Animating Direct Manipulation in Human Computer Interfaces

Bruce Hunter Thomas

Bachelor of Arts, George Washington University

Masters of Science, University of Virginia

A Thesis Submitted for the Degree of Doctor of Philosophy

Department of Computer Science

Faculty of Science and Engineering

The Flinders University of South Australia

Adelaide, South Australia

Submitted 22, August 1997

Contents

List of Figures

List of Tables

Abstract

If judiciously applied, the techniques of animation can enhance the look and feel of computer applications that present a graphical human interface. Such techniques can smooth the rough edges and abrupt transitions common in many current graphical interfaces and strengthen the illusion of direct manipulation that many interfaces strive to present. To date, few applications include such animation techniques. One possible reason for this is that animated interfaces are difficult to implement: they are difficult to design, they place great burdens on programmers, and they demand high performance from underlying graphics systems.

This dissertation demonstrates that direct manipulation human computer interfaces augmented with techniques borrowed from cartoon animators are both simple to program and effective to use. My approach is to add support for animation techniques such as object distortion and keyframe interpolation at the lowest levels of a graphical interface toolkit, and to provide prepackaged animation effects such as animated widgets for common user interface interactions. The low level support for animation simplifies the programming tasks by extending the toolkit's graphics model and by providing management tools for time-dependent information. The prepackaged effects provide aesthetically pleasing animations ranging from animated interaction components to animated visual feedback for direct manipulation of graphical objects.

To determine if these tools and techniques were simple and effective, I built a prototype direct manipulation drawing editor with an animated interface and used the prototype editor to carry out a set of human factors experiments. The experiments showed that the techniques are practical even on standard workstation hardware, that the effects can indeed enhance direct manipulation interfaces.

Keywords

animation, direct manipulation, graphical user interfaces, graphics, toolkits

"I certify that this thesis does not incorporate without acknowledgement any material perviously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text."

Candidate:

Bruce Hunter Thomas

"I believe that this thesis is properly presented, conforms to the specifications of thesis presentation in the University and is *prima facie* worthy of examination."

Supervisor:

Dr. Paul Calder

Acknowledgements

I wish to acknowledge the many people who helped me through the fun and strange times of my PhD studies. First and foremost I would like to thank my wife Judy and my son Sam for being patient with my late nights and weekends at the terminal. Judy as well provided me with great insights into improving my writing. Next, I would like to thank my supervisor Dr. Paul Calder, who has opened my eyes to many ideas concerning computers and life in general. Victor Demczuk provided me with a great service by helping with the statistics in the human factors experiments. I would also like to thank the School of Computer and Information Science at the University of South Australia for all their support. Finally a thank you to the many people who helped me in many ways: David Jacobs and the Thursday Night Dinner Club, Upi Weston, Tiffany Winn, Craig Anderson, Micheal Read, Elena Trichina, Chris Skekette, George Milne, John Roddick, Bruce Tolley, Dean Farrow, John Hammersmith, Richard Frost, David Stotts, Prasun Dewan, Raymond Thomas, and Sally Stearns.

animate *v.* **1.** to give life or movement to, to make lively, *an animated discussion.* **2.** to motivate, *he was animated by loyalty.* **3.** to produce as an animated cartoon. **animator** *n.* □ **animated cartoon** a film made by photographing a series of drawings, giving the illusion of movement.

animation *n.* **1.** animating. **2.** liveliness. [85]

Chapter 1

Introduction

This dissertation explores how techniques borrowed from cartoons and computer animation can enhance the experience of interacting with a computer. However, I am not concerned with portraying animated data, as would be the case when operating animation editing tools (such as an algorithm animation editor) or when using animation to supplement the presentation of otherwise static information (such as an animated help system). Rather, my goal is to apply animation to the interface itself—to enhance or augment the effectiveness of human interaction with applications that present a graphical interface.

1.1 Motivation

Like other recent workers [18, 19, 31], I have set out to extend current direct manipulation style user interfaces with animation. In her book *Computers as Theatre* [37], Brenda Laurel argues that both the computer and the human are active agents working together to achieve some common goal. It is the goal of the designer of an application's interface to facilitate these two active agents in their effort to collaborate as closely as possible. Laurel suggests the use of dramatic techniques to engage the interest and participation of the human agent in these computer/human activities. Animation can be used to enhance some of these dramatic effects; for example users are more willing to suspend their disbelief if the graphics of an interface move in a smooth and realistic fashion. As viewers, we often suspend our disbelief while watching a scene of a movie, or computer screen, which is perceived to be untrue or physically impossible. While watching Superman fly in a movie, we know people can not fly, but we suspend our belief system in order to enjoy the movie. If it is too difficult to suspend disbelief, the movie is no longer enjoyable. Effective animation smoothes the motion of objects on a computer screen and simulates more closely the human agent's expectation of the physical world, leading to an easier suspension of disbelief and a readier acceptance of what is shown on the screen.

Lon Barfield, Eddy Boeve and Steven Pemberton make the observation in *Objects, Invariants and Treacle: Animation in the Views System* [8] that human visual perception has been developed over millions of years. As visual perception developed, it grew more sensitive to moving objects than static ones, and this development should be exploited in user interfaces. Current interfaces can be improved in this respect by animating the transitions between state information in an interface. Standard state information has a "before and after" nature; for example, a push button has an on and off state. Barfield, Boeve, and Pemberton's View System provides a general system-wide way to add "fill-in" animations. These fill-in animations supply continuity between the before and after state information that helps to better convey the change in state of the interface to the user.

Chris Crawford argues in *Lessons from Computer Game Design* [22] that computer interfaces for standard applications have something to learn from interfaces of computer games. His paper states three lessons that may be of use in determining how animation could be included in computer interfaces:

- interfaces should move away from the keyboard;

- there should be a greater reliance placed on graphics and sound;
- the *intensity of interaction* should be emphasised.

In a similar vein, Robert M. O'Keefe and Ivan L. Pitt have shown that users have a strong preference for displays with animations or graphics in visual interactive simulation applications [46].

John Stasko in *Animation in User Interfaces: Principles and Techniques* [64] cites the absence of good tools for building animations into user interfaces and the relative difficulty of using the few existing ones as major hurdles restricting the addition of animation into user interfaces. The addition of the dimension of time into these tools makes the development particularly challenging.

1.2 The Problem

Most direct manipulation applications use a WIMP (Windows Icons Menus Pointing) style interface. A user interacts with a WIMP style interface using three techniques: by operating predefined components (widgets) for common tasks such as issuing commands and setting state; by direct manipulation operations such as moving or resizing application objects; and by indirect manipulation operations such as aligning or changing colour of application objects.

Many WIMP style interfaces use predefined widgets for ease of programming and to obtain a consistent *look-and-feel* between applications. These widgets range from components such as push buttons, pulldown menus, and text labels. Examples of widget toolkits include the OLIT widget toolkit for OPEN LOOK [69], the Motif toolkit [26], and the InterViews toolkit [39].

Direct manipulation aims to give users the illusion that they are directly interacting with application data [60]. For example, a user can move a graphical object in a drawing editor by selecting and moving it with the mouse. Sketchpad [70] by Ivan Sutherland is one of the earliest examples of an application using direct manipulation operations. The interaction with the user was changed from typing textual commands to drawing with a light pen and manipulating buttons and dials. This form of interaction changed the user's manipulation of data from symbolic processing to perceptual and kinetic processing (seeing the data graphically and changing it directly with a tool in their hand). Current computer drawing applications commonly support direct manipulation operations.

Direct manipulation works best when there is a straightforward mapping from user actions to visible changes on the screen; typical examples include dragging an object to a new position, stretching it to make it larger, or rotating it around an origin. But some changes have no obvious mapping (like changing an object's colour), and others require more precision than can readily be achieved by direct action (like rotating an object by exact quarter turns or moving it to exactly align with another). One way to permit such operations is to provide commands that carry out the desired change, together with a way to invoke the commands as needed. In the context of graphical interfaces, the commands are usually invoked by pressing a button in a toolbar or selecting a choice from a menu. This style of interaction is called *indirect manipulation*, since an action on one object (a button, for instance) causes a change to another object.

1.2.1 Research Questions

The effects of animation on the user's perception of the interface can be profound, like the effects of other aesthetic elements such as colour. On the one hand, this influence suggests that great improvements are possible; on the other hand, it warns that equally great disasters can happen. Just as there are good and bad uses of colour, so there will be good and bad uses of animation. Sensibly applied, animation should make an interface more graceful and enhance a user's enjoyment.

But inappropriately applied, it will seem childish and drive them away. In addition, the process of constructing animations is a time consuming and highly skilled task. Easy-to-use and powerful tools are required to reduce this complex task to a manageable size.

In this investigation, I have chosen to address the following four questions:

1. *What portions of the user interface are to be animated?* Animation could encompass the entire visual portion of the user interface, as is the case with a virtual reality interface. Alternatively, small regions of the interface could be animated; for example, current web browsers use animation to draw the user's attention to a particular portion of the screen. A related issue is what forms of visual information in the interface are to be animated: images, text, control components (widgets), or drawn graphics?

2. *How should the animated feedback be presented to the user?* Once the animated portion of the user interface is determined, the visual feedback animation effects need to be identified. Cartoon animators use a large arsenal of animation techniques and effects to produce pleasing and entertaining animations. A judicious application of these techniques and effects can enhance the illusion of direct manipulation that many human computer interfaces strive to present.

3. *Where should the animation tools be placed in the software programming model?* Animation tools could be part of the low-level graphics library or of a high-level workbench. The question can be viewed in a different form: should the software tools support the construction of animated objects or the animation rendering process. Once the location in the programming model has been determined, the kind and form of the software tools can be specified.

4. *Is the animated interface an improvement?* An empirical study is required

to determine if the animated interface is an improvement over existing forms of direct manipulation interfaces. This study should include objective results of improvement of a user's ability to use a direct manipulation software application, and subjective results of how a user "feels" about using interfaces that include animated visual feedback.

1.2.2 Research Goals

This dissertation demonstrates that direct manipulation human computer interfaces augmented with techniques borrowed from cartoon animators are both simple to program and effective to use. I set the following four research goals:

1. determine the portions of the user interface to be animated;
2. describe the animation effects for visual feedback to be presented to the user;
3. design and implement a set of easy-to-use but powerful software tools;
4. evaluate the effectiveness of animation effects in a substantial application.

1.3 Structure of the Dissertation

This work builds on the work of animators from the studios of the likes of Warner Brothers and Disney. Chapter 2 provides an introduction to the animation process, including how computers can aid this process, and where animation has been previously applied to a user interface. Chapter 3 defines three sets of animation effects for the improvement of visual feedback in graphical user interfaces: *smoothness in the interface*, *manipulated objects with substance*, and *anticipating the result of a manipulation*. Chapter 4 describes tools that are required to support animation in a user interface. Many of these tools support animation in general, but they are defined in such a way as to make them practical as an extension to an existing user interface toolkit. Chapter 5 describes the implementation issues to support the animated visual feedback effects discussed in Chapter 3. Chapter 6 goes on to describe a substantial application, a drawing editor, with an animated interface developed to demonstrate the animation effect in a "real life" example. The chapter also presents the design and results of a set of human factors experiments to evaluate the animation effects in the context of the drawing editor.

1.4 The Contributions

This work makes four key contributions in the context of animating direct manipulation graphical interfaces:

1. It describes a set of animation effects to extend the visual feedback for direct manipulation interfaces.
2. It shows these animation effects to be effective and enjoyable for users.
3. It presents the design and implementation of a set of software tools to support these animation effects.
4. It shows that convincing animation effects can be effectively incorporated into a significant application, a drawing editor.

Chapter 2

Animation Techniques for User Interfaces

Computer interfaces construction can profit from techniques and principles used by cartoon animation artists. For many years graphic artists have been employed to design static icon images. It would be beneficial to extend this practice to the dynamic nature of interfaces. This chapter is an overview of the following key concepts: animation, computer-generated animation, and the current use of animation in user interfaces.

The first section of this chapter gives a general overview of the process of creating an animation, whether computer-generated or non-computer-generated. Problems specific to computer-generated animations are then discussed, and descriptions of relevant techniques and principles taken from the general art of animation are presented. Descriptions of different forms of animation that can be applied to computer interactions are then introduced. Finally, examples of existing systems are presented to elaborate on the techniques and types of animations used in the different interfaces.

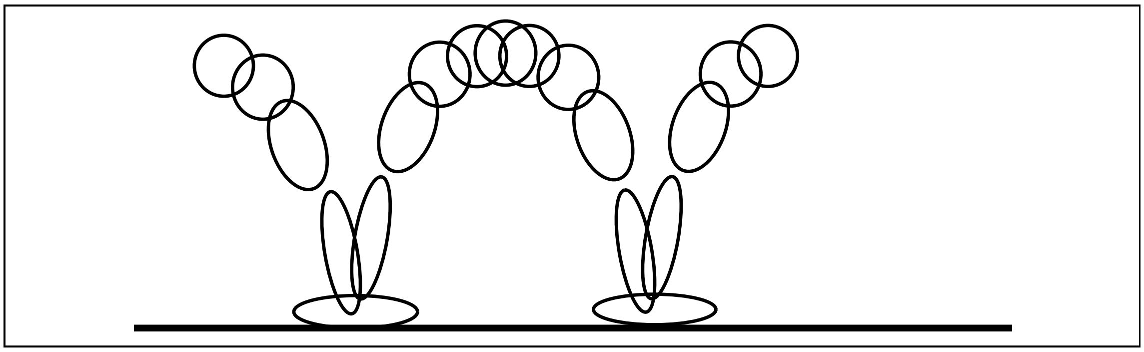


Figure 2.1: Animation of bouncing ball

2.1 Animation

Animation is achieved by presenting images at a rate fast enough to give the illusion of motion. These images are referred to as frames in an animation or movie. They can be taken from photographed objects, drawn by hand, or rendered by a computer. However the images are created, they are ordered and shown at an appropriate rate to produce an animation. This section provides a basic understanding of how animations are generally created.

2.1.1 The Mechanics of Animation

This section describes some of the skills and techniques animators use in their art. Defining the appropriate amount of change in an object's position between each frame of an animation is the core objective in creating a convincing illusion of movement. The basic natural laws of motion also must be considered in order to determine the amount of change between frames.

A common first exercise for animators is to investigate the rate of change of a single object, and this object many times is a bouncing ball. Figure 2.1 demonstrates a very simple animation of a bouncing ball. Static images cannot fully convey the dynamic nature of an animation, but several successive frames from the bouncing ball animation are superimposed in the figure to suggest the overall animation. This short animation can be repeated as many times as desired, to give the illusion of a ball bouncing along a hard surface. Note how the ball loses speed on the way up the path and gains speed on the way down, how the ball compresses as it hits the surface, and how the ball stretches along the direction of motion.

The pace of individual movements can add information and meaning to the viewer about that particular action. For example, a elephant might move more slowly than a mouse, conveying the sense of a larger size and mass. In the case of the bouncing ball example, a fast bouncing ball with very little squashing could give the impression of a hard glass marble bouncing, while a slow bouncing ball with a large amount of squashing could give the impression of an under-inflated basketball. The process of determining the pace of these actions is referred to as the *timing* of the animation. These are examples of timing of an individual animated object, but timing is also used to describe the pace of groups of animated objects and actions.

The second issue to be considered is the general order in which the frames are created to make an animation. This may be divided into the two main categories of keyframes and straight ahead action. *Keyframe* animation is the process of drawing the major or *key* drawings of the animation first. The *in-between* drawings are then drawn to fill the gaps between these keyframes. *Straight ahead action* animations are drawn starting at the first drawing and working on each successive drawing to the last.

The process of drawing the keyframes of an animation allows the animator to plan an animation. A keyframe defines an extreme point in the motion of an object. The positions of the objects in the in-between frames are interpolated from the keyframes. Figure 2.2 shows an example of possible keyframes of the bouncing ball example: the ball hitting the floor, the top of the arc, and the middle time positions. It is normal to have more than one in-between frame between each keyframe.

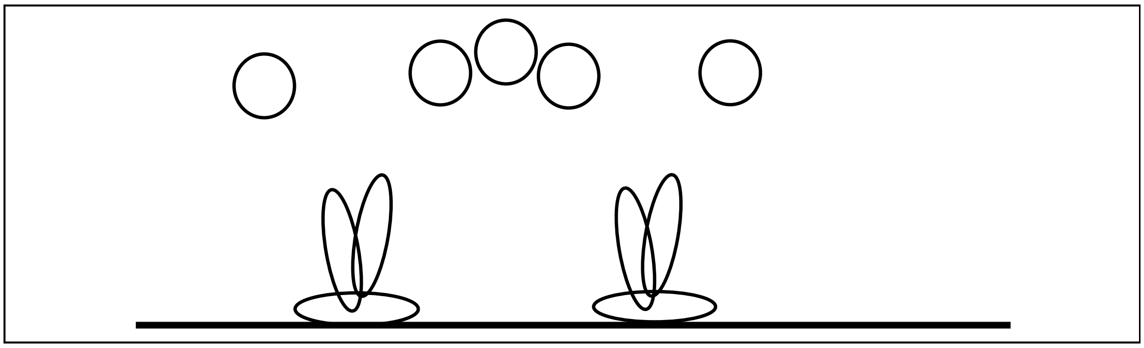


Figure 2.2: Keyframes of the bouncing ball

The final operation to be discussed is the reuse of existing frames. This is an important shortcut in the process of constructing an animation, especially considering that a single ten minute animation on 35mm film requires 14,400 frames or 900 feet of film [38]! Two standard methods of reuse are using *cycles* of drawings and *holding* the same drawing for multiple frames of an animation. As an example of using cycles of drawings, the bouncing ball in Figure 2.1 can be repeated as many times as needed to form a continuous animation. Each new cycle merely has to move the width of the arc either to the left or right, depending on the direction of the animation.

2.1.2 Animation Effects

Animators use a broad range of effects to enhance the illusion of the animation [81]. This section presents a subset of effects used in both hand-drawn and computer-generated animation: *anticipation*, *squash and stretch*, *motion blurring*, *exaggeration*, *slow-in/slow-out*, *arcs*, *secondary action*, *follow through*, and *overlapping action* [35]. Figure 2.1 uses a number of these effects, and is used as an example throughout this section.

Anticipation prepares the view for the next movement in the animation, so they can expect the movement before it happens. This can be used to broadcast the nature of the action or to attract the attention of the viewer to a particular portion of the screen, by giving the viewer visual or audio cues as to what is about to happen in the animation. The anticipation may be manifested as internal tension, or potential energy. In Figure 2.1, the ball is like a coiled spring about to be released. Once the ball is allowed to "spring" back into shape, it shoots up along the arc. This is an example of using *squash and stretch* to depict the feeling of *anticipation*.

Squash and stretch distorts a moving object to suggest its rigidity and mass. An object is *squashed* to show its inertia or flexibility. In the case of the bouncing ball, the distortion is high on impact. Similarly, when a boulder is dropped on the head of a cartoon character, the squash effect shows it to be massive. A rule of thumb for animators is that to appear convincing, the cross sectional area of an object should remain constant during a squash operation.

The *stretch* effect is also used to demonstrate an object rapidly accelerating, and when motion blurring is not available the stretch effect is an alternative mechanism for temporal anti-aliasing [35]. In the bouncing ball example, the ball is elongated along its trajectory. On the way down, the ball appears to be pulled towards the ground, and on the way up, the ball seems to leap off the ground. The Disney animators claim this adds visual realism to the final animation. If an object moves substantial distances between successive frames, then it will seem either as if there are multiple copies of the object at different positions blinking on and off, or as if it jumps across the screen and flickers when drawn. This effect, called *temporal aliasing* or *strobing*, can be reduced by artificially *stretching* the object in the direction of motion to fill in the gaps.

Another common technique for removing temporal aliasing is *motion blurring*, which smears or blurs the object's features in the direction of motion. If an object moves rapidly without motion blurring effects, the object will seem to disappear from one place and reappear at a new location. The user is then forced to reevaluate the context of the object. If the same action is displayed with motion blurring, the viewer perceives the object as moving.

Animators often make actions “larger than life.” Surprisingly, these *exaggerations* of the actions, moods, emotions, sounds and colours make an animation seem more realistic. A balanced set of exaggerations for the objects is required to stage the animation correctly. An animation is pleasing and believable if these actions are appropriately exaggerated. Exaggeration in Figure 2.1 occurs in the amount the ball is squashed. The squashing is more than would happen in a real life situation, but this exaggeration emphases the act of the ball hitting the floor.

The animation effect *slow-in/slow-out* deals with the spacing of animated objects at the extreme poses. Slow-in/slow-out draws on the natural laws of physics. For example, objects do not change speed or direction instantaneously: there must be a finite acceleration. Starting from rest, an object is required to accelerate up to its final speed; to come to rest, it must decelerate to a zero speed.[1] As stated above, actions in an animation must be appropriately exaggerated. Animators do not strictly stick to natural laws, but they apply large than life accelerations to present an animation with the correct *feel.* In the case of the bouncing ball, the speed of the ball is greatest just before it hits the ground. Because of the force of gravity, the average speed of the ball in Figure 2.2 is greater towards the ground than at the apex of the arc. The ball slows down as it approaches the apex of the arc. When the ball is squashed on impact with the floor it decelerates to a stopped position, then accelerates back to its speed just before impact.

To give animations a smoother, less stiff look, objects are animated to move along *arcs* rather than straight lines. This form of motion emulates natural objects, for example the parabolic arc of the bouncing ball or the sinusoidal bobbing of a human head during walking.

[1]For a more complete list of natural laws in cartoons see “Loony laws govern ’toon world physics” - *The Institute*, Oct 1994 page 12.

Secondary action is action that results from another action. If the bouncing ball were to come to a sudden stop, it would vibrate from its inertia. A basic animation principle is that an action is never brought to a complete stop before another action starts. In the case of a person throwing the bouncing ball to start the example, the person's hand keeps moving in an arc once the ball is released. This is an example of a *follow through action.* The person then starts into the next action, for example running forward. This is a form of an *overlapping action.* *Moving hold* is a combination of overlapping and secondary action. When a character is strikes a pose in an animation, the character can not merely be drawn frozen in one position. The key is to make two drawings of the pose, one more extreme than the other, yet both containing all the elements of the pose. The idea of moving hold is to hit the pose, then drift into an even stronger pose.

2.1.3 Staging and Directing Animations

Karp and Feiner [34] investigated how to automatically plan the staging and direction of computer-generated animations. The planning of the presentation encompasses the principles of *staging*, *appeal* and *motivation.* *Staging* an animation consists of presenting the overall effect of the animation clearly to the viewer. *Appeal* is the ability to draw and hold the attention of the audience. Simply put, "Does the animation work for the viewer?" *Motivation* is the process of providing an animation that convinces the audience that a given movement is the only possible kind for that situation.

The authors state, "We believe that presentation decisions for animation should rely on a knowledge of filmmaking techniques that have been developed to design visually coherent presentations." This concept is in keeping with the spirit of Lasseter [35] and other computer animators. This section discusses these filmmaking techniques, the basic building blocks of film, and the techniques for gluing these blocks together.

The basic units for describing portions of a film are *frames, shots, scenes,* and *sequences.* The number of frames per second is known as the *frame rate.* The frame rate of standard video systems is 25 or 30 frames per second and 24 frames per second for film. A continuous stream of frames from start to stop is known as a *shot.* One or more shots that represent the same event are called a *scene.* Shots with different camera angles and fields of view can be joined together to construct a scene. A set of scenes build up a *sequence.* A sequence forms a major segment of a movie, as a chapter is a major portion of a book. Finally, a film is the set of sequences placed in the proper order.

Different shots can direct a viewer's attention to a specific action or position on the screen. Shots from different angles of a subject allow the viewer to build up a 3D mental model from a set of 2D images of the subject. The number, type, and frequency of changes, or *transitions*, between shots helps to maintain visual interest for the viewer. The viewer scans each new shot for additional information for their mental model of the subject. To keep the interest high, a shot rarely lasts more than several seconds.

Five examples of the types of transitions are *cut, fadein, fadeout, dissolve*, and *wipe.* A *cut* is the joining of two shots with no transition effect; the last frame of the first shot is immediately before the first frame of the second shot. A *fadein* transition starts with a blank screen, commonly black, and then over a few seconds the shot reveals itself, strengthing from a faint image to the full intensity of the shot. The reverse of a fadein is the *fadeout* transition, where a shot begins with the full intensity of the shot and reduces to a blank screen. A *dissolve* transition performs a fadeout of the first shot, while simultaneously the second shot performs a fadein. A *wipe* has the effect of "pushing" the first shot off the screen while the second shot expands from the side or the centre to fill the view.

All of these transitions provide different illusions for the viewer and give the filmmaker different options for manipulating the timing and staging of the film. Further, two shots can be shown simultaneously with the use of multiple viewports. Two common multiple viewports used include inserting one shot into a sub-region of a second shot, and splitting the screen by having one shot on the left side and the other shot on the right side.

Finally, continuity editing of the film is performed to minimise jarring transitions and to maintain consistency throughout the film. For example, a person should be wearing exactly the same clothes from one shot to another, during a scene. A shot of a moving car from the left side of the car should not be followed by a shot from the right side. A transition such as this leaves the viewer with the impression that the car is going in the reverse direction. An intermediate shot from the front or rear of the car is needed to help with the transition. To minimise jarring transitions, Karp and Feiner describe a set of rules that have been developed to join shots together. These include the following:

1. Objects should be moving in the same direction in both shots.
2. A subject's direction of gaze or gesture should be in the same direction in adjacent shots.
3. Changes to camera position or field of view should be substantial enough to appear to have some purpose.
4. Objects should appear on the same side of the screen in adjacent shots.
5. Cuts should not interrupt a steady flowing action.
6. Cuts in steady flowing action should have overlapping frames as a transition.

2.2 Computer Animation

Computer animation is the process of designing, constructing, and recording animations with the aid of computers. Many applications have been enhanced with computer animation: scientific visualisation, visual simulation, animated video, film production, advertising, education, and interactive courseware.

In the process of constructing computer animations, a computer may assist the animator in rendering the images, but the design and artistic qualities are still required from the animator. The generation of these animations can be divided into two separate tasks:

1. the determination of objects and actions in the virtual animation world;
2. the presentation of these objects and actions.

This section outlines the process of specifying computer animations in terms of objects and actions for computer animation packages or programs. The animation process includes visual effects performed on a graphical object, and the manipulation of the camera or eye. A good overview of basic computer animation principles and techniques can be found in *Computer Graphics - Principles and Practice* by Foley, van Dam, Feiner, and Hughes [33].

2.2.1 Computer Graphics for Drawing Frames

The first form of assistance that a computer may give during the construction of an animation is the rendering of each frame. Generally the frames are defined by free-hand drawing or geometric models. Of most interest to the work of this dissertation is the rendering of images defined by geometric models. This section gives an insight into the process of defining one image using geometric models, by showing some of the common animation parameters to be specified by the animator. The examples are drawn from personal experience using the *imagine* [59] system,

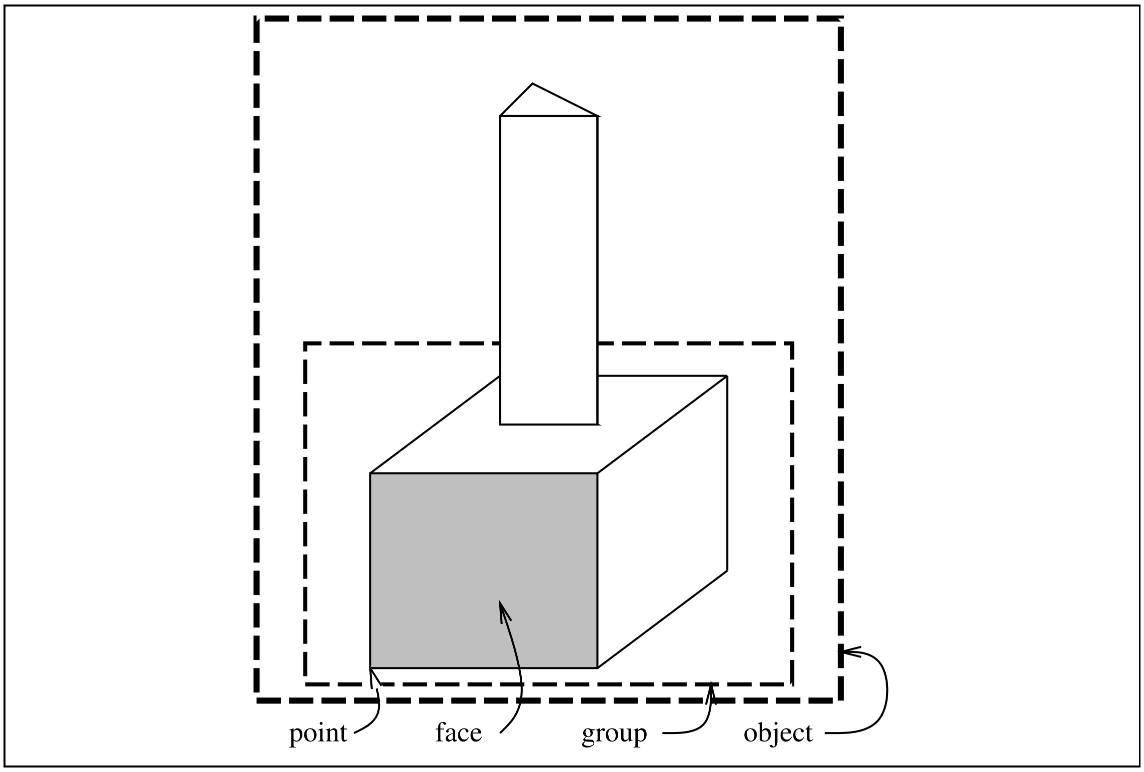


Figure 2.3: An example of a 3D graphical object

one example of this type of animation package. The animation parameters used in this section are typical of this genre of animation systems.

First the physical shape and structure of the objects to be rendered are specified. The fundamental unit for defining a unique position is a point, which can be defined with two values (x,y) for 2D spaces and three values (x,y,z) for 3D spaces. (Cartesian coordinate systems are the norm, but polar coordinate systems can be used.) Two points define a line, and an ordered list of points defines a polygon or spline. A face can be defined as a polygon with four co-planar points. A set of faces can be placed in a group, and a number of groups together constitutes a graphical object. Each face has a number of attributes, such as *colour*, *transparency*, *texture*, and *material*. Figure 2.3 illustrates these concepts.

The position, orientation, and scale of a graphical object are changed with transformation operations. Figure 2.4 is an example of these three transformation operations. The star on the left is moved to the right, rotated 90 degrees, and

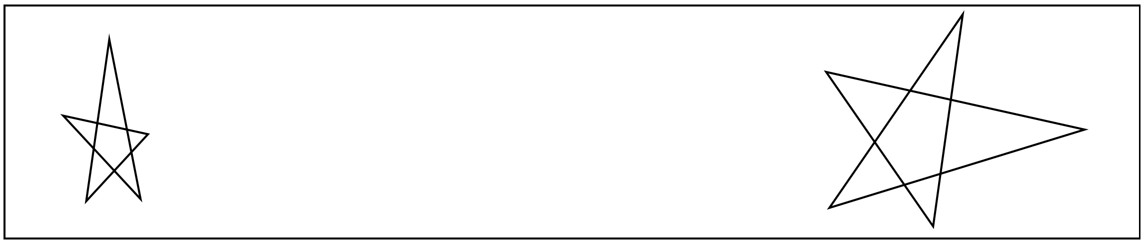


Figure 2.4: An example transformation of an object

scaled to a larger size. Since this transformation can be applied to any level of the graphical object's structure, control can be as local or global as needed. These transformation operations are conveniently represented by a *transformation matrix*, which is the product of one or more translation, scaling, and rotation matrices. For 2D graphics this matrix can be described with a 3 by 3 matrix; a 4 by 4 matrix is needed for 3D graphics.

The *lighting* of a 3D scene is defined by one or more light sources. Each light source has attributes to define its position, direction, colour, and diffusion. Finally, the camera, or eye, is defined by its position, orientation, and focus. Objects, lights, and cameras all have to be specified to make a complete image.

2.2.2 Computer Aided Animation

Computers can aid the keyframe animation process by providing an easy-to-use interface for the specification of the keyframes and by performing the interpolation between the keyframes. The following description shows how computer aided animation can be generated by manipulation of geometric models.

First, the keyframes to be rendered are defined, by creating an image for each keyframe as a set of graphical objects specified in an internal coordinate system. The physical shape and structure of an object is denoted in the manner outlined in the previous section. The position, orientation and size are defined by a transformation matrix associated with the graphical object. Parameters are determined for

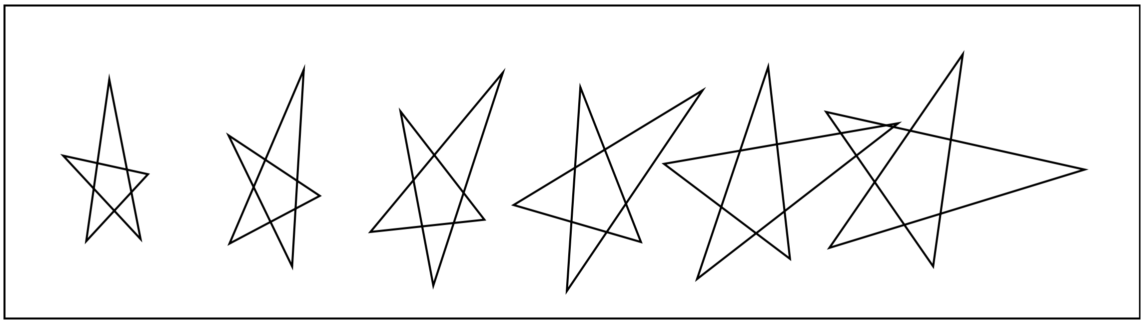


Figure 2.5: An example of interpolation for a 2D graphical object

each object, including colour, transparency, and texture. The lighting and camera position and orientations are stipulated.

Second, the interpolation between the keyframes is specified, by determining the amount of motion and change between keyframes as increments over time. The changes can be, for example, to the object's colour, shape, size, or position. Figure 2.5 shows how the example in Figure 2.4 could be generated by interpolation. The interpolation process requires the start and end transformation matrix information and a given number of in-between frames. In this example, the animator might specify the interpolation as follows:

1. move the object to the right by 200 units,
2. rotate the object about its centre 90 degrees,
3. scale the object about its centre by 2.0, and
4. perform these changes over 5 frames.

Animations in 2D have five degrees of freedom: translate x, translate y, rotate, scale x, and scale y. In contrast, there are nine degrees of freedom in 3D animations: translate x, translate y, translate z, rotate about the x axis, rotate about the y axis, rotate about the z axis, scale x, scale y, and scale z.

Functions or graphs can specify the rate of change over time; this allows animators to include effects such as slow-in/slow-out. Curved motion of an object's position over time is specified as a curved line, which is referred to as its *trajectory* or *path*. Changes to the camera can also be specified with a trajectory, giving camera effects such as panning, zooming, or trucking (moving the camera.) The lighting can change over time in position and colour in the same manner as graphical objects. Effects such as sunrises and sunsets can be achieved by moving light sources.

Finally, rendering the animation is the process of drawing each image with the option of also capturing them on some lasting medium, such as film, hard disc, or video. Two of the emerging standards for storage of animations in digital form are MPEG [27] and QuickTime [30]. The rendering process combines all the animated objects and animated scenes into one total animation. The start and stop times of each of the planned trajectories is combined onto one time line. The changes of lighting and camera angles are coordinated at this point. When all the frames have been created, the *timing* of the entire animation can be determined. The timing defines such aspects as the exact frame an object starts to move. These forms of timing can affect the entire *feel* of the animation.

2.2.3 Computer Controlled Animation

The amount of control the animation system has over the changing graphical object may range from assistance, as in the previous section, to autonomous actors directing graphical objects by high level commands [33]. Examples of computer controlled animations of varying degrees of complexity are described in this section. These include *full explicit control*, *procedural control*, *actors*, *constraint based systems*, and *kinematics and dynamics*.

The animator can specify the changes to the graphical objects by *full explicit control*. This is the simplest form of animation control. The animator defines the full set of attributes for every graphical object, for each keyframe, with the animation system supplying no extra help.

The process of *procedural control* is to have different elements of the graphical structure communicating with each other. In this way a dynamic model of the system can be described. A classic animation of this form is of spheres in a box modeling gas molecules. At the start, each sphere sets its initial position and velocity and is programmed with a procedure to determine what its current velocity is at a given time. These procedures can be the same for each sphere. The model is updated at predefined intervals. In the case of recording an animation, the length of the interval is the frame rate. At each update, the system calls each procedure with the current state of the system, and each sphere determines how its velocity will change, given this configuration of the model. Complex behaviour can be modelled with very simple procedures.

Actors are a more complex and powerful form of procedural control. Instead of a procedure being called for each object, each object has a separate process or program controlling it. This removes the rigid predefined intervals of the procedural control and allows the graphical objects to be more easily specified to have autonomous control. For example, a bird could be programmed to fly around the top of a tower in ever widening circles. With the exception of bumping into the bird, the rest of the graphical objects can ignore the bird's movements. Animators use actors to decompose the animation task into smaller separate processes.

Constraint based systems allow the specification of the movement based on relationships between objects [11, 12, 43]. A simple example of this is a swinging pendulum under the influence of gravity. The constraint of this system is that the pendulum and the pivot point must always remain the same distance apart. Taking into account the constraints and forces, a *constraint solver* is used to determine the next position of the pendulum system. Once again, this can be set to coincide with the frame rate. A more complex example is a human figure modelled as a set of multiple pendulums attached to the figure's trunk. Other constraints, such as in which direction of each the pendulums can swing, can also be added.

Kinematics and dynamics allow the specification of the movement based on natural laws. *Kinematics* defines a framework to define positions and velocities of points, and *dynamics* extends this framework to work under a set of natural laws, for example Newtonian mechanics [28]. In the bouncing ball example, the animation could be defined as a set of forces and velocities. A set of equations describe the velocity of the ball and the set of forces acting on it. The surface of the floor, defined by a plane, supplies a reactive force when pressed. The force due to gravity is a second force. The spring constant of the ball is defined to determine how high the ball bounces back up. Friction and drag can also be defined. With these forces and velocities, the position and shape of the ball is defined for the entire animation. The set of equations can be as simple or complex as required for realism.

2.3 Animated Interaction

Animated interaction includes a wide spectrum of applications and effects, ranging from the complex and highly detailed animations of computer games, virtual reality, and 3D simulations to simple animation effects, such as the spinning globe in the Mosaic web browser [44] to inform the user that the application is still active. Animated interactions are used to engage the user, provide additional information and allow the user to better use information [64].

Ronald Baecker and Ian Small in their article *Animation at the Interface* [6], outline different ways in which animation can be added to the computer interface. They list the major roles for animation as follows:

1. animation of structure: exploring complex environments;
2. animation of process: visualising algorithms and programs; and

3. animation of function: making interfaces more comprehensible.

Animation of structure in an interface enables a user to explore complex real and imaginary environments. A virtual reality environment with only static frames would be of little use or enjoyment. Animation allows the user to replace the natural visual stimulus with computer-generated images. This enables the application to place users in a new visual world; they can interact with this world to perform a required task. These tasks can range from tele-presence robotic control to shooting down robots in a computer game.

Animation of process enhances the understanding of an evolving process with changing state information. The process could be detecting weather patterns, assembling a car, or programming a computer. Each of these tasks has the user apply pattern matching abilities to the tasks. Often the patterns can only be recognised when the information is presented in the correct order and at a speed that enhances the details of the pattern. This may require the information to be presented at a greater rate than recorded, as in the case of viewing satellite photos in weather prediction, or at a slower speed than recorded, as when viewing state information of an executing computer program. John Stasko [64] points out animations that provide feedback in real time about the status or state of a system need to provide an accurate chronological summary of the operation, and that explanatory or online help animations should have speed control for the user; this enables users of different levels of ability to acquire the information at different rates.

Animation of function is concerned with making the operation of interfaces more comprehensible through the addition of animation. Short film clips or animations may assist comprehension through the presentation of explanatory material, such as for online help, manuals, or textbooks. Animations allow the presentation of dynamic information. For example in the case of a manual, this information could be a movie of how to remove the covering from an electric motor. This information is presented to the user in a more legible and understandable manner than using traditional paper manuals. As a tool for the creator of the manual, this method of providing information could save time, money and make a better product.

Animated interactions in this dissertation are categorised into *autonomous* and *tracked* animations. An *autonomous* animation is a predefined animation initiated by the user. For example, the starting of an application may present an animation signaling the loading of data. A *tracked* animation augments a user's action, such as a direct manipulation operation on a graphical object. For example, a user might grasp a graphical object and drag it across the screen, during which the tracked animation *stretches* the object to help prevent temporal aliasing. This section describes autonomous and tracked animated interaction and examines in more detail how they relate to Baecker and Small's taxonomy. The taxonomy is further broken down into several subroles that are illustrated by scenarios.

2.3.1 Autonomous Animations

Autonomous animation is the most common form of animation in the user interface of traditional computer applications. Autonomous animations are usually initiated by user action, deterministic in length, deterministic in content, and repeatable. Since the content of the animation is predetermined, an autonomous animation can be thought of as a *canned* animation. For example, a user tells a Web Browser to retrieve a file from a remote host. While the Web Browser is retrieving the file it signals its actions by spinning a logo, such as a globe, or by showing the progress as an animated sliding. The animation can, however, be parameterised to help make it more general. For example the name of the file could be a parameter and added to the title bar of the animated sliding scale. The parameters are used to change the attributes of the animation, but not the essence of the animation.

Autonomous animations are categorised, in this section, into the following areas: improving information legibility of changing information, smoothing of abrupt changes to visual interfaces, and providing improved information with animated help.

Improving Information Legibility

Much of the information people interpret and respond to is dynamic in nature. The domain of software visualisation includes the visualisation of abstract algorithms, and often incorporates the use of animation as a means of displaying the dynamic nature of the information [52]. The execution of a computer program has a temporal dimension; the statements of a computer program are executed in a certain order for a predefined input data set. A programmer's understanding of the execution of a program can be enhanced if the state information is presented in conjunction with the temporal information [82, 83]. Animations have also been used to visualise the temporal aspect of Petri Nets [51]. Tokens in the Petri Nets are animated to move to new locations at each firing. Systems with message passing protocols can animate the instances of messages moving between processing nodes or objects. Animation can also be used as an abstraction to show how algorithms process data. The animation of sorting algorithms was an early example of this abstraction; recent sorting algorithm animations are now moving from 2D images to 3D fully rendered images [14, 66].

Smoothing

Baecker and Small suggested that animation may be used during transitions from one process to another in a user interface, thus smoothing the transitions. Animated transitions can focus the user's attention on changes to the working environment, such as location and identity of newly created entities and new areas of interest. Further, they can improve the legibility of the interface by maintaining object consistency during changes.

Transitions may cause dramatic changes to large portions of a computer's display. Consider the operation of a popup menu. At one instant, the menu is represented by just a button; the next instant a complete menu window is visible, obscuring the underlying information. The user is momentarily startled; a small but significant disruption occurs before the new appearance of the screen is digested and work can proceed. With experience, users learn to ignore the disruption, but some of its effect remains. The overall impression is of an abrupt, hard-edged interaction that is jarring and tiring.

Animation can help smooth these rough edges by more gradually changing the screen's appearance from the old state to the new. Researchers [23, 55] have shown that a smooth change, even if it occurs rapidly, shifts the user's interpretation of the change to the perception system, allowing the cognitive system to stay focused on the task at hand. The overall effect is that the interface seems smoother and more pleasant to use.

Traditional graphical user interface toolkits provide a set of standard widgets for the application programmer, such as push buttons, pulldown menus, status buttons, and scrollbars. Each of these widgets provide graphical information reflecting the current state of the widgets; for example, a menu button is highlighted when it is activated. The widgets can be animated to smooth the change of state [86].

Components well suited to the application of animation are buttons, windows, and menus [19, 76, 78], which change appearance in predictable ways. A canned animation sequence can be custom-crafted for each type of component. For example, instead of a pulldown menu suddenly appearing when the user clicks on its tag in a menu bar, the menu might slide out from behind the menu bar and slide back when no longer needed. This sequence, which is repeated each time the menu is operated, can be designed by expert graphic and animation artists and built into the implementation of the widget. Other examples include a check box that animates the drawing of the check mark as if it were stroked with a pencil, a toggle button that flips over to reveal its alternate state on the reverse side, and a palette that rotates to bring alternate choices into view. Application programmers can use the animated widgets as easily as they currently use static widgets.

Animators routinely use a range of techniques to improve the realism of their work, and these techniques can be used to improve the dynamic nature of user interface components. For example, the slow-in/slow-out principle that is used in calculating the positions and sizes of intermediate frames in animations to emphasise the acceleration and deceleration can be applied to an animated menu that grows to full size. The menu would grow slowly at first, quickly in the middle, and then slowly again as it approaches full size. The effect would make the menu action seem smoother without increasing the overall duration of the animation.

A second example of smoothing is in the interface for *information visualisation.* The work of George G. Robertson, Stuart K. Card, and Jock D. Mackinlay[17, 40, 57] pioneered the area of 3D information visualisation. The effectiveness of this work relies heavily on good interactive animation. One of the 3D information visualisation tools they developed the Cone Tree [57], which uses smoothly rotating cones to display large hierarchies in a limited space. The perceptual phenomenon of object constancy enables the user to track the substructure relationships without thinking about it. When a rotation animation is complete, no time is needed to reassimilate the new configuration of the information space. The authors make two claims [57]: "*It seems clear that interactive animation can effectively shift cognitive processing load to the perceptual system. And it seems plausible (but not yet proven) that 3D can be used to maximise effective use of screen space.*"

Animating Help Systems

Baecker and Small's concepts may be applied to improve online help systems. Firstly, animation may be used to identify an application quickly and vividly when it is invoked. Secondly animation may provide more interesting, understandable, and communicative tutorials and explanations. Finally, the use of animation may improve the information content of icons or symbols used to perform actions. Animated help and software visualisation are two of the major areas in that animation has been used to improve information legibility. This section reviews three sets of studies that have investigated this issue.

Susan Palmiter and Jay Elkerton [47, 48], and Patricia Baggett [49], evaluated the differences in the retention of information learned from an online help system with animated demonstrations verses one with written instructions. They found that animated demonstrations helped the user with the initial learning and that users preferred animated demonstrations over written instructions, but the retention of information seven days later was better for text-only users.

Patrick Waterson and Claire O'Malley [87, 88] also found that, for graphically oriented applications, animated demonstrations improved the learning task for the immediate test phase. They found that subjects taught to use the MacDraw graphical application by using animated demonstrations completed tasks in a shorter time than users taught with online textual information. However in contrast to Palmiter *et al.*, this advantage was sustained for a week after the immediate test phase of the experiment [88]. The authors noted that further investigation is needed to compare different types of interfaces and tasks.

John Stasko, Albert Badre, and Clayton Lewis [61] used an empirical study to determine if animation assisted students in learning about a *pairing heap* data structure as an implementation of a priority queue data structure, and in particular, if it was more beneficial to procedural understanding than declarative. The results of the study showed that students felt the animation assisted them in understanding the algorithm, and the animation increased their interest in the lesson. However the study found that the quantifiable evidence for supporting the effectiveness of algorithm animation was “disappointing.” The difference between text-only lessons and text plus animation lessons was not large and was not statistically significant. This supports some of the findings with respect to animated help systems; people like animated interfaces, but the benefits of those systems are limited. Like the previous studies into animated interfaces, the authors believe animation can be useful, but the application of animation to the system could be improved.

