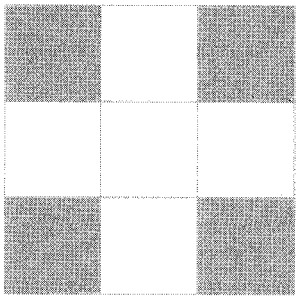

Fundamentals of Computer Graphics

Peter Shirley
School of Computing
University of Utah

with

Michael Ashikhmin
Michael Gleicher
Stephen R. Marschner
Erik Reinhard
Kelvin Sung
William B. Thompson
Peter Willemsen

Kelvin Sung
CSS, UofB
Aug 2005

A K Peters
Wellesley, Massachusetts

Editorial, Sales, and Customer Service Office

A K Peters, Ltd.
888 Worcester Street, Suite 230
Wellesley, MA 02482
www.akpeters.com

Library of Congress Cataloging-in-Publication Data

Shirley, P. (Peter), 1963-
Fundamentals of computer graphics / Peter Shirley ; with Michael Ashikhmin . . . [et. al.].--2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-56881-269-8
1. Computer graphics. I. Ashikhmin, Michael. II. Title.

T385.S434 2005
006.6'96--dc22

2005048904

Printed in the United States of America
09 08 07 06 05 10 9 8 7 6 5 4 3 2 1

Kelvin Sung

18

Building Interactive Graphics Applications

While most of the other chapters in this book discuss the fundamental algorithms in the field of computer graphics, this chapter treats the integration of these algorithms into applications. This is an important topic since the knowledge of fundamental graphics algorithms does not always easily lead to an understanding of the best practices in implementing these algorithms in real applications.

We start with a simple example: a program that allows the user to simulate the shooting of a ball (under the influence of gravity). The user can specify initial velocity, create balls of different sizes, shoot the ball, and examine the parabolic free fall of the ball. Some fundamental concepts we will need include mesh structure for the representation of the ball (sphere); texture mapping, lighting, and shading for the aesthetic appearance of the ball; transformations for the trajectories of the ball; and rasterization techniques for the generation of the images of the balls.

To implement the simple ball shooting program, one also needs knowledge of

- Graphical user interface (GUI) systems for efficient and effective user interaction;
- Software architecture and design patterns for crafting an implementation framework that is easy to maintain and expand;
- Application program interfaces (APIs) for choosing the appropriate support and avoiding a massive amount of unnecessary coding.

To gain an appreciation for these three important aspects of building the application, we will complete the following steps:

- analyze interactive applications;
- understand different programming models and recognize important functional components in these models;
- define the interaction of the components;
- design solution frameworks for integrating the components; and
- demonstrate example implementations based on different sets of existing APIs.

We will use the ball shooting program as our example and begin by refining the detailed specifications. For clarity, we avoid graphics-specific complexities in 3D space and confine our example to 2D space. Obviously, our simple program is neither sophisticated nor representative of real applications. However, with slightly refined specifications, this example contains all the essential components and behavioral characteristics of more complex real-world interactive systems.

We will continue to build complexity into our simple example, adding new concepts until we arrive at a software architecture framework that is suitable for building general interactive graphics applications. We will examine the validity of our results and discuss how the lessons learned from this simple example can be applied to other familiar real-world applications (e.g., PowerPoint, Maya, etc.).

18.1 The Ball Shooting Program

Our simple program has the following elements and behaviors:

- **The balls (objects):** The user can left-mouse-button-click and drag-out a new ball (circle) anywhere on the screen (see Figure 18.1). Dragging-out a ball includes:
  - (A): initial mouse-button-click position defines the center of the circle;
  - (B): mouse button down and moving the mouse is the dragging action;
  - (C): current mouse position while dragging allows us to define the radius and the initial velocity. The radius R (in pixel units) is the distance to the center defined in (A). The vector from the current position to the center is the initial velocity V (in units of pixel per second).

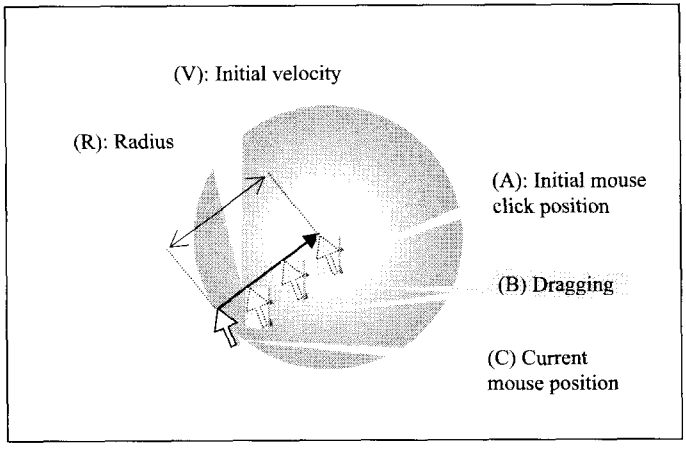

Figure 18.1. Dragging out a ball.

Once created, the ball will begin traveling with the defined initial velocity.

- **HeroBall (Hero/active object):** The user can also right-mouse-button-click to select a ball to be the current HeroBall. The HeroBall's velocity can be controlled by the slider bars (discussed below) where its velocity is displayed. (A newly created ball is by default the current HeroBall.) A right-mouse-button-click on unoccupied space indicates that no current HeroBall exists.

- **Velocity slider bars (GUI elements):** The user can monitor and control two slider bars (x- and y-directions with magnitudes) to change the velocity of the HeroBall. When there is no HeroBall, the slider bar values are undefined.

- **The simulation:**

  - **Ball traveling/collisions (object intrinsic behaviors):** A ball knows how to travel based on its current velocity and one ball can potentially collide with another. For simplicity, we will assume all balls have identical mass and all collisions are perfectly elastic.
  - **Gravity (external effects on objects):** The velocity of a ball is constantly changing due to the defined gravitational force.
  - **Status bar (application state echo):** The user can monitor the application state by examining the information in the status bar. In our application, the number of balls currently on the screen is updated in the status bar.

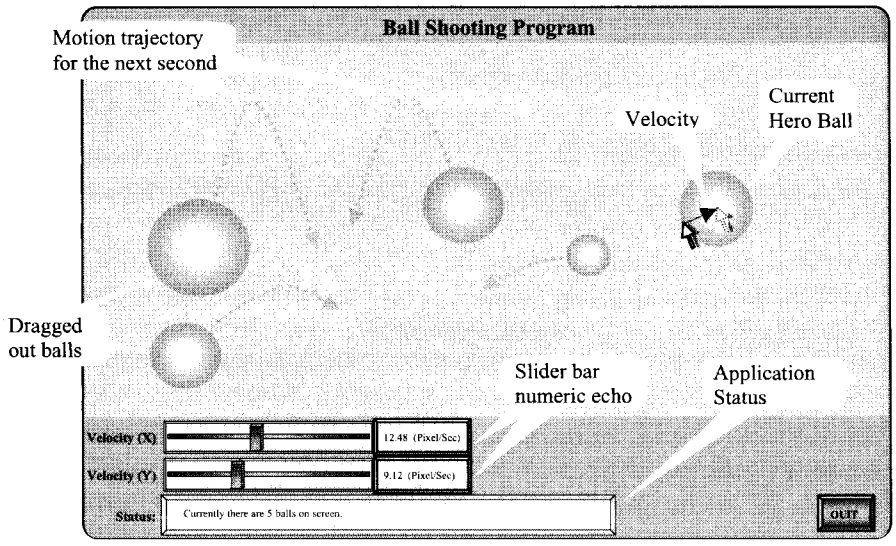

Figure 18.2. The Ball Shooting program.

Our application starts with an empty screen. The user clicks and drags to create new balls with different radii and velocities. Once a ball travels off of the screen, it is removed. To avoid unnecessary details, we do not include the drawing of the motion trajectories or the velocity vector in our solutions. Notice that a slider bar communicates its current state to the user in two ways: the position of the slider knob and the numeric echo (see Figure 18.2).

We have now described the behavior of a simple interactive graphics application. In the rest of this chapter, we will learn the concepts that support the implementation of this type of application.

18.2 Programming Models

For many of us, when we were first introduced to computer programming, we learned that the program should always start and end with the *main*() function—when the *main*() function returns, all the work must have been completed and the program terminates. Since the overall control remains internal to the *main*() function during the entire life time of the program, the type of model for this approach to solving problems is called an *internal control model*, or *control-driven programming*. As we will see, an alternative paradigm, *event-driven programming* or an *external control model* approach, is the more appropriate way to design solutions to interactive programs.

In this section, we will first formulate a solution to the 2D ball shooting program based on the, perhaps more familiar, control-driven programming model. We will then analyze the solution, identify shortcomings, and describe the motivation for the external control model or event-driven programming approach.

The pseudocode which follows is C++/Java-like. We assume typical functionality from the operating System (*OperatingSystem::*) and from a Graphical User Interface API (*GUISystem::*). The purpose of the pseudocode is to assist us in analyzing the foundation control structure (i.e., if/while/case) of the solution. For this reason, the details of application- and graphics-specific operations are intentionally glossed over. For example, the details of how to *UpdateSimulation*() is purposely omitted.

18.2.1 Control-Driven Programming

The main advantage of control-driven programming is that it is fairly straightforward to translate a verbal description of a solution to a program control structure. In this case, we verbalize our solution as follows:

> while the user does not want to quit (A);
>
> parse and execute the user's command (B);
>
> update the velocities and positions of the balls (C);
>
> then draw all the balls (D);
>
> and finally before we poll the user for another command,
>
> tell the user what is going on by echoing current application state to the status bar (E)

```
(A): As long as user is not ready to quit
(B): Parse the user command
(C): periodically update positions and velocities of the balls
(D): Draw all balls to the computer screen
(E): Sets status bar with number of balls

while user command is not quit
    parse and excute user's command
    if (OperatingSystem::SufficientClockTimeHasElapesd)
      UpdateSimulation()        // update the positions and velocities
                                // of the all the balls (in AllWorldBalls set)
    DrawBalls(AllWorldBalls)    // all the balls in AllWorldBalls set
    EchoToStatusBar()           // Sets status bar: number of balls on screen
```

Figure 18.3. Programming structure from a verbalized solution.

Figure 18.3 shows a translation from this verbal solution into a simple programming structure. We introduce the set of *AllWorldBalls* to represent all the balls that are currently on the computer screen. The only other difference between the pseudocode in Figure 18.3 and our verbalized solution is in the added elapsed time check in Step (C): *SufficientClockTimeHasElapsed*. (Recall that the velocities are defined in pixels per second.) To support proper pixel displacements, we must know real elapsed time between updates.

As we add additional details to parse and execute the user's commands (B), the solution must be expanded. The revised solution in Figure 18.4 shows the details of a central parsing switch statement (B) and the support for all three commands a user can issue: defining a new HeroBall (B1); selecting a HeroBall (B2); and adjusting current HeroBall velocity with the slider bars (B3). Undefined user actions (e.g., mouse movement with no button pressed) are simply ignored (B4).

Notice that HeroBall creation (B1) involves three user actions: mouse down (B1), followed by mouse drag (B1-1), and finally mouse up (B1-2). The parsing of this operation is performed in multiple consecutive passes through the outer while-loop (A): the first time through, we create the new HeroBall (B1); in the subsequent passes, we perform the actual *dragging* operation (B1-1). We assume that mouse drag (B1-1) will never be invoked without mouse button down (B1) action, and thus the HeroBall is always defined during the dragging operation.

The *LeftMouseButtonUp* action (B1-2) is an implicit action not defined in the original specification. In our implementation, we choose this implicit action to activate the insertion of the new HeroBall into the AllWorldBalls set. In this way the HeroBall is not a member of the AllWorldBalls set until after the user has completed the dragging operation. This delay ensures that the HeroBall's velocity and position will not be affected when the *UpdateSimulation*() procedure updates all the balls in AllWorldBalls set (C). This means a user can take the time to drag out a new HeroBall without worrying that the ball will free fall before the release of the mouse button. The simple amendment in the drawing operation (D1) ensures a proper drawing of the new HeroBall before it is inserted into the AllWorldBalls set.

When we examine this solution in the context of supporting user interaction, we have to concern ourselves with efficiency issues as well as the potential for increased complexity.

