From Painting to Widgets, 6-DOF and Bimanual Input Beyond Pointing

Bret Jackson* Daniel F. Keefe†

June 3, 2020

In this chapter, we present an approach to designing expressive 3D user interfaces that make use of handheld input devices tracked in 3D space to go beyond a simple pointing metaphor. We show how using the State Design Pattern can be useful for implementing interfaces in this style, and we provide examples from recent work highlighting particular design considerations. We hope that the reader will come away with the knowledge and inspiration to design more complex expressive 3D user interfaces.

1 Introduction

What would you do if you could hold a magic wand in your hand? Better yet, one in each hand! From the early days of VR, hardware and software designers have continued to return again and again to the concept of VR wands – handheld, tracked six degree-of-freedom input devices – as a primary means of input. Note, by six degree-of-freedom, abbreviated 6-DOF, we mean 3 degrees for positioning (location) plus 3 degrees for rotating (orientation). The names of these devices (e.g., wand, stylus, 3D mouse, game controller) and number of buttons, joysticks, trackpads and other options have changed over the years, but there seems to be something fundamentally appealing and exciting about the opportunity to interact with the computer by moving one (or more) 6-DOF tracked devices through the air.

Despite this fundamental appeal and the consistent evolution of hand input VR hardware over the years, we find that the majority of VR applications simply do not take advantage of the amazing opportunity provided by holding a magic wand in one's hand. The most obvious thing to do with one of these devices is to point. Thus, there are countless examples of VR laser pointers and flying metaphors (i.e., point the wand in the direction you wish to fly). These metaphors are a fine and useful starting point, but what else is possible? How do we go beyond pointing and/or integrate it within larger, more complex interfaces? Surely, there must be more we can do with a magic wand in our hands!

Conceptually, 6-DOF stylus input can support many interactions that go beyond or build upon pointing. We encourage readers to consider designing interfaces that support more expressive, complex poses and gestures, often made in a body-centric way, and we provide several examples later in the chapter. Many of these "beyond-pointing" interfaces will take advantage of user interaction

*Department of Mathematics, Statistics, & Computer Science, Macalester College, St. Paul, MN, 55105
†Department of Computer Science & Engineering, University of Minnesota, Minneapolis, MN 55455

within arms reach, perhaps involving direct manipulation with virtual content. Beyond-pointing interfaces built upon 6-DOF input devices can also be used to define spatial relationships (3D poses, orientations, distances) relative to one's body; grab objects and rotate, translate, scale, and twist them; and make sweeping movements and gestures over time. All these types of interactions may be made directly in 3D space or relative to some form of virtual widget. What's more, with two 6-DOF tracked devices, one in each hand, a bimanual user interface can be created whereby each of these beyond-pointing interactions can be combined to create simultaneous and composite interactions. Just as we might hold a piece of pottery in one hand while painting on it with the other, in a composite 6-DOF interaction, one hand can set the context for the other. This potential to harness natural, body-scale, and coordinated (e.g., bimanual) input is where we believe VR will shine.

Given this broad design space and great potential, why don't we see more beyond-pointing interfaces in current VR applications? We believe there are several reasons. First, it requires a different approach to programming. In desktop-based programming, our most common computing paradigm, the assumption is that the user controls just a single cursor. Thus, most programs rely upon this single cursor to set focus for input events and so on. In recent years, as multi-touch user interfaces have become more widely utilized, it has been exciting to see this assumption challenged more and more often, and new multi-cursor infrastructure for desktop, tablet, and phone platforms is emerging. This is useful for VR, as many new VR programmers may at least be familiar with the concept of multi-cursor input now. Yet, there are still important differences. For example, in VR, each cursor has more degrees of freedom; users often stand up without access to a keyboard or other complementary devices; and often it is dark or the view of our hands is blocked, so we work with input devices we cannot see. Whereas pointing and clicking is the dominant interaction mode in desktop computing, the VR situation is different, and user interface concepts such as gesture and spatial proximity become much more important. Thus, a different approach is needed when designing and implementing theses VR interfaces.

How then does one design and program useful VR beyond-pointing interfaces? Answering this question is the key message of this chapter. Our approach is to first introduce a general framework and terminology that can apply to any VR bimanual user interface. Then, we walk through a series of three examples that build upon each other to demonstrate how sophisticated 6-DOF, beyond-pointing, bimanual VR user interfaces can be built upon this framework.

2 VR Bimanual UI Framework and Terminology

The framework we promote is event based. Inputs, such as button presses, generate events; the VR program listens for these events and then responds appropriately. We also treat movements of the 6-DOF trackers as events, similar to the way modern windowing systems report mouse move events. Since this movement is often nearly continuous, this means that an application can expect to respond to tracker move events almost every frame.

Although event-based user interfaces are common in desktop-computing, there are several important nuances in VR. For example, the need to work with multiple 3D coordinate systems and the need to interpret input relative to the current digital and physical context. It is possible today to purchase a VR input device with more buttons than we have fingers. Many developers will thus be tempted to simply assign one system action to each button. This is fine for expert users who are willing to put time into training and who will eventually develop a mental map of the input devices, but we advocate a different approach. We demonstrate several examples where the context in which input

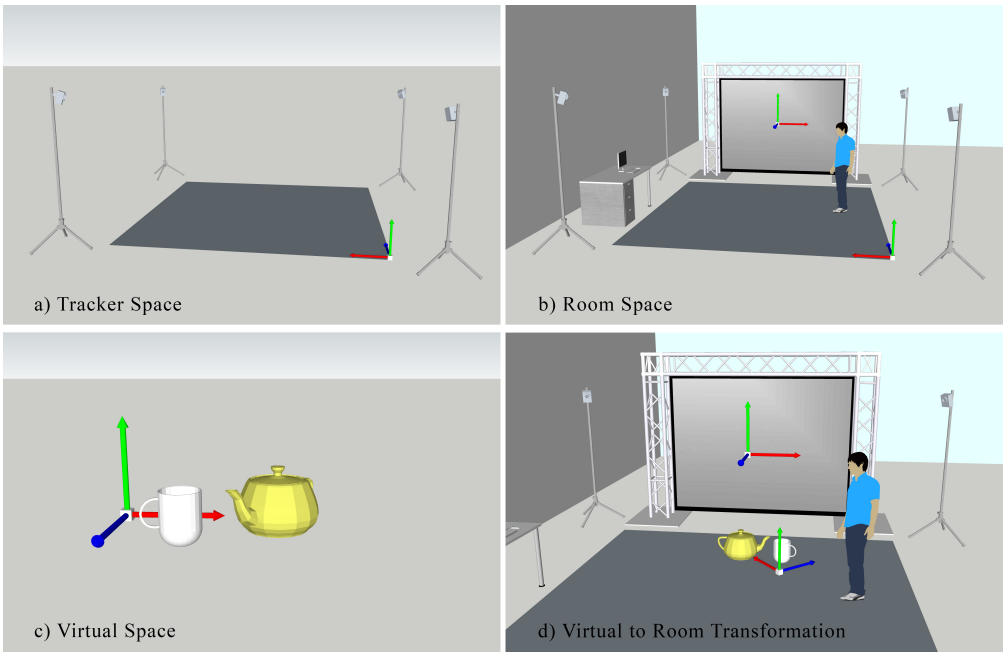

Figure 1: (a.) Tracker Space is defined relative to the specific 3D tracking hardware. (b.) Room Space is defined relative to the physical VR display. Sometimes this is the same as Tracker Space, but often it differs slightly, for example, in a projection-based VR display, the origin of Room Space may be defined conveniently to be located at the center of the display or the center of the walkable area. The virtual cameras setup in the VR graphics engine will be defined in Room Space; so, Room Space provides the essential link between tracking hardware and graphics rendering. Some virtual content (e.g., cursors, menus) may be defined in Room Space coordinates when the intent is for their virtual positions to stay consistent within the physical room. (c.) The main content of the virtual scene is typically defined in a separate coordinate system called Virtual Space. (d.) This Virtual Space content may be scaled, translated, or rotated relative to Room Space in order to make large viewpoint adjustments (travel through the virtual world).

events occur can be used to disambiguate users' intent. Thus, context (from both the physical and virtual worlds) is important. The state machine design pattern that we suggest for programming VR interfaces helps programmers to track and respond appropriately to changes in context.

