tapping on it while it's not selected, remove strokes by first selecting a sketch and then drawing over the strokes 3) arrows - draw starting at a source sketch and overlap other sketches to create arrows directed from the source to all intersected sketches. The user can tap on the buttons to select the tool and double tap their pen (in our implementation) to switch between 1) and 2). The currently-active icon displays beneath the user's pen as a cursor (also when hovering the pen) to help the user remember the current state.

7.3.7.4 Safe Area

This blank zone lets the user pan, rotate, and zoom the canvas at all times, which is useful if the entire screen is currently covered.

7.3.8 Toolbar

This is a flat menu of quick actions.

- **copy** (◦, ◦) copy the root sketch object along with all labels
- **copy attached** (◦, ◦) copy the entire hierarchy of objects rooted at the touched sketch
- **delete** (◦, ◦) delete the touched sketch
- **delete all** (◦, ◦) delete all user-created freehand, text, and number sketches
- **scale** ◦ - multitouch scaling on a sketch is disabled by default. Tap to toggle.

- **flip left / right** $(\circ, \circ)$ Swap the left/right orientation of a sketch. We assume a sketch's forward direction from the side view is to the right.
- **attach / detach** $(\circ, \circ)$ child-parent-attach / detach the first touched sketch to the second touched sketch
- **toggle show attached** $\circ$ - display line segments between attached objects
- **enable camera / disable camera** $\circ$ - toggle a background front-facing camera on the iPad (purely experimental)
- **toggle labels** $\circ$ - hide / show labels on all sketches
- **toggle system labels** $\circ$ - show / hide additional system-level metadata on all sketches such as system ID (debugging)
- **pause** $\circ$ - pause / unpause the simulation in the world. All objects stop moving, making it easier to edit and move objects yourself
- **save thing** $(\circ, \circ)$ - save the touched object as an example of whatever its label is to spawn it later

7.4 Language Semantics

The key interactions in DrawTalking resolve around the interplay between speech and drawing, and how the user provides their intent to the machine. The main interaction technique is inspired by the core takeaways from 6 and 6.3. Namely: when people narrate and sketch, they provide semantics and user intent to the machine as language and direct manipulation input. Once the machine knows user intent, it knows the content of the sketched scene and how to perform operations on it when the user narrates. In the context of natural speaking and sketching, the user can tap into the computational capability of the interface with little additional effort than the norm, and the machine likewise has the benefit of knowing for sure what objects represent according to user intent, without the need for automated inference. In our implementation of DrawTalking, that computational capability is procedural animation, simulation, computer graphics, and interface control.

7.4.1 Semantics Interpretation

DrawTalking receives a structured subset of English syntax and builtin primitives for several verbs, adjectives, and adverbs, taken from our content search, games, and visual programming systems. In general, these run the gamut between transformations, create/destroy/appear/mutation, procedural movements, tweens, drawing-program edit operations, and UI camera behavior, among others.

So the machine has information to manipulate each sketch, we have the user narrate to assign semantic labels to their sketches: possibly multiple *nouns* reflecting names and influencing selection, *adjectives* reflecting attributes and influencing selection and behavior, and *adverbs* reflecting *values* that modify the effect of each adjective. i.e. these labels perform more or less the function they do in language, but in the context of a procedural world simulation. Once this mapping is built, all else follows (and the user does not need to add or change labels unless desired.)

Then, when the user narrates further with reference to these labels, the machine can select the objects assigned these labels and *parameterize verbs* in the user's sentences using adjectives and adverbs. For example, an adjective might represent movement speed (e.g. "fast") and an adverb "very" might amplify *how* fast. We can treat adjectives as variables, and adverbs as modifiers.

Verbs are equivalent to a continuously running *function*. Each verb primitive individually and dynamically interprets noun, adjective, and adverb labels on objects differently according to the builtin behavior of the verb. i.e. receiving or ignoring relevant *parameters*. To support *interactivity*, adjectives and adverbs are evaluated continuously. For example, "fast" can be removed and replaced with "slow" upon an event or a new speech command, which will alter the speed of a sketch currently moving under the influence of a verb.

We also support conjunctions, sequences, loops, conditions (rules), elapsed time, number ("a", "the"), numerical counts, hierarchical relationships, and distinctions between individual objects and types of objects.

[8] We took creative liberties and focused on playful interactions that might fit into rough sketching.

Given user speech input, the interface looks at the whole semantic structure to express a command for immediate execution or rule defining future behavior based on an event. In general, a command will identify the sketches (and corresponding attributes and modifiers) and what their semantic roles should be for each verb in the sentence. The results of commands are driven by our small domain-specific language of parts-of-speech and how we felt they ought to act and be interpreted in our proof-of-concept interface. Any implementation of ideas in DrawTalking might opt for different visuals and simulations.

. (In 13 we provide a manual of the features we introduced to participants in a study.)

7.4.2 Nouns, Pronouns, Deixis

Nouns identify which sketches (or types of sketch) the interface should select, as if in a *query* (6.3 D9.5).

Quick selection via language is extremely important for achieving the fluidity and control in the interface. It enables the user to reference and manipulate an unbounded number of objects independent of their location, simply based on their properties. We believe this is a natural use case for speech, as it mirrors how we communicate via language: we talk about and imagine objects that aren't there. The language control, compared with traditional UI and direct input

[8] To the programming languages audience, this is equivalent to assigning symbols to variables, where symbols are semantic labels, and variables are sketches. The system is in charge of invoking interpreting users' spoken sentences as – essentially – functions, whose arguments are the objects whose labels are referenced in the sentence. We can think of the concept of this interface as an interactive-time compiler from spoken language (in this case, English) to any domain-specific-language (DSL) (in this case, the one we've implemented for simulation and animation.)

techniques, shines here.

Using the article "The" or "a"/"an" differentiates between selection of specific targets and randomly selected targets.

"all" lets you refer to every single object of a certain label.

We can select all objects with a label at once based on whether the noun is singular or plural, or if a certain number of objects is specified. e.g. { "The 5 dogs" } . Conjunctions let you specify objects of multiple labels. { "The 2 dogs and the 4 frogs..." } . If not enough objects exist, we assume it's valid just to select as many as possible.)

The special reserved noun "I" lets you do operations without a specific object.

The noun "thing" can refer to any object.

We can also use pronouns to refer-back to previously-mentioned objects (via co-reference resolution), which is more natural to us than always naming objects explicitly.

Lastly, if we want to select objects ourselves and quickly start a command, we can simply touch an object and say "This <verbs>." The system picks-up that the sketch being pointed to should be guaranteed to be selected.

Overall, there several multimodal ways to select objects (by the user and machine) to fit many flows.

7.4.3 Adjectives and Adverbs

7.4.3.1 Impact on Selection

We can use adjectives and adverbs to disambiguate between objects with the same label. For example, a "boy" is different from a "happy boy" or a "very happy boy."

7.4.3.2 Impact on Simulation

One or more adjectives describe the attributes of an individual object. For simplicity, we chose to treat these as discrete values representing magnitudes for a particular builtin **characteristic** (e.g. magnitude, speed, size, distance, height, accuracy). Adjectives are labels which map to a given characteristic. 1.0 is the default value. Anything above normally increases the effect. Anything below decreases it. We selected visually-appealing values by hand, but any method for defining these values could work. For example, "fast" is 1.5.

Verbs in the end are given the flexibility control how to use and or ignore adjectives. Animations and simulations caused by verbs can continuously evaluate these parameters on the selected objects. For example, if the user removes the label "fast," a verb can change its animation while it's running – to reflect a slowdown of the object if the adjective corresponded to speed, or lower the height if the verb was something like "jump" and the adjective was "excited." The user does not need to redo an entire command just to see this result. This is important as well when commands start to script objects to change state automatically.

Adverbs modify adjective values. A *very* fast character will have an amplified speed value. Humorously so, we opted for chained adverbs to modify multiplicatively: A *very very very very* ... fast character will move absurdly fast. As for adjectives with negative correlation e.g. "slow," adverbs with greater magnitude will multiply inversely to create a smaller value. See 7.4 for an example.

Note: we handpicked adjectives and adverbs based on the animations and simulations we wanted, but these are just labels mapped to values. At run-time, we can easily add adjectives and adverbs with their own values to create wider ranges of parameters, so long as verbs understand a given **characteristic**.

7.4.3.3 Labeling by Deixis, continued

We can also use deixis to label objects with adjectives and adverbs. e.g. **{** This is a very very happy dog. **}** will attach the labels "dog," "happy, very."

7.4.3.4 Special Adjectives

In pilots, users suggested providing speech-controls for directly manipulating system-level features. To hook into system functionality, some adjectives have special behavior. For example, any sketch is placed in either world space or screen space. It is desirable to create a UI / HUD element, or to freeze an object independently of the camera for convenience, by fixing it to screen space. Labeling an object as "static" achieves this, and unlabeling moves the object back to world space. Other words like "invisible" will automatically hide an object.

Additionally, for certain potentially-repetitive actions like "take" (as in taking several objects to a location), to disambiguate between 1) repeatedly affecting random object of a certain label 2) repeatedly affecting a multiple unique object of a certain labels until a task is done to all objects. We could find several context-based solutions to this, but our choice was simply to use the adjective "new" to describe repeating a task to multiple objects in sequence, as in collecting all trash and putting it in a waste-basket.

7.4.4 Verbs

Verbs correspond to actions and instructions that tell the system what to do with arguments passed-in (sketches, sketch types). The simplest example might be, "The object jumps." Note the speech is in narrative third-person form. The verbs can start long-running behaviors, short-term-animations, or control interface functionality directly. Verbs are essentially functions that can do anything, and have access to the entire interface and whatever the implementation of DrawTalking exposes. We build-in a library of simple primitives sourced from sketch-based videos, animation, game prototyping languages. Any implementation of DrawTalking could provide its own primitives via coding or scripting.

Verbs are responsible for driving the behavior of objects in the scene. They do the final assignment of the selected nouns (objects or types of objects) to different semantic roles: source (the object doing the action), direct-object (the object directly receiving the action), object (an object participating in an action, as in the object of preposition in "X jumps *on* Y), among others.

The verbs we chose to implement generally fall into a few categories:

7.4.4.1 Animation

Fixed-time or continuous movements, transforms, tweens, effects. e.g. move, follow, rotate, jump, flee

7.4.4.2 State Changes

System-level or instantaneous functionality for creation, destruction, hiding/showing of objects, transformation of objects into other objects, stopping, among others.

7.4.4.3 Events

Verbs like "collides with" or "overlap" are reserved events that occur when objects intersect with each other. "Press" is a reserved event by default that occurs when the user touches an object.

7.4.4.4 Numerical

(In)equalities (e.g. "equal," "exceed"), arithmetic to affect and check values of number sketches.

7.4.4.5 Adjective and Adverb Verb Modifiers

Adjectives and adverbs spoken as part of a sentence affect verbs the same way labels on objects would, but are fixed into the command permanently (and aren't evaluated continuously). Adjectives are "interpreted" in the context of the verb. For example, "move slowly right" will cause an object to move rightwards at a slow pace because a direction and speed category adjective are provided as arguments. But if the user said "destroy right," that would not make sense in the context of that verb's implementation, so "right" is ignored. **i.e. modifiers are context-sensitive, and verbs are flexible to logical errors in users' speech**

7.4.4.6 Preposition Verb Modifiers

A verb combined with a preposition (e.g. "jump on") represents a potentially unique variant of the verb, and just as in English, might express a very different action. The verb is responsible for choosing the correct variant if it exists. Otherwise, it assumes a default. It's up to the user to make "sense." For a verb definition like "jump," the preposition matters: an object jumps "on" (top) "below" (bottom) or "beside" (nearest side) of an object.

7.4.4.7 Selection by Preposition via Parent-Child Hierarchy

Objects can be attached hierarchically in a child/parent relationship to represent spatial relationships, or as in 6.2, arrows as well. We use these hierarchies and drawn arrows to represent relationships such as "belonging to" or "spatial proximity." In sketching, objects might be placed anywhere on the canvas for convenience or by choice, irrespective of their relationships (6.3). We do not want to force the constraint that objects inside or belonging to others literally overlap each other as in a physical space. When the system searches for objects described as belonging to or spatially located with respect to another, it will search for objects in these scene hierarchies to find the right one. For example: (7.5)

If we attach a blade sketch to a windmill, and a blade to a counter (as in a kitchen), then narrates **{** "The blade on the windmill rotates clockwise" **}** , we will traverse windmills only to find a blade, correctly ignoring the blade on the table.

7.4.5 Verb Conjunctions

Use "and" as a conjunction between verbs to execute operations in parallel. This is a way to synchronize animations and events to end at the same time.

7.4.5.1 Past and Present-Tense Are The Same

We treat past and present tense verbs the same for simplicity, as in the context of narrative, explanation, or storytelling describing something now. This has the benefit of reducing errors caused by inaccuracies in the speech recognition, as present and past tense recognized words are equivalent.

7.4.5.2 Verb Stopping Conditions

To stop an action, we tell an object to "stop" or perform a new action that overrides the old. For example, if an object is moving to a target, telling the object to move to a new target overrides the previous instance of the action. Some builtin verbs are compatible with others e.g. an object can rotate AND move, or an object can move up and move right, but not to two different targets. Note that this is also part of our DSL according to what "felt right," but this in no way precludes other possible implementations or configurations.

7.4.6 Sequences

Use "then" or "and then" to specify events that should happen in series.

7.4.7 Timers

We can specify the duration of a verb by adding "for X seconds," where X is a real number. e.g. { "The circle moves up for 11.18 seconds and then the circle jumps." }

7.4.8 Loops

We can repeat an action forever { "(forever|endlessly|over and over) the dog jumps } , repeat a certain number of times { "10 times the dog jumps excitedly" } , or for a duration { "For 5.3 seconds ..." } .

For nested semantic structures, there can be ambiguity in fully natural language in deciding how to interpreted looped and nested sequences. We simplify this like so: saying loop keywords before inner structure ends up looping around all nested language, whereas after creates an unnested sequence of loops:

keywords before:

{ "5 times A and then 2 times B" } **means** { Repeat the following 5 times: A happens once and then B happens 2 times. }

keywords after:

{ "A 5 times and then B 2 times" } **means** { A happens once and then B happens 2 times. }

We stop loops by using the verb "stop" on the object performing a looped action or by deleting the targets of actions. For example, an object will stop following a just-deleted object.

Some verbs like "follow" are implicitly loops in their implementation because they run continuously.

7.4.9 Rules (Conditionals, Events)

A rule represents an event-based trigger that continuously checks the current actions and attributes of particular types and objects, and then runs a command when a condition is satisfied.

This enables the user to specify a command for automatic execution in the future *when*, *as*, or *after* an event has completed. The rules are lazily evaluated as well, meaning we can refer to labels we haven't created or used yet. This gives the user the freedom to define the scene incrementally based on the logic they think they will need, without knowing in advance which objects they specifically need to draw. We can apply rules to individual objects, or, more generally, to objects with specific labels or types. This is incredibly useful because enables us to build worlds that start to look like simulated games, which behave independently, or in tandem with user interaction.

A common case is to specify behavior upon collision between multiple types of objects. Then, the rule runs the command on the relevant objects whenever the condition is satisfied.

e.g.

In the case of rules below, nouns represent *types*, which are specified with plurals without articles.

{ When *dogs* collide with *treats dogs* destroy *treats* **}**

or

{ *Dogs* destroy *treats* **when** *dogs* collide with *treats* **}**

or for a more natural input:

{ When *dogs* collide with *treats* ***they*** destroy *them* **}** .

The first two introduce no ambiguity, whereas the last form could be misunderstood in more complex cases and results may vary depending on the NLP capabilities.

Here, *any* objects with the labels "dog" and "treat" are candidates for this rule.

If we wanted this rule applied to a specific dog only:

{ When *the dog* collides with *treats dogs* destroy *treats* **}**

If the specific dog is deleted, the rule is automatically deleted. If the specific dog's label is removed, *the rule still works*, as the selection was persistent on the unique object. This allows for some creative transformations of the same object. (See 8.1.7 later.)

To aid in testing and iteration, the user can view, temporarily disable/enable, or delete rules via a context-sensitive interface (see 7.6).

7.4.10 Definition by Composition and Chaining

Using the rule syntax, we can define new verbs or customized variants of existing by composing existing verbs in the library. For example, we might find the word "eat" more fitting than "destroy" as in the above7.4.9.

"Eat" is not defined, however, we can define it ourselves by:

{ When things eat treats things destroy things **}** , which would be a simple alias of "destroy" for all objects, or we could customize it for dogs specifically as in **{** When dogs eat treats dogs destroy treats **}**

But it would make more sense to have the dog move to the target before "eating" it:

{ When dogs eat treats dogs move to treats and then dogs destroy treats **}**

It is also useful to chain triggers together separately for flexibility. e.g. we can make the above livelier by adding a jump upon the end of the above event: **{** After dogs eat treats dogs jump twice **}**

Lastly, to reduce ambiguity, we let the user specify when all objects of a certain type should be affected by a rule: saying that *all* objects of a type should do something as a result of a rule being a trigger will command every object with a certain type label.

7.4.11 As an English-Scripting Translation Target : Scoping and Decoupling of Language Input from Tech

We clarify why a closed-set of capability and English syntax still helps us work towards a "natural" interface, as opposed to using complementary technologies e.g. generative models. Firstly, we wish to keep focused on the the workflow aspect and not on precision or assuming a particular interpretation of the user's intent. Introducing boundless capability might be more "magical," but it's more unpredictable. Starting with a subset of English and known behavior keeps things more deterministic, scoped, and introspectable, and yet in practice, an open sandbox like DrawTalking still affords a wide range of expression and interactive capability. Furthermore, for the range of use cases dealing with thinking, improvisation, play, and open-ended design, it is important to keep the user in-control and the system interactive. A large language model, in contrast, is excellent at generating an open-set of material to achieve a certain result. It can give you results based on the expectations reflected by a mass amount of user data and content. If the goal is to arrive at a particular result, this is excellent. However, what we want is to support a workflow which puts the user in the loop of the entire creative process. Large language generate models create results independently from the user beyond a prompt, so there is no obvious entry point for direct user input and manipulation. This prevents us from exploring how to enable interactive functionality, as a generated result is challenging to change or introspect precisely. A closed subset of English and behavior is advantageous in that the user can learn and fully understand the primitives and know what they do for sure, which means they can master the system, and we as researchers can better observe them and learn what expectations are and are not met when using the tool. An open-set of generative behavior, in contrast, would be very challenging to study and reproduce, and the user would not be able to predict the output as well as a closed-set language. Overall, generative models would be a good choice if the focus were not on enabling a full user-in-the-loop process. In spite of the smaller set of functionality, by composing the primitives together interactively, it's possible for the user to build-up more complex interactive results with fairly foreseeable and learnable visual results.

The system is not limited to structured input, however, in the sense that the user is able to edit their spoken input using the *transcript view* to create the system-understood commands. As most input (we've found) is short, this editing via multi-touch and pen is fast in-practice, or unnecessary if the user sticks to the known syntax. Compared with large models, which as of writing run in seconds, our scoped English subset and rule-based interface runs at interactive game-speed. So it allows us to simulate the feel of speed better than alternatives, which is crucial for introducing our approach from the workflow perspective – how it feels as to how accurate it looks. This makes sense for the rough sketching domain.

Overall, external models are promising complements to a rule-based system like ours, due to the rich range of content, they can output, but they are less understandable. We choose our approach deliberately for our use case and research.

As technology for more sophisticated language processing evolves and speeds-up to game-levels of interactive time, external systems (e.g. generative models) could easily fit-in to create a best-of-both worlds with user control mixed with larger content spaces.

Since our language input and primitives confirm to a structured spec, we can provide a translation target for generate systems to compile to. External models could generate structured English systems like ours understand, directly from fully-natural English – or provide more flexible language inputs, behaviors, and visuals, independent of our interface. Our interface design allows for such translation layers in the future, while enabling us to study and play with the feel of the interface now7.7.

7.5 Invoking Commands, Language Views

Speech input runs *continuously* by default to let the user focus on their narration. (A top right indicator on the screen shows when recognition is on, and the ignore speech button 7.5.1.1 toggles it on/off).

Our design puts the user's intent first, so the system does not run a new command from the user's input without their consent via the language-action confirmation button 7.5.1.1. This serves the dual purpose of letting the user decide how to divide their sentences.

The user can immediately invoke commands from their ongoing speech by double-tapping this button. Completing a command will automatically clear the *transcript view* to prepare the next command.

However, DrawTalking has an optional direct interface to the user's language semantics. This provides a visual of the user's input history, editing capability, and transparency into the system's understanding of the input prior to a command. The *transcript view* is a live transcript for visualizing and quick editing of speech input, and the semantics diagram is for visualizing and editing/correcting the system's understanding of input at the semantic roles level. The interface elements are made always-accessible within reach to keep interaction quick even when using these interface components.

The first tap takes the currently-selected text in the *transcript view* as the input (all text by default). This generates the semantics diagram representing the command and the mapping between semantic roles and sketches. At this point the user can remap sketches if they were mapped incorrectly by the system. The second tap confirms that the result is correct and executes the command. At any point in these steps, the user can cancel and clear the *transcript view* with the 7.5.1.1 button.

7.5.1 Balancing Fluidity and Precision Trade-offs

This interface design lets the user make their own trade-offs between confidence in precision and accuracy, and interface fluidity and speed, independent of the quality of the underlying language processing implementation. We decoupled from the language processing technology and provided these options for recovery and ambiguity resolution so the user could make these choices.

For example, if the machine could read the user's intent perfectly, the user could skip these steps entirely. Realistically, the back-end speech recognizer and language processing implementation will influence precision, and the better technology becomes, the better our design will perform. In general, the more specific and unambiguous the user is when phrasing input and referring to objects, the more confident they can be that the system will do the right thing.

Still, we believe that even in ideal scenarios, it is useful to provide full transparency and control in case the user changes their mind (as discussed in 6.3).

7.5.1.1 Language Buttons (Input)

- **language action** stages language input for confirmation ; confirms
- **discard** discards language input ; cancels a staged command
- **ignore speech** toggles speech recognition off / on
- **find** finds sketches and rules. See

7.5.1.2 Language Buttons (Editing)

- **select all** selects all text in the *transcript view*
- **selection clear** deselects all text in the *transcript view*
- **selection invert** selects all non-selected text and deselects all previously-selected text in the *transcript view*
- **label things** auto-labels the semantic diagram's sketch selections. See 7.5.3.2

7.5.2 Transcript View

The *transcript view* is a scrollable live transcript of the user's speech input. By default, all words are considered part of the next command. If the user misspeaks or the underlying recognizer misinterprets, they should be able to deselect words and try again without repeating themselves. This way, the user can speak naturally and still be able to input a valid command. The user can label sketches with (∘, ⟶) between a sketch and multiple nouns or adjectives in the transcript. Directly linking a sketch to a noun in the transcript is also the most precise way to give a particular sketch a role within a new command because it forces the system to use the user's mapping. This is most often unnecessary and undesirable for speed since it's normally faster to let the system do the mapping, but it provides additional optional control to the user over the result.

Our solution is to permit quick editing at the word level. Each word in the transcript is a toggle/on button individually. Drag with touch at a start word over to an end word, and the entire range of text will toggle off if the start word was on, and off if the start word was on 7.8

Additional buttons for quick selection and deselection have been added to speed-up text editing if desired 7.5.1.2.

It might be common to remove large sections of words if the user is addressing an audience and does not want to treat all of their speech as a command. The user can select only the pieces they want from the transcript fairly quickly and treat those as part of a command.

A possible use case that would benefit greatly from quick text editing is one-many presentation. The user could speak fully naturally to an audience, but quickly remove words to aid the machine in processing a command.

As per our note on translation layers, external libraries could potentially preprocess natural speech to output system-understood commands with higher probability. But the *transcript view* could let the user edit if need-be, and sometimes the user might be able to edit with small changes faster than external tools. It is generally good to have options.

7.5.2.1 Search Query

In a large unbounded canvas, it's common to want to find moving objects off-screen, want to move quickly between different parts of the scene, or interact with objects without moving to them directly. To address these, we can use language as a search query. Tapping the find button (7.5.1.1) will display in a grid a preview of all sketches corresponding to the nouns (+ adjectives) in the text transcript. These previews are proxies to the original sketches: tapping with touch will move to and center the camera on the referenced sketch, wherever it is. Using erase on a preview will also delete the referenced sketch. Lastly, we can also copy by dragging a preview with the pen onto the canvas. Tapping with the eraser or dragging with the pen will apply the operation to all previews in the query pane: meaning, erase all sketches, or copy all of them. (This is a way to erase or copy multiplicatively en-masse. See 7.9).

We chose, as a test of how to show context-sensitive views without taking more space, the find button will display previews of rules if you select the word, "rule." The eraser will delete a rule, and the pen will toggle off and on a rule. This is a way for the user to test if a rule is doing the expected thing, or even to toggle rules interactively off or on as part of a user-in-the-loop performance of an animated scene.

7.5.2.2 Typing Input

As per 6.3 D8, it's useful to provide alternative inputs to correct for others' weaknesses, so we enable input by keyboard. If speech recognition fails[9], typing to edit individual words is a quick way to correct without repeating yourself. It might also be desirable if the user sometimes does not want to narrate some content.

7.5.2.3 Overall

The *transcript view* overall represents balancing user precision and speed. It enables fine-grained control of speech-input to keep the user always in-control. At minimum, the user can ignore this interface and use exclusively deixis for labeling and structured English for the system to understand. In between, they might use this interface purely to correct or edit their input occasionally, or to let them speak fully naturally to an audience and remove parts of their speech to help the machine. At most, the user could choose to use this part of the interface to have complete control over commands and not care about narration or speed, but rather accuracy.

[9] We use a builtin speech recognizer and do not contribute out own.

7.5.3 Semantics Diagram

The user should be made-aware of the system's understanding of their input (7) to remain in-control. To fulfill this need, we display a diagram representing system understanding of a command before the user confirms to execute the command. The diagram is spatially-independent from the main canvas, meaning it's a picture-in-picture that can be panned and zoomed separately to reveal other parts of the diagram, while letting the user move the main canvas separately. When the user stages a command with the language action button, the system displays a simplified semantic role diagram representing its understanding of the input. The flat tree diagram represents the relevant words with semantic roles linked to the sketches that should fill those roles. We designed the diagram to look almost like the original input to make it readable at a glance.

Words mapped to individual sketches contain a vertical list of proxy sketches referring to the sketches in the scene. This is designed to give a preview of the system's mappings without introducing visual clutter in the main canvas.[10]

Types or references of sketches are marked with an asterisk and do not contain a proxy list, as these do not refer to individual objects.

If the user thinks the diagram is correct, they can confirm the command to move-on. For most sentences and inputs, we find it is reasonable to skip the diagram step and just confirm twice, as speech recognizer errors are handled by the *transcript view*, and the user can often avoid any errors just by being more specific in their language (which is what we do in real-life to avoid ambiguities).

The diagram offers error feedback and recovery (6.3D5). If there are "fatal" errors, such as

[10] Language visualization is an ongoing research area that might be a good future direction. In previous versions of the interface, arrows always connected the words in the diagram to sketches directly in the scene rather than proxies. We do not claim that the final is necessarily better, as some might argue that joint attention is better with all information in one spot. However, we found that this was a necessary trade-off, as too much visual information was confusing and made it challenging to distinguish between UI and content in the canvas.

missing sketch/word mappings, the diagram will highlight the respective word in red, meaning the user cannot proceed with the command. This will commonly happen if the sentence refers to something that does not exist with the correct label. If the command references verbs that don't exist in the system (and the user is not defining a rule), the system will suggest existing verbs with similar meanings by displaying a list of candidate verbs for the user to select. For example, the word "hop" is not in the library, but the system will recommend "jump."

7.5.3.1 Remapping

The user can choose to (∘, →) between proxy sketches and a diagram's word to remove them from the diagram, and between sketches in the scene to a word to map the sketches themselves. In this step, *any* sketch will be valid, irrespective of its label. So the user can do the final sketch mapping themselves with this diagram interface if they desire.

7.5.3.2 Auto-labeling

The diagram, in this way, lets the user choose to defer the task of labeling to the last moment. However, it's useful to do the labeling. To support this version of the workflow, the user can tap the 7.5.1.2 "label things" button to label all mapped sketches with the words they're connected to in the diagram, all at-once.

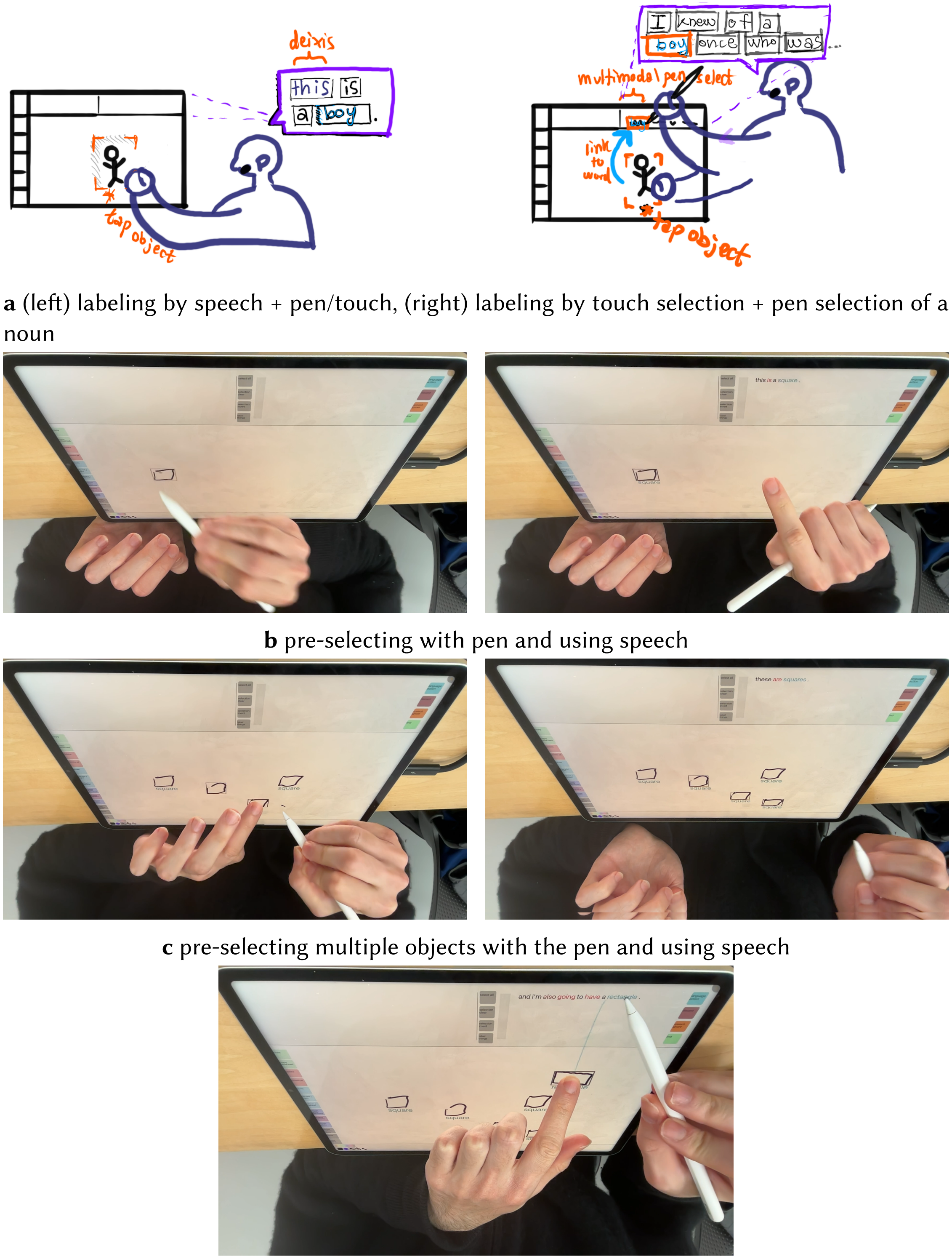
7.5.3.3 Search or Edit by Proxy

The user can also treat the diagram as a spatially-independent search and editor. By tapping a proxy sketch, an arrow is displayed linking the proxy to the referenced sketch, wherever it is in the scene (on or off-screen). Using the eraser on a proxy will delete the original sketch.

In more mature versions of the interface, on might edit a sketch via proxy with the pen.

7.5.4 Overall

We designed the DrawTalking interface to support workflow flexibility, system transparency, and error recovery at every stage. The *transcript view* and semantics diagram expose semantics and editing capability to the user if they want it and satisfy our goals towards putting user-control first. The user can trade-off fluidity and precision. Our design allows the user to choose between these extremes. On one end, it provides additional fine-tuning capability to the users who want deliberate control and direct manipulation of their language as a form of data input. On the other end, a user can skip past these levels of control and see results as fast and as precisely as the underlying implementation can handle. This way, our approach is independent of the language processing technology that powers it, but as technology improves, the speed and precision experienced while using our approach should as well.



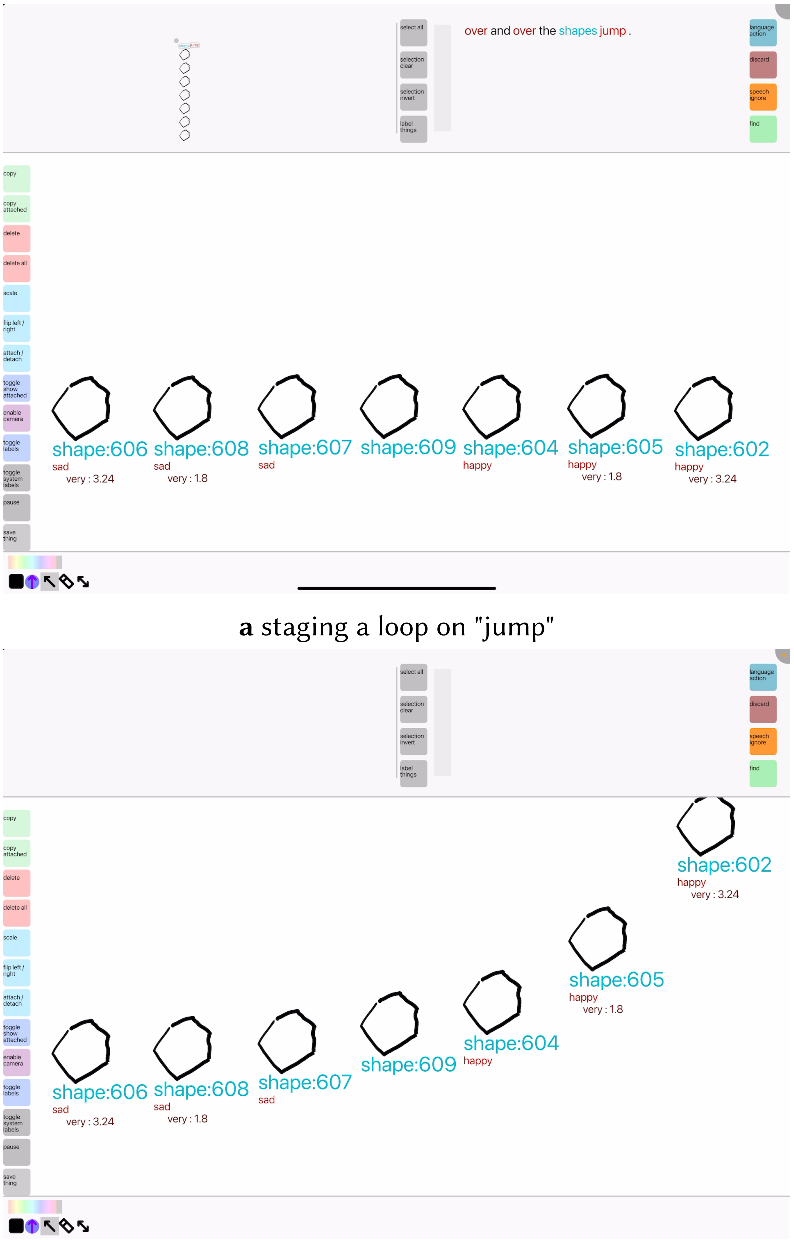
a (left) labeling by speech + pen/touch, (right) labeling by touch selection + pen selection of a noun

b pre-selecting with pen and using speech

c pre-selecting multiple objects with the pen and using speech

d labeling by linking an object to text directly, with selection + pen

Figure 7.3: Labeling sketches with semantics via 1) touch/pen + speech 2) link to text with pen

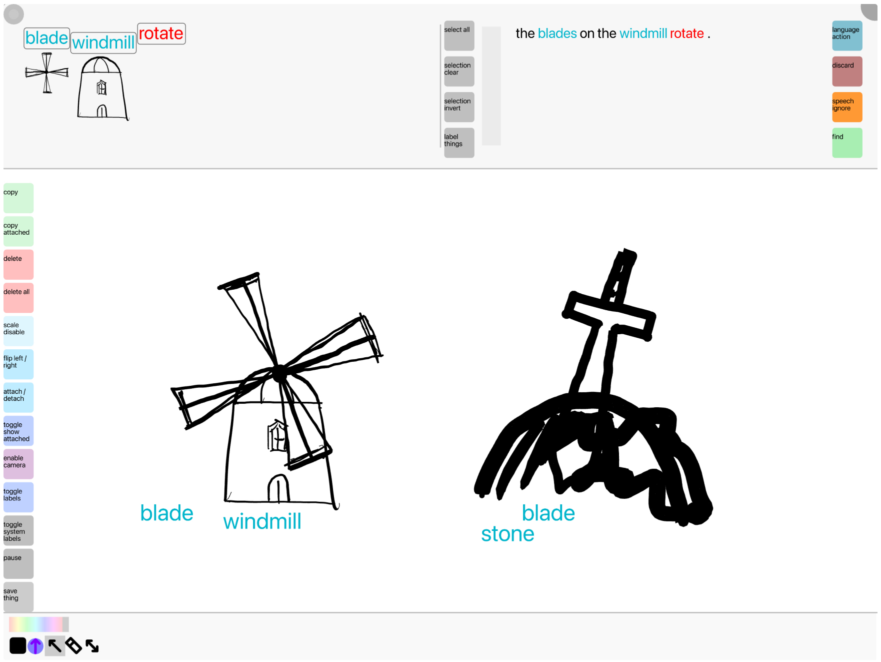


over and over the shapes jump .

a staging a loop on "jump"

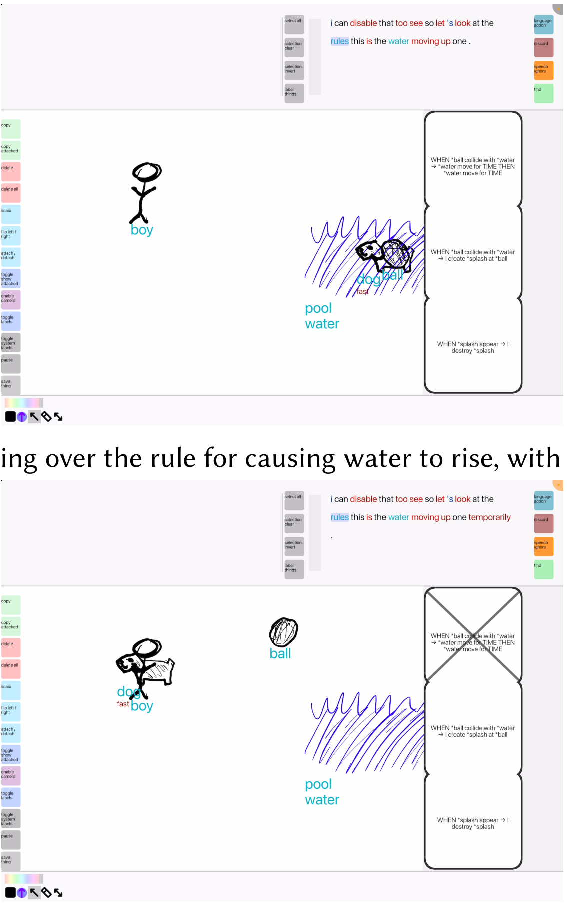
b we have the "jump" speed and height change based on the current adjectives and adverbs

Figure 7.4: Adjective effects - e.g. the verb "jump" is affected from left-to-right as shown via debug labels. "sad" has the inverse effect to "happy."



the blades on the windmill rotate .

Figure 7.5: Disambiguation by hierarchy - The system disambiguates between like-named objects by looking for context from other information, such as spatial hierarchy. Here, the correct blades are selected since the sentence describes blades attached to a windmill.



i can disable that too see so let 's look at the

rules this is the water moving up one .

WHEN *ball collide with *water → *water move for TIME THEN *water move for TIME

WHEN *ball collide with *water → I create *splash at *ball

WHEN *splash appear → I destroy *splash

a Hovering over the rule for causing water to rise, with the pen

i can disable that too see so let 's look at the

rules this is the water moving up one temporarily

.

WHEN *ball collide with *water → *water move for TIME THEN *water move for TIME

WHEN *ball collide with *water → I create *splash at *ball

WHEN *splash appear → I destroy *splash

b The rule has been disabled dynamically and temporarily no longer takes effect.

Figure 7.6: Rule Panel - view all existing rules, toggle them with the pen to enable/disable them, use the eraser to delete.

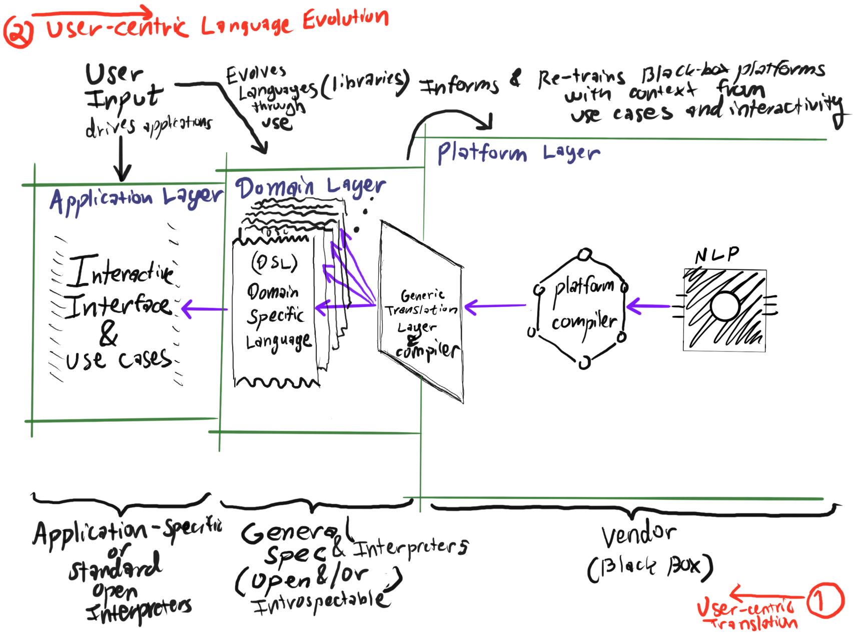
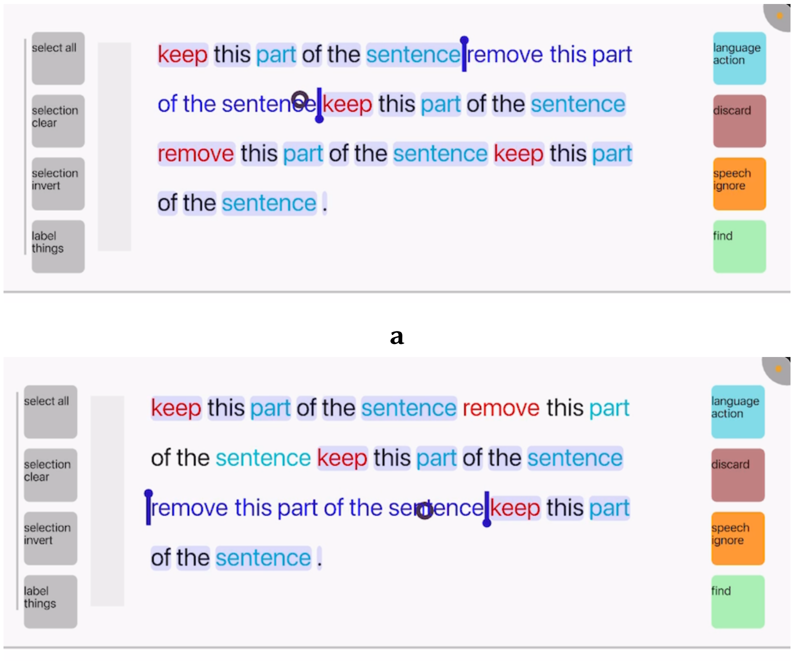

② User-centric Language Evolution
User Input
drives applications
Evolves Languages (libraries) through use
Informs & Re-trains Black-box platforms with context from use cases and interactivity
Application Layer
Domain Layer
Platform Layer
Interactive Interface & Use cases
(DSL)
Domain Specific Language
Generic Translation Layer & compiler
platform compiler
NLP
Application-Specific or Standard open Interpreters
General Spec & Interpreters (open &/or Introspectable)
Vendor (Black Box)
User-centric Translation ①

Figure 7.7: Translation Layers - From right-to-left, we can think of our prototype as an example of taking platform-specific language libraries or models, converting them into a generic format/DSL, and then sending to our domain-specific application.



keep this part of the sentence remove this part
of the sentence keep this part of the sentence
remove this part of the sentence keep this part
of the sentence .

a

keep this part of the sentence remove this part
of the sentence keep this part of the sentence
remove this part of the sentence keep this part
of the sentence .

b

Figure 7.8: *transcript view* selection/deselection - the user can omit words quickly from the next command by deselecting individual regions within the *transcript view*.

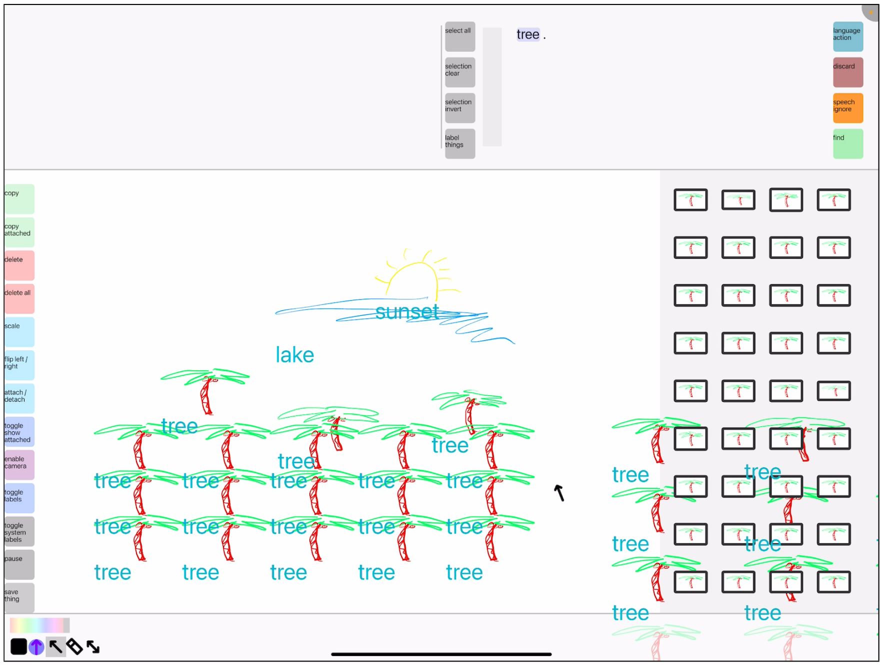


Figure 7.9: The "find" panel lets the user locate objects by name (selecting the word or multiple words in the *transcript view*) and warp to them by tapping on their entry. It also lets the user copy the individual objects or all of them to fill a scene quickly with cloned-objects. Here, the user is cloning trees into the scene multiplicatively, as the clones dynamically appear in the find panel.

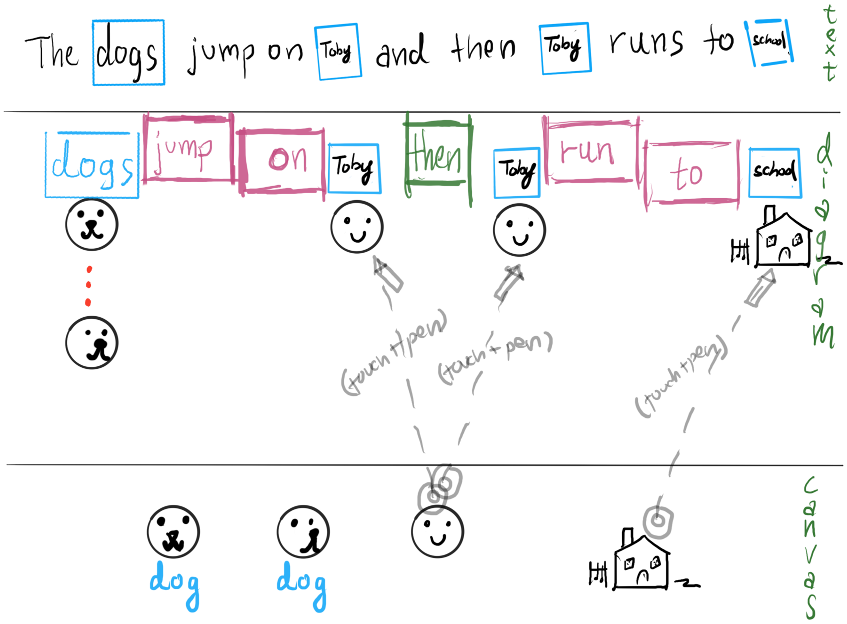


Figure 7.10: Editable Semantics Diagram. This displays a simplified form of the user's input text based on what the machine understood. Each noun in the diagram has a vertical list of "proxy" icons referring to the objects the machine selected. The user can edit the diagram to refer to different objects using a (∘, ⟶) operation. Note that objects with mismatching labels can be substituted-in. In this mock-example, two dogs exist in the scene, but no objects labeled "Toby" or "school" exist. The user in this scenario connects the desired sketches to the diagram to complete the command.

8 Putting It All Together by Example: Demos and Things

Here, we go through how to create sample scenarios and setpieces for mechanics built-up in DrawTalking. We show how the workflow fulfills the criteria for an interactive systems contribution [Olsen 2007] — particularly, 1) reduced solution viscosity (in which our use of language and sketching enables flexible design iteration and achieves *expressive leverage and match* with user intent[1]), and 2) power in combination (in which the primitives are capable of being recombined to express things with greater complexity). Please see 13 for a list of existing functionality as we go along.

Through the design, iteration, and feedback/pilot process, we consistently felt the *draw* of such an interface was partly in the playful feel and rough aesthetic – being able to create logical simulations and setpieces just with rough sketches and simple speech. We hypothesized that the range of application domains would be broad, ranging from playful interactive storytelling, animation creative exploration in design, and illustrative explanations, with all of these incorporating user-interaction and control. But the domain that brings together all these elements i.e. "playful," and "creative," "simulation," "ideation" fits well with game prototyping, mixed with a rough sketching, almost storyboard-like aesthetic. So, our examples tend towards playful, game-like worldbuilding and game mechanics. We start with simple examples based on rough-sketched

[1] This essentially describes being able to express more with less and "how close the means for expressing design choices are to the problem being solved"

characters and build-up to more advanced examples approximating functionality you could expect in more full-fledged game prototypes. We show how DrawTalking's approach supports a blend of rough sketching and language with computational world-building.

Note: we skip describing the labeling step, as this can happen at any time before commands operate on given objects.

8.1 Simple Set-Pieces and Mechanics

We start with simple mechanics to show how DrawTalking works. Let's assume the user wants to iterate on a few animated scenes using rough figure sketches of characters and animals, e.g. Rosie the dog. Note that in the end, these small scenes could be composed together into a larger whole, so these examples illustrate an iteration process on small pieces.

8.1.1 A boy and dog play fetch.

We illustrate with an "infinite game of fetch"8.1 how we can build-up looping simulation sequences with full interactive control by the user. Note that the examples continue from one another unless otherwise-stated.

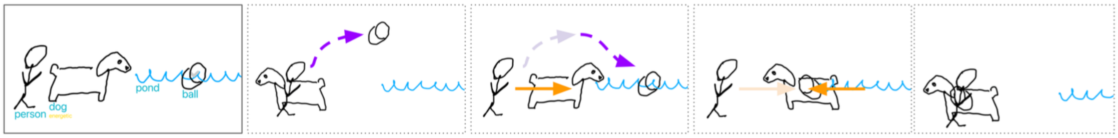


Figure 8.1: Infinite Loop created by "Over and over the person throws the ball into the pond and then the dog gives the ball to the boy."

8.1.1.1 The simplest sentence.

The user might begin by sketching a dog (e.g. Rosie the English Springer Spaniel) and testing it with a basic action. (table 8.1)

draw	dog
objects	dog
speech	"The dog jumps two times."
confirm	
command begin / end:	The dog jumps in-place two times.

Table 8.1: Instructions: dog jumps two times

Note that the result maps to exactly what the user said.

8.1.1.2 The Simplest Multi-object Sentence.

Here is a multi-object command. The user can insert themselves into the simulation and move objects while the command is happening, but the actions will still adjust dynamically to the positions and states of the objects to complete the task, meaning the user does not need to worry about precision or accurate "physics." The logic takes priority. (table 8.2)

draw	pool
objects	dog, pool
speech	"The dog jumps into the pool."
confirm	
command begin	
user moves the pool	path between *dog* and *pool* adjusts dynamically as user moves the target to adjust the scene
command end	The dog is atop the pool after having moved in an arc

Table 8.2: Instructions: dog jumps into pool

8.1.1.3 Indirect Object Commands

This is an example of directing an object to perform a simple task using other objects. (table 8.3)

draw	boy, ball
objects	dog, pool, boy, ball
speech	"The boy throws the ball into the pool."
confirm	
command begin	The boy moves to the ball and then throws it in an arc towards the pool.
command end	

Table 8.3: Instructions: boy throws ball

8.1.1.4 Infinite Sequences

It can become infeasible for the user to direct all steps of a simulation themselves. This example shows how the user can **offload repeated tasks to the machine** by creating the equivalent of a loop 8.1. (table 8.4)

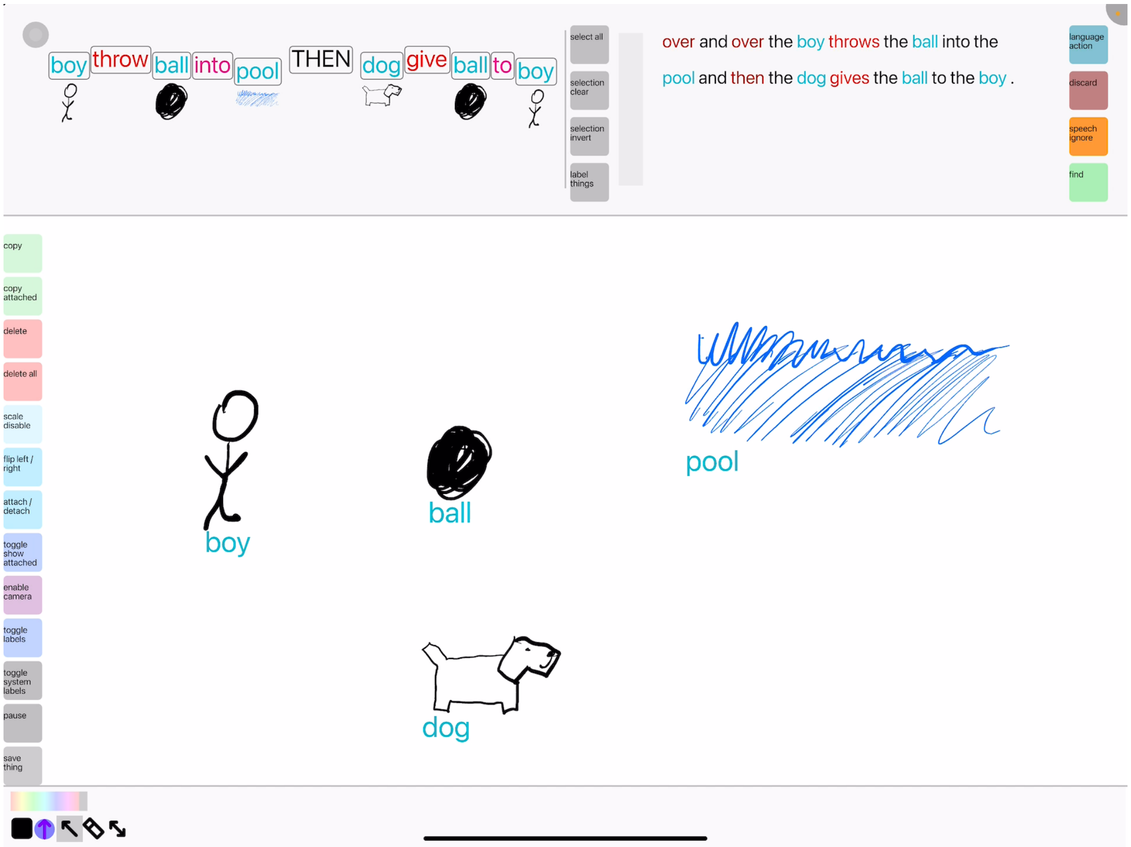
objects	dog, pool, boy, ball
speech	"Over and over the boy throws the ball into the pool and then the dog gives the ball to the boy."
confirm	
command loop begin	The boy moves to the ball and then throws it in an arc towards the pool. Afterwards, the dog moves to the ball and then gives it to the boy.
command loop repeat	

Table 8.4: Instructions: boy throws ball, dog gives ball to boy - loop

See an image here: 8.2

Moving any objects in the scene will dynamically alter the trajectories and paths of the simulated objects. This lets the user fine-tune or perform interactively while the machine directs the logic of the scene for them, all without needing to redo the commands to make adjustments.

