How to: use the graphics classes in `canvas.h`

H

H.1 The Graphics Library: TOOGL 1.0

The documentation in this section describes version 1.0 of TOOGL, the Tapestry Object Oriented Graphics Library (for Exploring and Experimenting)[1].

The graphics library consists of several classes for drawing and animating shapes. These classes provide support for client programs to create and manipulate shapes and images, and for the shapes to interact with the program and the user via the keyboard and mouse. The classes are built on a graphics engine underneath them doing the drawing and event-processing, the engine is not part of the library. The current implementation uses an engine created by a group at Carnegie Mellon University. The principal author of the CMU graphics engine is Geoff Washburn, the package is accessible via the following URL, this will also be linked to the website for this book (see below);

```
http://www.cs.cmu.edu/afs/cs/user/mjs/ftp
```

The current TOOGL classes are fully functional, but may evolve as they're more extensively used. In particular, the origin is currently fixed in the upper-left corner, with x-coordinates increasing to the right and y-coordinates increasing down the screen. Coordinates are expressed in pixels rather than in an absolute measure like centimeters. In the future the ability to choose the coordinate system will become part of the TOOGL classes and coordinates will be specified in centimeters or inches.

If you're reading this as part of *A Computer Science Tapestry*, the pictures of the screen images created by the graphics classes will be in black-and-white. For full-color pictures, and a much more extensive set of examples, including animations rather than still screen captures, see the supporting web pages at the following URL:

```
http://www.cs.duke.edu/csed/tapestry
```

The programs and examples in this Howto show the functionality of the graphics classes by using language features like arrays/vectors and inheritance. It's possible, however, to introduce every C++ concept with a graphical example, so that the first graphics programs might have no control statements, just shapes drawn on a canvas. Again, for a fuller treatment see the website for the book.

[1]TOOGL is pronounced too-gull, not too-gee-ell.

H.2 Using the Canvas Class

The basic window for drawing with TOOGL is an instance of the class `Canvas`, accessible by using `#include"canvas.h"`. A `Canvas` object is not double-buffered, and is intended for drawing shapes or figures once rather than as part of an animation. For drawing, redrawing, and animation, use the class `AnimatedCanvas` described in Sec. H.3.

H.2.1 Canvas Basics

A `Canvas` object is constructed by specifying its width, height, and distance in the x- and y-direction of the upper-left corner of the canvas from the upper-left corner of the screen; constructor parameters are integers. Any number of `Canvas` objects can be created in the same program. When a `Canvas` object is used, the standard console window is still visible, and all standard output streams are functional, so text and graphical output can be easily mixed. However, if the console window covers part of a `Canvas` window, the `Canvas` window may be erased when the console window is moved. To ensure that all windows in a program are visible, the first `Canvas` created in a program displays the message “click with mouse to begin”. Before clicking, you should move windows so that they don’t overlap, ensuring that the console window won’t erase any part of a `Canvas` window.

Syntax: Canvas constructor

```
Canvas can(width, height, x, y);
```

H.2.2 Drawing, Styles, and Colors

Program H.1, *circles.cpp*, draws seven circles, each in a different color. Circles are drawn in a filled style, in which the entire circle is filled with a color, and a frame style, in which just the outline of the circle is drawn. As the output of *circles.cpp* in Fig. H.1 shows, the default style of drawing uses filled figures, as though `Canvas::SetFilled()` had been explicitly called. The circles on the left of Fig. H.1 are filled, while the circles on the right are framed because the method `Canvas::SetFrame()` changes the drawing style just before the circles on the right are drawn. The largest radius circle must be drawn first when the filled style is used or else each drawn circle would completely obscure the circles drawn previously. To ensure that the graphics window remains visible after drawing has finished, the method `Canvas::runUntilEscape` keeps the graphics window showing until the escape key is pressed when the graphics window has the focus[2]. It’s also possible to use the free function `WaitForReturn()` in *utils.h*, which pauses until the user presses the return (or enter) key[3].

[2]A window *has the focus* when it is the active window. In most windowing systems you make a window active by clicking in the title bar of the window, or in the window itself.

[3]On many systems the return key must be pressed twice.

Program H.1 circles.cpp

```
#include "canvas.h"

// show simple Canvas functions, change style and color of drawing

void circles(Canvas& c, const Point& p, double size)
// post: series of circles drawn on c, centered at p
//       initial size = size (decreased by 20% for each one
{
    color spectrum[] = {CanvasColor::RED,    CanvasColor::ORANGE, CanvasColor::YELLOW,
                        CanvasColor::GREEN,  CanvasColor::BLUE,   CanvasColor::INDIGO,
                        CanvasColor::VIOLET};
    int k;
    for(k=0; k < 7; k++)
    {   c.SetColor(spectrum[k]);
        c.DrawCircle(p,size);
        size *= 0.80;
    }
}

int main()
{
    const int WIDTH = 250, HEIGHT = 150;
    Canvas c(WIDTH, HEIGHT, 20,20);
    circles(c, Point(WIDTH/4, HEIGHT/2),   WIDTH/4);
    c.SetFrame();
    circles(c, Point(3*WIDTH/4, HEIGHT/2), WIDTH/4);
    c.runUntilEscape();
    return 0;
}
```

circles.cpp

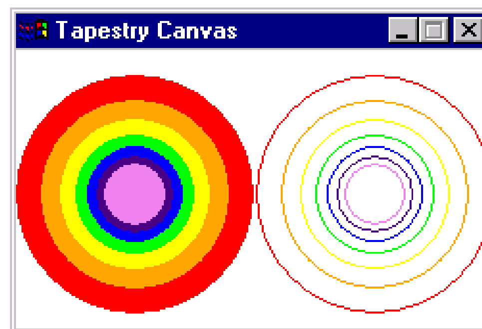

Figure H.1 Circles drawn in different colors and styles using *circles.cpp*, Prog. H.1

Table H.1 DrawXXX methods for the Canvas class. All methods are void.

