Abstract of "Systems for Sketching in 3D" by Jonathan Cohen, May 2000.

In this thesis we describe two systems for sketching in 3D with a 2D input device.

The first system allows the user to draw 3D stick figures. This system includes a novel technique for specifying 3D curves with 2D input from a single viewpoint. The user first draws the curve as it appears from the current viewpoint, and then draws its shadow on the floor plane. The system correlates the curve with its shadow to compute the curve's 3D shape. This method is more "natural" than existing methods in that it leverages skills that many artists and designers have developed from work with pencil and paper.

The second system, *Harold*, combines ideas from existing techniques and introduces new concepts to make an interactive system for creating 3D worlds. The interface paradigm in Harold is drawing: all objects are created simply by drawing them with a 2D input device. Most of the 3D objects in Harold are collections of planar strokes that are reoriented in a view-dependent way as the camera moves through the world. Virtual worlds created in Harold are rendered with a stroke-based system so that a world will maintain a hand-drawn appearance as the user navigates through it. Harold is not suitable for representing certain classes of 3D objects, especially geometrically regular or extremely asymmetric objects. However, Harold supports a large enough class of objects that a user can rapidly create expressive and visually rich 3D worlds.

We conclude with a discussion of 3D sketch-based interfaces in general, and propose a framework for analyzing them. From this analysis, we propose interface design principles and a general interface design methodology. We demonstrate this methodology by designing a system for sketching motion paths in 3D. Systems for Sketching in 3D

by Jonathan Cohen

Senior Thesis

Providence, Rhode Island May 2000 © Copyright 2000 by Jonathan Cohen

Acknowledgements

Robert Zeleznik, John Hughes, Lee Markosian, and Ronen Barzel have contributed many of the ideas and some of the text of this thesis. I wish to thank each of them for their generosity with their time and for their interest in my work. Parts of this thesis have appeared previously as [10] and [9].

On a personal note, I'd like to thank Joe Kaneko, Sharif Corinaldi, Michael Rymer, Ben Smith, Justin Bailey, Marc Majzner, Rafi Pristoop, Mark Henle, and my family for all the good times.

Thanks to the artists and art students who used Harold and gave us feedback, especially Michael Legrand and Noah Raford. Thanks also to Lee Markosian and J.D. Northrup for providing the stroke-based rendering system used in Harold. Thanks to Loring Holden for all of his help with the development of these systems and for his general good humor. Special thanks to my advisor John Hughes and to my second reader Nancy Pollard. Finally, thanks to Andy van Dam and the entire graphics group for their support.

Contents

List of Figures						
1	\mathbf{Intr}	roduction	1			
2 Sketching 3D curves						
	2.1	Introduction	3			
	2.2	Previous work	4			
	2.3	Overview of the system	4			
		2.3.1 Drawing curves	5			
		2.3.2 Details of shadows	7			
		2.3.3 Correlating curves with shadows	8			
		2.3.4 Strut manipulation	13			
	2.4	Extending the system to stick figures	13			
	2.5	Basic operations	14			
	2.6	Discussion	16			
	2.7	Future Work	17			
3	Har	cold: sketching 3D environments	20			
	3.1	Introduction	20			
	3.2	Related work	23			
	3.3	The user's view of the system	23			
	3.4	Implementation details	27			
		3.4.1 Storing collections of strokes	27			
		3.4.2 Billboard transformations	27			
		3.4.3 Bridge transformations	29			
		3.4.4 Rendering strokes	30			
		3.4.5 Terrain	32			

	3.5	Limita	ations and discussion	33					
	3.6	Future	e work	34					
4	Ske	tch-bas	sed interfaces	38					
	4.1	Interfa	ace components	38					
		4.1.1	Input device layer	39					
		4.1.2	Feedback layer	41					
		4.1.3	Gesture set layer	42					
		4.1.4	Inferencing layer	43					
		4.1.5	Model layer	44					
	4.2	Case s	tudy: sketching a motion path	46					
5	Con	nclusio	n	49					
Bibliography 50									

List of Figures

2.1	A single stroke creates the initial curve
2.2	A second stroke defines the curve's shadow $\ldots \ldots 5$
2.3	An overdrawn stroke
2.4	The system merges the overdrawn stroke into the original curve 5
2.5	Demonstrating image-space alignment
2.6	Critical points
2.7	The shadow surface
2.8	The curve stays on the surface
2.9	The curve has more critical points than the shadow 10 $$
2.10	An invalid curve-shadow pair $\ldots \ldots \ldots$
2.11	Warping the shadow
2.12	The curve defines a surface $\ldots \ldots \ldots$
2.13	An ambiguous curve-shadow pair $\ldots \ldots \ldots$
2.14	Strut manipulation
2.15	A simple stick figure
2.16	A simple 3D stick figure created in our system $\dots \dots \dots$
2.17	Splitting and manipulating curves
2.18	A tetherball
2.19	A camera path through a virtual environment
2.20	Misaligned curve and shadow $\ldots \ldots 19$
9.1	Creating a hillbaard
ე. I ე. ე	A bridge billboard
ე.⊿ ე.ე	A bridge bribboard
3.3 9.4	Ground strokes 24
ა.4 ე.5	Equing terrain
3.5	The interface table for Harold
3.6	Examples of the stroke styles in Harold

3.7	A mountain scene made in Harold 3	35
3.8	A dragon scene in Harold 3	36
3.9	A conceptual sketch of an outdoor scene	36
3.10	The same scene from a different viewpoint	37
3.11	A moon scene created in Harold 3	37
4.1	The components of a sketch-based interface	39
4.2	The interface process	ŧ0
4.3	Torso model made with Skin	15
4.4	Frog motion path	17

Chapter 1

Introduction

Interactive 3D computer graphics has grown out of the needs and ideas of the engineering and CAD communities. The original interactive graphics systems (most of which were in 2D, e.g. Sutherland's Sketchpad system [41]) were created with mechanical design as their prototypical applications. This history is reflected in many of the concepts and paradigms that permeate modern 3D computer graphics. An engineer, for example, is likely to be familiar with concepts from differential geometry such as tangent vectors, differentiability, and manifolds. The result is that a typical interactive modeling tool's interface might assume some understanding of, say, spline patches, surface modeling, or algebraic surfaces.

This engineering-based approach has led to rapid development in the field of interactive computer graphics by allowing programmers and researchers to exploit computers' strength, mathematical modeling. The result has been the creation of extremely powerful and expressive computer graphics tools. A side effect, however, had been that the burden for translating mental models and visual design into a form computers can understand is placed primarily on human artists and designers.

Current popular graphics systems such as Maya [1] or 3D Studio [2], while powerful and useful, force users to learn to think in terms of the computer's mathematical models. While talented artists can learn to use these systems and are able to create expressive and complex images, using these systems is much more difficult for those without the time or training required to learn about the mechanics of subdivision surfaces and spline patches.

As computing power increases, and with growing interest in the creation of computer systems usable by people with minimal technical training, researchers have begun to explore ways of interacting with computers that allow users to express themselves more "naturally." For this to happen, computers must take over the task of converting these natural instructions into operations on mathematical models. In computer graphics, this research direction has materialized in the form of sketch-based systems. The appeal of sketch-based interfaces for computer graphics is summarized by Zeleznik et al. in [47], "Traditionally, people have attacked conceptual design with paper and pencil, not with computers, even though computer models offer numerous advantages. The reasons for this include the low overhead of a single-tool interface (pencil), the lack of special knowledge needed to draw, the ease with which many kinds of changes can be made, and the fact that precision is not required to express an idea."

The problem that we investigate in this thesis is how to create interactive computer graphics systems that allow the user to focus exclusively on the visual and intuitive tasks of specifying "what something looks like," while letting the computer infer the necessary mathematical models. What types of mathematical models are suited to these tasks? What is hard about this problem, and what constraints exist on possible solutions? What approaches have been tried, which approaches are feasible, and which are not? Are there general principles we can apply to help design an interactive system to suit our particular needs?

In the rest of this thesis, we discuss two systems that we have created, one for sketching 3D stick figures and curves, and the other, *Harold*, for sketching 3D scenes. In the final chapter, we survey other sketch-based systems and propose some general principles that may be useful for designing interactive systems for sketching in 3D.

Chapter 2

Sketching 3D curves

2.1 Introduction

Specifying 3D curves is one of the most important tasks that a 3D user interface must support. Curves are used in modeling and CAD systems to specify surface patches [2, 1], as skeletal shapes for implicit surfaces [6], and to define controls for object deformations [38]. Animation systems and VR applications use curves to specify motion and camera paths [11, 31, 22].

Many authors have recognized the importance of specifying curves directly [14, 16, 49, 5, 47, 23]. Although sketched curves are imprecise by nature, sketching allows a user quickly to create a curve that is close to the desired result, even if she has little experience with the underlying curve representation. A novice user can quickly create approximate curves because little overhead is required to learn the interface. A trained artist can apply her existing drawing skills to produce accurate curves because the interface more closely matches the one she is used to – namely pencil and paper.

The technique we present is an extension of the idea used in [17, 47] that a point in 3D can be determined from its image-space projection together with that of its "shadow." (The "shadow" is just the vertical projection of the point onto some horizontal surface.) We apply this idea to a connected set of 3D points to define a curve. With this approach, the user sketches a curve directly into a scene in two strokes: first drawing the curve as it appears from the current viewpoint, and then sketching its appearance unchanged. The user can then refine portions of the curve by over-sketching either its projected image or that of its shadow. Although this technique is less precise than existing ones, it lets

the user quickly sketch a reasonably correct shape that may be further refined with more conventional methods.

2.2 Previous work

Some techniques for editing curves are indirect in that they require the user to modify parameters, e.g. spline control points or knot values, that in turn affect the curve's shape. Other techniques allow direct manipulation of the curve itself, such as the overdrawing paradigm described by Baudel [5] and direct manipulation of spline curves [14, 16, 49]. The technique we present falls into this latter category.

Although much work has been done in sketching 2D curves [5, 23, 4, 37], few systems have addressed the issue of sketching curves in 3D. One notable exception is the 3-Draw system [35], which uses Polhemus trackers attached to a stylus to allow a designer to sketch in 3D with arm motions.

Commercial 3D modeling systems such as Maya and 3DStudio [1, 2] give the user a variety of techniques for creating and editing curves. Most of these, however, are indirect (e.g. the user edits spline control points or intersects two surfaces). In Maya a user can sketch a curve directly onto a user-defined plane, or more generally onto a surface. Of these techniques, only the latter constitutes a direct method for specifying nonplanar curves. But the user can draw on a surface only where nothing occludes it. To draw all the way around a sphere, for example, the user must draw from multiple camera positions. Thus, there are restrictions on the types of curves that can be sketched from a single view.

