A Tutorial for InterViews Programmers
Part II: An Application Using Structured Graphics

John Vlissides

Abstract

This tutorial covers the design and implementation of a simple drawing editor application that uses a variety of InterViews components, including structured graphics. The tutorial first outlines the features of the application. Then it discusses the library components it uses that have not been described previously. Finally, the tutorial presents the structure and implementation of the application in detail.

1 Introduction

This tutorial describes minidraw, a substantial but straightforward application that employs several InterViews interface elements and structured graphics as well. Minidraw is a simple drawing editor that lets the user draw, move, and rotate rectangles. The implementation supports a direct manipulation interface that allows the user to point at and manipulate rectangles as if they were real objects. The user can also pan and zoom the drawing in its entirety.

The purpose of this tutorial is to present one way to design and build a non-trivial application. The tutorial reinforces object-oriented concepts by showing how they are used to simplify the implementation of complex behavior. It uses InterViews components to construct an interface that incorporates the basic elements found in many applications. You should read and understand the material presented in the preceding tutorial before delving into this one.

2 Overview

2.1 User Interface

Figure 1 shows how minidraw looks on the screen. The user draws and manipulates rectangles in the **drawing view**, the rectangular area on the left that occupies most of the screen space. To the right of the drawing view is the **tool palette** containing buttons for choosing the current drawing tool. The **rectangle tool** is used to create a rectangle, the **move tool** is used to move an existing rectangle, and the **rotate tool** is used to rotate an existing rectangle.

To add a rectangle to the drawing, the user first engages the rectangle tool by clicking the left mouse button on the **rect** button in the tool palette. The rectangle is drawn by first left-clicking and holding down the left mouse button in the drawing view to specify a corner. As the mouse is moved with the button held down (known as **dragging** the mouse), minidraw continuously draws a rectangle with opposing corners defined by two points: the point where the mouse was first clicked and the current mouse position. When the mouse button is released, a rectangle defined by the original point and the current mouse position is inserted into the drawing.

With the move tool engaged, the user can move an existing rectangle by left-clicking on it and dragging it to a new position. Minidraw animates the operation by drawing a bounding box that follows the trajectory of the mouse. Similarly, the user left-clicks and drags on an existing rectangle to rotate it; the operation is animated with a rotating bounding box that follows mouse motion.

Below the tool palette is a **panner**, which lets the user scroll the drawing in two dimensions at once as well as change the magnification at which it is viewed. Minidraw also provides a **pop-up menu** containing three **commands**: **clear** deletes all the rectangles in the drawing view, **normal size** returns the drawing view to its original magnification if it has been zoomed, and **quit** terminates the program's execution. The user presses the right mouse button anywhere in the drawing view to pop up the menu.

2.2 Architecture

Minidraw consists of four main classes:

1. The **Minidraw** class reads input and assembles the components of the interface in a

Research supported by the Quantum project through a gift from Digital Equipment Corporation.

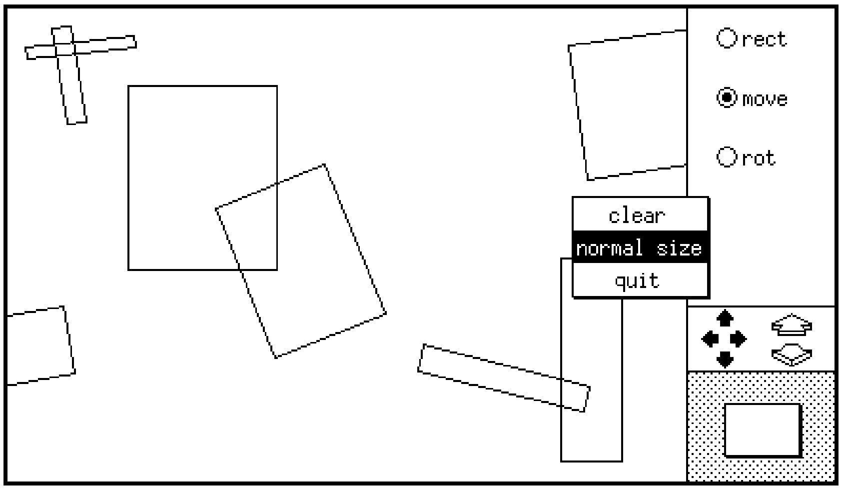

Figure 1: Minidraw user interface

manner analogous to the **Hello** class in the introductory tutorial.

2. The **View** class is the interactor that implements the drawing view. It contains and manages the user's drawing.

3. Each tool is represented by its own class. The **Tool** class is the base class from which the different tools are derived. The derived classes include **RectTool**, **MoveTool**, and **RotateTool**.

