Megan Fanning
Brandon Swanson
Miranda Weldon

Documentation for ETA-Team Endless Runner Mid-Point Project Check

Table of Contents

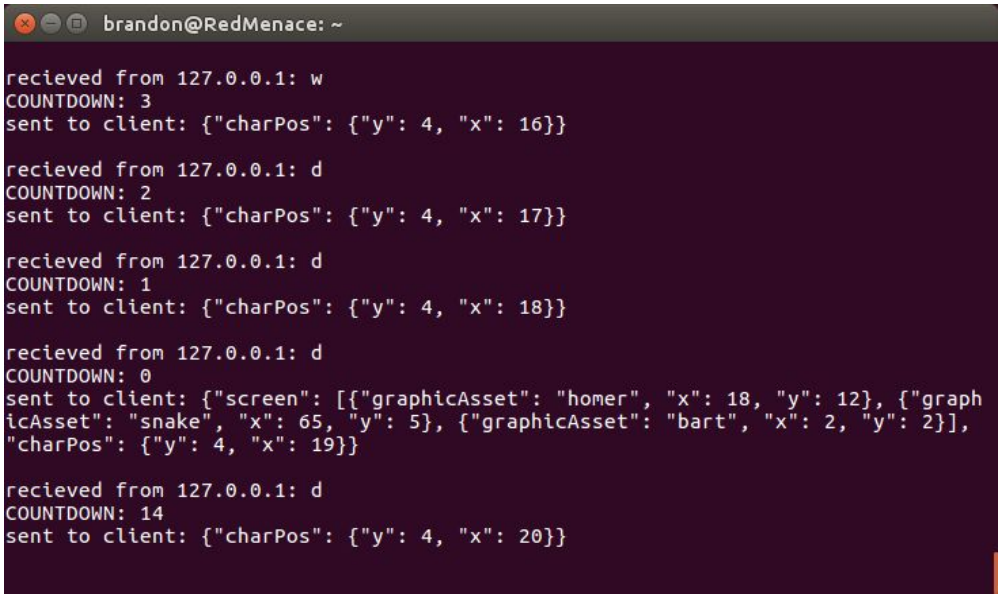
To run program

First run server.py, this will print out the IP and port (we have hard-coded the ip to local and the port to 9999 for testing).
Next begin the client.py in a seperate terminal/flip-session on the same host computer (flip1$ python client.py 127.0.0.1 9999).
Once the server and the client connect the game will begin (a curses screen will open and display game ASCII art).

Running the program on flip server

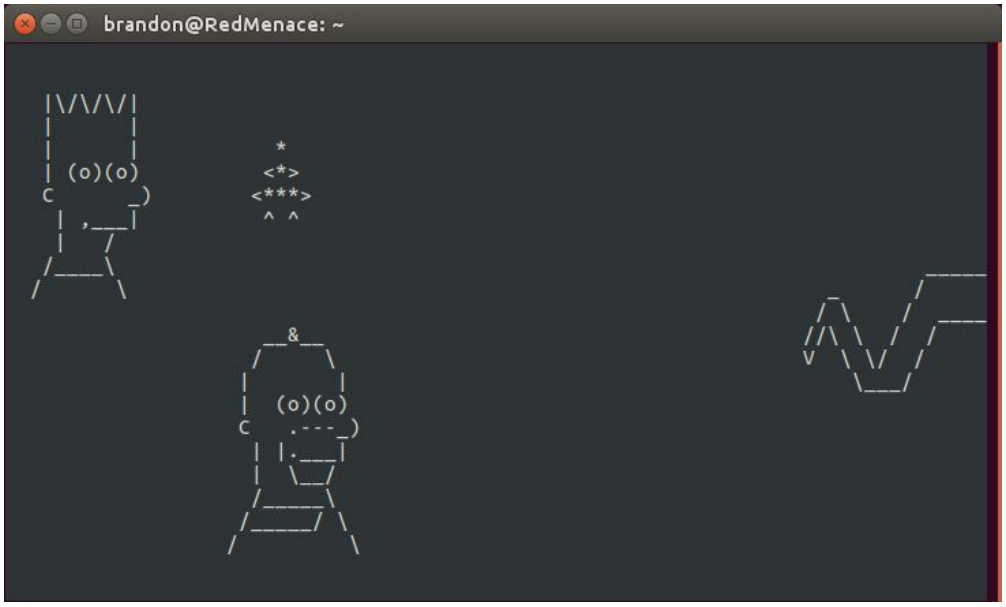
After transferring the project code to your home directory and establishing two ssh sessions with the same flip server (flip1, flip2 etc)

```
$ ssh <ENGRName>@flip1.engr.oregonstate.edu
$ cd ETA-Endless-Runner/
$ python server.py
```



