Pinball

CS32 Assignment 2

Due Dates:

Assignment Out:	Feb. 5, 1998
Design Check:	Feb 12, 1998 (See a TA by 11:59pm)
Design Handin Due:	Feb 14, 1998 (11:59pm)
UML Design Due:	Feb 19, 1998 (11:59pm)
Pinball Due:	Feb 26, 1998 (11:59pm)

Introduction

Your task is to write a pinball simulator that would make even the Who's Tommy proud. Well, okay, it just has to satiate him for about ten seconds. Since this is the second assignment in CS32, it is a significant step up in time and difficulty from Minesweeper. Thus, **starting late would be a very bad thing to do**. "Long is the way, and hard, that out of hell leads up to light." (Milton: Paradise Lost, line 432) Because of this, we've provided you with a number of graded checkpoints to guide you along the path towards a completed project and an enlightened soul.

Not an example of our support code.

Read this document many times. It is your primary source of wisdom during this project. Remember to be careful while reading: in the words of our beloved J. J. Rousseau, "I warn the reader that I have not mastered the art of making myself clear to the person who refuses to pay attention."

This assignment is designed to introduce you to a medium scale (*cough* big) C++ program, to test your ability to design and implement C++, to test your ability to design object-oriented programs and augment and modify those designs as needed, to give you experience with appropriate C++ programming and debugging tools, and to let you produce a program that you might enjoy playing.

Design Overview

In all honesty, Pinball is a challenging assignment. To do it in three weeks, you'd better have a crackerjack design. It's gotta be tight - more tight than a Kennedy on New Year's Eve. It's gotta

be cohesive, like the Wutang. It's gotta work, like the Crimson Permanent Assurance corporate pirates. And, when all is said and done, it's gotta be done in UML (Universal Modeling Language) with Rational Rose. There are three design checkpoints. The first is a TA checkpoint, which will help us make sure that you're heading on the right track. For this checkpoint, you should have a drawing of your design (hand-drawn is okay) at a high level of abstraction available for us to peruse. The second checkpoint is the final design checkpoint, where you will have to be able to explain, in detail, how your pinball machine is going to work. You should have a detailed written design available for us to evaluate, since your design will have to stand up to the notorious TA Tribunal. This checkpoint is to be turned in to the bin -- you will not be seeing a TA this time, though you are encourage to talk to a TA about your design before this deadline. For the final checkpoint, you will be required to submit an Rational Rose-ified version of your design, replete with object relationships and class hierarchies. Rational Rose will be discussed at great length during the Thursday, February 12th lecture.

Code Overview

What does coding a pinball simulation in CS32 entail, you ask? Good question.

First, you must become familiar with the support code, which is described in detail below. The support code, called the PIN Library, takes care of a lot of the nitty-gritty work of writing a pinball game. It allows you to concentrate on managing the internal logic of the game rather than worrying about the various drawing primitives, how to detect intersections between the ball and the various objects, and how to simulate gravity and force-added objects. The header file that we are providing, `pin.H`, provides some additional documentation and details the exact calling sequences. Because we're nice, friendly people, and because we want to destroy a rain forest or two, we've attached a copy of pin.H to the back of this handout.

After understanding the support code, you'll want to know about the level file format. (Honestly, these two go hand-in-hand.) No decent computer pinball game has just one board! As a result, we've provided you with a standard file format that can construct complex pinball boards. At the heart of the level description format is a way of describing pinball objects and the state transitions they should take. Part of the fun of playing a pinball game, after all, is figuring out how to get the most points and make the most noise. As the programmer of Pinball, you get to write software which can take an arbitrary set of state transition rules and execute them.

