Katherine Ward

Prof. Anthony Ellertson

GIMM 110 Interactive Programming

7 November 2019

In “Doki Doki Animu,” from the stub game “Charge,” the changes start with the menu. My first chronological changes were to the story, but interacting with the game, the menu is first. In the ChargeDoc class, I begin by defining variables to use later. In the main constructor function, I call “init" (60). “Init" is where, line by line, I create the start menu’s variable, set its x and y coordinates to zero, and add it to the stage. I then call “mouseFollower” (114) and add the event listeners to the buttons. The start menu itself is a movieclip with the buttons: “story,” “start,” and “how to play.”

Before delving into the different start screens, “mouseFollower” (114) needs to be explained as it is called in almost every function that alters the screen. “MouseFollower” is my first visible physics change. It is a very simple zeno friction function that interacts with a single movieclip of a heart. In the class function, I defined “heart1,” “heart2,” and “heart3”— all of them of “Heart.” “MouseFollower" adds each to the stage as children at the mouse’s x and y. I make sure they cannot block the mouse from clicking by setting mouse.enabled equal to false on each of the children, then add an enter\_frame event listener. “MouseFollowerZ" (129) is where the physics begin to move. From here, I used the zeno friction math, offsetting the first heart from the mouse, the second from the first, and the third from the second. In the offset, I found having “heart1’s” x and y plus 20 each, “heart2’s” plus 30, and “heart3’s” plus 25 looked the best. After setting the offset to where the hearts would move, I subtracted the current x and y coordinate respectively and divided the result by five, completing the function. This fix does not interact with the game in any way, but I found the cosmetic appeal very charming and additive to the game’s experience.

Back to the start menu, the player has three options to click. Clicking “play” calls “startGameHandler” (98) which removes the button’s parent, startMenu, and event listener then calls “createGame" (106). “CreateGame" sets the chargeGame class to the variable “game” and adds it to the stage. The only change I made here was specifying the child be added to the stage and adding “mouseFollower" (114) after the game was instantiated to ensure the hearts would follow the mouse in the game proper. Clicking “story” calls “createIntroClip" (23). Lines 24-26 do the same thing as “startMenuHandler” with the exception of defining the intro movieclip’s variable in the class instead of in the function and adding the intro, not the game. The next line, 27, ensures the movieclip plays until the first ‘stop()’ in the clip’s actions. In the intro, clicking the built in “menu” button calls “createStartMenu” (73). This function is the same as “init" (60) except for removing the tutorial in its first line. I set it up this way to reduce the amount of children on the stage while avoiding null object errors. The “next” button in the intro plays the clip’s next frame by calling “nextFrm” (52) which specifies the intro to play. A peer mentor helped me figure this one out. “NextFrm” is called until, in the clip’s layers, the next button disappears revealing the “play” button. The play button is under the next button the whole time in order to avoid another null object error. Animate does not like interacting with objects that do not exist when it thinks they should. This play button calls “startGameHandler" (98) too. Lastly, clicking “how to play” in the start menu calls “createTutorial” (87). The tutorial variable is assigned in the class, so this function adds it to the stage, adds the mouse follower, sets the movieclip to frame one, then adds the event listeners to the “menu” and “next” buttons. “Next” calls “playNext” (56). “PlayNext” is the same as “nxtFrm” (52) but instead specifying the tutorial to play. Since creating the tutorial does not delete the menu under it, the menu button only needs to call “deleteTutorial” (94) which simply removes the tutorial child from the stage. This way reduces the stage’s children and should make the game run faster.

Before getting into the actual game, a quick fix I did was in the Parallax class. All I did was comment out lines 54-72, effectively removing the parallax on the y-axis. I had to rearrange the stages in Stage1 and alter the units’ spawn point (ChargeGame lines 186 and 208) to accommodate this change, but it was still simple.

Now, we excavate ChargeGame. To start generally, wherever there is anything to do with a guy created, I multiplied it thrice. If the name had guy, it was changed to reflect SII, PBB, and CDD— the class names of my main units. This starts in line 23. At line 51, I created all the same variables from ChargeDoc since I restart the game by basically copying ChargeDoc into ChargeGame at line 403. “Init” was changed to “startChargeDoc” (403), but aside from that, everything else is the same. The restart conditions are called in “endGame” (378). Simply, if your or the enemy’s health reaches zero, checked in lines 224-229, “endGame” removes the “update” enter\_frame event and the added\_to\_stage event listeners and sets the win and lose screen movieclips to variables. When one health or the other reaches zero, the update also puts through a condition string variable accordingly. An if/elseif statement sorts them. A “win” will create the “winScreen” (387), and a “lose” will create “loseScreen” (393). Each sets the coordinates and adds the appropriate screen to the stage and places a mouse event listener on the nested buttons. The buttons have different names depending on the screen, but they both call “startChargeDoc” (403). After the if/elseif statements, I add “mouseFollower” (481), so it will be visible over either screen.

