Gizmoball3D
Preliminary Release Documentation

Team: se042
David Tsai
Eric Tung
Ragu Vijaykumar

TA: Stefanie Tellex

April 29, 2003

I. Requirements

1.1 Overview

Gizmoball is an adaptation of pinball, an arcade game in which the object is to keep a ball moving around the playing space without falling off the bottom of the game. Gizmoball extends the functionality of pinball by allowing players to create their own custom playing fields, as well as specify specific triggers between specific game pieces.

They layout of the game consists of one overall graphical user interface. However, functionality of the user interface is modified on whether the user is in the editing mode of the program or the simulation mode of the program. This toggle buttons between these two modes are the play and stop buttons, where play will switch the user to the simulation mode and the stop button will switch to back to the editing mode of the game. The environment of the game is set to model physical reality, with configurable gravity vectors and coefficients of friction. These environment variables directly affect the trajectory of the ball in the playing space and hence simulate the ball in an actual physical space. Also, balls will collide with other objects in the playing space, as well as the boundaries of the playing space if boundaries are provided. Conservation of momentum is established in the playing space, with gizmos have a mass that can be considered infinite when compared to the ball.

Users may also pause the game at anytime and switch perspectives to view the action of the game from different angles. The effect of pausing the game is that all pieces of the board are “frozen” to their current location, and no further simulation is run. All games are able to be saved and loaded to a local disk.

1.2 Revised Specification

1.2.1 Pieces

There are seven default pieces that can be used in the game: Absorber3D, Ball3D, CubicalBumper3D, Flipper3D, OuterWall3D, SphericalBumper3D, and TriangularBumperA3D. These pieces are three dimensional analogs of the pieces in the original specification of Gizmoball. Flippers are able to be rotated to function as either left flippers or right flippers. The default coefficient of reflection for all pieces is 1.0, except for flippers, which is 0.95. These parameters are able to be configured by the user.

1.2.2 Playing Space

The playing space is a 3 dimensional space that represents the playing field. The playing space is able to be resized to any dimensions; however, it must remain an rectangular prism. The default size of the playing field is 20 L by 20 L by 20L, for a total of 8000 possible positions to place gizmos and balls. The origin of the space is located in the lower-right hand corner, as seen in Figure 1, so that the space operates like a real physical system. In addition, the playing space is complemented by a set of environmental parameters, namely a gravity vector and two coefficients of friction. By enabling a gravity vector, the user is able to specific not only the magnitude, but also the directionality of gravity in the 3-dimensional playing space. The default gravitational vector is -25 L/sec^2 $\hat{y}$ and the default coefficients of friction are μ_1 = 0.025 sec^{-1} and μ_2 = 0.025 L^{-1}.

1.2.2.1 Editor Mode

The program will begin by entering the edit mode for the user. In this mode, the user is able to add, delete, and configure pieces on the board. All pieces snap to the grid that is displayed. The user will also be able to configure environmental variables, such as the magnitude and direction of gravity in the playing space, and the coefficients of friction. All pieces are given wire-frames and bounding boxes that represent their range of influence in the playing space. Pieces whose ranges of influence are overlapping are displayed to the user as bright red bounding boxes. The play button is disabled until all pieces are in valid locations. The user is also able to create triggers between gizmos and key connects that perform specific actions during simulation mode from keyboard input.

1.2.2.2 Simulation Mode

Once the play button is hit, the program switches to the simulation mode and begins to activate the physical environment. All environment parameters influence the ball in it's trajectory around the playing space, and gizmos are able to trigger one another after collision with a ball. The user may interact with the simulation by key presses that trigger specific actions as configured in the edit mode. The simulation runs at 34 frames per second, but this setting is dynamically changed to give the smoothest animation during the game play. Details of how the physical parameter of all balls in the playing space are altered is given in Section 3.2. The user is able to switch perspectives in the simulation mode, as well as pause, stop, or replay the animation.

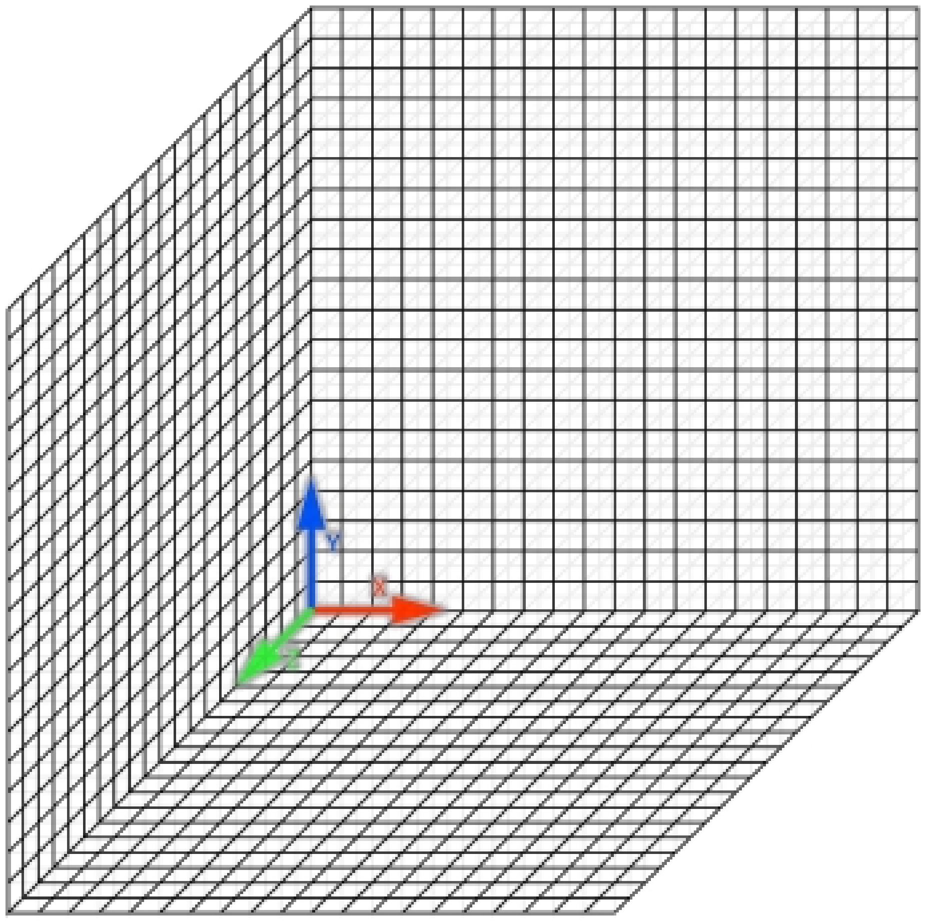

Figure 1: Playing Space and the directionality of the positive axes. The intersection of the three axes represents the inner bottom left corner of the rectangular prism.

1.2.3 Save / Load

In light of the modified specification to the playing, we still provide access to loading files of the standard 6.170 Gizmoball file format into the 3D playing space. However, playing spaces may only be saved to the g3d format, which is yet to be completely determined.

1.2.4 Extra Features

There are several optional features that have been added to the program. Mouse-overs explaining the button functionality have been added, along with visual clues as to which button is to be clicked. Perspective buttons have been added to view the playing space from a variety of different angles. In addition to the physically seen features, improvements have been made to the engine processor to provide smooth animation for up to 50 balls and 400 gizmos simultaneously.

1.3 User Manual

1.3.1 Running Gizmoball3D

Running Gizmoball3D requires two things:

- A machine that can run Java applications and more specifically, *.jar files. If you cannot run other standard Java applications, visit http:java.sun.org and download the Java Runtime Environment for your platform.
- A machine that has the Java3D API installed. If your machine does not have the API, visit http:java.sun.com/products/java-media/3D/download.html and download the appropriate Java3D API package for your platform

If these requirements are met, then you are set. Simply execute Gizmoball3D's JAR file to start the program.

1.3.2 Getting Acquainted with the Graphical User Interface

After executing the JAR file, a window will appear. This is Gizmoball 3D's graphical user interface (GUI). On the left of the GUI is the list of pieces that you can add to the game. On the bottom-left of the GUI is the gizmo properties panel. Information about the currently selected gizmo will be displayed here. On the bottom-right corner of the GUI are the camera controls. These controls can be used to help you position and navigate the 3D universe's camera and help you obtain the optimal view. The black region at the top-right of the GUI is the 3d canvas. The canvas is your window to the 3D universe and displays the playing arena. It is here that all the action will take place.

1.3.3 About the 3D Arena

Gizmoball3D is a pure 3D application that uses a full 3D physics package, a 3D engine, and of course, 3D graphics. The usual playing board for a standard Gizmoball implementation is a 20 x 20 grid. Gizmoball3D transforms the "playing field" into a full-fledged 20 x 20 x 20 "playing arena." The arena is setup in a standard Cartesian coordinate system, where +x is to the right, +y is up, and +z is coming toward the user.

When you first load the program, you will see 3 planes heading out in the +x, +y, and +z directions. Each plane, called a *grid walls*, is drawn in a 20 x 20 grid layout. A unit of space on each grid represents one unit length in the 3D arena. When navigating a 3D universe, it can be easy to lose your orientation. In order to help you navigate, the grid walls have been color-coded in bright, primary colors:

- The color of the x-y plane: Blue

- The color of the y-z plane: Red
- The color of the x-z plane: Yellow

If at any time you are confused about what are you looking at in the arena, turn on the grid walls. You should then quickly regain your orientation.

1.3.4 Camera Control

At any time, you can manipulate the virtual camera to change your views:

- “P” button: Switches camera to perspective view
- “F” button: Switches camera to front view, facing the x-y plane
- “S” button: Switches camera to side view, facing the y-z plane
- “T” button: Switches the camera to top view, facing the x-z plane
- “Lock” button: toggles between locked parallel-projection views of the F/S/T view modes and standard perspective-projection

NOTE: Incomplete Feature: “Free-roaming” camera manipulation abilities.

1.3.5 Editor Mode

Before you can play a live game of Gizmoball3D, you have to build the game first. You can do this by adding various gizmos to the arena and giving them different properties. You can also change some properties of the arena itself.

Adding a Gizmo:

On the left of Gizmoball3D’s GUI is a list of possible game pieces that you can add to the arena. These game pieces include a ball and various types of gizmos. To add a piece, click on the button of the piece you want to add, then click anywhere on the 3d arena panel. The gizmo you selected will be created at the location you clicked, along with its bounding box. The new gizmo will have the default properties of its type.