Efficiency Concerns. Typically a user interacts with an application in bursts of activity—continuous actions followed by periods of idling. This can be explained by the fact that, as users, we typically perform some tasks in the application and then spend time examining the results. For example, when working with a word

```
        main()  {
(A):
            while ( GUISystem::UserAction != Quit )  {
(B):
                switch (GUISystem::UserAction)  {

(B1): Define new
Hero Ball           // Begins creating a new Hero Ball
                    case GUISystem::LeftMouseButtonDown:
                        HeroBall = CreateHeroBall()      // hero not in AllWorldBalls set
                        DefiningNewHeroBall = true
    (B1-1) Support
    for drag actions
                    // Drags out the new Hero Ball
                    case GUISystem::LeftMouseButtonDrag:
                        RefineRadiusAndVelocityOfHeroBall()
    (B1-2) Implicit     SetSliderBarsWithHeroBallVelocity()
    Action
                    // Finishes creating the new Hero Ball
                    case GUISystem::LeftMouseButtonUp:
                        InsertHeroBallToAllWorldBalls()
                        DefiningNewHeroBall = false

(B2): Select
current Hero Ball   // Selects a current hero ball
                    case GUISystem::RightMouseButtonDown:
                        HeroBall = SelectHeroBallBasedOnCurrentMouseXY()
                        if (HeroBall != null)
                            SetSliderBarsWithHeroBallVelocity()
(B3): Set Hero
Ball Velocity       // Sets hero velocity with slider bars
                    case GUISystem::SliderBarChange:
                        if (HeroBall != null)
(B4): Undefined             SetHeroBallVelocityWithSliderBarValues()
actions are ignored
                    // Ignores all other user actions e.g. Mouse Move with no buttons, etc
                    default:
                }  // end of switch(userAction)

(C):                // Move balls by velocities under gravity and remove off-screen ones
                    if ( OperatingSystem::SufficientClockTimeHasElapesd)
                        UpdateSimulation()

(D):                DrawBalls(AllWorldBalls)
                        // Draw the new Hero Ball that is currently being defined
(D1): Draw the      if (DefiningNewHeroBall)
new Hero Ball           DrawBalls(HeroBall)

(E):                EchoToStatusBar()    // Sets Status Bar with number of balls currently on screen

            }  // end of while(UserAction != Quit)
        }  // end of main() function. Program terminates.
```

Figure 18.4. Programming solution based on the control-driven programming model.

processor, our typical work pattern consists of bursts of typing/editing followed by periods of reading (with no input action). In our example application, we can expect the user to drag out some circles and then observe the free-falling of the circles. The continuous while-loop polling of user commands in the *main*() function means that when the user is not performing any action, our program will

still be actively running and wasting machine resources. During activity bursts, at the maximum, users are capable of generating hundreds of input actions per second (e.g., mouse-pixel movements per second). If we compare this rate to the typical CPU instruction capacities that are measured at 10^9 per second, the huge discrepancy indicates that, even during activity bursts, the user command-parsing switch statement (B) is spending most of the time in the default case not doing anything.

Complexity Concerns. Notice that our *entire solution* is in the *main*() function. This means that all relevant user actions must be parsed and handled by the user command-parsing switch statement (B). In a modern multi-program shared window environment, many actions performed by users are actually non-application specific. For example, if a user performs a left mouse button click or drag in the drawing area of the program window, our application should react by dragging out a new HeroBall. However, if the user performs the same actions in the title area of the program window, our application should forward these actions to the GUI/Operating/Window system and commence the coordination of moving the entire program window. As experienced users in window environments, we understand that there are numerous such non-application specific operations, and we expect all applications to honor these actions (e.g., iconize, re-size, raise or lower a window, etc.). Following the solution given in Figure 18.4, for every user action that we want to honor, we must include a matching supporting case in the parsing switch statement (B). This requirement quickly increases the complexity of our solution and becomes a burden to implementing any interactive applications.

An efficient GUI system should remain idle by default (not taking up machine resources) and only become active in the presence of interesting activities (e.g., user input actions). Furthermore, to integrate interactive applications in sophisticated multi-programming window environments, it is important that the supporting GUI system automatically takes care of mundane and standard user actions.

18.2.2 Event-Driven Programming

Event-driven programming remedies the efficiency and complexity concerns with a default *MainEventLoop*() function defined in the GUI system. For event-driven programs, the *MainEventLoop*() replaces the *main*() function, because all programs start and end in this function. Just as in the case of the *main*() function for control-driven programming, when the *MainEventLoop*() function re-

```
UISystem::MainEventLoop() {

        SystemInitialization()
(A): For application        // For initialization of application state and
initialization              // registration of event service routines

        loop forever {

(B): Continuous
outer loop                  WaitFor ( GUISystem::NextEvent)
                            // Program will stop and wait for the next event

                            switch (GUISystem::NextEvent) {
(C): Stop and wait
for next event                  case GUISystem::LeftMouseButtonDown:
                                    if (user application registered for this event)
                                        Execute user defined service routine.
(D): Central parsing                else
switch statement                        Execute default UISystem routine.
                                ⋮
                                case GUISystem::Iconize:
                                    if (user application registered for this event)
                                        Execute user defined service routine.
Every possible                      else
event                                   GUISystem::DefaultIconizeBehavior()
                                ⋮
                            } // end of switch(GUISystem::NextEvent)
                        } // end of loop forever
                    } // end of GUISystem::MainEventLoop() function. Program terminates.
```

Figure 18.5. The default *MainEventLoop* function.

turns, all work should have been completed, and the program terminates. The *MainEventLoop*() function defines the central control structure for all event-driven programming solutions and typically cannot be changed by a user application. In this way, the overall control of an application is actually external to the user's program code. For this reason, event-driven programming is also referred to as the external control model.

Figure 18.5 depicts a typical *MainEventLoop*() implementation. In this case, our program is the user application that is based on the *MainEventLoop*() function. Structurally, the *MainEventLoop*() is very similar to the *main*() function of Figure 18.4: with a continuous loop (B) containing a central parsing switch statement (D). The important differences between the two functions include:

- (A) SystemInitialization(): Recall that event-driven programs start and end in the *MainEventLoop*() function. *SystemInitialization*() is a mechanism defined to invoke the user program from within the *MainEventLoop*(). It is expected that user programs implement *SystemInitialization*() to initialize the application state and to register event service routines (refer to the discussion in (D)).

- (B) Continuous outer loop: Since this is a general control structure to be shared by all event-driven programs, there is no way to determine the termination condition. User program are expected to override appropriate event service routines and terminate the program from within the service routine.
- (C) Stop and wait: Instead of actively polling the user for actions (wasting machine resources), the *MainEventLoop*() typically stops the entire application process and waits for asynchronous operating system calls to re-activate the application process in the presence of relevant user actions.
- (D) Events and central parsing switch statement: Included in this statement are all possible actions/events (cases) that a user can perform. Associated with each event (case) is a default behavior and a toggle that allows user applications to override the default behavior. During *SystemInitialization*(), the user application can register an alternate service routine for an event by toggling the override.

To develop an event-driven solution, our program must first register event service routines with the GUI system. After that, our entire program solution is based on waiting and servicing user events. While control-driven programming solutions are based on an algorithmic organization of control structures in the *main*() function, an event-driven programming solution is based on the specification of events that cause changes to a defined application state. This is a different paradigm for designing programming solutions. The key difference here is that, as programmers, we have no explicit control over the algorithmic organization of the events: over which, when, or how often an event should occur.

The program in Figure 18.6 implements the left mouse button operations for our ball shooting program. We see that during system initialization (A), the program defines an appropriate application state (A1) and registers left mouse button (LMB) down/drag/up events (A2). The corresponding event service routines (D1, D2, and D3) are also defined. At the end of each event service routine, we redraw all the balls to ensure that the user can see an up-to-date display at all times. Notice the absence of any control structure organizing the initialization and service routines. Recall that this is an event-driven program: the overall control structure is defined in the MainEventLoop which is external to our solution.

Figure 18.7 shows how our program from Figure 18.6 is linked with the predefined *MainEventLoop*() from the GUI system. The *MainEventLoop*() calls the *SystemInitialization*() function defined in our solution (A). As described, after the initialization, our entire program is essentially the three event service routines (D1, D2, and D3). However, we have no control over the invocation of these routines. Instead, a user performs actions that trigger events which drive

```
(A) System Initialization:
    (A1): Define Application State:
        AllWorldBalls: A set of defined Balls, initialze to empty
        HeroBall: current active ball, initialize to null

    (A2): Register Event Service Routines
        Register for: Left Mouse Button Down Event
        Register for: Left Mouse Button Drag Event
        Register for: Left Mouse Button Up Event
                        // We care about these events, inform us if these events happen

(D) Events Services:
    (D1): Left Mouse Button Down// service routine for this event
        HeroBall = Create a new ball at current mouse position
        DrawAllBalls(AllWorldBalls, HeroBall) // Draw all balls (including HeroBall)

    (D2): Left Mouse Button Drag // service routine for this event
        RefineRadiusAndVelocityOfHeroBall()
        DrawAllBalls(AllWorldBalls, HeroBall) // Draw all balls (inlucding HeroBall)

    (D3): Left Mouse Button Up  // service routine for this event
        InsertHeroBallToAllWorldBalls()
        DrawAllBalls(AllWorldBalls, null)       // Draw all balls
```

Figure 18.6. A simple event-driven program specification.

these routines. These routines in turn change the application state. In this way, an event-driven programming solution is based on specification of events (LMB events) that cause changes to a defined application state (AllWorldBalls and HeroBall). Since the user command parsing switch statement (D in Figure 18.7) in the *MainEventLoop*() contains a case and the corresponding default behavior for every possible user actions, without any added complexity, our solution honors the non-application specific actions in the environment (e.g., iconize, moving, etc).

In the context of event-driven programming, an event can be perceived as an asynchronous notification that something interesting has happened. The messenger for the *notification* is the underlying GUI system. The mechanism for receiving an event is via overriding the corresponding event service routine.

For these reasons, when discussing event-driven programming, there is always a supporting GUI system. This GUI system is generally referred to as the Graphics User Interface (GUI) Application Programming Interface (API). Examples of GUI APIs include: *Java Swing Library*, *OpenGL Utility ToolKit (GLUT)*, *The Fast Light ToolKit (FLTK)*, *Microsoft Foundation Classes (MFC)*, etc.

From the above discussion, we see that the registration for services of appropriate events is the core of designing and developing solutions for event-driven programs. Before we begin developing a complete solution for our ball shooting program, let us spend some time understanding *events*.

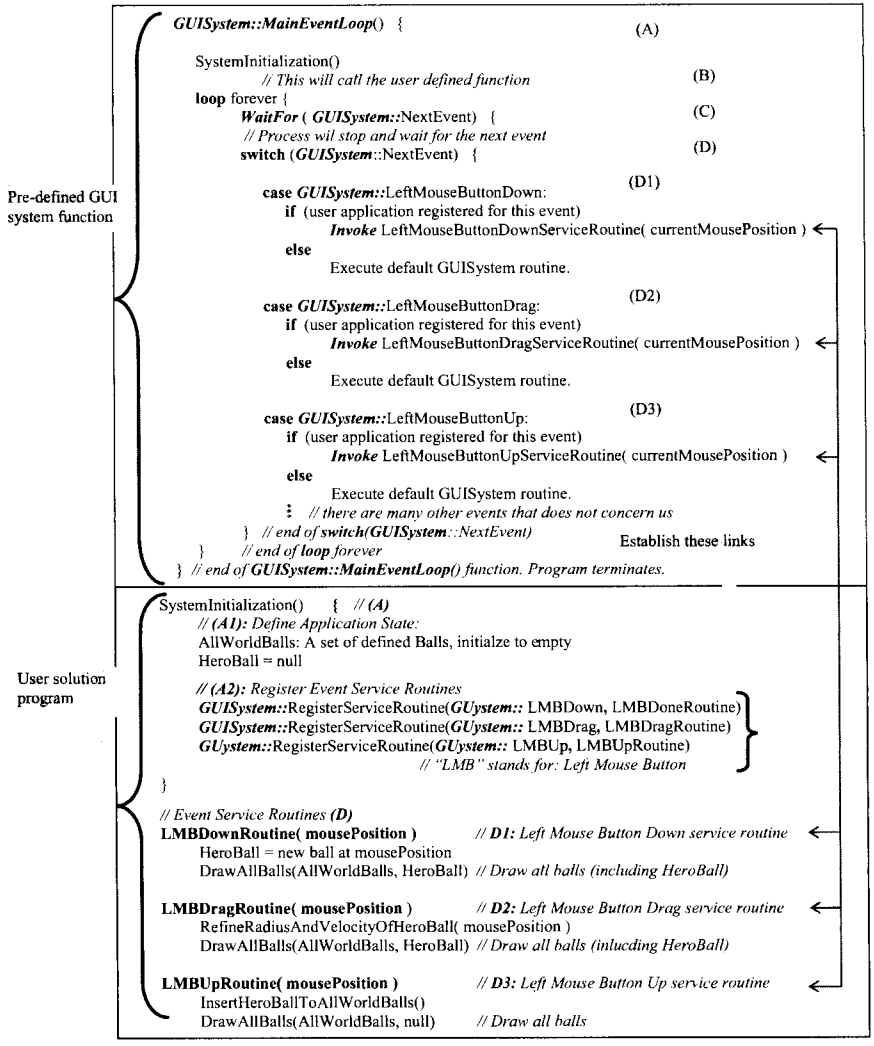

Figure 18.7. Linking MainEventLoop with our solution.

Graphical User Interface (GUI) Events

In general, an application may receive events generated by the user, the application itself, or by the GUI system. In this section, we describe each of these event sources and discuss the application's role in servicing these events.