We can continue to iterate on the scene. For example, labeling the dog "energetic" will speed her-up.



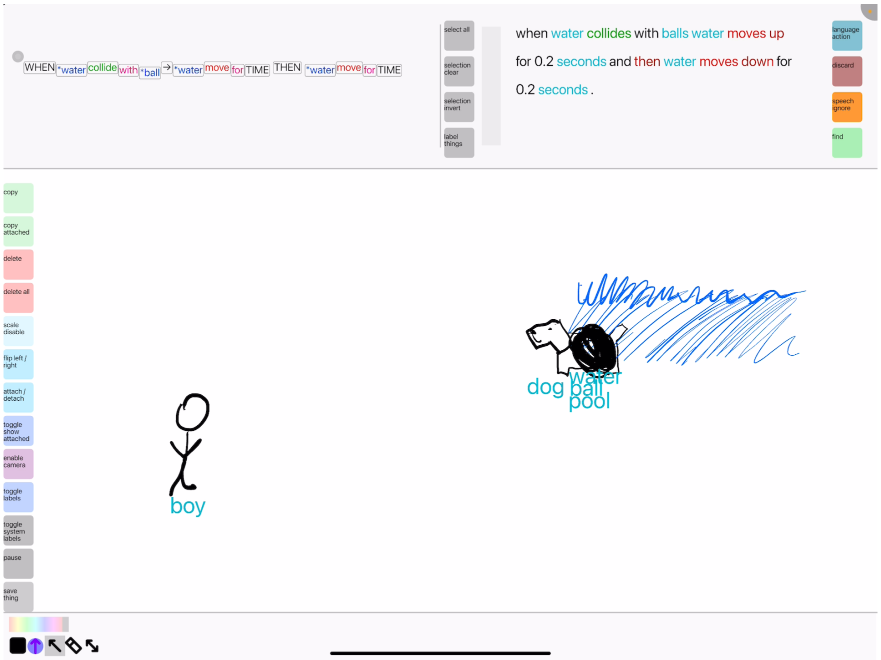
over and over the boy throws the ball into the pool and then the dog gives the ball to the boy .

Figure 8.2: Dog fetch infinite sequence

8.1.1.5 Embellishments by Event-based Control

The user might want to prototype some visual embellishments. For example, causing the pool to rise and sink upon impact with the ball using a *rule*. We show how to try this idea independently of the running simulation. (table 8.5)

See 8.3.



WHEN *water collide with *ball → *water move for TIME THEN *water move for TIME

when water collides with balls water moves up for 0.2 seconds and then water moves down for 0.2 seconds .

Figure 8.3: Dog fetch infinite sequence with rule for rising water upon collision with balls

objects	dog, pool, boy, ball
speech "This is water" and **Touches pool**	assigns additional label "water" to pool
objects	dog, (pool,water), boy, ball
speech	"When water collides with balls water moves up for 0.2 seconds and then water moves down for 0.2 seconds"
confirm	
command begin/end	Instantly creates a general rule that will trigger a rise/fall effect when ANY object labeled as "ball" collides with ANY object labeled as "water."
command loop begin/end	The dog playing fetch loop is still running, but now when the ball lands in the water, the rule causes the rising and falling.

Table 8.5: Instructions: water rises when ball collides with water

8.1.2 Randomization with Pond Creatures

If we move to a different section of the canvas, we can play with other ideas.

For example, we can create a set-piece involving other creatures (e.g. frogs and butterflies) moving with some basic random behavior. (table 8.6)

We'll define objects up-front for brevity.

8.1.2.1 Hopping Frog

draw	frog, lily_pad
objects	frog, butterfly, lily_pad
mass-copy lily_pad	We search for the lily_pad and copy en-masse
objects	frog, Many(lily_pad)
object movement	The user moves the lily_pads around.
speech	"The frog hops to a lily_pad every second."
confirm	
create verb alias	The word "hop" us unknown, but the interface suggests the word "jump" to the user for the closest approximation, and the user accepts.
command loop begin	This creates an implicit loop: the frog will hop to a random object with the label "lily_pad", wherever it is.
command loop repeat	A new lily_pad is selected
draw	lily_pad
objects	frog, Many_Plus_One(lily_pad)
command loop repeat	A new lily_pad is selected, which will include the one just added, and exclude any whose label might've been removed

Table 8.6: Instructions: frog hops to a random lily pad

We've shown another instance of how the user can individually control elements during iteration time and let the machine handle some nuances in the behavior if they do not require an exact visual – in this case, the task of selecting another target randomly is desirable for a simple behavior guiding a frog hopping around.

8.1.2.2 Layering with Butterfly

We might want to layer different behaviors on top of currently-running simulations. (table 8.7)

draw	butterfly
objects	frog, butterfly, Many(lily_pad)
speech	"The butterfly follows the frog"
confirm	
command loop begin/end	The butterfly follows the frog forever
speech + Touch the butterfly	This butterfly is slow
confirm	
command begin/end	The butterfly caught up with the frog too quickly, so the user decides to slow-it down as the simulation is running

Table 8.7: Instructions: frog hops, butterfly follows frog

Now we have a pond scene.

See 8.4.

8.1.3 Logical Gadgets, UI, Numerical Inequalities

8.1.3.1 Lightbulb - Definitions for Custom Verbs and Dynamic Property Assignment

Using similar mechanics as above, we could create a flickering lightbulb. "Flicker" is not defined, but we can define it in-terms of "appear" and "disappear."

Simply: "when things flicker, things disappear for 0.2 seconds and then things appear for 0.2 seconds.

Now, when we draw a lightbulb and tell it to "flicker forever," it will use our definition of "flicker".

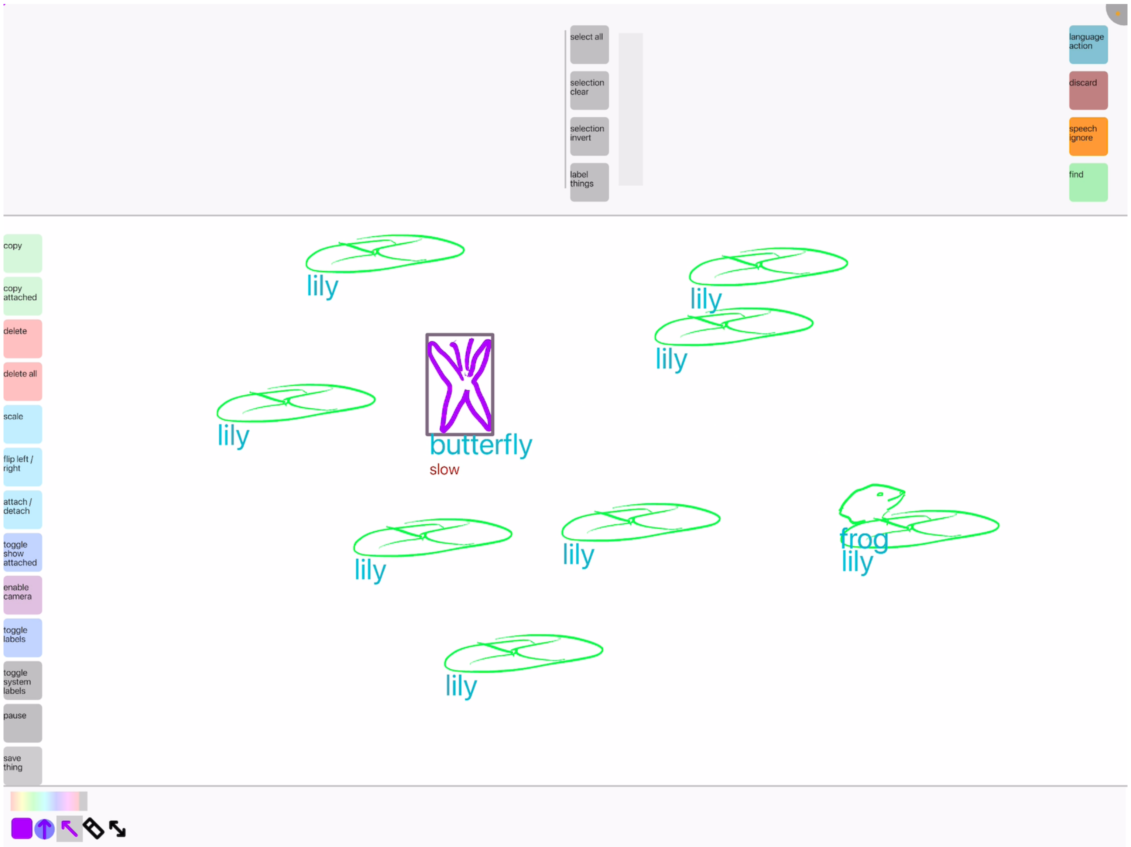


Figure 8.4: The frog infinitely hops to a random lily pad as the butterfly follows the frog.

We might want to create a static button in screen-space 7.4.3.4 to trigger the lightbulb as well. { "When I press this button the lightbulb flickers" } .

We can also create our own version of an on/off switch by using our own objects and rules.

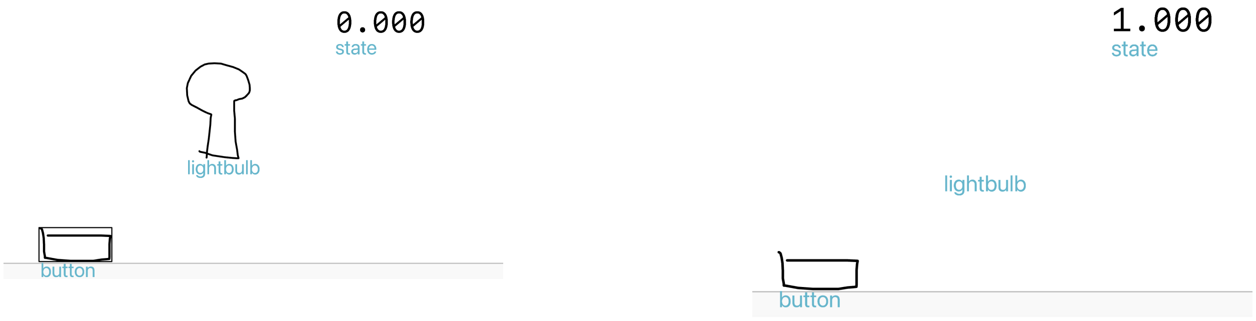
Let's create a number sketch and label it "state."

When its value is 1.0, the light will disappear, at 0.0, it will appear. "appear" and "disappear" will be treated as events. We create a rule stating that when we press the button, the "state" will increase. Another rule will be created so when the "state" equals 2.0, it will equal to 0.0 (immediately set to 0.0).

Using existing primitives, we've just created the equivalent of modulo boolean!

Note: we could also achieve the same by dynamically assigning adjectives to trigger events.

In this case, we define:



a The user just pressed the button at the initial state ...

b ... and the state increased

Figure 8.5: Visual Boolean as Visualized by a Lightbulb Sketch

{ When lights become broken they flicker forever. } (to cause endless flickering), { When lights become functional they stop flickering and then they appear. } (to stop the flickering and leave the light in the appear state).

Now, if we want our button to toggle all lights in the scene, we can rely on the current states of the lights to disambiguate:

{ When I press the button all broken lightbulbs become functional. } , { When I press the button all functional lightbulbs become broken. }

Lastly, we purposely left the library of adjectives minimal, so it's not obvious to the system that "repaired" should be removed when "broken" is added. We can define this behavior ourselves (and opens up possibilities for our own world logic or custom words).

{ When things become functional things become not broken } , { When things become broken things become not functional } .

OR (since pronouns here are unambiguous:)

{ When things become functional they become not broken } , { When things become broken they become not functional } .

.

Now, the lightbulbs have on/off states represented by "broken,"/"functional" rather than an external number object.

This alternative to the first approach allows for control of many more objects of the same type at the same time. The button, also, is not required. Any speech command directing one of these lightbulbs to "become" an adjective (become "broken") will trigger these state changes.

8.1.3.2 Windmill - Custom Objects and Spawning

Here we create a windmill that rotates it blade when user-created wind objects collide with it. It's an example of creating a chain reaction with rules and custom-labeled objects.

We create the base of a windmill, with an independent blade sketch attached to it as a child **{** "The blades attach to the windmill" **}** or using the "attach"/"detach button.

We create and save a template for a custom object with (○, ○) between a sketch and the "save" button. This will save the object mapped to its noun and adjective labels. When we use the verb "create," the system will clone an object based on the specified labels. We sketch a little wind swirl and label it "wind," then save it.

For our rules:

{ "As wind collides with blades blades rotate" **}** - "as" meaning continuous collision.[2]

We test by moving our wind sketch through the windmill to verify the blades rotate.

However. we realize that the blade do not stop rotating after the wind stops colliding.

We add a rule:

{ "After wind collides with blades blades stop rotating" **}**

to end the rotation upon collision end.

[2] To create different behavior for left/right collision, we can create additional invisible collider objects on either side of the windmill, so the rule could refer to these specifically. Our engine does not expose collisions with the relative coordinate (yet) but easily could to allow rules such as "As wind collides with ... from the left side"

We delete the original wind sketch because we no longer need it. We could also create several other wind sketches and save them to add variety ("create" will randomly-selected equivalent options).

Next, we create a spawnpoint for the wind by drawing a simple box and labeling it anything. e.g. "boundary." This should just be a point for reference, so we can make it invisible with **{** "This thing disappears" **}**

We'll use a static UI button again to test the mechanics. To make the wind move rightwards from the wall, we can define: **{** "When I press the button I create wind at the boundary" **}** **{** "When wind appears wind moves right." **}**

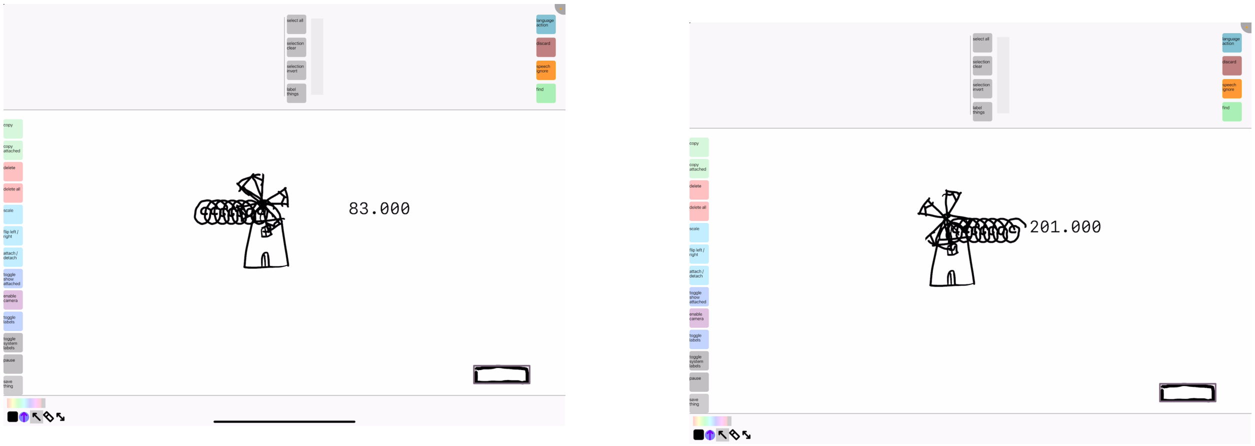
When we press the button, a new wind sketch will spawn and move rightwards through the windmill blades, causing them to rotate.

We notice that we're creating too many wind objects. We can create another boundary (call it "second boundary") off-screen on the right, which destroys the wind on collision. **{** "When wind collides with second boundaries, second boundaries destroy wind." **}**

Optionally, we can create another number object and perhaps increase its value as wind collides with the windmill blades, to represent "power."

We're left with a little windmill gadget we can play with and use as a potential mechanic for generating power. See 8.6

This example is possibly the most advanced so far because it involves more of a chain reaction. Notice, however, that the workflow is largely the same as before. The sketching, language, and rule-building operations can all happen at different times, and the user has time to think and work-out how to proceed. We've shown how at each step the user might realize a limitation of their current design and immediately address it.



a The user's custom wind sketches moving from the left; the user has spawned the wind by pressing the button in the bottom-right multiple times.

b The wind sketches moving rightward across the windmill's blades have caused the power to increase

Figure 8.6: A windmill's blades rotate as wind swirls continuously collide with the blades, causing the power counter to increase

8.1.4 Takeaways

We've shown how starting with a simple example, the user is able to construct increasingly complex functionality, delegate it to the machine, and continue trying other things in parallel. We can create interactive gadgets and hooks into our semi-automated scenes to let us have a hand in the simulation as well. We've also shown that using simple language also helps us use and navigate editing features in the interface. Moreover, based on the above, we could easily combine scenes and logic to continue building.

Note that the visual fidelity and quality of the animations are a separate concern from the functionality behind DrawTalking's control mechanisms. We can see our approach applying to more production-oriented use cases in other implementations.

8.1.5 Examples towards Prototyping of Gameplay Mechanics

In this section, we walk-through examples that are designed to show how one might use the controls to iterate on mechanics for arcade-style games and interactive diagrams, or to illustrate more specialized examples of how we might apply our approach.

We will show 1) elements of a 2D platforming game, 2) how one might try-out and repurpose mechanics (using the classic games Pong[noa 2022], Breakout[AtariAdmin 2022], and Space Invaders[noa 1980] as examples), 3) event-based character behavior, 4) a molecule-matching lesson, 5) a blend of several examples into one world.

As for why move to games from sketching at a whiteboard sketching with animations:

We wanted iterative and playful tasks that would result in runnable artifacts across a broad range of domains (i.e. a playable mini-game prototype). Many whiteboarding tasks might work: e.g. designing an interactive mechanic, improvising an animated story, describing physical properties and behavior, explaining a concept with motion graphics. Game prototyping via rough sketching is an adequate umbrella because it can involve a little bit of all of those examples. As we found in 6, much of the content people need is achievable with simple primitives and capabilities with a rough aesthetic aesthetic. The rough sketching medium lends itself well to quick and playful trying-out of mechanics, and the language controls should enable the user to represent many of the requirements in worldbuilding (selection, rules, variables, dynamic behavior) The introduction of interactive rules to our interface led us to extrapolate from this more explanatory content into more simulated, game-like content. Overall, game prototyping is representative of several other use cases for interactive animation and explanation, and it involves programming (or programming-like activity), so it was a good domain for examples. All-in-all, we tried to be creative in designing simple interactive examples to showcase our approach, so game-like simulations made the most sense.

Games and game-like simulations, furthermore, demonstrably require more multitasking and exploration from the user because they tend to have more possible outcomes to explore and elements to track. As a result, the user is likelier to benefit from the machine's ability to automate parallel tasks like procedural animations and simulations for the user. However, the workflow remains the same. The number of required primitives and inputs stay relatively low and manageable in size compared with the simpler examples. This is because the user has the freedom to "try things out." They can continue to take their time to layer and recombine functionality over their iteration process, rather than staging all things at-once. The approach "scales-up" with more complex functionality. Nothing stops us from creating more advanced functionality in a fleshed-out implementation.

Also, games and simulation-building is essentially a form of programming. These examples (though still simple) provide a proof-of-concept of how an implementation of our techniques might support programming-like tasks. We hypothesize that this would make programming more accessible for learner-onboarding and provide a quick way for a larger audience to express themselves. (In section 10, some participants liken the approach to beginner-tools like [Maloney et al. 2010], but find it simpler and more natural to understand once learned.) This would give us a positive sense that our approach starts to work towards the greater vision for natural interfaces while also serving the needs of more audiences.

8.1.6 Interactive Gameplay Mechanics Prototyping

This example illustrates how one might start prototyping and visualizing game mechanics and abilities that could feasibly be part of a 2D game. The dog will have the ability to collect items for the boy, and we'll try a couple of ideas for how this will work. The boy will have standard jumping abilities. We'll have a simple life system with respawning at checkpoints upon damage, and a points and lives counter HUD.

8.1.6.1 Character Abilities

Let's create a mechanic in which the dog fetches collectibles for the boy, similar to the earlier example.

We'll try a couple of ideas:

First we'll have a point counter (via a number object called "score," made static in the corner of the screen.

Let's say we want the dog to collect biscuits to get points.

We create and save a biscuit sketch.

Now define our collection logic:

{ When dogs collect biscuits the score increases and then I destroy biscuits **}** , **{** When dogs collide with biscuits dogs collect biscuits **}**

test the logic with an automated command (after we've drawn a biscuit and copied-in several)

{ Over and over the dog moves to a new biscuit. **}**

–and we see the score increasing.

The dog becomes a companion that collects biscuits for the boy. Now that we've tried the idea, we can make it easier to repeat by creating a button to trigger the action.

In the corner of the canvas we create a static UI button labeled "button" and use the rules **{** "When I press the "button" a boy throws a ball at a biscuit and then a dog moves to it." **}** [3]

But collecting any biscuit would be overpowered. So let's define a collection radius called "proximity"[4]: we draw a circle around the boy, make it invisible, and attach it to the boy.

We can use the speech input to tell the interface to center the collider precisely on the boy by narrating: **{** "The proximity moves to the boy and then it attaches to the boy." **}** We can always edit the collider later or recenter it conditionally.[5]

[3] Using "a" ensures that if we accidentally delete these characters, other sketches with the same label can be commanded instead.

[4] Probably easier to type this one

[5] We can also see how automating the collider could've led to other interesting behavior, for example, if the boy

We add the following rules:

{ "When proximities collide with biscuits, biscuits become nearby." } and { "After proximities collide with biscuits, biscuits become not nearby." }

Now we revise the above just to have the dog move: { "When I press buttons a dog moves to a nearby biscuit." }

Now we've balanced the mechanic so the boy's dog companion only picks-up a biscuit that is close-enough to the boy. See 8.7.

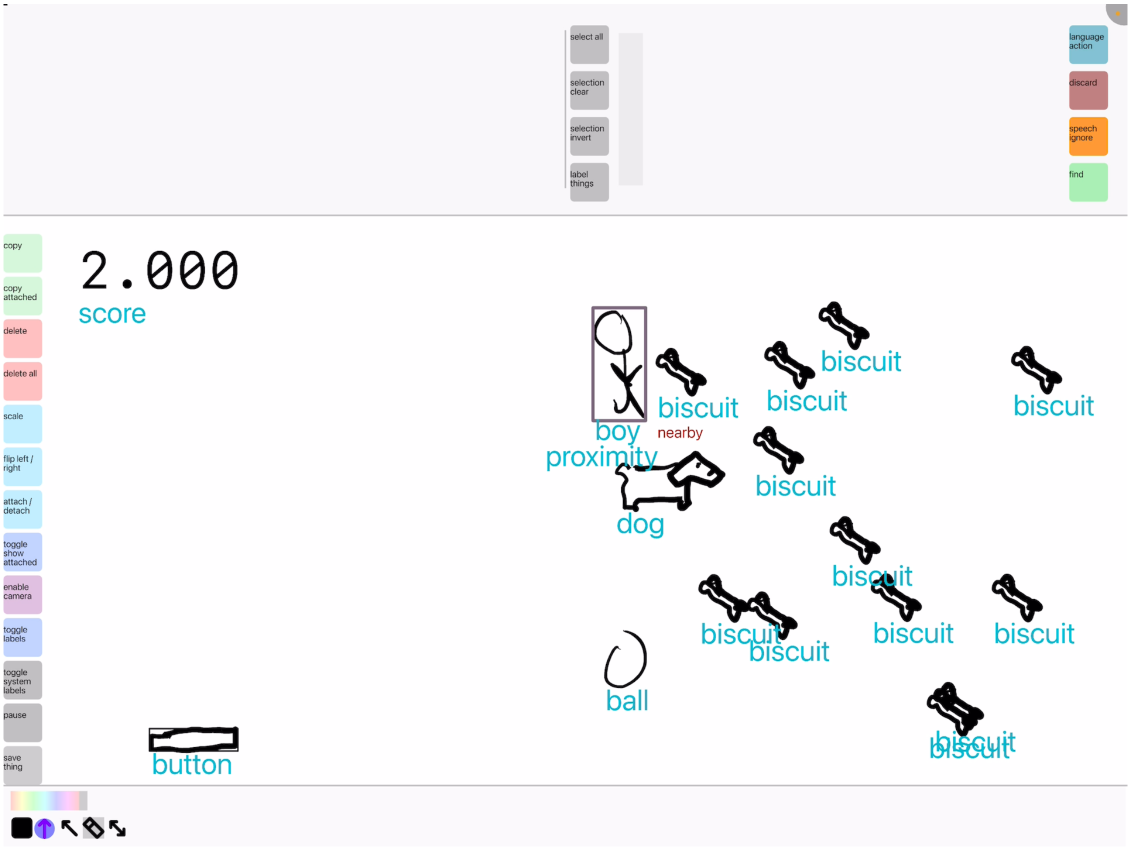


Figure 8.7: The dog fetching biscuits as a mechanic. This demonstrates a mechanic bounded by rules, as well as how to use custom collision objects. When the boy's proximity collider touches biscuits, they become labeled as "nearby." When the boy's proximity collider no longer touches biscuits, the "nearby" labels are removed. The dog will only collect the "nearby" biscuits.

Here we've seen how Sketching+speaking is useful not only for driving animations, but also

could collect points by swinging the collider with "revolve around."

telling the system to do precise operations for you and create reusable behavior.

8.1.6.2 Custom Movement, Movement Control, Collisions

We some way for the user to control them more easily.

For control, the user can again define a full button UI with a few static button sketches. Left-/right/up/down movement can be mapped to buttons labeled for those movements, representing a directional pad drawn on-screen. Releasing those buttons can direct the boy to stop moving. **{** "When I press this button the boy moves left" **}** (tapping the specific button to disambiguate with the other button), **{** "When I press the button the boy stops moving" **}** (tapping the specific button again). Repeat for the other buttons.

This can start to approximate an 2D overhead map game.

If we'd like to playtest, we can direct the camera to follow the main character using the context-sensitive version of "follow", simply: **{** "The view follows the boy and dog." **}** . This will make sure both characters remain in-view.

8.1.6.3 Lives, Checkpoints. Obstacles

Using essentially the same ideas from previous examples, we can create a life counter that decreases when the character collides with a obstacle. When the life counter depletes to 0, the system can spawn a user-defined game over screen text.

Normally, losing a life in a game might send a player back to a previous checkpoint. We can do the same with an invisible "spawnpoint" sketch and any number of checkpoint sketches. **{** "When boys collide with checkpoints a spawnpoint teleports to checkpoints" **}** . **{** "When boys collide with obstacles all life_counters decrease and then boys teleport to a spawnpoint." **}**

8.1.7 Ghost Enemies - Transforming Sketches

Lastly, let's give the boy enemies.

When we "transform" a sketch into another sketch, it retains its identity, but is replaced with the visual representation and set of labels of the target. This is useful for creating and automating contextual behavior changes, as well as dynamically changing graphical representation of objects independent of other actions they're performing.

To try this out, we create a ghost enemy that is only dangerous during the night.

We create and save sun and moon sketches (and delete the moon). The sun should transform into the moon and vice-versa. We can treat the sun/moon object just as a button as in other examples that the user presses themselves, or we can alternatively trigger the change based timed collisions between objects, among other ways.

This will create a day/night cycle (user-driven, or automated).

A side-note on transformations: we can also implement basic keyframe animation states by using ordinal adjectives as labels. e.g. transforming "first frames" into "second frames" into ... every so-and-so fraction of a second upon an event.

Now we create and save "cursed villager" and ghost sketches.

Define: **{** "When moons appear all cursed villagers transform into a ghost. **}** , **{** "When suns appear all ghost transform into cursed villagers" **}** .

Let's make these enemies follow the boy upon appearance.

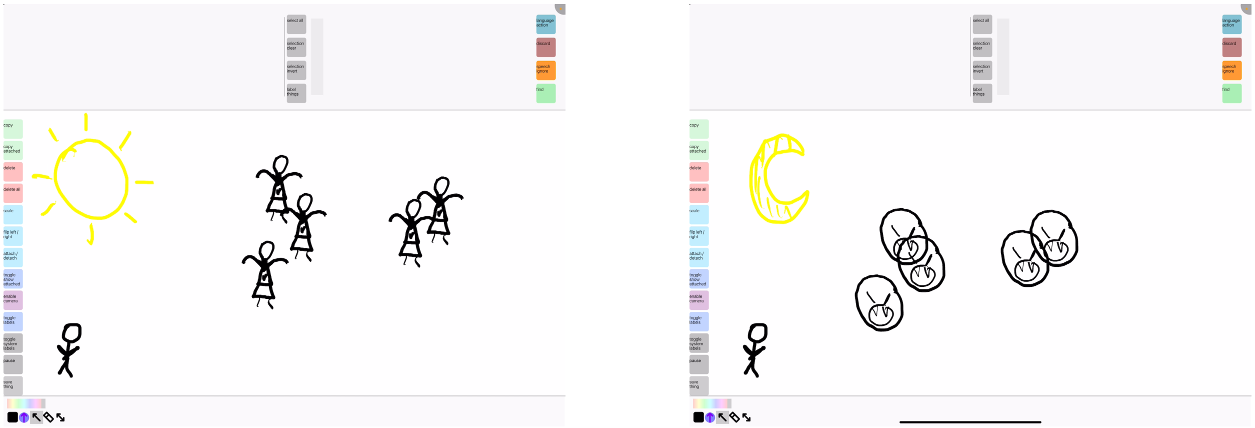
We define just that: **{** "When ghosts appear ghosts follow a boy **}** .

Now if we create several cursed villagers, they will transform into ghosts when the sun transforms into a moon, and vice-versa.

See 8.8.

We could go on to make the scene even more interesting by, for example, having a particular enemy causing regular villagers to become cursed upon collision, and maybe by having our goal be to cure all of the villagers by giving them potions.

With several of these little working prototypes, we can see how various ideas could work with each other, independent of the complexities of a full engine.



a The sun is out, so the villagers are represented as villagers, and are not following the main character

b The moon is out, so the villagers have transformed into ghosts, and are now following the main character.

Figure 8.8: Sun/moon and villager/ghost cycles This demonstrates dynamic changes in behavior on the same objects.

8.1.7.1 Takeaway

We've taken one of the very first simple "dog plays fetch" examples and shown a possible path to extending it into a much more complex set of custom functionality — all within the same environment. We can continue to add objects and define more behavior as we play with our simulations and animations.

We've essentially shown a proof-of-concept for how our approach could help people prototype game mechanics quickly using just language and rough sketching. Different implementations could expand the functionality greatly and still use the same sorts of inputs to control them.

8.1.8 Pong, Breakout, and Space Invaders - Reuse and Recombination of Mechanics

In this particular set of examples inspired by pieces of classic arcade games, we'll focus on another example of how the user can transform one idea into another using the existing primitives.

We chose these games because they're simple and well-known.

8.1.8.1 Pong Transformed into Breakout

Pong requires a paddle per player, a score count per player, level boundaries for walls and goals, and a ball to reflect off the boundaries and goals.

We can have the scene set-up like so:

— such that each player has an up/down button to move the paddle as an alternative to moving the paddle themselves using touch. (We define press and release behavior to move the paddles up/down and stop them accordingly.)

The start button causes the game to start with the ball moving fast to the right.

For the walls, we define **{** "When walls collide with balls walls reflect balls" **}** and for each player goal one way to to phrase two rules is like so:

{ "When balls collide with first/second goals second/first scores increase and balls stay" **}** – which increases the correct scores and cancels all velocity on the ball[6]

{ "When balls collide with goals balls teleport to a center and then balls move right fast" **}** resets the position of the ball and then sets it moving rightward at a faster speed than the default.

Lastly, we define reflection on the paddles:

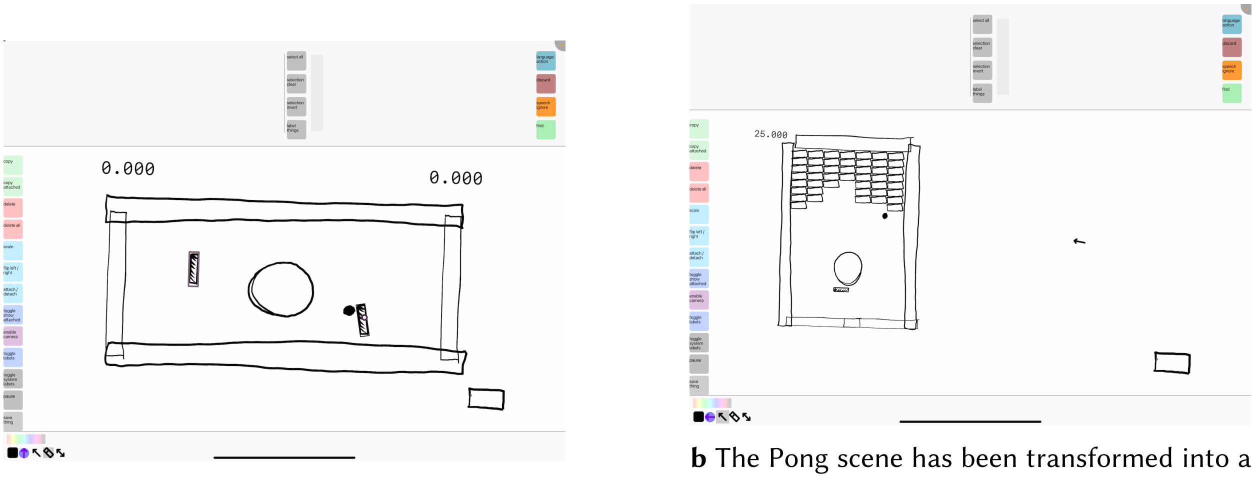
{ "When balls collide with paddles paddles reflect balls **}**

And we have a simplified form of the game Pong.

See 8.9a.

To transform it into Breakout, we could simply remove the second player sketches and goals, replace the rules on the buttons (or even rotate the canvas 90 degrees), and move a couple of colliders. The additions to Breakout are the field of breakable blocks and the game end condition (reach the goal behind the blocks).

[6] We decided that "stop" and "stay" would be useful with different meanings. "Stop ends an action, and stay specifically nullifies velocity."



a Pong-inspired mechanics

b The Pong scene has been transformed into a variant of Breakout, reusing many of the same rules, and objects

Figure 8.9: Pong- and Breakout-inspired mini-games The bottom-right button starts the ball's movement. Labels have been disabled here to de-clutter.

We sketch a quick block for the user to break with the ball and to reuse the wall reflection behavior from before we add a second label "wall," then save it.

To speed-up our drawing practice, we can sketch a temporary rectangular "region" and say { "I pack the region with blocks" } to fill the region with the block object we just defined. To make the blocks destructible and increase the point count, create a rule { "When balls collide with blocks balls destroy blocks and then the points increase." }

See 8.9b.

Lastly, we could create a text object with the word "win title," save it, and spawn it when the ball reached the goal. { "When balls collide with goals balls stay and then I create a win title" }

Now we've transformed a sketch of a working Pong clone into a Breakout clone reusing a mix of our own previously-defined functionality. This demonstrates a fair level of flexibility to try-out mechanics without needing to start over or worry about future changes in decision-making.

8.1.9 Space Invaders Elements Built from Breakout Elements

A common trick in games is to have objects appear to warp between sides of the screen to achieve an animation effect or looping movement. Let's create a looping version of the moving enemies from Space Invaders, which requires some timed events and sequencing.

We create and save an alien sketch.

Now we want to clone several of these and have them move in a snake pattern until they reach the bottom of the screen. Then they should teleport back to the start.

We make invisible walls on the left, bottom, and right sides and a spawnpoint at the beginning. We also make colliders representing triggers for moving left and right. The bottom zone teleports aliens back to the startpoint upon collision.

The following rule will create the spawning logic: **{** "When aliens collide with the bottom zone aliens teleport to a startpoint." **}**

For the movement logic we create our own "start" verb definition: **{** "When aliens start aliens move right. **}** **{** "When aliens collide with walls aliens stop and then aliens move down" **}** **{** "When aliens collide with r_walls aliens stop and then aliens move left." **}** **{** "When aliens collide with l_walls aliens stop and then aliens move right." **}**

See 8.10.

Now, we can augment the paddle from breakout by creating a shoot button that causes our own bullet object to spawn from the paddle. (This is similar to the previously-described examples). When the paddle collides with an alien, that might be a lose condition.

Note that we can think about this looping behavior as a building block for other games and scenarios as well. For example, Frogger requires spawning of random cars and obstacles that move from left to right as the title character navigates.

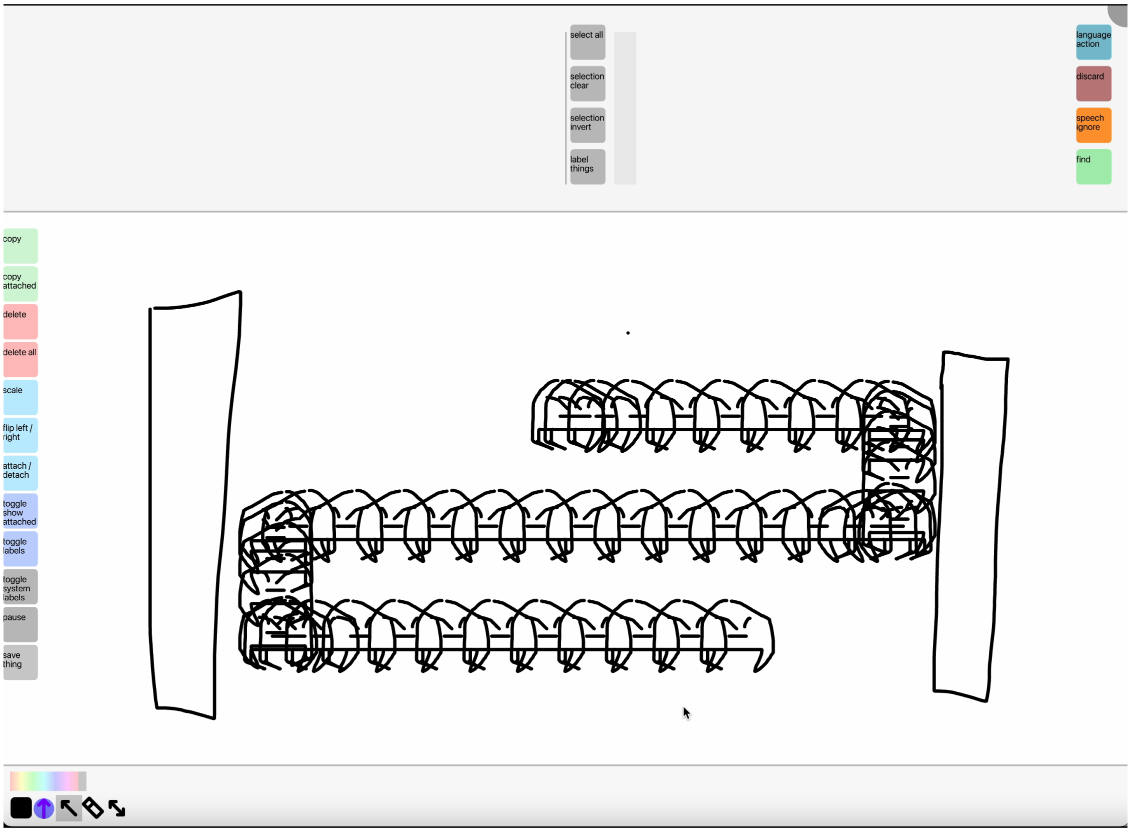


Figure 8.10: Space Invaders basic zig-zag-motion. This shows how one might start to approximate Space-Invaders-inspired motion by using walls as triggers for changing objects' trajectories.

8.1.10 Interactive Lesson

For one more example, we can create an interactive molecule matching game that could function as an exercise for an interactive lesson.

Let's create a box and a checkbox for it to transform into if the user gets the matching correct.

If we save a sketch structure of hydrogen and oxygen atoms attached together and name it "water," we can use this as a template for matching

Now using the verb "form" we can define: **{** "When atoms form water the box transforms into a checkbox" **}**

With this rule, the interface will check whether a group of sketches labeled as atoms form the same hierarchical attachment structure as the "water" sketch, with the correct "hydrogen" and "oxygen" labels. If so, the condition is fulfilled and the box becomes the checkbox.

See 8.11.

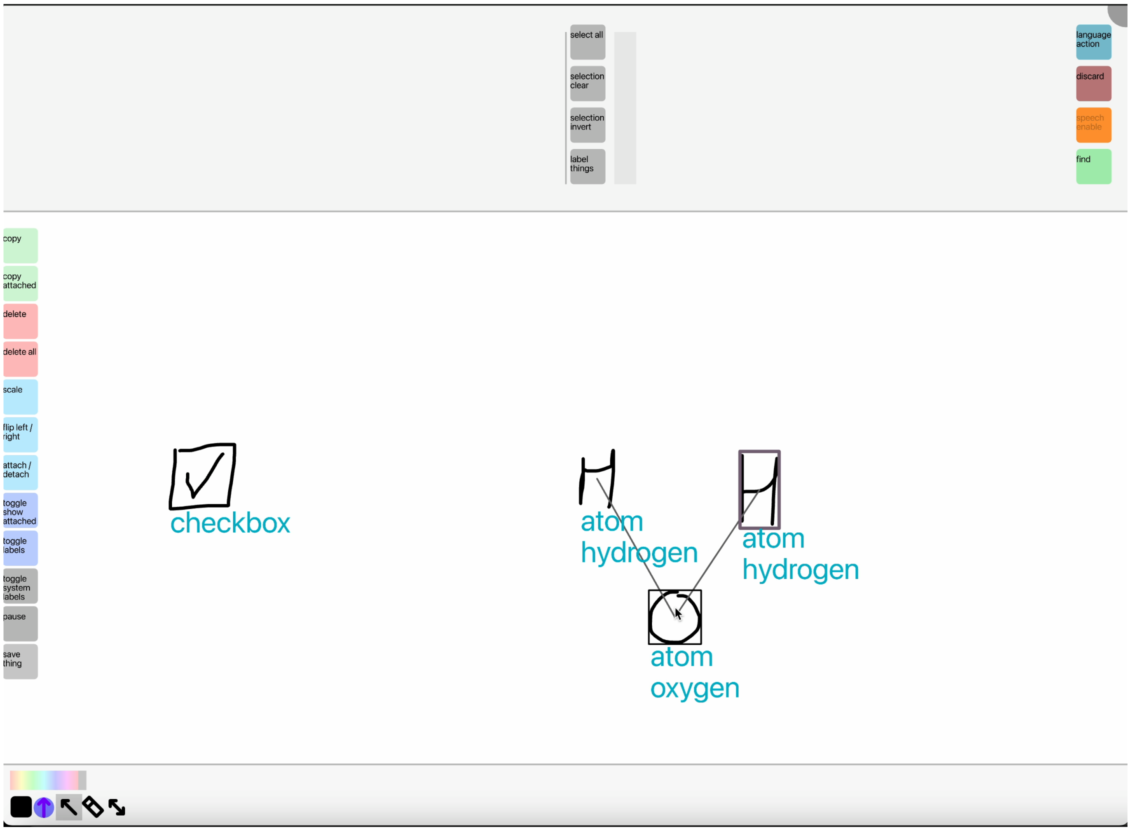


Figure 8.11: Molecule Matching Lesson Prototype - The structure of a water molecule has been saved prior. The shown mechanism spawns a checkbox if the user has fulfilled the rule: to create at least one correctly-structured and correctly-labeled hierarchy of sketches matching the water molecule. We believe this could be something one might want in an interactive lesson notebook or textbook.

This is just one example we've tried that shows how we could turn these interactive simulations into learning games that could fit in a digital notebook or worksheet use cases.

8.1.11 Overall Takeaway

In this section, we've illustrated the DrawTalking workflow through several examples showcasing capability as well as potential user iteration processes. We started from simple and small simulations and built towards larger interactive game-like prototypes. We've shown how using the same set of primitives and operations, we can flexibly and fluidly try-out ideas just through sketching and language. Due to the capabilities of the system, the user can manage several objects at once, define and reuse behavior, and explore several potential use cases. To reiterate, the approach to user/machine control using sketching and speech is our contribution. If we imagine our approach applied to more fleshed-out interfaces with larger sets of capability, it does not escape our notice that sketches and commands could easily represent variables and functions in different computing environments. In other words, as a takeaway from these examples, we consider how we can scale and export to other interfaces, and still preserve the functionality of our approach.

9 System Architecture and Implementation

We start by presenting the system architecture in the abstract (in terms of input and language processing layers) independently from the specific content features. Then we follow with our concrete implementation. By splitting the explanation we hope to clarify where our approach could help in the implementation of systems with different content needs, but which might benefit from a similar style of user interactivity as ours.

9.1 Abstract System Architecture

9.1.1 User Input and Object Selection

The system uses a combination of inputs. Language is captured and converted into text via continuous speech-recognition through a microphone, typing at a keyboard, or interaction with the application (e.g. manipulation of text objects). Direct manipulation input controls the application directly and represents what the user is touching or pointing to. e.g. using stylus/pen and multi-touch input. Concrete information such as position and orientation of the input event is stored along with the ID of whatever a user selects with a given direct input. Events are stored in-order per input-modality and retained so we can match sequences of selections with sequences of words in the future. Specifically, we match sequences of selection events with sequences of deictics to label or command objects in-order.

9.1.2 Translating Language into Application-Agnostic Form into Application-Specific Commands

The language processing component (in the abstract) comprises roughly 4 steps that translate a natural language data structure into something usable by the application as a command (9.1, 9.2).

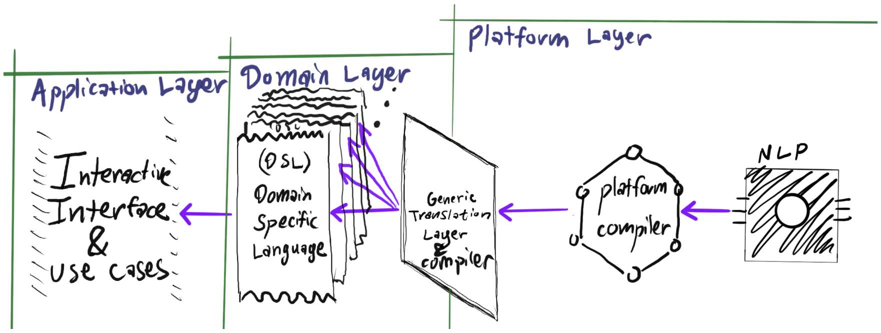

Application Layer
Domain Layer
Platform Layer
Interactive Interface & use cases
(DSL) Domain Specific Language
Generic Translation Layer & compiler
platform compiler
NLP

Figure 9.1: Abstract Model for Language Processing Stages

[**In** Natural Language Raw Text, **Out** S1] The system receives language input and annotates it using any method (modern models or classic NLP, libraries, etc.). It outputs a data structure (directed graph) encoding the dependencies between and semantic roles of words. Note that this representation does not prescribe a use case for the information. Instead, it's meant to be an abstract simplified structure built according to a known spec e.g. a simplified English grammar. Our implementation of DrawTalking uses the Spacy (v 3.1.4)[Montani et al. 2023] library to output a dependency tree per-sentence, along with a library called "coreferee" to fill-in coreference information.

[**In** S1, **Out** S2] The system traverses S1 and generates a new generic graph structure S2 from the information in S1. Each node entry in the graph structure represents a nested hierarchy of semantic units containing information such as labels (the word actually used) and part of speech

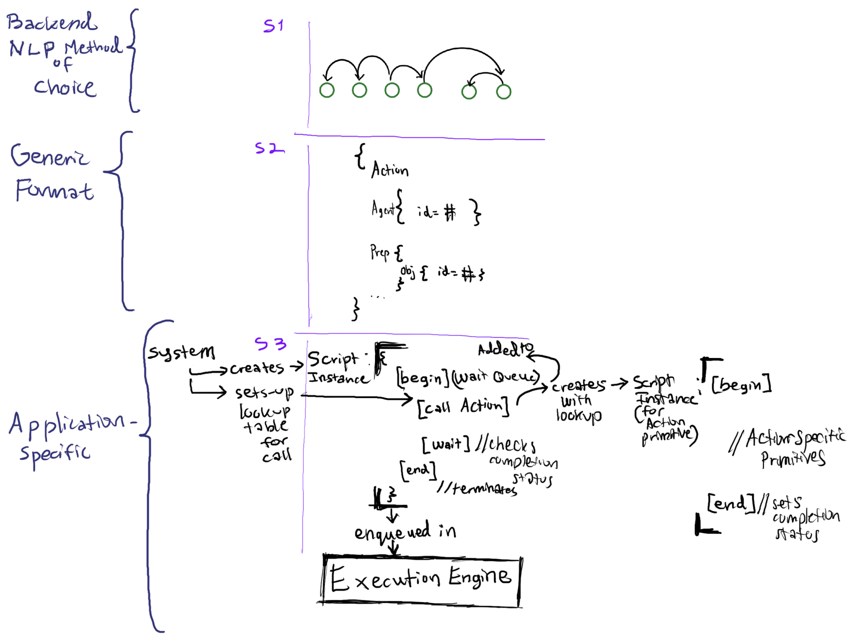

Backend NLP Method of Choice
S1
Generic Format
S2
{ Action
Agent { id= # }
Prep {
obj { id= # }
}
...
}
Application-Specific
S3
system
creates → Script Instance : {
sets-up lookup table for call
[begin] (wait Queue)
Added to
[call Action]
creates with lookup → Script Instance (for Action primitive) : [begin]
// Action Specific Primitives
[end] // sets completion status
[wait] // checks completion status
[end] // terminates
}
enqueued in
Execution Engine

Figure 9.2: System-Specific Language Processing into Intermediate Structures

(such as noun, verb, etc.). For example, ACTION nodes contain a label for the verb. Noun-like child nodes such as AGENT, DIRECT_OBJECT, INDIRECT_OBJECT, PREPOSITION, and so on have a label for the noun. Noun-like nodes might contain values including but not limited to numbers or entity IDs (if the user specifies specific objects in their language). ACTIONs and noun-likes can contain a PROPERTY, which is where we lookup and store information about adjectives and adverbs (For more details on the format, see 13).

[**In** Incomplete S2, **Out** S2 with System-Context] Feedback] Upon creating the structure for S2, there might be unspecified placeholders for objects, so we call it "incomplete." The system will now look at application-context such as user-input and objects. At this stage, the system fills the structure with concrete entity IDs as-needed. (In our case, the sketching application has spatial-visual context, so objects have positions and some notion of physics, so adjectives referring to a property like velocity are going to depend on the presence of these physical properties.). A query sub-system is needed to register and lookup objects based on their labels. For each noun-like entry in the traversal, the system queries for objects with the given noun and adjective labels, and returns the IDs of all objects in the world that match those labels. These are used to complete S2. Also, this is where the application can differentiate between object instances and object types (e.g. we differentiate using plural and singular), randomness, and how to interpret adjectives as what kinds of properties (e.g. as numbers, text labels). Deictics or direct linking are used to infer which objects the user referred to at a given moment and assign entity IDs to parts of the graph;

[**In** Incomplete S2, **Out** Complete S2 with User Feedback] After this process, the user should be able to do last-second editing and correction of the intermediate structure. The system can output user feedback, (which is what the 7.10 is).

[**In** S2, **Out** S3] Lastly, the application traverses the structure and generates a final application-defined structure S3 that it can evaluate – in our case, a mix between scriptable virtual machine and animation engine akin to [Perlin and Goldberg 1996; Resnick et al. 2009] or a runnable node-based program [cycling74 2018]. S3 can We'll call these scripts. S3 can retain S2 for reference. When traversing nodes in S2 labeled ACTION (the verbs) we lookup the appropriate previously-defined script for that action and insert a reference to it in S3. The arguments for that action, i.e. semantic role keys mapped to object ids or types, are inserted into a lookup table specific to the ACTION script instance so when the action executes, it knows what objects to modify. Loops and timed waits are also inserted into S3. The application evaluates the final structure according to its own interpretation. This results in any potential application-specific side-effect – in our case, animation, simulation, rule creation, application state-changes and so on.[1] The application can also retain the generic S2 structure to help generate commands later.

[1] In consequence, a different implementation might interpret the exact same data structure and see different parallel visual results.

9.2 Concrete Implementation

9.2.1 Hardware and Software

DrawTalking is implemented for the Apple iPad (2022 iPad Pro - A2436 - iPadOS 16) [2], with the main code written mostly in C++ and C, bridged with Objective C++ and Swift to access platform-specific APIs.[3] Speech-recognition happens on-device (via the SFSpeechRecognizer API[4]).

9.2.2 World System

Briefly to give a sense of the underlying data structures for entities in the system: an entity in DrawTalking is called a *Thing*, which belongs to a *World* context. The entire system stores references to Things by ID for pointer stability. Things are typed at runtime and can refer to any user-interactive or system-level object. i.e. freehand sketch, UI element, a component in S39.1.2. Each type has a default "run" function called on the Thing per simulated-frame.

Things contain flags for setting visibility, user-interactivity permissions (whether an object can be moved or deleted), among others. All Things have default and dynamically-attachable event-handlers for touch/pen interactions.

Things have standard representation data including pose and model transforms, and in our case, storage for polycurves (points, colors, widths, etc. as separate buffers). Things have type-associated data storage for persistent state that are initialized when the runtime sets a Thing's type. e.g. "velocity," "rotation_direction." The data are decoupled so different versions can be swapped-in or rolled-back.

[2] This is the best iPad model as of 2023 that supports stylus-hover operations

[3] Although platform-agnostic code was not required, we wanted to write most of the code in a portable-enough systems language to moving to another platform in the future would not be as prohibitive. Targeting wasm for web platforms is a possibility if we re-implement the "platform layer" — the Objective C++ and Swift for different devices.

[4] SFSpeechRecognizer in on-device mode as of 2023 auto-times-out a task and resets the results after a couple seconds of silence, but we want continuous speech recognition for the user to speak a command at their own pace. To simulate continuous speech recognition, we just start the task again and concatenate the previous incomplete input to the new results.

Things store all language info as properties using a unified query library that performs efficient lookups across many properties and facilitates creation of rules. We used open-source library for such a data structure called "flecs."

The system has basic collision entry, exit, and overlap checks using colliders, which are attached to Things dynamically.

Things can also send messages to each other similarly to a naive version of Erlang's message queue system or Smalltalk's messages. Or they can defer operations as callbacks to be executed at a specific point in the system's frame loop in the future (via poll queues). (e.g. it's important to "schedule" the deletion of an object so it occurs before or after the main execution loop, which is where we assume the world is stable.)

Things can also call a "getter/setter" on other Things with parameters without knowing the specifics of the other Thing's type. This is often how the system achieves offsets for actions such as "jump onto." The source object calls the getter for position with a "spatial_above" argument to get the position with the correct offset. The object being requested handles for itself how to get that value.

In light of the fact that these underlying elements of the system are not specifically a contribution, we won't dive more into specifics. For those curious, we've included some of the actual code definitions, here: A.5 in the event they help clarify the rough structure of how Things are evaluated.

9.2.3 Language Data Structure Transformations

9.2.3.1 Transforming S1 into Incomplete S2

S1 is Spacy's dependency tree of word tokens and dependencies, along with coreference pointers to resolve pronouns. This is stored as a list of stable pointers. We transform S1 into S2 with recursive descent starting at the root token (usually the verb). This behaves like any classic depth-first programming language compiler. The routine constructs S2 depth-first, branching into different parts of the tree, and matching token dependencies and parts-of-speech. This "compiler" stage is entirely hand-written. The implementation does not matter as long as S1 (whatever NLP library) can get to the simplified S2 form reliably. For the vast majority of sentence structures we've tried and demonstrated, our compiler works. Additionally, if some word dependency patterns aren't handled, they are ignored rather than treated as an error case. This is often a good thing because some natural language includes words that do not contribute to a meaningful command, so the user has some leeway to speak in a less structured way and still get a working command.