Another simple fix I made was changing the text format of the scores on lines 101-115. This took longer than I thought it would, but I found the format and the correct way to write the code through the official documentation and a Republic of Code tutorial. Continuing with more isolated fixes, each of the unit buttons along the top of the screen has a separate cooldown for when they can be summoned again. The initial counter values are set in lines 71-78 and utilized in the private function “counter” (293). It is either called from line 194 in “createGuy,” or line 222 in “update.” Each of the unit buttons, created and added 84-96, has an event listener, added 122-124, that calls their respective “create[unit]Handler” (161-174). The handlers call “createGuy” (176) and set the parameters for the unit side, name, class, and combat style. If the unit is a good guy, they will be created on the left side and “counter” will be called with the string “clicked” and the unit name string. With the “clicked” parameter, the unit’s counter will set to zero and the button will disable and lower its alpha value. If the counter is called from “update,” then the string parameters will be “unclicked” and “N/A.” These parameters will simply add to each counter until it reaches its total and reactivates the buttons. In the last “else” of these parameters (329), when the enemy counter reaches its maximum, it resets the counter and calls “createGuy” with the enemy’s parameters. In creating separate cooldowns for each unit, the player will have to pay closer attention to when each unit is available, adding a layer of difficulty to the game.

The last of my changes to ChargeGame are the “pause” and “unpause” functions (361, 370), and the “start” function (144). The “start” function was created with the help of a peer mentor when my stage kept returning as null. We added an added\_to\_stage event listener which called “start.” All this function does is call add “update” as an enter\_frame event listener and adds the pause button, the unit buttons, and the text fields to the stage. The important part is that “start” creates the stage that everything else interacts with whenever I created children. When the pause button is clicked, the “pause” function removes the enter\_frame and added\_to\_stage event listeners, adds the pause screen to the stage, and adds mouse event listeners to the nested buttons. The menu button calls “startChargeDoc” again, and the unpause button calls “unpause.” “Unpause” removes its event listener, removes the pauseScreen child, then adds the enter\_frame and added\_to\_stage event listeners back to the stage. In doing it this way, the units’ movieclips still loop, but they do not move forward. The SII music notes do not stop, though, since they are controlled by an enter\_frame in a different class.

Moving to the Guy class, each unit has a unique speed, health, and damage. I set these variables in lines 15-26 and set them in 63-80. By defining them in the class, they are easier to change and really helped when I was debugging the fighting since I could just change one value. When the guy is created, line 52 sets a variable for the called unit’s class name. In 63-80, I use if/else statements to check which unit is being called, then assign the appropriate values to speed, health, and damage. In line 78, instead of setting the enemy’s speed to one value, I decided to randomize their speed. This makes the enemies more interesting than all of them being the same. The next big change is my ranged unit. Starting at line 136 in the ChargeGame class, when the SII unit is created, it sets the parameter of “ranged.” In Guy, line 187, this is used. When the combat parameter is “ranged,” the unit will start attacking 200 pixels away instead of when they touch the enemy. This brings us back to line 124 in the “attack” function. I create a new className variable and set my physics class, “DokiPhysics,” to an appropriate variable. In line 127, if the attacking unit is not named “SII,” damage will be dealt as it was in the stub game. Otherwise, if the class name is “SII,” line 133 adds the physics class’ function “singSong” itself as a child to the playing field with the parameters of the target, the unit’s damage value, the chargeGame variable, and the unit’s x and y coordinates. In DokiPhysics, I initially defined the song attack in line 15 of the class, so I could set it immediately after assigning the appropriate parameters to variables within the function. After setting the projectile attack’s x and y, I add it as a child of the DokiPhysics class object. I then make sure the attack movieclip plays from frame one and name it “song” (94, 96). In line 98, I add an enter\_frame listener to the object and return the function in line 99 to ensure Guy receives the outcome. The event listener calls “songSung” which moves the object forward (103-104). It checks every frame whether or not the object has reached the target, and when the x value of the song is greater than or equal to the target’s, it calls “songDamage” in line 107. Starting in line 111, “songDamage” calls “takeDamage” from the Guy class, removes the “songSung” event listener, then removes the song object. A problem I encountered with this is that the song will only deal damage to the initially assigned target, even if the target has died. The song will keep moving across the screen, unable to damage the other enemies. I remedy this in line 116 by using the “getStatus” function in Guy. If the target’s status is “die,” the song will be removed through its parent.

