Gizmoball 3D: Final Design Document

se25: Ruth Dhanaraj, Chris Varenhorst, Xiao Xiao

Contents

1 Requirements

1.1 Overview

Gizmoball is a game of 3D pinball. In each game, there is a 20L by 20L by 10L board surrounded on all six sides by walls containing balls and gizmos. The gizmos include square, circular, and triangular bumpers, flippers, and absorbers, which can take in a ball and shoot it out. Flippers actions can be trigger be either key presses, or contact by the ball with other gizmos.

Gizmoball allows the user to load an existing level by loading an XML file in the correct format. Users may also define their own levels using the Gizmoball build mode. In build mode, users may add, move, and delete balls and gizmos to the game board. The user may also change the orientation of gizmos, change the velocity of balls, as well as add triggers and targets to gizmos.

1.2 Revised Specifications- Extensions

1.2.1 Z-Movement

Our physics and rendering engines support full 3D movement, including z-gravity. In XML files, gizmos and balls can be specified to be placed anywhere within the bounds of the game board. However, our build mode only deals with objects in the z=0 plane of xy-grid. We made this decision because the interface of build mode would be very complicated and unintuitive if we allowed objects to be placed anywhere in 3-space. For example, multiple objects could share the same xy-coordinates but have different z-coordinates, which would make selecting and moving objects difficult. We feel that for most users, our current build mode provides ample resources with an easy-to-learn interface to create interesting game levels. More advanced users can directly write their own XML files for further options.

1.2.2 Multiple Balls

Although it is not required for this year, we extended our system to support multiple balls. Our physics engine does the correct calculations to simulate interaction between moving objects. We made this decision because multiple balls adds a significant amount of visual interest to the game.

1.2.3 Absorbers and Multiple Balls

Each absorber can only hold one ball in it at a time. When an absorber already has a ball in it, other balls just bounce off of it.

1.2.4 Texture

Our renderer supports texture loading from images and renders gizmos and balls with textures. We have made three sets of skins for our game; these can be selected from the Skins menu.

1.2.5 XML

In addition to the parameters specificed in the requirements document, our xml format supports the following extensions. All of our extensions have default values, and therefore are not required in xml gizmo description. These parameters may not be valid for all objects, but the xml reader will ignore invalid parameters.

1. `coRef = <Double>` – specifies the coefficent of relection of a gizmo. This can override the default value laid out in gizmo specification.
2. `frozen = [true|false]` – specifies whether an object is frozen or not. All obejcts besides ball and frozen by default, and all balls are not frozen unless they are inside in absorber. The XML format lets any ball exhibit the frozen property, but the xml format supports this.

The board tag also supports the `FPS = <unsigned int>` property. This can be used to change the display frames per second in the gamesapce. The default value is 30.

1.3 Revised Specifications - Clarifications

1.3.1 Absorber Orientation

When absorbers change orientation, they rotate on the XY plane about their center. Even though they rotate, the ball is always shot out of the top face. The orientation can be specified in the XML. However, in build mode, this makes no difference as only absorbers of size 1 grid unit is allowed to be placed in the game board.

1.4 User Manual

There are two modes in our system - play mode and build mode. In play mode, the user can load a game, watch the ball move around, and trigger objects with keys if the triggers are set. In build mode, the user can create a customized game level. To switch between modes, click on the *Play/Pause* button. In build mode, the buttons on the side panel are active and a grid is displayed on the game board. In play mode, all buttons on the side panel are inactive, and no grid is displayed.

1.4.1 Building a Level

New Level:

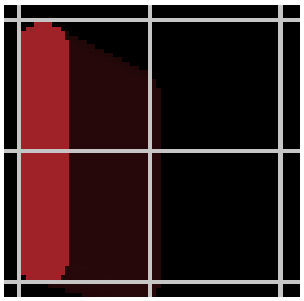
A user may build a customized level in our system. To do that, click on *File* > *New* in the file menu, and an empty game board will load in build mode.

Adding Objects:

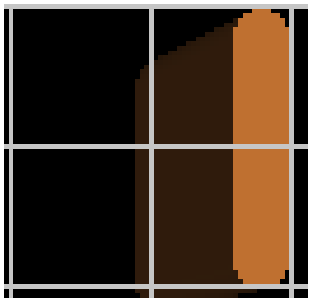
The buttons at the top of the side panel allows the user to add gizmos and balls. To add an object, click on its corresponding add button and click on a spot on the game board. Gizmos will snap to the grid such that the their top left front corner is aligned with the top left front corner of the clicked on grid box. Balls will snap to the grid such that they are placed in the exact center of the clicked on grid box. Objects can only be added and moved on the $z = 0$ plane. If the user does not click on another button in the side bar, other objects of the same type as the added object will be added when the user clicks on other spots on the game board. If a user tries to add an object in an

already occupied grid, nothing happens.

Flippers occupy more than one grid space. In fact, both left flippers and right flippers occupy four grid spaces to take into account the theh arc of their motion. To add a left flipper, click on the grid where you want the top of the left flipper to be. To add a right flipper, click on the grid immediately to the left of where you want the top of the right flipper to be.



(a) The left flipper occupies all four grids shown here. This flipper was added by clicking on the top left corner grid.



(b) The right flipper occupies all four grids shown here. This flipper was added by clicking on the top left corner grid

Moving Objects:

To move an object, click on the *Move* button and select a game object in the grid. The object will turn white, indicating that it is selected. Click on another spot on the game board to move the object to that spot. If the new spot is already occupied by another object, nothing happens. If the user does not click on another button in the side bar, other objects may be moved without reclicking the "Move" button.

Deleting Objects:

To delete an object, click on the *Delete* button and click on a game object in the grid. The object will immediately be removed. If The user clicks on an empty grid, nothing happens. If the user does not click on another button in the side bar, the user may delete other objects without reclicking the "Delete" button.