9.2.3.2 Transforming Incomplete S2 into Complete S2

As mentioned, S2 is completed by incorporating deixis and direct labeling to fill-in ID placeholders in S2 for things the user referred to. The system depth-first-traverses S2 to do this, handling each type of element based on its type13. The noun-like elements are where the routine tries matching entities or types of entities. In this same step for completing S2, the routine constructs a user interface for the semantics diagram7.10. At the end of this process, the system has a completed S2, which is placed into a pending state for the user to edit (optionally), then confirm or discard. If confirmed, the system feeds S2 to the execution engine (the world's animation/simulation system) to generate and run the final command.

9.2.4 Execution Engine

We implemented our own execution engine for DrawTalking called "Make The Thing." Note that it represents one specialized instance of a solution and could potentially be replaced by any engine that understands how to use the more generic S29.1.2. That said, to give a sense of how the system as it is works, we've included this section for the documentation factor.

The system runs an execution and simulation loop over all Things per-timestep. It's not exposed to the user in any way[5]. A collision detection and physics update step is performed, all rules and scripts are evaluated (see "process_scripts" and "process_things" here: A.5), and all Things' transforms are updated and all Things are prepared for rendering . This includes visual user-interactive objects like freehand sketches and UI (designated as "actors"); rules; animation building blocks like move, rotate, etc; and system-level building-blocks for that represent "instructions" such as starting a loop, starting a script, waiting for the latest scripts to finish, or returning a completion status to end a script.

Each Thing is actually an invisible "node" in an evaluation graph (similar to a visual blocks-based languages) with in/out data flow ports. S39.1.2 structures are actually compositions of these world graphs. The world graphs are runtime "programs" run in a vm/stack-machine-like frame loop. The root graph comprises actors, and all other behind-the-scenes Things (instructions) run in sub-graphs. This "everything is a Thing" approach is to keep the ID system unified across all elements in the entire system.

A verb actually refers to a script, which is is a blueprint responsible for selecting and initializing an instance for a script containing its own evaluation graph "program." Initialization involves selecting the right template based on the verb arguments (e.g. "jump," "jump on") and filling a lookup-table of variables (e.g. source, target, height, speed) from S29.1.2. It then generates an instance of a script and evaluation graph for that script instance. The evaluation graph might define how to achieve a "jump" or "follow" animation. An evaluation graph is roughly a list of primitive Things (instructions), port connections between instructions, and events that should occur. (The graph execution order is defined by explicit ordering of the Thing instructions, followed by a topological sort on the in/out ports. This is to keep execution order/behavior deterministic, as otherwise loops in the ports might cause unpredictable behavior, as is the case in many blocks languages that don't have explicit execution order.) Then, the system schedules the script in-

[5] but could represent an easy first step to open-up more advanced live editing for programmers, but this was out-of-scope.

stance to run on the next simulation step. The instance has a status for run state representing "not begun," "running," "canceled," "terminated."

The world evaluation graph structure is essentially S39.1.2 and a script is the glue code and container that initializes it from S2. S3 is fully-generated at runtime for each command.

The script evaluation step is based on a hierarchy of task queues used by something close to stack machine (all top-level actors are run last, separately). When a script instance is added to the queue at the root level, it will be placed on top of the stack, given its lookup table of arguments (accessible by every instruction in the evaluation graph for that instance script), and run immediately (this triggers a "begin event" for a verb to inform rules). The script has an instruction idx into the evaluation graph's list of Things. By default, most Things return a status of "proceed," which will move to the next instruction after the current instruction is done. So, the current Thing instruction is run, the instruction index is moved, and then the next instruction is run, and so on, until a different status is returned. E.g. a "terminate" status, which ends the script entirely (and fires an end event). The script can also be set to a terminate status from the outside. Other status types might tell the machine to suspend and wait to continue next frame, or to jump to a different instruction. (So infinite animations will just keep running). Loops are achieved with begin/end Thing instructions that push/pop to the stack and check whether repetitions are necessary. (begin/end refer to each other and keep loop counter states)

Within a script, there is often a Thing instruction with the type "Call" that references a different script to call, followed with another instruction called "Wait." Call will spawn and initialize a new script instance, parent it to the current running script instance, and schedule it to run on the root task queue. This new script is also added to a local pending-task queue that belongs to the parent script. The parent script moves to "wait." Wait checks the local queue to see if all scripts within it have completed. If not, it suspends the parent script, which blocks it until the next simulation step. Otherwise, if all scripts within the local queue have completed, the wait instruction returns a "proceed" status, which moves the parent to the next Thing instruction as usual. Conjunctions are actually implemented as an uninterrupted sequence of Call Thing instructions (to spawn multiple scripts at once). Sequences (and then) are implemented as nested blocks of "Wait"s. Parallel sequences e.g. ("X happens and then Y happens AND A happens and then B happens AND"...) are scripts with multiple instruction pointers. i.e. the stack is really a stack of contexts containing usually 1, but possibly more independent instruction indices evaluating the same script. Multiple instructions running on the same script necessitates storing the data for each instruction state separately, or else side-effects might affect the wrong arguments. This is why we decouple the data for each Thing. The machine saves and restores copies of the data layout for each instruction state, so only the currently-running instruction modifies its data. This is absolutely an area for improvement, as more complicated engines might try to parallelize execution. This is doable by restructuring the engine, but is out-of-scope for this project since performance was acceptable for our examples, and the current engine was straightforward-enough to develop and test.

Adjectives/property values are pulled dynamically from Thing instructions to update arguments to the script (i.e. looking at the source and target Things to animate, pulling their properties' labels, mapping them to values)

9.2.4.1 Client-Server Architecture

The language data-flow(9.1) is achieved through a client-server approach([6]. DrawTalking the application runs on the iPad and communicates with a local server (2021 macbook pro 16" M1 Max - model MacBookPro18,2 - macOS 14). (9.3 describes the data flow more abstractly. 9.4 describes the system implementation more concretely.)

Language text along with an ID for the current unstaged command is sent to the serverside continuously as the user speaks, or after they type (via TCP for ordering — the performance is good enough for now). This includes before the user has decided to confirm a command so

[6] since most NLP is available only through Python-based libraries

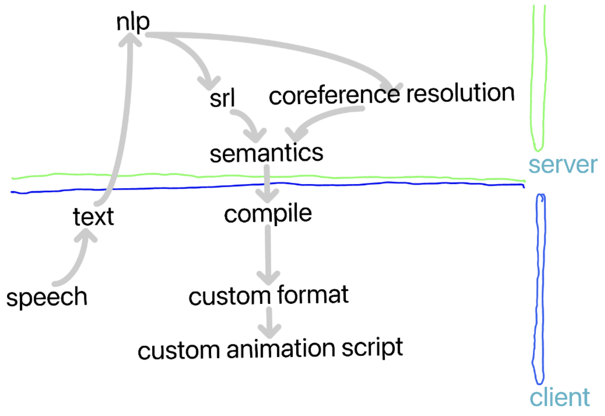


Figure 9.3: Language Processing Data-Flow

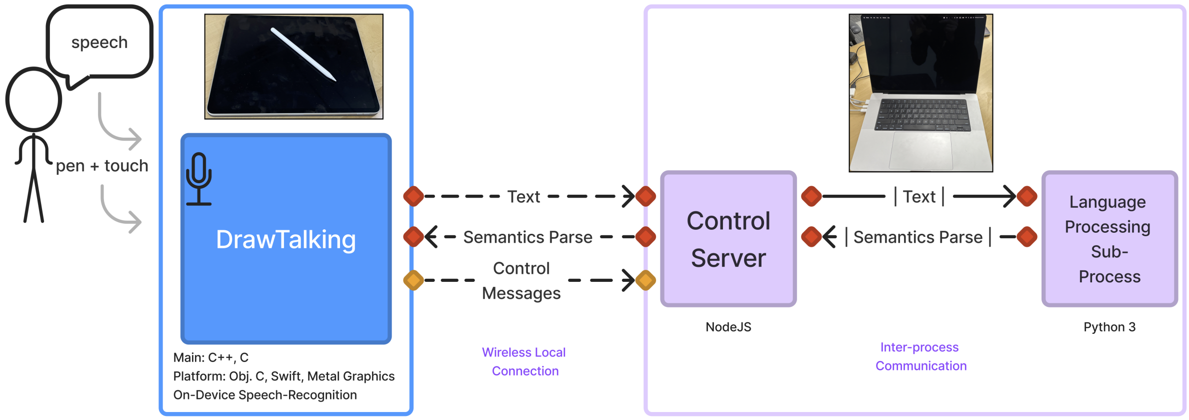

speech
pen + touch
DrawTalking
Main: C++, C
Platform: Obj. C, Swift, Metal Graphics
On-Device Speech-Recognition
Text
Semantics Parse
Control Messages
Wireless Local Connection
Control Server
NodeJS
| Text |
| Semantics Parse |
Inter-process Communication
Language Processing Sub-Process
Python 3

Figure 9.4: Concrete Details for Client/Server Text data are sent to a server for natural language processing over a data channel and forwarded to an NLP-focused sub-process on the same machine The semantic parses return to the client. A command channel handles canceling parses and other events independently from the NLP.

the data are ready as soon as possible, or for functionality that can use the live speech without confirmation. For example, deixis matching is one case in which the user does not need to confirm.

The serverside runs natural language processing on the text on-receipt

(via Spacy [Montani et al. 2023] to get dependency parsing structure and coreference resolution, as well as NLTK+Wordnet [Loper and Bird 2002] for finding synonyms to suggest alternative verbs). This generates S19.1.2, which is sent pack to the client in JSON format.

Note that as technology evolves, we might use entirely different algorithms[7], and better on-device approaches might become available that don't necessitate offloading to another machine.

Upon receipt of S1, client processes S1 into S29.1.2[8]. The system updates the *transcript view* (to enable up-to-date matching by linking), runs deixis matching checks, and saves S2 to await user commands. If a new S1 arrives for a given command, previous ones are ignored.

If the user presses the discard button, 7.5.1.1 a control message is sent to the serverside on another port that cancels unfinished NLP for a stale command and prevents it from returning to the client. This is necessary to prevent the server from falling behind on outdated data.

9.2.4.2 Rendering

We wrote a custom renderer with a simple abstraction layer API around Metal, the GPU API for Apple devices. We did this to have the most control over the pipeline for render passes, composition, and effects, and the least bloat during development. The abstraction layer allows for possible changes in the platform and creativity with the API.

9.2.4.3 Other Explorations in Implementation That Showed Potential

The keyboard input field in DrawTalking also serves as a command-line prompt for developer commands, which could help expert users program or customize the environment. We used this to develop ideas faster or create test cases. Example commands ranged from creating entities with a specific type and position quickly to setting variables directly to enabling/disabling a dark mode. These might all be desirable things for a power user who is a programmer.

We also created a parallel version of DrawTalking for macOS using the same project code with minor platform-specific input and API changes. The experience is not optimized for a system without a pen and multi-touch. However, desktop supports access to the entire system and makes more interesting capabilities easier to implement.

[7] but e.g. LLMs cannot yet achieve as fast a response

[8] in another thread so as not to bottleneck the application user input, simulation, and rendering.

For example, recompilation and reloading of code as dynamic libraries has been implemented. This enables users (who are comfortable as programmers) to extend functionality at run-time — i.e. create new verbs and interpreters or pull-in any functionality from other applications.

We also support texture loading from disk (on iPad and macOS). Although this feature might be expected in production-ready systems, we found the core of this project was the control + rough sketching aspect, and not the content. Loading of images proved to be a distraction from that interactivity focus. However, we tried this functionality, and it works. This includes loading an image from a URL or REST service such as a generative AI. Loading images, definitely can help supplement the content the user can create themselves and fit into a more feature-complete version of the application.

Additionally, in an early tests to see how multiuser collaboration could work, we tried connecting an iPad running DrawTalking to our mac running DrawTalking. The mac version treated the iPad as a camera feed, and we displayed the iPad's view onto an object on the mac's side so the mac's scene contained the iPad's scene within a sketched object. The iPad miniature view could feasibly become a "portal" for users to move between each other's scenes, watch as audiences, or reach-in remotely to help with minor changes. A regular camera feed from any device could work, which means DrawTalking could also implement functionality from video-conferencing or a streaming solution.

9.3 Afterword on Implementation

We've implemented DrawTalking around the idea of decoupling the concept of our approach from the specific details of the implementation. We believe that by thinking in terms of translation layers, we've shown how future advances in technology can improve the pipeline in terms of speed and performance, independently from the approach. For example, we decided to use a fully-rule based NLP library rather than a large model to favor interactive time speed and fluidity over naturalness of speech input; the loss in interactive time hurts the experience too much. It is faster by an expert user (i.e. us) to edit a natural sentence into a system-understood one, in most cases. However, large-language models, once faster or used in creative combinations with other methods, are a good option. They can feasibly translate from natural language into an intermediate format like ours and skip S19.1.2 directly to S29.1.2, or they can reduce a natural language to only system-handled versions of S1. Then the same applications like DrawTalking could use their own interpreters of the deterministic input. Or other applications could reuse other deterministic interpreters without coupling with the technology. We think of DrawTalking's language system as a proof-of-concept that shows this kind of decoupling could allow for improvements to the experience without affecting the core approach. It allows for parallel tracks of development and research without establishing hard-dependencies.

10 | User Feedback

10.1 Study Design Motivation

In designing a study, we wanted to learn about the participant's exploration process and impressions as a first-time user over a short-time period. We are starting from our experimental design, which fulfills our specialized design goals and criteria towards a vision for future interfaces. Our focus was **not** on comparing the system in terms of performance, as there isn't a specific use case we are evaluating in this stage. We are also aware that participants will need more time to learn to use the interface than an initial round of exposure would commit. Instead, we are interested in evaluating the feasibility of the user experience in isolation. i.e. how do users respond to *this* interactive approach?

We wanted to learn based on user feedback whether our approach would be perceived as something with potential that is acceptable and useful. Moreover, why and in what ways might people like and use the approach, from their own points-of-view?

To that end, we designed an open-ended exploration session in which the researcher introduced and guided the participant through increasingly complex features that they could try-out with their own creative direction. Discussion during the entire session would be allowed. The goal was to give the participant enough exposure to the interface to arrive at working artifacts from the process and elicit meaningful discussion. Since this would be a short exploratory session, the emphasis was not on reaching expert performance, but rather on getting the exposure.

By the end they should identify for us through longer-form discussion what the use cases and comparable workflows and tools are, from their perspective. We specifically did not prescribe specific use cases for the participant so as to bias them less.

We believe this form of verbal feedback is valuable because it can extend our own understanding of the tool with specific anecdotal and experiential feedback. In summary, we are interested in identifying use cases from the participants' perspectives. Do they perceive potential in the tool, and what is the main utility of the tool and what potential does it have for them, personally? From there, we can identify themes, directions, and additional user needs.

Some of the factors we anticipated observing and learning about were the following:

1. use cases and utility for the workflow (elicited from the participant)
2. understanding of the mechanics in a relatively-short time period
3. creativity as evidenced by explorations and artifacts from having tried the tool
4. the first-time user's exploration process
5. preferences for working with different parts of DrawTalking for tasks (e.g. labeling by speech vs labeling by linking)
6. expectations for and adaptations to language the controls
7. suggestions for improvements

— with the expectation that we'd learn additional feedback through discussion.

10.2 Study

10.2.1 Participants

We invited ten participants P1$_{expl}$-P9$_{expl}$ to an in-person study via an online form.

The form helped us select candidates with relatively-high confidence in speaking and sketching, with interest in the topic, and with a fair mix of professional/academic backgrounds and statuses. It asked candidates to state their academic and professional backgrounds and provide a short description of why sketching mattered to them (e.g. as a hobby, profession-related task, in general life, etc.). It also asked candidates to self-rate their confidence in speaking and sketching to help us decide whether they'd be comfortable using the tool.

Each participant was compensated 30 USD for their time and feedback.

The following summarizes the participants' backgrounds:

P1$_{expl}$ A very-experienced professor (at a higher-level institution) in computer science, interactive graphics, and new visual media (e.g. AR, VR storytelling and games) — focuses on teaching students new to programming. Uses many visual aids in classes.

P2$_{expl}$ Digital fine arts university student (design and computer science), drawing is a hobby. Uses sketching for game design; planning gameplay features and figuring out how they should work. e.g. diagramming and framework design. Experienced with tablet drawing.

P3$_{expl}$ Experience as a digital designer, economics and digital art university student,

not professionally-trained in art, uses a digital canvas (Procreate) to record visual ideas.

P4$_{expl}$ Tech / Design university student. Does design for web, desktop, apps in-general. Has done digital illustration and drawings often since childhood.

P5$_{expl}$ A very experienced professor in computer programming for interactive graphics. Does live-coding and streaming for internet-based education as well. Uses and authors open-source coding libraries for interactive graphics. Teaches programming for creating visuals rather than designing visuals by hand.

P6$_{expl}$ Student in interactive media. Draws and paints using the physical medium, with lifelong interest and experience. Experienced with digital graphical design work. Low experience with programming.

P7$_{expl}$ Industry creative expert. 15+ years experience as a set designer and in the theater industry. Digital illustrator, spatial experience designer, AR/VR immersive projects, projects in entertainment and fine-arts. Creative directing, collaborations with companies and academia. Non-programmer, some experience with visual blocks-based interfaces.

P8$_{expl}$ Robotics and machine-learning researcher, works with developing ML models, often needs to visualize ML inputs/outputs and illustrates rough sketches to think-through policies (for robots). No prior experience using a tablet.

P9$_{expl}$ Robotics and machine-learning PhD student, used tablet-based sketching interfaces.

10.2.2 Physical Setup

We set-up DrawTalking at a table and wanted the session to approximate side-by-side collaborative whiteboarding. The participant sat on the researcher's right facing the table, with the iPad and pencil in-front of them, within the view of the researcher. The server (macbook) was placed in-front of the researcher. Screen and microphone recording of the session would be done on the iPad on-device. The researcher had a DrawTaking Info sheet on the table for reference to show the participant available features to try (after the training phase).

10.2.3 Procedure

We conducted a semi-structured study of 1-1.5 hours. It involved 4 components: 1) A brief introduction of the study by the researcher and a self-introduction by the participant; 2) a training phase to acquaint the participant with the basic drawing-specific interface elements 3) integrating the full speech-input; an open-ended collaborative session between the researcher and participant exploring system features with increasing complexity, towards eliciting discussion about the user experience. The expectation was for the participant to become well-enough acquainted with the system (not necessary achieve mastery). The participant could ask questions, interject with thoughts and feedback, and try-out features. The exact content was improvised according to the participant's initial drawings and explorations of a feature list, with some structure imposed by the researcher to help the participant explore the the system; 4) a discussion-only segment with the participant for more focused feedback on the full experience.

The study was divided into the following sections, for a total of roughly 60-80 minutes:

10.2.3.1 Introduction *(~5 minutes)*

First, the researcher introduced the purpose of the study as an exploration of new sketching and speaking interactions for tasks such as interactive animations. No specific use cases were defined in an attempt to keep the participant open-minded. To make the participant comfortable with the study, the researcher emphasized that we were *not* measuring for skill or performance, but rather we were looking for feedback on the experience. Lastly, The participant was asked to give a brief overview of their background and experience.

10.2.3.2 Drawing Training *(~10 minutes)*

The researcher explained and demonstrated exclusively the drawing/canvas controls, then invited the participant to try the controls until they were visibly comfortable with the controls.

The researcher asked the participant to draw 5+ objects (characters, scenery, props) they might wish to interact with in the scene.

10.2.3.3 Collaborative Exploration *(~30-45 minutes)*

The researcher and participant explored all of DrawTalking's features collaboratively, improvising based on the drawings the participant created in the scene. The participant was able to think-aloud throughout the entire section and interject with questions and comments. First, the researcher demonstrated both of the workflows for labeling, using a variant of a basic command such as "jump on" or "move to as an example." The participant would be asked to repeat the same command using other objects. Afterwards, the researcher would progressively introduce more complex features (conjunctions, sequences, loops) ending with rules. After each feature, the participant would be asked to do similar commands with the freedom to look at a paper manual for inspiration. If the participant had an idea they wanted to try that diverged from this pattern, they were allowed to explore. The researcher would alternately suggest some ideas or help the participant reason-through how to achieve a given effect using the controls. The researcher was allowed to assist in correcting for speech/recognition/language errors, so long as the participant could demonstrably perform the same operations. The expectation for this study was not mastery, but rather understanding and comfort with the controls. The ideas was to evaluate the concept qualitatively, so the quality of language processing technology should not be a blocking point for the participant.

10.2.3.4 Post-Discussion *(flexible to end)*

After the participant created working commands using rules and demonstrated a fair understanding, the session naturally transitioned into a semi-structured discussion of the experience. The researcher asked specifically 1) which labeling technique the participant prefers (linking or deixis); and broadly 2) how the participant envisions using the techniques in their own work/life;

3) what use cases and scenarios the participant envisions. Beyond these, the researcher would continue the discussion open-endedly. Lastly, the researcher would show prerecorded videos of demos made in DrawTalking or demonstrate additional more advanced examples to elicit more feedback (e.g. windmills, game control buttons and UI, characters collecting coins for points, etc.).

After all sessions, we collected screen-shots and feedback (comments) according to themes related to 10.1.

11 | Results

Each participant learned to use DrawTalking's mechanics (from simple verb commands to rules) and developed a strong understanding of the ideas behind the tool during the session. Participants were visibly excited and offered invaluable feedback in defining the strengths, use cases, potential, and future directions for the approach.

Participants collectively identified and discussed related use cases including educational applications, videogame prototyping and paper prototyping, user interface design, presentations, language-learning, and visual-oriented programming. All participants likened using the interface to a form of play or rapid prototyping.

11.1 Participant Discussions

11.1.1 P1$_{expl}$

P1$_{expl}$ notes how working with DrawTalking is like using language with the feel of building blocks to program a scene. It has potential for creating interactive games and could be complemented by new language technologies to make it even more expressive. **P1$_{expl}$**: *It's like **language and Legos put-together.***

You could totally make something for kids to make their own videogames and build a videogame creator. That would be totally cool.

Right now the grammar's structured. You have to conform. I wonder if you could use a language model to parse human speech and transform it into the desired grammar. So I could speak more conversationally. That might free people from having to learn the specific grammar. Potentially lower the bar of entry.

I think it's awesome the way it is.

If you build this as a basic generic toolset that can handle basic actions, the artwork generation, and the exporting options – I think it could be a good engine (especially if you throw LLMs in the front-end.) You could sit here and have a conversation and build up just using language your interactions and then you send that out for code. That'd be fun. I'd do this all day.

Asked about additional use cases,

P1$_{expl}$ suggested using DrawTalking as a way to play with **digital manipulatives** to explain mathematical or physics concepts, citing the windmill demo as a good example. Furthermore,

P1$_{expl}$: *"I can completely see you using this to teach English as a Second Language (ESL). Learn vocabulary, practice. From a reading perspective for kids whose native language is not English [it is useful] to have the names of the objects below the objects. "*

11.1.2 P2_{expl}

P2_{expl}, as someone who taught kids programming via Scratch[Resnick et al. 2009], reflects on how DrawTalking compares as a visual-oriented programming language and provides several parallels with their own practices.

P2_{expl}: *When we make games ... there's something called a* ***paper prototype*** *where we make a bunch of pieces of the players and the objects.* ***We just move it around by hand to kind of simulate what it would be.*** *[DrawTalking] is kind of like that but on steroids a bit. So it's very nice to be able to kind of have those tools to* ***help you with it without needing to manually make it.***

P2_{expl}: *It's like using Scratch I used to teach Scratch to middle scholars while I was in high school and it's kind of the same experience of figuring out like all these these — 'this is an if statement, this is like a for loop, I can make this condition go here.' This really is like a coding language.*

When asked "how is it different:"

P2_{expl}: *Both a bit limited* ***and*** *more flexible. It's flexible because you can kind of make whatever and using your speech patterns makes it more flexible even if there's restraint on the way you speak. It feels a lot more freeing that I can kind of just make game objects at whenever I want. I can make them look however I want, and it's very easy to do that.*
I think in Scratch you have to do sprites and then import them in, put them in the game and then attach code to them.
The thing with the inflexibility right now might be due to the limited vocabulary. There's still a lot you can do with this, but some of it feels a little not close to how we'd normally talk.
Specifying the subject again it makes sense but is obviously a limitation.
The more I use it the more comfortable I would get with it. It's just like a new programming language.
You can definitely make a game wireframe using this. *Going back to the paper prototype. Sometimes when we're testing we playtest the paper prototype instead of actually having a program.*

And we'll give players a keypad. Like paper arrows that they press. and we'll move the stuff for them. This [the static on-screen programmable buttons] is kind of that but it skips the middleman.

When asked how they'd use the tool:

P2$_{expl}$: ***[I'd] probably use it to make little prototypes*** *(used to mess around with Scratch). Making tiny little games to see if concepts work. (Referring to a boss they're currently programming for a videogame): Would probably interesting to draw a little boss and program him to move around and react to stuff in his environment. Test edge cases.*
The logic flows pretty similarly to what I'm currently using, which is Unity.
It's good to figure out what the interactions I want would be.

When asked how the interactions might apply to other tools:

P2$_{expl}$: *[They'd help me]* ***find things.*** *(Referring to the ability to warp to objects by name) Unity, Maya, Z-brush. Their UIs are dense and difficult to work with. So it [the mapping] might be helpful to find stuff. Like windows or find objects if you've lost them in a really big scene. This was pretty interesting that you thought to add this [find features]. Some software doesn't have find option in their menus. It was still easy to use to tap and hit find.* ***This would've solved a lot of issues when I was first learning to develop.*** *When I draw I would show people my ideas with Procreate.* ***Procreate is manual (e.g. using lasso) and can't control things in tandem, which this [DrawTalking] solves. If I had to make interactions happen I would have to do slowly step-by-step. If I had to do the same thing (werewolf sun/moon demo) I'd have to draw it so this really ups the efficiency.*** *Also because of the way the system is implemented, there's things that happen that you don't expect that are a result of a computer program, which is actual useful for me because a lot of things can go wrong when I'm programming systems (edge cases) but seeing them happen with the logic / rules I program into this software can make me account for those things more easily and make me think about those things.*
Old Scratch games did not have actual interaction like a videogame. e.g. you walk down this path. premade events. This could lead to a lot more interactive storytelling whereas most stories that we

think of are down a straight path. This could lead to alternative paths. Also unexpected things because of the interactions you've created. I definitely could see interesting things being made in terms of narrative for this application.

P2$_{expl}$ overall likened DrawTalking to an interactive game and storytelling environment with useful automation and simple-to-understand controls and UI. They draw strong parallels with the visual programming environment Scratch, where the speech controls are the main differentiator — easier despite restrictions on the language structure because DrawTalking requires far fewer steps to do certain operations and solves issues related to multitasking.

11.1.3 P3$_{expl}$

P3$_{expl}$ found the interface straightforward and felt it was fast in comparison to other creative applications they'd used. He also likened DrawTalking directly to learning programming and finding several creative possibilities after the initial steps in this kind of learning process.

P3$_{expl}$: *Learning this reminded me of learning how to program. When you learn a new term/new language. You kind of forget what you learned before so you need to remind yourself of what came before. Once you learn all the features and use them for a couple of times, you get pretty good at it. You remember everything you keep coming up with new ideas.*
At the beginning it takes time to remember the things, but once you learn all of them they keep inspiring you to come up with new ideas.

P3$_{expl}$: *"This could be a brief storyboard for game design."*

P3$_{expl}$ describes DrawTalking as adaptable to other applications. It could be a potential plug-in for other applications that could use its mix of language and direct interactivity features, especially if it were to become more mature. It could integrate with other AI.

P3$_{expl}$: *This is kind of like... In Adobe AfterEffects[video creation software] if you were to move something. you have to do a lot of things. set a keyframe. set the way it's going to go.But in here, it just takes one second for me to "say it" and it's going to do it. I definitely can see this being used in*

creative software that you can just say things and it's going to do it for you.
When there's AI, now you can select the area and say "remove the scarf." I can see this[DrawTalking] being similar. built-in function for any tool interactive AI
The interaction of this does things other AI cannot do. (direct interactivity) I see this as a builtin function in basically any tool.
I can see this replacing other AI that use prompts
If this is more mature. I definitely see this easier to use than other AIs.

P3$_{expl}$ also perceived DrawTalking's workflow as fluid because it successfully supported multitasking between drawing and definition by pen, touch, and speech:

P3$_{expl}$: *I was also very impressed by how while you are still drawing you can say out loud and just name it. I'm surprised by how these two things can happen at the same time. // ... this is a smooth experience that when I'm drawing that I say something and then it's made.*

P3$_{expl}$ describes DrawTalking's workflow as quicker and easier than other things they'd used in the past.

11.1.4 P4$_{expl}$

P4$_{expl}$ felt very creatively-engaged with the tool. Like others, they needed time to understand the possibilities and encountered a learning process since to them, the bare-bones UI was not self-revealing. However, once they got acquainted, they immediately started considering the storytelling and animation potential. This shows a trade-off between complexity/obviousness in the UI and and the learning curve. Also like others, P4$_{expl}$ compared DrawTalking with Scratch in terms of how they thought about problems. **P4$_{expl}$**: *While I was making this I definitely felt very immersed and* ***I was thinking about a whole story in my head and all the different things that I could do with this program.*** *// The thing that limited me the most was the unfamiliarity with the controls so it does take some getting used to and I think that maybe having a more developed UI might be part of it since right now it's kind of bare-bones.* ***But I think that once you get a***

hang of the mechanisms, there's definitely a lot of fun things you can do with it such as telling/animating a story from a storybook, for example. // I think it's a very good tool for exploring your imagination. // I'm thinking about this in a very programming/Scratch way.

When asked about other use cases:

P4$_{expl}$: *I imagine this could be good for making a dress-up game.*
A lot of this is kind of more in the educational classroom in terms of use cases. ... demonstrate visual interaction to other people. I would only do that in an educational setting.
This is good for kids in elementary school for them to practice in a computer class (or something) and I think that in that case it would be really important to make sure that they know how to use the program and to introduce them to the terms (key words) slowly, as well as demonstrating.
When I was first doing it, it was a bit of a slow process. I'm also curious if there's a way you can make it so that there's a simpler, straightforward way. it seems there are multiple ways to name.
I think the the linking/tap difference was confusing sometimes.
As a beginner maybe I only want to be introduced to certain features.

Specifically, P4$_{expl}$ is suggesting that DrawTalking introduce fewer features more slowly, or in a way integrated into the application. We agree that a short 1-hour study cannot easily cover all features in an ideal pace. P4$_{expl}$ again emphasizes that once they had enough practice, they were able to see several use cases within education (e.g. for kids), videogames, and storytelling.

P4$_{expl}$ goes on:

P4$_{expl}$: *When I'm working in an art file with lots of layers. "This layer is this/that layer is that" for efficiency's sake a good tool I can use.*
Presentation [of a] slide deck. Group certain things. e.g. "this is the text. this is the graphic."

where the use cases move more into application-level control and productivity.

Overall, P4$_{expl}$ emphasizes a positive experience and was able to learn and understand the system. As with many new tools and programming languages, DrawTalking is new/unfamiliar enough to require some time to learn, but from observation we found that the hour-long session was about enough.

11.1.5 P5$_{expl}$

P5$_{expl}$ is excited by DrawTalking being used as a coding playground.

P5$_{expl}$: *I could imagine playing with this for a really long time and enjoying exploring it.*
I just want to play with this. drawing and animating and thinking about rules. It's like a toy.
This is like a natural language way of adding logic and code to a scene.
What's kind of cool about this beyond drawing an interface and using it – I can execute different rules of things that are happening. scene is playing out here AND physical thing is happening.

On the topic of being used for presentation:

5: *"[It could be used for] live explanation [and] presentation about the basics of foundational game design concepts."*, suggesting it would be exciting to involve the audience as participants in the talk by asking them to speak to the tool. e.g. a *"room of 5th graders. get them to make fun animations and interactions."*

When asked about how DrawTalking might integrate into the participant's work:

5: *If I wanted to use this as complementary in teaching e.g programming in P5.js*[1]*, how would it fit-in? Lot of value here in oh let's look at what is a conditional statement? ... let's look at this tool where we can build these objects in and try using natural language to build these rules and see what it does.*
and now I could look at how text-based coding does the same thing. This might help students be creative and think-through ideas and understand the concepts.
Then it could apply into their learning to code.
Mixing visual elements with language helps me understand it. more of that could be good.
drawing connections between visually as I'm speaking.

[1] JavaScript graphics programming framework

There, P5$_{expl}$ describes how DrawTalking could be used directly as a visual environment for learning programming directly. They go on to suggest that an evolution of the DrawTalking interface could convert user interactions and content sketches into actual code or a webpage. e.g. "Export it to P5.js code: e.g. a 'star class' with a point [x,y] with rules." We believe this is a strong direction to avoid lock-in with a particular application of DrawTalking if it were developed further. We agree that a promising direction might be to treat the interface more as a development phase with the ability to export to traditional code and code generation.

When asked about more use cases and potentially extending beyond sketching:

P5$_{expl}$: *[A] tool to actually design an animation. build whole set of rules and drawings I could export and maybe it would be a game.*
UI elements make a lot of sense
audio/webcam support?
This interface is a starting point for developing all of these ideas [for speech and drawing control]. This could be refined or placed into other contexts. [It's a] rapid prototyping environment.
Connect sketches to a controller for an LED panel? Physical/tangible interfaces.

P5$_{expl}$ also suggested additional domains and research directions, with which we agree.

P5$_{expl}$: *e.g. [Consider] this is a framework. Build on top of it for specific applications.*
e.g. Focus [it] for particular context.
e.g. Keep as-is but refine (is there a version without need for paper printout with just self-guided)

Lastly, P5$_{expl}$ is curious what a truly UI-less version of this tool is, which we believe to be a valid future research direction.

11.1.6 P6$_{expl}$

P6$_{expl}$ praised the sense of playfulness and felt that it was *fun discovering different actions or different code that [they] could write with the speech and drawing. [The experience was] visual and tactile.* (Again we see the participant likening DrawTalking to a form of coding.) They describe

DrawTalking as accessible to non-programmers. **If you don't know the language of code, you can just speak English ... create actions that will animate something.** Additionally, they felt the direct integration of drawing was in some sense more accessible than using code because it's *hard to draw a shape in JS exactly as you envision it, but here you can draw an object and give it actions. [For] code, [you] have to write every single detail into it.* i.e. the participant prefers directly drawing what they want to using procedural code.

P6$_{expl}$ appreciated the speed of speech as opposed to text in spite of additional potential for error, and also appreciated the availability of the keyboard to correct words. (We showed this feature.)

In terms of use cases: *Game design. Roughly flesh-out a scene. Sketch-out how everything would move around. You could do it very quickly here.* Also, *museum interactive exhibits., presentations*

P6$_{expl}$ also liked the ability to create buttons to perform actions. *[I] felt the buttons were useful so you didn't need to redo the actions over and over. [I] felt more in-control. If giving a presentation, [I] might make some buttons in-advance*

This highlights an interesting flexibility in DrawTalking: **the ability to create some elements in advance and some immediately as-needed.**

11.1.7 P7$_{expl}$

P7$_{expl}$ as a professional artist thinks DrawTalking is a step towards helping non-programmers build game worlds intuitively.

P7$_{expl}$: *I can see it in the realm of a game engine.*
Me instructing a game engine – someone like me who's not a programmer and who is intimidated by doing C# or even Unreal Blueprints and using visual scripting language like blueprints – this is a really clean interface that I think can achieve you can get 90- or 80% the way there by instructing say selecting a character – saying I want this character to limp or if I'm doing an animation to go to the top of a mountain and I want it to be slower or faster. It just makes the user experience cleaner than

having to use all these knobs and buttons or things like that or using scripting language or having to actually write code.

You [the researcher] do not force to work with the scripting language. You can get close to what you want without actually having to do visual scripting or programming.

You don't want friction. Points to a bigger thing. A tool should be intuitive that anyone can pick it up and use it to some extent. Like you can pick up a pencil. Hammer and nail. Without teaching. Not all tools are like that. A car is intuitive to use and very complex. We're all trying to get to that.

I look at it from the point of view of someone who uses Unreal and it really frustrates me to navigate that interface. So the possibility of something where I can speak to it and have a conversation with it and it can do what I want it to do is really enticing.

P7$_{expl}$ sees DrawTalking's approach as a potential bridge between art and programming for non-programmers. They describe that it has less friction than traditional visual interfaces. The main mode of interaction is speech, the participant believes the direction of using speech as the main way to describe intent to the machine as a promising path forward for addressing common frustrations.

P7$_{expl}$ pointed-out that in the future they'd like to see a version of DrawTalking that required no learning curve:

A tool should be intuitive so that anyone can pick it up and use it to some extent. Like you can pick up a pencil. Hammer and nail. Without teaching. Not all tools are like that. A car is intuitive to use and very complex. We're all trying to get to that.

This would be a worthy direction. In-large part the learning curve (we observed) came from the limitations in vocabulary, which is tied to the language processing component. However, as P7$_{expl}$ says, not all tools are instantly-learnable. For example, language is learned gradually over time. We believe it makes sense that given the comparison with programming languages, DrawTalking might require some exploration over time since it does not map 1-1 to full natural language. Based on the current approach and as mentioned, it might be possible to support more natural forms of speech, thereby reducing the learning time.

However, we still need to account for learning how to use language in new contexts. It is unclear whether we can simply perfect an interface by supporting greater language flexibility. As P4$_{expl}$ said, increasing the feature set and level of flexibility might have the inverse effect of making the interface harder to learn. P7$_{expl}$ touches-upon big open-questions in interface design — how to balance between making interfaces self-revealing and simple, and making them powerful, but potentially more complex and challenging to learn?

11.1.8 P8$_{expl}$

P8$_{expl}$, a robotics and machine learning researcher, expressed that DrawTalking could easily help create animations and presentations very easily **P8**$_{expl}$: *"If I was an educator, lecturer, this would be an amazing tool to explain things to people".* More specific to their work, the sketching capabilities could help in their process of building and trying-out machine learning models for directing robots in manual tasks. P8$_{expl}$ felt the ability to move boxes (sketches) around anywhere, name them, and roughly sketch mathematical equations on them, was already compelling due to the convenience of spatial organization and the ease of tagging sketches.

Additionally, P8$_{expl}$ clearly could see how one could create a videogame e.g. a racing game.

When asked more directly about applicability to their work, P8$_{expl}$ said more domain-specific grammars would be necessary for working with finite state machines (FSM)s. Additionally, having more granular control over the definition of walls and obstacles, paired with pathfinding algorithms could make DrawTalking usable to help with reinforcement learning-based obstacle avoidance. If such additions to DrawTalking's featureset and DSL were made-available, then this *could be used for animations for debugging possible problems [in robotics].* i.e. **DrawTalking could be a visual debugger for real computer programs.**

P8$_{expl}$ goes on: *For long animations could be complicated, but for first sketches for the system, could be useful for picking-up [detecting] possible problems [with the models / algorithms].* . This is understandable, as the current version of DrawTalking does not include all production-level features for saving and managing animations, but this could be integrated. The question would be — what is the best way for saving much more complex state beyond singular rules. P8$_{expl}$ suggests that simply extending the rules view to visualize and enable re-editing of dependencies between rules might be enough. We find this idea of treating DrawTalking as a simulation and controller of external processes and rules as an exciting possible direction.

11.1.9 P9$_{expl}$

P9$_{expl}$ is a robotics and machine learning PhD student. They saw the potential DrawTalking as a step in the iteration process towards creating videos or animations. For example, creating and performing animations (even simple movements), and recording them for composition in video. **9**: *"If I want in my presentation to show something ... there aren't going to be videos on the internet for that. With this[DrawTalking] just by recording the screen I can literally make a video in two minutes."*

Secondly, P9$_{expl}$ felt DrawTalking would also be useful as a form of game — a way to design and play games with friends as part of a social experience. e.g. to create "random figures, stories, see what happens.

P9$_{expl}$ discussed avenues for integrating DrawTalking's concepts into other contexts: *this idea can be extended to all kinds of [things] with more actions, movements — if for some game development where they already have the gadgets set-up. Just incorporate the language and control interface here. We can easily create animations with the fancy stuff they have.*

—indicating that they considered DrawTalking to serve as generic functionality.

When asked about possibly incorporating into robotics (their line of study), P9$_{expl}$ agreed, but with the caveat that for real-world environments, they might want to have even more control and specification of commands for safety at the expense of automation:

For potential tangible *interfaces, physical things like robots might need more specification (for* *safety? accuracy?). Would be more tolerant of extra specification to make sure it's correct. Definitely don't want surprises. People won't want it to be fully magical. [They want] more control.*

This suggests that our emphasis on user control was necessary, and that it likely requires further development for higher-risk contexts where safety is crucial. It also suggests that there is a use case for reducing automation. Not everything has to feel "magical" or instantaneous to be useful. Overall, configurability seems to be a major consideration moving forwards with this and similar interfaces.

11.2 Points / Observations

We discuss additional points and observations.

11.2.1 DrawTalking as Programming by Speech

That participants generally described DrawTalking as a form of accessible programming using language, with the ability to move things around tangibly and physically — yet we never described the interface as a "programming" interface. This indicates that participants understand the capabilities and metaphors underpinning DrawTalking. P1$_{expl}$ immediately understood that the use of "over and over" is like a while loop. P3$_{expl}$ asked whether labeling was like assigning variables. Most other participants self-described actions as "functions." P7$_{expl}$, a non-programmer familiar with some visual programming interfaces, interjected that DrawTalking is "true visual programming." Most participants became especially excited once more complex examples using repetitions and rules came into play, showing greater levels of programmability.

11.2.2 Labeling by Speech is Generally More Intuitive Than by Linking: Speech (Deixis) vs Linking

In the context of this study, participants overwhelmingly preferred labeling sketches using speech (deixis) to labeling using linking with touch and pen. All users settled on using speech after trying both methods. Participants generally reported wanting to learn one of the methods and staying consistent, but felt both methods were useful. They thought using speech + touch was more natural and direct. They appreciated pen-linking, however, because it guaranteed that labeling was possible as long as the desired word was available; it offered flexibility in the way the user could speak without being required to refer to an object directly. These guarantees and flexibility could be most useful in presentation. In pilots prior to this study, we had roughly the same feedback: speech is more intuitive, but linking might be less fatiguing and more useful when precision is required over a longer period.

Note that multimodal pen+touch and linking style operations have been available and used since at least SketchPad for drawing and visual programming interfaces. The linking operation was not self-revealing in this case, but was still useful.

P1$_{expl}$: *[Linking] is harder to remember, but [I] would get used to it. [Speech] is much more direct. "boom" (taps an object) – command associated with that.*

P2$_{expl}$: *[For] talk[ing] while drawing when other people are watching, [it's] easier to say, "This is going to be a 'blank'." Linking goes another way not conventional to drawing. But it is a nice guarantee that if something goes awry you can fix it. "This is a" feels much more intuitive.* P2$_{expl}$ thinks linking might be more useful in cases the system might be more confused like with compound words. *"In most cases '**This** is a' would be better. [but] In some cases ['this'/'that'] might be too obvious. When something doesn't require explanation. (might feel a bit bad)"*

P3$_{expl}$ was "more impressed" by "this is a" compared with linking.

P4$_{expl}$ in particular much-preferred labeling by speech and suggested that linking might require greater dexterity and practice as someone unfamiliar.

I think that drawing while naming made a lot more sense to me.

I think the the linking/tap difference was confusing sometimes.

Kids will have smaller hands (multiple selection might be hard).

As a beginner maybe I only want to be introduced to certain features.

P5$_{expl}$ reported that *"it feels much more intuitive to do it this way"* using speech. However, they understood linking as useful when one might want to label without directly referencing the object. P5$_{expl}$ gave their own example, saying *I often talk about arrows in the context of vectors in my teaching* and simultaneously linking a sketch of an arrow in the scene to the word "arrows" in the *transcript view*.

P6$_{expl}$ again would use tapping more (speech + touch) because it mapped better to how they'd speak with someone, but appreciated both: *"[Speech/tapping are] more intuitive. That's how you would talk to someone else. Having both is nice as a back-up."*

P9$_{expl}$ said that different people might choose different methods based on their workflow (drawing everything first and then labeling, or drawing elements and labeling, element by element), but preferred simply picking one method (speech). Linking with the pen was less discoverable. *"Some people might talk about what they are drawing (this/that), some people might draw everything first and then label. I might prefer to draw all objects first and then label them all at once. Might not need functionality to do at the same time. Linking is less obvious, discoverable. If they[users] know, should be intuitive. Happy labeling one way. (Tap, this is a ...). Might not need to go back and forth. Because I know one way ... pretty convenient way ... to label.*

11.2.3 User Opinions on UI

11.2.3.1 Semantics Diagram

Participants understood and appreciated the semantics diagram because it provided a simple representation of their command and a view into whether the machine understood a command correctly.

P2$_{expl}$ felt it was very important, especially for beginners and offered the suggested that it might be integrated more compactly into the *transcript view* to representing larger commands and for reducing redundant information. (Although if the transcript is empty, we still need a representation.) **P2**$_{expl}$: *[The diagram] is pretty important. Especially if you're getting to used to the program, you might say something wrong and it interprets it in a different way and you can correct it that way [using the diagram]. It's pretty important. As the program becomes more nuanced and more development goes into it maybe it's not as necessary because it's already shown with the subject or the noun or the noun highlighted in different ways (e.g. transcript view).*

P5$_{expl}$ was enthusiastic about how the diagram annotated their input and helped them understand the relationships between their visuals and speech. *"Mixing visual elements with language helps me understand it. More of that could be good... drawing connections between [things] visually as I'm speaking."*

P7$_{expl}$ felt the diagram was effective at communicating the system's understanding in a simplified and easy-to-process way: *It's isolating words. Showing what it's going to do. It's taking the verbs and cutting all the other stuff out.*

P8$_{expl}$ specifically described the semantics diagram as the equivalent of a program debugger. *"This I really liked. I think this is a really cool debug thing. It's very intuitive. I immediately understand what happens [in the system] when I read it."*

The feedback suggests that the semantics diagram successfully helped participants understand the system's behavior and feel more comfortable that it was about to do the right thing, and if not, that it was correctable. As to the exact design of the diagram, some feedback suggests we might simplify it further or provide more editing capability through the diagram, depending on how complex the command is. e.g. collapse the diagram into the *transcript view*. In short, this visualization and correction interface is appreciated.

11.2.3.2 UI Design Layout

We acknowledge that as a prototype, DrawTalking has not yet gone through a UI beautification step. Participants felt that changes to the text, colors, and borders would make the experience feel friendlier, but no one felt this detracted from the experience. Rather, the perception was that the goal was to make all functionality available at the top level, as if for a developer's menu. Our understanding was that some participants in this case wanted less on-screen to avoid being overwhelmed. P5_{expl} was curious what a 0-UI version of DrawTalking might be, if the technology permitted it.

In terms of the effectiveness of placing UI elements at the top-level, participants wanted to see functionality, such as the rules view, more easily accessible at the top level. Our implementation nested it under button mixed with a keyword selection. This reportedly did not detract from the experience, but some participants wanted to see more such functionality at the top-level since it was useful.

The takeaway here is that there's a delicate balance between keeping functionality spatially-local and instantly accessible, and keeping the interface clean. Preferences will vary wildly per user and per function. In the future, we mind consider a more dynamic and customizable UI to meet more users' needs at different levels of familiarity.

11.2.4 Habit: Clearing / Discarding the *transcript view*

Participants were (correctly) aware of the *transcript view* so they knew what speech the machine understood.

We observed that participants habitually cleared the *transcript view*. On the one hand, this is good because this is one way to ensure well-formed commands. P1$_{expl}$ said that they used "discard" and "speech" toggle buttons a lot because they wanted to "make the text clean." P5$_{expl}$ similarly said they wanted it to be "blank." In some sense, this suggests that the design of the interface works. The user chooses when their speech should be processed or discarded. However, the comments also suggest that having an always-on transcript is a double-edged sword: it puts the user into the habit of keeping the input clean for the machine due to a preference for a clean transcript — but it raises the question of whether we can more automatically hide this information contextually so it's only shown when needed. We believe that based on P1$_{expl}$'s feedback, it might be useful to hide the transcript automatically when speech recognition is toggled-off – but optionally in case the user wants to access the words. Alternatively, we could separate the transcript from an optional history for this purpose. Overall, participants e.g. P1$_{expl}$ understood and used the interface as intended and successfully came-up with their own workflows for turning on/off the speech input.

11.2.5 Potential for Adaptability and Extensions

Participants envisioned how DrawTalking could be integrated into or with other technologies to support greater capability on-top of its existing interactions. Participants generally considered DrawTalking a complementary system or potential plug-in as a feature in other applications. Alternatively, DrawTalking could be enriched by different choices in input processing.

Some participants described a full pipeline for what they say a fully-matured DrawTalking might look like. Namely, the key features to support next would be a robust import/export system for both the interactive animations created in DrawTalking, as well as external content libraries (images and actions). This would be most useful after creating the initial sketches. Possible use of language models could help make language input more natural or supply a wider range of generative content alongside the user-sketched content. Alternatively, DrawTalking could be integrated into existing feature-rich engines, purely as an interactive control mechanism.

P1$_{expl}$: *After the kid prototypes the game, they're going to want to drop in more professional assets. Maybe it's not the beginning – the playful piece of it.*
Would there be a way to export actions and interactions to code? e.g. take actions and dump into JavaScript. Your playful world suddenly becomes a template for building something in an open-ended environment that can be manipulated using actual code.

On large-language model integration: *Right now the grammar's structured. You have to conform. I wonder if you could use a language model to parse human speech and transform it into the desired grammar. So I could speak more conversationally. That might free people from having to learn the specific grammar. Potentially lower the bar of entry.*
I think it's awesome the way it is. i.e. LLMs could supplement DrawTalking, but DrawTalking is a good foundation.

Finally: *If you build this as a basic generic toolset that can handle basic actions, the artwork generation, and the exporting options – I think it could be a good engine, especially if you throw LLMs (Large Language Models) in the front-end. You could sit here and have a conversation and build up just using language your interactions and then you send that out for code. That'd be fun. I'd do this all day.*

There, P1$_{expl}$ describes the creative process in DrawTalking as something one could export into more production-ready code and assets. Using LLMs to process language input could support more conversational speech. As of writing, LLMs are too slow to process speech at interactive time compared with the libraries we used paired with structured speech parsing. In-spite of the less-natural input requirements, our existing prototype is faster. However, if we assume LLMs will reach interactive time (less than half a second) for simple tasks like speech transformation, this would be an excellent possibility for a more fully-developed implementation of DrawTalking.

P5$_{expl}$ wanted to see the ability to export to real code including variables and rules. Similarly to P5$_{expl}$, they ask: *"Have you thought of using language models with this? [Tell it:]your job is to* *take any generic way people are describing things and turn it into this precise syntax."*

P9$_{expl}$ suggested the possibility of treating the user's rough sketches as a phase 1 of an interaction process. Phase 2 would focus on replacing the rough-sketched assets and animations with higher-quality 3D models and motions. On the topic of LLM-integration, however, P9$_{expl}$ cautions, *"if I want to make a very specific video (e.g. a character 'digs gracefully'), an LLM is not trivial"* P9$_{expl}$ describes DrawTalking overall as an extensible concept that could be adapted as controller for functionality already existing in many other applications: *"this idea can be extended to all kinds of [things] with more actions, movements.* ***If for some game development where they already have the gadgets set-up ... just incorporate the language and control interface here. We can easily create animations with the fancy stuff they have."***

Our takeaway is **that participants understand DrawTalking to be an generic intermediate interface for controlling and programming elements across many possible applications.**

11.2.6 Participants' Quality-of-Life Suggestions

As a non-production tool, DrawTalking could be improved with many production-level features to make it more complete and faster-to-master. For example, ... Participants felt that a built-in tutorial system and in-application listing of possible verbs (as opposed to a manual) would speed-up their learning process. Offering an optional master list per-object of active verbs being performed would provide additional insight into the current state of the world (P1$_{expl}$).

11.2.7 Flexibility to Speak with an Agent or Created Characters

We chose to support narrative third-person form inspired more by how we might present or tell a story, but it would be useful to have the flexibility address the machine in any possible way for different use cases. For example, some participants found themselves trying to speak to one of their sketch characters in imperative form (i.e. telling the character to do something rather than describing it). This indicates they were at some level immersed in the world they were creating, rather than always detached.

P1$_{expl}$ suggested, we *can think of how the machine could alternatively talk back to you: 'What should I call this object?'* — meaning sometimes it's useful to have direct communication with a machine agent as when thinking-aloud.

We agree that this greater flexibility in input would be very useful when extending the audience and scenarios DrawTalking or similar approaches might be used.

11.2.8 Possible Immediate Grammar Extensions

We noted categories of vocabulary and sentence structure that were missing, but likely simple to implement in a future revision or with the aid of more feature-rich language processing. Namely:

1. *synonyms/analogues* e.g. "touch" or "hit" might be equivalent to "collide with" or "overlap." "separate" might be an event analogous to "after X collides with Y." "follow" might sometimes be analogous to "move to" even if it is implemented as a continuous action. Others are simple addition such as "point at" that might alias "rotate to." "If" could alias "when." The interface already provides some suggestions for remapping unknown verbs to known ones, but we could extend this further with a larger library or using more powerful models in the backend. However, the meaning of these verbs might change per-person and per-use case, and might require much more contextual information. The user might wish to be able to swap-out a vocabulary and/or an implementation might want to integrate models for predicting context.

2. *passive tense structure*

3. *imperatives* Sometimes the participant might want to command their own character directly e.g. **P2**$_{expl}$: *"When the squirrel collides with the collider, stop"* or the machine (often if an editor operation) as in "Make a copy of the star." We can transform these different forms into narrative form with an implicit subject, or support them directly. It would be good to support all manner of speech input, whether it's narrative, dialogue with elements, direct dialogue with the machine. We simply chose to focus on the first.

4. *implicit subjects* It would be more natural to remove the need to re-specify the subjects in different clauses of a sentence. For example, "When X collides with Y, X <action>" could be replaced with "When X collides with Y, it <action>". "X <action> and then X <action2>" could be replaced with "X <action> and then <action2>." **This already works** in DrawTalking for most of these types of cases. However, as is the case in actual English, less specification introduces possibilities for ambiguity. "It" above could refer to X instead of Y, but in the original example, it's unambiguously clear that "X" is the subject. There is a trade-off between "naturalness" and possibly confusing the machine. In this case, re-specification is much more likely to work.

5. *additional speech-labeling capability* We could think of ways to attach more properties faster to sketches by recognizing that e.g. "This is my house" might establish an ownership relationship between a character representing the user (such as an avatar) and the house sketch.

6. *additional English-as-A-Second-Language (ESL) Flexibility* DrawTalking assumes we can differentiate between singular and plural forms of words using articles (The, an) to effect different commands. In many languages, these don't exist or are less intuitive to ESL users. We can think about ways to achieve similar effects using context even if the user forgets or omits altogether these parts-of-speech. The underlying NLP libraries we used support some flexibility here, but are more prone to fail if the English input has grammatical errors and

too many fragments. Handling these kinds of errors was out-of-scope, but is an important thing to consider for fully free-form speech input and a larger potential audience.

7. *Multiple Commands at Once* To support additional flexibility, we could allow the user to create and segment multiple commands at the same time. This might be useful to create (or suggest) related commands faster. For example, one might want to set-up a UI button with an event upon press, and a stop event upon release. It would be good to set-up both of these together.

8. *More complex prepositions, idiomatic phrases* We omitted structures such as "all of the" in e.g. "The character jumps onto all of the platforms" for simplicity, but it's reasonable to recognize this use of prepositions to specify number.

9. *Recurring Inverse Operations* We might often say something like "This moves up and down." Interpreted literally, up and down directions would cancel each other out. For some cases, it might be obvious that this is actually meant to be a looped sequence, but that might require additional contextual information or built-in known structures.

11.3 Artifacts

See appendix 13 for artifacts from the sessions.

11.4 Summary

Participants identify DrawTalking as a new approach to working-out visual problems and interactively programming worlds using speech. It's promising in part because it doesn't require explicit coding experience, yet affords many of the capabilities of programming and visual thinking. It enables spatial interaction and control in a creative environment. Participants felt it was intuitive, fluid, and flexible once they overcame learning it for the first time, and could see its potential across many use cases.