The interface we present could complement existing 3D modeling systems by providing additional flexibility for directly sketching nonplanar curves.

2.3 Overview of the system

We support four basic operations for sketching curves: drawing a new curve in some plane, "overdrawing" a section of an existing curve, redefining a curve's entire shadow, and overdrawing a section of a curve's shadow. Figures 2.1, 2.2, 2.3, and 2.4 illustrate the steps involved in creating and editing a curve. To distinguish between operations that edit the shadow and operations that edit the curve, the user selects either shadow mode or curve mode via a menu or keyboard shortcut.

When the user draws a stroke in curve mode or shadow mode, the system determines whether the stroke is an "overdraw" by checking whether it starts and ends near and nearly



Figure 2.1: A single stroke creates the initial curve.



Figure 2.2: A second stroke defines the curve's shadow and hence its 3D shape.



Figure 2.3: The dashed line indicates an overdraw stroke.



Figure 2.4: The system blends the overdraw with the original curve to get the final result.

parallel to an existing curve or shadow. If so, we merge it into the existing curve using a method similar to that described in [5]. In curve mode, if the stroke is not an overdraw, the system interprets it as a new curve that is projected onto a plane; the plane is determined by a set of heuristics described below.

To define a shadow, the user (in shadow mode) draws a stroke beneath the curve to be modified. If the stroke appears to be an overdraw, the system blends it into an existing shadow. If there is no existing shadow with which the new shadow can be merged, we test whether its endpoints lie approximately below some curve's endpoints. If so, we take this to mean than the the curve's shadow was entirely redrawn. (If not, the stroke is rejected.) Finally, we reproject the curve back into the scene to match its new shadow.

2.3.1 Drawing curves

We represent 3D curves as parameterized polylines, i.e., as piecewise linear curves defined by a mapping from $[0,1] \rightarrow \Re^3$. Before they are used to define curves or shadows, input strokes are smoothed in the following way. First we filter the stroke to remove all points whose screen-space distance is less than some threshold (e.g. 25 pixels) from the previous point. We fit a Catmull-Rom spline [13] to the remaining points and sample the spline every few pixels to generate a smooth-looking polyline.¹

Some kinds of curve strokes, such as a straight line segment connecting two points, are simple to interpret without extra information. Our system should make it relatively easy for a user to draw these types of curves. However, for more complicated curves, we are not all that concerned with being able to interpret them correctly at first, since the user will redefine them by drawing a shadow. Here we discuss how we interpret strokes to place 3D curves initially. Later, we describe our technique for editing these initial curves to create more complicated shapes.

We have experimented with several heuristics for placing a 3D curves initially. In general, these heuristics involve choosing a plane and projecting the stroke onto that plane to produce a 3D curve. The system has a list of rules that it applies to each input stroke to determine an appropriate plane. The advantage of this design is that we can rapidly prototype several different schemes for specifying how to interpret input strokes just by changing the heuristics. A drawback, however, is that a heuristic tend to be specific for particular classes of curve. For example, a set of heuristics tailored to make it easy to draw human forms might not work very well for drawing six-legged insects, because the objects have different kinds of curves. One solution would be to have different sets of heuristics from which the user could select depending on the type of curve to be drawn, although this adds a lot of complexity to the system.

Although we have experimented with several different sets of heuristics, we list only a few representative examples here. A general heuristic that applies to most stick figures is:

Heuristic 1 If either endpoint appears to lie on an existing point or curve, we take this as intentional and place the endpoint in 3D so that it lies on the existing object. We then choose a plane that contains the endpoint (or points). Since one or two points do not uniquely determine a plane, we choose, among all planes containing them, the one that is most nearly screen-parallel.

A general heuristic that produces easily understandable results, yet is too simple to be of much use is:

Heuristic 2 If neither endpoint appears to lie on an existing point or curve in the scene, we determine which plane to use from the angle of the camera. If the camera is looking

¹This smoothing step, while independent of the overall technique, is important since noise in the input device propagates to the final 3D curves.

down, we use the floor plane, and if the camera is at an oblique angle, we use the plane perpendicular to the floor plane that is most nearly screen-parallel.

A heuristic for drawing curves coming up off the floor plane is:

Heuristic 3 If both endpoints of the stroke lies over the floor plane, cast a ray through each endpoint and intersect these two rays with the floor plane to get two points p and q. Say q is closer to the camera position. Choose the vertical plane containing q that is most nearly screen-parallel. Project the stroke onto this plane. If all points of the projected stroke lie above the floor plane (i.e., have non-negative y-value), then use this projection as the curve. If not, project the stroke onto the floor plane, and use this projection as the curve.

This rule makes it easy to sketch a leg for a human form, for example. The user begins a stroke at the foot and extends it upward to the hip. The resulting curve will lie in a vertical plane facing the camera as intended.

2.3.2 Details of shadows

A shadow is a 3D curve obtained by projecting another 3D curve along a fixed vector, which we call the *projection vector*, onto some surface, called the *projection surface*. In this discussion, we always use the world Y axis as the projection vector, and we always project onto the floor plane. These choices are arbitrary – we could just as easily use the world X vector and let the user draw shadows on a wall. Also, note that in all of our examples, the shadow is a planar curve. This assumption is not necessary for any of the algorithms described below. Thus, we could project shadows onto rolling terrain, for instance.

The key feature of this system is the ability to edit a curve via its shadow. As noted in [17, 47, 46], a point's location is determined uniquely by its appearance from an oblique camera position and by its shadow. We extend this idea to curves: the shape of a 3D curve is determined by its image-space projection and its shadow.² Thus, to modify a curve's shape in our system, the user redraws its shadow. This redefines the curve's shape while leaving its appearance from the current camera position unchanged.

It can be difficult to draw a valid shadow for a given curve. To facilitate this, we draw vertical guidelines at both ends of the curve. These lines provide feedback that helps the user align the shadow with the curve. Also, the matching algorithm does not require that the curve and shadow be exactly aligned, only that they be "close," as explained in the next section.

²In certain cases described below, the curve's 3D shape is not determined uniquely.

2.3.3 Correlating curves with shadows

Once a curve's shadow or image-space projection has been redefined, we project the curve back into the scene using the following method.

We assume either a perspective or orthogonal projection, with the restriction that the camera's "look vector" is not close to parallel with the projection vector. (Recall that the projection vector is the vector along which we project curves to get their shadows.) Further, in the case of a perspective projection, we require that the vanishing point for lines parallel with the projection vector be off-screen. This allows us to define a left-to-right ordering of 3D points (see figure 2.5). To test if a point A is left or right of another point B, we project A into the image. Then we take the line parallel to the projection vector running through A's world location and project this line into the image. This line (call it l) partitions the image into two sections, one to the left and one to the right. If the image-space projection of B is to the left of l, we say B is *image-space aligned*. If B is within a distance d of l, we say B is aligned to within d of A. ³

We define a point p on a curve to be a *critical point* if all points in some neighborhood of p on the curve lie to the right or left of the point in image space. Note that by this definition, the first and last points of a curve are critical points (see figure 2.6). A *span* is the section of a curve between two critical points.

The key observation we use to match a curve with its shadow is that the shadow defines a ruled surface formed by extruding the shadow along the projection vector. An interior critical point in the shadow corresponds to a silhouette of the surface, as shown in figure 2.7.

To redefine a curve's 3D shape from its shadow, we must project the curve onto this possibly many-layered surface. To do this, we must determine onto which layer of the shadow surface we should project each point of the curve. In general, a critical point on the shadow must correspond with a critical point on the curve. This is because the curve must turn around where the shadow surface turns around in order to stay on the shadow surface, as shown in figure 2.8. Note that there may be critical points in the curve that do not correspond to critical points of the shadow, as in figure 2.9.

If there is no way to project the curve onto the surface so that the resulting 3D curve is continuous, the shadow is invalid and we reject it. Figure 2.10 shows an invalid shadow.

To create the correspondence between layers of the shadow surface and spans of the

³Note that in a perspective projection, *aligned to within* d is not a symmetric relation. In other words, A may be aligned to within d of B, while B is not aligned to within d of A.



Figure 2.5: A and B are image-space aligned through the line l, and C is image-space right of both of them.



Figure 2.6: A, B, and C are critical points of this curve.



Figure 2.7: The shadow defines a ruled surface with a silhouette above the interior critical point of the shadow, B.



Figure 2.8: The curve must turn around at B to stay on the surface.



Figure 2.9: The curve may have more critical points than the shadow and still be valid.



Figure 2.10: There is no way to project this curve onto the surface to get a continuous 3D curve.



Figure 2.11: The dashed line indicates the original shadow. The solid line indicates the shadow after it has been adjusted to match the curve.

curve, we need to match critical points of the shadow with critical points of the curve. First, we find the critical points of the shadow and curve. Let c_0, c_1, \ldots, c_n be the list of critical points of the curve, sorted by parameter value, and s_0, s_1, \ldots, s_m be the sorted list of the shadow's critical points.

To begin, we verify that c_0 corresponds with s_0 . (At this point, we may have to reverse the parameterization of the curve to make these points match.) We iterate through the critical points of the shadow in order and attempt to match each point s_i with some critical point on the curve. We do this as follows.

First, let c_j be the next unmatched critical point on the curve. If c_j is aligned to within 25 pixels of s_i , then c_j is matched with s_i , and we go on to s_{i+1} . Otherwise, we test if c_j is between s_{i-1} and s_i (in terms of the image-space left-to-right ordering). If not, then there is no valid match because there is no shadow underneath some span of the curve adjacent to c_j . Also, if c_j is the final endpoint of the curve and is unmatched, then there is no valid match. Finally, we increment j and repeat for the next critical point on the curve.

The system can match a shadow with a curve even when the shadow does not align exactly with the curve, as follows. After creating a correspondence between critical points, we deform the shadow so that all matching critical points are precisely image-space aligned. This is done by rotating and scaling each span of the shadow to align it with the corresponding span of the curve, as shown in figure 2.11.⁴ This step allows the user to sketch an approximate shadow, leaving it to the system to ensure that curve and shadow are precisely aligned.

We now have a valid aligned shadow and a correspondence between each span of the

⁴For this reason, our previous claim that the shadow surface may be non-planar must be qualified. We need to be able to warp the shadow in such a way that it still lies on the projection surface. When the projection surface is planar, this is easy because we can scale along a vector orthogonal to the plane normal and rotate about an axis parallel to the plane normal. When the projection surface is non-planar, this may be more difficult.



Figure 2.12: The curve defines a surface containing all rays from the viewpoint through each point on the curve. We intersect this with the shadow surface to get the final 3D curve.