Method	Prototype
DrawPixel	`(const Point& p);`
DrawRectangle	`(const Point& p1, const Point& p2);`
DrawCircle	`(const Point& center, int radius);`
DrawEllipse	`(const Point& p1, const Point& p2);`
DrawTriangle	`(const Point& p1, const Point& p2, const Point& p3);`
DrawPolygon	`(const tvector<Point>& a, int numPoints);`
DrawString	`(const string& s, const Point& p, int fontsize=14);`
DrawPieWedge	`(const Point& p, int radius, double startRad, double endRad);`

H.2.3 Drawing Shapes and Text

The DrawXXX methods described in Table H.1 make lines, curves, and other shapes appear on a Canvas object. Once set, the color and the style (filled/frame) in a canvas apply to all drawings, though both the color and style can be changed between invocations of DrawXXX methods using SetColor(..), SetFramed(), and SetFilled() as shown in Prog. H.1. Colors are described in Sec. H.3.8.

The DrawPieWedge[4] method draws a segment of a circle, whose center and radius are specified. Parameters startRad and endRad specify the angles (in radians) of the segment. For example, the call below draws a quarter-circle centered at point p with radius 100[5].

```
c.DrawPieWedge(p, 100, 0.0, PI/2);
```

Many of the Canvas methods are shown in *drawshapes.cpp*, Prog. H.2, which draws randomly-sized different shapes at random locations. Two runs are shown in Fig. H.2: the screen capture on the right uses the default filled drawing mode, the capture on the left uses the code shown in *drawshapes.cpp* where the call c.SetFrame() uses the framed, outline style for each figure.

Program H.2 drawshapes.cpp

```
#include "canvas.h"
#include "prompt.h"
#include "randgen.h"
#include "dice.h"
```

[4] The DrawPieWedge method is called by the StatusCircle class declared in *statusbar.h* and used in Prog. 6.15.

[5] The constant PI and functions to convert degrees to radians can be found in *mathutils.h*, see Howto G.

May 28, 1999 09:20 owltex Sheet number 17 Page number 793 magenta black


```
// fill screen with random shapes

Point getPoint(Canvas& c)
// postcondition: return a random point in Canvas c
{
    RandGen gen;
    return Point(gen.RandReal(0,c.width()), gen.RandReal(0,c.height()));
}

void drawShape(Canvas & c)
// postcondition: random shape/random size drawn on c
{
    const int NUM_SHAPES = 4;   // # different shapes
    const int MAX_SIZE = 30;    // max size of a shape
    Dice shapeDie(NUM_SHAPES);  // for randomizing selections
    Dice sizeDie(MAX_SIZE);     // for randomizing size

    Point p1(getPoint(c));
    Point p2(p1.x + sizeDie.Roll(), p1.y + sizeDie.Roll());

    switch (shapeDie.Roll())
    {
        case 1 :
          c.DrawRectangle(p1,p2);
          break;
        case 2 :
          c.DrawEllipse(p1,p2);
          break;
        case 3 :
          c.DrawCircle(p1, sizeDie.Roll());
          break;
        case 4 :
          c.DrawTriangle(p1,p2,getPoint(c));
          break;
    }
}

int main()
{
    const int WIDTH=  200, HEIGHT= 200;
    RandGen rnd;
    Canvas c(WIDTH,HEIGHT,20,20);
    int numSquares = PromptRange("# of shapes: ",1,1000);
    int k;
    for(k=0; k < numSquares; k++)
    {   c.SetFrame();
        c.SetColor(CanvasColor(rnd.RandInt(0,255), rnd.RandInt(0,255),
                               rnd.RandInt(0,255)));
        drawShape(c);
    }
    c.runUntilEscape();
    return 0;
}
```

drawshapes.cpp

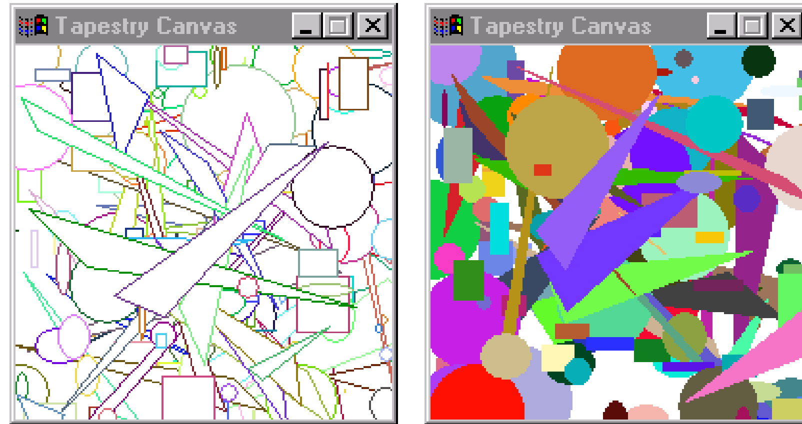

Figure H.2 Many shapes of random size and random color.

Using DrawText. As shown in Table H.1, the DrawText method has an optional parameter that specifies the font size. Prog. H.3, *grid.cpp*, shows the DrawLine and DrawText methods used to create the labeled grids in Fig. H.3.

Program H.3 grid.cpp

```
#include "canvas.h"     // for Canvas
#include "strutils.h"   // for tostring(int)
// illustrates line and text drawing in Canvas class
int main()
{
    const int GRID_SIZE = 200;
    const int SIZE=  20;   // fudge dimensions to make room for text
    Canvas c(GRID_SIZE+SIZE, GRID_SIZE+SIZE,100,100);
    int j;
    for(j=0; j <= GRID_SIZE; j+= SIZE)
    {   c.SetColor(BLACK);
        c.DrawString(tostring(j), Point(0,j));  // draw text labels
        c.DrawString(tostring(j), Point(j,0));
    }
    c.SetColor(BLUE);
    for(j=0; j <= GRID_SIZE; j+= SIZE)
    {   c.DrawLine(Point(j,0), Point(j,GRID_SIZE));  // horizontal line
        c.DrawLine(Point(0,j), Point(GRID_SIZE,j));  // vertical line
    }
    c.runUntilEscape();
    return 0;
}
```