2.3.2 Tracked Animations

The purpose of tracked animations is to augment a user’s actions, which are usually direct manipulation operations on graphical objects. This section explores the current state of the art of using tracked animations in user interfaces. First, the concept of feedback from traditional direct manipulation interfaces, as a form of tracked animation, is examined. The use of tracked animations to make directly manipulated objects appear to have substance is then investigated. This strengthens the belief of the user that they are applying actions to “real” objects. Finally, the use of tracked animation to enhance the sense of realism of artificial environments is discussed. This is especially important to the interfaces of simulations, games and virtual reality.

Direct Manipulation and Tracked Animations

In his seminal paper *Direct Manipulation: A Step Beyond Programming Languages* [60], Shneiderman identifies the essential ingredients of direct manipulation: immediate response to actions, incremental changes, and reversible effects. In essence, the technique works because it removes intervening machinery between the user and their task. For example, in a WIMP style interface, the user can gesture with a mouse pointing device to move a graphical object on the screen. Users believe that they are directly interacting with application data, even though they are interacting through a remote tool, the mouse. Central to this tight coupling between the user and the application, which Brenda Laurel calls suspension of disbelief [37], is the feedback that the application generates to track the manipulation.

A key component of WIMP style interfaces is the tracked animation of manipulated objects. Consider the common operation of moving an object by dragging. The application must provide continuous feedback during the drag so that the user knows where the object is at all times. The continuous feedback is a simple form of tracked animation. Simple feedback techniques based on drawing object outlines are cheap to implement. More ambitious schemes, such as drawing the full-featured object during the drag, give the user better graphical displays, but they require more computational resources.

Tracked animations can be used for more than just moving objects around a display. In graphical editors, tracked animation can be extended to scaling, skewing, rotating, and constructing a graphical object. At the most fundamental level, the cursor movement attached to the mouse movement is a tracked animation, as is the highlighting of menu items in a pulldown menu.

Substance

Simple feedback techniques based on drawing object outlines tend to give the impression that a direct manipulation operation is happening to a surrogate object rather than the real thing. The drawing of a full-featured object during the operation gives the object solidity but can still fail to convey a sense of substance; somehow it seems *too easy* to manipulate the object. To be convincing, solid-seeming objects must do more than look solid; they must also *feel* solid. Cartoon animators often use techniques that mimic physical effects, such as inertia and friction, to reinforce the illusion of substance [38]. Cartoon objects start moving slowly and come to a gradual stop, they move in curves rather than along straight lines, and they stretch and squash to suggest interaction with their environment. Two research projects have taken these ideas and investigated how to incorporate them into traditional user interfaces: the Self programming environment and the Artkit user interface toolkit.

The Self programming environment [18] supports cartoon-style animation techniques. Chang and Ungar list three cartoon animation principles that apply to interface animation: the principles of solidity, exaggeration, and reinforcement. These principles are characterised as follows:

1. Characters and objects should seem *solid.*
2. *Exaggerating* the behaviour of user interface objects makes the user interface more engaging.
3. The interface should *reinforce* the illusion of reality.

Self's interface uses solid drawing, squash and stretch, motion blur, dissolves, anticipation, follow through, slow-in/slow-out, and arcs to strengthen the impression that programmers of Self programs are manipulating solid objects.

Artkit [31] offers tools to the user interface application programmer to incorporate a similar set of animation techniques into their applications. Artkit includes all the animation effects from Self with the exception of the dissolve technique, and it includes pacing functions to control timing aspects of the animation effects. These functions allow the programmer to map the speed of a graphical object along a curved path to a non-linear time function.

	Immersive	**Nonimmersive**
Games	Immersive 3D painting	Doom
Virtual Reality	Telerobotics	Molecular modeling
Visual Simulations	Flight simulations	Factory simulations

Table 2.1: Examples of different environments for games, virtual reality, and visual simulations

Artificial Environments

Games, virtual reality, and visual simulations are examples of interfaces where the animation itself is the total display portion of the user interface. Applications in these three areas can be described as providing either an immersive or a non-immersive artificial environment. In the case of an immersive environment, the vision stimulus from the outside world is replaced by artificial computer rendered images [50]. In contrast to this, a nonimmersive environment [56] has the user interact with a conventional graphics workstation using a monitor, keyboard and mouse. Table 2.1 shows some examples of the breakdown of applications versus environments.

Baecker and Small described four scenarios for using animation in virtual environments,

1. Animation to help in exploring complex environments, as in factory simulation applications.
2. Animation to help in exploring viewpoints to determine the relationship between an entity and its environment, as in molecular modeling applications.
3. Animation to help visualise how an object, structure, or system would look under various conditions of light and shade, as with immersive 3D painting.
4. Animation to examine alternative futures, to experience hypothetical environments visually, or to simulate activities too dangerous or difficult to view directly, as in Doom, flight simulation and tele-presence robtics applications.

The concept of a WIMP interface has little or no meaning in an immersive environment. As James Foley [25] asks, *"What would a text editor look like through an immersive virtual reality interface?"* Recent work has examined how to incorporate the concepts of text and buttons into immersive virtual reality interfaces [41, 54, 89], but the core of an immersive environment is that the visual and audio surroundings are the interface itself. The nonimmersive environments range from fully interactive games to data visualisation. The data visualisation end of this spectrum overlaps with the concepts of information visualisation. Interactive simulations [42] and games go beyond the scope of traditional interfaces, but these forms of applications can be used as guides in determining how animation can improve interfaces [22].

The three application domains described above rely on a tight coupling between the user and the application; the user should have a sense of "entering" into the application domain. In the case of a game such as Doom [32], the player responds quite quickly to stimulus from the screen and audio output [53]. The entertainment value comes from the user believing they are in the playing space. The playing space must react to the user input. This is even more the case for immersive virtual reality systems. When a person picks up an object in a virtual world, the object must react in a sensible way, such as following the grasped hand. A dropped object must come under the influence of gravity and drop to the ground. Tracked animations are required to supply this form of visual feedback.

2.4 Examples of Systems with Animated Interaction

This section examines five systems that incorporate animation in their interface. The Information Visualizer shows how 3D graphical animation is used. The TANGO system provides an example of algorithm animation for software visualisation.

Artkit is presented as a toolkit that allows an application programmer to incorporate tracked animations into their application. A description of some examples of animated help is also presented. Finally the Whizz system for building animated interactive applications is described.

2.4.1 Information Visualiser

The Information Visualiser, by George G. Robertson, Stuart K. Card, and Jock D. Mackinlay[17], explores a new user interface paradigm that has come into being through exploitation of a new generation of graphical workstations. These workstations can provide cost effective 3D graphical animation on the desktop. The problem domain addressed by the Information Visualiser project is that of applications that require the user to retrieve, store, manipulate, and understand large quantities of information. The Information Visualiser uses large workspaces, agents, real-time interaction, and visual abstraction to improve the access to large information spaces.

The *large workspace* is delivered to the user in the form of virtual workspaces [16, 29, 55] and denser visual information spaces. The virtual workspaces take the form of virtual desktops or *rooms*, that allow the user to organise their individual electronic work environment to suit their tasks. This concept has been embedded in many workstation window managers, for example "tvtwm - Tom's Virtual Tab Window Manager for the X Window System." [36] In the Information Visualiser, the denser visual information spaces are supported with user interfaces constructed with 3D graphical animations.

To help ease the cognitive load of the user, autonomous *agents* allow the offloading of tasks to concurrent processes on the user's workstation. These tasks include such aspects as searching, clustering, and design of information presentation. The 3D graphical animations, autonomous agents, and the user's task all present information to the user in real-time at different rates. A system that provides *real-time interaction* must provide "impedance matching" to guarantee that information is presented to the user at the proper rate. *Visual abstractions* are required in order to speed pattern detection of information from a global perspective.

The Information Visualiser provides a number of applications:

1. Cone Tree - hierarchical structures
2. Perspective Wall - linear structures
3. Data Sculpture - continuous data
4. Office Floor Plan - spatial data
5. Information Grid - unstructured data

The Cone Tree and Perspective Wall system are examined in detail.

Animation is central to the Information Visualiser's ability to present large quantities of data in an understandable form. Robertson, Card, and Mackinlay argue the need for smooth interactive animation because it can shift a user's processing of information from the cognitive to perceptual capacity, freeing cognitive processing capacity for application tasks. An example of this shift is shown in the Cone Tree. The Cone Tree is a vertical or horizontal tree structure of records in which the 2D layout of each tree and subtree is replaced by a 3D cone structure. Figure 2.6 shows the way a Cone Tree appears to a user. The Cone Tree system uses a fully rendered 3D model with perspective view, lighting, texturing, and depth cuing. The user selects a record with the cursor, and the Cone Tree associated with that record smoothly rotates to expose the information. The smooth animation of rotation allows the user to connect the new view with the old view; conversely a sudden change of view point would require the user to spend time reassimilating the new point of view of the object.

To support the visualisation of linear information structures, the Perspective Wall[40] integrates detailed and contextual views. An example of how the user

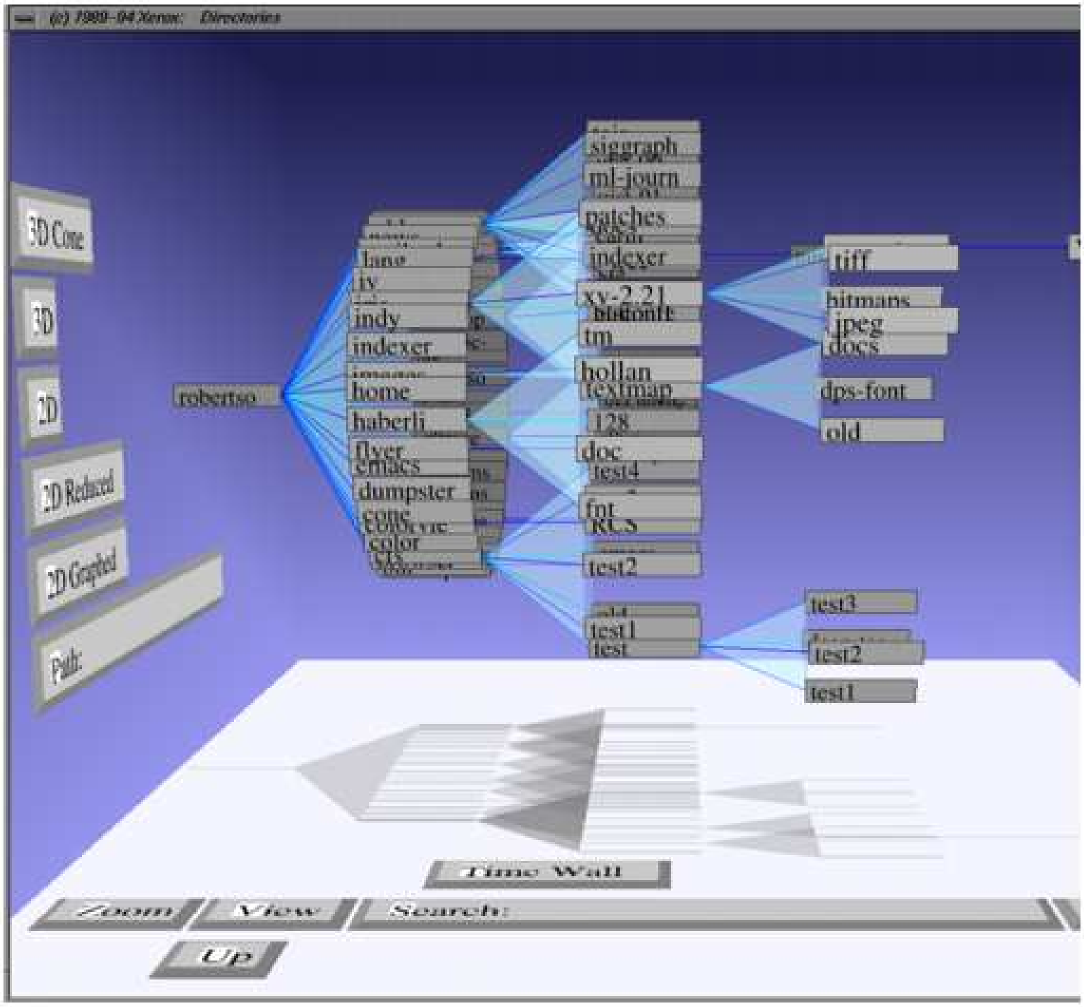


Figure 2.6: Illustration of a Cone Tree (reprinted with the permission of the authors.)

views the Perspective Wall is shown in Figure 2.7. As in the case of the Cone Tree system, the Perspective Wall system is a fully rendered 3D model with perspective view, lighting, texturing, and depth cuing. The user operates the interface by selecting an item. The system then moves the selected item to the centre panel with the aid of animation. The animation is smooth and is similar to a player piano scrolling the rolled music sheet to the centre view.

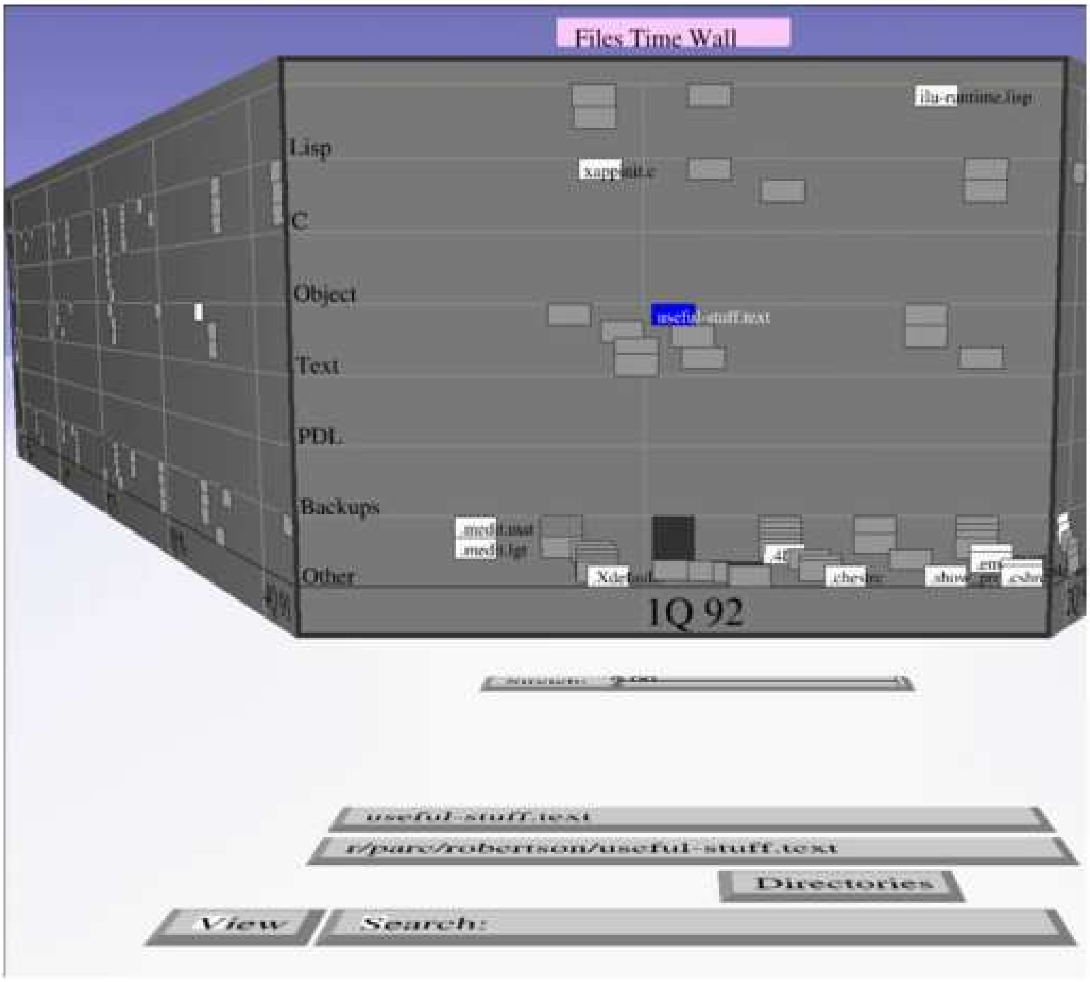


Figure 2.7: Illustration of a Perspective Wall (reprinted with the permission of the authors.)

2.4.2 TANGO

The *TANGO* system for algorithm animation is a successor to the BALSA system [13], the first major interactive software visualisation system [52]. The *TANGO* system was written by John Stasko [63]. The four primary goals of TANGO are:

1. to be a model for specifying and executing algorithm animation;
2. to be user-oriented;
3. to support smooth, continuous transitions and movements of graphical objects; and
4. to build upon a widely available teaching and research platform.

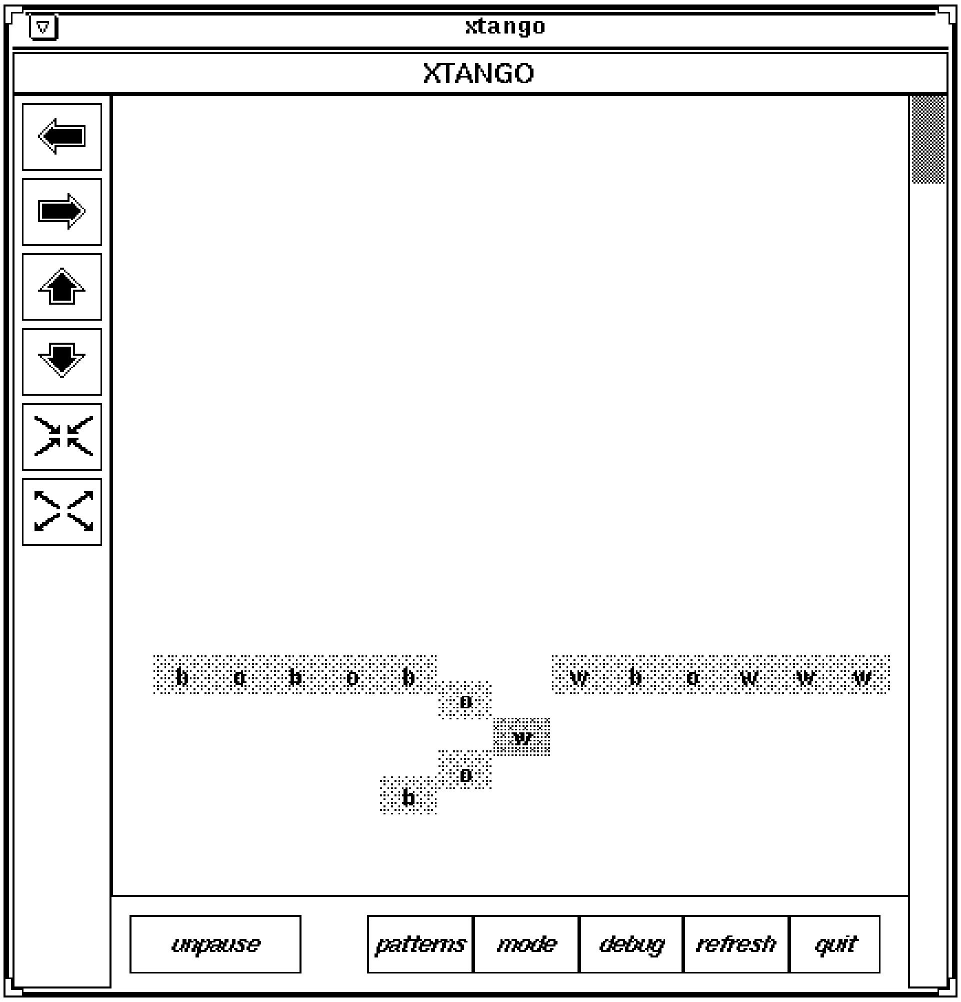


Figure 2.8: Example of a Xtango animation

John Stasko in his paper *A practical animation language for software development* [62] describes a language for specifying animations. This language describes algorithms, using locations on the screen, graphical images, paths that modify the image transitions, and transitions of images. Figure 2.8 shows an X-Window version [65] of TANGO, XTANGO, animating a version of the Boyer-Moore pattern matching algorithm.

Locations specify the 2D position of a *graphical image* in a window. *Graphical images* are defined by a collection of lines, circles, rectangles, and text. A graphical image may be composed of a collection of images, making a *composite image. Paths* define the change in position, size, colour, and visibility. An example of a path with

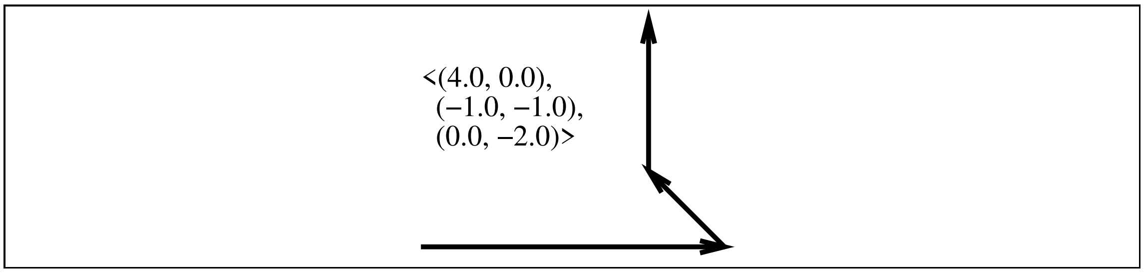


Figure 2.9: Example path for changing position

three different changes of position is shown in Figure 2.9. The current system can specify paths composed of straight lines, clockwise arcs, and anticlockwise arcs. The example path first moves four units to the right, then one unit up and one unit to the left, and finally two units up. *Transitions* combine the graphical image, path, and effect to form an action. Some common effects, or transition types, include move, resize, fill, colour, and alter visibility. These simple transitions can be combined to form a *composition*, which allows for concurrent execution of transitions. For example, a graphical object could simultaneously translate to right of the screen, grow to twice its original size, and change colour from blue to red.

2.4.3 Artkit

Hudson and Stasko implemented an extension to the Artkit toolkit to provide *flexible, robust,* and *reusable* animation abstractions. The extension provides a choice of animation techniques such as simple motion blurring, squash and stretch, arcing trajectories, anticipation, follow through, and slow-in/slow-out transitions. Central to this work are tools for the application programmer to deal with time for the control of the animation. Artkit provides powerful tools in its toolkit to model time in the Unix and X windows domains.

The central animation abstraction is the *transition.* This abstraction is an extension to the *transitions* used by *TANGO*. Transitions define changes over time, for example translations.

An *interface object* is the graphical object that is animated. A *time interval* is the time over which the transition takes place. A *trajectory* is defined as a curve and a pacing function. A *curve* defines the path an animation takes on the screen, and a *pacing* function controls the timing aspect of the animation effect. The curve can control not only the position of the graphical object, but other aspects such as the scale of the graphical object.

The abstraction of time is a time interval model, similar to that used in temporal logic systems [3]. The starting time point of an animation can be defined as a point in time, a future point in time relative to the start of another transition, or a point relative to the end of another transition. The end time point of an animation can be defined as a point in time, a future point in time, or a fixed duration from the start point. The rate of change can be specified by a pacing function in terms of a spline, that allows the application programmer to define a slow-in/slow-out effect.

2.4.4 Animated Help and Icons

Help systems in an application provide online documentation of the operation of the system. Animation aims to improve existing help systems by providing animated scenarios for insight into the operation of the user interface. These animations can be demonstrations of how to perform an operation, or a simple movie that improves the understanding of the intended purpose of an icon. For example, if a user requests a demonstration of how to copy a graphical object in an editor, the application could open a new demonstration area window, then demonstrate the correct sequence of mouse actions:

1. The “create new object” menu item is activated.
2. A new object is created, by a sequence of mouse actions.
3. The “select object” menu item is activated.
4. The new object is selected by clicking on it.

5. The "copy object" menu item is activated.
6. The location of the new object is defined by a mouse click.
7. The new object is drawn in the new location.

During the animation the help system would move the cursor automatically, and menus would be pulled down, highlighted, and activated. The system would present the illusion that the application was performing the task.

The implementation of animated demonstrations takes the form of a set of predefined images or a set of predefined events to drive the application [10, 67, 68, 73]. In the first form of animated demonstrations, the help system plays a prerecorded movie of predefined images demonstrating how the system behaves. In the second form, the help system takes control of the application and supplies a set of inputs that emulate the actions of a user performing a required task. In both cases, the user is led to believe that the application is under the control of the help system and operations appear as if a human initiated them.

Animated icons [4, 7] provide more information about the nature and use of the operations attached to the icons by providing short canned movies on the face of the icon. Instead of having a static image for an icon, the image is dynamic. Figure 2.10 is an example of an icon button to invoke a scaling operation; the sequence of superimposed images depicts the scaling of an object. The images in the figure are single frames in the canned movie. However, some investigators have noted problems with the activation of the movies on the animated icons. For example, one way the canned movie is activated is to place the cursor over the icon. This has the advantage of only animating the selected icon, so that the screen is not filled with distracting motion. However, the cursor hides the image on the icon; users tend to move the cursor off the icon to see the movie, which causes the movies to stop.

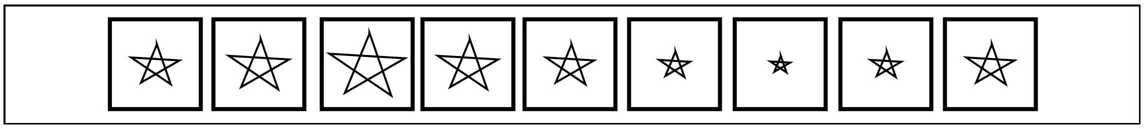


Figure 2.10: An example of an animated icon for a scaling operation

2.4.5 Whizz

Whizz [19, 20] is a system developed by Stéphane Chatty for building animated interactive applications. Chatty breaks interface behaviour into three categories: time-driven animation, application-driven animation, and user-driven animation. The stream-event model, in the Whizz system, was developed to deal with the problems comprising discrete versus continuous information flow and the combination of the above three interface behaviours. The objects of the system have defined input and output *plugs.* Each plug is typed for discrete or continuous information. An object may handle both forms of information.

The Whizz system uses a musical metaphor for defining a base set of objects: dancers, instruments, rhythms, tempos, and notes. Dancers produce visual effects and are generally graphical objects. Notes contain attribute information such as points, colours, or numbers. Dancers can respond to notes by changing, for example, shape or colour. Instruments emit notes as defined by their characteristics and state. For example, an instrument could emit a set of notes defining the trajectory of a moving object, a dancer. The tempo sets the frame rate of the animation, and the rhythm attaches the tempo information to an instrument.

An example given in the paper *Defining the dynamic behaviour of animated interfaces* [19] is of a speedometer connected to an integer generator. Figure 2.11 shows how the components are connected. The Active Value component emits integer notes, which are fed to the Integer-to-Point instrument. This instrument performs the conversion to position notes, and feeds these to the speedometer dancer. The speedometer dancer updates the dial as the new notes arrive.

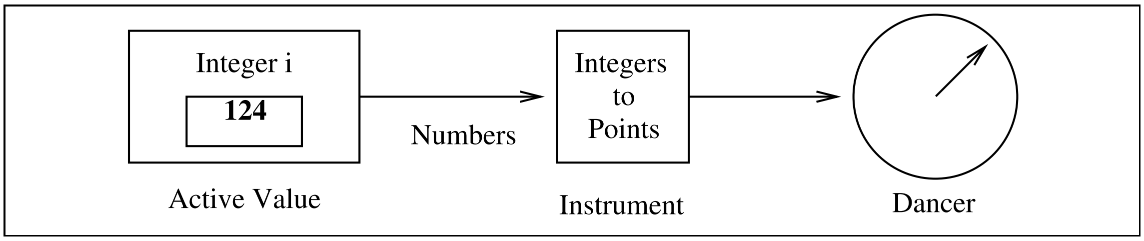


Figure 2.11: An example using components from Whizz to build a speedometer

Whizz has been used to develop an experimental system for air-traffic management. The experimental system displays intersections of flight paths. The user can grab an airplane and move it along its flight path. The system simultaneously updates the positions of the other planes on the screen to show where all planes are at the same moment. This allows the operator to predict how close airplanes will come to one another. This example is a mixture of application-driven animation and user-driven animation.

2.5 Summary

Animation has been applied to different areas of graphical user interfaces. There are a number of general animation effects and techniques used by animators across all forms of animations; these include the following: such as anticipation, squash, stretch, motion blurring, exaggeration, slow-in/slow-out, arcs, secondary action, follow through action, overlapping action, and moving hold. Computers have been used to assist animators by automatically generating images, assisting with interpolation between keyframes, and controlling the motion of graphical objects.

Animation can be incorporated into interfaces in two ways: firstly, as autonomous animations that improve information legibility, smoothing of transitions, and animated online help systems, and secondly, as tracked animations that provide directly manipulated graphical objects with substance and visual feedback in artificial environments where the use of animation dominates the interface.

Chapter 3

Applying Animation in Graphical User Interfaces

This chapter presents a set of animation effects targeted at two main areas of the user interface: widget components and graphical object manipulation. To make animation effects for user interfaces useful and pleasing to the user, I have developed effects based on traditional cartoon animation principles.

In the spirit of Chang and Ungar's principles for using cartoon animation in user interfaces (see Section 2.3.2), I have defined four principles for animating direct manipulation operations [75]:

1. **The principle of attachment** states that the objects being manipulated should at all times remain attached to the pointer.
2. **The principle of reluctance** states that objects should, in general, seem reluctant to change.
3. **The principle of smoothness** states that objects must change in a continuous fashion.
4. **The principle of anticipation** states that the result of a user's action must be obvious at all times.

The principle of attachment maintains the impression that the user is always in control of the action. The principle of reluctance reinforces the illusion of substance by suggesting that changing an object requires effort on the part of the user. The principle of smoothness reduces cognitive load by removing large and unexpected changes in visual information presented to the user. Finally, the principle of anticipation supplies enough information to allow users to confidently apply manipulation operations to visual data.

The chapter starts with a presentation of an exercise to build a videotape prototype of animated interfaces and discussing the results of this exercise. The remainder of the chapter is broken down into three sections describing three animation effects: *smoothing changes in the interface*, *giving manipulated objects substance*, and *anticipating the result of a manipulation.*

3.1 Animation Prototypes

To understand the type of effects needed to support animation in a user interface, I first investigated what portions of the user interface could be animated. Once the set of interface components to be animated is determined, low level software toolkit support can then be incorporated into an existing user interface toolkit. To visualise a set of possible animated interface components, I created animated sequences for a range of widget and direct manipulation tasks. A prime motivation for prototyping with conventional animation techniques was the ability to apply traditional animation techniques to different portions of graphical user interfaces, without being limited by the performance of current hardware platforms or implementation techniques. Without these limitations, I hoped to remove preconceived notions about how an interface should *look-and-feel.* Two different animation techniques were used: full 3D rendered graphical models and frame-by-frame images.

The full 3D rendered graphical model prototypes were rendered using the *imag-*

ine [59] animation package on an Amiga 3000 machine. The animations contained 3D models of interface objects such as menus, buttons, cursors, and pages, which were placed in short animation sequences that showed how a user might interact with the objects. The animations were fully rendered with lighting and shadows. The exercise quickly showed that even simple animations are difficult and time consuming to produce, and that slight defects can destroy the entire effect. For example in one video, a 3D cursor squashes a cube menu to activate the cube to rotate. In an early animation, the cube started to squash just one frame ($\frac{1}{30}sec$) before the cursor came in contact with the cube. This small defect destroyed the illusion of the cursor pushing on the cube.

The frame-by-frame animations used were constructed with idraw, a standard drawing tool, that is part of the InterViews package. They contained idraw-generated sequences of individual frames, stored as bitmapped images. The images were antialiased to help maintain detail in finely detailed graphics, such as reduced scale versions of the text characters that are difficult to read. Animation sequences were produced by reading all the images for a sequence into memory, then replaying the stored images at video rates. The final videos were produced on a DEC Alpha 3000 with a J-300 Multimedia card [21]. The J-300 card produces a PAL or NTSC composite video signal with gen-locking, which captures the video signal onto videotape in real-time.

The two aspects of graphical interfaces targeted for particular attention were predefined components (widgets) used by applications for common tasks such as issuing commands and setting state, and graphical feedback that accompanies direct manipulation operations such as translating or resizing application objects.

The videos experimented with a set of animated widgets that apply different animation techniques to common tasks. For the 2D animations, there were animations that showed translation, snap and drag, and menus. The translation examples

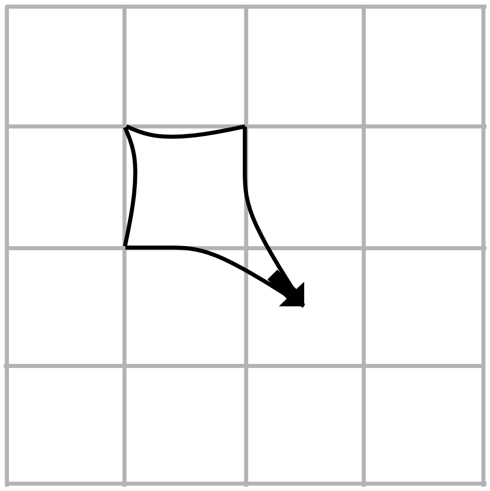


Figure 3.1: A frame from the animated snap and drag video

depicted different squash and stretch techniques and a set of motion blur effects. The snap and drag example demonstrated the effectiveness of squash and stretch in showing constraints to a user; Figure 3.1 shows one frame from the animation video. A set of sliding and stretching effects were used to produce an animation that depicts a range of pop up and pull down effects to expose the menu window. Other examples included 3D menus that appeared to swing out of view like a door on a hinge, and a 3D cube menu that was depicted with a 3D cursor. When activated, the cube menu was animated to rotate, squash and stretch. In the video, the 3D cursor "flys" through space on a 3D trajectory.

3.2 Smoothing Changes in the Interface

Existing standard widgets, such as buttons and menus, are carefully designed with a consistent and aesthetically pleasing *static* appearance, but little thought has so far been given to their *dynamic* appearance. Animation can help smooth abrupt changes in appearance by more gradually changing the screen's appearance from the old state to the new. These forms of smoothing animations have been investigated by a number of researchers. Whizz [19, 20] provides tools for building animated interactive applications. The Self animated widgets [18] provide menus that grow, dialog boxes that dissolve, arrows that grow and shrink smoothly, and objects that can be "hit" and bounce about from moving arrows. In the "buttonfly" 3D graphical menu system for Silicon Graphics workstations [72], activating a submenu button brings up a new menu by smoothly flipping the button over, exposing a new set of buttons.

This section describes four animation effects for widgets that I have incorporated into a standard graphical user interface toolkit to smooth the interaction: menus that open smoothly, buttons that change geometrically, buttons based on flip book animations, and 3D rotating palette menus.

3.2.1 Menus that Open Smoothly

The operation of an animated pull-down menu is similar to that of a standard static pull-down menu. The key difference is in how the menu is exposed. With the static menu, the entire menu appears as fast as the display can draw it; the effect is as if the menu suddenly appears at full size. With the animated version, the menu expands from zero to full size over a short but noticeable interval of time. Figure 3.2 demonstrates the animated version of a pull-down menu. As the menu expands, its contents scale smoothly to keep the window full. Because a user requires a small but finite amount of time to perceive and understand even an instantaneous change, animating the transition makes it seem smoother without increasing the overall duration of the interaction [17].

The duration of the expansion and the number of frames in the animation sequence are under program control. For informal testing, the animations consisted of 10 frames over a half-second duration. This produced a smooth and satisfying effect. Informal testing showed that if the animation duration was too long, the act of using the menus felt sluggish and seemed to slow the user down. If it was

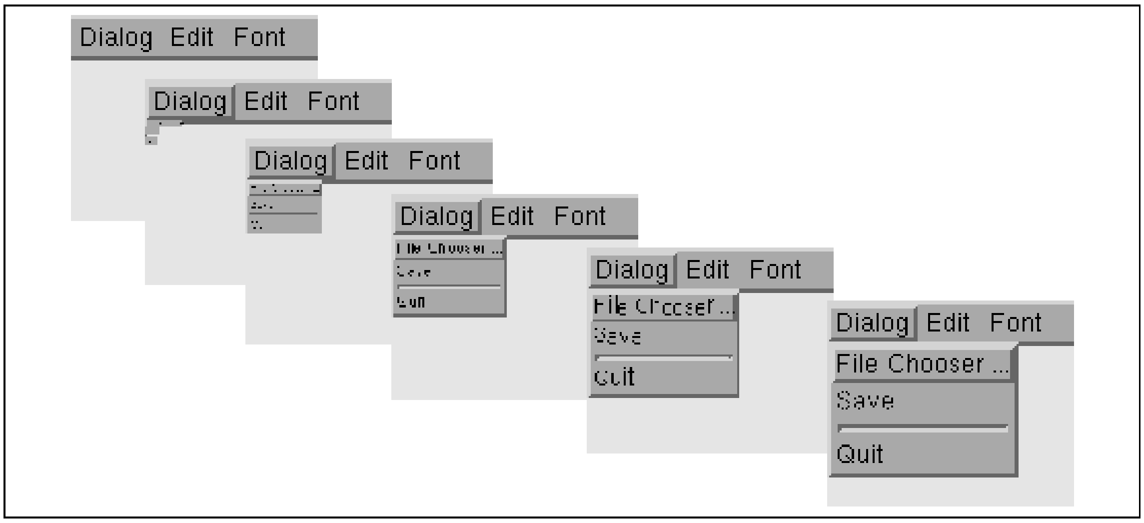


Figure 3.2: An animated menu

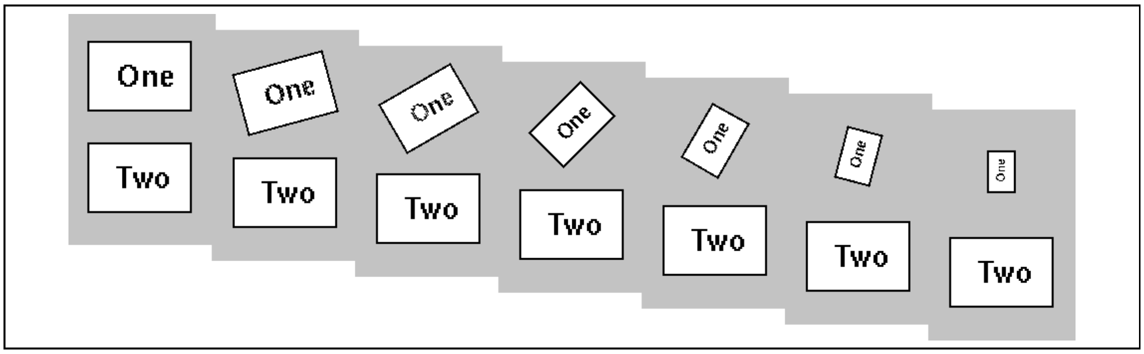


Figure 3.3: A palette button that resizes and rotates when pushed

too short, the effect of the animation was largely lost.

3.2.2 Buttons that Change Geometrically

Many of the button animation effects I have experimented with can be specified using a basic set of graphical transformations: scaling, translating and rotating. For example, the animated push buttons shrink to about ninety percent of their original size when pushed and return to full size when released. The effect suggests that the push button is "bouncing" away from the user in response to the push.

The palette button uses a more complicated transformation. In the animation, the button is reduced to forty percent of its original size and rotated 90 degrees in an anticlockwise direction. The effect is as if the button twists and recedes when

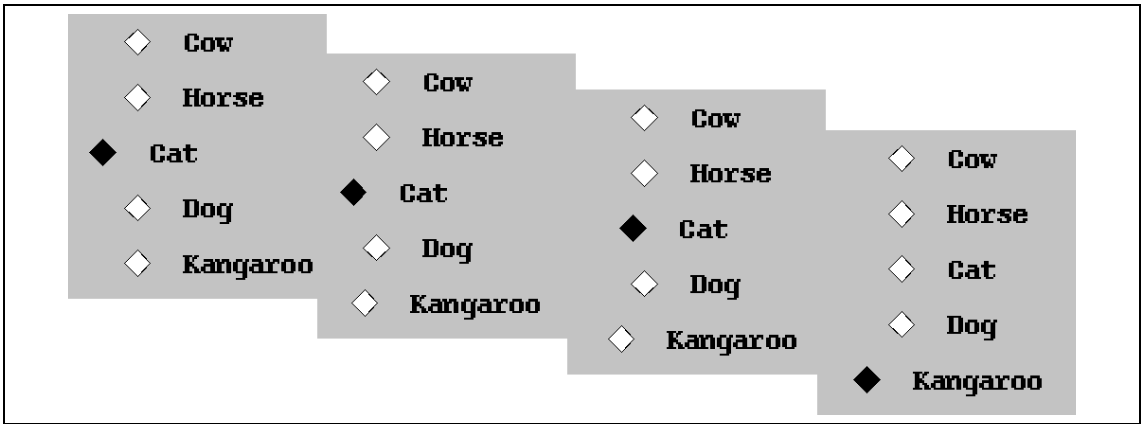


Figure 3.4: Animated radio buttons

pushed. A frame by frame example of a palette button animation is shown in Figure 3.3.

Two strategies are used to highlight a single button chosen from a list of options with radio and check box style buttons. In the operation of a radio button, only one button can be selected at a time. When a new button is chosen, the previous one is deselected. Figure 3.4 shows how the transition from the selected "Cat" button is smoothed to the state where the "Kangaroo" button is selected. During the animation, the "Cat" button slides to the right and the "Kangaroo" button to the left.

Similarly, the check boxes perform simple translations to highlight which button is selected, but in the case of check boxes, multiple buttons can be selected at once. Figure 3.5 shows the frame by frame animation of the transposition of the label "Column D" and box icon of the check box button. The motion of the translation smoothes the transition state, and the transposition of the label and box icon further highlights that the button is selected, although the list of labels is potentially more difficult to read.

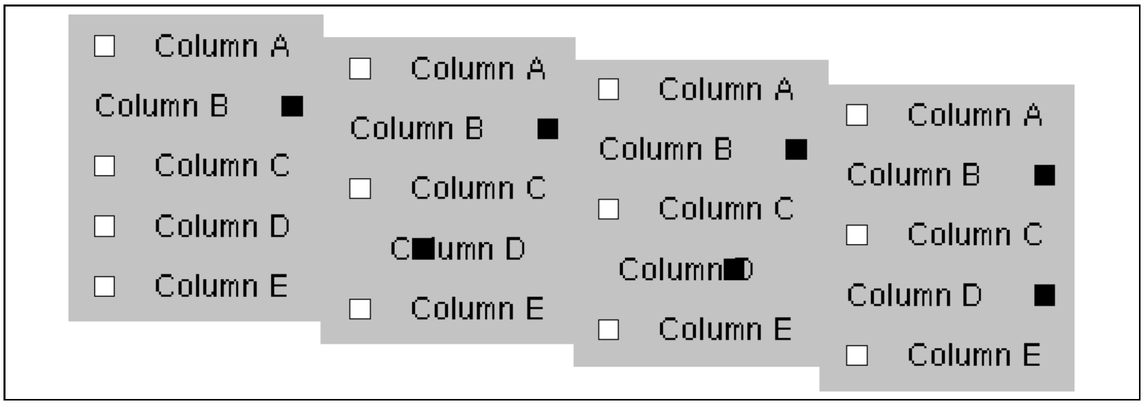


Figure 3.5: Animated check boxes

Figure 3.6: Button label animation

3.2.3 Buttons Based on Flip Book Animations

There are some animations that cannot be represented by transforming graphical objects. In cases such as these, frames of the animations are individually created, similar to the techniques used in old paper flip books.

Flip book style animation is more general than transformation style animation, since each frame can be individually hand crafted. A number of flip book animations have been constructed for buttons that change border thickness, colour, or the label text. Figure 3.6 depicts a set of frames from an animation, where the label on the button marked “White” is replaced by the label “Black” one letter at a time.

3.2.4 Rotating Palette Menus

The palette menu relies on a simple orthographically projected 3D effect to present more choices without increasing the display area required by traditional menu sys-

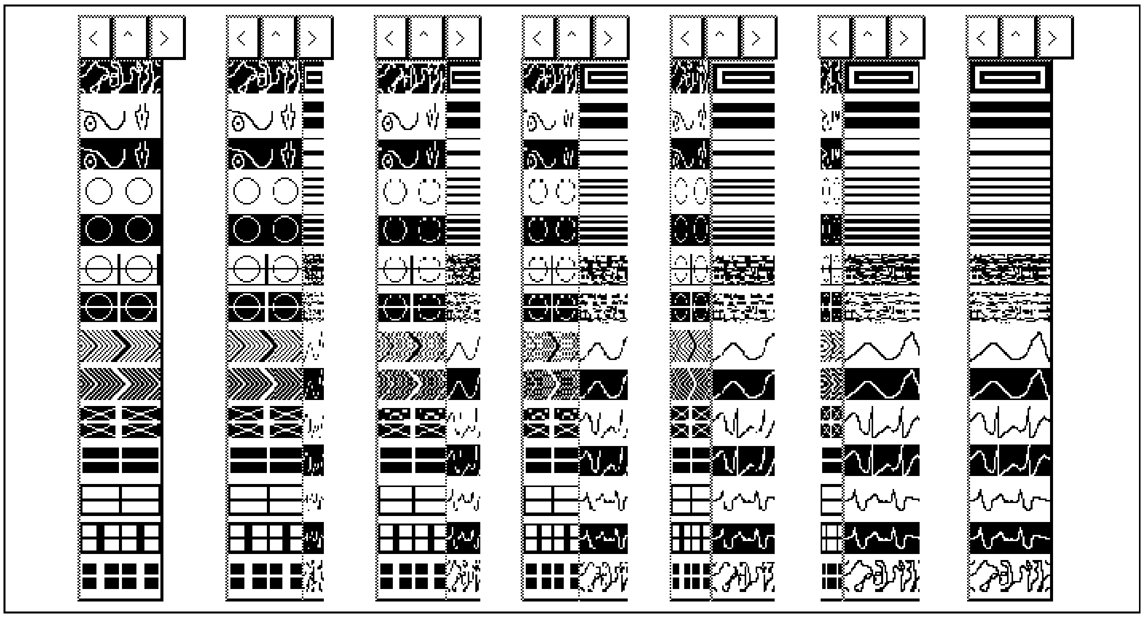


Figure 3.7: Palette menu animation

tems. The technique achieves a suitable 3D "feel" without the extra computational cost of full 3D perspective projections.

The approach is to group the choices in panels that are arranged as if on the sides of a prism. Figure 3.7 shows an example of a palette menu and an animation to bring a new panel into view. The menu includes 56 patterns, grouped into four panels of fourteen. Normally, only one side of the prism is visible at any one time, as demonstrated in the left hand frame of the figure. If the desired pattern is not visible, the buttons on the top of the menu labelled with "<" and ">" may be used to bring other hidden panels to view by smoothly rotating the menu around its vertical axis. In the figure, the menu rotates towards the left to bring to view the panel that was previously hidden on the right.

The approach of using several sets of choices and allowing users to switch between them is not new. A common example occurs in many graphics editors that use this approach to fit large palettes of choices into the narrow spaces beside or below the main drawing window. However, searching through the panels for a particular choice is often clumsy because the user has no cues about which panel to select.

The palette menu is based on the ideas that underlie the Cone Tree [57]. The concept is that smooth changes with a 3D rotation helps a user keep track of choices that have temporarily been rotated out of view. In informal use of the palette menu, the approach seems promising.