NOTE: Incomplete Feature: Right now, the gizmos just spawn at the origin.

Editing a Gizmo’s Properties via the 3D Canvas:

Now that you have a gizmo in the arena, you can move it around. Move your mouse over the gizmo you want to edit and then gizmo’s bounding box will be highlighted. Click the gizmo you want to be moved and while holding down the mouse button drag it to a new location. To help facilitate the movement of a gizmo in the arena, the following mouse-button arrangement is used:

- Left-click: Moves the gizmo in the x-y plane
- Right-click: Moves the gizmo along the z direction

NOTE: Incomplete Feature: We might change the button arrangements to be more flexible in the final release. Right now, even if you change camera views, the mouse buttons do not change their behavior. We might want to make it such that left-click isn’t hard coded into movement along the x-y plane only, but is relative to the current camera view.

Bounding Boxes and Invalid Gizmo Placement:

Every gizmo has a bounding box which is drawn in a white outline. When the mouse hovers over the gizmo, the bounding box of the gizmo will be highlighted. When you click the gizmo, the gizmo itself will be highlighted. When a gizmo’s bounding box intersects another gizmo’s bounding box, both of the gizmos’ bounding boxes will turn red. This indicates that you have made an invalid placement. To fix the problem, move one of the gizmos away. Also, if you drag the gizmo outside of the 20 x 20 x 20 playing arena, it will also highlight red. In that case, move it back into the playing arena. Gizmoball3D does not allow you to start playing a game unless there are no invalid placements.

NOTE: Incomplete Feature: Right now, a gizmo can be dragged outside of the 20 x 20 x 20 playing arena. Furthermore, a game can still be played even if there are invalid placements.

Editing a Gizmo's Properties via the Gizmo Properties Panel:

When you click on a gizmo, it will become highlighted in the arena. Furthermore, in the gizmo properties panel in the bottom-left of the GUI, the current gizmo's properties can be changed.

In here, you can add key triggers to gizmos and connect action commands between gizmos.

After making your changes, hit the "Update" button to commit the changes you made.

NOTE: Incomplete Feature: The properties panel has not been implemented yet.

To remove a gizmo, click the "Remove Gizmo" button.

Removing all Gizmos:

If you want to clear the arena of all gizmos without resetting any arena properties you might have made, click the "Remove All Gizmos" button.

Starting a new Arena:

To start a new arena, in the menu, select File -> New Arena

1.3.6 Play Mode

When you are done editing your game, press the "Play" button to start the game.
Play around with the camera controls to get the view you want.
Press the key bindings that you made in Edit mode to trigger the actions you specified.
To pause the game, press the "Pause" button.
To stop the game, press the "Stop" button. This will reset the gizmos to their previous state.
To exit back to Edit mode, press the "Back to Editor" button.

1.3.7 Loading and Saving Games

Loading a Gizmoball3D Arena in the standard 6.170 GB file format:
In the menu, select File -> Load. A dialog will pop up for you to select the file you want to load.

*Loading a Gizmoball3D Arena in the customized *.g3d file format:*
In the menu, select File -> Load *.g3d. A dialog will pop up for you to select the file you want to load.

Saving a Gizmoball3D Arena:
In the menu, select File -> Save As *.g3d. A dialog will pop up for you to save the file.

1.3.8 Other Menu Options

Drawing the Grid Walls: Draws the grid walls.
Draw the Outer Wall Gizmos: Displays the outer wall gizmos that bound the arena.
Switching between Wireframe drawing and Polygonal drawing: Toggles drawing modes.
Snapping to Grid: Toggles snap-to-grid alignment for gizmo placement.

(Incomplete Feature: Snap-to-grid has not been implemented yet. Also, we will have more menu options, such as modifying arena properties).

1.3.9 Getting Help

Upon mouse hovering over a button, a “mouse tip” will pop up and give a brief text description of the button. To turn of this feature, uncheck the box.

1.3.10 Exiting Gizmoball3D

In the menu, select File -> Exit.

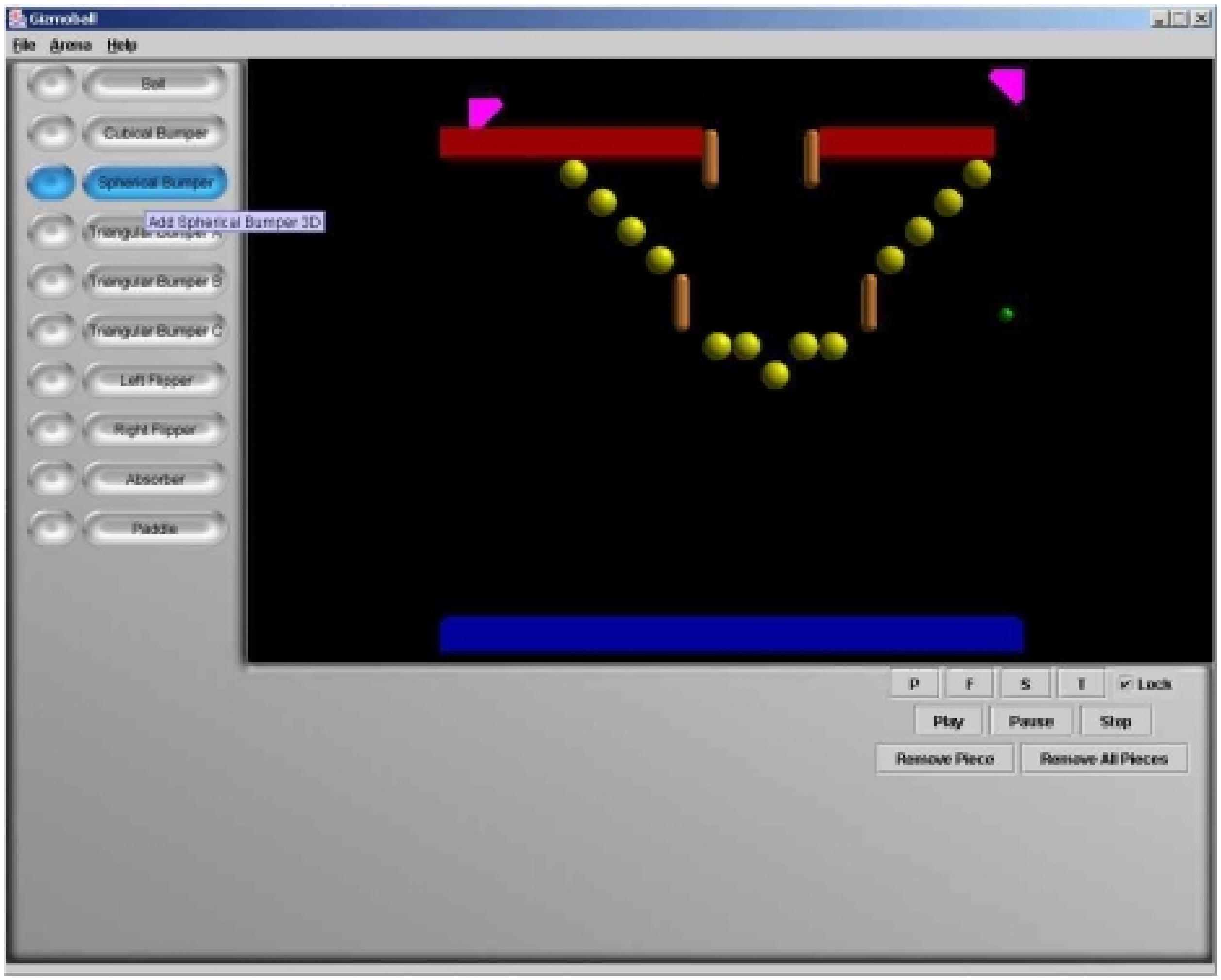

Figure 2: The graphical user interface of Gizmoball Note the highlighted button on the right with the pop up explanation as to what the functionality of the button is.

1.4 Performance

Due to the 3-dimensional simulation of our Gizmoball implementation, many of the modules are highly optimized for memory and time management. Since there are significant more possibilities for trajectories of balls and actions that happen during the simulation, more weight was given to speed optimizations than memory optimizations. These optimizations are handled by the simulation engine and are varied dynamically by the engine to give the smoothest animation possible while minimizing time and memory concerns. One key design implementation is the minimized amount of calls to the garbage collector by not creating and destroying memory-loaded objects frequently in the engine. The amount of memory consumed for m balls and n gizmos is approximately O(m+n) while the amount of time consumer is $O(m^2 + kmn)$, which k is the percentage of gizmos in a ball's trajectory. However, new algorithms are being written to reduce the time to $O(jm^2 + kmn)$, where j is the percentage of balls in a given ball's trajectory. These optimizations are dynamic to the properties of the board, and are not solely hard-coded into the simulation engine.

1.5 Problem Analysis

The problem object model shown in Figure 3 is a conceptual model of the Gizmoball application. Note that this object model was made to encapsulate Gizmoball the game, regardless of whether it is 2-dimensional or 3-dimensional. The underlying problem is still the same, with 3-dimensional labeled items being replaced by their 2-dimensional counterparts, i.e. Canvas instead of Canvas3D.

