Gizmoball Final Design Document

Austin Clements, Albert Leung, Dan Ports

April 27, 2004

Contents

List of Figures

1 Requirements

1.1 Overview

Gizmoball is similar to a traditional pinball arcade game with the notable addition that it gives users the ability to construct their own game layouts by placing "gizmos" on the game board. Additionally, Gizmoball can easily be extended with support for new gizmos and new looks and feels, opening up virtually unlimited possibilities.

The Gizmoball user interface is divided into two modes — playing mode and editing mode. Playing mode allows users to play the game, using the keyboard to control the gizmos. Editing mode allows users to edit their game board, as well as to create connections between gizmos that can lead to a variety of interactions, and to assign key triggers to gizmos to allow them to be triggered via the keyboard while playing. Because of the ability to edit games, Gizmoball also includes the ability to save and load game layouts in an XML file format.

1.2 Glossary

Gizmo The fundamental atomic game object. Gizmos are first-class, meaning that everything on the game board, including the ball, is a gizmo, and all gizmos are treated identically by the game system.

Squark The sound made by the collision of two gizmos.

Tick The process of updating all gizmos in a board for some time period.

Zero-knowledge system A component of Gizmoball that uses other components, but has only a weak coupling because it deduces information about the other components at run-time.[1]

Frobber A component of the editor that is displayed around the selected gizmo that allows its properties to be intuitively set. For example, the rotation frobber, when clicked, rotates the current gizmo.

Frobnicate To use the frobber menu of a gizmo.

Frobber menu A menu similar in design to a pie menu, but persistent and containing frobbers. A frobber menu has a fairly small maximum radius to make it possible to select other gizmos.

Interaction mode A method of interaction between the user and Gizmoball . Either playing or editing.

Renderer The part of the system responsible for displaying the game board to the user and mediating user interaction with the game board. These are designed to be easily replaceable if a different look and feel is desired.

Drawer A component of a renderer that is responsible for drawing some component (such as a particular gizmo type) of the game board.

[1] Almost, but not quite, entirely unlike a zero-knowledge cryptosystem

1.3 Revised Specifications

A number of minor revisions and clarifications have been made to the original specifications.

- When switching from playing mode to editing mode, all moving gizmos will freeze as they are. This includes flippers, which will stop at partial orientations if necessary. However, all gizmos will retain their state (and the editor will reflect this in the properties). One ramification of this is that the ball will have the velocity it left off with when editor mode was switched to. If no edits are made, the game should resume precisely as it left off when edit mode was switched to.
- The original specifications required gizmos to be laid out on a grid in the editor. However, this has been changed to allow the user to disable the grid if they wish to place gizmos in any valid location.
- When flippers are triggered while they are flippering, they will immediately reverse direction. This is the most like how real pinball games work, therefore it will probably be the most intuitive for users to use.
- Because of first-class gizmos, the ball is no longer a special entity in the system. It is simply another gizmo that operates by the same rules that the other gizmos do. The generality of the collision system (see Section 2.5.2) makes this possible without additional work.

The following sections cover the user interface specifications in greater detail.

1.3.1 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.

1.3.2 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.

1.3.3 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 *frobber* menu (see Glossary 1.2) 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 frobber menu interface. The connecting feature supports both click-and-drag and click-and-click to indicate the two gizmos involved. This

