6.170 Laboratory in Software Engineering
Fall 2005
Final Project: Gizmoball
Due: See Schedule

Contents:

Note: Some 6.170 students have acquired Repetitive Strain Injuries (RSI) over the course of the final project in the past. Don't let it happen to you. It hurts. Please read the MIT server about RSI before embarking on the final project.

Introduction

This handout describes one of the two final project choices you have this term, Gizmoball. For information on RSS Client, the other project, refer to handout in the projects section.

The goal of the project is to design, document, build, and test a program that plays Gizmoball. Gizmoball is a version of pinball, an arcade game in which the object is to keep a ball moving around in the game, without falling off the bottom of the playing area. The player controls a set of flippers that can bat at the ball as it falls.

The advantage of Gizmoball over a traditional pinball machine is that Gizmoball allows users to construct their own machine layout by placing gizmos (such as bumpers, flippers, and absorbers) on the playing field. These machine layouts may also form complicated "Rube Goldberg" contraptions that are intended to be watched rather than played. (If you don't know what a Rube Goldberg machine is, see http://www.anl.gov/Careers/Education/rube/ or http://www.rube-goldberg.com/). As an optional extension (after you have designed, documented, implemented, and tested all required functionality), you may create new varieties of gizmos that can be placed on a playing field.

Gizmoball Overview

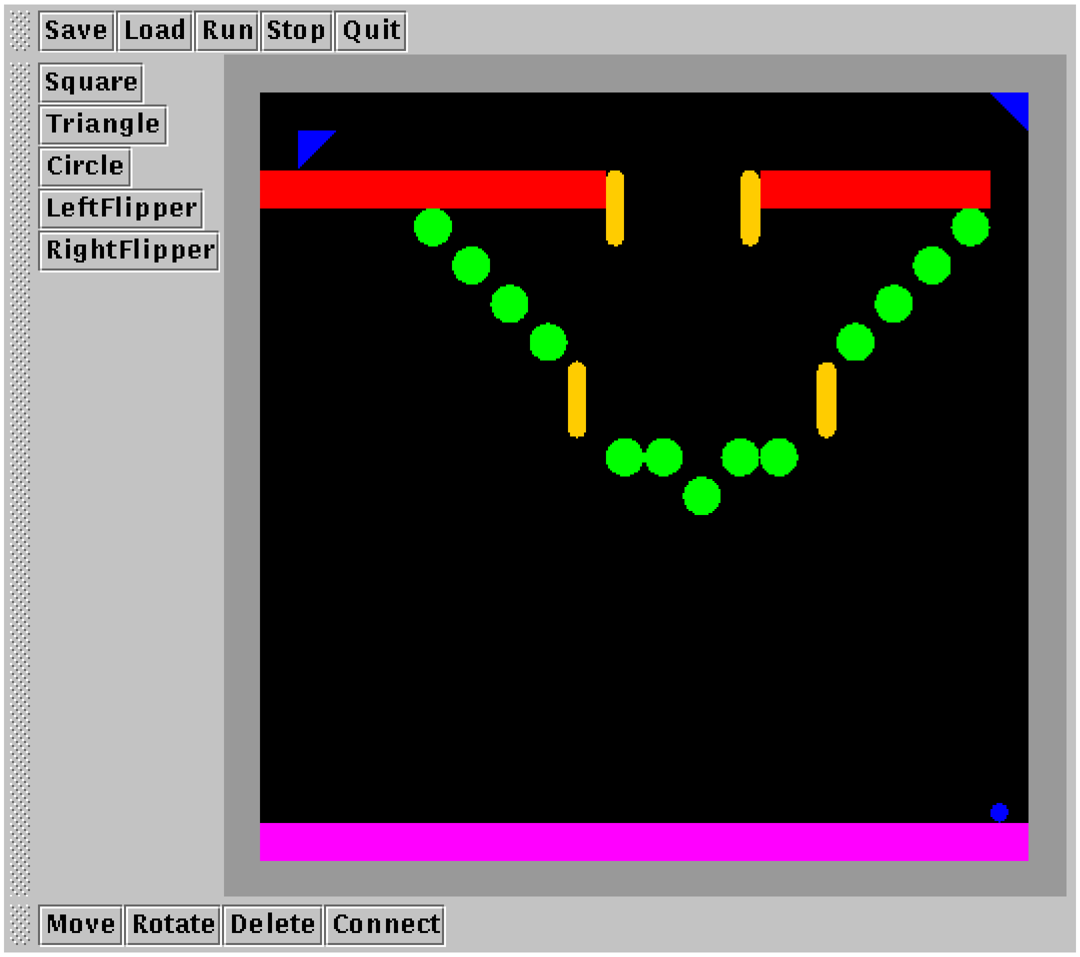
Because this project is in part a design exercise, the assignment specifies what the user should be able to do and leaves it up to you to figure out what modules and interfaces are appropriate. This section gives an overview of Gizmoball. A more detailed specification is given in Appendix 1. To enable automated testing, your implementation must support a XML file format (defined in Appendix 2), in addition to the loosely-specified graphical user interface.

Gizmoball has a graphical user interface with two modes, **building mode** and **running mode**.

In building mode, a user can:

- create and edit square, circular, and triangular **bumpers** on the playing surface,
- create and edit **flippers**,
- **connect** the action of flippers and bumpers to triggers, such as a keyboard key being hit or one of the bumpers being bumped, and
- **save and load** the user's game configuration to and from a file.

In running mode, the user can **play** the game.



A screenshot of one implementation of Gizmoball.
Your implementation will look different (depending on your choice of user interface),
and your ball motion may not match the animation given exactly.

The picture above illustrates the most important features of Gizmoball.

