**CMSC 331 Project 2**

**Ruby Specifications**

**Capture the Flag**

Gaurang Bhatt, Michael Norris, Eric Schuler

Description:

We created an implementation of capture the flag where two players compete to attempt to carry the flag to three of the four corners of the map to win. A player captures the flag by standing on it. After capturing the flag, it follows behind the player and they need to do the best they can to defend it from being captured by the other player.

*Overall Server Setup*

We used Ruby 1.9.1 with the gem WxRuby. Gaurang and Michael used an installation of Ruby which had WxRuby preinstalled. We have our source code in the src folder and some tests which are not used by the main program in the tests folder. In the res folder we have img for tiles and music for the game music.

We run the program by calling ruby src/main.rb

*Variables*

We used a mix of local variables and instance variables.

The instance variables were declared in the initialize methods (constructors)

For example, the player class constructor included the following:

**def** initialize(id, pt)

@playerID = id

@icon = tiles.getObject("player" + id.to\_s())

@currentLocation = pt

@hasFlag = ***false***

@mostRecentLocation = @currentLocation

@cornersVisited = Array.new(0, ***nil***)

@update = ***true*** *#if they need to be redrawn*

**end**

Instance variables in Ruby can be used by methods inside the class, but they cannot be directly accessed outside of the class. Accessors and mutators were provided to alter the variables to a certain extent.

**def** setUpdate(boolean)

@update = boolean

**end**

**def** getUpdate

**return** @update

**end**

Local variables were variables which we used temporarily, such as when setting up the main menu. Since these variables were never used again, they were used as local variables instead of instance variables.

*# get the initial height of the panel*

initialHeight = ***self***.get\_size().get\_height() / 2

*# create buttons*

btnNewGame = MainMenuButton.new(...)

The instance variables have a @ symbol at the start of their name, while the local variables do not.

*Conditional Statements*

If statements were used in our game extensively. Most of the time they helped to check if certain conditions were true before an action occurred. For example, when a player presses a button to move a player, the game checks to see if the player is on the edge of the map before moving it.

*# checks if the button pressed is the W key and if the player is not on the*

*# upper boundary of the map*

**if**(value == ?W.ord **and** p1Location.y > 0)

newLoc = [p1Location.x, p1Location.y - 1]

*# checks to see if the new location is occupied by another player*

**if** **not** overlap(newLoc, p2Array)

playerUpdate(@p1, newLoc)

**end**

*# checks if the button pressed is the S key and if the player is not on the*

*# lower boundary of the map*

**elsif**(value == ?S.ord **and** p1Location.y < MAPSIZE - 1)

newLoc = [p1Location.x, p1Location.y + 1]

*# checks to see if the new location is occupied by another player*

**if** **not** overlap(newLoc, p2Array)

playerUpdate(@p1, newLoc)

**end**

...

*# checks if the button pressed is the up arrow key and if the player is not*

*# on the upper boundary of the map*

**elsif**(value == K\_UP **and** p2Location.y > 0)

newLoc = [p2Location.x, p2Location.y - 1]

**if** **not** overlap(newLoc, p1Array)

playerUpdate(@p2, newLoc)

**end**

...

**end**

Ruby also has a case conditional statement, similar to the switch block in Java. This would have been equally effective. However, if/else statements have essentially the same format as in other popular languages, so they were better in our situation.

*Validation*

The primary validation needed in our game is when the user presses a key to move their player. This is accomplished by sending the key code from the key press event to the game object. The game object handles the key code in an if statement, seen above. The method moves the player in a different direction based on the key code given. If the key code given does nothing in the game, then nothing is done.

This is our key event listener from the MinimalApp class:

*self.evt\_key\_up() {*

*|event| getKeyPress(event)*

*if (@game.get\_game\_over)*

*sleep(5)*

*make\_new\_game*

*end*

*}*

This is the function that it calls which passes the input to the game object

def getKeyPress(event)

keyCode = event.get\_key\_code

@game.input(keyCode)

end

end

The game input function is very lengthy, but all it does is compare the key that was pressed and maps it to a movement of a player

#Take in a key value from the key event in the App and process it.

def input(value)

p1Location = @p1.get\_current\_location()

p2Location = @p2.get\_current\_location()

