Glen Chandler

Java Software Development

Final Project Report

To begin, this project is a simple arcade like game where a player shoots a ball, and another player attempts to block the ball by jumping. This game is two dimensional and implements a simple physics engine that allow objects to move and interact in a way that simulates reality in a very simple way. The game starts with a menu that allows you to view highscores, set highscores, and a help button that allows opens a windows with general instructions. Once the play button is pressed, the window with the game opens, and the menu window closes. A countdown starts while the player shoots balls using the mouse, and once the timer runs out, the game window closes and the menu window opens, displaying the score just received. This is the main idea of the game, allowing for players to challenge each other's highscores, while being actively involved in the game themselves.

The program uses java libraries alone, and the graphics user interface is based solely on repaint methods, buttons, panels and other standard java gui options. My approach to the project was based heavily around object oriented programming. I use polymorphism with a super class "ShapeObj", and an array of ShapeObj including all of its child classes. This array is passed all around the code so that methods can manipulate the objects values, and also call their methods such as get methods, and each ShapeObj's own repaint method.

First of all, ShapeObj is an abstract class with private members of x and y, referring to the x and y position on the frame, a private "weight" variable, which is a multiplier to gravity in the PEngine class, magX and magY variables, which are references to the objects velocity in the x and y direction (the rate of change of x and y), rotation and rotationMag, which refer to the current degree angle of an object,

and the degree of which the current angle is changing. ShapeObj has abstract setter and getters for each of these private variables, and one ‘paintObj’ method which will allow for custom paint to be implemented for each class that inherits from ShapeObj. The paintObj method is not abstract and includes a default of draw oval at x and y position, so that shapes will default to a circle, if it does not have a paintObj method. Lastly, each Obj has a hitbox, which is used within the class for repaint.

The Ball class inherits from ShapeObj, and implements all the abstract methods. It has its own radius variable which it uses for the size of drawing in the paintObj method. It also has ‘hitbox’ variable of type Rectangle(java.awt). The hitbox is used in the paintObj() method and is used for rotation by the AffineTransform class, which allows for object rotation on a frame. However, since ball object is a perfect circle, it is difficult to see it rotate.

SHRectangle class also inherits form ShapeObj, and includes a hitbox similar to ball’s, and instead of a radius, includes an xSize and ySize. Here the hitbox is made using xSize, ySize, and x and y positions just as it is in ball. Class ‘Arms’ also inherits from ShapeObj, but includes a unique member of type ShapeObj called ‘bindObj’. In the constructor, SHRectangle is given a reference of an object in which it can use. This allows for the Arms class to set its own x and y in reference to another objects x and y. This allows for an object to be perpetually attached to another object.

Next is the GameRun class, which is the masterclass for the actual basketball game to run (seperate from menu). To begin, the array shapes is created as an array of type shapeObj[]. This object is given new elements of type shapeObj[], like ball, rectangle and arm. All the constructors are called for specific variables, and they are all set to positions on the screen. Classes ButHandler, PEngine, and PaintingClass are all initialized to objects (will be explained later). The frame is created in which the game will be ran, and things will be painted on, and the infinite loop is called where the PEngine method

is called to process the objects, and the PaintingClass method is called to paint things to the frame. A counter is made and incremented for each loop of the infinite loop, and if it reaches 6000 (roughly 60 seconds), infinite loop breaks. After this, the game frame is set to invisible, and the menu frame is removed, where the menu constructor is called to create a new main menu.

The PaintingClass object is important in that it does all the painting for the main frame within one JFrame. First, shapes, (the ShapeObj) array is passed so it has access to the objects variables, and more importantly methods. A few String members are initialized to be used for the background painting and the countdown timer, like font and other things. Most of these are handled in the constructor of PaintingClass. However, once repaint is called from the GameRun class, PaintingClass calls the super cnstructor, (which clears the screen), and then calls auxillaryPainting, an in class method that uses Graphics to draw the basketball hoop, score board, and others. Then, a for loop runs that loops through every object of the ShapeObj[] array (implementing polymorphism). For each object, repaint is called, which calls each objects specific repaint, allowing for multiple separate drawings and paint methods on the screen per frame.

Each class's repaint method is different in that they both draw a different shape, using different members from that class. FillRectangle or fillOval is called depending on the class, and then transform (object of type AffineTransform) is called with transform.rotate(). The parameters for this are the degree of rotation, provided by ShapeObj object, and x and y coordinate where the rotation occurs. In these are also supplied by class member variables, but in the case of 'Arms' class, it is provided by another object, allowing it to bind to another object's x and y coordinate. After this an object of type Shape(java.awt) is assigned to transform.createTransformedShape(hitBox), which is the conceptual size of the each object. This allows for a rotation to occur just for the object. Then, g2d.fill(transformed) is

called to fill this hitbox, which in the case of SHRectangle, is already a rectangle. However, in the case of ball, must be called using fillOval with parameters of member variables, rather than the hitbox itself. This is mainly how the painting to the frame works, and how the frame is able to draw multiple objects in each frame, and clear it each frame and draw all the objects in a different way.