- The gizmo palette on the side provides the user with a variety of operations (**square, circle, triangle, flipper**) for placing gizmos in the playing area.
- The "modifications" toolbar on the bottom provides the user with a variety of operations (**move, delete, rotate**) for editing the gizmos in the playing area.
- The modifications toolbar also provides a **connect** button that connects the trigger of one gizmo to the action of another. After this button is pressed the user can connect gizmos together. For example, the user might press the connect button, then click on one of the circular bumpers, and then click on one of the flippers. As a result, every time the bumper's trigger is activated (which occurs when a ball hits the bumper), the flipper's action (to rotate around its pivot) will occur. Alternatively, the user might press the connect button, then press a key, then answer a question about whether the up or down keypress is of interest, then click on one of the flippers. As a result, every time the user depresses (or, respectively, releases) that key, the flipper will move. Several triggers may activate the same gizmo.

- The purple bar across the bottom of the playing area is the **absorber** gizmo. When a ball enters the absorber, the ball stops moving and is held in the absorber's lower right-hand corner. The absorber gizmo's action is to shoot a ball it is holding (if any) straight up in the direction of the top of the playing area. Connecting the absorber to itself allows the game to loop continually: every time a ball enters the absorber, it is immediately shot out again.
- The menu at the top of the window allows the user to **save** or **load** game configurations and to **run** or **stop** the game. In the animation, a game is in progress: the ball is the small blue circle which started in the lower right-hand corner. The ball bounces off the red, green, and blue bumpers, and is hit by the yellow flippers.

Awards

Each team may enter its Gizmoball implementation into a class contest for one or more of the following awards:

- **Best Design:** Awarded for the project with the best abstraction, modularity, extensibility, simplicity, etc. The quality of the final report is also considered.
- **Best Gizmoball Game/Usability:** Awarded for the project with the best game-play and best user interface. Part of your submission for this prize should be an input file that sets up the playing area; the prize is for the playable game itself, not for the construction kit.
- **Most Artistic:** Awarded for the project that is the most beautiful or fascinating to watch. Part of your submission for this prize should be an input file that sets up the playing area. When run with this input file, the ball or balls should bounce around forever without the user needing to press any keys.

Whether your program implements only the basic required functionality orextra gizmos etc. will **not** be considered when making the design award. However extra functionality that improves game-play or "Rube Goldberg" artistry will be an asset in competing for the other two awards.

Grading, Deliverables and Schedule

You'll do your project in phases, with the following milestones:

Phase	Deliverables	Due date
Preliminary Design	Preliminary design document	Wed. Nov. 9
Preliminary Release	Source code, specifications, unit tests	Mon. Nov. 21
Final Release	Final design document, source code, specifications, unit tests, user manual, webstart packaging	Mon. Dec. 12

Each team will receive a single grade for the final project, determined as follows:

Category	Deliverable	Due date	% project grade	Graded on
Design	Preliminary design document	Wed. Nov. 9 at 12 noon	15%	Are key issues identified?
	Final design document	Mon. Dec. 12 at 9AM	15%	Is design clean, robust and flexible?
Team work	Weekly meetings	weekly	10%	Did team work well together? Did all members participate constructively?
Packaging	User Manual	Mon. Dec. 12 at 9AM	8%	Is the tool easy to use? Is the user manual clear and helpful?
	Webstart delivery of executable	Mon. Dec. 12 at 9AM	2%	Is the client packaged correctly?
Implementation	Specifications, preliminary	Mon. Nov. 21 at 12 noon	5%	Are important interfaces identified and crucial parts well documented in Javadoc?
	Specifications, final	Mon. Dec. 12 at 9AM	5%	Are important interfaces well documented in Javadoc?
	Unit tests, preliminary	Mon. Nov. 21 at 12 noon	5%	Are there good unit tests for non-trivial classes?
	Unit tests, final	Mon. Dec. 12 at 9AM	5%	Are there good unit tests for non-trivial classes?
	Source code, preliminary	Mon. Nov. 21 at 12 noon	15%	Is the code clean and well structured? Basic functionality working?
	Source code, final	Mon. Dec. 12 at 9AM	15%	Is the code clean, well-structured and low in defects?

Here is what each of the deliverables should contain:

- **Preliminary design document**: includes basic decomposition, rationale (succinct informal narrative explaining why certain design decisions were made, and what alternatives were considered and rejected); and work allocation and milestones

(how tasks will be divided amongst team members, and the dates on which tasks are expected to be completed). You should also include sketches of what your GUI will look like (if you draw by hand, make arrangements with your TA for turning that sketch in).It should be possible to read the document linearly, so you should minimize the use of forward references, and should provide enough overview and explanatory material to make the design artifacts comprehensible. The purpose in preparing the preliminary design is to identify and explore important issues, so the document will be judged on how well it does this, rather than on the quality of the design per se.
- **Final design document**: includes basic decomposition, rationale (succinct informal narrative explaining why certain design decisions were made, and what alternatives were considered and rejected); a brief description of your strategy for testing and validation; and a post mortem (a discussion of which design decisions turned out well, and which turned out badly, and why; whether the milestones were met, and what you did when they were not). Any changes between the preliminary and final design should be noted and explained.
- **User manual**: a standard user manual, intended for users not familiar with any of the course material. The better the design of the user interface of your client, the less you will need to explain in the user manual.
- **Specifications**: every public method should have at least a minimal Javadoc specification. A careful pre-post specification should be written for any subtle or important method.
- **Unit tests**: each test should have a brief comment explaining its purpose.
- **Preliminary Source code**: will be judged by its correctness, clarity of structure, and the judicious use of runtime assertions and representation invariants.
- **Final Source Code**: will be judged by its correctness, clarity of structure, and the judicious use of runtime assertions and representation invariants. You will also need to package your application using the Java Web Start (JAWS) tool.

