6.170 Final Design

Se021: Edmond Lau, Chang She, Vincent Yeung, Min Zhang 11-26-02

Requirements

Overview

The final project is a program that plays GizmoBall. Gizmoball is a version of pinball, an arcade game in which the object is to keep a ball moving around in the game, without falling off the bottom of the playing area. The player controls a set of flippers that can bat at the ball as it falls. The advantage of Gizmoball over a traditional pinball machine is that Gizmoball allows users to construct their own machine layout by placing gizmos (such as bumpers, flippers, and absorbers) on the playing field.

There are two kinds of flippers: left flippers that flip counterclockwise and right flippers that flip clockwise. There are also three kinds of bumpers that the user can place on their own customized board: circular, square, and triangular bumpers. Additionally, the user can add as many balls as he wants to the board. The user can also rotate flippers and bumpers or move any gizmo to another location on the board. He can also construct triggers by connecting one gizmo with another so that when one gizmo is activated, the other is as well. For example, the user can connect a bumper with a flipper so that when the ball hits the bumper, the flipper automatically flips up. He can connect keypresses with gizmos as well so that for example, he could push the "f" key to flip a certain flipper on the board.

Finally, the user can save and load boards. There are some pre-designed boards that come with GizmoBall for the user to begin playing immediately.

Revised Specifications

<u>General</u>

The number of pixels per length L in the display will be set and made constant.

Multiple balls will be allowed and supported; this was not entirely obvious from the initial requirements.

<u>Playing Area</u>

No change/assumption made

<u>Building Mode</u>

To get the keys to connect correctly to the gizmos, the user must enter xset –r on the command line.

Running Mode

Frame rate will be approximately 20 frames per second. That is, the display will be refreshed approximately 20 times per second to support smooth animation. The internal physics modeling (movement of balls given their velocities, collision detection, and triggers) will update at discrete time intervals, which will be at least as frequent as the display refresh rate.

Changes in velocity due to gravity and friction are applied on small discrete time intervals. Half of the effects of gravity and friction are applied at the beginning of every cycle and the other half is applied at the end of every cycle. This is done so that the ball bounces to the same location every time it hits the outer wall, rather than bouncing to variable locations depending on when gravity was applied during that bounce. Both gravity and friction are approximated by their linearized forms for small changes in time. The number of physics loops run per turn is 10, and there is a frame update every 50 seconds.

The maximum ball velocity of 200 L/sec can be impractical since our frame rate will not be able to keep up with it (ball jumps up to 10L per frame). Furthermore, at such high speeds, there could potentially be slowdown with the collision detection mechanism. However, the physics generally should support such high speeds even if the display can't keep up with it.

Standard Gizmos

No change/assumption made

Gizmoball File Format

No change/assumption made

Problem Analysis

Underlying Problem

The underlying problem is finding a way to model the behavior of the Gizmoball system. In Gizmoball, there can be many gizmos and balls on the board. Additionally, there are walls lining the outer edge of the board and a physical system that governs the way the ball moves around the board. There is also a trigger system that allows a gizmo to be triggered by key presses or other gizmos.

We chose to analyze this problem by first describing the objects that Gizmoball contains. The bumpers, absorbers, and flippers can be categorized as "gizmos" because they are similar in that they cannot move around while Gizmoball is running and that they change the velocity of the ball upon impact. The outer walls can also be categorized as gizmos because they behave similar to a gizmo in that they cannot move, and that they change the velocity of the ball upon impact. The ball can be categorized separately storing fields for its location and velocity.

After coming up with a list of objects, we had to analyze the system more in depth for other entities. First, we chose to label Board the container for the gizmos and balls, and Display the interface that allows the user to view all the gizmos and balls. The physical system that governs

the way the ball moves around the board was labeled Physics Loop and the set of triggers that system contains was labeled the Trigger System. Additionally, there are "triggers" relations between key presses and gizmos to represent the triggers as well.