3.3 Giving Manipulated Objects Substance

When moving objects in the Macintosh Finder and many other applications that use a direct manipulation interface, a simple outline of the object follows the pointer around the screen. For some movement tasks, this technique conveys the necessary feedback information. It does not, however, attempt to convince users that they are moving objects. Instead, the impression is more like moving a surrogate and having the object follow when the action is complete. More ambitious animated feedback can make the interaction more convincing.

For interface objects that move spontaneously or along predefined paths, the application of animation techniques is straightforward. Chang and Ungar [18] describe some excellent examples in the Self programming environment, which uses cartoon style animation techniques to strengthen the impression that programmers are manipulating solid objects. But in the context of direct manipulation, where users are in control of object movements, animated interface designers are faced with a dilemma: the need for immediate and accurate response to user actions is at odds with the need for objects to have realistic-seeming behaviour. If a user tries to drag an object rapidly across the screen, for example, how should the interface represent the inertia of this object resisting its movement? Or if a user tries to move an object that is fixed in position, how should the interface suggest the restraint?

The approach I have taken is to consider how a cartoon animator would depict such behaviour. For example, to suggest movement caused by someone dragging an

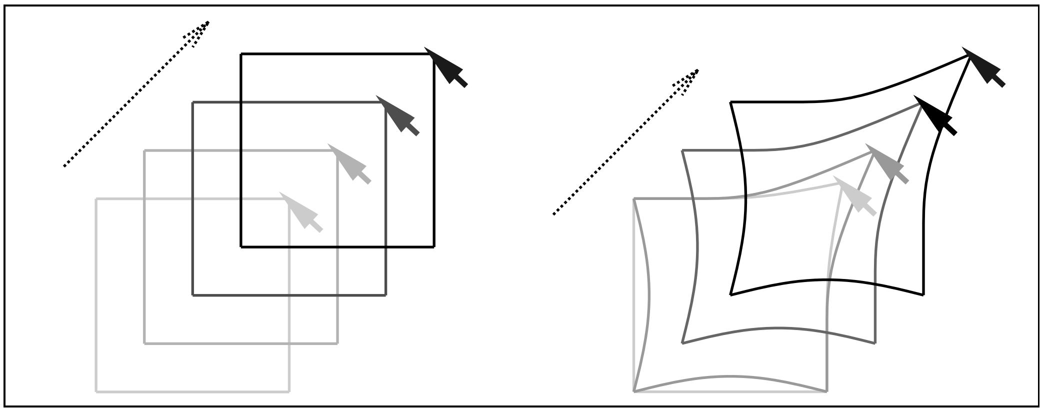


Figure 3.8: Animating a move operation

object, an animator could show the object distorted in the direction of the pull, as if the object is reluctant to change. To suggest an attempt to move a fixed object, the animator could show the object leaning in the direction of the pull while its base stays fixed in place. With techniques like these, an application can create the illusion of a greater sense of substance for its objects while still allowing users to feel that they are in control.

This section looks at animation effects for the following four direct manipulation operations: *translating*, *scaling*, *rotating*, and *shape editing*. All of these operations change the visual characteristics of the object under the user's control. Adding animation to the visual feedback allows the application to display more information about the interaction. In addition to the current state of the objects, the animation can also suggest the direction and rate of change caused by the user's actions. This extra information gives the user a better understanding of the results of an action on an object. Appendix A describes how to obtain some short video clips of these effects.

3.3.1 Translation

Figure 3.8 shows a direct manipulation move operation on a square. These static images cannot fully convey the dynamic feel of the interaction; several successive

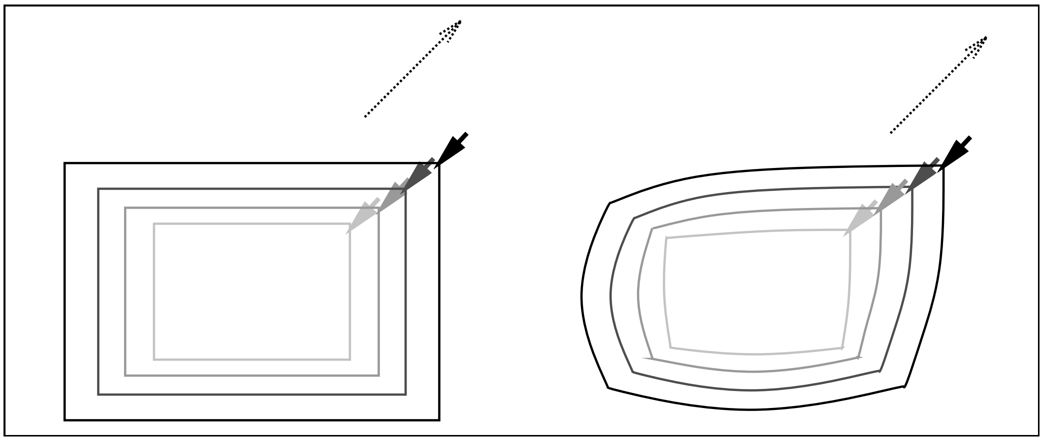


Figure 3.9: Animating a scaling operation

frames from the interaction have been superimposed to suggest the effect. The shading of frames is intended to imply the passage of time, with the most recent frame being the darkest. The diagram on the left side of the figure shows the move operation without the animation effect; the square merely moves to follow the mouse. The diagram on the right side of the figure shows the move operation with animation; the corner of the square stays attached to the mouse point (the principle of attachment) while the bulk of the object lags slightly behind (the principle of reluctance.) This animation gives the effect of manipulating a heavy "rubbery" object that distorts as it is pushed and pulled. Although the effect does not correspond exactly to a physical model, this simple algorithm gives the impression that the shape is made of elastic material, with weights attached to the vertices, causing them to lag behind the movement.

3.3.2 Scaling and Rotating

The animation effect for translation is also effective when used in conjunction with other common direct manipulation operations. For example, Figure 3.9 illustrates an operation to make an object larger by scaling. The diagram on the left side of the figure is an unanimated version of the operation and the diagram on the right is an animated version. As in the translation animation effect example, the part

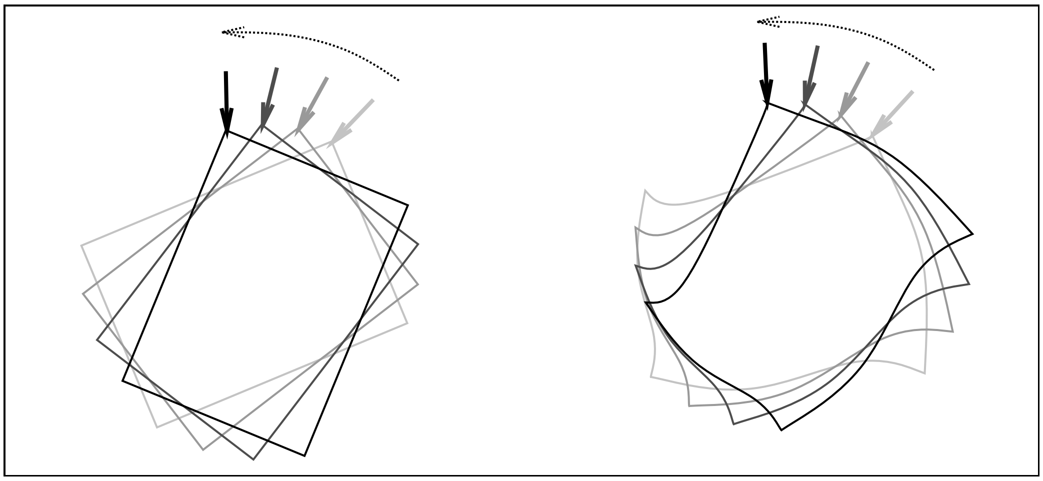


Figure 3.10: Animating a rotate operation

of the object that is "grasped" is controlled by the mouse, while the bulk of the object lags behind.

Figure 3.10 shows an operation to rotate an object anticlockwise about its centre. An unanimated version of the operation is shown on the left diagram of the figure and an animated one on the right. Once again the animation effect is to have the object lag behind so as to give the impression that the object has inertia.

3.3.3 Shape Editing

Operations that change an object's shape can also benefit from animation effects. In this section, animation effects for two simple kinds of shape-editing operations are suggested: moving a single vertex of a polygon and changing the size of a rectangle.

Moving a vertex entails changing the two adjacent sides of the polygon. Animation of this operation is achieved by distorting the sides that are changing. Figure 3.11 demonstrates the effect of moving the upper right vertex of a square. The left diagram shows an unanimated version in which the vertex is moved from the original position to the final position (the black polygon has the vertex in the final position.) The right diagram shows the animated version. As for all these an-

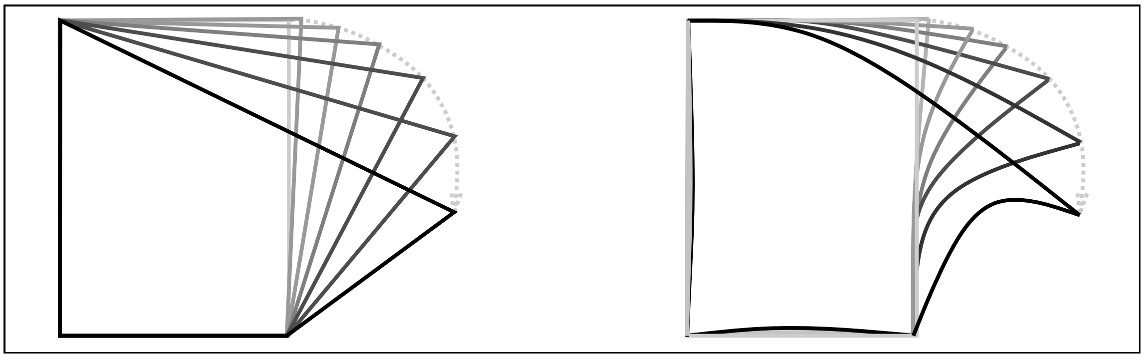


Figure 3.11: Animating a vertex editing operation

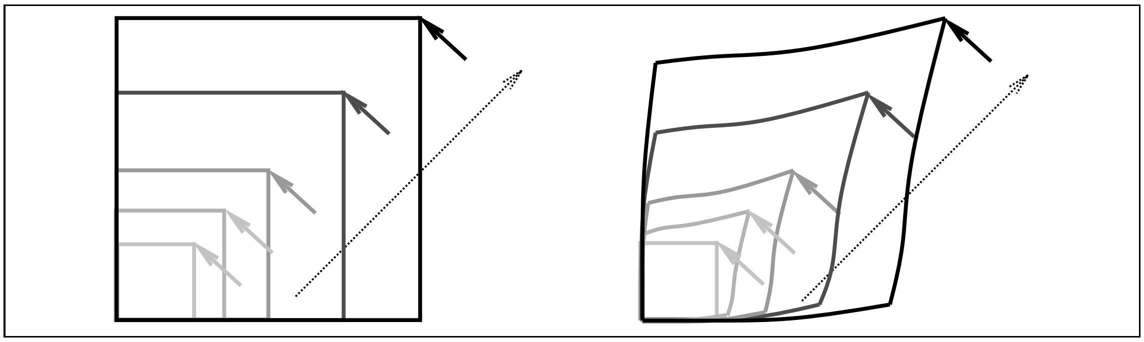


Figure 3.12: Animating a stretching operation

imations, the magnitude of the effect depends on how fast the action is performed. Rapid movement causes more distortion, as it must do if the effect is to suggest a "real" object.

Moving a single vertex affects the object only locally, but as shown in Figure 3.11 the bulk of the object distorts slightly to indicate the "stress" it is suffering. Other shape editing operations have a greater nonlocal effect on the object and hence offer more opportunities for animation. For example, the operation to resize a rectangle by stretching it changes its width and height while still keeping it rectangular. This operation affects three vertices and all sides of the object. The regular nature of rectangles, however, allows for an extension of the basic animation described in the previous sections.

Figures 3.12 and 3.13 show the animation of rectangle resize operations. The left diagram in both figures shows a nonanimated version of the rectangle growing and shrinking. The right sides of the figures show animated versions. Note how the

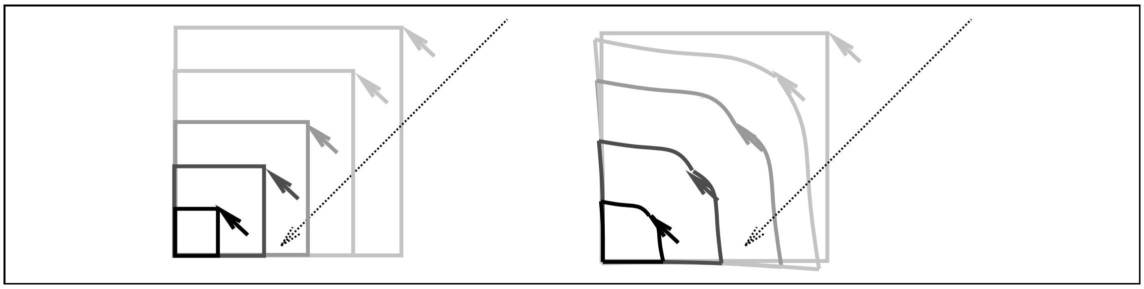


Figure 3.13: Animating a shrinking operation

vertices adjacent to the grasped vertex also participate in the action. When the rectangle is growing in size, they are drawn in slightly; when it is shrinking, they are pushed out. These animations, which correspond to the "stretch and squash" principle of cartoon animation, suggest the intuitive idea of the conservation of mass; if you press on an elastic object at one place, it will bulge out somewhere else. The overall effect conveys more strongly the idea that the object's natural shape, a rectangle, is not being changed by the editing operation.

3.4 Anticipating the Result of a Manipulation

The concept of anticipation is fundamental to the animation world, as it is for static drawings. Imagine how a cartoon artist might depict a young boy fighting to raise a mouthful of food where the plate of food has a mind of its own and refuses to move. The boy might pull the fork to his mouth, but the drawing would lead the viewer to believe that the fork would snap suddenly back to the plate if released. To help the viewer visualise the forces within the scene, the boy's arm would be bent to indicate his struggle, and the food would look stretched, indicating its tension. The effects combine to provide the viewer with a sense of anticipation as to the outcome of letting go of the fork. Similar effects can provide additional information and compelling visual cues to enhance graphical user interfaces.

Anticipation can be used to enhance operations that represent forms of geometric constraints outside the user's control. For example, if an object is constrained to one fixed location, animation can provide feedback to indicate that the object cannot be moved or altered by the user. When a user tries to move the pinned object, the application is attempting to keep it stationary. Animation can be used to suggest the resulting action when the user releases the pinned object. This section presents the animation effects to support visualisation of the constraints inherent in the pinning, snap and drag, and indirect manipulation operations.

3.4.1 Pinning

Consider how an interface might portray an attempt to move an object that is fixed in place – that is, the object is pinned. One response to this attempt might simply be to prevent the object from following the mouse. However, this lack of visible feedback might be misinterpreted as the result of a failure to “grasp” the object correctly. A user might make several attempts at the operation before realising the true cause of the lack of response. Another strategy might be to allow the object to follow the mouse, but then to snap it back to its original place when released. This approach avoids the problem with lack of feedback, but can lead to surprises when a carefully placed object suddenly jumps back to a previous position. This pinning effect supports the principle of anticipation.

Figure 3.14 shows an animation effect that avoids both problems. As the user attempts to drag the pinned object, the grasped point stays attached to the mouse but the bulk of the object stays fixed. The effect is as if the user is pulling on a corner of an object that is anchored in place. The feedback provides extra information; it makes it clear that the user is attempting to move the object, but that the attempt will not succeed. When the grasped point is released, the object will spring back to its original shape.

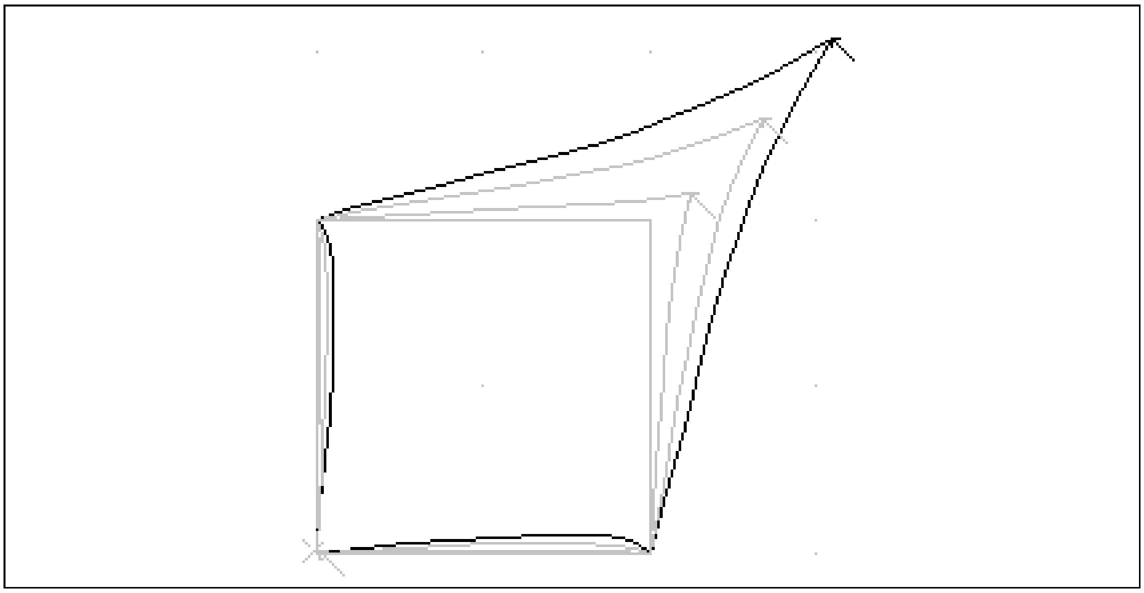


Figure 3.14: Animating an attempt to move a pinned object

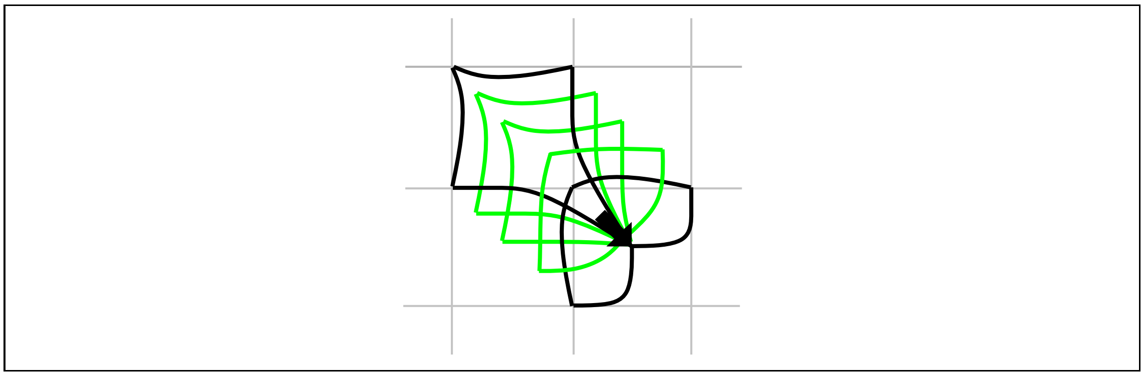


Figure 3.15: Animating a move under the influence of “gravity”

3.4.2 Snap and Drag

Snapping is a technique commonly used as an aid for accurate positioning in direct manipulation systems. Key features on the object being manipulated are snapped to nearby “hot spots” such as other objects or regularly spaced grids. Graphical feedback in systems that use snapping is difficult to implement without animation because of the tension between the position constraint implied by the snapping action and the need to accurately track user input. One possibility is to prevent the object from moving at all until it is dragged sufficiently far from a grid point, then to have it suddenly snap to the next point. However, with this scheme users may once again be uncertain whether the object has been "grasped". Another possibility is to allow the object to move freely, but to make it snap to the nearest grid point when let go. This scheme, though, may lead to surprises when an object unexpectedly jumps when released.

The use of the pinning animation effect enables both of these pitfalls to be avoided. For example, Figure 3.15 shows the effect of moving an object under the influence of a gravity field that causes the object to snap from the upper left grid position to the lower right position. As the user pulls the object away from a grid point, the grasped corner stretches while the remaining vertices of the object stay fixed. When pulled beyond a certain distance, the gravity suddenly "lets go" and the object snaps to the next grid point. The figure also shows a simple animation to smooth the transition from one grid position to the next grid position. Other effects such as motion blurring and follow through could be added to the animation. The feedback makes it clear what the current state of the interaction is, what the user must do to achieve the desired result, and what will happen if the object is released at any time.

3.4.3 Indirect Manipulation

The visual changes caused by direct manipulation are easy for users to understand because the user focuses directly on the affected object and tracks the change as it occurs. But for changes caused by indirect manipulation, where the user's attention is focused elsewhere and the change is instantaneous, the effect can be confusing.

Sudden visual changes can be disruptive because the user must subconsciously digest the new state of the screen; it takes a brief but significant moment before the new state is assimilated and the user can proceed with another task. For example, Figure 3.16 shows an operation where objects have been rearranged so that they abut left-to-right. By inspection, it is not obvious which objects in

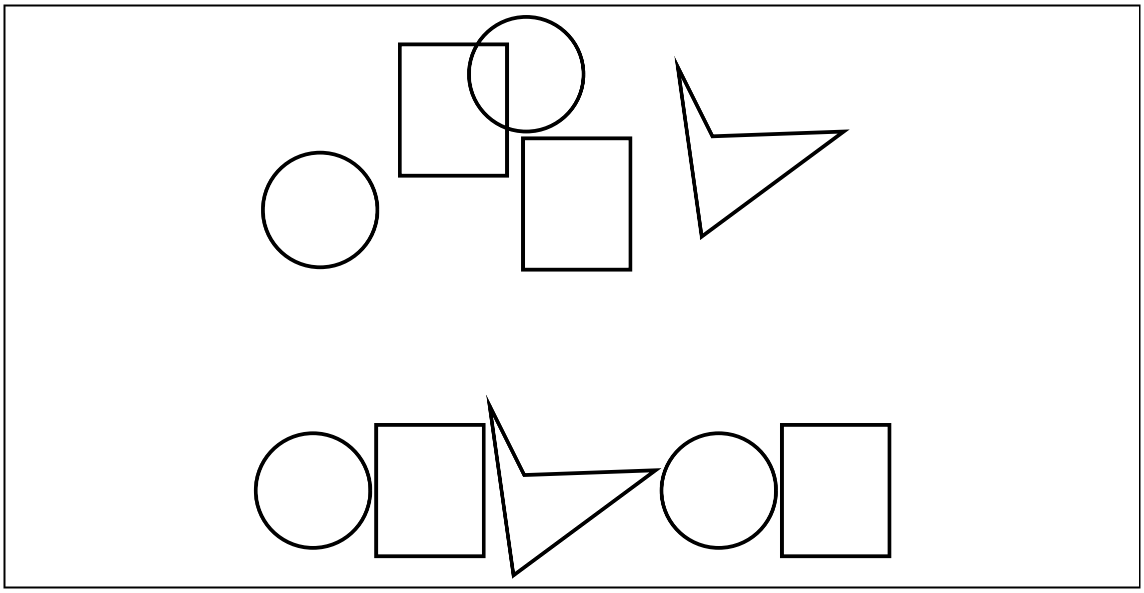


Figure 3.16: Before and after an alignment operation

the initial configuration (the top diagram) correspond to the objects in the final configuration (the bottom diagram.) If the screen instantaneously changes from one configuration to the other, the user may well be confused. Moreover, if the initial and final state of the screen are similar it may not be obvious just what has changed or how the change came about. This problem is exhibited in an operation that rotates a square through a quarter turn or flips it about an axis of symmetry.

The approach I have taken to smooth a change caused by indirect manipulation is to consider what the operation would look like if it was carried out by direct manipulation [80], albeit less quickly and precisely. For example, if an indirect manipulation causes an object to move to align with another object, then the effect portrays the change as if the object was dragged by one of its corners. As with direct manipulation, the object is distorted to suggest that it is slightly rubbery and is somewhat reluctant to change. A similar idea was used by Myers *et al.* [43]. They employed animated motion to smooth the transition between geometric states of the graphical objects. A key difference in my work is the addition of distortion of the graphical objects to add extra information for the user.

Figure 3.17 shows a situation where the square is moving to the left so that it

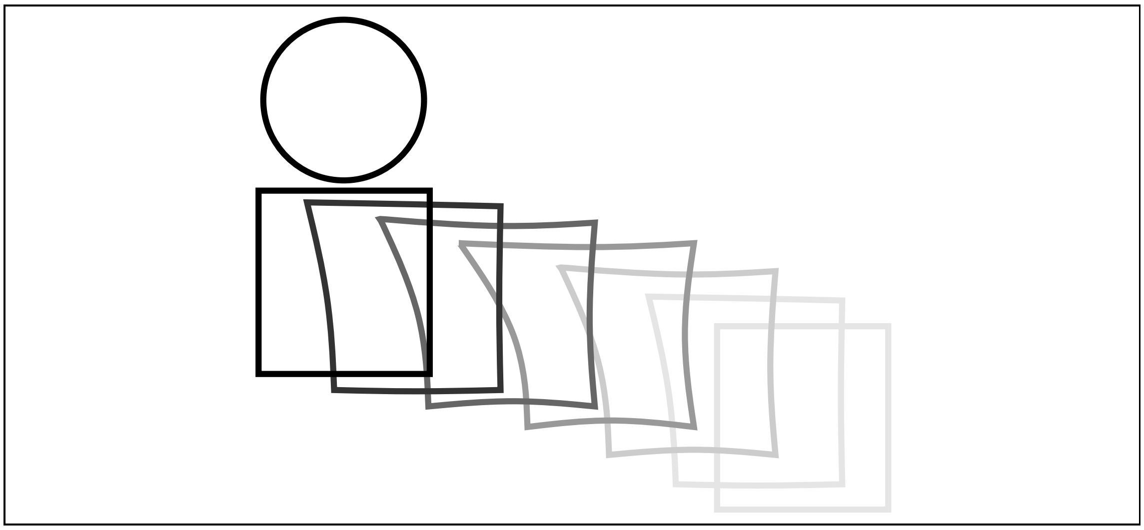


Figure 3.17: Animating an alignment operation

is aligned with the circle. In use, the distortion of the manipulated object gives the impression that the object is being towed along by its corner but is reluctant to change. Whereas without the distortion, the impression is more like the object is moving spontaneously. The net result is that the distortion helps convey the impression that the action, although indirect, is still under the control of the user.

When the user performs a complex operation, the final configuration of the objects is often not quite what was intended; the user might make several attempts before achieving the desired effect. One way to deal with such operations is to adopt a trial-and-error approach, with a heavy reliance on an 'undo' facility; if the final result is incorrect, it can be undone and a new attempt made. At best, such a working style is cumbersome. Instead, I use animation to show the effect of an operation while it is being considered; the animation suggests both what would happen if the operation is committed and what would happen if it is cancelled. By showing both states, this effect reduces cognitive load and minimising the use of short term memory.

The visual feedback to display the above approach uses an extension of the pinning technique previously used to show the effect of constraints on objects. In order to suggest that an operation has not yet been completed, the object is shown

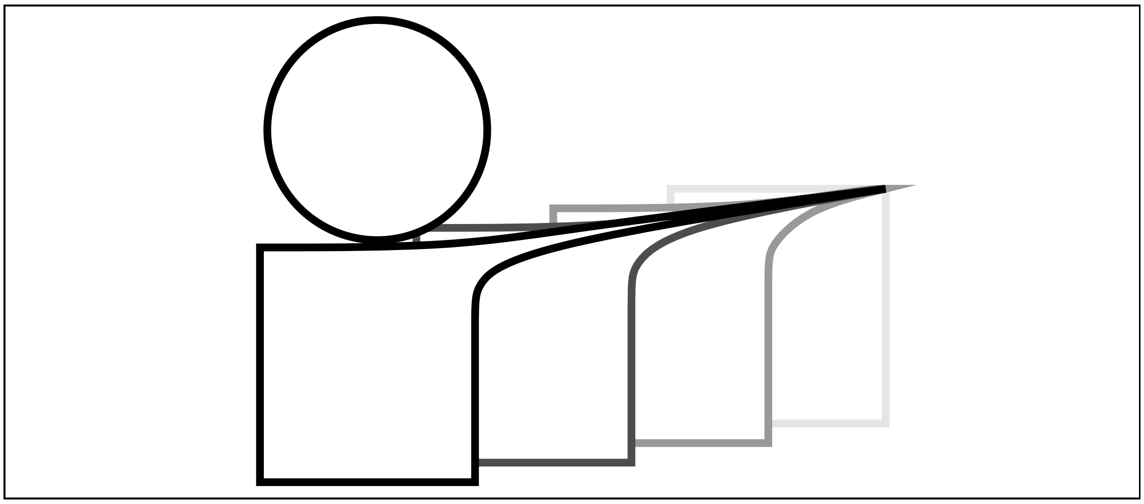


Figure 3.18: Previewing an alignment operation

in the position it will adopt if the change is committed, but with a *telltale* corner stretched back to its original position, as if it is attached by an elastic string. The effect suggests both possible outcomes to the trial operation:

1. If the change is committed, the telltale will retract into the object, showing that the new position has been accepted.
2. If the change is cancelled, the object will be released and the telltale will drag it back to its original position.

Figure 3.18 illustrates the idea in the context of an alignment operation, where the square is moving to the left to align with the circle. The figure shows the situation at the point where the animation for the trial operation has been completed, but before the operation has been committed. The position of the square clearly shows what the final alignment would look like, but its top right corner is stretched back to its original position to show that the object has not yet released its hold on that place. If the operation is committed, the stretched telltale retracts, leaving the square correctly aligned. But if the operation is cancelled, the square springs back to its original position.

The potential benefits of these techniques is twofold: the “liveness” of the objects helps strengthen the sense of direct manipulation that the interface conveys, thus making the interface more satisfying; and animating the effect of the user's actions on objects tells more about the process of the interaction and its outcome, improving the user's understanding of the effect of operations and leading to fewer surprises and mistakes.

3.5 Summary

I have proposed a set of animation effects for direct manipulation interfaces, and constructed videotapes of different animated components to demonstrate how the animations might look. The vidiotapes suggested two areas for applying animation to the user interface: predefined interface components (widgets) and graphical object manipulation.

To see how animation could be incorporated into widgets, I constructed prototypes of animated widgets, including menus that open smoothly; bouncing push buttons; push buttons based on flip book animations; palette buttons that twist; radio buttons that slide in and out; check boxes where the label and box icon are transposed; and 3D rotating palette menus.

To strengthen the illusion of substance for directly manipulated objects, I defined a set of distortion effects that animate translating, scaling, rotating, and shape editing operations. To suggest the anticipated result of indirect operations (such as alignment) and constrained operations (such as snap and drag), I used animation to suggest the tension between the user's action and the object's constraint.

Chapter 4

Supporting Animation in Graphical User Interfaces

This chapter describes a set of algorithms and implementations that support animation in graphical user interfaces. The major contribution is an extended drawing model for graphical user interface toolkits, which supports a subset of the traditional animation effects described in Section 2.1. The subset of effects chosen are targeted at algorithms that can be implemented on current hardware and software technology. Furthermore, the animation effects and software tools are extensions of current user interface toolkit technology, enabling this research to build on past graphical user interface research.

The animation effects described in Chapter 3 can be generated using a combination of object distortion, motion blurring, and keyframing. These techniques can be supported with tools that provide coordinate system warping, blurred drawing, drawing speed control, interpolation, and event recording. To make it easy to incorporate the animation effects into an application, I have incorporated these five tools into a graphical user interface toolkit.

Coordinate system warping, blurred drawing, and drawing speed control can all be implemented at the graphics layer of the toolkit. Apart from its simplicity, this approach has the advantage that the code that draws objects is unaffected by the presence of these tools. Therefore the effects can be applied to simple graphics as well as to more complex graphics.

I used the InterViews toolkit [15, 39] as the basis of my work, but the animation effects could also be implemented on other platforms. To illustrate the kinds of changes required, the chapter presents complete details of the extensions and modifications I made to the InterViews toolkit to support coordinate system warping, drawing speed control, interpolation, and event recording tools. A set of proposed tools and implementation details to support blurred drawing are discussed but not implemented.

4.1 The Original InterViews Drawing Model

The InterViews toolkit [15, 39] has a powerful graphics model that can be extended to include software tools to support animation. The primary responsibility for supporting the InterViews drawing model is placed in the Canvas and Transformer classes. It is these two classes that have been extended to supply support animation.

The Canvas class implements the InterViews 2D affine drawing model, which defines an interface to a set of drawing operations that are similar to those of PostScript [1]. Canvas supports two basic forms of drawing: line-based figures and pixel-based images. The line-based figures can be stroked or filled and are based on PostScript-like paths. The pixel-based images can be of the form of stencils, colour images, or text. All of these forms of images are based on bitmapped images.

Figure 4.1 shows the source code for an example the draws a simple square using Canvas operations. The figure is drawn relative to the position (x,y). The square is constructed by first starting at position (x,y) and continuing to add line segments with the **Canvas::line_to** method at positions (x,y+50), (x+50,y+50), and (x+50,y). The **Canvas::close_path** method adds another line segment that closes

```
Canvas *c;

c->new_path();
c->move_to(x,y);
c->line_to(x,y+50);
c->line_to(x+50,y+50);
c->line_to(x+50,y);
c->close_path();
c->stroke(color,brush);
```

Figure 4.1: Example of using a Canvas object

the path back to the first position (x,y). The **Canvas::stroke** method performs the "drawing" of the figure, with the supplied colour and brush parameters.

All drawings in InterViews are subject to coordinate transformation. The Transformer class represents a 3×3 matrix for use in translating 2D coordinates. The Transformer class supports affine transformations such as rotation, scaling, and translation. Conceptually, a Transformer represents a succession of discrete mapping steps applied one after the other. For example, a Transformer might represent a translation, followed by a rotation, followed by a translation. One characteristic of affine mappings is that they can be computed as matrix products in a homogeneous coordinate system; as a consequence, any sequence of affine mappings can be represented as a single matrix operation. Because of this property, the original Transformer stores only a single matrix; successive transformations on the Transformer modify the matrix to reflect the new aggregate operation.

Figure 4.2 is an extension to the example in Figure 4.1. It shows the source code required to rotate the square -45 degrees about its centre point, referred to as **mid_x, mid_y**. To help highlight the new source code, the source code from the previous example is indicated by the use of italic font. The example first defines the centre point of the square and then a Transformer object for the transformations. The Transformer is translated to bring the square's centre to the

```
Canvas *c;

Coord mid_x = x + 25.0;
Coord mid_y = y + 25.0;
Transformer t;
t.translate(-mid_x,-mid_y);
t.rotate(-45.0);
t.translate(mid_x,mid_y);
c->push_transform();
c->transform(t);

c->new_path();
...
c->stroke(color,brush);

c->pop_transform();
```

Figure 4.2: Example of using a Transformer object

origin of the coordinate system, rotated by -45.0 degrees, then translated back to its original position. The Transformer now contains a matrix to rotate the square about its centre. The **Canvas::push_transform** method pushes a new copy of the current Transformer object on the top of the Canvas's stack of transformers. The Transformer is applied with the **Canvas::transform()** method as a pre-multiply matrix operation to the top Transformer object on the stack. After the object is drawn, the Transformer is removed from the top of the stack of transformers with the **Canvas::pop_transform** method.

4.2 Pseudo-3D

Convincing animations require rapid and high-quality graphics. However, many current toolkits use graphics models that are adequate only for static 2D images. User interfaces have been improved by extending the graphics model into the third spatial dimension, amply demonstrated by the Information Visualizer [58]. I have

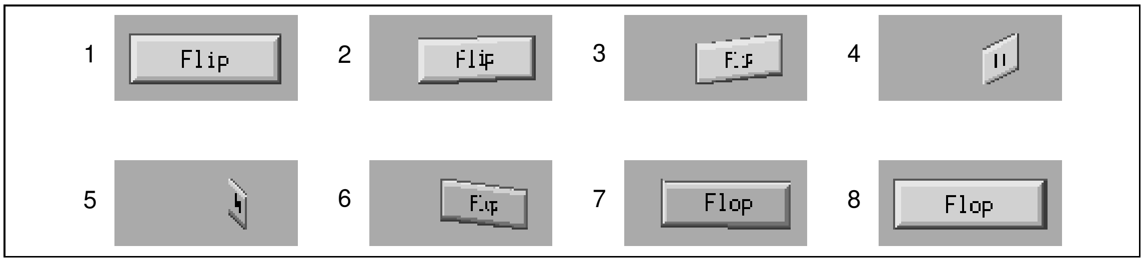


Figure 4.3: 3D path animation

implemented a simple pseudo-3D capability on top of the InterViews 2D graphics model [78].

The pseudo-3D mechanism allows for simple orthographically projected 3D graphics with only 2D graphics hardware/software systems support. Figure 4.3 shows an example of an animation that displays a button flipping over to expose its underside; the animation requires that the button label text be drawn at arbitrary angles and sizes. The pseudo-3D mechanism allows applications to specify graphics transformations using a full 3D transformation matrix; the transformed object is drawn in an orthographic projection on the 2D surface. I have built 3D-like interactive objects using this pseudo-3D mechanism.

This section describes the method by which a 2D graphical object may have a 3D transformation applied to it and displayed using only 2D transformations. Such an approach would not be possible if the graphics model underlying the toolkit did not support 2D transformations. Thus, this pseudo-3D mechanism relies heavily on the InterViews graphics model, which allows arbitrary 2D transformation of all its objects, including text and images. However because InterViews does not support high quality 2D transformations of text and images for example, it does not use antialiasing), text is difficult to read for certain transformations.

4.2.1 Pseudo-3D Mechanism

The pseudo-3D technique takes a 2D object described in a plane and transforms it into three space using a given 3D transformation matrix. The object in three space is then orthographically projected back on to the original plane for displaying. The basic idea is that, given a 3D transformation matrix, a 2D transformation matrix can be constructed directly from elements of the 3D transformation matrix under certain conditions. More formally, if a 3D graphical transformation,

$$\mathbf{M} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \tag{4.1}$$

is applied to a point that lies in the plane $z = 0$, and the transformed object is viewed along the z axis under orthographic projection, then the transformation can be replaced by an equivalent 2D transformation,

$$\mathbf{D} = \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix}. \tag{4.2}$$

Appendix B contains a description that shows that these two matrices are equivalent under the following constraints:

1. The points to be translated must be of the form [x y 0 1], that is, the graphics to be translated must be 2D in nature. All standard widgets are of this form.

2. The projection must be orthographic. An *orthographic projection* is one that transforms 3D objects onto a 2D plane, where the direction of the projection is normal to the viewing plane. The *viewing plane* is defined as the plane onto which the 2D objects are projected.

3. The viewing plane must be normal to the *viewing vector*, the direction of observation of the viewing plane.

4. The viewing plane must be a plane defined as z = 0.

5. The viewing vector must be the z axis.

6. The *view reference point* must be located at x = 0, y = 0 and z = 0. The view reference point is defined as the point on the viewing plane that describes the origin of the 2D viewing coordinate system.

4.2.2 Implementation

I have added the DDDtransformer and Face classes to the InterViews toolkit to support the pseudo-3D technique. The DDDtransformer class allows the programmer to specify and manipulate 3D graphical transformations, while the Face class allows for a glyph to be displayed in pseudo-3D. The DDDtransformer class, an extension of a 3D transformation class described by Tony Albrecht in his Honours Thesis, *A 3D Graphics Extension to the InterViews Toolkit* [2], is subclassed from the original 2D Transformer class to include the z dimension; Figure 4.4 shows the class interface.

As with the original InterViews Transformer class, the DDDTransformer class stores a transformation matrix and provides a set of access methods. The access methods provide 3D versions of the Transformer class methods, using the suffix "3D" appended to the end of the method name.

The *Face* class is defined to take an InterViews 2D drawable object, a Glyph, described in the $z = 0$ plane and transform it using a given 3D transformation. Figure 4.5 shows the Face C++ class definition. The constructor has as its parameters a glyph and a DDDTransformer object describing the pseudo-3D placement of that glyph. The DDDTransformer object contained in a Face object can be changed with the **Face::update** method. The current DDDTransformer object being applied to the glyph can be retrieved with the **Face::current_dddtransformer** method. The **Face::visible** method returns a boolean telling if the front of the glyph is exposed;

```
class DDDTransformer : public Transformer
{
public:
    DDDTransformer(); /* identity */
    DDDTransformer(const DDDTransformer &);
    DDDTransformer(float a00, float a01, float a02,
                   float a10, float a11, float a12,
                   float a20, float a21, float a22,
                   float a30, float a31, float a32);
    virtual void add3D(const DDDTransformer &);
    virtual void subtract3D(const DDDTransformer &);
    virtual void multiply3D(const DDDTransformer &);
    virtual void premultiply3D(const DDDTransformer &);
    virtual void postmultiply3D(const DDDTransformer &);
    virtual void shear3D(float sh_x, float sh_y);
    virtual void translate3D(float dx, float dy, float dz);
    virtual void scale3D(float sx, float sy, float sz);
    virtual void rotate3D(float angle, int axis);
    virtual void transform3D(float &x, float &y, float &z);
    virtual void transform3D(float x, float y, float z,
                             float &tx, float &ty, float &tz);
    virtual void matrix3D(float &a00, float &a01, float &a02,
                          float &a10, float &a11, float &a12,
                          float &a20, float &a21, float &a22,
                          float &a30, float &a31, float &a32) const;
};
```

Figure 4.4: DDDTransformer C++ class definition

a simple cross product of the x and y axis of the glyph's coordinate system is used to determine if its z axis is pointing into or out of the screen.

4.3 Distorting Graphical Objects

The distortion of graphical objects can enhance a user's impression of substance when interacting with objects in an application [77]. For example, if an object is dragged by a corner, it might distort slightly as if it had inertia; if it collides with other objects, they might wiggle as if bumped. The overall effect is that the

```
class Face {
public:
    Face(Glyph *, DDDTransformer * = nil);
    virtual ~Face();
    void update(DDDTransformer *);
    DDDTransformer * current_dddtransformer();
    boolean visible() const;
};
```

Figure 4.5: Face C++ class definition

user has a stronger sense that "things are happening" and that real work is being done [76].

One way to suggest interaction between physical objects and a user would be to represent the objects with a physical model and the user's interaction as forces acting on the objects. Then the responses of the objects could be determined by simulating the physical system. However, a much simpler model is sufficient to produce pleasing and effective visual feedback.

The approach I have taken is to draw the manipulated objects as usual, but to warp the coordinate system of the drawing surface so that the view appears distorted. The effect is as if the objects were drawn on a rubber sheet, and the sheet was then stretched by pulling at certain points. This technique is widely used to distort images [90]. The warping of the coordinate system is implemented as a transformation applied to all underlying drawing primitives. To add a particular animation effect to an interaction, an application need only calculate a set of vectors that characterise an appropriate warp. Apart from its simplicity, this approach has two key advantages when compared with an approach based on a physical model: the code that draws the warped objects is unaffected by the presence of the warp, and the warping effect applies equally well to complex graphics as to a simple rectangle. This section describes the calculations required to perform the warp

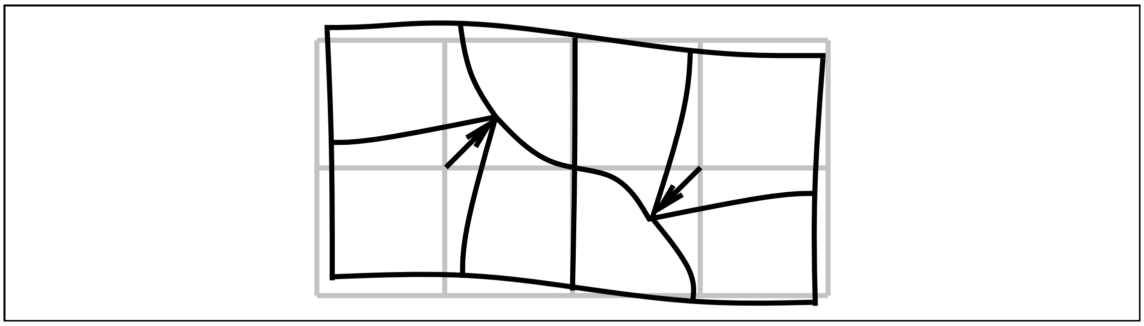


Figure 4.6: The effect of a set of warp vectors

transformations and the extension to the InterViews toolkit.

4.3.1 Computing the Warp Transformation

The warp transformation is characterised by a set of bound vectors [1] that describe the transformations applied to key points in the coordinate space. Transformations for points that do not coincide with vectors are calculated by interpolating between the vectors. Figure 4.6 illustrates the idea. In the example, the grid is defined by the grey lines, and the warp vectors by the black arrow-headed lines. The transformed grid is shown in black. Points that coincide with warp vectors are transformed by the specified amounts. Other points are transformed by intermediate amounts that depend on their relative distances from the specified vectors.

The implementation of the warp transformation computes a displacement vector for each transform point that is a weighted average of the warp vectors [2]. For a point q and a set of warp vectors p_i, each defined by a location p_{il} and a displacement p_{id}, the transformed point q' is computed as follows:

$$q' = q + \sum_{i=1}^{n} w_{qi} p_{id} \tag{4.3}$$

[1] A bound vector is defined as pair of values: a vector and a location of the tail of that vector.
[2] Thanks to my colleague Murk Bottema of the Cooperative Research Centre for Sensor Signal and Information Processing (CSSIP) for suggesting the algorithm.

where the vector weights, w_{qi}, are computed as

$$w_{qi} = \frac{c_{qi}}{\sum_{j=1}^{n} c_{qj}} \tag{4.4}$$

which depend on a set of "closeness" coefficients

$$c_{qi} = 1/|q - p_{il}| \tag{4.5}$$

The coefficient c_{qi} represents the "closeness" of the point q to the transform vector p_{il}. In the current implementation, closeness is computed as the reciprocal of the distance between q and p_{il}. The undefined case when the distance is zero is treated specially. This formulation ensures that the vector weights remain in the range $[0.0, 1.0]$, and that the sum of the weights is 1.0.

Apart from its simplicity and relative efficiency (the algorithm has complexity $\mathcal{O}(n)$ for n warp vectors), the implementation of the warp leads to several properties that simplify the selection of a suitable set of vectors for a desired warp effect.

1. Points coincident with one of the warp vectors are displaced by the amount of that vector.
2. Points equidistant from two vectors are affected equally by those vectors.
3. In the limit, the combined effect of distant vectors approaches the mean of all vectors.

To understand why these properties hold, consider the behaviour of the vector weights as q varies. If q approaches one of the vector locations, p_{il}, then c_{qi} grows without bound; thus w_{qi} approaches 1.0 while all other weights approach 0.0. In the limit, when q is coincident with p_{il} only that vector affects the transformation. Conversely, if q becomes very distant from all vectors compared with the distance between the vectors, then all c_{qi} become equal and thus all w_{qi} approach $1/n$.

In the above warp calculation, the "range" of each vector is the same, and the distance calculation uses Pythagoras' theorem. The rate at which the vector's influence decreases with distance is the same for all vectors. To gain more control of an individual vector's influence over the vector field, the calculation of c_{qi} is extended as follows:

$$c_{qi} = 1/(|q - p_{il}|)^y \tag{4.6}$$

where

$$y = \frac{f}{r_i}. \tag{4.7}$$

The model is extended so that each vector's range is controlled by a separate *range coefficient* r_i and an overall *warping factor* f. Using the range coefficient r_i, particular vectors can be 'short-range' (their influence is confined mainly to the area close to the vector) and others can be 'long-range' (their influence extends over a much greater distance). The overall strength of the warping can be adjusted by the warp factor f.