Figure 2.13: The shadow and curve do not determine a unique 3D curve. This is because all four endpoints are image-space aligned.

shadow and some span of the curve. Just as the shadow defines a particular surface, we can think of the curve as defining a unique surface containing all rays extending from the camera through the image-space projection of the curve. We intersect each portion of the curve surface with the corresponding layer of the shadow surface to produce a section of the 3D curve, as shown in figure 2.12. Because we use a piecewise linear representation for our curves, we intersect these two surfaces by breaking them up into planar segments and intersecting the corresponding segments. We splice all such sections together to get the final 3D curve.

Near a critical point on the shadow, the tangent plane to the shadow surface is oriented nearly edge-on to the camera. This has the effect of magnifying noise in the 2D input: that is, small variations in the input stroke result in large variations in depth for the 3D curve. To alleviate this problem, we remove points that are nearly aligned to critical points of the shadow, replacing them with a smooth spline that joins neighboring sections of the 3D curve. Finally, we perform the same filtering and smoothing operations described above to improve the smoothness of the final curve.⁵

In certain cases this algorithm may produce an unintended result. This can happen when the curve and shadow have multiple critical points that are image-space aligned. One example is shown in figure 2.13. In a case such as this, the image-space curve and shadow do not define a unique 3D curve. Our algorithm will find one possible 3D curve, but it

⁵Because we perform these filtering and smoothing steps, the appearance of the curve is not constant – it often changes by a few pixels after each edit.



Figure 2.14: The three vertical lines are struts. The dashed line indicates the original position of the shadow. Dragging the center strut downwards moves the shadow along the floor while fixing the appearance of the curve.

might not be the intended one.

2.3.4 Strut manipulation

We can consider the problem of correlating a curve with a shadow to be an instance of the general problem of matching features in two signals. From this point of view, we wish to extract the "salient" features of the image-space curve and shadow, register the two curves to align these features, and finally calculate the final 3D curve.

In our current system, we take into account only the critical points of the image-space curve and shadow. That is, critical points are the only features we consider to be salient. The system ignores other features, such as bends, because we assume that the user will draw such features in correct alignment (figure 2.20 shows what happens when this is assumption is incorrect).

It might be useful to provide automatic registration of such features – one possible way to do this would be to adapt the dynamic time warping algorithm from [8] to register the curve with the shadow.

Though we do not currently support this more general notion of signal matching, we do let the user explicitly align certain features. To facilitate this, we allow the user to draw *struts*, which are lines parallel to the projection vector that connect a fixed point on the shadow with a fixed point on the curve. After placing a strut, the user may drag its bottom up or down. This has the effect of adjusting the shadow while leaving the appearance of the curve unchanged, as shown in figure 2.14. The shadow is affected just in the span between the two neighboring struts.

2.4 Extending the system to stick figures

The techniques was have described allows a user to sketch interesting 3D curves and edit them in an intuitive way. In this section, we described how we have extended these ideas to support sketching stick figures.



Figure 2.15: A simple stick figure.

Stick figures seem easy to draw. A five-year-old child can draw stick figure animals and people. Yet despite their simplicity, stick figures are an important step in creating more complex drawings. Experienced artists often begin drawing human forms by sketching their underlying structures as stick figures. Stick figures nicely convey shape information, proportions, and relative position of different parts of an object. They also easily express connectedness, i.e., it is easy to see in figure 2.15 that the arm is connected to the torso at the shoulder joint.

Stick figures have a natural extension from 2D to 3D. Instead of a drawing made up of connected 2D strokes, we want a model made up of connected 3D curves. Our goal then becomes designing an interface that would enable a user to sketch 3D stick figures with the same ease with which she would sketch 2D stroke with pencil and paper.

We formally define a stick figure as a graph G = (V, E) where each vertex $v \in V$ corresponds to a point $p_v \in \Re^3$, and each edge $e = (u, v) \in E$ corresponds to a continuous curve C_e of finite length where the two endpoints of C_e are p_u and p_v . Note that this graph may contain more than one connected component, and more that one edge between two edges. Informally, we blur the distinction between a vertex v and a point p_v , and refer to both as a *point*. Similarly, we refer to and edge e and a curve C_e as a *curve*. Thus, we can think of a stick figure as a network of connected points and curves.

Figure 2.16 shows a stick figure created by our system.

2.5 Basic operations

The interface for our system is similar in style to the Sketch system [47]. The user is presented with a perspective view of the scene, which initially consists of only a floor plane. We use a three button mouse with the right button for camera controls, the middle button for manipulation, and the left button for drawing operations. For camera controls, we use



Figure 2.16: A simple 3D stick figure created in our system.

the technique described in [45] because it maps a full set of World-In-Hand camera controls to a single button.

Drawing operations are invoked by depressing the left button, dragging the cursor, and releasing. The sequence of mouse positions defines a *drawing stroke*, which is a 2D polyline on the screen. We interpret a stroke as either a curve or a point. A stroke that fits entirely inside a bounding box with width and height less than 10 pixels is considered to define a point; all other strokes are considered to define curves.

When we recognize an input stroke as a point, we cast a ray from the eye-point through



Figure 2.17: Drawing a point on the existing curve splits the curve and adds a new point. After the operation, there are two separate curves. The user can then drag the center point to manipulate both curves.

the center of the input stroke. We intersect this ray with the scene to determine the location of the 3D point. For ease of picking, we treat curves as tubes with some small radius and points as spheres with a small radius (10 pixels). If the ray intersects an existing curve, we split the curve into two sections, and join the sections with the new point (see figure 2.17).

When we recognize an input stroke as a curve, we apply the system of heuristics described in section 2.3.1. At either end of a curve, we create a point if one does not already exist at that location.⁶

Curves may be manipulated in several ways. With the middle mouse button depressed, the user may click on a curve C and drag it. All curves and points in the same connected component as C are translated parallel to the film plane to maintain pick correspondence. Every curve has a corresponding shadow curve projected onto the floor plane. We treat the shadow curves as interactive shadow widgets [17], where the user can drag a shadow widget with the middle mouse button depressed, and the connected component of the corresponding curve will be translated parallel to the film plane. The user can also edit a curve using the interaction techniques described previously in this chapter, such as overdrawing, shadow editing, and strut manipulation.

Manipulating points is accomplished by clicking on a point with the middle mouse button and dragging it. If a point is clicked on, it is translated parallel to the film plane to maintain pick correspondence with the cursor, and all connected curves are adjusted so that they are geometrically connected to the point's new position. This is accomplished as follows. If a curve C has endpoints p and q, and q is moved to q', we find an affine map that maps q to q' and leaves p fixed. (We use the map that is composed of the rotation of (q-p) to (q'-p) about the axis $(q-p) \times (q'-p)$ followed by a uniform scale about p by the amount $\frac{|q'-p|}{|q-p|}$.) We then apply this map to C.

2.6 Discussion

Color Plates 2.18 - 2.20 curves created with the curve drawing techniques given in this chapter.

This system is well suited for applications that require fast specification of approximate 3D curves. Applications that require more precise curves might still benefit from this technique, because of the lack of overhead required and the simplicity of the interface. In one scenario, the user would quickly sketch an approximate curve, then refine its shape with

⁶A heuristic may cause a curve to be projected into the scene such that its endpoint coincides with an existing point.

more conventional techniques.

We mentioned previously that our technique can be extended to allow shadows on walls or nonplanar surfaces. There is also no reason to restrict the user to drawing the curve and shadow from the same point of view. The user might draw the shadow from an overhead camera position (thus specifying the shadow more accurately), then sketch the curve from an oblique viewpoint.

A limitation of this method is that it can be quite hard to judge what the shadow should look like for complex 3D curves, especially from an oblique viewpoint. We have observed that users in our lab, even those with artistic training, have considerable difficulty drawing corkscrews and other spiraling shapes. In such cases, a better solution might be to use a 3D input device such as a Phantom or a 3D tracker.

2.7 Future Work

We use context-sensitive commands to indicate over-sketching operations and keyboard modifiers to indicate modes and to differentiate between different editing operations. Although this works, it is neither consistent nor supported by user studies. We would like to find a more streamlined user interface, perhaps using marking menus or gestural commands [34]. We would also like to have more users try this system, especially users with artistic training but little experience with computer graphics tools.

A continuing project is to use this curve-sketching technique and stick figure drawing program within a sketch-based free-form modeling system. We believe this interface is a good starting point from which to build a modeling system that leverages a user's talent with pencil and paper to create more complicated shapes than was possible with the original Sketch system [47].

If this system were a stand-alone application to let an artist quickly sketch stick figures, we would want it to capture the artist's "style." An artist may put subtle features in a curve to express certain ideas. When viewing a stick figure built out of planar curves from certain viewpoints, however, the stylistic variations are lost. From the original viewpoint, a curve contains all the details that were drawn. As the camera's position approaches the plane the curve lies in, the curve appears flattens out eventually becomes a straight line, losing all detail information. Further, nonplanar curves drawn in our system must be smoothed to reduce noise, but stylistic variations often consist of high frequencies. Smoothing a curve therefore has the effect of reducing the level of expressiveness of the system. The system described in the next chapter, Harold, seeks to overcome these limitations by storing strokes



Figure 2.18: The rope in the tetherball scene was drawn from the viewpoint in (a). Figure (b) shows the same scene from a different viewpoint.

not as 3D curve, but as "billboards" that look the same from all directions. Thus a stroke maintains its stylistic appearance as the viewpoint changes.



Figure 2.19: The user has sketched a camera path through this virtual environment. The curve was created from the viewpoint in (a). Figure (b) shows the scene from a different viewpoint.



Figure 2.20: In figure (a) we have intentionally misaligned the bends in the shadow and the curve. Note in figure (b) how the shape of the 3D curve has two bends while the sketched curve and shadow each have one.

Chapter 3

Harold: sketching 3D environments

3.1 Introduction

In the children's book *Harold and the Purple Crayon* [25], a small boy, Harold, creates a world by drawing it with his purple crayon. He explores and expands this world by walking into and through it, drawing all the time. In essence, this is the ultimate virtual environment, allowing users both to experience and to create a virtual world around them. An ideal VR system would be capable of this much expressiveness and interactivity, but this problem is, at present, intractable. We therefore state a more restricted form of this problem: Given a scene drawn in 2D from a single point of view, we would like to reconstruct the scene interactively from novel viewpoints. In other words, we would like to be able to draw a scene, move around it, and have everything just "look right."