Returning to Guy, starting line 158 in the “die” function, I set my physics class t a variable and use getQualifiedClassName again to set the dying unit’s class name to a variable. With this, I use an if statement to ask if the unit’s class name is “CDD,” my defense unit (165). If it is, I set the “cddX” and “cddY” variables, defined in lines 42 and 43, to the unit’s x and y and send them through to the handler function, “rockDeath,” in DokiPhysics (173). I add the class’ function to the stage in the same way I added the song projectile in line 133. Going into the DokiPhysics class again, “rockDeath’s” physics math was taken largely from the shoot game. After initializing the rock object, adding it to the stage, randomizing its appearance by choosing a random unique frame, and setting the object at the correct x and y to look normal, the math starts (36-46, 49). Lines 49 and 50 calculate the x and y differences between the unit and enemy base’s x and y. Line 52 uses the Pythagorean theorem to find the hypotenuse between the two targets. This is divided by 15 in line 55 to create the power of the x velocity, then multiplied by the cosine of the rock’s rotation divided by 180 times PI. This is supposed to make the rock appear to arc, but it returns a static number. The y velocity is created by dividing the two target’s y coordinates by the difference between them to create a constant motion to the enemy base no matter where the unit is destroyed. After all this, an enter\_frame is added to the rock that calls “rockThrown” and I make sure the function is returned to Guy (61, 63). In “rockThrown,” in lines 68-69, I add the velocities to the rock’s x and y values. I subtracted ten from the x velocity in order to make it slow down and be more visible to the player. The if statement on line 75 checks to see when the rock has hit the enemy base, and when it does, it subtracts from the enemy score, updates the enemy health text field, removes the “rockThrown” event listener, then sets the rock’s y value to 1000. Setting the y value instead of destroying the child is a bit messier, but I could not access the rock’s parent to remove it, and this is just as effective. The entire rock handler did not work for a very long time, but a peer mentor eventually figured out how to fix it. By making ChargeGame a public variable and putting it into the physics functions as a parameter, it is not instantiated as a new object as much and does not overlay the projectiles. This is also why the physics classes are added to the stage as children instead of created on the stage; it lessens game layers that can get in the way. I understand much better why this is necessary to the visibility of the projectiles, and was able to use this knowledge in the “singSong” function and in helping classmates.

My last fix to the CDD unit is the way they move. Instead of progressing at a steady pace like the other units, I used a zeno friction function to create an uneven movement to simulate a heavy object being pushed— like delinquents shoving a boulder. Using the stub game’s movement code in Guy’s “update”, checking if the status is equal to “walk” then adding the speed to the unit’s x, I added a check to make sure the unit is a good guy and not named “CDD” (250). If it is not, them movement continues as normal. As an aside, if the unit does not have the “good” parameter, the seep is negated in order that the enemies move left. This is how I created unique enemy units. Back to CDD, if the unit’s name is “CDD,” “moveCounter” is called (253-254). In the class function, I defined a move counter that works exactly like the button counters (44). If the counter is zero, the function calls “moveCDD” (288) then adds to the counter. If the counter is between zero and 50, it just adds to the counter— equal to or above 50 and the counter resets to zero. “MoveCDD” sets the variable “cddTarget,” defined line 45, and adds an enter\_frame listener to the unit that calls “updateCDD” (295). This last function adds the zeno friction math. Subtracting the unit x value from the target x value, dividing it all by 8, then adding the total to the unit’s x value every frame creates a smooth glide into place. The counter makes sure the unit moves in intervals, instead of adding the movement factor every frame. Before using the external counter, the zeno friction just added itself every frame and created a slower unit. When the unit moves as it does in intervals, it adds variety to the main units and creates a more fun experience.

The last change I made, and the easiest, was the Greensock animation. The animation itself shifts the unit backwards, shifting the alpha to zero. I added this in addition to the death animation. As the unit dies, they fade out. I simply had to put the generated Greensock code in the “update’s” “if(status == “Die”)” function (220). Once the health hits -13, the animation starts, and the unit is purged when the health equals -15. I had to play around with these values until it looked right. Before I did changed the purge health value, the Greensock moved the units too far off the screen, lengthening the playing field, and interfering with the parallax. It took a lng time to realize the issue, but it was a simple fix once I had. Next, I use getQualifiedClassName once again, in line 226, in order to make the enemy shift to the right as they die. Before I had, not only did it look odd, but the Greensck flung them to the left side of the playing field, hitting the player’s base, and messing up the score. This was another problem that was a very simple fix but took a long time to figure out.

All in all, “Doki Doki Animu” took a lot of work and learning to finish. I am very grateful to the peer mentors and my classmates for helping me figure out my issues, and I will be able to use everything I have learned from this in our next projects.