Some general points:

- The work you hand in for the preliminary design and release should differ from that of the final release in its state of completeness, not in its quality. It's very important to get into the habit of working methodically. If you just hack like mad, and hope to make your code clean and elegant at the end, you won't succeed.
- For the preliminary release, you will be expected to demonstrate some basic functionality. Here is the minimum that we expect:
    - Demonstrate key-press triggering of a flipper on the screen. When a key is pressed, the flipper should rotate 90 degrees; after the key is released, the flipper should rotate back to its original position. You should be able to trigger it a second or third time by pressing the key again after it has returned to the original position. (You need not demonstrate connecting the key to the flipper in build mode.)
    - Demonstrate a working absorber, ball motion, gravity, and friction. In running mode, with no bumpers or flippers on the screen and the ball sitting still in the absorber, you should be able to press a key, observe the ball shoot up out of the absorber, slow down as it rises, fall back to the absorber, and return to its original position. Also demonstrate that you can

    shoot it out a second time. (Note that you do not yet need to support configurable gravity or friction constants.)
    - Handle ball collisions with bumpers and the walls. Proper handling of ball-flipper collision is not required at this stage. During running mode, a ball shot out of the absorber must behave properly when it collides with bumpers or with the outer walls.
    - Demonstrate loading files in the standard format. Given a test file, your implementation should display the gizmos specified in that file at the specified locations on the screen. You should be able to load and display all the standard gizmos.

  Your animation in run mode should be smooth and adequate to demonstrate the features required above.

- Your TA will judge the usability and correctness of your client largely during a demo at the end of term. You will have about 15 minutes to show off your work, to be followed by about 30 minutes of questions and discussion directed by your TA.
- After the preliminary release, we will give you an **amendment**: a request for an additional feature. How easy it is to accommodate it will depend on well you have designed your client to anticipate reasonable increments of functionality. In judging your final release, the new feature will be considered one of the required features.
- All deliverables should be handed in electronically and (with the exception of the code) as hardcopy to your TA (double sided and stapled). Late handins will be heavily penalized; for the final release, because of end-of-term constraints, late handins will not be accepted.

Weekly Meetings with TA

Each team will meet with its TA once a week for an hour. You should contact your TA to schedule these meetings. To receive full participation credit:

- All team members must be present at all meetings.
- All team members must answer questions and participate in the discussion at each meeting.
- A clear progress document that is useful for productive discussions must be handed in each week at the meeting. At the first meeting a draft of the Preliminary Design will serve as the progress document.

Although the progress document must be clear, it is short and informal. This document will form the basis of discussion during the meeting, and the TA will keep it on file as a record of progress made. The team should bring multiple copies to the meeting, one for each team member and one for the TA. This progress document should include the following information:

- A description of all the new issues that have been discovered during the previous week. This includes both a list of newly discovered bugs, and a list of unresolved design issues.

- A description of all the issues that have been solved over the past week. This includes a list of bugs that were fixed, and how they were fixed, and a list of design issues that were resolved, and how they were resolved.
- A list of all the issues from previous weeks that are still unresolved.
- A plan for the next week, with specific actions and goals for each team member.
- An assessment of success at meeting the previous week's plan.
- The document may also contain any other material that you feel describes your progress, such as object model or MDD fragments showing changes to the design.

Resources

This section is full of information and links that will help you complete your final project.

Provided code

Animations in Java are quite challenging. You will use the `java.awt` and `javax.swing` packages to construct your graphical user interface (GUI). We have provided you with a demonstration program in Example.java that shows how to animate the movement of a ball bouncing around the window. It also demonstrates how to get your program to listen to user events, such as clicking on a toolbar button, pressing a key or dragging the mouse. All members of your group should be able to compile and execute this demo GUI.

We have also provided you with a library of physics routines (see Appendix 3) for calculating the dynamics of elastic collisions. You are welcome to use this code as is, or modify it in any way that you like.

Hints

General

- **Design**

  A careful design will save you a lot of time in the long run. It's well known that a small mistake made early in a project can become a big problem if it's not caught until much later. The preliminary design is a major part of the project (more so than its proportion of the grade might indicate). Do it very carefully, trying to anticipate problems that may arise. Then the rest of your project will be more straightforward and more fun.

- **Prototype**

  One of the largest challenges for this kind of design problem is figuring out where the "gotchas" are. If you are having difficulty imagining how to structure one part of the design it sometimes helps to build a small prototype. Plan to throw away your prototypes. Once you've figured out how to do design something correctly, it rarely makes sense to try to retrofit a hacked up, broken version.

- **Validate early and often**

  Validation shouldn't be an afterthought! You may choose a design because its implementation will be easier to test. Make sure you validate your code as you implement.

- **Document early and often**

  Incomplete documentation is better than no documentation at all. If a potential problem or subtlety occurs to you, but you don't have time (or are unable) to formulate it properly, then just add a few sentences in your document describing the issue. Later, if you have time, you can go back and fix it.

- **Don't overdocument**

  Don't include any redundant material. For example, there's no need to explain the difference between black-box and glass-box testing. Just indicate which of your test cases fall in each category. Similarly, being rigorous is not the same as belaboring the obvious. You can assume that your TA knows what a set or a stack is. There's no need to explain something from scratch when you can use standard terms and notions.

- **Have fun being on a team**

  Enjoy being part of a team. Run new ideas past your partners, and discuss problems with them. Read and discuss each others code. A good way to find a bug is to ask someone else to look at your code. Start early!

- **Communicate effectively**

  Each meeting you hold with your team members (or your TA) should have

  - an agenda. Don't get together unless you know the reason. This will help you avoid wasting time.
  - a designated leader to facilitate the meeting. The leader ensures that the meeting stays on track, encourages all group members to participate, and helps to resolve problems.
  - a secretary who takes notes and distributes them to the remainder of the group afterward. These notes highlight the important decisions made, issues resolved (and not resolved), etc. They ensure that all decisions are agreed upon by everyone and that everyone is aware of the issues raised at the meeting.

  The roles should rotate among the group members; in 6.170, no one individual should perform any of the roles disproportionately often.

- **Prioritize**

  We had fun putting together this project. Our goal was to provide you with a project that is both very challenging and offers many opportunities for you to be creative. We encourage you to experiment. Make your implementation of Gizmoball as beautiful to watch, and as fun to play as possible. That said, **make sure you get the basic functionality working** before you add bells and whistles. The best way to approach extensions to the project is to make your initial design flexible and extensible.

Coding

You should acquire background knowledge about Swing before attempting to code your GUI. You can see Sun's Swing tutorial (particularly the quick tour).