S1: The User. These are events triggered by the actions a user performs on the input devices. Notice that input devices include actual hardware devices (e.g.,

mouse, keyboard, etc.) and/or software-simulated GUI elements (e.g., slider bars, combo boxes, etc.). Typically, a user performs actions for two very different reasons:

- **S1a: Application specific.** These are input actions that are part of the application. Clicking and dragging in the application screen area to create a HeroBall is an example of an action performed on a hardware input device. Changing the slider bars to control the HeroBall's velocity is an example of an action performed on a software-simulated GUI element. Both of these actions and the resulting events are application specific; the application (our program) is solely responsible for servicing these events.
- **S1b: General.** These are input actions defined by the operating environment. For example, a user clicks and drags in the window title-bar area expecting to move the entire application window. The servicing of these types of events requires collaboration between our application and the GUI system. We will discuss the servicing of these types of events in more detail when explaining events that originate from the GUI system in S3c.

Notice that the meaning of a user's action is *context sensitive*. It depends on where the action is performed: click and drag in the application screen area vs. slider bar vs. application window title-bar area. In any case, the underlying GUI system is responsible for parsing the context and determining which application element receives a particular event.

S2: The Application. These are events defined by the application, typically depending on some run-time conditions. During run time, if and when the condition is favorable, the supporting GUI system triggers the event and conveys the favorable conditions to the application. A straightforward example is a periodic alarm. Modern GUI systems typically allow an application to define (sometimes multiple) *timer events* . Once defined, the GUI system will trigger an event to wake up the application when the timer expires. As we will see, this timer event is essential for supporting real-time simulations. Since the application (our program) requested the generation of these types of events, our program is solely responsible for serving them. The important distinction between application-defined and user-generated events is that application-defined events can be *spontaneous*: when properly defined, even when the user is not doing anything, these types of events may trigger.

S3: The GUI System. These are events that originate from within the GUI system in order to convey state information to the application. There are typically

three reasons for these events:

- **S3a: Internal GUI states change.** These are events signaling an internal state change of the GUI system. For example, the GUI system typically generates an event before the creation of the application's main window. This provides an opportunity for the application to perform the corresponding initialization. In some GUI systems (e.g., MFC) the *SystemInitialization*() functionality is accomplished with these types of events: user applications are expected to override the appropriate windows' creation event and initialize the application state. Modern, general purpose commercial GUI systems typically define a large number of events signaling detailed state changes in anticipation of supporting different types of applications and requirements. For example, for the creation of the application's main window, the GUI system may define events for the following states:
  - before resource allocation;
  - after resource allocation but before initialization;
  - after initialization but before initial drawing, etc.

  A GUI system usually defines meaningful default behaviors for such events. To program an effective application based on a GUI system, one must understand the different groups of events and only service the appropriate selections.

- **S3b: External environment requests attention.** These are events indicating that there are changes in the operating environment that potentially require application attention. For example, a user has moved another application window to cover a portion of our application window, or a user has minimized our application window. The GUI system and the window environment typically have appropriate service routines for these types of events. An application would only choose to service these events when special actions must be performed. For example, in a real-time simulation program, the application may choose to suspend the simulation if the application window is minimized. In this situation, an application must service the minimized and the maximized events.

- **S3c: External environment requests application collaboration.** These are typically events requesting the application's collaboration to complete the service of *general user actions* (please refer to S1b). For example, if a user click-drags the application window's title bar, the GUI system reacts by letting the user "drag" the entire application window. This "drag"

operation is implemented by continuously erasing and redrawing the entire application window at the current mouse pointer position on the computer display. The GUI system has full knowledge of the appearance of the application window (e.g., the window frames, the menus, etc.), but it has no knowledge of the application window content (e.g., how many free falling balls traveling at what velocity, etc.). In this case, the GUI system redraws the application window frame and generates a Redraw/Paint event for the application, requesting assistance in completing the service of the user's "drag" operation. As an application in a shared window environment, our application is expected to honor and service these types of events. The most common events in this category include: Redraw/Paint and Resize. Redraw/Paint is the single most important event an application must service, because it supports the most common operations a user may perform in a shared window environment. Resize is also an important event to which the application must respond because the application is in charge of GUI element placement policy (e.g., if window size is increased, how should the GUI elements be placed in the larger window).

18.2.3 The Event-Driven Ball Shooting Program

In Section 18.2.1, we started a control-driven programming solution to the ball shooting program based on verbalizing the conditions (controls) under which the appropriate actions should be taken:

while favorable condition, parse the input ...

As we have seen, with appropriate modifications, we were able to detail the control structures for our solution.

From the discussion in Section 18.2.2, we see that to design an event-driven programming solution we must

1. define the application state;
2. describe how user actions change the application state;
3. map the user actions to events that the GUI system supports; and
4. override corresponding event service routines to implement user actions.

The specification in Section 18.1 detailed the behaviors of our ball shooting program. The description is based on actions performed on familiar input devices

(e.g., slider bars and mouse) that change the appearance on the display screen. Thus, the specification from Section 18.1 describes items (2) and (3) from the above list without explicitly defining what the application state is. Our job in designing a solution is to derive the implicitly defined application state and design the appropriate service routines.

Figure 18.8 presents our event-driven programming solution. As expected, the application state (A1) is defined in *SystemInitialization*(). The AllWorldBalls set and HeroBall can be derived from the specification in Section 18.1. The *DefiningNewHeroBall* flag is a transient (temporary) application state designed to support user actions across multiple events (click-and-drag). Using *transient* application states is a common approach to support consecutive inter-related events.

Figure 18.8 shows the registration of three types of service routines (A2):

- user-generated application specific events (S1a);
- an application defined event (S2);
- a GUI system-generated event requesting collaboration (S3c).

The timer event definition (A2S2) sets up a periodic alarm for the application to update the simulation of the free falling balls. The service routines of the user-generated application specific events (D1-D5) are remarkably similar to the corresponding case statements in the control-driven solution presented in Figure 18.4 (B1-B3). It should not be surprising that this is so, because we are implementing the exact same user actions based on the same specification. Line 3 of the *LMBDownRoutine*() (D1L3) demonstrates that, when necessary, our application can request the GUI system to initiate events. In this case, we signal the GUI system that an application redraw is necessary. Notice that event service routines are simply functions in our program. This means, at D1L3 we could also call *RedrawRoutine*() (D7) directly. The difference is that a call to *RedrawRoutine*() will force a redraw immediately while requesting the generation of a redraw event allows the GUI system to optimize the number of redraws. For example, if the user performs a LMB click and starts dragging immediately, with our D1 and D2 implementation, the GUI system can gather the many *GenerateRedrawEvent* requests in a short period of time and only generate one re-draw event. In this way, we can avoid performing more redraws than necessary.

In order to achieve a smooth animation, we should perform about 20–40 updates per second. It follows that the *SimulationUpdateInterval* should be no more than 50 milliseconds so that the *ServiceTimer*() routine can be invoked more than 20 times per second. (Notice that a redraw event is requested at the end of the *ServiceTimer*() routine.) This means, at the very least, our application is guaranteed to receive more than 20 redraw events in one second. For this reason, the

```
SystemInitialization()  {      // (A)
    // (A1): Define Application State                          A1: Application
            AllWorldBalls: A set of defined Balls, initialze to empty   } State
            HeroBall = null
            DefiningNewHeroBall = false

    // (A2): Register Event Service Routines
        // S1a: Application Specific User Events
            GUISystem::RegisterServiceRoutine(GUISystem:: LMBDown, LMBDownRoutine)
            GUISystem::RegisterServiceRoutine(GUISystem:: LMBDrag, LMBDragRoutine)
A2S2: Defines
a Timer Event
            GUISystem::RegisterServiceRoutine(GUISystem:: LMBUp, LMBUpRoutine)
            GUISystem::RegisterServiceRoutine(GUISystem:: RMBDown, RMBDownRoutine)
            GUISystem::RegisterServiceRoutine(GUISystem:: SliderBar, SliderBarRoutine)
        { // S2: Application Define Event
            GUISystem::DefineTimerPeriod(SimulationUpdateInterval)
            GUISystem::RegisterServiceRoutine(GUISystem:: TimerEvent, ServiceTimer)
                                    // Triggers TimerEvent every: SimulationUpdateInterval period
        // S3c: Honor collaboration request from the GUI system
            GUISystem::RegisterServiceRoutine(GUISystem:: RedrawEvent, RedrawRoutine)
}

// Event Service Routines (D)
LMBDownRoutine( mousePosition )    // D1: Left Mouse Button Down service routine
    HeroBall = CreateHeroBall (mousePosition)
    DefiningNewHeroBall = true                          D1L3: Force a
    GUISystem::GenerateRedrawEvent                      Redraw Event

LMBDragRoutine( mousePosition )    // D2: Left Mouse Button Drag service routine
    RefineRadiusAndVelocityOfHeroBall( mousePosition )
    SetSliderBarsWithHeroBallVelocity()
    GUISystem::GenerateRedrawEvent      // Generates a redraw event

LMBUpRoutine( mousePosition )      // D3: Left Mouse Button Up service routine
    InsertHeroBallToAllWorldBalls()
    DefiningNewHeroBall = false

RMBDownRoutine ( mousePosition )   // D4: Right Mouse Button Down service routine
    HeroBall = SelectHeroBallBasedOn (mousePosition )
    if (HeroBall != null)  SetSliderBarsWithHeroBallVelocity()

SliderBarRoutine ( sliderBarValues )   // D5: Slider Bar changes service routine
    if (HeroBall != null)
        SetSliderBarsWithHeroBallVelocity( sliderBarValues )

ServiceTimer ( )                   // D6: Timer expired service routine
    UpdateSimulation( )                 // Move balls by velocities and remove off-screen ones
    EchoToStatusBar( )                  // Sets status bar with number of balls on screen
    GUISystem:: GenerateRedrawEvent     // Generates a redraw event
    if (HeroBall != null)               // Reflect propoer HeroBall velocity
        SetSliderBarsWithHeroBallVelocity( sliderBarValues )

RedrawRoutine ( )                  // D7:Redraw event service routine
    DrawBalls(AllWorldBalls)
    if (DefiningNewHeroBall)
        DrawBalls(HeroBall)             // Draw the new Hero Ball that is being defined
```

Figure 18.8. Programming solution based on the event-driven programming model.

GenerateRedrawEvent requests in D1 and D2 are really not necessary. The servicing of our timer events will guarantee us an up-to-date display screen at all times.

18.2.4 Implementation Notes

The application state of an event-driven program must persist over the entire life time of the program. In terms of implementation, this means that the application state must be defined based on variables that are dynamically allocated during run time and that reside on the heap memory. These are in contrast to local variables that reside on the stack memory and which do not persist over different function invocations.

The mapping of user actions to events in the GUI system often results in *implicit* and/or undefined events. In our ball shooting program, the actions to define a HeroBall involve left mouse button down and drag. When mapping these actions to events in our implementation (in Figure 18.4 and Figure 18.8), we realize that we should also pay attention to the implicit mouse button up event. Another example is the HeroBall selection action: right mouse button down. In this case, right mouse button drag and up events are not serviced by our application, and thus, they are undefined (to our application).

When one user action (e.g., *"drag out the HeroBall"*) is mapped to a group of consecutive events (e.g., mouse button down, then drag, then up) a finite state diagram can usually be derived to help design the solution. Figure 18.9 depicts the finite state diagram for defining the HeroBall in our ball shooting program.

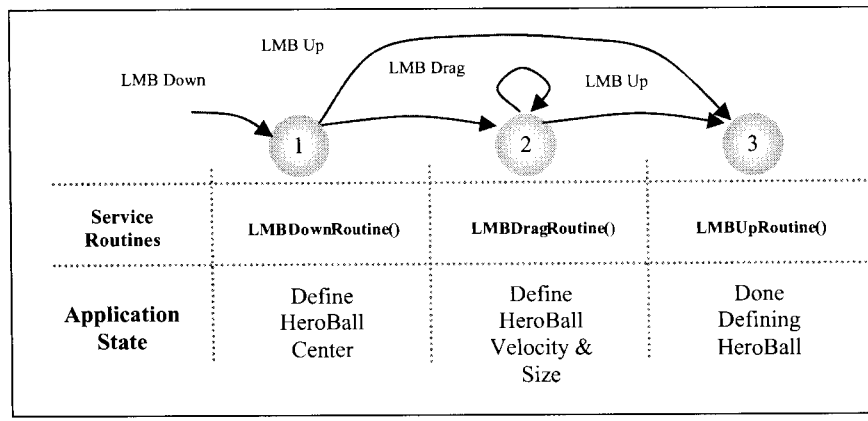

Figure 18.9. State diagram for defining the HeroBall.

The left mouse button down event puts the program into State 1 where, in our solution from Figure 18.8, *LMBDownRoutine*() implements this state and defines the center of the HeroBall, etc. In this case the transition between states is triggered by the mouse events, and we see that it is physically impossible to move from State 2 back to State 1. However, we do need to handle the case where the user action causes a transition from State 1 to State 3 directly (mouse button down and release without any dragging actions). This state diagram helps us analyze possible combinations of state transitions and perform appropriate initializations.

