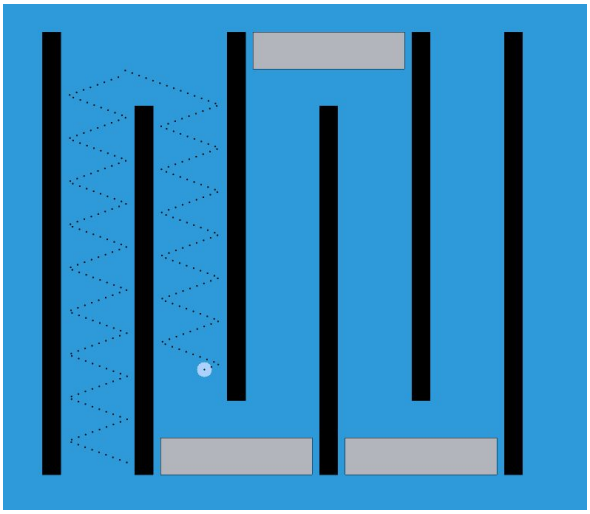
Not Okay: A Collision-Based Game

Brianna Karpowicz & Gabrielle Leon

Intent



The intent of this project is to recreate a version of the iOS application "o k a y?" This application is a game in which users launch a ball at different shapes and utilize positioning of the ball to attempt to make contact with all of the shapes on the screen in one launch. As the ball makes contact with each shape, the shape disappears. The object of each level is to contact, and thus clear, each object on the screen. This project utilizes the physics and mathematics of collisions with straight and angled walls as well as code for user interfaces and animation. While the interface itself is very simple, as we intended to model the game after the minimalistic style of the original app, the gameplay is challenging.

High Level Description

Guide was used to create a menu page (`NotOkayGUI`), which allows the user to select each level, view a help screen, or have a random level of a given difficulty generated for them. The help screen utilizes gifs of the game as well as pushbuttons to demonstrate how to play. The gifs are played using `gifplayer`, a piece of shared code uploaded to the Matlab file exchange by Vihang Patil. Because each level differs so much in number and placement of shapes, types of walls encountered, conditions to make the shapes disappear, and other special features, scripts rather than functions were used to design each level. Each level script plots different shapes that serve as the walls using patch and line. The script also utilizes keyboard commands, where the user can hit spacebar to launch and "r" to clear selections and pick a new initial state.

All of the levels call to `userSelect`, `calculateVelocity`, `initialPlot`, `updatePlot`, and `winnerScreen`. `userSelect` utilizes `ginput` to allow the user to choose a starting location and direction of velocity, but prevents the user from selecting a starting position inside of a shape. It then plots the .png image of a ball at this initial location and plots an arrow indicating the chosen direction of velocity. `calculateVelocity` determines the initial velocity components using the positions selected for the ball and the end of the arrow. `initialPlot` initializes the display of the previous state of the ball to create a trail. `updatePlot` updates the visualization of the ball and its trail as computed by `updateBallState`. `winnerScreen` determines whether or not the level has been won by considering the position of the ball and the presence of the shapes. It brings up a celebratory image if the user beats the level and options to restart the level or go back to the main menu if the user loses. There are also sound effects for both winning and losing a level.

To determine the ball's dynamics, `updateBallState` calls to `findCollision` to determine if a collision occurs and what the state of the ball will be after said collision. For vertical and horizontal walls, `findCollision` uses basic physical equations to determine the new state. For collisions with angled walls, `findCollision` calls to `tiltAxis`, which utilizes a rotation matrix to shift the coordinate system counterclockwise such that the walls are now pseudo-vertical. `findCollision` is then recalled and the calculations are performed as if the wall and ball are in the vertical plane before being fed back into `tiltAxis` where the calculated values are rotated back into the original coordinate system. In levels that contain circles, the level script calls upon `circleWalls`, which creates a 36-sided polygon using a rotation matrix to rotate a tangent line until the circle is completely circumscribed. Each side's endpoints are stored in a matrix which is outputted by the function and added to the walls matrix in the level script. This polygon

represents a rough estimation of all possible tangent lines and allows `findCollision` and `tiltAxis` to continue to be used.

The last five levels contain different aspects that contribute to gameplay. In level 16, there are permanent black walls that do not disappear when a collision occurs. In levels 17 and 18 the white walls need to be "activated" by passing the ball through the wall before a collision can occur with it. Level 20 contains a vortex that causes the ball to disappear if it enters it, and if the ball goes offscreen, it will come back on screen from the other side of the screen at the same angle. These levels all have pop-up windows that describe the new feature before the user can play.

Technical Challenges

The main challenge of this project involved implementing proper dynamics to make the movement of the ball as realistic as possible. Angled walls posed the biggest initial challenge, since it was important that this math was reliable in order to properly be called upon later by the functions for circular walls. Ultimately it was decided that rotating the coordinate axis system was the most robust solution. However, occasionally the wall was not rotated perfectly due to inherent imprecision in the mathematics performed by Matlab, and `findCollision` would fail to recognize it as a completely vertical wall, causing the ball to pass through the shape entirely or begin colliding with the interior of the shape. To fix this issue, rounding to three significant figures proved to be the quickest solution that still allowed the visual accuracy of the ball's movement to persist.

Circular walls also presented a challenge in terms of determining the collision and proper resulting dynamics. To solve this, a 36-sided polygon was created around the circle consisting of a short tangent line every 10 degrees. This gives some error in determining collisions since there is a small area between the circle and the polygon, but because of the size of the circle and the number of sides, the visual error is minimal. Similarly, circles were challenging in that they were difficult to determine boundaries in which to prevent the user from initially placing the ball - an important limitation for proper game play. In order to be consistent with the existing `userSelect` function, a square was plotted around the circle in which the user cannot place the ball initially. This causes there to be a small area outside of the circle in which the ball cannot be plotted, but the levels can still be won with other placement, so consequences are minimal.

Finally, in the last five levels, it is possible that the ball will get stuck bouncing between two permanent walls or returning to the other side of the screen infinitely. In these cases, the user would be stuck without a way to quit the level unless he or she chose to close the GUI completely. Instead, a push button was added that would cause the `winnerScreen` function to be forcibly called.

Omitted Features

The coefficient of restitution was kept at one, therefore creating elastic collisions. In order to ensure that the ball is able to make it entirely through each level without stopping, keeping the magnitude of the velocity constant was the most logical. We also decided not to implement moving walls in the last levels but instead included other novel features. We made this change because moving walls required implementation of parallel function evaluation in order to keep the walls animated and still allow the user interface to be interacted with and the ball to be animated simultaneously. Since our levels were designed in scripts, this proved to be especially complicated and ultimately it was decided that other features would have the same desired effect - to increase the fun and challenge of the game.