4. Each command is represented by its own class. The **Command** class is the base class from which different commands are derived. The derived classes include **ClearCmd**, **NormalSizeCmd**, and **QuitCmd**.

This set of classes represents those *defined* in minidraw and does not include InterViews classes *used* in the program. Figure 2 depicts the various classes schematically. We will describe the library components used in minidraw that have not been dealt with previously before discussing the minidraw classes themselves.

3 Library Components

3.1 Interactors

Several interactor subclasses are used in minidraw. The Minidraw class is derived from **MonoScene**, since it is the outer-level interactor that will compose the application's user interface. The View class is derived from **GraphicBlock**, an interactor that contains structured graphics. GraphicBlocks support scrolling, magnification, and reduction of the graphics they contain. The GraphicBlock class is part of the structured graphics library.

A panner lets the user pan and zoom the drawing. The inset clear rectangle represents the visible area in the view, while the gray area surrounding it represents the total area of the drawing. Dragging the rectangle within the gray area makes different parts of the drawing visible. The four small arrows in the top left of the panner scroll the drawing incrementally in one of four directions. The other two arrows in the upper right control zooming. Pressing the inward-pointing arrow enlarges the drawing; pressing the outward-pointing arrow reduces it.

The minidraw Tool class is derived from the **RadioButton** subclass of button to provide an interface for selecting the current tool. Radio buttons allow the user to select an option from several mutually-exclusive choices. Each radio button displays descriptive text to the right of a circle, which is filled when the radio button is pressed. All three radio buttons use the same buttonstate; when a radio button is pressed it sets the buttonstate's value to its own. A radio button indicates that it is pressed only when the buttonstate's value equals its own, so other radio buttons will unselect themselves automatically.

A **PopupMenu** interactor implements the pop-up menu containing the commands. Pop-up menus can contain **MenuItem** interactors that display a string of text and highlight themselves as the cursor enters their canvases. The PopupMenu

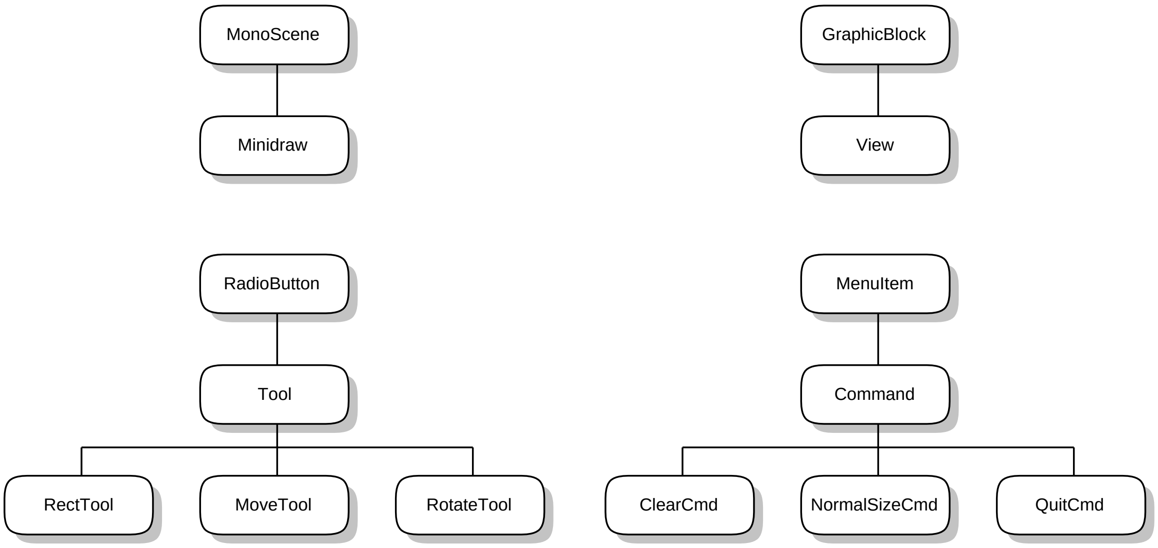

Figure 2: Classes defined in minidraw

class' Popup operation inserts the menu into the world at a location specified by an Event object. The menu removes itself automatically from the world when it receives an upclick. If the upclick occurred in one of its menu items, then the item will perform its Do operation. Subclasses of MenuItem redefine this operation to carry out the item's intended function.

