TRON Preliminary Specification

# Introduction

*Concept overview*

The main goal of our project is to utilize Java to build a TRON simulated game upon the computer, allowing for networking as well, and will be done on a grid-based world. TRON (or light cycles) is a game of pursuit and evasion played on a computer screen. It appeared on the movie TRON in 1982. TRON was chosen as a domain to investigate machine learning because it is a simple one yet presents several features that are known difficulties for machine learning: A dynamic, real-time environment where a perfect strategy is not known. Human and machine intelligence can show their particular weaknesses and strengths chasing each other in this competitive arena. Our team will consist of two software development leads, and one project manager. The complete date of this project is set to be June 2nd, 2014.

*Networking Overview*

Because the implementation of the game is fairly simple, our group decided to add network functionality in order to meet all the requirements of a higher end game. The finished game contains two TRON players on the field, one local and one remotely controlled by an external computer. The networking was mainly structured through the use of IP addresses and ports. This was implemented through the usage of the ServerSocket and Socket classes in the java networking library. All the networking classes are implemented as threads in order to keep up a continuous data transfer required by the routine. The end product is capable of efficiently transferring player location-based data across the machines so that both computers can successfully render their own “remote player”

*AI Overview*

In order to test the graphical and game play aspect of the game, an AI player was implemented. This AI player inherits the TronPlayer class and adds multiple levels of functionality such as choosing when to turn. The AI helps test various game functionalities such as player drawing, winning, losing, scoreboard, background, boundaries, and key input.

# Structural Design

Efficient data storage and transfer are critical to the project because of its high refresh rate and networking implementation.

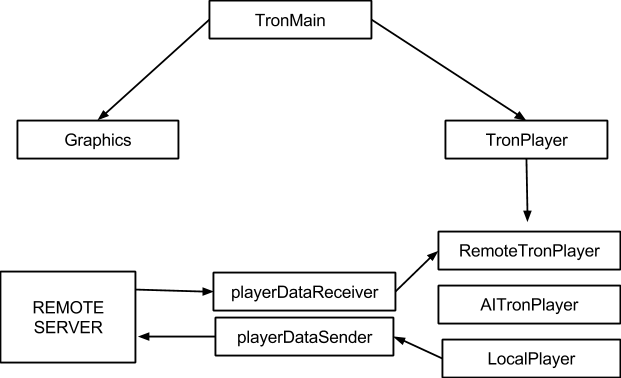
Here are the main storage structures:

*Players*: ArrayList that holds all the players on the fields. Currently the code only supports two players at a time but is kept flexible for above 2 players’ implementation. The main use of the ArrayList structure is to facilitate the process of traversing through all the players for the detection algorithms in the AI player as well as traversing through all the players for the action method to be called upon in the graphics class

*Locations:* Another ArrayList that holds all the locations. In order to prevent increased latency due the continuous restructuring nature of an ArrayList, it will have a predefined length set to be the length of the tail. *Locations* is constantly updated and iterated. It is used in AI detection algorithms, sending and receiving data, rendering player tails, and crash testing methods. Because it is used and updated in a wide list of threads, concurrent modification errors are bound to occur. Because the code repeatedly refreshes itself, a commodification error has little to no effect on game play and is quickly recovered the next cycle around. In order to take advantage of this, the code implementation ignores instances of a commodification error.

*Jframe f:* On a more abstract level a Jframe is used as a container to hold a panel. The panel draws and stores all the graphical dependencies of the code.

# Object Oriented Model



For this portion, we will be mainly using TronPlayer objects in our design. Each TronPlayer will be able to access different methods to turn, as determined by the user. They will also have set colors, either Red or Green. We will also have AITronPlayer objects, which will be controlled by the CPU - this makes decisions based on the User’s specific turns - and will be substituted by the other player once networking is implemented. Inheritance between AITronPlayer and TronPlayer classes also exists.

# Class Descriptions

As seen above the structure is centered around the TronMain class. Much of the intensive graphics are done either in the individual players or the graphics class, both initialized by TronMain. TronMain also initializes the sender and receiver classes in order to communicate with the remote host.

The TRON Payer class is the core of the player computations including location and. Each of the derived methods adds a specific set of functionalities, such as keyboard input in the LocalPlayer, and detection algorithms in the AITronPlayer.

## TronMain

TRON Main runs the game. It holds an array of all the players, establishes connections between local and remote players if needed, creates the JFrame, calls the Display class, and adds a keyListener to change the direction of your player. In TronMain, you can select 3 modes of play. The first is a local player v. computer game. The second will ask for the information needed to establish a connection with another player. The third will run a network game using preset data. TronMain has several methods that serve to initialize various aspects of the game.

Methods:

1. main()
   1. The central aspect of the code. Main initializes the ArrayList of TronPlayers and takes in player input asking for local play(AI), advanced (setup remote metrics), and default (predefined remote metrics)
   2. calls the spawn methods that spawn either 1 or 2 players based on player input
   3. calls the display method which runs the main game loop
2. setDisplay()
   1. Sets up the display as a panel and the frame around it. TronMain itself acts as a frame in which to place the panel.
   2. also sets up a keylistener to the frame to be utilized for the local player
3. printDefaults()
   1. Simple text representation of the defaults
4. killGame()
   1. kills the game
5. spawnTwoPlayers()
   1. spawns a local and remote player, also calls spawnConnection()
6. spawnOnePlayer()
   1. spawns a local and an AI, does not set up a connection
7. spawnConnection()
   1. sets up a sender and receiver from the respective classes

## Display

The Display class draws the players, boundaries, scores, and tails onto the JFrame. It stores an array of players and through each player object, it can ask each player to draw itself, give its score, and update its location and tail array. This class also has a reset method, which resets the players in their original starting positions after one player wins a round. A round ends when one player either crashes into a player’s wall or into the boundaries of the field.