Do not try to use the realtime clock in order to determine timing information. Instead, arrange to receive a timer event every 1/framesPerSecond and proceed to do the simulation and screen updates in response to this event. If you get behind and time slows down, so be it. A simple way to set this up is do use the `javax.swing.Timer` class, as in the example GUI. Using this approach will simplify the implementation of your code and will also avoid the need to deal with synchronization issues in a multi-threaded program.

If you are using Swing and wish to paint your own component, as you will need to do in order to actually draw the board, gizmos, and ball, you should extend `javax.swing.JComponent` and implement your own paint routines. In order to do this you will need to override the `paint` method of your `JComponent` to paint the board. The painting is done by calling methods on the supplied `java.awt.Graphics` object. Unless you explicitly turn it off, Swing components are automatically double-buffered to reduce flicker. If you do not understand this, do not worry about it. In addition to `Graphics` Java has an alternative graphics context `java.awt.Graphics2D` which provides more sophisticated capabilities than the traditional `Graphics` object. Note that the calls your components receive to `paint(Graphics)` will always have a `Graphics2D` passed as the argument, so if you want to work with `Graphics2D`, you may simply cast the `Graphics` object. You may implement Gizmoball using either style of graphics, but here are some differences which you might want to consider:

- The `Graphics` object works in terms of integer values for pixels allowing you to more directly control which pixels are updated.
- `Graphics2D`, on the other hand, accepts floating point values to define geometric shapes to be rendered and performs the rasterization itself. This is somewhat more automatic, but also makes it more difficult to directly set individual pixels.
- The `Graphics2D` class also allows `AffineTransform`s to be applied to it. (An affine transform is a geometric transform that preserves parallel lines.)

In order to respond to mouse and keyboard actions from the user you will want to create and install `MouseListener`, `MouseMotionListener`, and `KeyListener` all of which can be found in the `java.awt.event` package. Information about Java keycodes can be found in the documentation for `java.awt.event.KeyEvent`.

Keypress

The specifications for handling keyboard input in Gizmoball require that an object connected to a key is triggered when that key is pressed *or* released. This provides behavior similar to that of a real pinball game: hitting the button causes the flipper to swing upward and releasing the button causes the flipper to return to its rest position.

Keyboard events

The Java specifications for `java.awt.event.KeyEvent` describe three types of key events, `KEY_PRESSED`, `KEY_TYPED`, and `KEY_RELEASED`. The documentation suggests that `KEY_PRESSED` events occur when a key is actually depressed by the user and `KEY_RELEASED` events occur when the key is released. It would therefore seem reasonable to trigger when receiving a `KEY_PRESSED` or `KEY_RELEASED` event for a given key bound to a gizmo.

Unfortunately, most Java runtime environments fire multiple `KEY_PRESSED` and in some cases multiple `KEY_RELEASED` events when the user has only pressed the key once. Additionally, in some environments you may never receive the `KEY_RELEASED` events for an upstroke. This is because the behavior of `KEY_PRESSED` and `KEY_RELEASED` is system dependent. The behavior occurs through an interaction with the operating system's handling of key repeats that occur when you hold down a key for a period of time.

On Windows and MacOS

On Windows and MacOS, Java will produce multiple `KEY_PRESSED` events as the key is held down and only one `KEY_RELEASED` when the key is actually released. For example, holding down the 'A' key will generate these events:

```
    PRESSED 'A'
    PRESSED 'A'
    ...
    RELEASED 'A'
```

On Unix

On Unix, multiple pairs of `KEY_PRESSED` and `KEY_RELEASED` are received as the key is held down:

```
PRESSED 'A'
RELEASED 'A'
PRESSED 'A'
RELEASED 'A'
...
PRESSED 'A'
RELEASED 'A'
```

Test Program

If you want to explore the behavior of your in your environment, you can use the `KeypressTest` class provided by the staff. The application will dump all keyboard events to the console for inspection.

The source code is available at `KeypressTest.java`.

Solutions

You should feel free to handle this nuance of the Java API as you see fit. One easy solution is to shut off the operating system's automatic key press repeat mechanism and thereby cause the `KEY_PRESSED` and `KEY_RELEASED` events to more closely correspond to the actual actions of the user.

- **Unix/Linux**: Type "`xset -r`" to shut off autorepeat. To re-enable autorepeat use "`xset r`"
- **Windows**: Go to the Control Panel's `Accessibility Options` applet. On the `Keyboard` tab select the `Settings...` button for `FilterKeys`. Select `Ignore quick keystrokes and slow down the repear rate`. Select the `Settings...` button next to that option. Make sure `No keyboard repeat` is selected. Slide the `SlowKeys` slider to `Short` (0.00). Press OK twice. Check `Use FilterKeys` and press OK. To enable and disable these changes, simply check or uncheck the `UseFilterKeys` checkbox.
- **MacOS**: Go to System Preferences and select `Keyboard and Mouse`. Select `Keyboard`. Drag the `Delay Until Repeat` slider to the `Off` position.

Asking the end user to perform settings such as these is acceptable, but should be included in your Gizmoball documentation.

An alternative solution is to take advantage of a special key listener decorator provided by the staff. The class is available in compiled form in the `gizmo.jar` file as **`staffui.MagicKeyListener`**. Refer to the documentation for MagicKeyListener or use the provided source code as your own starting point.

Appendix 1: Detailed Requirements

General

Your implementation must support two modes of execution: *building* and *running*. In building mode, the user can add gizmos to the playing area and can modify the existing ones. In running mode, a ball moves around the playing area and interacts with the gizmos.

Playing Area

To describe dimensions in the playing area, we define L be the basic distance unit, equal to the edge length of a square bumper. Corresponding to standard usage in the graphics community, the origin is in the UPPER left-hand corner with coordinates increasing to the right and DOWN.