3.2 Rubberbands

Minidraw uses **rubberband**s to animate the manipulation of rectangles. A rubberband varies its appearance as its **tracking point** is changed via the Track operation. In practice, the tracking point reflects the current mouse position. Rubberbands are *not* interactors; they are simply objects that interactors use to provide dynamic feedback.

Three subclasses of the Rubberband base class are used:

1. **RubberRect** is a rectangle defined by two points corresponding to opposing corners. One corner remains fixed while the other follows the tracking point. In minidraw, a rubber rectangle is used to animate the creation of a rectangle in the drawing.

2. **SlidingRect** is a rectangle of fixed size, shape, and orientation that slides around the screen as the tracking point moves about a reference point. A sliding rectangle is used to animate the movement of a rectangle in the drawing.

3. **RotatingRect** is a rectangle of fixed size and shape that rotates to follow the tracking point. A rotating rectangle is used to animate the rotation of a rectangle in the drawing.

3.3 Structured Graphics

Structured graphics (as opposed to *immediate-mode* graphics, as provided by painters in InterViews) lets the programmer treat graphical elements as objects. For example, a circle can be represented by a data structure that stores a center and a radius. Operations such as Draw, Copy, and Translate make it possible to display and manipulate the circle. A structured graphics package supports composition of primitive graphical elements into hierarchies that can be used like primitives. Structured graphics thus offer an abstraction that makes it easier to write programs such as drawing editors that let the user manipulate distinct graphical elements.

The InterViews library includes a sub-library called the **Graphic library** that provides a set of classes supporting object-oriented structured graphics. All structured graphics objects are derived from the **Graphic** base class. In this tutorial we will use the **Rect** and **Picture** subclasses of graphic. Rectangle objects are simple unfilled rectangles. Pictures are graphics that contain other graphics, allowing hierarchies of graphics to be built.

```
#include <InterViews/button.h>
#include <InterViews/menu.h>
#include <InterViews/Graphic/grblock.h>

class Picture;  // forward declarations
class RotatingRect;
class Rubberband;
class RubberRect;
class SlidingRect;
class Tool;
class View;
```

Figure 3: Header file information

Detailed information on the Graphic library appears in the InterViews Reference Manual [1] and is not repeated here. Later in this tutorial when we are discussing the implementation of minidraw, we will give brief explanations of what's going on wherever graphic objects appear. You should consult the reference manual if you encounter unfamiliar terms or concepts.

4 Implementation

The minidraw implementation comprises about 450 lines of C++ code. Two files are used: the class declarations are kept in minidraw.h, and their definitions are in minidraw.c. Figure 3 shows the information appearing at the beginning of minidraw.h. Class declarations follow the forward declarations. We will cover the class declarations first, then the implementation of the main program and globals, and finally the definitions of each class.

4.1 Class Declarations

Minidraw Class Declaration

The Minidraw class declaration appears in Figure 4. The Minidraw class encapsulates the entire application just as the Hello class did in the first tutorial. The minidraw object performs the following functions:

- It is inserted into the world in the main routine.
- It composes the user interface.

```
class Minidraw : public MonoScene {
public:
   Minidraw();
   void Popup(Event&);
   Tool* CurrentTool();
private:
   Interactor* Interior(View*);
   Interactor* ToolButtons(View*);
   void CreateMenu(View*);

   PopupMenu* menu;
   ButtonState* curTool;
};
```

Figure 4: Minidraw class declaration

- It pops up the command menu, which executes the appropriate command (if any).
- It keeps track of the current tool.

The member variables in the class declaration include the menu object and a buttonstate whose value represents the currently selected tool. Of the private operations, Interior composes the user interface, ToolButtons creates the palette of tools, and CreateMenu creates the command menu. Public operations include: the constructor; Popup, which pops up the command menu at the current cursor position; and CurrentTool, which returns the buttonstate's value.

View Class Declaration

Figure 5 shows the View class declaration. A view object is used to display and manipulate the drawing. The view implements operations that tools and commands use to affect the contents of the drawing. It also coordinates events during direct manipulation and detects the event that pops up the command menu.