grid.cpp

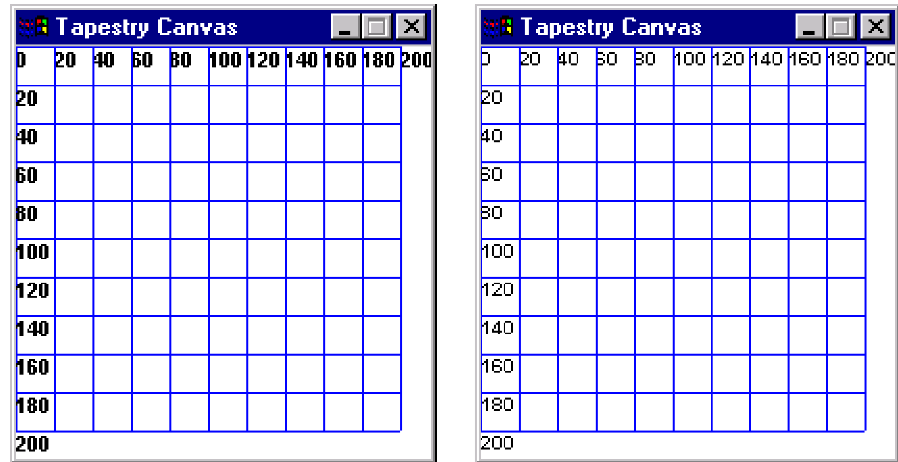

Figure H.3 Grids drawn with *grid.cpp*, on the left with with default font size of 14, on the right with a font size of 12.

H.3 Using the `AnimatedCanvas` Class

The class `AnimatedCanvas` supports the same methods that `Canvas` supports[6] but is double-buffered so it can be used for animations. Although both classes support the notion of `Shape` objects that draw themselves, these shapes don't work very well without double buffering.

Double buffering is a technique that uses two canvases, one for displaying and one for drawing, that makes flicker-free animations possible. All drawing takes place on an off-screen drawing buffer, which acts just like a canvas, but isn't visible. When drawing on the off-screen buffer is complete, the buffer is displayed on the visible canvas very quickly using an operation called **bitblt**, pronounced "bit blit".

H.3.1 The `Shape` Hierarchy

Client code draws directly on a `Canvas` object using one of the `DrawXXX` methods from Table H.1. In contrast, shapes are added to an `AnimatedCanvas`, and each shape knows how to draw itself on the canvas (using one or more of the `DrawXXX` methods). Client programs don't normally draw on an `AnimatedCanvas`, although it's possible to do so. It doesn't make sense to draw directly because each time the off-screen buffer is copied on-screen, any drawings made directly on the canvas will be erased.

[6]The class `Canvas` is actually a subclass of `AnimatedCanvas` without double-buffering. This means `Canvas` doesn't support animations. Both classes are subclasses of a class `BaseCanvas` that communicates with the underlying graphics engine.

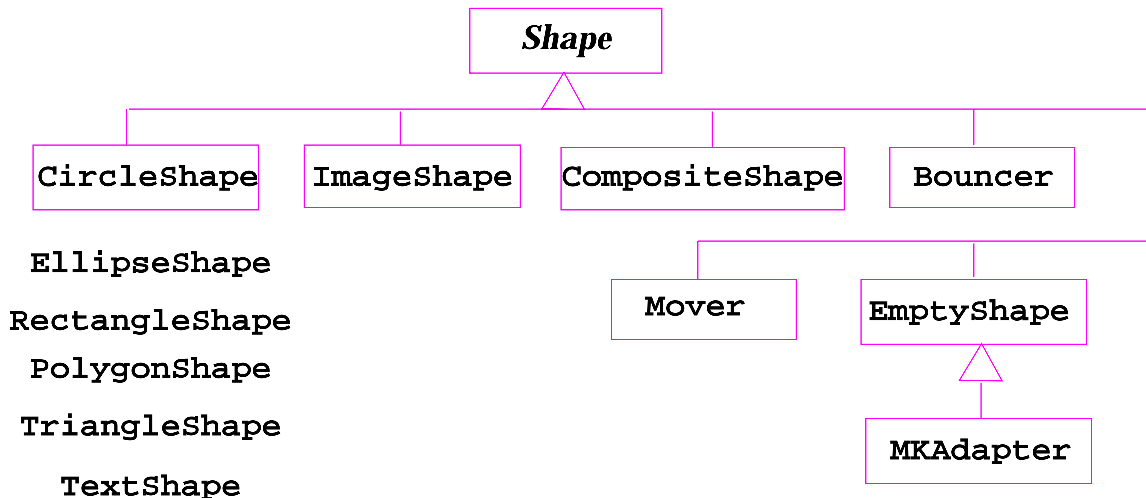

Figure H.4 The hierarchy of shapes in *shapes.h* used with the `AnimatedCanvas` class. The shapes on the left encapsulate a method of the corresponding name.

Instead, shapes are added to an `AnimatedCanvas` which then cycles through all the shapes asking each shape to draw itself. Animations are possible because a shape can draw itself at different locations. The double-buffering makes it seem as though the shapes are moving although what's actually happening is that all the shapes are erased, redrawn at new locations, and then displayed again.

The different shapes are accessible in *shapes.h* which is included as part of *canvas.h*. The shape inheritance hierarchy is shown in Fig. H.4. The classes on the left correspond to a `DrawXXX` method, the other classes extend the kind of shape objects and the behavior of shape objects.

H.3.2 Properties of `Shape` Objects

Every shape has a current position, a color, and a **bounding box**. The abstract base class `Shape`, from which all classes in the hierarchy in Fig. H.4 derive, is shown as Prog. H.4 which shows just the superclass. As the declaration shows, all derived classes must implement the following methods (they're abstract, or pure virtual, in `Shape`):

- `draw(AnimatedCanvas& c)` is the method that an `AnimatedCanvas` object calls, passing itself (the canvas), so that a shape object can draw itself.
- `setLocation(const Point& p)` sets the location of a shape object, see the related `bbox` method.
- `getLocation()` returns the current location of a shape object.
- `bbox()` returns the bounding box of a shape object. The bounding box is a

(minimal) rectangle that surrounds the shape. The bounding box is used to draw and detect overlap with other shapes.