The playing area must be at least 20 L wide by 20 L high. That is, 400 square bumpers could be placed on the playing area without overlapping. The upper left corner is (0,0) and the lower right corner is (20,20). When we say a gizmo is at a particular location, that means that the gizmo's origin is at that location. The origin of each of the standard gizmos is the upper left-hand corner of its bounding box, so the location furthest from the origin at which a gizmo may be placed is (19,19) on a 20L x 20L board. The origin of a ball is at its center.

During building mode, Gizmos should "snap" to a 1 L by 1 L grid. That is, a user may only place gizmos at locations (0,0), (0,1), (0,2), and so on.

During running mode the animation grid may be no coarser than 0.05 L by 0.05 L. Suppose that the ball is at (1,1) and is moving in the (1,0) direction -- that is, left to right -- at a rate of .05L per frame redraw. Then the ball should be displayed at least in positions (1,1), (1.05,1), (1.10,1), and can be displayed at more positions if you wish the animation to be smoother. Rotating flippers can be animated somewhat more coarsely; see the precise description of flippers below. If the ball is moving faster than the animation grid size per frame redraw, it need not be redrawn in each animation grid position.

Building Mode

In building mode the user can:

- Add any of the available types of gizmos to the playing area.
  - An attempt to place a gizmo in such a way that it overlaps a previously placed gizmo or the boundary of the playing area should be rejected (i.e., it should have no effect).
- Move a gizmo from one place to another on the playing area.
  - An attempt to place a gizmo in such a way that it overlaps a previously placed gizmo or the boundary of the playing area should be rejected (i.e., it should have no effect).
- Apply a 90 degree clockwise rotation to any gizmo.
  - Rotation has no effect on gizmos with rotational symmetry. For example, circular bumpers look and act the same, no matter how many times they have been rotated by 90 degrees.
- Connect a particular gizmo's trigger to a particular gizmo's action.
  - The standard gizmos produce a *trigger* when hit by the ball, and exhibit at most one *action* (for example, moving a flipper, shooting the ball out of an absorber, or changing the color of a bumper). The trigger that a gizmo produces can be connected to the actions of many gizmos. Likewise, a gizmo's action can be activated by many triggers. The required triggers and actions for the basic gizmos are described below.
  - Note that triggers do not "chain". That is, when A is connected to B and B is connected to C, a ball hitting A should only cause the action of B to be triggered.
- Connect a key-press trigger to the action of a gizmo.
  - Each keyboard key generates a unique trigger when pressed. As with gizmo-generated triggers, key-press triggers can also be connected to the actions of many gizmos.
- Delete a gizmo from the playing area.
- Add a ball to the playing area.
  - The user should be able to specify a position and velocity.

    - An attempt to place the ball in such a way that it overlaps a previously placed gizmo or the boundary of the playing area should be rejected (i.e., it should have no effect). There is one exception in the standard gizmo set: a stationary ball may be placed inside an absorber.
- Save to a file named by the user.
    - You must be able to save to a file in the standard format given in Appendix 2. You may, if you wish, define an extension to the standard format that handles special features of your implementation. If you do so, the user must have the choice of saving in the standard format or in your special format.
    - The saved file must include information about all the gizmos currently in the playing area, all of the connections between triggers and actions, and the current position and velocity of the ball.
- Load from a file named by the user. You must be able to load a game saved in the standard format.
- Switch to running mode.
- Quit the application.

Running Mode

In running mode, the user can:

- Press keys, thereby generating triggers that may be connected to the actions of gizmos.
- Switch to building mode at any time.
    - If the user requests to switch to building mode while a flipper is in motion, it is acceptable to delay switching until the flipper has reached the end of its trajectory.
    - Similar short delays in order to finish transitional states of gizmos you create are also acceptable.
- Quit the application.

In running mode, Gizmoball should:

- Provide visually smooth animation of the motion of the ball.
    - The ball by default must have a diameter of approximately 0.5L.
    - Ball velocities must range at least from 0.01 L/sec to 200 L/sec and can cover a larger range if you wish. 0 L/sec (stationary) must also be supported.
    - An acceptable frame rate should be used to generate a smooth animation. We have found that 20 frames per second tends to work well across a reasonably wide range of platforms.
- Provide intuitively reasonable interactions between the ball and the gizmos in the playing area. That is, the ball should bounce in the direction and with the resulting velocity that you would expect it to bounce in a physical pinball game.
- Continually modify the velocity of the ball to account for the effects of gravity.
    - You should support the standard gravity value of 25 L/sec^2, which resembles a pinball game with a slightly tilted playing surface.
- Continually modify the velocity of the ball to account for the effects of friction.

- You should model friction by scaling the velocity of the ball using the frictional constants *mu* and mu_2. For sufficiently small *delta_t*'s you can model friction as $V_{new} = V_{old} * (1 - mu * delta\_t - mu_2 * |V_{old}| * delta\_t)$.
- The default value of *mu* should be 0.025 per second.
- The default value of mu_2 should be 0.025 per L.

Standard Gizmos

There are seven standard gizmos that must be supported: bumpers (square, circular, and triangular), flippers (left and right), absorbers, and outer walls.

A coefficient of reflection of 1.0 means that the energy of the ball leaving the bumper is equal to the energy with which it hit the bumper, but the ball is traveling in a different direction. As an extension, you may support bumpers with coefficients above or below 1.0 as well.