#for checking overlap

p1Array = [p1Location.x, p1Location.y]

p2Array = [p2Location.x, p2Location.y]

if(value == ?W.ord and p1Location.y > 0)

newLoc = [p1Location.x, p1Location.y - 1]

if not overlap(newLoc, p2Array)

playerUpdate(@p1, newLoc)

end

elsif(value == ?S.ord and p1Location.y < MAPSIZE - 1)

newLoc = [p1Location.x, p1Location.y + 1]

if not overlap(newLoc, p2Array)

playerUpdate(@p1, newLoc)

end

elsif(value == ?A.ord and p1Location.x > 0)

newLoc = [p1Location.x - 1, p1Location.y]

if not overlap(newLoc, p2Array)

playerUpdate(@p1, newLoc)

end

elsif(value == ?D.ord and p1Location.x < MAPSIZE - 1)

newLoc = [p1Location.x + 1, p1Location.y]

if not overlap(newLoc, p2Array)

playerUpdate(@p1, newLoc)

end

elsif(value == K\_UP and p2Location.y > 0)

newLoc = [p2Location.x, p2Location.y - 1]

if not overlap(newLoc, p1Array)

playerUpdate(@p2, newLoc)

end

elsif(value == K\_DOWN and p2Location.y < MAPSIZE - 1)

newLoc = [p2Location.x, p2Location.y + 1]

if not overlap(newLoc, p1Array)

playerUpdate(@p2, newLoc)

end

elsif(value == K\_LEFT and p2Location.x > 0)

newLoc = [p2Location.x - 1, p2Location.y]

if not overlap(newLoc, p1Array)

playerUpdate(@p2, newLoc)

end

elsif(value == K\_RIGHT and p2Location.x < MAPSIZE - 1)

newLoc = [p2Location.x + 1, p2Location.y]

if not overlap(newLoc, p1Array)

playerUpdate(@p2, newLoc)

end

end

end

*Loops*

When we draw the background initially, we run a nested for loop to draw each tile. This loop (which is located in the update\_background function) is also called when the window is resized by way of the evt\_paint listener. In both of those instances, we want all of the tiles to be drawn, so we use a for loop to draw the tile to the screen. The values i and j are scaled by 32 and the pair (i\*32, j\*32) is the top left corner for where the tile is drawn.

for i in 0..(MAPSIZE-1)

for j in 0..(MAPSIZE-1)

self.drawToTile(dc, @map.getTile(i, j), i, j, false)

end

end

*Functions*

Functions in ruby were very fun for us, especially when we can call a function without parentheses. We attempted to make all of our functions have very specific names so we wouldn’t have to expend any effort trying to remember which function to use. We were unsure in the beginning about the naming structure for ruby functions as opposed to ruby variables so there is a little discrepancy between our naming conventions. Most of our functions have underscores, but some of our early functions are camelCase and we didn’t have the time to go back and adjust their names.

The corner\_handler function sees if the player who currently has the flag is standing on a corner. If the player is standing on a corner, it calls the player’s visit\_corner function which deals with adding the corner to the player’s array of corners if the player has not visited it. After, it changes the map tile. If we wanted to optimize our code, we would only change the map’s tile if it was not already changed. Instead, if a player keeps moving between a corner, the map keeps updating that tile.

def corner\_handler

point = @who\_has\_flag.get\_current\_location

pointArr = [point.x, point.y]

#see if player is standing on corner

corner\_num = corner\_number(pointArr)

#puts corner\_num

if corner\_num != -1

#if they are on a corner, update it if they haven't been there.

@who\_has\_flag.visit\_corner(corner\_num)

end

if corner\_num == 0

@map.top\_left\_pressed

elsif corner\_num == 1

@map.top\_right\_pressed

elsif corner\_num == 2

@map.bottom\_left\_pressed

elsif corner\_num == 3

@map.bottom\_right\_pressed

end

if @who\_has\_flag.has\_visited\_three\_corners == true

@game\_over = true

@frame\_reference.draw\_win\_message(@who\_has\_flag.getPlayerID)

end

end

This is the player’s visit\_corner function which is called by corner\_handler. The function iterates over the players cornersVisited array and returns true and adds the corner to the player’s array if they have not been to the corner, else it returns false. This function demonstrates the use of Ruby’s each method over an array.

# simulates the player visiting a specific corner

# returns true if the cornerNum is added

# returns false if the cornerNum is already in the array

def visit\_corner(cornerNum)

# iterate through the array to search for cornerNum

@cornersVisited.each {

|c|

if (c == cornerNum)

return false

end

}

# add the cornerNum to the array and return true

@cornersVisited = @cornersVisited + [ cornerNum ]

return true

end

*Arrays/Lists*

The Map class contains a two-dimensional array of Tile objects.

*#each row is a row of tiles*

tiles = Tiles.new

@grass = tiles.getTile("grass")

*#MAPSIZE = 19 #20 tiles from 0 to 19*

@backgroundTiles = []

**for** i **in** 1..MAPSIZE

@backgroundTiles.push([])

**for** j **in** 1..MAPSIZE

@backgroundTiles[i-1].push(@grass)

**end**

**end**

This is part of the constructor for the map class. The array is built using nested for loops. The Map class contains an accessor and mutator for a specific tile.