Event-driven applications interface with the user through physical (e.g., mouse clicks) or simulated GUI elements (e.g., quit button, slider bars). An input GUI element (e.g., the quit button) is an artifact (e.g., an icon) for the user to direct changes to the application state, while an output GUI element (e.g., the status bar) is an avenue for the application to present application state information to the user as feedback. For both types of elements, information only flows in one direction—either from the user to the application (input) or from the application to the user (output). When working with GUI elements that serve both input and output purposes, special care is required. For example, after the user selects or defines a HeroBall, the slider bars reflects the velocity of the free falling HeroBall (output), while at any time, the user can manipulate the slider bar to alter the HeroBall velocity (input). In this case, the GUI element's displayed state and the application's internal state are connected. The application must ensure that these two states are consistent. Notice that in the solution shown in Figure 18.4, this state consistency is not maintained. When a user clicks the RMB (B2 in Figure 18.4) to select a HeroBall, the slider bar values are updated properly; however, as the HeroBall free falls under gravity, the slider bar values are not updated. The solution presented in Figure 18.8 fixes this problem by using the *ServiceTimer*() function.

Event service routines are functions defined in our program that cause a *callback* from the MainEventLoop in the presence of relevant events. For this reason, these service routines are also referred to as *callback* functions. The application program registers callback functions with the GUI system by passing the address of the function to the GUI system. This is the registration mechanism implied in Figure 18.7 and Figure 18.8. Simple GUI systems (e.g., GLUT or FLTK) usually support this form of registration mechanism. The advantage of this mechanism is that it is easy to understand, straightforward to program, and often contributes to a small memory footprint in the resulting program. The main disadvantage of this mechanism is the lack of organizational structure for the callback functions.

In commercial GUI systems, there are a large numbers of events with which user applications must deal, and a structured organization of the service routines can assist the programmability of the GUI system. Modern commercial GUI sys-

tems are often implemented based on object-oriented languages (e.g., C++ for MFC, Java for Java Swing). For these systems, many event service registrations are implemented as sub-classes of an appropriate GUI system class, and they override corresponding virtual functions. In this way, the event service routines are organized according to the functionality of GUI elements. The details of different registration mechanisms will be explained in Section 18.4.1 when we describe the implementation details.

Event service routines (or callback functions) are simply functions in our program. However, these functions also serve the important role as the server of external asynchronous events. The following are guidelines one should take into account when implementing event service routines:

1. An event service routine should only service the triggering event and immediately return the control back to the *MainEventLoop*(). This may seem to be a "no-brainer." However, because of our familiarity with control-driven programming, it is often tempting to anticipate/poll subsequent events with a control structure in the service routine. For example, when servicing the left mouse button down event, we know that the mouse drag event will happen next. After allocating and defining the circle center, we have properly initialized data to work with the HeroBall object. It may seem easier to simply include a while loop to poll and service mouse drag events. However, with all the other external events that may happen (e.g., timer event, external redraw events, etc.), this monopolizing of control in one service routine is not only a bad design decision, but also it may cause the program to malfunction.

2. An event service routine should be *stateless*, and individual invocations should be independent. In terms of implementation, this essentially means event service routines should not define local *static* variables that record data from previous invocations. Because we have no control over when, or how often, events are triggered, when these variables are used as data, or conditions for changing application states, it can easily lead to disastrously and unnecessarily complex solutions. We can always define extra state variables in the application state to record temporary state information that must persist over multiple event services. The *DefiningNewHeroBall* flag in Figure 18.8 is one such example.

3. An event service routine should check for invocation conditions regardless of common sense *logical* sequence. For example, although logically, a mouse drag event can never happen unless a mouse down event has already occurred, in reality, a user may depress a mouse button from outside

of our application window and then drag the mouse into our application window. In this case, we will receive a mouse drag event without the corresponding mouse down event. For this reason, the mouse drag service routine should check the *invocation condition* that the proper initialization has indeed happened. Notice in Figure 18.8, we do not include proper invocation condition checking. For example, in the *LMBDragRoutine*(), we do not verify that *LMBDownRotine*() has been invoked (by checking the *DefiningNewHeroBall* flag). In a real system, this may causes the program to malfunction and/or crash.

18.2.5 Summary

In this section we have discussed *programming models* or *strategies for organizing statements of our program.* We have seen that for *interactive* applications, where an application continuously waits and reacts to a user's input actions, organizing the program statements based on designing control structures results in complex and inefficient programs. Existing GUI systems analyze all possible user actions, design control structures to interact with the user, implement default behaviors for all user actions, and provide this functionality in GUI APIs. To develop interactive applications, we take advantage of the existing control structure in the GUI API (i.e., the *MainEventLoop*()) and modify the default behaviors (via event service routines) of user actions. In order to properly collaborate with existing GUI APIs, the strategy for organizing the program statements should be based on specifying user actions that cause changes to the application state.

Now that we understand how to organize the statements of our program, let's examine strategies for organizing functional modules of our solution.

18.3 The Modelview-Controller Architecture

The event-driven ball shooting program presented in Section 18.2.3 and Figure 18.8 addresses programmability and efficiency issues when interacting with a user. In the development of that model, we glossed over many supporting functions (e.g., *UpdateSimulation*()) needed in our solution. In this section, we develop strategies for organizing these functions. Notice that we are not interested in the implementation details of these functions. Instead, we are interested in grouping related functions into components. We then pay attention to how the different *components* collaborate to support the functionality of our application.

In this way, we derive a framework that is suitable for implementing general interactive graphics applications. With a proper framework guiding our design and implementation, we will be better equipped to develop programs that are easier to understand, maintain, modify, and expand.

18.3.1 The Modelview-Controller Framework

Based on our experience developing solutions in Section 18.2, we understand that *interactive graphics applications* can be described as applications that allow users to interactively update their internal states. These applications provide real-time visualization of their internal states (e.g., the free-falling balls) with computer graphics (e.g., drawing circles). The *modelview-controller (MVC)* framework provides a convenient structure for discussing this type of application. In the MVC framework, the *model* is the application state, the *view* is responsible for setting up support for the model to present itself to the user, and the *controller* is responsible for providing the support for the user to interact with the model. Within this framework, our solution from Figure 18.8 is simply the implementation of a controller. In this section, we will develop the understanding of the other two components in the MVC framework and how these components collaborate to support interactive graphics applications.

Figure 18.10 shows the details of a MVC framework to describe the behavior of a typical interactive graphics application. We continue to use the ball shooting program as our example to illustrate the details of the components. The top-right rectangular box is the model, the bottom-right rectangular box is the view, and the rectangular box on the left is the controller component. These three boxes represent program code we, as application developers, must develop. The two dotted rounded boxes represent external graphics and GUI APIs. These are *the external libraries* that we will use as a base for building our system. Examples of popular Graphics APIs include OpenGL, Microsoft Direct-3D (D3D), Java 3D, among others. As mentioned in Section 18.2.2, examples of popular GUI APIs include GLUT, FLTK, MFC, and Java Swing Library.

The model component defines the persistent application state (e.g., AllWorldBalls, HeroBalls, etc.) and implements interface functions for this application state (e.g., *UpdateSimulation*()). Since we are working with a "graphics" application, we expect graphical primitives to be part of the representation for the application state (e.g., CirclePrimitives). This fact is represented in Figure 18.10 by the application state (the ellipse) partially covering the Graphics API box. In the rest of this section, we will use the terms model and persistent application state interchangeably.

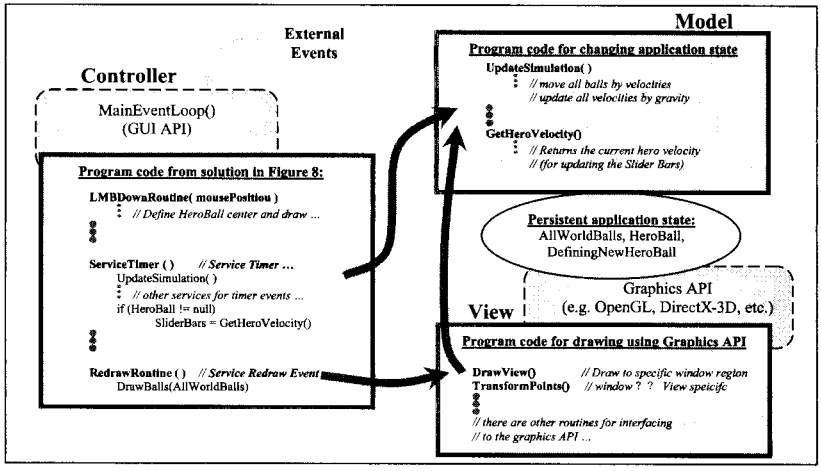

Figure 18.10. Components of an interactive graphics application.

The view component is in charge of *drawing* to the *drawing area* on the application window (e.g., drawing the free falling balls). More specifically, the view component is responsible for initializing the graphics API transformation such that drawing of the model's graphical primitives will appear in the appropriate drawing area. The arrow from the view to the model component signifies that the actual application state redraw must be performed by the model component. Only the model component knows the details of the entire application state (e.g., size and location of the free falling circles) so only the model component can redraw the entire application. The view component is also responsible for transforming user mouse click positions to a coordinate system that the model understands (e.g., mouse button clicks for dragging out the hero ball).

The top left *external events* arrow in Figure 18.10 shows that all external events are handled by the *MainEventLoop*(). The relevant events will be forwarded to the event service routines in the controller component. Since the controller component is responsible for interacting with the user, the design is typically based on event-driven programming techniques. The solution presented in Section 18.2.3 and Figure 18.8 is an example of a controller component implementation. The arrow from the controller to the model indicates that most external events eventually change the model component (e.g., creating a new HeroBall or changing the current HeroBall velocity). The arrow from the controller to the view component indicates that the user input point transformation is handled by the view component. Controllers typically return mouse click positions in the device coordinate with the origin at the top-left corner. In the application model, it is more convenient for us to work with a coordinate system with a lower-left origin.

The view component with its transformation functionality has the knowledge to perform the necessary transformation.

Since the model must understand the transformation set up by the view, it is important that the model and the view components are implemented based on the same Graphics API. However, this sharing of an underlying supporting API does not mean that the model and view are an integrated component. On the contrary, as will be discussed in the following sections, it is advantageous to clearly distinguish between these two components and to establish well-defined interfaces between them.

18.3.2 Applying MVC to the Ball Shooting Program

With the described MVC framework and the understanding of how responsibilities are shared among the components, we can now extend the solution presented in Figure 18.8 and complete the design of the ball shooting program.

The Model

The model is the application state and thus this is the core of our program. When describing approaches to designing an event-driven program in Section 18.2.3, the first two points mentioned were:

1. define the application state, and

2. describe how a user changes this application state.

These two points are the guidelines for designing the model component. In an object-oriented environment, the model component can be implemented as classes, and *state of the application* can be implemented as instance variables, with "*how a user changes this application state*" implemented as methods of the classes.

Figure 18.11 shows that the instance variables representing the state are typically private to the model component. As expected, we have a "very graphical" application state. To properly support this state, we define the CirclePrimitive class based on the underlying graphics API. The CirclePrimitive class supports the definition of center, radius, drawing, and moving of the circle, etc. Figure 18.11 also shows the four categories of methods that a typical model component must support:

```
class ApplicationModel    {                                          Class built on
    private:                                                         Graphics API
    // Application's private state
        vector<CirclePrimitive>   AllWorldBalls     // World balls, initially empty
        CirclePrimitive           HeroBall          // The Hero Ball
        bool                      DefiningNewHero   // If LMB drag is true

    public:                                                          1. Application
        bool  IsDefinningHeroBall()                                  state inquires
                // If LMB drag is true, same as if we are in
                // the middle of defiing the a new hero ball
        bool  HeroBallExists()
                // Current hero ball is not null
        int   NumBallsOnScreen()
                // Number of balls currently on screen
        float HeroVelocityX()
                // Hero's velocity, x-component
        float HeroVelocityY()                                        2. Application
                // Hero's velocity, y-component                      state changes
                                                                     from user events
        void  CreateHeroBall (mousePosition)
                // Creates new hero ball, with center at mousePosition
                // radius and velocity are initialized to zero
        void  DragHeroBallTo (mousePosition)
                // Refine radius and velocity of hero ball based on
                // hero's center and current mousePosition
        void  SetHeroBallVelocity (velocityX, velocityY)             3. Application
                // Sets current hero ball velocity                   state changes
                // if there is no current hero ball, nothing happens from application
        void  InsertHeroToAllWorld( )                                events
                // Done defining HeroBall, insert into WorldBallSet
        void  SelectHeroBall (mousePosition)
                // Sets hero ball to be the one currently under
                // mousePosition sets to null if none exists
        void  UpdateSimulation( )                                    4. Application
                // Move balls by their velocities, update velocity   state visualization
                // by gravity and remove off-screen ones
        void  RedrawApplicationState( )
                // Draw all the freefalling balls (including the HeroBall)
                // to the desired region on the application window.
}
```

Figure 18.11. The model component of the ball shooting program.

1. **Application state inquiries.** These are functions that return the application state. These functions are important for maintaining up-to-date GUI elements (e.g., status echo or velocity slider bars).

2. **Application state changes from user events.** These are functions that change the application state according to a user's input actions. Notice that the function names should reflect the functionality (e.g., CreateHeroBall) and not the user event actions (e.g., ServiceLMBDown). It is common for a group of functions to support a defined finite state transition. For exam-

ple, CreateHeroBall, DragHeroBall, and InsertHeroToWorld implement the finite state diagram of Figure 18.9.