Conceptual Model

The abstract object model describes one way to think about the problem. The domains and their corresponding relations are:

Display — Display describes the set of interfaces that can be viewed by the user. In Gizmoball, only one display will be used. Display is used to show objects from our back-end to users. The user interacts with the display since it is the only front-end part of our model. The rest of the domains are not visible to the user. Display is related to board in that it shows the board to the user.

Board — Board describes the set of containers for various objects. Only one board will be used. The board stores objects and has handles to those objects. Board is related to Gizmo and to Ball by the contains relation in that a board stores a set of gizmos and balls. Board is also related to Location by the grid-of relation since a board is simply a grid (in this problem, a 10x10 grid) of locations.

KeyPress — KeyPress describes the set of keys on the keyboard. A key press is the action of hitting a key on the keyboard. A key press is related to a gizmo in that the key press can trigger the action of the gizmo. Gizmoball can contain a large amount of key presses. In fact, any key on the keyboard can be connected to any triggerable gizmo allowing that gizmo to be triggered when the key is struck.

Gizmo — Gizmo describes the set of devices that can be added to a board. Adding these devices changes the way the way the ball bounces on the board. Bumpers, Absorbers, Walls, and Flippers form a complete subset of all gizmos. These gizmos each have their own individual properties. A gizmo is related to another gizmo by the triggers relation in that a gizmo can trigger the action of another gizmo similar to the way a key press triggers a gizmo. A gizmo is related to the location domain by the located-at relation, describing its specific location on the board.

Location — The set of all locations. In Gizmoball, a location is represented by a grid square. The grid square can store only one gizmo but it can store as many balls that fit within the area of the grid square.

Ball — The set of balls. A ball is a circular object that moves around the board depending on the physics of the board. A ball is related to a location by the located-at relation, describing its specific location on the board.

There are other sets that are simply subsets of the Gizmo domain:

Bumper — A bumper is a specific kind of gizmo that has the physical property that whenever a ball hits it, it simply bounces off in the appropriate direction. Bumpers can have different geometries. The only geometries represented in Gizmoball are square, triangle, and circle. Bumpers can be rotated but only the triangular bumper's physical properties will change upon rotation.

Flipper	A flipper is similar to a bumper in that the gizmo can simply bounce off the flipper. However, flippers can be triggered by key presses or other gizmos. When a flipper is triggered, it flips 90 degrees in one direction. Thus, if there is hits a ball while it flips, it changes the velocity of the ball. There are two types of flippers: left flippers and right flippers. Left flippers rotated counterclockwise when triggered while right flippers rotate clockwise. Flippers can also be rotated into four different positions within the grid square.
Absorber	An absorber swallows a ball when the ball runs into it. Absorbers can be created of any arbitrary size of grid squares. An absorber can store a ball and when triggered, will release that ball, shooting it out at a certain velocity.
Wall	A wall acts similar to a bumper except it is only placed on the outer edges of the board. When a ball hits a wall, the ball bounces off similar to the way it bounces off a bumper. Walls cannot be rotated or triggered.

Other Contraints

There are several constraints that must be imposed on the problem domain. First, the ball has both size and velocity. A ball can have any location while a gizmo can only have integer locations.

Another constraint is that outer walls cannot be triggered by other gizmos or key presses. There are only four outer walls, one on each of the four sides of the board. The outer walls cannot be modified by the user but still behave as a gizmo.

Finally, balls and gizmos cannot share a location unless the gizmo is an absorber. This prevents instances where our display is confused about which object to display when they share the same location. However, it does allow absorbers and balls to share the same location since this was given in the specs; thus, the display should only show the absorber and not the ball that shares its location. The absorber thus can store the ball and fire the ball when the ball shares a location with the absorber.

Alternative Models

An alternative model was to get rid of the notion of a board completely and have gizmos and balls store their locations internally. This idea was rejected because it made it much more accessing the contents of a particular grid location a linear search. Furthermore, by having a board, we can decouple the display from the actual gizmos and balls.

Another alternative model was to have outer walls as a separate domain rather than having them as a subclass of gizmo. This idea was rejected because it unnecessarily created more different objects that the board would have to store. An outer wall behaves exactly the same as a gizmo except its location cannot change and it cannot be triggered. However, the physical properties are the same.