Server script running on flip server

```
$ ssh <ENGRName>@flip1.engr.oregonstate.edu
$ cd ETA-Endless-Runner/
$ python client.py 127.0.0.1
$ CURSES-INPUT: w,a,s,d  q/esc to exit
```



Client script connected to server and rendering game elements received from server

Controls

players direct movement using wasd or arrow keys to move, q or esc key to quit.

Game play mechanics

One player will control the vertical axis and the other player will control the horizontal axis of movement. The player must navigate around obstacles and move to the exit of the screen to progress through the levels. For the purposes of this demo there is a single user who can move any direction.

Game play mechanics for mid-point demo:

- For testing purposes we will only have one client connecting who will control both horizontal and vertical movement.
- In this demo the player is able to move around a screen with a few randomly generated hazards/obstacles that may be overlapping.
- The player can move freely through obstacles and off screen.
- During this demo the obstacles/hazards displayed on the screen will be re-generated by the server and transmitted to the client to be rendered. The server does this every 15th round trip interaction between client and server. The purpose of this is to demonstrate the process of the server transmitting a new room with a collection of drawable entities and the curses client updating its local game model used for rendering each frame.

Networking

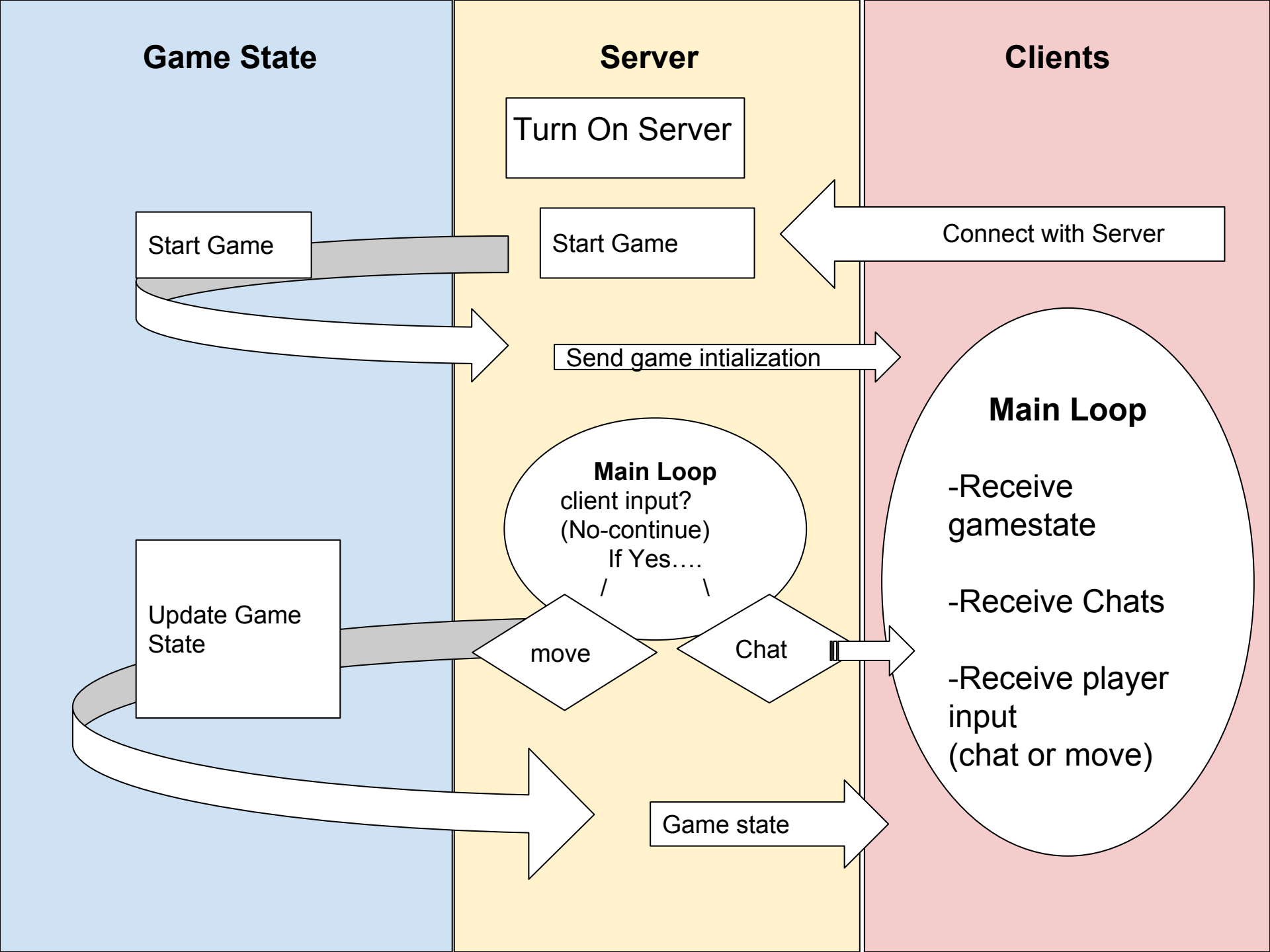
The network client accepts: strings of user input and passes them to the server as either strings (for messages) or constants (for directions) from the curses interface.The server returns: JSON formatted files containing information about player location and graphics.
Functionality: Upon accepting user input the client sends the data to the server, the server then parses user input for movement keys. If a movement key is found then the game state is updated and this update is sent back to the client.

