Technical Documentation - 4InARow

Niklas Ertle
309681

1. load.py
imports csv to read and write in a file. CSV is a file type to store values (comma separated values).

class Data

- __init__(filepath) : creates a list and calls update()
- update() : loads the file contents into the list
- find(index) : returns the first value it finds for the key ‘index’
    - example: input: "width" returns: "1200"
- findColorList(themeName) : collects the theme and translates hex colors to tuples
    - returns a dictionary
- listThemes() : lists all the themeNames in the file
- formatSave(themeName, colorDictionary) : parses Dictionary into list and Translates colors from int tuples to Hex String
    - example: input: themeName = "classic"
      colorDictionary = { "background" : (50,50,50),
      "player1_1" : (50,50,150), "player2_1" : (150,50,50),
      "player1_2" : (150,150,150), "player2_2" : (150,150,150),
      "normal_text" : (10,10,10), "buttons" : (100,100,100),
      "chat" : (100,100,100) }
      output: [[--- classic ---], [background,323232], [player1_1,323296],
      [player2_1,963232], [player1_2,969696], [player2_2,969696],
      [normal_text,0a0a0a], [buttons,646464], [chat,646464],
      [--- classic ---]]
- save(themeName, colorDictionary) : uses formatSave to format before saving the colorDictionary to the file
    - if the theme already exists in the file, the section will be updated
    - else the theme section will be added at the end of the file

2. game.py

imports random, time and threading

class Game

- __init__(id) : initializes all game variables
- move(player, number) : ‘player’ makes move at position ‘number’
- checkState() : check if somebody won or there are no more moves left
- checkHorizontals(player) : check if player has 4 in a horizontal row
- checkVerticals(player) : check if player has 4 in a vertical row
- checkDiagonals(player) : check if 'player' has 4 in a diagonal row
- checkNoMoreMoves(): returns True if there are no free fields left
- winner(player) : increments ‘wins’ for ‘player’ and ends round
- restart() : starts new round
- newMsg(msg) : adds 'msg' to the chat
- newCmd(msg) : analyzes 'msg' and does something to the game depending on its contents

3. server.py

imports socket, json and _thread
also uses the **Game** class from game.py and **Data** class from load.py

- initializes all variables to connect multiple clients and host multiple games
- function threaded_client(conn, player, gameId) : handles clients separately
    - always gets called in a new thread
    - sends the client feedback about which player it is
    - while True loop : runs as long as the connection is open
        - waits to receive 'data' from client
        - if game is no longer running or ‘data’ is null, then break loop
        - if 'data' is "get", then send the game state as json
        - if 'data' is a number in the valid range, then call game.move(player, data) and send back the game state as json
        - if 'data' starts with "message", then call game.newMsg("'player',rest of 'data'") and send back the game state as json
        - if 'data' starts with "command", then call game.newCmd("'player',rest of 'data'") and send back the game state as json
            - if the second element in 'data' is "close", then break the loop instead
    - closes connection when exiting the loop

- while True loop : runs as long as the server is open
    - waits for new connection
    - for every other connection a new game is created
    - start new thread threaded_client(conn, player, gameId)

4. network.py
imports socket

class Network
- __init__(server, port) : initializes all variables for a connection and then connects to the server
- getPlayer() : returns the player number
- connectToServer() : sends clients address and gets the player number on the server
- send(data) : sends 'data' from client to server and returns the games state from server to client

5. button.py
imports pygame

class Button
- __init__(text, x, y, color, textColor, width) : sets up a button
- draw(win, font) : draws button on screen
- click(pos) : check if pos overlaps with button

6. client.pyw
imports pygame, pygame_gui, json, time, threading
also use class Network from network.py, class Button from button.py and class Data from load.py

dummy class Game
- __init__(**entries) : builds a holding replica for the variables in game.py
    - **entries = all variables given to the class

class Client
- ref(x) : returns the relative x value depending on the base layout to the current layout
    - returns x * currentWidth / 1600
    - makes all elements (x, y, width, height) scalable relative to the window size
- __init__()
    - loads layout information from the properties.txt
    - sets up the layout and colors of the user interface
    - builds connection to the server specified in the properties.txt
    - updates the servers information about the player

- drawBackground(window, colors, p, width, height, buttonPos_x, buttonPos_y, btns)
    - draws background and separation lines
    - draws background of the chat with players secondary color
- drawChat(window, colors, font, messages, messagePos_x, messagePos_y, messageIndex)
    - draws a chat background and the chat text in the senders primary color
- drawElements(window, colors, game, font, p, btns, userNameThis, userNameOther)
    - draws game information on the UI
- drawGame(window, colors, game, buttonPos_x, buttonPos_y, winAnimation)
    - draw circles with players primary and secondary color

- redrawWindow(game) : call all draw methods and update display
- clientCommand(command)
    - depending on the contents of the command, the client will do different things
- toggleFrame() : is called in new thread
    - animates win situations
- rescale() : recalculates all element positions and sizes
- input(game) : evaluate all pygame events
- quit() : breaks main loop and sends server a close signal
- sendMessage() : calls clientCommand(), sends a command or message to the server depending on the input
- scrollUp() : goes up in the message list to reload them into the message input
- scrollDown() : goes down in the message list to reload them into the message input
- toggleFullscreen() : reinitializes the window to switch between Fullscreen and window mode
- clickButton(event, game) : player input to make a move
- adjustChat(event) : calculates the chat beginning and ending index
- updateUser(game) : update some clients variables with the server variables
- updateChat(game) : updates the clients message list to the servers chat list
- main() : runs client
    - while loop : runs as long as the client is running
        - updates the gamestate with the servers game variables
        - keeps framerate at 60 fps

script creates the object Client c and starts the main()