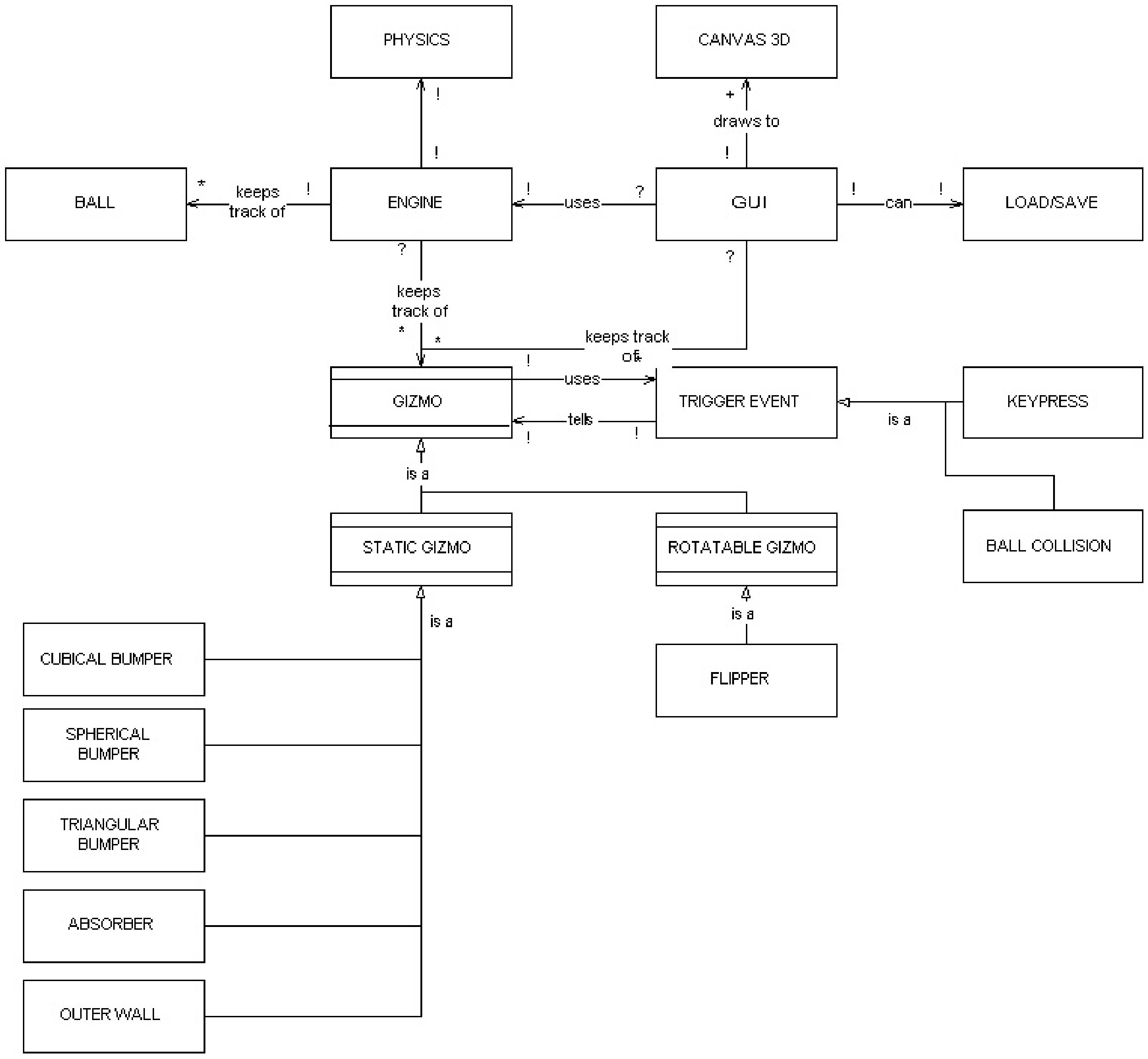

Figure 3: Problem Object Model for Gizmoball

II. Design

2.1 Overview

The Gizmoball software system is divided up into 5 separate modules: backend, engine, gui, physics, and plugins. Each module was chosen to be completed by a different team member and designed such that each could work at his own pace without having to depend on the progress of another team member. This also ensured maximum modularity and decoupling, as each team member was forced to set up only minimal connections to other modules in order to gain full functionality of the software. A brief overview of each module is given below:

- Backend – The backend package actually contains two modules and a collection of interfaces that can be used to create pieces of different types. The two modules are the event module and stream module, with all interfaces stored in the top level backend package.
  - Types: Interfaces used to create all the types of pieces that can be implemented by piece plugins.
  - Event: This module handles events that the user specifies and casts the necessary messages to all key listeners. Currently the only type of events it listens for are key events.
  - Stream: This module contains all the interfaces and classes needed to accurately load and save files of varying formats.
- Engine: This module contains the physical simulation processing engine, as well as separate drivers and pieces that can be used to run a text based simulation of the engine. The engine is responsible for simulating real-time physics in the playing space.
- Graphical User Interface (GUI): This module contains the classes used to create a user interface, as well as facilitate facile user interaction for the software. Specific actions of the graphical user interface are to properly handle mouse behaviors and keyboard input and transfer that information to the necessary parts of the software in a clean and modular fashion.
  - Backend: This module contains interfaces for proper functionality with the graphical user interface. Classes that implement interfaces in here are guaranteed to work properly with the graphical user interface.
- Physics: This module is used to simulate real-time physics in a 3-dimensional environment. Methods from this package are used directly by the engine to simulate the physical environment of the playing space. The package decouples the implementation of playing-field physics from the rest of the software system.
  - Backend: This module contains interfaces for proper functionality with the physics simulation engine. Any classes that implement the non-abstract interfaces are guaranteed to be simulated correctly by the simulation engine.
  - Exceptions: This module contains all gizmo-physics specific exceptions
- Plugins: This module contains all the actual gizmos themselves, as mentioned in Section 1.1.1, as well as all the pre-defined game-play modes that can be initialized. By having gizmos as plugins, we have inherently decoupled them from the rest of the modules since to work properly, all they must do is implement communicator interfaces of each of the modules.

To promote the modularization of our software, several design patterns were used.[1]

- Mediator: The mediator pattern facilitated keyboard interactions between the user, graphical user interface, and gizmos themselves to act in the proper manner. This allowed maximum decoupling as can be allowed with behavioral communication between multiple objects.

[1] Design Patterns are used as described in the Design Patterns: Elements of Reusable Object-Orientated Software by Gamma, Helm, Johnson, and Vlissides

- Observer: The observer pattern supported the triggering functionality for gizmos. Each gizmo potentially could be connected to every other gizmo. Upon collision with a ball, one method is called upon the gizmo (subject), and the gizmo redirects the call to all its connecting gizmos for the appropriate behavior (observers). In addition, a parameter is passed from gizmo to gizmo so that receiving gizmos are able to have information as to which gizmo triggered them and for what purpose.

- Adapter: The adapter pattern was used to wrap functionality of the physics module in the engine to the Gizmoball system. Collision-detection as well as collision triggering was wrapped by the engine to handle multiple physical component objects.

- Singleton: Due to the extremely complex nature of 3-dimensional physics, the engine was designed to employ the singleton design pattern such that only one instance of the engine may exist per runtime environment of the Gizmoball software.

- Flyweight: One of the abstractions the engine can make to the playing space is partitioning the space into rectangular prism block units. There is no correlation as to how the engine may wish to partition this space and how the graphical user interface partitions this space. Once again, due to the 3-dimensional playing space and all possible ball locations, the amount of partitions created and destroyed per time slice can have a severe performance lag on the system. Therefore, all partitions used by the engine are canonical representations of the partition space.

- Iterator: The Iterator design pattern was used within all packages to traverse collections of balls, gizmos, key bindings, collisions detections, and triggering between gizmos.

2.2 Runtime Structure

The runtime structure is illustrated below in Figures 5-10 on the following pages. Domains, relations, and constraints are all listed below:

2.2.1 Domains

- Piece: An interface specifying methods that are inherent to every piece on the board – this is an abstract interface in that it can only be used to type existing objects, but not directly implemented.
- Ball: A sub-interface of piece specifying additional methods that are applicable to generic balls.
- Gizmo: An abstract interface specifying methods that are common to all gizmos in the playing space - this is an abstract interface in that it can only be used to type existing objects, but not directly implemented.
- StaticGizmo: A sub-interface of Gizmo for non-moving gizmos
- RotatableGizmo: A sub-interface of Gizmo for rotating gizmos within their bounding box
- MobileGizmo: A sub-interface of Gizmo for translational moving gizmos within their bounding box
- PieceFactory: An interface providing methods to produce factory classes for gizmos
- PieceGraphics An interface specifying what methods must be implemented for pieces to be accurately drawn by the gui
- PiecePhysics: An interface specifying what physical methods are common to all objects in the playing field
- BallPhysics: An interface specifying the physical principles governing all balls in the playing space

- GizmoPhysics: An interface specifying the physics governing all gizmos. However since making a purely Gizmo type is not useful to the engine, this interface is declared abstract to reflect so.
- StaticGizmoPhysics: An interface designed to model gizmos that will not move with velocity during the course of the game.
- RotatableGizmoPhysics: An interface designed to model gizmos that will only move with angular velocity during the course of the game.
- MobileGizmoPhysics: An interface designed to model gizmos that will only move with translational velocity during the course of the game.
- Parsable: An interface specifying how a piece may convert its internal state into a string representing that information
- Xeon: A static engine that simulates the physical environment in the playing space
- Collisions: A class that collects and stores information related to all collisions that occur during periods between animations
- Partition: A class modeling a subset of the space represented by a rectangular prism to the engine
- GraphicArena: A GUI canvas object that represents the editor and simulation view of the game
- KeyHandler: A mediator between gui components and the gizmos themselves for setting up key board interactions

2.2.2 Relations

- Balls: list of balls in the playing space
- Gizmos: list of gizmos in the playing space
- PartitionMap: partitioning of the playing space by the engine
- Physical Components: The 3d physical objects that comprise the wire-frame of this object

2.2.3 Constraints

- No two pieces may have overlapping wire frames at any time during the simulation.
- All pieces must implement any of the concrete interfaces to function correctly. It is not enough to implement the abstract-declared interfaces.