The abstract object model can be found on the following page:

User Manual

Overview:

This application runs in two modes, building and running; building mode is turned on when the application is started. The user can toggle between the two modes by clicking on the stop and run buttons.

Building mode (on by default; enable by clicking stop button):

- The user is able to construct their own GizmoBall board by placing various gizmos (square, circular, and triangular bumpers, absorbers, left and right flippers) and balls at different locations.
- By right-clicking a placed gizmo and opening up a pop-menu, one can move, rotate (applies only for triangular bumpers and the flippers), or delete the gizmo.
- Similarly, balls can be moved and deleted, but they can also have their initial velocity changed from the default of 0 L/sec in both the horizontal and vertical directions to values of up to 200 L/sec in magnitude.
- In build mode, the user can save and load board configurations they have constructed in the standard GizmoBall file format.
- The following features have not been implemented in this preliminary release but will be available in the final release. The user can assign an existing gizmo as a *trigger* for another gizmo (possibly the same one). It is also possible to set *keypress triggers* to a gizmo. Finally, the gravity and friction constants can be changed from their default values.

Running mode (enable by clicking run/lightning button):

- Balls will move according to the physics of gravity, friction, and any collisions with bumpers, flippers, other balls, and the bounding outer walls. A ball that flies into an absorber will be absorbed and released only when the absorber is triggered.
- Pressing keys on the keyboard corresponding to previously connected keypress triggers will trigger the appropriate gizmos.
- In run mode, building operations (e.g. adding gizmos and balls, saving/loading), will be disabled.

Assumptions:

- GizmoBall should be run on the *Athena* platform at MIT on accounts with access to the se021 group directory. It may be possible to run GizmoBall on other platforms, but no guarantees are made about how the user interface icons will appear in that case.
- Repeating key events when a key is held down should be disabled by executing `xset -r` on Athena.
- For performance reasons, it may be preferable to run GizmoBall on more modern Athena workstations.

Design:

Runtime Structure

The code object model is included in the following page.

Here is a list of additional textual constraints not directly expressible in the code object model:

For any instance gbg of GizmoBallGUI:

- gbg.TS = gbg.board.TS = gbg.physicsLoop.TS
- gbg.display.gbgui = gbg
- gbg.physicsLoop.board = gbg.Display.currentBoard = gbg.board

For any instance d of Display

- d.gbgui.display = d

For any instance p of PhysicsLoop

- p.reflector is in p.board.gizmos or p.board.balls
- p.ballToReflect is in p.board.balls

For any instance b of Board

- b.gizmos contains { OuterWall.NORTH, OuterWall.WEST, OuterWall.EAST, OuterWall.SOUTH }
- b.gizmos contains every Gizmo in b.TS.producers
- b.gizmos contains every Gizmo in b.TS.observers
- for any Absorber a in b.gizmos: if a.myBall!=null, then a.myBall is in b.balls
- for all g1!=g2 in b.gizmos, g1.locations does not intersect with g2.locations
- for any g in b.gizmos of type CircularBumper, TriangularBumper, SquareBumper, LeftFlipper, RightFlipper, and Absorber, all Vect v in g.locations and g.corner must satisfy the following: 0<=v.x()<=19 && 0<=v.y()<=19

For instances of each of CircularBumper, TriangularBumper, SquareBumper, LeftFlipper, RightFlipper, and Absorber, the sizes of both circles and segments are fixed.

Module Structure:

This section documents Gizmoball's syntactic structure as described in the attached module dependency diagram and elucidates the justifications behind the design choices and design patterns employed for the purposes of decoupling, extensibility, partitionability, and similar software engineering goals.