Figure 4.7 shows the warping of the grey square with different range coefficients while keeping the vectors constant and the warp factor set to 1.0. One zero magnitude vector is placed at the position indicated with the black dot. The other three corners have vectors with values indicated by the arrow-headed lines. The thin line object in the middle diagram has a range coefficient of 1.0 for all four warp vectors; this is the default setting for a warp vector. The ratio of range coefficients between the different warp vectors provides the means for 'short-range' and 'long-range' effects. In this example, the ratio is derived from the range coefficient of the zero magnitude vector compared to the range coefficient of the vectors with non-zero magnitudes. The object with the 1/4 ratio has a range coefficient of 0.25 for the zero magnitude vector and a range coefficient of 1.0 for the other three vectors;

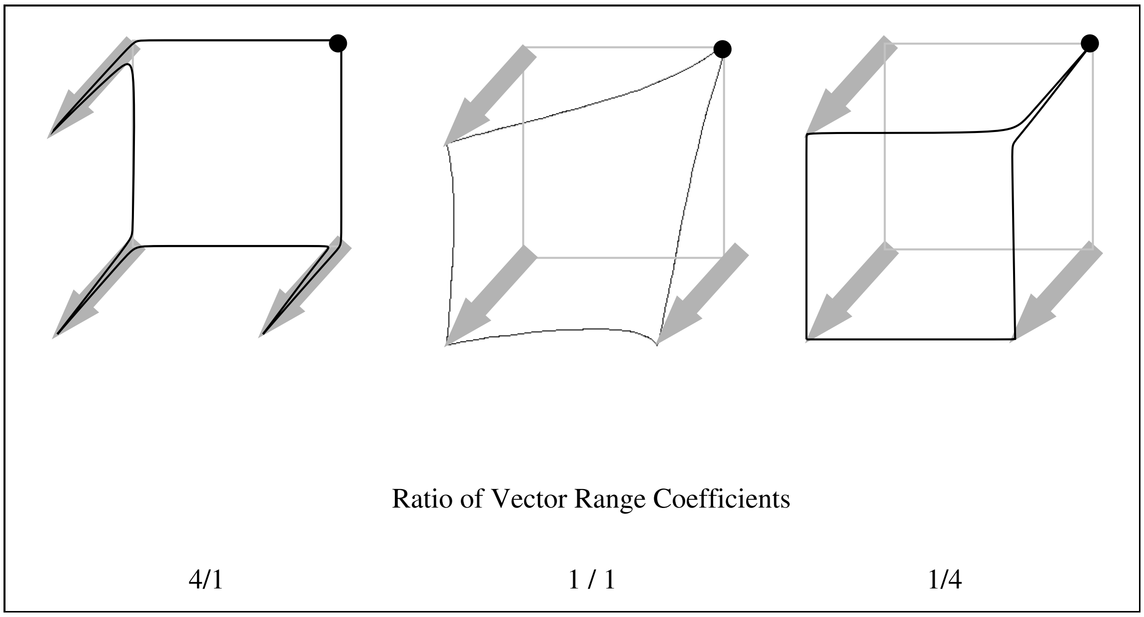


Figure 4.7: Changing the ratio of the vector range coefficients (r_i) for the warp calculation

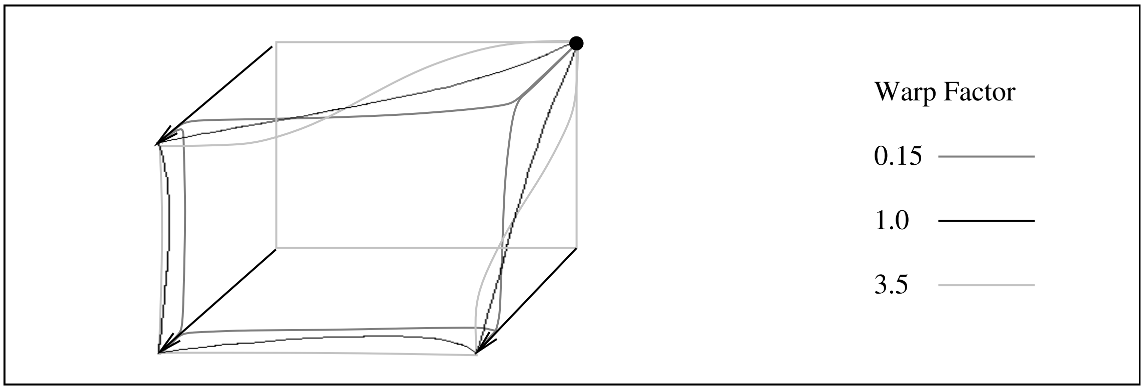


Figure 4.8: Changing the warp factor (f) for the warp calculation

this produces a short-range for the zero vector and a long-range for the remaining three vectors. The reverse effect is shown in the 4/1 ratio diagram, where the effect only significantly distorts the region very close to the non-zero vectors.

The *warping factor* provides a means to change the overall strength of the warp field. Figure 4.8 shows three different warp factors: 0.15, 1.0, and 3.5. With larger warp factors, the result of the warping effect spreads over a larger distance. In contrast, a small warp factor restricts the warping to a more localised region.

4.3.2 Implementation

To support the warping of graphical objects, the InterViews drawing model must be extended. This section describes three modifications to the original InterViews drawing model: the addition of a warp transformation to the Transformer class; the extension of the drawing routines of the Canvas class to support warped drawings; and a new Warp class to encapsulate the manipulation of a set of warp vectors. Finally, the section examines an example of using the new Warp class and warp transformations.

New Transformer Class

I extended the InterViews drawing model to include the new warp transformation. Since all drawing in InterViews is subject to coordinate transformation, this one addition to the application programmer's interface is sufficient to add warping capabilities to all InterViews objects. The addition of a single method, **Transformer::warp**, to the Transformer class provides support for the new warping transformation.

The original InterViews Transformer class supported only affine transformations, such as rotation, scaling, and translation. As the warp mapping is not affine, combinations of transformations that involve warping cannot be represented as a single matrix. The implementation of the Transformer class is therefore modified to enable it to keep a list of transformation items. Successive affine transformations are combined into a single matrix item as before, while warp transformations are represented by individual warp items. The modified **Transformer::transform** method iterates through the list of items applying each transformation in turn. This implementation retains the efficiency of the original implementation if the Transformer represents only affine transformations (by far the most common case), but it allows for arbitrary transformations when necessary.

Changes to the Canvas Class

Extending the Canvas class to draw path-based figures under the warping transformation requires only a slight modification. Affine transformations, as in the original Canvas drawing model, have the property that straight lines remain straight after transformation. To draw a straight path segment, therefore, the original Canvas has merely to transform the segment end points and draw a line between the new points. Since the warping transformation is non-affine, this assumption can no longer be made. Instead, the algorithm recursively bisects the segment until the warped sections can be approximated by straight lines.

Drawing image-based graphics is more difficult. The original Canvas implementation constructed transformed images and then used a back-transformation scheme to populate the transformed image with data from the original image. This scheme can not be used for warp transformations because the warp transformation does not, in general, have an inverse. Instead, an approximation to the warped image is computed by transforming the coordinates of each pixel in the original image and filling the corresponding area in the transformed image. The current algorithm is slow and produces poor results under some warp conditions, but it is sufficient to demonstrate the effect of the warp.

New Warp Class

The new *Warp* class has been defined to represent a set of warp vectors; Figure 4.9 shows the C++ class definition. A Warp object initially contains no vectors. The default warp factor is set to 1.0, if not specified. Calling **Warp::add_vector** adds a vector positioned at the point (**x**, **y**), with displacement components **vx** and **vy**, and an optional range coefficient as the last parameter. The **Warp::init_warp** method initialises the warp object to an empty set of vectors, or to a given set of vectors **w**. The method **Warp::vector_enclosing_box** returns through its pa-

```
class Warp {
public:
    Warp (Coord warp_factor = 1.0);
    ~Warp ();
    virtual void add_vector(Coord x, Coord y, Coord vx, Coord vy,
                            Coord range = 1.0);
    void init_warp();
    void init_warp(Warp * w);
    void vector_enclosing_box (Coord & left, Coord & bottom,
                               Coord & right, Coord & top);
    virtual void transform_point (Coord x, Coord y,
                                  Coord & tx, Coord & ty) const;
    void warp_factor (int wf);
    int number_of_vectors();
    boolean identity();
};
```

Figure 4.9: The new Warp C++ class definition

rameters the degree to which the drawing area of a glyph will be increased in each dimension due to the warping. Given a point (**x**,**y**), the **Warp::transform_point** method returns transformed point, (**tx**,**ty**).

The warp factor of the object is changed by the **Warp::warp_factor** method. The number of vectors currently in the warp object is obtained from the **Warp::number_of_vectors** method. Finally a test to determine if the warp object has an empty set of vectors is supplied by the **Warp::identity** method.

Warping Example

Figure 4.10 shows a warped version of the square drawing example. In this example, four vectors are added at the four vertices of the square. The effect of the warping transformation is shown in Figure 4.11. The light grey square is the unwarped version of the square, and the warped version is black. The zero magnitude warp vector at vertex (x+50,y+50) is denoted by the black dot. The three warp vectors

```
Transformer t;
Warp w;

w.add_vector(x,    y,    -10, -10);
w.add_vector(x,    y+50, -10, -10);
w.add_vector(x+50, y+50,   0,   0);
w.add_vector(x+50, y,    -10, -10);
t.warp(&w);
c->push_transform();
c->transform(t);

c->new_path();
c->move_to(x,y);
c->line_to(x,y+50);
c->line_to(x+50,y+50);
c->line_to(x+50,y);
c->close_path();
c->stroke(color,brush);

c->pop_transform();
```

Figure 4.10: An example of applying a warp

of magnitude (-10, -10) at the remaining three vertices are denoted by the arrow-headed lines.

4.4 Time Management

The inclusion of mechanisms for recording and manipulating information about time are fundamental for supporting animation in a user interface toolkit. This section first discusses the mechanism for managing a history of time-stamped event information. The tools required for the management of frame rates during the generation of animations are then discussed. The section goes on to describe keyframe animation tools for supporting the changing of graphical objects over time, and concludes with the description of how to specify autonomous animations.

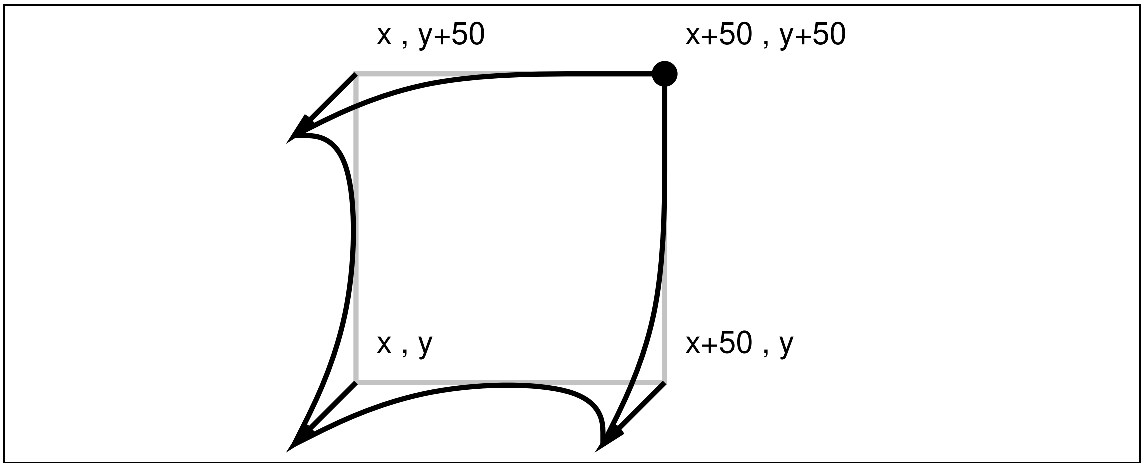


Figure 4.11: Graphical depiction of Figure 4.10

4.4.1 Time-Stamping and History Management

Direct manipulation of a graphical object can change its position, orientation, and size over time. These changes happen at discrete moments over a continuous time frame while using the application. There are situations where applications may be required to interpolate spatial information to retrieve spatial information at arbitrary points of time, such as when creating animations that model real-world behaviour.

I have defined a class *History* for the creation, addition, access, and interpolation of time-stamped event information. Figure 4.12 shows the class definition. A set of time-stamped event information is referred to as a *history*. The **History::start_history** method starts a new history collection. Invoking this method on a history that has already collected information resets the history to the initial state. There are three **History::add_event** methods for adding time-stamped event information specified in one of three ways:

1. as position, rotation, and scaling;
2. as position and velocity; and
3. from an event record.

The number of time-stamped events in the history can be obtained through

```
class History {
public:
    void start_history();
    void add_event(Coord x, Coord y, Coord angle,
                   Coord scale_x, Coord scale_y,
                   unsigned long timestamp);
    void add_event(Coord x, Coord y,
                   Coord velocity_x, Coord velocity_y,
                   unsigned long timestamp);
    void add_event(const ivEvent& e);
    int number_of_events();
    ObjectPosition * list_of_events();
    void past_event(Coord time_units, Coord & x, Coord & y,
                    Coord & angle,
                    Coord & scale_x, Coord & scale_y);
    void past_event(Coord time_units, Coord & x, Coord & y,
                    Coord & velocity_x, Coord & velocity_y);
}
```

Figure 4.12: History C++ class definition

the **History::number_of_events** method, and a pointer to a list of the history information is returned by the **History::list_of_events** method. Finally, the **History::past_event** methods are defined for retrieving interpolated spatial and velocity values at some increment of time, **time_units**, in the past.

4.4.2 Frame Rate Management

The demand on computational resources is a particular problem for direct manipulation style interfaces. This problem is compounded by the addition of animation in the interface. A user moving a graphical object across the screen forces that object to be drawn many times. The number of times the graphical object is drawn per second determines the *frame rate* of the animation. Computational resources are constrained by the current available resources and the complexity of the graphical object, both of which may change over time. A direct manipulation application may therefore be required to accommodate changes of both kinds.

Researchers [64, 71] have investigated two approaches for ensuring that an application maintains the minimum frame rate required to create the illusion needed for direct manipulation. The first approach is based on timing the rate events on arrival, and the second measures the time required to perform the drawing of a graphical image. In both of these cases, the application makes a decision as to whether it is able to maintain the required frame rate. If the minimum frame rate is not maintained, steps can be taken by the application to reduce the amount of required computation, thus allowing the frame rate to increase.

The first approach, based on recording the rate events arrive, takes into account all aspects of the current computational load on the computer. This approach records event times for direct manipulation operations and ignores all other events. The advantage of this approach is that the application adjusts resources based on the total computational load, and in an indirect manner takes into account such aspects as other processes and network delays. The disadvantage of this is that it produces a crude measure of computational load. This is the approach I have using in my system.

The second approach, based on recording the time required to perform the drawing, focuses on the resources directly associated with rendering the graphical image. An advantage of this approach is that adjustments made will almost certainly significantly affect the frame rate, since they are based on a calculation directly related to the resources for rendering. On the other hand, the "time to draw" calculation does not take into account other computational resources.

The Pacers system [71] uses the second approach to address the problem of automatically maintaining the frame rate of an animation when there are insufficient computational resources to fully draw complex graphics and respond to user input in a timely fashion. It dynamically adapts the presentation quality of the graphics depending on available resources. For example, if there is insufficient time to draw a solid object, the pacer might omit fine grained detail or draw an object outline instead.

4.4.3 Keyframe Animation

As previously described in Section 2.2.2, computers can aid in the animation process by performing the calculations of the transformations that specify the in-between frames that constitute an animation sequence. The flipping button in Figure 4.3 is an example of such an animation sequence. To support the animated interface components, I defined a set of classes that provide three kinds of animations: animation by transformation, animation by flip book, and animation of resizing windows. Others have used different techniques, such as dissolves and motion blurring [18].

The Path and Path_Animator classes , together with a Bézier time mapping function and 3D transformation interpolation algorithm, support animation by transformation of drawable objects. Animation by flip book is supported with the new Deck_Animator class, and animation of resizing windows is supported with the new Window_Animator class.

Path Class

A *Path* object is used to define a series of keyframes for an animation. For example, the button flipping over to expose its underside in Figure 4.3 was implemented with a Path object. A Path consists of an initial DDDTransformer object and a series of tuples that define the path's keyframes. Each keyframe tuple, stored as a PathItem object, contains a DDDTransformer object, a duration in seconds, and the number of frames of the animation required to transform the graphical object to this new geometric configuration. The DDDTransformer object defines the new position,

```
class Path
{
public:
   Path(DDDTransformer * start,
        DDDTransformer * finish,
        float duration, int steps);
   void append (PathItem * frame);
   float total_duration();
   int total_steps();
   void cycle(boolean cycle_state);
}
```

```
class PathItem
{
public:
   PathItem(DDDTransformer *,
            float duration,
            int steps);
   DDDTransformer * matrix();
   float duration();
   int step();
 }
```

Figure 4.13: Path and Path_Item C++ class definitions

orientation, and scaling. Figure 4.13 shows the Path and PathItem C++ class definitions.

The PathItem class constructor is passed the keyframe tuple information. This tuple information may be retrieved through a set of methods. The constructor for the Path class specifies the **start** transformation matrix, the **duration** in seconds, the number of **steps**, and the first keyframe **finish**. The **Path::append** method adds an additional keyframe to the animation. The **Path::total_duration** method returns the sum of all the durations contained within the list of path items, and the **Path::total_steps** method returns the total number of steps. Finally, the **Path::cycle** method sets the Path's cycle state to true or false. If true, the animation continues after the last path item is finished by resetting the path to its initial conditions; if false, the animation stops once the total number of steps have been reached.

Path_Animator Class

A *Path_Animator* object coordinates the animation of a drawable object with the information specified by a Path object. The major responsibility of a Path_Animator is the calculation of the in-between frames for the keyframes specified by the Path.

```
class Path_Animator {
public:
    Path_Animator(Glyph *, Path *);
    void flip_to(int index);
    void append_path(Path *new_path);
}
```

Figure 4.14: Path_Animator C++ class interface

The Path_Animator calculates a transformation matrix for each in-between frame based on the percentage of time between keyframes and an interpolating function. These calculations are described in more detail in Appendix C.

A Path_Animator can control multiple paths. Typically, the paths represent different trajectories for a drawable object. For example, one path might specify the transformation of a button from an "off" to an "on" state, and another might specify the transformation, perhaps by a different trajectory, back to the "off" state. Figure 4.14 shows the C++ class definition. The Path_Animator constructor is passed a Glyph object to animate and a Path object to define the initial animation. The **Path_Animator::append_path** method adds a new path to the end of the list. The **Path_Animator::flip_to** method is used to begin an animation path specified by the **index** parameter. In the above "on" and "off" button example, the first trajectory may be specified with the value zero and the second trajectory path with the value one. The Path_Animator uses an interpolating function, described later in this section, that results in a simple slow-in/slow-out transition between the keyframes. An example of the Path_Animator list structure is shown in Figure 4.15

Bézier Timing Function

To allow for the slow-in/slow-out effect desirable in animations, Path_Animator uses a time mapping function based on a third order Bézier function [9]. This function calculates the amount of change for each of the in-between matrices over

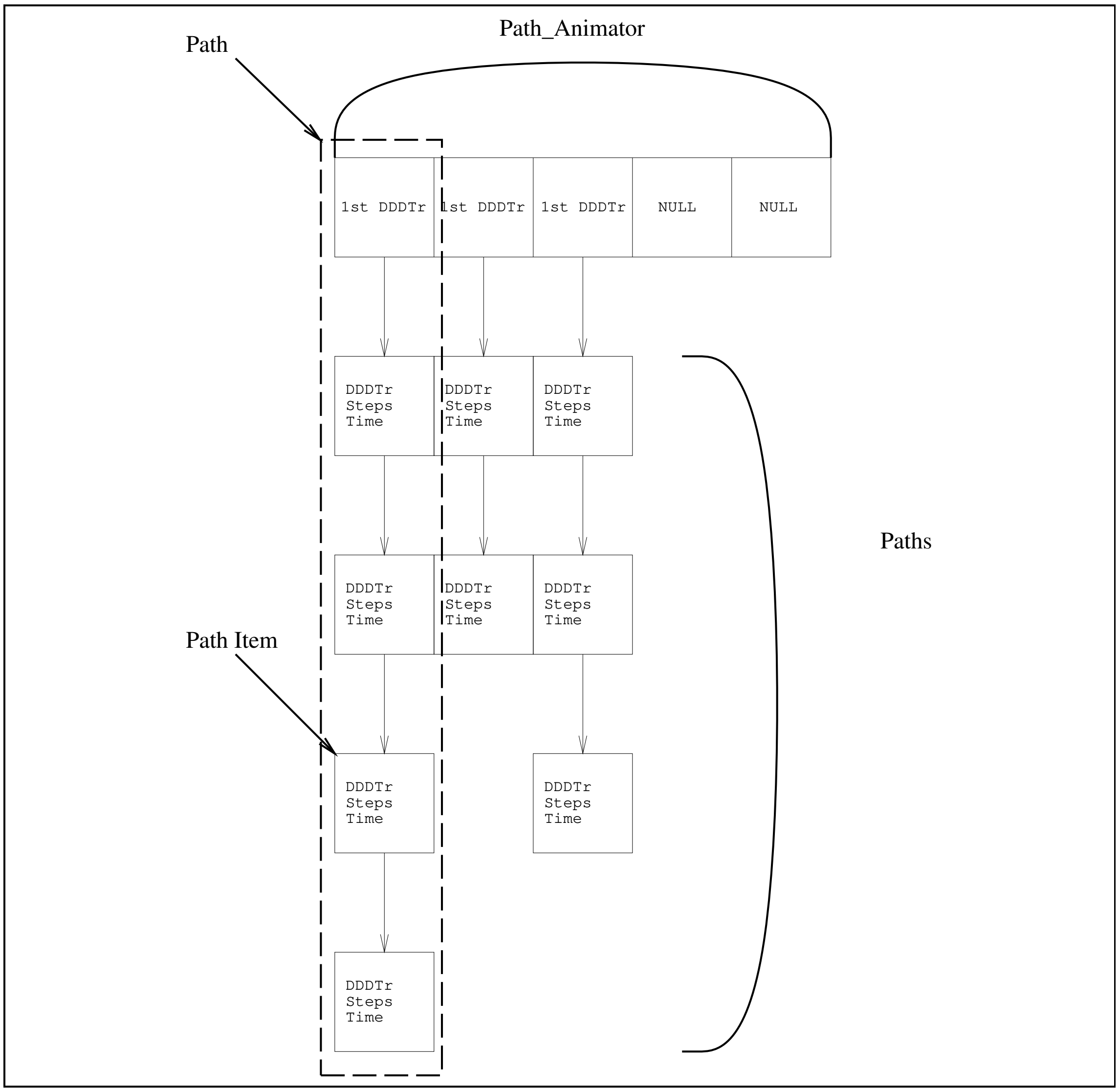


Figure 4.15: Path_Animator list structure

time. The Bézier function maps the percentage of change relative to the proportion of elapsed time, where time remains linear and the amount of change is mapped onto a Bézier curve.

Figure 4.16 shows an example of a Bézier curve. The curve is defined by the four control points x_1, x_2, x_3, and x_4. The curve has the desirable slow-in/slow-out effect, where $d\mathcal{B}/dt$ is small for t close to 0.0 and 1.0. In other words, the rate of change is small near the beginning and end of the interpolation.

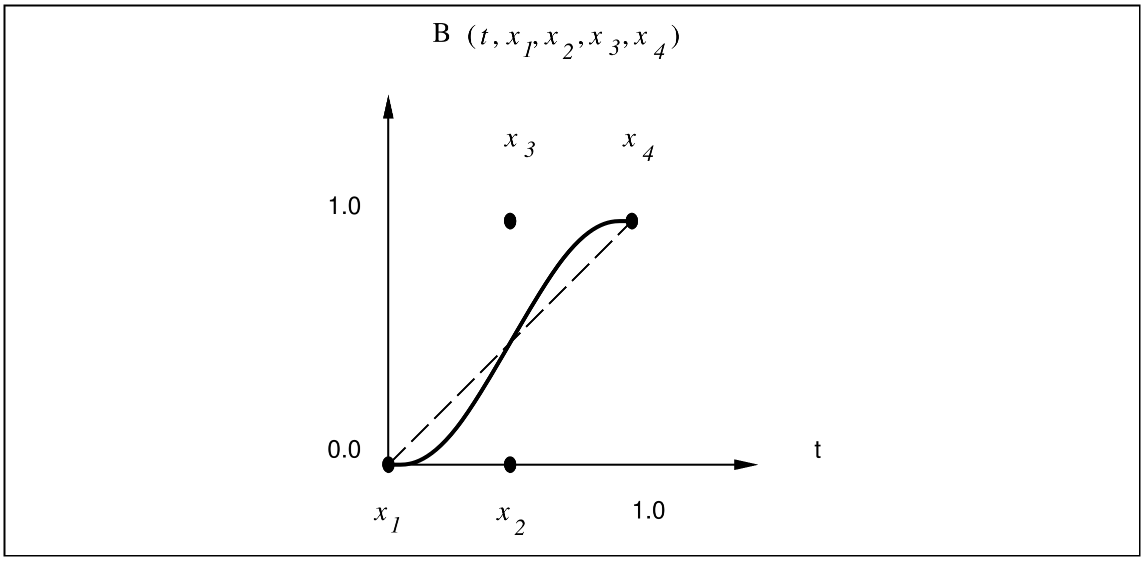


Figure 4.16: Example of a Bézier curve

Deck_Animator

The *Deck_Animator* class is used to build flip-book-style animations, where the frames in the animation are each specified by a separate glyph. Figure 4.17 shows the Deck_Animator C++ class definition.

The **Deck_Animator** constructor is passed an InterViews Deck as its only parameter. A Deck is a composite glyph that arranges its components like a stack of cards, with only the topmost glyph visible. The glyphs contained in the deck are stored in the correct order for presentation as an animation. The **Deck_Animator::duration** method sets the total number of seconds for the animation. The **Deck_Animator::flipper** method causes the animation to proceed to the given **end_index**. If the **end_index** is less than the current index in the deck, the animation is performed in reverse order.

Window_Animator Class

The *Window_Animator* class allows for the animated opening and closing of a window. This class supports animations that can be described by rescaling the window, and its contents, to fit a specified rectangle. Figure 4.18 shows the Win-

```
class Deck_Animator {
public:
    Deck_Animator(Deck *in_deck);
    virtual ~Deck_Animator();
    virtual void duration(float);
    void flipper(int end_index);
};
```

Figure 4.17: Deck_Animator C++ class definition

```
class Window_Animator {
public:
    Window_Animator(Window *);
    void start(boolean in_forward, Rectangle start,
               Rectangle end, float duration, int steps);
};
```

Figure 4.18: Window_Animator C++ class definition

dow_Animator C++ class definition. The constructor is passed an InterViews Window object as the window to animate. The **Window_Animator::start** method begins an animation that is described by a **start** and **end** rectangle, a **duration**, and a number of **steps**. The boolean parameter **in_forward** specifies whether the animation goes forward (opening the window) or backward (closing the window.) At each frame of the animation, the contents of the window are resized so that they fill the rectangle.

As drawing resized images is potentially slow, Window_Animator stores the resized images as X Windows pixmaps for future use. The first time the animation is performed, it proceeds relatively slowly; however subsequent animations run at full speed because the stored images can be displayed rapidly. The Window_Animator uses the InterViews caching for the storage of pixmaps. However because InterViews does not frees the cached pixmaps, the memory storage could grow large. A simple least recently used algorithm or first-in first-out algorithm could be implemented to replace old pixmaps.

4.4.4 Autonomous Animation

As defined in the previous chapter, an *autonomous animation* is an animation that is controlled by the application and not by the user. The Window_Animator, Deck_Animator, and Path_Animator all use autonomous animations. InterViews and most user interface toolkits use an event-based model to control the application program. The basic main function of the application program is as follows:

```
program begins
    initialise program and user interface components
    loop forever
        block on an event
        dispatch the event to correct user interface component
    end loop
end program
```

The program first creates the user interface components needed for the initial configuration of the application, then goes into an infinite loop. At the top of the loop is a function that blocks while waiting for an event. For most applications, these events are initiated by keyboard presses or interactions with the mouse. Once an event is detected, the user interface toolkit encodes the event data into an event-specific data structure and passes it to the correct component.

To achieve the necessary response from the interface, the processing of the events by the interface components must be quick in order to allow the application to receive and process the next event. A component can not retain control of the program for any great length of time, which poses a problem for programs that perform autonomous animations. Two solutions to this problem have been proposed: the use of lightweight thread processes to control the animation; or timers to place events on the incoming event list. A solution based on threads provides a generic and powerful mechanism for multiple processes controlling the user interfaces, but has the disadvantage of being complex. Instead, I used timers, which produces a usable but simpler solution.

In the X-Windows environment, timers are supported in the Xt Intrinsics library [24]. The functions **XtAppAddTimeOut** and **XtRemoveTimeOut** allow adding and removing of timers that initiate callback functions after a given length of time. The InterViews toolkit does not rely on the Xt Intrinsics library; instead it supplies two methods for the same purpose: **Dispatcher::startTimer** and **Dispatcher::stopTimer**.

The flipping button provides an example of using timers. When the button is first "clicked" by the user, this causes the Path_Animator associated with the button to start the animation by starting a timer. When the timer fires, the rotation animation occurs as follows:

1. Rotate the button $\frac{1}{7}$ of 90 degrees about the y axis.
2. Redraw the button.
3. If the animation is not done, reset and start timer.

4.5 Motion Blur

As discussed in Section 2.1.2, motion blurring can provide temporal anti-aliasing, which results in smoother-seeming movement for a given frame rate. Conversly, motion blurring allows the frame rate of an animation to be reduced, while maintaining its legibility.

Several forms of motion blurring techniques have been investigated to determine their effectiveness in improving the sense of smoothness of motion. During the initial prototyping experiments, frame-by-frame animations were created of movements using several different types of blurring, then the animation was recorded onto video tape to assess the effect at full-speed. Figure 4.19 shows frames from

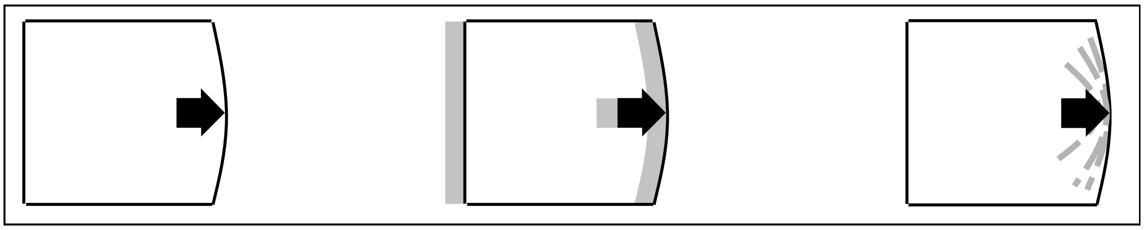


Figure 4.19: Motion blurring techniques

several motion sequences. The first frame shows the translation opertion without a motion blurring effect. The second frame uses a uniform grey smear to suggest the movement of the object relative to its position in the previous frame. The third frame uses a shock-wave like effect to suggest the movement.

As with the distortion of graphical objects, the motion blur effect could be implemented as a Canvas level operation. The application programmer would be able to apply the motion blurring effect to an arbitrary set of standard drawing primitives. The choice of when and where to apply motion blurring in an application is left to the application programmer. Based on informal testing, I propose two forms of motion blurring effects.

1. The smear motion blurring effect is a copy of the object with a thicker outline and a lighter colour, as suggested by Figure 4.20.
2. The stutter motion blurring effect is constructed from multiple copies of an object with a lighter colour, as illustrated by Figure 4.21.

A possible programming model is to push and pop the motion blur effect in the same manner as a graphical transformation. This allows the underlying graphics drawing system to know when to start and stop displaying motion blur. The following is a possible algorithm for providing motion blur:

1. The application programmer calls the **Canvas::push_motion_blur** method.
2. All X level drawing primitives are stored in the correct order.
3. The application programmer calls the **Canvas::pop_motion_blur** method.

Figure 4.20: Smear motion blurring technique

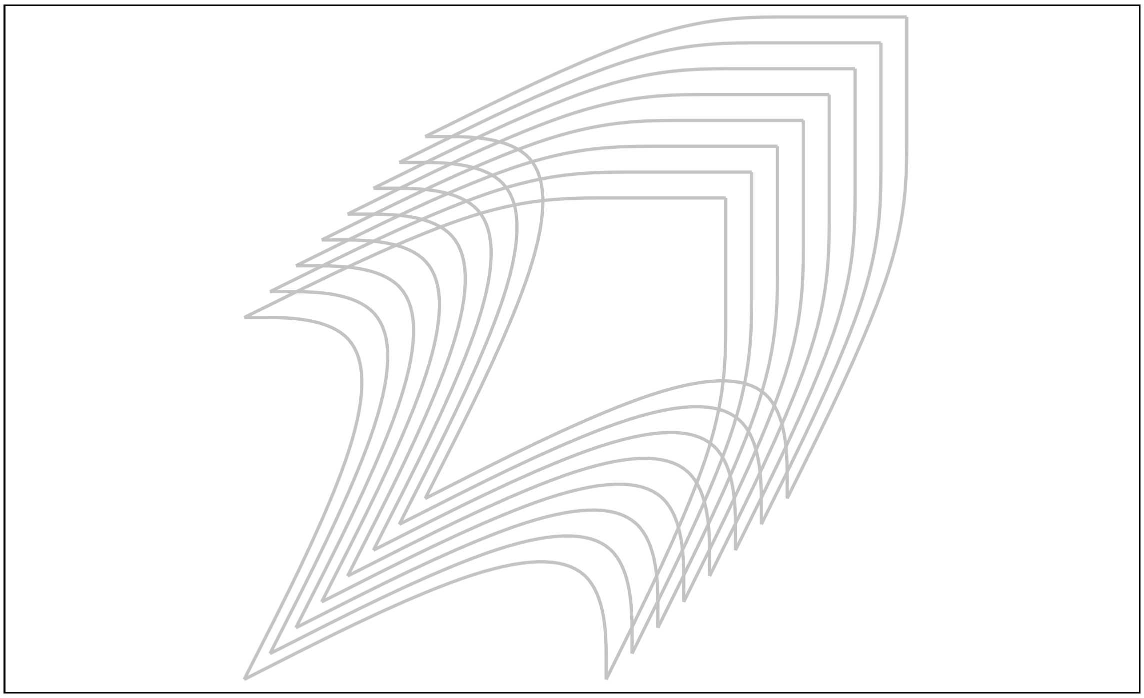


Figure 4.21: Stutter motion blurring technique

4. The Canvas draws a copy or copies of the graphical object in the motion blur style.

4.6 Efficiency

Drawing warped graphics is necessarily more expensive than drawing non-warped affine graphics, both because of the additional transformation step and because certain assumptions about properties of the distorted view, such as the straightness of "straight" lines, no longer hold. However, even a straightforward implementation of the warping and animation algorithms on standard workstation hardware has adequate performance for many applications. For example, a drawing editor used as the example application in this dissertation, running on a Sparcstation 2 with no special graphics hardware, achieved animation frame rates of around 10 per second for simple objects. This was possible even when the X11 connection was carried on a moderately loaded ethernet to a separate server.

Even the most elegant algorithm can fall prey to a large data set. To help prevent this, two forms of speedups for the graphics algorithms have been introduced. The first of these is an optimisation of warping effects for a common set of direct manipulation operations: translate, rotate, and scale. These three warping effects are used in most direct manipulation applications and serve as a model for optimisations of other warping effects. The second speedup is a set of methods to allow the application programmer to reduce the rendering time of a graphical object. To make these methods more general, and therefore more useful, they have been placed at the lowest level of the drawing system, the Canvas class. The remainder of this section describes these two forms of efficiency gains.

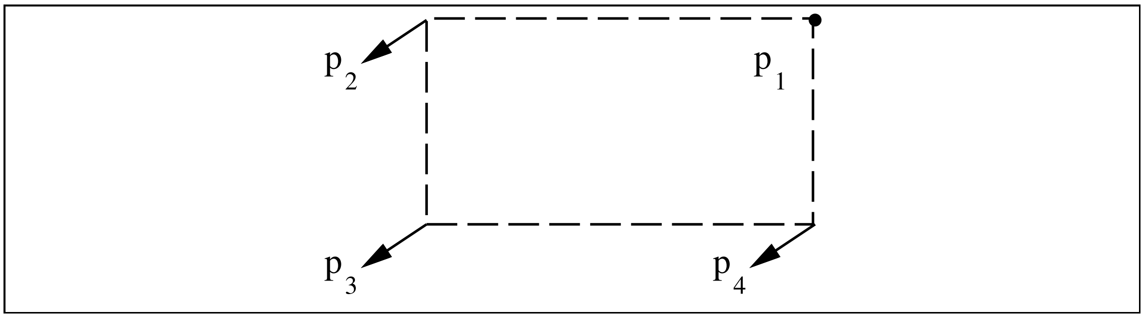


Figure 4.22: Symmetry of translation vectors

4.6.1 Optimisation of Warping Transformations

One technique for warp transformation speedup that has been implemented relies on symmetries in the warp vectors. The idea is to look for equal magnitude warp vectors at the time the transformation is applied. In such cases, some of the computation of the transformation displacement is redundant and can be avoided. Since the warp vectors used for many common manipulation effects often have equal magnitudes, the benefit can be significant.

For example, consider the warp vectors in Figure 4.22, typical of those used for a translate operation. Since $|p_1| = 0$ and $p_2 = p_3 = p_4 = p$, the computation of the transformation simplifies as follows:

$$\begin{aligned} q' &= q + \sum_{i=1}^{n} w_{qi} p_i && (4.8)\\ &= q + w_{q1} p_1 + \sum_{i=2}^{n} w_{qi} p_i && (4.9)\\ &= q + 0 + p \sum_{i=2}^{n} w_{qi} && (4.10)\\ &= q + p(w_{q2} + w_{q3} + w_{w4}) && (4.11) \end{aligned}$$

Although the complexity of the algorithm remains linear in the number of vectors, the simpler computation reduces the number of arithmetic operations. For other effects, some simplification occurs although the symmetries are fewer and thus the benefit less.

Another technique for warp transformation speedup that has been investigated is the substitution of a more easily drawn shape for the warped figures of path-based graphics [79]. For example, a Bézier curve could be substituded for each segment of a warped path, then the curve drawn as if no warping is present. In this way, the drawing only incurs the cost of the warping transformation when calculating the control points of the curve, a considerable saving over the usual cost of computing the transformation at many in-between points. For example, if a warped path segment is represented as a cubic Bézier curve, only four warp transformations are computed, the end points and two control points. This is in contrast to the many transformations that the true warping algorithm would typically need.

The approach works reasonably well under "mild" warp conditions, but the approximation of the Bézier curve to the warped curve degenerates for "severe" warps, which leads to unsatisfactory visual artifacts, such as coincident points not remaining coincident.

4.6.2 Controlling Drawing Time

Drawing complex graphical objects can slow down the rapid feedback needed to make a direct manipulation interface engaging. Making the graphical objects simpler to draw is one way to improve the speed of rendering. This in itself is not a new idea; many direct manipulation interfaces use a simple XOR enclosing rectangle instead of the full graphical object. The novelty of the approach I have taken is to control the reduction of drawing time over a continuum to simpler graphical objects. The idea is to reduce the complexity of the graphical object enough to make the rendering suitable for direct manipulation, while maintaining the maximum amount of visual information for the user.

These mechanisms are implemented at the lowest level of the graphics system, the InterViews Canvas class. I have implemented five drawing degradation meth-

```
void Canvas::roughness(Coord rf);
Coord Canvas::roughness();
void Canvas::thin_lines (boolean on);
boolean Canvas::thin_lines ();
void Canvas::no_images(boolean on);
boolean Canvas::no_images();
void Canvas::untransform_char(boolean on);
boolean Canvas::untransform_char();
void Canvas::small_object_area(Coord area);
Coord Canvas::small_object_area();
void Canvas::remove_small_objects(boolean on);
boolean Canvas::remove_small_objects();
void Canvas::push_enclosing_box();
void Canvas::pop_enclosing_box();
void Canvas::push_enclosing_box (int level);
void Canvas::engage_enclosing_box_level(int level);
void Canvas::enclosing_box(boolean on);
boolean Canvas::enclosing_box();
int Canvas::engage_enclosing_box_level();
```

Figure 4.23: Canvas methods for the reduction of drawing time

ods:

1. adjust the roughness of curves,
2. draw thin lines instead of thick lines,
3. draw transformed images and characters as if untransformed,
4. remove small objects, and
5. replace groups of objects with a bounding rectangle.

The methods are implemented as extensions to the Canvas class, and as such are available to the application programmer, see Figure 4.23. The idea is that an application that wishes to reduce the cost of drawing can choose an appropriate degradation method. The remainder of this section discusses in detail each of the five methods.

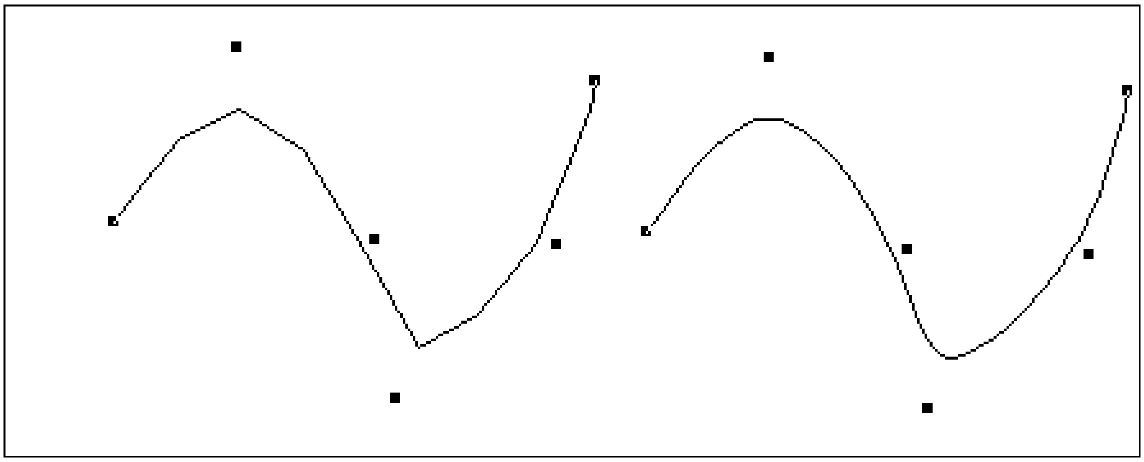


Figure 4.24: A rough curve (left diagram) and a smooth curve (right diagram)

Adjust Roughness of Curves

In InterViews, curves are approximated by a series of straight line segments. Drawing curves with fewer lines can reduce the cost of drawing complex objects. The toolkit has a roughness factor that determines how "roughly" a curve is drawn. Previously, the roughness factor was defined as a constant. I extended the Canvas class to include methods to set and test the roughness factor.

Figure 4.24 shows the effect of a curve drawn with a few lines in the left diagram compared to the curve with more lines in the right diagram. InterViews Canvas draws curves as third order Bézier curves. The algorithm checks to see if the Bézier end points and control points are approximately in a straight line. If so, it draws a line between the end points; otherwise it bisects the curve and recursively draws each half.

Increasing the roughness factor relaxes the required straightness of the line, and decreases the number of times the curve is bisected. This roughness factor affects both the curves and the warping of stroked figures.

Draw Thin Lines

To improve the efficiency of drawing complex stroked figures, I added a binary setting to the InterViews Canvas class that can force all line objects to be drawn with thin lines. In X11, a thin line is specified as a line with a width of zero and is drawn with a thickness of one pixel of the display device. As stated in the "Xlib Programming Manual" by Adrian Nye [45], thin lines are the fastest way to draw a line, and are implemented with an unspecified device-dependent fast algorithm. A second feature of this binary setting is that all lines are drawn as solid lines. The setting is controlled by methods added to the extended Canvas class to set and test the thin line effect.

Draw Transformed Images and Characters as if Untransformed

Drawing transformed colour raster images or single-colour bitmaps is expensive; I added additional controls to Canvas to adjust the drawing of these objects. A simple speedup can be achieved by replacing images with a simple rectangle of the same size as the image. A second speedup is to display text characters without any rotation or scaling transformations, thus allowing the more efficient X Window text display routines to be used. In the original InterViews toolkit, characters that have been scaled or rotated are displayed as bitmaps, and in all other cases they are displayed with standard X Window text display routines. Figure 4.25 shows an example of rotated text in the left diagram and rotated text with characters drawn untransformed in the right diagram. Note the characters retain their spatial position, but they are all in their normal vertical orientation. Methods have been added to the Canvas class to support these two operations.

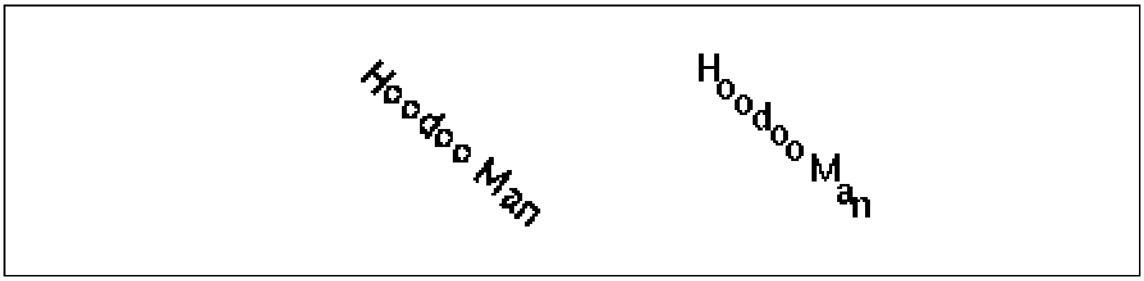


Figure 4.25: Example of untransformed rotated text

Remove Small Objects

A common algorithm to speed up the drawing of complex objects is to remove smaller graphical objects in the drawing and replace them with simpler graphical objects (sometimes referred to as *greeking*.) A simple strategy is to replace objects below a given area threshold with a bounding box. The user is able to maintain the overall context of the drawing because of the larger graphical objects remaining. This method is based on the idea that large graphical objects supply the user with sufficient information to "fill-in" the gaps from their memory of the complete drawing, and that smaller changes to the visual display are less disruptive to the user.

The new **Canvas::small_object_area** method sets the threshold area size in square printer's points [3]. This threshold area sets the size of the graphical objects to be replaced by enclosing rectangles. The algorithm allows for a continuous range of thresholds from the size of the drawing surface to an area of zero. The **Canvas::remove_small_objects** method activates the replacement process. Other methods return the current settings for the respective values.

Replace Groups of Objects with a Bounding Rectangle

A similar concept to that of removing graphical objects with small areas is the grouping of objects together and then rendering them with a simpler graphical

[3]InterViews uses a floating point coordinate system in standard printer's points, one printer's point is equal to $\frac{1}{72}$in.