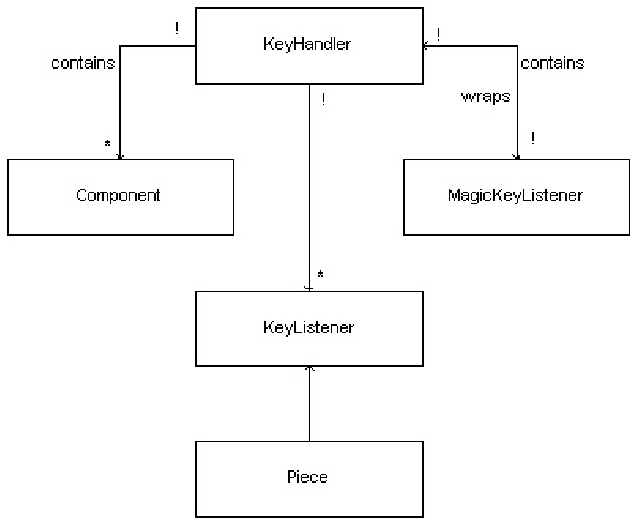

Figure 4: Event Code Object Model

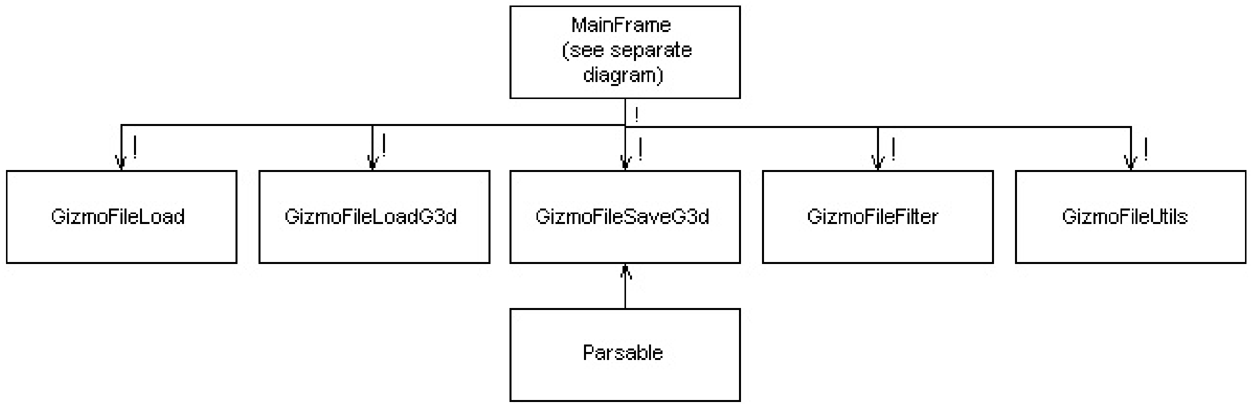

Figure 5: Stream Code Object Model

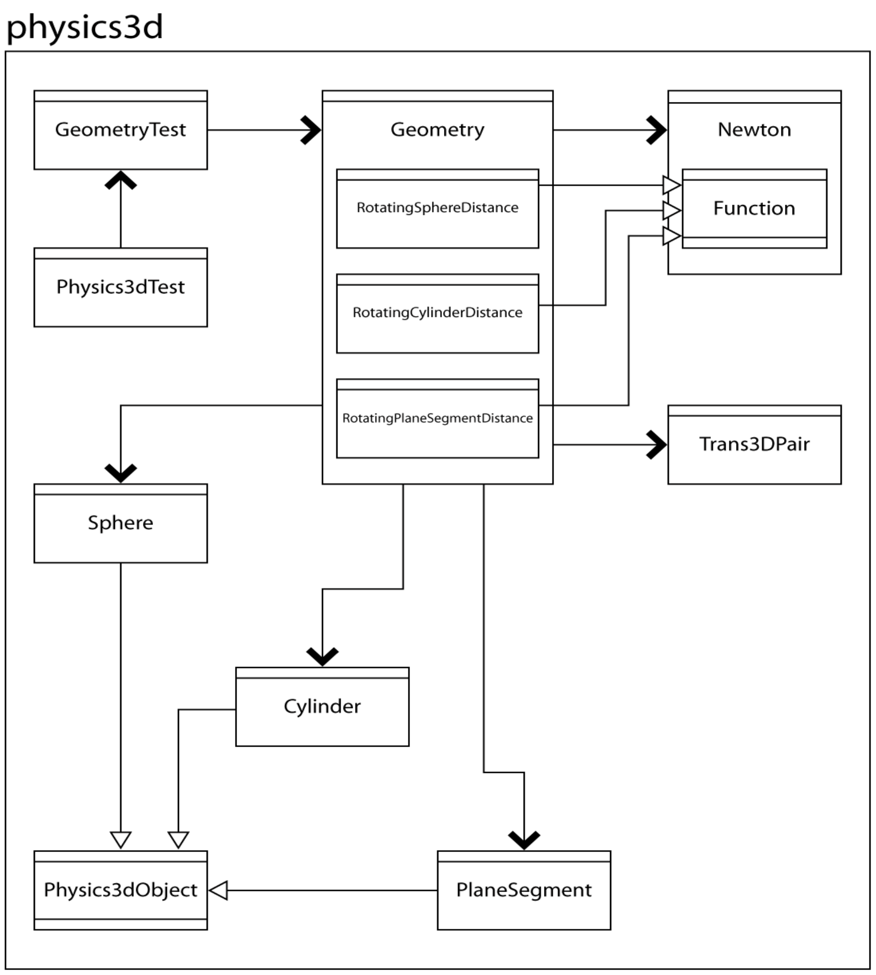

Figure 6: Physics Code Object Model

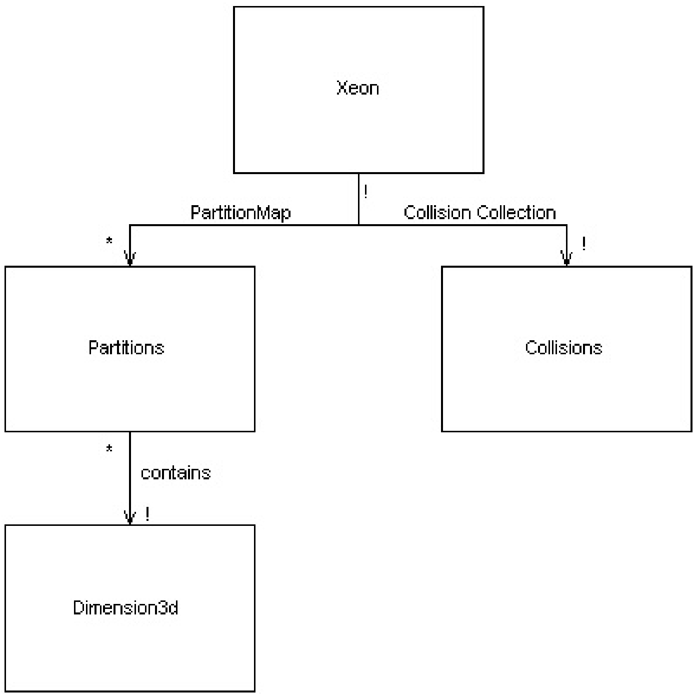

Figure 7: Engine Code Object Model

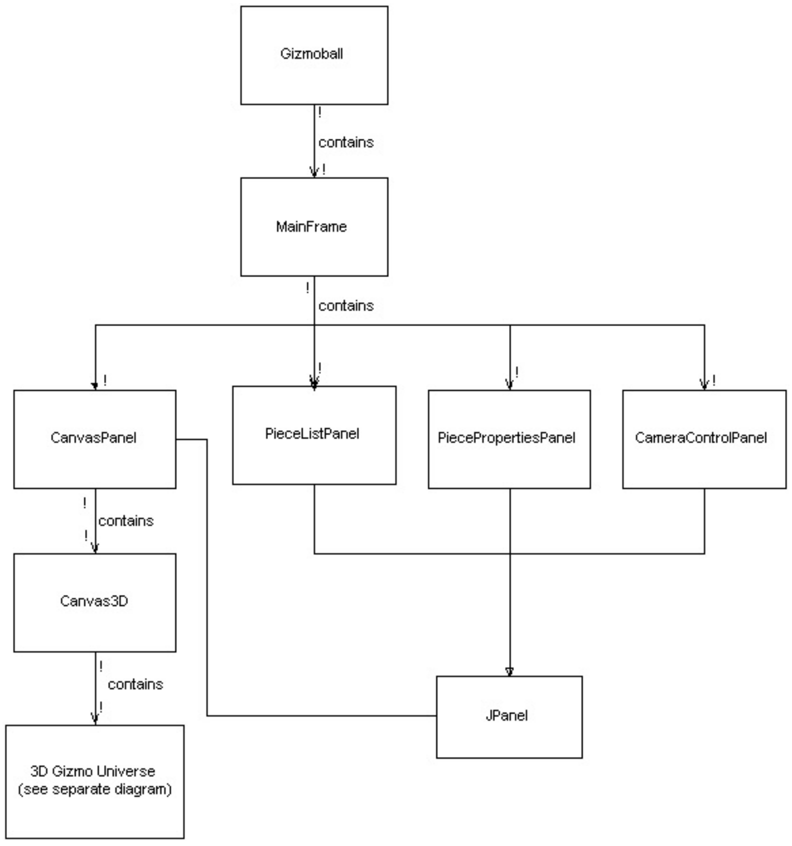

Figure 8: Graphical User Interface Code Object Model

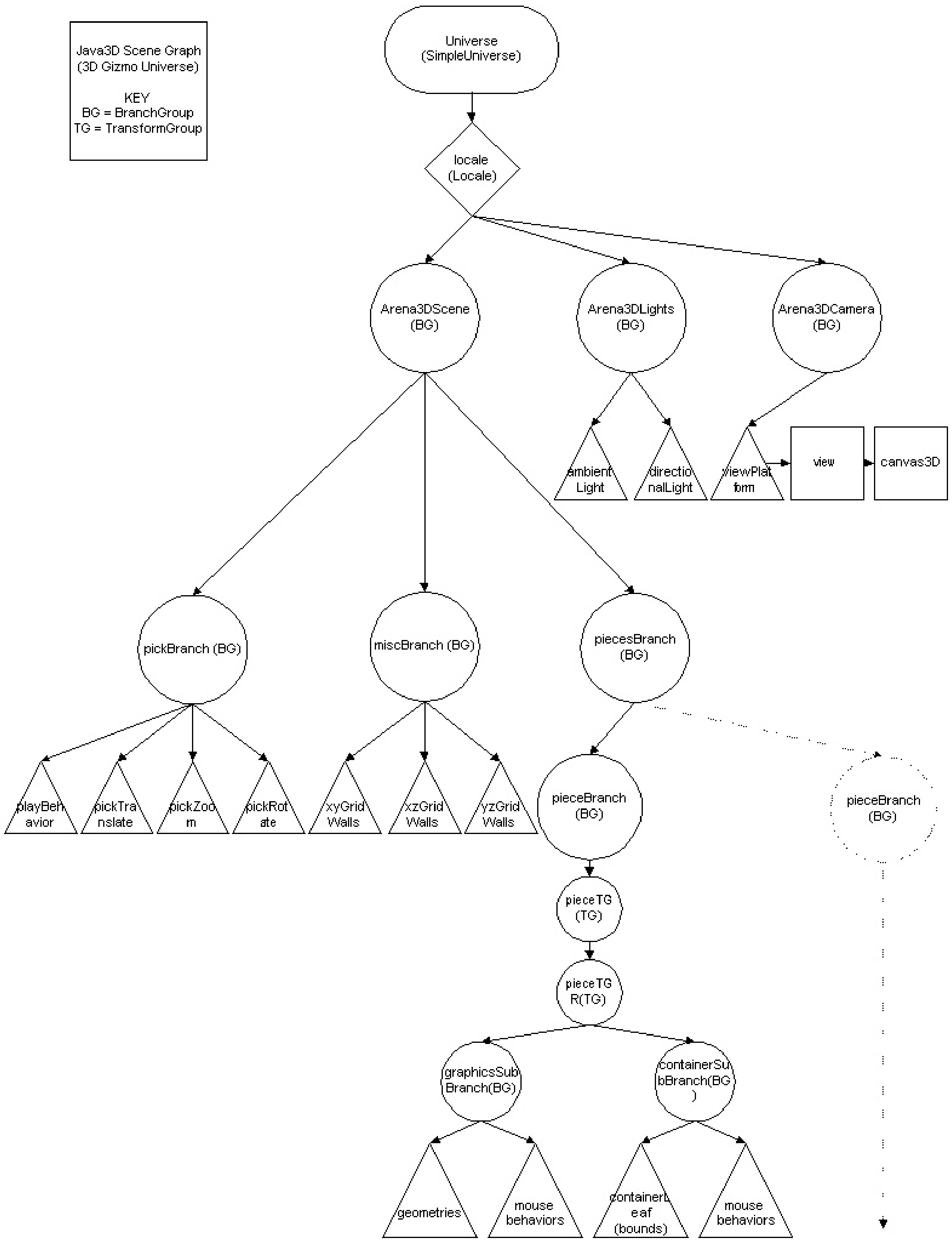

Figure 9: J3D Scene Graph Code Object Model

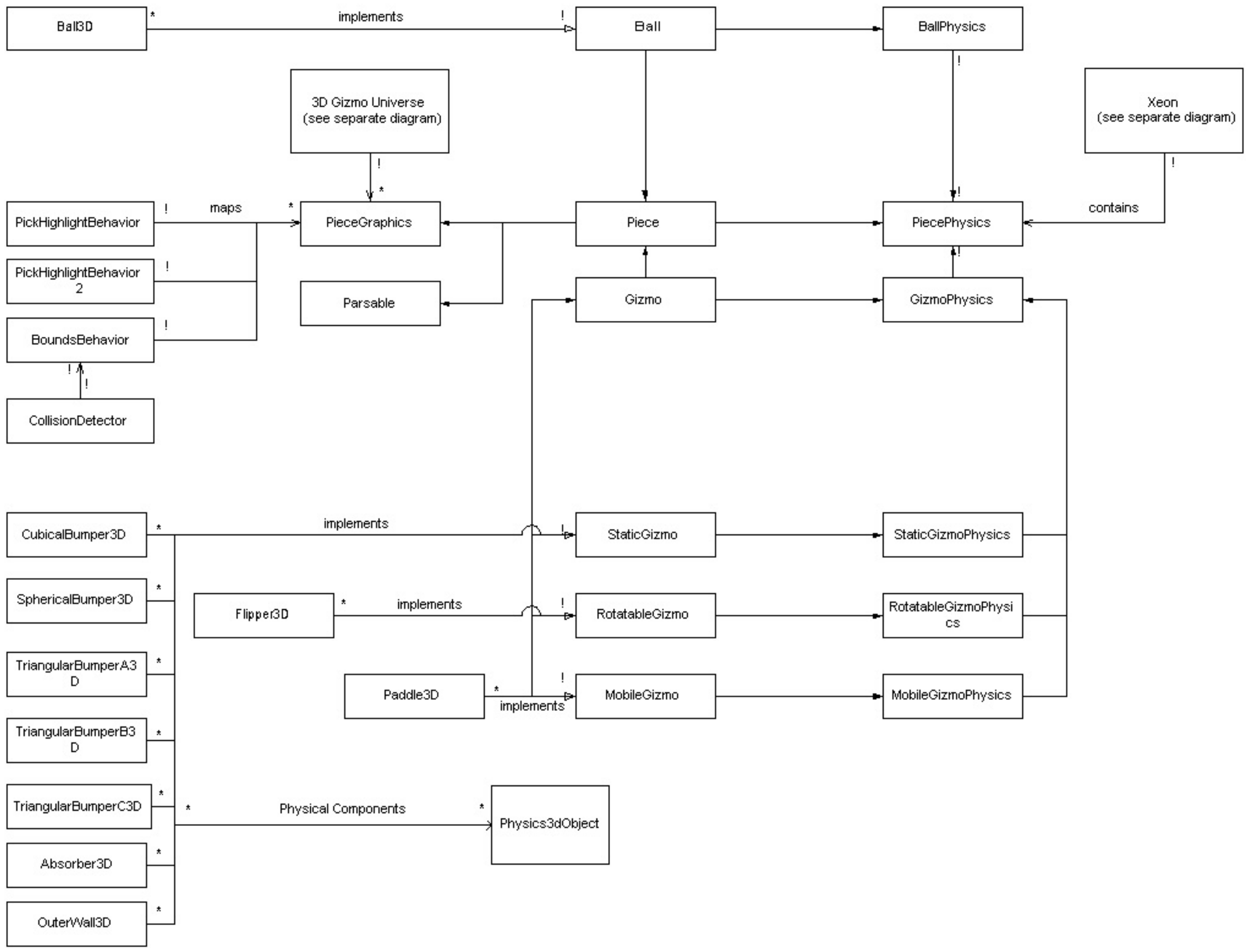

Figure 10: Plugins Code Object Model

2.3 Module Structure

The module dependency diagram is illustrated below in Figure 11.

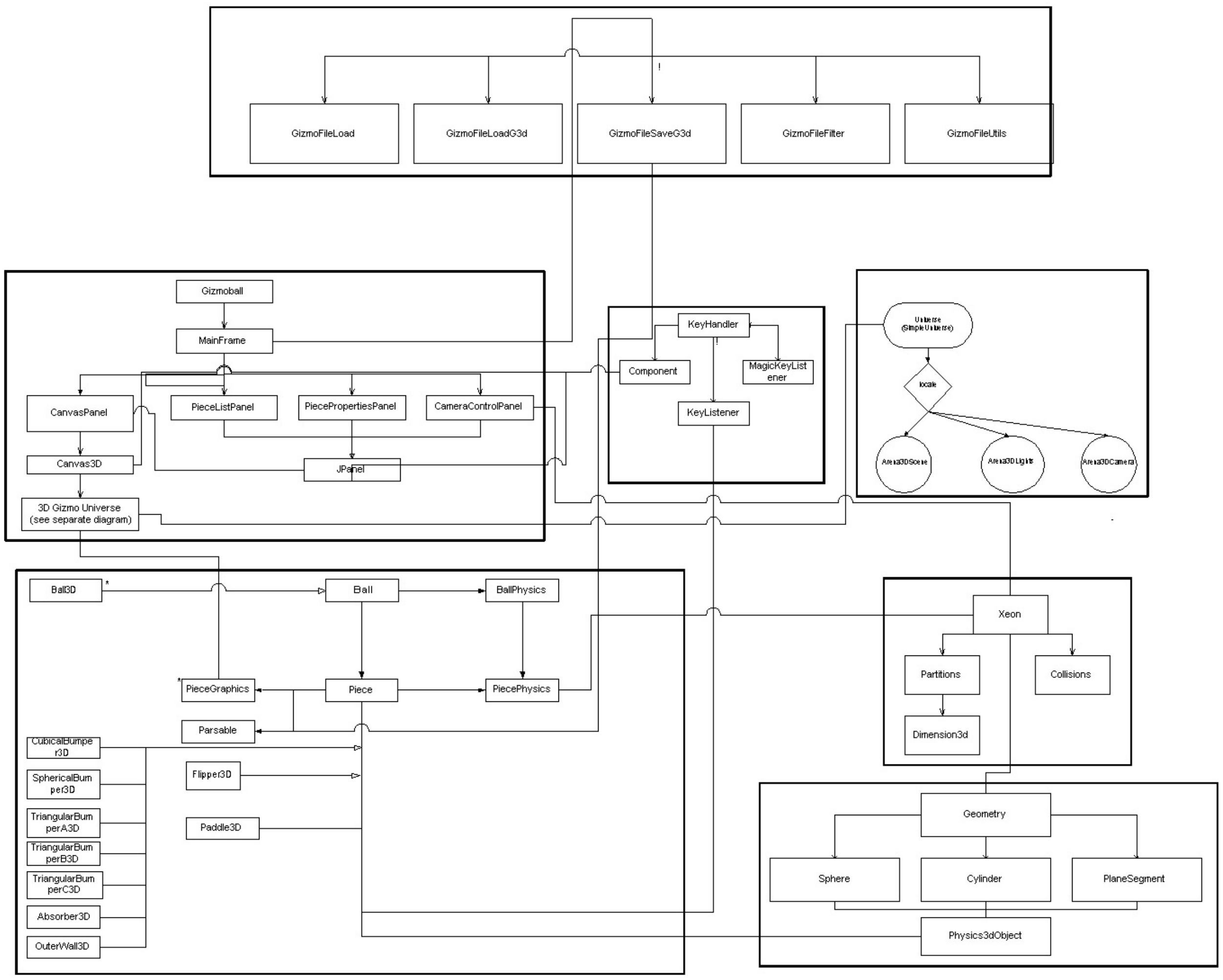

Figure 11: Module Dependency Diagram

III. IMPLEMENTATION

3.1 Overview

The actual implementation of the Gizmoball system consisted of 5 top level packages, reflecting the modules described in Section 2.1. In order to promote black box modularity, each package was designed to communicate with one another through a set of interfaces.

3.2 Backend

The backend serves to hold the interfaces of the different types of gizmos, as well as provide a location for code that extends the functionality of Gizmoball. It consists of a hierarchy of interfaces, some of which are declared abstract to notify designers not to implement these interfaces directly, but rather use one of the direct sub-interfaces of the those abstract interfaces. These abstract interfaces should only be used for generic typing across different pieces. Due to the nature of the Java programming language, there is no way to prevent the designer from directly implementing the abstract interfaces; however it is noted in the Java documentation of the Gizmoball software system that only use of the concrete classes will guarantee proper functionality with the GUI and the physics engine. Any attempts to use the abstract interfaces will be notified to the designer at runtime, but not compile-time. The two abstract interfaces are Piece and Gizmo, while Ball, StaticGizmo, RotatableGizmo, and MobileGizmo are all concrete interfaces. Although the basic functionality of gizmo ball has only a couple of classes implementing each interface, and only one class implementing the Ball interface, it is easily possible to create new types of gizmos and balls and have proper behavior in the Gizmoball system. By having interfaces, we can ensure maximum extensibility to third-party developers. There is also an interface to help build factory methods for different gizmos. The backend also contains two inner packages that help facilitate keyboard input and generic saving and loading of files.

