Assignment: A8

Multiplayer Demo

This assignment introduces network play and multiplayer functionality: a game server and clients. It illustrates single-room multiplayer network game play. The server displays the game on a single (projection) screen which must be visible by all the players. The server listens to the state of mouse and keyboards on each client. These client-state communications are primarily one-way in that the game state is not broadcast out to the client. Each client computer has a blank Pygame window (and supporting game loop) that detect mouse and keyboard events. Clients are typically on laptops.

This one-way approach offers a relatively simple implementation without significant latency problems. True two-way game play usually involves game-state broadcasting, multiple-engine synchronization, and/or a combination of both. Preliminary testing of PodSixNet indicated that state broadcasting would result in rendering delays if there were a large number of game objects (like streams of bullets; and who doesn't want streams of bullets?). And since this was to be applied in classroom/lab situation, this one-way approach seemed a practical compromise.

Python language topics:

- Another application of dictionaries
- Asynchronous socket service clients and servers (Wow. That sounds complicated, but we'll use PodSixNet which does all the heavy lifting for us.)

Problem statement:

(This assignment still uses Pygame and its event handling, but that's about it. So, we won't be building onto the previous assignment here.)

- Review the introductory help page for PodSixNet.
  http://mccormick.cx/projects/PodSixNet/

- Study the "Whiteboard" example (the three files) in the PodSixNet examples folder.

- To get the "Whiteboard" example running you will likely have to open a port in your Windows firewall. Then start the server and each client from a separate command window. Specify the host machines IP address (if localhost doesn't work) and port number as shown in this example:

  WhiteboardServer.py localhost:3333 or WhiteboardServer.py ???.???.??.??:3333
  WhiteboardClient.py localhost:3333 or WhiteboardClient.py ???.???.??.??:3333

  Note that if you have a general application-based firewall rule setup for Python (or Pygame), this may conflict with any attempt to make a specific port rule. These general rules sometimes get setup if Python attempts to connect to the internet for some reason, and if you blocked it when it tried to connect, this will cause a problem here. So if you are having trouble connecting, first look for an application based rule; delete it; then set up a firewall rule for a specific port. This should be an "inbound" rule in the Windows 7 firewall and an "exception" in the XP firewall. Port 3333 is used in the example above.

  In this Whiteboard example the server is broadcasting state data out to the clients. The server does not render; all the clients render. This is a two-way example, where each client sends drawing-related updates to the server

and then the server broadcasts the overall whiteboard state out to all the clients. Each client renders based on what it hears from the server.

- Create a similar drawing demo; look at the video associated with this PDF. Make this demo distinct from the Whiteboard example in that only the server renders and the server does NOT broadcast state data out to the clients. Clients send keyboard and mouse state data to the server; basically one-way data flow.
    - Clients:
        - Send state info to the server 100 times per second
            - mouse x and y pixel location
            - up/down state of the left mouse button
            - up/down state of keys a, s, d, and w.
    - Server:
        - Make a FIFO list structure in the server to keep track of the user's mouse locations.
            - When the user's left mouse button is down, add the incoming mouse position to the FIFO. Keep the length of the FIFO at 200; use the "pop" method to delete the oldest records from the list.
            - Make a list of these FIFO lists to keep all the client data in one place.
        - Draw the FIFO list as circles one time per game loop. Have a different color for each client.
        - Draw the client keyboard state as text U or D (up/down) for each of the four keys.
        - Keep track of the activity of each client; if they stop talking, stop rendering them.

Algorithmic description:

Many of the ideas for this demo are in the problem statement above, but here are some guidelines for coding the client/server functionality.

The data flow, for drawing on the server, is from client to the server. But there is also an initial connection from the client that triggers a server "send" to the client; this contains an ID number that allows the client to name itself. This ID is also used to establish a color for each client as it connects to the server.

- The initial connection (client to server to client)
    - The server will need a "GameServer" class that inherits from the parent class "server." Here the "Connected" function will run whenever a client connects. "Connected" executes the "send" method of the "channel" object. This sends a dictionary that contains the client ID that will be used to establish the client color.
    - Client will need a NetworkListener class that inherits from the parent class "ConnectionListener." On instantiation, it connects to the server. It should have a Network_hello function that runs one time when the connection is established; here the client ID is received and the client color is established.
- Main data flow (client to server)
    - Client: The data dictionary that is sent by the client should be updated based on the mouse and keyboard states gleaned from the Pygame event queue. In the game loop, update the dictionary, and then send it with the "send" method of the "connection" object.
        - The dictionary needs to have an "action" key that identifies the name of the receiving function for the server. For example, I used the value of "CN" to indicate the receiving function will be "Network_CN."
        - The dictionary also needs to have an identifying key that associates it with the sending client.

    - Server: The server receives the client's dictionary in the Network_CN function. This function stashes this incoming data into a server object and updates the FIFO data that represents the client cursor history. This function also keeps track of the activity of each client (counting the sends); rendering can be inhibited from those clients that have stopped sending data.
- Hiding inactive clients: clients can be flagged as inactive if their send count hasn't changed. Then based on this flag, the server can stop rendering inactive client's. The server object that stores the client data has a method called "checkForQuietClients" that checks for client activity. This can be called every few tenths of a second in the game loop to check for quiet clients. Quietness can be caused by shutting down the client window (and also by simply mouse-dragging the client window).

  Note that the approach described here for identifying inactive clients is used in all of the final assignments but not some of those between this one and those. There is an earlier approach to this that didn't get edited out of some of the intermediate assignments (TBD).

  Also note the Whiteboard example uses a very different approach for keeping track of disconnected clients. This involves WeakKeyDictionary objects and appears like it might be a bit abstract for beginning programmers. The counting approach described above seems more straightforward.

Python code: (see source code line on web page…)

If you can, try to work through this problem without looking at the source code. If that works (you learn); great. But peeking is encouraged here. First, start with the help page (see link above). Then, get the Whiteboard demo running. However, there's a lot going on there that we don't need. So focus on this PDF before spending too much time to completely understand the code in the Whiteboard example.

Obfuscated code or incremental code didn't seem useful for this assignment, so no screen shots here.