Yekwon Park
James Kaufman
Louis Piper Carson

==

Project Title: **Pictionary Royale**

==

Minimum deliverable is standard pictionary. Several players play at the same time. In the first round, one player is randomly selected to be the drawer. They are given a word to draw a picture of. Based on that picture, the rest of the players must try to guess what the word is. If they can guess the word before time runs out, they are awarded a point. The drawer gets points for each person to correctly guess the word. Enough rounds are played so that every player gets a chance to draw, and whoever has the most points in the end wins.

Maximum deliverable has most of the same features: drawers, guessers, pixel drawings, and chat messages. However, now multiple players will be selected as the drawers in each round. They will all draw the same word at the same time, while the rest of the players guess based on all of the pictures. At the end of the round, each of the guessers vote on which drawing was the best, and points are given accordingly.

==

Specific Problems to be solved

==

How can a group of friends play pictionary together?
This problem includes:

- How to send user mouse input to all clients?
- How to keep score and verify answers?

==

Addressing these problems

==

Note: client meaning the software run by the user, server meaning external software that client chooses to connect to

1. Peer-to-peer method (First proposed Design Solution)
    a. Each client is responsible for its current user and nobody else
        i. knows if it is the current user's turn to draw or to guess
        ii. broadcasts its state and any relevant data to anyone who will listen
    b. Each client is given a unique ID
        i. ID used to determine turn order and assign points

For this method to solve the first problem above, a separate Erlang process would probably have to monitor each one of the drawers, spawning processes to notify each of the guessers after each individual change. Another process overseeing the entire game could also solve the second main problem, keeping track of all the correct answers and their corresponding points.

2. Client-server method (Second proposed Design Solution)
    a. Each client is capable of starting or joining a pictionary server
        i. Server is responsible for delegating work/data flow between all clients
        ii. Client is only responsible for giving its data to the server and displaying received data
    b. Each client is capable of receiving input from its user and sending that input to the server
        i. drawing mode
            1. input is received as mouseclicks
            2. output is the drawing and all other users' guesses
        ii. guessing mode - input is received as guesses
            1. input is the guess
            2. output is only the drawing

This method solves the given problems above very simply. Every time a change is made by the drawer, this requires a call to be made to the server, which can then simply update its state with the new drawing changes, and have all the clients update their information accordingly. Similarly, the server would itself be controlling the game state, so the server will always get every single one of the answers, and easily be able to keep track of the scores during the game.

==

Decision and rationale

==

We chose to use the client-server design because it fits more naturally to the problem and is easier to implement. A client-server approach allows for both the drawer's input and the guessers' inputs to be handled together by a single entity - this naturally reflects the traditional elementary school pictionary approach, where a single entity - a 'moderator' of sorts - acts to pick the next drawer, grant points to individuals and resolve timing disputes ("i guessed first" "no, i guessed first" issues)

We can also use producer-consumer design in this way - input is sent to the server from the outside, the server produces messages for each designated client and the consumers send these messages. The clients then process the messages accordingly.

This design was also chosen over peer-to-peer as it provided an easier way to split the work between multiple people and then combine the solutions later: one person could work very heavily on the client individually while another could work heavily on just the server, and then the two solutions could start to be integrated together. We decided it was easier to split up the work and have good individual testing capabilities with a client/server model than with peer-to-peer.