3.2.1 Piece Interface

This is the top-level abstract interface used by all playing pieces. It is only used for developers to see what is common to all pieces in the Gizmoball system and should not be directly implemented. Methods that are common to all pieces in the Gizmoball system are enumerated below:

- Get and set the location of the piece in 3d space. These are absolute pixel coordinates as used by the Java3D coordinate system.
- Get and set whether this object is a physical object anymore – this will notify the engine whether to take into account the piece in processing the physical environment of the simulation.
- Setup physics and graphics to set up the pieces for proper rendering and physical simulation when the Gizmoball system switches from editing mode to simulation mode
- Update physics and graphics to animate the pieces in the playing space
- Reset physics and graphics to restore the pieces to their original locations before the simulation was run
- Access to physical components the comprise the wire-frame of the piece for proper simulation
- Access to bounding box that represents range of motion for a piece. Note for gizmos, bounding boxes may not change locations due to the physical simulation of the engine, whereas for balls they may.
- Rotate 90 degrees in the x, y, and z planes to allow for all possible configurations of the pieces.
- Enable and disabling of piece behaviors
- Draw wire-frame or fill mode for different possible graphical renderings of the pieces in the playing space

3.2.2 Gizmo Interface

This is a abstract sub-interface of piece, but a top-level interface for all gizmos in the playing space. It also should only be used for developers to see what methods and actions are common to all gizmos in the Gizmoball system and should not be directly implemented. Functionality specific to gizmos is enumerated below:

- Get and set coefficient of reflection
- Registering a ball collision and triggering functionality to other gizmos
- Listens and processing keyboard input from the user

3.2.3 Static Gizmo Interface

This is a concrete sub-interface of the Gizmo interface and is to be directly implemented in gizmos that possess no velocity component that must be considered in the real-time physics simulation of the playing space. All functionality of the static gizmo is inherited from the abstract Gizmo and Piece interface.

3.2.4 Rotatable Gizmo Interface

This is a concrete sub-interface of the Gizmo interface and is to be directly implemented in gizmos that possess only an angular velocity component that must be considered in the real-time physics simulation of the playing space. All functionality of the rotatable gizmo is inherited from the abstract Gizmo and Piece interface, along with the extended functionality below.

- Get and set the angular velocity
- Get and set the axis of rotation

3.2.5 Mobile Gizmo Interface

This is a concrete sub-interface of the Gizmo interface and is to be directly implemented in gizmos that possess only a translational velocity component that must be considered in the real-time physics simulation of the playing space. All functionality of the mobile gizmo is inherited from the abstract Gizmo and Piece interface, along with the extended functionality below.

- Get and set the translational velocity

3.2.6 Ball Interface

Balls are extremely different from gizmos in that they have complete range of motion in the playing space, and are considered to have some finite mass associated with them. Balls possess a linear velocity component and are free to change the bounding box coordinates dynamically throughout the game. Balls have the following functionality:

- Get and set mass
- Get and set linear velocity

3.2 Engine

The engine is designed to simulate a real-time physical environment in the playing space, while being as completely separated from other modules as is allowed. One of the key designs of the engine is that it can be used to simulate generic graphical playing spaces. The engine only requires a few bits of information about the playing space to make it completely abstract the space into its own view. Once it has created its vision of the space, it can simulate the physical environment of the space based on time increments given to it. Some of the key design points about the engine is minimal calls to the garbage collector by reusing data structures, using only canonical representations for various immutable objects, and approximating collisions that happen in a time interval smaller than is discernable to the human eye. Once the engine has been set up with the list of balls and gizmos, as well as the maximum size of the playing space, simulating the physical environment can be run. The physics engine algorithm is described below:

Algorithm Physical_Simulation

```
While (the amount of time in the time slice is > 0) {

    For every ball in the physical system {
      Find the minimum time until collision between any ball and any other ball or gizmo
    }

    Simulate all physical components of balls and gizmos up until this minimum time
```

```
        If minimum time is less than the time slice {

            Find all possible collisions that will occur within some tolerance of time that is indiscernible to
            the human eye

            Process all collisions and trigger gizmos that were involved collisions
        }

        time slice = time slice – minimum time
    }

    Update all balls
    Update all gizmos
```

Many optimizations were made in the code to this overall algorithm for physical simulation. Instead of every ball checking against every other ball and every other gizmo, a process that would take O(m(m+n)) time, where m is the number of balls in the system and n is the number of gizmos in the system, the algorithm will only check for gizmos in the immediate localized trajectory of the ball. This significantly reduces the amount of time checking for possible collisions and adds for a smoother animation. Time performance was enhanced by creating all memory intensive objects at setup and reusing them throughout the simulation, as well as processing all collisions that occur within a fraction of a millisecond between each other. Representation Invariants and Abstraction Functions can be found in Appendix A.

3.3 Graphical User Interface

The GUI of our Gizmoball system consists of one module, with many packages to separate the main graphical engine from graphical rendering and interacting tools. Upon initialization of the program, the user is presented with the layout of the 3-dimensional playing space, and various buttons and menu items to configure the playing space as he / she desires. Camera controls as well as simulation controls are located in the bottom left hand side of the interface, while editing functions such as addition of gizmos and gizmo properties comprise the rest of the interface. Easy and simple drop down menus allow the user to view the playing space with grids for easy placement of the pieces, as well as allowing the user to save or load files.

The primary concern of the GUI is to provide an aesthetically pleasing and user-friendly environment for constructing the gizmo arena. A clean finish for the background provides for a polished look that the user can respect and admire. Buttons were configured to provide maximum visual indications of functionality. The playing space is seen as a window into a 3-dimensional world, helping separate the properties panel from the playing space visually. In terms of functionality, the user has many perspectives to choose from to design his playing space, as well as a locking mechanism that will lock his perspective to a zoomed in 2-dimensional perspective of each face of the board for fine configuration of the balls and the gizmos in the three dimensional space. Grid panels are placed to guide the user in placing his gizmos in convenient locations around the space, but these panels are removed once the play button is pressed to provide maximal visual enjoyment of the interactions between the balls and the gizmos. However, with menu bar choices, the user may opt to choose to keep the grids in place, as well as display the outer walls both during editor and simulation mode.

We will spend the next couple of weeks fine tuning the graphical user interface to provide an environment that conforms to Nielsen’s 10 Usability Heuristics. This will include a progress bar for when play is hit so the users will not worry the program crashed as their simulation is being checked and loaded into the system.

3.4 Physics 3D

The physics module was written to support 3 dimensional physics. There are several geometrical objects that are supported within the physics 3d package, as enumerated below:

- Cylinder: A basic symmetrical cylinder with base radius r and height represented as the distance between the two circles. Location of the cylinder is based on the location of the two bases, where the cylinder is the object between the two points of radius r. NOTE: Line segments are constructed by making a zero-radius cylinder.
- PlaneSegment: A triangular plane segment described by three vertices in absolute space. The plane is the area of space that is coplanar to all three points and contained within the bounds of all three points.
- Sphere: A sphere with radius r and center c, where r is a floating-point number and c is a 3-dimensional point in space. NOTE: Single points are constructed by making a zero-radius sphere.

To provide extensibility and modularity, all geometrical objects in the physics package implement the Physics3dObject interface. Also, 3-dimensional transform matrices were used instead of a conventional 3-dimensional vector to represent locations to allow for rotations along axis, i.e. spheres that spin while having translational motion.

In addition to the various physical objects, the physics package has a Geometry class that has many useful static methods to determine times until collisions between spheres and any other physical object, as well as processing a collision and determining resulting velocities between a sphere and any other physical object. Representation Invariants and Abstraction Functions can be found in Appendix A. See Appendix B for physics algorithms that are used to calculate both times until collisions between spheres and various objects, as well as how systems react after a collision between a sphere and another physical object.

3.5 Plugins

3.5.1 Pieces

We approached the problem of handling pieces by considering them as plugins to the interface. Therefore, implementation of the modules was given such that anything that implements their interfaces would work with them. Therefore, pieces only have to implement the basic interfaces to achieve full functionality with the system. We developed each of the basic pieces as mentioned in Section 1.2.1. Representation Invariants and Abstraction Functions for each of the gizmos can be found in Appendix A.

3.5.2 Modes

Although not fully implemented yet, this package will contain pre-made game play modes the user can opt to choose when running the Gizmoball 3D system. Games such as araknoid, pong, and many others will be available to the user through a drop down menu in the GUI. These games will employ many of the gizmos already written, as well as conceivably new 3-dimensional gizmos.

3.6 Triggering System

3.6.1 Gizmo Connections

In the editor mode of the game, the user will be able to have gizmos interact with one another by a process known as gizmo triggering. In this system, if there is a ball collision with a gizmo, any gizmos that are connected to the gizmo involved in the collision will be “triggered” and perform an action visual on the screen. In our system, every gizmo keeps track of all gizmos that it is supposed to notify. This method employs the Observer design pattern in order to provide maximum functionality. Gizmos are notified with a parameter of type Action, which is an immutable class with instructions as to how the gizmo should act upon being triggered. This allows for gizmos to perform multiple actions depending on the gizmo that triggered them.

3.6.2 Key Bindings

Key bindings are an integral part of user interaction with the playing space. After the play button is pressed, any keys that are pressed will trigger gizmos in the playing field to perform specific actions. This interaction, although facilitated by the graphical user interface, is handled through the backend package *events* to serve as a communicator between what gizmos should be listening for events based on a key press, and what items should be listening for key presses. This decouples the dependence between the graphical user interface and gizmos, because GUI elements can add themselves to the to be listeners of key events, and redirect those events to the key handler, which will disseminate the key handling events to the gizmos themselves. Since all actions are handled by the key handler, clearing lists of key bindings and disabling listening is extremely facile.

3.7 Streams

3.7.1 Save / Load

The load and save features of Gizmoball are designed to be flexible and robust. Each Gizmoball game piece has different properties, and each one is capable of un-parsing its properties into a String. GizmoballFileSaveG3d can then be implemented easily and efficiently by just iterating through a list of all the game pieces and asking them to un-parse their string representations into the output file. This is handled in a modular fashion because gizmos implement the Parsable interface

GizmoballLoad and GizmoballLoadG3d are implemented in a neatly organized command structure that breaks down an input string and calls the appropriate methods depending on the string tokens. It should be noted that GizmoballLoad needs to take into account the fact that the 6.170 file format has positive y going downwards, whereas our graphics/physics/engine packages interpret positive y as going upwards. GizmoballLoad automatically makes the appropriate conversions and outputs the right layout on the screen.