Interfaces:

The introduction of the Gizmo interface, which is implemented by the seven standard gizmos, provides substantial decoupling between individual gizmos and the Board as well as between individual gizmos and the PhysicsLoop. The main design goal of this interface is that no other modules besides the GizmoFactory will ever need to depend on particular implementations of the Gizmo interface. For example, in adding, removing, editing, or rotating gizmos, the Board only needs to interact and call methods on the Gizmo interface and not the individual gizmos. Furthermore, the Physics Loop can retrieve the geometric model representation of the gizmos and resolve collisions with a mere dependence on the generic Gizmo interface. The generic interface allows for simplicity of coding by the client of Gizmos since all Gizmos share the same methods; it also provides the added benefit of system extensibility, since new Gizmos can be easily added with modifications occurring only in GizmoFactory.

The Drawable interface decouples the back-end Gizmo and Ball modules from the front-end Display module. In particular, since the Display module only needs information from the Gizmo and Ball regarding how to draw and clear each one on the display, the Display does not need to know about any other functionality of Gizmos and Balls. This interface also provides system extensibility since the Display module then only has a weak name dependence on the Gizmos and Balls; thus, if any modifications occur in the backend, the front-end Display module can be readily updated.

The Triggerable interface decouples the Gizmos from the TriggerSystem. All registration, unregistration, and triggering of actions is completed with the one method of the Triggerable interface. For more information detailing the usage of this interface, refer to the paragraph regarding the blackboard design pattern in the next section and to the section titled TriggerSystem.

The Collidable interface, implemented by the balls and the Gizmos, provides a uniform interface by which the PhysicsLoop can control the interaction among the components on the board. In particular, since all collision detection and collision resolution is actually delegated to the objects themselves, the PhysicsLoop does not need to know whether the objects it is detecting collisions for is a ball or a gizmo; it only needs to rely on the Collidable interface for collisions.

Design Patterns:

The factory object design pattern employed by the GizmoFactory decouples the front-end GUI from the back-end Gizmo objects. Since the GUI must have the functionality to add Gizmos to the board, the absence of a factory object would generate an undesirable module structure: either 1) the GUI must perform direct calls to the individual constructors of different gizmos, creating undesirable dependences from GUI to Gizmos, or 2) the Board API must be narrow and contain

gizmo-specific methods for adding objects, thereby creating excessive overhead. The factory object pattern eliminates this necessity and instead allows the GUI to simply create and pass in a GizmoFactory to the board for the addition of objects. This design choice also facilitates future extensibility if additional Gizmos are to be added.

The blackboard design pattern utilized by the TriggerSystem decouples Gizmos that are connected together by triggers and actions. In essence, when the user signals to the GUI that two gizmos are to be connected, the GUI notifies the Board to register the pair of Gizmos on the TriggerSystem blackboard; at that point, the Gizmo that performs actions is added as an observer to the TriggerSystem. The incorporation of the Observer pattern using the Triggerable interface further provides decoupling since the TriggerSystem does not need to know the type of consumers that are receiving the messages. The producer, in this usage of the blackboard pattern, will primarily be the PhysicsLoop, which needs to leave a message on the blackboard during its collision resolution phase whenever a collision occurs; if the object that is collided with is registered as an observer on the TriggerSystem, the object reads the message off of the blackboard and behaves accordingly. The absence of such a blackboard would mean that the a Gizmo itself would need to store information regarding which other Gizmos it is connected to, which is once again a dependence to be avoided.

Another powerful tool that increased the simplicity, extensibility, and partitionability of the backend structure is the Template Method. The utilization of this design choice rests on the recognition that different Gizmos shared similar implementations for many common methods, especially those methods in the Collidable interface, combined with the recognition that the same basic Gizmo files would need to be continually modified by different members at the same time, which increased the difficulty of partitioning the work without bumping into the conflicts due to the line-by-line checks of CVS. To resolve this issue, an AbstractGizmo class with template methods for the collisions and default implementations for the hook methods necessary for the collision detection and resolution of Gizmos with zero angular velocities was introduced. All Gizmos by extending AbstractGizmo would thus have those methods implemented by inheritance; only the flippers, when they are rotating, and the absorber need to override the minor hook methods. The benefits of this design choice are far-reaching. Code for collision detection and resolution was simplified and cut down by nearly a factor of seven due to inheritance; furthermore, work could be more easily partitioned with less fear of CVS conflicts now that the bulk of the collision detection and resolution occurs in a separate file.

