Wii Educational Tools
Final Report

Colorado School of Mines
Field Session Summer 2010

Joseph Cirone
Seth Daugherty
Chris Loew
Gilliane Oswald
Kyle Shepard

Table of Contents

Abstract …………………………………………………………………………………… Page 1
Introduction …………………………………………………………………………… 2
Functional Requirements …………………………………………………………… 2
    Framework ………………………………………………………………………… 2
    Binary Tree Game ………………………………………………………………… 2
    Network Routing Game …………………………………………………………… 3
Non-Functional Requirements ……………………………………………………… 4
Use Cases ……………………………………………………………………………… 5
    Binary Tree Game ………………………………………………………………… 5
    Network Routing Game …………………………………………………………… 6
System Architecture ………………………………………………………………… 6
    Wiimote Manager ………………………………………………………………… 6
    Wii Event Parser ………………………………………………………………… 7
    Game Event Parser ……………………………………………………………… 8
    Game Event Listener …………………………………………………………… 11
Game Design ………………………………………………………………………… 11
    Binary Tree Game ………………………………………………………………… 11
    Network Routing Game …………………………………………………………… 14
Testing ………………………………………………………………………………… 18
Implementation Details ……………………………………………………………… 18
Lessons Learned ……………………………………………………………………… 19
Future Work …………………………………………………………………………… 20
Glossary ………………………………………………………………………………… 21
Appendix

Abstract

The Wii Can Do It! educational games are an interactive, fun tool to teach introductory Computer Science concepts. These games will be used in the upcoming CSCI 101 courses offered to CSM students as part of the new distributed science core. The course will be taught by the project client, Keith Hellman. We developed two games, one to illustrate network routing concepts and one to introduce binary trees. They are multiplayer games that use Wiimotes for user input and are programmed in Python for use on Linux platforms. We also developed a "Wiimote framework," which simplifies the interaction between Wiimotes and the games. This framework should simplify development of future games by allowing work to focus solely on game logic.

The framework interprets input from a Wiimote and converts it to a standardized "Wiimote Event". A second portion of the framework takes a stream of these Wiimote events and translates them into game events. Our input interpretation method is a solid foundation for the client's future Wiimote games. To minimize development time, we used two existing libraries, which allowed us to focus on the games and the underlying framework. The first library is cwiid, a library that interacts with the Bluetooth stack and Wiimotes(Game controllers for the Wii console system that transmit Bluetooth signals). The second library is pygame, a 2D graphics library and game engine, which was used to develop the graphical interface for both games.

Introduction

CSCI 101 is a new course offered during the fall 2010 semester designed to give students a general understanding of basic computer science concepts. To make the class more interactive and engaging for students, the project client, Keith Hellman, is looking to design several games that interface with the Nintendo Wii controllers ("Wiimotes"). The goal of team "Wii Can Do It!" was to develop a Wiimote framework that can be easily integrated with more of these games as the class evolves. This framework takes care of all the input from Wiimotes, so future developers can focus primarily on game logic. Another goal was to use this framework to implement two initial games that can be used in the fall. One game has players racing to create a balanced binary tree, and the other game has players working to route packets over a simulated network.

Functional Requirements

This project can be broken into three major components: the Wiimote framework, the balanced tree game, and the network game.

Framework

The framework consists of four parts: Wiimote Manager, Wii Event Parser, Game Event Parser, and Game Event Listener. These components work together to convert the stream of raw binary data broadcast from the Wiimotes into a consistent format that is suitable for game development. Since the Wiimote uses both acceleration data to track a user's motion and button presses, this framework needed to condense this information into simple events. The Wiimote Manager and Wii Event Parser interpret data from the Wiimote and pick out Wii Events. Wii Events are an abstraction to describe the simplest user actions such as "player 1 pressed A", or "player 3 flicked the Wiimote to the right". The Game Event Parser receives a stream of these events, looks for sequences of events, and creates a "Game Event". Game Events are simply function calls within the game logic, along with any associated parameters like the direction of Wiimote movement. To build a game that uses Wiimotes, a developer simply needs to create a file telling the Game Event Parser which game function to associate with what sequence of Wiimote actions; all other low-level Wiimote details are abstracted away by the framework.

Binary Tree Game

The objective of the binary tree game is to make a fun way to introduce binary tree concepts to students. The game requires users to manually manipulate several aspects of creating and balancing a tree, such as using rotation algorithms to balance the tree and determining what a "worst case" scenario for inserting numbers into a tree would be.

This game is currently implemented with a single game mode, which is a three-on-three game. Each team is supplied with a single node that starts a tree, an insert slot holding the first number to be inserted into the tree, and a queue containing numbers that need to be inserted into the tree. The first player of each team, called the sorter, is supposed to “sabotage” the opponents queue by rearranging the numbers so that the opponents have to rotate their tree as much as possible to re-balance the tree. The second player of each team, called the inserter, must insert new data into the tree. The player flicks their Wiimote left or right to correctly sort the number in the insert slot based on the number in the node beneath it, starting at the root node of the tree. Insertions are restricted to the proper leaves of the tree. That is, left children must have values less than their parent nodes and right children must have greater values. The third player, the rotator, has to balance the tree, when insertions unbalance the tree. They do so by traversing the tree and rotating select nodes. A prototype of what the game will look like is shown in Figure 1, and more detailed gameplay is described in the Use Cases section below.

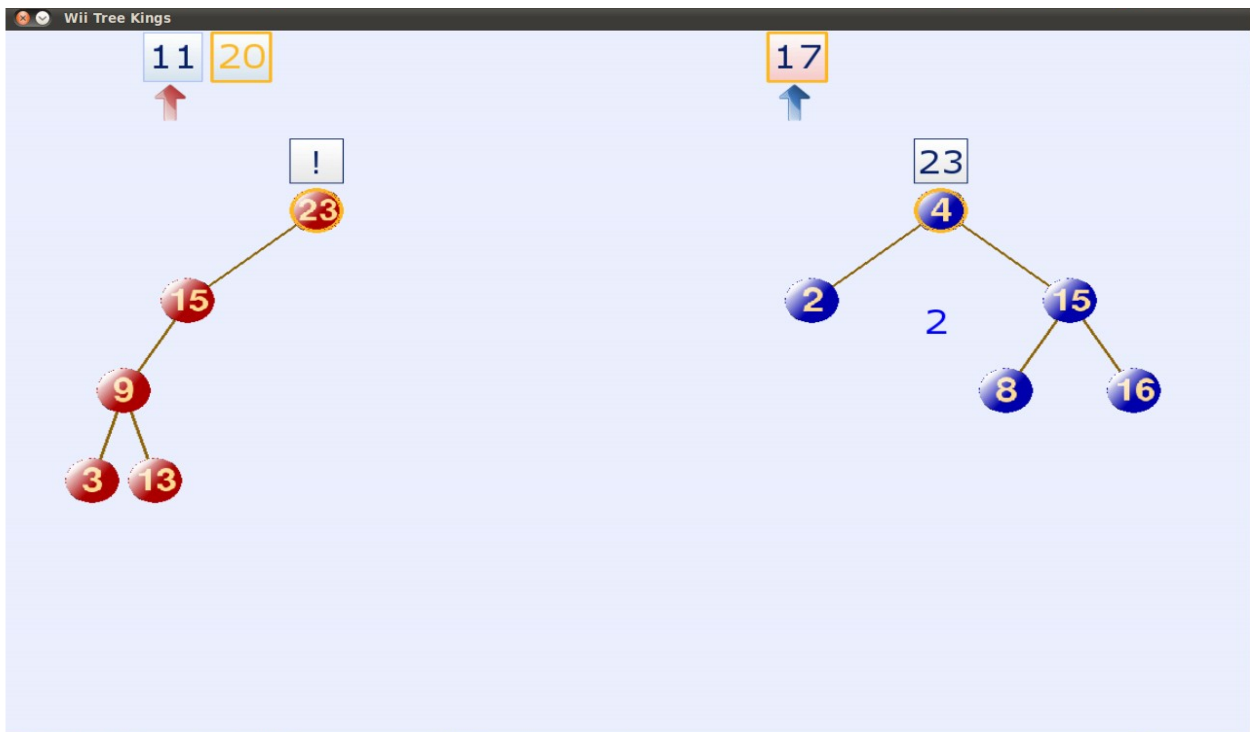

Figure 1: Screenshot of the binary tree game. The red team's tree is unbalanced, so the rotator must balance it before the game can continue.

Network Routing Game

The network game is a port of an existing Wiimote game written in C# for Windows. Using our framework required such an extensive overhaul of the existing game that we ended up coding the game from scratch, using the existing code simply as a guide to the logic of the game. The goal of this game is for players to route packets across a simulated network in the most

efficient manner possible. In this game, each player controls a node in the network. When packets arrive at a player's node, he or she has to flick the Wiimote in the proper direction to send to the destination node. Each connection, or edge, is assigned a weight, and the player needs to try to use a least cost path to get the packets to their destination. A screenshot of our game is shown in Figure 2. (A screenshot of the original game is included in Figure 11 in the Appendix.)