We found, generally, as we moved to increasingly complex programmatic features such as repetitions and rules, the participants became more excited e.g. **1**: *"I want to try **that**"* — pointing to rules on the info sheet13.

The comparisons with other visual programming-like environments made sense – e.g. Scratch – and we consider this a good thing considering that Scratch is often used professionally as a programming-onboarding tool or as a model for other such tools. Participants identified (when comparing with Scratch) that the speech controls combined with drawing all-in-one generally made the tool even more accessible and simpler in terms of number of steps. This line of discussion (when it occurred) suggested that knowledge from Scratch was transferable to DrawTalking. DrawTalking was also perceived better in particular by $P7_{expl}$ in the sense that it doesn't expose an explicit coding interface or textual blocks. The semantics diagram, despite its visual similarity to node diagrams in blocks visual programming, was perceived more as a visual indicator rather than a programming interface. $P7_{expl}$ appreciated that it simplified the English input while preserving the readability. Had DrawTalking exposed more of a coding interface, it's clear that it would be perceived as less accessible to non-programmers. Note, that Scratch is generally an onboarding tool for learning programming, whereas here, the feedback suggests that DrawTalking could be a model for achieving programming capability without needing to learn the traditional model of textual coding, other than the logic present in language. We consider this a very promising for supporting wider audiences.

The use cases that people identified all aligned with our original goals of supporting creative exploratory use cases involving imagination, prototyping, and working-out problems through drawing, or, alternatively for more deliberate activities like presentation and learning. The use cases fit into multiple categories across prototyping, visual thinking, design, and iteration, explanation, and system-level control. The most distinct categories deal with real-time creativity, real-time presentation to an audience, and the system-level application control (a lower-level element of control). We can also think in terms of "personal/private" e.g. thinking-to-oneself or a with a friend/collaborator, out-loud, versus "public" where there is an audience. Our user testing looked at the first. This indicates how DrawTalking's approach could be incorporate at many levels of interactive control.

Our use of language for labeling was cited as a natural intuitive input, and the visual feedback afforded by the semantics diagram gave users confidence that the machine was doing the right thing, and that the user could correct errors if not. Participants felt the flattened UI and system control by speech reduced the complexity they encountered in their experiences other comparable tools and environments.

Participants also understood DrawTalking as a complementary tool that could plug-into other applications, and could serve as a feature to enrich the controllability of other UI, animation/game engines, toolkits, or systems. In the reverse, participants noted that without changing the core ideas in DrawTalking, extending it with more powerful language models in the future (given speed improvements) could make it even more flexible in terms of capability and language flexibility. In short, DrawTalking's primary benefits come from the interaction techniques for control and programming by speech. The combination of speech and drawing control was liked and useful based on feedback. Additionally, there is potential to transfer these general features into other domains or systems, even including those with no specific relation to drawing.

We also identified additional potential audiences including children and ESL learners who could be the target of future extensions of DrawTalking.

As technology evolves, DrawTalking might become increasingly accessible and natural — either the interface itself or as a general feature. As-designed, DrawTalking can accept a specific input inform, and external tools can convert to that input. This might make it easier to test interoperability with language models to support more natural and complex speech inputs, while keeping the research and results reproducible.

Lastly, we identified areas for improvement in the interface and language understanding modules and grammars themselves. Participants felt that DrawTalking's capability far outweigh the restrictions on the language structure and grammar. We observed that participants learned labeling by speech immediately. Most participants felt that the system needed time to learn mainly due to the language structure limitations and the large number of possibilities of what could be created in the environment. Even with the limited verbs and and other parts-of-speech, they felt the system was rich with things to try. As several participants stated, the experience of trying DrawTalking for the first time was like learning a new programming language, so it's arguably normal for there to be a short-term learning curve. All participants were confident that with more time with the tool they could achieve mastery. i.e. the interactions were clear; the feature set just needed more internalization. Participants such as $P2_{expl}$ wanted to take DrawTalking home to explore it more fully. We expected this outcome.

We consider it a success that all participants were able to understand and learn DrawTalking's mechanics, identify several use cases, and speak deeply about the interfaces strengths, limitations, and perception. It's a strong indicator that DrawTalking is a useful, understandable tool, that participants can identify how it might improve or otherwise make their workflows and experiences more interesting, productive, and enjoyable. It's also a success to see that participants identified and enjoyed the programming elements, and overwhelmingly likened it to a playful exploratory experience that could work in several contexts, and that might be enjoyable to children.

12 | Discussion and Future Work

The core concept behind DrawTalking is the control mechanism: users speak to label sketches with semantics; the machine uses this information to empower the user with computational capability; the user's speech and direct manipulation via pen and touch directs the world with the support of that computational capability. Our design places the user in control and gives them the flexibility to make decisions for how to create the world – how should it look and how should it behave.

Through our explorations and evaluations, we've identified that beyond our initial motivations, DrawTalking has become multiple things:

DrawTalking is a specification for input/output (IO) that combines elements of drawing, language, and AI mediation.

It is also a creative-social medium inspired by storytelling, presentation, and making-believe.

We can also think of DrawTalking as an interactive approach to programming by speech without code. User study participants compared it directly to visual programming environments like Scratch and discussed many use cases for the interaction concept, ranging from playful exploration to demanding creative productions and games.

Sketches in DrawTalking, one might observe, are just containers with variables encoding semantics, values, and flexible representations. The machine searches for and evaluates these "variables."

We've designed our implementation of DrawTalking to accept a generic format to which multiple natural language frameworks and models could likely "compile" to. Then, different implementations of DrawTalking-like applications could interpret a more deterministic, stable input as long as the underlying technology could continue compiling reliably, even if the technology changed. Using deterministic inputs in general could aid in the the development and (reproducible) research of systems and reduce dependencies on the underlying language technologies.

User study participants, also, suggested how the underlying functionality of DrawTalking could be extended by other technologies (e.g. language models), or be integrated directly into other applications. Through the combination of drawing and programming elements, DrawTalking acts as a step in a creative iteration process, potentially with links to other applications in the pipeline. Unified interactivity in the style of DrawTalking might enable a seamless dataflow of content between applications, facilitating a more fluid iteration loop at a system-level.

All of the above can be thought of, collectively, as a metaphor and proof-of-concept for a unified operating system.

We think that DrawTalking calls back and pays tribute to visual computing, language interfaces, and intelligent sketching. These are fundamental ideas developed since (at least) SketchPad[Sutherland 1963], Visual Programming[Sutherland 1966], SHRDLU[Winograd 1972], and the Smalltalk environment[Ingalls 2020]. Smalltalk is a particularly apt comparison, as it was a programming language with human-readability in-mind, a mental model inspired by communication between biological organisms, a unified programming environment with interoperability between applications, and the progenitor of more visual-oriented and playful variants for kids (Scratch). DrawTalking in a sense tries to synthesize these ideas together in light of newer technologies, enabling easier communication, creating expression, and thinking. As Bret Victor argues in his talk on humane computing interfaces([Victor 2014]), humans have a wealth of senses and modes of thought. We should consider how to bring them together. We tried to take some steps towards this with DrawTalking.

We've demonstrated initial working proof-of-concept in the space of rough sketching, but we think of the interface as a more general. It is a potential model for how user-centered human-AI collaboration could work in future computer systems. We hope our research can raise questions about how to balance user control with machine automation; how best to use the advancing capabilities we're seeing as artificial intelligence and compute evolve.

We can think about the various improvements we've identified so-far in the prior user testing, in the short-term. There are several more far-reaching directions we could take as well:

12.1 Future Work

The next steps in research should look into how to make the workflow more natural and fluid. The far-reaching "vision," so-to-speak, continues to be to find ways to integrate computing into our everyday interactions to increase our range of creative expression and communication.

In terms of what a true interface of this sort might look like–it might be fully-invisible, with no UI at all, or it might incorporate machine agents, or both.

Much like how DrawTalking was inspired by how the machine could use the semantics in people's natural speaking patterns, we should look for more opportunities for the machine to introspect natural human behavior at low cognitive cost to the user. Simultaneously, we should consider the privacy and safety of the user.

12.1.1 Towards Incorporating Language snf Multimodal Models

Towards making input to a DrawTalking-inspired system more "natural": this might involve to integration with complementary technologies such as large language models to support a wider palette of inputs and features. At present, these models do not return results fast enough to achieve interactive time, so our current implementation better-approximates the speed we'd like to see for quick interactive sketching+speaking. However, we've already begun thinking about how to use them. Participants in our user study suggested that an LLM translate fully natural language into the more structured, domain-specific representations at the application level. We've successfully segmented sentences, simplified structures, and disambiguated context with some reliability. We could output the reformatted sentence to DrawTalking by generating commands directly or by first processing through an intermediate rule-based model or compiler.

We've also thought about asking LLMs to generate commands. We've tried prompting the LLM OpenAI ChatGPT with a simple scene description corresponding to names and IDs of objects we might create in DrawTalking. Given a narrative and the task to output DrawTalking-formatted commands, we've found that the model can often generate several intermediate commands from a single prompt to create a more believable narrative with less user-specification. For example, following several carefully-articulated prompts to the bot, we can describe a scene with just a pirate and a treasure chest that is underground. We can also tell the bot that it has the ability to create objects as needed. We can say, *"The pirate gets the treasure."*, which often will output commands that generate a shovel sketch, direct the pirate to move to the shovel and pick it up, then dig underground to the treasure. Here, the bot generated hidden context leading to several intermediary animation commands. We use the word, "often," though, as we find the output is still unreliable and easy-to-break. There are ways to coerce an LLM into creating more deterministic outputs by combining with rule-based approaches. Encoding a scene model into the memory of an LLM might not be the ideal way to proceed, but this test was encouraging.

Another use case we've tried was prompting the LLM to take a sequence of speech commands and user interactions in DrawTalking and output a full story based on the input. The LLM could help provide some suggestions for creativity-support. This is something we could do today because it does not necessarily need interactive speeds.

We gather that LLMs will be promising for storytelling use cases because they've likely been trained on gigantic collections of stories, and therefore have many examples of context for filling-in intermediate steps. However, this still doesn't mean the LLM will be able to "be creative" in the event that the user wants a more unexpected or illogical result, or even something very specific.

We think the LLM might still need to remain in the role of "advisor" rather than "director," which is why we still need a way to tune between levels of user control and machine automation.

Lastly, multimodal models could help more easily identify and store information about the relationships between sketches and objects given the sketch (or a representation of it) as input. (We can rasterize the scene and feed as an image.)

12.1.2 Towards Greater Editability and Context-Sensitivity

Depending on who the user is and what their background is, the domain-specific language will likely need to change. We've already encountered this with user study participants who wanted greater flexibility in the language input, with looser requirements on the use of articles to specify different types and objects.

DrawTalking's DSL was made with the assumption that the user would want side-view 2D sketch animation, for the most part. If the user wanted to reinterpret the verbs from a different "camera angle," we'd need support for a greater number of verb variants, with the ability to detect the context from the user's previous commands. This is achievable with our current rules-based approach. For example, if the user talks about looking at a "world map, top-down" the system might infer that the camera should face downwards and that the verbs should make sense for "top-down" viewpoints.

If we were to add sketch recognition and image recognition, the user's cultural background, geographical location, and/or profession might influence suggestions for how a sketch should look and how it should behave. For a simple example, a "temple" will look different depending on whether the user's narration is currently describing ancient Greece or Thailand. The machine could store this contextual information without requiring the user to re-specify in a prompt, and automatically a "house" would generate the most fitting image.

We think, however, the next unsolved question is how to support more automatic generation, suggestion, or editability of animation based on context, *without* explicit

coding. Moreover, how should these libraries be defined? Research in cognitive science has tried understanding how people formulate "concept libraries" for deconstructing objects and tasks [Wong et al. 2022]. Taking closer look into how we think about language and visuals at a cognitive level may help because regardless of whether we use a large library or LLMs to aid in code-generation, there might be too many assumptions at-play. The user will almost always find a point at which they want something different and new. (Even with our user study, we found several cases in which a logical mapping between some simple verbs like "follow" contextually did not align with what they wanted e.g. "move to and then stop.") We believe a solution might require more rigorous theoretical study, while building open-libraries from a combination of users' previous sketches, programming by demonstration, coding by experts, and suggestions from language models. i.e. it will be a combination of everything. We would be excited to see how we might find ways to create new verbs using pure language, beyond the programming-by-composition supported in our implementation.

12.1.3 Collaboration

In our user study, we simulated a collaborative whiteboarding scenario with the researcher and participant at the same board. However, we would be interested in seeing how to adapt to multiple user and spatial configurations. How does one collaborate in-person with multiple people talking, at the same board or at different boards? How would a team-based collaboration work? A 1-1 conversation or a brainstorming session over a diagram? This might require ownership semantics on top of sketches to track information about each contributor. This might also become important when asymmetric roles come into play, for example, in a teacher/student environment. Furthermore, we might need to devise ways to hide information selectively, per-user.

We've also started to play with the idea of connecting multiple devices, where iPads could be remote "portals" visible on others' screens. It would be a worthy challenge to find intuitive ways for multiple users to edit the same world at once.

Furthermore, DrawTalking should in-principle support any natural language so anyone could collaborate. Should the representation and behavior of content change, per-user, depending on their language? Can we auto-translate from one language to another to overcome language barriers, in addition to helping language learners practice with a single target language? Can we also aid in communication with people who suffer from hearing impairments?

12.1.4 Cross-Applications / Cross-Experience

A next logical step would be to demonstrate focused proofs-of-concept for the use cases identified. For example, generically plugging-in DrawTalking functionality into a production application such as Adobe After Effects, or making our tablet application talk to a version of DrawTalking for 3D scenes. There are many possible design considerations for how to make communication between applications sensible given different visual representations. One possibility is to treat the tablet mode as a map for navigating and editing a scene, with a connected desktop or immersive device as a 3D environment for a user to experience. This form of asymmetric control and views could facilitate interesting types of improvisational and procedural storytelling, controlled by an external user, and experienced by an audience. Take a role-playing-game, for example. This could also be a mode of teaching, or playing or designing of games

12.1.5 Multimodal Content

We've focused on freehand sketching, but DrawTalking objects can have a flexible representation mapped to any object: we can consider spatial *audio* design, digital musical instrument automation and music composition, and even physical objects (Internet-of-Things or robotic) control.

12.1.6 Immersive Experience Control

Spatial-immersive computing (XR, AR, VR) offer new ways to introduce computing into a real-world, spatial context. This is promising because it could place computing capability directly in the hand of the user without any device, other than a pen. Participants in our study highlighted that one of DrawTalking's strengths included the combination of spatial manipulation, interactivity, and natural inputs. This translates well into immersive technologies. We're excited at the potential for non-intrusive social engagement and collaboration with the addition of the ideas we've explored in DrawTalking. Research and industry have long tried to transition from mouse/keyboard/controller input to more natural inputs like hand gestures and gaze. We're still in the process of developing a scalable and affordable solution to a trackable pen, as of writing. How would DrawTalking's controls translate to immersive environments? We believe the interoperation between speech, text, and gesture is a promising direction. Rather than have explicit text and semantics views, we might want to create equivalents that fit into a real-world environment better, and compare against the current captions-like visuals. Further into the future, non-invasive brain-computer interfaces[1] might make it possible for the user to register intent without interaction with a UI, period. A version of DrawTalking combined with fully-natural input would likely feel freezing. This is one reason why thinking about a no-UI version of DrawTalking would be most exciting. Imagine a version of DrawTalking that registers intent perfectly. This is partly why we think it's important to think about and prototype the possibilities now, to help understand use cases for the future.

On another note, DrawTalking's labeling mechanic might complement scene-understanding algorithms. If the user labels their own environment, we can potentially rely less on computer vision to understand the full environment. Rather, labels could be combined with computer vision, or in some cases, they might be enough. This could help side-step privacy concerns with

[1] See CTRL Labs

exposing camera and sensor data to applications.

12.1.7 Targeted User Studies

We universally identified that DrawTalking might be a strong educational and creative tool for children. We would like to learn how DrawTalking might help children learn program, as well as teachers create engaging learning experiences. It would be motivating to design and run longitudinal and co-design studies with educators and artists using a more polished version of DrawTalking, to learn about people's experiences over a longer period of time after they've achieved mastery.

Developing alternative versions and improvements to DrawTalking for collaborative purposes or specifically for presentation might help us uncover additional needs and pain-points.

Another direction might be to study DrawTalking-like interactions in a wizard-of-oz setup– the researcher (as a human) can potentially match the user's intent closely and type perfectly-formatted commands from the user's speech, possibly at lower latency than could an LLM. It might be easier to accelerate design of the interactions and users' true response to them with little to no technical hindrances. The disadvantage of this approach is that it would not help explore usability or directly benefit people.

We believe all of the above directions combined could be worth studying using ideas from DrawTalking as a starting point.

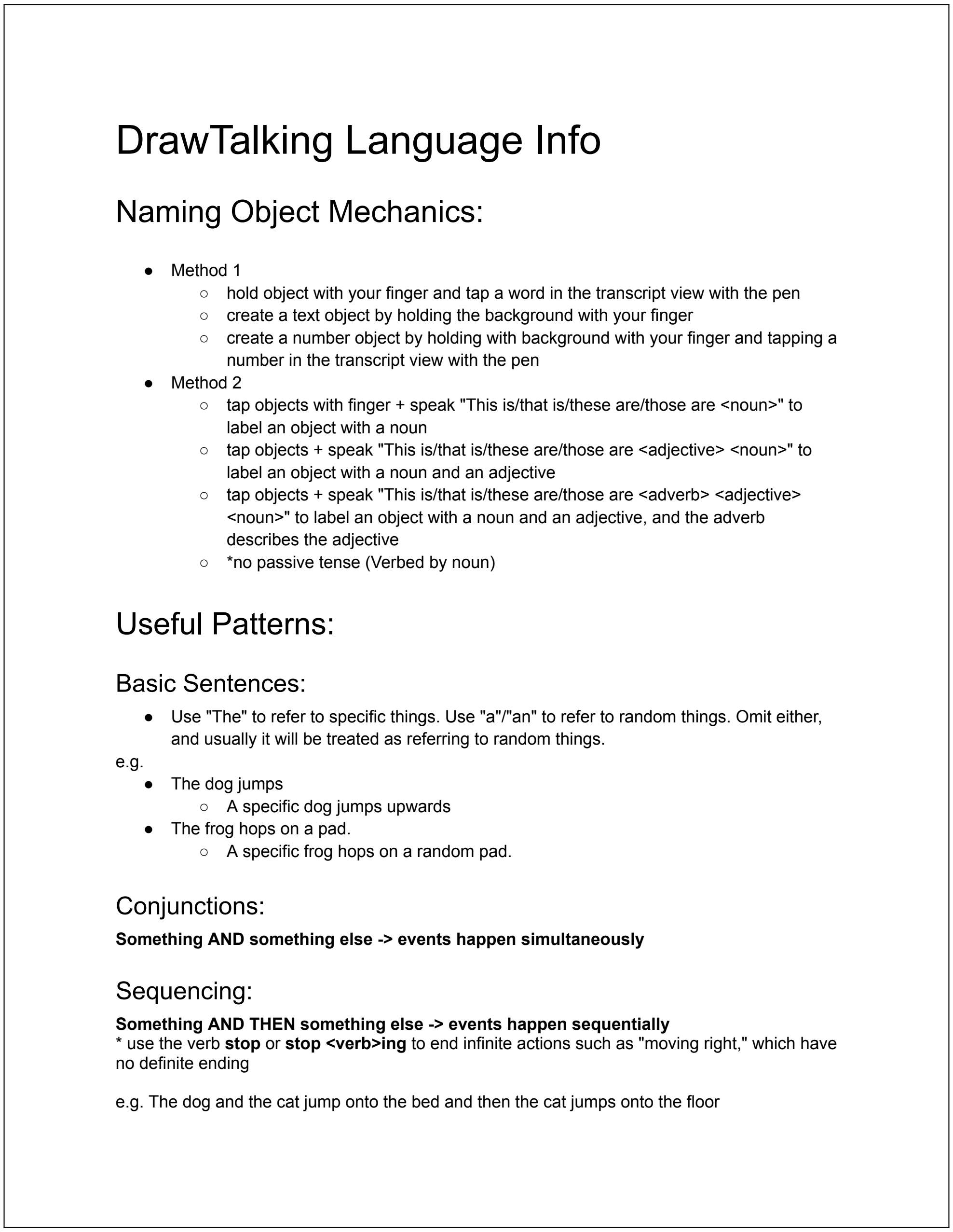
13 | Conclusion

In this thesis, we have introduced and demonstrated a tablet implementation of DrawTalking. DrawTalking enables the user to create, program, and interact with their own worlds using sketching and speech improvisationally. We were inspired by our use of speech to explain and tell stories, and to "make-believe" to communicate information. We wanted to design interaction techniques that would help extend our creative capabilities with computation, while keeping the user in control. We drew upon work in dynamic and programmatic illustration, visual programming, and natural language speech interfaces. Simultaneously, we looked at sketching in the wild. We arrived at an interaction technique in which the user embeds semantics into sketches with their speech to label them. This telegraphs to the machine what those sketches are and how they should behave. The user can then narrate and describe rules to direct the machine in building-up and animating the world. This enables simple storytelling, explanation, and programming using speech, multitouch, and the pen. We found through our design and testing process that our approach generalized to many use cases and reapplications involving human-AI collaboration. DrawTalking, in the end, synthesizes decades of work in HCI and describes a possible approach to designing a full computing system marrying human interaction and machine intelligence, where the user remains in control. We establish our work as a proof-of-concept and starting point that we hope will serve as a helpful blueprint for future natural interface.

In closing, when designing human-computer interaction(s), perhaps most importantly we should ask : "what are the roles of the human and the machine," and "how do we privilege the user’s intent?"

We believe technology should just be a tool for improving lives. A positive reminder for the future: even as we find new ways to create, think, and automate via machine, none of this is possible without the human element contributing its ingenuity and creativity. We re-emphasize that cross-disciplinary collaboration between human-centered, and more computation-centered sub-fields is necessary for the best outcome.

Appendix: User Study Sheet — Sept. 2023



DrawTalking Language Info

Naming Object Mechanics:

- Method 1
  - hold object with your finger and tap a word in the transcript view with the pen
  - create a text object by holding the background with your finger
  - create a number object by holding with background with your finger and tapping a number in the transcript view with the pen
- Method 2
  - tap objects with finger + speak "This is/that is/these are/those are <noun>" to label an object with a noun
  - tap objects + speak "This is/that is/these are/those are <adjective> <noun>" to label an object with a noun and an adjective
  - tap objects + speak "This is/that is/these are/those are <adverb> <adjective> <noun>" to label an object with a noun and an adjective, and the adverb describes the adjective
  - *no passive tense (Verbed by noun)

Useful Patterns:

Basic Sentences:

- Use "The" to refer to specific things. Use "a"/"an" to refer to random things. Omit either, and usually it will be treated as referring to random things.

e.g.

- The dog jumps
  - A specific dog jumps upwards
- The frog hops on a pad.
  - A specific frog hops on a random pad.

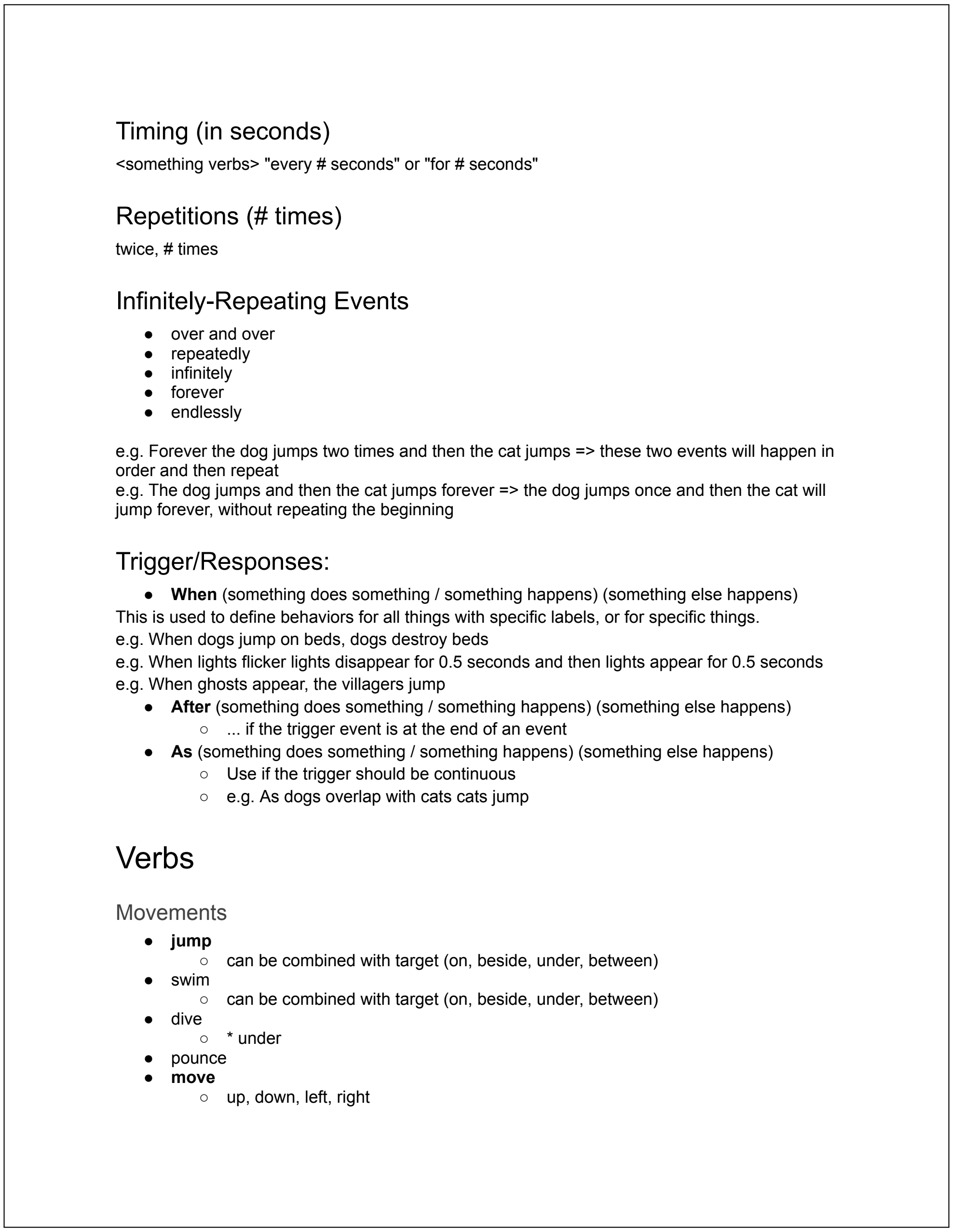
Conjunctions:

Something AND something else -> events happen simultaneously

Sequencing:

Something AND THEN something else -> events happen sequentially
* use the verb **stop** or **stop <verb>ing** to end infinite actions such as "moving right," which have no definite ending

e.g. The dog and the cat jump onto the bed and then the cat jumps onto the floor



Timing (in seconds)

<something verbs> "every # seconds" or "for # seconds"

Repetitions (# times)

twice, # times

Infinitely-Repeating Events

- over and over
- repeatedly
- infinitely
- forever
- endlessly

e.g. Forever the dog jumps two times and then the cat jumps => these two events will happen in order and then repeat
e.g. The dog jumps and then the cat jumps forever => the dog jumps once and then the cat will jump forever, without repeating the beginning

Trigger/Responses:

- **When** (something does something / something happens) (something else happens)

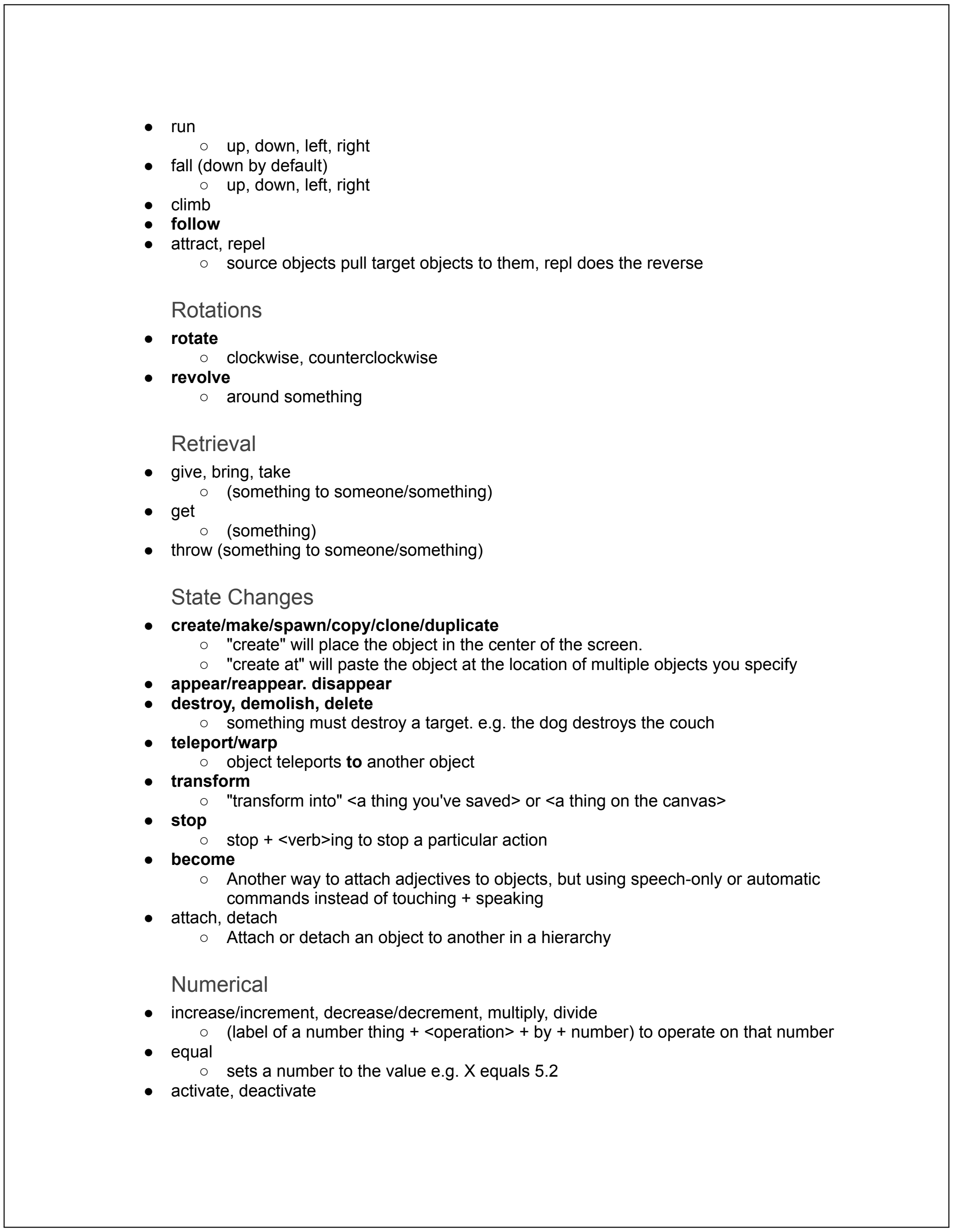
This is used to define behaviors for all things with specific labels, or for specific things.
e.g. When dogs jump on beds, dogs destroy beds
e.g. When lights flicker lights disappear for 0.5 seconds and then lights appear for 0.5 seconds
e.g. When ghosts appear, the villagers jump

- **After** (something does something / something happens) (something else happens)
    - ... if the trigger event is at the end of an event
- **As** (something does something / something happens) (something else happens)
    - Use if the trigger should be continuous
    - e.g. As dogs overlap with cats cats jump

Verbs

Movements

- **jump**
    - can be combined with target (on, beside, under, between)
- swim
    - can be combined with target (on, beside, under, between)
- dive
    - * under
- pounce
- **move**
    - up, down, left, right



- run
  - up, down, left, right
- fall (down by default)
  - up, down, left, right
- climb
- **follow**
- attract, repel
  - source objects pull target objects to them, repl does the reverse

Rotations

- **rotate**
  - clockwise, counterclockwise
- **revolve**
  - around something

Retrieval

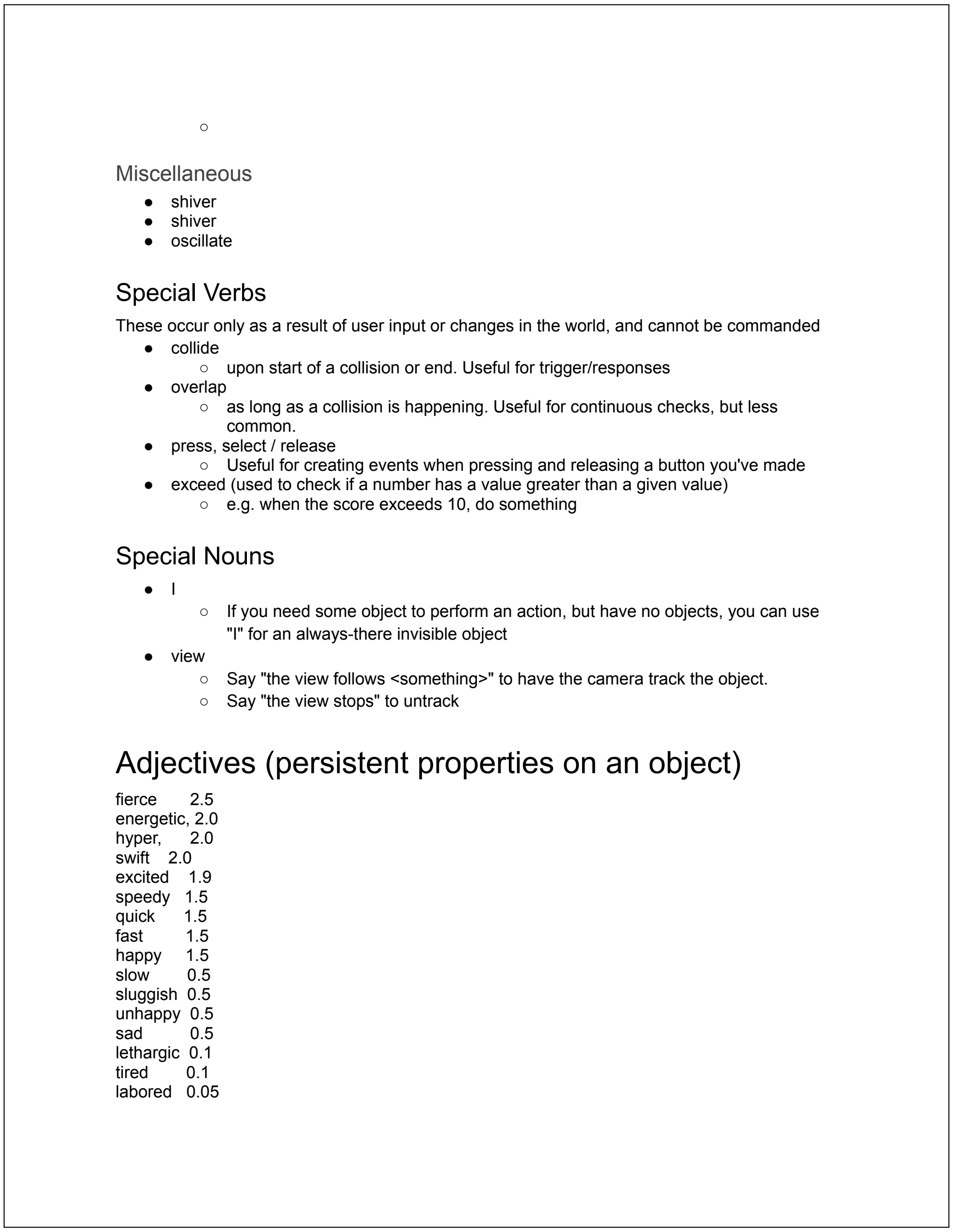
- give, bring, take
  - (something to someone/something)
- get
  - (something)
- throw (something to someone/something)

State Changes

- **create/make/spawn/copy/clone/duplicate**
  - "create" will place the object in the center of the screen.
  - "create at" will paste the object at the location of multiple objects you specify
- **appear/reappear. disappear**
- **destroy, demolish, delete**
  - something must destroy a target. e.g. the dog destroys the couch
- **teleport/warp**
  - object teleports **to** another object
- **transform**
  - "transform into" <a thing you've saved> or <a thing on the canvas>
- **stop**
  - stop + <verb>ing to stop a particular action
- **become**
  - Another way to attach adjectives to objects, but using speech-only or automatic commands instead of touching + speaking
- attach, detach
  - Attach or detach an object to another in a hierarchy

Numerical

- increase/increment, decrease/decrement, multiply, divide
  - (label of a number thing + <operation> + by + number) to operate on that number
- equal
  - sets a number to the value e.g. X equals 5.2
- activate, deactivate



○

Miscellaneous

- shiver
- shiver
- oscillate

Special Verbs

These occur only as a result of user input or changes in the world, and cannot be commanded

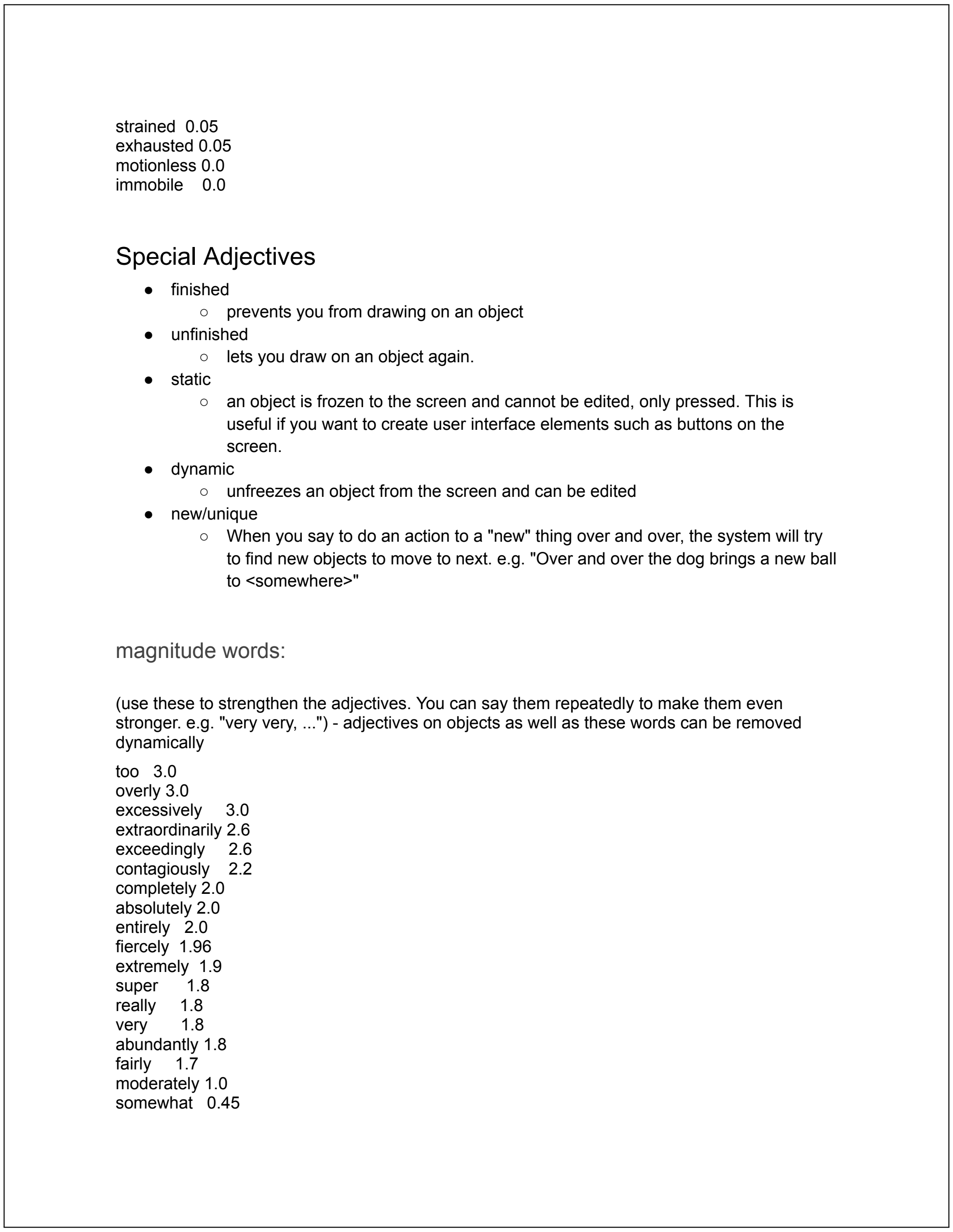
- collide
    - upon start of a collision or end. Useful for trigger/responses
- overlap
    - as long as a collision is happening. Useful for continuous checks, but less common.
- press, select / release
    - Useful for creating events when pressing and releasing a button you've made
- exceed (used to check if a number has a value greater than a given value)
    - e.g. when the score exceeds 10, do something

Special Nouns

- I
    - If you need some object to perform an action, but have no objects, you can use "I" for an always-there invisible object
- view
    - Say "the view follows <something>" to have the camera track the object.
    - Say "the view stops" to untrack

Adjectives (persistent properties on an object)

fierce	2.5
energetic,	2.0
hyper,	2.0
swift	2.0
excited	1.9
speedy	1.5
quick	1.5
fast	1.5
happy	1.5
slow	0.5
sluggish	0.5
unhappy	0.5
sad	0.5
lethargic	0.1
tired	0.1
labored	0.05



strained	0.05
exhausted	0.05
motionless	0.0
immobile	0.0

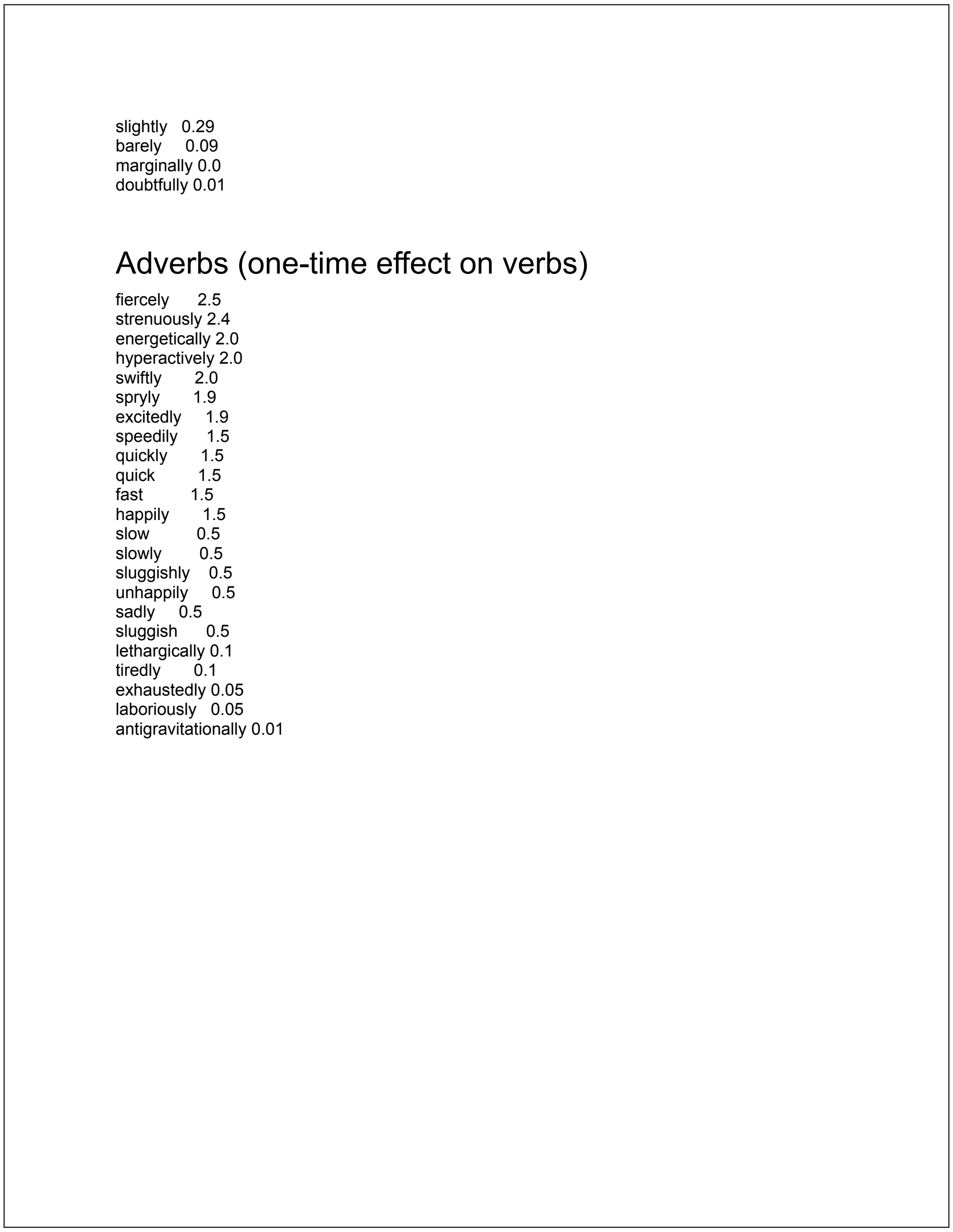
Special Adjectives

- finished
  - prevents you from drawing on an object
- unfinished
  - lets you draw on an object again.
- static
  - an object is frozen to the screen and cannot be edited, only pressed. This is useful if you want to create user interface elements such as buttons on the screen.
- dynamic
  - unfreezes an object from the screen and can be edited
- new/unique
  - When you say to do an action to a "new" thing over and over, the system will try to find new objects to move to next. e.g. "Over and over the dog brings a new ball to <somewhere>"

magnitude words:

(use these to strengthen the adjectives. You can say them repeatedly to make them even stronger. e.g. "very very, ...") - adjectives on objects as well as these words can be removed dynamically

too	3.0
overly	3.0
excessively	3.0
extraordinarily	2.6
exceedingly	2.6
contagiously	2.2
completely	2.0
absolutely	2.0
entirely	2.0
fiercely	1.96
extremely	1.9
super	1.8
really	1.8
very	1.8
abundantly	1.8
fairly	1.7
moderately	1.0
somewhat	0.45



slightly	0.29
barely	0.09
marginally	0.0
doubtfully	0.01

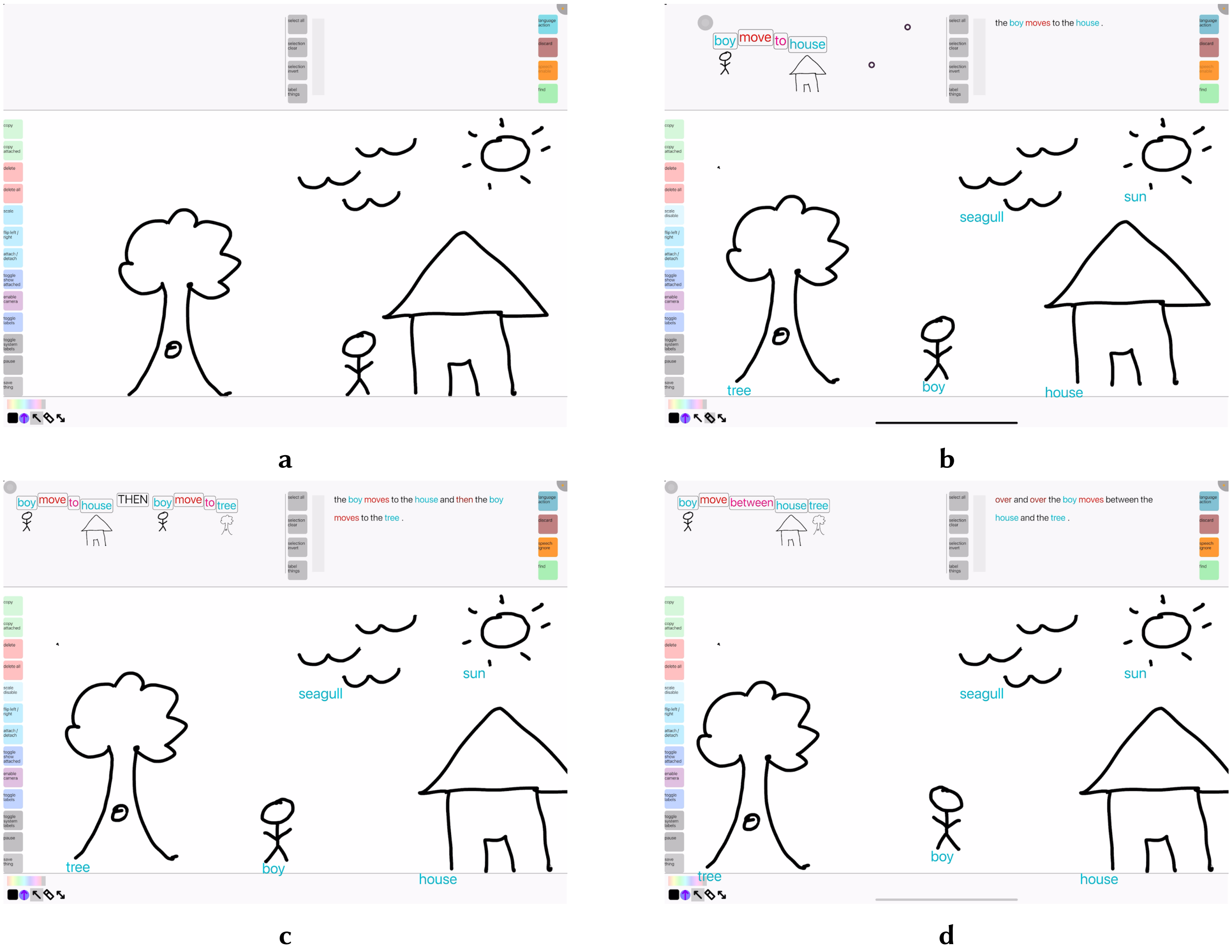
Adverbs (one-time effect on verbs)

fiercely	2.5
strenuously	2.4
energetically	2.0
hyperactively	2.0
swiftly	2.0
spryly	1.9
excitedly	1.9
speedily	1.5
quickly	1.5
quick	1.5
fast	1.5
happily	1.5
slow	0.5
slowly	0.5
sluggishly	0.5
unhappily	0.5
sadly	0.5
sluggish	0.5
lethargically	0.1
tiredly	0.1
exhaustedly	0.05
laboriously	0.05
antigravitationally	0.01

Appendix: User Exploration Artifacts

The following are selected screens from each of the sessions, captured on-device:

Session P1$_{expl}$



a

b

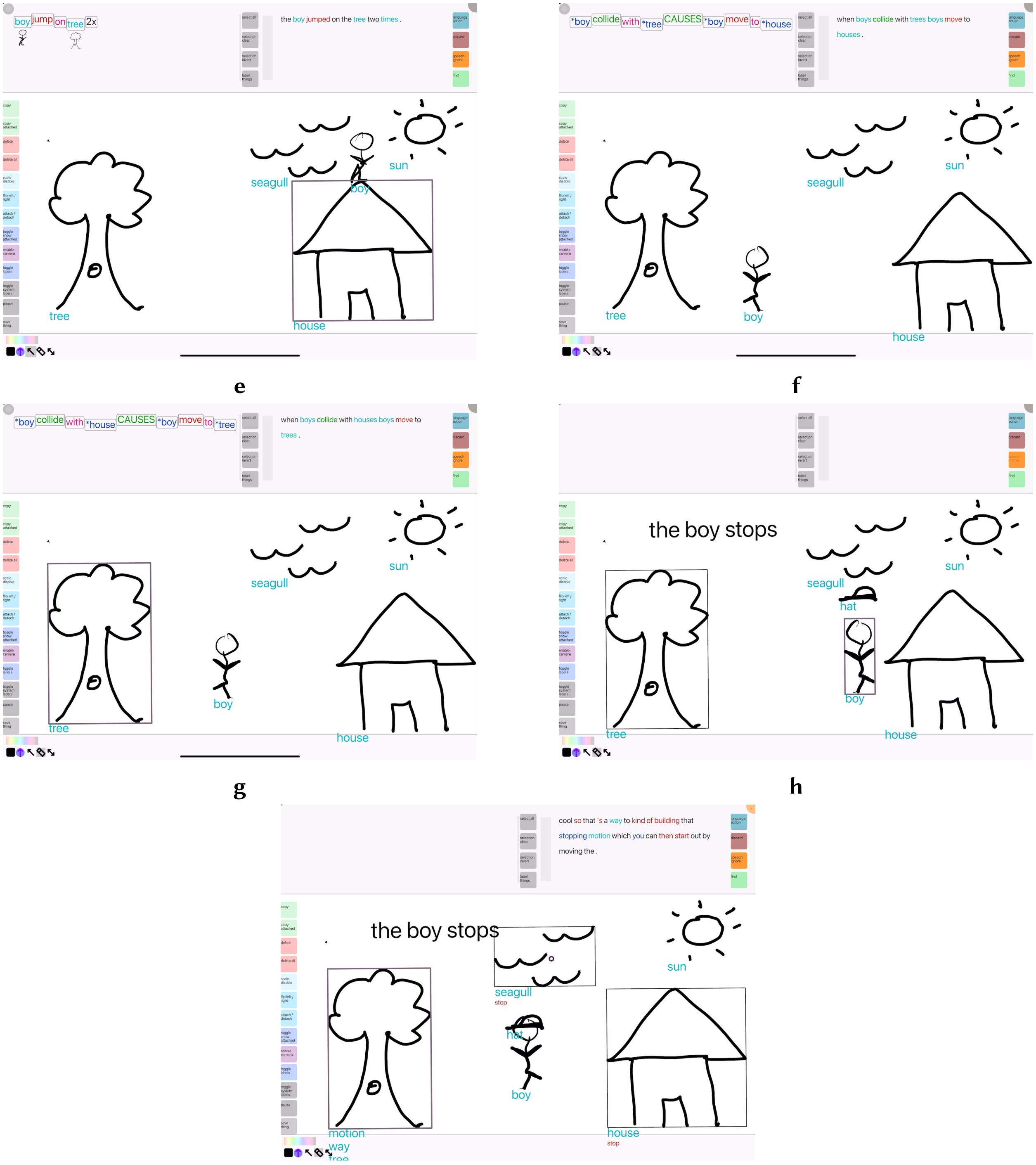
the boy moves to the house .

c

the boy moves to the house and then the boy moves to the tree .

d

over and over the boy moves between the house and the tree .



boy jump on tree 2x
seagull
sun
boy
tree
house
*boy collide with *tree CAUSES *boy move to *house
*boy collide with *house CAUSES *boy move to *tree
the boy stops
hat
motion
way
stop

e

the boy jumped on the tree two times .

f

when boys collide with trees boys move to houses .

g

when boys collide with houses boys move to trees .

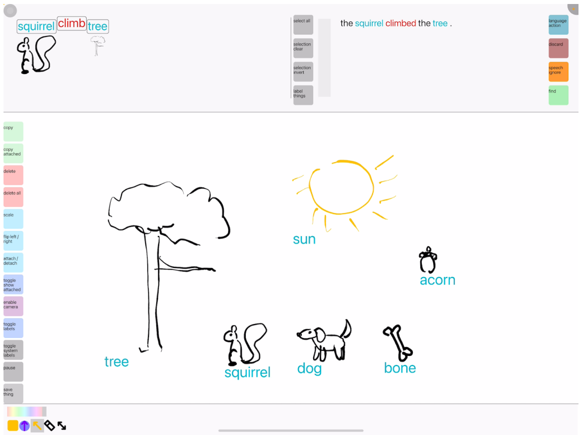
h

i

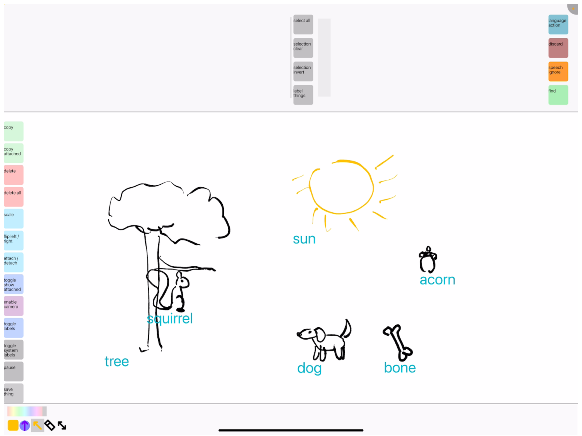
cool so that 's a way to kind of building that stopping motion which you can then start out by moving the .