==

Development Plan

==

1. 1st Deliverable
    a. Simple pictionary.
        i. N total players, 1 drawer, N - 1 guessers.
        ii. Point tallying system based on correct/incorrect guess
        iii. Round time set to ~30 seconds.
    b. Client
        i. Display + Chat area
            1. Display handles input if server tells it that it is the drawer
            2. Display updates if server tells it that it is the guesser
            3. Chat area accepts input if server tells it that it is the guesser
            4. Char area displays guesses if and only if the server tells it that it is the drawer (prevent cheating but also provide artist feedback)
    c. Server
        i. N connected clients
            1. Ordered
            2. Score kept for each
        ii. 2 states - waiting, drawing
            1. waiting - before game starts and between rounds
            2. drawing - time when drawer can draw, guessers can guess
        iii. Award points for correct guess (once per client per round)
            1. Award artist some points per correct guess (~1/N of a correct guess)
2. 2nd Deliverable
    a. Pictionary Duel
        i. N total players, M drawers, N - M guessers
        ii. Point tallying system based on correct/incorrect guess AND time passed AND how many previous correct guesses have been received
        iii. Round time is configurable by server host
        iv. Art critique/voting at end of round
        v. All drawers have same word to draw
    b. Client
        i. Mostly the same, except there are now M drawing areas and 1 chat area

    c. Server
        i. Mostly the same, except there are now 3 states - waiting, drawing, voting
3. Extras
    a. Gamemodes
        i. Server host can input their own words
        ii. Blackout mode - can only see a small area around the mouse at any given time (for both drawers and guessers)
        iii. Invisible ink mode - Brush Strokes disappear after a few seconds (only for guessers)
        iv. Mashup mode - each drawer is given a separate word to draw

Division of Labor

1. 1 person for client
    a. UI
        i. Basically a grid of N panels, one for each drawer and a final one for chatroom / guessing / voting area
        ii. Server creation / selection
    b. Server interface
        i. Erlang backend that sends messages to server
        ii. Receives messages and calls corresponding python functions
2. 3 people for server
    a. gen_server implementation
        i. listen for new client connections and create consumers
    b. handle producer/consumer model
        i. 2 P/C queues
            1. data output queue - handles sending state to clients
                a. producer - server output process
                b. consumers - client process dispatcher
            2. data input queue - handles receiving messages from clients
                a. producers - server-client input processes
                b. consumer - server runtime
    c. handle multiple clients and input/output synchronization
        i. Keep track of connected clients
        ii. Heartbeat signals to kill client-handling processes for disconnected clients
        iii. Keep score, award points, tell clients current leaderboard states
        iv. Choose drawer, tell drawer what to draw
    d. implement all states
        i. waiting - stalls for N seconds
        ii. drawing - receives drawing messages ONLY from drawer and guessing messages ONLY from guessers

Most if not all concurrency related details will be isolated to the server itself. Client will more or less act as a sequential process.

- For very large numbers of drawers, might need to handle drawing input concurrently. consider this to be an ‘extra’ deliverable

As much of the labor will be focused as possible on the server, as this is where a majority of the work will happen. All team members will eventually work on the client at least once, but only one person will be ‘dedicated’ to just the client at any given time.

Timeline:

1. Client
    a. November 12th - N panel GUI and Chat demo [complete]
    b. November 19th - User input and handling; panels support drawing; chat supports message sending [in progress - chat implemented, input framework created]
    c. November 26th - Implement server communication process in Erlang
        i. Drawing messages received are drawn to the panel associated with a given client ID
            1. Drawing messages contain a list of pixels that have changed and what they have changed to
                a. Perhaps can be simplified down to a couple straight lines
        ii. Chat messages received are drawn to the chat panel if in waiting mode, or if current client is the drawer
2. Server
    a. November 12th - gen-server that can accept new client connections and dispatch/receive messages from those clients; dummy states implemented that match the spec. [complete]
    b. November 19th - game algorithms implemented. can choose drawer, keep state for a given amount of time, assign points etc [behind schedule. can add dummy players and choose drawers]
    c. November 26th - data input/output queues and P/C model finished; client communication tested and finalized