The module dependency diagram can be found on the following page.

The Physics Loop:

The physics loop serves as the main module for modeling the physical interactions between the Balls and the Gizmos as they move and collide around the board. To perform relevant calculations, it relies on the procedural abstractions provided in the physics package. The physics loop depends on the Board for the sets of gizmos and balls, the Collidable interface so that it can manage the motions and interactions between balls and gizmos, and the Trigger System so that gizmos with triggers can be notified to perform the appropriate actions. This section documents how and in what order motion, collision detection and resolution, triggering, friction/gravity, drawing, and other factors are handled.

Although the frame rate is 20 frames per second, the physics loop runs every 5 milliseconds, or ten times per frame. At the beginning of each loop, the effects of gravity and friction over the course of 2.5 milliseconds are applied using simple mathematical formulas; at the end of each loop, after the path of the balls has already been charted, their effects over 2.5 milliseconds are applied again. The earlier decision to run the physics loop at the frame rate and to adjust the velocity of the ball for gravity and friction at the beginning of each loop actually turned out to be too coarse of an approximation, causing the balls intolerable aberrations in the movement of the ball. The further subdivision of the frame rate by 10 to obtain the physics loop's running rate combined with the application of gravity and friction at the beginning and end of each loop better models the continuous effects of gravity and friction. This design choice for slowly incrementing the effects of acceleration linearizes the ball's trajectory but still maintains the integrity of the physics because 2.5 milliseconds is a sufficiently small time period.

After the initial application of gravity and friction, the physics loop then enters into the collision detection phase. For each ball, the physics loop iterates through all of the gizmos on the board, obtaining the set of geometric shapes (i.e. circles and line segments) that model the gizmo and calculating the time until collision with each of the shapes given the ball's current trajectory. The physics loop next calculates the time it takes (if any) for two balls to collide based on their current trajectories. The calculation of these times until collision occurs through delegation to the balls and gizmos; the physics loop calls the balls' and gizmos' common methods of the Collidable interface, which utilize the timeUntilCollision() methods of the physics.Geometry class. From all of these times, the physics loop takes the minimum time to collision. If this minimum time is greater than the time remaining in the physics loop, the physics loop simply updates the positions of the balls and the gizmos based on their current velocity and angular velocity, respectively, in the passage of the remaining time; this session of the physics loop then terminates. If the minimum time to collision is less than the time remaining, however, the physics loop proceeds to the collision resolution phase.

In the collision resolution phase, the physics loop first uses the balls' velocities and the gizmo's angular velocities to update the positions of the balls and the gizmos based on their respective velocities and angular velocities, due to the passage of the minimum time calculated in the collision detection phase. At this point, a ball and the object it is about to hit are exactly adjacent to one another. Next, the physics loop once again delegates responsibility to the object the ball is about to hit using a method in the Collidable interface, which updates the ball's velocity based on its collision with the object by employing the reflect() methods in the physics.Geometry class. The physics loop then leaves a message on the Trigger System blackboard, so that the object

connected to the trigger (if any) can pick up the message and activate itself. Further discussion of the Trigger System can be found in the Trigger System section. Finally, if time still remains in the current 5 millisecond loop, the physics loop repeats the collision detection phase until no time remains; otherwise, the physics loop terminates.

As mentioned above, the display phase occurs once for every 10 runs of the physics loop. Currently, the entire display is redrawn for each frame, which works fine for board customizations that are not too complex. However, for the final release, part of the project plan includes optimizing the display phase redraw only those objects that have changed state in the 10 runs of the physics loop in order to support more complex board configurations. This can be implemented by creating a change log mediator between the physics loop and the display. The physics loop will communicate to the log the objects that have changed state, and the display will retrieve from the log the largest bounding rectangle that encompasses the objects that have changed state and only redraw that rectangle.