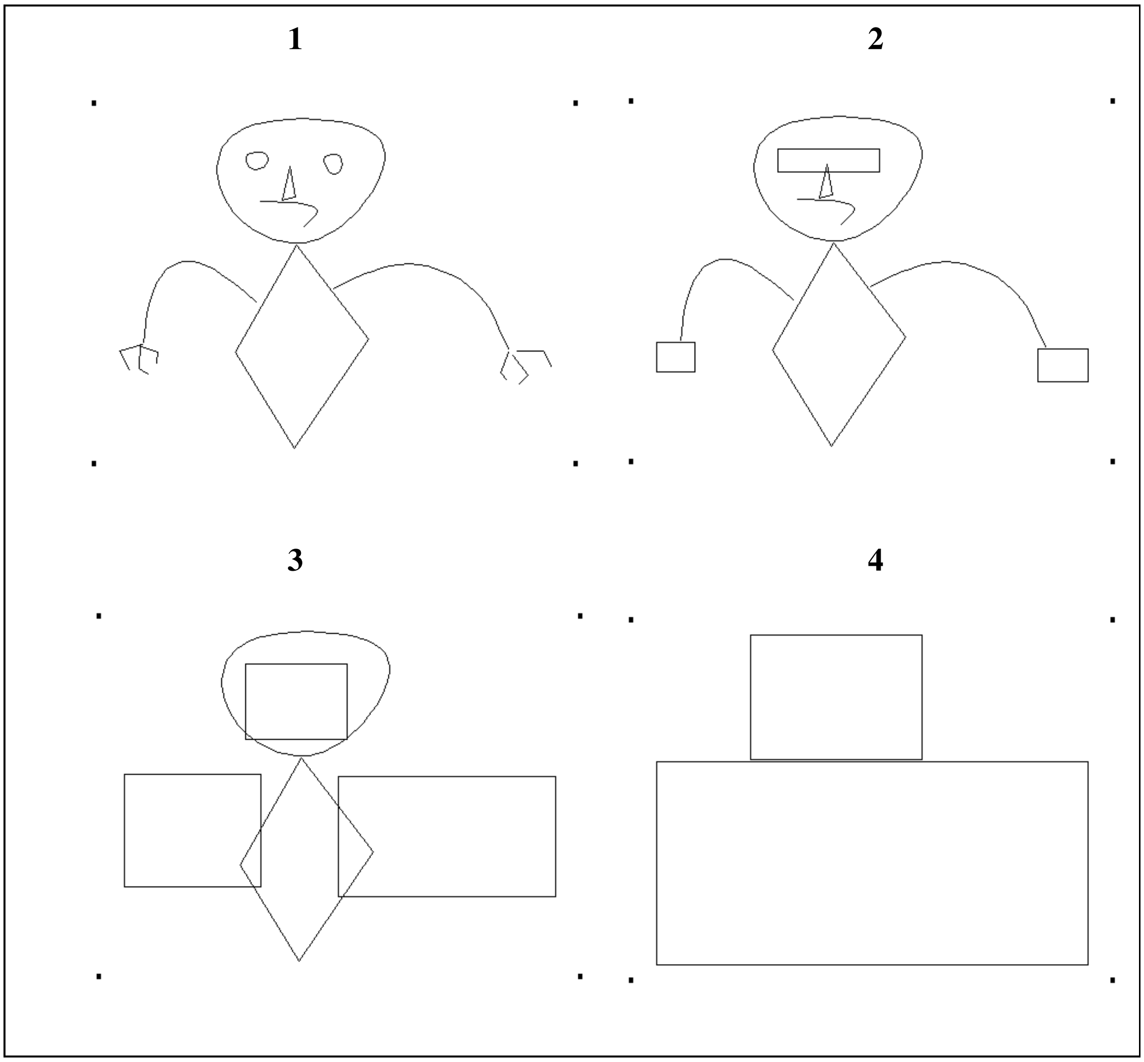


Figure 4.26: Example of enclosing boxes for a complex graphic

object. As with the previous effect, the simpler graphical object is an enclosing rectangle. Figure 4.26 shows an example of four different levels of grouped objects. Diagram 1 shows the full graphical object. In diagram 2, the "hands" and "eyes" have been grouped together and replaced with rectangles. Diagrams 3 and 4 remove more of the detail.

The methods to mark the beginning and end of the graphical objects groups are **Canvas::push_enclosing_box** and **Canvas::pop_enclosing_box**. **Canvas::push_enclosing_box** sets the level at which the enclosing boxes are then to be substituted. The level can also be set with the **Canvas::engage_enclosing_box_level** method. To start and stop the effect, the **Canvas::enclosing_box** method is called. The values noting whether the effect is on and the current level at which the enclosing boxes are substituted can be retrieved with other methods.

4.7 Summary

The animation effects described in Chapter 3 can be easily incorporated into applications by providing toolkit support for warped drawing, blurred drawing, drawing speed control, interpolation, and event recording. To show how these tools can be incorporated in an existing toolkit, I extended the InterViews user interface toolkit. The extended InterViews supports pseudo 3D drawing, a coordinate warping transformation, event recording and interpolation, and control that trades drawing quality for drawing speed. Motion blurring is not currently implemented.

Chapter 5

Implementing Animation in Graphical User Interfaces

To support the animation effects from Chapter 3, I used the algorithms and the InterViews toolkit extensions described in Chapter 4 to implement an animated widget kit, and warping effects for a range of direct manipulation and indirect manipulation operations.

5.1 Smoothness in the Interface

Overall the InterViews toolkit provides an excellent basis on which to build animations that smooth transitions in a user interface: the widget kit mechanism offers an easy way to try out the new smoother components in the context of existing applications; and the underlying graphics model is both powerful enough to draw the objects and efficient enough to achieve acceptable speed, even with low-end hardware. This section describes several of the implementation issues in building autonomous animations to smooth the transition of interface components. An overview of the Animated Widget Kit, which encapsulates the animated widgets previously, described is presented. This overview leads on to the implementation details of the software tools for providing keyframe animation by geometric transformations. Finally, the algorithms for constructing and operating the palette menu widget are studied in detail as an example of a complex animated widget.

5.1.1 The Animated Widget Kit

The Animated Widget Kit [74, 78] is an extension of the InterViews widget kit [39]. InterViews widgets are created by an object factory called a WidgetKit; application programs call WidgetKit methods to create standard widgets such as menus, buttons, and scrollbars. This approach insulates the application from the detailed implementation of the widgets and gives it a degree of independence from the specific look-and-feel of the components. For example, an application might call the WidgetKit PushButton method to create an instance of a push button widget. In general, the application does not care whether the object returned is a Motif-like push button or an OpenLook-like push button; it just incorporates the component into the structure of the interface. Indeed, the InterViews mechanism provides several types of WidgetKit, each producing widgets that conform to a different look-and-feel standard, and it allows the choice of the type of push button created to be deferred until run-time. The user can specify the look-and-feel of the application in a preferences file or even as a command line argument.

This indirect way of creating widgets provides an easy way to incorporate animation into existing applications; simply create a new type of widget kit and make it available as another run-time look-and-feel choice. This approach allows new animated widgets to be tested in existing InterViews-based applications without needing to modify the application code in any way. The Animated Widget Kit extends the original WidgetKit in two ways: animated versions of the standard components are included, and several new kinds of widgets are added to the kit.

To demonstrate these animated widgets, I created an expanded version of the idemo[1] application that creates instances of the new types of widgets offered by the animated kit. Figure 5.1 shows several of the new widgets: the push buttons "bounce" slightly when pushed, the scroll bars slide smoothly instead of jumping, and the menus grow smoothly instead of suddenly popping open at full size. The selected radio buttons are shifted to the left, and check boxes swap sides with their labels when selected. Some of the palette buttons have been replaced with buttons that twist when selected. The palette button labelled "Static" changes its label character by character to "Dynamic." The figure also shows a palette menu (on the right) that uses animation to switch between several sets of choices of fill patterns, as might be found in a graphics editor. I chose a subset of possible animation effects to investigate the usability animation in a graphical user interface, but other effects such as anticipation and follow-through could be incorporated into the widget kit.

5.1.2 Geometric Transformations

The animated versions of the palette buttons, check boxes, radio buttons, and palette menus all use the Path, Path_Item, and Path_Animator classes. The geometric transformations used for the buttons and menus are specified using a 3D coordinate system. Each end point of an animation path is defined with a DDDtransformer object, which can define arbitrary scalings, translations, and rotations in 3D. The frames that animate the transition from one point to another along the path are calculated by interpolating the end point matrices. Note that this approach is far from a full scale 3D graphics rendering model: it does not support perspective, lighting and shading, or many other essential components of such systems. Nevertheless, it is powerful enough for these simple animation effects and can be implemented without special hardware or software.

The animated versions of the widgets use the slow-in/slow-out effect at the

[1]The idemo application is a "periodic table" of the major interface components from the InterViews widget kit.

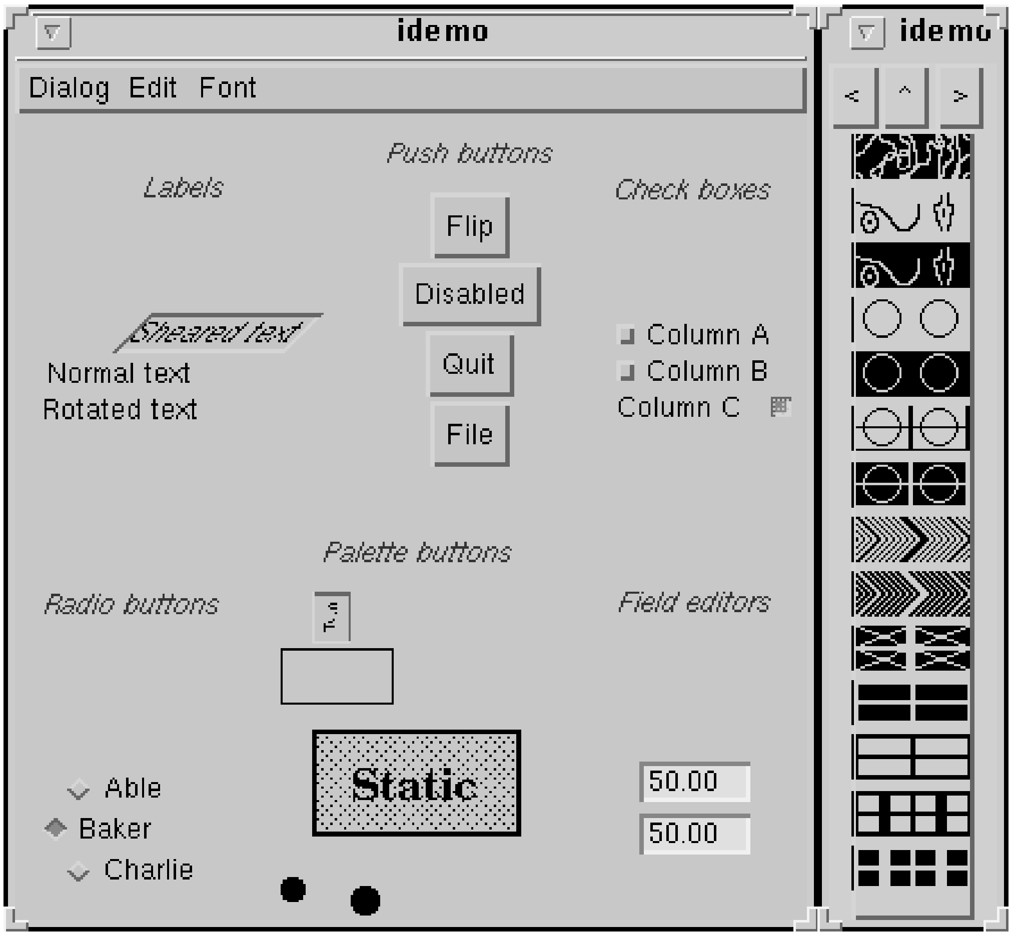


Figure 5.1: Enhanced idemo with animated widgets

beginning and the end of their animation sequence. This effect involves moving objects more slowly at the beginning and end of their movement than in the middle. Bézier functions are used to define this effect for the geometric transformation keyframe animations, by mapping over time the percentage of change in location, orientation, and size for each drawable object in the intermediate frames.

5.1.3 Palette Menu

The *palette menu* is composed of three buttons and a Cube object, which in turn is constructed from panels of standard InterViews widgets, forming the sides of a rectangular prism. The rotation of the entire Cube object is achieved simply by a 3D transformation.

```
class Cube {
public:
    Cube( Glyph * side1, Glyph * side2,
          Glyph * side3, Glyph * side4,
          Glyph * top,   Glyph * bottom );
    virtual ~Cube();
    void set_draw(float X, float Y, float Z);
};
```

Figure 5.2: Cube C++ class definition

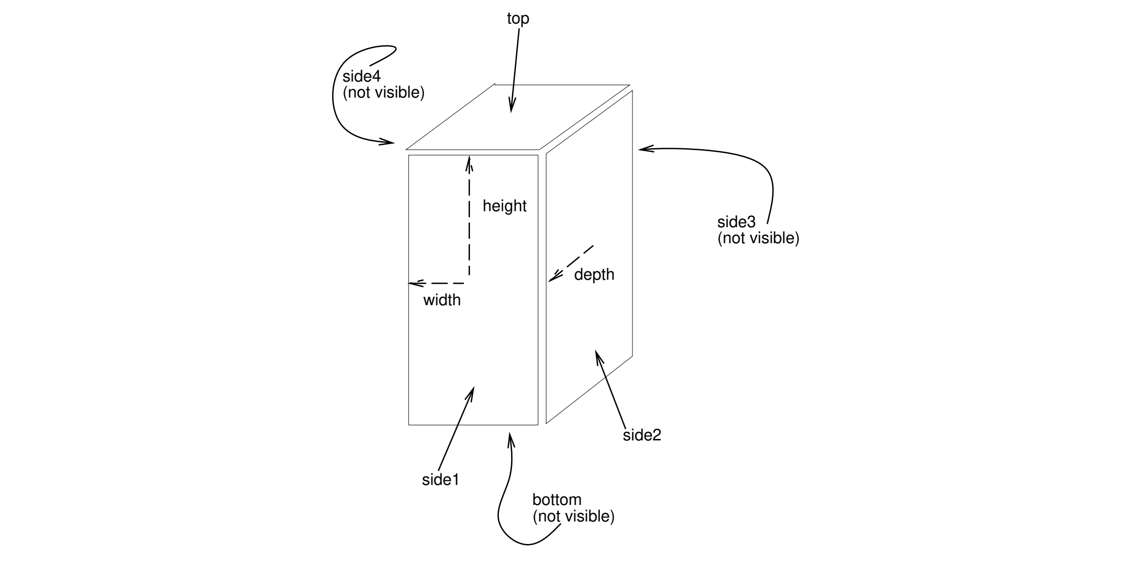


Figure 5.3: Cube dimensions

The C++ class definition for the Cube is shown in Figure 5.2. The Cube is constructed from six InterViews glyphs (drawable objects), passed as parameters. These six glyphs are used to build a group of six Face objects, which are arranged into the configuration shown in Figure 5.3. In the case of the palette menu example, the Face objects side1, side2, side3, and side4 are boxes containing buttons with bitmap patterns as their labels.

The original orientation of the Cube object is with side1 exposed and parallel to the x-y plane (the window plane), while all other glyphs are hidden. To change the orientation of the Cube, the **Cube::set_draw** method applies absolute rotations of the amount given as an angle parameter, in the range of 0 to 360 degrees. The rotations are applied as post matrix multiplications in the order y axis, x axis and then z axis.

The algorithm to draw the Cube requires each of the faces of the Cube to be placed in the correct location and orientation. The location of the lower left corner of each face object, before the 3D transformations, is defined as the Cube's origin ($x = 0,\ y = 0,\ z = 0$). The pseudo-3D transformations require the transformed graphical object to initially be on the $z = 0$ plane and to be projected back onto the $z = 0$ plane. The algorithm for the 3D translation and rotation of each face on the Cube object is shown in Figure 5.4. The *width*, *height*, and *depth* values are defined in Figure 5.3.

5.2 Manipulated Objects with Substance

The animation effects for the translation, rotation, scaling, and shape editing direct manipulation operations are all implemented by warp transformations. The basic process of drawing the warped objects is as follows:

1. Calculate the derived warp vectors and make a Warp object.
2. Make a copy of the Transformer object on the top of the Canvas Transformer stack.
3. Apply the Warp to the Transformer.
4. Push the Transformer onto the Canvas Transformer stack.
5. Draw the graphical object.
6. Pop the Transformer off the Canvas Transformer stack.

The above process is the same for each of the four effects. The only difference is that the calculation of the warp vectors varies between shape editing and the other three effects. This section describes how to calculate the set of warp vectors for each

```
draw_cube
    draw side1_face
        translate side1_face to (0.0, height, depth)
        if side1_face is visible draw it
    draw side2_face
        rotate side2_face about its centre 90.0° around the y axis
        translate side2_face to (depth, 0.0, 0.0)
        if side2_face is visible draw it
    draw side3_face
        rotate side3_face about its centre 180.0° around the y axis
        translate side3_face to (width, 0.0, 0.0)
        if side3_face is visible draw it
    draw side4_face
        rotate side4_face about its centre −90.0° around the y axis
        translate side4_face to (depth, 0.0, 0.0)
        if side4_face is visible draw it
    draw top_face
        rotate top_face about its centre 90.0° around the x axis
        translate top_face to (width, 0.0, 0.0)
        if top_face is visible draw it
    draw bottom_face
        rotate bottom_face about its centre −90.0° around the x axis
        translate bottom_face to (width, 0.0, 0.0)
        if bottom_face is visible draw it
```

Figure 5.4: Draw cube algorithm

of these effects. The first section describes the calculation of the warp vectors for translation, rotation, and scaling. The next two sections describe the calculation of warp vectors for rectangle resizing and editing a vertex. The final section describes the implementation issues for supporting follow through animation effects to help smooth these effects.

5.2.1 Translation, Rotation, and Scaling

The calculations of the warp vectors for the translation, rotation, and scaling effects all use the same set of algorithms. Three different methods of calculating the warp vectors have been investigated: *previous position in time*, *inertia*, and *friction*. Each of these provide a different look-and-feel to the warping of an object. The *previous position in time* method stretches the vertices not grasped by the user back to their position at a set time interval in the past. The *inertia* model applies a force to the object at the grasped point and considers the graphical object itself as having the elastic property of a spring. Each of the vertices not grasped by the user are stretched away from the grasped point as if in the direction of a reactive force. The *friction* method models a simple static friction force. The grasped point has to stretch a defined distance away from the object before the object starts to move. Dynamic friction, the friction occurring once an object starts moving, is modelled in the same way as static friction in this system.

As an example of how each of these methods change the visual effects of warping, Figure 5.5 shows each of the methods applied to a square object dragged along a curving path. The top diagram shows the operation carried out with no warping. For the previous position in time method, the object is warped in the direction of the object's path, as shown in the next diagram of the figure. The third diagram of the figure shows the inertia method the object responds to a change in direction by swinging outside the path of the mouse. Finally, in the bottom diagram of the figure, the friction method has the square move only after the grasped point has stretched past the defined minimum distance.

Each of the methods performs the following steps for each frame:

1. Determine the new configuration for the object.
2. Compute an "old" position for the object, based on the particular method.
3. Place a vector that points from the grasped vertex of the new object to the mouse point.
4. Place vectors that point from each other vertex of the new shape to the corresponding vertices of the old shape.

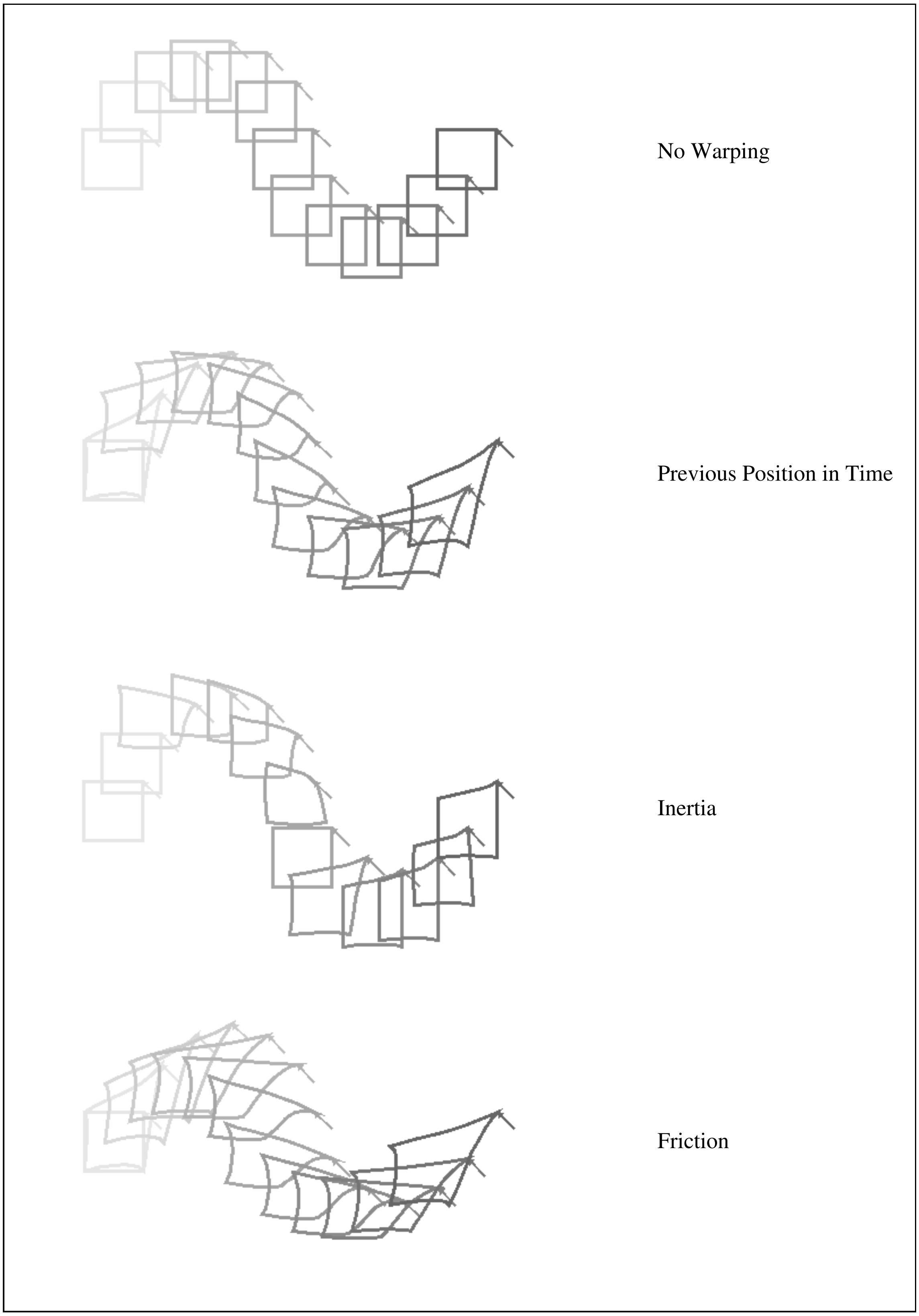


Figure 5.5: Previous time, inertia, and friction warp effects

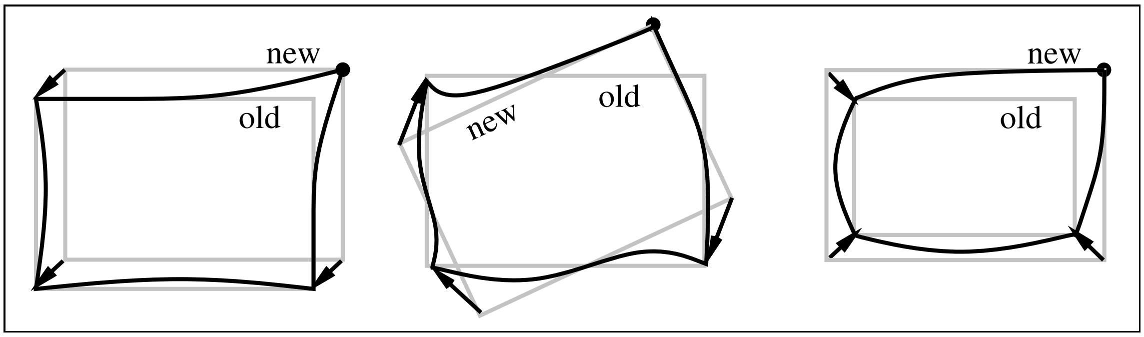


Figure 5.6: Calculating warp vectors for translation, rotation and scaling

The previous position in time method uses a simple algorithm that nevertheless gives very convincing results for common shapes and operations. Figure 5.6 shows the set of vectors for typical translate, rotate, and scale direct manipulation operations; in each case the rectangle is grasped by its top right corner. The grey outlines show a rectangle at its old and new configurations. The black dots on the top right corners represent the zero warp vectors at the mouse point, and the arrow-headed lines show the positions and sizes of the other warp vectors. Finally, the solid strokes show the shape of the distorted rectangle that is drawn under the warped transformation. The "new" configuration is determined as if no animation is in effect. The "old" configuration is simply the configuration of the shape at some time in the past.

Section 4.4.1 describes the history mechanism for storing translation, rotation, and scaling information, which includes a means of interpolation to retrieve this information at arbitrary points of time in the past. At each user initiated event that changes the configuration of a graphical object, the algorithm stores the current position, orientation, and scale factor and retrieves from the history object information about the configuration at a previous time. Then it uses the current and previous configuration data to construct current and previous transformation matrices. Finally, it transforms each vertex of the old and new shape to compute the warping vectors.

The *inertia* method uses an algorithm that more closely models the physical

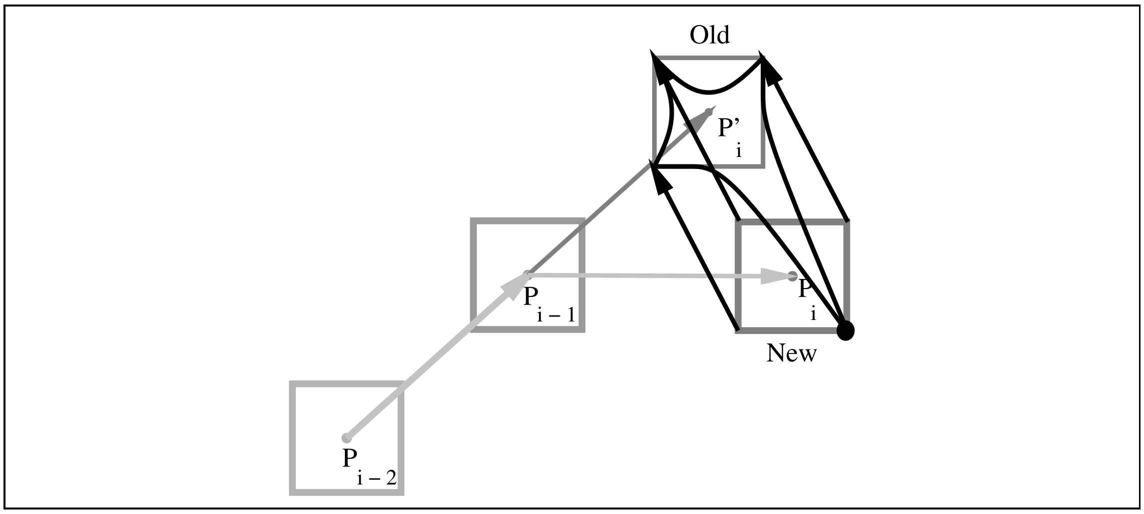


Figure 5.7: Calculating inertia method warp vectors for translation

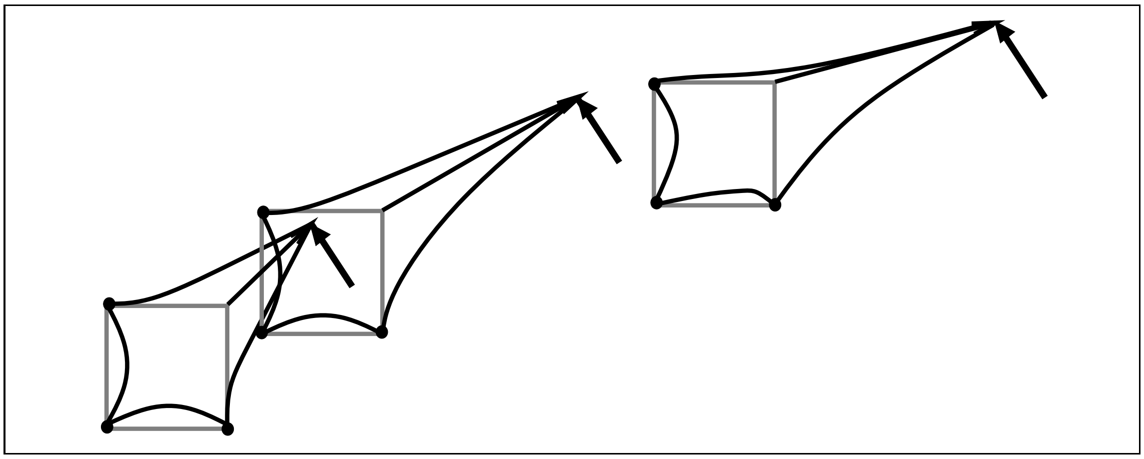


Figure 5.8: Calculating friction method warp vectors for translation

world. The "new" configuration is determined as if no animation is in effect. Figure 5.7 shows a few frames from a translation operation with the object at positions p_i, p_{i-1}, and p_{i-2}. The "old" configuration, p'_i, is determined as if the graphical object were to continue with the velocity it was experiencing at p_{i-1}. The grey arrow-headed line between p_{i-1} and p_{i-2} indicates the displacement between these points and serves as a velocity vector; the positions retrieved from the history are at regular intervals of time. The inertia "feel" to this method comes from the object appearing to maintain its previous velocity.

The *friction* method simulates a frictional force that tends to keep the object from changing position. In operation, the grasped vertex must move a minimum

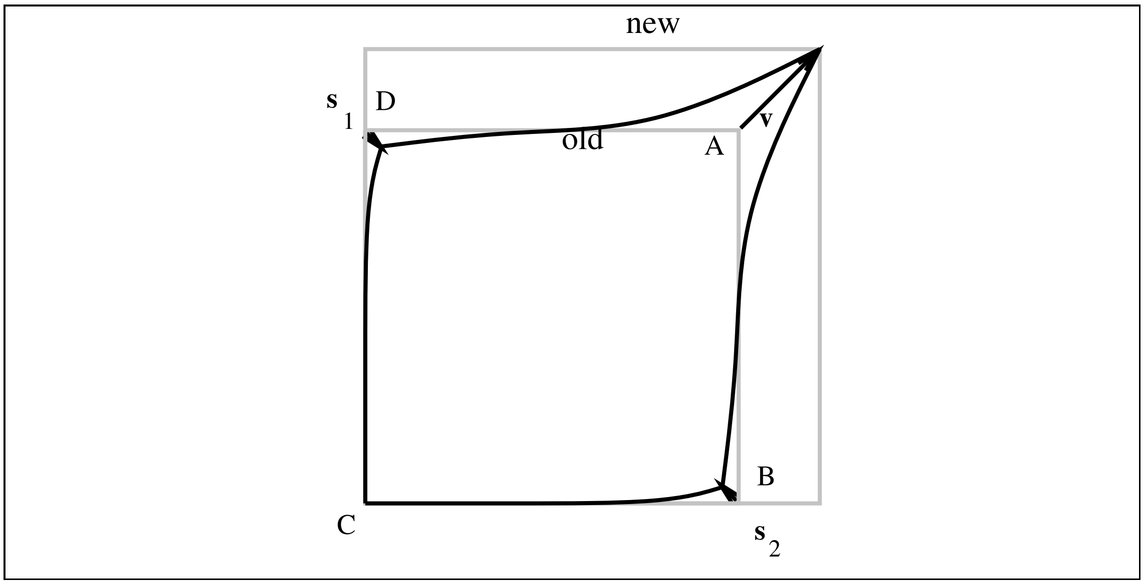


Figure 5.9: Constructing a growing rectangle effect

distance before the object as a whole starts to move. The effect is as if the object trails behind the mouse by a fixed distance. The "new" configuration is determined by moving the object in the direction of the mouse pointer so that it is no further than the specified distance. If the mouse is less than that distance, the object does not move. For the friction method, the "old" configuration is the same as the "new" configuration. Therefore, there are zero vectors at all vertices except the grasped one and a vector that points from the grasped vertex to the mouse point. Figure 5.8 shows this idea with three frames from a translation operation with a frictional animation effect. The grey squares indicate the "new" configurations of the objects.

5.2.2 Rectangle Resizing

The regular nature of rectangle resizing allows for a more complex animation effect. The "stretch and squash" principle can be more fully implemented to adhere to the animator's rule of thumb – the cross sectional area of an object should remain constant during a stretch or squash operation.

The set of warping vectors for resizing rectangles deforms the graphical object

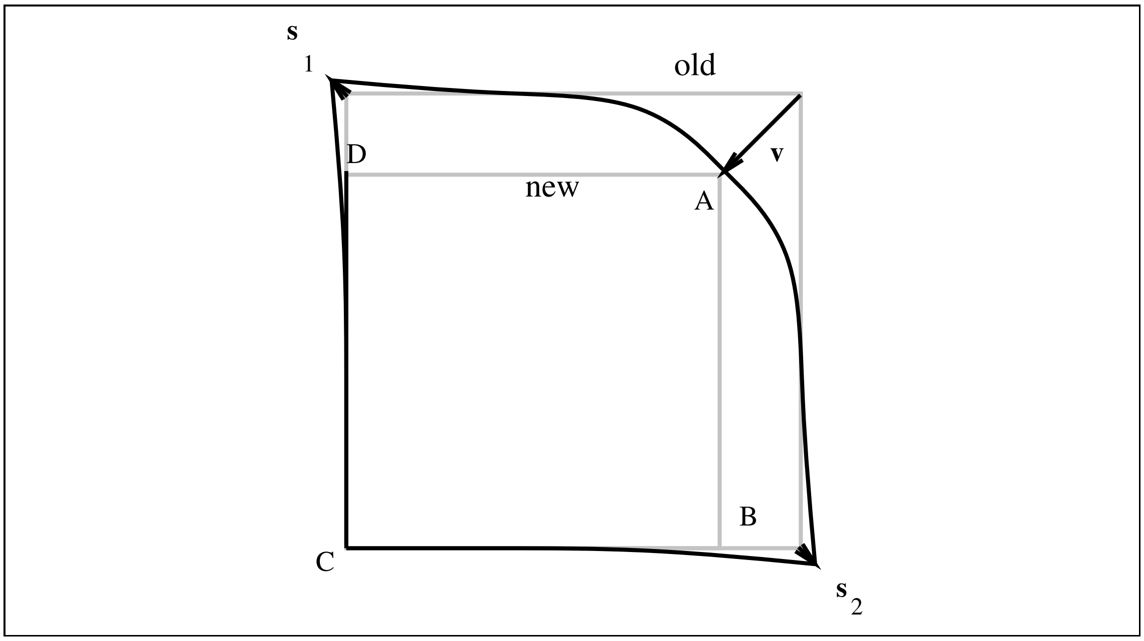


Figure 5.10: A shrinking rectangle effect

by stretching out the manipulated vertex and pulling in the two adjacent vertices, as described in Section 3.3.3. Figure 5.9 shows the configuration of warp vectors at the vertices A, B and D that produce this effect. A warp vector of zero magnitude is placed at vertex C and a vector, v, is placed from vertex A to the mouse point. The size of the warp vectors $\mathbf{s_1}$ and $\mathbf{s_2}$ at the vertices B and D is determined by the cursor displacement vector, $\mathbf{v}$, and a stiffness factor, f, as follows:

$$\mathbf{s_1} = f \cdot (v_x, -v_y) \text{ and} \tag{5.1}$$

$$\mathbf{s_2} = f \cdot (-v_x, v_y). \tag{5.2}$$

These warp vectors $\mathbf{v}$, $\mathbf{s_1}$ and $\mathbf{s_2}$ are applied to the position of the vertices A, B and D in the previous time frame.

The above equations are appropriate for the effect of pulling a rectangle in or out. Figure 5.10 shows the process in reverse; the user is making the rectangle smaller. Notice the grasped corner is pushed into the graphical object, and the upper left hand and lower right hand corners are pushed out.

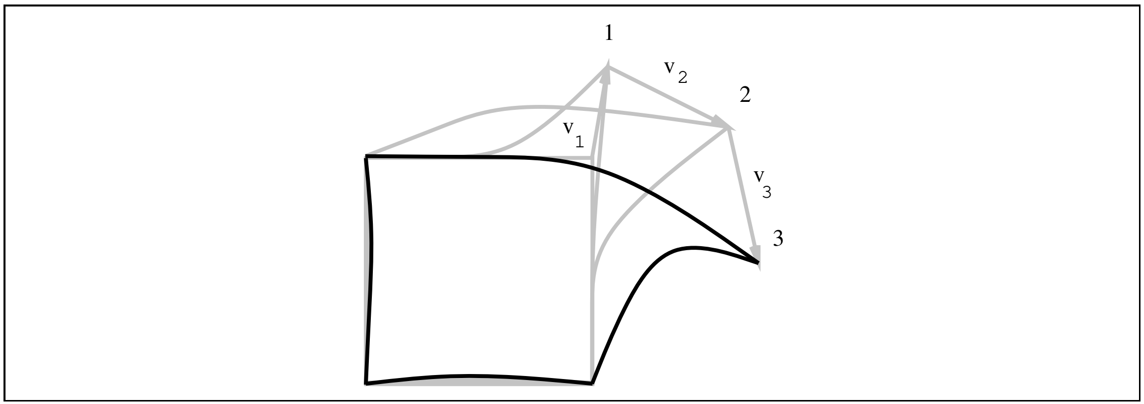


Figure 5.11: Vectors for animating a vertex editing operation

5.2.3 Vertex Editing

A vertex editing operation is one in which a single vertex of an object is moved. For example, Figure 5.11 depicts the editing of the upper right vertex of a square. For each frame in the animation, the graphical object is drawn with the vertex being edited in its previous position, but with a warp vector from the last position of the grasped vertex to the cursor position and zero vectors at all other vertices. Figure 5.11 shows the three vectors, v_1, v_2, and v_3, for three different animation frames during the edit vertex operation. When the operation is finished, the object is drawn unwarped with the vertex in its final position.

5.2.4 Follow Through

The follow through effect can be used to remove artifacts of rapid mouse movements while manipulating an object over a large displacement. For example, if there is a large mouse movement between events followed by no mouse movements, a warped object can appear to be frozen in the warped state. This problem can be solved by having the application continue to move the object with a follow-through effect until it has caught up to the cursor. The time during which the user is not initiating input events is referred to as *idle time* (Artkit uses a similar term.) The animation effects produced during idle time allow the application to smooth the animation so

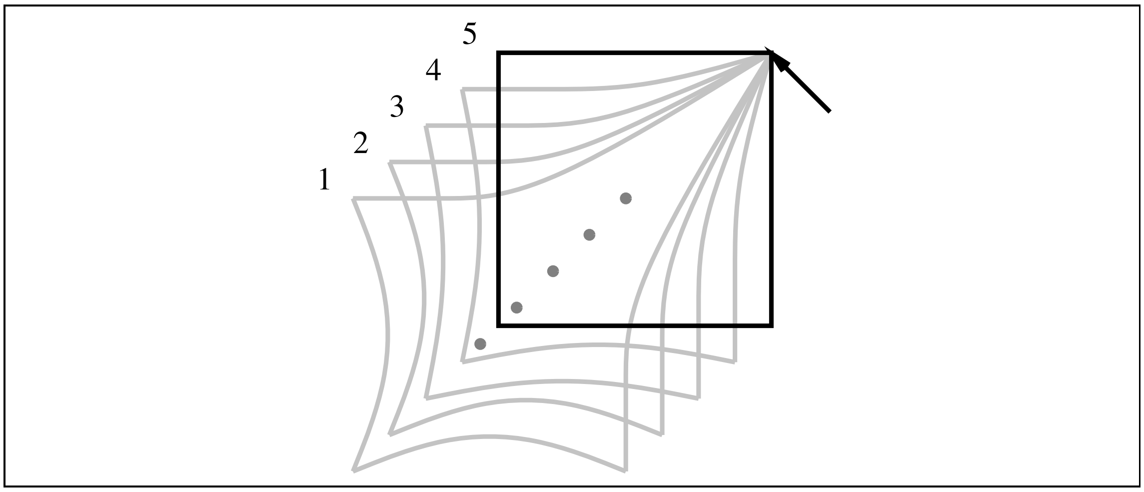


Figure 5.12: Follow through effect during an idle state

that an object has a more "realistic" feel.

An example of this follow through effect can be seen in Figure 5.12. The grey object '1' in the lower left has been warped and the cursor is positioned in the upper right of the object. The cursor remains stationary over the next four time frames. The object is animated to move to a position in which it is not warped, indicated with the black object '5'. The grey circles in the figure indicate the centre of the object as it moves through the animation. This example uses a linear interpolation of the position, but the Bézier time function described in Section 4.4.3 could be used to achieve a slow-in/slow-out effect.

Similar animations can be constructed for rotating, scaling, and shape changing. When the application determines that it is in an idle state (that is, when the user stops moving the cursor), a warped object is created as the start keyframe of the animation and an unwarped object is created as the end keyframe. The system proceeds to calculate a set of interpolated in-between frames and displays them at a predetermined frame rate.

Section 4.4.4 describes how timers can facilitate follow through effects under an event based programming paradigm. Consider for example the move manipulation. When the user drags the object, the following algorithm is executed for every X

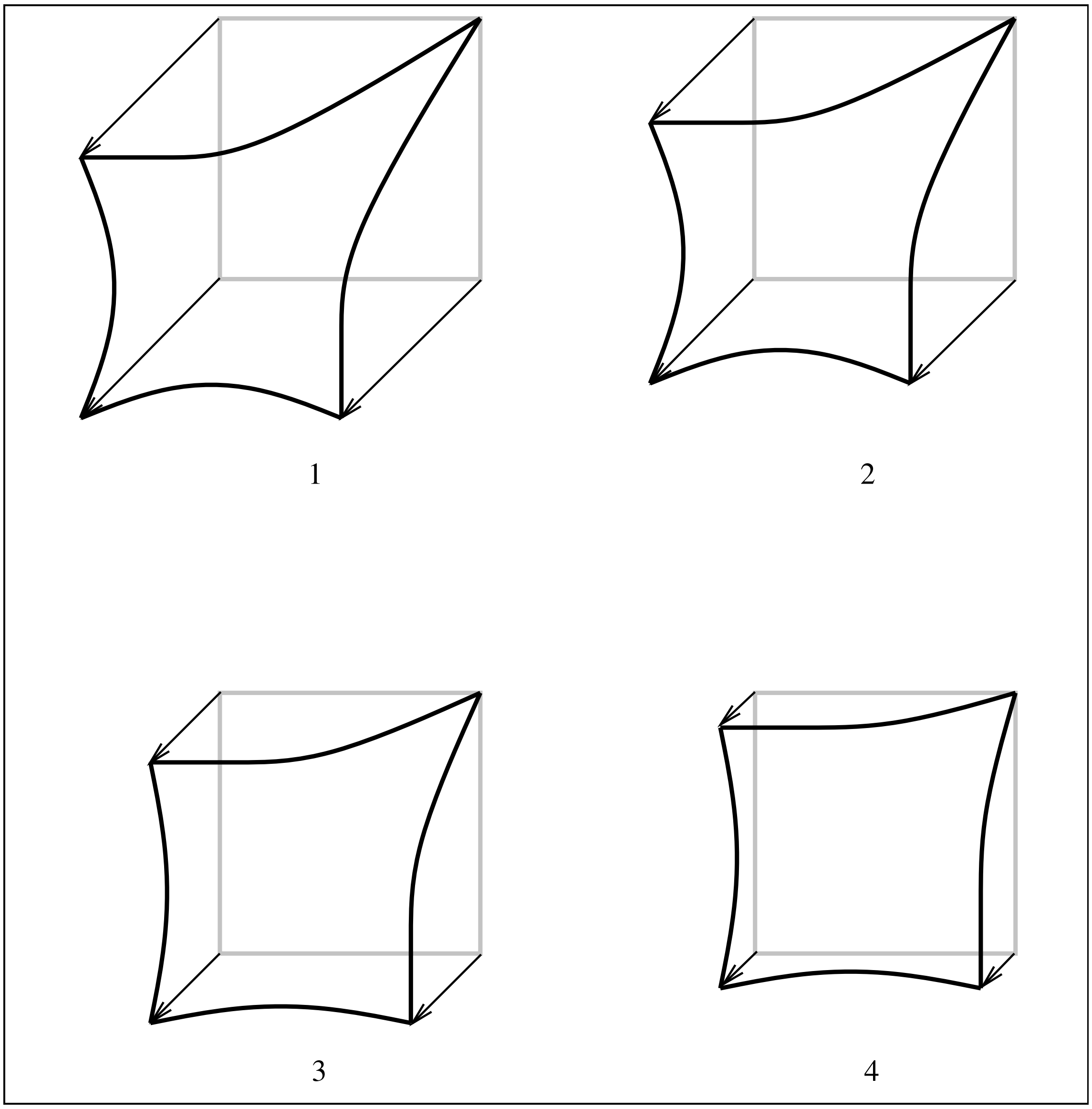


Figure 5.13: Vector set for a follow through effect during an idle state

event during the translation process.

1. Stop the timer.
2. Move the object.
3. Store the object's position in the history.
4. Redraw the object.
5. Reset and start the timer.

At the start of each individual move operation, the timer is stopped. The objects are then moved and redrawn in the new locations. The timer is reset and started again. If the timer goes off (because the object has not moved within the given time) a callback routine is invoked. The algorithm for the callback routine is as follows:

1. Retrieve positional information at a defined past interval of time.
2. For each moving object make and apply the correct Warp object.
3. Redraw each moving object.
4. Add event information to time history.
5. If the animation has completed
   (a) Then redraw each object without warping.
   (b) Else reset the timer.

Figure 5.13 show the vectors for each of the animation frames in Figure 5.12. A zero magnitude vector is placed at the grasp point (upper right hand corner), and the vectors defined by the difference between the past position and current position are placed at all other vertices.

5.3 Anticipating the Result of a Manipulation

This section describes the implementation of animation effects for the *pinning*, *snap and drag*, and *indirect manipulation* operations described in Section 3.4. The implementation uses warp vectors for each of the three anticipation animation effects. All of these effects use a common *pinning* effect to help provide visual feedback about the constraints, and they use animations to smooth the transitions.

5.3.1 Pinning

The simplest animation effect for conveying the sense of anticipation is the pinning effect, described in Section 3.4.1. This effect causes the ungrasped vertices of the

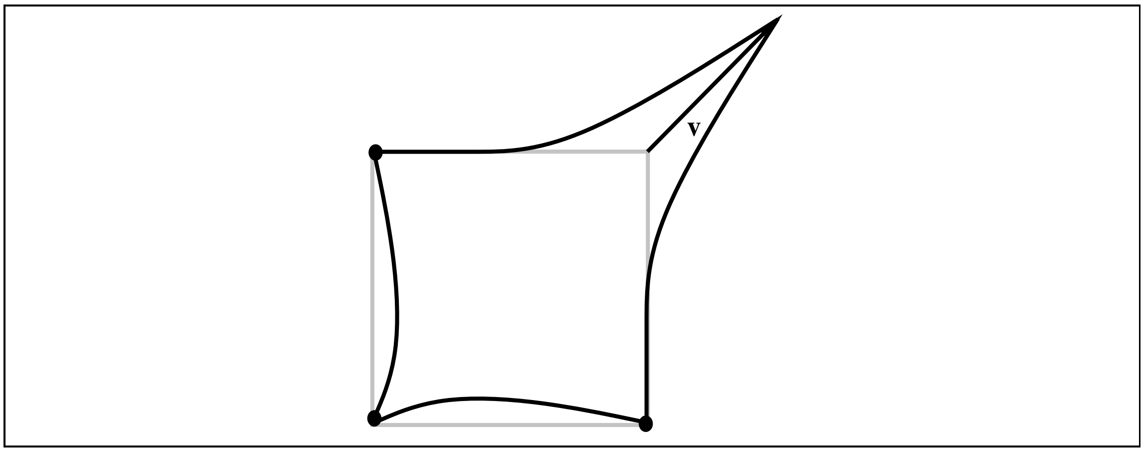


Figure 5.14: The set of warp vectors for a pinning operation

graphical object to remain fixed to the drawing surface, while the grasped vertex is moved by the user. The bowing of the edges of the graphical object as the vertex moves from its original position suggests the effect of the constraint. The creation of this effect involves two animations: the animation of the direct manipulation of the user's interaction with the vertex, and the animation of the grasped vertex back to its original position, when it is released.

