Megan Fanning

Brandon Swanson

Miranda Weldon

Final Report

Introduction

Our program is an “endless runner” that procedurally generates obstacles which the users sprite navigates past using their keyboard to move up, down, left or right. This program is networked in such a way that two clients can connect to a server and collaboratively navigate a character through a game world that is hosted on the server and displayed in both client applications.

Description of what your program does (from the user's perspective).

Players log in and control one axis of movement (horizontal or vertical). Players enter a room which contains enemies and obstacles. To avoid hitting objects they must send chat messages to their partners to alternate taking turns so they can safely traverse the room.

Usage instructions

First download the source code onto the flip server, you will want all of your connections to be on the same server since it’s set up to be played locally. Meaning if you connect to flip1.engr.oregonstate.edu to run the server, ensure that you explicity connect to flip1.engr.oregonstate.edu to run connected clients and not flip.engr.oregonstate.edu as that connects to arbitrary flip servers. (If you aren’t playing on flip you will need to change the python prefix to python2.7, if you use a tiled window manager use “floating window” mode to allow the curses client to properly size the image window.)

To Launch in two player mode:

- Step one: Launch server using the following command supplying an optional port number or allowing the kernel to allocate an arbitrary port number:

  ```
  $ python multithreadedserver.py [PORT-NUMBER]
  ```

  Once launched the server will display the ip address and port number necessary for connecting to it, as well as print logging information about network traffic and from the game state

- Step two: Launch 2 connected connects clients in separate connections to the same flip server.

  ```
  -$ python client.py 127.0.0.1 <PORT-NUMBER >
  -$ python client.py 127.0.0.1 <PORT-NUMBER >
  ```

To Launch in one player mode:

The provided shell script `singlePlayer.sh` will launch the server script in the background, with a single player flag supplied and with its logging output redirected, and then launch a connected client with full movement priveleges. Allowing you to play the game in a single terminal/flip-connection.

- Launch in command line use:

  `$ ./singlePlayer.sh <PORT-NUMBER>` or

  `$ bash singlePlayer.sh <PORT-NUMBER>`

- A valid port number must be supplied as the first and only command line argument in this case to allow the client to automatically connect to the initiated server.

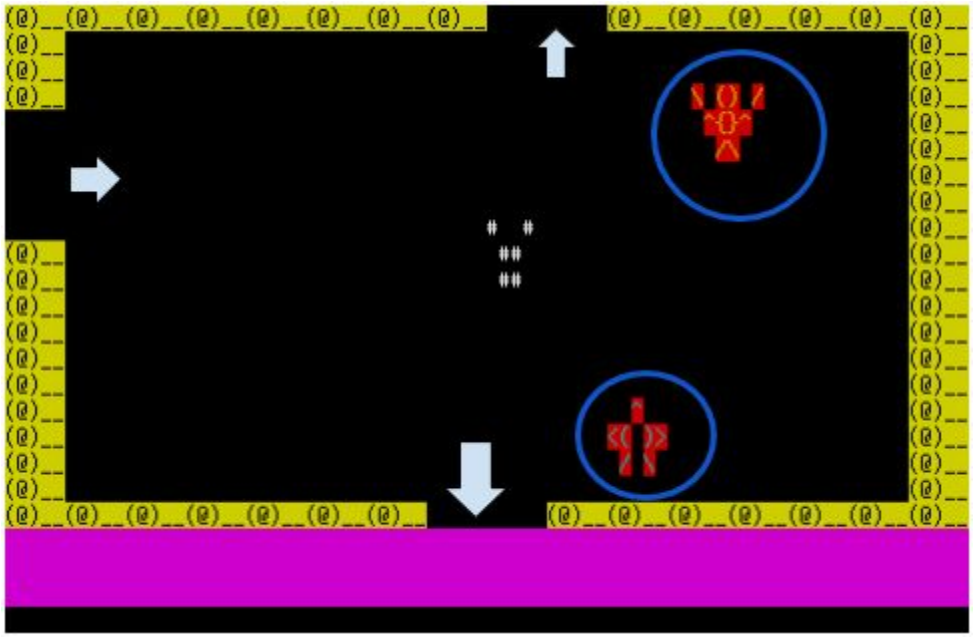
To play:

- Players can navigate the maze of rooms using either the arrows or wasd keyboard interface to move up down left or right. With vertical movement assigned to connected client, and horizontal movement to the other.