Trigger System

The TriggerSystem employs a form of the blackboard design pattern. In our model, there are multiple trigger producers and observers; with the current application requirements, these producers can be Gizmos or key presses, and the observers are Gizmos. A Triggerable interface is used to help decouple TriggerSystem from Gizmos and also allow room for changes. For example, consider what might happen if we were required to make Balls trigger observers as well. Having a Triggerable interface makes the change a matter of just having Ball implement the interface, and no change in TriggerSystem would be needed.

While our direct application of the blackboard pattern may seem to work relatively flawlessly at first, in actuality there are some subtle problems associated with situations where triggering is done too quickly in short periods of time. The system is then susceptible to unpredictable behavior and even crashing despite the fact that all our methods should work in theory. While this problem has not happened much so far in our preliminary release, we plan to, in an effort to make our application more robust, implement some form of a triggering queue in the TriggerSystem that will queue and possibly delay trigger events as they come in order to make sure that these events are not propagated to the observers at inappropriately high speeds. Then, our Triggerable classes will be able to behave the way they were planned to when their triggerAction() methods (required for Triggerable) were implemented.. We believe this solution involving a triggering queue will solve the problem of error-prone rapid triggering without having to compromise too many other issues, if any exists.

Testing:

Validation Strategy:

Overall, the testing of Gizmoball followed quite closely a bottom-up approach, whereby individual units were rigorously tested prior to both integrating those units together and testing more complex modules that depended on the smaller units. This section documents in relatively chronological order the validation strategy adopted for various modules of GizmoBall and the justifications behind test-related decisions.

In the GizmoBall design, the most fundamental units are the Gizmos and Balls since they can be tested independently of all other modules; consequently, most of the early unit testing focused on them. In testing these atomic components, a strong three prong-standard was adopted for selecting tests: 1) black-box testing that tested all of the obvious subdomains and also some of the more oblique ones; 2) glass-box testing with over 90 % branch and decision coverage and strong condition coverage, with special emphasis on the collision-related methods and the movement and rotation methods of more complicated gizmos such as flippers; and 3) regression testing, including making new test cases and checking against old ones, whenever any major changes were made in any of the basic units. These tests were performed using automated JUnit test suites which efficiently verified the algorithmic correctness of the code and illuminated obvious as well as some of the more subtle logic errors. Some degree of peer testing whereby a different team member tested the code of another team member also occurred, since one often overlooks errors in his own code. The motivation for a strong focus on the testing of these atomic units should be apparent. All of the more complex units strongly depend on the correctness of these unit modules; furthermore, considerable time is saved when bugs do not need to traced as far back as these fundamental building blocks.

Subsequently, the Board, which integrated the gizmos and the balls, was tested using a JUnit test suite. Statement and branch coverage of this module closed in to around 80%. Primarily, the difficulty in validating this module occurred in determining whether elements overlapped or not. Thus, testing the board involved first constructing an initial configuration on paper, setting it up in JUnit, and then trying to break the code by adding or moving Gizmos and balls into locations that overlapped with other units. This strategy would reveal the logic bugs that can occur in implementing procedures to determine overlaps.

To facilitate later automated testing, the GizmoParser module which was in charge of saving and loading files using the standard file format was tested. Perhaps the simplest to validate, testing involved reading in the input for a board, setting up the board, saving the board, and finally visually checking the input and the output text files to verify that they are indeed the same.

To perform integration testing on the PhysicsLoop module using pure automated JUnit test cases proved to be infeasible. There were simply too many variables to take into account, and would require manually planning a scenario on the board and calculating where each object would be after the passage of a certain amount of time or after a certain number of collisions. The complexity of the physics made the module difficult to validate. Thus, an alternative testing strategy was adopted. After the basic add and get methods of the board were tested, a display

prototype was created. Then, a simple test driver that added some balls and gizmos to the Board was animated by running the PhysicsLoop with the Display prototype. This revealed the obvious bugs in the PhysicsLoop and revealed any unreasonable approximations made by linearizing the effects of gravity and friction.