2.1 Definitions

Before demonstrating with examples how this framework can be applied to create exciting VR interfaces, let's begin with just a few definitions:

2.1.1 Coordinate Systems

Tracker Space: The raw data from the tracking system is reported in a Tracker Space coordinate system (origin and axes), as shown in Figure 1a. This coordinate system may align with the Room Space coordinate system, but does not always. In a calibrated system, the raw data is converted into Room Space before being reported in the tracker matrix.

Room Space: A coordinate system defined relative to the physical space the VR display hardware occupies. The Room Space coordinate system provides a way to relate tracking data to graphics,

since the virtual cameras used to render VR graphics are also defined in Room Space coordinates.

Virtual Space: A coordinate system used to define virtual content that may move relative to the physical room. Virtual Space content may be rotated, scaled, or translated relative to the Room Space coordinate system, but Room Space content will stay fixed within the room. A `virtualToRoom` 4x4 transformation matrix is used to encode the transformation between virtual space and room space.

2.1.2 Devices, Events, and Widgets

6-DOF: 6 Degree of Freedom. 3 Position (Location) + 3 Rotation (Orientation)

Tracked Device: A VR input device, such as a wand or stylus, that can be tracked in 3D space, often with 6 degrees of freedom. Often the device will have buttons or other inputs. When the device is moved, a `Tracker_Move` event is generated, and when a button is depressed and later released, `Button_Down` and then `Button_Up` events are generated.

Event: A discrete programmatic signal generated in response to user input, for example a press or release of a button or movement of a tracked device to a new location or orientation relative to its last move event.

Tracker and **Tracker Matrix:** Each tracked device has (or can be modeled as having) a 6-DOF tracker attached to it. The Tracker Matrix is the raw data, typically a 4x4 transformation matrix, reported by this tracking hardware. These are the data reported by `Tracker_Move` events. The tracker matrix is reported in Room Space coordinates.

Cursor and **Proxy Matrix**: An icon or other visual representation of the tracked device displayed in VR for the user to see. In some applications the cursor may be drawn at the exact location defined by the Tracker Matrix, but we will discuss several examples where it is useful to offset this location slightly. For example, imagine a snap-to-grid mode, where small positional movements of a physical tracker are ignored in order to keep the virtual tracker locked onto a grid point. Thus, we add the concept of a Proxy Matrix, which stores the current position (location) and orientation of a virtual proxy for the tracker. The cursor is drawn at the position and orientation specified in the Proxy Matrix, which is defined in Room Space and is often close to, but not necessarily the same, as the Tracker Matrix.

Widget: A virtual user interface object, such as a menu, that responds to and provides a visual target for user input. Widgets may be defined in Room Space so that they appear to occupy the same space as the user (often useful for menus), or they may be defined in Virtual Space so that they move together with a virtual scene.

2.1.3 Context-Based User Interface

Interface Context: A discrete set of circumstances defined by both the virtual environment and the physical environment that requires input events to be interpreted differently. The circumstances are often significant enough that each interface context would include a change in visual feedback for the user.

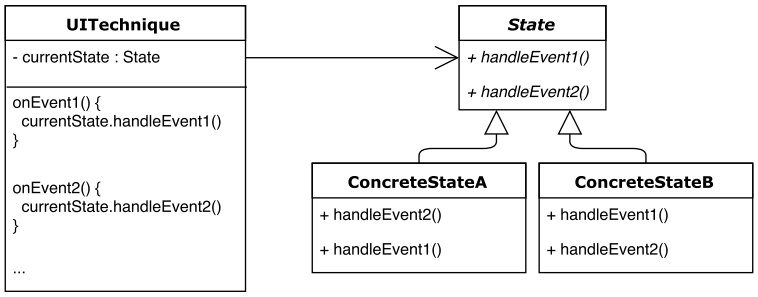

Figure 2: A generic illustration of the state design pattern applied to handling user input events. `UITechnique` always maintains one current state. When your program calls `UITechnique::onEvent1()` or `UITechnique::onEvent2()`, these calls are passed on to whichever `ConcreteState` class is pointed to by `currentState`.

State: We suggest an implementation based upon the state design pattern, where each state is a programmatic representation of the interface context.

2.2 State Design Pattern

There are many ways to implement beyond-pointing VR interfaces. We make two recommendations. First, and most importantly, recognize the importance of the interface context (defined above) in VR and choose a style of implementation that is designed to handle this context elegantly – in contrast, the approach we have too often seen is to handle state changes via one long conditional (`if`) statement, which does not scale well. Second, we recommend a specific approach that makes use of the state design pattern used extensively in object-oriented software design. Since this is the approach used in the remainder of the chapter, we begin with a brief description of the State Pattern.

"The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class." [Freeman et al., 2004] Figure 2 shows an example diagrammed in Unified Markup Language (UML).

In our approach, a client (e.g., your program's main loop), will notify `UITechnique` that some input event has occurred by calling `onEvent1()`, `onEvent2()`, etc. But, the way in which the user interface responds could be quite different depending upon which `ConcreteState` is currently active. Thus, your job as a software designer and programmer is to determine how to properly divide your desired functionality into states and to implement these in separate `ConcreteState*` classes. The remainder of this chapter provides a series of increasingly complex examples to demonstrate how this may be accomplished.

3 Example 1: Implementing a 3D Painting User Interface

Our first example demonstrates how to implement a basic 3D painting user interface. There is a long history of VR applications based on 3D painting metaphors. Early work in this style can be found in Clark's surface designing system [Clark, 1976], 3-Draw [Sachs et al., 1991], 3DM [Butterworth et al., 1992], and Holosketch [Deering, 1995]. A second generation of 3D painting user interfaces can be found in applications, such as BLUI [Brody and Hartman, 1999], CavePainting [Keefe et al., 2001, Keefe et al., 2005],

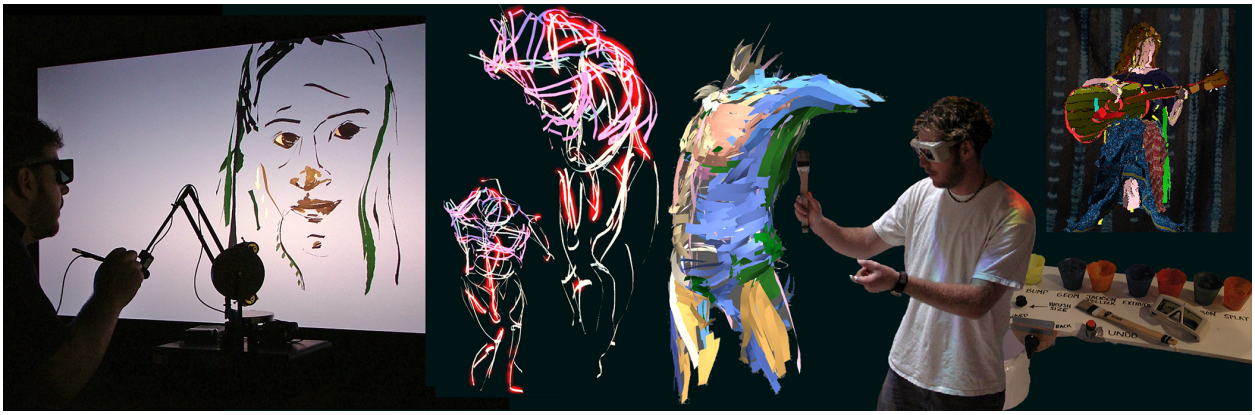

Figure 3: 3D Paintings created in VR using the CavePainting and Drawing on Air interfaces.