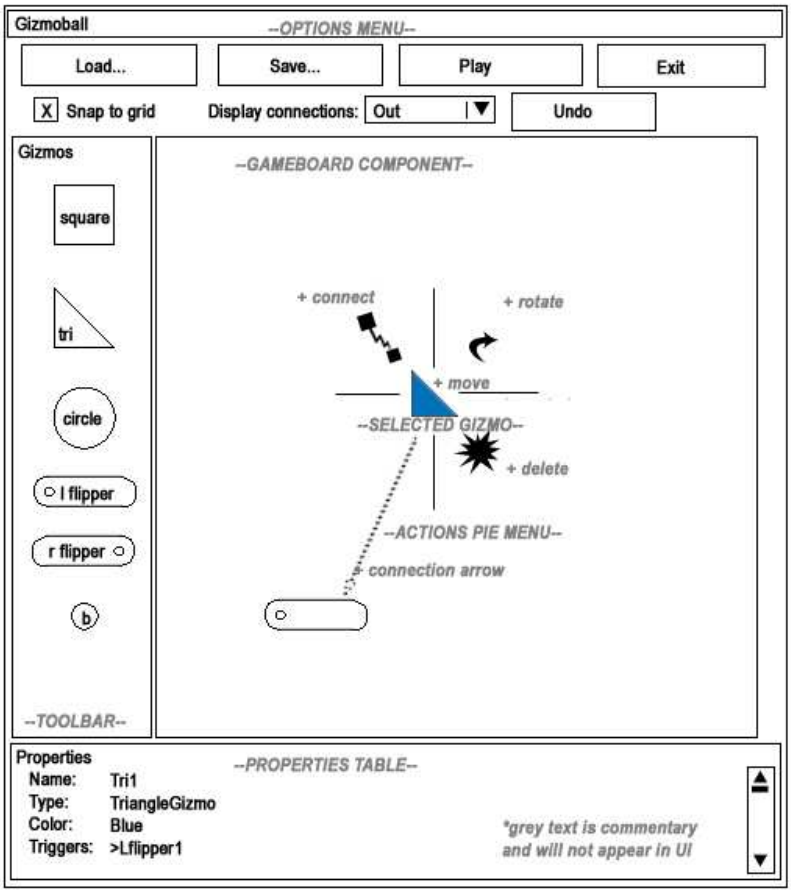

Figure 1: Editor user interface

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 frobber 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.

1.4 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 the gizmo on the gizmo toolbar. Moving the mouse over the gameboard area will reveal a gizmo trailing the pointer. Clicking the mouse again will place the gizmo in the current location. Gizmos will be X-ed to indicate that they cannot be added to the current mouse location. To cancel gizmo adding, press 'Esc' or click the gizmo over an invalid location. Gizmos may be aligned to a grid by activating the "Snap to grid" option, or they may be freely placed. Balls are also added as gizmos.

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 (see Section 2.2)

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

Transforming gizmos and setting properties (advanced) Gizmo properties can be set through the property list. Each property (name, position, orientation, color, triggers) will have a corresponding entry in the property list. Selecting the property will activate the appropriate input method (e.g. a text field for Strings, or a Color selector for colors). Input the desired value to set the property. This method allows properties not accessible through the focus menu to be set.

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 editing and playing To switch between the two modes, click the appropriate button in the menu. This button is labeled "Play" while in Editing mode, and "Edit" while in Playing mode.

2 Design

2.1 Overview

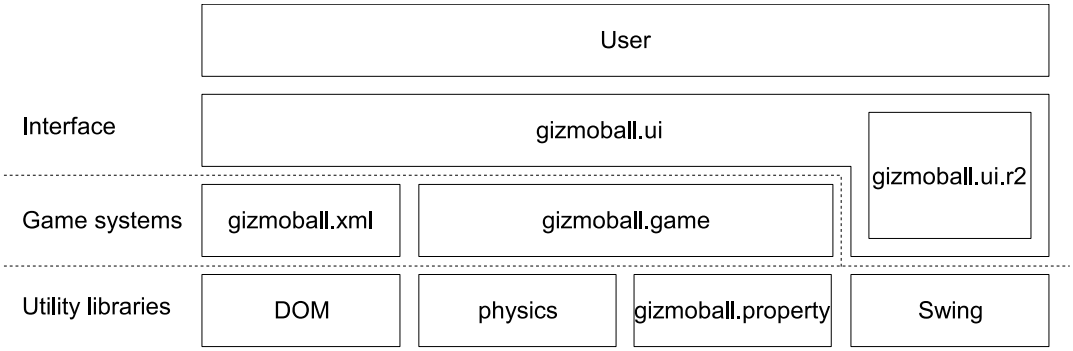

Figure 2: Gizmoball layer model

The design of Gizmoball is organized into three layers:

Utility libraries These libraries provide basic services to the Gizmoball game and interface. The physics library, Swing, and XML DOM are provided libraries. Additionally, this includes the `gizmoball.property` package (see Section 2.2), which encapsulates generic support for object property access. The property system is used throughout Gizmoball , particularly by the game system.