Setting Properties:

To modify properties of an object, click the *Select* button and click on the object to be modified. The selected object will turn white to indicate that it is selected. A panel will appear on the bottom of the side bar, where the user may set object properties. If the selected object is a ball, the only option on the bottom panel will be velocity. The fields will display x, y, and z components of the current velocity of the ball. To change the velocity, type in the desired components in the text fields and press the *Set Ball Velocity* button. If an invalid velocity is entered, nothing happens.

If the selected object is a gizmo, the details panel will have four sub-panes: Orientation, Triggers, Targets, and Delay. To modify one of these properties, click on the appropriate button on the side of the pane or click on the up arrow and select one of the four options that show up. When *Orientation* is clicked, the current orientation of the selected object is displayed in a combo box. Select another value in the combo box to change orientation.

When Triggers is displayed, the trigger pane appears, which displays the current triggers attached to the object as a list of buttons showing the name of the key and an up or down button depending on the type of trigger. There are two types of triggers: triggers on key press and trigger on key release. To add a key press trigger, click the *KeyDown* button. To add a key release trigger, click the *KeyUp* button. When clicked, the buttons will open a dialogue window. Press a key and press *select* to add the trigger. To remove a trigger, click on the button associated with it inthe list.

To add a delay to a trigger, click on the *Delay* button on the side. A slider will appear, showing the current trigger delay in seconds. To modify the delay, move the slider bar. Delay can be between 0 and 10 seconds.

To add targets to the selected gizmo, click on the *Targets* button on the side. The target selection panel will pop up with a list of buttons displaying the name of the target. Holding the mouse on the button displays the location of the target object. To add a target, click the *Add* button and an object in the grid. Adding another target requires reclicking the *Add* button. To remove a target, click the button associated with the

target in the list. To remove all targets, click the clear button.

1.4.2 Loading a Game

After starting our program, the user can load a premade game level to play by clicking on *File* > *Load* in the file menu. A file chooser will pop up, and the user many select an XML file in the valid form. If the game is currently in build mode, the user must switch to play mode from the Mode menu item. To change the look of the game, go to $File > Skins$. There are three different looks available- the default, natural, and cute. Skins only change the textures mapped to objects and do not affect game play.

1.4.3 Examples

Example 1: Adding a ball and several gizmos to a level

This example gives instructions for building the level shown in figure 1.

1. Open a new game level by going to $File > New$ in the file menu.
2. make sure that you are in build mode and that the grid is active. if not, click on the "Pause" button.
3. Add circle bumpers. Click on the *Add Circle Bumper* button and click on the locations in the grid where you want to place them.
4. Add triangle bumpers. Click on the *Add Triangle Bumper* button and click on the locations in the grid where you want to place the triangle buttons.
5. Add the absorber. Click on the *Add Absorber* button and click on the spot on the grid to place the absorber.
6. Add the left flipper. Click on the *Add Left Flipper* button and click on the grid location where you want the top half of the flipper to be.
7. Add the right flipper. Click on the *Add Right Flipper* button and click on the grid location immediately to the left of where you want the top half of the flipper to be.

8. Add the balls. Click on *Add Ball* and click on the grid where you want your ball to go.

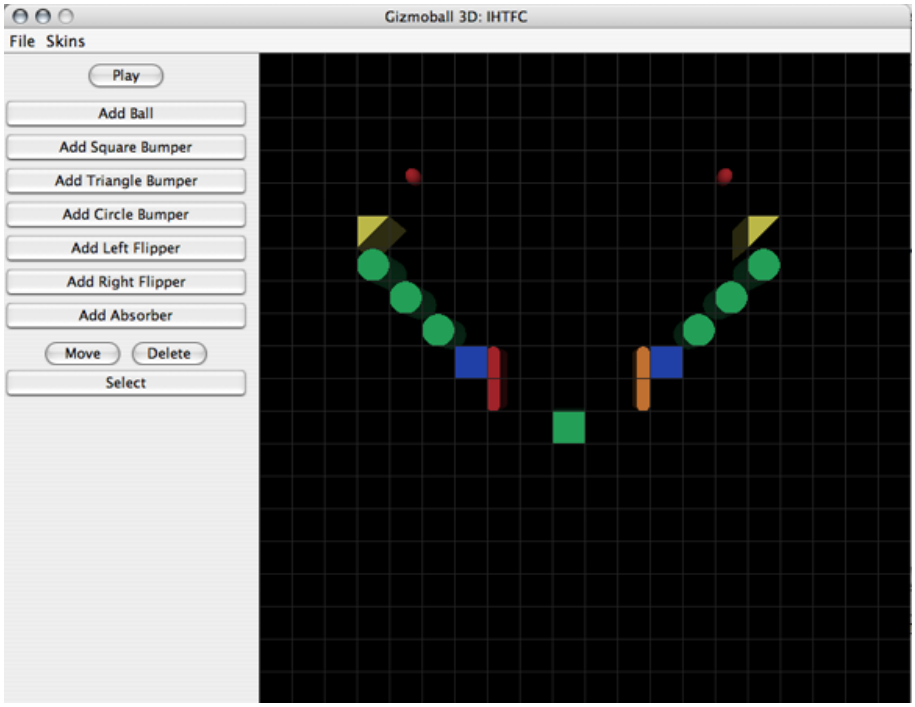


Figure 1: Follow the instructions in example 1 and look at this picture for where to place the GameObjects.

Example 2: Changing the orientation of an object

this example starts where example one left off.

1. Select the triangle bumper on the left by clicking the "Select" button and clicking the triangle bumper. Now, a panel will appear beneath the buttons in the side panel. This panel will have several buttons on the side.

2. Click on *Orientation* on the side of the bottom panel. The combo box will display "0 degrees" because that is the current orientation. Select "270 degrees". This will rotate the selected triangle bumper 270 degrees clockwise.

