Gizmoball

Preliminary Design Document

Austin Clements, Albert Leung, Dan Ports, Zack Skolnik

April 13, 2004

Contents

1 Requirements

1.1 Overview

Gizmoball is a standard implementation of the pinball arcade game with a few additions — the most notable, the ability to construct a user-defined machine layout by placing gizmos on the screen. A gizmo game board may also be setup for simulation, rather than for user interaction, a la Rube Goldberg. Optionally, we may decide to create new and exciting gizmos with various important features.

Gizmoball allows XML file loading and provides a graphical user interface with two modes — building mode and running mode. In building mode, a user creates and edits bumpers, creates and edits flippers, connects flippers and bumpers to respective triggers, and saves/loads current game configurations. In running mode, the game is either simulated or played.

1.2 User Interface

The User Interface is divided between two interaction modes, one for building boards, and the other for playing Gizmoball. A switch toggles between the two modes, appropriately adjusting the appearance of the interface and its behavior.

1.2.1 Playing user interface

In playing interaction mode, the interface will only respond to a limited number of menu items (load, save, mode-switch, exit) and the keys that have been connected to gizmos.

1.2.2 Editor user interface

The editing interaction mode expands the functions accessible to the user, revealing new areas of the interface that allow for modifications to the board. The load, save, mode-switch, and exit keys are still available, and the board from the play mode is frozen.

The first set of expanded features in the editing mode is a toolbar for creating gizmos. Gizmos can be dragged from the toolbar and placed onto the board.

To facilitate the transformation of gizmos, a pie menu is applied to the gizmo in focus. This method was chosen over the alternative of having tools to perform the actions — rotate, move, delete, set color, connect, etc. — because it improves access time (Fitt's law). Connections are also created using one of the options in the pie menu interface. The connecting feature supports both click-and-drag and click-and-click to indicate the two gizmos involved. This allows for greater flexibility. One possible problem was the potential confusion between connecting to a second gizmo, and selecting that gizmo, but this problem was resolved using modality of the connection creation method. This modality is indicated by an arrow from the trigger gizmo trailing the mouse pointer.

Ball velocity is also set using a pie menu option. When the option is selected, an arrow appears, with which the user can indicate direction and magnitude. The option snaps to logical values (0, vertical and horizontal angles, etc).

All gizmo information for the selected gizmo is displayed in a pane at the bottom of the screen. This will contain name, type, color, and the connections to and from the gizmo. The property list will be in a tree form, to allow the information to be better navigable. Properties can also be set through the list.