Figure A.1: P1$_{expl}$

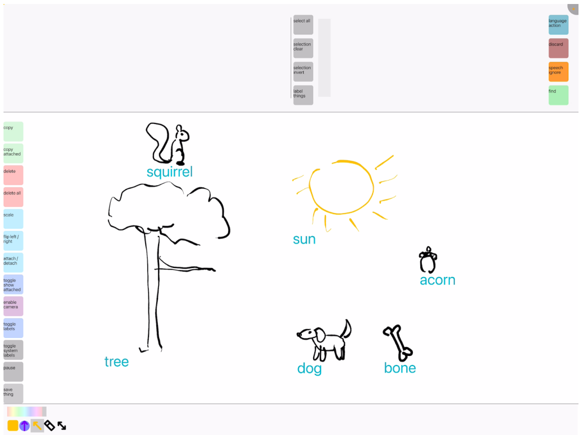
Session P2$_{expl}$


the squirrel climbed the tree .

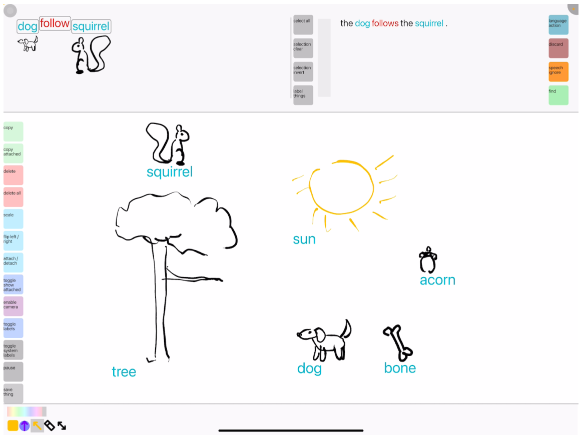
a



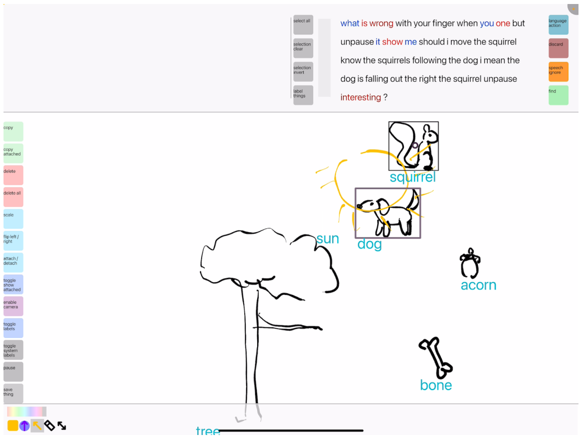
b



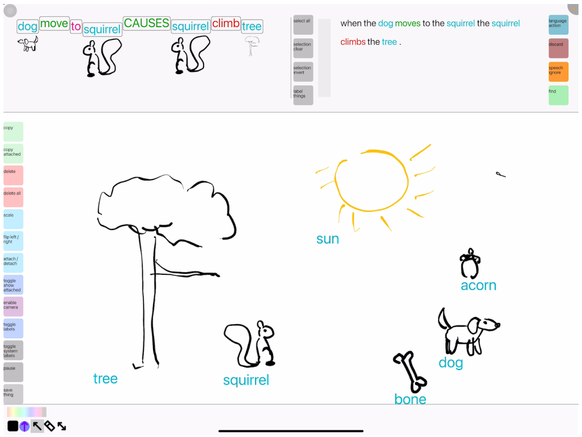
c


the dog follows the squirrel .

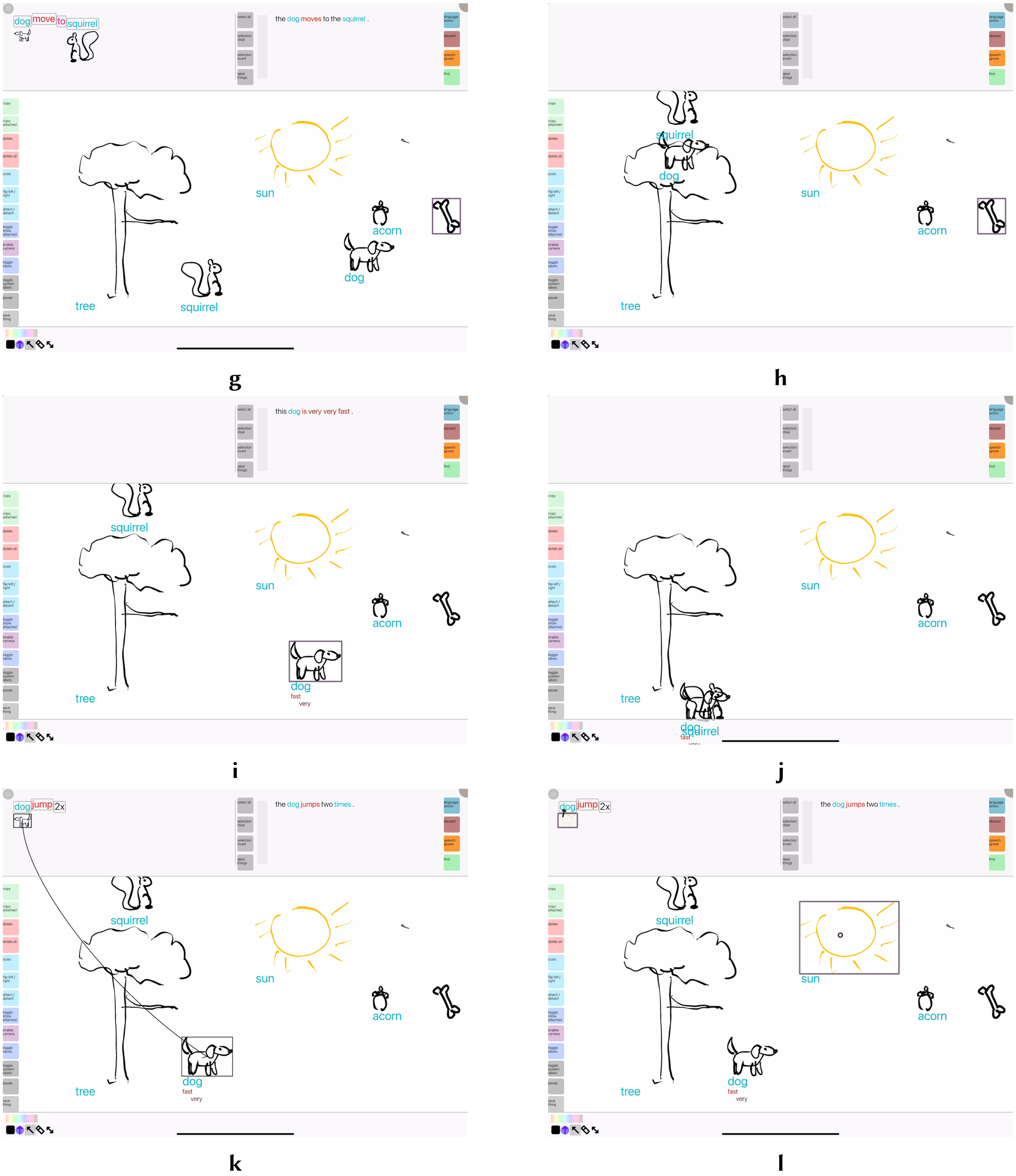
d


what is wrong with your finger when you one but unpause it show me should i move the squirrel know the squirrels following the dog i mean the dog is falling out the right the squirrel unpause interesting ?

e


when the dog moves to the squirrel the squirrel climbs the tree .

f



the dog moves to the squirrel .

g

h

this dog is very very fast .

i

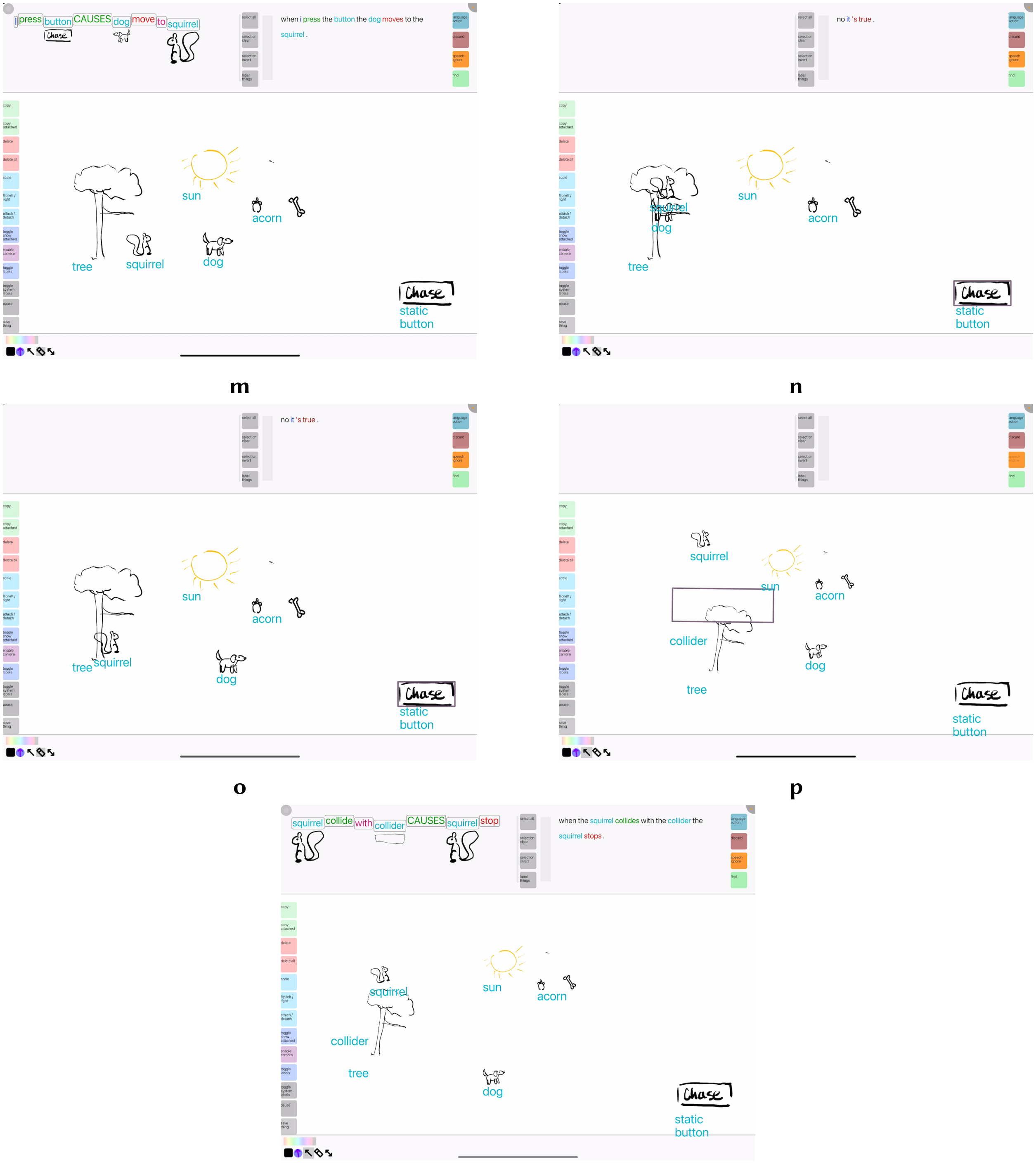
j

the dog jumps two times .

k

the dog jumps two times .

l



m

when i press the button the dog moves to the squirrel .

n

no it 's true .

o

no it 's true .

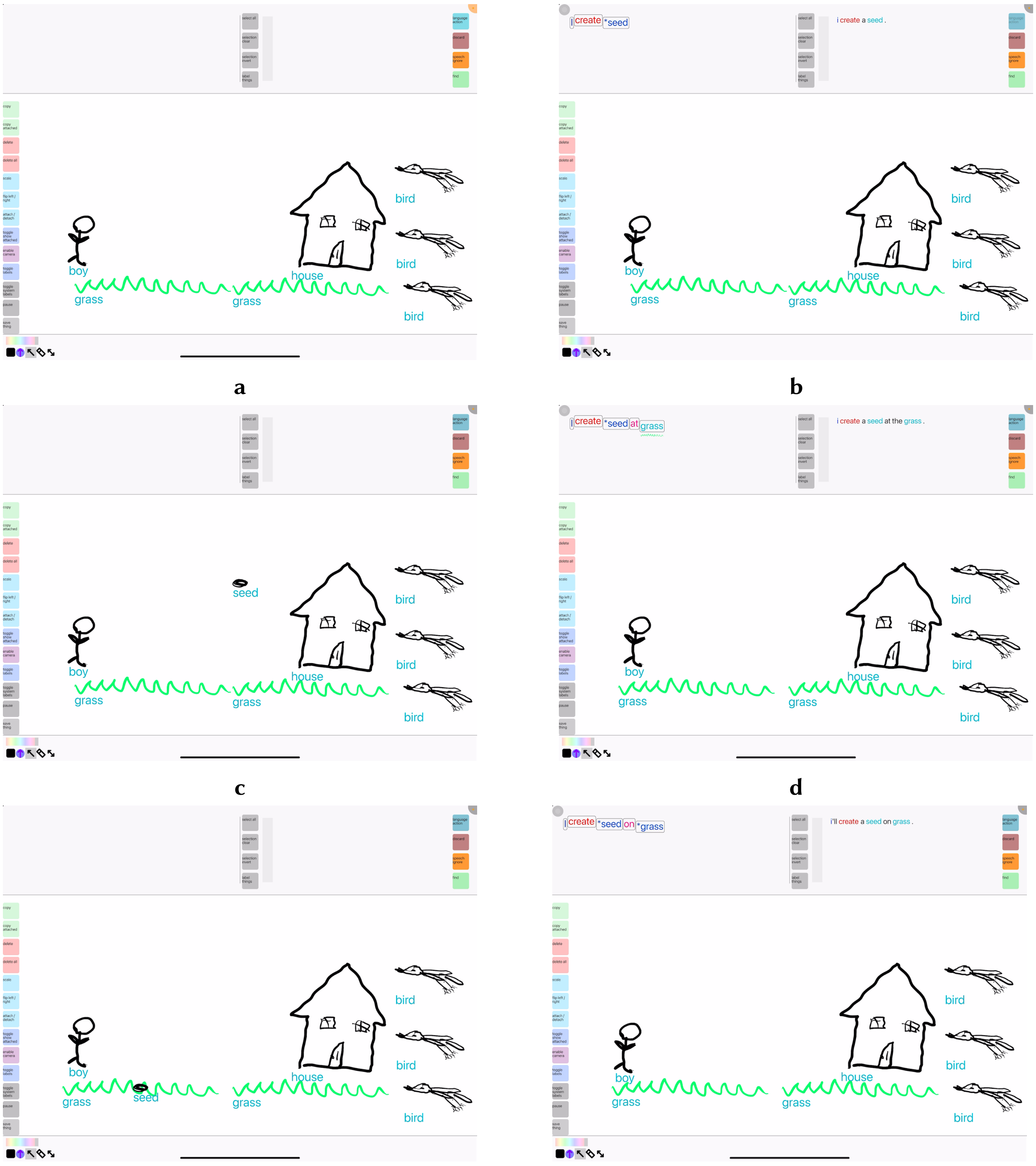
p

q

when the squirrel collides with the collider the squirrel stops .

Figure A.2: P2$_{expl}$

Session P3$_{expl}$



a

I create *seed

i create a seed .

b

c

I create *seed at grass

i create a seed at the grass .

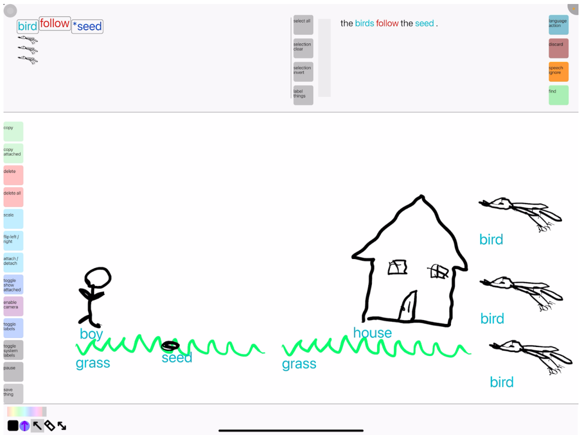
d

e

I create *seed on *grass

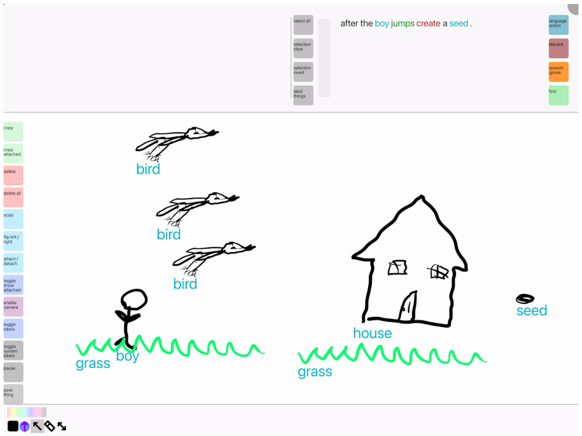
i'll create a seed on grass .

f



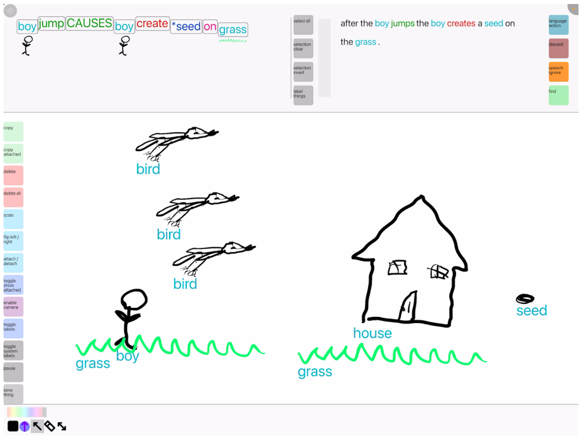
the birds follow the seed .

g



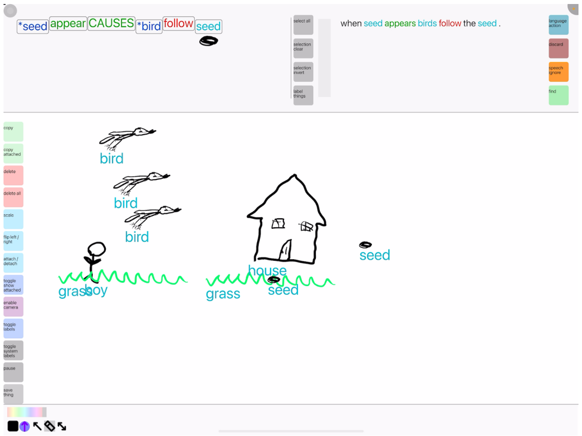
after the boy jumps create a seed .

h



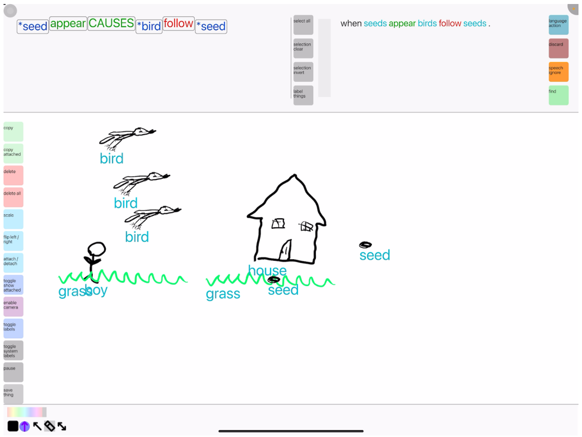
after the boy jumps the boy creates a seed on the grass .

i



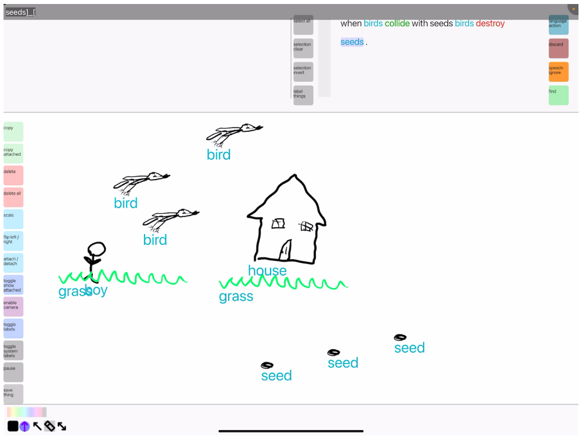
when seed appears birds follow the seed .

j



when seeds appear birds follow seeds .

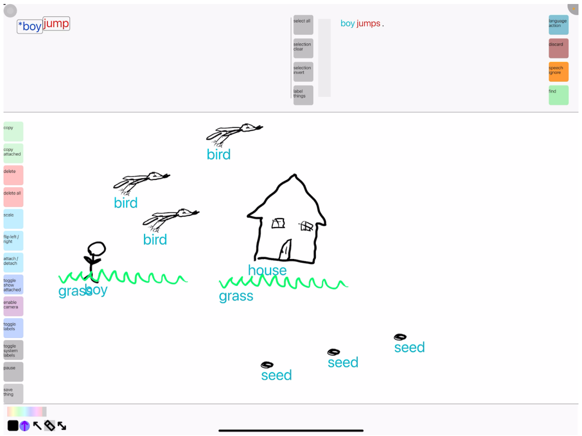
k



seeds]_[

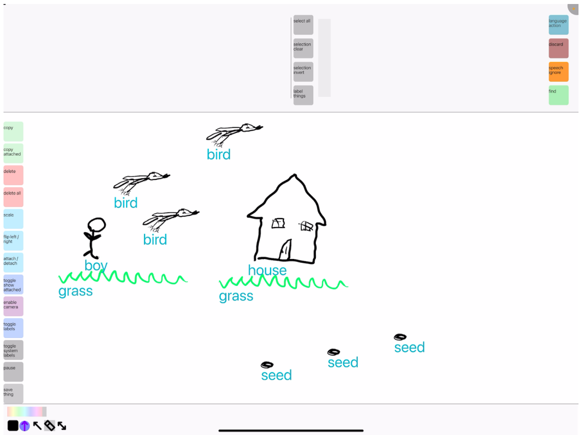
when birds collide with seeds birds destroy seeds .

l

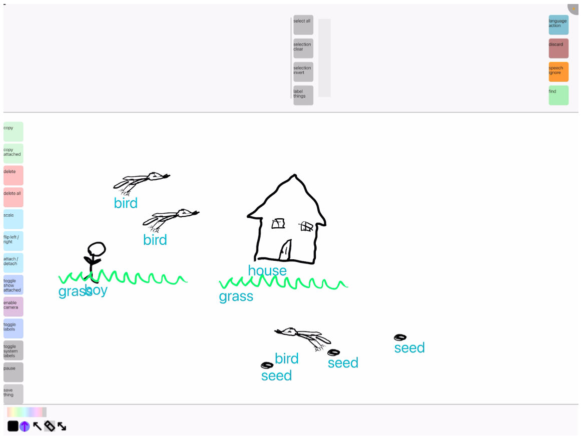


boy jumps .

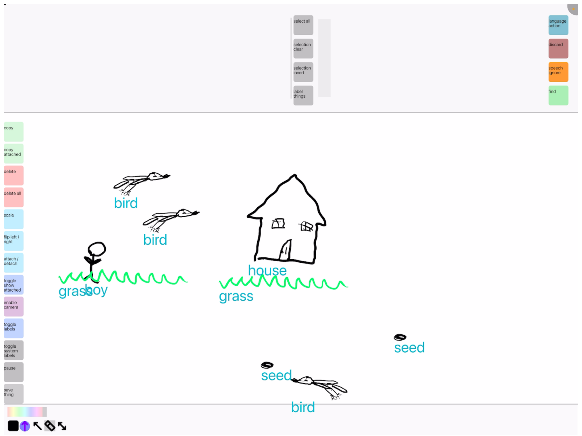
m



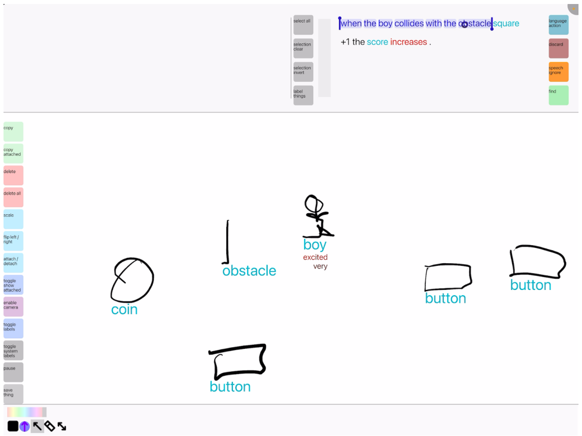
n



o

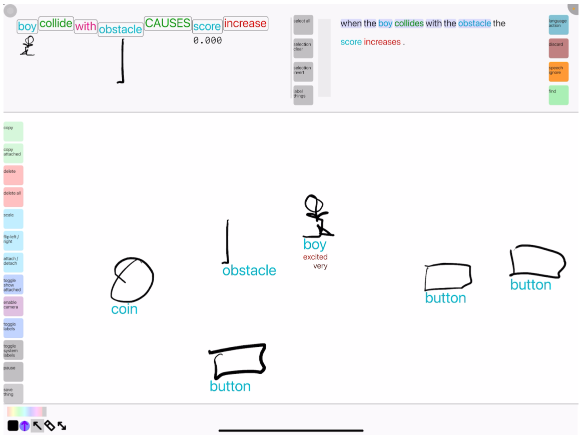


p


when the boy collides with the obstacle square

+1 the score increases .

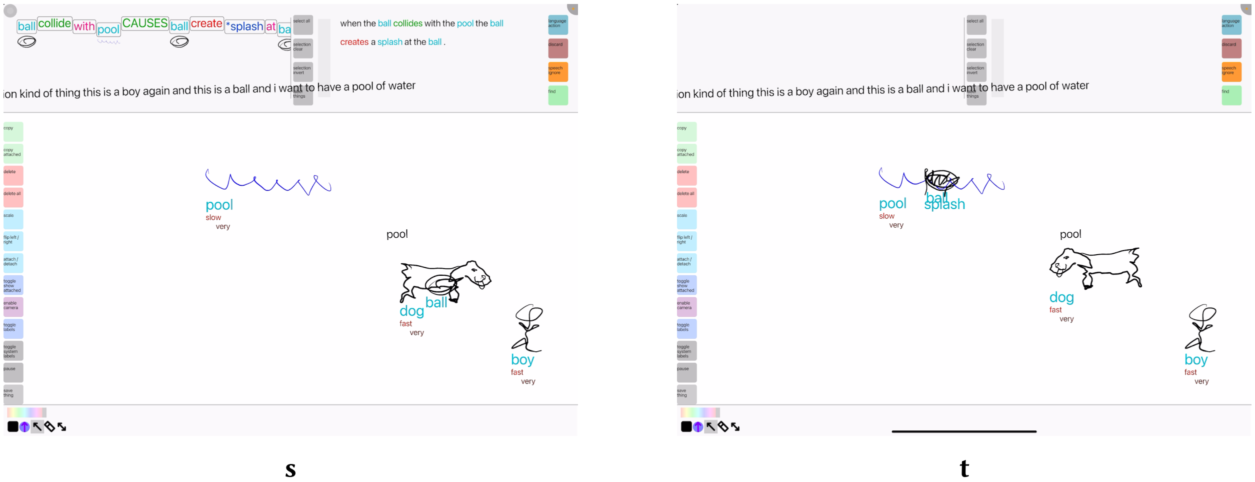
q



when the boy collides with the obstacle the

score increases .

r

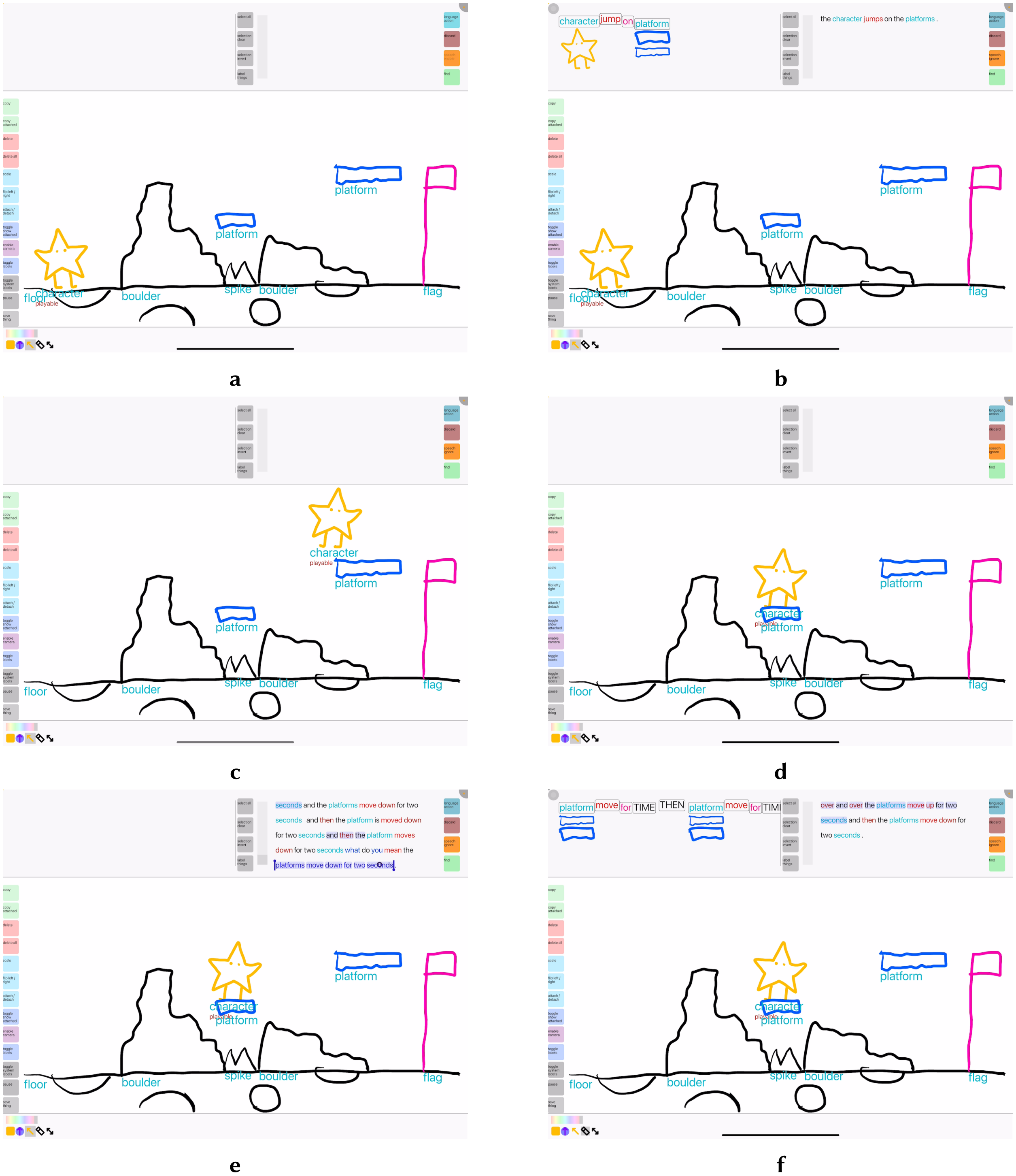


when the ball collides with the pool the ball creates a splash at the ball .

ion kind of thing this is a boy again and this is a ball and i want to have a pool of water

Figure A.3: P3$_{expl}$

Session P4$_{expl}$



a

b

the character jumps on the platforms .

c

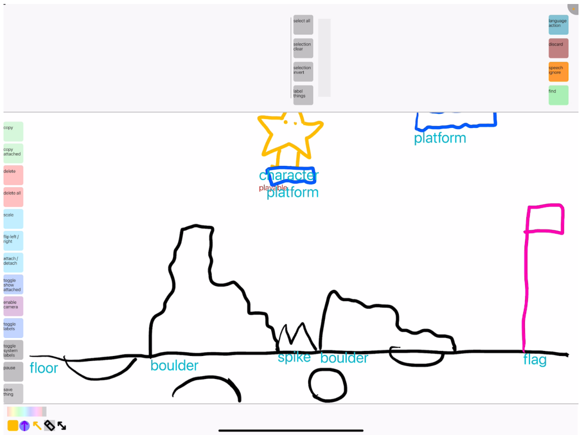
d

e

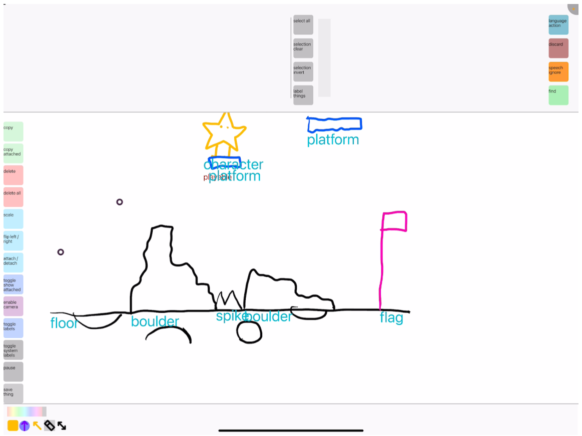
seconds and the platforms move down for two seconds and then the platform is moved down for two seconds and then the platform moves down for two seconds what do you mean the platforms move down for two seconds.

f

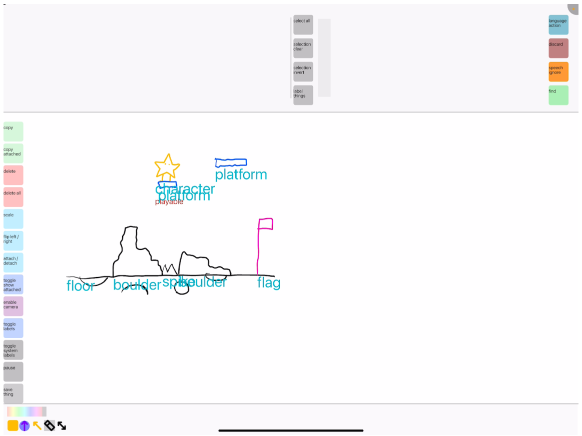
over and over the platforms move up for two seconds and then the platforms move down for two seconds .



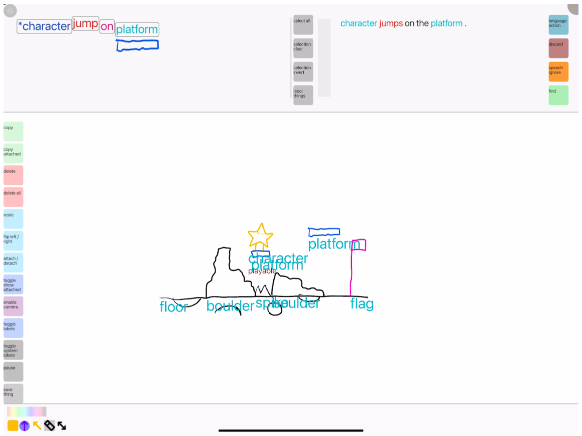
g



h

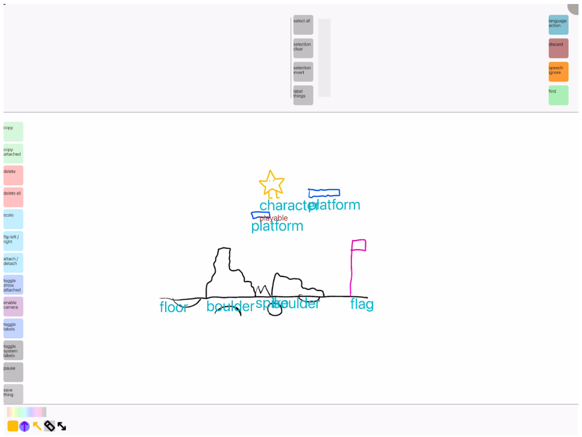


i

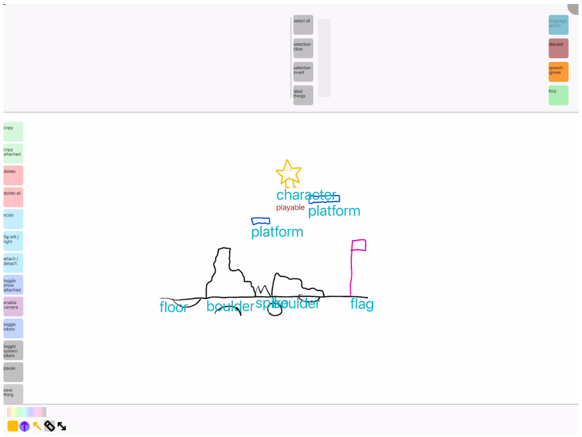

*character jump on platform

character jumps on the platform .

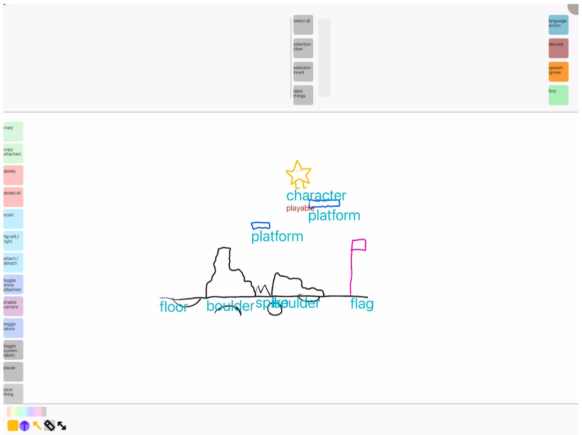
j



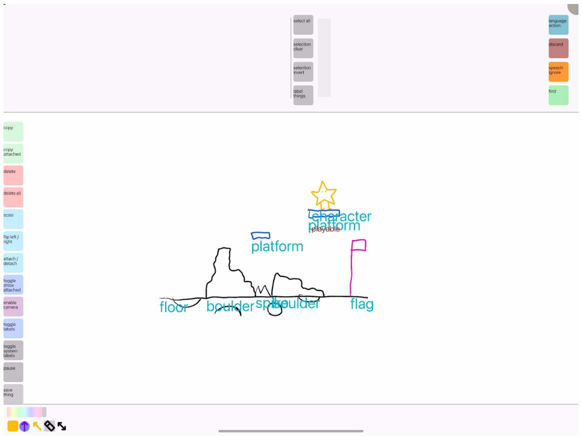
k



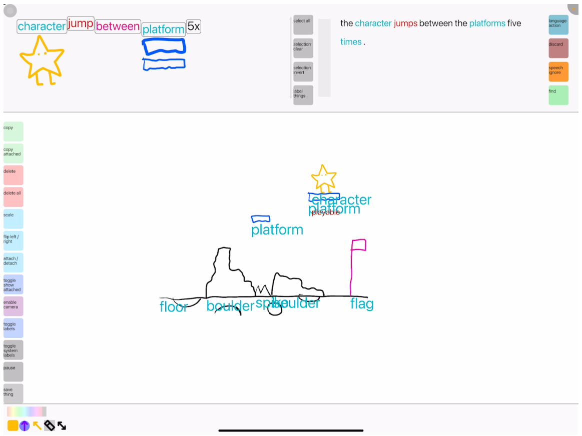
l



m

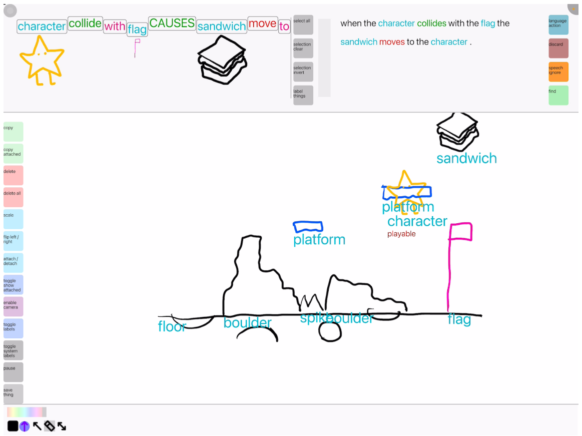


n



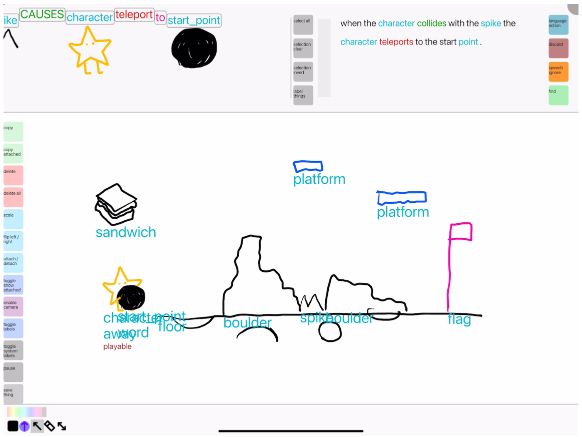
the character jumps between the platforms five times .

o



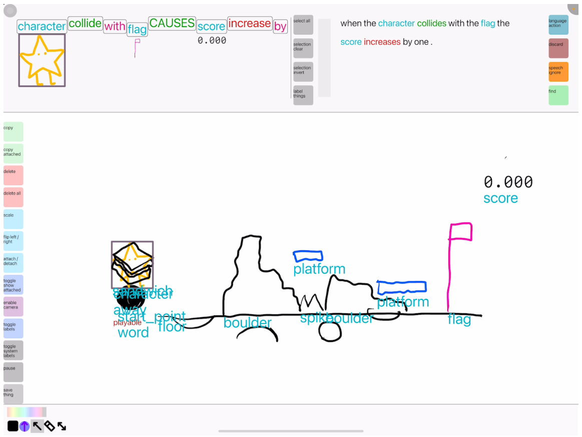
when the character collides with the flag the sandwich moves to the character .

p



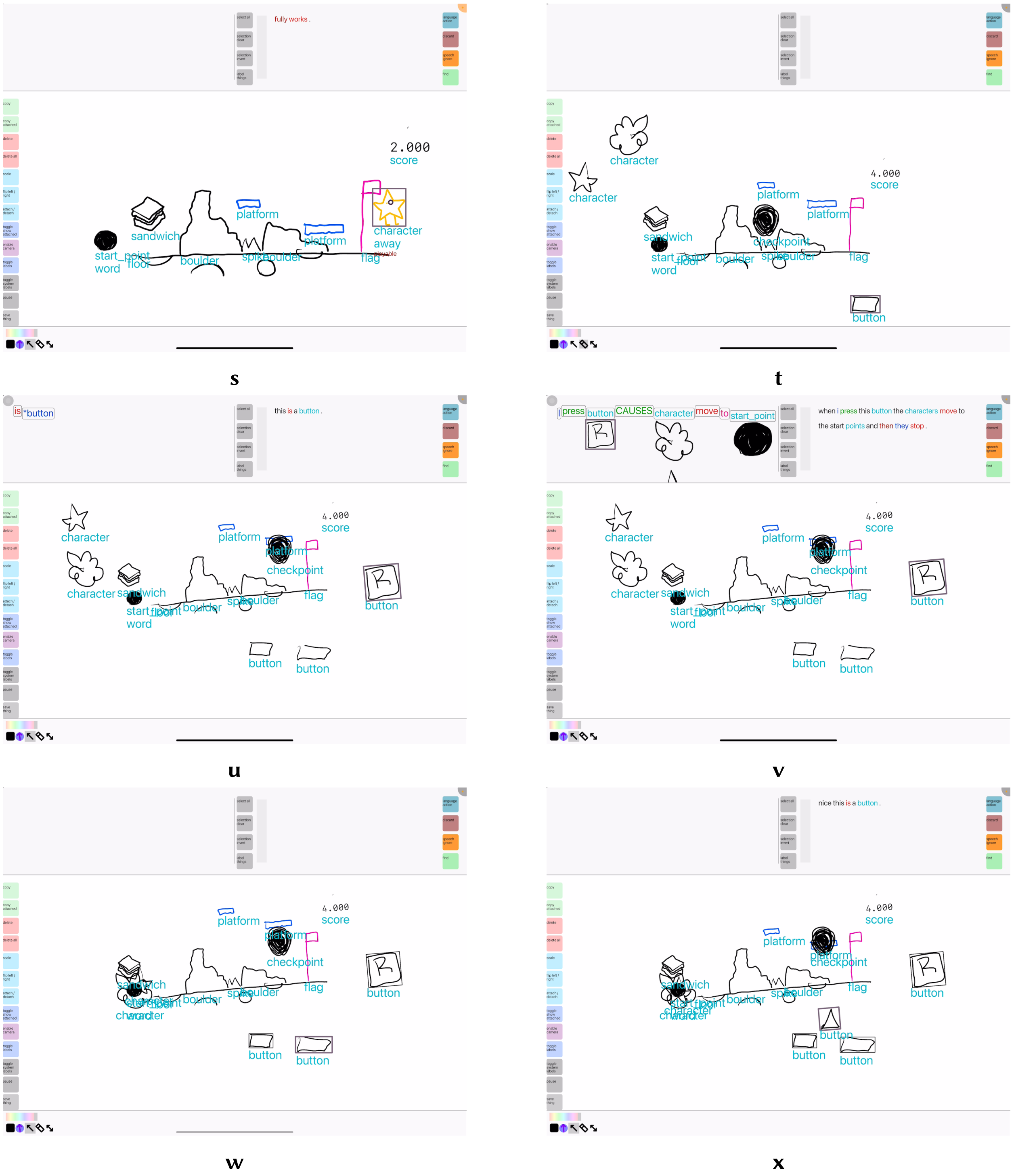
when the character collides with the spike the character teleports to the start point .

q



when the character collides with the flag the score increases by one .

r


s

fully works .

t

u

this is a button .

v

when i press this button the characters move to the start points and then they stop .

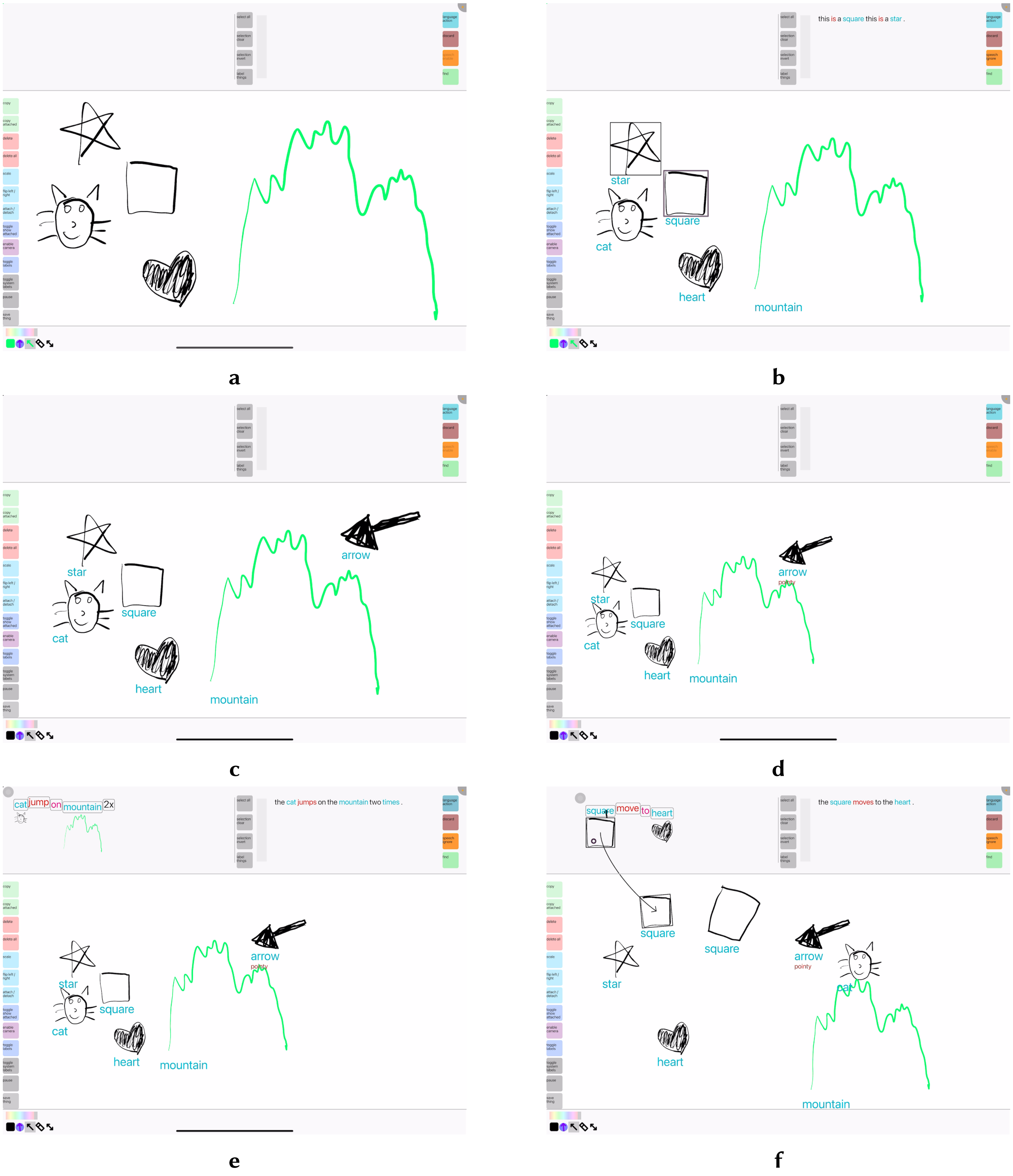
w

x

nice this is a button .

Figure A.4: P4$_{expl}$

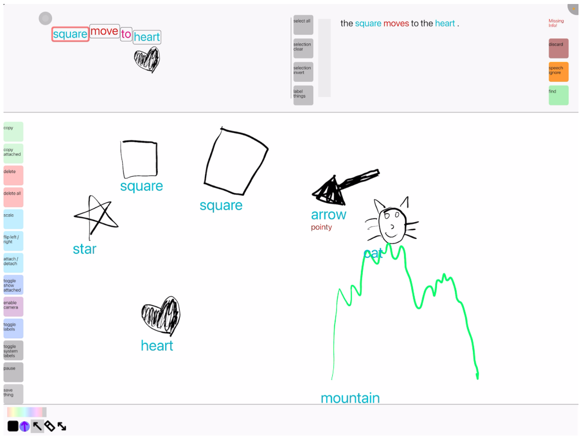
Session P5$_{expl}$



this is a square this is a star .

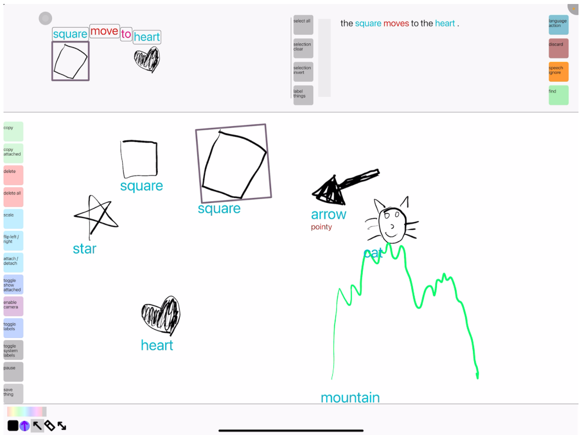
the cat jumps on the mountain two times .

the square moves to the heart .



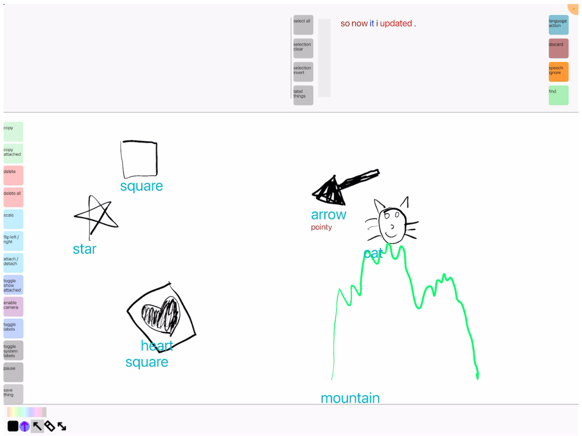
the square moves to the heart .

g



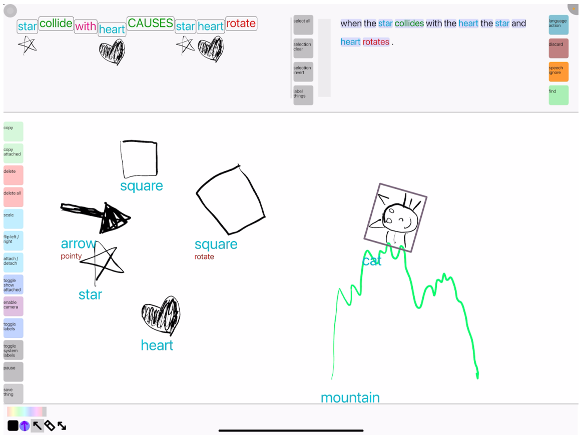
the square moves to the heart .

h



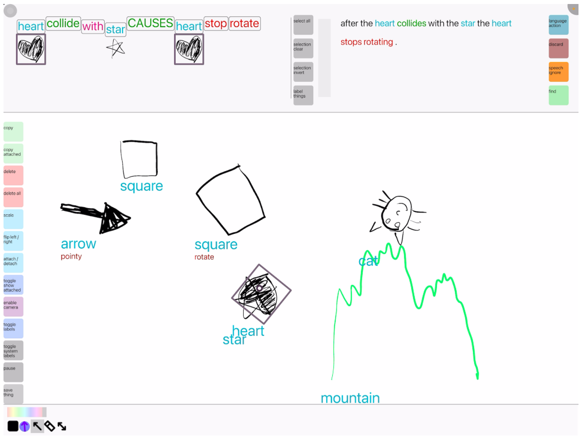
so now it i updated .

i



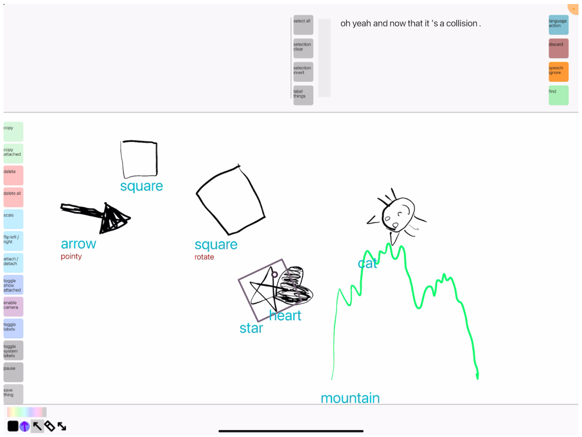
when the star collides with the heart the star and heart rotates .

j



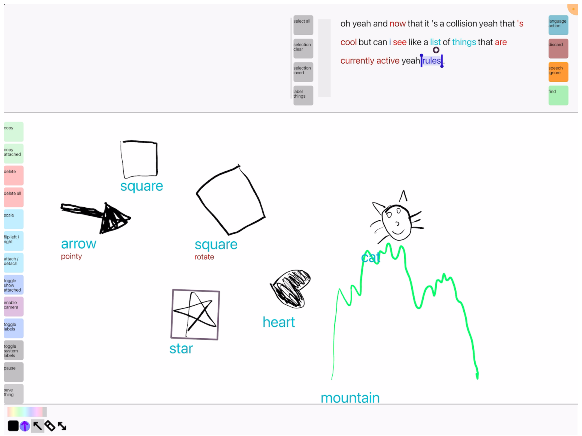
after the heart collides with the star the heart stops rotating .

k

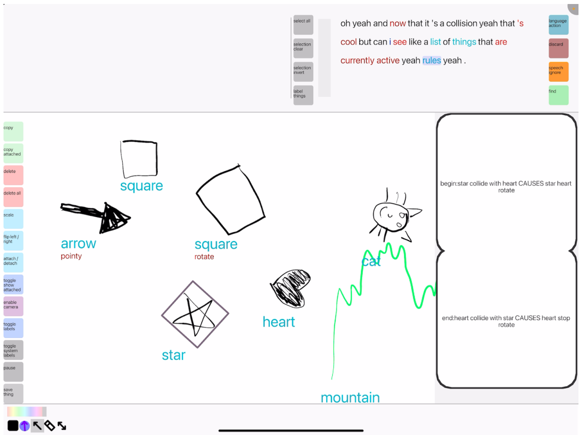


oh yeah and now that it 's a collision .

l


oh yeah and now that it 's a collision yeah that 's cool but can i see like a list of things that are currently active yeah rules

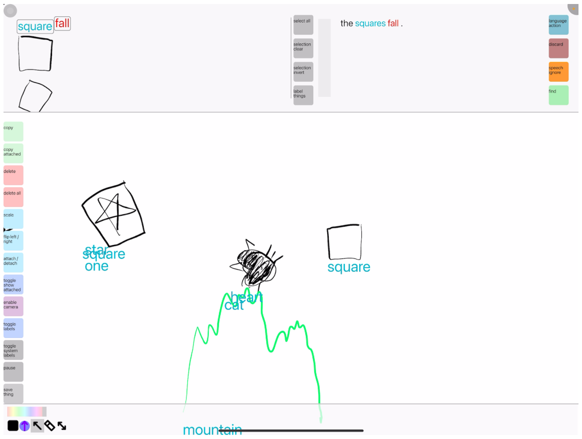
m


oh yeah and now that it 's a collision yeah that 's cool but can i see like a list of things that are currently active yeah rules yeah .