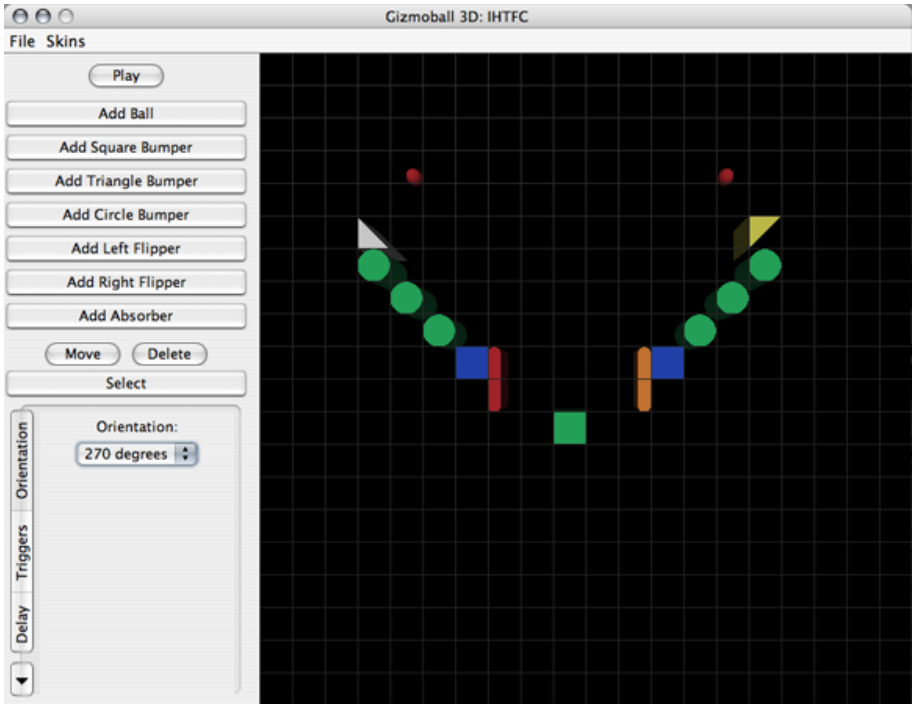


Figure 2: The orientation of the left triangle bumper has been changed to 270 degrees.

Example 3: Adding velocity to a ball

This example starts where example 1 left off.

1. Select a ball by clicking the *Select* button and clicking on a ball. Now, the bottom panel will appear.

2. The only option for ball is *Velocity*. The fields will all display zero because the default velocity for a ball is 0 for the x, y, and z components. Type the desired ball velocity into the text boxes for each of the components.

3. Click *Set Ball Velocity* to change the ball's velocity to what is in the text boxes.

Example 4: Adding triggers

This example starts where example 1 left off. In this example, we add triggers to a flipper such that it acts like a flipper in a normal pinball game:

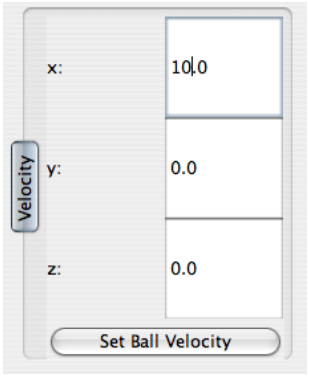


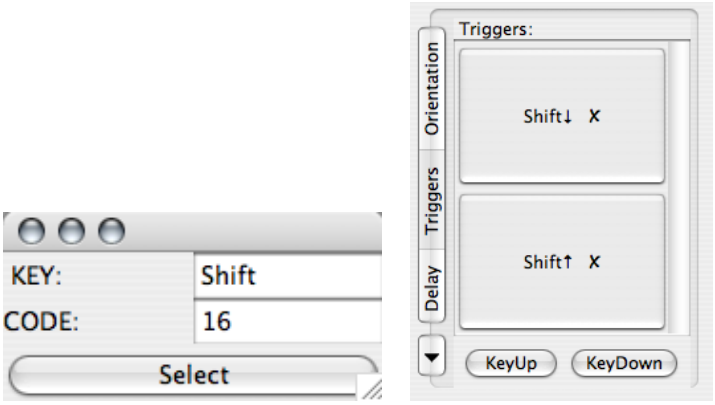
Figure 3: The detail panel showing the ball's changed velocity

1. Select a flipper by clicking the *Select* button and clicking on a flipper. Now, the bottom panel will appear.
2. Click on the *Triggers* button on the side. The Triggers panel will show up.
3. Click on the *KeyUp* button. A dialogue will pop up.
4. Press the key that you want to use to trigger the flipper and hit the *Select* button.
5. Click on the *KeyDown* button. The same dialogue will pop up.
6. Press the same key as you did for the KeyUp.

Example 5: Adding targets

This example starts where example 1 left off. In this example, we add connect a flipper's action to a circle bumper getting hit by the ball.

1. Select a circle bumper by clicking the *Select* button and clicking on a circle bumper. Now the buttom panel will appear.
2. Click on the *Targets* button on the side. The Targets panel will show up.
3. Click on the *Add* button and click a flipper.



(a) Dialogue for trigger key selection (b) The detail panel for Triggers

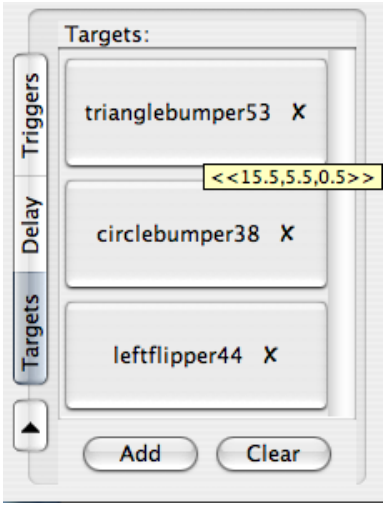


Figure 4: The dialog for trigger key selection. The numbers indicate the location of the triangle bumper and appear when the mouse (not shown) is held over the button.

1.5 Performance

On a MacBook Pro laptop running OS X with a 2 GHz Intel Core Duo processor and 512 MB of RAM, the default game level runs very smoothly. On Athena, the ball movement seems to be slightly slower than on the Mac. Our system should run smoothly for all game levels following the specifications of the final project, i.e. with a single ball.

With multiple balls, our system tolerates as many as 50 before animation starts looking slower. With multiple balls and multiple gizmos involving curved surfaces, animation is even slower. This is because calculating collisions with curved surfaces is much more computationally intensive than collisions of ball with flat surfaces.