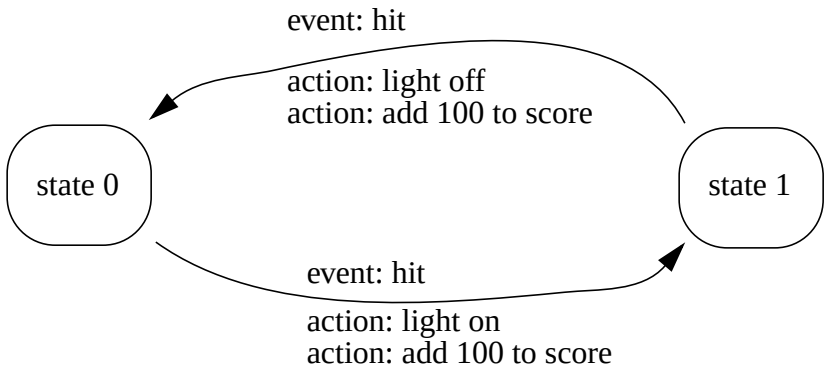
This said, coding will consist of three general tasks. First, you will have to write code to read in a level file, check for its validity, and construct pinball objects and internal information about the state logic of the game. Second, you will have to code routines for handling a general state machine. Finally, you will have to make use of the support code in order to correctly animate your board, test for user keystrokes, and perform other simulation tasks.

If you are careful with your object-oriented design, these three tasks won't be too tricky. For this project, you may (hint, hint) be better off with many small objects rather than a few big ones. Poor object-oriented design will lead to large code size and will make it difficult to complete the assignment on-time. A well-designed project will range from between three and six thousand lines of code, depending on the additions you make. Note that the better the design, the smaller the code.

Level File Format

Input files consist of two main parts. The first section of a file describes the objects that make up the pinball board. The second section of a file describes the logic of the game, e.g. what points are scored for hitting what objects, what internal states are present, when the player gets a free ball, when the game is over, etc. All input in the file consists of single line commands containing a command name followed by a set of arguments to the command. Blank lines and lines whose first non-blank character is a pound sign (#) are ignored.

So what's really going on here? The level file describes a finite state machine which controls the play of the game. For example, consider a common pinball device: when the ball hits a rebounder, it turns the light on and you get 100 points, and when you hit it again, you get another 100 points and the light goes off. Here's the finite state machine for the rebounder:



This finite state machine will be represented in the level file as a set of states and the set of transitions between them. You will notice, if you have looked at the sample level file we have provided you with (`/course/cs032/support/pinball/games/sample.game`), that after the finite state is set up there are a bunch of ACTION command lines. These lines specify what actions will occur when a state transitions into the given state. In our clever example an action would be giving the player some points or turning on/off the light.

Object Definitions

Objects are defined as 3D entities with respect to a fixed 2D board that is 10 units wide and 16 units high. The X coordinate of the board runs from -5.0 (left) to +5.0 (right); the Y coordinate from -8.0 (bottom) to +8.0 (top); the Z coordinate from 0 to whatever height is desired (probably 0.5) Most of the objects are defined using the OBJECT command:

Each object specification take a different number of parameters depending on what type of object it is. The handy table that follows shows you what parameters each object we provide you with expects to receive:

Command	Arguments
OBJECT	<name> WALL <width> <height> <#points> <x0> <y0> .. <xn> <yn>
	<name> BUMPER <width> <height> <#points> <x0> <y0> .. <xn> <yn>
	<name> BUTTONTARGET <x0> <y0> <x1> <y1> <width>
	<name> FLOORTARGET <x0> <y0> <radius>
	<name> FLIPTARGET <x0> <y0> <x1> <y1> <width> <height>
	<name> LIGHT <x0> <y0> <radius>
	<name> REBOUNDER <x0> <y0> <radius> <height>
	<name> VALVE <x0> <y0> <x1> <y1> <width> <height>
	<name> IMAGE <file> <x0> <y0> <x1> <y1>