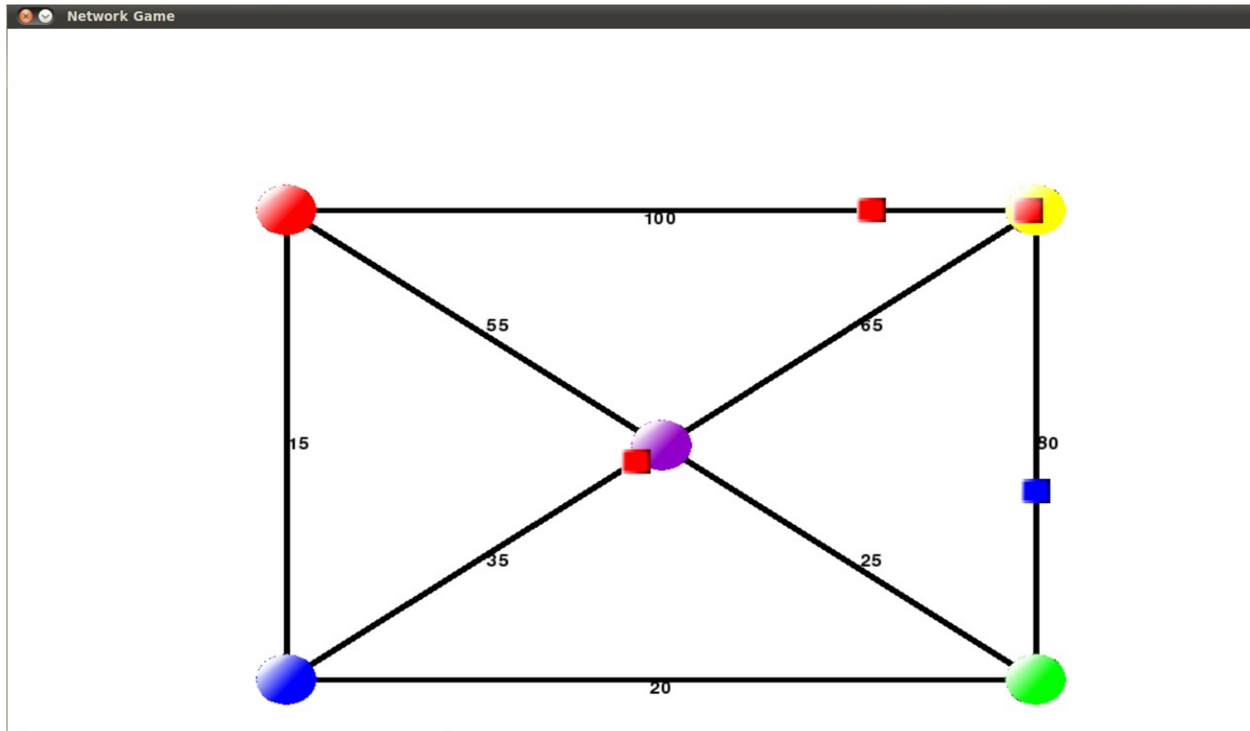

Figure 2: Screenshot of the network game. Packet colors indicate the destination of the packet.

Non-Functional Requirements

For documentation, we used a combination of docstrings for all classes and functions, as well as creating a full documentation of our software using reStructured Text. ReStructured Text is a simple markup language designed to be both readable in its raw form and easily converted to other formats like HTML. We used the Bazaar distributed version control system to store our code. All of the code was written in Python, to run on Debian systems.

Use Cases

Binary Tree Game

Player 1 manipulates the opponents’ queue to create an insertion pattern that causes for the most re-balancing of the opposing team's tree. They do this by switching numbers in the queue.

1. Player 1 selects a number in the queue
2. Player 1 flicks the number left or right
    a. If there is a neighboring number in the direction flicked, the two numbers switch locations
    b. If there is no number in that location, nothing happens.
3. Repeat 1 through 3 until sorter is satisfied with arrangement

The job of player 2 is to insert numbers in their queue into the tree

1. Player 2 receives a number to be sorted into their tree.
2. The player flicks the Wiimote down one branch of the tree
    a. If the player chooses to go down the correct path, the number will move to the node in the specified direction
    b. If the player chooses to go down the wrong path, no action will be taken, and the player will be unable to move the node down the tree for 3 seconds
3. Repeat 2 until the number reaches the end of the tree and is sorted
4. Return to 1 on condition that the tree is balanced

Player 3 balances the tree whenever it is needed

1. The tree is unbalanced, so player 2 is not receiving new numbers to sort
2. Player 3 flicks up, left, or right to traverse the tree to a node that needs to be rotated
3. Player 3 selects the node with the Wiimote, and rotates the Wiimote
4. The tree rotates based off of the direction the Wiimote is rotated.
5. Repeat 2-4 until the tree is balanced

Network Routing Game

1. Player receives a packet.
2. Player flicks Wiimote in a particular direction
    a. If there is a path in that direction, the packet is sent down that path
    b. If there is no path in that direction, no action is taken.

System Architecture

As mentioned above, the design three main components: the Wiimote framework, the Network Routing Game, and the Binary Tree Game. The Wiimote framework can be further broken down into four main pieces: the Wiimote Manager, the Wii Event Parser, the Game Event Parser, and the Game Event Listener. The Wiimote Manager handles synchronizing Wiimotes with the computer and direct interaction with the Wiimotes. The raw data from the Wiimotes is sent to the Wii Event Parser, which converts it to an abstraction we have dubbed a "Wii Event." The Game Event Parser receives these Wii Events and then converts them to "Game Events". Game Events are sent to the Game Event Listener, which makes the corresponding function call in the game itself. The details of these modules are explored below.

The project made use of two existing libraries to provide needed functionality. The first is Pygame to implement the two games. Pygame is a well-documented and popular two-dimensional game engine. This is a game engine that controls most of the games graphics. It also handles events from standard inputs, such as using the mouse and keyboard, used with the menu systems. We decided would not prove too difficult to learn the engine because of the documentation, and we could then focus on providing solid game logic as opposed to constantly troubleshooting graphics issues.

The other main library we used was cwiid, which provides an interface to communicate with Wiimotes. By using cwiid, we do not need to directly work with the Bluetooth stack in order to receive information from the Wiimotes. This library is the foundation of the Wiimote Manager.

Wiimote Manager

The entire Wii Event Parser is comprised of two parts. The first is the Wiimote Manager. This module keeps track of all Wiimote control classes currently associated with the program. The Wiimote control object contains the location of a Wiimote object made by the cwiid library, the id the manager has assigned it, and a callback function for cwiid to use to send data over a socket to the Wii Event Parser every time the Wiimote reports information. This function creates a raw data object, compresses it using Python's "pickle" module, and sends it over the socket designated for raw data. In this implementation, the socket acts much like a pipe, or a specific way to transfer data between the Wiimote Manager and the Wii Event Parser.

The purpose of the socket between the Wiimote Manager and the Wii Event Parser is for abstraction. With this implementation, the Wii Event Parser can read data from any source, as long as it is compressed into a raw data event and sent over a the appropriate socket. The raw data class itself is abstracted, with one instance of the class passing button press events and a second passing a stream of accelerometer events. Each Raw data event contains the timestamp of the event and the appropriate data for the event. The accelerometer raw data contains the acceleration values for the x, y, and z directions. The button class contains a binary string of what buttons are currently pressed on the Wiimote.

Wii Event Parser

The Wii Event Parser module reads data from the socket that raw data is being sent to, tests for appropriate events, and sends those events over a socket to the Game Event Parser. The parser contains a list of accelerometer, button control, and roll pitch classes – one for each possible Wiimote that is on the Bluetooth stack. These classes test the raw data for events in different ways:

Accelerometer.

The accelerometer class contains the logic in testing for a flick. For the current version, this means testing the accelerometer data against thresholds of what would constitute a flick of the Wiimote. If the data passes one of these flicks, the accelerometer data then calls a magnitude tracker class. This keeps track of if the accelerometer is in a lockout state, as well as computes the magnitude of a flick. The magnitude of a flick is essentially how hard the user flicked the Wiimote in a certain direction. The lockout of an accelerometer prevents multiple flicks from being registered with one motion by not allowing a flick event to be sent more than every three tenths of a second. The magnitude of a flick is found by repeatedly comparing a given acceleration to what was previously passed to the object, waiting for the value of the acceleration to decrease.

Button Control.

The button control class contains the prior state of what buttons are pressed in a set of boolean variables. When the class receives a new set of data, the button control first converts the information into a second set of boolean variables. Once this is complete, the class checks each button to see if there is a discrepancy between the two sets, firing off an event every time it sees a change.

Roll Pitch.

The Roll Pitch class takes the acceleration values of a given piece of raw data and calculates the approximate roll and pitch of the Wiimote, assuming it is being held still. This is done by taking the inverse tangent of two accelerometer values. The roll is calculated by taking arctan(x / z), while the roll is calculated by taking the arctan(y / z). The resulting roll and pitch value is then compared against a prior threshold, seeing how many roll events need to be fired, if any. The threshold for a roll is every .3 radians, and a pitch event is every .2 radians. This means

that every time the Wiimote gets rolled .3 radians in a direction, the parser sends out an event symbolizing the roll.

When an event is triggered, each of the subclasses listed above create a Wii Event, compress it using the pickle module, and send it over a socket to the game event parser. Wii Events represent the simplest action a user would perform with the Wiimote, such as “press A”, “release A”, “flick right”, and so on. The following events are currently supported by the Wii Event Parser.