1.6 Problem Analysis

One major goal of our project is to create a simulate pinball game that looks realistic. To achieve that, we had to solve two major problems- physic simulation and object rendering in 3D.

For physics, we were provided with a premade library that contained classes for objects representing different primitive shapes and classes that contained mechanisms to do collision calculations for the primitive shapes. Even though we had these tool, we still had to figure out when to call methods to calculate collisions and how often to calculate the collisions in order to make the movement appear smooth and realistic.

For rendering, we utilized the Java OpenGL (JOGL) libraries, which is capable of rendering primitive shapes like plane polygons, circles, spheres, and cylinders. We had to figure out the most efficient and extensible way of separating out objects for rendering- i.e. should we have methods that render each gizmo as a unit, or should we separate each primitive of a gizmo and render them using separate methods.

For both physics and rendering we had to figure out how to encapsulate 3D shape information for each gizmo with the gizmo classes.

Another problem that we had to solve is the delagation of responsibilities within our modules. Issues that we had to consider to solve this problem involved what information to keep with which classes and what classes important methods should belong to. For instance, should ball be responsible for its own movement and for telling objects that it collides with the update, or should a separate controller do that. Also, which classes should do which calculations. The design decisions that we made with regards to these issues tried to maximize modularity and extensibilitiy. When making decisions, we tried to minimized module dependencies. We also often tried to come up with possible

additions to our project (even if we were not going to implement them for any release) and thought about whether they would be supported in our framework.

For the user build mode, our goal is to create an easy to use interface that still had enough options for a user to create an interesting game. Because our physics and rendering models support full 3D movement, we had the potential to add a lot of customization options to our build mode. Thus, one problem that we faced is balancing extra functionality and options with ease of use.

2 Design

2.1 Overview

The most important principle that we tried to follow in our design is modularity. In general, we wanted objects to deal with the specifics of itself. For example, the game space need not know the internal workings of a ball, nor how flippers work. This way, objects have less dependencies on each other, which makes complexities easier to manage. It reduces the number of cases that objects need to consider when interacting with other objects, which decreases the chance for errors. We also tried to keep single classes from keeping track of too many things. This also makes complexity more managable, especially when testing and debugging.

2.1.1 Game Components

All balls and gizmos in our system extend the GameObject class. GameObject has fields for general information that all balls and gizmos have as well as methods shared by all the components. We made GameObject an abstract class because unlike interfaces, abstract classes may have fields and may be partially implemented. This is useful because many of GameObject's subclasses share the exact same methods and need to keep track of the exact same fields. Doing this allows us to reuse much code.

2.1.2 Shape

An important field for all GameObjects is its shape, which is used both in physics calculations and in rendering. Shape represents the abstract shape of an object. Because many GameObjects share shapes (i.e. all balls have the same shape, all square bumpers have the same shape), the Shape class has static Shapes for ball, bumpers, and flippers, which uses the interning design pattern. Only one copy of a ball, bumper, or flipper's shape is ever made each time the system is run, and all GameObjects that have those shapes refer to them. This design decision was made for efficiency. Not having to create multiple copies every time a shape is made saves space.

The Shape class also keeps track of the bounding sphere of a gizmo or ball, which is used in physics calculations

2.1.3 Physics

Our physics engine deals with the motion of balls around the game space. `GamePhysics` contains the static method `moveBall()` which moves a ball for one frame. This class is the bridge between the game component layer (i.e. `Ball`) and the physics3d library layer (i.e. `Sphere`).

2.1.4 Rendering

Rendering is done by primitives from the Physics 3D library using JOGL. There are 4 types of object that can be rendered: PlanePolygons, PlaneCircles, LateralCylinders, and Spheres. Each of these objects implement the interface PhysicsShape. In the Renderer, there are separate methods to render each PhysicsShape. We chose to render by primitives rather than by GameObjects because this system is more extensible and simplifies rendering. In this system, we don't have to change the Renderer code at all if we added another gizmo with a completely different shape. If we rendered by gizmos, we would have to add a new method to the renderer. Our design also reinforces the encapsulation pattern that classes don't know unnecessary information about other

classes. Renderer does not know or care what gizmos or balls look like. It just draws all the primitive PhysicsShapes that it is given.

2.1.5 Triggers

Triggers are an implied structure in our design. Each `GameObject` has an associate action, defined in its `actionPerformed` method and a set of targets. The `GameSpace` contains a mapping of key codes to the objects they trigger. An object's action is triggered either if, 1) the object is contained in another object's list of targets and a ball hits, or 2) the object is contained in the GameSpace's mapping of keys to objects and one of its key is pressed.

2.1.6 XML

XML reading and writing interact with the rest of the game entirely through the `gamespace`. XMLReader has a method to take an gamespace, and write that information to an XML file. The event driver XML reading model is used. `GameObjectClassification` is used to determine what factory to use to generate a new instance of that object. Using the property map object construction model (see above), all XMLReader needs to is convert the xml data into a property map and pass it on to the object.

XML writing constructs an XML document using the `DocumentBuilder`. Each gameobject has a `getBasicPropertyMap()` which stores that object's properties, XMLWriter only gets the objects from `gamespace`, and then using `getBasicPropertyMap()` and `GameObjectClassification` it has all the information it needs to create the Gizmo object tags. Connection information is gathered from `gameobject` in a similar way.

2.2 Runtime Structure

2.2.1 Components Package

GameObject:

GameObject represents a basic object that can be placed in the game board. A

GameObject keeps track of its center, orientation, velocity, and shape, which define where the object is and what it looks like. Orientation is composed of two parts- a vector around which to rotate and an angle to rotate by. Velocity is always zero for every GameObject except ball.

GameObject also has two boolean fields- visible and frozen. Visible indicates whether or not the object can be seen. For example, walls are not visible, neither are balls in absorbers. Frozen indicates whether the GameObject is affected by physics. Ball usually has frozen set to false unless it is in an absorber. All other GameObjects always ha frozen set to true.