* A **wall** is a flat surface of the given width and height centered along a line given by the sequence of points. Balls bounce normally off of walls.
* A **bumper** is a special type of wall consisting of an elastic cord stretched between posts. A ball hitting a bumper bounces off with some additional force.
* A **button target** is a vertical circular target that the ball can hit. It is placed along the line between (x0,y0) and (x1,y1) and has the given width. Note that because it is circular, its height is the same as its length.
* A **floor target** is a circular region in the floor that the ball activates by rolling over.
* A **flip target** is a rectangular target that has two states: in state 1 it is up and can be hit by the ball. In state 0, it is down and the ball moves smoothly over it.
* A **light** is a circular region on the floor that does not interact with the ball.
* A **rebounder** is a circular object that the ball can hit and will bounce off with additional force.
* A **valve** is a one way gate that provides no resistance to the ball in the forward direction and bounces the ball normally in the other direction.
* Finally, an **image** is a graphic which appears mapped onto the surface of the pinball board. <file> is the filename for the image, which must be in pnm format. (x0,y0) and (x1,y1) represent the two points which form the bounding rectangle of the image.

All these objects are tagged with the `name` parameter; this name is used later in the logic specifications.

All these specifications can be followed by any combination of the following standard arguments consisting of a keyword and value pairs:

Keyword	Values
COLORS	<primary color> <secondary color> <tertiary color>
FORCE	<additional force to be added on collision>
DRAWSTATE	<initial drawing state>
STATE	<initial object state>
SIDES	<sides to collide against>

* Each object uses between one and three colors; the **colors** parameter provides an initial setting of these colors.
* The **force** parameter allows the definition of an additional force to be added if the ball collides with this object with sufficient force. Specify additional force using a floating-point number (for example, FORCE 10.0 adds a hefty kick to a bumper).
* The **drawstate** specifies an initial state for drawing (i.e. for a flip target, whether it is up or down).
* Finally, each object maintains an integer state as a basis for the internal pinball logic. The **state** parameter allows setting the initial value of this state.
* The **sides** parameter specifies which sides of the object should be collided with. If omitted, all sides are considered. Possible values are FRONT, BACK, FRONT_BACK, EDGES, TOP, and ALL, all of which have corresponding constants defined in `pin.h`.

There are also three objects that are distinguished because they are affected by the user's key presses. These are defined using the commands:

Command	Arguments
LEFT	FLIPPER <x0> <y0> <w0> <x1> <y1> <w1> <height> <pivot>
RIGHT	FLIPPER <x0> <y0> <w0> <x1> <y1> <w1> <height> <pivot>
MAIN	SHOOTER <x0 > <y0> <x1> <y1> <width> <height>

* The first two define a trapezoidal **flipper** centered along the line from (x0,y0) to (x1,y1) with width w0 at its start point and w1 at its end point. The pivot location indicates where along the center line the pivot for the flipper occurs.
* The **shooter** is a box of the given width centered along the line from (x0,y0) to (x1,y1).

All these can have any of the standard parameters described above appended to their description.

There are three additional object commands:

Command	Arguments
BALL	<start_x> <start_y> <radius> <color>
TRIGGER	<name> [STATE <initial_state>]
TIMER	<name> <timeout in milliseconds> [<message>]

* The **ball** command defines the ball size and color and defines the position on the board where a new ball will appear (with zero velocity).
* The **trigger** command defines an object that does not appear on the board but that can be used as part of the state logic defined in the second part of the input specification.
* The **timer** command defines a named timeout of a given duration. If a message is specified here, it will be displayed until the timer expires.

Logic Definitions

The internal logic of a pinball game is what makes the game fun. The user's challenge in playing the game, in addition to the usual hand-eye coordination, is to deduce what the underlying logic is so as to maximize the number of points or to attain some hard-to-achieve (and high scoring) state. In order to allow you to design relatively complex pinball games of your own, our input language allows the definition of a rather complex state-action network. The basis for this network are the internal states of the objects defined above and the actions that are inherent to the game.

The logic is defined in two parts using the **state** and **action** commands:

Command	Arguments
STATE	<name> <state> <newstate> <event>
ACTION	<name> <state> <action>

The **state** command defines a transition from one state of the named object to another. The transition occurs when the specified event is seen. The states are integer values.