View inherits from GraphicBlock the ability to pan and zoom a graphic. A picture is the graphic used to represent the drawing. User manipulations on rectangles map directly to operations on rectangle graphics in the picture. For example, when the user draws a rectangle, a rectangle graphic is appended to the picture; when the user rotates a rectangle, a rotation is applied to the corresponding rectangle graphic.

```
class View : public GraphicBlock {
public:
   View(Minidraw*, Picture*);
   virtual void Handle(Event&);

   void Add(Graphic*);
   void Clear();
   void Move(Graphic*, float, float);
   void Rotate(Graphic*, float);

   Picture* GetPicture() {
      return (Picture*) GetGraphic();
   }
   void InitRubberband(Rubberband*);
private:
   void Manipulate(Event&);
   Minidraw* minidraw;
};
```

Figure 5: View class declaration

The private part of the View class declares a pointer to the minidraw object and a Manipulate operation that abstracts the code associated with direct manipulation. The public part declares the following:

- the constructor
- a redefinition of the Handle operation
- Add, for adding a rectangle graphic to the picture that represents the drawing
- Clear, for deleting all rectangles from the picture
- Move, for translating a graphic in the picture
- Rotate, for rotating a graphic in the picture
- GetPicture, which returns the picture
- InitRubberband, which initializes a rubberband so that it can draw itself on the view's canvas

Tool Class Declarations

Figure 6 shows the Tool base class and subclass declarations. Tool is derived from RadioButton so that we can insert tools directly into a box to make a palette. The Tool class adds operations that abstract the behavior common to all tools. Of the protected members, Select returns the "topmost" rectangle graphic in the view's picture that intersects the coordinate position in the event. Select is used to determine which rectangle the user clicked on. Also protected is a pointer to the view object in which the tool will be used.

The public members include the constructor and three virtual functions: Setup, Manipulating, and DoIt. These three operations define what the tool does when the user uses it; they are redefined in each Tool subclass to implement the tool's desired behavior. Setup performs any one-time operations prior to the actual direct manipulation loop. For example, the move tool's Setup operation calls Select to determine which rectangle is being moved. Manipulating defines what happens during direct manipulation and the conditions under which the manipulation ends: `MoveTool::Manipulating` animates the motion of a rectangle by calling Track on a rubberband; it returns false when the user upclicks. Finally, DoIt implements the result of the manipulation: `MoveTool::DoIt` determines from the rubberband how far the rectangle was moved and calls Translate on the selected rectangle graphic. These three operations are called by `View::Manipulate` as will be shown in Section 4.4.

The Tool subclasses for the most part simply redefine the base virtual functions. In addition, the rectangle tool maintains a rubber rectangle rubberband to animate rectangle drawing, while the move and rotate tools use a sliding and rotating rectangle, respectively, to animate their behavior. The move and rotate tools also include a target graphic that is initialized to the rectangle selected in Setup and translated or rotated in DoIt.

Command Class Declarations

The Command base class and subclass declarations are shown in Figure 7. Command is derived from MenuItem so that commands can be included in the pop-up menu. A command contains a pointer to the view. Command subclasses redefine the Do function (which is otherwise inherited from MenuItem) to implement their semantics.

4.2 Main Program and Globals

Figure 8 shows the include files, globals, and main program appearing in the `.c` file. The main program is similar to those of the earlier tutorial

```
class Tool : public RadioButton {
public:
   Tool(
      View*, ButtonState*,
      const char* label
   );
   virtual void Setup(Event&) { }
   virtual boolean Manipulating(Event&) {
      return false;
   }
   virtual void DoIt(Event&) { }
protected:
   Graphic* Select(Event&);
   View* view;
};

class RectTool : public Tool {
public:
   RectTool(View*, ButtonState*);
   virtual void Setup(Event&);
   virtual boolean Manipulating(Event&);
   virtual void DoIt(Event&);
private:
   RubberRect* rubRect;
};

class MoveTool : public Tool {
public:
   MoveTool(View*, ButtonState*);
   virtual void Setup(Event&);
   virtual boolean Manipulating(Event&);
   virtual void DoIt(Event&);
private:
   SlidingRect* slidingRect;
   Graphic* target;
};

class RotateTool : public Tool {
public:
   RotateTool(View*, ButtonState*);
   virtual void Setup(Event&);
   virtual boolean Manipulating(Event&);
   virtual void DoIt(Event&);
private:
   RotatingRect* rotRect;
   Graphic* target;
};
```

Figure 6: Tool base class and subclass declarations

```
class Command : public MenuItem {
public:
   Command(View*, const char* label);
protected:
   View* view;
};

class ClearCmd : public Command {
public:
   ClearCmd(View* v) : (v, "clear") { }
   virtual void Do();
};

class NormalSizeCmd : public Command {
public:
   NormalSizeCmd(View* v) : (
      v, "normal size"
   ) { }
   virtual void Do();
};

class QuitCmd : public Command {
public:
   QuitCmd(View* v) : (v, "quit") { }
   virtual void Do();
};
```

Figure 7: Command base class and subclass declarations

except it calls InitPPaint, a predefined function that initializes the structured graphics library.