Previous research has approached this problem in two primary ways, which we categorize as *geometric* and *image-based*. In the geometric approach, the system attempts to create a geometric description of the 3D scene from the user's 2D input. This is similar to many problems in computer vision, and is essentially the inverse of traditional computer graphics – given a rendering (or often a drawing), the system tries to recreate a geometric description of the scene. New renderings can then be obtained from arbitrary viewpoints. The Sketch [47] and Teddy [24] systems have demonstrated the feasibility of this approach for interactively creating 3D objects from 2D gestures.

A limitation of Sketch and Teddy, however, is that the inferred geometry is often incorrect, and these errors become more and more apparent as the viewpoint changes significantly from that from which the object was initially created. This reflects a fundamental drawback of purely geometric approaches – not all 2D drawings can actually be generated from 3D models. Rademacher [33] discusses how this problem arises when animators attempt to create static 3D models of cartoon characters, and proposes using dynamic view-dependent geometry to address it.

Image-based approaches avoid creating a geometric description of the scene, but instead redisplay the original input image, modified to reflect new camera parameters. The system described in [42] lets the user draw on the inside of a sphere, thus allowing an immersive experience as long as the camera's position remains fixed. In Tour Into the Picture [19], the user begins with a 2D image, which may be either a photograph or hand-drawn, and then specifies geometric constraints such as the vanishing point and horizon line. The user can then view the scene from novel camera locations within certain constraints (the user cannot, for example, turn around and look behind herself).

Our approach attempts to find a middle ground. Our system, Harold, like Sketch and Teddy, creates a 3D model of the environment. However, our world is populated by drawings, not 3D objects, and thus is similar to image-based methods, particularly [42]. The primary geometric primitive in our system is a *billboard*; these are commonly used in interactive systems to render complex yet unimportant objects with low overhead. A billboard is typically a plane with an image texture-mapped onto it that rotates about some point or axis to face the viewer as much as possible. Our billboards contain collections of planar strokes rather than textures. When the user draws a stroke over a billboard, we simply project the stroke onto the billboard and store it; then, to display the billboard, we re-render each stroke, rotated appropriately (see Figure 4.1). Thus, we avoid the problem of reconstructing what the backside of a tree looks like – the tree simply has no back. A consequence of this choice is that Harold, unlike Sketch and Teddy, works with a very small set of inferences about the user's strokes: for the most part, strokes are simply projected onto a surface and nothing more.

As noted by Zeleznik et al. [47] and Igarashi et al. [24], it is important that views of the scene be rendered in a non-photorealistic style, in order better to convey the imprecise and hand-drawn nature of the underlying geometric description. We take this notion even further and allow the user to draw objects using a variety of stroke styles. With the exception of distance cueing, we render the strokes exactly as they were drawn. Objects in Harold maintain the distinct stylistic appearance and subtleties imparted by the user, and our worlds thus maintain their intended style and character as the viewpoint changes.

The primary drawback of our approach is that the relationships between objects change as the viewpoint changes. For example, imagine a fenced-in area containing farm animals. If the fences were all billboards, they would rotate through one another as the viewpoint



Figure 3.1: Creating a billboard. (a) The trunk of the tree was drawn as a stroke starting at the base. The leaves were then added to the billboard. (b) That same billboard, seen from a different view; the billboard on which the stroke was drawn rotates so as always to face the viewer as much as possible.



Figure 3.2: The hammock, which is a *bridge* anchored to the two tree billboards, was created from the viewpoint in (a); (b) shows the objects from a new viewpoint.

changes, unwittingly freeing the enclosed livestock. We thus need some way of specifying fixed relationships among objects in our world. For this purpose, we use a *bridge billboard*, which is a collection of planar strokes that is anchored to points on two billboards (see Figure 4.2). Thus, we can string a fence between separate fence posts, or a hammock between two trees. Harold also has a primitive terrain-sketching facility, with which the user can sketch a heightfield terrain by drawing the silhouettes of hills or other features.

The entire Harold system is an amalgamation of simple components – there are no algorithmic subtleties, nor any complex constraint-maintenance mechanisms. Our contribution is thus an approach to 3D scene reconstruction that combines features of other approaches as well as novel ideas, and integrates them with a particular set of interface choices to create an interactive system.

3.2 Related work

Our system has a similar aesthetic to several 2D paint programs. We like the easy-touse and appealing visual interface of Kid Pix [18], which is a paint program designed for children. In Kid Pix, all operations are easy to find and have immediately observable consequences. Because this type of interface places a low cognitive load on the user, it encourages experimentation and exploration. Our system is similar to the AltaMira and Quantel Paintbox systems ([39] and [32]), although these systems are not fully 3D, in that they employ a similar notion of creating scenes by layering images on top of one another.

Interactively constructing 3D scenes from 2D input is an active research area. The Sketch and Teddy systems ([47] and [24]), as well as the system described in the previous chapters, allow the user to create 3D objects via simple suggestive gestures. In Teddy, for example, the user draws the silhouette of an object, and the system creates a plausible 3D shape with that silhouette. Previous work by Tolba et al. [42] is very similar in spirit and technique to Harold. In their system, user's strokes are projected onto a sphere centered at the camera. We have in fact incorporated this technique into Harold to let the user to draw objects in the sky. Tour Into the Picture [19] allows the user to "enter" a 2D image by warping the image to simulate new viewing parameters.

We use the stroke-based rendering system described in [30], which is related to the skeletal strokes technique described in [21, 20]. The ground in Harold is rendered using the shader described by Gooch et al. [15], which provides shape information while preserving a brightly colored hand-drawn appearance.

3.3 The user's view of the system

We now describe how the system appears to the user. The initial view of the world shows a "ground," which is a large planar region of the xz-plane, and a "sky," which is the inside of an enormous sphere.

The interaction metaphor in Harold is *drawing*. With the exception of clicking on toolbar icons to change color, stroke style, stroke width, and drawing mode, all operations are invoked either by clicking on an object or by drawing a stroke. There are three buttons: the drawing button, the camera button, and the eraser button.¹ Figure 4.5 summarizes the gestures, key-mappings, modes, and their meanings.

¹Our implementation uses left button for drawing, right button for camera controls, and shift-left button for erasing.



Figure 3.3: Ground strokes. (a) A ground stroke crossing the silhouette of the hill on the left. (b) The same stroke seen from a different view, showing how the system has filled in the "gap" with a segment.

The user places her cursor at some point of the screen; this point corresponds to a point of the world, either on the ground or in the sky. (Henceforth, we merely say that she places her cursor "on a point in the world," glossing over the correspondence induced by tracing a ray from the viewpoint through the point on the film plane and into the world.) She now begins to draw by dragging the cursor with the drawing button depressed. The style of the resulting stroke depends on the current color, the current stroke width, and which of three rendering styles is selected from the menu.

STROKES ON THE SKY. Suppose that her starting point was in the sky. The strokes are then interpreted as "drawing on the sky" and the result is a stroke on the sky, visible whenever she looks in that direction. As mentioned above, this is an implementation of [42].

STROKES ON THE GROUND. Strokes that begin on the ground are interpreted in one of three ways, depending on the drawing mode currently selected on the toolbar: drawing on the ground, drawing billboards, and drawing terrain.

Drawing on the ground. In drawing-on-the-ground mode, a stroke that starts on the ground is treated as laying down marks on the ground itself. This is useful for creating things like train tracks, sidewalks, etc. If the user's stroke crosses over a silhouette of the terrain, the projection of the stroke onto the ground terrain will be discontinuous. We make the projected stroke continuous by bridging any gaps with curves that look from above like straight line segments (see Figure 4.3). Since our terrain is a heightfield, for every x and z coordinate, there is a unique y coordinate. We can thus fill in all gaps with a line segment on the xz-plane, and then compute the height at each point along the line segment to create a 3D stroke that lies on the ground.

Drawing billboards. In billboard-drawing mode, a stroke that starts on the ground

creates a new *billboard*, anchored at the starting point of the stroke. The billboard's plane is perpendicular to the ground, and as perpendicular as possible to the eye-to-stroke-startpoint vector. Thus the billboard plane's normal vector, when the billboard is created, is

$$n = \mathbf{y} \times (\mathbf{y} \times (\mathbf{eye} - \mathbf{base})),$$

where \mathbf{eye} is the camera location, \mathbf{base} is the location of the base of the billboard and \mathbf{y} is the vertical unit vector in the world.

After each stroke is added to a billboard, a bounding rectangle for the strokes on that billboard is created, slightly enlarged, and displayed in a semi-transparent light gray. Any subsequent stroke that starts in this highlighted area is added to this billboard, regardless of the current drawing mode. This is important in creating billboards having several strokes that do not actually touch.

The interpretation of strokes has one small subtlety: the final meaning of a stroke is determined at the end of the stroke, when the mouse is released. If the mouse-up occurs *outside* of the highlighted area, and over a *different* billboard, then the stroke is interpreted as defining a *bridge billboard* between these two billboards, or simply a "bridge."

Drawing bridge billboards. A bridge is created, as just described, by a stroke that starts on one billboard (we'll call the start point S) and ends on a stroke in another billboard (we'll call the end point E). Both S and E are taken to be points on their respective billboards, not in the world; when the billboards turn to face the camera, the world locations of S and E change as well. When a bridge is created, the system determines the plane that contains the current world-space positions of E, S, and the \mathbf{y} vector. At creation time, this bridge billboard is highlighted, and subsequent strokes drawn on it are recorded just like those on any other billboard.

When the camera is moved, the world-space locations of E and S may change. When this happens, the bridge billboard is scaled in its horizontal direction and sheared along the vertical direction to maintain the correspondence of points of the billboard with the points E and S (see Section 3.4.3). This helps to maintain the apparent "connections" between strokes in the bridge billboard and strokes in the billboards at either end.

Drawing terrain. In terrain-drawing mode, a stroke that starts and ends on the ground creates a bump in an attempt to make the stroke a silhouette of the newly deformed ground (see Figure 4.4). All objects in the world are then lifted so that they remain coincident with the ground.

OTHER GESTURES. A few other gestures can be made with the drawing button. A single click anywhere in the scene un-highlights the highlighted billboard. A single click



Figure 3.4: Editing terrain. (a) The user draws a stroke starting on the ground that indicates the shape of a hill; (b) The hill is created by warping the ground to try to match the stroke.

on a billboard stroke highlights the billboard that contains that stroke. Colors can be dragand-dropped from the color palette onto the sky, the ground, or any stroke, to change their colors accordingly.

ERASER GESTURES. Clicking on a stroke with the eraser button removes it. Scribbling on a billboard with the eraser button depressed removes the entire billboard.