- buttons (up, down, left, right, a, b, +, -, 1, 2, home)
- “flick” (x, y, z)
- Roll
- pitch

This list can very easily be expanded to support other types of Wii Events, such as input from the Wii Nunchuck attachment, by adjusting the name of the event and flick.

Each Wii Event holds three pieces of information: the ID of the Wiimote (an arbitrary number assigned when a Wiimote is connected), the name of the event (e.g., “A”, “FlickX”, “Roll”), and a modifier. This modifier holds different information depending on the action that triggered the Wii Event, as described below.

Buttons
For button events, the modifier is simply +1 or – 1 to determine whether the button was pressed or released.

Flicks
For flick events in any of the three directions, the modifier will be a positive or negative number with the sign determining direction along the specific axis. The magnitude of the modifier is equal to the peak magnitude of the flick given by the accelerometer sensor.

Roll and Pitch
Just as with the button events, the modifier is merely a +1 or -1 to designate direction of the roll, for example the sign determines whether it is a positive roll (clockwise) or negative roll (counter-clockwise).

Once a Wii Event has been triggered, the Wii Event Parser sends the Wii Event over a network socket to the Game Event Parser.

Game Event Parser

The Game Event Parser is intended to transform sequences of atomic Wii Events into game logic. The Game Event Parser makes it very simple for game developers to bind sequences

of Wii Events to a game function call. The developer is spared from having to rewrite any logic that listens for combinations of Wii Events. Developers only have to create a Python file that lists all of a game's bindings, which are simply function names associated with a particular sequence of Wii Events.

For a more concrete explanation, consider the following list of bindings, which is the actual syntax for declaring bindings.

```
bind_functionA = ['ButtonAPress', 'ButtonARelease']
bind_functionB = ['ButtonBPress', 'FlickX', 'ButtonBRelease]
bind_functionC = [('ButtonAPress', 'ButtonBPress'), 'FlickX', ('ButtonARelease',
'ButtonBRelease')]
```

The first binding connects a function called *functionA* in the game class to a simple press of the A button. The middle binding represents selecting something and flicking it left or right, an action we use several times in our games. The last binding shows the flexibility of the Game Event Parser. Bindings can be composed of Python's collection data types to represent different variations of events. In this case, the tuple (*'ButtonAPress', 'ButtonBPress')* indicates that the A and B buttons should be pressed simultaneously.

A high level overview of the Game Event Parser is shown in Figure 3 below. The parser is constantly listening for Wii Events. Every time it receives one, it iterates through the list of bindings looking for sequences that match the received Wii Event. For each event binding, the parser stores a pointer to the next event in the sequence. If the received Wii Event matches the binding's next pointer, the program will move the pointer to the next event in the binding's sequence. If the received Wii Event does not match this pointer, then the user is performing a different sequence of actions, so the program resets the pointer to the beginning of the binding's event sequence.

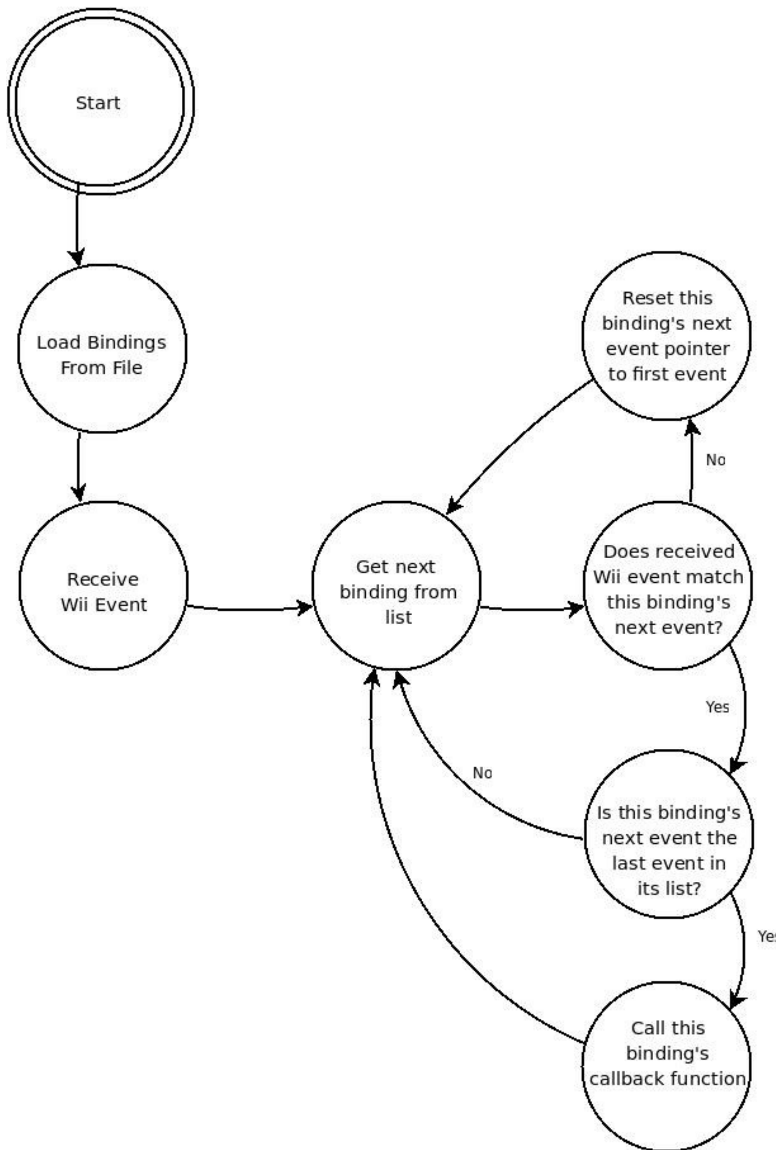

Figure 3:Game Event Parser data flow

The Game Event Parser relies heavily on Python's dynamic typing scheme. Since container types in Python do not care what type of data they store, bindings can be very flexible. Game developers can specify a set of Wii Events that should happen simultaneously simply by inserting a tuple inside the binding's event sequence list. Optional events, like “Press A” or “Press B”, can be created by inserting a Python set data type into the list.

Another unique Python feature the Game Event Parser uses is every objects’ power of introspection. The list of bindings is actually a subclass of the main Game Event Parser class. When the parser starts, the object will look at all of its internal member functions and variables. Variables that are prefixed by a specified word are considered to be bindings, so they are all gathered into a list to allow easy iterating. When a game function needs to be triggered, the Game Event Listener changes the scope of the variable name to the game class and calls the

corresponding game class method. A list of information about the sequence, such as the magnitude and direction of the flick, roll and pitch data, and duration of the sequence is sent as a parameter to the game function.

Some Wii Events, like pitch and roll, must be handled differently than simple events like pressing a button. For instance, The Wii Event Parser is configured to be sensitive to pitch and roll movements, so it sends pitch and roll Wii Events very frequently – as often as a few times per second. Normally, when the Game Event Parser receives a roll event, the player is not really trying to perform a roll; often, they intended to do a flick or are even holding the Wiimote relatively still. To account for all these extra roll events, the Game Event Parser filters out all roll events unless a binding explicitly asks for them. Also, roll events come from the Wii Event Parser in small chunks of 0.3 radians. If the player tries to perform a roll of, say, one radian, the Game Event Parser will receive 3 atomic Wii Events in rapid succession, which it will integrate into one event.

The Game Event Parser includes a small selection of "pseudo-Wii Events". These events do not correspond to actions on the Wiimote, but instead are events like "timeout" or "lock". If a binding specifies a timeout, the Game Event Parser will stop listening for Wii Events for the specified time. A "lock" will make the current binding the sole active binding; so it is impossible for other sequences to trigger game events. Our games are too simple to use this functionality, but we decided that a "pseudo-Wii Event" was a common enough use case to justify inclusion into the Game Event Parser.

Game Event Listener

The Game Event Parser runs in its own process, so it does not have access to the internal functions of the game object. In order to make game function calls, the parser sends Game Events to a simple utility called the Game Event Listener. The Game Event Listener runs as a thread in the game itself, so when it receives a Game Event, it is able to look up and call the proper function in game object.

The other function of the Game Event Listener is to start the two parsers. This is mostly a convenience for game developers; they do not need to start the parser processes because the Game Event Parser automatically starts them.

Game Design

Binary Tree Game

The Binary Tree Game has four main classes, the Player, Team, Tree, and TwoTeamVersusMode classes. Their part in the design of the program will be emphasized below, but a full UML is included in Figures 6-8 in the Appendix.

The Player class is an abstract class that contains a 'Wiimote ID' to connect the player to a Wiimote and abstract functions for all the game actions players can take. These methods are

defined in its three children classes, one for each role in the game: sorter, inserter, and rotator. This allows the same Wiimote inputs to do different actions based on a player’s role. The Player class also holds a current variable which specifies something unique to each player and a selected boolean that tells whether a player has selected the object they are currently on. Each player also has an association to their team. However, every instantiation of the Player class may require more variables(such as the Sorter Class described next). To allow for future extension, Player objects are initialized with a dictionary that must hold 'player' and 'team' as keys to the Player's Wiimote ID and Team object respectively. This dictionary can then hold more data that each of the individual subclasses may need as well. The Player class also holds dummy versions of all of its subclasses function that can be called by Wiimote actions. This is to ensure all Players can do whatever they want with their Wiimote without having an accidental call to a function that is not defined.