Surface Drawing [Schkolne et al., 2001], FreeDrawer [Wesche and Seidel, 2001], and Digital Tape Drawing [Grossman et al., 2002]. More recently, Drawing on Air [Keefe et al., 2007, Keefe et al., 2008b, Keefe et al., 2008a], Shape Modeling with Sketched Feature Lines [Perkunder et al., 2010], Mockup Builder [De Araùjo et al., 2012, De AraúJo et al., 2013] and Lift-Off [Jackson and Keefe, 2016] are representative of an ongoing emphasis within the VR research community on improving control in freehand 3D painting and drawing user interfaces. In parallel, 3D painting applications built in industry, including Tilt Brush [Google, 2017], have leveraged commodity hardware and modern graphics rendering to make the techniques widely accessible and even more compelling. The examples pictured in Figure 3 are from CavePainting and Drawing on Air.

3D painting interfaces are excellent examples for our discussion of how to build beyond-pointing VR interfaces. The common interaction in all of them – creating new virtual forms via sweeping, gestural movements of a tracked wand in space – requires interpreting 6-DOF input in a way that goes well beyond pointing. Artists report making dance-like, gestural movements that incorporate not just the 3D path of the brush but also the orientation along the path [Zen, 2004, Mäkelä et al., 2004, Keefe, 2011]. There are many possible extensions in these tools that also go beyond pointing, ranging from interaction with physical props and virtual widgets to bimanual precision 3D drawing interfaces.

Let's begin the technical discussion by describing the core hardware and input events we expect in a bimanual interface for 3D painting. Our description assumes that the user is holding two tracked devices, each with a single button. When the button on the device held in the dominant hand is pressed, the event named `DH_Down` is generated, and when it is released, `DH_Up` is generated. Likewise, the button on the device held in the non-dominant hand generates `NDH_Down` and `NDH_Up` events. Finally, `DH_Move` and `NDH_Move` events are generated whenever the tracking system reports a change in the position or orientation for the corresponding tracked device.

The basic 3D painting interface described in this example (Example 1) supports two key user interactions. First, the dominant hand is used to paint – when users press and hold a button on this tracked device, a stream of virtual "paint" begins to emerge from the brush, and as they move and twist the brush through the air, additional paint, which could take almost any form in VR, is deposited along the path. Second, the non-dominant hand is used to "grab onto" the virtual painting and move it around. We call this reframing the painting. While reframing, changes in the full position (location and orientation) of the non-dominant hand are applied to the `virtualToRoom` matrix, so it is possible to both translate and rotate the painting in order to get a good look at it

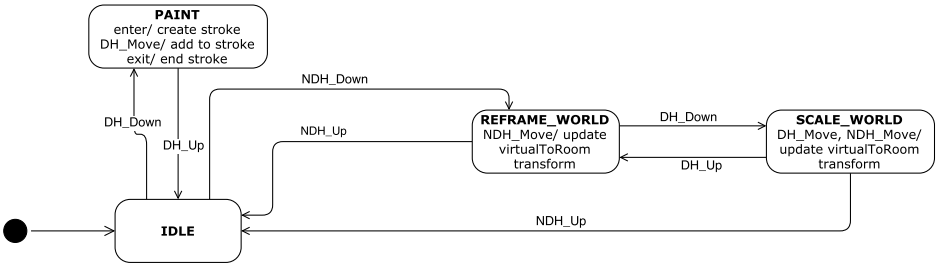

Figure 4: The finite state machine for a basic 3D painting user interface.

from different angles and place it in a comfortable position for making the next brushstroke. While reframing, it is also possible to enter a scaling mode. Simultaneously holding down the dominant hand button (so now, the button on each hand is depressed) activates a scaling mode where the scale of the `virtualToRoom` matrix is adjusted to scale the painting up or down in proportion to the distance between the two hands.

Figure 4 is a diagram of a state machine that can be used to implement this basic functionality. The interaction technique begins in the `IDLE` state. Notice that when a `DH_Down` event is generated, the state transitions to the `PAINT` state. This is the most important interaction in the whole user interface. When `PAINT` is entered, a new brush stroke is created. While in the `PAINT` state, each `DH_Move` event adds a new segment of geometry to this stroke. Finally, a `DH_Up` event triggers a transition back to the `IDLE` state.

Reframing is engaged from the `IDLE` state. A `NDH_Down` event triggers a transition to the `REFRAME_WORLD` state. From here, a `DH_Down` event triggers a further transition to the `SCALE_WORLD` state.

This last transition, from `REFRAME_WORLD` to `SCALE_WORLD`, is the first example in this chapter of a common pattern that we argue is a critical component of beyond-pointing user interfaces. Here, the `DH_Down` event triggers a different response depending upon the current state. In the `REFRAME_WORLD` state, `DH_Down` transitions to a scaling mode, whereas in the `START` state, `DH_Down` starts painting. Thus, the meaning of `DH_Down` is overloaded; it takes on a different meaning depending upon the context.

In this case, this overloaded behavior makes perfect sense. In general, the dominant hand is for painting and the non-dominant hand is for reframing, but once we have engaged a reframing operation, it is natural to extend this operation just a bit to adjust the scale in response to the motion of the two hands. There is no need to add a separate button to the input devices to support this, instead, we can reuse the same dominant hand button used for painting. This has an intuitive meaning for the user. Reframing and scaling can be described as, "grab on to the painting with one hand to translate and rotate, and grab on with two to scale it", and the user only needs to be able to locate a single button on each tracked device in order to perform all of the operations.

Now that we have designed an appropriate state machine, we can begin to translate the idea into actual code. Figure 5 is a class diagram that illustrates how this basic 3D painting state machine can be implemented using the State Design Pattern. `BasicPaint3DUI` serves as the main class that should be connected to the rest of your program. It includes a `currentState` member variable that points to an object of type `State`, which is an abstract class. There are four concrete implementations of `State`, one for each of the states defined in Figure 4. When `BasicPaint3DUI` is created, it will create one instance of each of the four concrete implementations and save a reference to each. Then, the

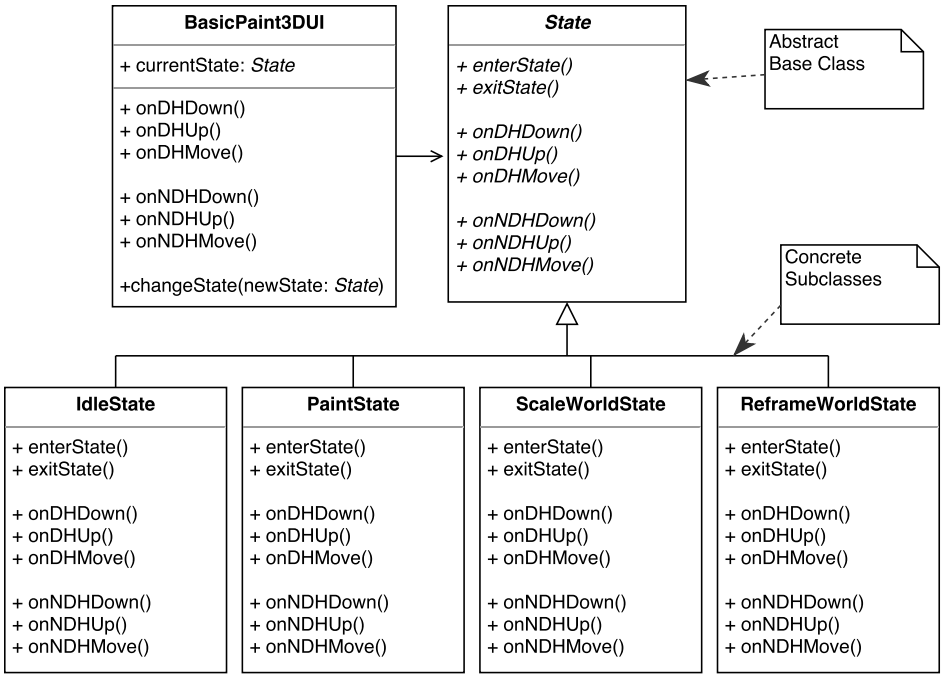

Figure 5: UML class diagram for the classes needed to implement the state machine diagrammed in Figure 4.

