**Design Document II**

**<** Space Crucible**>**

**REVISION HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision # | Author | Revision Date | Comments |
| 1.0 | Parth Patel | September 28, 2021 | initiated |
| 1.1 | Isaac Colon,  Meshwa Patel,  Yifan Zhang,  Kwadwo Gyasi-Danquah  Parth Patel | September 28, 2021 | Added server-side, client-side, and level editor data fields and methods |
| 2.0 | Kwadwo Gyasi-Danquah | November 22, 2021 | Initial revisions based on feedback |
| 2.1 | Parth Patel | November 23,2021 | Revise system overview, component description, class diagrams, use cases, state diagrams  Add some class’s data fields and methods |
| 2.2 | Isaac Colon | November 28, 2021 | Finish API writeups |

**Table of Contents**

[System Overview 4](#_Toc89033407)

[General Requirements 6](#_Toc89033408)

[Component Descriptions and Interfaces 6](#_Toc89033409)

[Class Diagrams for Components 8](#_Toc89033410)

## System Overview

Space Crucible is a two-dimensional, top-down perspective action-puzzle game with support for both single and multiple players. Levels will be defined as a series of square tiles on a fixed grid- however, player and monster movement will not be locked to discrete points on this grid. The combat will be in real-time and take place on tile-based levels. Space Crucible will have a science fiction theme and will emphasize run-and-gun combat style. Players take on the role of “Exterminators” and must use a variety of tactics to combat foes, from weaponry to coercing them to fight each other.

The objective is to explore the map while eliminating monsters, dodging traps, and finding the exit to proceed to the next level. Multiplayer is a key feature; levels can be designed that require more than one player to complete (in fact, the multiplayer-oriented level design will be prioritized). A level editor will be included so users can create their scenarios and save them in a simple text-based format. Levels can be compiled into level packs using a simple archive format called a “.WAD”, which contains and organizes any necessary graphics, sounds, music, and monster code (defined in scripts, rather than hard-coded Java).

Players will receive a launcher upon starting the game. The launcher will allow the user to pick a level pack, a specific level from the pack, choose a difficulty, and either launch a single-player session or join a multiplayer session by inputting the lobby code in the join lobby menu. Real-time action will use WASD or arrow keys to move the player, while the player can simultaneously aim using the mouse. Certain map tiles or objects can be interacted with to proceed in the level or trigger traps. An in-game chat will allow players to communicate with each other. A straightforward light system will allow tiles far from light “sources” to darken, obscuring important puzzle components or hiding sneak attacks. MIDI files will be used as background music.

There will be a Master server that will handle all clients connecting to the multiplayer game mode. Once a player has selected create a lobby or join a lobby option, they will be directed to a specific game server. Each lobby will have a game server that will handle all incoming connections and data from the clients. In multiplayer mode, the game server controls the whole game to ensure that all players are in sync and the host does not have an advantage over other players. The clients are only responsible for sending their input data to the game server; the game server handles moving the players and sending the updated rendering data back to the clients.

Players will be able to host their own lobbies by selecting Create Lobby option in the co-op mode menu. Each lobby will have its unique 4-digit code that the host can share with other players to invite them to the lobby. The 4-digit code is linked with the lobby’s IP address and port number. When the client makes a request to create a lobby, the master server sends the lobby info to the client, and then the client joins the lobby. The players who want to join an existing lobby will choose the Join Lobby option and input the unique 4-digit code. The join lobby request is sent to the master server, and the master server finds the lobby associated with the code and sends the lobby details back to the client. The lobby host will have access to the difficulty selection menu and start game option that other players won’t be able to see. Once the host selects the start game option, the level beings for all players in the lobby. If the host leaves the lobby before starting a game, the host privileges are passed on to the player who joined second.

Once the lobby host starts the level, new players can still join the ongoing level as long as they have the lobby code. This feature also allows players to leave and join the ongoing game as they wish. Another feature we implemented is if a player leaves in an ongoing game, the player will be replaced by a bot/AI player who will help the players finish the level. The bot player can follow the closet player and shoot and chase the visible enemies.

Server administrators can use the remote control (RCON) application to monitor the servers and send instructions to perform certain tasks. The RCON client can log into both the master server and the game servers using a lobby code (or MASTER for the master server) and a password (the master server’s host chooses the master password, the game server password is generated randomly upon lobby creation). When RCON is connected to the master server, it can monitor the number of game servers the master is hosting and check their lobby codes, RCON passwords, and add-on files. When connected to a game server, player positions and health and packet information can be monitored in real-time, chat can be sent to the players from the server, and settings such as game skill and level can be changed.