The Sorter Class is the first subclass of the Player Class. A Sorter object makes use of the dictionary mentioned above. Since a Sorter's purpose in the game is to 'sabotage' the opposing team's queue, a Sorter object must have a reference to that opposing team's queue. This was accomplished by passing an extra parameter in the dictionary that held the value for the opposing Team object. For a Sorter object, the current variable holds an index of the opposing team's queue that designates where in the queue the Sorter is. The Sorter's only other method to note is its shove method. This is the method called when a player flicks their Wiimote, regardless of direction. For a Sorter, this method either moves the number in the queue that the Sorter is currently on, or swaps the numbers in the queue if the Sorter has selected the number they are on.

The Inserter Class is the next subclass of the Player class used in the game. The Inserter holds an extra variable called center. This variable holds the root node from the Inserter's Team's tree. This variable determines where the Inserter starts sorting a number from, as well as where the start of the tree is to check to see if it is balanced. The Inserter's current variable holds the value of the number that needs to be inserted. The Inserter's shove function communicates with the Team's Tree object to check if the direction was the valid direction based off the Inserter's center node's value.

The Rotator is the final subclass of the Player Class. The Rotator's current variable designates the node in the Team's Tree that the player is currently at. The Rotator's shove method allows traversal of the tree, with left moving to the current node's left child, right moving to the right child, and up moving to the current node's parent. The Rotator also holds the two rotate functions, which allow reordering of the Team's Tree to try and balance it.

The Team class holds pointers to all three of its players and their game board, which consists of the team's queue and tree. The queue is merely a list of numbers rather than an actual collection version of a queue as used in the Network game. This is because collection queues do not allow easy manipulation of internal values, which is essential for the sorter to do their job. The Tree is its own class and will be discussed next. The Team class also handles all functions that affect its tree and then checks to see if the action has balanced the tree. If it has, the Team class will pop the next number from its queue and provide it to the inserter.

The Tree class holds most of the central game methods specifically checking if the tree is balanced and allowing rotation within the tree. Also, the Tree class allows insertions and checks if an Inserter has attempted their insertion correctly at every level through the tree.

The TwoTeamVersusMode class is the class that builds both Team objects and all 6 Players and handles all the Wiimote interaction for our Game Mode. It is also a subclass of the GameMode class, which allows for communication with the GUI and a way to return to the Main Screen of the Game. The TwoTeamVersusMode class first builds both teams and then proceeds to add players based on Wiimote input. Then, after all 6 players have been added, it starts the game. Now, the gameplay starts and players are able to perform their various actions, while the TwoTeamVersusMode class determines which actions correspond to which player. The TwoTeamVersusMode class also interacts with the GUI to tell it when to update the screen.

For a state-based description of the binary tree game's gameplay, see Figure 4. To start, players will connect their Wiimotes and then begin the game. Once this happens, a dialog will appear that prompts a player to fill each role for each of the two teams (sorter, inserter, and rotator). Once all 6 roles (one of each role for each team) are filled, the actual game begins.

Each team gets a random sequence of 6 numbers to sort as well as random number that is the root of their tree (for a total of 7 number to make a full tree). The sorter will be trying to sort the opposing team's sequence of numbers so that they spend the most time rotating the tree after insertions, giving the sorter's team more time to finish their tree. This sequence of numbers acts as a queue so that, whenever the tree is in a balanced state, the first number in the sequence can be placed in the 'insertion slot'. This 'insertion slot' is simply a space on the screen to tell the inserter for each team that they need to insert that number into their team's tree before they can continue. The inserter does this by flicking the Wiimote left or right to sort the node; the number then moves to the next node if it was sorted correctly (i.e., the player motioned left if the insertion number is less than the node at or right if greater) otherwise a timer appears, and the player is not allowed to continue sorting for 3 seconds. This prevents people from randomly flicking the Wiimote to sort the tree.

This continues until the insertion number is a leaf of the tree, at which point the game will check if the tree is still balanced or not. If it is balanced, the first number of the team's sequence will get placed in the insertion slot and the game starts over again. If the tree is not balanced, player 3 must rotate the tree so that it becomes balanced. Player 3 does this by selecting a node (pressing the A button while their cursor is on the node) and then rolling the Wiimote in the desired direction and then letting go of the node (releasing the A button). This process continues until a team reaches a balanced tree and has no numbers in their queue. At this point, that team has won, and the game returns to the main screen so play may continue if users wish.

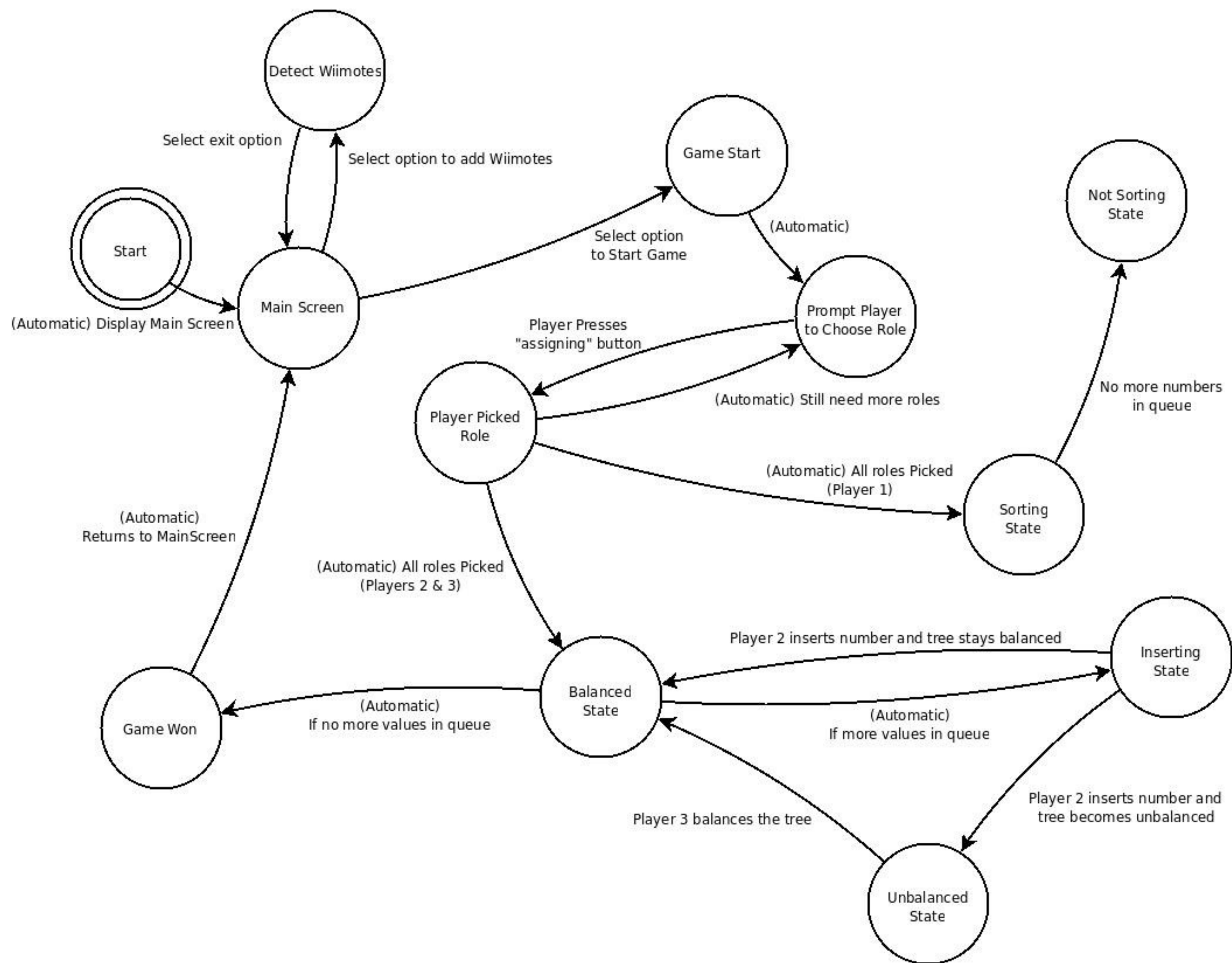

Figure 4: Binary Tree Game Finite State Diagram

Network Routing Game

The Network Routing Game has two minor classes, the Player and RouterNode classes. These classes are then used to help the major classes controlling game logic. The major classes are the Game and Map classes.

The Player class is used to represent human players in the game. This class holds a wiiID variable which is an integer that connects the Player to a Wiimote's data in the Wii Event Parser. Player objects also hold a color variable that connects them to a specific RouterNode when the Game is being played.

The RouterNode class represents a Network Node for use in the game. Objects of this class have a color variable that identifies what color Packet or, more appropriately, what string they accept, since in the Game logic Packets are merely strings of color names, “RED”, “BLUE”, “GREEN”, and so on. RouterNode objects also hold a queue that holds all the “Packets” at that node, as well as where the node is located on the screen.