CAMERA CONTROL. We wanted a driving-style interface for controlling the camera location in Harold. We rejected World-In-Hand controls because we wanted to give the user a sense of the size of the world around her. We wanted a technique that would blend aesthetically with the stroke drawing nature of Harold. Igarashi's technique [22] was appropriate because of its path-drawing interface; however, we adapted this technique so that the user could explicitly specify a point in the scene at which the camera would look, and we set the camera always to be two units above the ground (so that one unit is approximately equal to one meter).

To move the camera, the user draws a stroke on the ground with the camera button depressed. This path is displayed as a red line, and the gaps in the stroke that the user can't see are filled in as described above. The user then clicks on a point on the world. The camera moves along the drawn path at a constant speed (5 meters per second) and ends up looking at the point where the user clicked. A click on the sky while the camera is traveling along a path cancels the current path.

To swivel and tilt the camera, the user clicks on the sky with the camera button and "drags" the sky side-to-side to swivel, or up and down to tilt. We constrain the camera only to tilt 30 degrees up or down from the horizontal plane.

We also provide Doom-style camera controls [40] via the numeric keypad, so the user can optionally fly through the world. This particular choice of controls breaks the drawing metaphor, but is a simple way to give the user more freedom of movement in the virtual world.

3.4 Implementation details

3.4.1 Storing collections of strokes

All strokes are stored with an associated color, style, and width. The width can be in either pixels or world-space units. Strokes on billboard and bridges are stored in world-space width, while strokes in the sky and on the ground are stored in pixel width. If the width is in world-space units, we first compute the corresponding width in pixels that an object would have at that distance from the camera and then render the stroke. Thus strokes change their widths to indicate distance from the camera. We clamp all pixel widths so strokes are at least three pixels wide.

Since our toolbar indicates stroke widths by circles of different pixel radii, one issue that arises when storing strokes with world-space width is what does it mean to select a particular stroke width. We decided that selecting a stroke width in pixels should indicate the width at which a particular stroke would be rendered if it were at the same distance from the camera as the filmplane. Thus, when the user has selected a particular stroke width, strokes drawn with that width may have different pixel widths depending on their distances from the camera.

The collection of strokes associated with a billboard or a bridge is represented as a list of polylines in the coordinate space of the billboard, which is a copy of \Re^2 . The coordinates are then mapped into the world via the transformations described in the following two sections.

3.4.2 Billboard transformations

Whenever the camera moves, each billboard must transform itself to face the camera. While a number of possible transformations would achieve this affect, we chose a relatively simple transformation in which each billboard rotates around a single point fixed at the base of the billboard. To minimize the artifacts that occur when rotating a billboard, we determine the point of rotation by finding where the billboard's strokes touch the ground. This rotation point is updated every time a stroke is added to or removed from the billboard.

Since some billboards have more than one point where a stroke touches the ground and others have none, our algorithm for choosing a rotation point searches the collection of stroke polylines on the billboard for all points with a locally lowest y-value. We next find



Figure 3.5: The interface components: red arcs are strokes, red dots are clicks.

all such minima that are within a small distance of the ground (we use 0.5 meters), and of these determine L and R as the leftmost and rightmost (i.e., the points with the highest and lowest x-value in the coordinate space of the billboard). We choose the midpoint of Land R for the center of rotation. When no strokes have points near the ground, we simply choose as the center of rotation the lowest point over all strokes on the billboard.

Each billboard is represented internally as a copy of \Re^2 together with a basepoint b in \Re^3 ; for any viewpoint v, we build a rigid transformation that maps the origin of \Re^2 to b and maps \Re^2 to a plane through b that contains \mathbf{y} and the vector $(v - b) \times \mathbf{y}$. If we use coordinates $[x, y, 1]^t$ for points of \Re^2 and coordinates $[x, y, z, 1]^t$ for points of \Re^3 , then our transformation is represented by the 4×3 matrix

$$M = \begin{bmatrix} \gamma & 0 & b_x \\ 0 & 1 & b_y \\ \zeta & 0 & b_z \\ 0 & 0 & 1 \end{bmatrix}$$

where

$$\gamma = (v-b)_x / \sqrt{(v-b)_x^2 + (v-b)_z^2}$$

$$\zeta = -(v-b)_z / \sqrt{(v-b)_x^2 + (v-b)_z^2}$$

which maps the origin to b, the unit **y**-vector in \Re^2 to the unit **y**-vector in \Re^3 , and the unit **x**-vector to the horizontal unit vector in \Re^3 that is orthogonal to v - b.

3.4.3 Bridge transformations

A bridge is created with a pair of points S and E, each on a separate billboard. Let \hat{S} and \hat{E} denote the world-space locations of those points at the time the billboard is created. Let q denote the vector $\hat{E} - \hat{S}$ and $m = [q_X, 0, q_z, 0]$ denote its projection to the xz plane. Then we build a map from \Re^2 to \Re^3 defined by the transformation

$$N = \begin{bmatrix} \phi & 0 & \hat{S}_x \\ 0 & 1 & \hat{S}_y \\ \chi & 0 & \hat{S}_z \\ 0 & 0 & 1 \end{bmatrix}$$

where

$$\phi = q_x/||m||$$
 and $\chi = q_z/||m||$

which sends the origin $\mathbf{0}$ to \hat{S} , the unit **y**-vector of \Re^2 to the unit **y**-vector of \Re^3 , and the unit **x**-vector in \Re^2 to the unit vector in the direction m. The preimage of E under this transformation is some point e of \Re^2 . Since the preimage of S is the origin and the preimage of E is e, the preimage of q = S - E is $e - \mathbf{0}$; thus $N(e - \mathbf{0}) = q$.

To store a stroke made on a bridge billboard, we project each vertex onto the bridge's plane to get a point in \Re^3 . This points is then transformed by N^{-1} and recorded in \Re^2 ; when the bridge is to be redisplayed from some other view in which the world-space locations of E and S are now \bar{E} and \bar{S} , we build a new transformation \bar{N} that maps the origin to \bar{S} , the **y**-axis of \Re^2 to the **y**-axis of \Re^3 , and the point e to the point \bar{E} (or equivalently, maps the vector $e - \mathbf{0}$ to the vector $\bar{q} = \bar{S} - \bar{E}$. Letting $\bar{m} = [\bar{q}_x, 0, \bar{q}_z, 0]^t$ be the projection of the vector \bar{q} to the xz-plane, the matrix for this transformation is

$$\bar{N} = \begin{bmatrix} \bar{\phi} & 0 & \bar{S}_x \\ 0 & 1 & \bar{S}_y \\ \bar{\chi} & 0 & \bar{S}_z \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ \frac{\bar{q}_y}{q_y ||\bar{m}||} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{||\bar{m}||}{||m||} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where

$$\overline{\phi} = q_x/||m||$$
 and $\overline{\chi} = q_z/||m||$.

Reading right to left, the first matrix scales the domain in x so that after being transformed, the vertical line that contains e will map to the vertical line containing \overline{E} . The second matrix is a shear such that the vertical line containing the origin is unchanged and the vertical line containing e is raised so that e maps to \overline{E} . The third matrix is simply a rigid transformation like the one built for N above.

3.4.4 Rendering strokes

As described above, each stroke is defined by a sequence of points (a polyline) and a pixel width that is either stored or computed. This sequence of points is then rendered as a stroke using the method of Northrup and Markosian [30]. The three stroke styles we have implemented provide a fair range of expressiveness while maintaining interactive frame rates. Examples of these stroke types are shown in Figure 4.6.

For the marker style, we use a stroke with constant width and no mitering at the endpoints. For the ink style, we use a stroke with mitered endpoints and a width that tapers to 0. Since the width is tapered over the length of the stroke, the stroke gets wider near its beginning as its length increases. In our interactive system, this gives the effect of ink "bleeding" and spreading outward as the user continues to draw a stroke.



Figure 3.6: The top strokes are marker style, the middle strokes are ink style, and the bottom strokes are watercolor style.

For watercolor strokes, the stroke is drawn with increasing transparency along its length. We linearly fade out the transparency to 0.5 along the first 20 vertices of the stroke, and draw all subsequent vertices with 0.5 transparency. This implementation captures only a tiny fraction of true watercolor behavior (as modeled by Curtis et al. [12]), but is efficient enough for rendering in our real-time system.

Strokes are rendered by first building triangle strips, and then passing them to the rendering pipeline using OpenGL [7]. Since strokes are view-dependent, we cannot just cache them in display lists; they must be rebuilt for each frame. However, the user typically maintains a fixed camera position while drawing an object. To get a higher sampling rate for the input device in this case and improve the interactivity of the system, we we can take advantage of display lists and cache the triangle strips that have already been built.

3.4.5 Terrain

Terrain-editing strokes must start and end on the ground. Call the starting and ending points S and E. Just as in the bridge-definition rules in sections 3.4.3, these two points, together with the **y**-vector, determine a plane in \Re^3 , that we call the *projection plane*. The points of the terrain-editing stroke are projected onto this plane (this projection, which is a curve in \Re^3 , is called the *silhouette curve*); the shadow of the resulting curve (as cast by a sun directly overhead) is a path on the ground (we call this the *shadow*). Points near the shadow have their elevation altered by a rule: each point P near the shadow computes its new height (y-value) as a convex combination

$$(1 - w(d)) \cdot P_u + w(d) \cdot h$$

where d is the distance from P to the projection plane, h is the y-value of the silhouette curve over the nearest point on the projection plane to P, and w(d) is a weighting function given by

$$w(d) = \max\left(0, 1 - \left(\frac{d}{5}\right)^2\right).$$

This gives a parabolic cross-section of width 10 for a curve drawn over level terrain. Other choices for w would yield hills with different shapes that might be more intuitive, but this particular choice gives reasonable results in most cases.

Note that if the silhouette curve bends back on itself (i.e. it defines a silhouette that cannot be modeled using a heightfield), then the variation of height along the shadow will be discontinuous. The resulting terrain then may have unexpected features.

3.5 Limitations and discussion

Figures 4.7 and 4.8 show two scenes created using Harold. Both scenes took under 15 minutes to draw. Figures 4.9 and 4.10 show how Harold can be used for conceptual prototyping of outdoor scenes. Figure 4.11 shows a simple moon scene.

The major problem that arises in trying to reconstruct a 3D scene from 2D input is trying to determine what an object looks like from a new point of view. One can think of billboards as providing a crude solution to this problem – we approximate the appearance of an object from a new viewpoint as its appearance from the initial viewpoint. Thus billboards work well for objects that look approximately the same from all directions, e.g., trees, flowers, telephone poles, and even roughly drawn characters. When this assumption of approximate radial symmetry does not hold, as in the case of a house, say, billboards look odd and visually disturbing.