The program will be written in Java and will support all desktop operating systems (i.e., Windows, macOS, and Linux). There will be separate executable JAR files for the master and game servers, client programs.

A Java IDE will be necessary to develop this project, preferably a common one shared by all developers. The Java LibGDX library is a game development library that includes graphic, sound, music, and networking functionality and will be the framework we use to develop the game. An open-source library, DoomStruct, exists to manipulate .WAD files, which will be used to access game data neatly in an archive format. The KryoNet library is used for network operations, exchanging packets between the game client and the master and game servers over the TCP protocol.

## 

## General Requirements

* Desktop or Laptop running Windows, Mac, or Linux operating systems
* Master and Game Servers will run on a Linux machine (headless, desktop or laptop)
* Java runtime environment
* Keyboard and mouse to control in-game movement
* Optional - Internet connection to access multiplayer mode

## Component Descriptions and Interfaces

Space Crucibles comprises of three high-level components: client application, master server, and game server.

**Client Application:** The game features two modes, single-player and multiplayer. In the single-player mode, the client application by itself is enough to play the game. In multiplayer mode, the user running the game will need an internet connection to connect to both the master server and game server to play the game.

In the single-player mode, the client application controls the game logic, renders the game data, and handles client input. This way, the client can play the game without having access to an internet connection.

In the co-op mode, the multiplayer feature, the client application will be responsible for connecting and sending input data to the servers and rendering the data that the server sends back.

When the player selects the co-op option in the main menu, they will be able to join/create a lobby where they will wait until the lobby host starts the game. If the user creates a lobby, they will become the lobby host and have access to start the game and change difficulty options. Once the host starts the game, all players will spawn and move using WASD and aim and shoot using the mouse and left click, respectively. The player can attack the AI monster until its health goes down to zero and move forward. If the player’s health reduces to zero, the player sets to the beginning of the level. The player can clear the level once they have passed all the obstacles and reached the endpoint. In the settings, the player can control the sound and audio. If required, they can turn it down to zero or adjust it accordingly. The game will also have a text chat where the players will communicate with each other. After the game starts, the AI monsters, controlled by the server, will attack the player while they are finishing the level. WASD or arrow keys are used to move the character. The player can aim using the mouse towards the AI monster.

Users can create their level packs using a level editor. Users' new levels will be compiled into level packs using an archive format- ‘.WAD’. To play the user-created levels, they can load them using the addons option inside the settings menu. Once the level is loaded, the user can either go into single-player and play the game or go to multiplayer and play the level. When the invited players try to join the lobby with a custom level loaded, the server automatically downloads the level files from the host and loads them to allow the user to enter the lobby.

**Master Server:** The master server will be hosted on a Linux machine. We will be using KryoNet, a Java networking library. It will provide efficient TCP client/server network communication using NIO. There will be only one instance of the master server running at all times. The master server will be responsible for keeping track of all game servers and redirecting all clients to the game servers. When a game server is no longer being used, it will ping the master server, and the master server will be able to reuse it to create another lobby.

**Game Server:** The game servers will also be hosted on a Linux machine, and they will be created using the Kryonet library. We will create as many instances of the game servers as needed. The game server will control the game logic, keep all players in sync, handle input data by updating the game, and send the render data back to all clients. The game server will be the brains of the game in multiplayer and control everything to ensure everything stays in sync and fair for all players. The Game Logic runs at approximately 55 ticks per second. At each tick, we will send the player’s x and y coordinates to the server and update the timer if the player has moved.