Each GameObject keeps track of its own set of Targets, which it triggers when hit. It also has an integer field called delay, which indicates how long to delay an action when it is triggered. When a ball collides with a GameObject, the object calls its onCollision method, which tells its targets to perform their action.

Most accessor and mutator methods for fields are defined in GameObject. These are inherited by the subclasses of GameObject. The stepFrame() method, which indicates what each GameObject does in each frame is define as empty in GameObject. This is because most GameObjects (bumpers, absorbers) do nothing in each frame. Flippers and Balls override this method. actionPerformed(ActionEvent e), which incidates what a GameObject does when triggered is empty in GameObject for the same reason.

In build mode, GameObjects must snap to a grid by their top left front corners. Thus, GameObject has a method called getTLF(), which returns the top left front corner as calculated from the center and the difference between the center and the top left front corner. GameObject has a default getTLF() method, which works for all bumpers. Flippers, Absorbers, and Balls override the default.

Also, GameObjects has an abstract getOccupiedPositions method, which is used in build mode to keep objects from being placed on top of each other. Because GameObjects have different sizes and shapes, this method is defined in the subclasses.

Ball:

The Ball object represents a Ball the game. Ball extends GameObject but differs

from other GameObjects because it can freely move around the GameSpace, bouncing off other objects and getting affected by gravity and friction. Ball controls its own movement in the stepFrame() method, which it overrides. It does calculations using the static moveBall method in GamePhysics.

SquareBumper, CircleBumper, TriangleBumper:

Bumpers all extend GameObject and do not override anything. They each return their own Shape from the static Shape accessors in the Shape class

Flipper, LeftFlipper, RightFlipper:

Flipper is an abstract class that extend GameObject. Flipper has a boolean field called homeState that indicates whether or not the flipper is "down" or "up". It also has an angle called current, which keeps track of the current angle of the flipper, which is between 0 and 90 degrees. Flipper's action() method moves the flipper up and down based on triggers. Flipper's getDiff() method is abstract because Left and Right flippers are different in regards to the positional relationship between the top left front corner and the center. Note that the center for a flipper is not the center of gravity but rather the center of rotation. Flipper also has an abstract getGOClassification, which returns the GameObjectClassification, used for XML IO and loading textures. LeftFlipper and RightFlipper both extend Flipper. Their only difference is the way their getDiff() method is defined and what getGOClassification returns.

Absorber:

The Absorber object extends GameObject. It has an additional GameObject field called holding, which keeps track of what the absorber is holding. Absorbers override GameObject's onCollision method to take in a ball.

Walls:

Walls extend GameObject. For Walls, the visible field is always set to false. the getOccupiedPositions() returns a position not contained in the GameSpace (-1, -1, -1) because walls lie just outside of the playing space.

GameSpace:

GameSpace is basically a container for GameObjects. It keeps track of all the

GameObjects in its bounds with a Set and it keeps track of the six walls with a set of Walls. GameSpace is always the same size- 20L by 20L by 10L. This is hardcoded into the game with the placement of the default walls in its constructor. GameSpace also keeps track of all the key triggers used for GameObjects. GameSpace has a Map<GameObjectClassification, String>, which specifies the location of textures to be rendered for it. In addition to the standard accessor and mutator methods, GameSpace has several methods of note. GameSpace's stepFrame() is called at every time frame. It in turn goes through all of its GameObjects and calls their stepFrame() method. writeXMP(File f) writes the current state of GameSpace to a file specified by f.

GameObjectClassification:

GameObjectClassification is an Enum that identifies GameObjects.

2.2.2 Object Construction

Because of the large number of properties each object has, passing them as a comma-separated list in the constructor (along with all abbreviated forms) quickly became cumbersome.

Instead, we chose to construct `GameObject`s by passing a map of properties. GameObject's constructor first sets properties to `defaults()`, some default properties (i.e. orientation is zero, velocity is zero, frozen is true, visible is true, coefficient of reflection is one...). Then the `GameObject` overrides defaults with the properites passed to it. It uses the method `getDiff()` to find the center of the object – the x, y, z-coordinates passed to it in XML are typically for the top-left-front corner.

$$tlf + diff = center$$

It uses the method `shape()` (which is abstract and implemented by each object) to set its `shape` field.

Ball

`Ball`s override the defaults to set `frozen` to `false` – unlike most other objects in the game, `Ball`s respond to gravity and other forces. `Ball`s are also specified with their

center, not a top-left-front corner, so `getDiff()` is overridden to return $ZERO$.

Flipper

`Flipper`s define `getDiff()` differently, depending on their orientation and whether they are left- or right-flippers. Both override `defaults()` to remove orientation – orientation is a required field – and to change the coefficient of reflection to 0.95.

2.2.3 Physics Package

GamePhysics:

GamePhysics computes the motion and collisions for a ball for one frame of animation. The algorithm pseudocode is:

```
do:
find the next collision
if it is before the end of the frame, then:
  move the ball until the collision time
  reflect
loop until the end of the frame
```

`GamePhysics` uses the helper class `Collision` to encapsulate the properites of a collision: the ball, the object, the shape (i.e. a polygon on the object), and the time of the collision. In some cases, the `reflect()` function still leaves the ball's normal vector pointing into the object. When this happens, we move the ball for a fraction of a frame and then resume normal physics operations.

PhysicsShape:

PhysicsShape is an interface implemented by the objects that are used as primitives to create Shapes. Those include PlaneCircle, PlanePolygon, Sphere, LateralCylinder, and Torus. The methods specified in PhysicsShape include getShapeClassification(), which returns the ShapeClassification of a PhysicsShape, as well as some methods useful in PhysicsCalculations shared by the implementing classes.

ShapeClassification:

ShapeClassification is an Enum that identifies the PhysicsShapes used by Physics and Rendering. Those include PlaneCircle, PlanePolygon, Sphere, LateralCylinder, and Torus. Torus is never rendered by the Renderer but it is used in physics calculations.

Shape:

A Shape object includes a bound, a PhysicsShape, and a List of PhysicsShapes. The List of PhysicsShapes is defined about the origin and describe what the Shape looks like. The Shape class has several static Shapes, which define the shapes of the Ball, bumper, and flipper objects. These Shapes are interned, i.e. there is only one copy of them. Shape also has static methods to return the Shape of any GameObject, including absorber, which is not interned because absorbers can be of arbitrary size.