Figure 5.14 shows the configuration of the warp vectors for the pinning effect during the manipulation of the vertex. The dots represent zero warp vectors at the pinned vertices. The grey outline shows the original square and the black figure represents the final warped object. The warp vector v is calculated from the difference between the current position of the cursor and the original position of the grasped vertex.

The snap back animation is triggered when the user releases the grasped vertex. The algorithm uses a linear calculation for the change in the warp vector, $v \cdot \frac{F'}{F}$, but the Bézier timing function in Section 4.4.3 may be used to produce a slow-in/slow-out effect. The algorithm for the animation is as follows:

Set $F' = F$ the number of frames in the animation and start the timer.

1. Calculate a Warp object with a warp vector of $v \cdot \frac{F'}{F}$.
2. Apply the warping to the pinned object.

3. Redraw the pinned object.

4. If $F' = 0$

   (a) Then redraw the pinned object without warping.

   (b) Else reset the timer and decrement F'.

5.3.2 Snap and Drag

The snap and drag effect uses the pinning effect to indicate the grid points to which the graphical object is currently snapped. Again the implementation uses two animations: the animation of the direct manipulation of the user's interaction with the vertex, and the snapping animation of the graphical object snapping to a set of grid points.

The configuration of the warp vectors is identical to those used for the pinning effect. The only difference is that the warp vector v is determined by the difference between the current position of the cursor and the most recent position of the snapped grasped vertex.

The triggering of the snapping animation occurs when the user moves the grasped vertex past a defined distance from the position of the snapped grasped vertex. The algorithm for snapping a given point (x, y) to a grid is done in two parts, one for the x dimension and the other for the y dimension. The final snap point in the x dimension for a point between grid points x_i and x_{i+1} is as follows:

$$x = \begin{cases} x_i & if\ x \leq \left(\frac{x_i + x_{i+1}}{2}\right) \\ x_{i+1} & if\ x > \left(\frac{x_i + x_{i+1}}{2}\right) \end{cases} . \tag{5.3}$$

The equation for the y dimension of a point between grid points y_i and y_{i+1} is similar:

$$y = \begin{cases} y_i & if\ y \leq \left(\frac{y_i + y_{i+1}}{2}\right) \\ y_{i+1} & if\ y > \left(\frac{y_i + y_{i+1}}{2}\right) \end{cases} . \tag{5.4}$$

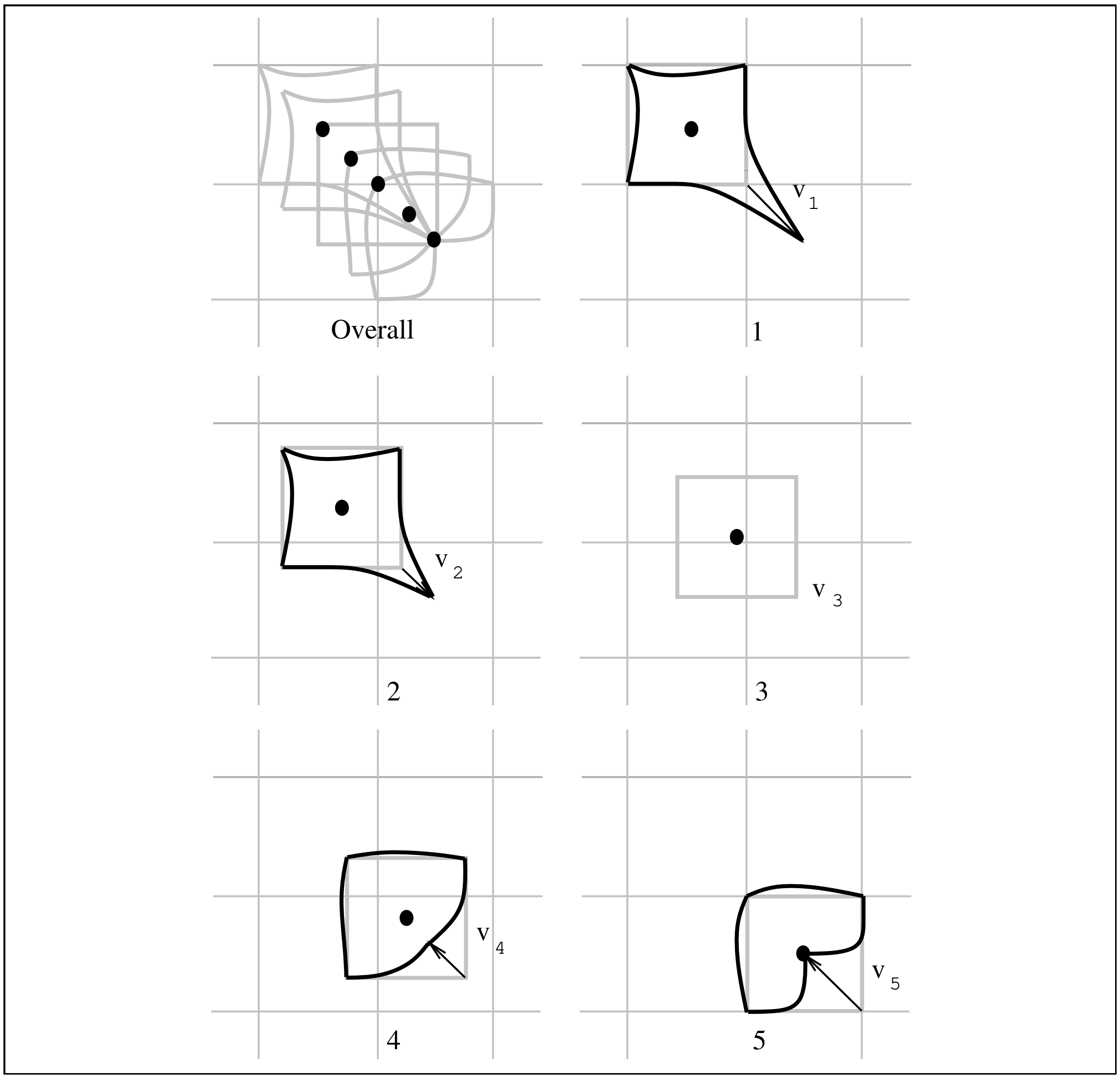


Figure 5.15: Interpolation points and warping vectors while moving under the influence of "gravity"

The animation for the snap effect is similar to that used for the translation operation during idle time. The graphical object is moved by a timer-activated callback. The amount the object moves between each frame is calculated as a proportion of the total snap distance and the number of frames in the animation. In Figure 5.15, the diagram labelled "overall" shows five frames of this animation superimposed on each other. The individual frames of the animation for the snap effect are shown in the diagrams marked '1' through '5'.

The warp vector v is calculated as the difference between the current position of the cursor and the most recent position of the original grasped vertex during the animation. Warp vectors of zero magnitude are placed on the other vertices at their most recent positions. Notice that the tip of the warp vector where the user has grasped the object does not move. Diagram '1', in Figure 5.15, shows the graphical object pinned to the original snap position; vector v_1 is calculated as the difference between the original position of the grasped point and the current cursor position. Diagrams '2' through '5' show corresponding warping vectors v_2, v_3, v_4, and v_5 (warp vector v_3 has a magnitude of zero.) Typically, the cursor moves during the animation; in this example it remains still for simplicity.

5.3.3 Indirect Manipulation

The animation effects for indirect manipulations involving object geometry rely on two techniques: autonomous animations and warping. Autonomous animations generate intermediate frames that convey the illusion that an indirect manipulation operation is being performed as a direct manipulation operation. Warping distorts the appearance of the objects to further suggest the effect of direct manipulation.

The indirect manipulation effects are based on an *arm*, *disarm*, and *activate* push button model. Arming of a button occurs whenever the user presses the push button. It is at this point that the preview animation is started. If the user moves the mouse off the push button and then releases it, the push button is disarmed and the undo animation is started. This disarm feature allows an immediate undo of the current command. If the cursor is over the button when the user releases it, the button is activated and a confirmation animation is initiated. As with the implementation of animations for pinning and snap and drag, the discussion centres around the calculations of the warp vectors during the animations. The animation effects described in this section are considered in the context of three examples of

Indirect manipulation	Transformation	Incremental value
horizontal alignment	translation x axis	inc_x
vertical alignment	translation y axis	inc_y
rotation about centre	rotation	inc_a
horizontal flipping	scaling x axis	inc_{sx}
vertical flipping	scaling y axis	inc_{sy}

Table 5.1: Incremental values for translate, rotate, and scale

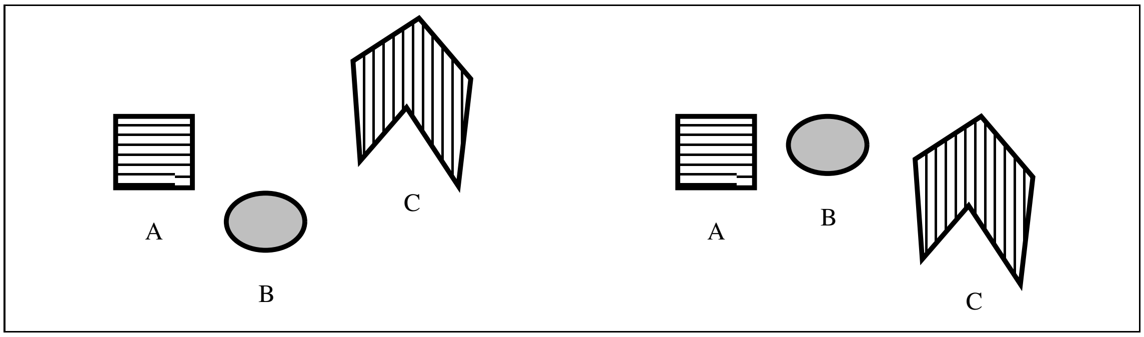


Figure 5.16: Example of align object tops

indirect manipulations involving object geometry: aligning, rotating, and flipping.

The overall algorithm for each of the three operations calculates a set of incremental values for translation, rotation, and scaling. These are shown in Table 5.1. The incremental values are applied to the manipulated objects at each frame of the animations. The indirect manipulation aligning operations require the setting of incremental values inc_x and inc_y, based on the amount the graphical object moves in each of these directions during each frame. Similarly, the rotation operation requires the setting of the inc_a incremental value, and the flipping operations require the setting of the incremental values inc_{sx} or inc_{sy}. Applying a -1.0 scale factor in the x direction flips a graphical object horizontally, and a -1.0 scale factor in the y direction flips a graphical object in the vertical direction.

The calculation of the warp vectors and the animations are based on these five incremental values. Each indirect manipulation uses a different algorithm for calculating the incremental values.

The arming animation is slightly different for each operation. For alignment, the arming animation entails the translation of a group of objects so that they align to a position relative to a particular object. For example, the diagram on the left of Figure 5.16 shows a set of unaligned graphical objects. The diagram on the right shows the alignment of all the objects' tops to the top of object **A**. The graphical object to which all other objects are aligned to is known as the *first* object, and remaining objects are referred to as *aligning* objects. The basic alignment animation algorithm is as follows:

1. Find the reference point of the *first* object.
2. Set the first object's incremental values to zero.
3. For each of the remaining aligning graphical objects.
   (a) Set the rotate and scale incremental values to zero.
   (b) Set the translation incremental values for the particular alignment.
   (c) Determine which vertex is to be the simulated grasped point.
4. Call the timer function.

For each form of alignment, Table 5.2 shows the definition of the reference point of the first graphical object in the column *first*, the reference point for the aligning graphical objects in the column *aligning*, and the incremental values for the effect in the column *incremental values*. The *midpoint* is the centre of an object. For each of the *aligning* objects, the grasped vertex is the closest vertex when the object is pulled toward the *first* object. When an *aligning* object is pulled away from the *first* object, the grasped vertex is the furthest vertex from the *first* object.

The basic form of the arming rotate animation is similar to the alignment animations, but the rotate animation does not use the *first* and *aligning* objects. Instead, the animations are applied to the entire group of objects, and the concept of a single grasped vertex is not used. In the animation, all the vertices appear to be grasped simultaneously. The rotation incremental values are defined in Table 5.2 and are set as follows:

Operation	**First**	**Aligning**	**Incremental values**
vertical	y = y of midpoint	y1 of midpoint	$inc_y = \frac{(y-y1)}{steps}$
center	x = x of midpoint y = y of midpoint	x1 of midpoint y1 of midpoint	$inc_x = \frac{(x-x1)}{steps}$ $inc_y = \frac{(y-y1)}{steps}$
left	x = x of left	x1 of left	$inc_x = \frac{(x-x1)}{steps}$
right	x = x of right	x1 of right	$inc_x = \frac{(x-x1)}{steps}$
bottom	y = y of bottom	y1 of bottom	$inc_y = \frac{(y-y1)}{steps}$
top	y = y of top	y1 of top	$inc_y = \frac{(y-y1)}{steps}$
abut left	x = x of left	x1 of right	$inc_x = \frac{(x-x1)}{steps}$
abut right	x = x of right	x1 of left	$inc_x = \frac{(x-x1)}{steps}$
abut up	y = y of top	y1 of bottom	$inc_y = \frac{(y-y1)}{steps}$
abut down	y = y of bottom	y1 of top	$inc_y = \frac{(y-y1)}{steps}$
clockwise 90			$inc_a = \frac{-90.0}{steps}$
anticlockwise 90			$inc_a = \frac{90.0}{steps}$
flip vertical			$inc_{sy} = -2.0$
flip horizontal			$inc_{sx} = -2.0$

Table 5.2: Incremental values for the animations

1. For each graphical object in the group.

   (a) Set the scale and translate incremental values to zero.

   (b) Set the rotation to the incremental value for the particular rotation.

2. Call the timer function.

The same algorithm used for the arming rotate animation is also used for the arming flip animation, with the exception that the flip sets a scale incremental value instead of the rotate incremental value, as shown in Table 5.2.

Each of the algorithms for the arming animation uses a timer function **tick** to perform the update operations. The **tick** function uses a separate set of incremental values inc_{sx}, inc_{sy} inc_a, inc_x, and inc_y to calculate the warp vectors for each of the graphical objects in the group. The number of steps and duration of the animation is predefined. This algorithm uses a linear calculation for the change of the incremental values, but as with the pinning effect, the Bézier timing function in Section 4.4.3 may be used to produce a slow-in/slow-out effect. The algorithm for the **tick** function to update each frame of the arming animation is as follows: For each graphical object i in the group.

1. Apply incremental scale $inc_{sx}(i)$ and $inc_{sy}(i)$.
2. Apply incremental rotate $inc_a(i)$.
3. Apply incremental translate $inc_x(i)$ and $inc_y(i)$.
4. If the vertex is the telltale vertex,
   (a) Then add vector v_i at the vertex.
   (b) Else if the vertex is not the grasped one then add vector $v'(i)$ at the vertex
   (c) Else at the grasped vertex add a zero magnitude warp vector.
5. Set the warp transform and redraw object.
6. If no more steps
   (a) Then redraw each object without warping.
   (b) Else reset the timer.

Figure 5.17 shows an alignment operation to align the top of the square with the bottom of the circle. Frame 1 shows the positions of the graphical objects before the alignment. Once the animation starts in Frame 2, the telltale vertex vector v_i is shown as the arrow headed line, and the dots indicate zero magnitude warp vectors. The telltale vertex vector is calculated as the difference between the position of the vertex i in its original position and its current position. The other vertex vectors, shown as dashed arrow headed lines, are calculated as the difference between the position of the vertex in its past position and its current position. Frames 3, 4 and 5 continue the animation until the two graphical objects are aligned. Note that currently only the alignment animation makes use of telltales.

The range coefficients of the Warp object are used to help strengthen the visual effect of the telltales. For example, Figure 5.18 shows the effect of changing

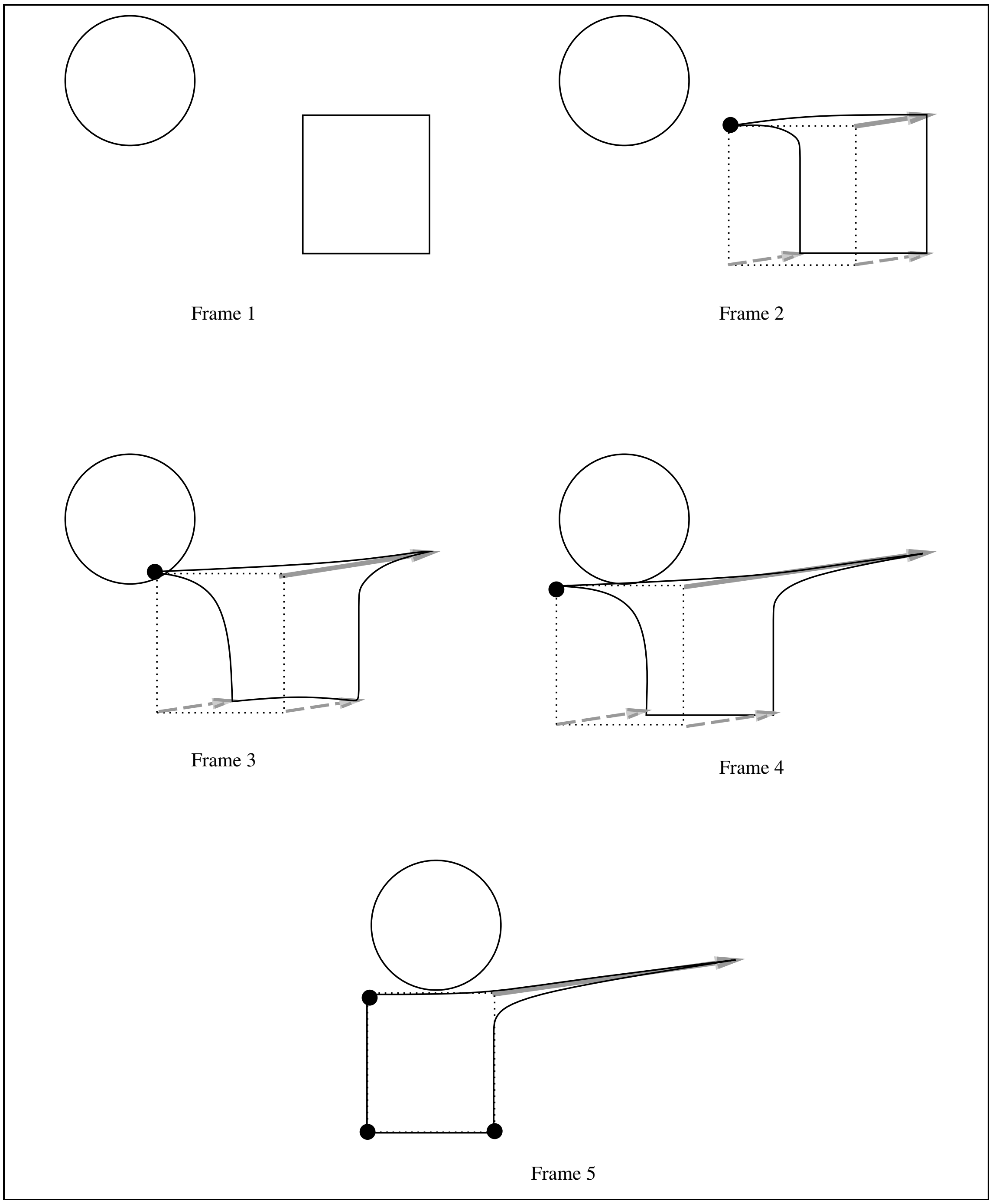


Figure 5.17: Example of using telltale warping vectors

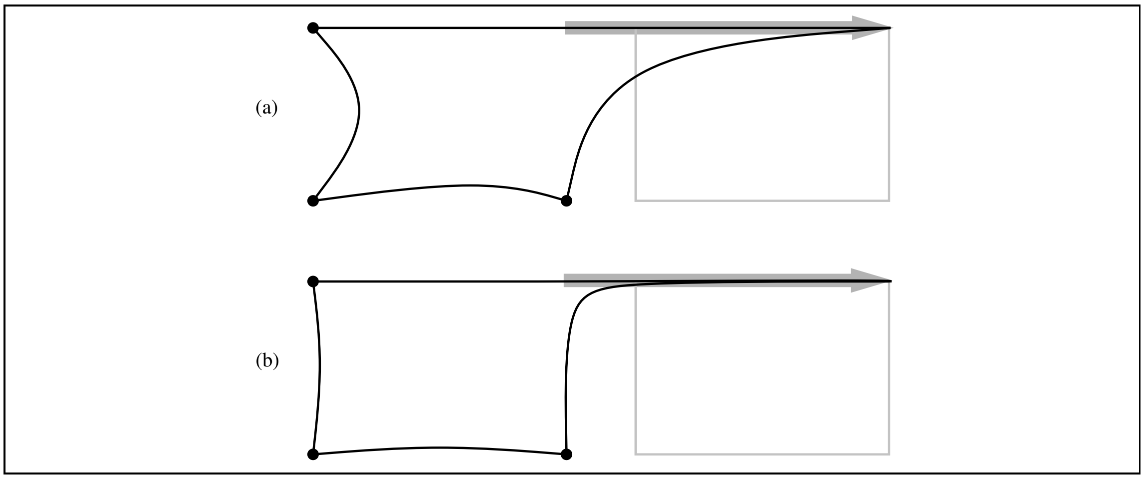


Figure 5.18: The effect of vector range coefficients

range coefficients while keeping the vectors constant. In both the top and bottom diagrams, the figure illustrates the set of vectors that portray an object being considered for alignment. Three of the object's corners are pinned to the positions they will take if the alignment operation is committed. The dots in the figure represent vectors of zero magnitude, and arrow headed line is the telltale warp vector. The top-right corner has a telltale to show its original position. This pre-alignment location shows the position that the object would have if the operation was cancelled.

Diagram (a) of Figure 5.18 shows the curvature of the warping when all vectors have a range coefficient of 1.0; diagram (b) shows the effect with the pinning vectors having a range coefficient of 1.0 and the telltale formed from a vector with a range coefficient less than one. The overall effect in part (b) gives a better preview of potential results of the alignment operation because the bulk of the object is under the influence of the long-range pinning vectors and hence close to its final position. To produce a telltale that looks convincing, the range coefficient is set to 0.25 for the telltale vertex vector, and all other warp vectors have a range coefficient set to the default 1.0.

The animation for *disarming* is the reverse of the *arming* animation. It is a matter of reversing the five incremental values. The alignment, rotation, and flipping disarming animations use the same algorithm:

1. For each graphical object in the group
   (a) Set the rotate incremental value to $-inc_a$.
   (b) Set the translate x incremental value to $-inc_x$.
   (c) Set the translate y incremental value to $-inc_y$.
   (d) Set the scale x incremental value to $-inc_{sx}$.
   (e) Set the scale y incremental value to $-inc_{sy}$.
2. Call the timer function.

The **unwarp_tick** timer function used in the above algorithm is similar to the **tick** timer function.

Finally, the *activate* animation is invoked when an indirect manipulation operation is committed. Currently, the alignment operation is the only animation that uses an *activate* animation; the animation simply retracts the telltale vertex. The algorithm of this animation is as follows:
For each graphical object in the group.

1. For each vertex in the graphical object.
   (a) If the vertex is the telltale
      i. Then add vector v at the vertex.
      ii. Else add a zero magnitude vector at the vertex.
2. Apply this Warp object and redraw the graphical object.
3. If no more steps
   (a) Then redraw each object without warping.
   (b) Else reset the timer

The telltale vertex vector v_i is calculated as the difference between the original and final position of the vertex scaled by $\frac{s_t - s_c}{s_t}$, where s_t is the total number of steps and s_c is the current step number. The range coefficient for vector v_i is set to 0.25, and a range coefficient of 1.0 is set for the zero magnitude vectors.

5.4 Summary

The animation effects for the widget components and graphical object manipulation described in Chapter 3 can be implemented with the tools described in Chapter 4. A new animated widget kit, compatible with the existing suite of InterViews look-and-feel widget sets, enables applications to use the new widgets without change. To incorporate the new widgets, an application need only be recompiled against the new libraries; it will then operate in its original manner but with a new animated look-and-feel for its widgets. A set of distortion and follow through effects effects help application programmers give their interfaces substance. The effects, which model physical phenomena such as delay, inertia, and friction are suitable for common direct manipulation operations such as translating, scaling, rotating, and shape editing. Keyframe animations and distortion can also provide anticipation cues by suggesting constraints such as pinning, snapping or the outcomes of indirect manipulation operations such as alignment and flipping.

Chapter 6

Demonstrating Animation in Graphical User Interfaces

To explore the ideas of using animation in the user interface, I used the animation effects described in Chapter 3 to provide animated visual feedback in a drawing editor. The editor demonstrates the effectiveness of the use of animation for a subset of user interface operations, and it shows that the techniques are practical even on standard workstation hardware.

The animated drawing editor was used two experiments that tested the benefits of using animation in the user interface. The first experiment was designed to test the effectiveness of the animations in providing feedback about the process of direct manipulation movement operations. The subjects in the experiment were asked to undertake a task with different forms of animated visual feedback. The measure of effectiveness was the time required to successfully complete the operation. The second experiment was designed to measure users' preferences for a gamut of animation effects. Subjects were encouraged to adjust the amount of animation to their liking while performing a range of editing tasks; the different settings they chose were then recorded.

6.1 Adraw: an Animated Drawing Editor

To demonstrate the use of cartoon-style interaction techniques in a significant application, I constructed a drawing editor, *adraw*, with an animated user interface. The editor incorporates the animation effects described in Chapter 3. In particular, it incorporates the animated widget kit and the animated visual feedback for both direct and indirect manipulations.

Appendix F describes some of the implementation issues involved in building the animated drawing editor. In particular, the issues of defining the directly manipulatable graphical objects to be animated and the manipulator objects to manœuvre these graphical objects are discussed.

Adraw is a drawing editor that can be used to create and edit drawings made up of graphical objects like text, images, splines, rectangles, and polygons. The design of the editor is based on the InterViews drawing editor *idraw*; Figure 6.1 shows the interface. The main portion of the window, the drawing area, is a portrait view of an A4 page. In a column along the drawing area's right side is a set of radio buttons for engaging the drawing tools, and above the drawing area is a set of pull-down menus containing commands. A panner on the bottom right side allows for the panning and zooming of the drawing area, while a set of indicators along the top display attribute information for different editing commands.

Before a drawing tool can be used, it must be engaged by selecting its radio-button or by typing the hot key. Once engaged, the tool can be used by clicking the left mouse button in the drawing area. The *Select*, *Move*, *Scale*, *Rotate*, *Pin*, and *Edit* tools are used to manipulate existing graphical objects, while the *Text*, *Polygon*, *Spline*, *Rectangle*, *Closed Polygon*, and *Closed Spline* drawing tools place the editor in a mode to create new graphical objects. Finally, the pull-down menus *File*, *Edit*, *Structure*, *Font*, *Brush*, *LineColor*, *FillColor*, *Align*,*View*, and *Warping* contain commands for editing a drawing or for controlling the editor's execution.

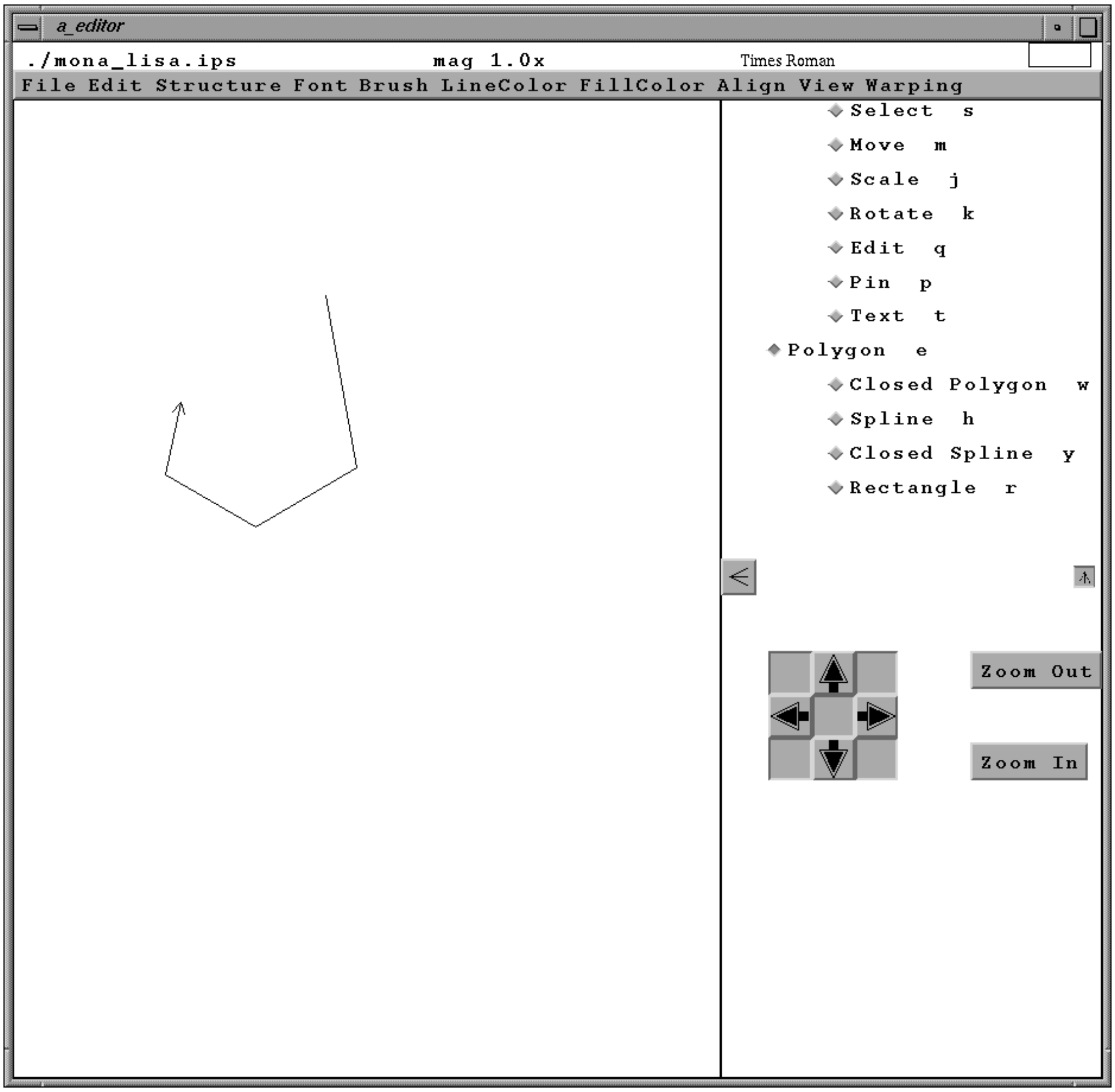


Figure 6.1: The adraw main window

This section describes the use of the editor in three sections: direct manipulation operations with animation effects, indirect manipulation operations with animation effects, and operations without animation effects.

6.1.1 Direct Manipulation Operations with Animation Effects

This section describes the direct manipulation operations that use animation effects to supplement the traditional visual feedback. Presented first are a set of direct manipulation operations used to modify the geometric properties of graph-

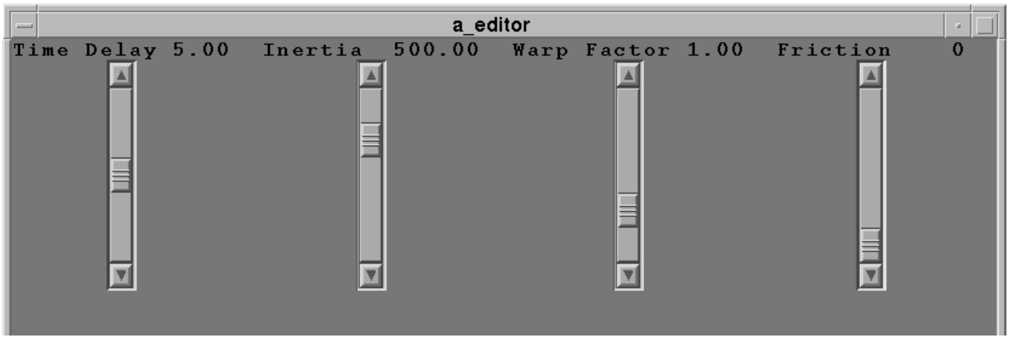


Figure 6.2: Popup window containing control tools for warping effects

ical objects. The direct manipulation operations that produce or use constraints are then discussed. Finally, the techniques that allow controlled degradation of drawing quality are examined.

The operations *move*, *scale*, *rotate*, *create rectangle*, and *edit vertex or control point* perform standard direct manipulation operations. To perform these operations, the user must first select the graphical objects with the left mouse button and then drag the mouse. The move operation translates the selected graphical objects from one position to another. The scale operation resizes selected graphical objects about their centres. The rotate operation changes the orientation of selected graphical objects about their centres. The create rectangles operation creates rectangles by dragging them open. The edit operation allows vertices and control points to be shifted to new locations. All of these operations were augmented to include the different forms of animated visual feedback described in Section 3.3. Figure 6.2 shows the dialog box for controlling and displaying warping parameters. This set of sliders controls the amount and look of the warping effect.

In addition, cartooning effects were included to help visualise different constraints (such as pinning and gravity) in the editor. The *pin* operation allows the user to specify that graphical objects are to be attached to the drawing surface, making their position, orientation, and size unchangeable. This operation in itself has no animation effect, but it causes move, rotate, and scale operations to use

Figure 6.3: Popup window containing control tools for controlled degradation of rendering quality

the pin animation effect described in Section 3.4.1. The *gravity on/off* menu item toggles the gravity state of the drawing surface to on or off. Gravity constrains the tool operations to the grid, whether or not the grid is visible. The gravity animation effect is presented in Section 3.4.2.

To investigate possible approaches for improving the responsiveness of direct manipulation operations, several approaches were implemented to reduce the drawing time of graphics rendering by adraw. One approach was to use a dialog box to control the settings of different effects that degrade the quality of the drawing when time is short. The other is a simple heuristic that engages different levels of roughness of curves. Figure 6.3 shows the dialog box responsible for the control if the degradation parameters. The set of sliders adjust the degree of the following degradation effects: the level at which enclosing boxes replace groups, the minimum area of drawn objects (smaller objects are not drawn), and the roughness of curves. The palette buttons *Thin Lines*, *No Images*, *Untransform Char*, *Small Objects Removed*, and *Enclosing Boxes* toggle the state of engagement for each of these forms of rendering degradation.

During a drag operation, the roughness of the curves is automatically adjusted

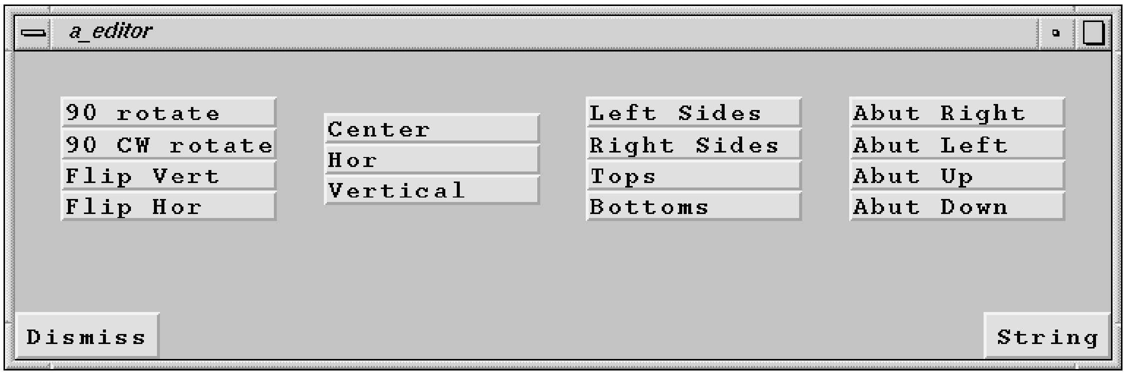


Figure 6.4: Popup window containing alignment tools

to improve the response time. The adjustment affects both the warping effect and the spline curves. The roughness factor is changed when adraw detects a lengthing of intervals between events during a drag operation.

6.1.2 Indirect Manipulation Operations with Animation Effects

The same cartoon techniques used with the direct manipulation operations were also applied to the indirect manipulation operations in adraw. The approach taken for indirect manipulation operations was to animate the change they caused as if the task was carried out by direct manipulation. Section 5.3.3 describes the forms of animation for indirect manipulation supported in the drawing editor. The indirect manipulation operations are initiated through a popup window containing a set of arm/disarm/activate buttons, as shown in Figure 6.4. Table 6.1 summarises the action for each of the buttons in the popup window.

The indirect manipulation operations that involve changes to the geometry of objects are rotate by 90 degrees; flip vertical and horizontal; and abut top, bottom, left, and right. In each case, the change is displayed in the same way that the user might see if undertaking the operation using direct manipulation. If several objects are involved they all changes together, as if simultaneously manipulated by several users. Thus, rotations are shown as if objects were dragged around by their corners,

Command	**Effect of the Command**
90 rotate	Rotates the selected objects 90 degrees clockwise.
90 CW rotate	Rotates the selected objects 90 degrees anti-clockwise.
Flip Vert	Flips the selected objects about its horizontal centre
Flip Hor	Flips the selected objects about its vertical centre
Center	Aligns the selected objects' centres
Hor	Aligns the selected objects' horizontal centres
Vertical	Aligns the selected objects' vertical centres
Left Sides	Aligns the selected objects' left sides
Right Sides	Aligns the selected objects' right sides
Tops	Aligns the selected objects' tops
Bottoms	Aligns the selected objects' bottoms
Abut Right	Abuts the selected objects towards the right
Abut Left	Abuts the selected objects towards the left
Abut Up	Abuts the selected objects towards to top of the screen
Abut Down	Abuts the selected objects towards the bottom of the screen

Table 6.1: Indirect manipulation commands

Red	**Green**	**Blue**
1.0	0.0	0.0
0.8	0.0	0.2
0.6	0.0	0.4
0.4	0.0	0.6
0.2	0.0	0.8
0.0	0.0	1.0

Table 6.2: Example of a linear transition of RGB values

movements as if objects were dragged by the vertex nearest to their destination point, and flips as if objects were turned inside-out.

Finally, adraw uses a set of animations to support the setting or changing of *line* and *fill* colours. A simple scheme that changes colour using linear interpolations in the Red-Green-Blue (RGB) colour coordinate space proved to be adequate. The animation smoothly varies each colour component from its initial to its final value. Table 6.2 shows an example of a translation between a red RGB value and a blue RGB value. A value of 1.0 is a setting of 100 percent for that colour.

Key	Function
Control D	delete until end of line
Control B	move cursor left one character
Control F	move cursor right one character
Control A	move cursor to beginning of text
Control E	move cursor to end of text

Table 6.3: Text entry edit functions

6.1.3 Operations Without Animation Effects

Adraw was rounded out with a set of unanimated operations to provide the functionality of a complete drawing editor. This section provides a brief description of each of the adraw operations that have no animation effects at this time.

The drawing tools *polygon*, *spline*, *closed spline*, and *closed polygon* create the defined graphical object. They all have the same mouse commands. Each left-click starts a new segment (i.e., adds a vertex or control point); each right-click removes the last vertex or control point added. The middle button finalises the graphical object.

The *select* operation provides a means for the user to select graphical objects for other operations. There are additional controls to toggle the adraw's grid on or off and to change the grid spacing. The *import_graphic* command can import images from files in the TIFF and X bitmap formats. The *text* mode allows the creation of text labels; the user uses a left-click to position the first character of text to be typed in. Table 6.3 defines the keystrokes for editing the text.

There are a set of operations for interacting with files associated with a particular drawing. These include operations to begin a new drawing or to restart the current drawing from the last saved version of a drawing. An existing drawing may also be loaded into the editor. The current drawing can be saved into a file with a specified name or into the file it was read from. Alternatively, a PostScript version of the drawing can be sent to a printer or file.

Adraw supports a standard set of graphical object manipulation operations:

cut, *paste*, *copy*, *duplicate*, *delete*, *select all*, and *undo*. The undo command is only applied to the last editing operation; a successive undo command reverts the drawing to the state before the undo command. Operations to move, scale, or rotate graphical objects by exact amounts are supported through a dialog box interface.

Structuring commands are included to modify the order in which graphical objects are drawn and the groupings of graphical objects. There are operations to bring the selected graphical objects to the front or back of the drawing so that they are drawn on top of or behind the other graphical objects. There is also an operation to *group* sets of graphical objects that allows the building of hierarchies of graphical objects, and there is an *ungroup* operation, which removes the groupings of any selected grouped graphical objects.

6.2 Experiment 1: Feedback Animation for Snap and Drag

The first experiment tested the idea that animation can provide information about the process of an interaction. This is particularly evident when an operation's effect on an object is subject to the presence of constraints. Section 3.4.2 describes how animation can provide feedback that suggests both the immediate affect of the user's action and the final outcome of the operation when the constraint is taken into account.

In adraw, the snap and drag constraint is provided for objects moving under the influence of a "gravity" field. This constrains the object's position to lie on a regular grid. To test the effectiveness of animations that suggest constraints, the editor was modified to use one of four feedback techniques for movement operations, as illustrated in Figure 6.5. These four feedback techniques are as follows:

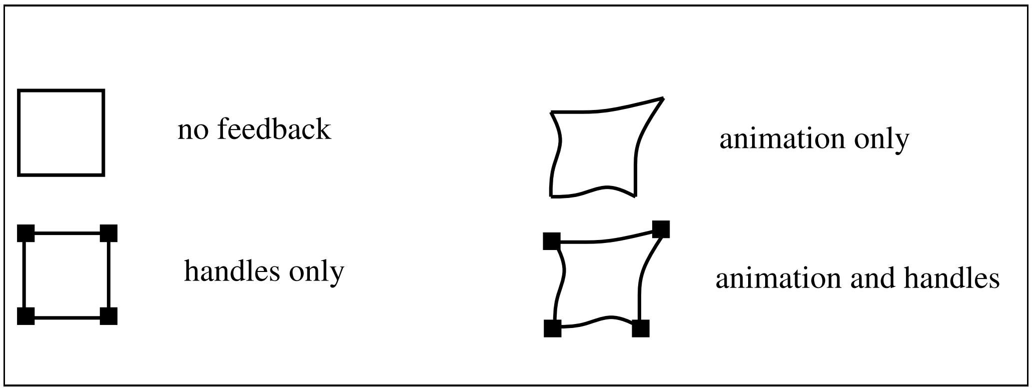
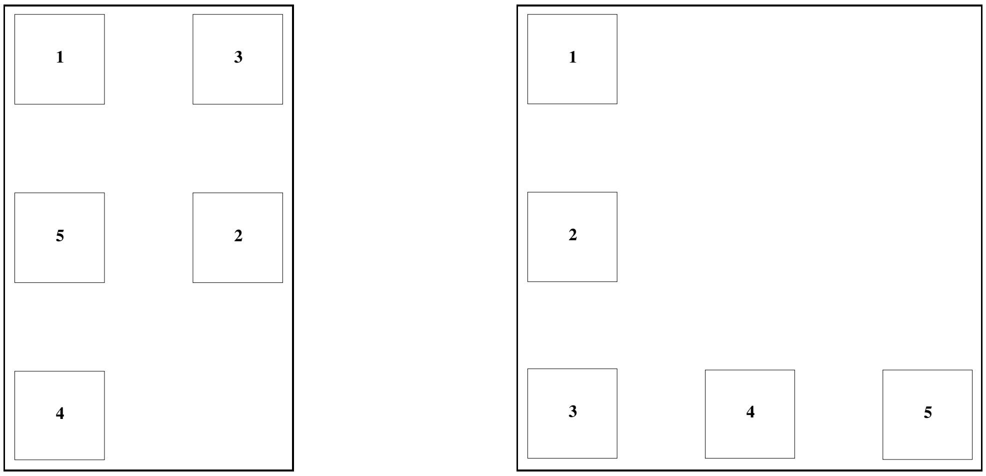


Figure 6.5: The four visual feedback effects for movement

1. **no feedback:** Objects being moved show no visible feedback at all; they merely follow the mouse if unconstrained, and snap suddenly to their allowable positions if under gravity.
2. **handles only:** Objects being moved show "handles" (small squares at the corners of the object) to indicate that the object has been selected, but the position of the object is as above.
3. **animation only:** Objects are animated so that the corner being dragged follows the mouse while the remainder of the object lags behind; if an object is constrained by gravity, it stretches to follow the mouse, then snaps to its new position.
4. **animation and handles:** Objects being moved show both handles and animation effects.

The experiment asked users to perform a simple rearrangement task under the influence of a gravity field. The experiment measured the users' performances by recording how long they took to complete the task and by noting instances of "misses" and "drops". A miss was defined as an attempt to move an object that failed to pick the intended target. A drop was defined as an incorrect release of an unmoved object, presumably because the user mistook the object's lack of observable response for a failure to pick it. The supervisor also asked users to complete an exit survey on their attitude to the various forms of feedback.



(a) start (b) finish

Figure 6.6: Example of the starting and finishing configurations for the rearrangement trials.

The experiment tested the following hypotheses:

1. Cartoon-style feedback animation decreases task completion time and reduces miss and drop rates when compared with feedback that does not use such animation.
2. Users prefer cartoon-style feedback animation over non-animated feedback.

6.2.1 Experimental Design

The experimental task was to rearrange the objects from the configuration shown in Figure 6.6 on the left to that shown on the right.

Subjects were tested individually, working on a Silicon Graphics Indy workstation in a quiet isolated office. The experiment supervisor read from a script to explain the requirements of the experiment and gave the following instructions:

1. Given the graphical objects in the starting formation, move the objects to the finishing formation, as indicated on a paper diagram.
2. Perform the task as accurately and quickly as possible.

3. The object labelled “1” will remain in the same location.

4. Objects can be “picked up” only by their outlines.

5. The grid is turned on; therefore the objects can only be placed at the grid points.

The subjects were asked to read an information sheet and a set of instructions, then asked if they had any questions. Each subject was then asked to sign a consent form.

To familiarise themselves with the editor and the feedback effects, each subject performed four trials of the editing task, one for each feedback effect. Then each subject performed five sets of trials, each consisting of ten repetitions of the task for each of the four feedback effects. Thus, each subject performed 50 repetitions of the task for each effect, or a total of 200 repetitions. To minimise ordering artifacts, the feedback effects were presented in a random order within each trial set. Subjects were given a one minute rest break between each trial set. The experiment took about one hour for each subject to complete.

The editor was instrumented to record, at each press and release of the mouse button, the coordinates of the mouse and the time in microseconds. In addition, the editor recorded at each mouse press the object that was picked (if any) by a mouse press, and at each mouse release the new grid point to which an object had been moved (if any). This information allowed for analysis of both inter-subject and intra-subject effects.