4.3 Minidraw Class Definition

The definitions of private minidraw operations appear in Figure 9. Interior composes the components into an hbox and returns it. The `controls` vbox composes the tool buttons and a panner. Panners change their height to reflect the drawing's aspect ratio; `Propagate(false)` keeps changes in the panner's height from affecting the screen space allocation (i.e., canvas) of interactors outside the vbox.

ToolButtons composes the tool buttons and fixed-size glue into a vbox, which is then placed into an hbox with fixed-sized glue on either side. Glue is used in both boxes to provide space between and around the tools. ToolButtons also initializes the curTool buttonstate to the rectangle tool so that it will be engaged when minidraw starts up.

```
#include "minidraw.h"
#include <InterViews/border.h>
#include <InterViews/box.h>
#include <InterViews/frame.h>
#include <InterViews/glue.h>
#include <InterViews/panner.h>
#include <InterViews/rubrect.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>
#include <InterViews/transformer.h>
#include <InterViews/world.h>
#include <InterViews/Graphic/picture.h>
#include <InterViews/Graphic/polygons.h>

void main (int argc, char** argv) {
   World* world = new World(
      "minidraw", argc, argv
   );
   InitPPaint();

   world->Insert(new Minidraw);
   world->Run();
}
```

Figure 8: Header file information, globals, and main program

Finally, CreateMenu builds the command menu by including commands directly into the pop-up menu.

The public member functions' definitions are shown in Figure 10. The constructor creates the view (with a new picture representing the drawing) and the command menu. Lastly, it inserts the interactor returned by Interior into itself. Popup pops up the command menu simply by calling the corresponding operation on the pop-up menu object. CurrentTool returns the value of the curTool buttonstate, which is the tool object engaged currently.

4.4 View Class Definition

The implementation of the View class appears in Figures 11 and 12. Figure 11 shows the basic operations, while Figure 12 shows those that support drawing editing features.

```
Interactor* Minidraw::Interior (
   View* view
) {
   VBox* controls = new VBox(
      ToolButtons(view),
      new VGlue,
      new HBorder,
      new Panner(view)
   );
   controls->Propagate(false);

   return new HBox(
       view,
       new VBorder,
       controls
   );
}

Interactor* Minidraw::ToolButtons (
   View* view
) {
   int space = round(.5*cm);
   RectTool* rectTool = new RectTool(
        view, curTool
   );
   curTool = new ButtonState;
   curTool->SetValue(rectTool);

   return new HBox(
      new HGlue(space, 0),
      new VBox(
         new VGlue(space, 0),
         rectTool,
         new VGlue(space, 0),
         new MoveTool(view, curTool),
         new VGlue(space, 0),
         new RotateTool(view, curTool),
         new VGlue(space, 0)
      ),
      new HGlue(space, 0)
   );
}

void Minidraw::CreateMenu (View* view) {
   menu = new PopupMenu;
   menu->Include(new ClearCmd(view));
   menu->Include(new NormalSizeCmd(view));
   menu->Include(new QuitCmd(view));
}
```

Figure 9: Minidraw class private member definitions

```
Minidraw::Minidraw () {
   View* view = new View(
      this, new Picture
   );

   CreateMenu(view);
   Insert(new Frame(Interior(view)));
}

void Minidraw::Popup (Event& e) {
   menu->Popup(e);
}

Tool* Minidraw::CurrentTool () {
   void* tool;

   curTool->GetValue(tool);
   return (Tool*) tool;
}
```

Figure 10: Minidraw class public member definitions

Basic Operations

The View constructor passes a nil sensor and the picture it was passed to the GraphicBlock constructor. Then it stores the minidraw object for future use, creates a sensor to detect up and down events,[1] and sets its preferred shape. The shape of the enclosing minidraw object is influenced most by the shape of the view, since the other interactors occupy a small fraction of a reasonably-sized minidraw window.

The Handle operation handles events within the view. It responds only to down events. A LEFTMOUSE button event initiates direct manipulation with the current tool, the details of which are abstracted by the Manipulate operation. A MIDDLEMOUSE event does nothing, since the interface does not use the middle button. Finally, a right-click in the view pops up the command menu; thus a RIGHTMOUSE event results in a call to Popup on the minidraw object.