Look back to the finite state machine on page 3. The rebounder starts in state 0. The event that moves it to state 1 is a collision with the ball, which is the HIT event type.

The possible event include:

Format	Description
HIT	Ball collides with the named object
TIMEOUT <name>	Named timeout object times out

Format	Description
NEWGAME	Start of a game
NEWBALL	Start of a new ball
ENDBALL	After a ball has exited
ENDGAME	When a game is complete

The **action** specification defines what occurs when an object first enters a given state. The action specification here can be one of:

Action Format	Description
SOUND <name>	Play the named sound file
START <timer>	Start the specified timer
CANCEL <timer>	Cancel the specified timer
COLOR <c1> <c2> <c3>	Change the colors of the specified object
DRAWSTATE <state>	Change the drawstate of the given object
SET <name> <state>	Set the named object to the given state
SCORE <value>	Add the given value to the current score
ENDBALL	End the current ball
NEWBALL	Start a new ball (ends the current one as well)
FREEBALL	Give the user a free ball
MESSAGE <text>	Display the given message

Going back to the example on page 3 (with which you should be rapidly becoming familiar), moving into state 1 causes two ACTIONs to occur. One adds 100 to the score (SCORE 100), and the other changes the color of the rebounder to make it look like it is “lit up” (by using the COLOR action).

The PIN Library

The PIN library that we are providing takes care of a lot of the nitty-gritty work of the pinball game, allowing you to concentrate on managing the internal logic of the game rather than worrying about the various drawing primitives, how to detect intersections between the ball and the various objects, and how to simulate gravity and force-added objects. The header file that we are providing, pin.h, provides some documentation and exact calling sequences. Note that it also contains some methods that we are not describing here that may be used for extensions to the basic game.

The file defines the following standard types to be used in the interface:

Type Name	Description	C++ equivalent
PinString	String parameter	string
PinColor	Color parameter — currently the name of the color as a string	PinString
PinBoolean	A Boolean value	bool
PinInt	An integer value	int
PinState	A drawing state	double
PinCoord	Coordinate value or value used in computing	double
PinTime	Time in milliseconds	long
PinScore	Score value	long
PinKeySet	Combination of pressed keys — OR of PinKey enumeration	long
PinSound	Sound (file name of a ulaw sound file in `/course/cs032/lib/sounds`)	PinString
PinSide	Description of the sides of an object	(enumerated type)
PinCallback	User-defined callback object to handle next event	PinCallbackInfo *
PinDraw	Top level control object for drawing, etc.	PinDrawInfo *
PinTimeOut	Object for defining time outs	PinTimeOutInfo *
PinComponent	Component drawn on the pinball surface	PinComponentInfo *
PinBall	The ball component	PinBallInfo *
PinComponentFactory	Factory for generating components	PinComponentFactory-Info *

A detailed description of the methods provided by this library is available online, and is also attached to the end of this document. In the following paragraphs we attempt to describe the basic functionality that you should be aware of while designing your system.

The **PinDraw** class provides a top level controller for the game. The user should create one of these as early as possible. Its constructor needs to be passed the initial argument set to scan for X11 arguments, and remove any X11 arguments it finds. When it is constructed it will cause a window to be displayed containing the board, a score region and a message region. PinDraw provides the basic functionality for managing the game. The playAudio() method plays a sound file either once or repeatedly as background music. The changeScore() method sets the score to

the specified value. The displayInfo() method displays a given message for a given amount of time.

Important Note: the PinDraw class isn't really a class. It is a defined as a pointer to a class of type PinDrawInfo. When you instantiate a PinDraw, you have to treat it as if it were it were a PinDrawInfo*, which it actually is.

To control the game, your application must define an object that is a subclass of PinCallbackInfo and pass this object to PinDraw using the setCallback() method. Then, once the game is set up, the startGame() method of PinDraw should be called. Finally, the mainLoop() method of PinDraw is invoked. Note that, like the Titanic, this method will never return. Once mainLoop() has been called, the PIN library will invoke the nextCycle() method of the user callback function every cycle (currently 100 milliseconds) and will then update the display. The callback method is called with an OR'ed combination of the keys currently pressed by the user.