3. **Application state changes from application (timer) events.** This is a function that updates the application state resulting from purposeful and usually synchronous application timer events. For the ball shooting program, we update all of the velocities, displace the balls' positions by the updated velocities and compute ball-to-ball collisions, as well as remove off-screen balls.

4. **Application state visualization.** This is a function that knows how to draw the application state (e.g., drawing the necessary number of circles at the corresponding positions). It is expected that a view component will initialize appropriate regions on the application window, set up transformations, and invoke this function to draw into the initialized region.

It is important to recognize that the user's asynchronous events are arriving in between synchronous application timer events. In practice, a user observes an instantaneous application state (the graphics in the application window) and generates asynchronous events to alter the application state for the next round of simulation. For example, a user sees that the HeroBall is about to collide with another ball and decides to change the HeroBall's velocity to avoid the collision that would have happened in the next round of simulation. This means, before synchronous timer update, we must ensure all existing asynchronous user events are processed. In addition, the application should provide continuous feedback to ensure that users are observing an up-to-date application state. This subtle handling of event arrival and processing order is not an issue for simple, single-user applications like our ball shooting program. On large scale multi-user networked interactive systems, where input event and output display latencies may be significant, the *UpdateSimulation*() function is often divided into pre-update, update, and post-update.

The View

Figure 18.12 shows the ApplicationView class supporting the two main functionalities of a view component: coordinate space transformation and initialization for redraw. As discussed earlier, the controller is responsible for calling the *DeviceToWorldXform*() to communicate user input points to the model component. The viewport class is introduced to encapsulate the highly API-dependent device initialization and transformation procedures.

```
class Viewport {
        private:
            // An area on application window for drawing.
            // Actual implemenation of the viewport is GraphicsAPI dependent.
        public:
            void        EraseViewport()
                            // Erase the area on the application window
            void        ActivateViewportForDrawing()
                            // All subsequent Graphics API draw commands
                            // will show up on this viewport
}
class ApplicationView {
        private:
            // a view's private state information
            Viewport    TargetDrawArea
                            // An area of the application main window that
                            // this view will be drawing to
        public:
            void        DeviceToWorldXform( inputDevicePoint, outputModelPoint)
                            // transform the input device coordinate point to
                            // output point in a coordinate system that the model understands

            void        DrawView( ApplicationModel TheModel)
                            // Erase and activate the TargetDrawArea and then
                            // Sets up transformation for TheModel
                            // calls TheModel.DrawApplicationState() to draw all the balls.
    }
```

Figure 18.12. The view component of the ball shooting program.

The Controllers

We can improve the solution of Figure 18.8 to better support the specified functionality of the ball shooting program. Recall that the application window depicted in Figure 18.2 has two distinct regions for interpreting events: the upper application drawing area where mouse button events are associated with defining/selecting the HeroBall and the lower GUI element area where mouse button events on the GUI elements have different meanings (e.g., mouse button events on the slider bars generate SliderBarChange events, etc.). We also notice that the upper application drawing area is the exact same area where the ApplicationView must direct the drawings of the ApplicationModel state.

Figure 18.13 introduces two types of controller classes: a *ViewController* and a *GenericController*. Each controller class is dedicated to receiving input events from the corresponding region on the application window. The ViewController creates an ApplicationView during initialization such that the view can be tightly paired for drawing of the ApplicationModel state in the same area. In addition, the ViewController class also defines the appropriate mouse event service routines to support the interaction with the HeroBall. The GenericController is meant to contain GUI elements for interacting with the application state.

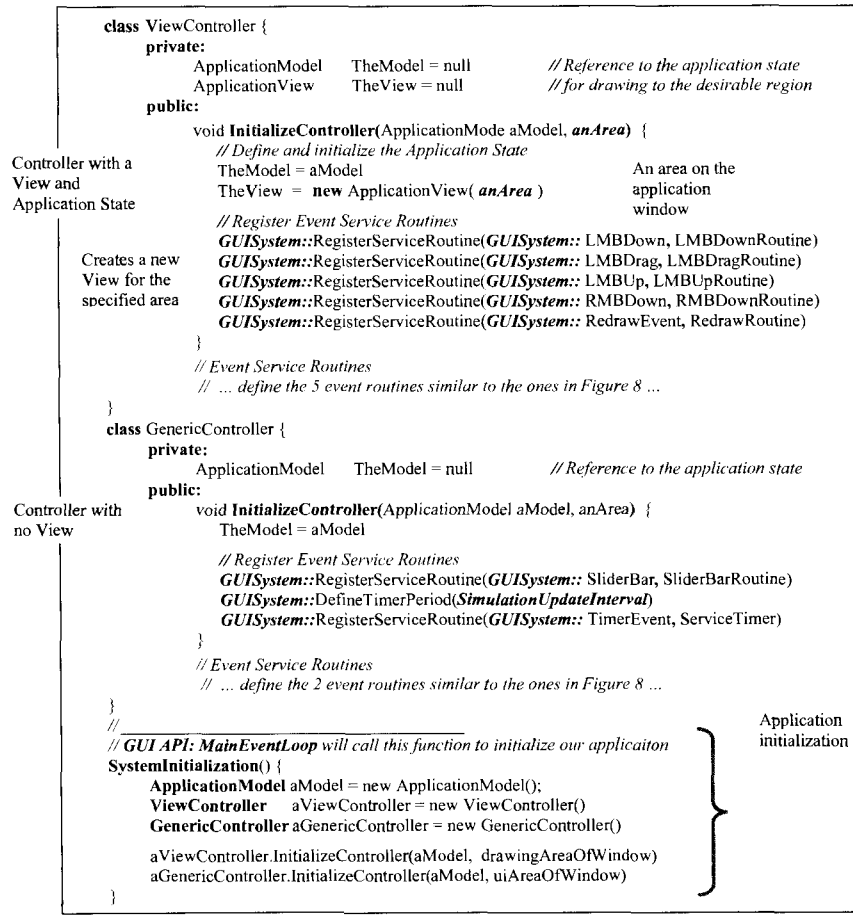

Figure 18.13. The controller component of the ball shooting program.

The bottom of Figure 18.13 illustrates that the GUI API MainEventLoop will still call the *SystemInitialization*() function to initialize the application. In this case, we create one instance each of ViewController and GenericController. The ViewController is initialized to monitor mouse button events in the drawing area of the application window (e.g., LMB click to define HeroBall), while the GenericController is initialized to monitor the GUI element state changes (e.g., LMB dragging of a slider bar). Notice that the service of the timer event is global to the entire application and should be defined in only one of the controllers (either one will do).

In practice, the GUI API MainEventLoop *dispatches* events to the controllers based on the *context of the event*. The context of an event is typically defined by

the location of the mouse pointer or the current *focus* of the GUI element (i.e., which element is active). The application is responsible for creating a controller for any region on the window that it will receive events directly from the GUI API.

18.3.3 Using the MVC to Expand the Ball Shooting Program

One interesting characteristic of the MVC solution presented in Section 18.3.2 is that the model component does not have any knowledge of the view or the controller components. This clean interface allows us to expand our solution by inserting additional view/controller pairs.

For example, Figure 18.14 shows an extension to the ball shooting program given in Figure 18.2. It has an additional small view in the UI (user interface) area next to the quit button. The small view is exactly the same as the *original large view*, except that it covers a smaller area on the application window.

Figure 18.15 shows that, with our MVC solution design, we can implement the small view by creating a new instance of ViewController (an additional ApplicaitonView will be created by the ViewController) for the desired application window area. Notice that the GenericController's window area actually contains

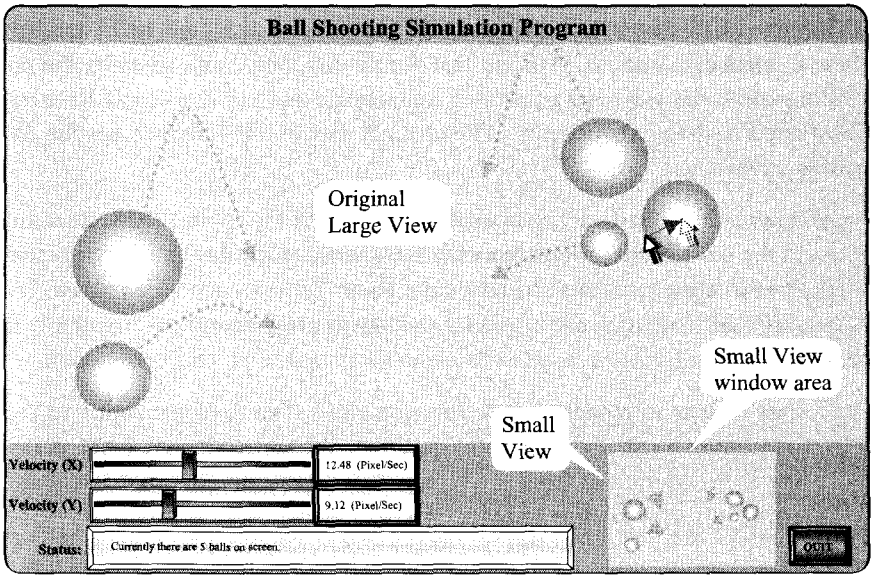

Figure 18.14. The ball shooting program with large and small views.

```
//
// GUI API: MainEventLoop will call this function to initialize our applicaiton
SystemInitialization() {                                          New instance of
    ApplicationModel  aModel = new ApplicationModel();            ViewController (and
    ViewController    aLargeViewController = new ViewController()  ApplicationView)
    GenericController aGenericController = new GenericController()

    aLargeViewController.InitializeController(aModel, drawingAreaOfWindow)
    aGenericController.InitializeController(aModel, uiAreaOfWindow)

    ViewController aSmallViewController = new ViewController()
    aSmallViewController.InitializeController(aModel, smallViewDrawingArea)
}
```

Figure 18.15. Implementing the small view for the ball shooting program.

the area of the small ViewController. When a user event is triggered in this area, the "top-layer" controller (the visible one) will receive the event. After the initialization, the new small view will behave in exactly the same manner as the original large view.

For simplicity, Figure 18.14 shows two identical view/controller pairs. In general, a new view/controller pair is created to present a different visualization of the application state. For example, with slight modifications to the view component's transformation functionality, the large view of Figure 18.14 can be configured into a *zoom view* and the small view can be configured into a *work view*, where the zoom view can zoom into different regions (e.g., around the HeroBall) and the work view can present the entire application space (e.g., all the free falling balls).

Figure 18.16 shows the components of the solution in Figure 18.15 and how these components interact. We see that the model component supports the operations of all the view and controller components and yet it does not have any knowledge of these components. This distinct and simple interface has the following advantages:

1. Simplicity. The model component is the core of the application and usually is the most complicated component. By keeping the design of this component independent from any particular controller (user input/events) or view (specific drawing area), we can avoid unnecessary complexity.

2. Portability. The controller component typically performs the *translation* of user actions to model-specific function calls. The implementation of this translation is usually simple and specific to the underlying GUI API. Keeping the model clean from the highly API-dependent controller facilitates portability of a solution to other GUI platforms.

3. Expandability. The model component supports changing of its internal state and understands how to draw its contents. As we have seen (Figures 18.15

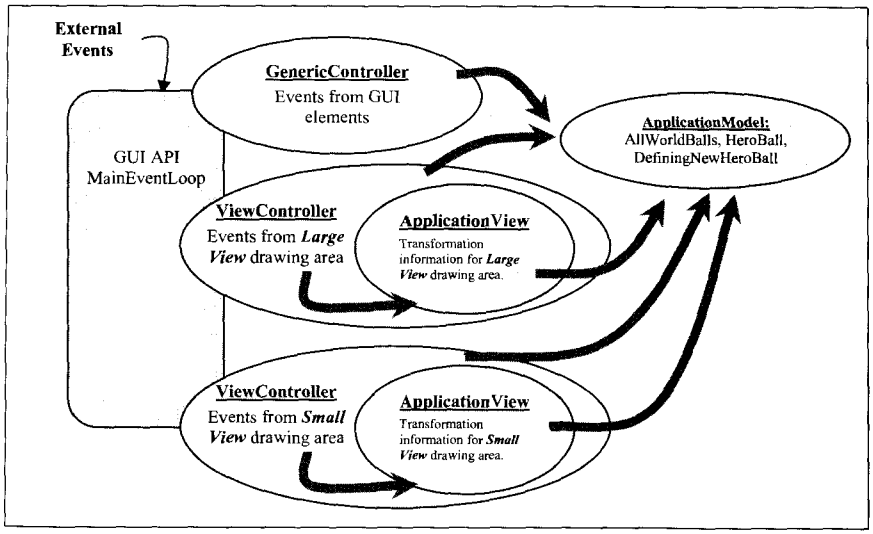

Figure 18.16. Components of the ball shooting program with small view.

and 18.16), this means that it is straightforward to add new view/controller pairs to increase the interactivity of the application.