A different approach may have more success in cases like this. The advantage of a system like Teddy, for example, is that for certain types of objects, Teddy can produce fairly accurate approximations of the objects' appearances from novel viewpoints. Another strategy is to constrain the camera so that the user views objects only from viewpoints from which the system can plausibly reconstruct an object's appearance. This is the approach taken in Tolba, Dorsey, and McMillan's system [42], and one we have incorporated to a certain extent in Harold. This is why in Harold the camera is constrained to stay at a constant height over the ground (with the exception of the optional flying controls); we have made the system's task slightly easier since we don't have to reconstruct the appearances of objects from above or below.

Another drawback of billboards is that they do not maintain a fixed relationship with one another as the camera moves. For example, when two large billboards are close together, they may intersect each other as the viewer moves and the billboards rotate. This can produce surprising visual effects. Our algorithm for choosing the center of rotation partially addresses this problem, as does the use of bridges, but there are some cases that neither approach handles particularly well.

Also, the interface in Harold is more obviously modal than systems like Sketch or Teddy. It is not clear whether this is a drawback or not. The notion that modes are "suggestive" rather than rigid may be somewhat awkward to some users as well: one can be in billboard mode and still perform operations that are not related to billboards, such as drawing on the sky.

3.6 Future work

An obvious direction for future work is to incorporate more techniques from systems such as Sketch, Teddy, or our 3D stick figure system. It is not clear how to do this in many cases, and it is clearly a nontrivial task. However, such a system could potentially be a very powerful modeling tool for creating visually rich 3D environments.

Although we have three types of brushes, it is clear that a richer mechanism for creating drawings would be useful. All of the features of ordinary paint programs could conceivably be incorporated, although certain issues arise in the context of our system that do not arise in 2D paint programs. For example, flood-filling a collection of strokes on a billboard is not as simple as flood-filling a region of a 2D image because there may be gaps between strokes that are not visible if the billboard is sufficiently far away. A naive flood fill would then unexpectedly flow outside of the intended area. These are interesting problems to investigate.

We have also experimented with view-dependent billboards. Instead of representing an object with a single billboard, we represent a object with multiple billboards with the same axis of rotation, where only one of these billboards is visible at a time. Associated with the i^{th} billboard is a vector b_i describing the "optimal" direction for viewing that billboard. From a camera position v, we find a vector from the billboard's axis of rotation to v. Call this vector b. We then choose billboard i according to

$$\operatorname*{argmax}_{i}(b_i \cdot b)$$

In other words, we display the billboard best suited for facing the current camera position. Once the billboard is selected, it is rotated about its axis to best face the camera as with regular billboards.

While this approach allows the user to create objects that don't look the same from all directions (e.g., the back of someone's head could have hair, while the front could have a face), it is not clear how to integrate view-dependent billboards into the current interface. For example, how does the user specify the "optimal" direction for viewing a billboard? The user might want some features to appear from certain views, and not from others, or from all views all the time. These complexities quickly bog down the simplicity of Harold's drawing interface.

Other directions we are exploring include extending the drawing metaphor to allow the user to "draw" animations, such as a river that appears to be flowing or rain that appears to be falling. It is conceivable that somehow we could take this even further and allow the



Figure 3.7: This scene was created in Harold. The mountains are painted on the sky, the fences are bridges strung between billboards, and the hut in the lower center is a billboard.

user to "draw" simple behaviors, thereby creating interactive 3D environments.

Finally, there are many possible extensions to the rendering system, such as adding automatic level-of-detail for strokes that are far away, improvements to the overall efficiency of the system, and incorporating other non-photorealistic rendering styles such as those described in [27].



Figure 3.8: This scene was created by an art student after using Harold for approximately 1 hour.



Figure 3.9: Here Harold was used to create an initial conceptual sketch of an outdoor scene. The scene was created by an architectural student within an hour of first using Harold.



Figure 3.10: The same scene from a different viewpoint.



Figure 3.11: A moon scene created in Harold. The bright colors and different stroke styles enable Harold to capture the liveliness of a child's drawing.

Chapter 4

Sketch-based interfaces

In general, the goals of a user interface include precision, speed, ease of learning, flexibility, expressiveness, and naturalness. While these are all important factors, we observe that certain tradeoffs are necessary. In the interfaces described in this thesis, we have traded flexibility and precision for naturalness, ease of learning, and speed. In this section, we would like to examine these tradeoffs in an attempt to explain why they are necessary, and to understand the dynamics of user-interface design choices. While this discussion may apply to interfaces for specifying 2D information as well, we will focus on sketching to specify 3D information.

We begin by defining what we mean by a *sketch-based 3D interface*. A sketch-based interface allows a user to specify 3D objects, scenes, or other information by directly specifying appearance and shape information via an input device. The goal of a sketch-based interface typically is not the creation of a detailed geometric model; rather it is often communication and rapid exploration of design possibilities.

4.1 Interface components

Figure 4.1 is a diagram showing the different elements of a sketch-based interface system. The top layer is the *Device Layer*. This consists of the actual input device or devices that the user interacts with. The second layer, the *Feedback Layer*, consists of how the system provides immediate feedback to the user's input, in the form of a cursor, haptic feedback, or other techniques. The *Gesture Set Layer* determines of how the data stream is segmented into different commands. This determines what the user must learn in order to use the system. The *Inferencing Layer* consists of how the system interprets input gestures to extract the parameters the system needs to execute a command. The bottom layer, the



Figure 4.1: The components of a sketch-based interface.

Model Layer, consists of the kinds of models the system can create and manipulate, where a "model" may include geometric, appearance-based, or dynamic descriptions of an object or an action.

For a particular modeling system, we can identify the space containing all models that the system is capable of producing. Our analysis of a sketch-based interface must take the structure of this space into account. Informally, we seek to describe the structure of the model-space, specifically by describing how the space can be parameterized, and by calculating its dimensionality.¹ If we think of a particular model as a point in this modelspace, then the modeling task to specify a point in model-space. We can think of an interface as a process that takes data obtained from input devices and determines values for the parameters of the model-space. This is shown schematically in figure 4.2.

Decisions made in designing one level of the interface affect the functioning of the other layers. Thus to design or analyze an interface, we cannot just look at each component separately; rather, we must take into account the overall performance of our interface from a functional point of view, as well as from a human-centered point of view. From the functional perspective, we view the interface as a function that maps from an input stream to a model-space. From the human-centered perspective, we consider the user experience, in which a user interacts with the interface to explore the model-space.

4.1.1 Input device layer

From the functional point of view, it is less ambiguous to map high-degree-of-freedom (DOF) input to a low-dimensional model space than to map low-DOF input to a high-dimensional

¹We use the term "dimension" here loosely, since model spaces often are not actually vector spaces.



Figure 4.2: This schematic diagram shows the components of the interface process. Raw data is segmented into gestures, while the system provides feedback to the user. The inferencing layer then maps these gestures to parameters in the model space, specifying a particular model.

model space. Mapping from high-to-low involves removing redundant information, while mapping from low-to-high involves somehow extracting extra information.

Using high-DOF input devices therefore has the potential to reduce the complexity of other parts of the system, because less inferencing may be required. High-DOF input can be obtained from 6-DOF trackers or 3D positional devices, as well as by aggregating data from several different input devices. This is the idea behind multi-modal interfaces in which the user interacts with the system simultaneously via several different channels.

In choosing an input device, it is also important to consider human factors such as accuracy, fatigue, how "natural" it feels, ease of learning, coordination, and suitability to a particular task. The input device is the physical component that the user interacts with, and from the user's point of view, is of primary importance. Thus, the system designer must choose input devices that have high enough DOF, while at the same time enabling a pleasant user experience.

One approach for choosing an input device is to pick a device with the same number of DOF as the model space. Balakrishnan et al. [3] describe a system for using a flexible metal strip as an input device to create 3D NURBS curves. In their system, the user poses the metal strip, and the curve takes on the strip's shape. Because there is a direct mapping between input parameters and model-space parameters, the inferencing layer is very simple. However, such as simple mapping fails to exploit any of the benefits of extrapolation – the system adds nothing to the user's input.

This is also the motivation behind the 3-Draw system [35], for example, where the user traces out paths with her hands to create free-form curves. Since a curve is defined by a

time-varying position in \Re^3 , the input is also a time-varying position in \Re^3 . Again, the inferencing layer is very simple – it involves an affine map applied to the input stream. 3D input devices are far from perfect, however. They suffer from lack of haptic feedback, fatigue, lack of precision, and the engineering difficulties in obtaining accurate positional data. Zhai's article [48] provides an overview of current 3D input device technology.

Multi-modal interfaces provide a richer data stream, but are rarely feasible for deployment on a large scale because few of the required input devices are commercially viable yet. Possible multi-modal interfaces include combining speech input with a 3D cursor to allow for a "put that there"-style interface, or combining 3D tracker with hand pose information. In Schkolne's work [36], for example, the user wears a CyberGlove with a 3D tracker attached to it. The user "draws" a surface by sweeping her hands through space. Operations such as changing color or erasing are distinguished by hand position. Two fingers extending means draw, while one finger extended means erase. LaViola [26] provides a comprehensive overview of multi-modal interaction in virtual environments.

The prevalence of 2D input devices such as the mouse or the tablet, along with people's natural abilities to draw in 2D with pencil and paper, makes them a natural choice for many systems. There is a lot of work on "natural" drawing-style interfaces using 2D input devices, including the previous chapters of this thesis. A technique for increasing the dimensionality of the input while using 2D devices is to use two of them, thereby creating a multi-modal interface. Zeleznik et al. [46] describe interaction techniques that use a stylus and a puck on a large WACOM tablet to perform camera manipulation and geometric modeling operations.

4.1.2 Feedback layer

While feedback is important for the user experience, it is also important for the effective functioning of the system. If we consider that the modeling task is not just to specify *a* model in the model-space, but to specify the *desired* model, then accuracy is important, and feedback affects accuracy.

Feedback is closely related to the nature of the input device. For example, a consideration when choosing a 3D input device for an interface is whether it provides haptic feedback. Without this, a user has difficulty attaining speed and accuracy. Haptics are a relatively new research area, and while few general principles have emerged yet, this may prove to be a fruitful direction to pursue. Miller and Zeleznik [29] suggest principles that may be used in the design of 3D haptic interfaces.

Co-location of the input device with the display also enhances the user experience

while improving accuracy. As noted in [44], in many situations performance advantages can be gained by providing immediate visual feedback co-located with the input device. Schkolne [36] postulates a theory of what he calls *perceptual thinking*, which states that sensory and proprioceptive perception are integral parts of the cognition required to perform a task. This theory suggests that a smaller distance between an action and its perceived consequences results in better performance.