- `clone()` returns a copy of a shape. The superclass `Shape` implements `clone` to return a NULL/0 pointer, which will cause immediate problems in most cases so subclasses should override `clone`.

In most cases at first you'll be using the classes in Fig. H.4 rather than creating your own classes.

Program H.4 abcshape.h

```
class Shape
{
    public:
      Shape();
      virtual ~Shape() {}

      virtual void draw(AnimatedCanvas& c)      = 0;
      virtual void setLocation(const Point& p) = 0;
      virtual Shape * clone();

      int            id  ()        const {return myCount;}
      virtual Point  getLocation() const = 0;
      bool           contains(const Point& p) const;
      bool           overlaps(const Shape& s) const;

      virtual Box    bbox()        const = 0;
      virtual string tostring()    const;

    protected:

      static int ourCount;
      int        myCount;
};
```

abcshape.h

H.3.3 Using Shapes: addShape and clone

Program H.5 shows how simple it is to create shapes with interesting behavior. A five-line program creates a bouncing ball, snapshots are shown in Fig. H.5[7]. The program creates a `CircleShape`, adds bouncing behavior to the circle by creating a `Bouncer` object from the circle, then adds the bouncer to the canvas and runs the program until the user presses the ESCAPE key. We'll discuss the `Bouncer` class in more detail in Sec. H.3.6, but the parameters are a shape, a direction in radians, and a velocity. One radian is approximately 57 degrees.

[7]See the website whose URL is given at the beginning of this Howto for access to an animation, or run the program.

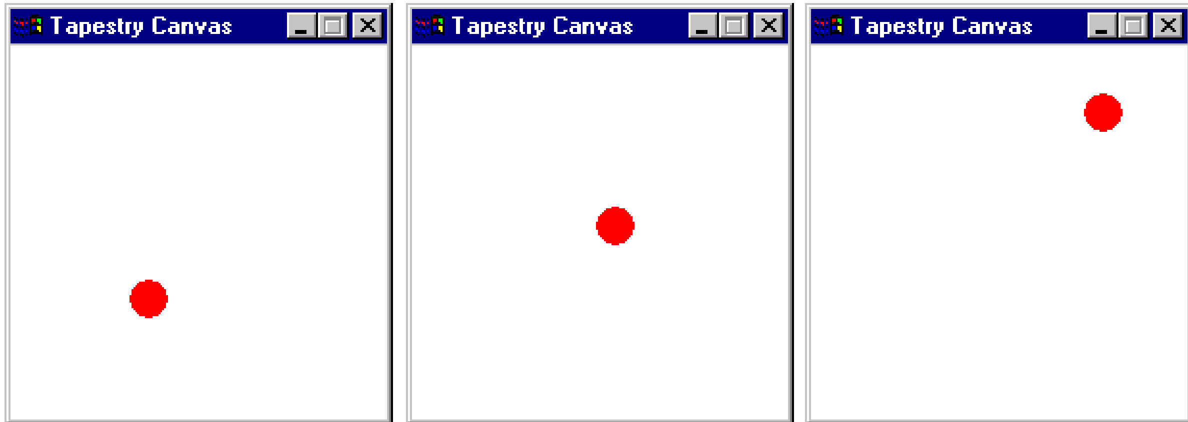

Figure H.5 Three snapshots of a bouncing ball in an `AnimatedCanvas`.

Program H.5 bouncedemo.cpp

```
#include "canvas.h"

// simple bouncer, one circle bouncing

int main()
{
    AnimatedCanvas canvas(200,200,20,20);
    CircleShape circle(Point(100,100), 10.0, CanvasColor::RED);
    Bouncer b(circle,1,2);
    canvas.addShape(b);
    canvas.runUntilEscape(10);
    return 0;
}
```

bouncedemo.cpp

An `AnimatedCanvas` object stores pointers to all the shapes it contains. Client code can create shapes on the heap by calling `new`, or can construct shapes as local (stack) variables as shown in Prog. H.5. Objects added to an `AnimatedCanvas` are cloned if they're not added as pointers[8]. In Prog. H.5, for example, the bouncer object `b` will be cloned by the `addShape` method. Normally client programs don't need to be concerned about cloning, but it's difficult to remove cloned shapes from a canvas. New shape classes may implement cloning improperly, so fewer bugs are usually encountered when shapes are allocated on the heap and not cloned.

[8]To be precise, `clone` is called when an object is passed by reference rather than by a pointer. Cloning can be circumvented using the address-of operator, but this is almost always a very bad idea.

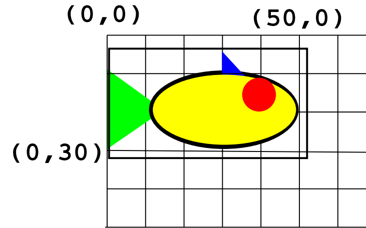

Figure H.6 The CompoundShape fish and its bounding box from *bouncefish.cpp*

H.3.4 The CompositeShape class

The class CompositeShape allows you to construct a new shape by combining several shapes together. Prog. H.6, *bouncefish.cpp*, shows how one fish is made from several shapes, then cloned to create an aquarium. Just as the addShape method clones shapes added to a canvas when the shapes aren't allocated on the heap, the CompositeShape::add method clones shapes not allocated on the heap. The CompositeShape fish that's part of *bouncefish.cpp*, Prog. H.6 is shown in Fig. H.6 with its bounding box, a snapshot of the bouncing fish is shown in Fig. H.7.