18.3.4 Interaction Among The MVC Components

The MVC framework is a tool for describing general interactive systems. One of the beauties of the framework is that it is straightforward to support multiple view/controller pairs. Each view/controller pair shares responsibilities in exactly the same way: the view *presents* the model and the controller allows the events (user-generated or otherwise) to change the model component.

For an application with multiple view/controller pairs, like the one depicted in Figure 18.16, we see that a user can change the model component via any of the three controllers. In addition, the application itself is also capable of changing the model state. All components must however, ensure that a coherent and up-to-date presentation is maintained for the user. For example, when a user drags out a new HeroBall, both the large and small view components must display the dragging of the ball, while the GenericController component must ensure that the slider bars properly echo the implicitly defined HeroBall velocity. In the classical MVC model, the coherency among different components is maintained with an elaborate protocol (e.g., via the observer design pattern). Although the classical

MVC model works very well, the elaborate protocol requires that all components communicate or otherwise to *keep track* of changes in the model component.

In our case, and in the case of most modern interactive graphics systems, the application defines the timer event for simulation computation. To support *smooth* simulation results, we have seen that the timer event typically triggers within real-time response thresholds (e.g., 20–50 milliseconds). When servicing the timer events, our application can take the opportunity to maintain coherent states among all components. For example, in the *ServiceTimer*() function in Figure 18.8, we update the velocity slider bars based on current HeroBall velocity. In effect, during each timer event service, the application *pushes* the up-to-date model information to all components and *forces* the components to refresh their presentation for the user. In this way, the communication protocol among the components becomes trivial. All components keep a reference to the model, and each view/controller pair in the application does not need to be aware of the existence of other view/controller pairs. In between periodic timer events, the user's asynchronous events change the model. These changes are only made in the model component, and no other components in the application need to be aware of the changes. During the periodic timer service, besides computing the model's simulation update, all components poll the model for up-to-date state information. For example, when the user clicks and drags with the left mouse button pressed, a new HeroBall will be defined in the model component. During this time, the large and small view components will not display the new HeroBall, and the velocity slider bars will not show the new HeroBall's velocity. These components will get and display up-to-date HeroBall information only during the application timer event servicing. Since the timer event is triggered more than 30 times per second, the user will observe a smooth and up-to-date application state in all components at all times.

18.3.5 Applying the MVC Concept

The MVC framework is applicable to general interactive systems. As we have seen in this section, interactive systems with the MVC framework result in clearly defined component behaviors. In addition, with clearly defined interfaces among the components, it becomes straightforward to expand the system with additional view/controller pairs.

An *interactive system* does not need to be an elaborate software application. For example, the slider bar is a fully functional interactive system. The model component contains a current *value* (typically a floating point number), the view component presents this value to the user, and the controller allows the user to in-

teractively change this value. A typical view component draws rectangular icons (bar and knobs) representing the current value in the model component, while the controller component typically supports mouse down and drag events to interactively change the value in the model component. With this understanding, it becomes straightforward to expand the system with additional view/controller pairs. For example, in our ball shooting program, the slider bars have an additional view component where the numeric value of the model is displayed. In this case, there is no complementary controller component defined for the numeric view; an example complementary controller would allow the user to type in numeric values.

18.4 Example Implementations

Figure 18.17 shows two implementations of the solution presented in Section 18.3.3. The version on the left is based on OpenGL and FLTK, while the version to the right is based on D3D and MFC. In this section, we present the details of these two implementations. The lessons we want to learn are that (a) a proper MVC solution framework should be independent from any implementation and (b) a well designed implementation should be realizable based on and/or easily ported to any suitable API.

Before examining the details of each implementation, we will develop some understanding for working with modern GUI and graphics APIs.

18.4.1 Working with GUI APIs

Building the Graphical User Interface (GUI) of an application involves two distinct steps. The first step is to *design the layout* of the user interface system. In this step, an application developer places GUI elements (e.g., buttons, slider bars, etc.)

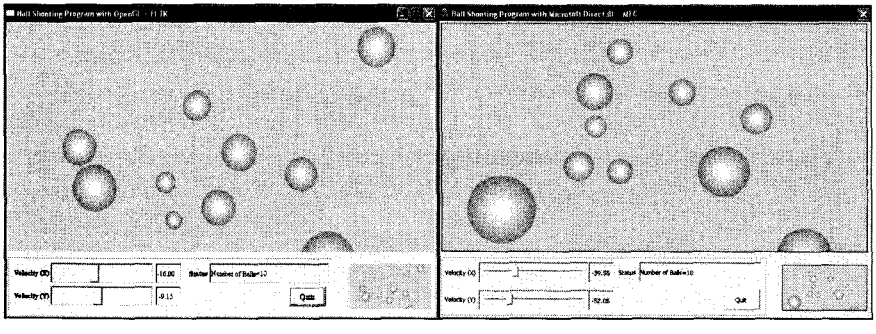

Figure 18.17. Ball shooting programs with OpenGL+FLTK and D3D+MFC.

in an area that represents the application window. The GUI elements are typically two-dimensional graphical artifacts (e.g., a 3D looking icon representing a slider bar). The goal of this first step is to arrange these graphical artifacts to achieve user friendliness and maximum usability (e.g., what is the best place/color/size for the slider bar, etc.). The second step in building a GUI for an application is to *semantically link* the GUI elements to the functionality of the application (e.g., update HeroBall velocity when the slider bar is dragged). In this step, an application developer builds the code for the necessary functionality (e.g., code for changing HeroBall velocity) and *registers* this code with the on-screen graphical artifacts (e.g., the slider bar). This is precisely the *event service registration* described in Section 18.2.2.

Modern GUI APIs support the building of a graphical user interface with a *GUI builder*. A GUI builder is an interactive graphical editor that allows its user to interactively place and manipulate the appearances of GUI elements. In addition, the GUI builder assists the application developer to compose or generate service routines and links those service routines to the events generated by the GUI elements.

Figure 18.18 illustrates the mechanism by which the GUI builder (in the middle of the figure) links the graphical user interface front-end (left side of the the figure) to the user-developed program code (right side of the figure). The patterned ellipse, the GUI Builder, is shown in the middle of Figure 18.18 The arrow pointing left towards the application (*A Simple Program*) indicates that the application developer works with the GUI builder to design the layout of the application (e.g., where to place the button or the status echo area). The arrows pointing from the GUI builder toward the *MainEventLoop* and *Event Service Linkage* mod-

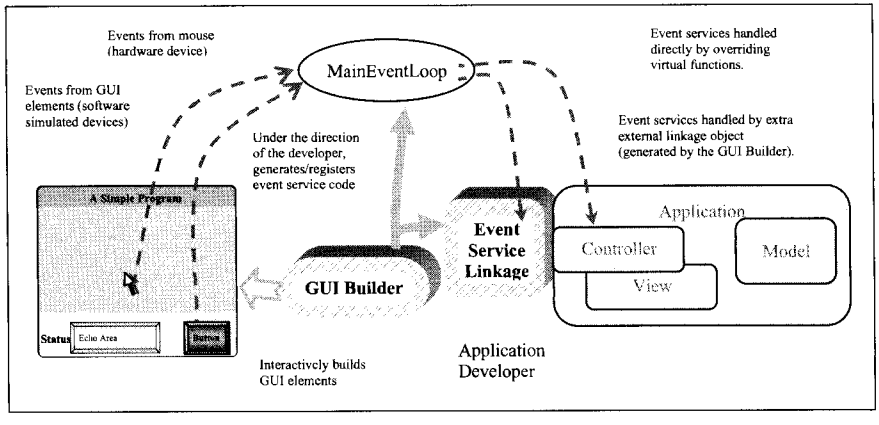

Figure 18.18. Working with a GUI API.

ules indicate that the GUI builder is capable of generating programming code to register event services. In Figure 18.18, there are two dotted connections between the mouse and the button GUI element through the MainEventLoop module to the event service linkage and the application controller modules. These two connections represent the two different mechanisms with which GUI APIs support event services:

1. External Service Linkage. Some GUI builders generate extra program modules (e.g., in the form of source code files) with code fragments supplied by the application developer to semantically link the GUI elements to the application functionality. For example, when the "button" of "A Simple Program" is clicked, the GUI builder ensures that a function in the "Event Service Linkage" module will be called. It is the application developer's responsibility to insert code fragments into this function to implement the required action.
2. Internal Direct Code Modification. Some GUI builders insert linkage programming code directly into the application source code. For example, the GUI builder modifies the source code of the application's controller class and inserts a new function to be called when the "button" of "A Simple Program" is clicked. Notice that the GUI Builder only inserts an empty function; the application developer is still responsible for filling in the details of this new function.

The advantage of an external service linkage mechanism is that the GUI builder only has minimal knowledge of the application source code. This provides a simple and flexible development environment where the developer is free to organize the source code structure, variable names, etc., in any appropriate way. However, the externally generated programming module implies a loosely integrated environment. For example, to modify the "button" behavior of "A Simple Program," the application developer must invoke the GUI builder, modify code fragments, and re-generate the external program module. The Internal Direct Code Modification mechanism in contrast provides a better integrated environment where the GUI builder modifies the application program source code directly. However, to support proper "direct code modification," the GUI builder must have intimate knowledge of, and often places severe constraints on, the application source code system (e.g., source code organization, file names, variable names, etc.).

18.4.2 Working with Graphics APIs

Figure 18.19 illustrates that one way to understand a modern graphics API is by considering the API as a functional interface to the underlying graphics hardware.

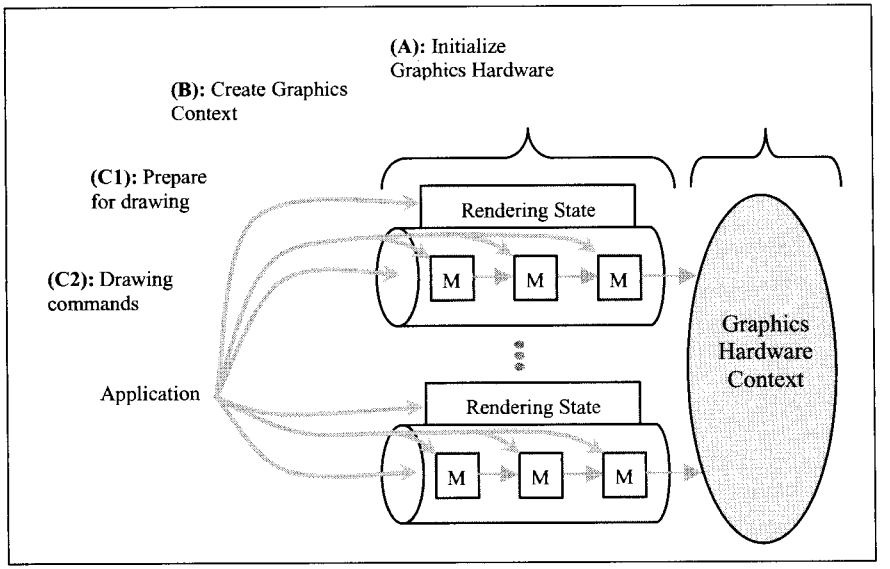

Figure 18.19. Working with a graphics API.

It is convenient to consider this functional interface as consisting of two stages: *Graphics Hardware Context (GHC)* and *Graphics Device Context (GDC).*

Graphics Hardware Context (GHC). This stage is depicted as the vertical ellipse on the right of Figure 18.19. We consider the GHC as a configuration which wraps over the hardware video display card. An application creates a GHC for each unique configuration (e.g., depth of frame buffer or z-buffer, etc.) of the hardware video card(s). Many Graphics Device Contexts (see below) can be connected to each GHC to support drawing to multiple on-screen areas from the same application.

Graphics Device Context (GDC). This stage is depicted as a cylindrical *pipe* in Figure 18.19. The multiple pipes in the figure illustrates that an application can create multiple GDCs to connect to the same GHC. Through each GDC, an application can draw to distinct areas on the application window. To properly support this functionality, each GDC represents a complete *rendering state.* A rendering state encompasses all the information that affects the final appearance of an image. This includes primitive attributes, illumination parameters, coordinate transformations, etc. Examples of primitive attributes are color, size, pattern, etc., while examples of illumination parameters include light position, light color, surface material properties, etc. Graphics APIs typically support coordinate trans-

formation with a series of two or three matrix processors. In Figure 18.19, the "M" boxes inside the GDC pipes are the matrix processors. Each matrix processor has a transformation matrix and transforms input vertices using this matrix. Since these processors operate in series, together they are capable of implementing multi-stage coordinate space transformations (e.g., object to world, world to eye, and eye to projected space). The application must load these matrix processors with appropriate matrices to implement a desired transformation.

With this understanding, Figure 18.19 illustrates that to work with a graphics API, an application will

(A) initialize one or more GHCs. Each GHC represents a unique configuration of the graphics video card(s). In typical cases, one GHC is initialized and configured to be shared by the entire application.

(B) create one or more GDCs. Each GDC supports drawing to distinct areas on the application window. For example, an application might create a GDC for each view component in an application.

(C) draw using a GDC. An application draws to a desired window area via the corresponding GDC. Referring to Figure 18.19, an application sets the rendering state (C1) and then issues drawing commands to the GDC (C2). Setting of the rendering state involves setting of all relevant primitive and illumination attributes and computing/loading appropriate transformation matrices into the matrix processors. A drawing command is typically a series of vertex positions accompanied by instructions on how to interpret the vertices (e.g., two vertex positions and an instruction that these are end points of a line).