The exit survey asked subjects to indicate their attitude towards the various feedback effects on a 7 point scale, with 1 being “strongly liked” and 7 being “strongly disliked”. See Appendix D for a copy of the exit survey.

The experiment was completed by 17 subjects (mean age 27 years, age range 19 - 42 years). Subjects were self-selected (on a first-come basis) from a pool of computer-literate students drawn from the undergraduate and postgraduate programs of the School of Computer and Information Science at the University of South Australia.

6.2.2 Results and Discussion

Initial analysis of the experimental data focused on task completion times. Analysis was carried out to compare the completion times for the four feedback effects and to compare the times between the five trial sets.

An ANOVA analysis (Tukey HSD test, SYSTAT for Windows version 5) showed no significant difference in execution time between the four feedback conditions. However, a significant difference was found between the time for the first trial and the second trial ($p = .005$), and between the first trial and trials 3, 4 and 5 ($p > .001$). This difference was most likely due to a learning effect: that is, the subjects took 1 set of 40 repetitions to stabilise their performance. The subjects showed no significant difference in execution time between trials 2, 3, 4, and 5.

For the attitude survey, the subjects showed a significant liking ($p > .001$) for the three feedback conditions ($mean = 2.97$) compared to the no feedback condition ($mean = 5.5$). The subjects showed a greater liking for the combined feedback of handles plus warp ($mean = 2.4$) than for either the handles alone ($mean = 3.1$) or warp alone ($mean = 3.4$), although the difference was not significant.

The lack of difference between the completion times for the four feedback effects is surprising, but may reflect the simplicity of the task. Perhaps subjects quickly learned to ignore the feedback (or lack of it) through performing the same task many times.

6.3 Experiment 2: Level of Animation Preference

To gauge users' liking for the animations a slider that allowed users to smoothly vary the amount of animation was added to adraw. At its minimum animation setting, the editor behaved like a conventional (unanimated) editor; at its maximum animation setting, manipulated objects were extensively distorted. Subjectively, the effect was as if the "weight" and "stiffness" of the objects varied: at low animation settings objects seemed light and rigid; at high settings they seemed heavy and rubbery.

The experiment encouraged users to adjust the amount of animation in the editor while performing a range of editing tasks. The changes users made to the animating setting were recorded as the trial proceeded. To determine user-interface satisfaction, the supervisor also asked users to complete a questionnaire about their attitude towards the editor and the animation.

The experiment tested the following hypotheses:

1. When given a choice, users choose an animation setting that corresponds to a clearly evident level of animation.
2. Users prefer animated interfaces over non-animated interfaces.

6.3.1 Experimental Design

The experimental task comprised the construction of three drawings of increasing complexity. Figures 6.7, 6.8, and 6.9 show the completed drawings for tasks one, two, and three. The remainder of this section describes how the experiment proceeded.

As in experiment one, subjects were tested individually, working on a Silicon Graphics Indy workstation in a quiet office. The experiment supervisor read from

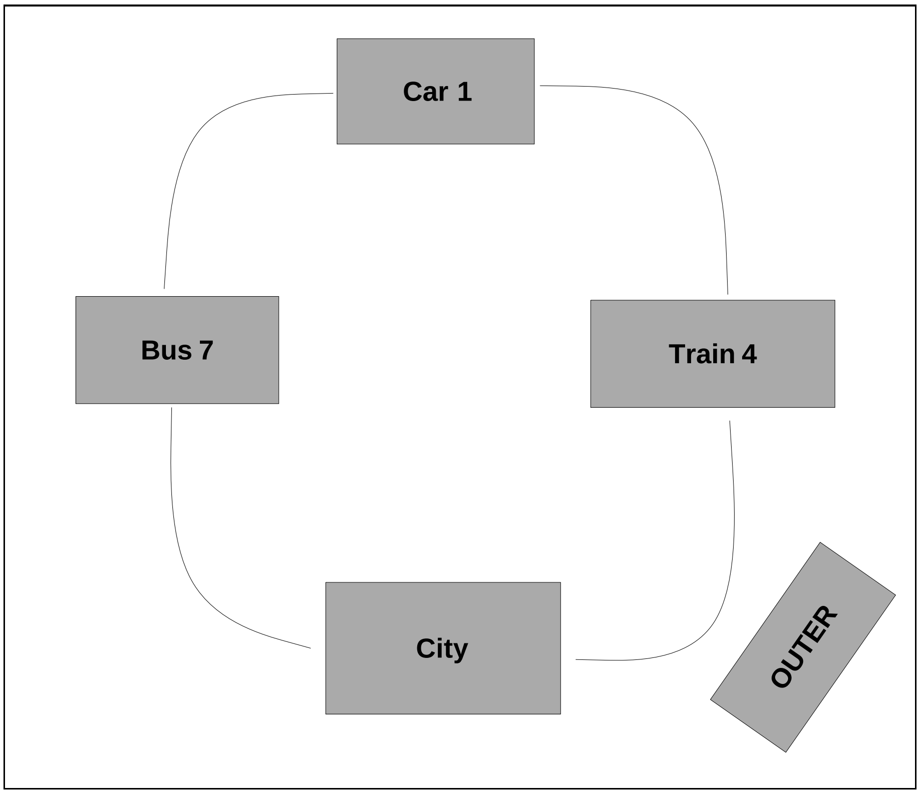


Figure 6.7: The first drawing task

a script to explain the requirements of the experiment. Subjects were asked to read an information sheet, then asked if they had any questions. The subjects were then asked to sign a consent form.

Subjects were told that they were to construct the three target drawings, which were presented to the users as paper handouts. They were also informed that they could adjust the amount of animation whenever they desired. At the start of each drawing task, the editor was set to zero animation, and the subjects were asked to construct and manipulate a simple rectangle to experiment with the effect of different animation settings. When they were happy with the effect, they were to delete the rectangle and proceed to construct the drawing. Subjects were given a two minute rest break between drawings. The experiment took about one hour for each subject to complete.

The editor was instrumented to record changes in the amount of animation, and the time at which the change occurred. This information mapped how subjects

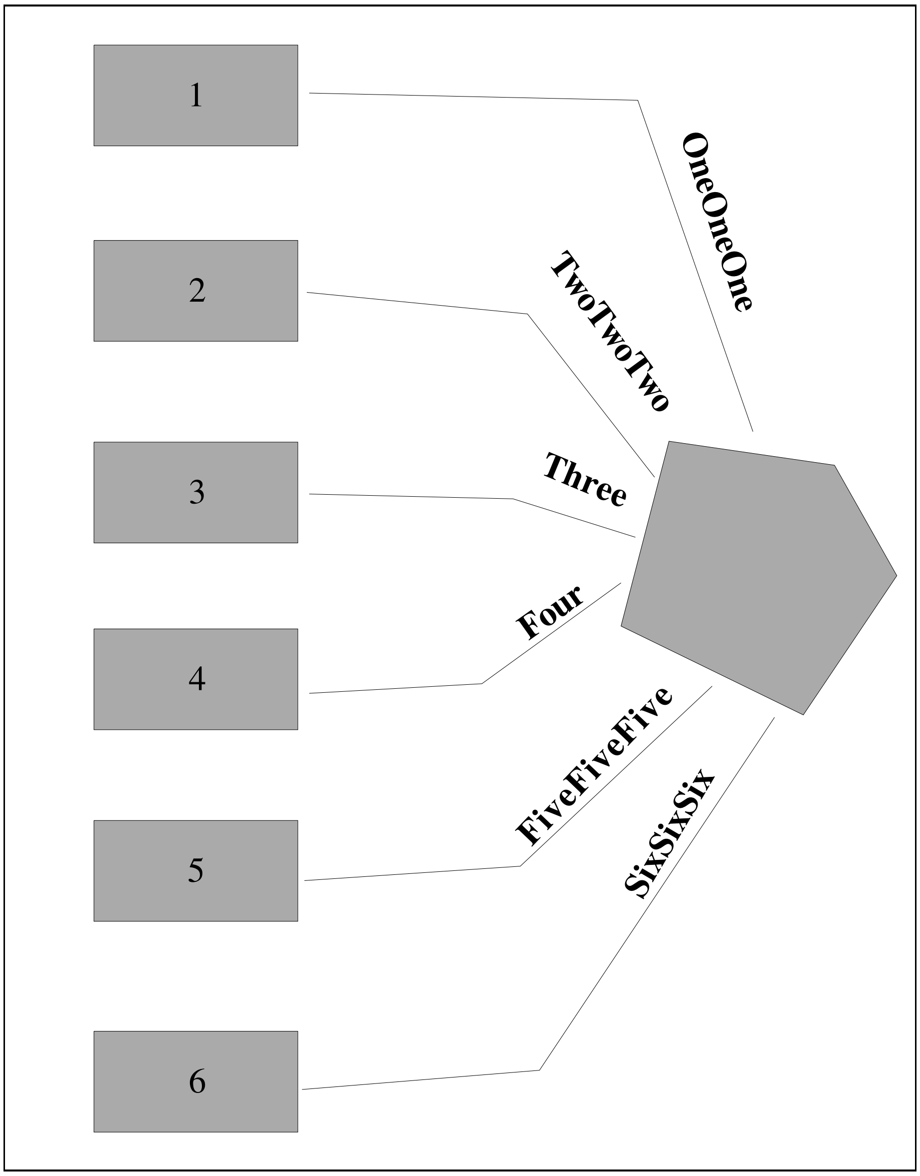


Figure 6.8: The second drawing task

adjusted the amount of animation over time, both at the beginning of a drawing task and during the execution of the task.

The questionnaire for user-interface satisfaction asked the subjects about their impressions of the animation effects and the overall drawing editor. A copy of the questionnaire is presented in Appendix E. Subjects recorded their responses on a seven point scale, with 1 corresponding to negative reactions ("terrible", "frustrating", "dull", "difficult") and seven corresponding to positive reactions ("wonderful", "satisfying", "stimulating", "easy"). Another set of questions asked subjects

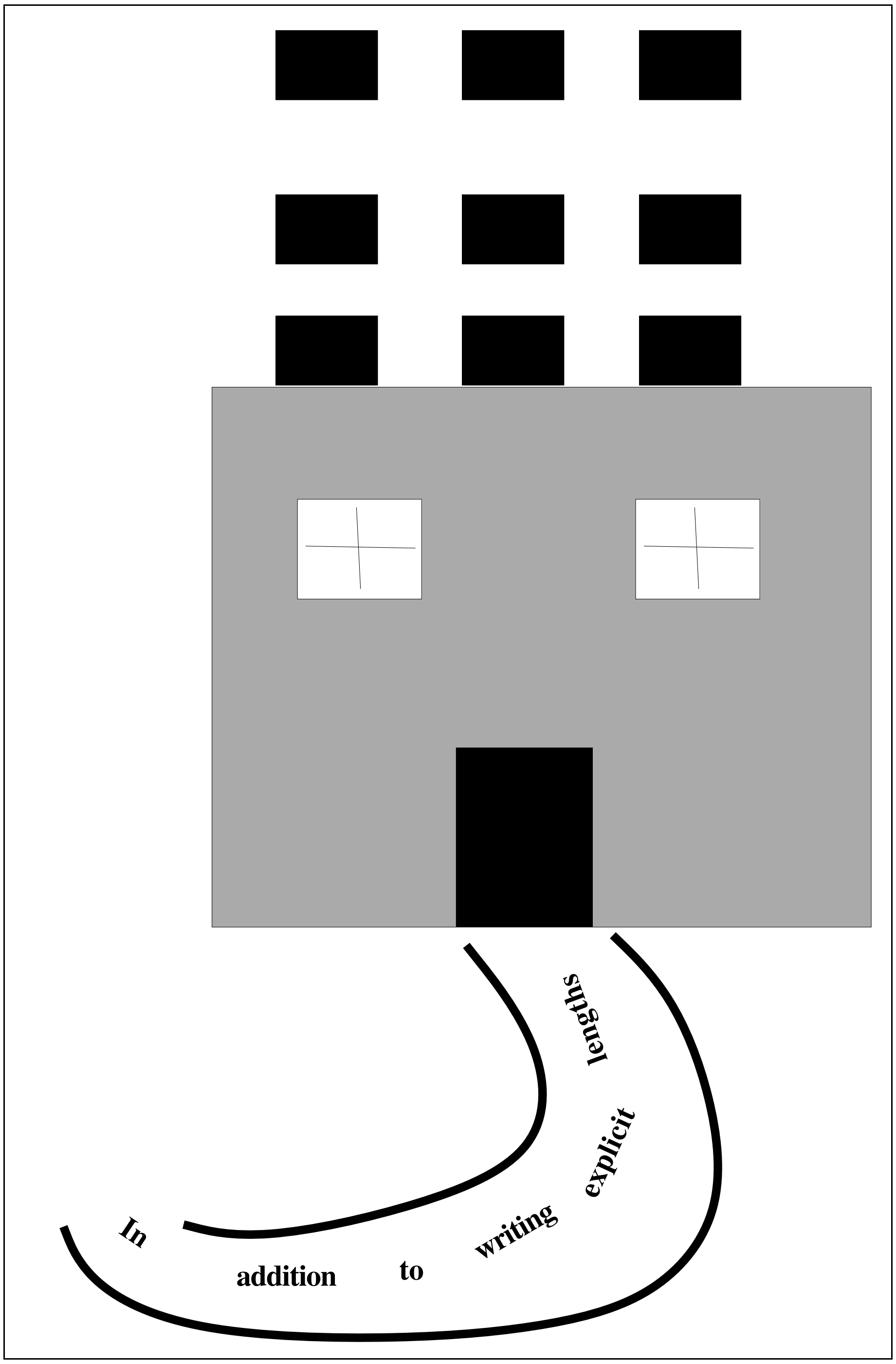


Figure 6.9: The third drawing task

to agree or disagree with statements about the animation.

Data was collected during the experiment from 13 different subjects (mean age 27 years, age range 19 - 42 years). On average, subjects had 12 years computing experience including an average of seven years experience with mouse-based interfaces and a current average of 24 hours per week using such interfaces. Subjects were self-selected (on a first-come basis) from a pool of computer-literate students drawn from the undergraduate and postgraduate programs of the School of Computer and Information Science at the University of South Australia.

6.3.2 Results and Discussion

Initial analysis of the experimental results showed that all subjects chose to experiment with the animation level at the beginning of the trial, most exercising the editor over the full range of animation settings. The usual pattern was for a subject to adjust the setting, then spend some time manipulating objects to get a feel for the animation at that level. Most subjects tried several different settings before adopting a final setting and proceeding to the editing task. Some made further adjustments during the task or at the end; others left the animation setting alone after the initial experimentation. Subjectively, the effect of the animation at different levels was such that a setting of 1.0 or more produced animation effects that were clearly evident during manipulation operations. For example, the diagram in Figure 6.10 shows a move operation with an animation setting of 1.0.

Table 6.4 summarises the subjects, preferences. For each drawing task, the table records the highest animation value the subject experimented with (max), the final value they settled on before beginning the task (final), and the amount of time (in seconds) that they spent adjusting the animation at the beginning of the task (time).

The data shows that, although each drawing task required the subject to reset

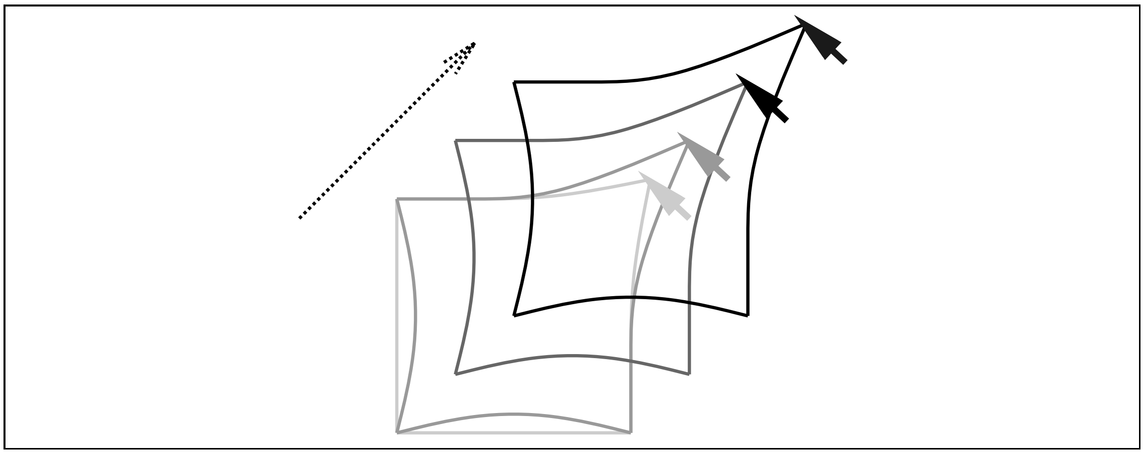


Figure 6.10: Animation of a move operation with an animation setting of 1

Subject	Drawing 1			Drawing 2			Drawing 3		
	max	*final*	*time*	*max*	*final*	*time*	*max*	*final*	*time*
30	4.5	2.1	20	14.9	2.7	14	3.2	3.2	1
31	19.0	4.3	73	4.0	4.0	67	5.0	5.0	69
32	10.7	4.8	61	19.0	4.9	42	4.8	4.8	2
33	10.5	6.0	42	5.4	5.4	2	5.2	5.2	1
34	19.0	0.9	46	4.3	1.9	33	2.0	1.8	23
35	5.0	2.3	35	2.1	2.1	57	2.2	2.2	3
37	19.0	2.5	33	4.8	3.2	16	19.0	3.0	13
38	11.5	7.5	72	2.9	2.9	1	2.9	2.9	7
39	9.3	0.0	54	8.0	0.0	59	6.3	0.0	44
40	19.0	3.5	84	3.0	0.8	8	1.2	1.2	1
41	11.0	6.3	69	7.2	2.8	26	6.4	5.4	1
47	11.4	2.7	37	4.7	4.7	1	2.4	2.4	1
48	20.0	1.5	97	2.0	1.3	89	4.0	0.9	39

Table 6.4: Subject animation preferences for the drawing tasks

the animation level (each task began with the level set to its minimum value), most chose to set the animation for each task to a similar level. Moreover, in the second and third tasks, the subjects tended to spend less time experimenting with the animation. Indeed three subjects in task 2 and seven in task 3 merely set the animation to about the same level as before and proceeded straight to the drawing task, although others spent about the same amount of experimentation time for each task.

The data also shows that all but one subject (subject 39) chose to set the animation level to a value above zero. (More detailed analysis showed that subject

39 completed part of the drawing task with the animation set at around 1.0). The average final animation setting over all subjects and drawing tasks was 3.0 ($minimum = 0.0$, $maximum = 7.5$, $sd = 2.1$).

The attitude questionnaire showed that subjects had a positive response to both the animation ($mean = 5.2, sd = 1.0$) and the editor as a whole ($mean = 4.9, sd = 1.0$). The more positive response to the animation compared with the editor (significant at $p = .001$) suggests that the animation itself was a contributing factor to the subjects' rating of the editor. Of various suggested characterisations of the possible benefits of animation in drawing editors, subjects rated it as "fun to use" ($mean = 6.3, sd = 0.9$) and "intuitive to use" ($mean = 5.5, sd = 0.9$). They were ambivalent about whether it improved productivity ($mean = 4.3, sd = 2.0$) and slightly positive that it encouraged them to work ($mean = 4.9, sd = 1.0$). The subjects agreed that "the manipulation of objects seemed smoother" ($mean = 5.2, sd = 1.5$) and that "the manipulated objects seemed to have substance" ($mean = 5.3, sd = 1.1$).

Further investigation of data shows no strong correlation between the level of final animation setting and the responses to the questionnaire. The graph in Figure 6.11 shows the average of the subject's final animation values (the dashed line) and their responses to the questionnaire (the solid lines). In the main, subjects who set the animation to a high level responded about as positively as those who set it to a low level. Even the subject who set the animation level to a zero was generally positive about the animation effects.

6.3.3 Users Comments

The questionnaire also asked subjects to make comments on the system and the animation effects. Some comments related to what the subjects liked about the animation effects:

- "It gives the objects a 'feel'".

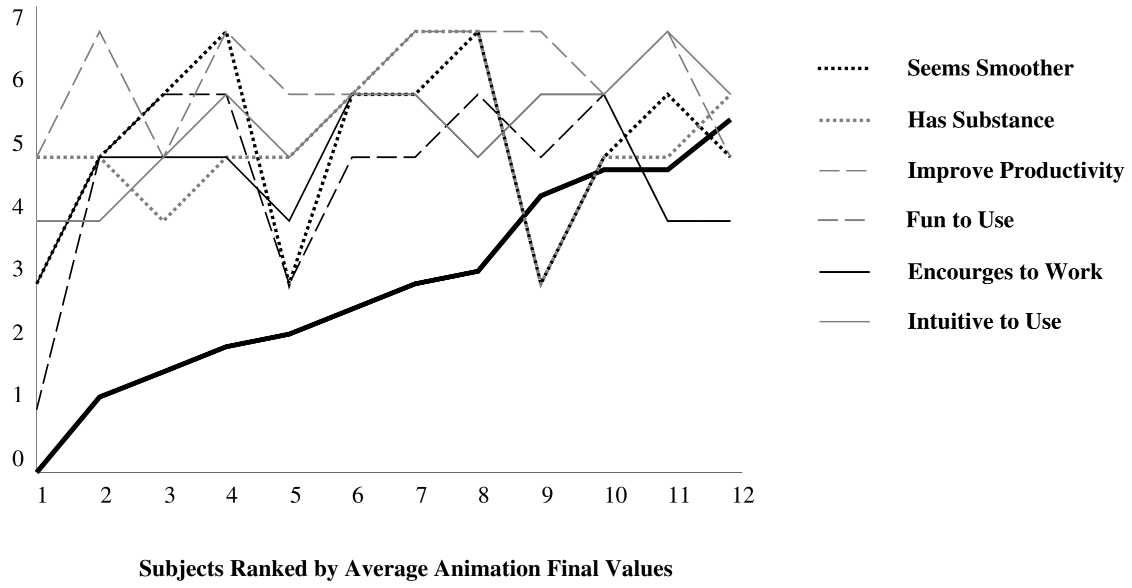


Figure 6.11: Graph of total *final* values and six questionnaire values

- "Objects seem 'softer'".
- "[Animation] makes the work you are doing look fun. [It] makes things more interesting."
- "It was easy to see what your action was doing when you move the object—it warps but still keeps a recognizable shape."
- "Good drawing tool. I enjoyed playing around with it."
- "[Animation] gives a sense of movement and speed."
- "It seems to be really good for the text rotations."
- "It can give the user more information ([as when] using gravity)."

Other comments related to how other applications could be improved with animation:

- "Objects could be animated when they are pushed against other objects."
- Animation could perhaps be used for "fall down menus or fold out menus."
- Animation could be used to show a "'peeling off' action when objects are duplicated." (currently, duplicates just appear)

- "Modifications to formatting of text [could be animated] to give a flowing effect. At the moment the text changes too rapidly in WYSIWYG editors."

Several subjects commented that "Shape distortion made placing objects slightly more difficult." A modification to the system could be to turn animation off when the object is moving slowly. A few subjects made the verbal comment that they would have liked separate control of the animation for different operations. For example, the animation for the rotation could be set to a different level than the animation for translation.

6.4 Summary

To show the effectiveness of the animation effects presented in this dissertation, I constructed a drawing editor, adraw, with an animated interface. The editor uses animation for its controls, and provides visual feedback for direct manipulation and indirect manipulation operations.

The editor incorporates a range of animation effects that augment the feedback for direct manipulation such as transforming objects and indirect manipulation operations such as aligning. The implementation showed that the cost of drawing animated feedback need not be prohibitive, and that good update rates can be achieved even on standard workstation hardware.

The results of two experiments on effectiveness and usability of the drawing editor were mixed. Although the first experiment did not conclusively demonstrate a significant difference between the animated and non-animated feedback, users were generally positive about the animation techniques. The results of the second experiment indicated that users prefer to adjust the drawing editor so that the effects of animation are clearly evident.

Chapter 7

Conclusion

The work makes four key contributions in the context of animating direct manipulation graphical interfaces:

1. It describes a set of animation effects based on those used by cartoon animators that extend the visual feedback for direct manipulation interfaces.
2. It shows the effects to be effective and enjoyable for users.
3. It presents the design and implementation of a set of software tools to support these animation effects.
4. It shows that convincing animation effects can be effectively incorporated into a significant application, a drawing editor.

The remainder of the chapter discusses the four contributions in more detail.

7.1 Animation Effects

Techniques used by cartoon animators can improve visual feedback in graphical user interfaces. Chang and Ungar listed three principles that apply to cartoon-like animation of user interfaces: the principles of solidity, exaggeration, and reinforcement. This work adds four more:

1. **The principle of attachment** states that the object being manipulated should at all times remain attached to the pointer.

2. **The principle of reluctance** states that objects should, in general, seem reluctant to change.

3. **The principle of smoothness** states that objects should change in a continuous fashion.

4. **The principle of anticipation** states that the result of a user's action should be obvious at all times.

In particular, this work has explored three ways of using animated visual feedback in graphical user interfaces: to smooth changes in the interface appearance, to suggest that manipulated objects have substance, and to clearly show the outcome of manipulation operations.

7.1.1 Smoothness in the Interface

The smoothness of the visual feedback in an interface can be improved by displaying change gradually. Standard cartooning techniques, such as squash and stretch, slow-in/slow-out, and exaggeration can further smooth the transition. These smoothing techniques reduce the cognitive effort needed for a user to assimilate the change by shifting some of the user's cognitive processing load onto their perceptive processing capacity.

To support smoothness in the interface, I constructed a new animated widget kit for the InterViews toolkit, and I developed tools to manage a history of time-stamped event information, to support the changing of graphical objects over time, and to construct autonomous animations. The new animated widgets included menus that open smoothly, buttons that flip and slide, check boxes with animated labels, and rotating palette menus.

7.1.2 Objects with Substance

Cartoon techniques can enhance the notion that graphical objects appear to be real and have substance. Distorting the representation of an object suggests that the object has inertia and flexibility. To help application programmers give their interfaces substance, I defined a set of distortion and follow through effects for common direct manipulation operations such as translating, scaling, rotating, and shape editing.

7.1.3 Anticipating the Result of a Manipulation

Good animation provides visual cues to the viewer as to what type of action is about to happen. In an interface, this technique can telegraph the result of a particular manipulation to the user. For example, animated visual cues can highlight constraints imposed by the application that will affect the outcome of a manipulation and they can suggest the outcome of an indirect manipulation operation.

To help programmers incorporate anticipation cues into interfaces, I designed software tools that use keyframe animation and distortion to suggest the way a manipulation affects an object. The keyframe animations provide visual continuity between different end states of a manipulation operation, and distortion suggests constraints such as pinning, snapping or the outcomes of indirect manipulation operations such as alignment and flipping.

7.2 Software Tools that Support Animation

To make it easy to incorporate animation effects into an application, I incorporated the animation technology into the lowest levels of the InterViews user interface toolkit. Specifically, I extended the InterViews drawing model, created a set of time management tools, and added an animated widget toolkit to the suite of existing look-and-feel toolkits. Although the work has been embodied as an extension to InterViews, the animation effects could also be implemented on other platforms.

7.2.1 Extending the Drawing Model

The InterViews toolkit uses a 2D affine drawing model. This work extends the drawing model to include non-affine warping transformations, and it adds a simple pseudo-3D capability that uses only the existing 2D rendering engine, and it provides some control over the quality of rendering. The warping model supports the kind of visual effects that animators use. Incorporating warping into the toolkit allows existing graphical objects in an application to be warped without increasing the complexity of the software model for the application programmer. The pseudo-3D transformation capability allows application programmers to construct applications that use 3D effects to describe smooth changes to objects. The rendering quality controls allow applications to progressively degrade rendering in order to maintain a rapid frame rates during manipulation operations.

7.2.2 Time Management Tools

InterViews, as with most current X-Windows user interface toolkits, provides only primitive support for manipulating objects that have a time-dependent behaviour. I have extended InterViews with a set of tools that help generate in-between frames in a keyframe animation, that support autonomous animations, and that save and retrieve information at arbitrary points on a time line.

7.2.3 Animated Widget Toolkit

The InterViews toolkit defines four look-and-feel widget sets: Motif, Silicon Graphics Motif, Open Look, and Monochrome. I added an animated look-and-feel that provides animated versions of the standard components. To use the new widgets, an application need only be recompiled against the new libraries; it will then operate in its original manner but with a new animated look-and-feel for its widgets.

7.3 An Application with an Animated Interface

To demonstrate the animation effects in a "real life" example, I developed a substantial application with an animated interface. I chose a drawing editor because of its heavy reliance on direct manipulation operations, and because it is complex enough to demonstrate these effects in multiple contexts. I performed a set of human factors experiments to evaluate the animation effects in the context of the drawing editor.

7.3.1 Adraw

The animated drawing editor, adraw, can be used to create and edit drawings made up of graphical objects like text, images, lines, splines, rectangles, and polygons. The editor uses the animated widget kit and incorporates the cartoon style techniques to smooth changes in the state of graphical objects, to give manipulated objects a sense of substance, and to help suggest the outcome of both direct and indirect manipulation operations.

7.3.2 Experiments

Two sets of experiments demonstrated that computer-literate users like the idea of cartoon-style animation in direct manipulation drawing editor interfaces. In particular, users felt that animation improved the visual feedback for constrained operations such as movement under the influence of gravity, and they rated the animation "enjoyable" and "intuitive" for a range of editing tasks. Moreover, when given a choice, users chose to set the magnitude of the animation effects to be clearly evident.

Although a user performance benefit has not yet been demonstrated for the animation techniques, the experiments suggested that such benefits are likely. Certainly, if the strength of users' responses is any guide, there are considerable gains to be made in applying animation principles to direct manipulation interfaces.

7.4 Future Work

This section identifies four areas for future work: automatic degradation of rendering quality to maintain rapid frame rate, other animation effects to improve current user interfaces, new applications that might profit from animation, and combining animation with other prosodic cues such as sound and gestures.

7.4.1 Reduction of Drawing Time

In this dissertation I have provided the low level infrastructure to support automatic degradation of rendering quality. More research is needed to develop better algorithms that can automatically apply these techniques to graphic-intensive applications.

7.4.2 Other Animation Effects

The work suggests several areas in which animation effects might be further developed. One extension would be to incorporate a major animation technique, motion blurring, into the extended InterViews toolkit. In addition, new types of animated widgets could be developed; animated menu items, animated text field editors, and Times-Square-like text labels are obvious candidates. Finally, more algorithms for constraint effects could be developed. For example, objects could push others out of the way when they bumped, or the cursor could stretch when attempting to move pinned objects.

The generalisation of the InterViews drawing model to incorporate warping suggests how other kinds of transformations could be incorporated. For example, imagine a "vanishing point" transformation that could be used to suggest perspective drawings, or a "fisheye" transformation that is capable of displaying objects under non-uniform scaling.

The indirect manipulation techniques presented in this dissertation work well where operations cause a change to an object attribute that can vary over a continuous range; the animation sets the attribute to successive intermediate values in the range. However, some operations remain problematic. For example, changing the font of a text label or the pattern of a filled rectangle has no obvious interpolation since these attributes are typically selected from a discrete palette of values. Further investigation is needed to determine how animation can benefit these and other operations.

7.4.3 New Applications

The animation effects trialed in the context of the drawing editor should be incorporated into a new set of applications. The use of animation to highlight constraints in an application seems to be a fruitful area to investigate. One area currently being investigated is a structured graphical editor for the mapping of logical parallel processes onto a physical parallel architecture [84]. An application like this has many constraints and it is hoped animation will provide better visual cues to the user. A non-editor direct manipulation application, such as a window manager, is another likely target for animation.

7.4.4 Prosodic Interfaces

This work is part of a larger effort to build tools that allow programmers to augment their application interfaces with prosodic cues. Prosody is concerned with the tempo, rhythm, and meter of poetry and music. The prosodic interfaces project aims to extend these concepts into the domain of user interfaces, using animation to smooth changes and suggest constraints, sounds to augment the sense of engagement of users with the interface and gestures to increase the expressiveness of user input. Underlying all these techniques—animation, sound, and gestures—is a sense of time that is largely lacking in current user interface tools. The prosodic interfaces project is developing a model for the representation of time and time-dependent effects that will underpin all three lines of investigation. In the short term, the project goals are confined to the application of these techniques to the user interface itself. In the longer term, the work will be extended to support applications that manipulate time-dependent data such as video and audio.

7.5 Concluding Remarks

Animation shows much promise in improving the usability of graphical user interfaces and making them more fun to use. The response that I have had to this work so far indicates that it is fruitful to continue to pursue this area of research.

Appendix A

Videos of Animations Effects for Direct Manipulation Operations

Three video clips have been prepared to demonstrate animation effects for translating, scaling, and rotating. These can be viewed as Quicktime movies from the following web page:

`http://www.cis.unisa.edu.au/staff/thomas.b.h./home.html`

Appendix B

3D and 2D Transformation Equivalence

This appendix demonstrates that, under certain conditions, a 3D transformation is equivalent to a simpler 2D transformation. If a 3D graphical transformation,

$$\mathbf{M} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \tag{B.1}$$

is applied to an object that lies entirely in the plane $z = 0$, and the transformed object is viewed along the z axis under orthographic projection, then the transformation can be replaced by an equivalent 2D transformation,

$$\mathbf{D} = \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix}. \tag{B.2}$$

Proof: Consider the equivalent 3D transformation matrix, M', that is the result of multiplying the transformation matrix M with the orthographic projection matrix O where,

$$\mathbf{O} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.3}$$

Thus,

$$\mathbf{M}' = \mathbf{MO} = \begin{bmatrix} r_{11} & r_{12} & 0 & 0 \\ r_{21} & r_{22} & 0 & 0 \\ r_{31} & r_{32} & 0 & 0 \\ t_x & t_y & 0 & 1 \end{bmatrix}. \tag{B.4}$$

Since the object to be transformed lies within the plane $z = 0$, all points on the object are of the form $\mathbf{P} = [\ x\ y\ 0\ 1\ ]$. Thus all points on the transformed object are of the form $\mathbf{P}' = \mathbf{PM}' = [\ x'\ y'\ z'\ 1\ ]$, where:

$$\begin{aligned} x' &= r_{11} \cdot x + r_{21} \cdot y + r_{31} \cdot 0 + t_x, \\ y' &= r_{12} \cdot x + r_{22} \cdot y + r_{32} \cdot 0 + t_y, \text{ and} \\ z' &= 0 \cdot x + 0 \cdot y + 0 \cdot 0 + 0. \end{aligned}$$

Note that, as required, the z component of the transformed object is zero (the object lies in the $z = 0$ plane), and that the x and y components of the transformed object do not depend on r_{31} or r_{32} since the z component of the original object is zero. Finally, note that this result is identical to that obtained by the 2D transformation

$$\mathbf{P}' = \mathbf{PD} = [\ x\ y\ 1\ ] \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix} = [\ x'\ y'\ 1\ ], \tag{B.5}$$

Where

$$\begin{aligned} x' &= r_{11} \cdot x + r_{21} \cdot y + t_x \text{ and} \\ y' &= r_{12} \cdot x + r_{22} \cdot y + t_y \end{aligned}$$

Appendix C

Calculations for the In-Between Matrices for the Path_Animator

The keyframe interpolation process provides in-between matrices for two given matrices. Consider an initial matrix

$$\mathbf{I} = \begin{bmatrix} I_{11} & I_{12} & I_{13} & I_{14} \\ I_{21} & I_{22} & I_{23} & I_{24} \\ I_{31} & I_{32} & I_{33} & I_{34} \\ I_{41} & I_{42} & I_{43} & I_{44} \end{bmatrix} \tag{C.1}$$

and a final matrix

$$\mathbf{F} = \begin{bmatrix} F_{11} & F_{12} & F_{13} & F_{14} \\ F_{21} & F_{22} & F_{23} & F_{24} \\ F_{31} & F_{32} & F_{33} & F_{34} \\ F_{41} & F_{42} & F_{43} & F_{44} \end{bmatrix}. \tag{C.2}$$

Figure C.1 shows an example of an initial transformation matrix **I** and a final transformation matrix **F**, displayed as unit vectors along the axes of each of their coordinate systems. A transformation consisting of a translation, scaling, and set of rotations is applied to the graphics in the coordinate system defined by **I**, to produce graphics rendered in the coordinate system defined by **F**.

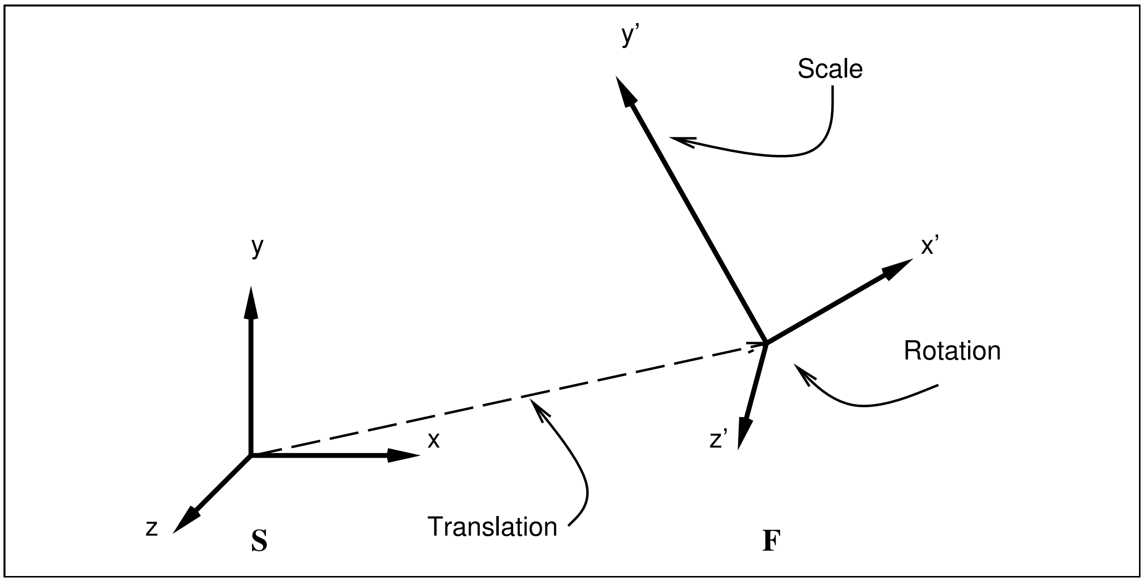


Figure C.1: Example of two coordinate systems

The interpolating function requires information about changes in translation (T), scaling (S), and rotation (R). Given **I**, **F**, and the interpolation function $\mathcal{B}$, the following nine quantities may be calculated to define the amount of change for each step of the animation: ΔT_X, ΔT_Y, ΔT_Z, ΔS_X, ΔS_Y, ΔS_Z, ΔR_X, ΔR_Y, and ΔR_Z.

The translation components, ΔT, are calculated by finding the incremental difference of the distance between the origins of the axes **I** and **F** as follows:

$$\Delta T_X = (F_{41} - I_{41}) \cdot \mathcal{B}, \tag{C.3}$$

$$\Delta T_Y = (F_{42} - I_{42}) \cdot \mathcal{B}, \text{ and} \tag{C.4}$$

$$\Delta T_Z = (F_{43} - I_{43}) \cdot \mathcal{B}. \tag{C.5}$$

The scaling components, ΔS, are determined directly from ratios of the magnitude of the unit axis vectors of **I** and **F**. The columns of the first three rows of a transformation matrix form the orthogonal unit axes vectors of its coordinate system. A change in the magnitude of these axis vectors corresponds to a scaling

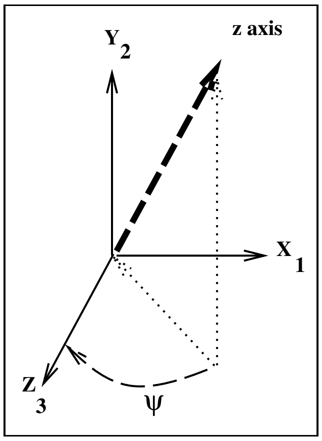


Figure C.2: Rotation about y axis

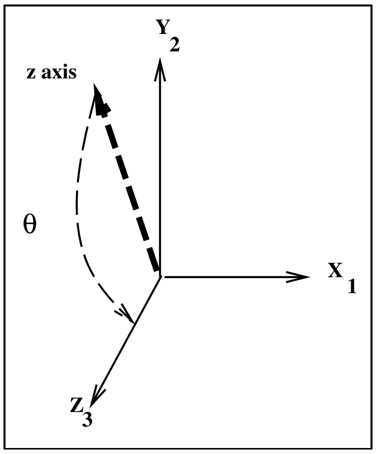


Figure C.3: Rotation about x axis

along that axis. The scaling components, ΔS, are determined as follows:

$$\Delta S_X = (S_{X_F} - S_{X_I}) \cdot \mathcal{B}, \tag{C.6}$$

$$\Delta S_Y = (S_{Y_F} - S_{Y_I}) \cdot \mathcal{B}, \text{ and} \tag{C.7}$$

$$\Delta S_Z = (S_{Z_F} - S_{Z_I}) \cdot \mathcal{B}, \tag{C.8}$$

where

$$S_{X_I} = \sqrt{{I_{11}}^2 + {I_{21}}^2 + {I_{31}}^2}, \tag{C.9}$$

$$S_{Y_I} = \sqrt{{I_{12}}^2 + {I_{22}}^2 + {I_{32}}^2}, \tag{C.10}$$

$$S_{Z_I} = \sqrt{{I_{13}}^2 + {I_{23}}^2 + {I_{33}}^2}, \tag{C.11}$$

$$S_{X_F} = \sqrt{{F_{11}}^2 + {F_{21}}^2 + {F_{31}}^2}, \tag{C.12}$$

$$S_{Y_F} = \sqrt{{F_{12}}^2 + {F_{22}}^2 + {F_{32}}^2}, \text{ and} \tag{C.13}$$

$$S_{Z_F} = \sqrt{{F_{13}}^2 + {F_{23}}^2 + {F_{33}}^2}. \tag{C.14}$$

Finally, the rotation components, ΔR, are determined by calculating a set of rotation angles that bring the axes described by the matrix **I** into alignment with the axes described by the matrix **F**. There are an infinite number of combinations of rotations that may be used to perform this alignment, and my algorithm uses a deterministic choice of combinations. The rotation ψ about the y axis is calculated first. This rotation brings the **I** z axis into the **F** y-z plane, as shown in Figure C.2. The angle of rotation is calculated as

$$\Delta R_Y = (\psi_F - \psi_I) \cdot \mathcal{B}, \tag{C.15}$$

where

$$\psi_I = \begin{cases} -arcsin(I_{31}/d_{I_1}) & \text{if } I_{33} >= 0.0 \\ 180.0 + arcsin(I_{31}/d_{I_1}) & otherwise \end{cases} \tag{C.16}$$

$$\psi_F = \begin{cases} -1.0 \cdot arcsin(F_{31}/d_{F_1}) & \text{if } F_{33} >= 0.0 \\ 180.0 + arcsin(F_{31}/d_{F_1}) & otherwise \end{cases} \tag{C.17}$$

$$d_{I_1} = \sqrt{{I_{33}}^2 + {I_{31}}^2} \tag{C.18}$$

$$d_{F_1} = \sqrt{{F_{33}}^2 + {F_{31}}^2} \tag{C.19}$$

The second angle, θ, of rotation is about the x axis, as shown in Figure C.3. This rotation brings the **I** z axis into alignment with the **F** z axis. The calculation of this angle of rotation is calculated as follows:

$$\Delta R_X = (\theta_F - \theta_I) \cdot \mathcal{B}, \tag{C.20}$$

where

$$\theta_I = \begin{cases} arcsin(I_{32}/d_{I_2}) & \text{if } I_{33} >= 0.0 \\ 180.0 - arcsin(I_{32}/d_{I_2}) & otherwise \end{cases} \tag{C.21}$$

$$\theta_F = \begin{cases} arcsin(F_{32}/d_{F_2}) & \text{if } F_{33} >= 0.0 \\ 180.0 - arcsin(F_{32}/d_{F_2}) & otherwise \end{cases} \tag{C.22}$$

and

$$d_{I_2} = \sqrt{{I_{33}}^2 + {I_{32}}^2} \tag{C.23}$$

$$d_{F_2} = \sqrt{{F_{33}}^2 + {F_{32}}^2} \tag{C.24}$$

The final rotation angle, ζ, about the z axis is calculated to bring the **I** y axis into alignment with the **F** y axis. Figure C.4 shows the final rotation. The calculation of this rotation is in two parts. First an intermediate y axis is calculated by rotating the **I** y axis, Eq C.16, and x axis, Eq C.21. The transformation matrices needed for this operation are as follows:

$$\mathbf{T}_I = \mathbf{R}_y(\psi_I)\mathbf{R}_x(\theta_I) \tag{C.25}$$

$$\mathbf{T}_F = \mathbf{R}_y(\psi_F)\mathbf{R}_x(\theta_F), \tag{C.26}$$

Where $\mathbf{R}_x(\theta)$ denotes a matrix that rotates about x axis by an angle of θ, and $\mathbf{R}_y(\psi)$ denotes a matrix that rotates about y axis by an angle of ψ. Finally, $\mathbf{R}_z(\zeta)$ denotes a matrix that rotates about z axis by an angle of ζ. The transformation matrices $\mathbf{T}_I$ and $\mathbf{T}_F$ are used to calculate the intermediate axes as follows:

$$\begin{bmatrix} y_{I_1} & y_{I_2} & y_{I_3} & 1 \end{bmatrix} = \begin{bmatrix} I_{12} & I_{22} & I_{32} & 1 \end{bmatrix} \mathbf{T}_I \tag{C.27}$$

and

$$\begin{bmatrix} y_{F_1} & y_{F_2} & y_{F_3} & 1 \end{bmatrix} = \begin{bmatrix} F_{12} & F_{22} & F_{32} & 1 \end{bmatrix} \mathbf{T}_F. \tag{C.28}$$

With these new axes, the final angle of rotation, ΔR_Z, about the z axis is calculated as

$$\Delta R_Z = (\zeta_F - \zeta_I) \cdot \mathcal{B}, \tag{C.29}$$

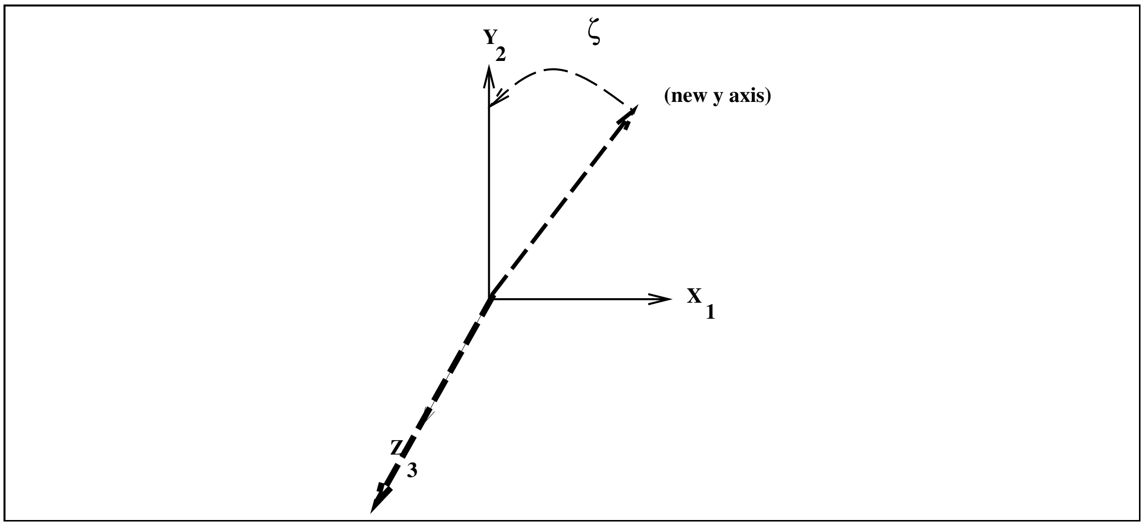


Figure C.4: Rotation about z axis

Where

$$\zeta_I = \begin{cases} arcsin(y_{I_1}/d_{I_3}) & \text{if } y_{I_2} >= 0.0 \\ 180.0 - (arcsin(y_{I_1}/d_{I_3}) & otherwise \end{cases} \tag{C.30}$$

$$\zeta_F = \begin{cases} arcsin(y_{F_1}/d_{F_3}) & \text{if } y_{F_2} >= 0.0 \\ 180.0 - (arcsin(y_{F_1}/d_{F_3}) & otherwise \end{cases} \tag{C.31}$$

and

$$d_{F_3} = \sqrt{y_{I_1}{}^2 + y_{I_2}{}^2 + y_{I_3}{}^2} \tag{C.32}$$

$$d_{F_3} = \sqrt{y_{F_1}{}^2 + y_{F_2}{}^2 + y_{F_3}{}^2}. \tag{C.33}$$

Once the nine components have been calculated, an interpolation matrix is constructed by the following transformation operations:

1. scale $\Delta S_X, \Delta S_Y, \Delta S_Z$;
2. rotate R_y;
3. rotate R_x;
4. rotate R_z; and
5. translate $\Delta T_X, \Delta T_Y, \Delta T_Z$.