Neither of these classes provides significant functionality, but they are absolutely necessary for the Game and Map classes to function.

The Game class handles all the information between the Map class, where all the actual game functionality happens, and the Game Event Parser, which calls functions in the Game class whenever a binding is completed (in the case of this game, the binding was a simple Flick Event). The Game class holds all the player objects and the Map object that is actually being played. The Game class has one major function, filterInput. The filterInput function is what is directly called by the Game Event Parser. This function takes the player ID sent by the Game Event Parser and checks it against the wiiID of all the player objects in the game. If the function finds a match, it continues and tries to find the RouterNode object in the Map that corresponds to the player's color. If it is able to find a RouterNode that means the player is actually playing. Next, the function normalizes the “flickx” and “flickz” data to be -1 if the value was negative or 1 if the data was positive (the data is already zero if there was no flick in the corresponding direction) and then tells its Map object to try and send a Packet from the RouterNode the Player controls.

This normalization is used to simplify the arrangement of the graph to only use 8 directions, since the accelerometer data used for the flicks is so sensitive. Different people can flick with different strengths and with a large and inconsistent range from the desired direction. Because of these variables, using the strengths of the different flicks to determine an angled direction is extremely difficult – if not impossible – to do without infuriating the user. However, this does not mean the graphical versions of the maps are limited to these eight directions. Edges may vary in their direction, but only one may be in each of the eight sectors of direction. This is all implemented within the Map class.

The Map class holds all the major functionality for the actual gameplay and all the variables that need to be tracked, such as the score, the maximum number of packets to add to the map, the number of packets the players have routed successfully, and the penalty for losing a packet. Map objects can make edges between two of their RouterNodes by use of the makeNeighbors function. This function takes the two RouterNode objects that will be connected and the weight between them. This function will take the coordinates of the two RouterNode objects and find the direction between the two, by subtracting the x and y values of the coordinates and dividing each by the magnitude of their associated vector. These new x and y values are then rounded to -1, 0, or 1 to allow the more varied displays mentioned above. After collecting this direction information, the makeNeighbors function builds a dictionary with both of the RouterNode objects as keys to their specific direction to the other node. The function finally adds the string “weight” as the key to the numeric value of the weight given as a parameter to the function.

Map objects also have the trySend function. This function is called by the filterInput function described above. This function takes a RouterNode and an x and y direction and then determines whether the provided node actually has an edge in the provided direction. If the node does, the Map starts a timer based on the weight of the edge. This timer corresponds to the length of time

the packet is traveling along an edge, so that higher weighted edges actually take longer to send packets. When this timer finishes it calls the tryReceive function. The tryReceive function increments the score for the players with the weight of the edge used, then, checks if the Packet received is the same color as the node, if so, it exits the function. If the Packet isn't the same color, the function checks if the node has room in its queue for the new Packet. If the node does not, the Map object adds the penalty for losing a Packet to the score. The score, in this game, gauges the overall performance of the team. It can be compared to the score of what a perfect game would be to see how well a group of players did. Otherwise, the node receives the Packet, and the Map checks if the node that received the packet is controlled by a computer. If the node is controlled by a computer, the Map finds the least-cost path for the Packet and then calls trySend function to send it in the right direction. Otherwise, the node waits for a player to flick their wiimote then calls the trySend function in the appropriate direction.

The final functionality that Map objects hold is the ability to add packets at random to random nodes. The Map continuously starts a timer for between 0 and 8 seconds after which the Map randomly picks a node that doesn't have a full queue and then randomly picks a Packet color that is not the color of the selected node. After both of these are picked, the Map object calls the tryReceive function and actually adds the Packet to the node's queue. The Map continues to do this until the max number of Packets have been added, after which it waits for all Packets the node's hold to reach their destinations and then ends the game.

The gameplay design for the routing game is rather simple, as illustrated in Figure 5. When players connect their Wiimotes, they are assigned a color; this color also designates which node in a map the player will control. Once all players have chosen their color and a level has been selected, the map will appear and packets will randomly appear on the screen at random nodes. Players must then ‘flick’ their Wiimotes in the direction of a neighboring node to route packets to the node with the same color. Score is determined by the weight of each edge used, so the lower the score the more efficient the routing (i.e., the lower the score, the better the team did). Some maps may have more nodes than the number of players; in this case, a simple script will control the non-player nodes and will send packets along the most efficient route automatically.

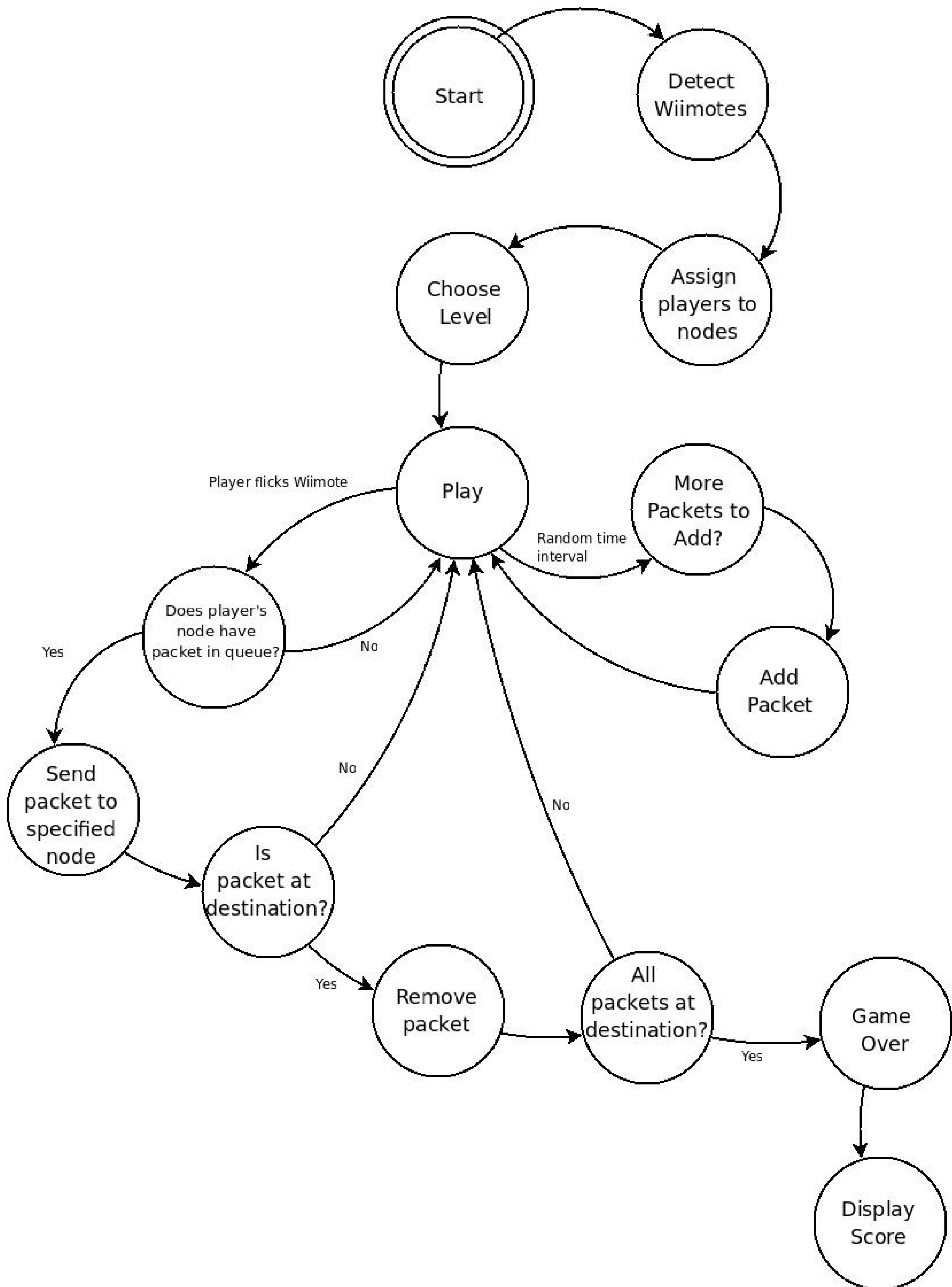

Figure 5: Network Routing Game Finite State Machine

Testing

We have several layers of tests for all the modules. The basic classes of the Wii Event Parser, such as Accelerometer and button control, have unit tests for their functionality. The network game also has mock-up unit tests. The Binary tree game also contains some unit tests for the rotate and sorting algorithms. Above that, there is logging at the Raw data, Wii Event, and Game Event levels. Each of these logs can be passed through the appropriate socket, allowing for recreations of bugs to be repeatedly tested to ensure it is fixed.

Implementation Details

As mentioned in the project requirements section above, the client had several implementation requirements. The most important, and the most obvious, is that all code should be written in Python. The client left the choice of Python version up to us. This may sound like a trivial decision, but Python 2.4, 2.5, 2.6, and 3.0 are all widely used on different systems, and code is not always portable between versions. We decided to develop on Python 2.5, because that was the version installed in the Alamode lab. However, one of our testing laptops ran version 2.6 and encountered only minor incompatibilities with starting and killing the processes that ran the parsers. Our framework and games should run with little difficulty on any implementation of Python 2.5 or 2.6. We did not test on Python 3.0.