Square Bumper

A square shape with edge length 1L
Trigger: generated whenever the ball hits it
Action: none required
Coefficient of reflection: 1.0

Circular Bumper

A circular shape with diameter 1L
Trigger: generated whenever the ball hits it
Action: none required
Coefficient of reflection: 1.0

Triangular Bumper

A right-triangular shape with sides of length 1L and hypotenuse of length Sqrt(2)L
Trigger: generated whenever the ball hits it
Action: none required
Coefficient of reflection: 1.0

Flipper

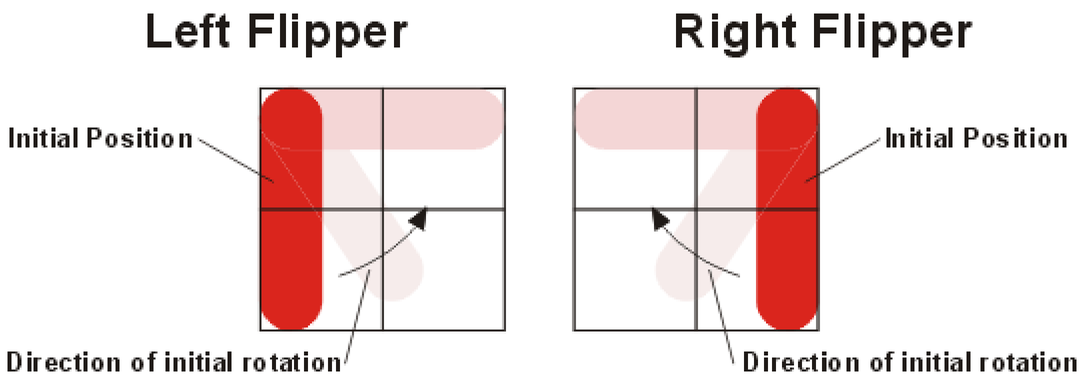
A generally rectangular rotating shape with bounding box of size 2Lx2L
Trigger: generated whenever the ball hits it
Action: rotates 90 degrees (see below)
Coefficient of reflection: 0.95 (but see below)

Flippers are required to come in two different varieties, left flippers and right flippers. A left flipper begins its rotation in a counter-clockwise and a right flipper begins its rotation in a clockwise direction.

During run mode, a flipper should never extend outside its bounding box. In edit mode the flipper should not be permitted to be placed in any way which would cause the flipper to extend outside of its bounding box during run mode, or would cause the flipper's bounding box to overlap with (the bounding box of) another gizmo.

The below pictures show flipper placements for various initial rotations. In run-mode, when a flipper is first triggered, it sweeps 90° in the direction indicated by the arrows. If triggered again, the flipper sweeps back 90° to the initial position.

In the pictures, the shape and design of the flippers are for illustrative purpose only -- your final design may differ.



Flipper initial placements and initial directions of rotation.

As with the three standard bumpers, a flipper generates a *trigger* whenever the ball hits it.

When a flipper's action is triggered, the flipper rotates at a constant angular velocity of 1080 degrees per second to a position 90 degrees away from its starting position. When its action is triggered a second time, the flipper rotates back to its original position at an angular velocity of 1080 degrees per second.

If its action is triggered while the flipper is rotating, the exact behavior is at your discretion. Here are some suggestions, but you are not limited to these options:

1. Ignore triggers while the flipper is in motion. This behavior may be undesirable for the user because a single press and release of a key might not cause the flipper to return to its original position.
2. Wait until the flipper finishes rotating (and responding to any previously-received triggers) before responding to the action. This behavior may be undesirable for the user because several quick keypresses in a row could cause the flipper to flip repeatedly for a long period of time.
3. Queue at most one trigger during the initial forward motion and have no queue during the return motion. With this model, a keypress which generated two triggers would cause the flipper to flip and return, but quick repeated keypresses would not tie up the flipper for a long time.
4. Respond to all triggers immediately. If a flipper is in a forward motion and is triggered, it will immediately switch to a backward motion. In this way, flippers with a key up and down as triggers will behave most like flippers in a real-world pinball game.

The standard coefficient of reflection for a flipper is 0.95. However, when computing the behavior of a ball bouncing off the flipper, you must account for the linear velocity of the part of the flipper that contacts the ball; therefore the ball may leave the flipper with a higher energy than it had when it reached it.

Absorber

A rectangle with integral-length sides
Trigger: generated whenever the ball hits it
Action: shoots out a stored ball (see below)
Coefficient of reflection: not applicable; the ball is captured

When a ball hits an absorber, the absorber stops the ball and holds it (unmoving) in the bottom right-hand corner of the absorber. The ball's center is .25L from the bottom of the absorber and .25L from the right side of the absorber.

If the absorber is holding a ball, then the *action* of an absorber, when it is triggered, is to shoot the ball straight upwards in the direction of the top of the playing area. By default, the initial velocity of the ball should be 50L/sec. (With the default gravity and the default values for friction, the value of 50L/sec gives the ball enough energy to lightly collide with the top wall, if the bottom of the absorber is at y=20L.) If the absorber is not holding the ball, or if the previously ejected ball has not yet left the absorber, then the absorber takes no action when it receives a trigger signal.

Absorbers cannot be rotated.

Outer Walls

Impermeable barriers surrounding the playfield.
Trigger: generated whenever the ball hits it
Action: none required
Coefficient of reflection: 1.0

A Gizmoball game supports *exactly one* set of outer walls. The user cannot move, delete, or rotate the outer walls. The outer walls lie just outside the playing area:

- There is one horizontal wall just above the y=0L coordinate.
- There is one horizontal wall just below the y=20L coordinate.
- There is one vertical wall just to the left of the x=0L coordinate.
- There is one vertical wall just to the right of the x=20L coordinate.

It is *not* required that the user be able to use the GUI to connect the trigger produced by the outer walls with any of the other gizmos. However, the standard file format *does* support this kind of connection.

Appendix 2: The Gizmoball File Format

Informal Description

Game files will be stored in a text file format known as XML which stands for *eXtensible Markup Language*. XML has a well-defined, treelike structure; thus, it is straightforward to check if an XML document is well-formed (as compared to an arbitary tab- or space-delimited file format). Because of XML's popularity, there are numerous parsers out there that convert the plain text of an XML file into usable objects. You can use the Xerces parser to read in the various xml files and create Java objects.

But before Xerces can create these Java objects, it makes sure that the file it is reading in validates against an XML Schema. An *XML Schema* is a file written in XML that defines the desired format for other XML files.

We provide you with an XML Schema gb_level.xsd that defines the text format for a level of gizmoball that your application must be able to load and save to. Any XML file that does not validate against the schema should be rejected by your application, and an appropriate error message should be displayed to the user.

If you are new to XML, then you may first want to read the w3schools tutorial on XML. The API for Xerces is located at http://xml.apache.org/xerces2-j/javadocs/api/index.html, but a good example of the parser in action is available here.