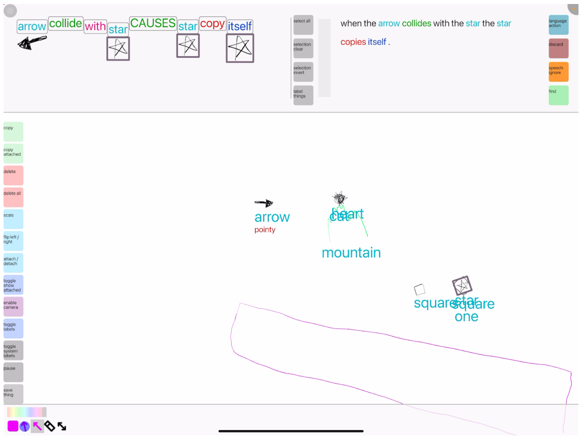
begin:star collide with heart CAUSES star heart rotate

end:heart collide with star CAUSES heart stop rotate

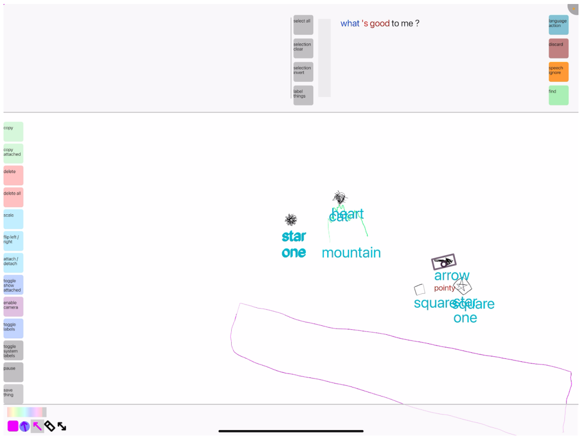
n


the squares fall .

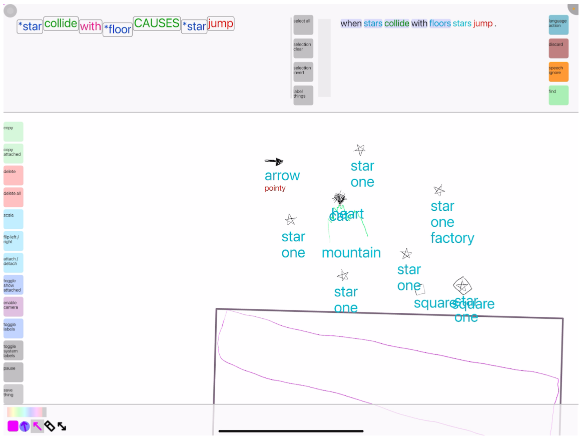
o


when the arrow collides with the star the star copies itself .

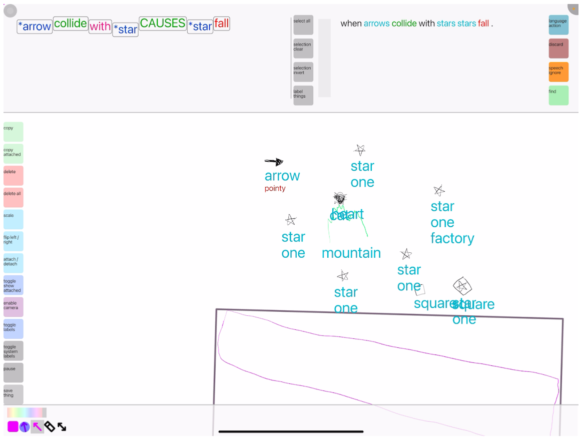
p


what 's good to me ?

q


when stars collide with floors stars jump .

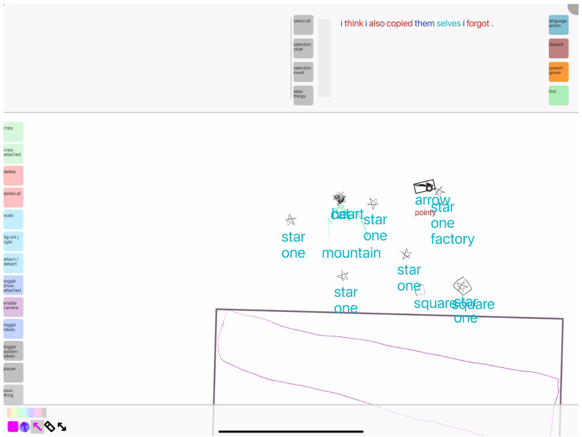
r



*arrow collide with *star CAUSES *star fall

when arrows collide with stars stars fall .

s

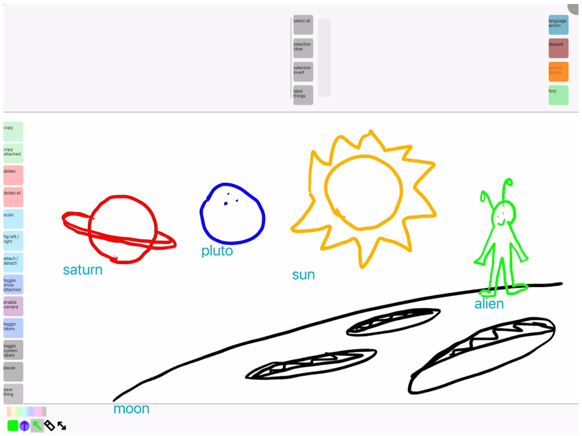


i think i also copied them selves i forgot .

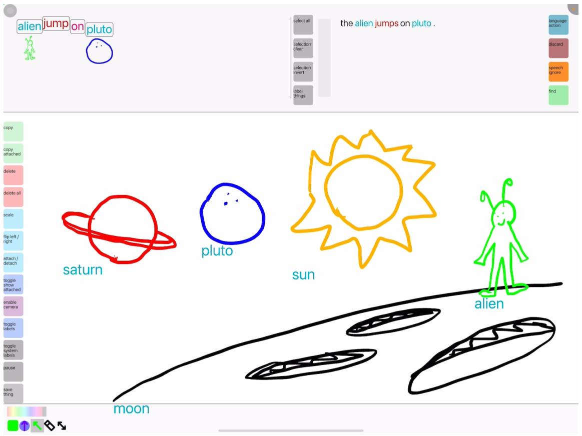
t

Figure A.5: P5$_{expl}$

Session P6$_{expl}$

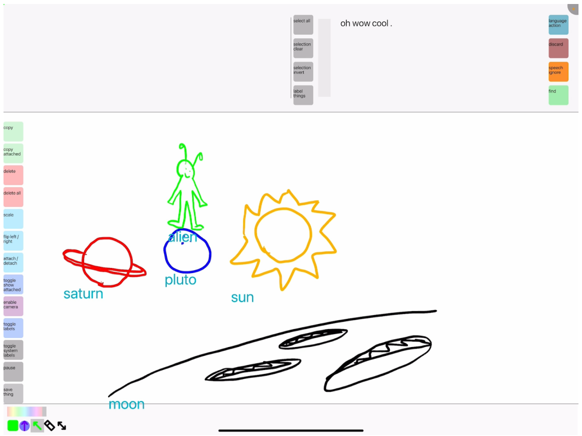


a



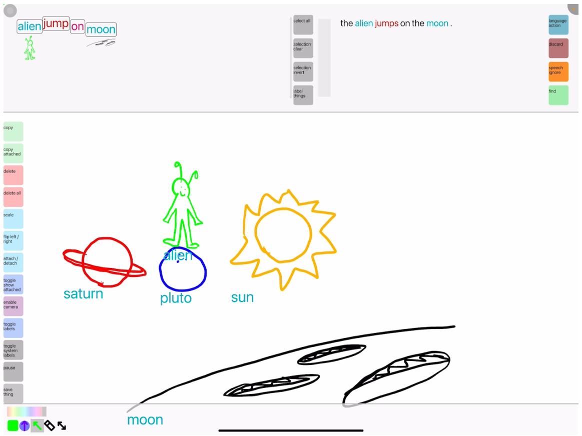
the alien jumps on pluto .

b



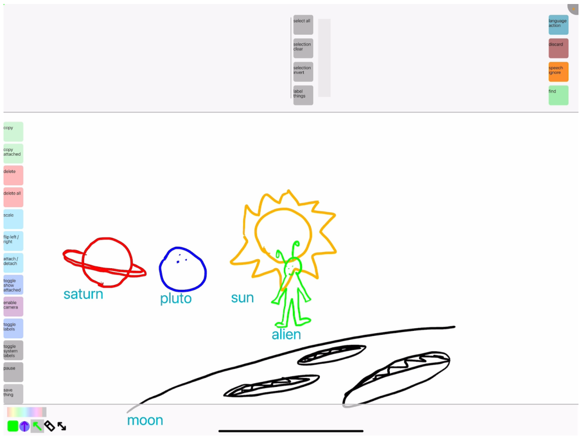
oh wow cool .

c

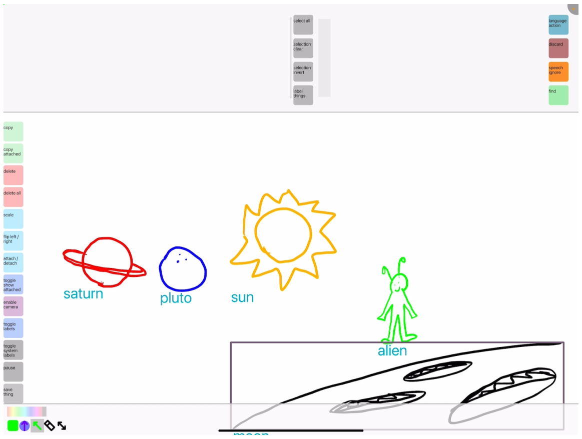


the alien jumps on the moon .

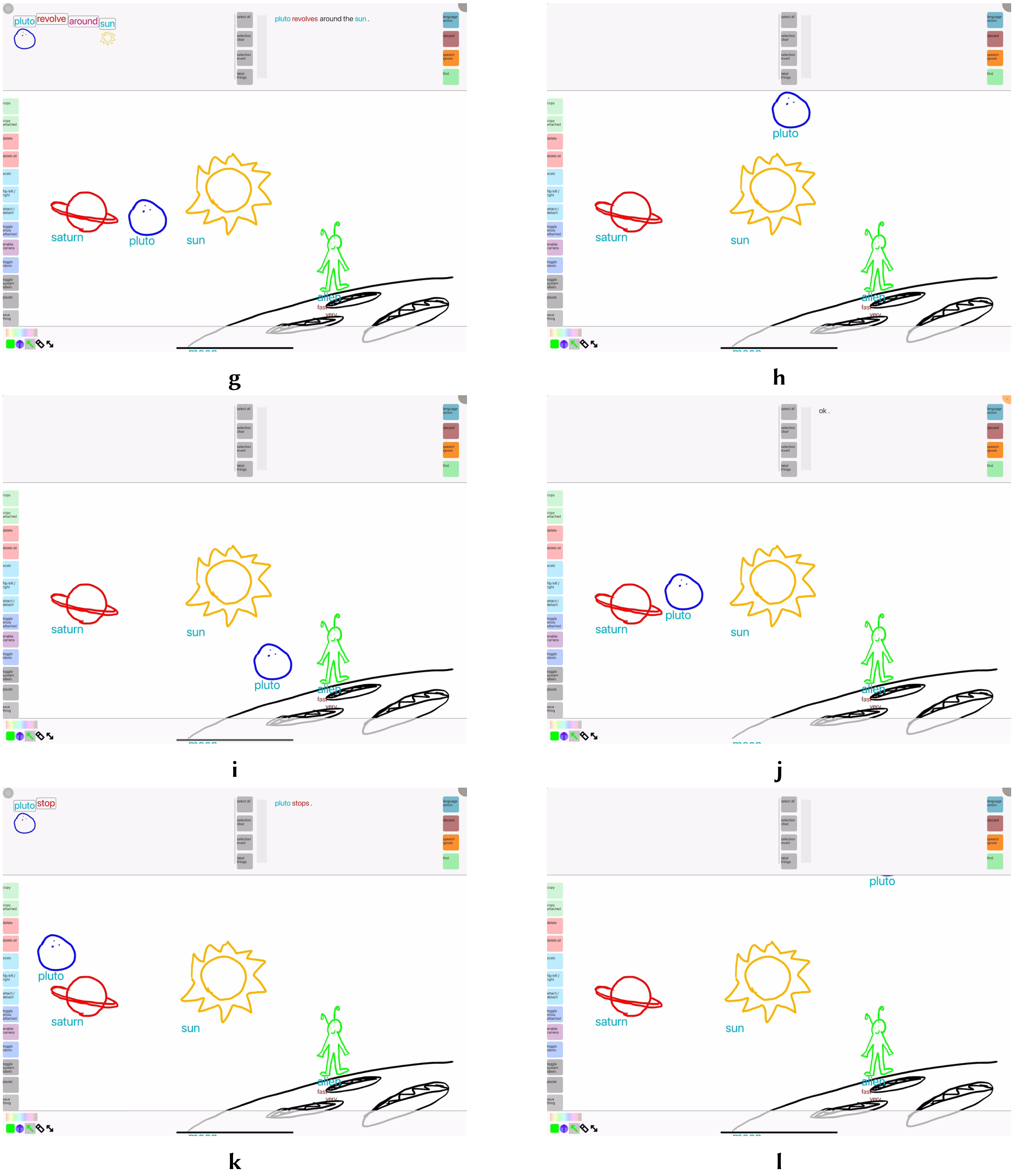
d



e



f



pluto revolves around the sun .

g

h

i

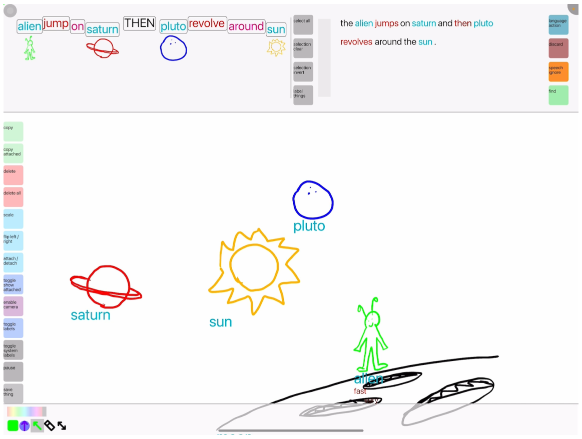
ok .

j

pluto stops .

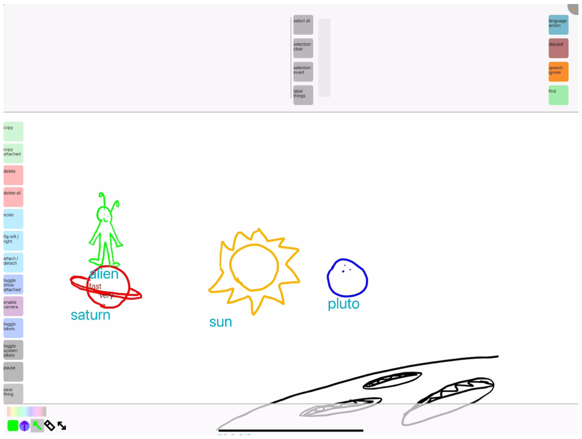
k

l

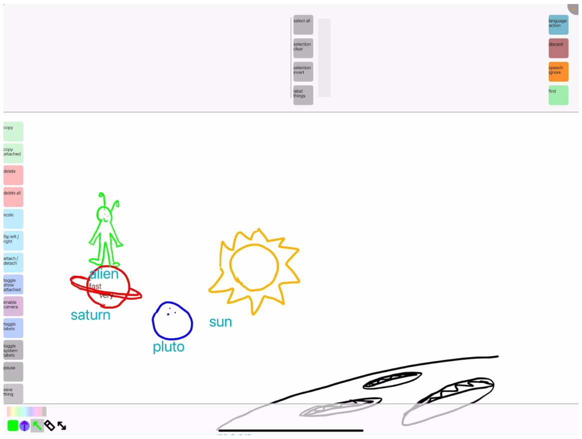


the alien jumps on saturn and then pluto revolves around the sun .

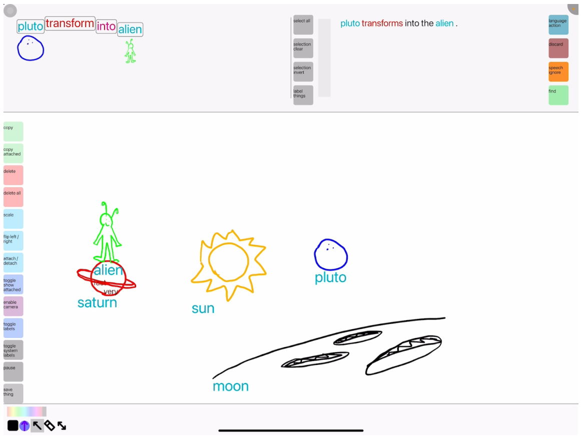
m



n

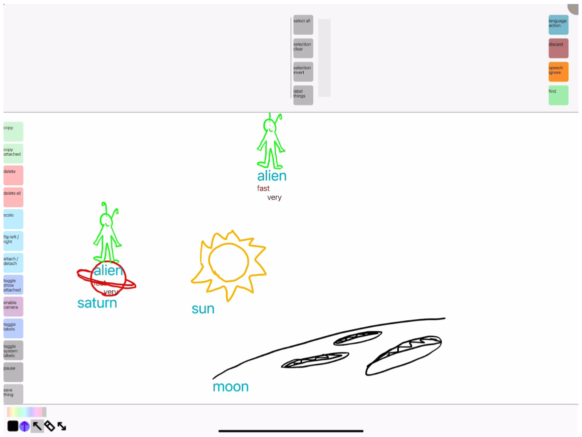


o

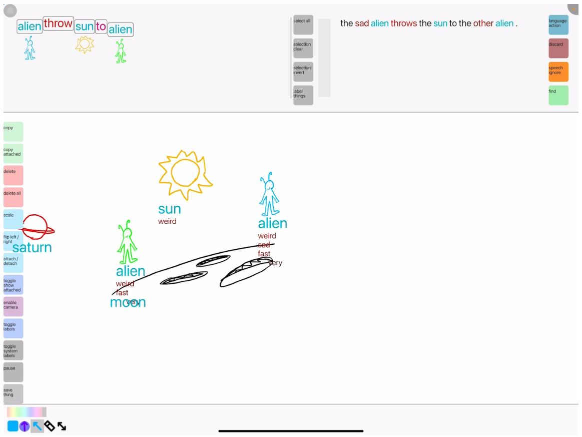


pluto transforms into the alien .

p

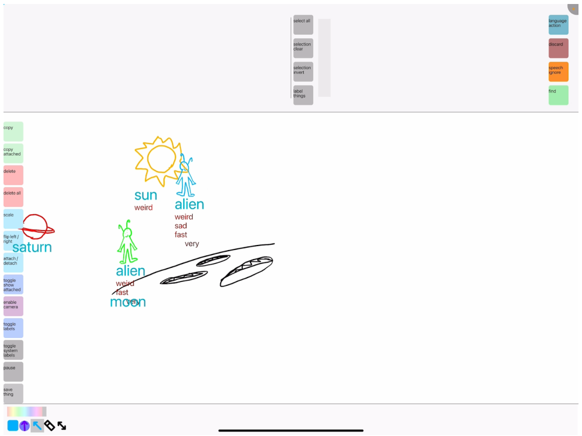


q

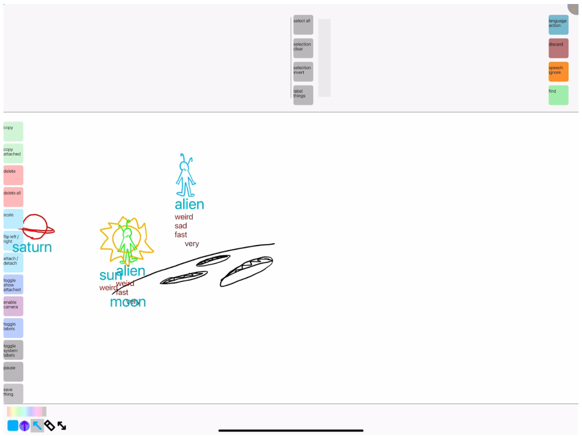


the sad alien throws the sun to the other alien .

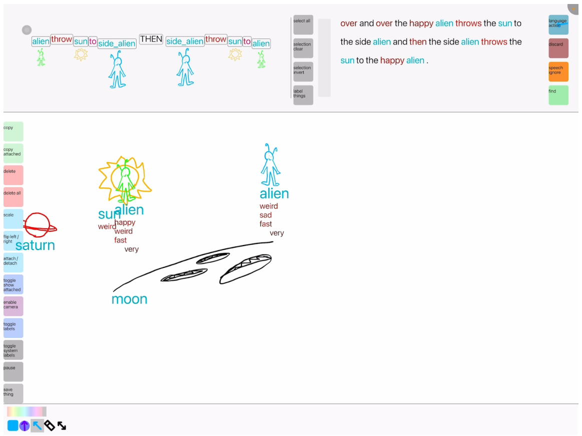
r



s

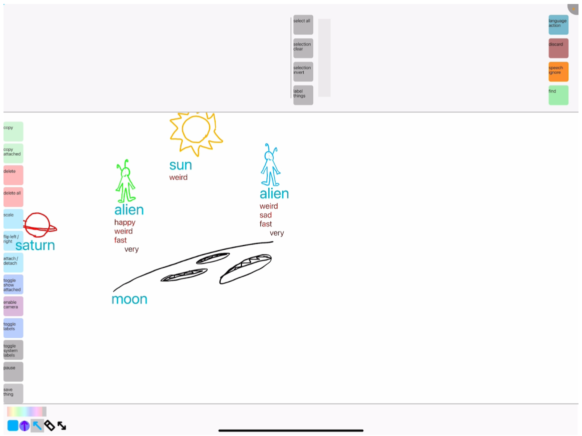


t

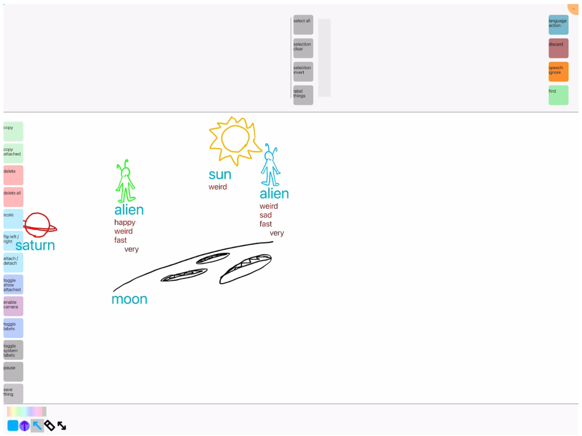

alien throw sun to side_alien THEN side_alien throw sun to alien
sun
weird
alien
happy
weird
fast
very
alien
weird
sad
fast
very
saturn
moon

over and over the happy alien throws the sun to the side alien and then the side alien throws the sun to the happy alien .

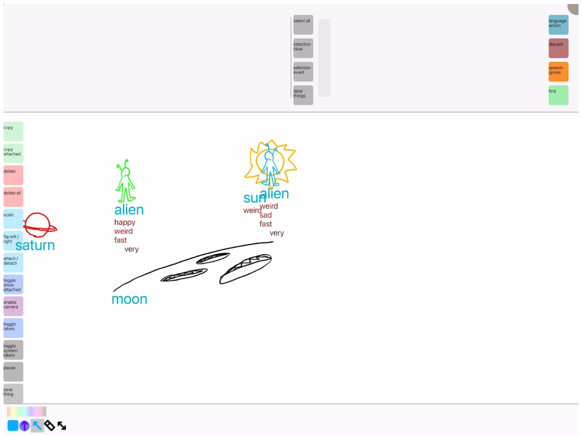
u



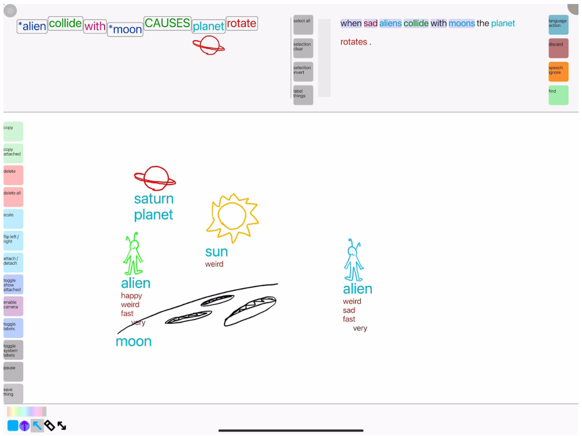
v



w

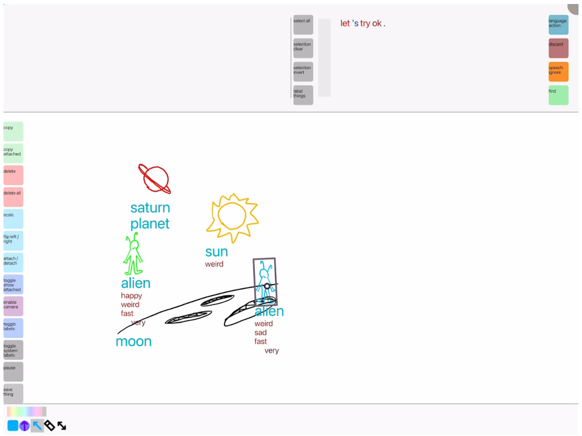


x



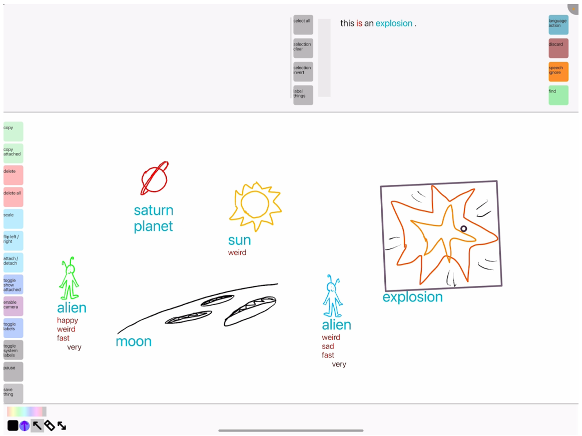
when sad aliens collide with moons the planet rotates .

y



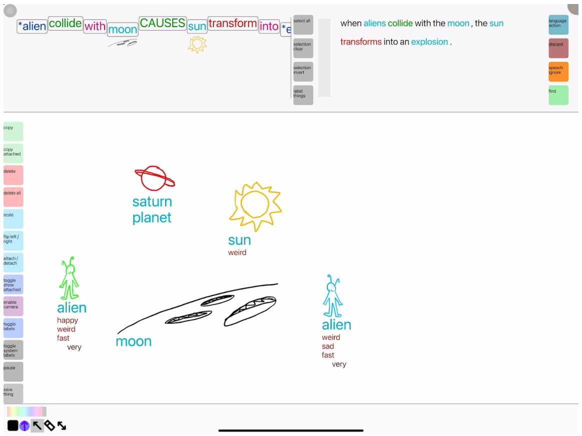
let 's try ok .

z



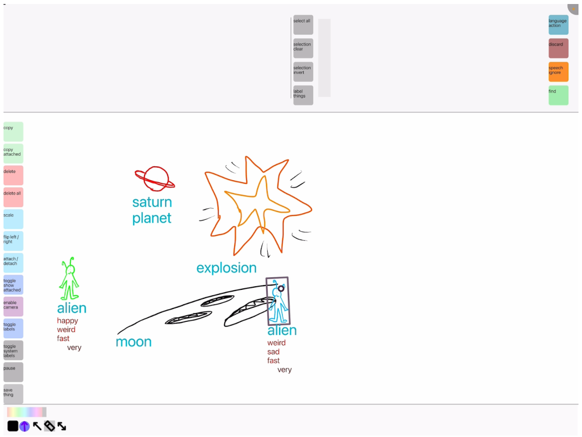
this is an explosion .

aa

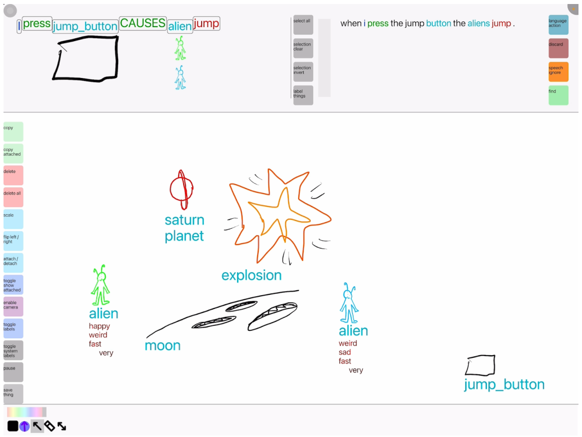


when aliens collide with the moon , the sun transforms into an explosion .

ab

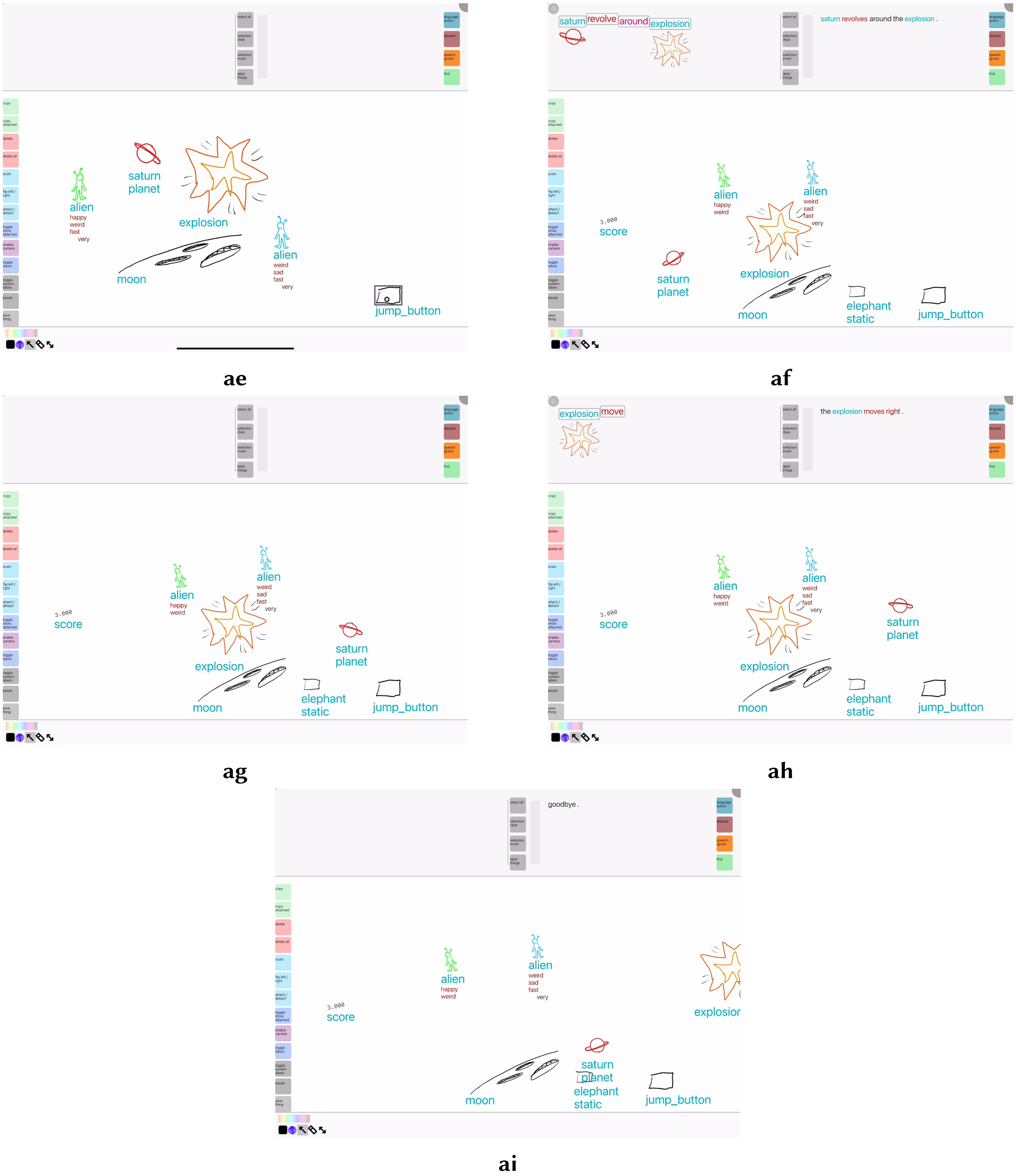


ac



when i press the jump button the aliens jump .

ad



ae

saturn revolves around the explosion .

af

ag

the explosion moves right .

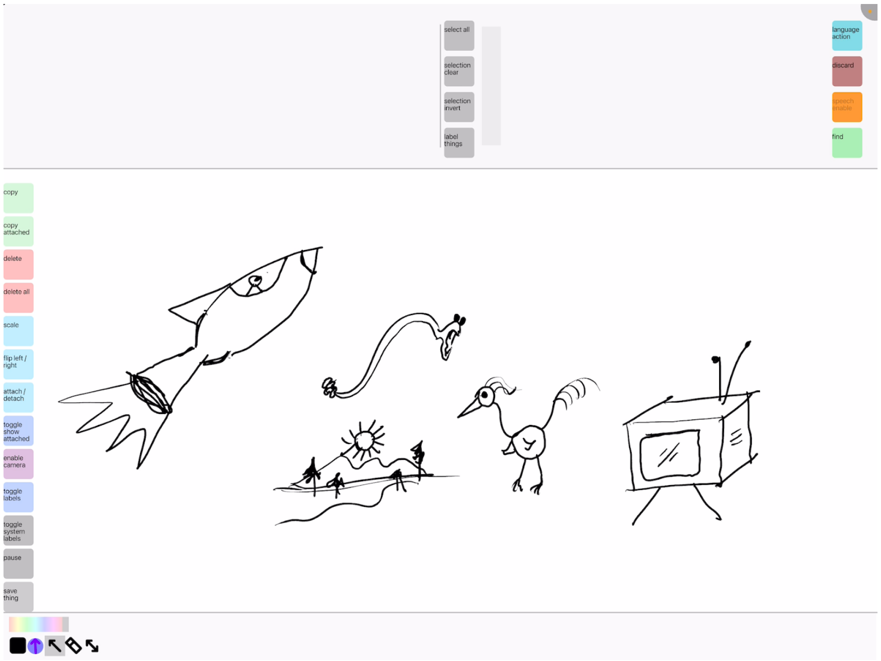
ah

goodbye .

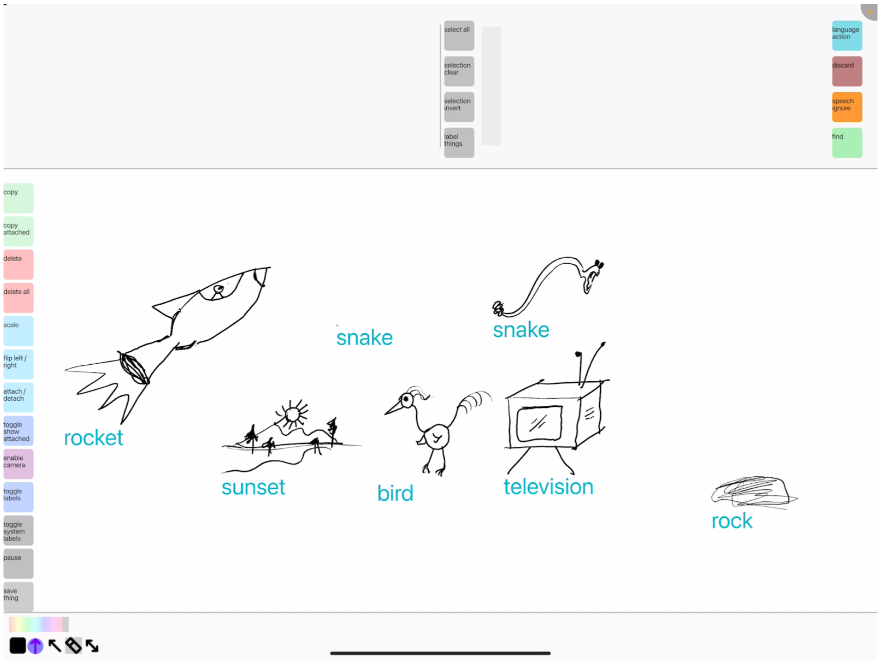
ai

Figure A.6: P6$_{expl}$

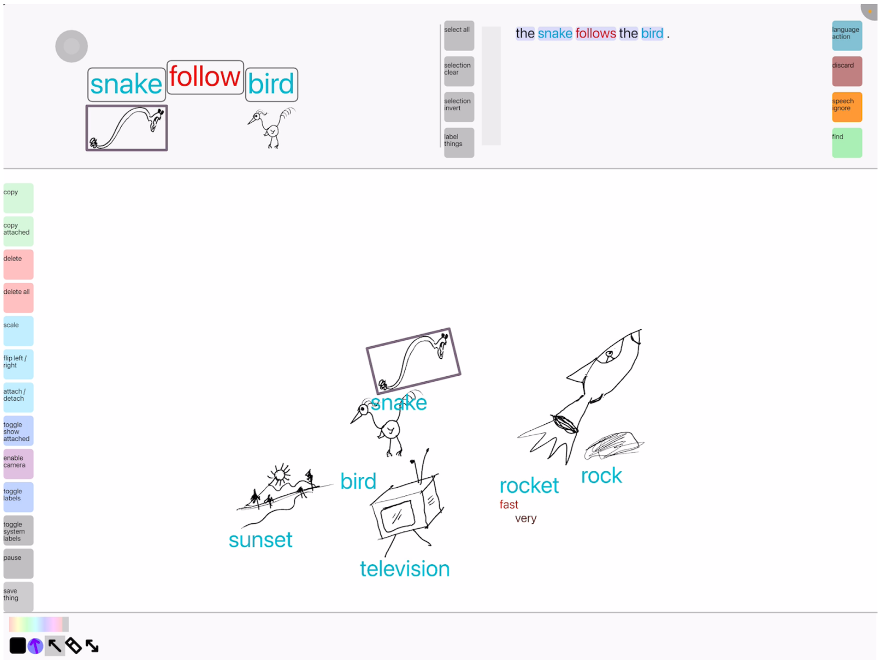
Session P7$_{expl}$



a

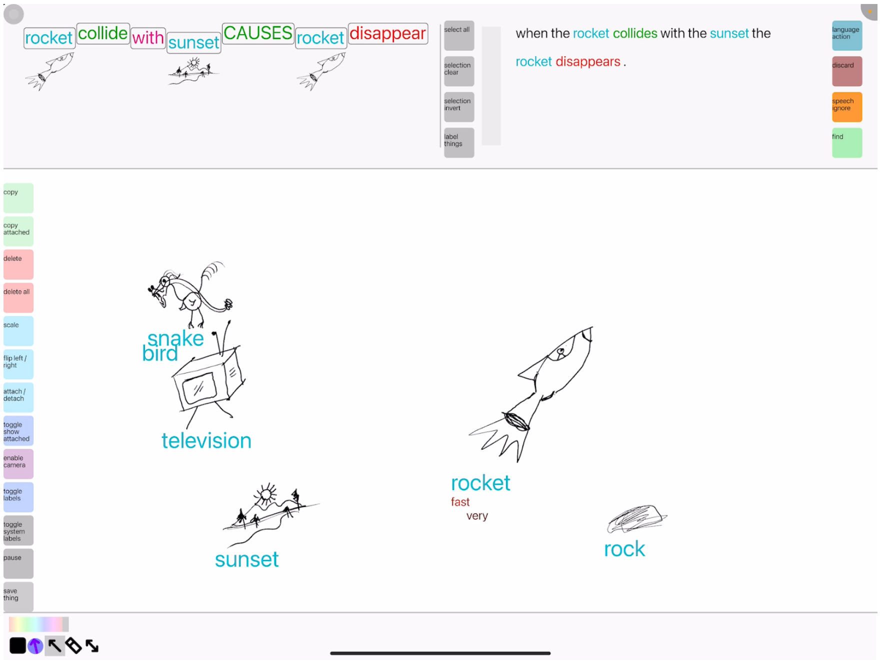


b



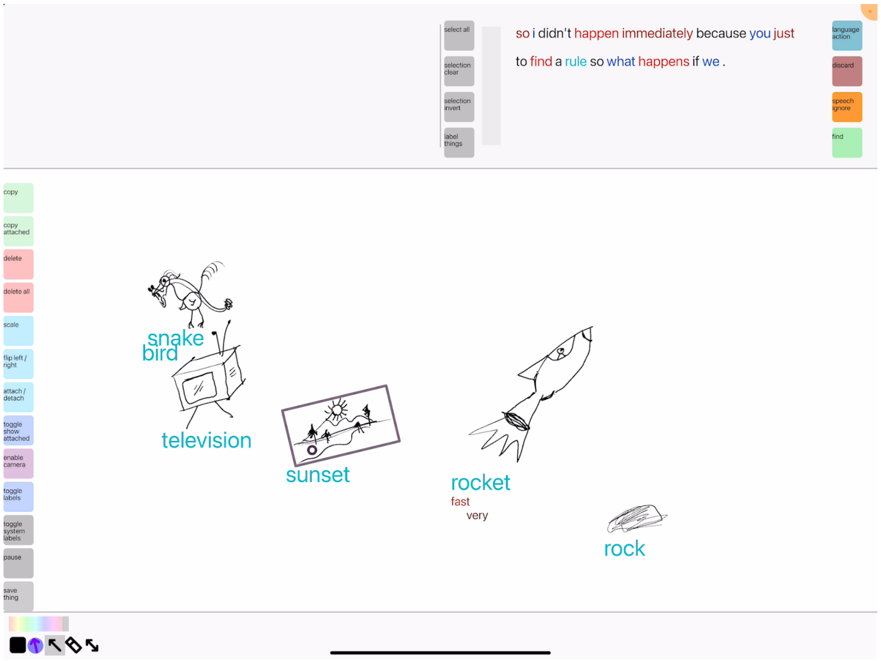
the snake follows the bird .

c



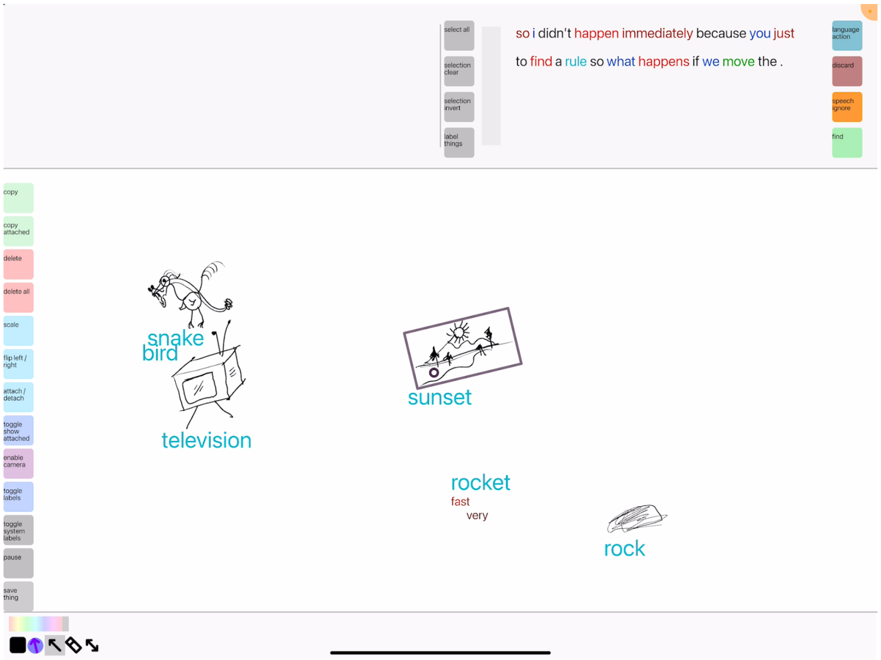
when the rocket collides with the sunset the rocket disappears .

d



so i didn't happen immediately because you just to find a rule so what happens if we .

e

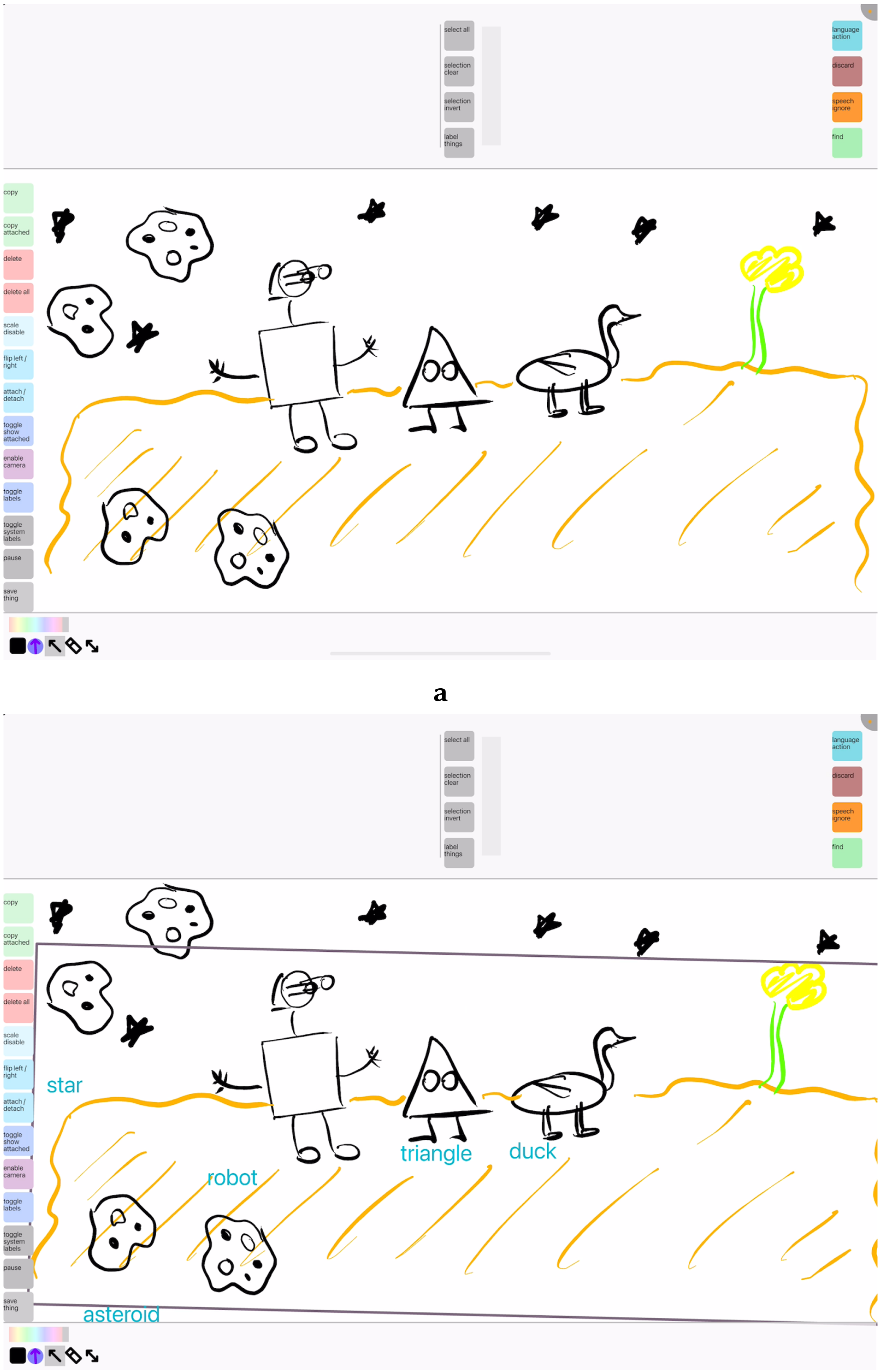


so i didn't happen immediately because you just to find a rule so what happens if we move the .

f

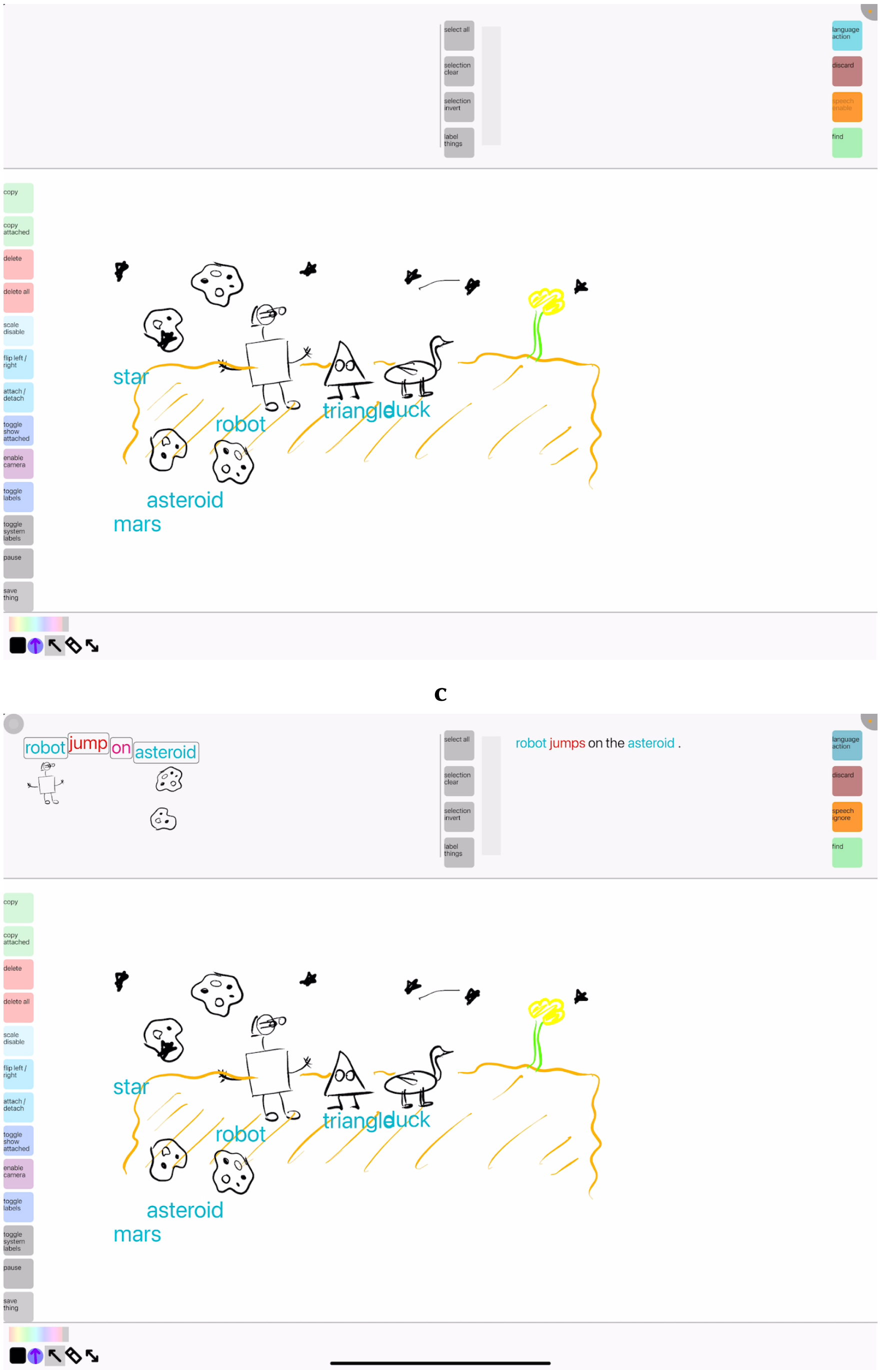
Figure A.7: P7$_{expl}$

Session P8$_{expl}$



a

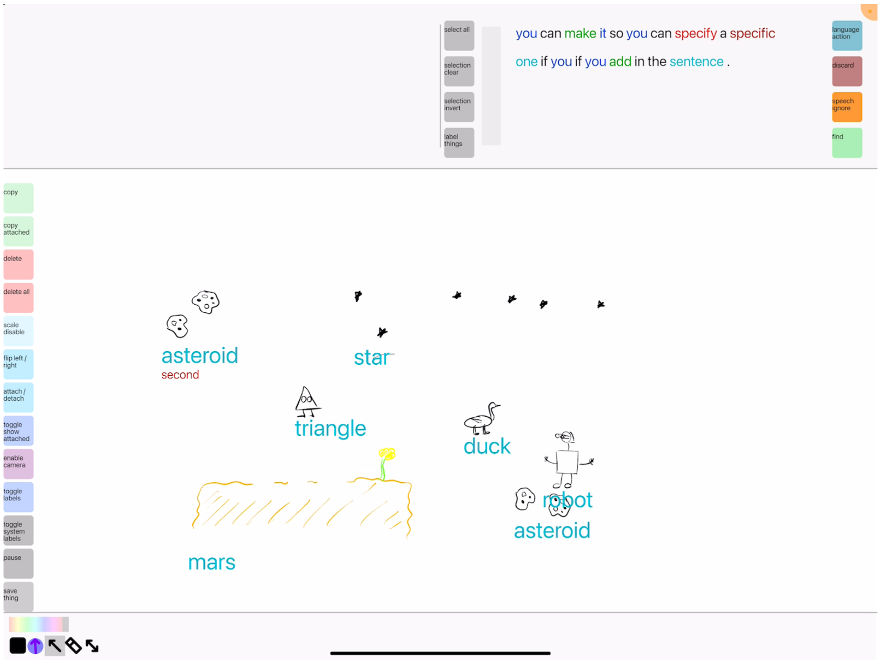
b



c

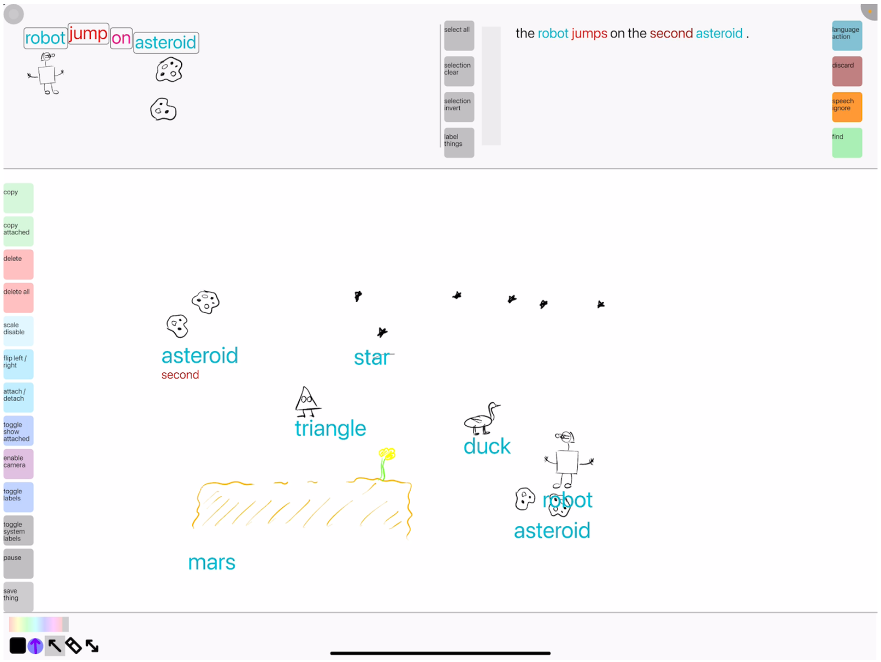
robot jumps on the asteroid .

d



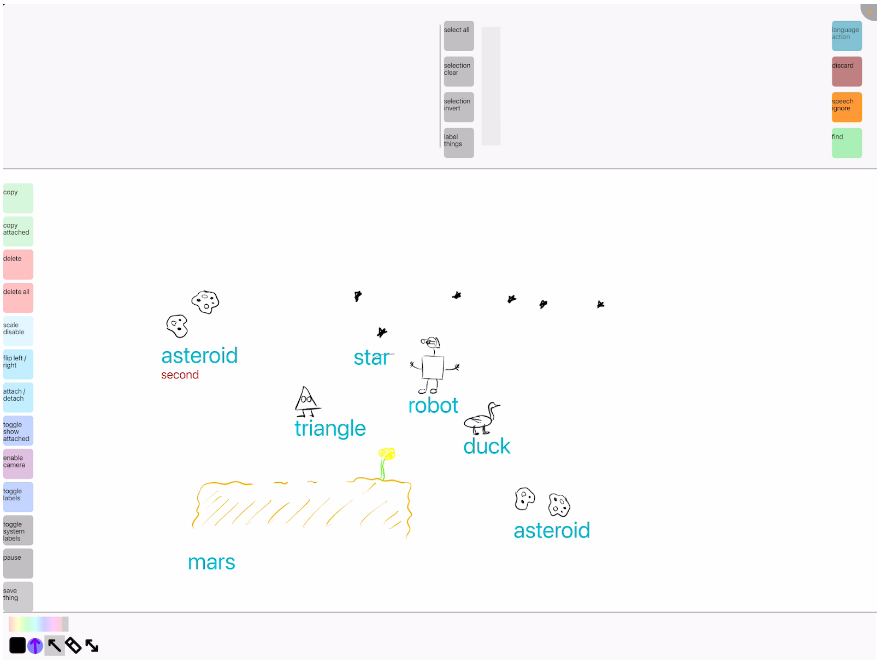
you can make it so you can specify a specific one if you if you add in the sentence .

e

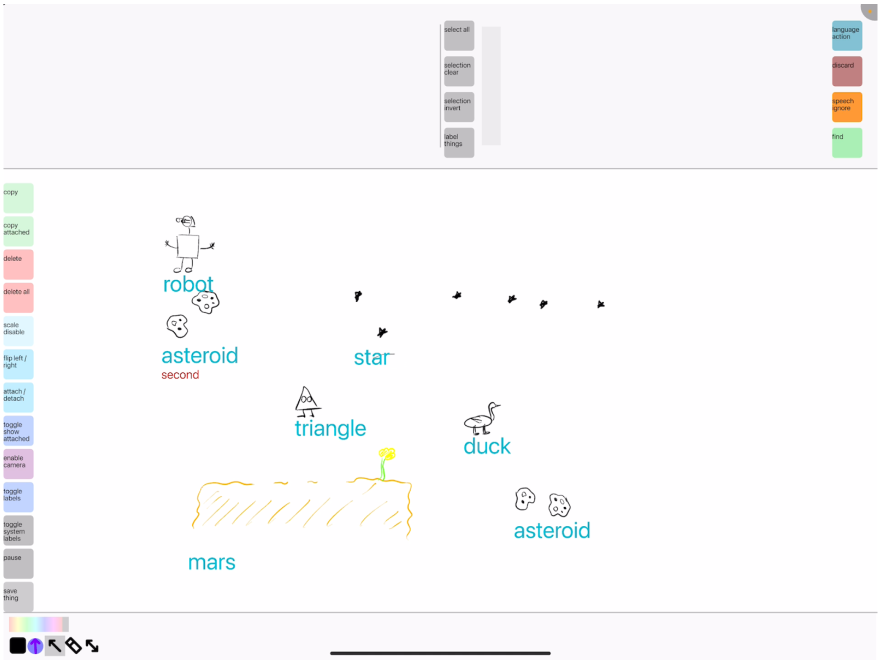


the robot jumps on the second asteroid .