Besides using Python, The client also specified that our project should be developed for Debian-based systems, because that is the operating system he will be using when teaching the class. Luckily, the Alamode lab runs Ubuntu, which is derived from Debian, so we had access to a suitable development environment. The only other implementation requirement the client specified was that we use a Python program called WMGui to read raw data from the Wiimotes. Investigating WMGui revealed that it is just a front-end to the cwiid library, so we used cwiid communicate with the Wiimotes.

After cwiid, the next most important implementation decision we made was to use the Pygame library to handle graphics for our games. We chose Pygame instead of other Python graphics libraries primarily because of the community surrounding Pygame. It is much more widely used than its competitors, which means it has much better documentation, tutorials, and support forums. Also, the complexity level of our games did not seem to warrant an OpenGL-based graphics engine, though our framework would certainly enable future game developers to take that route. Pygame was lacking in some respects, particularly in its drawing functions. For instance, Pygame does not support drawing anti-aliased circles. Overall for our games, Pygame was a very good fit because it balanced ease of use, features, and programmer documentation.

Apart from external libraries, we had a few design decisions that need explanation. First, we decided with the client that communication between all parts of the system should be done with network sockets. This made it very easy to develop and test each piece in parallel. For example, instead of needing to build from the ground up – from reading raw data towards the game event parser – we were able to develop each piece in parallel by sending “fake” input over socket. Also, the client has plans for future network-based games. Since sockets are a fundamental part of the framework, developing network games will relatively simple. To ease our development process, we chose to use the connection-less UDP protocol instead of the more reliable TCP. This allowed us to, for example, test the Wii Event Parser without needing to have

an instance of the Game Event Parser running and listening for connections. Also, the relative importance of individual packets is small, so the framework can easily tolerate a few lost packets. However, future developers may choose to change communication to TCP.

Another design decision needing justification is our use of processes instead of threads for each component of the framework. It might seem natural to have the game process start the Wiimote Manager, Wii Event Parser, Game Event Parser, and Game Event Listener in separate threads. When the game is over, the process would simply kill all the threads. This is the direction we took initially, but we quickly found that threads are very difficult to use properly. We spent countless hours tracking down errors resulting from threads before the client advised us to de-thread and use processes instead. In retrospect, it was foolish to use threads in the first place. Almost all communication between the components happens over sockets – as opposed to directly sharing memory – so threads are superfluous.

Running each component as a separate process greatly increased the flexibility of our games. The games can function as a stand-alone application, starting and stopping the subprocesses when needed, but the processes can also be run separately. For example, while developing, we would often want to run the game several times without the hassle of re-syncing Wiimotes every time. Since the Wiimote Manager runs in its own process, we were able to sync the Wiimotes once, leave the process running, and continue testing. This flexibility will also be very useful for the client when he is teaching with the games, because he will not have to waste valuable class time re-syncing Wiimotes.

The framework does still run one thread. Since the Game Event Listener directly calls functions in the game class, it needs to have access to the same memory space as the game. Games must start the Game Event Listener thread, which will start the Wii Event Parser and Game Event Parser processes. However, for the game developer, the burden of using the thread in the current framework is absolutely minimal; they need only create a Game Event Listener object then call that object's startup method.

Lessons Learned

- Threads, while easy to implement, are very difficult to implement *right*. Our original strategy of combining the parsers involved threads, and we ran into many issues with them not starting or stopping properly. Using processes instead is a much more reliable and simple way to allow for multiple functions to be executed at once.

- Sockets are very useful when trying to decouple a program. Especially once you remove the overhead of connecting and disconnecting, they provide an efficient, solid interface between parts of a program. They are also useful in their ability to abstract data, and allow for easy logging and use of those logs. This is because the process listening to the socket does not care how the data was created, it just reads it and performs the necessary functions.

- Python is a very powerful programming language, yet is very easy to use. Python also has a very extensive list of modules. If you need to do a specific task, there is a good chance of an existing Python module to do the work for you.
- It is very helpful to be the administrator of your development environment. We ran into many problems in the Alamode lab trying to get Bluetooth to work, among other things. In the end, the primary testing machine that interfaced with Bluetooth turned out to be a personal laptop that we could fully access and manipulate.
- Agile methods actually work. For developing the Wii Event Parser, it was very helpful to start by reading basic data, then take it up to reading data from a Wiimote, then managing several Wiimotes, then abstracting the parser and the Wiimote manager, until the module was finished. The "war room" method of having the whole team coding in the same room was very helpful for solving problems and working out design issues.

Future Work

From the beginning, this project was designed to be a foundation for future work. First and foremost, future developers could implement features that we could not complete within our project scope. Though we were able to implement the necessary modules for our project – the parsers, the binary tree game, the network game – there are many additional requirements that the client would like to see within the games themselves. For instance, the network game could have the ability to dynamically alter the map while the game is in progress. Also, there can be multiple play modes for the binary tree game. We feel that it will be very easy to write more programs, be it games or other teaching tools, using the Wiimote framework.

The framework has a very linear data flow, from Wiimote Manager to Wii Event Parser to Game Event Parser to Game Event Listener. However because of the modular design and communication with network sockets, the framework could very easily be expanded to a more complicated client-server pattern. For example, the parsers could be set up on a server, and multiple computers could run a Wiimote Manager that sends data to the server. Building network games would require minimal modification to the framework. This would also allow for more than seven Wiimotes (the maximum allowed on a single Bluetooth daemon) to play simultaneously. We feel that after enough polish, our framework could be made into a Debian package and distributed as an open source project.

Glossary

cwiid - A library written in C used to interface with Wiimotes. Handles all interaction with the Bluetooth stack.

pygame – A 2D graphics library and game-engine built in Python.

reStructuredText – A markup language that is easily readable in its unrendered format. For example, a bolded word is represnted as: *Bolded word**.**

Wiimotes – A Wiimote is a controller developed by Nintendo to interface with the Nintendo Wii Gaming Console via Bluetooth. The data sent by a Wiimote is the traditional button input of a game controller as well as accelerometer data in each of the 3 standard directions along the x, y, and z planes.

Appendix

Figure Title	Page Number
Figure 6: Binary Tree Game UML	*B*
Figure 7: Binary Tree GUI UML	*C*
Figure 8: The GUI Objects UML	*D*
Figure 9: Network Game UML	*E*
Figure 10: Wii Event Parser UML	*F*
Figure 11: Screenshot of Network Game in .Net	*G*