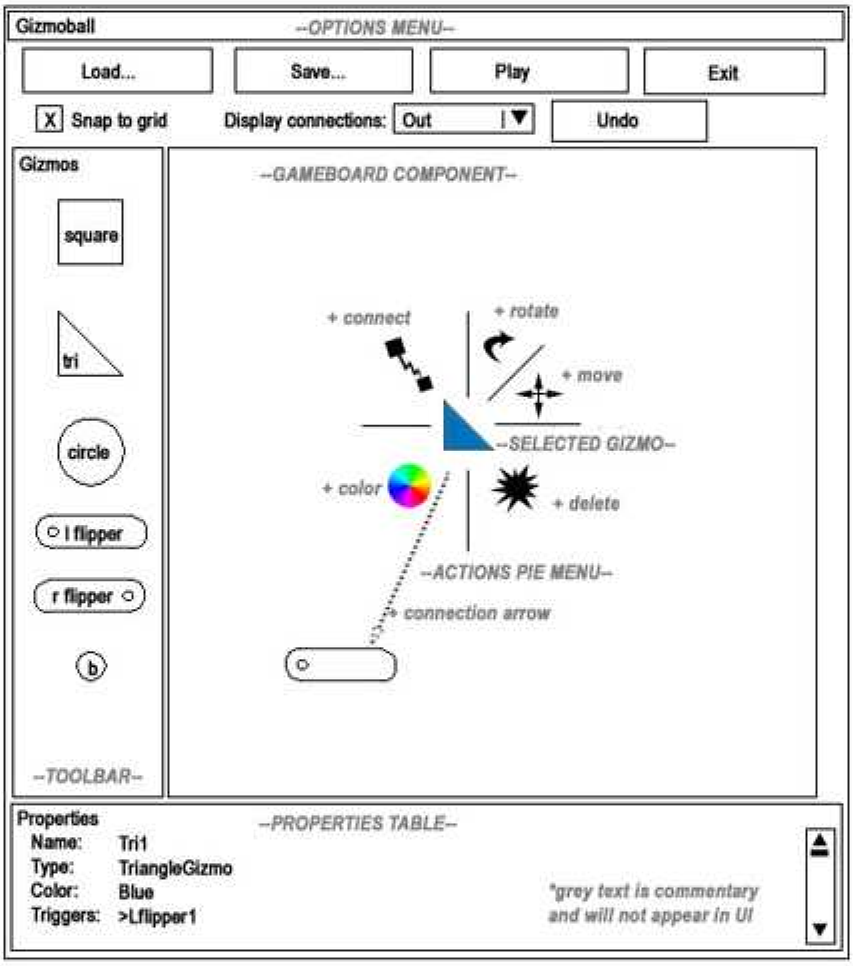


Figure 1: Editor user interface

1.2.3 Use cases

Loading a game board To load a game board, click "Load..." and navigate to the file you wish to load.

Saving a game board After editing or while playing, a game board can be saved by clicking "Save..." and typing the desired path and filename.

Adding gizmos to the game board To add a gizmo, click and drag it from the gizmo toolbar on to the desired spot on the game board. Gizmos will be X-ed to indicate that they cannot be added to the current mouse location; otherwise, releasing the mouse will place the gizmo. Gizmos may be aligned to a grid by activating the "Snap to grid" option, or they may be freely placed. Balls are also added in this way.

Transforming/deleting gizmos Clicking on a gizmo on the game board will select it. Once selected, a menu will appear around the gizmo displaying a set of options. The following options are allowed:

- rotate
- move
- delete
- set properties

To rotate or move a gizmo, select the corresponding option and drag the mouse to make the desired change. To delete a gizmo, click the delete option.

Connecting gizmos Once a gizmo has been selected, one of the displayed options is to connect the gizmo. To connect the selected gizmos trigger to another gizmo's action, click and drag the connect option to the gizmo whose action is to be triggered, and then release. An alternative method is to click the connect option on the first gizmo and then click the second gizmo. By either method, an arrow from the trigger gizmo will trail the mouse pointer to indicate that a connection is being created. Once the connection has been created, it will be indicated by an arrow (depending on the view mode) and will also appear in the property lists of the two gizmos.

Connecting gizmos to keys To connect a gizmo to a key-press, first select the gizmo. Then, locate the property in the property list that contains all the keys triggering the gizmo. Setting this property will allow the gizmo's action to be connected.

Undoing build options To undo adding, deleting, transforming, connecting, or changing properties, click the Undo button.

Switching between build and play To switch between the two modes, click the appropriate button in the menu. This button is labeled "Play" while in Build mode, and "Build" while in Play mode.