job of `BasicPaint3DUI` is simply to act as a pass-through object. When your program tells it that a `DH_Down` event has occurred by calling `onDHDown()`, it simply passes this event through to the current state, which will handle the event appropriately. In C++, the code to implement this pass-through feature can be very simple, as shown below.

```
void BasicPaint3DUI::onDHDown() {
  currentState->onDHDown();
}
```

Listing 1: Pass through an input event to the current state handler.

The implementation for passing other events through to the current state would follow the same pattern.

It is also important for `BasicPaint3DUI` to implement a method for changing the state, as shown below.

```
void BasicPaint3DUI::changeState(State *newState) {
  if (newState != currentState) {
    currentState->exitState();
    currentState = newState;
    newState->enterState();
  }
}
```

Listing 2: Handle a state transition.

The concrete implementations of `State` may then call `BasicPaint3DUI::changeState(..)` whenever they receive an event that should trigger a state change or otherwise determine that a state change is required.

With these key code snippets in mind, consider the following example implementation of `PaintState`, one of the concrete states in the basic painting interface.

```
class PaintState {
public:
  PaintState(BasicPaint3DUI *in_uiTechnique, State *in_startState) {
    uiTechnique = in_uiTechnique;
    startState = in_startState;
  }

  void enterState() {
    // create a new brush stroke object here.
  }

  void onDHMove() {
    // add geometry to the brush stroke object here.
  }

  void onDHUp() {
    uiTechnique->changeState(startState);
  }

private:
  BasicPaint3DUI *uiTechnique;
  State *startState;
};
```

Listing 3: Example implementation of the `PaintState`.

Of course in a real implementation the code needed to create a new brushstroke and add geometry to it is likely to be quite complex, so this `PaintState` class would need to interface with graphics libraries and be much more detailed in practice. Keeping this complexity isolated to a single class is another reason why treating the user interface code for the painting context as its own class is useful.

4 Example 2: Adding Proximity Events and Widgets

This second example builds upon the first, adding the features needed to turn the basic 3D painting interface of Example 1 into a complete application. The first addition is an interface to resize the virtual brush; it introduces the notion of proximity events, which are used several times throughout the remainder of the chapter. The second addition makes it possible to interact with 3D widgets (e.g., menus, color pickers, other virtual objects). As in Example 1, context will be important in both of these new features. If the artist user is in the middle of a grand, sweeping brushstroke and happens to move the brush into a widget, we wish to just ignore that contact with the widget and continue on with the painting operation. On the other hand, if the user is not actively painting at the moment, then moving the virtual brush within close proximity of a widget is a good indication that the user intends to work with that widget.

Figure 6 adds two new features to the same finite state machine pictured in Figure 4. The first feature, brush resizing, makes it possible for artists to adjust the thickness of the paintbrush, essentially adjusting the line weight of the virtual 3D marks it will create. Since this operation is

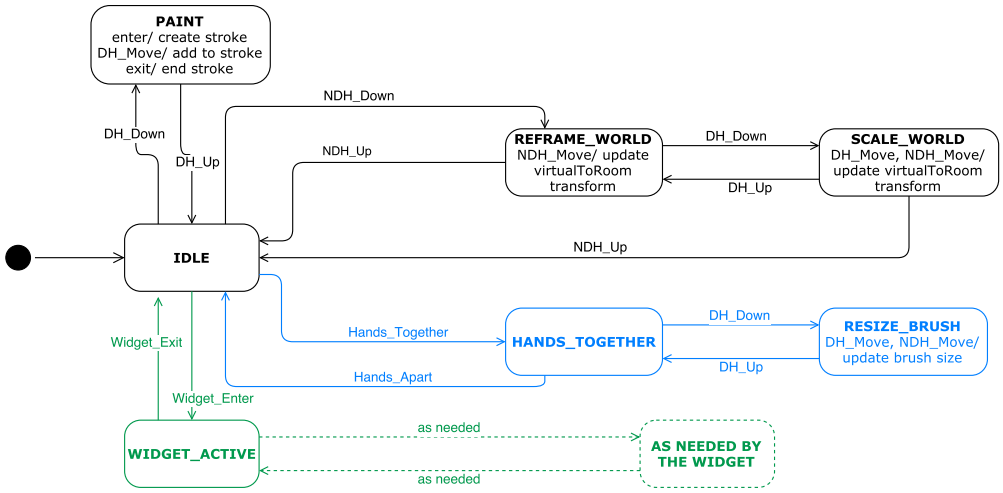

Figure 6: The basic 3D painting state machine is extended here to support brush resizing (blue) and interaction with a widget (green).

similar to scaling, a similar interface is used. The brush size is set to be proportional to the distance between the hands. The brush cursor is also adjusted during this operation – immediate visual feedback is important.

The most interesting aspect of this brush resizing interaction for our discussion is the way it is activated. Notice in Figure 6 that the `RESIZE_BRUSH` state is entered only after first entering a `HANDS_TOGETHER` state. This is a new state that we enter only when we determine that the two hands are within close proximity to each other, and serves to disambiguate the meaning of the `DH_Down` event. Since paint brushes are relatively small compared to the size of our bodies, users naturally put their hands close together in order to adjust the brush size. In contrast, the hands are rarely held close to each other while painting, since that is performed with the dominant hand while the non-dominant hand is at one's side. Thus, we use the proximity of the hands as a cue to disambiguate the user's intent. If the distance between the two hands is within a threshold (e.g., 15 cm), then we wish to transition from the `IDLE` state to the `HANDS_TOGETHER` state. From within the `HANDS_TOGETHER` state, a `DH_Down` event will trigger a transition to the `RESIZE_BRUSH` state. In contrast, if the hands are not within the proximity threshold, then a `DH_Down` event within the `IDLE` state will trigger a transition to the `PAINT` state as before. Visual feedback helps users understand how this proximity-based interface works, so it is recommended to include a change in the brush cursor to indicate that a button press will engage the resizing operation when within the `HANDS_TOGETHER` state.

There are several possible ways to implement this proximity-based functionality. Sticking with the event-based framework described thus far, we recommend detecting the instant the distance between the two hands falls under a threshold and treating this as a discrete `Hands_Together` event. Similarly, when the distance between the hands is greater than the threshold, this can also be treated as a significant event, which we call `Hands_Apart`. The advantage of this approach (generating new events) as opposed to including distance calculations and logic directly within the state machines is that the transitions remain simple – the state machine simply transitions whenever it receives the appropriate event. Of course, the distance calculations do need to happen somewhere, and for this we introduce the idea of Proximity Events – events, just like those generated from a regular input device, but generated dynamically in response to tracker input.