InitRubberband exists for a rather obscure reason. Since rubberbands are not interactors, they

[1] The view is really only interested in down events initially, but since an up event cannot occur without a preceding down event, it won't hurt to listen for up events as well. Thus, we can avoid an extra call to Catch by specifying event interest entirely in the constructor.

```
View::View (Minidraw* m, Picture* p) : (
   nil, p
) {
   minidraw = m;
   input = new Sensor(updownEvents);
   shape->width = round(15*cm);
   shape->height = round(10*cm);
}

void View::Handle (Event& e) {
   if (e.eventType == DownEvent) {
      switch (e.button) {
         case LEFTMOUSE:
            Manipulate(e);
            break;
         case MIDDLEMOUSE:
            break;
         case RIGHTMOUSE:
            minidraw->Popup(e);
            break;
      }
   }
}

void View::InitRubberband (
   Rubberband* r
) {
   r->SetPainter(new Painter);
   r->SetCanvas(canvas);
}

void View::Manipulate (Event& e) {
   Tool* tool = minidraw->CurrentTool();

   tool->Setup(e);
   Listen(allEvents);

   do {
      Read(e);
   } while (tool->Manipulating(e));

   Listen(input);
   tool->DoIt(e);
}
```

Figure 11: View class member definitions for basic operations

```
void View::Add (Graphic* g) {
   Picture* pict = GetPicture();
   Transformer* t =
      pict->GetTransformer();
   Transformer* corrected;

   if (t != nil) {
      corrected = new Transformer(t);
      corrected->Invert();
      g->SetTransformer(corrected);
   }
   pict->Append(g);
   Update();
}

void View::Clear () {
   Picture* pict = GetPicture();
   Graphic* cur;

   while (!pict->IsEmpty()) {
      cur = pict->First();
      pict->Remove(cur);
      delete cur;
   }
   Update();
}

void View::Move (
   Graphic* g, float dx, float dy
) {
   float mag = GetMagnification();

   dx /= mag; // correct for current zoom
   dy /= mag;
   g->Translate(dx, dy);
   Update();
}

void View::Rotate (
   Graphic* g, float angle
) {
   float cx, cy;

   g->GetCenter(cx, cy);
   g->Rotate(angle, cx, cy);
   Update();
}
```

Figure 12: View class member definitions for drawing editing operations

must be passed a painter and a canvas on which to draw. The rubberbands used by tools are drawn in the view, so InitRubberband tells a rubberband to use the view's canvas. The painter is also set, mainly for consistency—that way the tool specifies neither a canvas nor a painter when it creates a rubberband.

The Manipulate operation provides the framework in which tools perform their function. It calls the current tool's Setup function in preparation for the direct manipulation loop, in which an event is read and interpreted by the tool's Manipulating operation. The loop is exited when Manipulating returns false, after which DoIt is called. Before the loop begins, Listen is called so that the view will receive all events and pass them to Manipulating. After the loop, the view limits event interest to up and down events by listening with its original sensor.

Editing Operations

Add, Clear, Move, and Rotate are used by commands and tools to edit the drawing. The operations represent the beginnings of a core of primitives that can be used to implement diverse functionality. To extend minidraw into a real drawing editor, we add operations like Cut, Paste, and Duplicate to the view. Then tools and commands are added that use these operations individually and in combination.

Add adds a graphic to the drawing. In our implementation the graphic is always a rectangle, but Add can add any graphic object. At first it might look like all we need to do is Append the graphic to the view's picture, but things are complicated by the fact that the picture may have been zoomed or panned.

The view is a GraphicBlock, which implements zooming and scrolling by respectively scaling and translating the graphic it contains. The rubberband used to size the graphic is specified in canvas coordinates; therefore a graphic in canvas coordinates will be distorted when it is drawn in the picture's coordinate system. To remedy this, the graphic is given an inverted version of the picture's transformer to use, effectively making the graphic's size and shape specification correspond to the picture's coordinate system.

After the graphic is appended, Update is called to make the newly added graphic appear in the view. Since the View class does not redefine Update, the entire view will be redrawn.[2]

Clear simply removes every graphic from the view's picture, effectively deleting every rectangle in the drawing. The implementation iterates over the picture's elements, removing the current one on each iteration. Update is called to redraw the display once the picture is empty.

Move translates a graphic by the given amount in canvas coordinate units. The amounts must be changed to reflect the current magnification, so they are divided by the amount returned by GetMagnification, an operation defined by the GraphicBlock class.

Rotate is the simplest of the editing operations: it just rotates the graphic about its center by the given angle. The angle is independent of the picture's coordinate system, so no special treatment is necessary.

4.5 Tool Definitions

Tool Base Class Definition

The Tool base class defines two non-trivial operations, the constructor and Select, as shown in Figure 13. Recall that a tool is a kind of radio button. The constructor passes its buttonstate and label string to the RadioButton constructor along with the tool's value (itself). It then initializes its view member variable.

