**Graphics Game – Abhay Bhat – Maze Game**

Idea – A maze game where the player must navigate from the start to the finish of a maze. Upon completing the maze, the player wins the game.

Gameplay – A player is spawned in a ray-casted 3d environment. The player must navigate the maze until they reach the “Finish” tile, where the game ends as the player has won. Played can use W/S to go forwards/backwards, and A/D will let the player turn left/right respectively. This system of viewing is similar to old, 3d-raycasted games.

Rules – No “rules.” The game will ensure the player stays within the walls.

Objective – Get through the maze.

APIs:

* No external APIs not already provided by the java library
* java.awt libraries such as…
  + Canvas
  + Graphics
  + Color
  + Image
  + KeyListener & KeyEvent
* javax.swing libraries such as…
  + JFrame
  + SwingUtilities
  + ImageIcon
* And some more java STDLIB (standard libraries)
  + TimeUnit
  + Math

Pseudocode:

* Window class
  + Extends JFrame
  + Initializes various JFrame settings
  + Adds the game object and starts the game
* MazeGenerator class
  + Has a static method which returns a map/maze
* MazeGame class
  + Actual main class which has public static void main, which uses SwingUtilities to start “this” game
  + MazeGame implements “Runnable” interface, which means it can be run via a separate Thread, allowing for greater performance as well as well as the ability to properly utilize a game loop
  + Also extends canvas, because “this” game will have visuals drawn to it. We pass “this” to the Window class to add to the frame.
  + We declare various constants for the game and window as necessary
  + Start method creates the Player object and starts the game thread
  + Run() method, which implements method from Runnable, contains a game loop, which updates (“ticks”) and renders (draws) our game
    - While running game
      * Tick()
      * Render()
  + Tick() method updates variables. For now, my idea is that it will call a tick method in the Player
  + Render() method which actually draws graphics to canvas. Draws a background, then draws the 3d view, then draws a minimap, then the FPS. It needs to draw in this order because these are the layers that will appear on the screen. Primarily calls the player render methods
* Player class
  + Also a “Camera” class; this is the location where most of the calculations and 3d drawing will occur
  + Implements KeyListener, which allows it to respond to key events. In the MazeGame class, we attach it to the “Window” instance we make.
  + Contains instance variables to hold x/y pos, fov, and other relevant info for a camera.
  + Contains method implementations for KeyListener, setting the relevant Boolean to true when a key is pressed, such as setting forward to true when the up arrow is pressed
  + Tick() method which updates location based on what movement Booleans are true. If the forward key is pressed, we update x/y vars based on the direction the player is facing. We also move the direction left or right based on whether the corresponding key is pressed. We also have collision detection here: we check if the x/y pos we are going to is a “wall” in the map, and if it is, then we don’t update our x/y position.
  + Render() method where the real 3d drawing happens.
    - Use a “DDA” Algorithm, which finds the distance from a point to the nearest wall for a specific ray direction in an array.
    - Calculate the distance to the nearest wall intersecting the ray horizontally, then calculate the distance for the nearest wall intersecting the ray vertically.
    - This calculation is done by finding the nearest whole number index in a grid using math and trig. We then move in equal sized intervals until we hit a wall. This method is mathematically dependent on direction, which means we will need if statements in order to manipulate our math to be negative/positive based on direction.
    - We will decide on a constant number for how many pixels each vertical rectangle should be, and then we draw rays for each section. So with a screen width of 400 and with each rectangle being 4 pixels in length, we for loop 100 rays covering the entire FOV.
    - We then use the distance of the shorter ray, and then draw it to the screen. Based on the distance, the wall will appear bigger or smaller
    - The color we use for the wall can be dependent on various things, depending on what I deem fit.
      * Can be based on whether we intersect horizontally or vertically,
      * Can be based on a “flashlight” shape, etc
  + We have additional render methods as needed, such as for drawing a crosshair.
  + Additional helper methods for getting colors and doing more math