A Note on Magic Numbers and Other CS32 Pinball Voodoo

Inside your callback function, nextCycle(), you need to update everything **more than once**. Otherwise your game will move at a blinding crawl. When we say "update everything" we mean checking for and reacting to collisions, and updating the ball position. The screen will be updated once for each call of nextCycle(), and to get an accurate simulation, you must update the game more often than you redraw the screen. Try something around 10 or 20, and jack it up if your game isn't moving quickly enough for your Nintendo-based upbringing. Why, back in my day... (insert vehement Colecovision defense) - but I digress.

There are a few other magic numbers you'll need to tweak. They are cycle time, gravity, and ball cycles. Call setGravity() and cyclesPerSecond() on your PinBall, and call setCycleTime() on your PinDraw object. These important numbers obviously affect the gameplay in countless subtle ways. Umm, if, however, you just want a good basis to get the #$@%!# thing working, try setGravity(90), cyclesPerSecond(1000), and setCycleTime(50).

Here is some pseudocode for the nextCycle() method. Note that this is essentially what should get done each time the method is called, but delegating the actual implementation of the pseudocode to other methods or classes would be a good idea.

```
check to see what keys are pressed
repeat 10-20 times {
      tell PinBall to update
      for each component
            if ball is intersecting with component
                  collide ball with component
                  do any necessary state transitions
}
```

To create timeouts with the library, the application needs to create a new class that is a subclass of **PinTimeOutInfo**. Calling the activate() method on this class with a given time will

start the timeout. When the elapsed time (again in milliseconds) has expired, the timeOutCallback() method of the application's class will be invoked. The timeout can be canceled at any time using the cancel() method.

Components are created using a **PinComponentFactory** object that can be obtained from the PinDraw object using the getComponentFactory() method. Components can be created by the factory either by calling the specific routines or by using the simple create() method that takes an istream as its single parameter. The format of that stream is the name of the type of object followed by the object parameters. The format of the type name and parameters (coincidentally) match that of the OBJECT, LEFT, RIGHT and MAIN commands (except for the optional standard parameters that can follow these). The create() method of the factory will return a **PinComponent** to the calling function. Note that this is a polymorphic superclass of the actual, specific subclass of the component that was created. However, since all components are used equally we can treat it as the superclass type and all will be fine.

All the **PinComponent** objects support a small set of standard methods including changeColors to change the object's colors and changeState to change its state. Components are removed by deleting them. In addition all the components support two simple intersection methods. The first, intersect(), simply tests if the ball is touching with the given object. The second, bounce() does this test first and, if the ball is hitting the object, causes the ball to be reflected off the object and adds additional force if desired. Different forces can be added for the different sides of the object the ball collides with. The state parameter of the Shooter object indicates the where the plunger on the shooter is, with 100 indicating all the way up (not pulled back by the user), and 0 indicating all the way down. The state parameter on a flipper indicates the angle (in a clockwise direction) that the flipper is offset around its pivot from its initial position.

The **PinBall** object is a special case of a PinComponent. In addition to the standard methods defined above, it allows the programmer to specify the velocity and position of the ball explicitly and to inquire about the current velocity and position. In addition, the ball object handles its own motion and the effect of gravity. To get the ball to behave properly, the application should set the value of gravity using the setGravity() method and should specify the number of cycles (updates) that occur per second using cyclesPerSecond(). [To do an accurate simulation you must compute the intermediate position of the ball and the corresponding game actions much more frequently than you update the graphics — at least a factor of ten, possibly more if the ball is moving rapidly. The cyclesPerSecond() method takes a float argument that specifies this ratio.] Then, for each subcycle, the application should call the PinBall's update() method to move it to its new position and update its velocity due to gravity. [After the update it might be wise to test if the ball has collided with any of the other objects.]