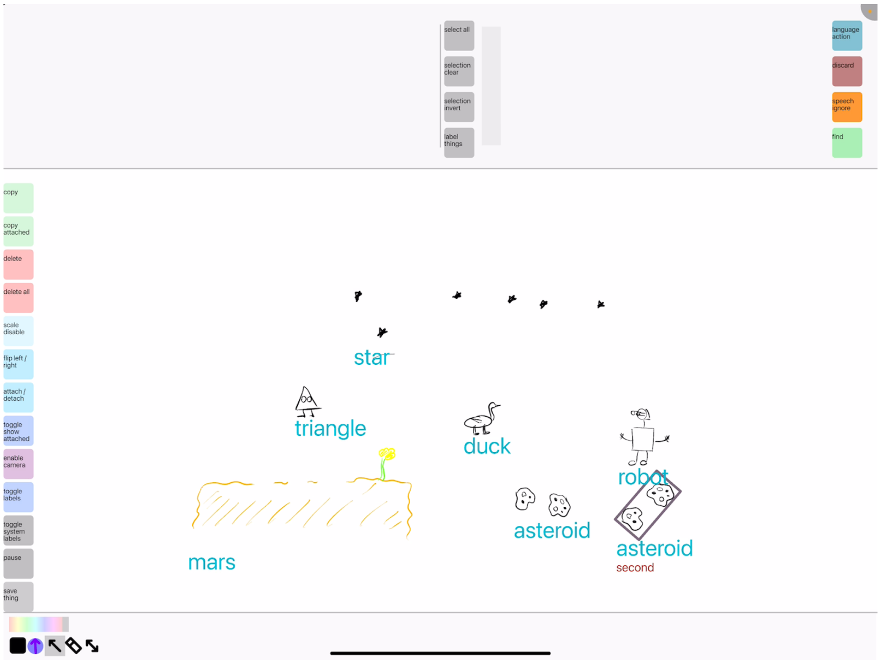
f


select all
selection clear
selection invert
label things
language action
discard
speech ignore
find
copy
copy attached
delete
delete all
scale disable
flip left / right
attach / detach
toggle show attached
enable camera
toggle labels
toggle system labels
pause
save thing
asteroid
second
star
robot
triangle
duck
mars
asteroid

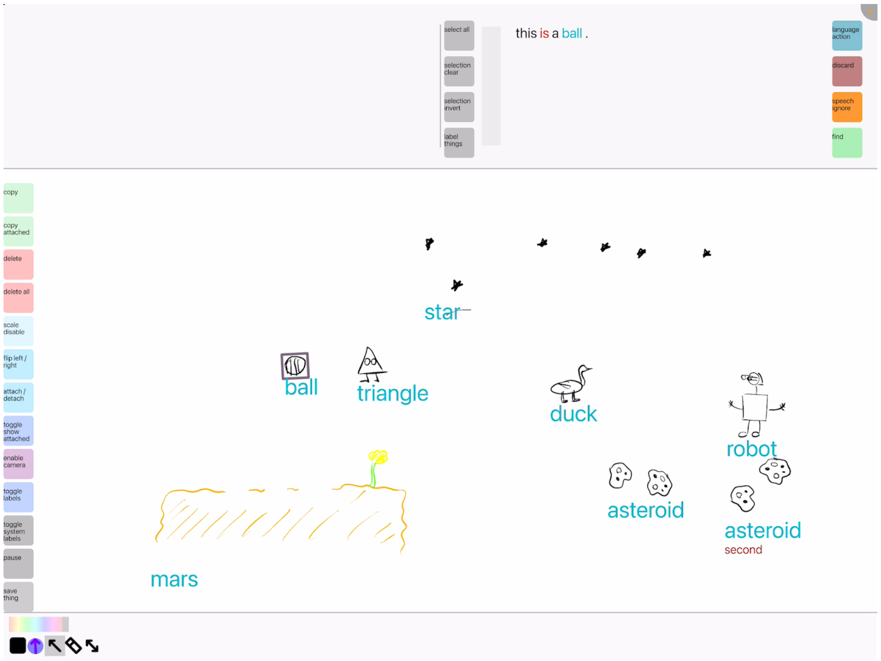
g


select all
selection clear
selection invert
label things
language action
discard
speech ignore
find
copy
copy attached
delete
delete all
scale disable
flip left / right
attach / detach
toggle show attached
enable camera
toggle labels
toggle system labels
pause
save thing
robot
asteroid
second
star
triangle
duck
mars
asteroid

h

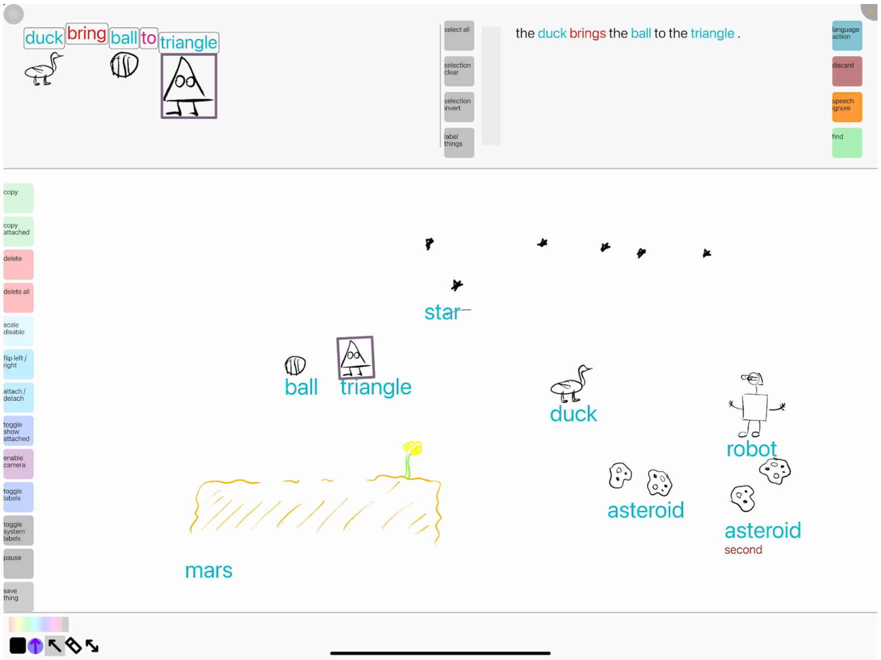


i



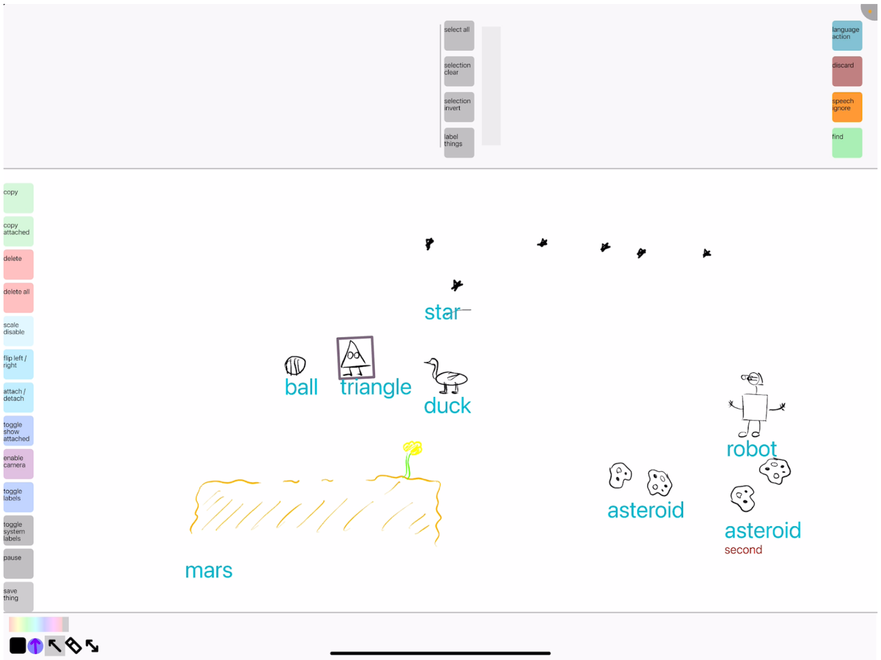
this is a ball .

j

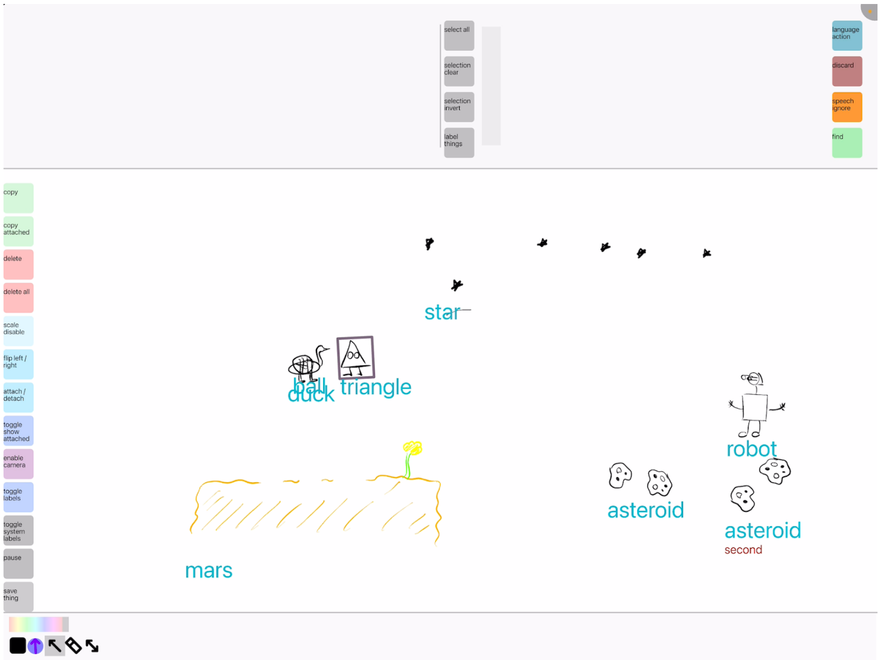


the duck brings the ball to the triangle .

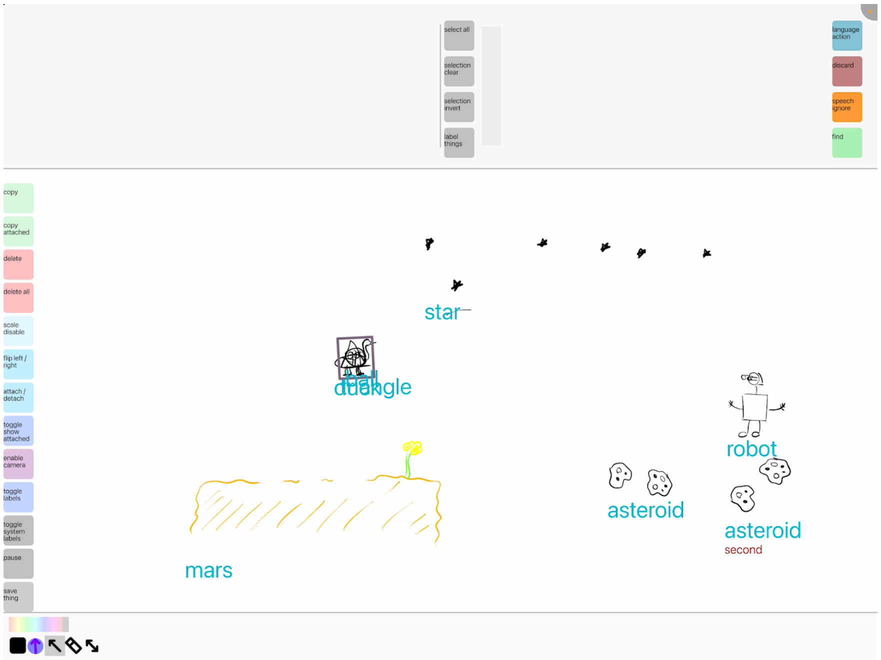
k



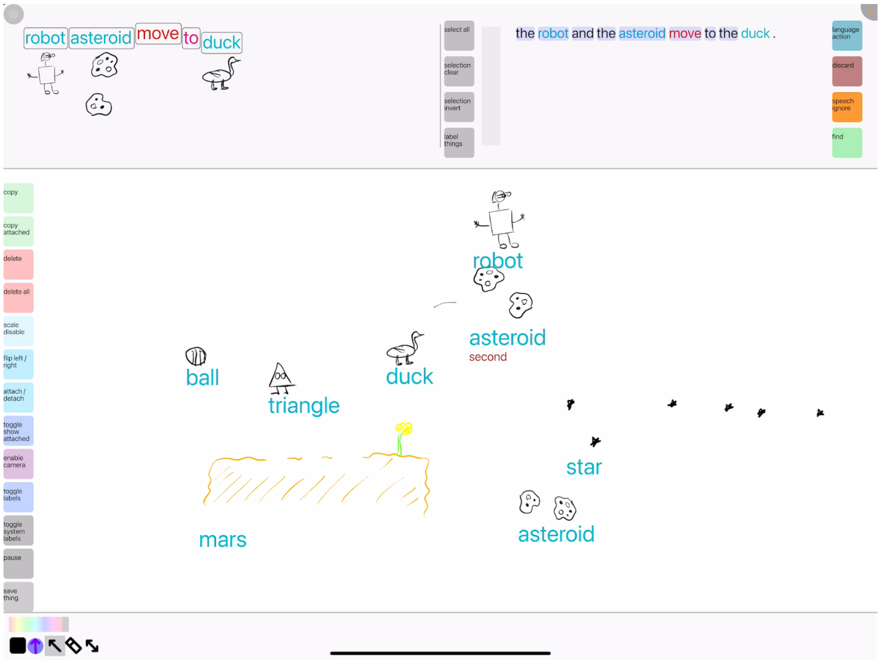
l


select all
selection clear
selection invert
label things
language action
discard
speech ignore
find
copy
copy attached
delete
delete all
scale disable
flip left / right
attach / detach
toggle show attached
enable camera
toggle labels
toggle system labels
pause
save thing
star
ball
duck
triangle
robot
asteroid
asteroid
second
mars

m

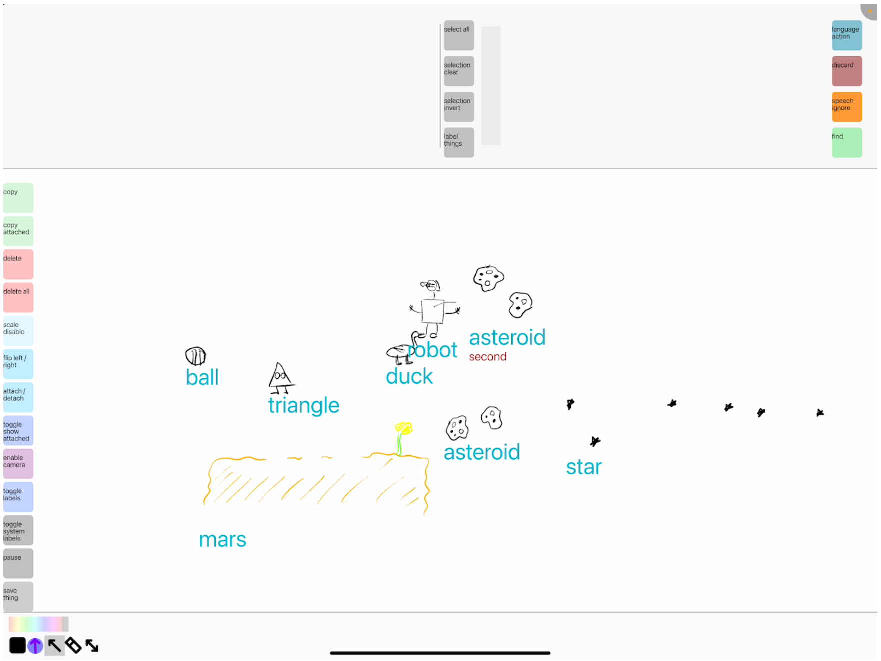

select all
selection clear
selection invert
label things
language action
discard
speech ignore
find
copy
copy attached
delete
delete all
scale disable
flip left / right
attach / detach
toggle show attached
enable camera
toggle labels
toggle system labels
pause
save thing
star
ball
duck
triangle
robot
asteroid
asteroid
second
mars

n

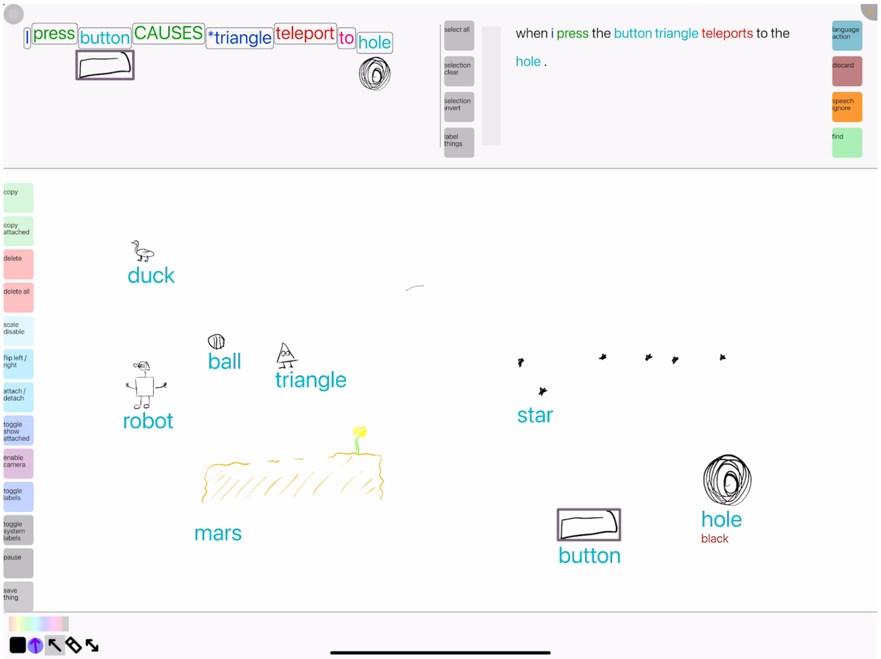


the robot and the asteroid move to the duck .

o



p

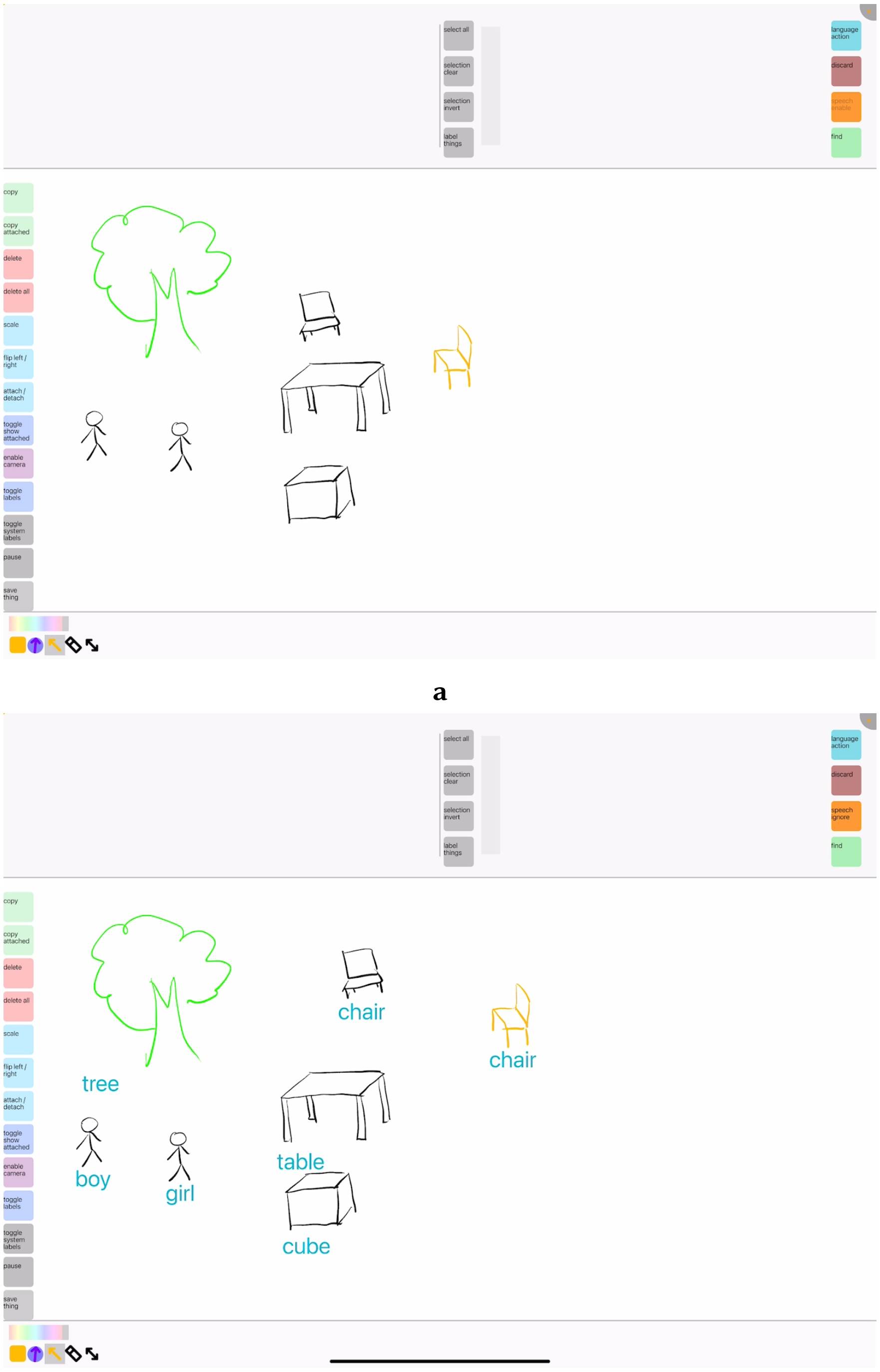


when i press the button triangle teleports to the hole .

q

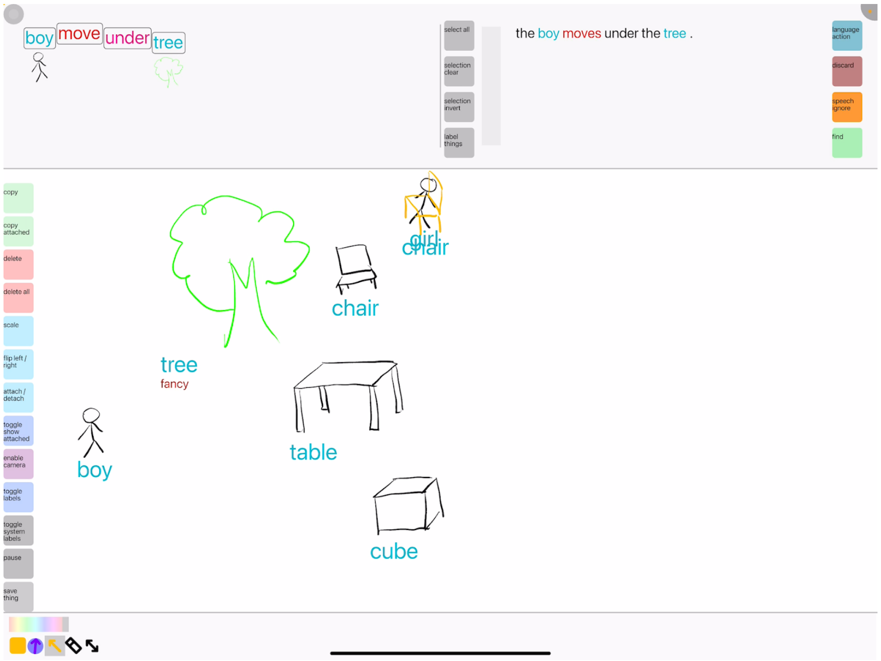
Figure A.8: P8$_{expl}$

Session P9$_{expl}$



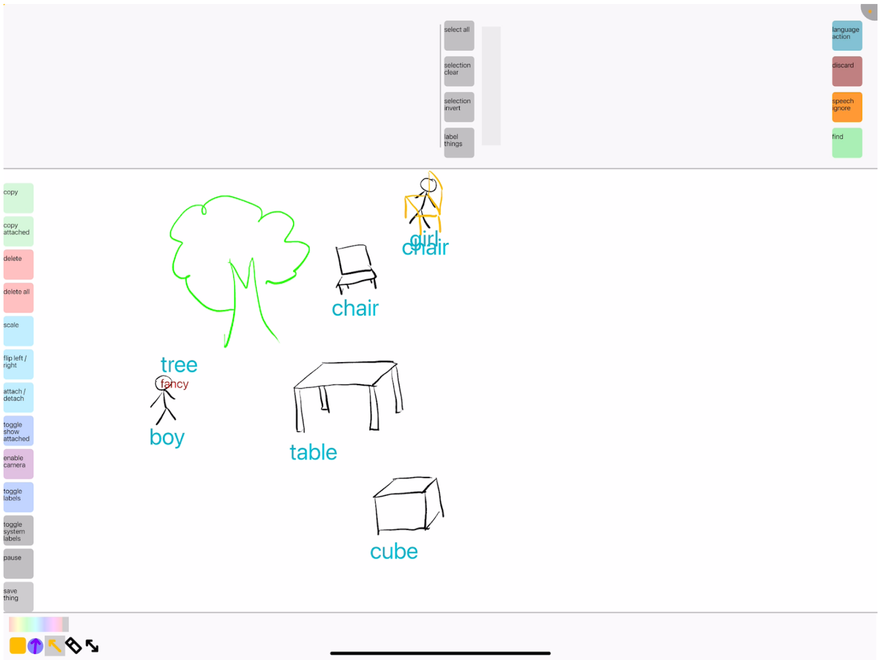
a

b

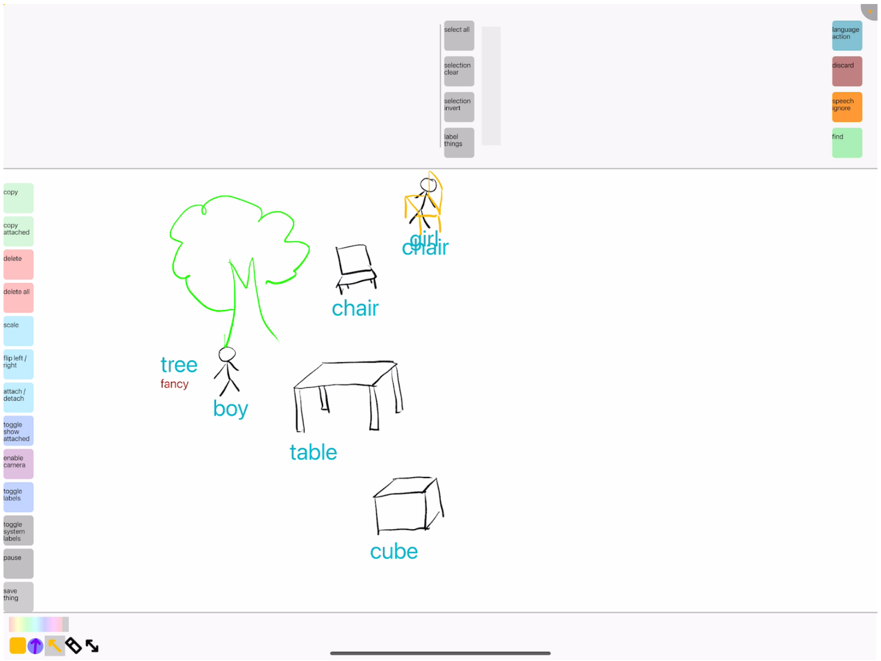


the boy moves under the tree .

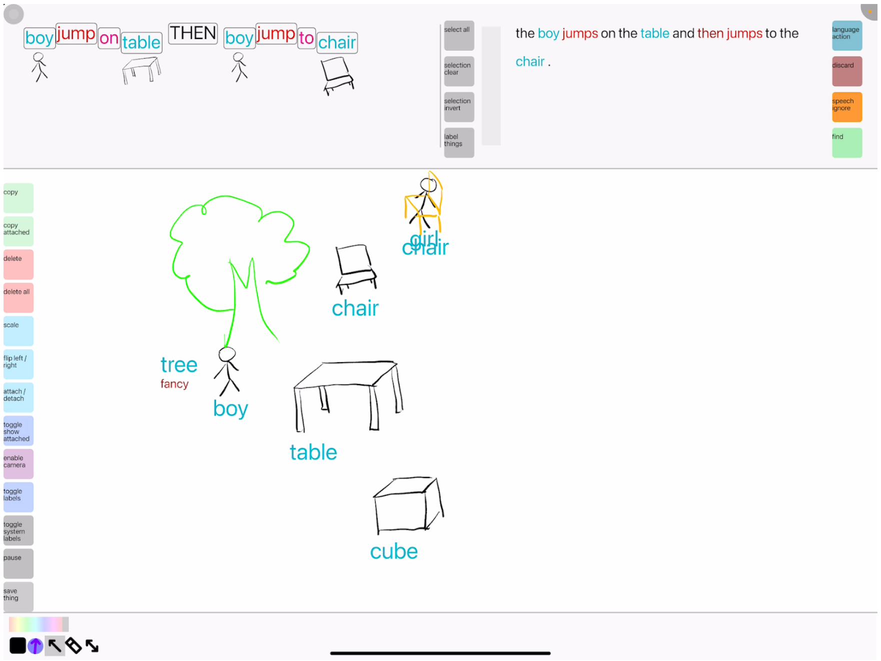
c



d

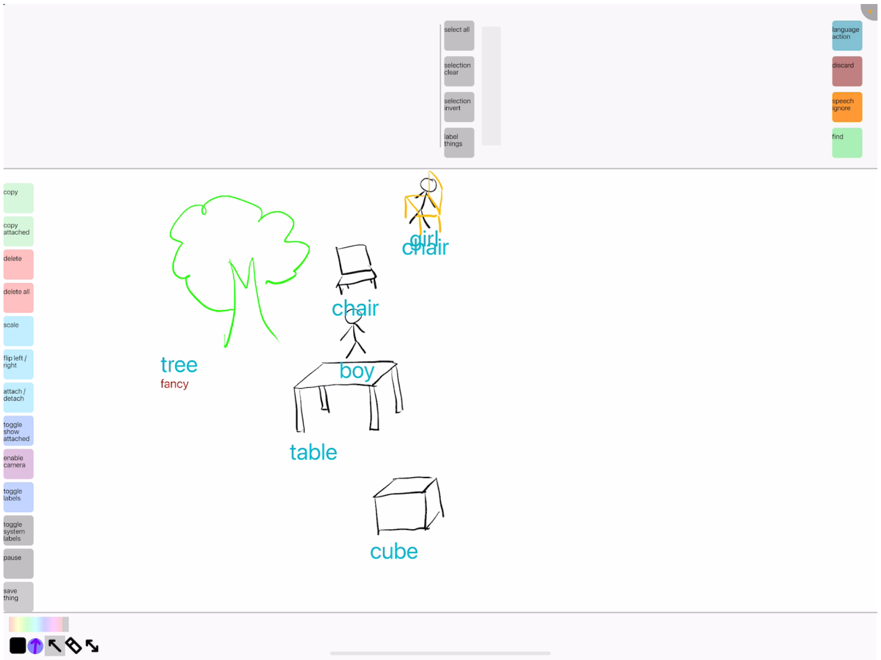


e

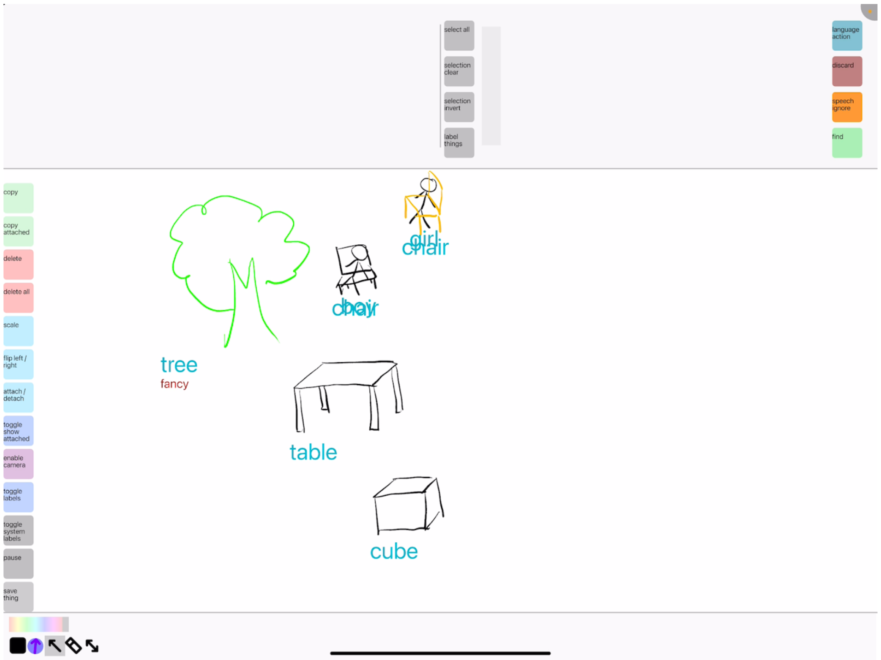


the boy jumps on the table and then jumps to the chair .

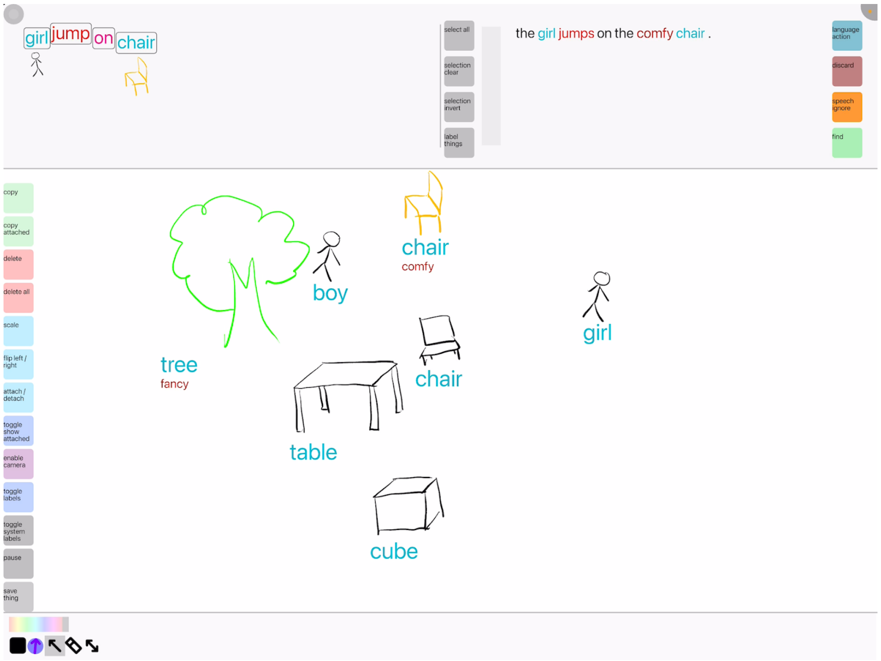
f


select all
selection clear
selection invert
label things
language action
discard
speech ignore
find
copy
copy attached
delete
delete all
scale
flip left / right
attach / detach
toggle show attached
enable camera
toggle labels
toggle system labels
pause
save thing
girl
chair
chair
boy
tree
fancy
table
cube

g

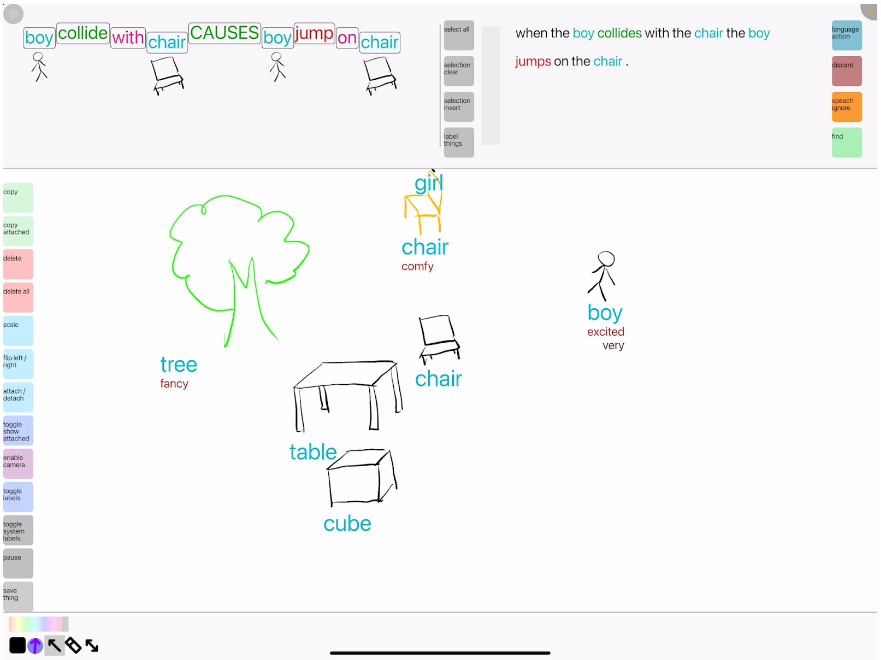

select all
selection clear
selection invert
label things
language action
discard
speech ignore
find
copy
copy attached
delete
delete all
scale
flip left / right
attach / detach
toggle show attached
enable camera
toggle labels
toggle system labels
pause
save thing
girl
chair
boy
chair
tree
fancy
table
cube

h



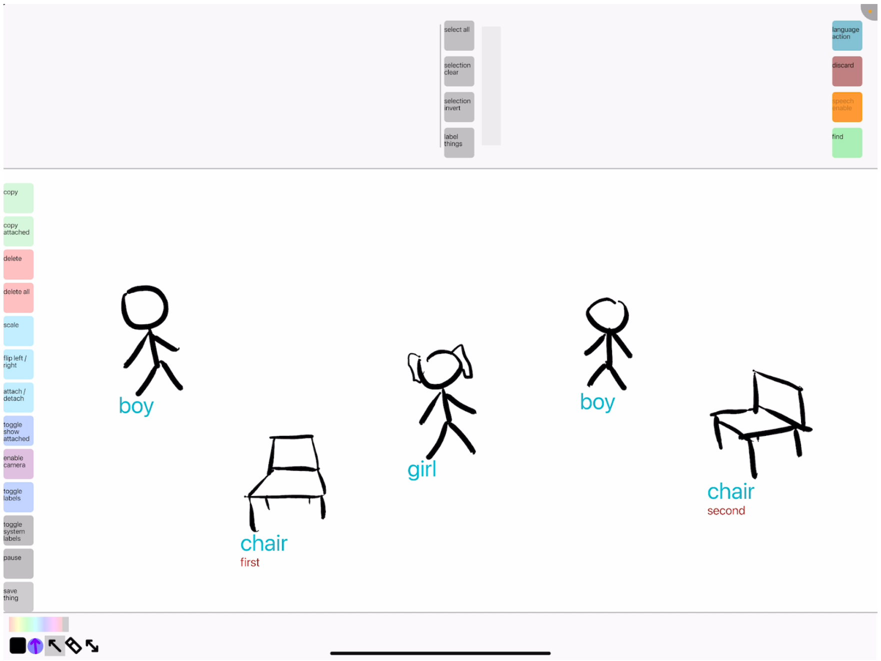
the girl jumps on the comfy chair .

i

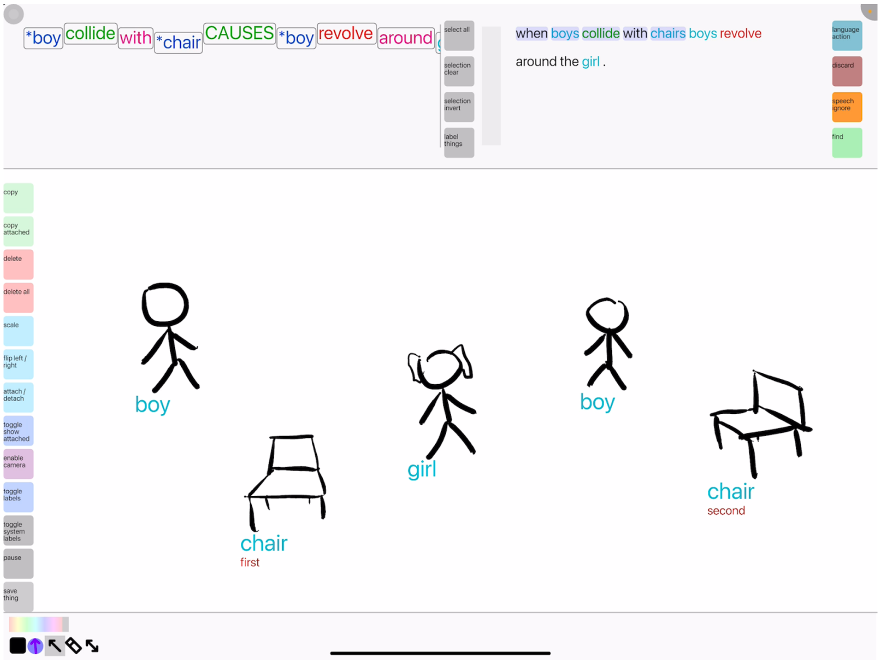


when the boy collides with the chair the boy jumps on the chair .

j



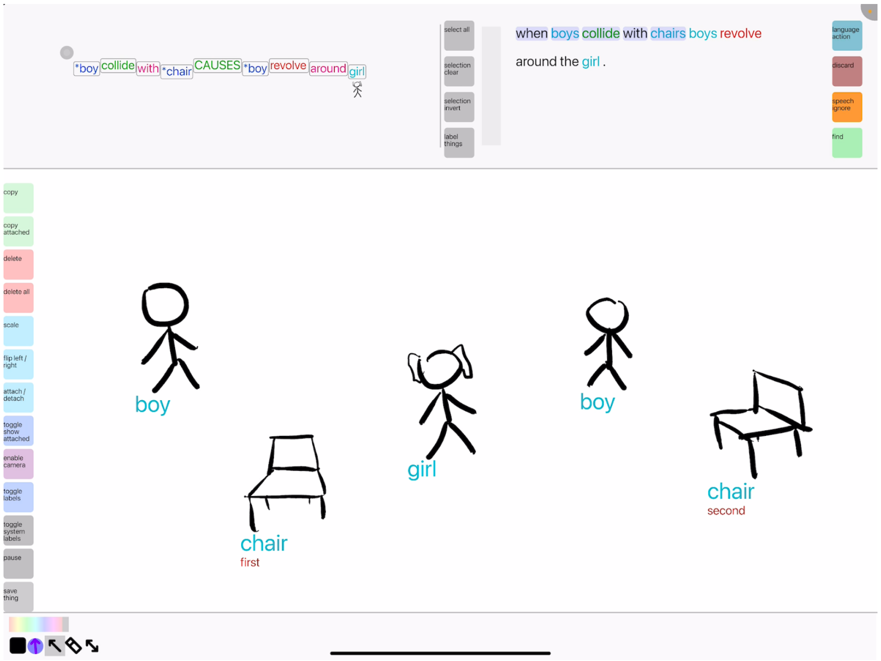
k



*boy collide with *chair CAUSES *boy revolve around

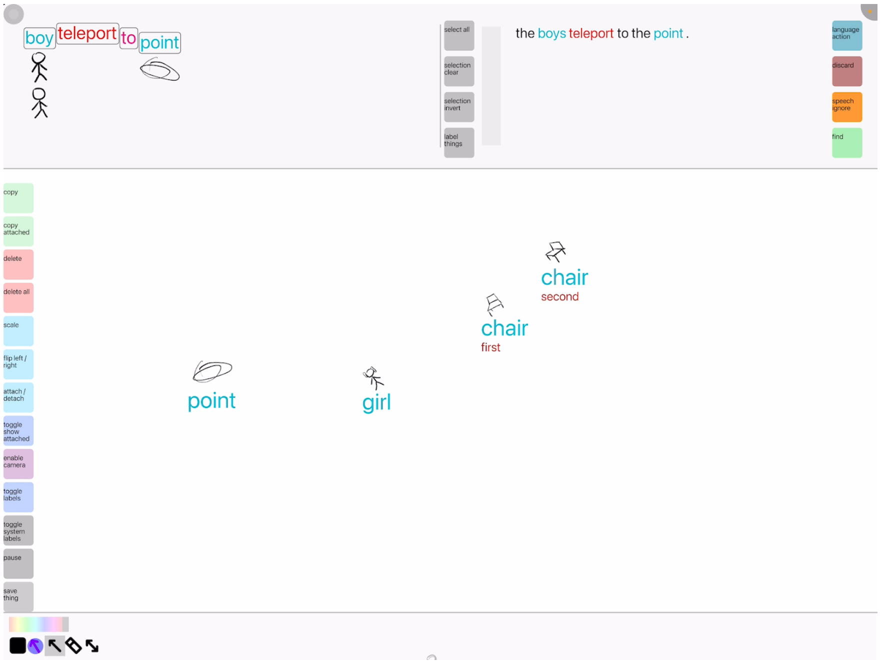
when boys collide with chairs boys revolve around the girl .

l



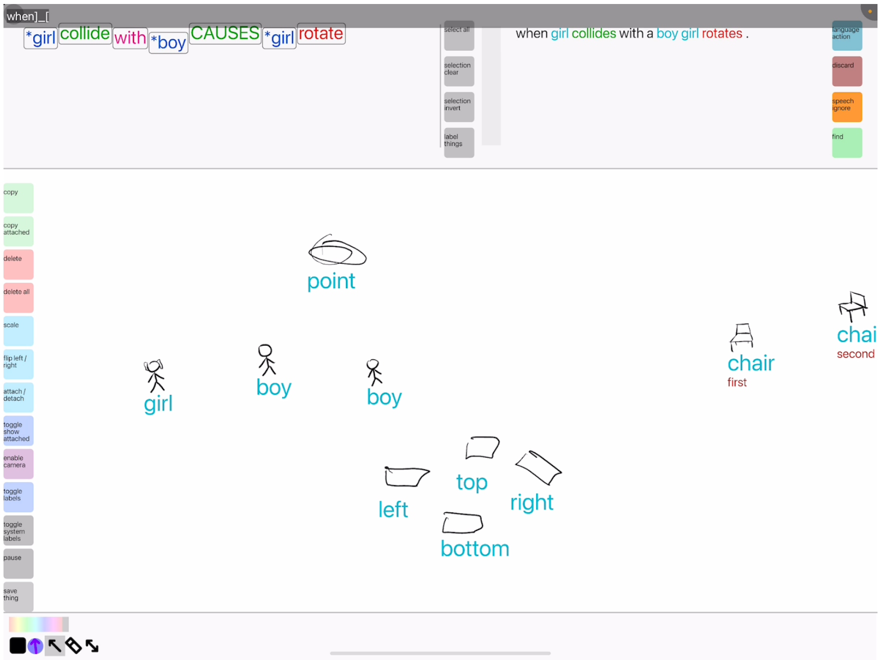
when boys collide with chairs boys revolve around the girl .

m



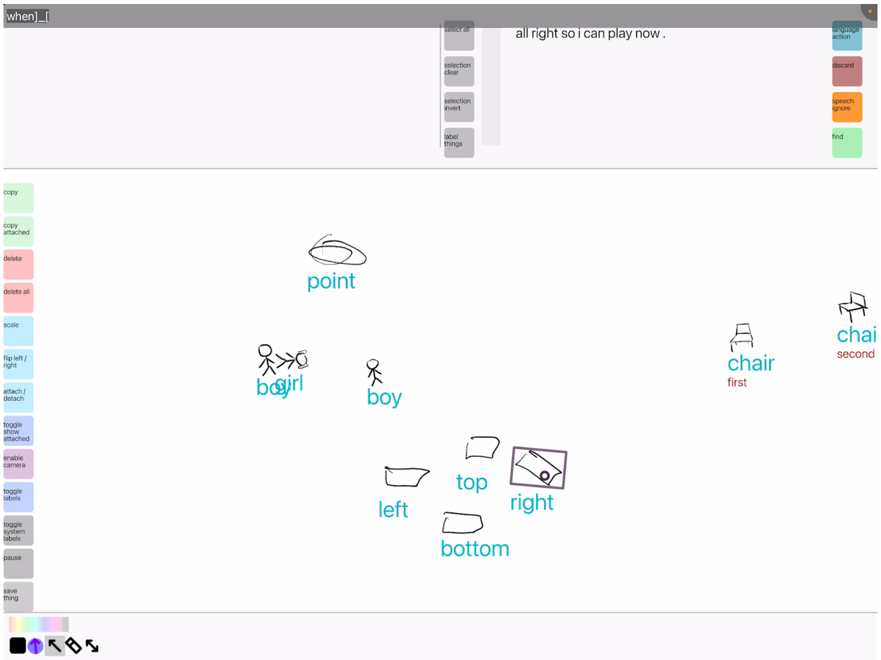
the boys teleport to the point .

n



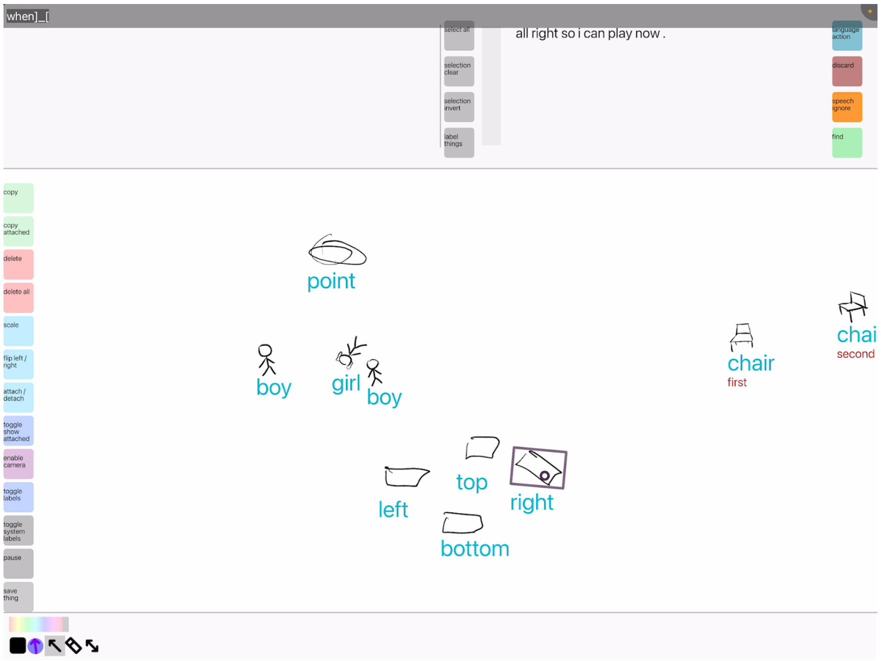
when girl collides with a boy girl rotates .

o



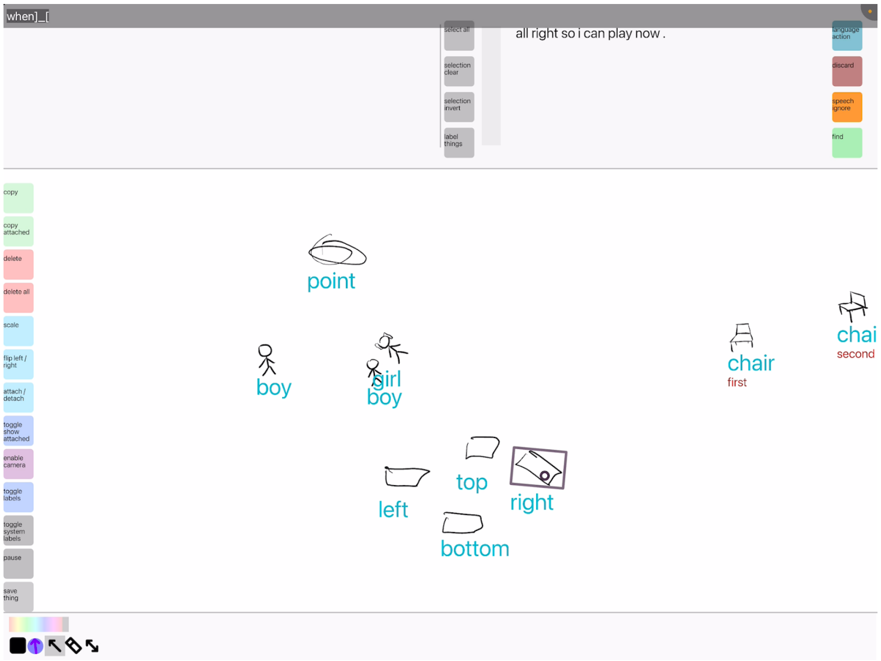
all right so i can play now .

p



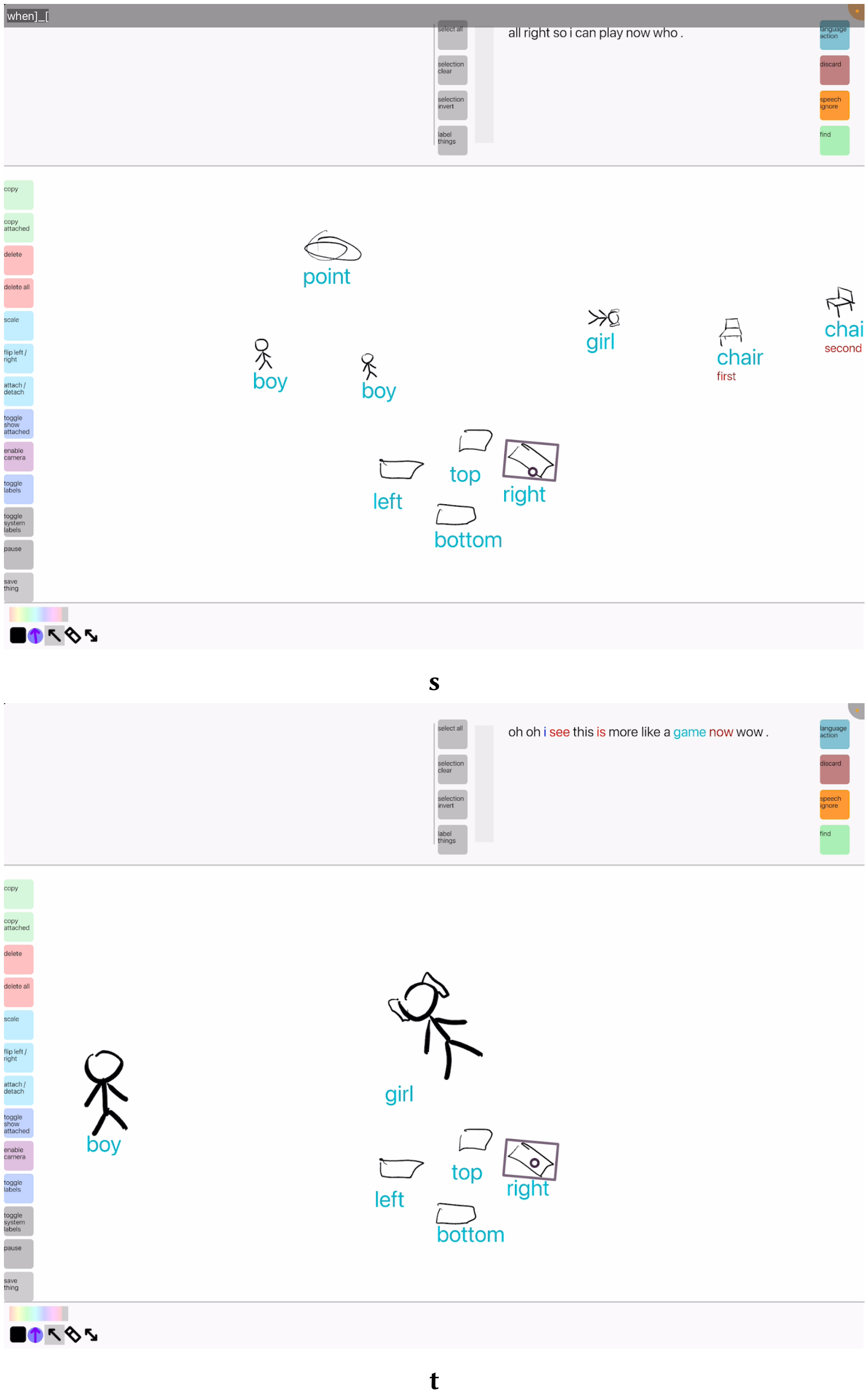
all right so i can play now .

q



all right so i can play now .

r



all right so i can play now who .

s

oh oh i see this is more like a game now wow .

t

Figure A.9: P9$_{expl}$

Appendix: Abstract Semantic Structure

The following defines the basic structure of S29.1.2, which is a simplified semantic structure graph representing an interpretable command. The concrete implementation of DrawTalking traverses this structure to generate final execution commands S39.1.2.