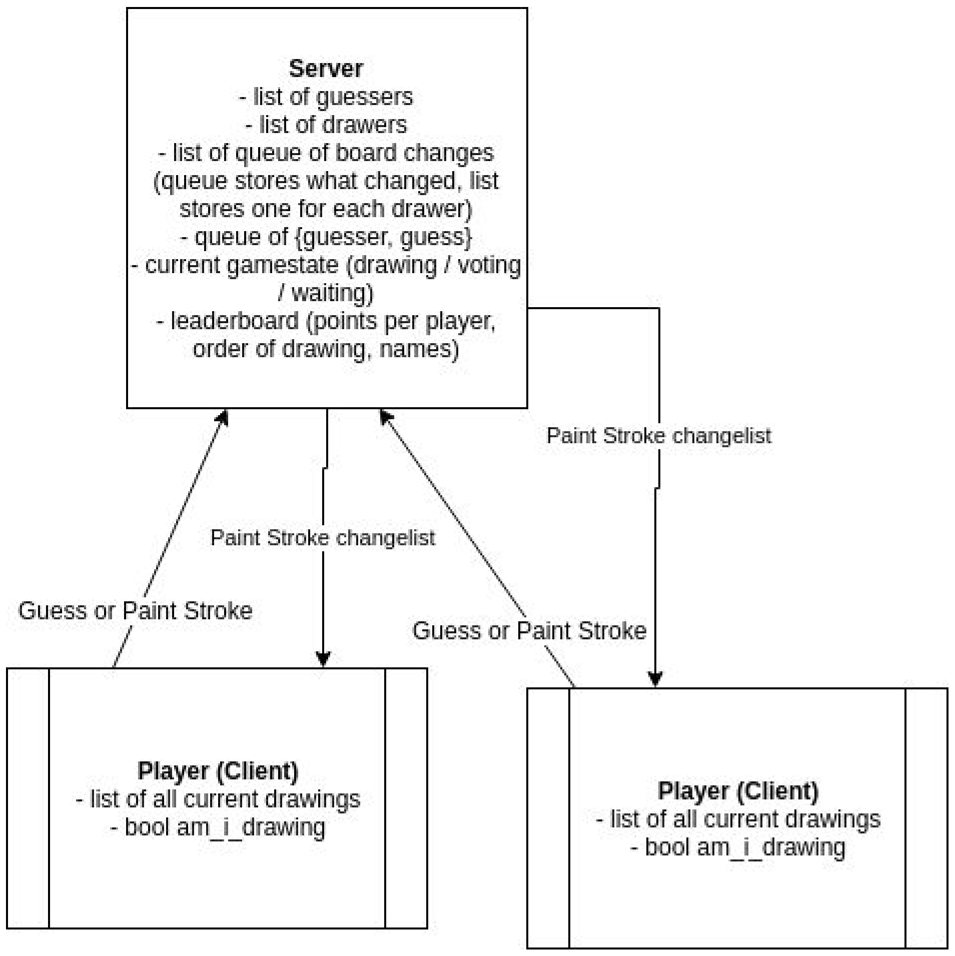
Presentation preparation

Finished presentation outline by November 28th

Meet to rehearse November 30th

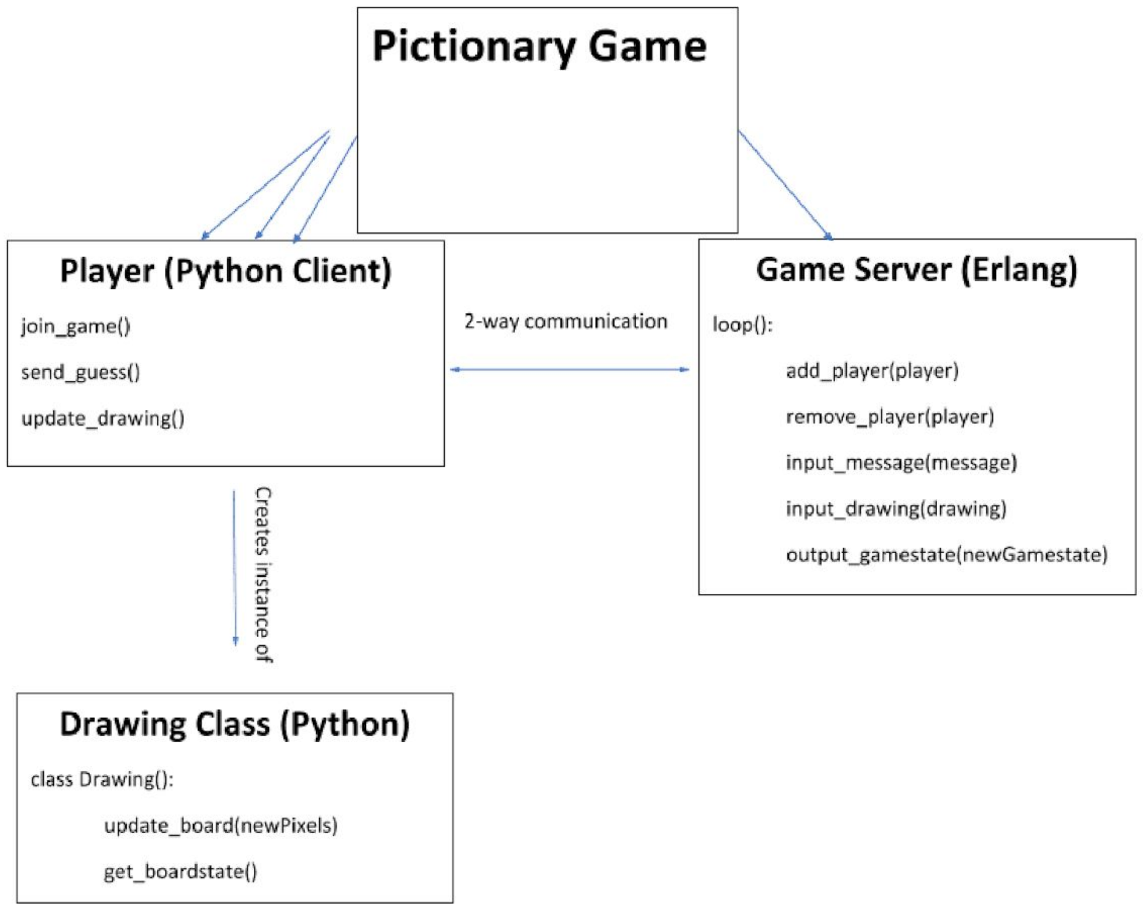
==
Class Diagram / Object Diagram
==

Object Diagram


Server
- list of guessers
- list of drawers
- list of queue of board changes (queue stores what changed, list stores one for each drawer)
- queue of {guesser, guess}
- current gamestate (drawing / voting / waiting)
- leaderboard (points per player, order of drawing, names)
Paint Stroke changelist
Paint Stroke changelist
Guess or Paint Stroke
Guess or Paint Stroke
Player (Client)
- list of all current drawings
- bool am_i_drawing
Player (Client)
- list of all current drawings
- bool am_i_drawing

A flowchart of data between given objects in the simulation. Each Player is a standalone client, possibly on different computers. Data is sent along the arrows.

Class Diagram



Final deliverable bundles the client and server together. User can choose to start their own server, or connect to an existing one when they launch the program. Game server is capable of the functions listed, and will use them within its loop function to maintain itself. Client consists of both the GUI area and a server communication backend, and some middleware classes that translate between them.

Changes to Design

As of now, no major changes to the design have been made. The class diagram and object diagrams still fully hold true for the structure of the program, and as such have not been updated since the initial project design submission. There were times where we considered changing to a python only design, but by that point we had worked so heavily with ErlPort and Erlang that we deemed it a bad decision. In hindsight, we may have adopted a design that developed both the client and server in only Python, so that we would not have had to deal with the frustration of ErlPort and getting it to work in conjunction with PyGame.