## Class Diagrams for Components

**A screenshot of a computer program

Description automatically generated with low confidence**

**Figure 1**: Master Server Class Diagram

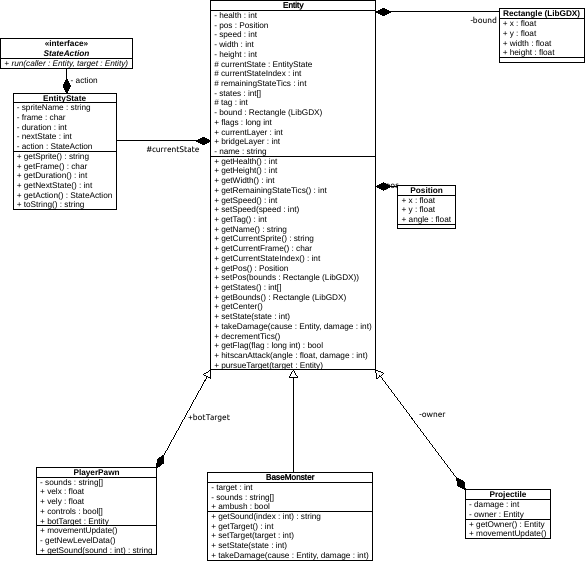
**Figure 1**shows how the MasterServer class communicates with the clients. The MasterServer is a class that represents a server; in Kryonet, a server has a threaded listener, and the listener has its own connected, received, and disconnected methods, so whenever a client connects, sends data, or disconnects, the listener handles it. So, when the user wants to access coop mode, a new SpaceClient is created, and that class tries to connect to the MasterServer. If the SpaceClient is able to connect, then the client can access the co-op mode menu; otherwise, an error is shown to the client. The MasterServer class also has a client/server relationship with the game server, where the master server plays the role of server, and the game server plays the role of a client. Whenever a new SpaceServer connects to the MasterServer, it records the id and port number in a HashMap. So, whenever a client requests to create a lobby, the MasterServer checks if a SpaceServer is available to be used. If one is found, the MasterServer sends the server details to the client. When a client tries to join a lobby using a lobby code, the master server checks if that lobby code corresponds to any active SpaceServer, and if it does, the MasterServer sends the details of that SpaceServer to the client. When all players leave the SpaceServer, the SpaceServer pings the MasterServer that it’s empty, so the MasterServer can make it available to use again.

A screenshot of a computer program

Description automatically generated with low confidence

**Figure 2**: SpaceServer Class Diagram

**Figure 2**shows how the SpaceServer class coordinates with the client and game logic to run the game. SpaceServer has two roles: a server that SpaceClients connect to and a client that connects to the MasterServer. When clients send requests to create or join a lobby, the MasterServer sends the details to connect to a SpaceServer. When the SpaceClient connects to the SpaceServer, they change the screen from TitleScreen to LobbyScreen, where they can see all of the players that have joined so far. When the lobby host starts the game, SpaceClient sends a ping to the SpaceServer to start the GameLogic loop and spawn all the players in the game. On the client-side, they switch from the LobbyScreen to the GameScreen. The GameLogic is controlled only by the SpaceServer, so when clients send input to the SpaceServer, it updates the player’s coordinates and sends the render data back to the SpaceClient for rendering. This cycle continues until the game is over or all players leave the server. Once all players have left the SpaceServer, it sends a ping to the master server that it's empty and ready to be reused.



**Figure 3**: Game Object Class Diagram

**Figure 3** shows the basic game object class hierarchy. An “Entity” is any map object that represents an active creature or person. These are split mostly between monsters, which will inherit from an abstract “BaseMonster” class, and players, who are represented by the PlayerPawn class. Both contain basic information such as health, position, speed, size, what graphics they use, and what angle they are facing. The abstract BaseMonster class defines methods which any monsters in-game will need to override in order to have functionality- for example, idle() represents an enemy waiting to see players, chase() is called when the monsters are pursuing their target, and so on. Target is a pointer to an Entity existing on the map that the monster is chasing. This may not necessarily be a player. The PlayerPawn class contains information such as which player client the entity is representing (via playerNumber), what keys, ammo, and weapons are currently in the player’s possession, and the methods called when the PlayerPawn is moving, using something on the map, attacking, or dying. Other classes include Item, the base class for anything that can be picked up by players, and projectiles, which contain a pointer to whatever Entity launched it (for example, so a player can take the credit for a kill).

**Master Server (MasterServer):**

The master server is the controller for all the other game servers. Whenever a client wants to create or join a lobby, it must first interact with the master server to make sure there is room for a new lobby (if they wish to host a new lobby), or that the lobby they want to join exists (if they wish to join someone else’s lobby).