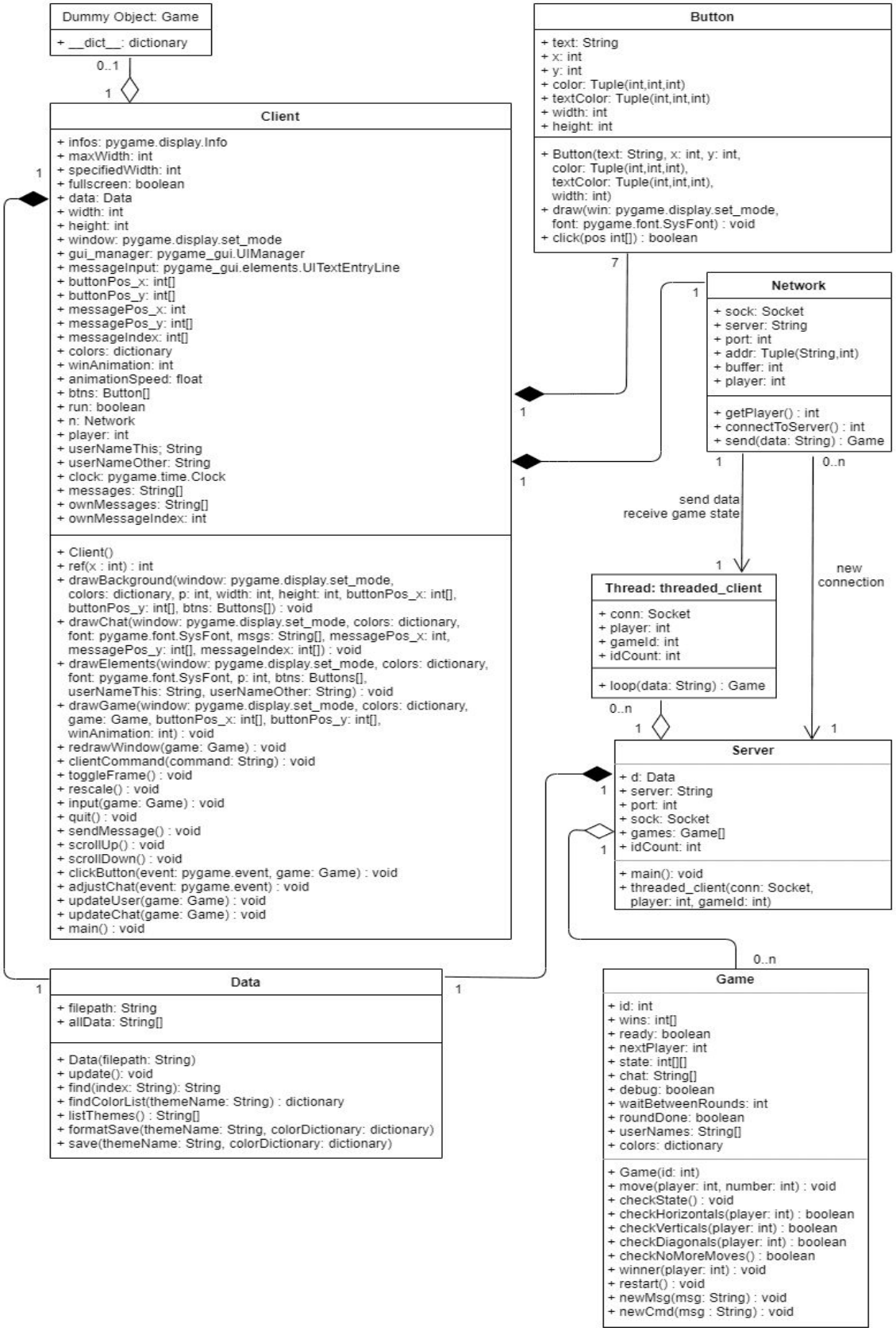
7. data transfer in this project

- open a Server
    - waits for connections
- open n Clients
    - creates a Network object, which handles all sending and receiving processes on the Client side
- the Network object connects to the Server and receives a player number
    - the server creates or starts a new game
- the Client sends frequently a package, asking for an update
    - the server sends back a copy of the games variables
- depending on the users input, the client can send 3 types of packages
    - a simple integer between 1 and 7
        - the server then makes a move in the game objects state at the numbers position for the player
    - a command via the message Input, which begins with "/"
        - if the command is to close the game, the server ends the game directly
        - otherwise the server forwards the command to the game to process it there
    - a message via the message Input, which the server forwards to the game to add it in it's chat list
- after each package, the server sends back the updated version of the game variables

8. UML Diagrams

the server.py script is not build in a object oriented way, but to illustrate the servers behavior the contents are represented as class

UML class diagram



UML sequence diagrams