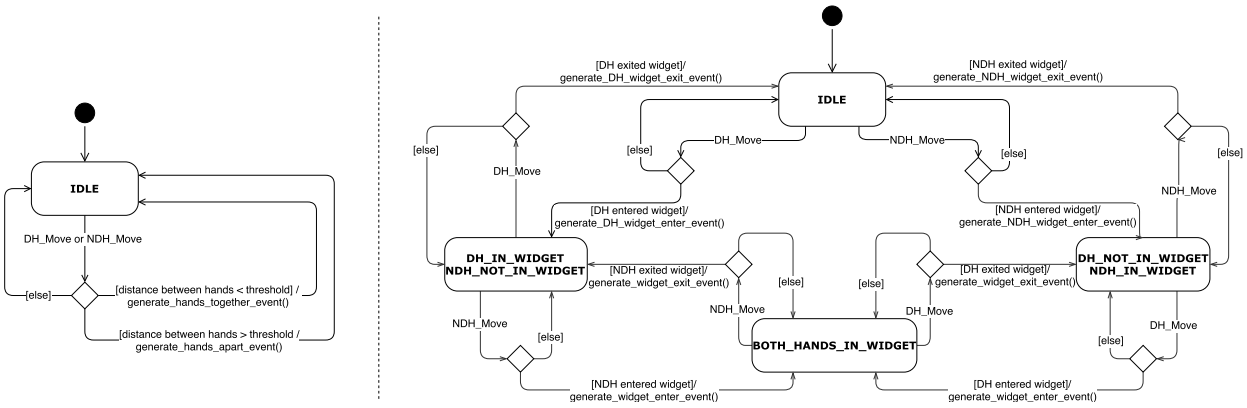

Figure 7: This Proximity Checker state machine, shown with two orthogonal regions running in parallel, acts as a virtual input device, interpreting `Tracker_Move` events from regular input devices and generating new events, for example, a `Hands_Together` event to signify the instant the tracked devices held in the hands comes within a short distance of each other.

The Proximity Events needed for the features described in this chapter can be generated by the state machine diagrammed in Figure 7. We call this our "Proximity Checker". The state machine contains two orthogonal regions running in parallel to output different types of Proximity Events.

For each new event received by the application, the event is passed first to the Proximity Checker, which acts as a virtual input device, and then on to the larger, application-oriented finite state machines, such as the one diagrammed in Figure 6. If the Proximity Checker generates new events, these are simply added to the current event queue. For example, notice in the left half of Figure 7 that the `Hands_Together` and `Hands_Apart` events mentioned earlier are generated in response to specific distance calculations. The same strategy is used to generate `Widget_Enter` and `Widget_Exit` events.

The second key feature introduced in the updated finite state machine diagrammed in Figure 6 is the ability to interact with widgets. For a 3D painting application, common widgets include a color picker, texture picker, menu of 3D style types (i.e., different styles of geometry for the paint), and menu of system control operations (e.g., load, save, print, undo, redo). All of these can be implemented with the same pattern illustrated in green within Figure 6. Here, the `Widget_Enter` event generated by the Proximity Checker is used to control the state transition. Because the system only responds to `Widget_Enter` when in the `IDLE` state, a widget (e.g., menu) can be activated as intended from this state but if we happen to hover over the menu while in the process of painting (i.e., while in the `PAINT` state), the `Widget_Enter` event is ignored, elegantly avoiding an unintended activation of the menu.

The Proximity Checker can be extended to test proximity to multiple widgets, each with its own `_Enter` and `_Exit` events. It is flexible for widgets needing to respond differently depending on how many hands are inside the widget. The right half of Figure 7 shows how a `DH_Enter_Widget` event is generated when the dominant hand enters a widget and a `NDH_Enter_Widget` event when the non-dominant hand enters.

The Proximity Checker can also be extended to test for more elaborate gestures. For example, `_Enter` events might be redefined to mean that a tracked device is approaching the location of the widget but does not yet fall within its bounds or to mean that a tracked device is pointing toward the widget or has performed some other more elaborate gesture. In some versions of the CavePainting

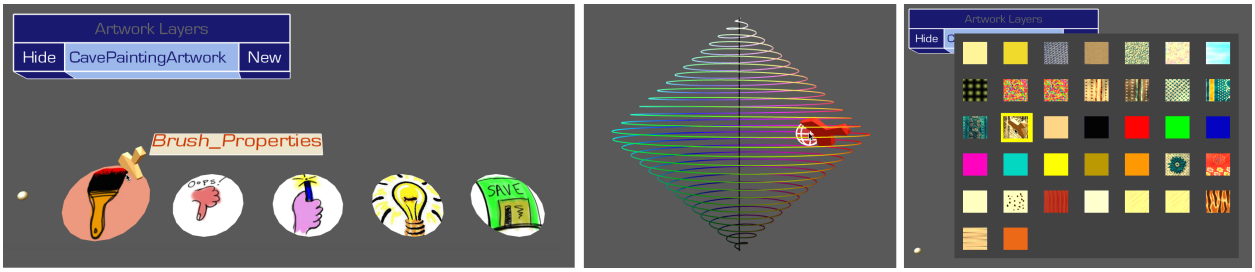

Figure 8: Example widgets used in CavePainting. Left: The blue "Artwork Layers" widget is an example of a 3D menu with a standard box-like layout. The palette of circular menus at the bottom activate in the location of the non-dominant hand when users flip their hand over, as if to look at their palm. The brush, controlled by the dominant hand, is currently hovering over the "Brush_Properties" sub-menu. Clicking and holding at this point would activate a range of choices displayed radially outward. Middle: A 3D color picker widget maps the Hue-Saturation-Value color space to a true 3D space, users work with this widget simply by moving the hand within the bounds of the double-sided cone. Right: A texture selection widget is used to change the visual appearance of the virtual paint strokes.

system a menu palette is activated at the location of the non-dominant hand when the user turns over their hand to look at their palm, a gesture that naturally defines a surface plane and evokes the idea of holding a palette in one's hand. Regardless of the method of activation, once the widget is activated, the system transitions to a `WIDGET_ACTIVE` state and further movements and button presses are handled as determined by the widget's state(s). Multiple widgets can be included in the design simply by duplicating the same pattern within the finite state machine (and renaming `WIDGET_ACTIVE` to be specific to each widget). Some widgets may be simple to implement, having just one state, and others may define additional states and transitions that hang off of their widget active state.

Figure 8 shows several example widgets from various versions of the CavePainting application developed at Brown University between 2000 and 2007. These widgets (various styles of menus, a 3D color picker) are relatively simple in that they can be implemented by adding just one or two "widget" states to the diagram in Figure 6. The next section builds upon these simple widgets, describing an example of a more complex, dynamic widget that includes multiple interaction modes and responds in real-time to the positions and orientations of both hands.

5 Example 3: Adding Constraints, Control, and Dynamic Widgets

This example presents additional possibilities for adding smart constraints, improving control, and integrating dynamic computation into VR widgets that are inspired by the Lift-Off 3D modeling interface [Jackson and Keefe, 2016] shown in Figure 9. The Lift-Off modeling interface makes use of several bimanual interactions and 3D widgets that go beyond pointing and are more complex than the previous examples. We will start by briefly summarizing the user interface followed by more in-depth discussions of specific implementation details.