Next, is the class PEngine, which handles the processing of how the objects move about the frame. This class has a constructor, which like most other constructors sets a private array of ShapeObj[] to the one initialized in the GameRun class. In the case of this, frame from GameRun is also passed so that if statements can also be used to detect properties of that frame. There are two major portions to PEngine. First there is the mouse click listeners, which are unimplemented except for mouse pressed and mouse released. These events are able to also call getX() and getY() in which the the initial click coordinates are set to two class member variables. Then when mouseReleased is called, getX() and getY() are compared to the saved coordinates of mousePressed(). Comparing these coordinates allows for a vector to be created where magX and magY of each object is set to the distance of x and y for mouse pressed and released. Also, the x and y coordinate of the currently selected object is set to the shooter position, which is drawn in auxillaryDrawing() in class DrawingClass, to simulate shooting the ball. After these calculations, the selected object, which is selected by a counter, will follow this trajectory, allowing to choose which way an object moves. After this, the objects movement is up to the PEngine functions which change the x and y coordinates, and the magnitudes when different checks occur. These checks are called from runEnvironment() of class PEngine. Additonally PEngine also contains pointsNum variable and a getter method, which is used by DrawingClass to draw to the screen the number of baskets made.

Next is the physics engine detections. The PEngine class already has a copy of the reference variable for the object array, so it is free to use and modify the various objects that will move around the screen. First is the barrier collide check. This method uses two if statements to check first if the objects are at or outside the frame barriers (using frame.getContentPane().getHeight()/ width) and then taking action. If the two left and right walls are hit, the x velocity (magX) is simply inverted, so that the object bounces off the wall and does not lose velocity from the bounce itself. If it hits the floor however, velocity is lost due to the implied impact that dampens another upward bounce (this is an example of the unrealistic aspect of the engine). This, however, stops the object from bouncing forever, and simulates gravity bringing objects to the ground. There is no ceiling check. However, since gravity is always increasing (adding 1) to an objects y velocity, what happens when the object is on the ground with a velocity of 0? If there is no check, gravity will constantly push the object through the ground, so there must be a check to assure that objects don't just sink through the ground but also don't get trapped on the ground after one bounce. So I added a simple if statement where if the velocity is negative, and the object is below the ground (even by one pixel), the floor resting state is set where rotation becomes 0, and the object is set to the y coordinate of the floor. This way, this stationary position on the ground is only initialized if the object begins to sink under the ground (if its velocity is negative and the object is out of bounds). This would only happen if there is no more upwards velocity to stop gravity from pulling the object to the ground, thus simulating an object lying stationary on the ground.

Next, is the moveObj() method. This takes the parameter time, and like all the other methods 'i', the index of which object to effect, thus avoiding many for loops in each function, and just one for loop in the controlling function of PEngine class. This method implements gravity, and wind resitance/ friction (the same effect in this case, just hinderance of moving in the x direction to prevent perpetual

movement in the x direction). To begin, there is gravity. I included an if statement where if magY is less than 15 and time % 2 == 0 (every two frames), then increase magY by 1 * the object's weight. This effectively simulates terminal velocity, and also stops an object from perpetually bouncing. There is also a check to see if the object's rotation is less than 90 to increase and if it's greater than 90 to decrease, making a simple stabilization for when an object flies through the air. Additionally, there is friction, which only activates every 40 frames, to accurately simulate a rolling motion, where rolling objects would not be stopped in 3 seconds. The friction set is a simple algorithm that sets the magnitude x to the original magnitudeX - the magnitudeX / the absolute value of the magnitudeX. (magX / Math.abs(magX)) will give a number that is either 1 or -1, since one will be positive and one will be negative, or both will be positive, and since they are the same number, this will result in a 1 or a -1. This is then subtracted from the original magnitude and set to the new one. This way it is simple to slow down an object regardless of the direction it is going without using an if statement. Finally, this function also calls for each object to move its x, y, and rotation by the appropriate mag of each. Simple movement of each object.

Finally there is collision detection. Collision testing runs simply a nested for loop, and one if statement. The outer loop goes through every object of the object array, and then the inner loop checks with every other object of the row, thus being able to make it so every object checks itself with every other object. The if statement simply checks to see if the x coordinate of one object is within the minimum / maximum range of the x coordinates of another object, while doing the same for y coordinates, it calls a collision event method while passing the indexes of the two interacting objects. The collision event then sets the affected object's rotation to the other objects's magnitude, then switches the velocity of the two objects, simulating them almost bouncing off of each other. There is an additional if statement that also checks to make sure that the objects colliding aren't of index 5 and 6

(arm and second player), but other than that, everything goes. The collision event also only occurs every other frame, to stop from objects getting stuck inside each other because they are being detected as colliding faster than they can move apart. Last but not least is the pointCheck method that simply checks to see if any objects are inside of the hoop using a range coordinate if check for where the basket would be. If the ball is in the basket coordinate at any frame, point number is incremented. There is also a static get method for pointNumber so that other classes can see and use the pointNumber that is changed only inside PEngine. There is a master method in the PEngine class that loops through every object and does all the checking, so to avoid events happening out of order so that every object can have all of its check first, and then move onto the next object.