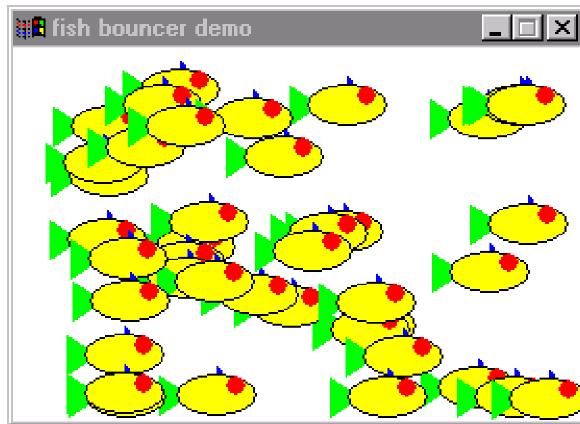

Figure H.7 An aquarium of 40 bouncing fish from *bouncefish.cpp*

Program H.6 bouncefish.cpp

```
#include <iostream>
using namespace std;
#include "canvas.h"
#include "randgen.h"
#include "prompt.h"
#include "mathutils.h"

int main()
{
    const int WIDTH=  300;
    const int HEIGHT= 200;
    AnimatedCanvas display(WIDTH,HEIGHT,20,20);
    RandGen rgen;

    display.SetTitle("fish bouncer demo");

    EllipseShape   body (Point(10,10), Point(50,30), CanvasColor::YELLOW);
    EllipseShape   bodyb(Point(9,9),   Point(51,31), CanvasColor::BLACK);
    CircleShape    eye  (Point(40,15), 5, CanvasColor::RED);
    TriangleShape  fin  (Point(30,5), Point(30,11), Point(35,11), CanvasColor::BLUE);
    TriangleShape  tail (Point(0,10), Point(0,30),  Point(15,20), CanvasColor::GREEN);
    CompositeShape fish;
    fish.add(fin);
    fish.add(tail);
    fish.add(bodyb);
    fish.add(body);
    fish.add(eye);

    // fish should start on grid, not bouncing

    const int MAX_FISH_X = WIDTH  - fish.bbox().width();
    const int MAX_FISH_Y = HEIGHT - fish.bbox().height();

    int numFish = PromptRange("how many fish: ",1,100);
    int k;
    for(k=0; k < numFish; k++)
    {   fish.setLocation(Point(rgen.RandInt(0,MAX_FISH_X),
                               rgen.RandInt(0,MAX_FISH_Y)));
        Bouncer fishb(fish, deg2rad(rgen.RandInt(0,360)), rgen.RandInt(2,7));
        display.addShape(fishb);
    }
    display.runUntilEscape(10);

    return 0;
}
```

bouncefish.cpp

H.3.5 Processing Mouse and Key Events

An `AnimatedCanvas` object cycles through all the shapes that have been added calling `Shape::draw` on each one. It's possible to have individual shapes respond to mouse clicks and key presses, but it's more often useful to add click/key behavior to an entire canvas. We'll discuss methods for doing this with TOOGL.

The simplest way to incorporate functions that respond to mouse presses and key clicks is to create a class that derives from `MKAdapter` (Mouse and Key Adapter) and add an instance of the new class to a canvas. In *circlefun.cpp* a simple class `MakeCircle` is created that responds to mouse clicks by creating a circle at the point of the click and labeling the center. The radius of the circle is chosen randomly as a multiple of five between five and thirty. Output from one run is shown in Fig. H.8

Program H.7 circlefun.cpp

```
#include <iostream>
using namespace std;

#include "canvas.h"
#include "dice.h"

// illustrate MKAdapter, make a circle where mouse is clicked

class MakeCircle : public MKAdapter  // stateless, make a circle where clicked
{
  public:
    MakeCircle()
    { }
    void processClick(const Point& p, AnimatedCanvas& ac)
    // post: circle of random radius created at mouse click point
    //       center labeled withcoordinates
    {
        Dice d(6);

        CircleShape circ(p,d.Roll()*5, CanvasColor::MAGENTA);
        ac.addShape(circ);
        ac.addShape(TextShape(p,p.tostring(),CanvasColor::BLACK));
    }
};

int main()
{
    AnimatedCanvas ac(200,200,20,20);
    MakeCircle mc;
    ac.addShape(mc);
    ac.runUntilEscape(10);

    return 0;
}
```

circlefun.cpp

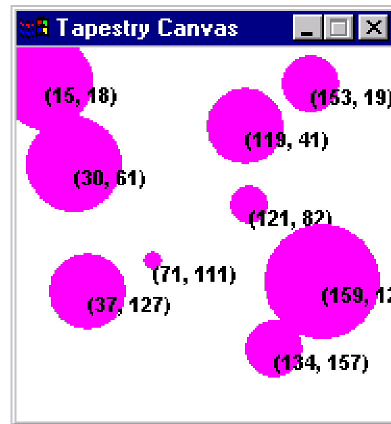

Figure H.8 Responding to mouse clicks by creating circles in Prog. H.7, *circlefun.cpp*

It's possible for a new class derived from `MKAdapter` to have a `processClick` method for responding to mouse clicks and a `processKey` method for responding to keys. Both methods are called by an `AnimatedCanvas` object which passes itself (the canvas) and either the point of the mouse click or the key press. An `MKAdapter` is also a `Shape` so that it can be added to a canvas via the `addShape` method. However, `MKAdapter` derives from `EmptyShape`, a shape with size zero and no drawing behavior. The `EmptyShape` class is often used to add behavior to a canvas via an invisible shape.

As a simple illustration of responding to key presses, Prog. H.8 implements a version of the toy *Etch-a-Sketch*. By pressing arrow keys, the user can create pictures by moving a drawing pen around the screen. This program uses a `Canvas` rather than an `AnimatedCanvas` because lines are drawn rather than shapes and we don't want to erase the lines via double-buffering. A rudimentary drawing using the program is shown in Fig. H.9