Analysis and Reflection

We were able to more or less reach our minimum deliverable. We have a mostly working pictionary that can work across multiple nodes on different computers. However, our maximum deliverable was a bit too ambitious with the lack of time for the development phase. We could have definitely added a lot more features to our model now if we had enough time to do so.

Our design overall was great and the implementation followed through with what we had in mind. The best design decision that we made was implementing this project with the client-server model, which made the networking aspect of the project simpler than if we had implemented it in the peer-to-peer model. Things that we wouldn't do next time are using the PyGame library, and using both Python and Erlang. The PyGame library caused us problems when trying to send messages between the client and the server, which would have been made simpler if we had implemented this project with just Python, since ErlPort was also tough to figure out.

Division of Labor

At the beginning, we made good boundaries with having certain people work on the client while others worked on the server. This way, the problems could be testing individually and then combined later. However, in the end, we had lots of troubles getting PyGame to work with ErlPort, so possibly combining the basic versions of our code through ErlPort earlier in the development process would have been beneficial. If we had done it that way, we could have possibly prevented us from having to modify large portions of the PyGame code so that it could be compatible with ErlPort.

==

Bug Report

==

As we mentioned a few times in other sections, as well as during the presentation, the most difficult bug integrated PyGame together with ErlPort in order to have the client and server interact. Specifically, PyGame wanted to run a while loop in the main thread of activation for the client's Python code. However, ErlPort required certain functions to be called in the Python in order for messages to be received from the server. Due to the nature of PyGame's loop, this ErlPort code to receive messages (which were necessary to be printed on the PyGame GUI) would not be executed until after the PyGame window was closed. To get around this, we had to change certain parts of the PyGame code as well as the server code, and had to use Python threading to have the client's Python code both listen for messages and run PyGame at the same time (and be able to share data safely).

==

Code Overview

==

In the main folder of the project there are two main important files:

- **sherl.sh** starts erlang with all of the necessary flags (sname, cookie, erlport, etc.)
- **launch.erl** holds the main functions launch:server() and launch:client() that startup the server for pictionary, and have the player join the game as a client, respectively

Within the server subfolder, there is one main important file:

- **server.erl** holds all of the code for the Erlang gen_server OTP part of the project, which handles the game loop, having players join the game, starting the game, counting the points, and handling and sending all messages to and from all of the players

Within the client subfolder, there are many distinct files:

- **chat.py** exports the chat class that allows for the sending and receiving of erlang messages, and handles drawing/writing to the GUI
- **recentlist.py** defines the RecentList class that is a list with a maximum age, where elements older than that age are removed
- **network.py** contains the NetworkHandler class, which is a wrapper for all communication with the Erlang server
- **event.py** has the EventHandler class, which handles all GUI events (keyboard input, mouse input, etc.)
- **panel.py** defines the Panel class, which handles drawing to the canvas on the GUI, adding multiple panels, removing panels, etc.
- **client.py** is the main part of the client, which uses all of the other classes in order to have a client join the game, update the GUI, and communicate with the server

How to Download and Play: https://github.com/lpiper01/PictionaryRoyale

1.) While in the main folder of PictionaryRoyale, run:

    ./sherl.sh nodeName

2.) In the erlang portal that comes up, run the following:

    c(launch).
    c(server).
    launch:server(cookie)

   Note that nodeName and cookie can be any input you wish, but are important to remember for later

3.) After the server is launched, run the following on the same node:

    launch:client(server, node(), cookie)

   “server” is an atom that is to be entered EXACTLY as above, node() is simply to specify the current node, and the cookie must be the same one as when the server was launched

4.) On a DIFFERENT node (so either a different terminal or a different computer on the same network) run the following:

    launch:client(server, NODE, cookie)

   This is the same as before, except NODE must be specified as the node that the server was created on (e.g. ‘nodeName@vm-hw07’)