5.1 Overview of the Lift-Off Interface

The Lift-Off workflow starts by creating 2D pencil-and-paper sketches (Figure 9a). These sketches are integrated into the VR environment as 3D slides placed in space. Slides are chosen and placed

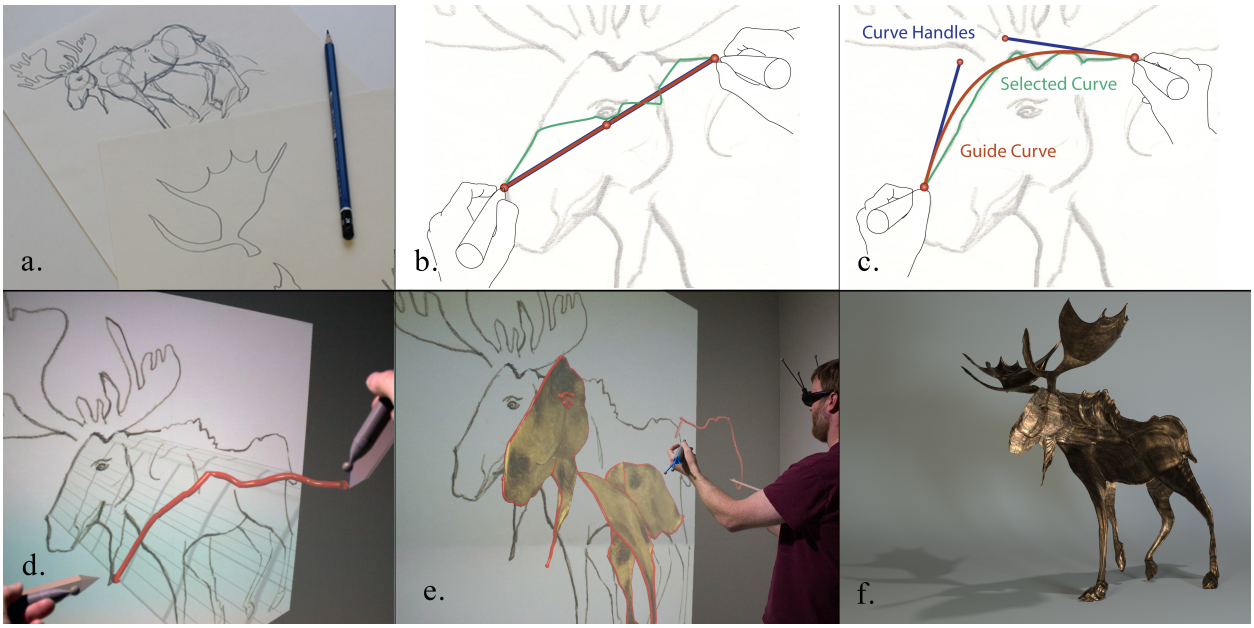

Figure 9: The Lift-Off 3D modeling interface. (a) 2D sketches are placed as 3D slides in the VR environment. (b) When both hands are within the activation distance of a slide, a guide curve (shown in red) is projected on the slide. (c) Rotating the tracked devices moves the curve handles changing the shape of the curve guide. The selected curve (shown in green) is influenced by the guide but is constrained to follow curve features identified automatically in the image. (d) The selected curve is lifted off the slide and placed in space as a rail. Rotation of the tracked devices changes the shape of the rail by bending to adjust depth. (e) Multiple rails create a wireframe. Surfaces are swept between rails to create the model. (f) The resulting 3D model rendered with Blender.

using the slide selection and placer widget shown in Figure 10.

After the slide is placed in space, the artist can select a curve from the 2D imagery. This is accomplished by placing both tracked devices inside the the activation area of the slide. The curve selection widget shows a cubic Bézier guide curve (shown in red in Figure 9b and c) that is projected on the surface of the slide. Each tracked device controls two of the Bézier control points (shown as the blue curve handles) to adjust the position and curvature. The guide curve gives visual feedback and functions like a magnetic rope. It pulls along the selected curve (shown in green) which settles onto the closest curve identified in the underlying pixel data of the slide texture. Clicking the button on the dominant hand's tracked device confirms the curve selection, and the curve now becomes a dynamic 3D widget called a rail that can be adjusted and bent in 3D space by rotating the tracked devices (Figure 9d).

After a series of these rail creation actions, the artist builds up a wireframe of connecting rails defining the outlines of the 3D model that they are creating (Figure 9e). Connections are facilitated by snapping the virtual cursors to the endpoints of existing rails, either in 3D space or projected onto the surface of the slide, when the tracked device is close to an existing endpoint.

Two final modeling operations are needed: first, the ability to divide a rail into two pieces to add new endpoints for future connections, and second, a way of creating surfaces between the rails to form the geometry of the 3D model. Approaching a rail with the tracked device will highlight it. Clicking and releasing a button will divide the rail. However, if the tracked device is swept away from the highlighted rail before releasing the button, then a surface will be created filling in the spaces between the particular rails indicated by the direction of the sweep gesture.

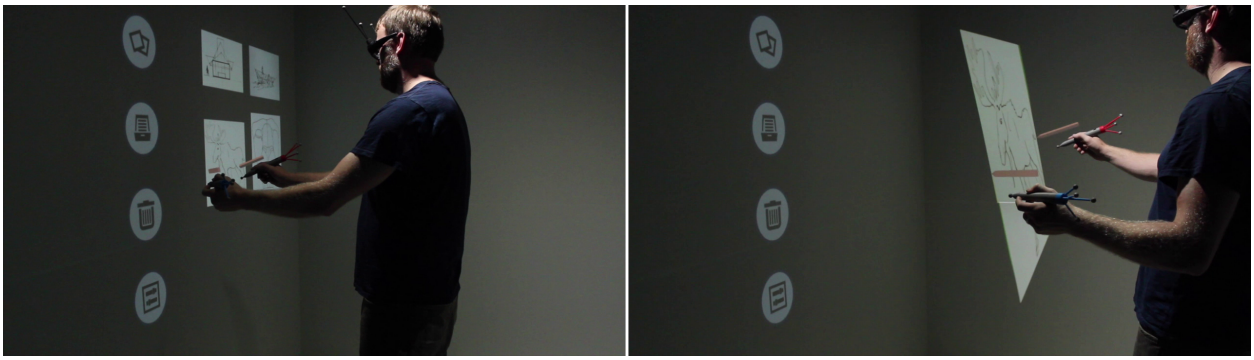

Figure 10: Left: A slide selection widget. Placing both tracked devices near a slide and creating a `DH_Down` event selects the slide, like lifting a picture off of a wall. Right: The selected slide sticks to the tracked devices until a second `DH_Down` event confirms the position. Rotating the tracked devices changes slide orientation and moving them further apart adjusts the scale. Note the red cursor representations of the tracked devices. The cursors are offset using a proxy matrix to avoid occlusions of the stereo images by the hands.

5.2 Implementing Gesture-Based State Changes

There are several interesting design and implementation details that make this interface more complex than the previous two examples. The complete Lift-Off finite state machine is diagrammed in Figure 13. Here, we explore the implementation details of a subset of the state machine used to split rails in two pieces or to create surfaces between them, as described in the previous section. The subset of the state machine is shown in Figure 11. This interaction illustrates an important difference from previous examples because changes from one state to another depend on gestural 3D movements, in addition to proximity or button presses.

Gestural state changes can be challenging to implement because they depend on device input over a sequence of time. The possible output states can also be divergent. For example in Figure 11 from the `RAIL_ACTIVE` state, a `DH_Down` event can signal the start of two possible transitions. If the `DH_Up` event is received before the tracked device is moved significantly, then the state transitions to the `SPLIT_RAIL` state. However, if the stylus is moved more than a specified distance from the rail, the system enters the `SURFACE_SWEEP` state and a surface is created. A `DH_Up` at this point will finalize the surface creation and return the state back to `IDLE`.

To implement this type of gestural state change, we introduce a new state called `DISTANCE_CHECKER`. In this state, the `DH_Move` event continues to update the cursor position, but it also calculates the distance the cursor has moved from the rail. If this distance increases above a threshold, then it will automatically transition to the `SURFACE_SWEEP` state.