Game State

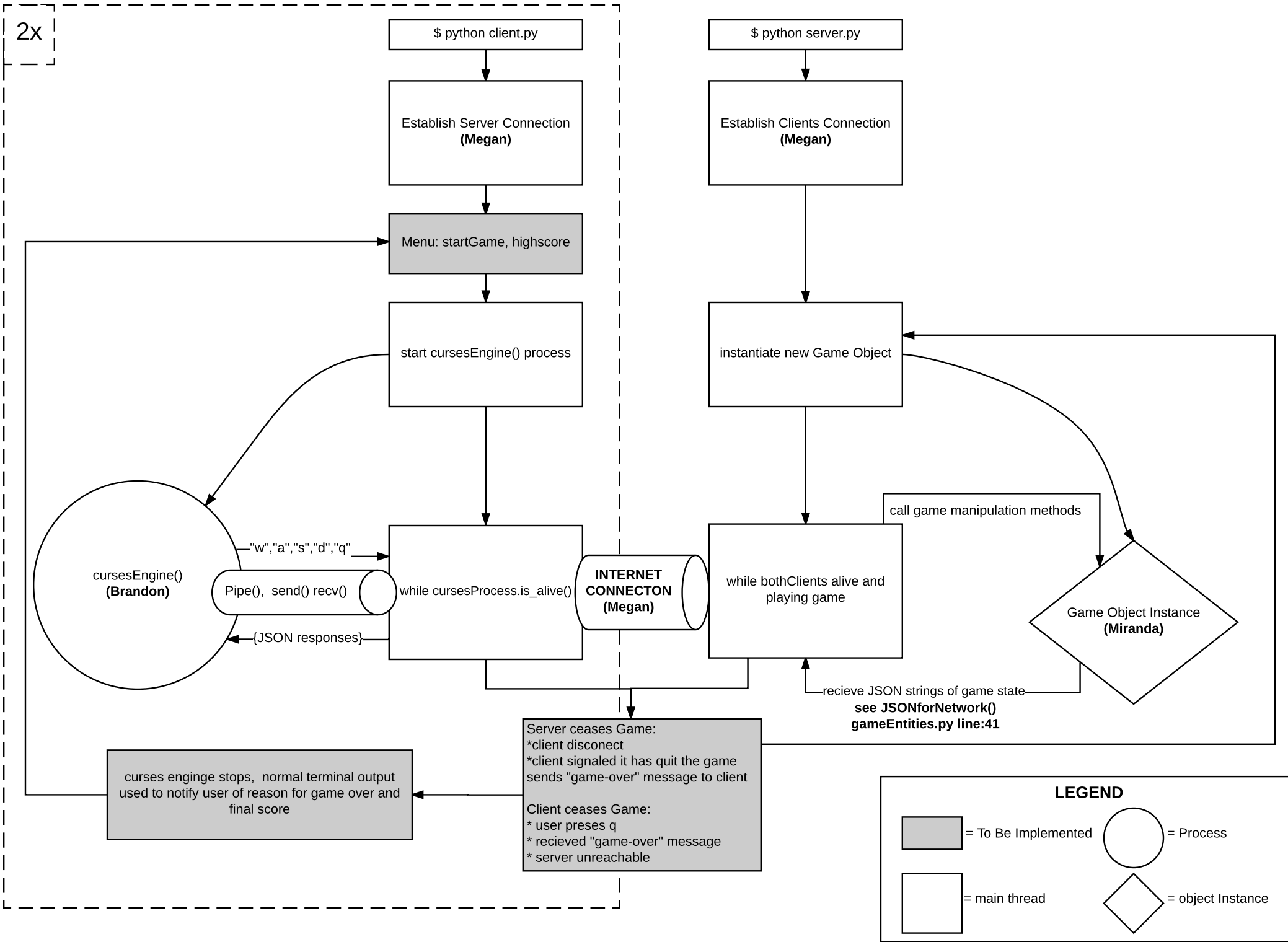
The Player class handles single space movement. The Obstacle class places and update obstacles in a room. The Grid class keeps track of dimensions (width and height) of playable grid. The Gamestate class keeps track of timing, scoring mechanisms, sets graphical assets, sends requests to update other classes' variables. All of these classes come together to keep track of all the internal game state variables that keep this game running consistently.

Curses Engine

The curses engine is launched as a separate process and manages collecting input from the user (non-blocking) and rendering the game state as it is transmitted from the server. The curses process communicates with the network interface over a Pipe connection allowing the network and curses component to communicate using a producer-consumer pattern. Rending game entities that share property definitions between server and client, defining ascii art in an external file library, and transmitting the game state from server to client “by-name” as opposed to “by-drawing” are all achieved through the combination of the GraphicAsset and GameEntity classes.


