Bust a Move: Project Summary and Design

For my final project, I created an arcade-style game based on the game “Bust a Move.” The game works by firing objects, in this case bubbles, of several different colors from the bottom of the screen to the roof of the screen where other bubbles are clinging to the roof in staggered rows, much like a brick-wall pattern. The goal of the game is to fire your bubbles at bubbles on the roof of the same color in order to pop them, and possibly leave other bubbles floating and therefore pop them as well. Once you have popped all of the bubbles on the roof, you win. However, every few shots that you do not cause a pop sequence that leaves another bubble floating, the roof is lowered towards the bottom of the screen, pushing all the bubbles down towards your launcher. If any bubbles end up at the level of your launcher, you lose the game.

Now, if your fired bubble collides with either the roof or another bubble already clinging to the roof, then the fired bubble will stop moving and become one of the bubbles clinging to the roof, either directly or through some other bubbles that are clinging to a bubble that is clinging directly to the roof itself. If this fired bubble is touching at least two other bubbles of the same color after it lands, either directly or through some other bubbles of the same color, then all of these matching-color bubbles will be popped. After these matching-color bubbles are popped, if any bubbles are left floating, that is, no longer touching the roof either directly or through some other bubbles of any color that are touching another bubble that is directly touching the roof itself, then those bubbles need to be dropped and subsequently popped as well.

For this color-matching pop test and the subsequent floating drop test, I used three main functions: ChainPop, PopMatching, and then FallCheck.

The function ChainPop takes in your fired bubble after it lands, does a breadth-first search to check for matching-color, touching bubbles and establishes a list of them. It accomplishes this using two lists in order to avoid backtracking while checking bubbles adjacent to the current bubble being checked. Then, decides if this group of bubbles needs to be popped based on the length of the list.

Function PopMatching pops every bubble in the list created by ChainPop, and checks every adjacent to each popped bubble, putting it into a new list to be checked for floating bubbles, making sure not to add the same bubble twice as one bubble may be adjacent to two popped bubbles at the same time.

Last in this sequence, function FallCheck does a breadth-first search for all bubbles touching each bubble in the list provided by PopMatching, decides whether or not this group of touching bubbles is touching the roof as a whole, and, based on this decision, either drops the entire group or marks each bubble in this group as “safe,” so that they will not be checked a second time during the same sequence because a bubble from the original list may have already been check because it was touching a another bubble earlier in the list that already had all bubbles it was touching check. After it’s done dropping all floating bubbles, it flushes the safe list so that the next fired bubble can use it.

After this sequence of functions has finished, all matching-color bubbles that were touching the fired bubble have been popped, and all bubbles that were left floating afterwards have been dropped.

Another non-trivial function from my game was the AtLanding function. This function is responsible for checking if your fired bubble is at a proper landing position as it travels across the screen. This function relies heavily on the adjacents list that every bubble on the game field has. As the field is initialized, each bubble pushes every bubble that is adjacent to it into its own adjacents list. So, as the fired bubble travels across the screen in the angle of the aimer, the game converts the screen coordinates into field coordinates, rounds the result to the nearest bubble on the field’s grid, and then checks that closest bubble’s adjacents list for any filled bubbles, that is, any bubbles that have a color. An empty bubble in the field grid still exists; however, it is simply not drawn if it does not have a color. If a filled adjacent bubble is found, a collision test is performed. My collision test works by performing a line-circle intersection test. The line is created with one endpoint at the fired bubble’s actual position and the other endpoint two units away in the fired bubble’s traveling direction. The circle is centered at the adjacent bubble’s center with a radius equal to the adjacent bubble’s radius plus the fired bubbles radius (half plus one half for one unit). By plugging the equation of the line into the equation of the circle and algebraically converting the equation into the quadratic equation, I can test if the line intersects the circle. If there are no solutions for the resulting quadratic equation, then there is no collision; if there is one solution, then either the line is tangent to the circle, indicating a collision, or one endpoint of the line is inside the circle and the other is outside, indicating a collision as well; if there are two solutions, then both of the endpoint of the line are outside the circle and the line intersects the circle, indicating a collision. Using this function, the game accurately tests when the fired bubble should land.

The last important function is the Ricochet function. As the fired bubble travels across the screen, if it hits a wall, it needs to bounce off. The first thing the game needs to know for this calculation is the distance from where the fired bubble started moving to the wall. For any trajectory, this can be found by taking the cosine of the angle of fire. This gives us the change in position of the fired bubble in the x-direction. If we record whether the fired bubble is traveling from the aimer, or a previous bounce off the opposite wall, we can assume the size of the gap is either half the width of the field or the same size as the width of the field, respectively. So if this change in x is greater than or equal to our known gap, we know the bubble is hitting a wall and needs to bounce off of it. Once we know the fired bubble needs to bounce, we simply subtract the last angle of fire from 180 to get the new angle of fire.

For the most part, the rest of the project is self-explanatory. I chose to keep the details of the program simplistic to avoid any complications over trivialities and to make sure I had enough time to make sure the mechanics of gameplay functioned the way I wanted them to. In the end, there are still a few bugs during the matching-color pop sequence and subsequent dropping of floating bubbles and perhaps still a bug with the victory and defeat triggers, but I’ve been unable to reproduce them at all after they occur. Overall, the game was a great success. It functions almost exactly as I intended it to and is just as fun to play as I expected it would be.

Bibliography

Wikipedia. Web. <http://Wikipedia.org>.

OpenGL. Khronos Group. Web. <http://opengl.org>.