**def** getTile(i, j)

**return** @backgroundTiles[i][j]

**end**

**def** set\_tile(x, y, img)

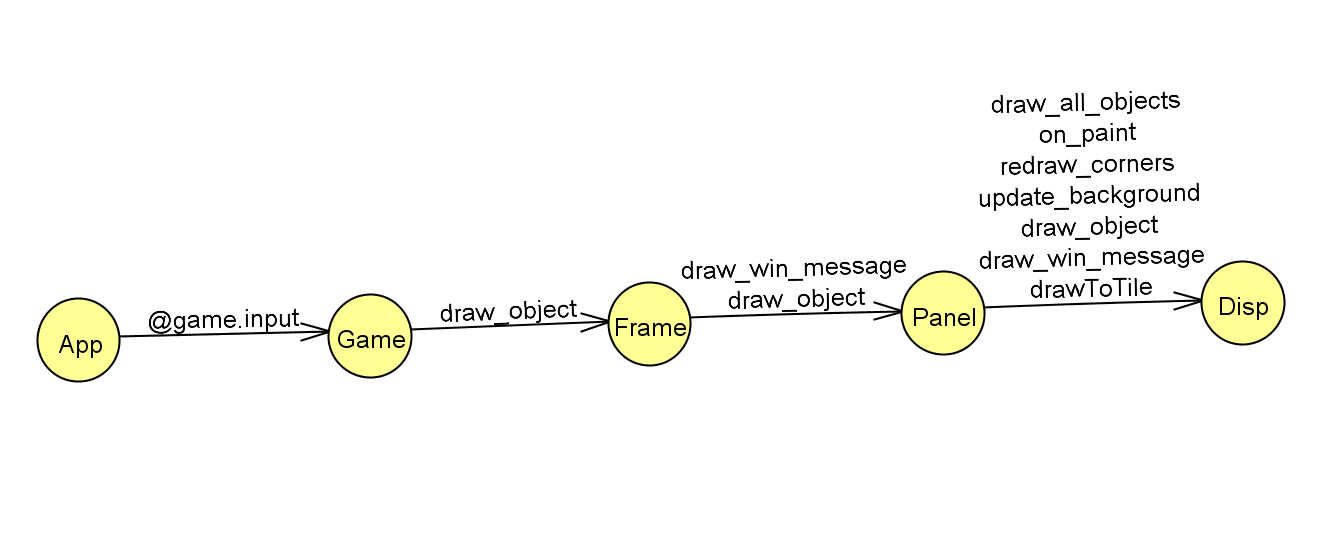
@backgroundTiles[x][y] = img

**end**

A two dimensional array effectively represents the x and y coordinates on a map. Also, each tile in the array is easily accessible through its index, which correspond to the coordinates.

*Graphics*

There is no game loop; the graphics are drawn based on events. There are two events for drawing, the first is the on\_paint event which is called from the GamePanel’s evt\_paint listener, and the second event for drawing is whenever the player pressed a key. When they press a key, the game may move a player, and when a player is moved then the draw functions will be called. Since one of these events originates in the app class, makes its way through the game class, then goes to the Panel, but the other one originates in the panel, we gave the panel a reference to the game so that it can quickly draw the game’s objects when it needs to, but we also gave the game a way to call the Panel’s draw functions as well. The following image demonstrates both the processes required for a graphic to be drawn based on a key press and also shows all of the drawing functions which have been created:



When we call the Game Panel’s draw object function we have two options. We can either draw an object with a tail or an object without a tail. The tail represents the previous location of the object. If an object is moving, we want its previous tile to be reverted to whatever it looked like before there was a player on it. We update its previous tile by asking the map class for the bitmap that should be at a specific location, then draw that bitmap to the screen. (The map class holds an active record of whether a tile should be grass or it should be a special pressed image, in the case of the corners). But we do not always want to draw the object’s tail. If the player has a flag, we do not simply want to draw the player, redraw its previous tile, then draw the flag and redraw its previous tile. The reason for this is that if the player’s most recent location was where the tile is, and the tile’s most recent location is where the player is, then the first object drawn will not be shown! Having the tail argument eliminates this bug.

def draw\_object(object, tail)

loc = object.get\_current\_location

rcnt = object.get\_most\_recent\_location

paint do | dc |

if (tail)

self.drawToTile(dc, @map.getTile(rcnt.x, rcnt.y), rcnt.x, rcnt.y, false)

self.drawToTile(dc, @map.getTile(loc.x, loc.y), loc.x, loc.y, false)

end

self.drawToTile(dc, object.getIcon, loc.x, loc.y, true)

end

end

It should be obvious that we are actively preventing continually drawing all of the map at once. The reason for this is that our game can move very fast and if we do not optimize the code and we just iterate over a paint\_everything\_at\_once method, some of the screen will not always be rendered fast enough between movements and the game looks very choppy. There are two occasions, though, when we allow the whole window to be drawn at once. These two occasions are when the on\_paint function is called, which is a listener for the GamePanel’s evt\_paint event. First, when the Panel is created the on\_paint function is called and it renders everything at once. First it draws the background, then the objects (players and flag) on top of it. The second occasion is when the Panel is resized – this also triggers an evt\_paint event. It was decided that the resizing event was significant enough that the whole window could be redrawn without being too much of a nuisance.