Game systems The game system consists of the `gizmoball.game` and `gizmoball.xml` packages. `gizmoball.game` (see Section 2.5) 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. The `gizmoball.xml` (see Section 2.6) package also resides in the game systems, and encapsulated support for loading and saving Gizmoball game board configurations in the standard XML format.

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` (see Section 2.3) package takes care of the overall interface, but abstracts out the mechanism for actually rendering the game board. Currently, this renderer abstraction is implemented by the $\mathbb{R}^2$ renderer in `gizmoball.ui.r2` (see Section 2.4), which displays a basic 2D game board.

2.1.1 Possible Changes

The editor is not yet implemented, nor is the mysterious amendment, therefore these could still affect the design of Gizmoball , however, we are confident that only minor changes will be necessary due to the generality and flexibility of our design.

We are considering revising the timing mechanism to adapt to the maximum update rate possible on a user's system by consuming idle CPU time[2], instead of the current fixed rate timer. This would only change the user interface because the game system is interval agnostic. Furthermore, not only would this let it provide smoother animation on faster computers, but it could smoothly adapt to slower computers without causing responsiveness problems.

2.2 Property System

The properties system exists to abstract out and unify the process of getting and setting "properties" of objects. It is designed to be a general support library, and as such has no dependency on any of the other Gizmoball components. However, as such, it is used throughout Gizmoball as a general mechanism to decrease coupling between the components.

The properties system leverages an approach similar to that of JavaBeans.[3] Like JavaBeans, it assumes objects will follow a particular design pattern with get and set methods for accessing their properties. It presents unified method to access the value of and set the value of any presented properties by name. The property system works through reflection, leveraging objects' existing get and set methods, thus allowing it to be combined with virtually any object without any change or penalty to code that does not need the unified access of the property system, while still incurring minimum changes to objects presenting their properties through the system. The run-time nature of the property system makes it more prone to certain types of errors that would normally be caught at compile-time, but it gives it a great deal more flexibility than pure compile-time code has.

There are three things objects are required to do in order to present their properties through this system:

- Extend `gizmoball.property.Propertyified`
- Report what properties the object supports (in the form of a list of property names)
- Call a method that in turns fires property observers when a set method is called

The property system is the base of the "zero-knowledge" systems in Gizmoball because, despite knowing nothing about the properties of objects at compile-time, it can present them for use by other systems. Thus, by leveraging the properties system, other components of Gizmoball can also be designed as zero-knowledge systems. All of the gizmos are written on top of the property system. This allows the game serializer (see Section 2.6) to get and set the properties of gizmos as they are

[2] While Machiavellian, this is standard practice for games

[3] With the advent of Sun's JavaBean technology, one can finally answer the immortal question "What does that have to do with the price of Beans in Africa?" Because the JavaBean API is certified 100% Pure Java and Java itself is portable, the price of Beans in Africa is the same as it is nearly anywhere else in the world. However, we decided that the price of Beans was nevertheless too high to warrant their use in Gizmoball, instead choosing to implement our own property framework similar in spirit to JavaBean's property capabilities, but taylored to fit our needs without the overhead incurred by the other capabilities of JavaBeans.

loaded and saved without requiring knowledge of the types of gizmos it is loading or saving. The editor uses it to deduce which frobbers to display for a gizmo, as well as to display the properties table and keep it up to date. Because these systems are completely decoupled, it is trivial to add new gizmos and change the properties that gizmos support; the remaining systems simply adapt to these changes at run-time.

We also considered using a non-reflection based property system which would have been more efficient. However, it would have required a great deal of additional support from objects implementing properties. Furthermore, it would not have fixed the only reasonable problem with the properties system, that is, the lack of compile-time checking. Because the property system is never used by Gizmoball under real-time circumstances, the decreased efficiency of the reflection based solution is acceptable.

2.3 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 two rows of option buttons: a universal set that handles system-level functions such as loading, saving, and switching between editing and playing modes; and a mode-specific set with controls that are only relevant to one mode or the other (e.g. for playing mode, a start/pause button; for editing mode, connection display options). 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 2.5.1). 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 (see Section 2.3.1). 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 poly-

morphism 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`.