The final matrix for the in-between frame of the animation can be determined as follows:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta T_X & \Delta T_Y & \Delta T_Z & 1 \end{bmatrix} \cdot \mathbf{R_y}\left(\Delta R_y\right) \cdot \mathbf{R_x}\left(\Delta R_x\right) \cdot \mathbf{R_z}\left(\Delta R_z\right) \cdot \begin{bmatrix} \Delta S_X & 0 & 0 & 0 \\ 0 & \Delta S_Y & 0 & 0 \\ 0 & 0 & \Delta S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{C.34}$$

Appendix D

Experiment 1 Exit Survey

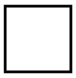
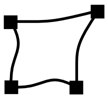
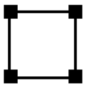
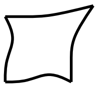
The exit survey presented in this appendix was given to each of the subjects at the end of the testing session for experiment 1. This survey asked subjects to indicate their attitude towards the various feedback effects on a 7 point scale.

Exit Survey

Please fill in this short survey.

Male ____ Female ____ Age ____

Indicate with a check how you liked the different visual effects shown in the editor. Please add comments on the back of sheet of what you thought of the visual effects.

		strongly like effect						strongly dislike effect
	no handles no warp	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7
	handles warp	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7
	handles no warp	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7
	no handles warp	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7

Appendix E

Experiment 2 Questionnaire for User-Interface Satisfaction

The questionnaire for user-interface satisfaction presented in this appendix was given to each of the subjects at the end of the testing session for experiment 2. The questionnaire asked the subjects about their impressions of the animation effects and the overall drawing editor.

Age: ______ Sex: Male ______ Female ______

1. Number of years experience using computers: ______
2. Number of years experience using mouse oriented computers: ______
3. Number of hours per week using window based computers? ______

Please circle the numbers which most appropriately reflect your impressions about using the graphical drawing editor. Not Applicable = N/A. There is room on the last page for your written comments.

Overall reaction to the system:

4.	terrible	1	2	3	4	5	6	7	wonderful	N/A
5.	frustrating	1	2	3	4	5	6	7	satisfying	N/A
6.	dull	1	2	3	4	5	6	7	stimulating	N/A
7.	difficult	1	2	3	4	5	6	7	easy	N/A

Overall reaction to the animation:

8.	terrible	1	2	3	4	5	6	7	wonderful	N/A
9.	frustrating	1	2	3	4	5	6	7	satisfying	N/A
10.	dull	1	2	3	4	5	6	7	stimulating	N/A
11.	interfering	1	2	3	4	5	6	7	helpful	N/A
12.	difficult	1	2	3	4	5	6	7	easy	N/A

With animation:

13. I felt more in control of the graphical object.	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A
14. The interaction was more natural.	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A
15. I had a better understanding of the outcome of operations.	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A
16. The manipulation of objects seemed smoother.	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A
17. The manipulated objects seemed to have substance.	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A

Animation in editors:

18. improves productivity	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A
19. is fun to use	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A
20. encourages me to work	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A
21. is intuitive to use	strongly disagree	1	2	3	4	5	6	7	strongly agree	N/A

Reaction to:

22. moving animation	strongly disliked	1	2	3	4	5	6	7	strongly liked	N/A
23. rotating animation	strongly disliked	1	2	3	4	5	6	7	strongly liked	N/A
24. scaling animation	strongly disliked	1	2	3	4	5	6	7	strongly liked	N/A
25. aligning animation	strongly disliked	1	2	3	4	5	6	7	strongly liked	N/A
26. colour animation	strongly disliked	1	2	3	4	5	6	7	strongly liked	N/A
27. gravity animation	strongly disliked	1	2	3	4	5	6	7	strongly liked	N/A

28. What other edit operations could be improved with the addition of animation?

29. What other applications could be improved with the addition of animation?

30. Please list three aspects you liked most about the animation.

31. Please list three aspects you liked least about the animation.

32. Please write any additional comments in the space below.

Appendix F

Implementation of Adraw

This appendix describes some of the implementation issues for adraw. The description is broken down into four major areas: the overall design of the application software, background to the InterViews *Glyph* class, the graphical object class *Gob*, and the manipulator object class *Manip*. The goal of this appendix is to provide an insight into how the software tools previously described may be used effectively.

F.1 Application Software

An overview of the layout of user interface components for adraw is shown in the Figure F.1. The status bar is a horizontal box containing text labels and a simple rectangle drawing to supply status information for the user about the current state of different attributes. The menu bar contains a set of pulldown menus to initiate editor commands. The set of radio buttons are used to engage the drawing tools, and the panner controls the viewing of the drawing surface. Finally, the drawing area is contained in the Renderer object.

The Renderer object controls the drawing of the hierarchy of Gobs (**G**raphical **Ob**ject**s**). These Gobs are extended glyphs, with methods to allow direct manipulation actions to be applied to them. The Renderer maintains a list, Goblist, of the top level Gobs in the hierarchy. A number of objects to control global manage-

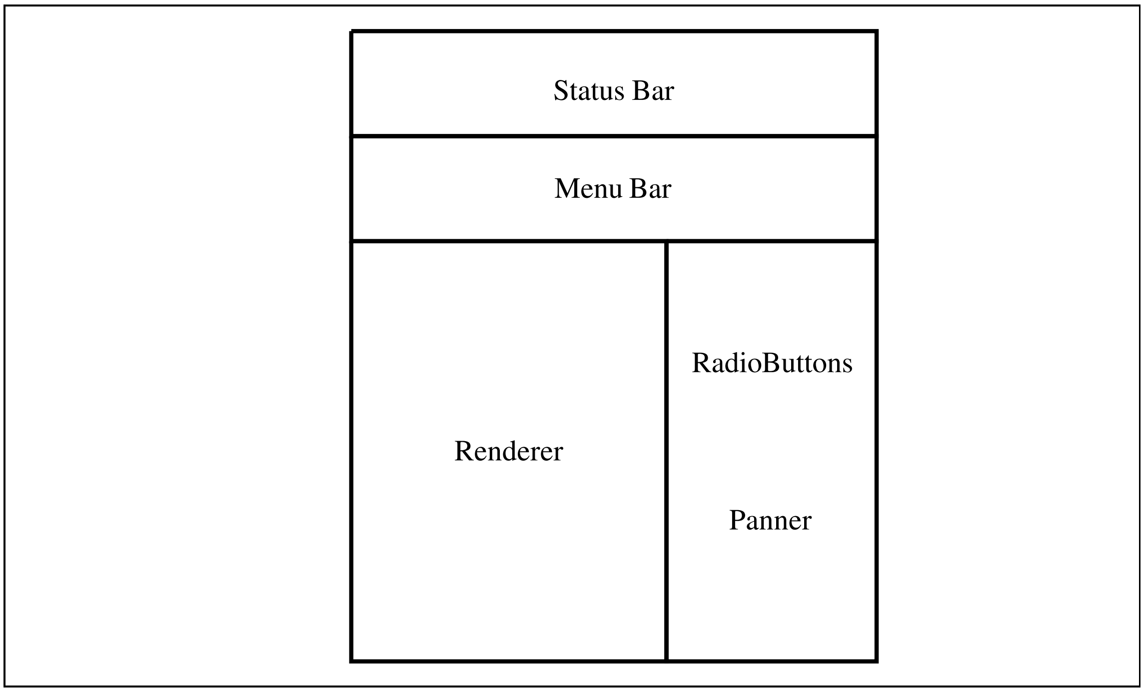


Figure F.1: Layout of the adraw user interface

ment of the Goblist are placed in the instance tree between the Renderer and the Goblist. Figure F.2 shows the object instance tree below the Renderer object. The **Patch** object performs global redraws and reallocations. The grid points are controlled and displayed with the **Grid** object. A black border around the Renderer is provided by the **Border** object. The **Printable** object sends PostScript versions of the drawing to a file or a printer. The view is adjusted with the **Tformsetter** object; the user adjusts the view with the panner or the *view* menu items.

An example of a drawing with a hierarchy of Gobs is shown in Figure F.3. This example shows the top level only having three objects: a group, an open spline and a closed spline. It is at this level that the direct manipulation operations are applied. The top level group object has four children: two groups of objects, a text label ("X Windows"), and a closed polygon. Each of the two group objects has a text label ("Hoodoo Man" and "Kool View") and a stroked figure. The object instance tree from the Goblist object is shown in Figure F.4. The leaves of the instance tree are single Gobs.

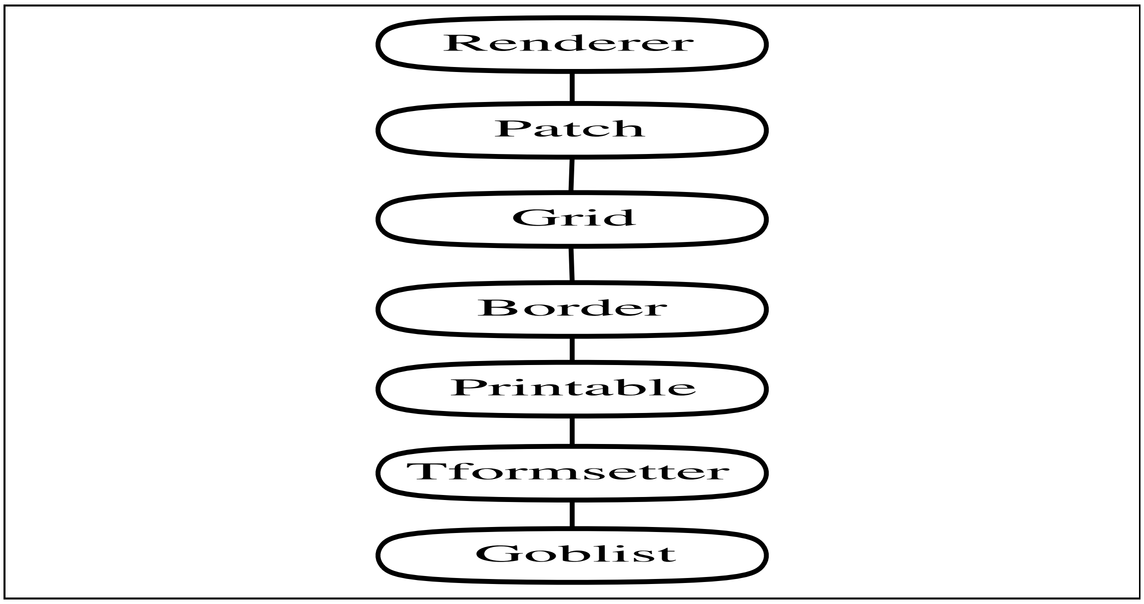


Figure F.2: The object instance tree for the Renderer's initial children

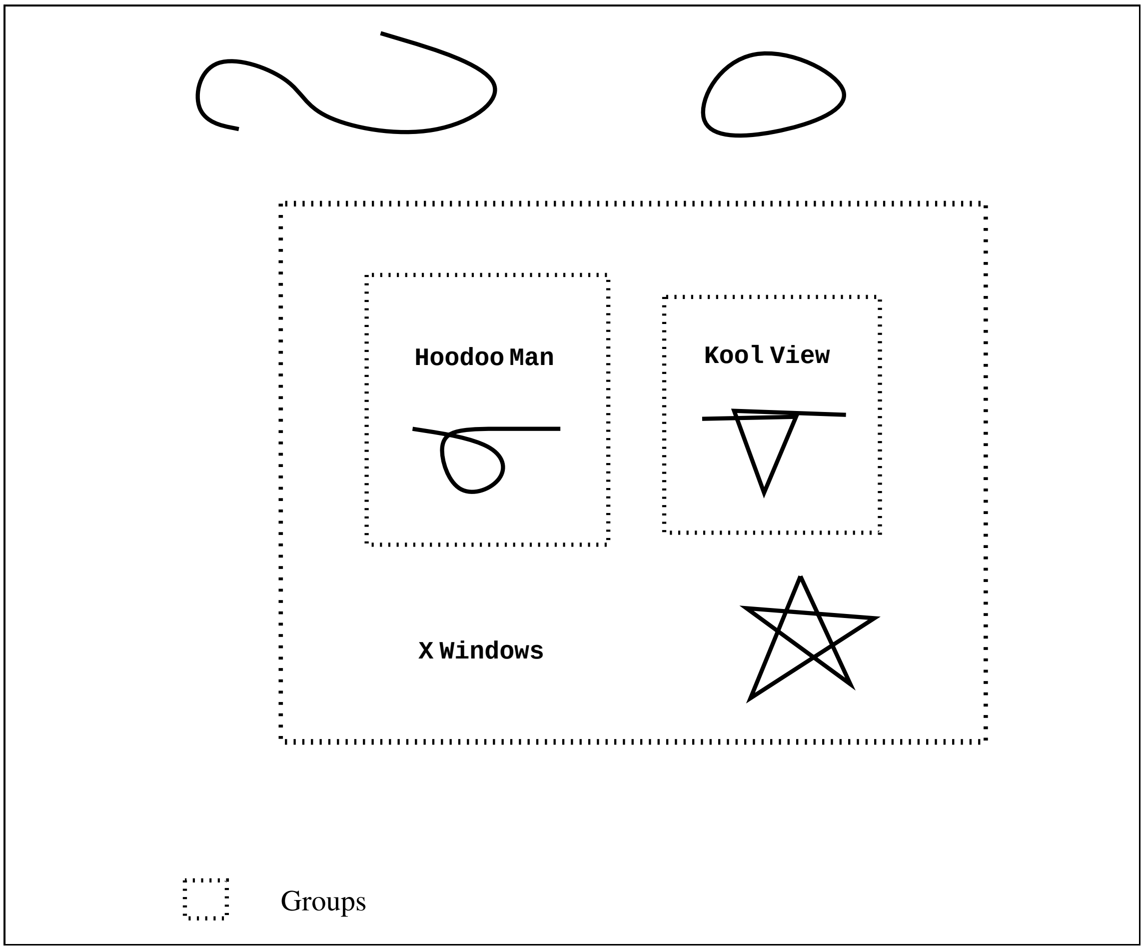


Figure F.3: An example of a Goblist with Groups as childern

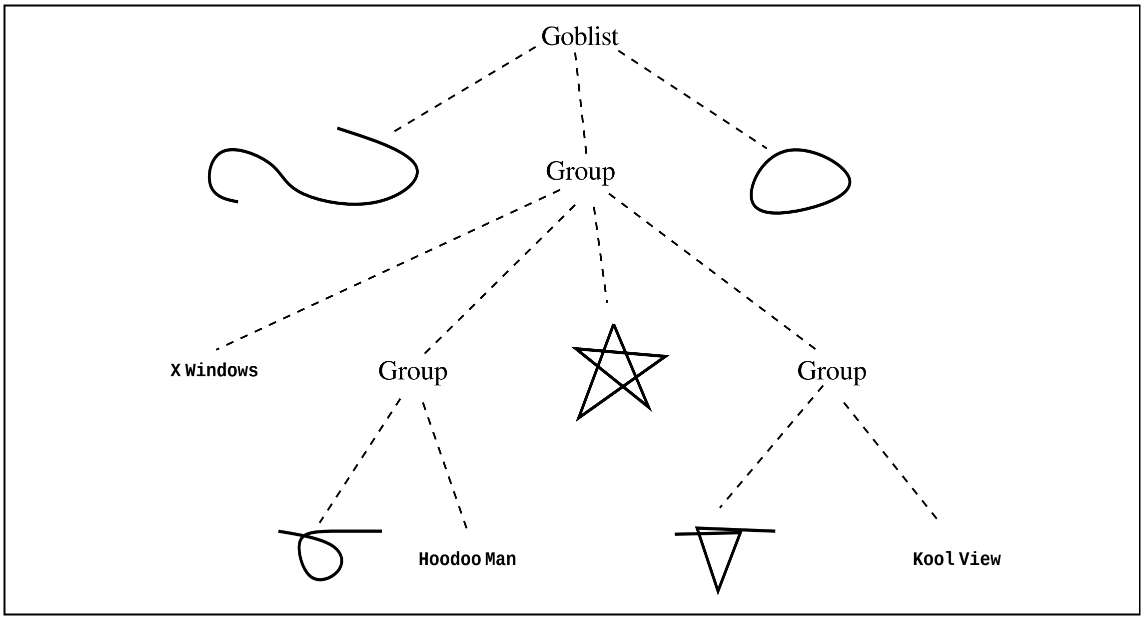


Figure F.4: The object instance tree for the example in Figure F.3

F.2 Glyph Class

This section provides a background to the InterViews Glyph class of that the Gob class is a subclass. InterViews is implemented as a library of C++ classes [5]. Chief amongst these classes is the *glyph* [15], which is the base class for a hierarchy of graphical objects that specify the appearance of InterViews-based applications. Glyph subclasses include simple graphical objects that define the elements of the interface, and composite objects that arrange other glyphs by tiling, overlaying, or stacking. An InterViews widget, itself a glyph, is composed of other glyphs that specify its appearance.

The Glyph class is a base class that contains no storage; its methods are passed parameters containing contextual information. Figure F.5 shows the five main methods for the Glyph class: **request**, **allocate**, **draw**, **print**, and **pick**. The **Glyph::request** method allows the Glyph to return its desired geometry through the **Requisition** parameter. The **Glyph::allocate** method tells the Glyph which portion of the canvas it has been allocated by way of the **Allocation** parameter; the method returns the actual area where it will be drawn (its extent) through

```
class Glyph : public Resource {
public:
...
    virtual void request(Requisition&) const;
    virtual void allocate(Canvas*, const Allocation&, Extension&);
    virtual void draw(Canvas*, const Allocation&) const;
    virtual void print(Printer*, const Allocation&) const;
    virtual void pick(Canvas*, const Allocation&, int depth, Hit&);
...
};
```

Figure F.5: A portion of the Glyph C++ class definition

the **Extension** parameter. In many cases the Allocation and the Extension are the same area on the screen. The **Glyph::draw** method requests the glyph to perform its drawing action on the given **Canvas**, at the location specified by the **Allocation** parameter. The **Glyph::print** method performs the same function, but to a **Printer** drawing surface instead of a display drawing surface. To enable a glyph to be identified by a rectangular region on the canvas, the **Glyph::pick** method is used. The **Canvas** and **Allocation** parameters of the **Glyph::pick** method perform the same function as with the **Glyph::draw** method. The **depth** and **Hit** parameters provide information for identifying the required glyph, and the **Hit** parameter returns information about which glyphs were "picked."

In the context of this dissertation, the most important Glyph method is **Glyph::draw**. A subclass of the Glyph class, Square, is defined here as an example of using the draw method to render on a canvas. This new Glyph draws a square 50 printer's points up and to the right from its base position.

Figure F.6 shows the source code for the **Square::draw**, using the same canvas drawing source code from the previous Canvas drawing example in Chapter 4. The **Canvas** parameter defines the rendering surface, and the **Allocation** parameter specifies the position to be rendered. The current position of the Square

```
void Square::draw(Canvas* c, const Allocation& a) const {
    Coord x = a.x();
    Coord y = a.y();
    c->new_path();
    c->move_to(x,y);
    c->line_to(x,y+50);
    c->line_to(x+50,y+50);
    c->line_to(x+50,y);
    c->close_path();
    c->stroke(colour,brush);
}
```

Figure F.6: Example of a Glyph draw method

glyph is obtained from the **Allocation**; the stroked figure is drawn relative to this position. As with the previous example, the figure is constructed using calls to **Canvas::move_to**, **Canvas::line_to**, and **Canvas::close_path**. Finally, a call to **Canvas::stroke** performs the "drawing" of the figure, with the supplied colour and brush parameters.

F.3 Gob Class

The **Gob** class is a subclass of MonoGlyph, which is in turn a subclass of Glyph. The Gob class extends the Glyph protocol to support directly manipulatable objects. The new functionality is broken down into six main areas: construction, duplicating, attribute setting, transformations, extension information, and warping support. The Gob class in turn is subclassed into six types of directly manipulable objects, as shown in Table F.1.

There are two options for where to place the attributes and special methods for the subclassed the Gob classes: in each subclass or in the Gob base class. Placing the attributes and special methods in the subclasses has the advantage that each subclass only stores and uses information it requires, but has the disadvantage of that operations cannot be performed generically on Gobs without unsafe type

Gob Class	Type of Drawing
Splinegob	Multi-point B-Spline stroked figure
Rectgob	Rectangle stroked figure
Groupgob	Group of Gobs manipulated as one
Polygob	Multi line segment stroked figure
Imagegob	Bitmap or TIFF image
Labelgob	Text label

Table F.1: Gob subclasses

Attribute	Splinegob Polygob	Rectgob	Groupgob	Imagegob Labelgob
selected	√	√	√	√
fill_color	√	√		
line_color	√	√		√
select_color	√	√	√	√
brush	√	√		
tr (Transformer)	√	√	√	√
draw_end_arrow	√			
draw_start_arrow	√			
warp	√	√	√	√

Table F.2: Gob subclass attributes

casting during execution. The superset design removes the type casting problem, but makes each object larger than it need be. For this implementation, I made the base class contain a superset of all attributes and methods needed by the subclasses. To see how effectively each subclass used the attributes and special methods, Table F.2 shows the base attributes supported by the Gob class and the subclasses that use them. The Gob class has 47 public methods, 40 of which are defined for all seven subclasses. Table F.3 shows the seven methods that are not defined for all subclasses, and the subclasses for which they have been defined. The remainder of this section describes the methods of the Gob's C++ class definition. The tables show that the superset design is justified in this instance because most subclasses use most of the attributes.

Method	Splinegob Rectgob Polygob	Groupgob	Labelgob	Imagegob
gob_to(x,y)	$\surd$			
gob_to(ch)			$\surd$	
ungob_last_point	$\surd$			
keystroke			$\surd$	
set_position			$\surd$	
move_point	$\surd$			
gob_component		$\surd$		

Table F.3: Gob subclass special methods

F.3.1 Construction

Figure F.7 shows the portion of the Gob class definition concerned with construction and initialisation. The constructor performs initialisation and sets the Gob's id value (each Gob has a unique id). The destructor cleans up any dynamically allocated memory. The **allocate**, **draw**, **pick**, and **request** methods provide the same functions as their Glyph counterparts.

The specific construction methods for Gobs entail manipulation of vertices and characters. The **Gob::gob_to** methods add either an additional vertex or control point to the end of a stroked figure or an additional character to the end of a Gob label. **Gob::ungob_last_point** removes the last point at the end of a stroked figure. Simple text edit manipulations for LabelGobs are provided by **Gob::keystroke**. **Gob::move_point** changes the specified vertex or control point of a stroked figure to a new position. Finally, **Gob::id** returns the Gob's id number, and **Gob::gob_component** retrieves a specified Gob from a group.

F.3.2 Duplicating

Figure F.8 shows the two duplication methods for Gobs. **Gob::copy** creates a new Gob subclass object, then copies the attribute information and returns a pointer to the new object. **Gob::copy_out** writes the Gob in adraw file format to an output

```
class Gob : public MonoGlyph {
public:
    Gob(const int id);
    ~Gob() {};
    virtual void allocate(Canvas*, const Allocation&, Extension&) ;
    virtual void draw(Canvas*, const Allocation&) const;
    virtual void pick(Canvas*, const Allocation& a, int depth, Hit& h);
    virtual void request(Requisition& requisition) const;
    virtual void gob_to (Coord x, Coord y);
    virtual void gob_to (char ch);
    virtual void ungob_last_point(){};
    virtual void keystroke(char key);
    virtual void move_point (Coord oldx, Coord oldy,
                             Coord newx, Coord newy);
    int id ();
    virtual Gob* gob_component(GlyphIndex) const;
```

Figure F.7: Gob C++ class definition (construction portion)

```
virtual Gob * copy (int & id);
virtual void copy_out(ostream * out);
```

Figure F.8: Gob C++ class definition (duplicating portion)

stream.

F.3.3 Attribute Setting

Figure F.9 shows Gob attribute methods that are used to define drawing characteristics of the graphical object. **Gob::set_line_color**, **Gob::get_line_color**, **Gob::set_fill_color**, and **Gob::get_fill_color** set and retrieve the line and fill colour attributes. The LabelGob uses the line_color attribute for its text colour. **Gob::set_font** and **Gob::set_brush** set the font and brush attributes. **Gob:: set_arrows** turns the beginning and ending arrow heads on and off. Finally, **Gob::set_position** sets the label cursor position in a LabelGob.

```
virtual void set_line_color(Color * );
virtual Color * get_line_color();
virtual void set_fill_color(Color * );
virtual Color * get_fill_color();
virtual void set_font (Font * , char *);
virtual void set_brush(Brush * );
virtual void set_arrows(boolean start_arrow, boolean end_arrow);
virtual void set_position(Canvas * , Coord );
virtual  void select();
void toggle_select();
virtual  void unselect();
boolean selected();
boolean set_select (Coord bx, Coord by, Coord ex, Coord  ey);
```

Figure F.9: Gob C++ class definition (attribute portion)

The Gob's select state information is set and retrieved by the following four methods: **Gob::select**, **Gob::toggle_select**, **Gob::unselect**, and **Gob::selected**. **Gob::set_select** determines if the extent of the gob is within a given rectangle. If the extent is within the rectangular region and the gob is unselected, the Gob's select state is set to true and the method returns a value of true. In the other cases, the Gob remains in its current selected state and the method returns false.

F.3.4 Transformations

Figure F.10 shows Gob methods that control a Gob's transformations. Each Gob maintains a Transformer object that is used to describe the current geometric transformation of the object. The **Gob::transformer** methods return and set the Gob's Transformer. The six values of the Gob's Transformer matrix can be directly set by **Gob::set_tr**. The following methods change the transformation coordinate system:

- **Gob::translate** moves the Gob relative to its current position.
- **Gob::rotate** rotates the Gob about its midpoint relative to its current angle.

```
Transformer  transformer();
void transformer(Transformer);
void set_tr(Coord m1, Coord m2, Coord m3,
            Coord m4, Coord m5, Coord m6);
virtual void translate (Coord x, Coord y);
virtual void rotate (Coord angle);
virtual void scale (Coord x, Coord y, Coord , Coord);
virtual void scale (Coord x, Coord y);
virtual void push_down_transformer(Transformer);
```

Figure F.10: Gob C++ class definition (transformer portion)

```
Extension * ext();
Coord left()const;
Coord right()const;
Coord top()const;
Coord bottom()const;
void mid_point (Coord & x , Coord & y);
```

Figure F.11: Gob C++ class definition (extension portion)

- **Gob::scale** scales the Gob about its midpoint or about a given point.

The **Gob::push_down_transformer** method premultiplies a given Transformer to the Gob's own Transformer. This method is used in the ungrouping operations, where the Transformer of the Groupgob is directly applied to each of its child Gobs.

F.3.5 Extension Information

The Gob object maintains its own extent information, and updates this information during the call to its **Gob::allocate** method. Figure F.11 shows the Gob methods that return extent information about the Gob. **Gob::ext** returns the complete extent. **Gob::left**, **Gob::right**, **Gob::top**, and **Gob::bottom** return the extremes of the Gob's extent. **Gob::mid_point** returns the Gob's midpoint.

```
  virtual void init_warp();
  virtual void set_warp(Warp *w);
  virtual boolean id_warp();
  virtual int get_warp_vertex(int start_point, Coord *x, Coord *y);
};
```

Figure F.12: Gob C++ class definition (warp portion)

F.3.6 Warping Support

Gobs do not determine their own Warp object. However, Figure F.12 shows Warp object maintenance routines that provide information and access methods for other objects. **Gob::init_warp** reinitialises the Gob's Warp object. **Gob::set_warp** sets the Gob's Warp object to a specified value. **Gob::id_warp** determines if the Gob is currently warped. The Gob determines which vertices or points are the base points for the bounded vectors used to warp itself, and returns these lists of points via **Gob::get_warp_vertex**. This method adds points to the given arrays of warp vectors, starting at the index **start_point** in the arrays.

F.4 Manipulators

The *Manip* class is an abstract class for defining objects to perform direct manipulation operations on Gob objects. Figure F.13 shows the class interface. There are currently eight subclasses of *Manip*: *Set_Att*, *Group*, *Align*, *Move*, *Select*, *Save*, *Edit*, and *Gob_To*. Table F.4 provides a brief description of each subclass.

The main role of the Manip class is to provide a consistent interface to manipulate the Goblist structure in the Renderer object. Manipulations are performed only at the highest level of the hierarchy of objects. Figure F.14 shows a code fragment illustrating how a subclassed Manip object performs a manipulation on selected Gobs. The **for** loop provides a means to examine every Gob at the top level

```
class Manip : public MonoGlyph {
public:
    Manip (ListGob * , Renderer *r , boolean warping=false);
    ~Manip ();
    void set_warp (boolean);
    void patch_it (MonoGlyph *g);
    void unpatch_it ();
    void redraw();
    void reallocate();
    Coord gravity_x (Coord x);
    Coord gravity_y (Coord y);
    Coord friction_y (Coord new_y , Coord old_y);
    Coord friction_x (Coord new_x , Coord old_x);
    virtual void idle (long , long);
    void set_ready_to_kill();
    void inertia(boolean _inert);
protected:
    void start_history();
    void end_history();
    void add_event(Coord x, Coord y, Coord angle,
                   Coord, Coord,unsigned long t);
    void add_event(Coord x, Coord y, Coord vec_x,
                   Coord vec_y, unsigned long t);
    void add_event(Transformer tr,  unsigned long t);
    void add_event(const ivEvent& e);
    int number_of_events();
    ObjectPosition * list_of_time_stamps();
    void past_event(Coord time_units, Coord & x1, Coord & y1,
                    Coord & angle, Coord & sx1, Coord & sy1);
    void past_event(Coord time_units, Coord & x1, Coord & y1,
                    Coord & vx1, Coord & vy1);
    void set_start_tr(Transformer tr);
};
```

Figure F.13: Manip C++ class definition

Manip Class	Type of Manipulators
Set_Att	Set or change the attribute of a selected set of Gobs
Group	Convert a selected set of Gobs into a group Gob
Align	Perform an indirect manipulation on a selected set of Gobs
Move	Change position, scale, or orientation of a selected set of Gobs
Select	Select one or more Gobs
Save	Write all Gobs out to a save file
Edit	Perform a cut, copy, duplicate, or paste operation
Gob_To	Add a vertex, control point, or letter to a Gob

Table F.4: Manip subclasses

of the hierarchy. Inside the loop is a test on the state of selection of a particular Gob. For Gobs that are selected, the manipulation is applied in three steps:

1. redraw the object before the manipulation,
2. apply the manipulation, and
3. redraw the object after the manipulation.

For manipulations that change the Gob's calculated **Extension**, the old and new areas defined by the two Extensions are redrawn. A Gob is redrawn by wrapping an InterViews **Patch** around the Gob, and the entire Goblist hierarchy is then reallocated. The area on the drawing canvas occupied by the Gob is then marked to be redrawn, and finally the **Patch** is removed.

A number of methods are implemented to support the drawing of the Gobs. The gravity algorithm defined in Section 5.3.2 is implemented in the Manip class through the two methods **Manip::gravity_x** and **Manip::gravity_y**. **Manip:: idle** is redefined by subclasses and is called during idle times in an animation. **Manip::set_ready_to_kill** is used to signal to the Manip object that it can free itself when all animations are complete.

The control of the warping animation is through a number of methods. **Manip::set_warp** tells the manipulator whether or not to apply warping animation, and the **Manip::inertia** method changes the warping to an inertia effect. The friction model of animation is implemented in a similar fashion as the gravity

```
for (ListItr(PolyGobList) i((*_list->main_list())); i.more(); i.next())
{
    Gob* g = i.cur();
    if (g->selected()) {
       patch_it(g);
       reallocate();
       redraw();
       unpatch_it();
       APPLY MANIPULATION TO g
       patch_it(g);
       reallocate();
       redraw();
       unpatch_it();
    }
}
```

Figure F.14: C++ source code for the manipulation of a gob hierarchy

model, using **Manip::friction_y** and **Manip::friction_x**. The time management methods were described in Section 4.4.

Bibliography

[1] Adobe Systems Incorporated. *PostScript Language Reference Manuel*. Addison-Wesley, Reading, Massachusetts, 1985.

[2] Tony Albrecht. *A 3D Graphics Extension to the InterViews Toolkit*. Flinders University, Bedford Park, South Australia, 1992.

[3] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–842, Nov 1983.

[4] Sherman R. Alpert. Self-describing animated icons for human-computer interaction: A research note. *Behaviour and Information Technology*, 10(2):149–152, 1991.

[5] AT&T. *C++ Language System Release 2.1, Product Reference Manual*.

[6] Ronald Baecker and Ian Small. Animation at the interface. In Brenda Laurel, editor, *The Art of Human-Computer Interface Design*. Addison-Wesley Publishing Co., Reading, MA, 1990.

[7] Ronald Baecker, Ian Small, and Richard Mander. Bringing icons to life. In *Proceedings of ACM CHI '91 Conference on Human Factors in Computing Systems*, pages 1–6, 1991.

[8] Lon Barfield, Eddy Boeve, and Steven Pemberton. Objects, Invariants and Treacle: Animation in the Views system. In *Proceedings of the HCI'93 Con-*

ference on People and Computers VIII, Tools and Techniques, pages 157–171, 1993.

[9] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. An introduction to the use of splines in computer graphics. University of Waterloo TR CS-83-09 / UC Berkeley TR UCB/CSD 83/13, Unversity of Waterloo and University of California, Berkeley, 1985.

[10] Krishna Bharat and Piyawadee "Noi" Sukaviriya. Animating user interfaces using animation servers. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Visualizing Information, pages 69–79, 1993.

[11] Alan Borning. Defining constraints graphically. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, Drawing and Animation Systems, pages 137–143, 1986.

[12] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, 1986.

[13] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems*, Visualization, pages 33–38, 1988.

[14] Marc H. Brown and Marc A. Najork. Algorithm animation using 3D interactive graphics. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Visualizing Information, pages 93–100, 1993.

[15] Paul Calder and Mark Linton. Glyphs: Flyweight objects for user interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 92–101, 1990.

[16] Stuart K. Card and D. Austin Henderson. Catalogues: A metaphor for computer application delivery. In *Proceedings of IFIP INTERACT'87: Human-Computer Interaction*, pages 959–964, 1987.

[17] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, Information Visualization, pages 181–188, 1991.

[18] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Animation/Visualization, pages 45–55, 1993.

[19] Stephane Chatty. Defining the dynamic behaviour of animated interfaces. In *Engineering for Human-Computer Interaction. IFIP TC2/WG2.7 Working Conference*, volume A-18, pages 95–111, Ellivuori, Finland, 10-14 Aug. 1992.

[20] Stephane Chatty and Michel Beaudouin-Lafon. Integrating animation with interfaces. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems – Posters and Short Talks*, page 70, 1992.

[21] Digital Equipment Corporation. *Multimedia Services for DEC OSF/1 AXP – Programmer's Guide*. Maynard, Massachusetts, 1994.

[22] Chris Crawford. Lessons from computer game design. In Brenda Laurel, editor, *The Art of Human-Computer Interface Design*. Addison-Wesley Publishing Co., Reading, MA, 1990.

[23] Misha Donskoy and Victor Kaptelinin. Window navigation with and without animation: a comparison of scroll bars, zoom, and fisheye view. In Steven Pemberton, editor, *Proceedings of ACM CHI'97 Conference on Human Factors*

in Computing Systems, volume Extended Abstracts, pages 279–280, March 1997.

[24] David Flanagan. *Volume 5: X Toolkit Intrinsics Reference Manual.* O'Reilly and Associates, Inc., Sebastopol, CA USA, 1992.

[25] James D. Foley. Interfaces for advanced computing. *Scientific American*, pages 127–135, October 1987.

[26] Open Software Foundation. *OSF/Motif Programmer's Reference.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[27] Didier La Gail. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[28] David Halliday and Robert Resnick. *Fundamentals of Physics.* John Wiley and Sons, Inc., New York, New York, 1974.

[29] D. Austin Henderson and Stuart K. Card. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics*, 5:211–243, 1986.

[30] E. Hoffert, M. Krueger, L. Mighdoll, M. Mills, J. Cohen, D. Camplejohn, B. Leak, J. Batson, D. Van Brink, D. Blackketter, M. Arent, R. Williams, C. Thorman, M. Yawitz, K. Doyle, and S. Callahan. Quicktime: an extenisble standard for digital multimedia. In *COMPCON Spring 1992*, San Francisco, CA, USA, February 92. IEEE.

[31] Scott E. Hudson and John T. Stasko. Animation support in a user interface toolkit: Flexible, robust and reusable abstractions. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Animation/Visualization, pages 57–67, 1993.

[32] id Software. Doom. http://www.idsoftware.com/.

[33] Steven K. Feiner James D. Foley, Andries van Dam and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., Reading, MA, 1990.

[34] Peter Karp and Steven Feiner. Issues in the automated generation of animated presentations. In *Proceedings of Graphics Interface '90*, pages 39–48, Halifax, NS, Canada, 14-18 May 1990.

[35] John Lasseter. Principles of traditional animation applied to 3D computer graphics. In *Proceedings of the ACM SIGGRAPH '87*, pages 35–44, Anaheim, CA, July 1987.

[36] Tom LaStrange, Jim Fulton, Steve Pitschke, Keith Packard, and Dave Payne. *tvtwm - Tom's Virtual Tab Window Manager for the X Window System*. Solbourne Computer.

[37] Brenda Laurel. *Computers as Theatre*. Addison-Wesley Publishing Co., Reading, MA, 1991.

[38] Kit Laybourne. *The Animation Book*. Crown Publishers, Inc., New York, NY, 1979.

[39] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pages 8–22, February 1989.

[40] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The perspective wall: Detail and context smoothly integrated. In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 173–179, 1991.

[41] Mark Mine. *Exploiting Proprioception in Virtual-Environment Interaction.* PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hil, Campus Box 3175, Sitterson Hall, Chapel Hill, NC 27599-3175 USA, 1997.

[42] Michael Moshell. Three views of virtual reality: Virtual envrionments in the US military. *IEEE Computer*, 26(2):81–82, February 1993.

[43] Brad Myers, Robert Miller, Rich McDaniel, and Alan Ferrency. Easily adding animations to interfaces using constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 119–128, Seattle, WA, USA, November 1996.

[44] National Center for Supercomputing Applications, University of Illinois in Urbana-Champaign. *NCSA Mosaic.*

[45] Adrian Nye. *Xlib Programming Manual for Version 11.* O'Reilly and Associates, Inc., Sebastopol, CA USA, 1990.

[46] Robert M. O'Keefe and Ivan L. Pitt. Interaction with a visual interactive simulation, and the effect of cognitive style. *European Journal of Operational Research*, 54:339–348, 1991.

[47] Susan Palmiter and Jay Elkerton. An evaluation of animated demonstrations for learning computer-based tasks. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, Systems for Training, pages 257–263, 1991.

[48] Susan Palmiter and Jay Elkerton. Animated demonstrations for learning procedural computer-based tasks. *Human-Computer Interaction*, 8(3):193–216, 1993.

[49] Susan Palmiter, Jay Elkerton, and Patricia Baggett. Animated demonstrations vs written instructions for learning procedural tasks: A preliminary investigation. *International Journal of Man-Machine Studies*, 34(5):687–701, 1991.

[50] Randy Pausch. Three views of virtual reality: An overview. *IEEE Computer*, 26(2):79–80, February 1993.

[51] James L Peterson. *Petri Net Theroy and The Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[52] B.A. Price, I.S. Small, and R.M. Baecker. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3), Sept 1993.

[53] Matthias Rauterberg, Marc Sperisen, and Markus Darwyler. From competition to collaboration through a shared social space. In *Proceedings of the 1995 East-West International Conference on Human-Computer Interaction*, pages II.94–II.101, Moscow, Russia, July 1995.

[54] Jun Rekimoto and Katashi Nagao. The world through the computer: Computer augmented interaction with real world environments. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 29–36, November 1995.

[55] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. The cognitive coprocessor architecture for interactive user interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, 3D/Gesture, pages 10–18, November 1989.

[56] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Three views of virtual reality: Nonimmersive virtual reality. *IEEE Computer*, 26(2):81–83, February 1993.

[57] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, Information Visualization, pages 189–194, 1991.

[58] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Information visualization using 3D interactive animation. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, Videos: Information Visualization, pages 461–462, 1991.

[59] Rick Rodriguez. *imagine, A Professional 3D Animation Rendering System for the Amiga – Tutorial Manual.* Impulse, Inc., Minneapolis, Minnesota, 1990.

[60] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–69, August 1983.

[61] John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? An empirical study and analysis. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, Understanding Programming, pages 61–66, 1993.

[62] John T. Stasko. A practical animation language for software development. In *1990 International Conference on Computer Languages*, pages 1–10, New Orleans, LA, USA, March 1990. IEEE.

[63] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, Programming by Demonstration, pages 307–314, 1991.

[64] John T. Stasko. *Trends in Software, Special Issue on User Interface Software*, chapter 5: Animation in User Interfaces: Principles and Techniques, pages 81–101. John Wiley & Sons Ltd, 1993.

[65] John T. Stasko and Doug Hayes. *XTANGO Algorithm Animation Designer's Package.* College of Computing, Georgia Institute of Technology, Alanta, GA 30332-0280, Oct 92.

[66] John T. Stasko and Joseph F. Wehrli. Three-dimensional algorithm animation. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems – Posters and Short Talks*, pages 79–80, 1992.

[67] Piyawadee Sukaviriya. Dynamic construction of animated help from application context. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 190–202, November 1988.

[68] Piyawadee Sukaviriya and James D. Foley. Coupling a UI framework with automatic generation of context-sensitive animated help. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Automatic Generation II, pages 152–166, November 1990.

[69] Inc Sun Microsystems. *OPEN LOOK Graphical User Interface Functional Specification.* Addison-Wesley Publishing Company, Inc., 1989.

[70] Ivan Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346, 1963.

[71] Steven H. Tang and Mark A. Linton. Pacers: Time-elastic objects. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Animation/Visualization, pages 35–43, November 1993.

[72] Thant Tessman and Gavin Bell. *Buttonfly.* Silicon Graphics Inc.

[73] Markus A. Thies. Animated help as a sensible extension of a plan-based help system. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 712–717, 1993.

[74] Bruce H. Thomas and Paul R. Calder. An animated widget kit for InterViews. In *Proceedings of the Computer Human Interaction Special Interest Group of the Ergonomics Society of Australia, OZCHI '94*, pages 203–208, Melbourne, November 1994.

[75] Bruce H. Thomas and Paul R. Calder. Using animation to enhance look and feel. Technical Report CIS-94-014, School of Computer and Information Science, University of South Australia, September 1994.

[76] Bruce H. Thomas and Paul R. Calder. Animated interaction for a graphical editor. In Helen Hasan and Cathy Nicastri, editors, *Proceedings of the Computer Human Interaction Special Interest Group of the Ergonomics Society of Australia, OZCHI '95*, pages 191–196, Wollongong, Australia, November 1995. Ergonomics Society of Australia CHISIG.

[77] Bruce H. Thomas and Paul R. Calder. Animating direct manipulation interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 3–12, Pittsburgh, November 1995. ACM SIGGRAPH and ACM SIGCHI.

[78] Bruce H. Thomas and Paul R. Calder. Animating widgets in the interviews toolkit. In *Proceedings of the 1995 East-West International Conference on Human-Computer Interaction*, pages 26–44, Moscow, Russia, Jul 1995.

[79] Bruce H. Thomas and Paul R. Calder. Graphical feedback for direct manipulation of simple polygons. Technical Report CIS-95-007, School of Computer and Information Science, University of South Australia, February 1995.

[80] Bruce H. Thomas and Paul R. Calder. Animating indirect manipulation in direct-manipulation editors. In John Grundy and Mark Apperley, editors, *Proceedings of the Computer Human Interaction Special Interest Group of the*

Ergonomics Society of Australia, OZCHI '96, pages 184–188, Hamilton, New Zealand, November 1996. Ergonomics Society of Australia CHISIG.

[81] Frank Thomas and Ollie Johnston. *Disney Animation: The Illustion of Life*. Abbeville Press, New York, NY, 1984.

[82] Elena Trichina and Bruce Thomas. 3D interactive animation for visualization of parallel design. Technical Report TR CIS-96-001, School of Computer and Information Science, University of South Australia, January 1996.

[83] Elena Trichina, Bruce Thomas, and Juha Oinonen. Visualization of data dependency analysis in parallel program design. In G. G. Grinstein and R. F. Erbacher, editors, *Visual Data Exploration and Analysis IV, IS and T SPIE 9th Annual Symposium*, volume 3017, San Jose, CA, USA, February 1997.

[84] Elena Trichina, Bruce Thomas, and Juha Oinonen. Visualization of data dependency analysis in parallel program design. In *The 2nd International Conference on Multimedia Information Systems (ICMIS 97)*, Motorola University, Schaumburg, Illinois, USA, USA, April 1997.

[85] George Turner and Beryl Turner. *Australian Oxford Paperback Dictionary*. Oxford University Press, 3rd edition, 1989.

[86] D. Volislav. Handmove: a system for creating animated user interface components by direct manipulation. In Leonard J. Bass and Claus Unger, editors, *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 151–166. Chapman and Hall, 1995.

[87] P. E. Waterson and C. E. O'Malley. Using animated demonstrations in multimedia applications: Some suggestions based upon experimental evidence. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2 of *III. Media*, pages 543–548, 1993.

[88] Patrick Waterson and Claire O'Malley. Using animated demonstrations to teach graphics skills. In *Proceedings of the HCI'92 Conference on People and Computers VII*, Graphical User Interfaces, pages 463–474, 1992.

[89] Matthias M. Wloka and Eliot Greenfield. The virtual tricoder: A uniform interface for virtual reality. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 39–40, November 1995.

[90] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA USA, 1992.