Program H.8 sketchpad.cpp

```
#include "canvas.h"

// illustrates MKAdapter with SketchAnEtch (stealing shamelessly
// from EtchASketch, a trademarked product)

class SketchPad : public MKAdapter
{
  public:

    SketchPad(const Point& start);    // begin to draw at start
```

```
    void processKey(const Key& key, AnimatedCanvas& c);

  private:
    static const int DELTA;          // each key-click moves this amount
    Point myPoint;                   // current point in drawing
};

const int SketchPad::DELTA = 2;
SketchPad::SketchPad(const Point& start)
  : myPoint(start)
{

}

void SketchPad::processKey(const Key& key, AnimatedCanvas& c)
// post: line drawn from oldpoint to newpoint in given direction
//       specified by key (arrow key)
{
    Point newPoint = myPoint;

    if (key.isuparrow())
    {   newPoint.y -= DELTA;
    }
    else if (key.isdownarrow())
    {   newPoint.y += DELTA;
    }
    else if (key.isleftarrow())
    {   newPoint.x -= DELTA;
    }
    else if (key.isrightarrow())
    {   newPoint.x += DELTA;
    }
    c.DrawLine(myPoint,newPoint);
    myPoint = newPoint;
}

int main()
{
    const int WIDTH=200, HEIGHT=200;
    Canvas c(WIDTH,HEIGHT,20,20);     // double buffering off
    c.SetColor(CanvasColor::BLACK);
    c.SetTitle("Tapestry SketchAnEtch");

    SketchPad sp = SketchPad(Point(WIDTH/2,HEIGHT/2)); // start in middle
    c.addShape(sp);
    c.runUntilEscape(10);

    return 0;
}
```

sketchpad.cpp

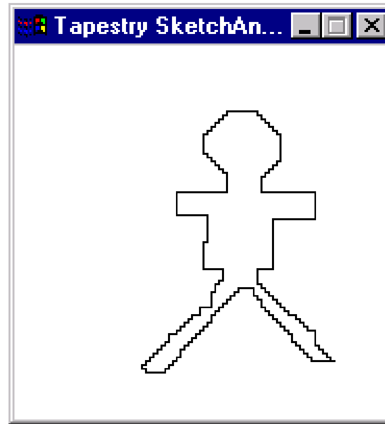

Figure H.9 Creative drawing using *sketchpad.cpp*.

H.3.6 Animations with Bouncer and Mover

The classes Bouncer and Mover make it relatively simple to write programs with animated shapes, either by using the classes directly or by creating new classes derived from them, but with motion behavior specific to a problem. A Bouncer object controls its own movement, a Mover object has its movement controlled by client code outside the class.

Bouncer Basics. When Bouncer objects are used, typically the objects are created, added to an AnimatedCanvas, and then the canvas is "run" for a set number of steps or until the user presses escape using, respectively AnimatedCanvas::run(int) or AnimatedCanvas::runUntilEscape(). Both functions take an optional second int parameter that specifies a millisecond delay between drawing cycles.

```
void AnimatedCanvas::run(int steps,int pause)
// post: all objects in canvas drawn for steps cycles
//       with pause millisecond delay (pause is optional)

void AnimatedCanvas::runUntilEscape(int pause)
// post: all objects drawn until user presses Escape
//       with pause millisecond delay (pause is optional)
```

A Bouncer object bounces off the borders of the window it's in, bouncing off so that the angle of impact equals the angle of reflection. Clients can subclass Bouncer to create different behavior when a border is hit. For example, objects could disappear from the left and re-appear on the right with the same y-coordinate, could change color, or could do nearly anything when hitting a border. There are two ways to change

behavior: by overriding Bouncer::update or one of the methods it calls. The update function is called just before any Bouncer object draws itself as shown in the code for Bouncer::draw.

```
void Bouncer::draw(AnimatedCanvas& c)
// post: bouncer updated and drawn
{
    update(c);
    myShape->draw(c);
}
```

As this example shows, a Bouncer object is a wrapper around an existing Shape object, stored in the bouncer as the instance variable myShape. The Bouncer class is an example of the design pattern **Decorator** [GHJ95].

Program Tip 8.1: (From [GHJ95]) Use *Decorator* to add responsibilities to individual objects dynamically and transparently and when extension by subclassing is impractical. For example, it would be impractical to create classes BouncingCircle, BouncingRectangle, BouncingText, and so on.

The word “transparent” here means that a shape object can be turned into a bouncing-shape without affecting other shapes and it can still be used as a shape.

Client programs can override update or one of the four methods update calls each time a bouncing object hits a border.

```
void Bouncer::updatetop    (AnimatedCanvas& c, Point& p)
void Bouncer::updatebottom (AnimatedCanvas& c, Point& p)
void Bouncer::updateleft   (AnimatedCanvas& c, Point& p)
void Bouncer::updateright  (AnimatedCanvas& c, Point& p)
```

A Bouncer subclass could, for example, override just updatebottom to change behavior when an object reaches the bottom of a canvas. It’s also possible to add behavior as shown in the example below from *backandforth.cpp* (this program is online, and not shown completely here.) A subclass of Bouncer, called BackAndForthBouncer, constructs objects that change appearance when they hit the left or right border. For example, by creating a left-facing fish that mirrors the fish from *bouncefish.cpp*, Prog. H.6, we can make the fish appear to swim back and forth as shown in Fig. H.10. We do this by simply storing two shapes and alternating between which one is displayed depending on the direction the object is moving. We override only two of the updateXXX methods as shown in the code that follows. Since we’re creating a new shape, we need to override clone as well or else the new class will have the same behavior as the Bouncer class since it inherits Bouncer::clone. All other inherited methods can be used as is.

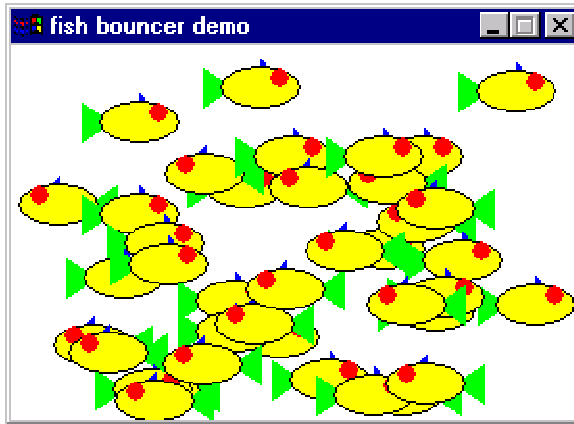

Figure H.10 Bouncing by creating a subclass as shown in *backandforth.cpp*

Program H.9 fishforth.cpp

```
// from backandforth.cpp
class BackForthBouncer : public Bouncer
{
  public:
    BackForthBouncer(Shape * left, Shape * right,
                     double angle, double velocity);
    virtual void updateright (AnimatedCanvas& c, Point& p);
    virtual void updateleft  (AnimatedCanvas& c, Point& p);
    virtual Shape* clone();
  protected:
    Shape * myLeft;
    Shape * myRight;
};

void BackForthBouncer::updateright(AnimatedCanvas& c, Point& p)
{
   myShape = myLeft;           // use left-facing shape
   myShape->setLocation(p);    // update location
   Bouncer::updateright(c,p); // bounce
}

void BackForthBouncer::updateleft(AnimatedCanvas& c, Point& p)
{
  myShape = myRight;           // use right-facing shape
  myShape->setLocation(p);     // update location
  Bouncer::updateleft(c,p);    // bounce
}
```

```
Shape* BackForthBouncer::clone()
{
  return
     new BackForthBouncer(myLeft,myRight,myAngle,myVelocity);
}
```