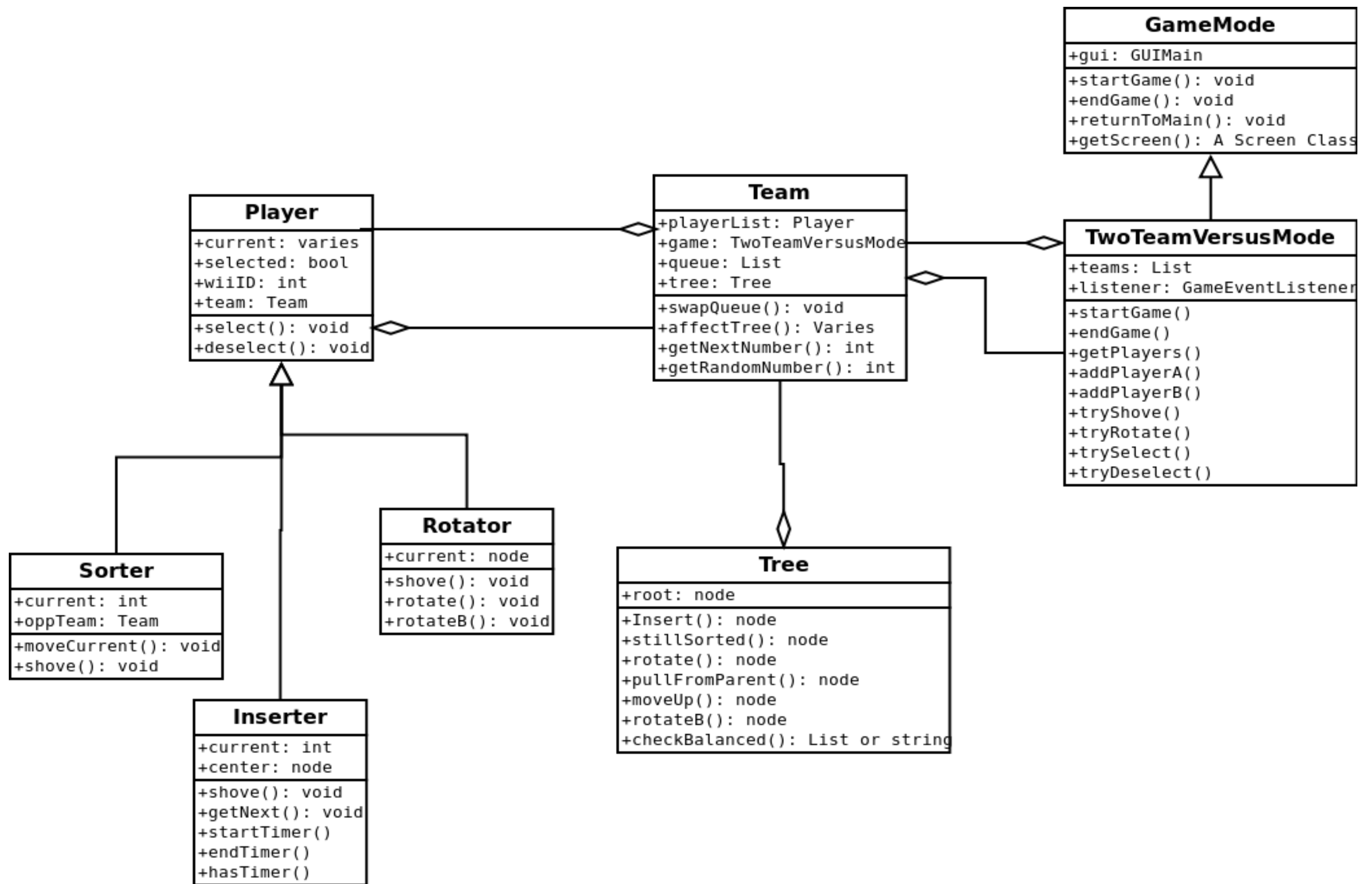

Figure 6: Binary Tree Game UML

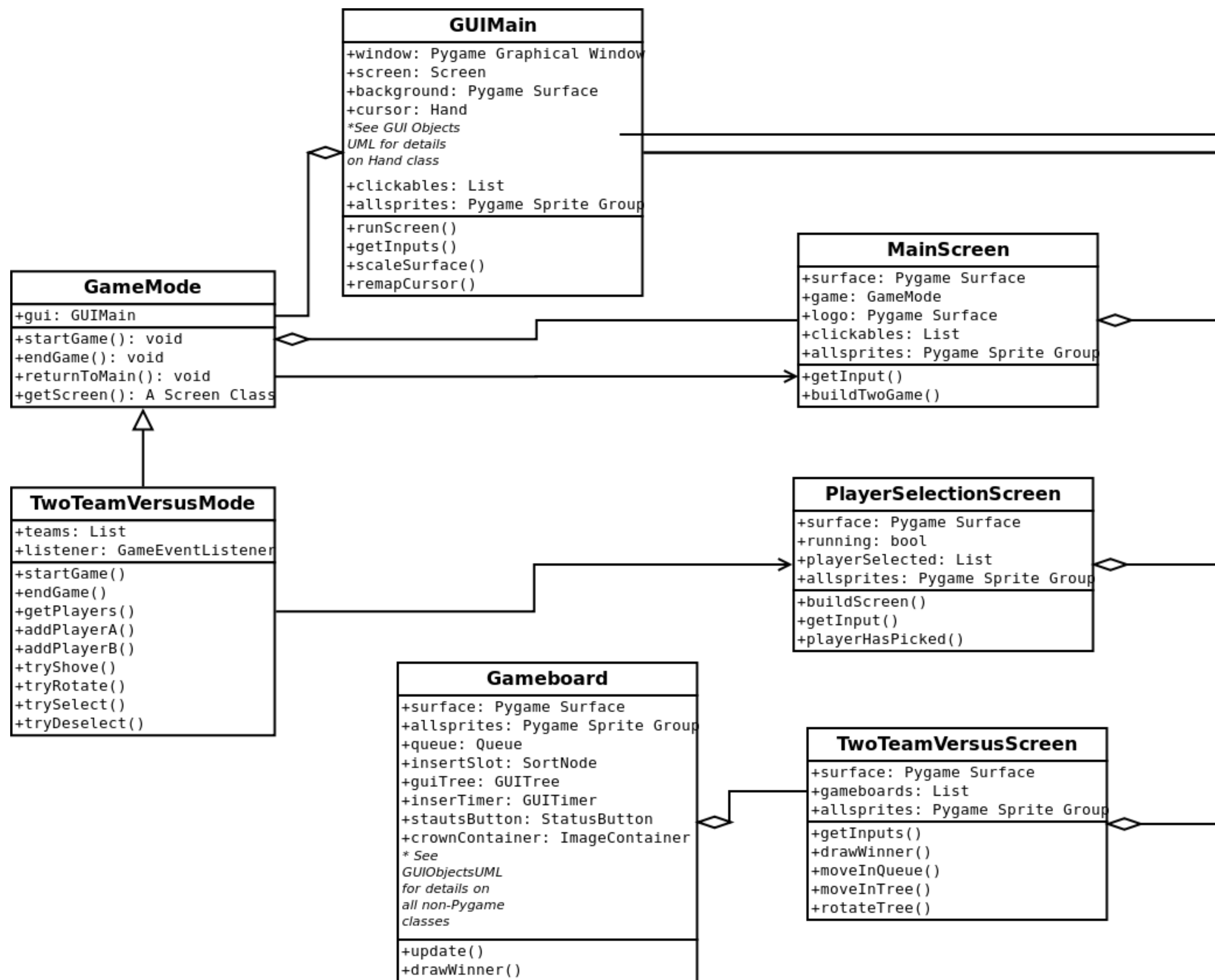

Figure 7: Binary Tree GUI UML

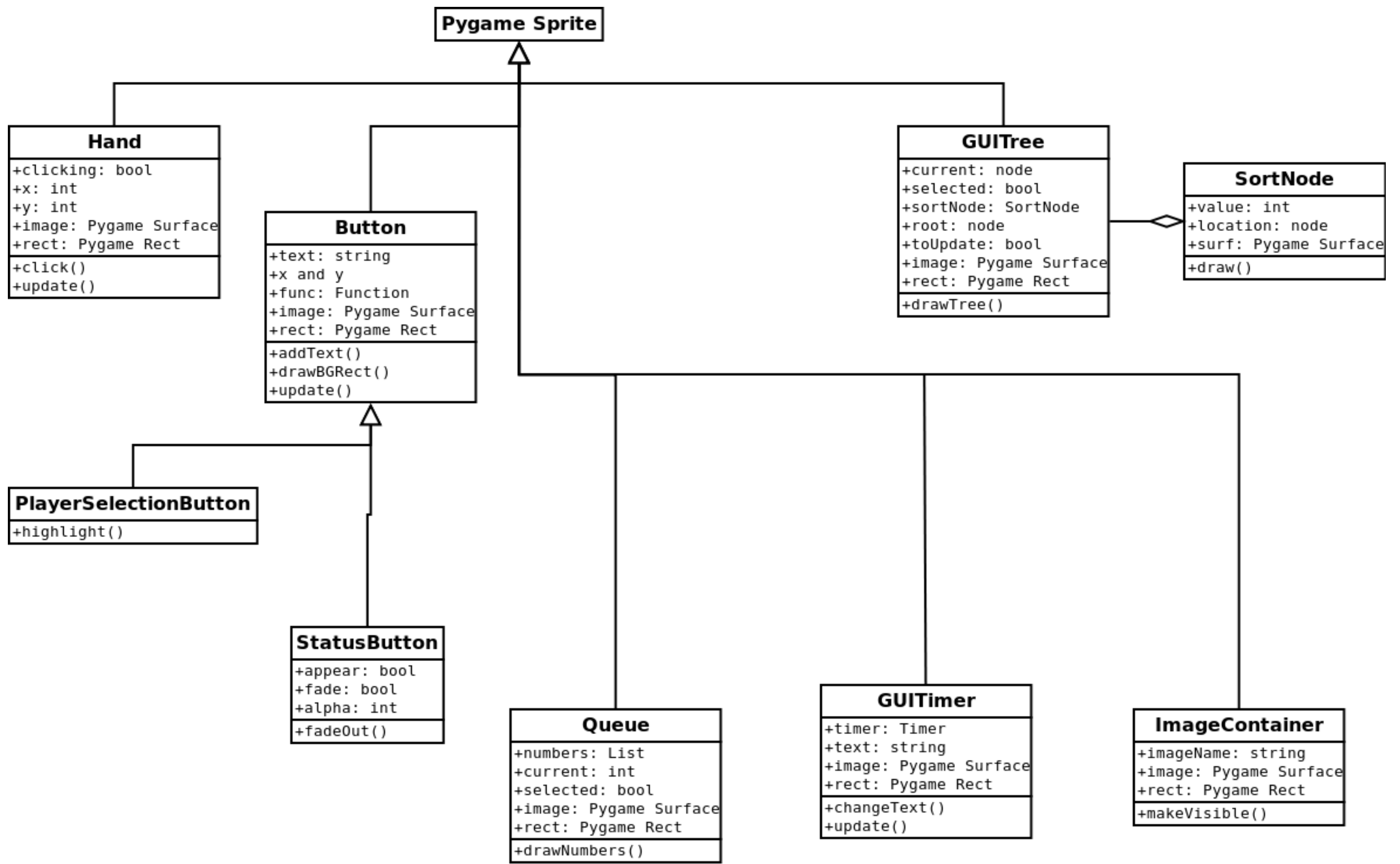

Figure 8: The GUI Objects UML