2 Design

2.1 Overview

2.1.1 Top-level design

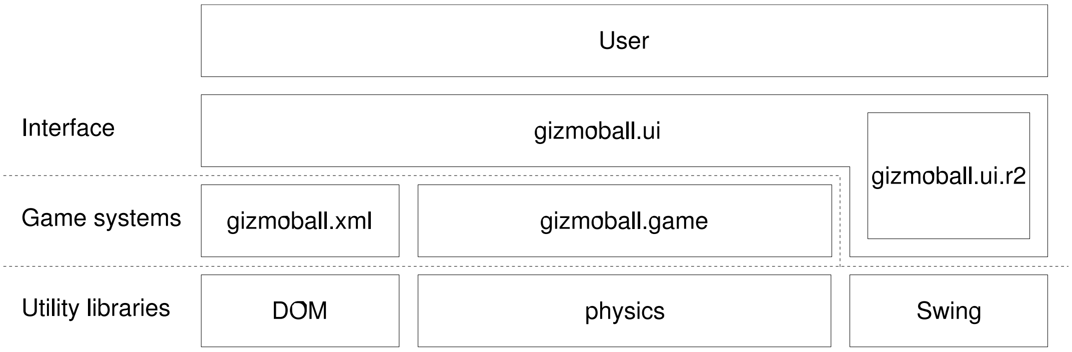


Figure 2: Gizmoball layer model

The design of Gizmoball is organized into three layers:

Utility libraries These libraries provide basic services to the Gizmoball game and interface. Both the physics library and Swing are provided libraries. Additionally, this includes the `gizmoball.xml` package, which encapsulates support for loading and saving Gizmoball board configurations in the standard XML format.

Game systems The game system consists of the `gizmoball.game` and `gizmoball.xml` packages. `gizmoball.game` lies at the core of Gizmoball and captures the game physics and the modeling of gizmos. Note that this component of Gizmoball is entirely independent of the user interface. Additionally, this is where the properties system is implemented, because it is closely coupled with the gizmos themselves. The properties system captures the properties of gizmos in an easily accessible manner. The `gizmoball.xml` package contains the game serializer, which is responsible for loading an saving game board states.

Interface The user interface is built on top of the game systems and the Swing library. It is responsible for driving the game system and presenting it visually to the user. The game system relies on the interface for timing the simulation and for information on interactions from the user (namely, key presses that result in triggers). The `gizmoball.ui` package takes care of the overall interface, but abstracts out the mechanism for actually rendering the game board. Initially, this renderer abstraction will be implemented by the $\mathbb{R}^2$ renderer in `gizmoball.ui.r2`, which will display a basic 2D game board.

2.1.2 Possible changes and unresolved issues

While some form of a gizmo property abstraction is necessary to make the editor function correctly, as it currently stands, the gizmos are an odd balance of a somewhat slow and tedious property table approach and much faster direct access to properties that is necessary for an efficient physics and display loop. Also, we are considering segregating the triggers from the properties, and abstracting the properties system out further from the gizmos. Thus, the properties system may still undergo revision.

Also, we are still considering the precise mechanism to use for collision management. Conceptually, a collision occurs *between* two objects, instead of simply one object colliding into another. However, this is difficult to express in Java while still maintaining proper modularity of collision management. One possible solution is to leverage dynamic polymorphism with a global collision manager. This adds overhead, but leverages table-driven programming for maximum flexibility. Another possibility is to simply define an ordering of gizmo types. For example, this might entail that a ball collides with an absorber, instead of the other way around, thus making only one of the two objects responsible for resolving the collision (if the two objects are of the same type, then it should not matter which one is told about the collision). However, this second approach could lead to inflexibility.

2.2 General user interface

The user interface for the Gizmoball game is implemented in the `gizmoball.ui` package. It is a generalized design that allows for multiple renderers to draw the game board state, and uses a shared interface framework with *interaction modes* for the game player and editor in order to achieve code reuse.

The root of the interface is the `gizmoball.ui.GizmoBallFrame` class. This class handles the initial bootstrapping of the system, and displays the main window. It contains a toolbar (implemented as a `JToolbar`) that handles system-level functions such as loading, saving, and switching between editing and playing modes. It also contains a `GameBoardComponent` that draws a gameboard on the screen. It also contains a `PropertyTable` that displays information about the selected gizmo, and an editor toolbar that selects gizmos to be placed.

The actual drawing of the gameboard is performed by a *renderer*. We separate the renderer from the UI package in order to make the design more general; by abstracting away the renderer, we can drop in a new renderer that provides different features. Initially, we intend to implement the $\mathbb{R}^2$ renderer (described in detail in Section 2.4), which draws a two-dimensional representation of the game board using the Swing API. If time permits, we may implement an $\mathbb{R}^3$ renderer using the Java3D APIs.