Feedback is related to the gesture set and inferencing layers. Feedback can be used to constrain the user to give only input that conforms to the gesture set. If the user provides invalid input, the system can correct the input and provide feedback to indicate how the input has been corrected. An interface also may provide feedback to the user regarding how the inferencing layer will interpret the input, such as by previewing the resulting model.

One such type of feedback is to provide geometric constraint information. In the Sketch system [47], for example, the gesture to create a cuboid involves drawing three lines on the screen aligned with screen-space projections of orthogonal axes of a fixed coordinate system. To aid the user in drawing these lines, the system "snaps" each input stroke to the closest axis by projecting it onto that axis. This indicates how that stroke will be interpreted by the inferencing layer. Further, the different axes are drawn with different colors to help the user immediately know which axis the system thinks she is drawing.

4.1.3 Gesture set layer

The gesture set layer determines how the input stream is broken up into separate input commands. The gesture set determines what the user must learn in order to interact with the interface. Therefore design choices made in this layer are judged by the cognitive load they place on a trained or novice user. For a trained user, a well-designed gesture set should "disappear" so that the user barely notices it at all. Thus, from the user's point of view, we want a gesture set that is idiomatic, intuitive, and consistent.

From the functional point of view, we want a gesture set that will make the inferencing layer's job easier. Thus the gesture set should be as unambiguous as possible, while also providing data in a format that is understandable. There are several techniques for disambiguating different gestures. We group them according to their qualitative effects on the user experience. The techniques roughly fall into modal, chordal, grammar-based, contextual, and probabilistic categories.

Modality involves having the user enter a mode, and then interpreting the input stream based on the current mode. Using a modal gesture set requires temporal reasoning on the part of the user. In a chordal gesture set, the occurrence of several input events simultaneously indicates different input gestures. A chordal gesture set is only possible when the input devices are capable of generating multiple simultaneous input events. A grammarbased gesture set requires that the incoming data be segmented into tokens, which are then parsed according to a grammar. Parsing a stream of tokens is similar to the modal technique, except that the modes in grammar-based interfaces tend to "disappear," and the user thinks in terms of multiple sequences of commands at a time. In a contextual gesture set, gestures are disambiguated depending on where they occur. Contextual gesture sets are similar to a modal gesture sets, except that they requires spatial, rather than temporal, planning. A probabilistic gesture set involves some type of machine learning algorithm to "guess" the user's intention. The goal with probabilistic gesture sets is to enable to system to respond to less structured input.

Consider modeling systems that apply 3D model reconstruction algorithms directly to a scanned image of a real pencil sketch, as surveyed by Wang and Grinstein [43]. The user draws with a pencil then indicates that the system should apply an inferencing algorithm to produce a 3D model. The gesture set the user must learn is virtually nonexistent.

The Sketch system is similar to these systems in that the input device is a stylus and the resulting models are 3D scenes, but its gesture set and inferencing engine are quite different. The design goal was to "remind" the user of pencil sketching as much as possible. Therefore the gesture set was designed to mimic the feel of a pencil sketch. Sketch uses a grammar-based approach to distinguish between gestures, but after sufficient practice the sequences of gestures merge together and feel like a single command. The bulk of the effort in designing Sketch came from designing a gesture set that would facilitate this learning process while being unambiguous.

4.1.4 Inferencing layer

The inferencing layer is closely related to the modeling layer. Inferencing involves taking recognized gestures and converting them to models. If we think of the model-space as somehow parameterized, then inferencing is the process of extracting values for model parameters from the input stream. A typical sketch-based interface must determine values for a high number of parameters from a lower-DOF input stream. The problem then becomes a problem of interpretation. How does the interface extract the desired information from this data stream accurately, quickly, and robustly?

For sketch-based interfaces, inferencing algorithms often involve projection or constraint

solving. Inferencing can be complex, as in the case of the algorithm in chapter 3, or Igarashi et al.'s algorithm for generating a polyhedral model from a silhouette curve [24]. Because it can be so complex, inferencing may be the least accurate and robust component of an interface. Therefore, interfaces in which most of the work is done in the inferencing layer tend to be less precise and more sensitive to bad input from the user.

Inferencing may also be context sensitive. That is, the inferencing layer may use information from models that have already been created to help it interpret the input data. Thus there can be feedback from models that have been created back to the inferencing layer. For example, to interpret a terrain stroke in Harold, the system intersects rays through the first and last points of the input stroke with the heightfield terrain. The results of these intersections depend on how the terrain has previously been edited. Thus there is feedback from the terrain model back to the terrain-stroke inferencing algorithm.

4.1.5 Model layer

As the number of different parameter values to be extracted from a single data stream increases, the modeling task becomes more difficult because either the user must provide more information, the inferencing layer must do more work, or both. Conversely, if the model-space has fewer DOF relative to the input stream, then a less ambiguous interpretation of user input is possible, resulting in greater precision and shorter time required to perform a modeling task. Thus there is a tradeoff between system's accuracy against the complexity of the model-space. It is harder to specify a model precisely in a complex space that in a simple space.

At the same time, since the reason for a sketch-based system is to enable the user to explore the model-space, it is important that the model-space contains interesting enough models that it is worth exploring. Therefore a goal in designing a sketch-based interface is to reduce the number of parameters in the model space while retaining a model space that contains interesting and expressive models. This is why it is easier to design a sketch-based interface that is domain-specific than it is to design a generic sketch-based interface. From within a given domain, the class of models tends to have a more coherent structure that the interface can exploit.

Consider drawing strokes on a billboard in Harold. A stroke is determined by the billboard it belongs to, its drawing style, and its path, which is a sequence of points lying in the billboard's plane. Since our models – in this case strokes lying on billboards – have a high degree of structure, the user can easily draw precisely what she intends. Also, the



Figure 4.3: The skeleton is shown on the left. The torso model on the right was edited at three levels of resolution.

inferencing step is relatively simple. A single click gesture specifies which billboard is to be drawn on, and toggle buttons on the side of the screen select the drawing style. Once these parameters are determined, we can simply project the 2D input stroke onto the billboard plane to generate a stroke path.

A generalized stick figure drawing system is much harder to design since it is difficult reduce a stick figure model to a small number of parameters. If we limit the type of stick figure to be drawn, however, we can perform this reduction and more successfully map user input to a small number of parameters. This is precisely what heuristics do, and why a given set of heuristics is only successful for a small class of stick figures.

A hierarchical approach seems to be a promising technique for expanding the class of models that may be created with a sketch-based system. This approach works well for specifying models in a space where the parameters are hierarchical – i.e., "less important" parameters depend on "more important" parameters. By requiring the important parameters to be specified first, the system can quickly narrow down the possible locations of the model within the model space. In some cases, the system can fill in default values for the dependent parameters and suggest an approximate model for the user. The user then specifies the dependent parameters to refine the model further. This process may be repeated until all parameters are specified.

In previous work, we introduced Skin [28], a free-form modeling technique. An example of a model created with Skin is shown in figure 4.3. Skin allows a user to organically "grow" a subdivision surface over an underlying skeleton. To create a surface using Skin, the user first specifies underlying skeletal shapes, which are created using the Sketch system. Skin is then grown over these skeletons by specifying topological information, distance from the skeleton, and other properties. The user can then subdivide the skin mesh and add more skeletal meshes at higher levels of resolution. With Skin, users can quickly specify complex models with structure at multiple different scales.

We can informally identify the parameters needed to specify a Skin model. Roughly, they are a Sketch-generated skeletal model and information about how Skin should grow over the skeletons. The information that determines skin growth, while complex, depends partially on the makeup of the underlying skeleton. Thus there is a natural hierarchy between the parameters that makes it easier to specify skin growth.

Furthermore, Skin uses a multi-resolution approach. At each level of resolution, the parameters that specify detail at that level depend on detail information from the previous level. Again, there is a naturally hierarchical relationship between parameters at different levels of resolution. By exploiting this structure, the Skin interface allows users to create a huge variety of models that have otherwise eluded creation via a sketch-based interface.

4.2 Case study: sketching a motion path

We can use this analysis to guide the design of a sketch-based interface. Our general approach will be to consider the desired model-space, the desired user experience, and then to design each layer to attain both goals.

A full design methodology for sketch-based interfaces is beyond the scope of this thesis, but we can provide a general outline for such a methodology. First, identify the space of models to be created. Second, decide on a parameterization for this space that minimizes the number of degrees of freedom and exploits hierarchical relationships if possible. Third, decide on the input devices, user feedback, and gesture set that best suit the desired user experience while maximizing the quality of the input data. Fourth, decide how to map input to each of the model-space parameters in an intuitive and direct way. This will reveal weaknesses with the original parameterization, or may point to additional parameters that need to be specified. This is an iterative process, requiring multiple repetitions. Ideally, user feedback should be incorporated into the evaluation of each of these steps.

As an example, consider the following problem: create a sketch-based interface to specify a motion path for a cartoon frog hopping over the ground. By motion path, we mean a 3D curve. Therefore, one approach would be to consider the motion path as a generic 3D curve, and use the 3D curve system described in chapter 3 to specify it. However, by recognizing that these motion paths have more structure than generic curves, we can do better.

A motion path consists of several "jumps," each of which is an arc that lies in a vertical



Figure 4.4: The frog motion path system. The motion path has been extended on the right by adding three more jumps.

plane. Two jumps are joined by a "landing" where the frog touches the ground and changes direction abruptly. So our space of possible motion paths is the space of continuous curves made up of arcs lying in vertical planes. Further, the curve must touch the ground where two arcs meet, and at the curve's endpoints. We can further assume that the curve should remain above the ground at all other times. We have now reduced the number of parameters needed to specify a frog path: A vertical plane can be specified by two non-vertically aligned points that it contains, and an arc can be specified by the plane that contains it and a 2D curve lying in that plane.

Now that we have proposed a model-space, we need to decide our intended user experience. The application is to be run on a desktop computer, so the obvious choice for input device is the mouse. We would like the user to just "draw" a motion path from a particular point of view directly into the scene by pressing a button and drawing with the mouse. It is reasonable for the user to draw a motion path one jump segment at a time because this matches a user's cognitive notion of a motion path containing several jumps, so our gesture set will be a sequence of mouse strokes with the drawing button depressed. We could also add a separate scribble gesture for erasing jump segments.

The inferencing layer will map from these mouse strokes to motion paths. Given a stroke on the screen, the system casts rays through the first and last point of the stroke and intersects these rays with the scene to obtain two points in \Re^3 . These two points define a vertical plane onto which the stroke is then projected. If any part of the projected stroke lies below the ground, we abort and indicate that the jump was invalid. At the landings on either side of a jump, we display a small red sphere. If a jump begins or ends inside

the sphere of another jump's landing, we join the two jumps together. This interface lets the user quickly build up a motion path from multiple jumps by exploiting the structure inherent in the space of desired motion paths. Figure 4.4 shows our system in action. Although we have not tested the system to any real extent, it is intuitive and easy to learn, while generally producing the desired results.