In practice, modern graphics APIs are highly configurable and support many abstract programming modes. For example, Microsoft's Direct3D supports a drawing mode where the matrix processors can be by-passed entirely (e.g., when vertices are pre-transformed).

18.4.3 Implementation Details

Figure 18.20 shows the design of our implementation for the solution presented in Section 18.3.3.[1] Here, the MainUIWindow object represents the entire ball shooting program. This object contains the GUI elements (slider bars, quit button, etc.), the model (application state), and two instances of view/controller pairs (one each for LargeView and SmallView).

[1] Source code for this section can be found at http://faculty.washington.edu/ksung/fcg2/ball.tar.zip

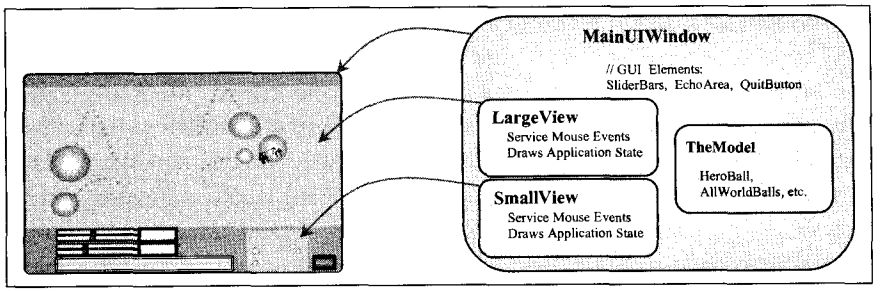

Figure 18.20. Implementation of the ball shooting program with two views.

OpenGL with FLTK

Figure 18.21 shows a screen shot of *Fluid*, FLTK's GUI builder, during the construction of the GUI for the ball shooting program. In the lower-right corner of Figure 18.21, we see that (A) Fluid allows an application developer to interactively place graphical representations of GUI elements (3D-looking icons); (B) is an area representing the application window. In addition (C), the application developer can interactively select each GUI element to define its physical appearances (color, shape, size, etc.). In the lower-left corner of Figure 18.21, we see that (D) the application developer has the option to type in program fragments to service events generated by the corresponding GUI element. In this case, we can see that the developer must type in the program fragment for handling the X velocity slider bar events. Notice that this program fragment is separated from the rest of the program source code system and is associated with Fluid (the GUI builder). At the conclusion of the GUI layout design, Fluid generates new source code files to be included with the rest of the application development environment.

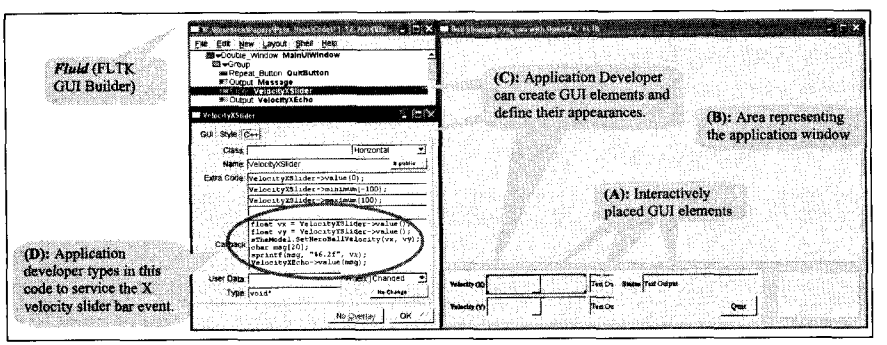

Figure 18.21. Fluid: FLTK's GUI Builder.

```
// Forward declaration of mouse event service routines
    void ServiceMouse(int button, int state, int x, int y); // service mouse button click
    void ServiceActiveMouse(int x, int y);                  // service mouse drag
class MainUIWindow {
    UserInterface    UI;          // This is Linkage Code generated by Fluid (GUI Builder)
                                  //    This object services events geneated by GUI elements
    Model            *TheModel;   // The application State (Figure 11)
    FlGlutWindow     *LargeView;  // These are View/Controller pairs that understand graphics
    FlGlutWindow     *SmallView;  //    outputs (GDC) and mouse events (controller)

    MainUIWindow(Model *m)   {                          // The constructor
        TheModel = m;                                   // Sets the model ...
        LargeView = new FlGlutWindow(TheModel);         // Create LargeView
        LargeView->mouse = ServiceMouse;          // callback functions for service mouse events
        LargeView->motion = ServiceActiveMouse;
        // Create SmallView ... exactly the same as LargeView (not shown)
        glutTimeFunc( // set up timer and services )   // Set up timer ...
    }
};
```

Figure 18.22. MainUIWindow based on OpenGL and FLTK.

Since these source code files are controlled and generated by the GUI builder, the application developer must invoke the GUI builder in order to update/maintain the event service routines. In this way, FLTK implements external service linkage as described in Section 18.4.1. In our implementation, we instruct *Fluid* to create a *UserInterface* class (.h and .cpp files) for the integration with the rest of our application development environment.

Figure 18.22 shows the *MainUIWindow* implementation with OpenGL and FLTK. In this case, graphics operations are performed through OpenGL and user interface operations are supported by FLTK. As described, the *UserInterface* object in the MainUIWindow is created by Fluid for servicing GUI events. TheModel is the application state as detailed in Figure 18.11. The two FlGlutWindow objects are based on a predefined FLTK class designed specifically for supporting drawing with OpenGL. The constructor of MainUIWindow shows that the mouse event services are registered via a callback mechanism. As discussed in Section 18.2.4, the FLTK (Fast Light ToolKit) is an example of a light weight GUI API. Here, we see examples of using callback as a registration mechanism for receiving user events.

FlGlutWindow is a FLTK pre-defined Fl_Glut_Window class object (see Figure 18.23) designed specifically to support drawing with OpenGL. Each instance of a FlGlutWindow object is a combination of a controller (e.g., to receive mouse events) and a Graphics Device Context (GDC). We see that the *draw*() function first sets the rendering state (e.g., clear color and matrix values), including computing and programming the matrix processor (e.g., GL_PROJECTION), before calling TheModel to re-draw the application state.

```
// Fl_Glut_Window is a pure virtual class supplied by FLTK specifically for supporting
// windows with OpenGL output and for receiving mouse events.
class FlGlutWindow : public Fl_Glut_Window {
  FlGlutWindow(Model *m);              // Constructor
  Model    *TheModel;                  // The application state: initialized during construction time.
  float    WorldWidth, WorldHeight;    // World Space Dimension

  void HardwareToWorldPoint(int hwX, int hwY, float &wcX, float &wcY);
                                       // Transform mouse clicks (hwX, hwY) to World Cooridnate (wcX, wcY)

  virtual void draw() {                // virtual function from Fl_Glut_Window for drawing
    glClearColor( 0.8f, 0.8f, 0.95f, 0.0f );
    glClear(GL_COLOR_BUFFER_BIT);      // Clearing the background color
    glMatrixMode(GL_PROJECTION);       // Programming the OpenGL's GL_PROJECTION
        glLoadIdentity();              //      Matrix Processor to the propoer transfrotm
        gluOrtho2D(0.0f, WorldWidth, 0.0f, WorldHeight);
    TheModel->DrawApplicaitonState();  // Drawing of the application state
  }
};
```

Figure 18.23. FlGlutWindow: OpenGL/FLTK view/controller pair.

Direct3D with MFC

Figure 18.24 shows a screen shot of the MFC resource editor, MFC's GUI builder, during the construction of the ball shooting program. Similar to Fluid (Figure 18.21), in the middle of Figure 18.24, (A) we see that the resource editor also supports interactive designing of the GUI element layout in (B), an area representing the application window. Although the GUI builder interfaces operate differently, we observe that in (C), the MFC resource editor also supports the definition/modification of the physical appearance of GUI elements. However,

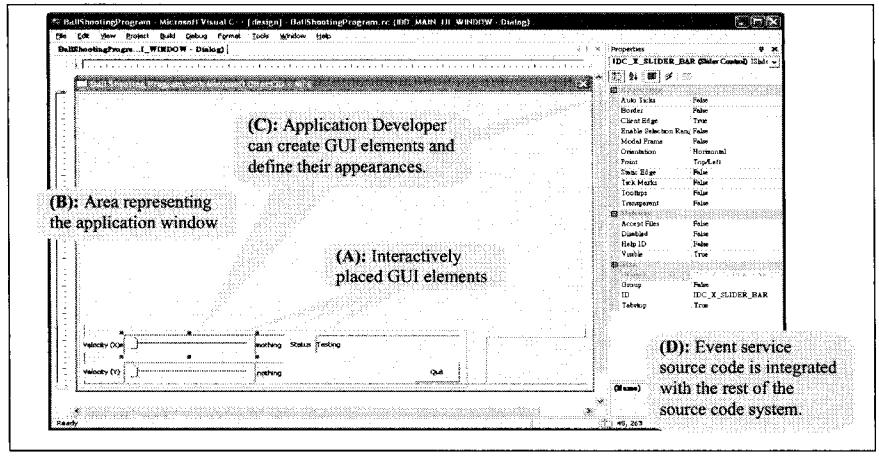

Figure 18.24. The MFC resource editor.

```
class MainUIWindow : public CDialog {
    Model          *TheModel;   // The application State (Figure 11)
    LPDIRECT3D9    TheGHC;      // This is the Graphics Hardware Context
    CWndD3D        *LargeView;  // These are View/Controller pairs that understand drawing
    CWndD3D        *SmallView;  //     with D3D (GDC) and UI element events (controller)

    CSliderCtrl    XSlider, YSlider;   // These are the GUI elements
    CStringEcho    StatusEcho;

        void    OnTimer();             // Override the Timer service function
        void    OnHScroll( ...);       // Override the Scroll bar service function
};
```

Figure 18.25. MainUIWindow based on Microsoft Direct3D and MFC.

unlike Fluid, the MFC resource editor is tightly integrated with the rest of the development environment. In this case, a developer can register for event services by inheriting or overriding appropriate service routines. The MFC resource editor automatically inserts code fragments into the application source code system. To support this functionality, the application source code organization is governed/shared with the GUI builder; the application developer is not entirely free to rename files/classes and/or to re-organize implementation source code file system structure. MFC implements internal direct code modification for event service linkage, as described in Section 18.4.1.

Figure 18.25 shows the *MainUIWindow* implementation with Direct3D and MFC. In this implementation, graphics operations are performed through Direct3D while user interface operations are supported by MFC. Once again, TheModel is the application state as detailed in Figure 18.11. *LPDIRECT3D9* is the Graphics Hardware Context (GHC) interface object. This object is created and initialized in the MainUIWindow constructor (not shown here). The two CWndD3D objects are defined to support drawing with Direct3D. We notice that one major difference between Figure 18.25 and Figure 18.22 is in the GUI element support. In Figure 18.25, we see that the GUI element objects (e.g., XSlider) and the corresponding service routines (e.g., OnHScroll()) are integrated into the MainUIWindow object. This is in contrast to the solution shown in Figure 18.22 where GUI elements are grouped into a separate object (e.g., the UserInterface object) with callback event service registrations. As discussed in Section 18.2.4, MFC is an example of a large commercial GUI API, where many event services are registered based on object-oriented function overrides (e.g., the OnHScroll() and OnTimer() functions).

CWndD3D is a sub-class of the MFC CWnd class (see Figure 18.26). CWnd is the base class designed for a generic MFC window. By sub-classing from this base class, CWndD3D can support all default window-related events (e.g., mouse

```
// CWnd is the MFC base class for all window objects. Here we subclass to create a D3D output
// window by including a D3D Graphics Device Context.
class CWndD3D: public CWnd {
  LPDIRECT3DDEVICE9  D3DDevice;   // This is the D3D Graphics Device Context (GDC)
  Model              *TheModel;   // The application state
  void InitD3D(LPDIRECT3D9);      // Create D3DDevice (GDC) to connect to GHC

  void RedrawView()  {            // Draws the Application State
        // Compute world coordinate to device transform
        D3DMATRIX transform = ComputeTransformation();
        D3DDevice->SetTransform(D3DTS_WORLD, &transform);
              // Programming the D3D_WORLD matrix with the computed transform matrix

        D3DDevice->Clear( bgColor, D3DCLEAR_TARGET);
        D3DDevice->BeginScene();
          TheModel->DrawApplicationState(D3DDevice);
        D3DDevice->EndScene();
        D3DDevice->Present();
  }

  void HardwareToWorldPoint(CPoint hwPt, float &wcX, float &wcY);
                          // Transform mouse clicks (hwPt) to world coordinate (wcX, wcY)

  void OnLButtonDown(CPoint hwPt);  // Override mouse button/drag service functions
  :
};
```

Figure 18.26. CWndD3D: Direct3D/MFC view/controller pair.

events). The *LPDIRECT3DDEVICE9* object is the D3D Graphics Device Context (GDC) interface object. The *InitD3D*() function creates and initializes the GDC object and connects this object to the *LPDIRECT3D9* (GHC). In this way, a CWndD3D sub-class is a basic view/controller pair: it supports the view functionality with drawing via the D3D GDC and controller functionality with input via MFC. The *RedrawView*() function is similar to the *draw*() function of Figure 18.23 where we first set up the rendering state (e.g., bgColor and matrix), including programming the matrix processor (e.g., *D3DTS_WORLD*), before calling the model to draw itself.