Select returns the graphic (i.e., rectangle) in the drawing that intersects the mouse coordinates in the event. Select assumes the event and its contents are valid. Because the user may have difficulty in hitting precisely on the rectangle of interest, we introduce a "slop" factor by checking for a hit within 4 square pixels. We create a **BoxObj** (defined in the Graphic library) and pass it to a call to LastGraphicIntersecting on the picture, which returns either the topmost intersecting graphic or nil if none intersect. Select then returns this graphic to the caller.

```
Tool::Tool (
   View* v,
   ButtonState* b,
   const char* label
) : (label, b, this) {
   view = v;
}

Graphic* Tool::Select (Event& e) {
   const int slop = 2;
   BoxObj area(
      e.x-slop, e.y-slop,
      e.x+slop, e.y+slop
   );
   Picture* p = view->GetPicture();

   return
      p->LastGraphicIntersecting(area);
}
```

Figure 13: Tool class member definitions

Rectangle Tool Definition

All Tool subclasses define a constructor and redefine the base class' Setup, Manipulating, and DoIt operations. The rectangle tool definition appears in Figure 14. Its constructor simply forwards the parameters it is passed to the Tool constructor, adding only its radio button label. It also initializes the rubber rectangle variable to nil.

Setup creates the (initially degenerate) rubber rectangle using the coordinate in the event, and then it calls InitRubberband to set its canvas and painter. Manipulating is straightforward: if the event is a motion event, then it calls Track on the rubberband to animate the sizing of the rectangle; otherwise it signals termination of the manipulation on an up event.

DoIt is responsible for creating a rectangle graphic (based on the rubber rectangle's final configuration) and adding the graphic to the drawing. Once GetCurrent is called on the rubber rectangle to obtain its size and shape, it is no longer needed, so it is erased from the screen and destroyed. Then a rectangle graphic is created, given a brush, and inserted into the view's picture using the Add operation on the view. Note how Add simplifies the implementation of the tool by abstracting the insertion operation.

Move Tool Definition

The MoveTool constructor (Figure 15) is nearly identical to the rectangle tool's: it specifies its radio button label and initializes its sliding rectangle rubberband to nil. The Setup operation uses Select

[2] `View::Update` could be defined to perform more sophisticated, incremental screen update using the **Damage** class, but we won't go into that in this tutorial.

```
RectTool::RectTool (
   View* v, ButtonState* b
) : (v, b, "rect") {
   rubRect = nil;
}

void RectTool::Setup (Event& e) {
   rubRect = new RubberRect(
      nil, nil, e.x, e.y, e.x, e.y
   );
   view->InitRubberband(rubRect);
}

boolean RectTool::Manipulating (
   Event& e
) {
   if (e.eventType == MotionEvent) {
      rubRect->Track(e.x, e.y);
   } else if (e.eventType == UpEvent) {
      return false;
   }
   return true;
}

void RectTool::DoIt (Event&) {
   Coord x0, y0, x1, y1;

   rubRect->GetCurrent(x0, y0, x1, y1);
   rubRect->Erase();
   delete rubRect;
   Rect* rect = new Rect(x0, y0, x1, y1);
   rect->SetBrush(psingle);
   view->Add(rect);
}
```

Figure 14: Rectangle tool class member definitions

to initialize target to the rectangle clicked upon by the user. If no rectangle is hit, Select will return nil; otherwise, its bounding box is used to initialize the sliding rectangle. Thus, the sliding rectangle will circumscribe the selected rectangle when the user first moves it. As in `RectTool::Setup`, InitRubberband is called to initialize the sliding rectangle's canvas and painter.

MoveTool's Manipulating routine is identical to the rectangle tool's except for an extra test to avoid doing anything if the target is nil. Thus, nothing happens while the button is pressed down if no rectangle is being moved.