Commands available to players:

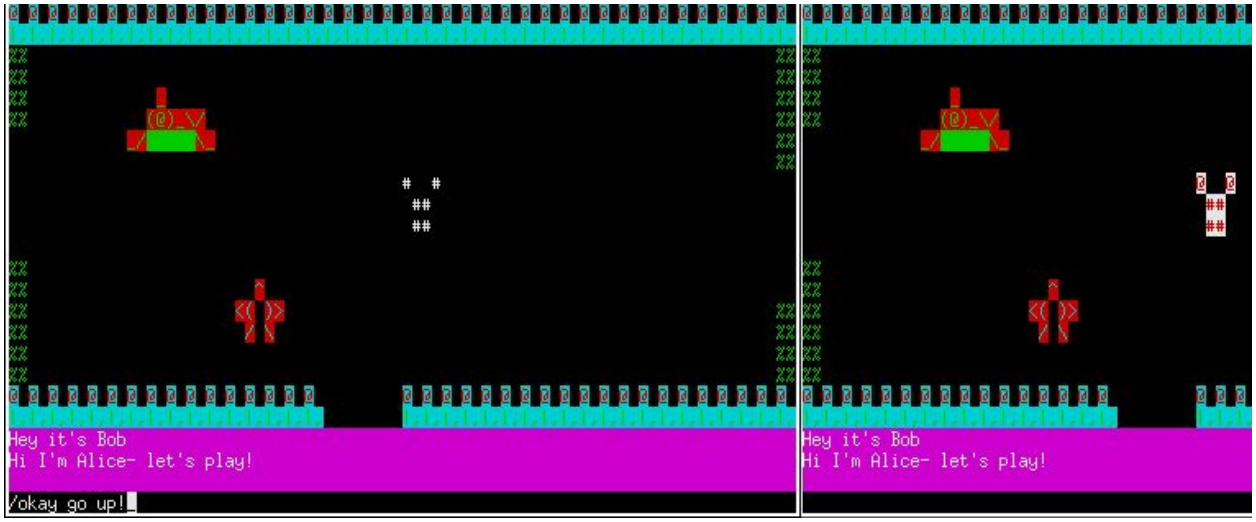
q	Quit, closes client connection
/[chat]	Any text prefixed with “/” is sent as a chat
W or up arrow	Moves player up
A or left arrow	Moves player left
S or down arrow	Moves player down
D or right arrow	Moves player right

- The goal is to avoid enemies (characters with the red backgrounds) and to make it through as many rooms as possible. Upon collision with an enemy there is a death animation and the highscores will print showing how many rooms the player managed to navigate and how they died