5.) Now, there should be two PyGame windows open on two different nodes. Note that there may be a delay before the messages on the two windows get to each other when they are first opened. Be patient here. Now, messages and drawings on the two nodes are sent freely to each other. From this point, however many more players want to join can run launch:client the same way. When you wish for the game to start, someone must type:

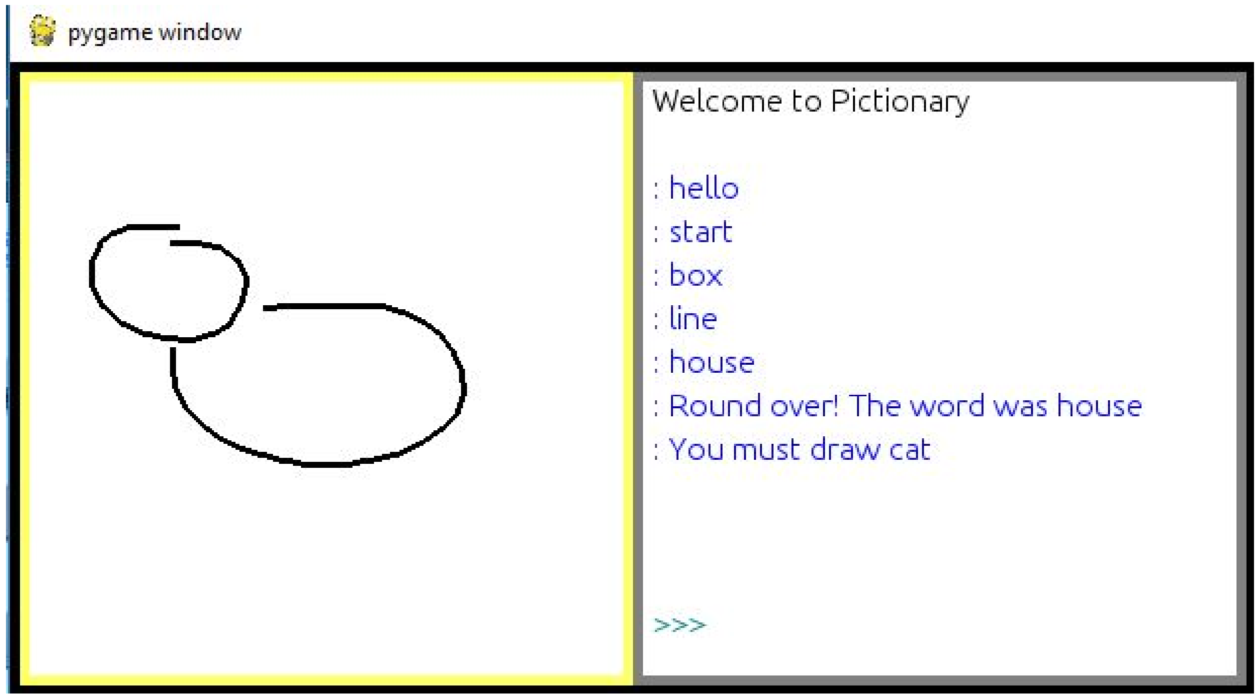
    start

   And enter this in their chat box. From that point, one player will be given something to draw, others must guess, and this will continue until everyone has drawn once and gotten their answer guessed. At that point, the game will be reset and someone can enter “start” (without quotes) in their chat box to start another game of Pictionary. Have fun!

Pictionary Royale

What it is: A multiplayer game where one player is designated as the drawer every round and the remaining players are “guessers” and type in the chat what they think the drawer’s given word is.

How it was implemented: Using ErlPort, both Python and Erlang were used to develop this project. The client-side was programmed in Python using provided Python modules such as PyGame. The server-side was implemented with Erlang due to the simplicity of the basic generic server OTP model the functional programming language provides.



```
Welcome to Pictionary

: hello
: start
: box
: line
: house
: Round over! The word was house
: You must draw cat

>>>
```