In conclusion, we see that Figure 18.20 represents an implementation of the solution presented in Section 18.3.3 while Section 18.4.3 presented two versions of the implementation for Figure 18.20. Although the GUI Builder, event service registration, and actual API function calls are very different, the final programming source code structures are remarkably similar. In fact, the two versions share the exact same source code files for the *Model* class. In addition, although the drawing functions for *CirclePrimitive* are different for OpenGL and D3D, we were able to share the source code files for the rest of the primitive behaviors (e.g., set center/radius, travel with velocity, collide, etc.). We reaffirm our assertion that software framework, solution structures, and event implementations should be designed independent of any APIs.

18.5 Applying Our Results

We have seen that the event-driven programming model is well suited for designing and implementing programs that interact with users. In addition, we have seen that the modelview-controller framework is a convenient and powerful structure for organizing functional modules in an interactive graphics application. In developing a solution to the ball shooting program, we have demonstrated that knowledge from event-driven programming helps us design the controller component (e.g., handling of mouse events, etc.), computer graphics knowledge helps us design the view component (e.g., transformation and drawing of circles, etc.), while the model component is highly dependent upon the specific application (e.g., free falling and colliding circles). Our discussion so far has been based on a very simple example. We will now explore the applicability of the MVC framework and its implementation in real-world applications.

18.5.1 Example 1: PowerPoint

Figure 18.27 shows how we can apply our knowledge in analyzing and gaining insights into Microsoft PowerPoint,[2] a popular interactive graphics application. A screen shot of a slide creation session using the PowerPoint application is shown at the left of Figure 18.27. The right side of Figure 18.27 shows how we can apply the implementation framework to gain insights into the PowerPoint application. The MainUIWindow at the right of Figure 18.27 is the GUI window of

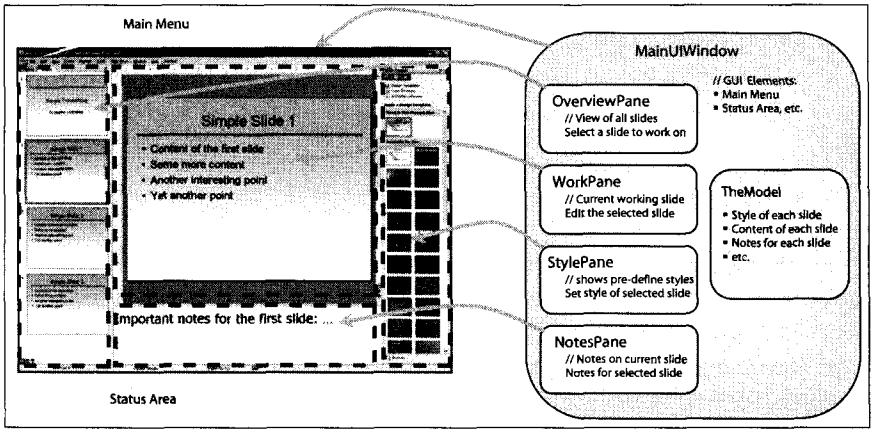

Figure 18.27. Understanding PowerPoint using the MVC implementation framework.

[2]Powerpoint is a registered trademark of Microsoft.

the entire application, and it contains the GUI elements that affect/echo the entire application state (e.g., main menu, status area, etc.). We can consider the MainUIWindow as the module that contains TheModel component and includes the four view/controller pairs.

Recall that TheModel is the state of the application and that this component contains all the data that the user interactively creates. In the case of PowerPoint, the user creates a collection of presentation slides, and thus TheModel contains all the information about these slides (e.g. layout design style, content of the slides, notes associated with each slide, etc.). With this understanding of TheModel component, the rest of the application can be considered as a convenient tool for presenting TheModel (the view) to the user and changing TheModel (the controller) by the user. In this way, these convenient tools are precisely the view/controller pairs (e.g., ViewController components from Figure 18.16).

In Figure 18.27, each of the four view/controller pairs (i.e., OverviewPane, WorkPane, StylePane, and NotesPane) presents, and supports changing of different aspects of TheModel component:

- **OverviewPane.** The view component displays multiple consecutive slides from all the slides that the user has created; the controller component supports user scrolling through all these slides and selecting one for editing.
- **WorkPane.** The view component displays the details of the slide that is currently being edited; the controller supports selecting and editing the content of this slide.
- **StylePane.** The view component displays the layout design of the slide that is currently being edited; the controller supports selecting and defining a new layout design for this slide.
- **NotesPane.** The view component displays the notes that the user has created for the slide that is currently being edited; the controller supports editing of this notes.

As is the case with most modern interactive applications, PowerPoint defines an application timer event to support user-defined animations (e.g., animated sequences between slide transitions). The coherency of the four view/controller pairs can be maintained during the servicing of this application timer event. For example, the user works with the StylePane to change the layout of the current slide in TheModel component. In the meantime, before servicing the next timer event, OverviewPane and WorkPane are not aware of the changes and display an out-of-date design for the current slide. During the servicing of the timer event,

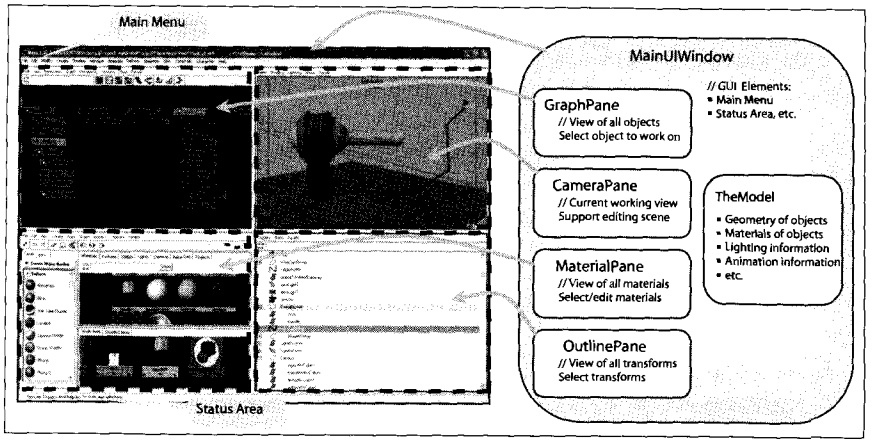

Figure 18.28. Understanding Maya with the MVC implementation framework.

the MainUIWindow forces all view/controller pairs to poll TheModel and refresh their contents. As discussed in Section 18.3.4, since the timer events are typically triggered more than 30 times in a second, the user is not be able to detect the brief out-of-date display and observes a consistent display at all times. In this way, the four view/controller pairs only need to keep a reference to TheModel component and do not need to have any knowledge of each other. Thus, it is straightforward to insert and delete view/controller pairs into/from the application.

18.5.2 Example 2: Maya

We now apply our knowledge in analyzing and understanding Maya[3], an interactive 3D modeling/animation/rendering system. The left side of Figure 18.28 shows a screen shot of Maya in a simple 3D content creation session. As in the case of Figure 18.27, the right side of Figure 18.28 shows how we can apply the implementation framework to gain insights into the Maya application. Once again we see that the MainUIWindow is the GUI window of the entire application containing GUI elements that affect/echo the entire application state, TheModel component, and all the view/controller pairs.

Since Maya is a 3D media creation system, TheModel component contains 3D content information (e.g. scene graph, 3D geometry, material properties, lighting, camera, animation, etc.). Once again, the rest of the components in the MainUIWindow are designed to facilitate the user's view and to change TheModel. Here is the functionality of the four view/controller pairs:

[3]Maya is a registered trademark of Alias.

- **GraphPane.** The view component displays the scene graph of the 3D content; the controller component supports navigating the graph and selecting scene nodes in the graph.
- **CameraPane.** The view component renders the scene graph from a camera viewing position; the controller component supports manipulating the camera view and selecting objects in the scene.
- **MaterialPane.** The view component displays all the defined materials; the controller component supports selecting and editing materials.
- **OutlinePane.** The view component displays all the transform nodes in the scene; the controller component supports manipulating the transforms (e.g. create/change parent-child relationships, etc.).

Once again, the coherency among the different view/controller pairs can be maintained while servicing the application timer events.

We do not speculate that PowerPoint or Maya is implemented according to our framework. These are highly sophisticated commercial applications and the underlying implementation is certainly much more complex. However, based on the knowledge we have gained from this chapter, we can begin to understand how to approach discussing, designing, and building such interactive graphics applications. Remember that the important lesson we want to learn from this chapter is how to organize the functionality of an interactive graphics application into components and understand how the components interact so that we can better understand, maintain, modify, and expand an interactive graphics application.

18.6 Notes

I first learned about the model view controller framework and event-driven programming from SmallTalk (Goldberg & Robson, 1989) (You may also want to refer to the SmallTalk web site (http://www.smalltalk.org/main/).) Both *Design Patterns—Elements of Reusable Object-Oriented Design* (Gamma, Helm, Johnson, & Vlissides, 1995) and *Pattern-Oriented Software Architecture* (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996) are excellent sources for finding out more about design patterns and software architecture frameworks in general. I recommend *3D Game Engine Architecture* (Eberly, 2004) as a good source for discussions on issues relating to implementing real-time graphics systems. I learned MFC and Direct3D mainly by referring to the online Microsoft Developer Network pages (http://msdn.microsoft.com). In addition, I find Prosise's

book *Programming Windows with MFC* (Prosise, 1999) to be very helpful. I refer to the *OpenGL Programming Guide* (Shreiner et al., 2004), *Reference Manual* (Shreiner, 2004), and FLTK on-line help (http://www.fltk.org/) when developing my OpenGL/FLTK programs.

18.7 Exercises

1. Here is the specification for dragging out a line:

   - Left mouse button (LMB) clicks define the center of the line.
   - LMB drags out a line such that the line extends in two directions. The first direction extends from the center (LMB click) position toward the current mouse position. The second direction extends in the opposite direction from the first with exactly the same length.
   - Right mouse button (RMB) click-drag moves the line such that the center of the line follows the current mouse position.

   (a) Follow the steps outlined in Section 18.2.3 and design an event-driven programming solution for this specification.

   (b) Implement your design with FLTK and OpenGL.

   (c) Implement your design with MFC and Direct-3D.

   Notice that in this case the useful application internal state information (the center position of the line) and the drawing presentation requirements (end points of the line) do not coincide exactly. When defining the application state, we should pay attention to what is the most important and convenient information to store in order to support the specified functionality.

2. For the line defined in Exercise 1, define a velocity that is the same as the slope of the line: once created, the line will travel along the direction defined by its slope. Use the length of the line as the speed. (Note that longer lines travel faster than shorter lines).

3. Here is the specification for dragging out a rectangle:

   - LMB click defines the center of the rectangle.
   - LMB drag out a rectangle such that the rectangle extends from the center position and one of the corner positions of the rectangle always follows the current mouse position.

- RMB click-drag moves the rectangle such that the center of the rectangle follows the current mouse position.

(a) Follow the steps outlined in Section 18.2.3 and design an event-driven programming solution for this specification.

(b) Implement your design with FLTK and OpenGL.

(c) Implement your design with MFC and Direct-3D.

4. For the rectangle in Exercise 3:

   (a) Support the definition of a velocity similar to that of HeroBall velocity in Section 18.1: once created, the rectangle will travel along a direction that is the vector defined from its center towards the LMB release position.

   (b) Design and implement collision between two rectangles (this is a simple 2D bound intersection check).

5. With results from Exercise 4, we can approximate a simple Pong game:

   - The paddles are rectangles;
   - A pong-ball is drawn as a circle but we will use the bounding square (a square that centers at the center of the circle, with dimension defined by the diameter of the circle) to approximate collision with the paddle.

   Design and implement a single-player pong-game where a ball (circle) drops under gravitational force and the user must manipulate a paddle to bounce the ball upward to prevent it from dropping below the application window. You should:

   (a) design a specification (similar to that of Section 18.1) for this pong game;

   (b) follow the steps outlined in Section 18.2.3 to design an event-driven programming solution;

   (c) implement your design either with OpenGL or Direct-3D.

6. Extend the ApplicationView in Figure 18.12 to include functionality for setting a world coordinate window bound. The world coordinate window bound defines a rectangular region in the world for displaying in the Viewport. Define a method for setting the world coordinate window bound and modify the *ApplicaitonView::DeviceToWorldXform*() function to support transforming mouse clicks to world coordinate space.

7. Integrate your results from Exercise 6 into the two-view ball shooting program from Figure 18.14 such that the small view can be focused around the current HeroBall. When there is no current HeroBall, the small view should display nothing. When user LMB click-drags, or when user RMB selects a HeroBall, the small view's world coordinate window bound should center at the HeroBall center and include a region that is 1.5 times the HeroBall diameter.