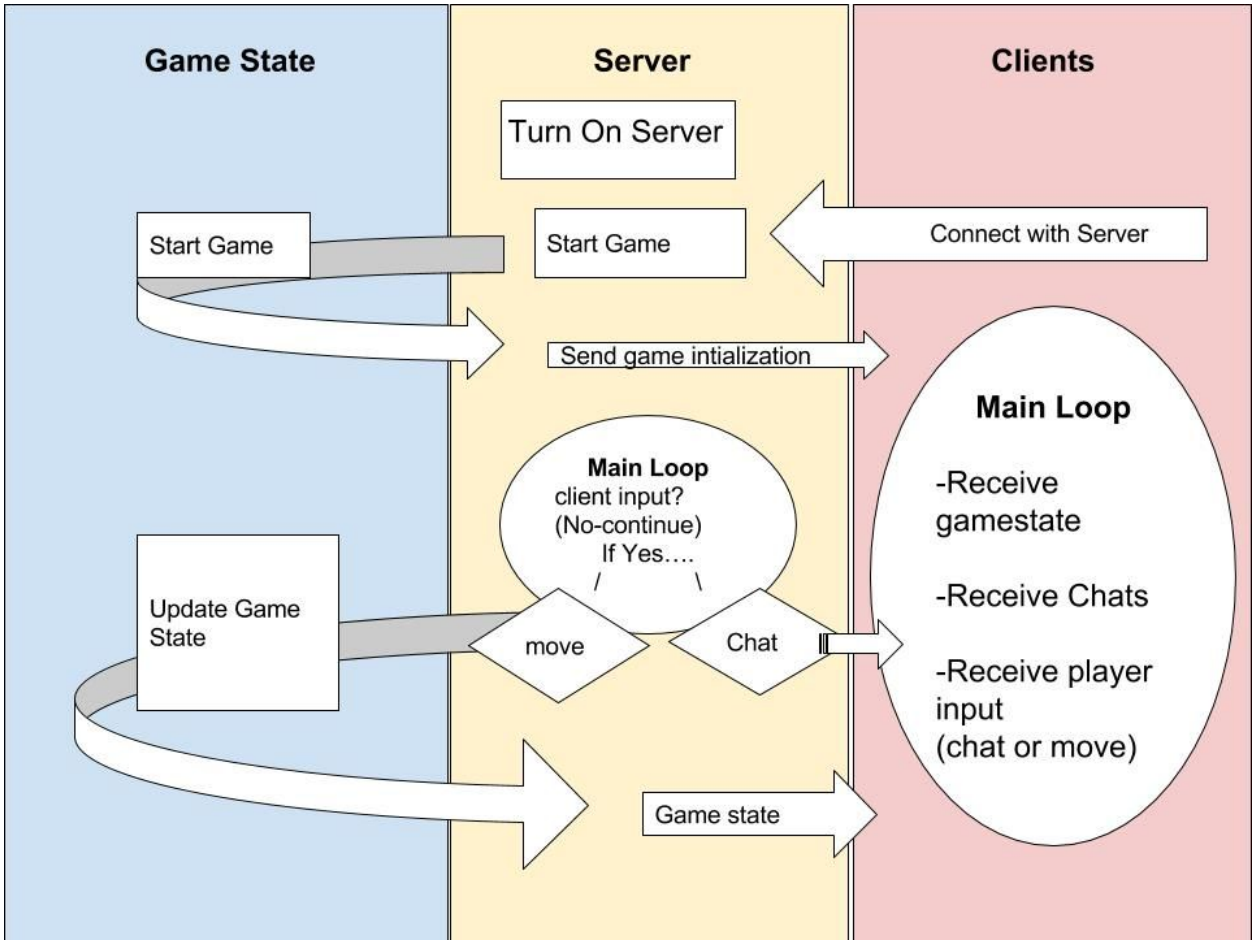
In the image above the players have entered the screen from the left and can exit to the top or bottom of the screen and must avoid the enemies (circled in blue, all enemies have a red background color besides the snowman who has red ascii and a white background)

The chat window is magenta,which contains a running log of chats, below that is a black field which players can view the messages they are drafting prior to sending.Players should use chat to communicate about how to move together to avoid obstacles. Two players screens are shown side by side below to demonstrate what the chat looks like (see below).



Description of how the software and systems function together.