2.2.4 Rendering Package

Renderer:

Renderer implements GLEventListener, which allows it to draw on the GLCanvas that it is registered with. Renderer also extends the abstract class ShapeRenderer, which has methods to draw the primitive PhysicsShapes. Renderer takes in a GameSpace and has a boolean parameter which indicates whether it is in build mode or play mode. Because Renderer implements GLEventListener, it implements four methods: init(), display(), displayChanged(), and reshape(). displayChanged() and reshape() are empty for this renderer.

The init() method of Renderer is only called once at the very beginning and sets up the parameters. It configures the lighting for the scene and the camera angle.

When the display() method of the GLCanvas with which an instance of Renderer is registered gets called, the display() method of Renderer draws the scene. It does this by going through all the GameObjects in GameSpace, taking their Shape, and drawing each of the PhysicsShape primitives for each Shape. When playMode is set to false, the Renderer also draws a 20L by 20L grid in the $z = 0$ xy plane. The display() method also tries to load textures at the beginning of every time it is called, but textures are only loaded in the beginning of each game or when skins are changed. Renderer

stores textures in a Map keyed by GameObjectClassification, which means that each GameObject always has a uniform texture all over.

ShapeRenderer:

ShapeRenderer is the class with methods to draw the primitive PhysicsShapes. We put these methods in ShapeRenderer instead of Renderer for extensibility reasons. If we had decided to implement drag and drop in the GLCanvas, we would have used another renderer, an OverlayRenderer, which draw the same PhysicsShapes. Even though we only have one Renderer that draws on GLCanvases, putting the methods that draw PhysicsShape in a separate class clears the cutter in the Renderer's code a bit.

Textures

The Textures class has static Maps of GameObjectClassification as keys and Strings of image locations as values. These maps are interned much in the way that static Shapes in the Shape class are interned. Note that the Textures class is different from the Texture object, which actually represents a Texture.

2.2.5 Swingui Package

MainGUI

MainGUI is the graphical user interface of our Gizmoball game. It keeps track of a GameSpace, a Renderer, a Timer, and a boolean, buildMode, indicating whether it is in build mode or play mode. The buildMode of boolean should always be consistent with the buildMode of MainUI's Renderer.

Visually, MainUI can be broken down into four parts: the GLCanvas, the FileMenu, the AddPanel, and the DetailPanel. The GLCanvas, canvasPanel, is where the game is rendered with Renderer. MainUI has a field for a MouseListener, currentListener, which keeps track of the MouseListener registered with canvasPanel. MouseListeners are used to add, move, delete, and select GameObjects in the GameSpace displayed in canvasPanel. Only one MouseListener may be registered with canvasPanel at a time. See next section on AddPanel for further discussion of these listeners.

The MainUI's Timer object, animation, is responsible for calling display() on the

GLCanvas and for calling GameSpace's stepFrame() method to update the physical model. In play mode, animation calls stepFrame() and display() 20 times per second. In build mode, animation only calls display() and not stepFrame().

The FileMenu of MainUI allows the user to start constructing a new game in build mode, load an XML file, and to save a game. Note that saving can be done both in build and play mode. Thus, a user can save the state of an existing game and pick up playing exactly where left off. The FileMenu also allows the user to change the look of the game by loading different textures. This is done by changing the texture map of MainUI's GameSpace.

AddPanel

AddPanel allows the user to add, move, delete, and select GameObjects in the GameSpace. AddPanel has a field for the MainGUI that it is a part of and a field for the currently selected GameObject. There is a JButton for adding each of the GameObjects and JButtons for move, delete, and select. The action for each button is done in a MouseListener, which can be registered with the GLCanvas of AddPanel's MainGUI.

AddPanel has a ButtonListener, which figures out which button was pressed. Every time a button in the AddPanel is pressed, the ButtonListener unselects any previously selected item and sets selected to null. It then unregisters the MouseListener currently registered with the MainGUI's GLCanvas and registers the MouseListener associated with the pressed button.

Each MouseListener performs an action indicated by its button. All the Add listeners listen for a MouseClick in the GLCanvas, and checks if the grid is occupied. If the grid is not occupied, it creates a GameObject and adds it to the GameSpace.

The MoveListener's first checks if selected is null after a MouseClick. If so, it "selects" the object in the clicked grid and highlights it, if there is an object. If selected is not null when the MoveListener hears a MouseClick, it moves the selected object to the clicked location, unless the clicked location is occupied. Note that the selection done by *Move* is different from the selection done by *Select*. The DeleteListener checks if a clicked grid is empty or not. If the grid is not empty, it deletes the object currently in the grid.

After a MouseClick in the GLCanvas, SelectListener first clears the currently selected object if there is one. Then, it checks if the grid clicked on is empty. If so, it selects the object by highlighting it and displaying the DetailPanel for it.

DetailPanel

The `DetailPanel` extends `JPanel` and contains one `JTabbedPane`. It is used to display detailed properties of a selected objct in the `GameSpace`. The `DetailPanel` displays only those panels that are relevant to the object under inspection. Thus, `Ball`s only have show the velocity panel, whereas gizmos have delay, orientation, target, and trigger panels; only `Absorber`s show the absorber panel.

OrientationPanel listens to a `JComboBox` in order to change an object's orientation.

DelayPanel uses a `JSlider` to set a gizmo's delay from 0 to 10 seconds.

TriggerPanel displays a list of buttons that represent the key triggers currently registered for this `GameObject`. Clicking on a button removes that trigger. If a user wishes to add a trigger, this pops up an AddKeyWindow, which listens to keypresses and adds a given trigger when the user presses Select.

TargetPanel displays a list of buttons that represent the targets of this `GameObject`. Clicking on a button removes that target. The add target functionality is implemented with a class called `AddTargetListener`, which is a `MouseListener` that adds the clicked object to the first object's targets list and unregisters itself.