Next is part of the player interface of the game itself. The KeyPressHandler class is a class that implements KeyListener so that it can effectively detect and do appropriate action whenever a key is pressed during the game. This implements four simple functions. First is the main action listener, in this case keyTyped, which is added to the frame in GameRun(), and is what also checks what key was pressed by using if statements, and e.getKeyChar() , ‘e’ being the event. If the char of the key pressed is ‘w’, then moveJump() is called. Which first, sets the player position to outside of the sitting on floor / sink through the ground area, to a position in space that it can move freely. After this, the magY of the object is set to -30, so that it will now move 30 pixels in the upward position against gravity. Rotation and rotation magnitude are set for visual effect, but the main effect is the shapes[5] (defender player SHRectangle), will now move upward with a velocity of negative 30. If the key ‘a’ is called, then magnitude is set just as jump is, but the x coordinate does not need to be changed at all. Two is simply just subtracted from the original magnitude x. Vice versa also occurs when pressing the key ‘d’, which will cause the object to slide right. Also it is important that these methods do not have for loops that

loop through all objects like most other methods in the program does. This just manipulate object shapes[5], the defender, so that on key presses, he is the only one affected.

Lastly, is the menu. None of this is started until the constructor of GameRun is called, which only happens once a button is pressed from the menu. The menu shows a label with the game title, a picture, a button, a label with previous score (obtained from PEngine's getPointNum() method), and a menuBar. First of all the menu bar consists of two drop down selections. First there is help, with has an option to see the help, which shows a brief description of the game, and a brief explanation of how to play the game. A text area is opened where a file with a help message is parsed to the text area. Under the HighScores drop down, there is see high scores, and save current score. First, see high score does the same thing that help does, and parses a file to a text area to read, but with no output file. However, save score opens a frame with a label "Name: " a text area, and a button. An in-method class with action listener is added to the button, and when the button is pressed, the textarea string is taken (player name), the time is taken from class Date, and the scores is taken from PEnigne.getPointNum(). This is all effectively parsed by using a printWriter set to a bufferedwriter, which is set to a fileWriter with accepts the filename as input. This effectively creates a file writer that can parse strings to a file, rather than overriding the file each time the method is called. After the information is written into highScores.txt, the name enter frame is closed, and the menu continues running as usual if you wish to do any more commands, or start the game by pressing the beginGame button, which will call the GameRun() constructor, officially starting the basketball game and executing everything else explained above. Additionally, this menu method is also called from a class MenuMain(), which just holds the constructor to menu and the method main().

In conclusion, I learned many things from this project. I learned how to implement frames easier and more seamlessly, how make frame by frame animations, how to use polymorphism efficiently, and how to use data across many different classes. One of the most important things I learned from this project was the importance of refined and precise algorithms. I came into the project knowing that most of the things I was doing probably had a very simple way of implementing it, things like collision detecting, gravity, rotation, and they all did. I kept a notebook by me at all times so that if I realized how to implement something I was stuck on, or refine something that wasn't working as well as I wanted it to, I could write it down and try it later. Especially in something like a physics engine, there are often very, very simple lines of code that using a small of amount of math magic can do something amazing. It took me a lot of time and though to figure as many of these out as possible and implement it into my code. Other than precise algorithms, I also learned how to use the internet very sparingly when looking for solutions to my problems. I found that anytime I did find a possible solution to something, I ended up not really understanding the implementation or the library used, and this would lead to not being able to understand my own code in a way that backed myself into a wall. That being said, I also found out the importance of knowing my own code front and back, so that I could look at a method and know exactly how it is used, and more importantly, where it was used. I took the architecture of my classes as seriously as I could, because I knew that if I ever lost sight of what I was doing, I would surely scrap the whole project and try to start again. This is why I tried to stick to concepts that, even if I didn't understand completely, I grasped the basic idea to. And this worked out very well for me, as the only real unknown libraries I used were AffineTransform, and Date, which I was not able to completely wrap my mind around them, but nonetheless found a working use for them. I found the importance of neat code, and I always tried to shave off unused code and make my code as short and precise as possible

without risking the structure of the program itself. While I learned many useful libraries and certain programming tricks that I used in my project, the most valuable lessons were the ones that I discussed above. I learned good programming techniques and strategies, and found out how to work through a large scale project in a satisfying, clear, concise, and simple way. I gave myself goals, met them, and kept moving forward. These were the most important lessons to me, where I did not just learn to write a relatively over simplified physics engine, but I learned to be a better programmer, and to make sure my code didn't just compile and run, but that it made me happy and satisfied. The abilities I gained from achieving these goals are surely lessons that I will carry with me for the rest of my life, inside of programming, and outside.