There are three main components to this design, the graphics, the networking and the game state.A typical game begins with turning on the server which instantiates a game object and begins accepting incoming connections. When the client connects it prompts the user for their user name which it sends to the server, it then receive from the server the current game state and creates a window in which curses runs showing the current game state. The network client launches the curses keyboard input and game-state rendering engine as a separate process to facilitate non-blocking keyboard input, screen rendering, and network updating. Once this initial setup is done user input is passed from the the curses input into the client, and then passed to the server and updates from the server are passed from the client script to the curses process, which updates its local model of the game state and updates its rendering. (see the graphic below which demonstrates the game flow between the three primary components.)



The above chart visually demonstrates the game 'flow'.

Graphics

Use of the Curses Library

For rendering of the game and collecting input from the user we used the Python wrapper around the curses terminal display/input library. This library allowed us to “draw” a screen using ascii characters in a coordinate grid, and collect keyboard strokes in a non-blocking manner. Early in the development we tested a few different versions of the render and input loop using different methods of non-blocking input and methods for regulate the refresh rate. The smoothest experience was found to be setting curses cbreak and nodelay mode, which allow keystrokes to be collected from a buffer asynchronously; and setting an update frame-rate by measuring the time since the last render. The engine we created is launched as a process connected to the networking processes via a Pipe object (an object wrapper around mkfifo interprocess communication) and after initializing a screen it continuously: queries the keyboard for input and when appropriate sends it to the network process, queries the network process for updates from the server and updates its local model to match the state of the server, and on a defined interval draws the game state to the terminal window using.

Graphics Challenges and Design Decisions

During the proposal phase we decided to follow the design pattern of Model-View-Controller to aid in keeping our design well encapsulated and strictly defining the responsibilities of our team members and the classes we created. The state of the game and the rules of the game would be defined and maintained on the server without concern for the methods of displaying the game (model), and the curses client rendering this game (view) without concern for the rules of the game. The responsibility of the controller, mapping user input to modifications to the game state and relay the changed state fell mostly on the networking classes.

Adhering to this design specification allowed us to work in co-operation but also safely make redesign to our respective components. But to preserve this level of encapsulation it was necessary to create a method for defining the graphical assets and instances of entities within the game that make use of these assets that could be shared between the game on the server and the rendering client. A class for managing an external library of graphical assets was created so that the game engine and client could both access a collection of drawable assets, where the game could use the height and width to determine how to place them in a new room, and the client could use the character arrays, background/foreground color arrays, and animation frames to render these assets. With this shared library we were able to transmit the information required for rendering the game as a JSON with renderable entities described by name and position, rather than requiring to transmit the pixel by pixel (character by character) grid of what the game world looks like. This allowed the game functions to maintain their

encapsulation by focusing on maintaining a model of the game and not a drawing; and allowed the rendering engine to continue to add features such as animation and color without breaking the establish game model and communication protocol.