2.3.1 Interaction Modes

Interaction modes provide a pluggable system of user control handlers that interface between events in the user interface and input domain and the internal game systems. The decision to adopt such a system stems from a need for different strategies of handling user interactions depending on both the current mode (playing or editing) and the current renderer ($\mathbb{R}^2$ or others). One can easily see how editing would require different functionality than playing: keypresses may be shortcuts rather than key triggers, and the gameboard component would need to handle mouse events to deal with gizmo placement and selection. Similarly, a 2d renderer would not require camera angle user controls that users of a 3d renderer would appreciate. The choice of an interaction mode system is also consistent with the overall emphasis on designing a system that is as general and expandable as possible.

Implementations of interaction modes act as the keyboard and mouse event listeners for the `GameBoardComponent`. As such, the interaction modes can handle user interactions from these input devices correctly. In addition, implementations of interaction modes have various methods that serve as relays from the GUI's components to the representation systems for the gameboard and display. This design allows the interface and the internal systems to be completely isolated; the interface and devices are unaware of underlying systems it controls, and the internal systems are unaware of buttons and input devices (or even a user).

2.4 $\mathbb{R}^2$ Renderer

The $\mathbb{R}^2$ renderer[4] 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. These drawers paint the gizmos onto the graphics context displayed by the gameboard component. Repainting of the gameboard is also controlled by the component and occurs at a rate of once per 50 milliseconds.

$\mathbb{R}^2$ 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 `R2GameBoardComponent` to the playing and editing interfaces expected by the user, as described in Section 1.3.1.

[4]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.

2.5 Game System

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

The design of the game system is extremely general, and can handle not only the specified Gizmoball behavior, but many additions of various kinds.

2.5.1 Architecture

The principal interface to this module is through the `GameBoard` class, which captures the state of a Gizmoball game board. A `GameBoard` can be constructed in an empty state, or it can be generated by a factory function in `gizmoball.xml.GizmoballReader` (Section 2.6).

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.

Each type of gizmo is represented by its own class which inherits from the `AbstractGizmo` class. A gizmo class must support three types of operations:

- simulation for a given time duration. The `GameBoard` will call the gizmo's `tick` method with the time duration.
- collision handling. A gizmo class must indicate (through the `canCollideWith` call) which other classes of gizmos it has collision handlers for. Then, for any gizmo that it has a handler for, it must be able to calculate and return the *time until next collision* with the other gizmo. Finally, it should also have a collision handler that resolves collisions with other gizmos it can collide with once they have occurred.
- triggering. The `GameBoard` handles the management and firing of triggers, but a gizmo must implement a (possibly trivial) `fireTrigger` handler that is called when it is triggered.

The gizmos are also *propertyified*, i.e. they contain properties as used in the properties system (Section 2.2). This allows properties of the gizmos to be accessed and manipulated by the editor and serializer in a standard menu. Gizmo properties include:

- the gizmo's name (a string used by the editor to identify a specific gizmo to the user)
- the gizmo's position on the board
- the gizmo's orientation, if appropriate for the gizmo type
- for a ball, its current velocity
- for an absorber, its size
- for a flipper, its handedness (whether it is a left flipper or a right flipper)

An `AbstractGizmoWithPolygonalGeometry` abstract class is used to represent gizmos that have polygonal geometry: square and triangular bumpers, absorbers, and the game walls. It provides an implementation of the collision routines that handles collisions between any of the sides of the polygon and the ball. For the absorber, we override the collision handler to capture the ball, but use the same time-until-collision method. Initially, we did not plan on using this design; however, it quickly became apparent that there was a substantial body of code that could be reused this way.

Initially we planned to use `AbstractFixedGizmo` and `AbstractTranslatableGizmo` classes that derived from `AbstractGizmo` to separate stationary and mobile gizmos. Upon further reflection[5], we concluded that this dichotomy did not provide significant benefits. This was especially true in light of the fact that there was only one translatable gizmo: the ball.

The `GameBoard` tracks the triggers in the system. Triggers are represented using a `AbstractTrigger` class that implements the triggering of the target gizmo, and `CollisionTrigger` and `KeypressTrigger` classes that capture the semantics of collision and keypress triggers. The `GameBoard` fires the triggers at the appropriate times.