Listing A.1: S2 Data Structure

```
S2_Element : Type_Definition {
    type :=
        CMD_LIST |
        ACTION |
        AGENT | DIRECT_OBJECT | OBJECT | INDIRECT_OBJECT |
        PREPOSITION |
        TRIGGER_RESPONSE | TRIGGER | RESPONSE |
        SEQUENCE_SIMULTANEOUS | SEQUENCE_THEN |
        PROPERTY | PLURAL | COUNT | SPECIFIC_OR_UNSPECIFIC | TIME |
        COREFERENCE

    value :=
        Number | // e.g. any of Float64, Float32, Uint64, etc.
        Thing_ID |
        Thing_Type |
        String |
        Boolean |
        // e.g. pointer or int ID to dynamic-allocated object
        Reference |
        List[Value_Type]

    // S2_Elements should be allocated with stable pointers, or use stable IDs
    parent : Reference(S2_Element)
    // Similar to JSON, but elements are always lists (can have multiple children for the same key
    // (although the layout is not a hard requirement)
    key_to_value := Map[String : List[S2_Element]]
    // property can refer to another property e.g. for coreference
    refers_to : Reference(S2_Element)
    user_feedback_ref : Reference(Anything)... // usually refer to some user feedback UI element
    token : Reference(Token) // optional: stable pointer or ID to the raw token in the language input used to create this element
}
```

Listing A.2: Possible structures for S2

```
// Note that each right-hand-side can be a list.

CMD_LIST -> any of the rest

ACTION -> any combination of
           SOURCE,
           DIRECT_OBJECT,
           INDIRECT_OBJECT,
           OBJECT,
           PREPOSITION,
           PROPERTY, // contains a "trait" with a string value for the property name e.g. how adverbs or adjectives are used.
           SEQUENCE_SIMULTANEOUS,
           SEQUENCE_THEN,
           TIME // usually how long the action should last
           COREFERENCE,

TRIGGER_RESPONSE -> TRIGGER + RESPONSE

TRIGGER, RESPONSE -> ACTION
// equivalent to ACTION
// (but handled differently to generate rules.
// TRIGGER should be used to generate rules.
// RESPONSE should be used to generate commands invoked in the future with arguments generated from rule evaluation)

PREPOSITION -> OBJECT

SOURCE, DIRECT_OBJECT, INDIRECT_OBJECT, OBJECT -> any combination of
PLURAL, // whether plural or not
COUNT, // number of elements
SPECIFIC_OR_UNSPECIFIC // referring to a specific object or not
PROPERTY

PLURAL, COUNT, SPECIFIC_OR_UNSPECIFIC, TIME are terminal

SEQUENCE_THEN -> ACTION
SEQUENCE_SIMULTANEOUS -> ACTION

COREFERENCE // contains a pointer to another node in the value, usually noun-like.
```

Listing A.3: Output from "Forever the person throws the ball into the pond and then the dog gives the ball to her." prior to object selection

```
{
    label=[], tag=[], type=[], kind=[], key=[], idx=[0] id=[1931]
    [CMD_LIST] = [
    {
        label=[], tag=[], type=[CMD], kind=[], key=[CMD_LIST], idx=[0] id=[1932]
        [ACTION] = [
        {
            label=[throw], tag=[VERB], type=[ACTION], kind=[ACTION], key=[ACTION], idx=[0] id=[1933]
            [PREPOSITION] = [
            {
                label=[into], tag=[], type=[into], kind=[], key=[PREPOSITION], idx=[0] id=[1934]
                [OBJECT] = [
                {
                    label=[pond], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[OBJECT], idx=[0] id=[1935]
                    value={
                        THING_INSTANCE=[609]
                    }
                    [SPECIFIC_OR_UNSPECIFIC] = [
                    {
                        label=[the], tag=[DET], type=[VALUE], kind=[SPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1936]
                        value={
                            FLAG=[true]
                        }
                    }
                    ,
                    ]
                    [COUNT] = [
                    {
                        label=[], tag=[], type=[VALUE], kind=[], key=[COUNT], idx=[0] id=[1937]
                        value={
                            NUMERIC=[1.000000]
                        }
                    }
                    ,
                    ]
                    [PLURAL] = [
                    {
                        label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1938]
                        value={
                            FLAG=[false]
                        }
                    }
                    ,
                    ]
                }
                ,
                ]
            }
            ,
            ]
            [SEQUENCE_THEN] = [
            {
                label=[give], tag=[VERB], type=[ACTION], kind=[ACTION], key=[SEQUENCE_THEN], idx=[0], @=[MUST_FILL_IN_AGENT] id=[1939]
                [PREPOSITION] = [
                {
                    label=[to], tag=[], type=[to], kind=[], key=[PREPOSITION], idx=[0] id=[1940]
                    [OBJECT] = [
                    {
                        label=[person], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[AGENT], idx=[0] id=[1968]
                        <coreference substitution> id=[1960]
                        value={
                            THING_INSTANCE=[606]
                        }
                        [SPECIFIC_OR_UNSPECIFIC] = [
                        {
                            label=[the], tag=[DET], type=[VALUE], kind=[SPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1969]
                            value={
                                FLAG=[true]
                            }
                        }
                        ,
                        ]
                        [COUNT] = [
                        {
                            label=[], tag=[], type=[VALUE], kind=[], key=[COUNT], idx=[0] id=[1970]
                            value={
                                NUMERIC=[1.000000]
                            }
                        }
                        ,
                        ]
                        [PLURAL] = [
                        {
                            label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1971]
                            value={
                                FLAG=[false]
                            }
                        }
                        ,
                        ]
                    }
                    ,
                    ]
                }
                ,
                ]
                [DIRECT_OBJECT] = [
                {
                    label=[ball], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[DIRECT_OBJECT], idx=[0] id=[1964]
                    <coreference substitution> id=[1955]
                    value={
                        THING_INSTANCE=[607]
                    }
                    [SPECIFIC_OR_UNSPECIFIC] = [
                    {
                        label=[the], tag=[DET], type=[VALUE], kind=[SPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1965]
                        value={
                            FLAG=[true]
                        }
                    }
                    ,
                    ]
                    [COUNT] = [
                    {
                        label=[], tag=[], type=[VALUE], kind=[], key=[COUNT], idx=[0] id=[1966]
                        value={
                            NUMERIC=[1.000000]
                        }
                    }
                    ,
                    ]
                    [PLURAL] = [
                    {
                        label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1967]
                        value={
                            FLAG=[false]
                        }
                    }
                    ,
                    ]
                }
                ,
                ]
                [AGENT] = [
                {
                    label=[dog], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[AGENT], idx=[0] id=[1951]
                    value={
                        THING_INSTANCE=[608]
                    }
                    [SPECIFIC_OR_UNSPECIFIC] = [
                    {
                        label=[the], tag=[DET], type=[VALUE], kind=[SPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1952]
                        value={
                            FLAG=[true]
                        }
                    }
                    ,
                    ]
                    [COUNT] = [
                    {
                        label=[], tag=[], type=[VALUE], kind=[], key=[COUNT], idx=[0] id=[1953]
                        value={
                            NUMERIC=[1.000000]
                        }
                    }
                    ,
                    ]
```

```
        [PLURAL] = [
        {
            label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1954]
            value={
                FLAG=[false]
            }
        }
        ,
        ]
    }
    ,
    ]
}
,
]
[DIRECT_OBJECT] = [
{
    label=[ball], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[DIRECT_OBJECT], idx=[0] id=[1955]
    value={
        THING_INSTANCE=[607]
    }
    [SPECIFIC_OR_UNSPECIFIC] = [
    {
        label=[the], tag=[DET], type=[VALUE], kind=[SPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1956]
        value={
            FLAG=[true]
        }
    }
    ,
    ]
    [COUNT] = [
    {
        label=[], tag=[], type=[VALUE], kind=[], key=[COUNT], idx=[0] id=[1957]
        value={
            NUMERIC=[1.000000]
        }
    }
    ,
    ]
    [PLURAL] = [
    {
        label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1958]
        value={
            FLAG=[false]
        }
    }
    ,
    ]
}
,
]
[PROPERTY] = [
{
```

```
                label=[modifier], tag=[ADV], type=[PROPERTY], kind=[], key=[PROPERTY], idx=[0] id=[1959]
                value={
                    TEXT=[forever]
                }
            }
            ,
            ]
            [AGENT] = [
            {
                label=[person], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[AGENT], idx=[0] id=[1960]
                value={
                    THING_INSTANCE=[606]
                }
                [SPECIFIC_OR_UNSPECIFIC] = [
                {
                    label=[the], tag=[DET], type=[VALUE], kind=[SPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1961]
                    value={
                        FLAG=[true]
                    }
                }
                ,
                ]
                [COUNT] = [
                {
                    label=[], tag=[], type=[VALUE], kind=[], key=[COUNT], idx=[0] id=[1962]
                    value={
                        NUMERIC=[1.000000]
                    }
                }
                ,
                ]
                [PLURAL] = [
                {
                    label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1963]
                    value={
                        FLAG=[false]
                    }
                }
                ,
                ]
            }
            ,
            ]
        }
        ,
        ]
    }
    ,
    ]
}
```

Listing A.4: Output from "Every few seconds the frog hops to a lily." prior to object selection

```
{
    label=[], tag=[], type=[], kind=[], key=[], idx=[0] id=[1018]
    [CMD_LIST] = [
    {
        label=[], tag=[], type=[CMD], kind=[], key=[CMD_LIST], idx=[0] id=[1019]
        [ACTION] = [
        {
            label=[hop], tag=[VERB], type=[ACTION], kind=[ACTION], key=[ACTION], idx=[0] id=[1005]
            [PREPOSITION] = [
            {
                label=[to], tag=[], type=[to], kind=[], key=[PREPOSITION], idx=[0] id=[1013]
                [OBJECT] = [
                {
                    label=[lily], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[OBJECT], idx=[0] id=[1014]
                    value={
                        THING_INSTANCE=[0]
                    }
                    [SPECIFIC_OR_UNSPECIFIC] = [
                    {
                        label=[a], tag=[DET], type=[VALUE], kind=[UNSPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1016]
                        value={
                            FLAG=[false]
                        }
                    }
                    ,
                    ]
                    [COUNT] = [
                    {
                        label=[], tag=[], type=[], kind=[], key=[COUNT], idx=[0] id=[1017]
                        value={
                            NUMERIC=[1.000000]
                        }
                    }
                    ,
                    ]
                    [PLURAL] = [
                    {
                        label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1015]
                        value={
                            FLAG=[false]
                        }
                    }
                    ,
                    ]
                }
                ,
                ]
            }
            ,
            ]
            [TIME] = [
            {
```

```
            label=[second], tag=[TIME], type=[INTERVAL], kind=[], key=[TIME], idx=[0] id=[1006]
            [PROPERTY] = [
            {
                label=[trait], tag=[ADJ], type=[PROPERTY], kind=[], key=[PROPERTY], idx=[0] id=[1008]
                value={
                    TEXT=[few]
                }
            }
            ,
            ]
        }
        ,
        ]
        [AGENT] = [
        {
            label=[frog], tag=[NOUN], type=[], kind=[THING_INSTANCE], key=[AGENT], idx=[0] id=[1009]
            value={
                THING_INSTANCE=[0]
            }
            [SPECIFIC_OR_UNSPECIFIC] = [
            {
                label=[the], tag=[DET], type=[VALUE], kind=[SPECIFIC], key=[SPECIFIC_OR_UNSPECIFIC], idx=[0] id=[1011]
                value={
                    FLAG=[true]
                }
            }
            ,
            ]
            [COUNT] = [
            {
                label=[], tag=[], type=[VALUE], kind=[], key=[COUNT], idx=[0] id=[1012]
                value={
                    NUMERIC=[1.000000]
                }
            }
            ,
            ]
            [PLURAL] = [
            {
                label=[], tag=[], type=[VALUE], kind=[], key=[PLURAL], idx=[0] id=[1010]
                value={
                    FLAG=[false]
                }
            }
            ,
            ]
        }
        ,
        ]
    }
    ,
    ]
}
```

```
    ,
    ]
}
```

Appendix: System Code Samples

Listing A.5: "Make The Thing" World Code Sample

```

struct World {
    usize step_count;
    Thing_Archetype_Collection archetypes;
    Thing_Collection things;

    Eval_Connection_Graph root_graph;
    Eval_Connection_Graph* current_graph = &root_graph;
    Eval_Connection_Graph* saved_graph = &root_graph;
    Runtime runtime;

    Message_Passer message_passer;

    mem::Allocator allocator;

    mem::Memory_Pool_Fixed memory_pool;
    mem::Allocator pool_allocator;

    mem::Pool_Allocation drawable_pool = {};

    mem::Buckets_Allocation buckets;
    mem::Buckets_Allocation message_allocation;
    mem::Buckets_Allocation arg_allocation;

    bool no_deletion_zone_on = false;


    MTT_String_Pool string_pool;

    float64 time_seconds = 0.0;
    float64 timestep = 0.0f;
    float64 time_seconds_prev = 0.0;
    uint64 time_ns = 0;
    uint64 time_ns_prev = 0;
    uint64 timestep_ns = 0;
    usize eval_count = 0;

    std::deque<Thing_To_Make_Info> to_make;
    std::deque<Destroy_Command> to_destroy;
    std::deque<mtt::Thing_ID> to_destroy_end;

    std::vector<To_Clear> to_clear;

    std::deque<Thing_ID> to_disable;
    std::deque<Thing_ID> to_enable;

    flecs::world ecs_world;

    flecs::entity IS_ATTRIBUTE_TAG;

    External_World ext_worlds[2];
    usize curr_ext_world_id = 1;

    b2World* physics_world;

    void (*on_thing_make)(mtt::World* world, mtt::Thing* thing);

    flecs::query<mtt::Thing_Info, mtt::Sensor> sensor_query;

    sd::Renderer* renderer;

    std::vector<mtt::Trigger_Response_Command> rules;

    void* user_data;

    Priority_Layer priority_layer = PRIORITY_LAYER_DEFAULT;

    std::vector<Priority_Layer> priority_layer_stack = {};


    mtt::Collision_System collision_system;
    mtt::Collision_System collision_system_canvas;

    mtt::Collision_System_Group_World_Canvas collision_system_group = {
        .world = &collision_system,
        .canvas = &collision_system_canvas
    };

    Instancing instancing;

    inline bool Thing_try_get(Thing_ID id, Thing** thing)
    {
        return mtt::Thing_try_get(this, id, thing);
    }

    inline Thing* Thing_try_get(Thing_ID id)
    {
        return mtt::Thing_try_get(this, id);
    }

    inline Thing* Thing_get(Thing_ID id)
    {
        return mtt::Thing_get(this, id);
    }

    inline void Thing_get(Thing_ID id, Thing** thing)
    {
        mtt::Thing_get(this, id, thing);
    }

    inline bool Thing_Archetype_try_get(Thing_Archetype_ID id, Thing_Archetype** arch)
    {
        return mtt::Thing_Archetype_try_get(this, id, arch);
    }

    inline void Thing_Archetype_get(Thing_Archetype_ID id, Thing_Archetype** arch)
    {
        mtt::Thing_Archetype_get(this, id, arch);
    }

    std::deque<mtt::Thing_ID> traversal_queue;
    std::vector<Thing*> thing_buffer;
    std::vector<Thing*> to_enable_list;
    std::vector<Thing*> to_disable_list;

    // need per-frame transient memory

    // per simulation step
    mem::Allocator allocator_temporary_[ALLOCATOR_TEMPORARY_COUNT];
    mem::Arena memory_temporary_[ALLOCATOR_TEMPORARY_COUNT];
    // only reset the arena after a few frames
    usize per_frame_reset_counter_ = 0;

    inline mem::Allocator& allocator_temporary(void) { return allocator_temporary_[0]; }
    inline void allocator_temporary_begin_frame(void) {

        if (per_frame_reset_counter_ >= 32) {
            per_frame_reset_counter_ = 0;
            mem::Arena_rewind(&memory_temporary_[0]);
        } else {
            per_frame_reset_counter_ = (per_frame_reset_counter_ + 1);
        }
    }

    mtt::Map<mtt::Thing_ID, Input_Triggers> input_triggers;

    Thing_Child_List saved_children;

    Thing_Archetype_Drawable_Instances archetype_drawables;

    dt::Word_Dictionary_Entry* collide;

    flecs::entity collide_tag;
    flecs::entity collide_begin_tag;
    flecs::entity collide_end_tag;

    dt::Word_Dictionary_Entry* select;

    flecs::entity select_tag;
    flecs::entity select_begin_tag;
    flecs::entity select_end_tag;

    dt::Word_Dictionary_Entry* overlap;

    flecs::entity overlap_tag;
    flecs::entity overlap_begin_tag;
    flecs::entity overlap_end_tag;

    dt::Word_Dictionary_Entry* equivalent_to;
    std::vector<Collision_Record> collisions_to_remove;
    usize collisions_to_remove_count;

    mtt::Map<mtt::String, mtt::Interaction_Trigger> interactions;


    mtt::Map_Stable<mtt::Thing_ID, Thing_Metadata> id_to_metadata;


    void (*custom_on_thing_make) = [](mtt::Thing* thing) {};


    void clear_all_of_type(mtt::ARCHETYPE arch);
    void clear_all_of_type_ignore_flags(mtt::ARCHETYPE arch);

    mem::Pool_Allocation field_list_pool;


    std::vector<bool(*)(
    mtt::World* world,
    float32 fixed_dt,
    float32 time_prev,
    float32 time,
    float32 realtime_dt,
    MTT_Core* core,
    void* ctx)> deferred_per_frame;

    bool show_verbose = true;
    bool show_attachment_links = false;
    bool show_debug = false;
    bool show_script_eval_print = false;

    mtt::Map_Stable<mtt::String, mtt::Set<mtt::Thing_ID>> thing_saved_presets = {};
};

struct alignas(16) Thing {
    // main ID
    Thing_ID id;
    // ID of the "type"
    Thing_Archetype_ID archetype_id;
    Thing_ID root_thing_id;
    // if this thing is a proxy,
    //this ID is the original Thing being proxied
    Thing_ID mapped_thing_id;
    // handle to the query library
    Entity ecs_entity;
    // contains info about how to "run" the Thing
    // based on its type, per simulation step
    Logic logic;
    // dynamic data description
    Field_List_Descriptor field_descriptor;
    // dynamic data (that are currently active/swapped-in)
    Field_List active_fields;
    // inputs and outputs for data flow
    Port_Descriptor ports;
    uint64 eval_index;
    usize eval_priority;

    // simulation graph
    Context_ID ctx_id = Context_ID_DEFAULT;
    Eval_Connection_Graph* graph = nullptr;
    Evaluation_Output* eval_out = nullptr;

    inline bool operator==(const Thing& other) const { return this->id == other.id; }

    // flags
    struct alignas(16) {
        bool is_resident;
        bool is_root;
        bool is_visible;
        bool is_locked;
        bool is_visited;
        bool is_user_destructible;
        bool is_user_drawable;
        bool do_evaluation;
        bool is_user_movable;
        bool lock_user_movement_if_not_root;
        bool lock_to_canvas;
        bool forward_input_to_root;
        bool should_defer_destruction = false;
        bool is_static = false;
        bool is_reserved;
        THING_FLAG flags;
    };
    // visuals
    Representation representation;
    // user input handlers
    Input_Handlers input_handlers;

    void (*message_handler)(Message* msg);

    // gets the world
    inline World* world()
    {
        return mtt::ctx();
    }


    Thing_ID next_id;
    Thing_ID prev_id;
    Thing_ID first_child;
    Thing_ID parent_thing_id;

    Thing_ID saved_parent_thing_id;


    Thing_Child_List child_id_set;

    void (*on_destroy)(Thing* thing);



    usize selection_count;

    void (*on_run_init)(Thing* thing) = nullptr;

    // syste
    MTT_String_Ref label;
};



struct Eval_Connection_Graph {

    Evaluation_Output output = {};

    typedef Thing* Eval_Op;
    std::vector<Eval_Op> sorted_things_direct = {};
    usize visited_count = 0;

    bool is_modified = false;


    Map<Thing_ID, Port_Input_List> incoming = {};
    Map<Thing_ID, std::vector<Port_Input_List>> outgoing = {};

    Map_Stable<Thing_ID, mtt::Set<Thing_ID>> incoming_execution = {};
    Map_Stable<Thing_ID, mtt::Set<Thing_ID>> outgoing_execution = {};
};

void evaluate_world(World* world)
{
    world->no_deletion_zone_on = true;

    broad_phase(&world->collision_system, 0);

    if (root_graph(world)->is_modified) {
        root_graph(world)->is_modified = false;

        sort_things_in_root_ctx(world, world->things.instances, *root_graph(world));
    }

    auto* rtime = Runtime::ctx();
    Signal_Mailbox_clear(&rtime->signal_mailbox);


    {
        std::stable_sort(
                        rtime->script_tasks.rules_list.begin(),
                        rtime->script_tasks.rules_list.end(),
                        Script_Instance_compare__ID_and_Priority());

        auto& task_list = rtime->script_tasks.rules_list;
        for (isize i = task_list.size() - 1; i >= 0; i -= 1) {

            auto* task = task_list[i];

            switch (task->status) {
                case SCRIPT_STATUS_CANCELED: {
                    Script_Instance_cancel(task);
                    MTT_FALLTHROUGH
                }
                case SCRIPT_STATUS_DONE_SHOULD_TERMINATE:
                case SCRIPT_STATUS_TERMINATED:
                case SCRIPT_STATUS_DONE: {
                    Script_Instance_terminate(task);
                    rtime->id_to_rule_script.erase(task->id);
                    Script_Instance_destroy(task);
                    std::swap(task_list[i], task_list[task_list.size() - 1]);

                    task_list.pop_back();
                    continue;
                }
                default: { break; }
            }

            process_script(world, task);
        }
    }

    while (!world->message_passer.system_messages_deferred_before_scripts.empty()) {
        auto& msg = world->message_passer.system_messages_deferred_before_scripts.front();

        Procedure_Input_Output io = {};

        Thing* sender = nullptr;
        world->Thing_try_get(msg.sender, &sender);
        io.caller = sender;
        io.input = (void*)&msg;
        msg.proc(&io);

        world->message_passer.system_messages_deferred_before_scripts.pop_front();
    }

    std::stable_sort(
                     rtime->script_tasks.list.begin(),
                     rtime->script_tasks.list.end(),
                     Script_Instance_compare__ID_and_Priority());

    auto& task_list = rtime->script_tasks.list;
    for (isize i = task_list.size() - 1; i >= 0; i -= 1) {

        auto* task = task_list[i];

        switch (task->status) {
            case SCRIPT_STATUS_CANCELED: {
                Script_Instance_cancel(task);
                MTT_FALLTHROUGH
            }
            case SCRIPT_STATUS_DONE_SHOULD_TERMINATE:
            case SCRIPT_STATUS_TERMINATED:
            case SCRIPT_STATUS_DONE: {
                Script_Instance_terminate(task);
                Script_Instance_destroy(task);
                std::swap(task_list[i], task_list[task_list.size() - 1]);

                task_list.pop_back();
                continue;
            }
            default: { break; }
        }

        process_script(world, task);
    }

    process_things(world, *root_graph(world));
}

void process_script(World* world, Script_Instance* script)
{
    Eval_Connection_Graph& G = script->source_script->connections;

    usize sorted_things_count = G.sorted_things_direct.size();
    if (sorted_things_count == 0) {
        return;
    }

    Evaluation_Output& eval_out = G.output;

    auto* prev_graph = mtt::curr_graph(world);
    mtt::set_graph(world, &G);

    Thing** sorted_things = &G.sorted_things_direct[0];
    usize port_index = 0;

    auto& thing_local_fields = script->thing_local_fields;

    bool is_done = true;
    const usize ctx_count = script->contexts.size();
    if (script->status == SCRIPT_STATUS_NOT_STARTED) {
        script->status = SCRIPT_STATUS_STARTED;

        if (script->agent != mtt::Thing_ID_INVALID) {
            auto& label = script->source_script->label;
            auto* rtime = Runtime::ctx();
            auto& task_list = rtime->script_tasks.list;

            for (isize i = task_list.size() - 1; i >= 0; i -= 1) {
                auto& tsk = task_list[i];
                if (tsk != script && !tsk->allow_duplicates_for_agent && tsk->label == label && tsk->agent == script->agent) {
                    Script_Instance_stop(tsk);
                }
            }
        }

        for (usize ctx_idx = 0; ctx_idx < ctx_count; ctx_idx += 1) {
            auto* const script_ctx = &script->contexts[ctx_idx];
            script_ctx->is_done = false;
            script_ctx->slot_idx = ctx_idx;
        }


        if (script->on_start != nullptr) {
            for (usize i = 0; i < sorted_things_count; i += 1) {
                mtt::Thing* thing = sorted_things[i];
            }

            auto res = script->on_start(world, script, script->args);
            auto [result_type, result_continuation, result_value] = res;
        }
    }

    if (script->on_begin_frame != nullptr) {
        auto res = script->on_begin_frame(world, script, script->args);
        auto [result_type, result_continuation, result_value] = res;
        switch (result_type) {
            case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_DONE_SHOULD_TERMINATE: {
                script->status = SCRIPT_STATUS_DONE_SHOULD_TERMINATE;
                is_done = true;
                goto LABEL_EXIT;
                break;
            }
            default: {
                break;
            }
        }
    }

    for (isize ctx_idx = ctx_count - 1; ctx_idx >= 0; ) {
        script->contexts[ctx_idx].slot_idx = ctx_idx;
        if (script->contexts[ctx_idx].is_done) {
            ctx_idx -= 1;
            continue;
        }

        Script_set_current_context(script, ctx_idx);


        auto* ctx_state = &Script_current_context_state(script);

        Script_Instance_start_or_resume(script);


        for (; ctx_state->instruction_idx < sorted_things_count; ) {
            Thing* const thing = sorted_things[ctx_state->instruction_idx];
            Thing_ID const thing_id = thing->id;

            {
                usize iteration = mtt::get_loop_iteration(script);

                mtt::set_active_fields(thing, &thing_local_fields[thing->eval_index][iteration]);
                thing->eval_out = &script->output;
                script->output.set_port_entries_index(iteration);

            }

            Port_Input_List* input_list = nullptr;
            map_get(curr_graph(world)->incoming, thing_id, &input_list);

            auto res = thing->logic.proc(world, thing, input_list, script, &ctx_state, nullptr);
            auto [result_type, result_continuation, result_value] = res;
            ctx_state = &Script_current_context_state(script);
            switch (result_type) {
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_ENTER_SCOPE: {
                    ctx_state->instruction_idx += 1;
                    break;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_EXIT_SCOPE: {
                    break;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_IMMEDIATE_SUSPEND: {
                    is_done &= false;
                    script->contexts[ctx_idx].is_done = false;
                    goto LABEL_CONTEXT_EVAL_END_STEP;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_JUMP: {
                    is_done &= false;
                    script->contexts[ctx_idx].is_done = false;
                    break;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_JUMP_IMMEDIATE_SUSPEND: {


                    is_done &= false;
                    script->contexts[ctx_idx].is_done = false;
                    goto LABEL_CONTEXT_EVAL_END_STEP;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_JUMP_IMMEDIATE_SUSPEND_NO_RESET: {


                    is_done &= false;
                    script->contexts[ctx_idx].is_done = false;
                    goto LABEL_CONTEXT_EVAL_END_STEP;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_DONE: {
                    script->status = SCRIPT_STATUS_DONE;
                    is_done = true;
                    goto LABEL_EXIT;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_DONE_SHOULD_TERMINATE: {
                    script->status = SCRIPT_STATUS_DONE_SHOULD_TERMINATE;
                    is_done = true;
                    goto LABEL_EXIT;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_DONE_WAS_STOPPED: {
                    script->status = SCRIPT_STATUS_DONE;
                    goto LABEL_EXIT;
                }
                case LOGIC_PROCEDURE_RETURN_STATUS_TYPE_CONTEXT_DONE: {
                    is_done &= true;
                    script->contexts[ctx_idx].is_done = true;
                    goto LABEL_CONTEXT_EVAL_END_STEP;
                }
                default: {
                    ctx_state->instruction_idx += 1;
                    break;
                }
            }


        }
        if (ctx_state->instruction_idx >= sorted_things_count) {
            script->contexts[ctx_idx].is_done = true;
        }
        is_done &= script->contexts[ctx_idx].is_done;

        LABEL_CONTEXT_EVAL_END_STEP:;

        ctx_idx -= 1;
    }

    LABEL_EXIT:;




    if (is_done) {
        if (script->status != SCRIPT_STATUS_DONE_SHOULD_TERMINATE) {
            script->status = SCRIPT_STATUS_DONE;
        }

        if (script->on_done != nullptr) {
            auto res = script->on_done(world, script, script->args);
            auto [result_type, result_continuation, result_value] = res;
        }
    }


    mtt::set_graph(world, prev_graph);
}

// process root-level Things (i.e. "actors" like freehand sketches
void process_things(World* world, Eval_Connection_Graph& G)
{
    Evaluation_Output& eval_out = G.output;

    usize sorted_things_count = G.sorted_things_direct.size();
    if (sorted_things_count == 0) {
        return;
    }

    auto* prev_graph = mtt::curr_graph(world);
    mtt::set_graph_to_root(world);

    Thing** sorted_things = &G.sorted_things_direct[0];
    usize port_index = 0;
    for (usize thing_idx = 0; thing_idx < sorted_things_count; thing_idx += 1) {


        Thing* const thing = sorted_things[thing_idx];
        Thing_ID const thing_id = thing->id;

        {
            eval_out.list[thing_idx].first_port_index = port_index;

            auto* const out_ports = &thing->ports.out_ports;
            const usize port_count = out_ports->size();
            usize next_port_index = port_index + port_count;
            while (eval_out.port_entries().size() < next_port_index) {
                eval_out.port_entries().push_back({});
            }
            for (usize i = 0; i < port_count; i += 1) {
                auto* const entry = (&eval_out.port_entries()[port_index + i]);
                entry->ID = thing_id;
                entry->out.type = (*out_ports)[i].type;
                entry->out.contained_type = (*out_ports)[i].contained_type;

                entry->is_ignored = false;
            }
            port_index += port_count;
        }
    }

    for (usize thing_idx = 0; thing_idx < sorted_things_count; thing_idx += 1) {
        Thing* const thing = sorted_things[thing_idx];
        Thing_ID const thing_id = thing->id;


        Port_Input_List* input_list = nullptr;
        map_get(G.incoming, thing_id, &input_list);
        auto return_status = thing->logic.proc(world, thing, input_list, NULL, NULL, NULL);
        (void)return_status;
    }

    mtt::set_graph(world, prev_graph);
}
```

Listing A.6: Scripting Structures

```

// the script template
struct Script {
    Script_ID id = Script_ID_INVALID;
    static inline Script_ID next_avail_ID = 1;
    static mtt::Map_Stable<Script_ID, Script> scripts;
    static mtt::Map_Stable<mtt::String, mtt::Map<SCRIPT_CALLING_CONVENTION, Script_ID>> scripts_by_name_and_calling_convention;



    Script_Lookup lookup_ = {};
    bool is_alias = false;

    bool allow_duplicates_for_agent = false;

    inline Script_Lookup& lookup(void)
    {
        return lookup_;
    }

    Script* alias_make(void)
    {
        // This works for now because an alias will just be readonly besides its lookup table
        Script* s = &Script::scripts[Script::next_avail_ID];
        s->id = Script::next_avail_ID;
        Script::next_avail_ID += 1;

        usize own_id = s->id;
        *s = *this;
        s->id = own_id;

        s->is_alias = true;
        s->ref_count = 0;

        return s;
    }



    std::vector<Script_Precondition> preconditions;

    void add_precondition(Script_ID script_id)
    {
        Script_Precondition pc = {};
        pc.id = script_id;
        preconditions.push_back(pc);
    }

    std::vector<Script_Precondition>& get_preconditions()
    {
        return this->preconditions;
    }

    Script_Label label = {};
    Script_Label sub_label = {};

    SCRIPT_CALLING_CONVENTION calling_convention = {};

    std::vector<Script_Contexts> contexts = {};

    Eval_Connection_Graph connections = {};

    bool preserve_lookup = false;
    bool share_lookup_with_sub_scripts = false;
    // if set to true, does not depend on child scripts
    // e.g. trigger/response drivers do not need to check child statuses
    bool detach_children = false;

    usize ref_count = 0;
    bool destroy_upon_ref_count_0 = false;
    bool is_infinite = false;



    mtt::Logic_Procedure_Return_Status (*on_start)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_begin_frame)(mtt::World* world, Script_Instance* script_instance, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_end_frame)(mtt::World* world, Script_Instance* script_instance, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_cancel)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_done)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_terminate)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;

    bool is_rule = false;
};

// instance of a running script constructed from a Script
struct Script_Instance {
    Script_ID id = Script_ID_INVALID;
    static inline Script_ID next_avail_ID = 1;
    static inline const usize PRIORITY_FIRST = 0;

    usize priority = PRIORITY_FIRST;

    Script* source_script = nullptr;
    Script_Instance* parent = nullptr;
    mtt::Thing_ID caller = mtt::Thing_ID_INVALID;

    uint64 creation_time = 0;
    // This is where the per-evaluation data are stored (passed between Things during evaluation)
    Evaluation_Output output = {};

    mtt::String label = {};


    std::vector<Field_List_Descriptor> thing_initial_fields = {};

    std::vector<std::vector<Field_List>> thing_local_fields = {};
    std::vector<Active_Action> active_action_list = {};

    mtt::Thing_ID agent = mtt::Thing_ID_INVALID;
    bool allow_duplicates_for_agent = false;
    void action(const mtt::String& a, mtt::Thing_ID src, mtt::Thing_ID dst);
    void action(const mtt::String& a, mtt::Thing_ID src);
    void remove_actions(void);
    void remove_actions(mtt::Thing_ID t_id);

    Script_Lookup lookup_ = {};
    Script_Lookup* curr_lookup = &lookup_;

    Script_Lookup lookup_initial = {};

    bool is_own_lookup = true;
    bool preserve_lookup = false;

    Script_Lookup& lookup()
    {
        return *curr_lookup;
    }

    void set_lookup_copy(Script_Lookup* lu)
    {
        lookup_ = *lu;
        curr_lookup = &lookup_;

        is_own_lookup = true;
    }

    Script_Lookup* shared_lookup()
    {
        return curr_lookup;
    }

    void set_shared_lookup(Script_Lookup* lu)
    {
        curr_lookup = lu;

        is_own_lookup = false;
    }

    void set_own_lookup()
    {
        curr_lookup = &lookup_;

        is_own_lookup = true;
    }

    std::vector<Script_Contexts> contexts = {};
    usize ctx_idx = 0;
    std::vector<mtt::Any> return_value = {};
    SCRIPT_STATUS status = SCRIPT_STATUS_NOT_STARTED;

    inline static mem::Pool_Allocation pool = {};
    inline static Script_Instance* make()
    {
        auto* s_i = mem::alloc_init<Script_Instance>(&Script_Instance::pool.allocator);
        s_i->id = Script_Instance::next_avail_ID;
        Script_Instance::next_avail_ID += 1;
        return s_i;
    }
    inline static void destroy(Script_Instance* s)
    {
        mem::deallocate<Script_Instance>(&Script_Instance::pool.allocator, s);
    }

    Script_Instance* init(Script* script, void (*post_init)(Script_Instance* s, void* data), void* data = nullptr);
    Script_Instance* deinit(void);

    mtt::Logic_Procedure_Return_Status (*on_start)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_begin_frame)(mtt::World* world, Script_Instance* script_instance, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_end_frame)(mtt::World* world, Script_Instance* script_instance, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_cancel)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_done)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;
    mtt::Logic_Procedure_Return_Status (*on_terminate)(mtt::World* world, Script_Instance* self_script, void* args) = nullptr;

    Script_Rules rules = {};
    Script_Rules* rules_ref = nullptr;
    std::vector<Rule_Var_Record_One_Result> rule_vars = {};
    bool rules_are_valid = true;
    bool rules_are_active = true;

    void* args = nullptr;

    inline bool is_rule(void)
    {
        return source_script->is_rule;
    }
};
```

Bibliography

(1980). Space Invaders game cartridge - CHM Revolution.

(2022). 50 Years of Fun With Pong. Section: Curatorial Insights.

, t. C. (2023). Scribblenauts. Page Version ID: 1166977807.

Aesop and Winters, M. (2023). Aesop's Fables Interactive Book Library of Congress.

Agrawala, M., Li, W., and Berthouzoz, F. (2011). Design principles for visual communication. *Communications of the ACM*, 54(4):60–69.

AtariAdmin (2022). New Insight into Breakout's Origins.

Bolt, R. A. (1980). Put-that-there: Voice and gesture at the graphics interface. *ACM SIGGRAPH Computer Graphics*, 14(3):262–270.

Buxton, W. (1992). Telepresence: Integrating shared task and person spaces. In *Proceedings of graphics interface*, volume 92, pages 123–129. Canadian Information Processing Society Toronto, Canada.

Chandrasegaran, S., Ramanujan, D., and Elmqvist, N. (2018). How Do Sketching and Non-Sketching Actions Convey Design Intent? In *Proceedings of the 2018 Designing Interactive Systems Conference*, DIS '18, pages 373–385, New York, NY, USA. Association for Computing Machinery.

Chang, Y. (2020). DrawmaticAR: automagical AR content from written words! In *ACM SIGGRAPH 2020 Real-Time Live!*, SIGGRAPH '20, page 1, New York, NY, USA. Association for Computing Machinery.

Cohen, P. R., Johnston, M., McGee, D., Oviatt, S., Pittman, J., Smith, I., Chen, L., and Clow, J. (1997). QuickSet: multimodal interaction for simulation set-up and control. In *Proceedings of the fifth conference on Applied natural language processing*, ANLC '97, pages 20–24, USA. Association for Computational Linguistics.

Compton, K. and Mateas, M. (2015). Casual Creators.

Coyne, B. and Sproat, R. (2001). WordsEye: an automatic text-to-scene conversion system. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 487–496, New York, NY, USA. Association for Computing Machinery.

cycling74 (2018). Max.

Davis, R. C., Colwell, B., and Landay, J. A. (2008). K-sketch: a 'kinetic' sketch pad for novice animators. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 413–422, New York, NY, USA. Association for Computing Machinery.

Dietz, G., Le, J. K., Tamer, N., Han, J., Gweon, H., Murnane, E. L., and Landay, J. A. (2021). StoryCoder: Teaching Computational Thinking Concepts Through Storytelling in a Voice-Guided App for Children. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–15, New York, NY, USA. Association for Computing Machinery.

Dietz, G., Tamer, N., Ly, C., Le, J. K., and Landay, J. A. (2023). Visual StoryCoder: A Multimodal Programming Environment for Children's Creation of Stories. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, pages 1–16, New York, NY, USA. Association for Computing Machinery.

Dreams (2020). Dreams. [Playstation 4]. https://indreams.me.

Fan, J. E., Bainbridge, W. A., Chamberlain, R., and Wammes, J. D. (2023). Drawing as a versatile cognitive tool. *Nature Reviews Psychology*, 2(9):556–568. Number: 9 Publisher: Nature Publishing Group.

Fraser, C. A., Markel, J. M., Basa, N. J., Dontcheva, M., and Klemmer, S. (2020). ReMap: Lowering the Barrier to Help-Seeking with Multimodal Search. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 979–986, New York, NY, USA. Association for Computing Machinery.

Gao, T., Dontcheva, M., Adar, E., Liu, Z., and Karahalios, K. G. (2015). DataTone: Managing Ambiguity in Natural Language Interfaces for Data Visualization. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 489–500, New York, NY, USA. Association for Computing Machinery.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., USA.

Gugenheimer, J., Stemasov, E., Frommel, J., and Rukzio, E. (2017). Sharevr: Enabling co-located experiences for virtual reality between hmd and non-hmd users. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 4021–4033. ACM.

He, Z., Rosenberg, K. T., and Perlin, K. (2019). Exploring Configuration of Mixed Reality Spaces for Communication. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA '19, pages 1–6, New York, NY, USA. Association for Computing Machinery.

Hinckley, K., Yatani, K., Pahud, M., Coddington, N., Rodenhouse, J., Wilson, A., Benko, H., and Buxton, B. (2010). Pen + touch = new tools. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 27–36, New York, NY, USA. Association for Computing Machinery.

Huang, F., Schoop, E., Ha, D., and Canny, J. (2020). Scones: towards conversational authoring of sketches. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*, IUI '20, pages 313–323, New York, NY, USA. Association for Computing Machinery.

Ingalls, D. (2020). The evolution of Smalltalk: from Smalltalk-72 through Squeak. *Proceedings of the ACM on Programming Languages*, 4(HOPL):85:1–85:101.

Insa, D., Silva, J., and Tamarit, S. (2016). Where you sit matters how classroom seating might affect marks. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 212–217, New York, NY, USA. ACM.

Ishii, H., Kobayashi, M., and Grudin, J. (1993). Integration of interpersonal space and shared workspace: Clearboard design and experiments. *ACM Transactions on Information Systems (TOIS)*, 11(4):349–375.

Jacobs, J., Brandt, J. R., Meěh, R., and Resnick, M. (2018). Dynamic Brushes: Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI EA '18, pages 1–4, New York, NY, USA. Association for Computing Machinery.

Kazi, R. H., Chevalier, F., Grossman, T., and Fitzmaurice, G. (2014a). Kitty: sketching dynamic and interactive illustrations. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, UIST '14, pages 395–405, New York, NY, USA. Association for Computing Machinery.

Kazi, R. H., Chevalier, F., Grossman, T., Zhao, S., and Fitzmaurice, G. (2014b). Draco: bringing life to illustrations with kinetic textures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 351–360, New York, NY, USA. Association for Computing Machinery.

Kim, Y.-S., Dontcheva, M., Adar, E., and Hullman, J. (2019). Vocal Shortcuts for Creative Experts. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 1–14, New York, NY, USA. Association for Computing Machinery.

Landay, J. A. (1996). SILK: sketching interfaces like krazy. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 398–399, New York, NY, USA. Association for Computing Machinery.

Lanir, J., Booth, K. S., and Findlater, L. (2008). Observing presenters' use of visual aids to inform the design of classroom presentation software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 695–704, New York, NY, USA. ACM.

Laput, G. P., Dontcheva, M., Wilensky, G., Chang, W., Agarwala, A., Linder, J., and Adar, E. (2013). PixelTone: a multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2185–2194, New York, NY, USA. Association for Computing Machinery.

LaViola, J. J. and Zeleznik, R. C. (2004). MathPad2: a system for the creation and exploration of mathematical sketches. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 432–440, New York, NY, USA. Association for Computing Machinery.

Lee, B., Kazi, R. H., and Smith, G. (2013). SketchStory: Telling More Engaging Stories with Data through Freeform Sketching. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2416–2425. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

Lee, J. H., Kim, H., and Bae, S.-H. (2022). Rapid design of articulated objects. *ACM Transactions on Graphics*, 41(4):89:1–89:8.

Leiva, G., Grønbæk, J. E., Klokmose, C. N., Nguyen, C., Kazi, R. H., and Asente, P. (2021). Rapido: Prototyping Interactive AR Experiences through Programming by Demonstration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 626–637, New York, NY, USA. Association for Computing Machinery.

Liao, J., Karim, A., Jadon, S. S., Kazi, R. H., and Suzuki, R. (2022). RealityTalk: Real-Time Speech-Driven Augmented Presentation for AR Live Storytelling. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, pages 1–12, New York, NY, USA. Association for Computing Machinery.

Little Big Planet (2008). Little Big Planet. [Playstation 3].

Little Big Planet 2 (2011). Little Big Planet 2. [Playstation 3].

Liu, X. B., Kirilyuk, V., Yuan, X., Chi, P., Chen, X. A., Olwal, A., and Du, R. (2023). Visual Captions: Augmenting Verbal Communication with On-the-fly Visuals. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI)*.

Loper, E. and Bird, S. (2002). NLTK: the Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics - Volume 1*, ETMTNLP '02, pages 63–70, USA. Association for Computational Linguistics.

Lyons, K., Skeels, C., Starner, T., Snoeck, C. M., Wong, B. A., and Ashbrook, D. (2004). Augmenting conversations using dual-purpose speech. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, pages 237–246, New York, NY, USA. Association for Computing Machinery.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4):16:1–16:15.

Montani, I., Honnibal, M., Honnibal, M., Landeghem, S. V., Boyd, A., Peters, H., McCann, P. O., geovedi, j., O'Regan, J., Samsonov, M., Orosz, G., Kok, D. d., Altinok, D., Kristiansen, S. L., Kannan, M., Bournhonesque, R., Miranda, L., Baumgartner, P., Edward, Bot, E., Hudson, R., Mitsch, R., Roman, Fiedler, L., Daniels, R., Phatthiyaphaibun, W., Howard, G., Tamura, Y., and Bozek, S. (2023). explosion/spaCy: v3.5.0: New CLI commands, language updates, bug fixes and much more.

Novick, D., Rhodes, J., and Wert, W. (2011). The communicative functions of animation in user interfaces. In *Proceedings of the 29th ACM international conference on Design of communication - SIGDOC '11*, page 1, Pisa, Italy. ACM Press.

Nunes, G. B. and Perlin, K. (2017). Atypical: A Type System for Live Performances. In *Adjunct Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17 Adjunct, pages 61–62, New York, NY, USA. Association for Computing Machinery.

Olsen, D. R. (2007). Evaluating user interface systems research. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 251–258, New York, NY, USA. Association for Computing Machinery.

Otsuka, K. (2016). Mmspace: Kinetically-augmented telepresence for small group-to-group conversations. In *Virtual Reality (VR), 2016 IEEE*, pages 19–28. IEEE.

Oviatt, S. (1999). Ten myths of multimodal interaction. *Communications of the ACM*, 42(11):74–81.

Perlin, K. (2016). Future reality: How emerging technologies will change language itself. *IEEE computer graphics and applications*, 36(3):84–89.

Perlin, K. and Goldberg, A. (1996). Improv: a system for scripting interactive actors in virtual worlds. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 205–216, New York, NY, USA. Association for Computing Machinery.

Perlin, K., He, Z., and Rosenberg, K. (2018a). Chalktalk : A Visualization and Communication Language – As a Tool in the Domain of Computer Science Education. arXiv:1809.07166 [cs].

Perlin, K., He, Z., and Zhu, F. (2018b). Chalktalk VR/AR. *International SERIES on Information Systems and Management in Creative eMedia (CreMedia)*, (2017/2):30–31. Number: 2017/2.

Raskar, R., Welch, G., Cutts, M., Lake, A., Stesin, L., and Fuchs, H. (1998). The office of the future: A unified approach to image-based modeling and spatially immersive displays. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 179–188. ACM.

Reas, C. and Fry, B. (2014). *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11):60–67.

Rosenberg, K. T., Kazi, R. H., Wei, L.-Y., Xia, H., and Perlin, K. (2024). DrawTalking: Building Interactive Worlds by Sketching and Speaking. arXiv:2401.05631 [cs] version: 1.

Ross, J., Holmes, O., and Tomlinson, B. (2012). Playing with Genre: User-Generated Game Design in LittleBigPlanet 2.

Saquib, N. (2020). *Embodied mathematics by interactive sketching*. Thesis, Massachusetts Institute of Technology. Accepted: 2021-01-06T20:15:54Z.

Saquib, N., Kazi, R. H., Wei, L.-Y., and Li, W. (2019). Interactive Body-Driven Graphics for Augmented Video Performance. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 1–12, New York, NY, USA. Association for Computing Machinery.

Saquib, N., Kazi, R. H., Wei, L.-y., Mark, G., and Roy, D. (2021). Constructing Embodied Algebra by Sketching. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–16, New York, NY, USA. Association for Computing Machinery.

Sarracino, J., Barrios-Arciga, O., Zhu, J., Marcus, N., Lerner, S., and Wiedermann, B. (2017). User-Guided Synthesis of Interactive Diagrams. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 195–207, New York, NY, USA. Association for Computing Machinery.

Scott, J. and Davis, R. (2013). Physink: sketching physical behavior. In *Adjunct Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13 Adjunct, pages 9–10, New York, NY, USA. Association for Computing Machinery.

Siddiqui, T., Luh, P., Wang, Z., Karahalios, K., and Parameswaran, A. G. (2021). From Sketching to Natural Language: Expressive Visual Querying for Accelerating Insight. *ACM SIGMOD Record*, 50(1):51–58.

Snyder, J. (2013). Drawing practices in image-enabled collaboration. In *Proceedings of the 2013 conference on Computer supported cooperative work*, CSCW '13, pages 741–752, New York, NY, USA. Association for Computing Machinery.

Solomon, C., Minsky, M., and Harvey, B. (1986). Logo works. *New York: MacGraw-Hill Book Company.*

Srinivasan, A. and Stasko, J. (2018). Orko: Facilitating Multimodal Interaction for Visual Exploration and Analysis of Networks. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):511–521. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

Stapleton, A. (2017). Deixis in Modern Linguistics. *Essex Student Journal*, 9(1). Number: 1 Publisher: University of Essex Library Services.

Sturdee, M. and Lindley, J. (2019). Sketching &amp; Drawing as Future Inquiry in HCI. In *Proceedings of the Halfway to the Future Symposium 2019*, HTTF 2019, pages 1–10, New York, NY, USA. Association for Computing Machinery.

Subramonyam, H., Li, W., Adar, E., and Dontcheva, M. (2018). TakeToons: Script-driven Performance Animation. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 663–674, New York, NY, USA. Association for Computing Machinery.

Subramonyam, H., Seifert, C., Shah, P., and Adar, E. (2020). texSketch: Active Diagramming through Pen-and-Ink Annotations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–13, New York, NY, USA. Association for Computing Machinery.

Suhm, B., Myers, B., and Waibel, A. (2001). Multimodal error correction for speech user interfaces. *ACM Transactions on Computer-Human Interaction*, 8(1):60–98.

Sutherland, I. E. (1963). Sketchpad: a man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, spring joint computer conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA. Association for Computing Machinery.

Sutherland, W. R. (1966). *The on-line graphical specification of computer procedures.* Thesis, Massachusetts Institute of Technology. Accepted: 2005-09-21T22:40:43Z.

Suzuki, R., Kazi, R. H., Wei, L.-y., DiVerdi, S., Li, W., and Leithinger, D. (2020). RealitySketch: Embedding Responsive Graphics and Visualizations in AR through Dynamic Sketching. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 166–181, New York, NY, USA. Association for Computing Machinery.

Tan, K.-H., Gelb, D., Samadani, R., Robinson, I., Culbertson, B., and Apostolopoulos, J. (2010). Gaze awareness and interaction support in presentations. In *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, pages 643–646, New York, NY, USA. ACM.

Turner, P. (2016). A Make-Believe Narrative for HCI. In Turner, P. and Harviainen, J. T., editors, *Digital Make-Believe*, Human–Computer Interaction Series, pages 11–26. Springer International Publishing, Cham.

Tversky, B. (2011). Visualizing Thought. *Topics in Cognitive Science*, 3(3):499–535. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1756-8765.2010.01113.x.

Victor, B. (2013). Stop Drawing Dead Fish.

Victor, B. (2014). Humane representation of thought: a trail map for the 21st century. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, UIST '14, page 699, New York, NY, USA. Association for Computing Machinery.

Walny, J., Carpendale, S., Henry Riche, N., Venolia, G., and Fawcett, P. (2011). Visual Thinking In Action: Visualizations As Used On Whiteboards. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2508–2517. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

Winograd, T. (1972). Understanding natural language. *Cognitive Psychology*, 3(1):1–191.

Wong, C., McCarthy, W., Grand, G., Friedman, Y., Tenenbaum, J., Andreas, J., Hawkins, R., and Fan, J. (2022). *Identifying concept libraries from language about object structure.*

Xia, H. (2020). Crosspower: Bridging Graphics and Linguistics. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 722–734, New York, NY, USA. Association for Computing Machinery.

Xia, H., Jacobs, J., and Agrawala, M. (2020). Crosscast: Adding Visuals to Audio Travel Podcasts. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 735–746, New York, NY, USA. Association for Computing Machinery.

Xia, H., Wang, T., Gunturu, A., Jiang, P., Duan, W., and Yao, X. (2023). CrossTalk: Intelligent Substrates for Language-Oriented Interaction in Video-Based Communication and Collaboration. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, pages 1–16, New York, NY, USA. Association for Computing Machinery.