5.3 Implementing Dynamic Widgets

Compared to the basic Proximity Checker used to activate widget states in Example 2, Lift-Off presents an added challenge. What makes this interaction different than the previous examples is that the slide and rail widgets are dynamic. They do not exist at the start of the program and their number depends on how many the user has created at runtime. Figure 12 extends the subset of the Lift-Off state machine shown in Figure 11. In this example the transition from `IDLE` to the `RAIL_ACTIVE` or `SLIDE_ACTIVE` states depends on the Proximity Checker calculating the distance to

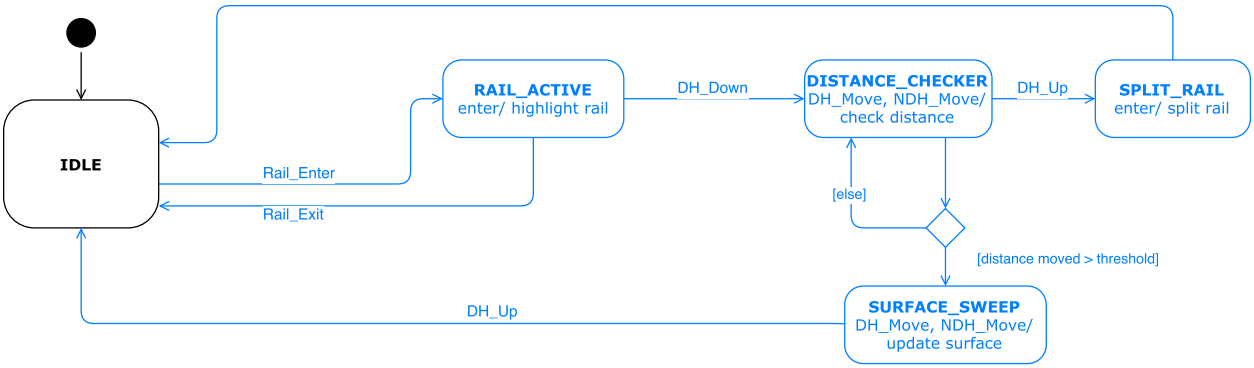

Figure 11: A subset of the Lift-Off modeling state machine used for dividing rails or sweeping surfaces between them.

the *closest* slide or rail to determine whether it is within a threshold. This requires additional computation, a common occurrence for dynamic widgets.

To implement this efficiently, the Proximity Checker, acting as a virtual input device, must contain a reference to a shared data structure holding a list of dynamic widgets, updated as they are created. The `3D_RAIL_PLACEMENT` state also needs to contain a reference to this data structure since it will add new rail widgets to the list.

At runtime, the list of dynamic widgets can be sorted to find the closest one using a custom comparison function. For each widget in the list, the function calculates the distance to the tracked object. Storing the widget data in a KD-Tree [Bentley, 1975] or other space-partitioning data structure will accelerate this process. The comparison function can also take into account the relative priority for different types of widgets. For example, the Lift-Off interface gives preference to slides when determining the closest object so that users can start to select new curves even when a 3D rail lies near a slide.

Implementing dynamic widgets also calls for interpreting the `DH_Move` events differently. For a static widget, the event data can report the tracker's location in Room Space (See the definition in Section 2.1.1). This makes sense when a static widget should always appear in the same position, for example near the right side of a display. However, for dynamic widgets the tracker events might need to be interpreted in Virtual Space. Here, the relative position of the slides and rail widgets directly depends on the virtual model being created, and the events must be interpreted in the same coordinate system, Virtual Space.

The slide widgets are dynamic in another way as well. Each slide contains a different 2D texture that is determined at runtime. As a result, the user interface to select curves must be flexible and controlled enough that users are able to indicate their desired selection regardless of the particular texture. In the next section, we describe how to implement contextual constraints (e.g. real-time image processing) to improve control of 3DUIs.

5.4 Contextual Constraints to Improve Control

A major challenge for beyond-pointing interfaces is that they can be difficult for users to control effectively. For example, CavePainting [Keefe et al., 2001] allows a user to create exciting gestural lines swept through the air, but it falls short of allowing an artist to draw a precise curve or even a perfectly straight line.

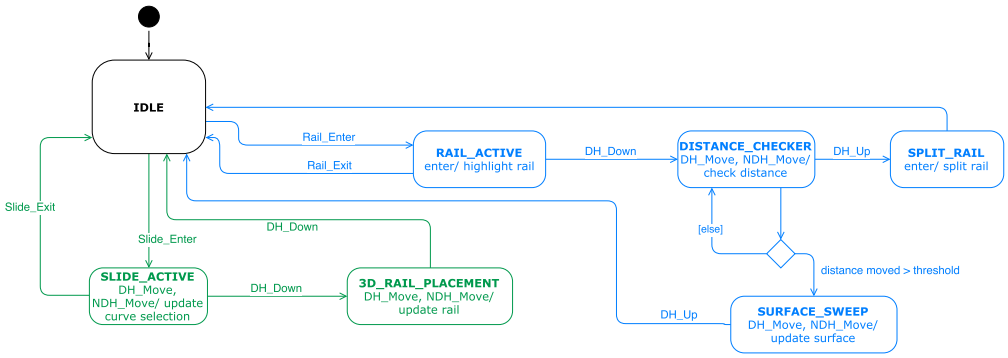

Figure 12: The Lift-Off modeling state machine is extended here to support rail creation (green).

Using Lift-Off as an example, we advocate for using smart contextual constraints to improve control while limiting the impact on freedom of expression. The use of this state-based framework for beyond-pointing interfaces makes them easier to implement because the interpretation of events (i.e. the interface context) is encapsulated inside a class representing a particular state.

Lift-Off makes use of contextual constraints in several ways. In the `SLIDE_ACTIVE` state, each hand only has 3-DOF (two for the xy-position of the guide curve endpoint on the slide texture and one for rotation to indicate the position of the curve handle). This level of constraint is possible because the actual selection is based on image-based computations to set the selected curve.

Control is further improved by adding the ability to snap the cursor to an endpoint of a previously selected curve when the cursor is nearby, constraining the movement to ignore small hand jitters. Snapping is accomplished by adjusting the cursor position. In this situation the cursor position differs from the tracked object position by means of a Proxy Matrix. Actions that are based on the tracker's position (like selecting curve endpoints) in either room or virtual space should take this Proxy Matrix into account.

Contextual constraints are also used in the `3D_RAIL_PLACEMENT` state. In this state, the user input is interpreted within the context of the guide surface (the transparent grid extending from the slide in the extruded shape of the selected curve, Figure 9d). Cursors are constrained to follow the closest point on the guide surface to each of the tracked objects' positions. The 3D rail is constrained to lie on the surface, although its shape can now be bent along this surface in 3D. These constraints make it possible to create precise curves and straight lines to model complex shapes with precision.

Lift-Off's finite state machine has a series of state transitions that linearly follow a series of steps. As seen in Figure 12, this still fits within the state framework described above. In fact, it would be extremely challenging to implement it without the state design pattern. This raises the question of when to add a new state to a program. We recommend creating a new state (i.e. a new program class) when the context of an interaction changes and input events must be interpreted differently. This is usually correlated with changes in visual feedback and the ways that the input should be constrained to add control. In this example, the `SLIDE_ACTIVE` and `3D_RAIL_PLACEMENT` states are distinct because the way they constrain the cursors and interpret the `DH_Move` and `NDH_Move` events is different.

When you want to enhance control in VR, turn to constraints. In developing user interfaces, you should find the things that you can do well with 6-DOF input and do them. Then, find the things that do not work very well and add constraints.