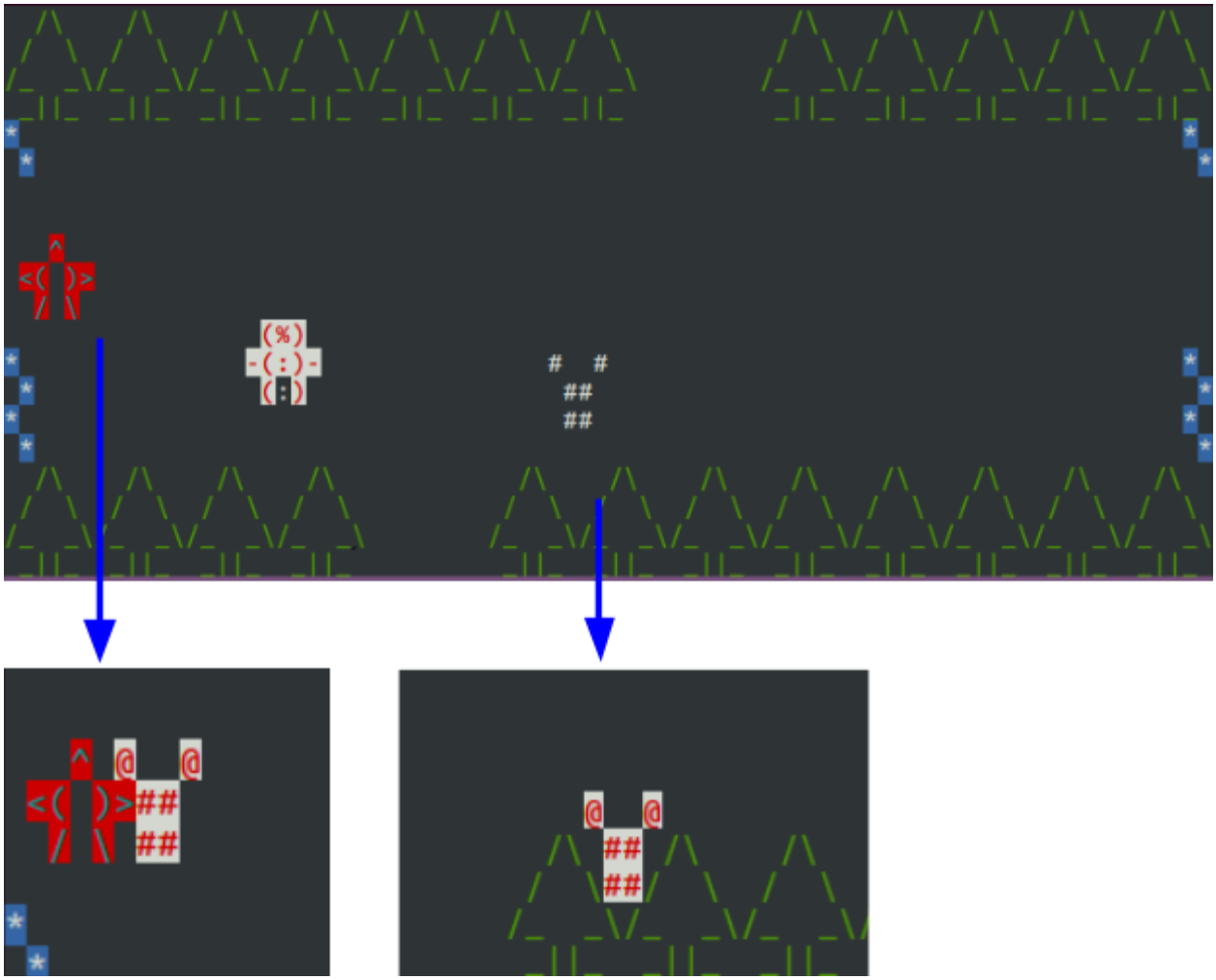
GraphicAsset class and GameEntity class Usages

Communication about the game state from the server to the client was accomplished with the help of the GraphicAsset class and its factory function that produced a GraphicLibrary dictionary subclass containing all successfully parsed GraphicAssets from available JSON files. Each of these assets could be used to create an instance of a GameEntities, these are an instance of the available decorations, hazards, or player constructed with a reference to a GraphicAsset and a coordinate position. As an example to elucidate the relationship between these two classes, after both the client and server have loaded the graphic asset from file for the “owl” enemy they both have an instance of the GraphicAsset, which contains necessary information for generating a room like the height, width and hitbox shape for that available asset. Then each time the server places an “owl” entity it only must specify the position of the owl and the owl graphic-asset and this position combination (an instance of GameEntity) can be used to get a set of points that comprise the hitbox at that entities position or in the case of the DrawableEntity sublcass in the curses client can be used to get a collection of ascii-character, y,x-position, and color to draw the entity in its position. The “owl.json” file never needs to be parsed from file again and any number of other owl entities can be instantiated and transmitted from server to client (which uses its own GraphicsLibrary instance from the parsed JSONs to instantiate its own instances) and then rendered

.