2.5.2 Collision System

As hinted at above, collision detection and resolution are performed using a method in which the individual gizmo classes have collision handlers for collisions with other gizmo types. The `canCollideWith` method indicates whether a gizmo has support for colliding with another type of gizmo. It can return three responses: *can collide*, indicating that it has a handler for collisions with the other gizmo type; *defer collisions*, indicating that it has no knowledge of the other gizmo type and defers collision handling to it; or *never collides*, which indicates that the gizmo knows it will never collide with the other gizmo type (this is used for optimizing the collision recalculation process).

The `GameBoard` maintains a list of *collision pairs*: pairs of gizmos that can collide. For each pair, one gizmo is designated as the *collider* and the other as the *collidee*; the collider's collision handlers are used to detect and resolve the collision with the collidee.

Prior to the adoption of this general, distributed collision system, two alternatives were considered. First, we considered using a separate *collision manager* class that would contain methods for handling the collision of any pair of gizmos using an approach similar to generic programming. However, this would have had the disadvantage of requiring all gizmo types to be known about by this collision manager, making it more difficult to add new gizmos. We also considered using a technique similar to the operator overloading mechanism of the Python language. In this model, collision resolution would first attempt to notify the collider of the collision with the colidee. If this reported that it did not know how to resolve such a collision, a *reverse resolution* would be attempted, in which the collidee would be notified of the collision with the collider. If both of these resolution attempts failed, then it would be assumed that the two gizmos could not collide with each other (such as two fixed bumper gizmos). However, the resolution methods would have been difficult to implement, could have led to bizarre cases of infinite recursion, and may have resulted in code duplication. The current system side-steps many of these issues by distributing the collision resolution into the gizmos themselves, and by separating the process of discovering which gizmos know how to collide with each other and the process of collision resolution.

[5] Not the kind used by the properties system

2.5.3 Game Simulation

After the `GameBoard` is constructed by either the serializer or the user interface (in edit mode), it can be simulated. The user interface module regularly calls the `GameBoard`'s tick method, which performs simulation for a specified time duration. Simulation is performed according to the description and procedure in Section 2.5.3 below. Once simulation is complete, the UI module may access the new states of the gizmos.

Game simulation is performed by a *modified dynamic adaptive continuous-time discrete-event simulator*[6]. This divides a given time interval into time segments bounded by *events*. The archetypical simulator event is a collision between two gizmos. In addition, the simulator makes use of two types of events that do not have an obvious concrete meaning: an event representing the end of the current tick, and periodic events to ensure that all collision times are recalculated regularly. The simulation of each gizmo is performed in continuous time in the intervals between discrete events.

A timer in the UI triggers the `GameBoard`'s `tick` method regularly. This initiates the following procedure:

1. If the specified tick time is greater than the maximum tick length (a constant), it is divided into multiple ticks whose length is the maximum tick length or less. The remainder of the procedure assumes that the tick has been divided up.

2. An *end-of-tick* event is placed in the queue `time` seconds in the queue.

3. The earliest event in the queue is repeatedly processed, until the *end-of-tick* event is reached:

   (a) All gizmos on the board are simulated up to the event. Their `tick` method is called. If it returns true for any gizmo, indicating that the gizmo's velocity changed, that gizmo is scheduled for collision recomputation.

   - If the event represents a collision, the collider gizmo's `collide` method is called. Collision triggers for both gizmos are fired. Both the collider and collidee are scheduled for collision recomputation.
   - If the event is a *recalculate-collision-time* event, both gizmos associated with it are scheduled for collision recomputation.
   - If the event is the *end-of-tick* event, the tick ends.

   (b) Collision recomputation is performed. First, every event involving any gizmo identified for collision recomputation is removed from the queue.

   (c) Next, every pair involving any gizmo identified for collision recomputation earlier in this processing algorithm is processed:

   - If the time to collision for the pair is less than the maximum tick length, a *collision* event is placed in the queue at the time the collision will occur.
   - Otherwise, if the time to collision for the pair is greater than the maximum tick length, a *recalculate-collision-time* event is placed in the queue, the maximum tick length in the future.

4. The remaining events in the event queue are stored so that they can be reused for the next event `tick` call.

[6] Buzzword bingo!