Each renderer has a `GameBoardComponent` class (e.g. `R2GameBoardComponent`) that extends the `gizmoball.ui.AbstractGameBoardComponent` abstract base class. This class handles all things related to the gameboard and its display on the screen. It contains the `GameBoard` object that represents the game board (see Section 2.3.1). It accesses the game board's state, and displays it in a renderer-specific manner.

Key presses and mouse clicks on the game board are also handled by the `GameBoardComponent`. However, we maintain only a single `GameBoardComponent`, regardless of whether the game is in the editor or player mode. This decision eliminates the need to simultaneously maintain or update state in two different game boards, and facilitates code reuse. Since Gizmoball obviously behaves very differently depending on whether the user is currently editing or playing, we use *interaction modes*, which derive from `gizmoball.ui.AbstractInteractionMode` and encapsulate how the game reacts to user input. The `GameBoardComponent` selects either the playing or editing interaction mode (for the $\mathbb{R}^2$ renderer, `R2PlayingInteractionMode` and `R2EditingInteractionMode`). The `GameBoardComponent` harnesses the power of dynamic polymorphism by passing click and keypress events to the interaction mode object, which chooses the correct action for that mode and invokes the appropriate method of the `GameBoardComponent`.

2.3 Game system

The game system module, implemented in the `gizmoball.game` package, contains the classes necessary to represent a game state and simulate it.

2.3.1 Architecture

The principal interface to this module is through the `GameBoard` class, which captures the state of a Gizmoball game board. An `GameBoard` can be constructed in an empty state, or it can be generated by a factory function in `gizmoball.xml.GizmoballReader` (Section 2.5).

Intuitively, a game board can be represented as a collection of gizmos, in addition to a few additional pieces of state information associated with the board itself. This is the representation used by the `GameBoard` class, a composite object that contains gizmos. We model the ball as a gizmo, so the ball does not need to be considered separately. Each gizmo maintains its own state information, including its position on the board, and the `GameBoard` primarily passes messages on to the appropriate gizmos.

Each gizmo is represented by its own class which inherits from the `AbstractGizmo` class. The `TranslatableGizmo` and `FixedGizmo` abstract classes derive directly from `AbstractGizmo` and express whether the gizmo can move about the screen, or is fixed in place (this distinction becomes crucial in collision handling, among other places). So, for example, we have a `BallGizmo` that extends `TranslatableGizmo`, and a `SquareBumperGizmo` that extends `FixedGizmo`.

2.3.2 Property system

The property system is a generalized framework for storing information about game entities. A property, represented by the `gizmoball.property.Property` class, is simply a name-value pair. The property can hold whatever type of value is appropriate for the specific property, including a list.

Each gizmo contains a set of properties (the implementation is contained within the `AbstractGizmo` class). Some examples of data that are stored in a gizmo's properties list include:

- the gizmo's name (a string used by the editor to identify a specific gizmo to the user)
- the gizmo's position and orientation on the board
- the gizmo's color
- for a `TranslatableGizmo`, its current velocity
- a list of incoming and outgoing trigger connections

The property system also implements the Observer design pattern. Any class implementing the `PropertyChangeObserver` interface can register to be notified when a gizmo's properties change. This is used, for example, by the editor user interface's property table in order to update itself at the appropriate time.

2.3.3 Game loop

Game simulation is performed in discrete time, with a time quantum known as a *tick*. A timer in the UI's `GameBoardComponent`, controlled by the `PlayingInteractionMode`, triggers the `GameBoard`'s `tick` method each quantum. This initiates the following procedure:

1. The `GameBoard` checks for key presses passed to it by the `GameBoardComponent`, and maps them to keypress triggers, if applicable. If such a trigger exists, the object is notified that it has been triggered. However, the trigger does not actually take effect until later in the game loop.