There are also a few helper functions that accept collections of these game entities as parameters and assist in performing essential game functions such as preparing a JSON string for network communication from a collection of entities and the player (also a subclass of GameEntity). On each movement of the player the collection of game entities is used to check for collisions between the player and any decoration or hazard. This is done by creating a dictionary that associates every point that is in each GameEntity in the collection of game entities and then checking for each point in the player’s current hitbox (updated after a request from the client to move one space in a specific direction) if there is an entity occupying that space. These hitboxes are generated automatically from the supplied graphic by applying a flood-fill algorithm that removed every point from its rectangular bounding box that can be reached without crossing an ascii-character used to represent the contour of the asset. This is stored in a 0,0 coordinate space and then each instantiated entity simply returns this calculated hitbox with their y,x delta applied.

An exciting consequence of using such granular collision detection (as opposed to simple bounding box detection), is that the players can truly navigate their characters within complex shapes and not find themselves colliding with corners/edges of the drawing array. See example below.



This image shows a Y shaped player character is able to get very close to this T shaped enemy, and is able to wedge itself between the peaks of the trees.

Graphic Asset Creation Tool

To facilitate the creation of graphic asset JSON files that would successfully parse we created a simple API that we could all access by hosting on GAE. This API accepts POST requests containing the necessary metadata (deadly (t/f), category) and an array of drawing frames with foreground and background colors and provides an HTML form for creating these requests. It then generates and returns a formated JSON that takes care of many of the arduous tasks that made creating these assets by hand rather un-enjoyable. These include removing unnecessary whitespace padding and adding padding as necessary to make the drawing arrays uniformly shaped, and most importantly automatically escaping necessary characters like backslash or quote marks. The API will also ensure that the supplied values and the generated JSON will successfully parse and comply with all constraints by running our parsing and testing script on the GAE server before returning the JSON to the user. A common

case for this test failing is if different frames of the drawings have differently shaped hitboxes. Because the server is unaware of what animation frame the drawing is on on the client (in fact the server makes no distinction between the animated and not animated assets) it is necessary to ensure that all frames of an entity occupy the same shaped area.

The source code for this web-app can be viewed within the submitted code in the folder /GraphicMaker and the produced graphic JSON files in /graphics.

The web app can be seen live at the following address:
http://etagraphicmaker.appspot.com/

Networking

The game networking was done using pythons SocketServer class, class I extended the TCP SocketServer class with the Threading mix in to create a custom request handler which used locking to allow two players to share the same game state object.

The Network accepts strings of user input and passes them to the server as either strings (for messages) or constants (for directions). The Server returns JSON formatted files containing information about player location and graphics which are interpreted by the graphics handlers. Messages are passed using a queue.

Game State

Description of class designs and interactions

The game state is maintained by a number of separate classes that come together under a central Gamestate class. An initialization of the Gamestate class will create a number of variables and in turn initialize a grid,a player, and various graphical assets. Then it will make a call to getNewGameRoom() in order to populate the first room for the player, which will then place the player within the playable grid. One of the main functions within the Gamestate class is get_change_request(), which handles player movement and coordinates any changes to the internal state and sends any updates (player death, left current room, ran into obstacles, etc.) back to the server to give to the client(s). The Gamestate also keeps track of timing in order to determine the total score and contains functions to interact with the server when there are any updates.

The Player class is significantly simpler, only really concerning itself with the x and y coordinates of a single instance. For this, it has functions to set, move, and remove the player. Notably, the remove function does not erase any variables but instead moves both coordinates to -1, outside the playable grid. The Obstacle class is very similar, except instead of handling movement it handle updates to its name and position and sends these updates back to the

server. The Grid sets the default screen size for the playable screen, currently set at 20 rows by 80 columns.