DoIt must move the target based on the sliding rectangle's final displacement. First a check is

```
MoveTool::MoveTool (
   View* v, ButtonState* b
) : (v, b, "move") {
   slidingRect = nil;
}

void MoveTool::Setup (Event& e) {
   Coord x0, y0, x1, y1;

   target = Select(e);
   if (target != nil) {
      target->GetBox(x0, y0, x1, y1);
      slidingRect = new SlidingRect(
         nil, nil, x0, y0, x1, y1, e.x, e.y
      );
      view->InitRubberband(slidingRect);
   }
}

boolean MoveTool::Manipulating (
   Event& e
) {
   if (
      target == nil ||
      e.eventType == UpEvent
   ) {
      return false;

   } else if (
      e.eventType == MotionEvent
   ) {
      slidingRect->Track(e.x, e.y);
   }
   return true;
}

void MoveTool::DoIt (Event&) {
   Coord x0, y0, dx, dy, dummy;

   if (target != nil) {
      slidingRect->GetOriginal(
         x0, y0, dummy, dummy
      );
      slidingRect->GetCurrent(
         dx, dy, dummy, dummy
      );
      dx -= x0;
      dy -= y0;
      slidingRect->Erase();
      delete slidingRect;
      view->Move(target, dx, dy);
   }
}
```

Figure 15: Move tool class member definitions

```
RotateTool::RotateTool (
   View* v, ButtonState* b
) : (v, b, "rot") {
   rotRect = nil;
}

void RotateTool::Setup (Event& e) {
   Coord x0, y0, x1, y1;
   float cx, cy;

   target = Select(e);
   if (target != nil) {
      target->GetBox(x0, y0, x1, y1);
      target->GetCenter(cx, cy);
      rotRect = new RotatingRect(
         nil, nil, x0, y0, x1, y1,
         round(cx), round(cy), e.x, e.y
      );
      view->InitRubberband(rotRect);
   }
}

boolean RotateTool::Manipulating (
   Event& e
) {
   if (
      target == nil ||
      e.eventType == UpEvent
   ) {
      return false;

   } else if (
      e.eventType == MotionEvent
   ) {
      rotRect->Track(e.x, e.y);
   }
   return true;
}

void RotateTool::DoIt (Event&) {
   float angle;

   if (target != nil) {
      angle = rotRect->CurrentAngle();
      rotRect->Erase();
      delete rotRect;
      view->Rotate(target, angle);
   }
}
```

Figure 16: Rotate tool class member definitions

made for a nil target, in which case nothing is done; otherwise the displacement is determined by comparing the sliding rectangle's original placement to its final (current) placement. This is done by calling GetOriginal and GetCurrent on the sliding rectangle. Since we can calculate displacement based on the change in position of only one point on the rectangle, we can call GetCurrent and GetOriginal with a dummy variable in place of the last two parameters.

After we determine the difference in the position of a point before and after manipulation, the sliding rectangle is no longer needed; it is erased and destroyed. Then the target is moved by the calculated displacement.

Rotate Tool Definition

The implementation of the rotate tool, shown in Figure 16, parallels that of the move tool. The constructor performs the same sort of initializations. Setup is identical except that a rotating rectangle is used instead of a sliding rectangle, thus requiring us to specify the point about which the rotating rectangle will rotate. Note that the mouse position in the event is supplied as a reference point for the rotating rectangle; thus, rotation will start from the point at which the mouse button was pressed.

The rotate tool's DoIt operation is even simpler than the move tool's, because rotating rectangles can calculate their total rotation based on the original and final orientations. Thus, we can simply get the angle of rotation from the rotating rectangle, erase and destroy it, and call Rotate on the view with the target and angle as parameters.

4.6 Command Definitions

The command definitions, shown in Figure 17, are particularly simple. The Command class is derived from MenuItem; the constructor simply passes the label and its horizontal alignment to the MenuItem constructor. The Command constructor then initializes its view member variable to the one it was passed.

Do is the only command operation that needs redefinition. The clear command redefines Do to call Clear on the view, while the normal size command's version calls the GraphicBlock SetMagnification operation with 1 as its argument. Finally, `QuitCmd::Do` exits the application.

```
Command::Command (
   View* v, const char* label
) : (label, Center) {
   view = v;
}

void ClearCmd::Do () { view->Clear(); }

void NormalSizeCmd::Do () {
   view->SetMagnification(1);
}

void QuitCmd::Do () { exit(0); }
```

Figure 17: Command base class and subclass member definitions

Exercises

1. Add tools for creating line and ellipse graphics. Use **RubberLine** and **RubberEllipse** rubberbands, respectively, to animate their creation.

2. Create a new tool, **DestroyTool**, that allows the user to click on a rectangle to remove it from the view's picture, after which it is deleted.

3. Implement a command, **CountCmd**, that does the following:

   (a) Counts the number of graphics in the view's picture.

   (b) Creates a dialog box containing a message and a pushbutton. The message should read: "The drawing contains x rectangles.", where x is the count determined previously.

   (c) Inserts the dialog box into the world, centered atop minidraw.

   (d) Removes the dialog when the user presses the pushbutton.

References

[1] Stanford University. *InterViews Reference Manual, Version 2.6*, 1989.