AbsorberPanel informs the user whether or not the `Absorber` already contains a ball or not. If it does, it displays a button that the user can click to remove it; if not, it displays a button the user can click to generate a new ball inside the absorber.

2.3 Module Dependency Diagram

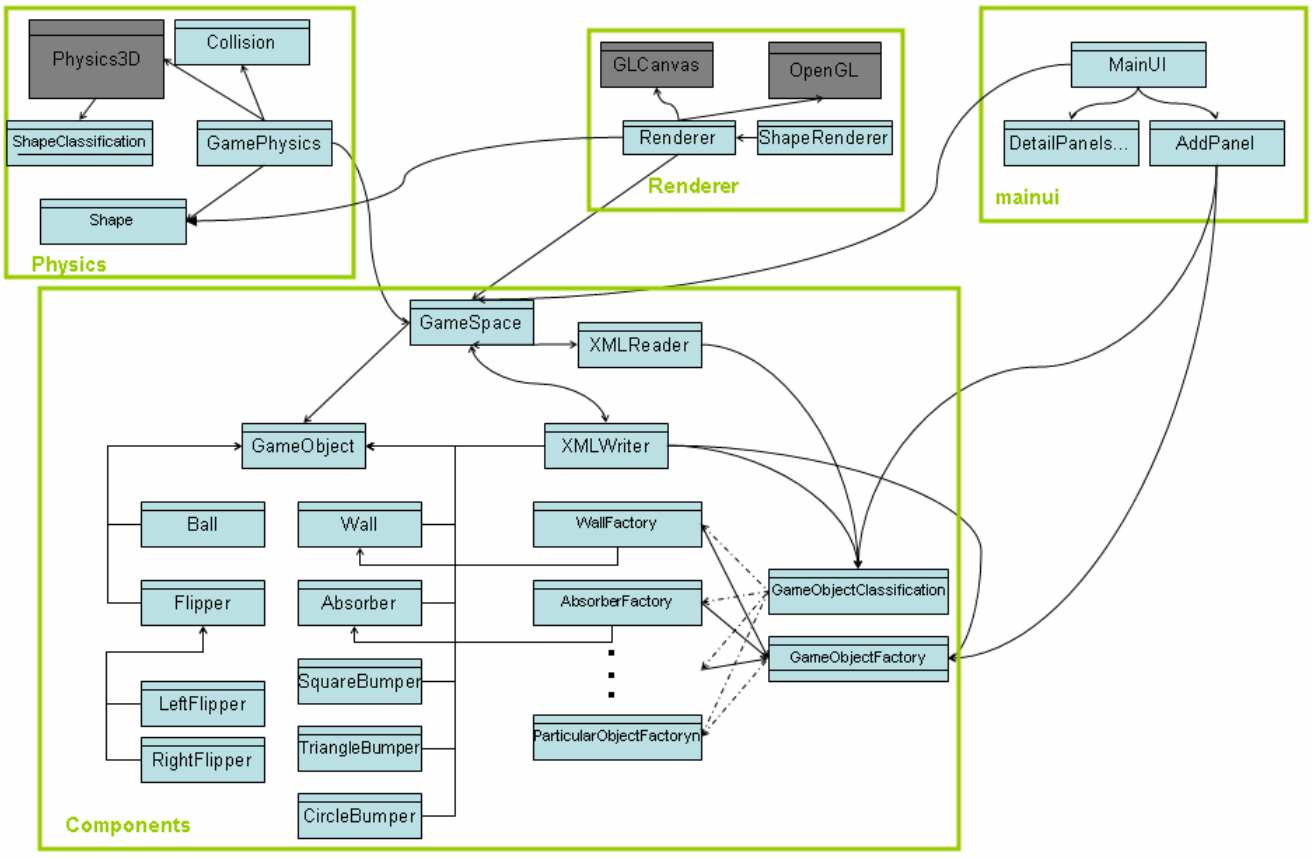


Figure 5: Module Dependency diagram (some less important classes are not shown)

3 Testing

3.1 Strategy

It is difficult to isolate many parts of system, such as the rendering and the physics engine. Thus, our testing strategy focused largely on integration testing. We did write some unit tests for the Components package for the preliminary release to test the functionality of commonly used methods. Our integration tests consists of a series of

XML files targeting specific aspects of our system. A list of these files along with their description can be found in the appendix. In addition, a large part of our testing involved trying out the system by hand and observing the result. Many of the bugs that we fixed were discovered this way.

We used both bottom up and top down testing. In the beginning, we started with unit tests on the components. We then moved on the integration tests. However, when our integration tests failed, we sometimes went back to write more unit tests to make sure that the basic components worked.

Most of our testing was black box testing, even for our integration tests. For each test, we had a specific specification in mind to test for.

3.2 Results

3.2.1 Components

We ran many unit tests on the game components before the preliminary release and are very confident that the system works. After we change the constructor structure, we could have done more testing on the new constructors.

3.2.2 Physics

The Physics3D library was already tested for us. To test our physics controller, we wrote some unit tests, which made sure that what our physics gave was not way off. Most of our testing for physics was done through integration testing with XML files. Through these, we found many bugs, almost all of which we were able to fix.

3.2.3 Rendering

Rendering testing is covered in our XML implementation tests. We made sure that all objects were drawn at the correct places with the correct orientations and surface normals. We are very confident that our rendering engine works properly.

3.2.4 SwingUI

Most of the testing for this package was done by hand. We went through the user interface and tried as many things as we could think of to test its functionality. We are fairly confident that our tests covered most of the possible strange things that a user could do with our GUI.

4 Reflection

4.1 Evaluation

In general, our team was successful in fulfilling the requirements of the project. In addition, we also implemented extra features the went beyond the requirement, such as multiple ball interactions and textures. Our renderer is extremely versatile and can support the rendering of any gizmo built from the basic primitive shapes in any position and orientation. The detail panel of our GUI is also particularly attractive because it fits a lot of information and functionality in a limited amount of space.

We realize that our system is not perfect. Our physics engine generally simulates the correct behavior but occasionally causes strange interactions. Our components structure could also have been designed differently to increase modularity, extensibility, and cleanness of code.