Our design decisions in each layer affect the functioning of the other layers. For example, if we change our model-space to require that jumps be parabolic arcs, our model-space would have fewer DOF, since a parabola is defined by three points. As a result, our inference layer would have to be redesigned. We could consider changing our interface further to require only three mouse clicks, or maybe two mouse clicks for the end point followed by dragging the peak of the parabola up and down, or some other technique.

While this small example may be simple, it illustrates the benefits of thinking about sketch-based user interface design in this way. By analyzing our model-space, we were able to determine what structure we could impose on it, and then exploit this structure to create an effective 3D sketch-based interface.

Chapter 5

Conclusion

In this thesis, we have defined 3D sketch-based interfaces, examined the particular problems that arise when trying to develop them, and presented several algorithms, interaction techniques, and systems to support 3D sketching. Our goal was to demonstrate that although sketch-based interfaces present certain difficulties, they also are a feasible and often desirable approach to problems that arise in computer graphics.

In chapter 2, we considered the problem of drawing 3D curves and stick figures in a way that exploits people's natural abilities to draw with pencil and paper. We presented new interaction techniques for defining and editing curves, as well as the underlying algorithms to support these interactions. We also showed how to extend these techniques to build a system for sketching 3D stick figures.

In chapter 3, we described Harold, a system for quickly sketching 3D scenes with a hand drawn appearance. Harold is successful because for certain types of scenes, billboards are a surprisingly effective object representation. Billboards also have numerous advantages that make them ideal for this type of application, such as that they maintain their intended appearance as the viewing conditions change.

Finally, in chapter 4, we presented a framework for analyzing and designing 3D sketchbased interfaces. Our analysis identifies the different components of a sketch-based interface, and focuses on how design choices at each component affect the overall functioning of the system and the user's experience. We demonstrated how this analysis could be used to design a simple interface for sketching motion paths.

Bibliography

- [1] Alias / Wavefront. Maya, 1.0 edition, 1998.
- [2] Autodesk. 3D Studio MAX, 1996.
- [3] Ravin Balakrishnan, George Fitzmaurice, Gordon Kurtenbach, and Karan Singh. Exploring Interactive Curve and Surface Manipulation Using a Bend and Twist Sensitive Input Strip. In Computer Graphics (1999 Symposium on Interactive 3D Graphics), April 1999.
- [4] Michael J. Banks and Elaine Cohen. Realtime Spline Curves from Interactively Sketched Data. In Computer Graphics (1990 Symposium on Interactive 3D Graphics), pages 99-107, 1990.
- [5] Thomas Baudel. A Mark-Based Interaction Paradigm for Free-Hand Drawing. In Proceedings of UIST 94, pages 185–192. ACM SIGGRAPH, 1994.
- [6] Jules Bloomenthal and Brian Wyvill. Interactive Techniques for Implicit Modeling. In Computer Graphics (1990 Symposium on Interactive 3D Graphics), pages 109–116, 1990.
- [7] "OpenGL Architecture Review Board". OpenGL Reference Manual, 2nd Edition. Addison-Wesley Developer's Press, 1996.
- [8] Armin Bruderlin and Lance Williams. Motion signal processing. In SIGGRAPH 95 Conference Proceedings, pages 97–104. ACM SIGGRAPH, August 1995.
- [9] Jonathan M. Cohen, John F. Hughes, and Robert C. Zeleznik. Harold: A World Made of Drawings. In Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment, June 2000. To be held in Annecy, France.

- [10] Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, John F. Hughes, and Ronen Barzel. An Interface for Sketching 3D Curves. In 1999 Symposium on Interactive 3D Graphics, pages 17–21. ACM SIGGRAPH, April 1999.
- [11] M. F. Cohen. Interactive Spacetime Control for Animation. In SIGGRAPH 92 Conference Proceedings, pages 293–302. ACM SIGGRAPH, July 1992.
- [12] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-Generated Watercolor. In SIGGRAPH 97 Conference Proceedings, pages 421–430. ACM SIGGRAPH, August 1997.
- [13] Gerald Farin. Curves and Surfaces for Computer Aided Geometric Design. Academic Press, third edition, 1993.
- [14] Barry M. Fowler and Richard H. Bartels. Constraint-Based Curve Manipulation. IEEE Computer Graphics and Applications, pages 43–49, September 1993.
- [15] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In Michael Cohen, editor, SIG-GRAPH 98 Conference Proceedings, pages 447–452. ACM SIGGRAPH, July 1998.
- [16] Cindy Grimm and Matthew Ayers. A Framework for Synchronized Editing of Multiple Curve Representations. In EUROGRAPHICS '98, pages C-31 - C-40, 1998.
- [17] Kenneth P. Herndon, Robert C. Zeleznik, Daniel C. Robbins, D. Brookshire Conner, S. Scott Snibbe, and Andries van Dam. Interactive Shadows. In *Proceedings of UIST* 92, pages 1–6. ACM SIGGRAPH, November 1992.
- [18] Craig Hickman. Kid Pix: The Early Years. http://pixelpoppin.com/kidpix/KPHistory/.
- [19] Youichi Horry, Ken ichi Anjyo, and Kiyoshi Arai. Tour Into the Picture: Using a Spidery Mesh Interface to Make animation from a Single Image. In SIGGRAPH 97 Conference Proceedings, pages 225–232. ACM SIGGRAPH, August 1997.
- [20] Siu Chi Hsu and Irene H. Lee. Drawing and Animation Using Skeletal Strokes. In SIGGRAPH 94 Conference Proceedings, pages 109–118. ACM SIGGRAPH, July 1994.
- [21] Siu Chi Hsu, Irene H. Lee, and N. E. Wisema. Skeletal Strokes. In Proceedings of UIST 93, pages 197–206. ACM SIGCHI, November 1993.

- [22] T. Igarashi, R. Kadobayashi, K. Mase, and H. Tanaka. Path Drawing for 3D Walkthrough. In *Proceedings of UIST 98*, pages 173–174. ACM SIGGRAPH, 1998.
- [23] T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Pegasus: A Drawing System for Rapid Geometric Design. In CHI'98 Summary (ACM Conference on Human Factors in Computing Systems), pages 24–25, 1998.
- [24] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. In SIGGRAPH 99 Conference Proceedings, pages 409–416. ACM SIGGRAPH, August 1999.
- [25] Crockett Johnson. Harold and the Purple Crayon. HarperCollins Juvenile Books, 1977.
- [26] Joseph J. LaViola Jr. Whole-Hand and Speech Input in Virtual Environments. Master's thesis, Brown University, 1999.
- [27] Michael A. Kowalski, Lee Markosian, J.D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. Art-Based Rendering of Fur, Grass, and Trees. In SIGGRAPH 99 Conference Proceedings, pages 433–438. ACM SIGGRAPH, August 1999.
- [28] Lee Markosian, Jonathan M. Cohen, Thomas Crulli, and John F. Hughes. Skin: A Constructive Approach to Modeling Free-form Shapes. *Proceedings of SIGGRAPH 99*, pages 393–400, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.
- [29] Timothy Miller and Robert Zeleznik. The Design of 3D Haptic Widgets. In Computer Graphics (1999 Symposium on Interactive 3D Graphics), april 1999.
- [30] J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. In Non-photorealistic Animation and Rendering. ACM SIGGRAPH, June 2000.
- [31] R. Pausch, T. Burnette, D. Brockway, and M. E. Weiblen. Navigation and Locomotion in Virtual Worlds via Flight into Hand-Held Miniatures. In SIGGRAPH 95 Conference Proceedings, pages 399–400. ACM SIGGRAPH, 1995.
- [32] Quantel. Quantel Paint Box. http://www.quantel.com.
- [33] Paul Rademacher. View-Dependent Geometry. In SIGGRAPH 99 Conference Proceedings, pages 439–446. ACM SIGGRAPH, August 1999.

- [34] Dean Rubine. Specifying Gestures by Example. In SIGGRAPH 91 Conference Proceedings, pages 329–337. ACM SIGGRAPH, August 1991.
- [35] Emanuel Sachs, Andrew Roberts, and David Stoops. 3-Draw: A Tool for Designing 3D Shapes. *IEEE Computer Graphics and Applications*, pages 18–25, November 1991.
- [36] Steven Schkolne. Surface Drawing: The Perceptual Construction of Aesthetic Form. Master's thesis, California Institute of Technology, 1999.
- [37] P.H. Schneider. An Algorithm for Automatically Fitting Digitized Curves. In A. Glassner, editor, *Graphics Gems.* Academic Press, 1990.
- [38] Karan Singh and Eugene Fiume. Wires: A Geometric Deformation Technique. In SIGGRAPH 98 Conference Proceedings, pages 405–414. ACM SIGGRAPH, July 1998.
- [39] Alvy Ray Smith. Varieties of Digital Painting. ftp://ftp.alvyray.com/PstScrpt/8_Paint.ps, 1995.
- [40] Id Software. Doom. Computer game, 1993.
- [41] I.E. Sutherland. Sketchpad: A man-machine graphical communication system. In Proceedings of Spring Joint Computer Conference, 1963.
- [42] Osama Tolba, Julie Dorsey, and Leonard McMillan. Sketching with Projective 2D Strokes. In Proceedings of UIST 99. ACM SIGCHI, November 1999.
- [43] W. Wang and G. Grinstein. A survey of 3D solid reconstruction from 2D projection line drawings. *Computer Graphics Forum*, 12(2), June 1993.
- [44] Robert C. Zeleznik. Sketching in 3D. Computer Graphics, 32(4), November 1998.
- [45] Robert C. Zeleznik and Andrew S. Forsberg. UniCam 2D Gestural Camera Controls for 3D Environments. In Computer Graphics (1999 Symposium on Interactive 3D Graphics), April 1999.
- [46] Robert C. Zeleznik, Andrew S. Forsberg, and Paul S. Strauss. Two pointer input for 3D interaction. In Computer Graphics (1997 Symposium on Interactive 3D Graphics), April 1997.
- [47] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. In SIGGRAPH 96 Conference Proceedings, pages 163–170. ACM SIGGRAPH, August 1996.

- [48] Shumin Zhai. Interaction in 3D Graphics. Computer Graphics, 32(4), November 1998.
- [49] J. M. Zheng, K.W. Chan, and I. Gibson. A New Approach for Direct Manipulation of Free-Form Curves. In EUROGRAPHICS '98, pages C-327 - C-334, 1998.