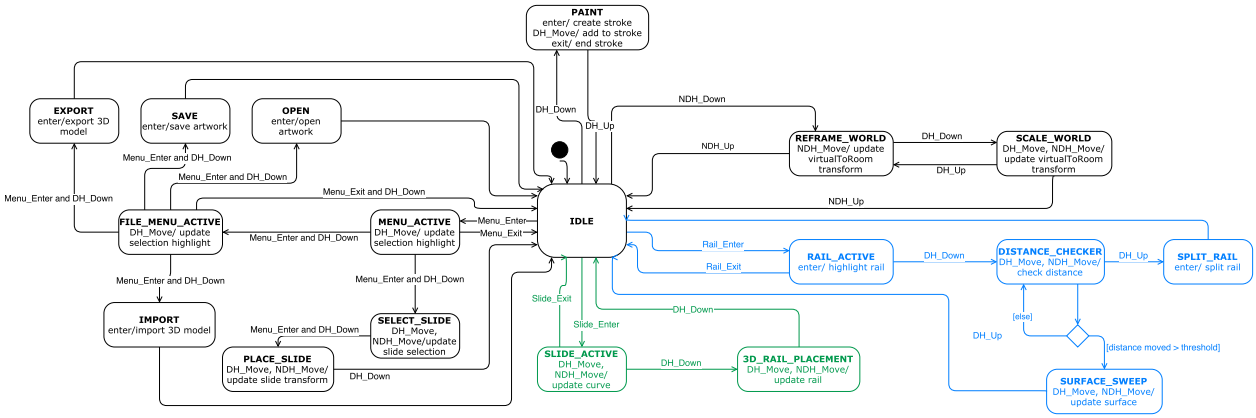

Figure 13: The Lift-Off modeling state machine builds on the basic 3D painting application. It has more complex transitions, including many states that are chained together. This transitional logic would be challenging to elegantly implement without the State Pattern.

5.5 Putting It All Together

In this example we have described how the Lift-Off interface can serve as inspiration for gesture-based state changes with dynamic widgets involving computation that can improve users' control through constraints. In addition to the rail creation process described in Section 5.1, Lift-Off also supports rail creation by directly painting rails in the air in the style of the 3D paint application in Example 1. It has additional states for reframing and scaling, as well as saving and loading models, and deleting the artwork. The full finite state machine for Lift-Off is diagrammed in Figure 13. In contrast to the previous two examples, this example shows how a more complex user interface can be implemented using states, and hopefully it provides motivation for adoption of this approach. Creating a complete system of this scale and complexity would be extremely challenging without the organizational structure of the state framework.

6 Conclusions

Many VR user interfaces can be implemented using this state-based UI framework. The examples presented here show only a few possibilities, but we hope that they have inspired the reader to create more complex user interfaces that go beyond pointing to accomplish more complex tasks. In designing new user interfaces, you should take full advantage of the amazing opportunity VR affords by holding magic wands in our hands, and you should use *both* hands. When you get two hands involved in the interaction, you should use state and the interaction context intelligently to provided significant functionality with a small number of buttons. Resist the urge to add one more button to the device for every new feature that comes along — your users will thank you when one button magically does what they want depending on the context.

References

[Bentley, 1975] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

[Brody and Hartman, 1999] Brody, A. W. and Hartman, C. (1999). BLUI: a body language user interface for 3D gestural drawing. In *Proc. SPIE, Human Vision and Electronic Imaging*, volume 3644, pages 356–363.

[Butterworth et al., 1992] Butterworth, J., Davidson, A., Hench, S., and Olano, M. T. (1992). 3DM: A three dimensional modeler using a head-mounted display. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pages 135–138, New York, NY, USA. ACM.

[Clark, 1976] Clark, J. H. (1976). Designing surfaces in 3-d. *Communications of the ACM*, 19(8):454–460.

[De Araùjo et al., 2012] De Araùjo, B. R., Casiez, G., and Jorge, J. A. (2012). Mockup builder: Direct 3D modeling on and above the surface in a continuous interaction space. In *Proceedings of Graphics Interface 2012*, GI '12, pages 173–180, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

[De AraúJo et al., 2013] De AraúJo, B. R., Casiez, G., Jorge, J. A., and Hachet, M. (2013). Mockup builder: 3d modeling on and above the surface. *Comput. Graph.*, 37(3):165–178.

[Deering, 1995] Deering, M. F. (1995). Holosketch: A virtual reality sketching/animation tool. *ACM Trans. on Computer-Human Interaction*, 2(3):220–238.

[Freeman et al., 2004] Freeman, E., Robson, E., Bates, B., and Sierra, K. (2004). *Head First Design Patterns: A Brain-Friendly Guide*. " O'Reilly Media, Inc.".

[Google, 2017] Google, I. (2017). Tilt brush.

[Grossman et al., 2002] Grossman, T., Balakrishnan, R., Kurtenbach, G., Fitzmaurice, G., Khan, A., and Buxton, B. (2002). Creating principal 3D curves with digital tape drawing. In *Proc. of the Conference on Human Factors in Computing Systems*, pages 121–128, New York, NY, USA. ACM.

[Jackson and Keefe, 2016] Jackson, B. and Keefe, D. F. (2016). Lift-off: Using reference imagery and freehand sketching to create 3D models in VR. *IEEE Transactions on Visualization and Computer Graphics*, 22(4):1442–1451.

[Keefe, 2011] Keefe, D. F. (2011). From gesture to form: The evolution of expressive freehand spatial interfaces. *Leonardo*, 44(5):460–461.

[Keefe et al., 2008a] Keefe, D. F., Acevedo, D., Miles, J., Drury, F., Swartz, S. M., and Laidlaw, D. H. (2008a). Scientific sketching for collaborative VR visualization design. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):835–847.

[Keefe et al., 2001] Keefe, D. F., Feliz, D. A., Moscovich, T., Laidlaw, D. H., and LaViola Jr., J. J. (2001). CavePainting: A fully immersive 3D artistic medium and interactive experience. In *Proceedings of I3D 2001*, pages 85–93.

[Keefe et al., 2005] Keefe, D. F., Karelitz, D. B., Vote, E. L., and Laidlaw, D. H. (2005). Artistic collaboration in designing VR visualizations. *IEEE Computer Graphics and Applications*, 25(2):18–23.

[Keefe et al., 2007] Keefe, D. F., Zeleznik, R. C., and Laidlaw, D. H. (2007). Drawing on air: Input techniques for controlled 3D line illustration. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1067–1081.

[Keefe et al., 2008b] Keefe, D. F., Zeleznik, R. C., and Laidlaw, D. H. (2008b). Tech-note: Dynamic dragging for input of 3D trajectories. In *Proceedings of IEEE Symposium on 3D User Interfaces 2008*, pages 51–54.

[Mäkelä et al., 2004] Mäkelä, W., Reunanen, M., and Takala, T. (2004). Possibilities and limitations of immersive free-hand expression: A case study with professional artists. In *Proceedings of the 12th Annual ACM Intl. Conference on Multimedia*, pages 504–507, New York, NY, USA. ACM.

[Perkunder et al., 2010] Perkunder, H., Israel, J. H., and Alexa, M. (2010). Shape modeling with sketched feature lines in immersive 3d environments. In *Proceedings of the Seventh Sketch-Based Interfaces and Modeling Symposium*, SBIM '10, pages 127–134, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Sachs et al., 1991] Sachs, E., Roberts, A., and Stoops, D. (1991). 3-draw: A tool for designing 3D shapes. *IEEE Computer Graphics and Applications*, 11(6):18–26.

[Schkolne et al., 2001] Schkolne, S., Pruett, M., and Schröder, P. (2001). Surface drawing: Creating organic 3D shapes with the hand and tangible tools. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '01, pages 261–268, New York, NY, USA. ACM.

[Wesche and Seidel, 2001] Wesche, G. and Seidel, H.-P. (2001). Freedrawer: A free-form sketching system on the responsive workbench. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '01, pages 167–174, New York, NY, USA. ACM.

[Zen, 2004] Zen, J. (2004). Painting in air. *Comput. Graph.*, 38(3):7–9.