2.6 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.

To decrease coupling, the XML reader and writer leverage reflection to find the gizmo classes and the property system to configure created gizmos and discover gizmo configurations, respectively. This way, the reader and writer need to know almost nothing about the individual gizmos, or even about which gizmos exist in the game. All of this information is simply inferred. Without this system, the reader and writer would have needed explicit knowledge of each supported type of gizmo.

Reflection is well suited to this system because the names of the gizmo classes are user-specified in the XML file, making it easy to add new gizmos to the system by simply changing the XML schema to allow them to be specified in the XML data. However, because naming conventions are enforced (as well as XML validation), this still restricts the user to valid inputs. The flippers are one exception to this rule because they are represented by a single class in the game system, but by two distinct tags in the XML format. This is an anacronism from the original specifications and may be fixed later (while retaining backwards compatibility). Currently this is handled by a special case syntax transformer which translates the incoming tags into an internal representation used for the flippers.

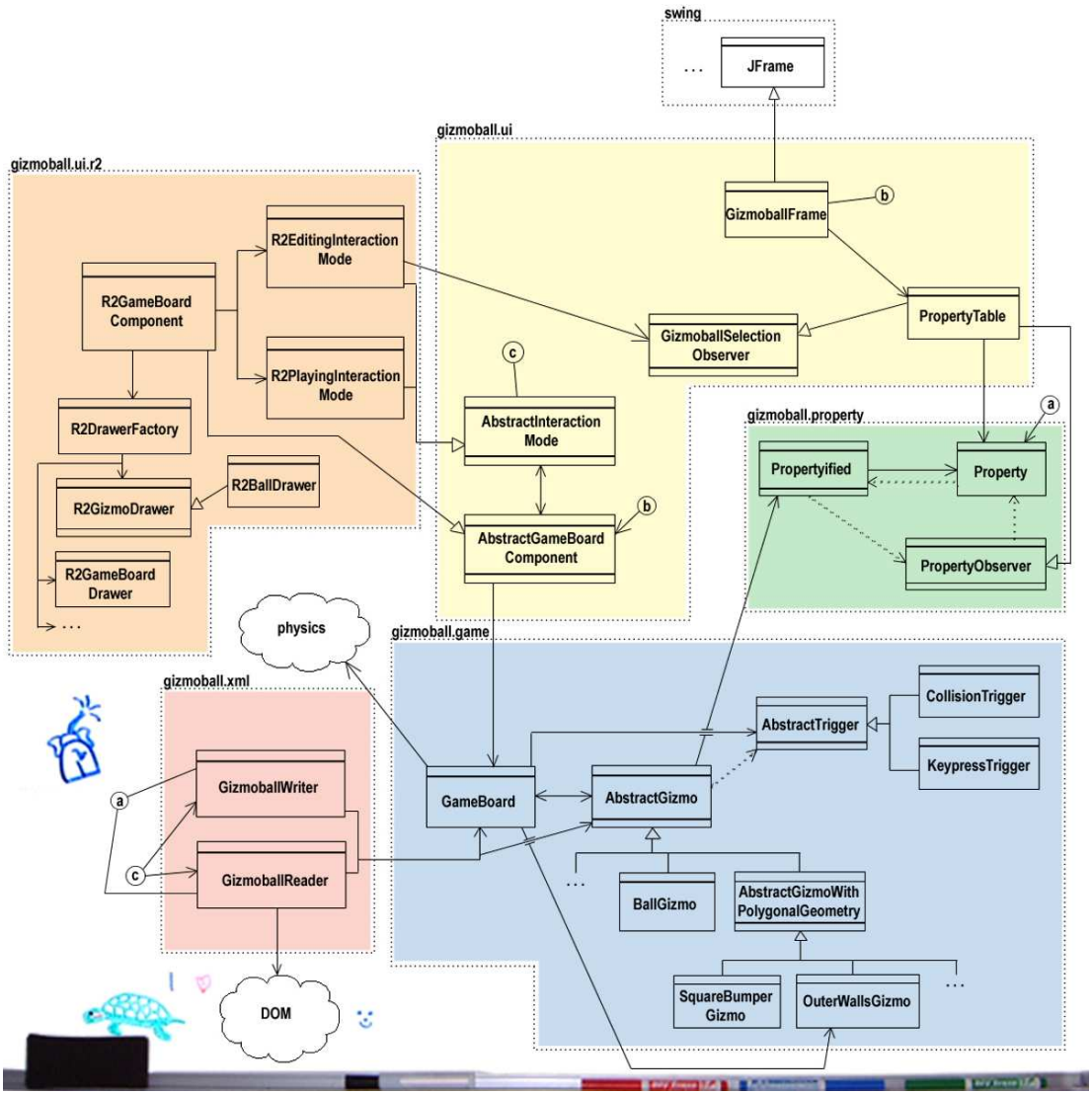

Figure 3: Module dependency diagram

3 Validation Strategy

Gizmoball is currently and will continue to be tested rigorously, comprehensively, and extensively. Because the implementation of the design is proceeding in parallel, with many different components being implemented simultaneously, a variety of different testing strategies are appropriate. Each module is unit tested as it is implemented, and integration tests are performed as modules are integrated. Automated regression tests are performed regularly as changes are made. Each of the top-down, bottom-up, and inside-out testing methodologies are used, comprising both black-box and glass-box tests.

In particular, we are careful to test all types of inputs, including those that can cause errors. Correct error behavior, including exceptions raised and error messages displayed, is verified.

3.1 Module Testing Strategies

User Interface The user interface is essentially developed top-down, starting with the top-level `GizmoballFrame` and working down to the interface with the game system module and $\mathbb{R}^2$ renderer. Hence, testing is performed in a top-down manner, using simple stub classes as necessary. Very little unit testing can be performed on the user interface, so tests are performed by hand. A test script has been written (and continues to be expanded as we implement new parts of the user interface) and is used to verify the correct operation of the user interface and the system as a whole.