fishforth.cpp

H.3.7 Canvas Iterator

As a final example of extending the class Bouncer by subclassing, we'll show how to process collisions using the Canvas iterator class. Part of the class MoleBouncer from *bouncefun.cpp* shows how to determine if a shape collides with another shape. The overridden update function is shown in Prog. H.10. The complete program isn't shown, but it's nearly identical to *circlefun.cpp*, Prog. H.7, except it creates MoleBouncer objects where the mouse is clicked instead of drawing circles. Two snapshots are shown in Fig. H.11.

Program H.10 molebouncer.cpp

```
class MoleBouncer : public Bouncer
{
  public:
    MoleBouncer(Shape& s, double angle, double v)
      : Bouncer(s,angle,v)
   {  }
    virtual void update(AnimatedCanvas& ac)
    {
        RandGen rgen;
        Iterator<Shape> it(ac.makeIterator());
        bool collided = false;               // collision or still bouncing?
        Point p = getLocation();
        double angle = getAngle();
        for(it.Init(); it.HasMore(); it.Next())
        {   // check for collision, but not with myself
            if (it.Current().id() != this->id() && it.Current().overlaps(*this))
            {   ac.removeShape(this);
                ac.addShape(
                  new MoleBouncer(
                      CircleShape(Point(rgen.RandReal(0,ac.width()/10),
                                        rgen.RandReal(0,ac.height()/10)),
                      RADIUS, CanvasColor::BLUE), 2*PI - angle, 4));
                collided = true;
                break;
            }
        }
        if (!collided)           // no collision, update
        {   Bouncer::update(ac);
        }
    }
};
```

molebouncer.cpp

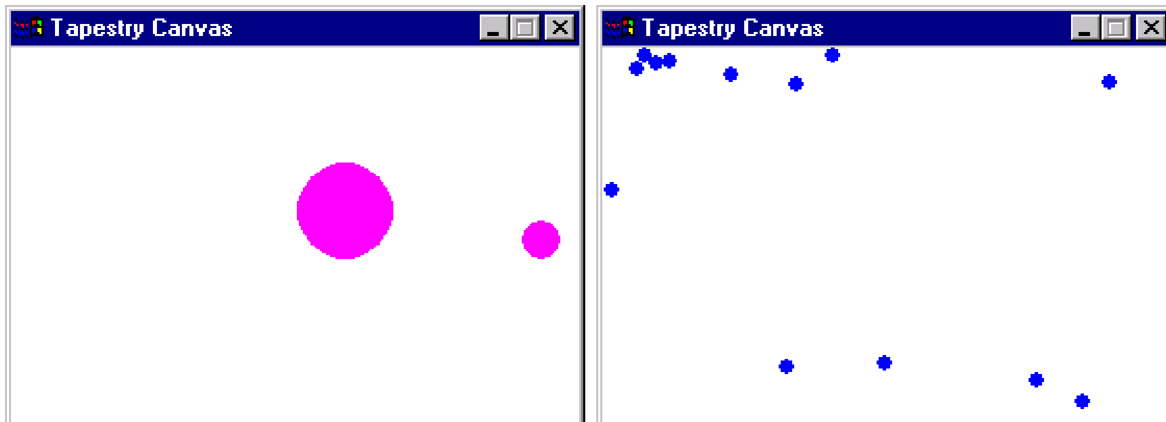

Figure H.11 Molecules bouncing in *bouncefun.cpp*

Collisions are detected using the method `Shape::overlaps` which is implemented in the abstract class `Shape` using bounding boxes. When a `MoleBouncer` object detects a collision, it removes itself and creates a new object that is placed initially in the upper left corner of the canvas. Mouse clicks create larger, magenta objects, collisions create smaller, blue objects.

If no collision is detected, an object updates itself as a normal bouncer by calling the parent method `Bouncer::update`. It's essential that objects avoid checking themselves for collisions since every object will collide with itself. All `Canvas` objects support the `makeIterator` method used in *molebouncer.cpp*. The method `Current` in the class `CanvasIterator` (see *canvasiterator.h*) returns a reference to a shape, so the shapes can be modified via the iterator.

***Mover* Basics.** Shapes can be wrapped (or decorated) by the `Mover` class and controlled by client programs rather than by the objects themselves, as when `Bouncer` objects are used. Client programs must call `moveTo` or `moveBy` and then explicitly ask an `AnimatedCanvas` to redraw its shapes. This differs from how `Bouncer` objects are moved, since bouncers are typically part of a canvas cycling through its shapes using `run` or `runUntilEscape`. In contrast, when `Mover` objects are used client programs usually call `AnimatedCanvas::run(1)` to redraw all shapes once.

```
void Mover::moveTo(const Point& p)
// post: mover is at location p
{
   myShape->setLocation(p);
}
```

May 28, 1999 09:20 owltex Sheet number 33 Page number 809 magenta black


```
void Mover::moveBy(double dx, double dy)
// post: mover moved by dx,dy from current point
{
   Point p = myShape->getLocation();
   p.x += dx;
   p.y += dy;
   myShape->setLocation(p);
}
```

H.3.8 Specifying Color with Class CanvasColor

.