Game State
Server
Clients
Turn On Server
Start Game
Start Game
Connect with Server
Send game intialization
Main Loop
-Receive gamestate
-Receive Chats
-Receive player input
(chat or move)
Main Loop
client input?
(No-continue)
If Yes….
move
Chat
Update Game State
Game state

ETA - Endless Runner InterConnection Scheme



ETA Endless Runner Network Message Protocol

Messages sent to server

Movement: “w”, “a”, “s”, “d”
Quit-Game: “q”
Chat-Message: / followed by chat string, example: “/hello other player”

Messages sent to client

JSON strings representing a dictionary containing 1 or more of the following key value pairs:
"screen" : [{}, {}] Array of dictionaries defining game entities with position and drawing asset
"charPos" : {} dictionary of y and x position of
"gameOver": {} dictionary containing reason for quitting and game score (k:v not yet defined)

Example:

```
{
   "screen":[
      {  "graphicAsset":"skull", "x":74, "y":9},
      {  "graphicAsset":"enemy", "x":23, "y":4},
      {  "graphicAsset":"clover", "x":57,"y":3},
   ],
   "charPos":{ "y":12, "x":28 }
}
```

A message like the one above (containing the “screen” key) indicates that the character has moved to a new room/screen and new elements will need to be drawn and should always be accompanied with the new “charPos” because the client invalidates the local record of the characters position after receiving a message to update the screen (since the character will likely enter a new screen at a different coordinate than that from which they exited).

To facilitate the Curses client be able to parse the messages from network the Curses engineer has developed and maintained a method for JSONifying relevant game states.
https://github.com/swanyriver/ETA-Endless-Runner/blob/master/gameEntities.py#L41-L49
It can be used with kwargs in order to send any combination of: screen, character position, game over messages (both charY and charX required to send character pos.)
Params expected types:
screen=[<GameEntity instances>, <>, ..] charY=Int charX=Int gameOver={}

Log output from test run of network protocol

(CURSES-GAME): char new pos (12, 23)
(CURSES NET-OUT): d
(NET MSG-FROM-CURSES):<type 'str'>d
(NET MSG-TO-CURSES):{"charPos": {"y": 12, "x": 24}}
(CURSES NET-IN): {"charPos": {"y": 12, "x": 24}}
(CURSES-GAME): char new pos (12, 24)
(CURSES NET-OUT): d
(NET MSG-FROM-CURSES):<type 'str'>d
(NET MSG-TO-CURSES):{"charPos": {"y": 12, "x": 25}}
(CURSES NET-IN): {"charPos": {"y": 12, "x": 25}}
(CURSES-GAME): char new pos (12, 25)
(CURSES NET-OUT): d
(NET MSG-FROM-CURSES):<type 'str'>d
(NET MSG-TO-CURSES):{"screen": [{"graphicAsset": "skull", "x": 74, "y": 9}, {"graphicAsset": "enemy", "x": 23, "y": 4}, {"graphicAsset": "clover", "x": 57, "y": 3}, {"graphicAsset": "enemy", "x": 7, "y": 6}, {"graphicAsset": "skull", "x": 43, "y": 4}, {"graphicAsset": "clover", "x": 35, "y": 15}], "charPos": {"y": 12, "x": 28}}
(CURSES NET-IN): {"screen": [{"graphicAsset": "skull", "x": 74, "y": 9}, {"graphicAsset": "enemy", "x": 23, "y": 4}, {"graphicAsset": "clover", "x": 57, "y": 3}, {"graphicAsset": "enemy", "x": 7, "y": 6}, {"graphicAsset": "skull", "x": 43, "y": 4}, {"graphicAsset": "clover", "x": 35, "y": 15}], "charPos": {"y": 12, "x": 28}}
(CURSES-GAME): new screens entities: (6) [<gameEntities.gameEntity instance at 0x7f20303e43f8>, <gameEntities.gameEntity instance at 0x7f20303e4290>, <gameEntities.gameEntity instance at 0x7f20303e42d8>, <gameEntities.gameEntity instance at 0x7f20303e41b8>, <gameEntities.gameEntity instance at 0x7f20303e4368>, <gameEntities.gameEntity instance at 0x7f20303e4128>]
(CURSES-GAME): char new pos (12, 28)
(CURSES NET-OUT): q
(CURSES GAMEOVER):this client pressed quit
(CURSES): curses screen exited
(NET): closing connection