$\mathbb{R}^2$ Renderer The $\mathbb{R}^2$ renderer and drawers were unit tested as well as tested with the rest of the UI. A separate application, `R2DrawerTestApp`, was created; it simply draws several gizmos. This was used to verify that gizmos could be drawn before performing integration tests with the UI.

Game System The game system was tested using stub gizmos. A simple `StubbyGizmo` that simply counted the number of times each of its methods was called was used to test adding and deleting gizmos and triggers. A more complex `ScriptedGizmo` whose responses to various queries could be scripted[7] was used for testing the simulator. This verified that simulation, including collisions, ticks, and recalculations were performed according to the specifications. The individual gizmos were integration tested with the game system framework.

Property System The properties framework was unit tested using a simple `PropertifiedClass` class that implemented properties. Its methods and the observer system were tested thoroughly.

Game Serializer The serializer was tested using a number of sample XML files that made use of all the entities that could be used in a Gizmoball XML file. Both valid and invalid XML files were used. In addition, the supplied example XML file was used to verify that the loaded configuration matched what was specified.

[7] As in the script to the play *Unit Testing: A Half-Second Play In Two Acts, Entirely Divorced of Any Physical Meaning*

4 Project Plan

Our project timeline has undergone revisions since the preliminary design. In particular, after our team suffered the loss of one of its members, a number of tasks were forced to be reassigned. Some of our design changes also forced timeline adjustments: for example, the new continuous-time game loop mechanism made the game loop and collision mechanism inseparable, so the earlier game loop milestone no longer made sense.

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	Austin	4/16	√
Agree on detailed specs		4/17	√
XML loader	Austin	4/23	√
Basic UI framework	Albert	4/23	√
Game system framework and trigger mechanism	Dan	4/23	√
Property framework	Austin	4/23	√
Player interaction mode	Albert	4/23	√
Standard gizmos [8]	Austin	4/25	√
Game loop and collision mechanism	Dan	4/25	√
$\mathbb{R}^2$ renderer/drawers [9]	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	$\overline{\text{Zack}}$ [10]	5/7	
Editor display	Dan	5/7	
Editor interaction mode	Albert	5/7	
Property table	$\overline{\text{Zack}}$	5/7	
Finish implementing amendment		5/9	
Implementation and Critique		**5/11**	

[8] Modulo flippers collisions

[9] Using just drawing primitives

[10] The complement of Zack (ie *not Zack*)