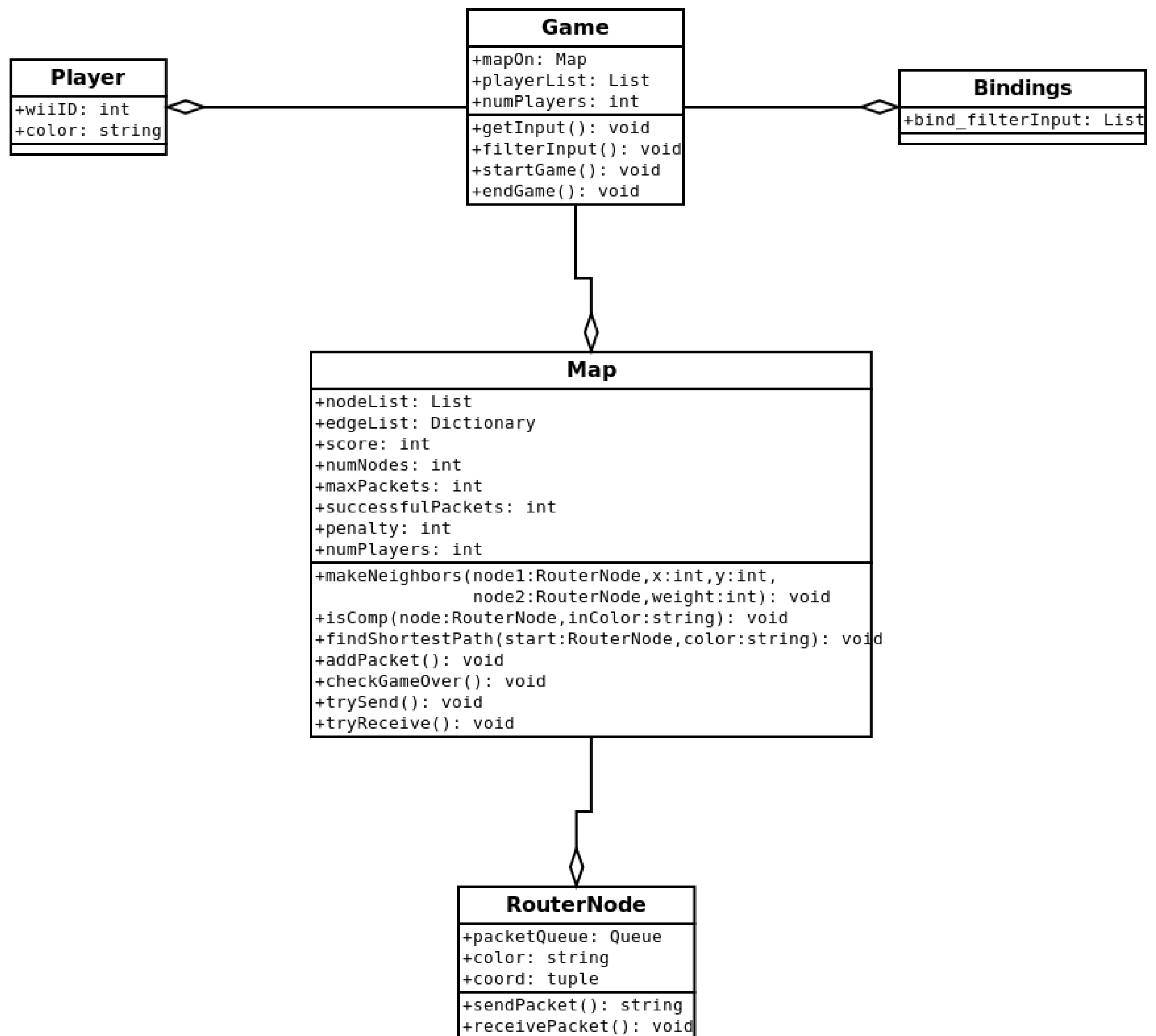

Figure 9: Network Game UML

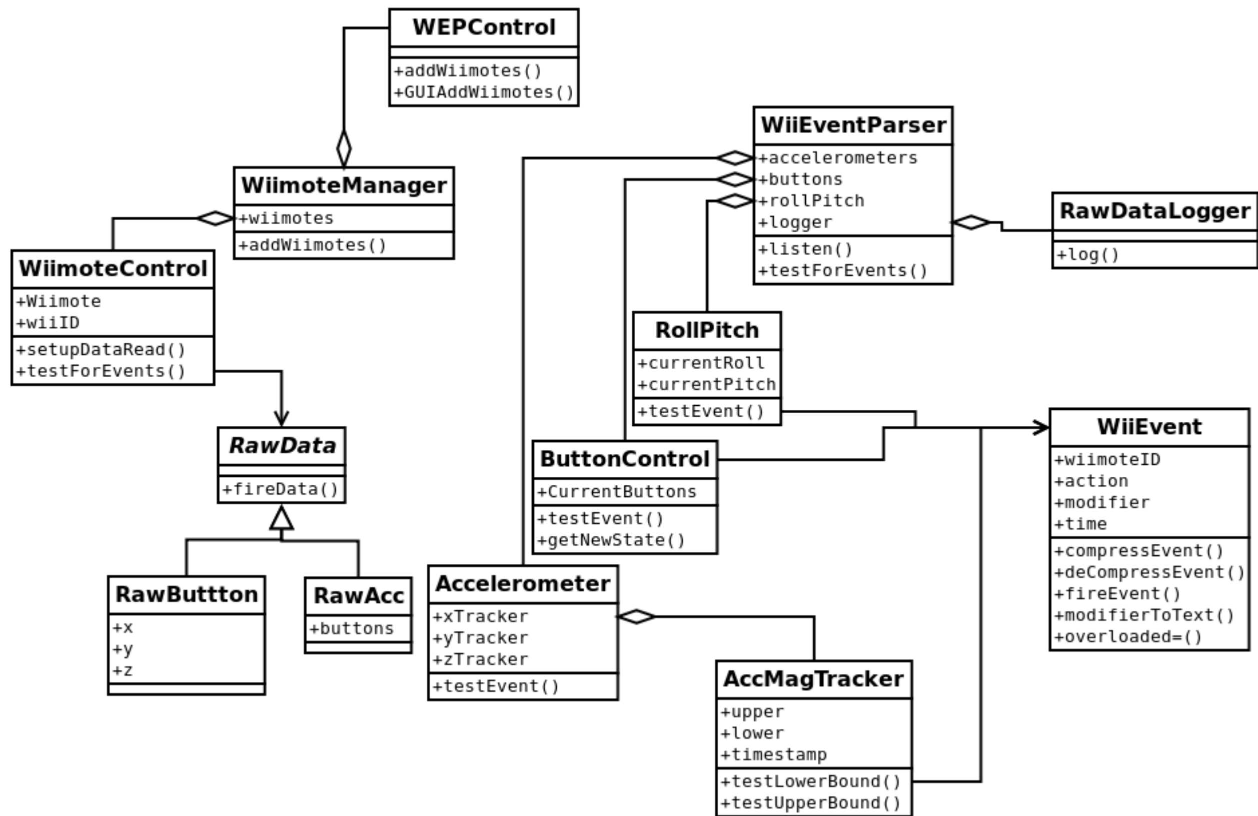

Figure 10: Wii Event Parser UML

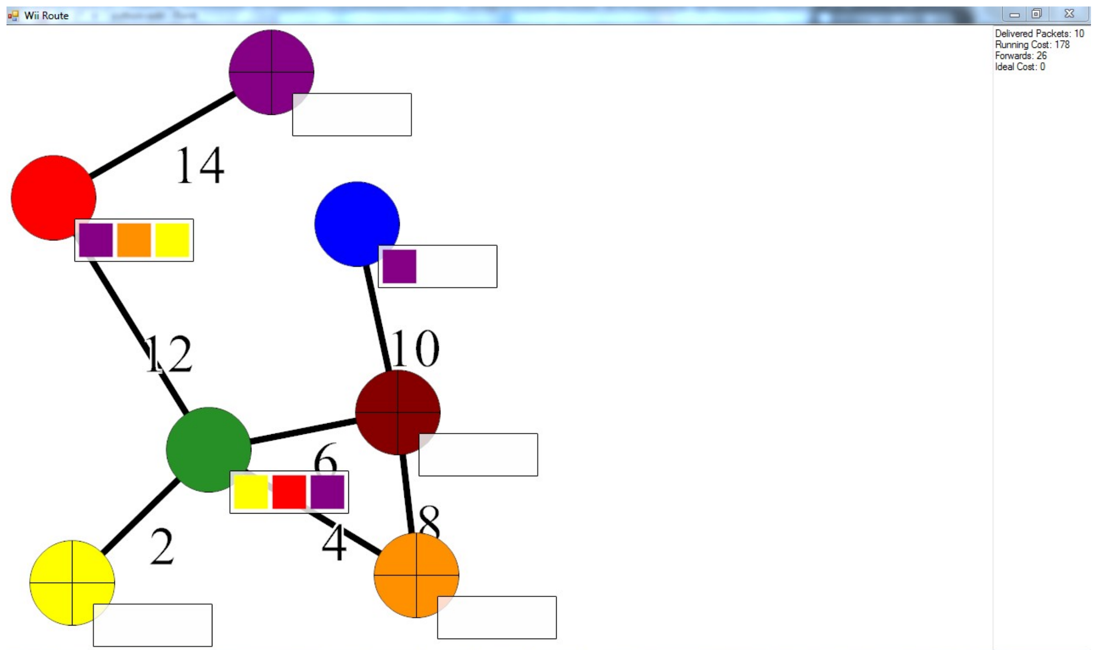

Figure 11: Screenshot of Network Game in .Net