While the backend was being tested, the front end GUI was also being tested quasi-independently (independently in terms of the aesthetics but dependent on the backend for the functionality). The nature of GUI design precluded systematic JUnit test cases. Instead, the tests required manually running through the program and testing the correctness of two areas, that A) everything looks correct and B) the functionality of items such as menus and buttons was correct. In fact, the design process reflected two similar major stages: A) getting the GUI and Display modules to display correctly, and B) getting the GUI and the Display modules' event handlers to function correctly. Thus, in the beginning when the focus was concentrating on getting the GUI and the gizmo and ball icons to look aesthetically pleasing, the simple testing strategy involved a continual cycle of editing the code and then running the visual elements every time a change was made to check that they appeared as desired. After the backend was up and running, the functionality of the event handlers was also tested by checking visually that every action has its intended effect.

At this point the TriggerSystem was tested to make sure that absorbers and flippers could be triggered by keypresses and other Gizmos. This was tested by running the sample formatted file provided in the project statement and checking that indeed the trigger events occurred. More rigorous testing of the TriggerSystem, such as its capacity in terms of triggering events, will occur before the final release.

Finally, the last step of integration testing performed thus far involved playing around with the GizmoBall editor, and checking for errors, program crashes, performance issues, etc.

Testing Results:

A substantial amount of testing has already been accomplished, as stated in the previous section. The Gizmo, Ball, Board, and GizmoParser modules have undergone an extremely thorough set of automated tests; and the PhysicsLoop and the frontend GUI and display modules have undergone a relatively thorough set of tests; the TriggerSystem has been tested for basic functionality.

However, several levels of testing still remain between now and the final deadline. In particular, representation invariants and run-time assertions must be added to all backend modules to ensure that they function as they should, especially the Gizmos, the Balls, and the Board. The TriggerSystem must be tested for overloading. The PhysicsLoop must be tested to ensure that all collisions are handled correctly and efficiently, especially when the board is overloaded with objects. The Display must be tested to see if its frame rate can match up to the desired rate of 20 frames per second when the board is filled with gizmos and balls. The GUI (excluding the Display) must be tested to see that it is extremely user friendly and displays relevant help messages; some degree of user testing should also take place to ensure that the operations are simple and intuitive. Nonetheless, the degree of confidence in the code is currently still quite high. Most of the major bugs appear to have been discovered, and gameplay is relatively smooth.

Once again, the remaining bugs and necessary optimizations will be discovered when the board is overloaded with gizmos and balls.

Reflection:

In light of our plan and the requirements we need to meet, we have made considerable progress, and we believe we are in good shape to meet the final deadline. As of the writing of this document, we have met most of our internal deadlines and we have achieved the majority of our goals in our preliminary plan. We have implemented all of the individuals modules and they have followed our planned MDD and OM fairly closely with a few changes. We have pretty exhaustively tested our basic objects such as Gizmos and Balls as well as done a wide array of integration testing, fixing the vast majority of bugs.

However, our software does have a few issues that are in the process of being dealt with. One such is that in rare situations the physics routine will enter an infinite loop and crash the program. Our plan is to pinpoint the location of the infinite loop, discover the cause of entry into the loop, and resolve this issue in one fell swoop of sweet sweet 170 coding. Another issue is the motion of the flippers. MagicKeyListener seems to behave in strange ways under certain situations so that if the trigger key is pressed in quick succession at exactly the right time interval, the flipper will flip to and stay in it's flipped position. The flipper will then flip to the original position when the trigger key is pressed after that but will still return to the flipped position unless the trigger key is pressed in quick succession at exactly the right time interval once more.

For the next few weeks we have three overall goals. First and foremost is to finish implementing all required functionality. Second is to establish stronger testing. And third is to refine our existing code. Still not implemented are the remaining GUI options such as connecting gizmos, setting ball size, setting the gravity, and also setting the friction. And as we finish those up, we need to be looking to implement more rigorous tests of our code. For that we need to establish representation invariants and abstraction functions for our ADT's. And of course there can never be enough fine tuning of existing code. Most important is the need to fine tune flipper movements and physics loop algorithms to try to speed things up. Also we will implement a queue for the trigger system to make it more efficient. And last but still very important is the need to find more ways to decouple individual modules.