To help with loading and saving, GizmoballFileUtils and GizmoballFileFilters are used to help handle extensions. This way, the GUI is able to filter out all non-relevant file formats when the user selects a file to load.

IV. TESTING

4.1 Strategy

Due to the diverse nature of the numerous modules that comprise the Gizmoball system, several testing strategies were employed to ensure the stability and effectiveness of the software suite. The physics module was extensively tested using testing files conforming to the JUnit Framework. The engine was also tested for proper behavior to invalid inputs using the JUnit framework. Our testing strategy was to test as soon as new implementations were developed for our modules. We are incorporating black box, glass box, and boundary case testing for all of our code. Every gizmo will have JUnit tests to assess for proper functionality. The engine is accompanied with a testing driver with pseudo gizmos to drive it with in order to test proper functionality with the physics module. All output is to the shell output as of now, but will be rewritten to file output so that test comparisons using the JUnit framework can take place. Additionally, we have written some g3d files that set the environment up with stress tests and possible problematic configurations of pieces to analyze what the inherent root of the problems are.

The graphical user interface will be tested exhaustively by performing every combination of button clicks as well as menu options. We will also ask some of our friends to interact with our system in an attempt to find programmatic faults and aesthetical problems. Enumerated below is our testing strategy:

- Event Handling: Event Handling classes are being tested using the JUnit framework.
- Streaming: The file load/save classes are being tested exhaustively using the JUnit framework for proper format of input files as well as logic of the parameters contained within the input files.
- Interfaces: Since interfaces are only collections of methods, only classes that implemented the interfaces were tested. Method declarations that have a signature that throws an exception were JUnit tested to make sure they threw the exception in the correct manner.
- Factory Objects: These classes are inherently very simple classes, as they only relay information passed to them to the proper constructor of the object they make. Therefore, minimal JUnit testing is being written for them.
- Gizmos: Gizmos are tested exhaustively using the JUnit framework for proper inputs, as well as encapsulation of internal fields. All methods from interfaces that the gizmo implements are tested using black box and glass box testing.
- Physics: The physics module was testing using the JUnit framework for every possible scenario that could occur in our physical simulation. However, since we are unaccustomed to 3-dimensional physics, this testing file is constantly updated when new physical simulation errors are witnessed within the GUI.
- Engine: The engine is tested using a simple JUnit framework to glass box test for proper exception throwing. The crux of the testing and debugging comes from a text based driver for the engine using classes that implement on the physics interfaces of required by the engine. This output will soon be logged into a text file and adequate JUnit tests will be written that compare the actual output with the expected output.

4.2 Test Results

So far, all of our testing has proved to be successful and while there are a few bugs that we foresee, we also know of numerous ways of fixing those bugs. There are no potential handicaps thus far. All testing files are or will be in a sub directory of each package labeled testing, so third-party developers will be able to test the code for themselves in an easy to use fashion. We will be employing both computer science students and non computer science students to test our software during this code validation process to eventually provide the highest quality product at deployment.

V. REFLECTION

5.1 Evaluation

We approached this problem from a very rigorous and meticulous standpoint, thinking of how to make the program both modular and extensible to third-party developers, while also providing user-satisfaction to the client. Each person was assigned a particular module to work on from the start, and this process was key in that it forced modularity to occur as we knew that we would have to integrate the modules together eventually. Once each of us has completed his module, instead of immediately integrating, we had team meetings in a classroom with nothing but a printout of the code, and each person was expected to completely explain his module in a top-down hierarchal method. Not only did this allow each person to truly understand what he was doing with the module, but it allowed conversation to occur such that all team members could see the big picture of the software system.

One of our first design decisions was to make Gizmoball extremely extensible, even more so than the specifications that were given to us. As a result, we decided to support a 3-dimensional space, along with a dynamically changing board size instead of the hard coded 20L x 20L configuration. Many other configurations are allowed to achieve maximum simulation capabilities of the Gizmoball engine. We see that even though Gizmoball is a game and for pure enjoyment, one could simulate many scientific principles using the same system.

Every team member was uniquely assigned to a particular module based on his interest to the key issues of that module, whether it be graphical interaction, real time physics modeling, or writing complex algorithms for real-time simulation. This allowed us to easily overcome the basic requirements of the Gizmoball system and take the concept of Gizmoball one level further, or in our case, one dimension further. None of the team members would settle for anything less in performance or graphics just because a 3-dimensional package was used – it was decided early on that if we were to do 3-D, it would work just as well or better than had we done the system in 2-D. Due to this determination and hard work, we have so far seen all our visions of the Gizmoball system come through as realities.

Although the system may look complex and daunting, once the user has become accustomed to the interface, he or she will find it a very powerful tool to use. From the developer's standpoint, the modules are decoupled in a very rigid fashion, and many extensions can easily be added in. One of our strongest features for developers is that not only can new gizmos be easily created and dropped into the plugins folder of gizmos with very little addition of code to the gui and none to the engine, the physics package can be appended to support rotation, if the user would wish to add features like spin to pieces. In this case, only the physics module and new spin gizmos would need to be changed; however, the engine would support this functionality in its current build.

5.2 Lessons

We saw from very early on that since we were overextending ourselves, we should employ as many design patterns as possible to make sure that the system would be modular and easy to use. This paid off in the end since we could easily decouple each module from one another. Any time that a design module was implemented, it was tested for performance as well as correctness. This led to few bugs and provided an unimpeded path towards our goal.

One of our unforeseen problems however was the loss of a valuable teammate during the software development phase. This caused us to have to reorganize and split up the lost team member's work across all three of us, and introduced an even heavier load of work for this project on each of us. This increased the stress level for each team member and temporarily slowed our progress – however, once milestones were hit and the integration was proving itself to be successful, our stress quickly disappeared and we were able to continue at our original pace. As a result, we managed to stay on top of our original schedule for the project. Thus far, we have learned valuable lessons in the areas of effectual design, implementation, and teamwork.

5.3 Known Bugs and Limitations

Since we are still both building and testing our Gizmoball system, there are many known bugs that must be resolved. Our most current limitation is that the user must have the Java3D implementation for his OS installed to use our software, as well as attempting to resolve time and memory overhead issues before final deployment. Some of known issues that must be resolved are enumerated below:

- Have balls only check for against other balls whose immediate trajectory intersects with current ball's trajectory
- Support complete rotation for gizmos without hard-coding every possible configuration, i.e. derive algorithmically how to wire frame graphical gizmos with their physical counterparts.
- Delay absorber firing to allow time for first ball fired to clear absorber fire area
- Support balls with 0 L / s during simulation – to have balls that sit on the floor not bounce around
- If a ball gets caught between a flipper and a wall, make sure this situation doesn't crash the system as it will in the current implementation of flipper
- Add properties panels for the user interface to fully customize the gizmos on the playing field.
- Add progress bars for loading of the program and for switching between editing and simulation mode
- Texture mapping for pieces
- All things mentioned in the Section 1.3 – User Manual

VI. PROJECT PLAN

Thus far, we have completed the majority of our milestones.

Milestones	**Who**	**Date**	**Completed**
- Preliminary GUI interface design decided upon and implemented - Learn how to integrate the Java3D API with Swing. - Research the Java3D API. This includes: i) Designing and implementing a scene branch. ii) Learning the different intricacies of the API (mouse behaviors, mouse picking, setting up cameras, setting up lights, how to manipulate 3D objects, etc.) - Be able to add and delete gizmos, or at least simple 3D shapes (gizmo placeholders) onto the playing arena. - Be able to interface with the backend (i.e. have a functional "play" feature that is able to send and retrieve information from the backend). - Have functionality 3 (balls bouncing) working for the preliminary release	Group	4.20	Yes
Get ball -rotating object collision detection/reaction methods written and tested	Eric	“	Yes
Finish engine that handles gizmos and physics	Ragu	“	Yes
Finalize GUI layout design	Ran*	“	Almost
Snap to grid, lighting up of occupied/forbidden areas of the side grid walls	David	“	No
Optimize for speed	Ragu	4.27	Yes
Develop method invocation module	Ragu **	“	See Note
Program interface of buttons for GUI completed	Ran*	“	Yes
Save/Load Feature working, Key triggers working, Property Panel/Camera control completed	David, Ragu	“	Yes
Photoshopped icons, images, and overall polish and beautification of the GUI.	David/Ran*	“	Yes
-Have real gizmos coded -Have all 4 required functionalities of the preliminary release done	Group	“	Yes
Preliminary Release	Group	4.29	Yes
Debugging and Testing	Group	5.4	
Optimization/Improvements (add more modes)	Group	5.4	
Final Documentation	Group	5.11	
Implementation and Testing	Group	5.13	
User Interface and Quality of Play	Group	“	
Final Report	Group	“	

* - David has taken over Ran’s graphical user interface roles, as he was best suited for the job with his Photoshop experience and being the project leader of the graphical user interface module.

** - Due to time constraints, and the loss of the team member, we will not be able to complete the method invocation module by Demo day.

Appendix A: Representation Invariants and Abstraction Functions

A.1 Physics

Geometry
RI: maximumForesight >= 0.0 && searchSlices > 0

Sphere
AF: An ADT representing a sphere in cartesian space. Spheres have a location, an orientation, and a radius.
RI: radius >= 0 && trans != null

Cylinder
AF: An ADT representing a sphere in cartesian space. Cylinders have two endpoints (p1 and p2) as well a radius.
RI: radius >= 0 && p1 != p2 && p1 != null && p2 != null

Plane Segment
AF: An ADT representing a triangle in cartesian space. Plane Segments have three vertex points (p1, p2, and p3) from which a normal vector is derived in a right-handed fashion: normal = p1->p2 x p2->p3;
RI: p1 != null && p2 != null && p3 != null && p1, p2, p3 non-collinear

A.2 Engine

Partition
RI: All variables are not null;
Dimension3d.x > 0 && Dimension3d.z > 0 && Dimension3d.z > 0
If partition is created in initialize, x,y,z = Integer.MIN_VALUE
If partition is created in getBlocks, x,y,z > Integer.MIN_VALUE
AF: An ADT representing a rectangular prism in space. The partition block is in reference to the origin point that it is initialized with, which represents how many dimensional units a point is away from the starting point.

Engine:
RI: All variables != null && 1 instance of engine exists
AF: Contains a playing space that is rectangular prism starting from The origin and extending to the upper corner. Playing space is represented as a HashMap from partition locations of the space to any gizmo objects in that space. Note that origin and upper corner are relative points and pieces can extend outside this rectangular prism if needed.

