**Introduction**

The main influence to the design was based on automatic move and reactive concept. As every game needs to be able to do certain basic tasks to create an environment for the user to enjoy, the first step was to discovery a framework to build on top of. The LEDS and the option to interact with the entity or simply sit back and let the AI traverse through the logic creating captivating eye candy brought us to the astro sensor extension. The amount of accuracy that the AI should have is what lead to the algorithms chosen to accomplish the goal that we set out to achieve. The designed implemented for the AI consists of two central parts. The collision methods being one of the central parts create a circumstance for reactive decision to be made. The second key method is the path that is chosen to get to the target which is random. These two methods combined create a reactive AI once set into motion.

**Concept**

The diagram created was meant to depict and provide insight into the inner workings of the reactive AI and the structure that drives it. Reactive AI is one of the most basic AI implementation, it was a perfect choice for an entity that is event driven. The AI is set into motion upon entering a Boolean based while loop. Each sequential pass across the central parts mentioned above, set various entity states into effect. The first method that is encountered is the handle events(HDE) method, this method increments the coordinates array and sets an entity heading. Once a position and direction has been established, these methods call the collision check methods and the process begins.

The first collision method of importance is the target collision(CD); the secondary collision check is for movement(SM). The movement collision check has two main functions, one is to avoid the bounding borders and the other is to avoid colliding with itself.

The bounding boarder’s detection or often referred to as the bounding box test creates rectangular edge limits defined by the left, top, bottom and right grid limits. The self-awareness aspect of the collision detection is pure chance, while random choice of movement is made and screened by safe move(SM) flags are set depending on a hit or miss condition. As the snake gets a clear set of flags and a random path is generated, the entity moves into the game logic(GM) method that increases the entity segments and increments the positioning of the entity to a new position coordinates. The last part of the design is accomplished with rendering(RD) the stored positions and checking the flags for a reset(RE) condition.

**Intention**

We originally wanted to modify a snake game written in c to respond to motion instead of keyboard input but, due to its simplicity, we decided to implement an AI algorithm. For motion detection, we used the RTIMULib user library written in c++ to parse input from the GPIO pins; this greatly simplified the process of coding motion detection. Given how easy this was to implement, my team member and we decided some other functionality would need to be added for this to be substantial enough for a term project. This was when we regrettably decided to try and write artificial intelligence for the snake program in the language it was coded in, c. we tried to do this without modifying any of the functions previously written for the snake game in order to maintain separation of code. The snake AI algorithm was much less straightforward to implement and took many weeks of my time.

**Methodology**

The approach we took for the motion sensing program was simply to replace any functions that handled keyboard polling with my own motion detection algorithm. This algorithm heavily relied on the use of the RTIMULib user library, an open source library for interfacing a 9-dof, 10-dof or 11-dof IMU to embedded Linux systems. All of the setup for using this library was given in an example program which polled the various sensors and displayed rotational data through print functions. This code was placed in the main function to initialize the required objects. The function we wrote to handle motion events takes as a parameter an RTVector3, a data structure containing three floats which represent degrees of rotation in radians. In order to capture the change in rotation, another variable was assigned the value of the last RTVector3 passed to the function. The algorithm then checked whether or not roll was strictly greater than pitch by taking the absolute value of the difference of the current RTVector3 and the last RTVector3. After determining the which difference was greater, the algorithm checked whether or not this difference was strictly greater than a hardcoded minimum. This check allowed the algorithm to ignore any unintentional movement and only process change in rotation above a minimum threshold. After determining which direction the snake was to move, a call to change\_dir() was made passing this direction.

The algorithm for snake AI maintains a path array on the stack in main(), and uses two integer variables for indexing. This array is populated with direction values in the function handle\_events(), where a while loop attempts to create a valid path from the snake head to the apple. In the cases where a direct path to the apple cannot be found, handle\_events() would try to find a safe path to its tail by heading in the complete opposite direction of it. If both paths had collisions on them, the function would then switch on the current heading to find a safe move. This switch statement at the end of handle\_events prevents the snake from running into a wall if the apple is flush with a wall the snake is currently heading towards. In either case, the path is first checked for collisions in safe\_move(); if there are none, it is added to the path array and the index count is incremented and returned. In main, a while loop iterates given the global condition running is true. At each iteration, if path[i] is equal to 0 the path is computed in handle\_events(); if a path already exists, change\_dir() is called passing the current direction referenced by path[i] and then increments the index. The function change\_dir() sets snake.heading to the value of its argument, and then game\_logic() advances the snake in memory according to the new direction. For reasons unknown, we was unable to refactor the code in handle\_events() to separate the different conditions encountered when calculating a safe path. When another function was called to implement this functionality, the snake would be unable to react fast enough to avoid a collision. We both tried implementing an algorithm that would allow the snake to wrap around the edges of the matrix if no direct path to the apple could be found. While this seemed like a good idea at first, the actual implementation of it was not at all trivial. We soon gave up on the idea and instead ended up trying different combinations of the original algorithm that would chase the apple, with a variant where the snake would head in a direction opposite of the tail. This process involved a lot of trial and error with the result being an algorithm that may or may not have functioned better than the naive chase the apple and avoid collisions. The main problem we face with the application now is trying to come up with a way that allows the snake to always have a safe path after eating the apple. As of right now it just mindlessly chases the apple if there is a safe path, paying no attention to any traps it may be falling into. To solve this problem,

**Conclusions**

If we had realized ahead of time the difficulty of implementing an AI algorithm, we probably would have tried to do something else for this project. Nonetheless, we learned a lot about c and the difficulties of implementing AI algorithms in simple games. In hindsight, we probably should have used a graph algorithm that calculated the longest path from the head of the snake to the tail; one that would take shortcuts if a path to the apple presented itself. Although we tried many different approaches to solving the longest path problem, we were unable to get any to work for this application.

**Bibliography**

www.github.com/RPi-Distro/RTIMULib