2. Each gizmo's `tick` method is called. This simulates one quantum of motion, including friction and gravity. Each gizmo's motion is independent of the other gizmos on the board; we do not rely on information from other gizmos. Thus, the order in which the gizmos' `tick` methods are called should not matter. Each gizmo's previous position information is also retained; this will be used in collision resolution if necessary.

3. Collisions between two objects are detected. Since a collision is an event involving a *pair* of objects rather than a single object, some arbitration is performed based on the types of the gizmos in order to determine which gizmo's `collide` method will be called.[1] (For example, a collision between two translatable gizmos ought to be handled differently than a collision between a translatable gizmo and a fixed gizmo.) The physics package is used to resolve their collision and determine the new position and velocity of each object. Each object involved in a collision is notified that it has been triggered.

4. Triggers are processed. Every object that had been notified that it was triggered, either by a keypress or a collision, performs whatever action it performs in response to triggers.

2.4 $\mathbb{R}^2$ Renderer

The $\mathbb{R}^2$ renderer[2] is Gizmoball's basic game board renderer that displays 2 dimensional representations of the game board. It provides the default implementation of the `AbstractGameBoardComponent` and `AbstractInteractionMode` abstractions.

The display mechanism works through a system of "drawers". One drawer exists for each type of gizmo and contains the methods necessary to render the respective type of gizmo to the $\mathbb{R}^2$ AWT graphics context. Because all of the drawers implement a common interface, the process of actually rendering the board is simple a matter of traversing the set of drawers. Furthermore, a factory abstraction around the production of drawers also unifies the process of creating drawers from the game board components. The `R2DrawerFactory` is responsible for taking a gizmo from a live game board and producing an instance of the appropriate `R2GizmoDrawer` subclass.

$\mathbb{R}^2$ also provides the necessary interaction modes for game playing and editing. Both interaction modes, `R2EditingInteractionMode` and `R2PlayingInteractionMode` sit atop the interaction mode abstraction provided by the general user interface. They mold the behavior of the

[1] This design has not yet been entirely finalized.

[2] We considered calling this the $\mathbb{N}^2$ renderer, because the Java Virtual Machine is technically an integer machine, incapable of representing true reals, but decided it was best to remain outside of the `double` abstraction provided by Java.

R2GameBoardComponent to the playing and editing interfaces expected by the user, as described in Section 1.2.