Collisions:
RI: All variables != null && (colliding object instanceof BallPhysics) && collided object instance of (PiecePhysics)
AF: This is ADT for handling multiple collisions between BallPhysics objects and PiecePhysics objects. The list of collisions is maintained by hash maps that map a BallPhysics object to the Physics3dObject shape of the BallPhysics object, the PiecePhysics object, and the Physics3dObject shape that is collided with in the PiecePhysics object

A.3 Gizmos

Ball:

```
AF:
  An ADT representing a ball in Cartesian space.
  Ball3D has a location, radius, mass, velocity, and appearance.
RI:
    lowerCornerLocation != null &&
    radius != null && radius >= 0 &&
    mass != null && mass >= 0 &&
    velocity != null
    app != null &&
    isVisible != null
```

Cubical Bumper:

```
AF:
  An ADT representing a cubical bumper in Cartesian space.
  CubicalBumper3D has a location, dimensions, coefficient of reflection,
  and appearance.
RI:
   lowerCornerLocation != null &&
  xLength != null && xLength >= 0 &&
  yLength != null && yLength >= 0 &&
  COR != null &&  && COR >= 0 &&
  app != null &&
  isVisible != null
```

Spherical Bumper:

```
AF:
  An ADT representing an spherical bumper in Cartesian space.
  SphericalBumper3D has a location, radius, coefficient of reflection,
  and appearance.
 RI:
   lowerCornerLocation != null &&
  radius != null && radius >= 0 &&
  COR != null && && COR >= 0 &&
  app != null &&
  isVisible != null
```

Triangular Bumper:

```
AF:
  An ADT representing a triangular bumper in Cartesian space.
  TriangularBumperA3D has a location, dimensions, coefficient of
  reflection, and appearance.
RI:
   lowerCornerLocation != null &&
  xLength != null && xLength >= 0 &&
  yLength != null && yLength >= 0 &&
  COR != null &&  && COR >= 0 &&
  app != null &&
  isVisible != null
```

Outer Wall:

```
 AF:
  An ADT representing an outer wall bumper in Cartesian space.
  OuterWall3D has a location, dimensions, coefficient of reflection, and
  appearance.
 RI:
  lowerCornerLocation != null &&
  xLength != null && xLength >= 0 &&
  yLength != null && yLength >= 0 &&
  COR != null &&  && COR >= 0 &&
  app != null &&
  isVisible != null
```

Flipper:

```
 AF:
  An ADT representing a flipper in Cartesian space.
  Flipper3D has a location, coefficient of reflection, and appearance.
 RI:
  lowerCornerLocation != null &&
  COR != null && COR >= 0 &&
  app != null &&
  isVisible != null
```

Appendix B: Physics Algorithms

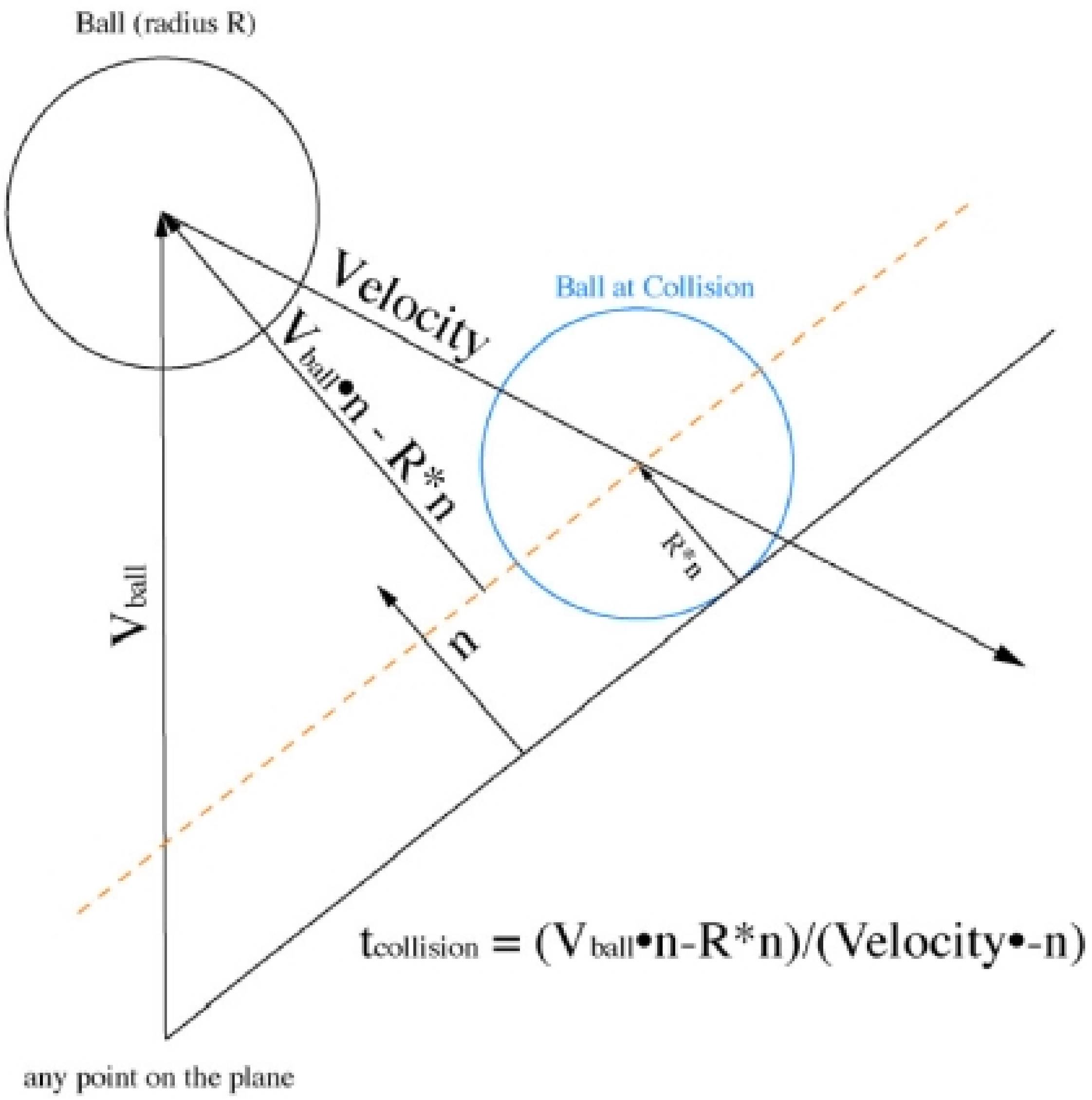

Figure 11: Moving Ball / Non-moving Plane Collision

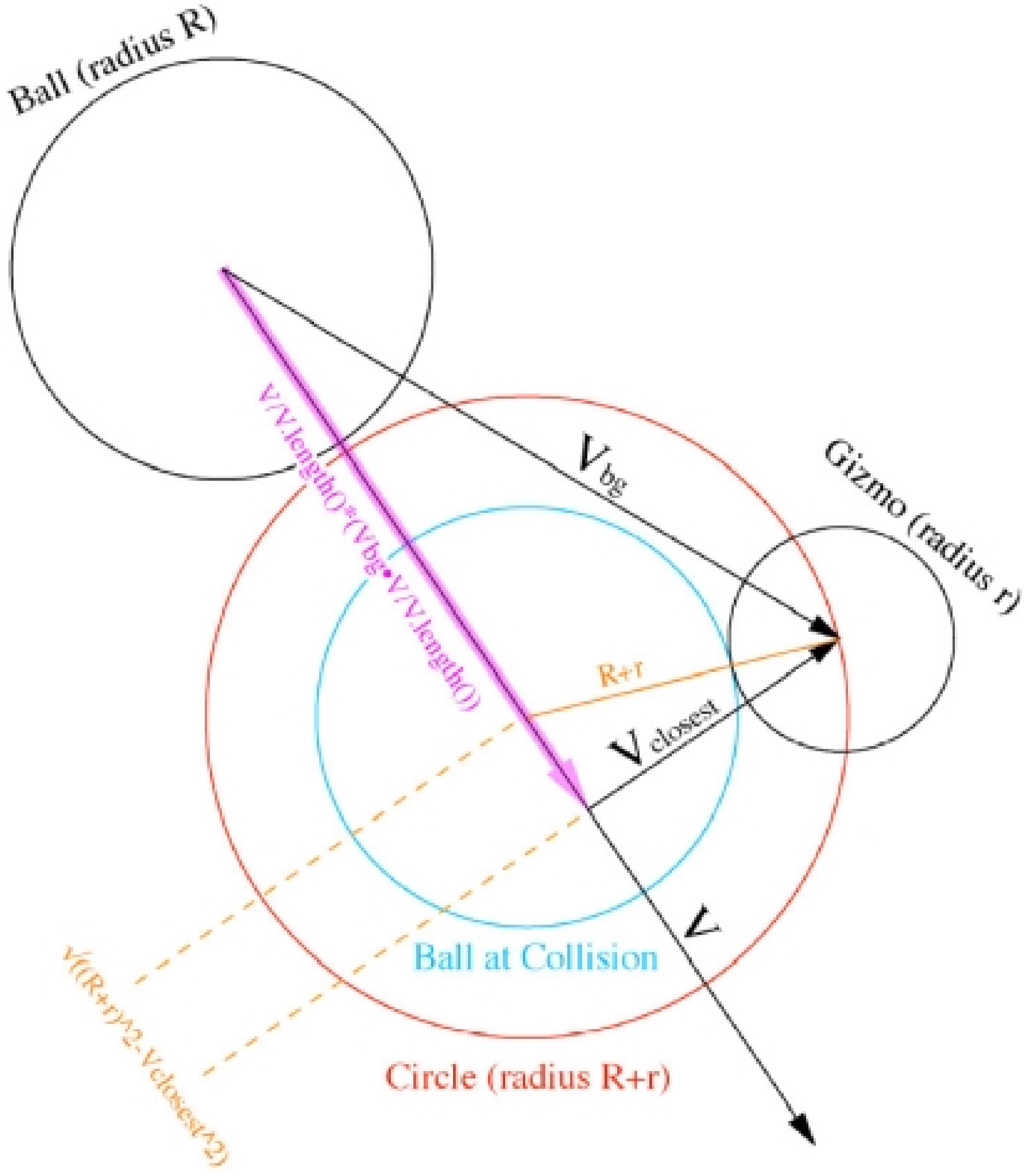

Figure 12: Moving Ball / Non-moving Sphere Collision