4.2 Lessons

4.2.1 What We Learned

A working system above all else:

We spent a lot of time toward the beginning of the project trying to think of the best way to make our system as extensible as possible. We tried to think of as many edge cases as possible, sometimes cases that we would never have to implement. Two things occurred as a result. First, we became pressed on time toward the end of the project because we focused too much time thinking of optimizations rather than implementing.

Second, the final design that we settled on is not the best for the specifications that we had to implement because we tried to consider too many hypothetical cases.

Testing before committing:

About a week before the final deadline, a team member committed some changes without testing them and caused the entire system to fail. Testing may not be the most convenient or gratifying aspect of project development, but it is extremely important. Along the same lines, we should probably have written more unit tests, which would have simplified debugging at the end.

Always keep up to date

During the final project, sometimes we forgot to update before a work session, which sometimes caused CVS synchronization issues, which took time and were annoying to fix. As a corollary, each team member should have had a better understanding of other team members' code. We had several instances where by the time one team member discovered an inefficient design design made by another team member, it was already too late to change the system.

4.2.2 Alternative Designs

Components Package

Currently, there is nothing distinguishing a gizmo from a ball. We should have included a Gizmos abstract class that is a subclass of GameObject to group all gizmos together. That way, GameObject would not need to have so many fields that are often unused or unnecessary in its subclasses. For our preliminary release, we divided GameObject into Fixed and Mobile objects. That was a result of us anticipating a possible extension involving other types of moving objects. Instead of getting rid of the Fixed and Mobile distinctions entirely, we should have modified them.

Within GameSpace, we currently store all GameObjects together in one set. We should have stored Ball separately from other GameObjects. Actually, for our preliminary design, we separated the storage of Fixed and Mobile objects. When we got rid of the Fixed and Mobile objects, we grouped all GameObjects together without thinking

through the implications. Because Ball is stored along with all the other GameObjects, retrieving the Ball(s) from the GameSpace takes $O(n)$ time, where n is the number of GameObjects in the GameSpace. This is extremely inefficient.

Another thing that we could have done was to hash stationary objects by position. That way, during each time frame, the ball only has to look at the objects close to it to check for collisions as opposed to checking every object in the game space.

4.3 Known Bugs and Limitations

4.3.1 Bugs

Balls going through surfaces:

When we have multiple balls, balls go through walls on rare occasions. This happens when constant pressure is applied to the ball from multiple sides. We are not clear what the root case of this is, but because it happens so rarely and because multiple balls is not actually a requirement for the system, we did not pursue the bug further. To see an example of this bug, load the bouncing balls xml file. The balls bounce on top of each other, but once they loss energy the a ball is pushed between the ball above it, and the wall below it, the bottom ball pops through the bottom wall, disappearing from the gamespace.

Game freezing:

Very rarely, our game would freeze after running for a few minutes, usually if there are too many objects with curved surfaces in the game space. From System.out.println's, we suspect that the cause of this is GamePhysics caught in an infinite loop. After some debugging, this seem to happen when an object reflects off of a surface, into the same surface again. Real world physics says this cannot happen, but we some suspect some strange numerical artifact is making this happen in Physics3D.

4.3.2 Limitations

GUI

Our GUI actually does fulfill the required functionality. These limitations simply indicate what we would have liked our GUI to do in the ideal case. Even though our system supports full 3D movement and rendering, our GUI does not allow for full 3D editing. This is because the interface would have been very complicated, both for the programmers and for the users. We actually did come up with a way to simplify full 3D editing, but the math for selecting objects in the z-dimension did not work out, and we did not have time to fully investigate the issue.

Also, our build mode only supports building new files. Given an imported XML file or even a previously built file after running, our build mode may not be able to select objects in the existing level. Users may still add to an existing level and freely modify the newly added objects.

5 Appendix

5.1 XML Format

Taken from the official 6.170 Gizmoball project description.
For more detail, see http://web.mit.edu/6.170/www/assignments/gizmoball/gizmoball.html#file-format

```
<board properties...>
      <ball>
      <ball>
      <ball>
      ...
      <gizmos>
          <gizmo types ..... >
          ....
      <gizmos/>
      <connections>
          <connect>
```

```
        ...
        <keyConnect>
        ...
    <connections/>
</board>
```

The main difference from the xml spec given to us is that we support multiple balls before the gizmo tag. For an explanation of the additional attributes our XML format supports, see the section of revised specification.

5.2 Module Specifications

5.3 Test Cases

A List of our XML files with a brief description:

1. **absorber_contains_ball.xml**

   Tests that a ball can be placed into an absorber in build mode and that the ball moves along with the absorber when the absorber gets moved.

2. **absorbertest.xml**

   Tests that the absorber can shoot things out on a key trigger.

3. **bellcurve.xml**

   Tests multiple ball interactions and interactions with many circular bumpers. It also looks cool.

4. **bounce_balls_test.xml**

   Tests that balls bounce properly over square bumpers and walls

5. **bucket_of_balls.xml**

   Tests that balls bounce properly with triangular bumpers and with each other.

6. **delay_flipper_test.xml**

   Tests that flipper can have delay on their key triggers. When the space bar is pressed, the right flipper flips after a few seconds, which is followed by the left flipper flipping.

7. **flipper_orientation_test.xml**

   Test level with 4 left flippers and 4 right flippers, each in a different orientation. All flippers are triggered with the space bar on key up and key down. Tests whether the flippers in different orientations behave appropriately when triggered.

8. **flipper_test.xml**

   Checks that moving-flipper and ball interact correctly.

9. **momentum.xml**

   Contains a row of balls with one ball at the end that has initial x velocity. The ball with x velocity hits the other balls and a ball at the other end pops out.

10. **orient_triangle_test.xml**

    Tests that build mode allows triangle bumpers to be placed in any orientation.

11. **pong.xml**

    Shows off the full 3D physics interactions.

12. **round_object_test.xml**

    Tests that balls interact properly with circular bumpers.

13. **zgravity_test.xml**

    Shows that gravity in the z coordinate works properly.