All graphics are drawn using the Bitmap object in WxRuby. Each player has a bitmap associated with them, so when they are passed to the GamePanel it draws their bitmap onto their respective tile based upon the player’s current location.

*Data Structures*

The MapOne class is a data structure that was created to easily facilitate the drawing of the map. The map has an array which is two dimensional and each element is a reference to a Wx::Bitmap. When we call the update\_background function in the Game Panel, it iterates over all of the tiles in the map and draws them at their specific point.

class MapOne

def initialize

#each row is a row of tiles

tiles = Tiles.new

@top\_left = tiles.getTile("topLeft")

@top\_right = tiles.getTile("topRight")

@bottom\_left = tiles.getTile("bottomLeft")

@bottom\_right = tiles.getTile("bottomRight")

@grass = tiles.getTile("grass")

#MAPSIZE = 19 #20 tiles from 0 to 19

@backgroundTiles = []

for i in 1..MAPSIZE

@backgroundTiles.push([])

for j in 1..MAPSIZE

@backgroundTiles[i-1].push(@grass)

end

end

end

def set\_tile(x, y, img)

@backgroundTiles[x][y] = img

end

def top\_left\_pressed

set\_tile(0, 0, @top\_left)

end

def top\_right\_pressed

set\_tile(MAPSIZE - 1, 0, @top\_right)

end

def bottom\_left\_pressed

set\_tile(0, MAPSIZE - 1, @bottom\_left)

end

def bottom\_right\_pressed

set\_tile(MAPSIZE - 1, MAPSIZE - 1, @bottom\_right)

end

#Resets the corners to grass.

def reset\_corners

set\_tile(0, 0, @grass)

set\_tile(MAPSIZE - 1, 0, @grass)

set\_tile(0, MAPSIZE - 1, @grass)

set\_tile(MAPSIZE - 1, MAPSIZE - 1, @grass)

end

#Returns the tile at i, j

def getTile(i, j)

return @backgroundTiles[i][j]

end

end

This data structure stores references to the Wx::bitmaps, provides easy access through its functions, and enables smooth updating of its state and the game’s representation of corner presses.

*OOP*

Our game has a few different classes that represent different aspects of the class. The Player class contains atrributes and behaviors for a player of the game. A player object knows its current location and immediate past location, which is used for displaying a flag behind the user. The player knows if it has the flag or not, and it knows which corners of the map it has visited (part of the goal of winning the game). Finally, the player object contains an Image object for its icon on the map.

A main function of the Player class is move(), which updates the current location and sets the immediate past location to the former current location. There is also a tag() function, which removes the flag from the user and erases the data for corners visited.

Here is the constructor for our Player to show our OOP

We also included a flag class to represent the game flag. Like the player, a Flag object keeps track of its current and immediate past location, as well as its icon. In addition, the flag object knows which player controls it, which would initially be nil. The main function for the Flag class is move(), which updates the location attributes.

#A Player holds a player's location, moves their avatar, keeps track

#of whether they hold the flag or not, and keeps track of which tiles

#they have been to.

class Player

# Creates a new Player object

# param id -- int used as playerID

# param img -- Image used as player icon

# param pt -- Point used as current location of player on map

def initialize(id, pt)

tiles = Tiles.new

@playerID = id

@icon = tiles.getObject("player" + id.to\_s())

@currentLocation = pt

@hasFlag = false

@mostRecentLocation = @currentLocation

@cornersVisited = Array.new(0, nil)

@update = true #if they need to be redrawn

end

The Game class contains two player objects, one flag object, and a map object. Each game object updates the locations of the players and the flag. It also handles the key codes obtained from the key press event handler. A game can check if the game has been won or not.

The main driver class contains some derived classes. For example, there is a GamePanel class which inherits from Panel. Also, the GameFrame and MainMenuFrame classes inherit from the Frame class. The MainMenuButton and MainMenuFont classes, which inherit from the Button and Font class respectively, are used for the buttons on the main menu. Finally, there is a class which inherits from the App class. We have given all of these classes their own attributes.

We could have implemented a GameObject class which would have had derived classes of Player and Flag. Then, if we wanted to draw all of the objects, we iterate over the array and call the draw function. Instead, the Player and Flag classes are similar and we just made sure they had similar methods so that they could be moved and drawn easily.