Finally, the **PinKey** enumerated type defines a bunch of values which each correlate to a particular key the user can press. Examples of this are PIN_KEY_LEFT_FLIPPER or PIN_KEY_QUIT. They also have values defined for them, which you will notice are in hex. These values specify powers of 2. So, if you think of each value in binary, each value has a 1 in one digit of the number. The PinKeySet variable you are passed in will be the bitwise-OR of whatever keys are pressed at that moment. This is called a bitfield. So, to determine if a given key was pressed, you should do a bitwise-AND of the PinKeySet value and the key you wish to check.

Project Requirements

Your pinball program should take the name of a pinball game file from the command line. The program should set up and then play the given game. The setup stage involves setting up key commands with the defineKeyMap() method, parsing the specified file, exiting gracefully if there are any parse errors, and constructing objects based on what you read in. When parsing the file, you should make use of the PinComponentFactory class, which is capable of instantiating a PinComponent directly from an istream, using the create() method. As you parse, you should keep track of the states, actions, and state transitions for each object. All objects should start in state -1. In addition, you can use from-state -1 to mean a "don't care" state (i.e. if the specified trigger event takes place, you should follow the given state transition, regardless of which state the particular object is in.) Be careful only to have one state change per trigger event per object specified... if you don't, you'll be working with a non-deterministic state machine! (But for extra credit you could always consult Savage, Models of Computation: Exploring the Power of Computing, pp. 156, on converting a NFSM to a DFSM. [That's a joke. Please don't do that.])

You should handle the basic set functionality that levels can specify. This means you will have to create your own subclass of the PinCallback class and send it to your instantiation of the PinDraw (top-level) object, using the setCallback() method. Your subclass of PinCallback must override the nextCycle(), which takes as its input a bitfield representing the set of keys that the user currently has pressed. This is where you'll check to see if any state transitions should be taken, and if so, carry out the appropriate action(s), such as writing a message to the screen, updating the score, changing object colors, jumping up and down and shouting "Oogah", and so forth. You can subclass the PinTimeOut class and override the timeOutCallback() method to handle chores which need to go off after a specified length of time. Timeouts make pinball games more interesting and also make it easier to start a pinball game, while at the same time giving the user warning that their first ball is about to appear. The remainder of the support code found in `pin.h` should be self-explanatory; if you have any questions, however, feel free to see a TA or (*even better*) post to the CS32 newsgroup.

We'd also like you to design a game file of your own. After all, part of the fun of programming pinball should be in playing your own "dream machine." A very small portion of your project grade will be devoted to this. It is advisable to do this first, using the demo to scope it out, so that you have a good understanding of what each object is and how to write and use a finite state machine.

There are a slew of possibilities for extra credit here. (Please detail them in your README file and check with a TA before proceeding.) These could include extensions to the game logic such as counters and new arithmetic tests, or additions to the level file format, or just about anything else you can dream up. What you can accomplish is bounded only by your creativity.

You can locate the support files for this program in /course/cs032/support/pinball. There, you will find the PIN Library header file, a (very useful) makefile, and one or two test files that you can use to get your software up and running.

Handing In

To hand in your UML design, type
`/course/cs032/bin/cs032_handin design2`
To hand in your code and header files, your README file, and your executable, type
`/course/cs032/bin/cs032_handin pinball`
at the promt. You must be in the directory that contains all files in order for this to work. Handin your program electronically by 11:59pm on February 26, 1998. You have until the next day at 5pm to turn in a hardcopy of your code and README file. Please staple your README file on top, and turn your entire packet into the CS32 bin, near the TA Room.

Parting Wisdom

In the immortal words of St. Bernard of Clairvaux, “Arouse yourself, gird your loins, put aside idleness, grasp the nettle, and do some hard work.”

Tho’ much is taken, much abides; and tho’
We are not now that strength which in old days
Moved earth and heaven, that which we are, we are,--
One equal temper of heroic hearts,
Made weak by time and fate, but strong in will
To strive, to seek, to find, and not to yield.
- Tennyson