Random Room Generation Algorithm

By the time of the midpoint review we were able to define any collection of hazardous and decorational entities, and transmit those in a defined JSON format to all connected rendering clients, which were capable of displaying on screen any combination of entities within the defined coordinate space. The curses engine, networking client and server, and GameState class were able to coordinate the movement of the player character on screen and detect when that player had collided with obstacles or hazards. What was left was creating a system of rules and procedures for generating an environment that could turn all of the interworking systems and instances of drawable entities at a specific location into a playable game, with objectives and failure conditions.

The goal of our room generation algorithm was to allow the player to move between different rooms that were composed of elements from different themes or categories. These rooms would have varied amounts of exits in varied locations with walls surround the room composed of decoration assets from the selected category. These decoration assets would prevent the player from moving through them but are safe for the player to collide with. The player increases their score by moving between more and more rooms but as the game progresses more enemies are placed in each room and any collision with these enemies will mean game-over for the player. These rooms are generated according to a systematic approach that guarantees each room will adhere to certain constraints to maintain the game play features we desired, so while random choices are used it might be more accurate to classify this as a procedural generation process.

Step by step process for generating game rooms

- A category of graphic assets is randomly chosen from those available in the graphics library
- Assets from this category are chosen for creating the outer walls
- Between 1 and 3 of the walls (that the player hasn't entered from) are chosen to contain exits
- Walls are placed with location of selected gates placed randomly and a gate reserved for the players entrance (that forbids the player from retreat)
- A Safe-Path is created for the player that leads from its entry to at least one other exit, Sometimes this path takes the most direct route, other times it takes the most indirect route.

- A coordinate set is generated that includes all of the already placed decorations hitboxes (also coordinate sets, not rectangular) and a collection of coordinates that includes every point that the player's hitbox will include if it were to traverse the defined Safe-Path
- Using this coordinate set (exclusion set) enemies from the selected category are placed by:
    - Creating the set difference of all possible coordinates and the exclusion-set (possible-placements set)
    - (step 2) Choosing an enemy from the category randomly
    - Choosing randomly from the set of points defined by:
        - All points in in possible-placements such that placing selected enemy at that point would not cause its hitbox to intersect with any of the entities used to make the exclusion-set *
        - Placing the enemy there would not result in the enemy being partially “off-screen”
    - If that set of feasible-placements is empty then there is no where remaining in this generated room that the selected enemy will fit and that enemy is removed from the set of possible enemies to be placed
    - Add enemy to game-state at randomly selected location and update the exclusion-set to include the hitbox of the newly placed enemy
    - Continue placing enemies (return to step 2) until either:
        - The desired number of enemies have been placed
        - OR for all enemies from the selected category there is no location it can be placed that would not be overlapping with some other entity

The most crucial part of this generation strategy was pre-calculating the reserved exit and safe path for the player before placing enemies. This has allowed us to gradually increase the difficulty of the game during a player's session but maintain the concession that a skillful player could continue to traverse rooms endlessly. As the number of enemies placed grows the contours of this patch can become easier to spot, but there are still moments of surprise in the game available when enemies block all but one of the available exit, or a room is generated with only one exit and it appears to be blocked by an enemy but after venturing over the player discovers that their avatar can just barely (even exactly) fit through the available gap. And randomly choosing to place this safe-path directly through the room or around the outside forces the player to pay attention as sometimes there will be a great deal of enemies directly in their path between their entrance and the rooms exit.

Demo of Random Room Generation

https://www.youtube.com/watch?v=5hVaeZNO5u8&feature=youtu.be

The above video demonstrates and explains room generation that was described above.

Unit Tests

During development we created a number of testing utilities to test functionality, prevent regression, and test parts of the system in isolation. They are as follows:

Room Generation Test

Purpose:
Launches a curses client and continually generates rooms with an increasing amount of enemies, allowing us to quickly verify the functioning of the room generation algorithm and put the function through extensive stress testing.