Methods:

1. Display()
   1. constructor sets up some basic metrics
2. paintComponent()
   1. The major drawing method of the game, it recursively calls itself and iteratively calls the updatePlayers() method.
   2. For the PaintComponent, we will be using a black background with Red and Green colored TronPlayers as the visuals. This will be done on an apple, using a panel.
3. updatePlayers()
   1. calls action on all the players iteratively
4. drawScoreBoard()
   1. draws and updates the scoreboard
5. reset()
   1. clears the entire playing field of all the players, has not been implemented yet

#### 

## playerDataSender

The sender class sends step by step location data to the remote server. The main functionality is embedded in the send method which uses a predefined IP and port to forward the location data to the correct remote game. The sender class is implemented as a thread that is designed to close if the socket connection is disrupted.

methods:

1. PlayerDataSender(int remotePort)
   1. constructor that creates a remote socket it is able to send to this section of the thread will not exit until the connection is made
2. send( int[] location)
   1. sends a location which consists of an x and y coordinate. Checks if socket is open before it sends
3. kill()
   1. sets run flag alive to false
4. run()
   1. the thread aspect of the method that runs until alive is detected to be false under which case it closes the connection and stops the thread

## PlayerDataReciever

The receiver class is in charge of receiving remote locational data and pushing it to the remoteTronPlayer objects location data. Because it should be capable of accepting connections from remote ports, it implements a ServerSocket. Because this has to stay up in a continuous loop, it is also implemented as a thread.

methods

1. PlayerDataReciever(int localPort)
   1. sets up a SocketServer based on localPort
2. kill()
   1. closes the server
3. run()
   1. continuously tries to accept a remote connection. Parses and sends the appropriate locational data to the remote player object when received

## TronPlayer

TronPlayer is the object that holds the information that is needed for Display to draw the players on the board. It holds the x/y coordinates, direction, color, score, and name. Also, it updates the tail that follows the head of the player.

*TronPlayer General Methods:*

* hitWall(Display d)
  + This essentially is a boolean determining whether or not a TronPlayer object has crashed into a wall.
  + It traverses through a list of all of the players in the game, and detects if one of them has hit a wall by checking the relative location compared to the boundaries.
* action(Graphics g, Display d)
  + This void method simply sets the TronPlayer in a direction provided by another action method (e.g. AITronPlayer’s nearWall()
  + If a player has hit a wall or boundary, it crashes and resets the game.
  + Finally, as the players move, they have a trail left behind them, creating temporary walls to use to beat opponents.
* outOfBounds(Display d)
  + Returns whether or not a player has hit a wall as a boolean.
* reset()
  + Resets the game.
* crash()
  + Crashes the game. The difference between this and reset() is that crash() is a void method that sets a boolean to true once the game has ended, which is used to restart the game.

## LocalPlayer

Local Player, which extends TronPlayer, is your player. Its action method is called when display wants to draw the player. In this method, the player adds new points to the tail based on its current location and direction. Also, it determines whether the player is dead by scanning if it has hit a boundary or wall (tail). Finally, the method sends its newly added location to the remote player.

methods:

1. action(Graphics g)
   1. sets new X and Y coordinates based on the direction and velocity
   2. checks if the player hits a wall or boundary
   3. adds new coordinate points in tail ArrayList if the tail length is less than the specified max length
2. outOfBounds - boolean
   1. if the player is outside of the boundaries
3. hitWall - boolean
   1. if the player hits a wall

## RemotePlayer

The RemotePlayer class just updates the tail positions of the remote player

method:

1. addLocation( int[] xy )
   1. adds an array with two values to act as a point into the arrayList of tail points

## AITronPlayer

The AI TRON Player acts like a normal TronPlayer, but it is able to determine whether to move. It turns when it approaches a boundary or wall. It is also able to determine which direction to turn based off of the distance between itself and walls/boundaries. If it’s closer to a wall on the left it will turn the opposite direction.

methods:

1. action(Graphics g)
   1. moves the AITronPlayer and adds a new point to the tail
2. nearBoundary - boolean
   1. determines if the AI TRON Player is near one of the edges
3. nearWall - boolean
   1. determines if the AI TRON Player is near one of the tails
4. turnTowardsPlayer
   1. makes the AI turn towards the closest direction to the player
5. smartTurn
   1. turns left or right depending on which quadrant it is in and which direction it is moving
6. outOfBounds - boolean
   1. checks if the player is outside of the boundaries
7. hitWall - boolean
   1. checks if the player hits a wall

# Testing

Testing of this code is primarily done through the use of the AI TRON player which checks aspects such as background, scoring, crashing, and refresh speed. The networking aspect was tested through extensive trial and error through the use of multiple computers. A series of code-based tests will be run using the JUnit structure, testing the various classes and methods.