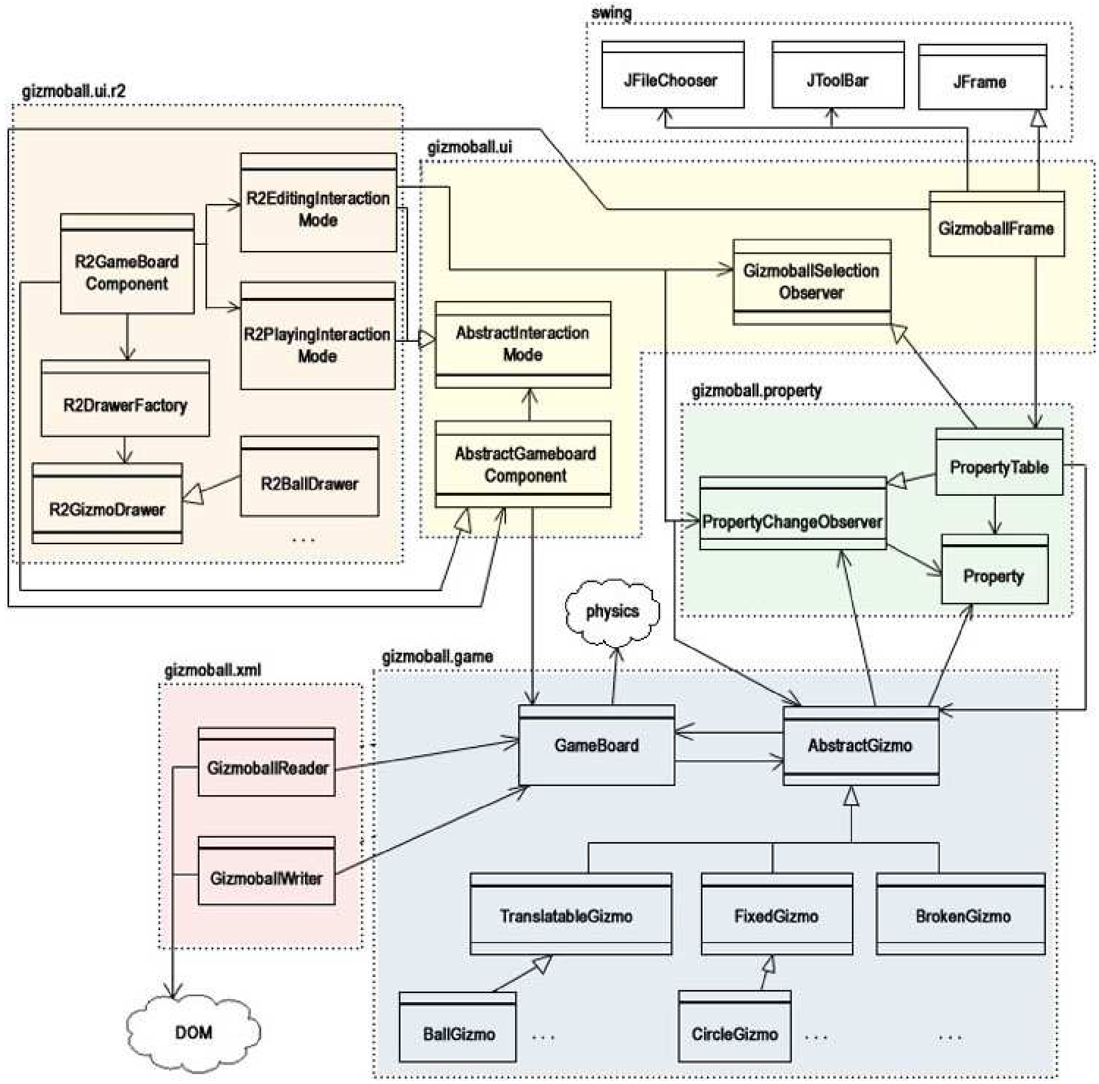
2.5 Game serializer

The game serializer, contained in the gizmoball.xml package, is responsible for loading and saving game board states in the standard Gizmoball XML format. This package itself uses Java's DOM API for reading and writing the necessary XML data.

The loading and saving of game board states is divided into two classes: GizmoballReader and GizmoballWriter. GizmoballReader is a static factory responsible for loading an existing game state. It operates by using the standard GameBoard constructor to construct an empty board, then populating it with gizmos and connections as the XML data is traversed. GizmoballWriter performs the reverse, taking in a file (which may or may not exist) and an already populated GameBoard and writing the appropriate XML data to the file. Both of these processes can be initiated by the user through the user interface.

3 Implementation Overview

3.1 Module Dependency Diagram



3.2 Project Plan

Task	Who	When
Preliminary design		**4/13**
Detailed specs for XML loader	Austin	4/16
Detailed specs for game internals	Dan	4/16
Detailed specs for UI	Albert	4/16
Detailed specs for property framework	Zack	4/16
Agree on detailed specs		4/17
XML loader	Austin	4/21
Basic UI	Albert	4/21
Game loop	Dan	4/21
Property framework	Zack	4/21
Standard gizmos	Austin	4/25
Collision mechanism [3]	Dan	4/25
$\mathbb{R}^2$ renderer/drawers [4]	Zack	4/25
Player interaction mode	Albert	4/25
Preliminary release		**4/27**
Plan and schedule amendment		4/28
Finish flippers	Austin	5/7
XML saver	Austin	5/7
Prettify $\mathbb{R}^2$ drawers	Zack	5/7
Editor display	Dan	5/7
Editor interaction mode	Albert	5/7
Property table	Zack	5/7
Finish implementing amendment		5/9
Implementation and Critique		**5/11**

[3] Modulo flippers
[4] Using just drawing primitives