To Launch:
```
$ python Unit-Tests/testRoomGenerate.py
```

Client Fuzz Testing

Purpose:
Allow the networking code to be tested without the need for running multiple curses terminal windows. The fuzz client is able to simulate any number of connected clients and allow us to ensure that the server was properly responding to incoming messages and forwarding game status and chat messages to all connected servers.

To Launch:
```
$ python multithreadedserver.py <PORT-NUMBER>
```

In a separate terminal connect 2 clients via:
```
$ python Unit-Tests/fuzzClient.py 127.0.0.1 <PORT-NUMBER> 2
```

Graphic Library Testing

Purpose:
The parsing of graphic assets was designed to fail gracefully, The graphic asset factory function responsible for parsing all of the available JSON assets would return a collection of all of the assets that had failed without error. In order to be able to investigate the state of the graphics library and debug the reason for a new asset not appearing the game the graphicAssets.py file contains testing routines to be run when the script is run by python rather then imported by another script.

An example of using this to find problems with bad assets and displaying the available graphics with their generated hitboxes and animation frames can be seen in the included source code at: Unit-Tests/graphicsTestSample.txt

To Launch:
To view the full graphics library launch:
```
$ python graphicAssets.py -d
```

To view a verbose output of errors and successes parsing view included graphicsTestSample.txt as it includes examples of failed parsing or launch:
```
$ python graphicAssets.py
```

To view a single asset rendered in the curses client launch:
```
$ python graphicAssets.py -f graphics/<FILENAME>
```

Listing of which software tools used to create the software.

- Program written in Python.
- Graphics library:Curses
- Networking library used : TCPServer, Socket, Pipe, threading mix in
- Game Objects and highscores were saved to JSON files, game graphics were made using a applet (http://etagraphicmaker.appspot.com/) which was created by Brandon.
- Testing done both on local machines (arch Linux and ubuntu distributions) and on the school's linux server, Flip. Flip server (linux) will be the deployment platform (for both client and server scripts)
- Used a Github hosted repository for development collaborating and used a master/dev/topic branch strategy.

Description of what each Team member accomplished

- Megan: Designed, implemented and tested both a single threaded server and a multithreaded server and clients, handled client input and output (including limited directional movement, chat, and usernames), integrated the calls to the game state. Additionally Megan generated over 20 game image assets. Megan did game testing and debugging.
- Brandon: Implemented the curses non-blocking rendering and user input engine, allowing images and chat to be displayed while players control player movement. This included creating classes and functions that allowed for an external library of graphic assets that could be shared by the client and server along with tools for easily creating and testing this graphics library. Brandon also created a message queue for relaying

game updates and chat messages to all connected clients, testing frameworks, highscore function, and room randomization method.
- Miranda: Designed and implemented the game state, created player, object and enemy classes and created functions to generate movement.

Closing Remarks

Our design mostly stayed the same throughout, the only real deviation is we found that we needed a piping interface to send message between the threads in the server because of the multiple threads we couldn't directly send message between players over the server.One aspect which we hadn't anticipated was that after gameplay the player could not play again without killing the server because of our design and handling of threading. Brandon and Megan worked together to come up with a wrapper to handle resetting the game state which could called internally by the threads upon player death or game termination. We found when we did have unexpected issues working together to pair program often lead to better solutions because we each were experts in our own section of the program but often didn't know the other aspects of the design as well and by working together we could leverage the groups knowledge to find a solution.

One of the hardest aspects of this program was debugging the interactions between players and designing a thread safe way of allowing players to share the movement of one object. There were several design iterations and once completed we did many careful play throughs to ensure that the players notifications of the game state changes were constant. Ensuring that information was being passed smoothly end to end in this program required all of us carefully discussing each aspect of our design and making sure that our interfaces were transparent. The multiplayer aspect of this project forced us all to more carefully examine our design decisions as it was a necessity that all of the components operated in a non-blocking asynchronous fashion, a paradigm that has been uncommon in work for other classes. Overall this plan was executed smoothly with few deviations and only minor surprises.