The following is an example of a very simple gizmoball level file:

```
<board>
 <ball name="Ball" x="1.8" y="4.5" xVelocity="-3.4" yVelocity="-2.3" />
 <gizmos>
  <squareBumper   name="Square" x="0"  y="2" />
  <circleBumper   name="Circle" x="4"  y="3" />
  <triangleBumper name="Tri"    x="1"  y="1"  orientation="270" />
  <leftFlipper    name="FlipL"  x="10" y="7"  orientation="0" />
  <rightFlipper   name="FlipR"  x="12" y="7"  orientation="0" />
  <absorber       name="Abs"    x="0"  y="19" width="20" height="1" />
 </gizmos>
 <connections>
  <connect sourceGizmo="Square" targetGizmo="FlipL" />
  <keyConnect key="32" keyDirection="up" targetGizmo="Abs" />
 </connections>
</board>
```

The `ball` tag specifies the initial position and velocity of the ball. Because the ball can be at intermediate points within a particular square, the coordinates are specified as floating point numbers. For example:

```
 <ball name="Ball" x="1.8" y="4.5" xVelocity="-3.4" yVelocity="-2.3" />
```

places a ball with name Ball, center at (1.8,4.5), and an initial velocity of 3.4L per second to the left and 2.3L per second upward.

Each gizmo has a `name` and a location (`x` and `y` coordinates) where it will be placed. The `triangleBumper` and the `flippers` all require an `orientation`. This `orientation` can be "0", "90", "180", or "270" degrees, and refers to the clockwise rotation of the gizmo.

Triggers can be connected to actions with the `connect` tag. In the example above, FlipL's action will be triggered whenever the ball hits the bumper named Square.

The `keyConnect` tag specifies that the action of a gizmo is associated with a particular key being pressed or released. For example:

```
        <keyConnect key="32" keyDirection="up" targetGizmo="Abs" />
```
specifies that the gizmo named "Abs" should be activated whenever the space bar key is released ("32" is the decimal number that represents a space in ascii). Type `man ascii` and scroll down to the "Decimal" section to view all the mappings from decimal numbers to ascii characters).

Because you might also want to allow the outer walls to trigger various actions, the special identifier "OuterWalls" is reserved for it:

```
        <connect sourceGizmo="OuterWalls" targetGizmo="GIZ" />
```
This command would cause the ball hitting any of the outer walls trigger the action of the gizmo named by "GIZ".

The main `board` tag can optionally take arguments for gravity and friction. If the board was described with:

```
        <board gravity="16.0" friction1="0.0" friction2="0.0">
```
the gravity in the game would be reduced to only 16L/sec^2 and all effects of friction would be removed.

Here are the contents of the gizmoball file for the example shown at the beginning of this document. It specifies a triangular bumper in the upper right-hand corner, a bunch of circular and square bumpers, and a few flippers. The actions of the upper flippers are triggered by the "space" (ascii 32) key, the actions of the lower flippers are triggered by the "q" (ascii 81) and "w" (ascii 87) keys, and also by hitting some of the circular bumpers. The action of the absorber is triggered both by the "delete" key (ascii 127) and *also by the absorber itself!* This allows the game to run continuously. Every time the ball hits the absorber, the absorber immediately shoots the ball back upwards again.

```
<board>
 <ball name="Ball" x="1.0" y="11.0" xVelocity="0.0" yVelocity="0.0" />
 <gizmos>
  <squareBumper name="S02"  x="0"  y="2" />
  <squareBumper name="S12"  x="1"  y="2" />
  <squareBumper name="S22"  x="2"  y="2" />
  <squareBumper name="S32"  x="3"  y="2" />
  <squareBumper name="S42"  x="4"  y="2" />
  <squareBumper name="S52"  x="5"  y="2" />
  <squareBumper name="S62"  x="6"  y="2" />
  <squareBumper name="S72"  x="7"  y="2" />
  <squareBumper name="S82"  x="8"  y="2" />
  <squareBumper name="S132" x="13" y="2" />
  <squareBumper name="S142" x="14" y="2" />
```

```
   <squareBumper name="S152" x="15" y="2" />
   <squareBumper name="S162" x="16" y="2" />
   <squareBumper name="S172" x="17" y="2" />
   <squareBumper name="S182" x="18" y="2" />
   <circleBumper name="C43"   x="4"  y="3" />
   <circleBumper name="C54"   x="5"  y="4" />
   <circleBumper name="C65"   x="6"  y="5" />
   <circleBumper name="C76"   x="7"  y="6" />
   <circleBumper name="C99"   x="9"  y="9" />
   <circleBumper name="C109"  x="10" y="9" />
   <circleBumper name="C1110" x="11" y="10" />
   <circleBumper name="C129"  x="12" y="9" />
   <circleBumper name="C139"  x="13" y="9" />
   <circleBumper name="C156"  x="15" y="6" />
   <circleBumper name="C165"  x="16" y="5" />
   <circleBumper name="C174"  x="17" y="4" />
   <circleBumper name="C183"  x="18" y="3" />
   <triangleBumper name="T" x="19" y="0" orientation="90" />
   <triangleBumper name="T2" x="1" y="1" orientation="0"  />
   <leftFlipper  name="LF92"  x="9"  y="2" orientation="0" />
   <rightFlipper name="RF112" x="11" y="2" orientation="0" />
   <leftFlipper  name="LF87"  x="8"  y="7" orientation="0" />
   <rightFlipper name="RF137" x="13" y="7" orientation="0" />
   <absorber name="A" x="0" y="19" width="20" height="1" />
  </gizmos>
  <connections>
   <connect sourceGizmo="C43"   targetGizmo="LF87" />
   <connect sourceGizmo="C54"   targetGizmo="LF87" />
   <connect sourceGizmo="C65"   targetGizmo="LF87" />
   <connect sourceGizmo="C76"   targetGizmo="LF87" />
   <connect sourceGizmo="C109"  targetGizmo="LF87" />
   <connect sourceGizmo="C1110" targetGizmo="LF87" />
   <connect sourceGizmo="C139"  targetGizmo="LF87" />
   <connect sourceGizmo="C99"   targetGizmo="RF137" />
   <connect sourceGizmo="C1110" targetGizmo="RF137" />
   <connect sourceGizmo="C129"  targetGizmo="RF137" />
   <connect sourceGizmo="C156"  targetGizmo="RF137" />
   <connect sourceGizmo="C165"  targetGizmo="RF137" />
   <connect sourceGizmo="C174"  targetGizmo="RF137" />
   <connect sourceGizmo="C183"  targetGizmo="RF137" />
   <connect sourceGizmo="A"     targetGizmo="A" />
   <keyConnect key="32"  keyDirection="down" targetGizmo="LF92" />
   <keyConnect key="32"  keyDirection="up"   targetGizmo="LF92" />
   <keyConnect key="32"  keyDirection="down" targetGizmo="RF112" />
   <keyConnect key="32"  keyDirection="up"   targetGizmo="RF112" />
   <keyConnect key="87"  keyDirection="down" targetGizmo="RF137" />
   <keyConnect key="87"  keyDirection="up"   targetGizmo="RF137" />
   <keyConnect key="127" keyDirection="down" targetGizmo="A" />
   <keyConnect key="81"  keyDirection="down" targetGizmo="LF87" />
   <keyConnect key="81"  keyDirection="up"   targetGizmo="LF87" />
  </connections>
</board>
```