The class CanvasColor supplies over 130 named constants for identifying and using colors. Each constant is specified using *RGB* values, supplying a value between 0 and 255 for red, green, and blue that contribute to the color. See the header file *canvascolor.h* for details and note that the CanvasColor class encapsulates the colors used by the CMU graphic engine that runs underneath TOOGL. Part of the header file is shown below.

```
class CanvasColor
{
  public:
    CanvasColor(unsigned char red = 0, unsigned char green = 0,
                unsigned char blue = 0)
      : myRed(red), myGreen(green), myBlue(blue)
    { }
    // see canvascolor.h for details, note all data is public

    string tostring( ) const;

    unsigned char myRed;
    unsigned char myGreen;
    unsigned char myBlue;

    static const color SNOW;
    static const color GHOSTWHITE;
    static const color WHITESMOKE;
    static const color GAINSBORO;
    static const color FLORALWHITE;
    static const color OLDLACE;
    // about one hundred more constants
    ..
};
```

H.3.9 The class Key in *key.h*

The class Key encapsulates key presses when used with TOOGL. Part of the class declaration is shown below, the entire class is accessible in *key.h*. As the declaration shows, a Key object responds to a wide variety of query/predicate methods for determining what key it represents.

```
class Key
{
  public:
    enum Kind{ascii, escape, function, arrow, none};

    Key();
    Key(char ch);
    Key(char ch, Kind k);

    char aschar()     const;   // the key as a character
    bool isnothing()  const;
    bool isasc()      const;
    bool isfunction() const;
    bool isarrow()    const;
    bool isescape()   const;
    bool iscontrol()  const;
    bool isleftarrow()    const;
    bool isrightarrow()   const;
    bool isuparrow()      const;
    bool isdownarrow()    const;
    bool ishome()         const;
    bool isend()          const;
    // more
};
```

Bibliography

[AA85] Donald J. Albers and G.L. Alexanderson. *Mathematical People*. Birkhäuser, 1985.

[ACM87] ACM. *Turing Award Lectures: The First Twenty Years 1966-1985*. ACM Press, 1987.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretion of Computer Programs*. MIT Press, McGraw Hill Book Company, second edition 1996.

[Asp90] William Aspray. *Computing Before Computers*. Iowa State University Press, 1990.

[Aus98] Matthew H. Austern *Generic Programming and the STL* Addison-Wesley, 1998.

[Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[Ben88] Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.

[Boo94] Grady Booch. *Object Oriented Design and Alnaysis with Applications*. Benjamin Cummings, second edition, 1994.

[BRE71] I. Barrodale, F.D. Roberts, and B.L. Ehle. *Elementary Computer Applications in Science Engineering and Business*. John Wiley & Sons Inc., 1971.

[Coo87] Doug Cooper. *Condensed Pascal*. W.W. Norton, 1987.

[DR90] Nachum Dershowitz and Edward M. Reingold. Calendrical calculations. *Software-Practice and Experience*, 20(9):899–928, September 1990.

[(ed91] Allen B. Tucker (ed.). *Computing Curricula 1991 Report of the ACM/IEEE-CS Joint Curriculum Task Force*. ACM Press, 1991.

[EL94] Susan Epstein and Joanne Luciano, editors. *Grace Hopper Celebration of Women in Computing*. Computing Research Association, 1994. HopperBook@cra.org.

[Emm93] Michele Emmer, editor. *The Visual Mind: Art and Mathematics*. MIT Press, 1993.

[Gö95] Denise W. Gürer. Pioneering women in computer science. *Communications of the ACM*, 38(1):45–54, January 1995.

[Gar95] Simson Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, 1995.

[GHJ95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Programming* Addison-Wesley, 1995

[Gol93] Herman H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, 1993.

[Gri74] David Gries. On structured programming - a reply to smoliar. *Communications of the ACM*, 17(11):655–657, 1974.

[Har92] David Harel. *Algorithmics, The Spirit of Computing*. Addison-Wesley, second edition, 1992.

[Hoa89] C.A.R. Hoare. *Essays in Computing Science*. Prentice-Hall, 1989. (editor) C.B. Jones.

[Hod83] Andrew Hodges. *Alan Turing: The Enigma*. Simon & Schuster, 1983.

[Hor92] John Horgan. Claude e. shannon. *IEEE Spectrum*, April 1992.

[JW89] William Strunk Jr. and E.B. White. *The Elements of Style*. MacMillan Publishing Co., third edition, 1989.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, third edition, 1997.

[Knu98a] Donald E. Knuth. *The Art of Computer Programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, third edition, 1998.

[Knu98b] Donald E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, third edition 1998.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[KR96] Samuel N. Kamin and Edward M. Reingold. *Programming with class: A C++ Introduction to Computer Science*. McGraw-Hill, 1996.

[Mac92] Norman Macrae. *John von Neumann*. Pantheon Books, 1992.

[McC79] Pamela McCorduck. *Machines Who Think*. W.H. Freeman and Company, 1979.

[McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993.

[Neu95] Peter G. Neumann. *Computer Related Risks*. Addison Wesley, 1995.

[Pat96] Richard E. Pattis. *Get A-Life: Advice for the Beginning C++ Object-Oriented Programmer*. Turing TarPit Press, 1996.

[Per87] Alan Perlis. The synthesis of algorithmic systems. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[RDC93] Edward M. Reingold, Nachum Dershowitz, and Stewart M. Clamen. Calendrical calculations, ii: Three historical calendars. *Software-Practice and Experience*, 23(4):383–404, April 1993.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Rob94] Eric S. Roberts. Loop exits and structured programming: Reopening the debate, 1994. submitted for publication.

[Sla87] Robert Slater. *Portraits in Silicon*. MIT Press, 1987.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1987.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[Mey96] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[Wei94] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin Cummings, 1994.

[Wil56] M.V. Wilkes. *Automatic Digital Computers*. John Wiley & Sons, Inc., 1956.

[Wil87] Maurice V. Wilkes. Computers then and now. In *ACM Turing Award Lectures: The First Twenty Years*, pages 197–205. ACM Press, 1987.

[Wil95] Maurice V. Wilkes. *Computing Perspectives*. Morgan Kaufmann, 1995.

[Wir87] Niklaus Wirth. From programming language design to compiler construction. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.