Semantics

We now describe each element defined in the specification, in turn.

`<board gravity="FLOAT" friction1="FLOAT" friction2="FLOAT">`

Defines a board. There must be exactly one of these tags in a valid gizmoball level file. The gravity of the board is set to `gravity` L/sec^2 (default 25.0) in the downward direction. The global friction constants are set such that mu and mu2 (as described in the friction formula) are `friction1` and `friction2`, respectively (both have a default value of 0.025).

The `board` tag encloses zero or one `ball` tags, followed by a `gizmos` tag and a `connections` tag.

`<ball name="STRING" x="FLOAT" y="FLOAT" xVelocity="FLOAT" yVelocity="FLOAT" />`

Creates a ball whose center is (`x`,`y`) and whose velocity is (`xVelocity`, `yVelocity`). Within the file, the `name` must be unique, and may be used later to refer to this specific ball.

`<gizmos />`

The `gizmos` tag is just used as a container for zero or more gizmos (`squareBumper`, `circleBumper`, `triangleBumper`, `rightFlipper`, `leftFlipper`, and `absorber`). It takes no attributes.

```
<squareBumper   name="STRING" x="INTEGER" y="INTEGER" />
<circleBumper   name="STRING" x="INTEGER" y="INTEGER" />
<triangleBumper name="STRING" x="INTEGER" y="INTEGER"
orientation="0|90|180|270" />
<rightFlipper   name="STRING" x="INTEGER" y="INTEGER"
orientation="0|90|180|270" />
<leftFlipper    name="STRING" x="INTEGER" y="INTEGER"
orientation="0|90|180|270" />
```

Creates the given gizmo with its upper left-corner at (`x`,`y`) and with the given orientation. Within the file, the `name` must be unique, and may be used later to refer to this specific gizmo. The "0" orientation for each orientable gizmo is:

triangleBumper

One corner in the north-east, one corner in the north-west, and the last corner in the south-west. The diagonal goes from the south-west corner to north-east corner.

leftFlipper

pivot in north-west corner, other end in south-west corner

rightFlipper

pivot in north-east corner, other end in south-east corner

When specified, the `orientation` attributed indicates a clockwise rotation of the whole gizmo compared to its default orientation.

`<absorber name="STRING" x="INTEGER" y="INTEGER" width="INTEGER" height="INTEGER" />`

Creates an absorber with its upper left-hand corner at (`x`,`y`) that is `width` wide and `height` tall. `width` and `height` must both be greater or equal to 1 and must not cause the absorber to extend off of the board. Within the file, the `name` must be unique, and may be used later to refer to this specific absorber.

`<connections />`

A tag that has no attributes. It is just a container for zero or more `connection`s and `keyConnection`s.

`<connect sourceGizmo="STRING" targetGizmo="STRING" />`

Makes the gizmo named by `targetGizmo` a consumer of the triggers produced by the gizmo described by sourceGizmo. That is, every time a ball hits `sourceGizmo`, `targetGizmo`'s action will happen.

```
<keyConnect key="INTEGER"  keyDirection="down" targetGizmo="STRING" />
<keyConnect key="INTEGER"  keyDirection="up"   targetGizmo="STRING" />
```

Makes the item named by `targetGizmo` a consumer of the trigger produced when the key represented by `key` is pressed (or released, respectively).

The formal definition of the file format can be found in the schema gb_level.xsd. Basically, the schema defines which elements it expects to see in an XML file and notes where the format may be extended. (These extension points are denoted by either `<xs:any>` or `<anyAttribute>`.) You don't have to understand the schema unless you want to extend it to support any new Gizmoball features you've designed. If you want to extend the schema, the XML Schema Tutorial will be helpful.

Appendix 3: The physics package

The provided physics library consists of immutable abstract data types such as `Angle`, `Vect`, `LineSegment`, and `Circle`, as well as a class `Geometry` that contains static methods to model the physics of elastic collisions between balls and other circles and line segments. You are welcome to use or not use this code as you please, and to modify it to meet your needs.

Documentation for the `physics` package can be found on the MIT server.

Source for the physics library can be found on the MIT server. The jar file of our code is available in the projects section. While this source is provided in the event that you wish to examine or modify it, we strongly discourage you from modifying it. In the past, students who have not used the physics library as-is have had poor results on their projects. Most groups will *not* need to copy the source code to their own directories, add it to their CVS repositories, or compile it, but will just use the `gizmo.jar` file and examine the specifications.

Amendment

The amendment has been provided in the projects section.

Errata

Errata has been provided in the amendment sheet in the projects section.