**Attributes:**

* **Server** server
  + A KryoNet Server object that represents that server instance itself. This object is used to send and receive data from clients and game servers.
* **HashMap<String, ServerEntry>** servers
  + A map of each server's port, list of add-on files and their hashes, their RCON access password, and their IP address to the server’s lobby code.
* **HashMap<String, Integer>** ports
  + A map of each server (by lobby code) to its port
* **ArrayList<Integer>** availablePorts
  + A list of available server ports remaining, according to the port range given in the launch parameters.
* **HashSet<Integer>** rconConnections

A set of all connection IDs which belong to remote control clients (as opposed to game clients)**Methods:**

* **public MasterServer(int minPort, int maxPort, String password)**
  + Constructor for the master server. Adds every game server on its local IP from the minimum port to the maximum port as an available server. Initializes the KryoNet server, allowing game clients to connect and create or join lobbies on available game servers within the port range.
  + No return value, as it is a constructor.
  + Will halt program if there is an IOException (caused by Kryonet connection/bind/packet code)
* **public String createRandomLobbyCode()**
  + Creates a random four-character join code for a lobby.
  + Returns code as a String
* **public Boolean isPortAvailable(int port)**
  + Checks if given port is available for use in a new lobby
  + Returns whether the above condition is true
* **public void handleRCON(String message)**
  + Precondition: At least one RCON client must be running.
  + Parses an RCON message into a command and performs one of many actions depending on the command and given parameters.
  + If command is parsed correctly, send a reply message
  + If command is unparsable, gracefully ignore
* **public void sendRCON**

**Game Server (SpaceServer):**

The game server, represented by the SpaceServer class, is an instance of a multiplayer game. Before the game starts, this server’s’ clients will remain in lobby mode, but once the host starts the game, the clients will begin the actual multiplayer game.

**Attributes:**

* **private Server** server
  + Just like the master server, the core functionality of the SpaceServer relies on a KryoNet Server object.
* **private Client** serverClient
  + The game server also contains a client to the master server, in order to keep the latter up to date with its status
* **public ClientData** clientData
  + Contains the ids, player numbers, and names of all clients on the server. This object can be sent as-is as a packet to clients as needed.
* **public HashSet<Integer>** connected
  + Contains the set of connection IDs representing all connected **player** clients.
* **public HashSet<Integer>** rconConnected
  + Contains the set of connection IDs representing all connected **RCON** clients
* **public HashSet<Integer>** disconnected
  + Contains the set of connection IDs representing all disconnected clients
* **public List<Integer>** idToPlayerNum
  + Contains list of player clients’ connection IDs. The index in the list will represent the playernumber of the client with the ID stored at that value.
* **public HashMap<Integer, String>** ips
  + The ip of each client, mapped to their connectionID
* **private boolean** gameStartedByHost
  + True if game if game is running, false if still in lobby mode
* **public Date** startTime
  + The starting time for the server as a Java Date object
* **public long** packetsReceived
  + The total lifetime packets received, used for monitoring
* **public long** packetsSent
  + The total lifetime packets sent, used for monitoring
* **public volatile AtomicInteger** packetsReceivedLastSecond
  + Packets received in the last second, used for live monitoring. It is volatile because a Timer object users this information exactly every second and resets the value to zero when it does so, so atomic operations must be used.
* **public volatile AtomicInteger** packetsSentLastSecond
  + Packets sent in the last second, also volatile and uses atomic operations
* **private Timer** packetTimer
  + Sends live server monitoring data to remote control every second
* **private File** serverReport
  + Server log file, which records server status over time until the server closes.
* **private Writer** fileWriter
  + Writer class used to write to server log
* **private Thread** gameLoop
  + The Thread responsible for running the GameLogic timer

**Methods:**

* **public SpaceServer(int tcpPort)**
  + Precondition: port is open on the network and is unused, so a new server can be created.
  + Sets up serverside GameLogic, creates server report files, creates KryoNet server object to connect with clients, schedules per-second packet data and player info reports for administrators.
  + There is no return type, as this is a constructor.
  + Postcondition: a game server is set up and ready to host when it receives an order from the master server
  + If log file cannot be created (IOException), then it is gracefully ignored.
* **private void updatePackets()**
  + Precondition: Server is up and running
  + This method is scheduled to run every second. It sends the average packets sent and received per second for the server’s runtime to all RCON clients, as well as the packets sent and received in the last second.
  + The method also logs this information in the server’s log file, unless the file is not accessible (IOException)
* **private void createGameLoopThread()**
  + Precondition: Server constructor has been called
  + Creates a new Thread object which will start the GameLogic loop when run.
  + Postcondition: lobby host is now able to start the game
* **handleRCON and sendToRCON work the same as their equivalents in the master server**

**Game Logic (GameLogic):**

The GameLogic class is a class full of static methods and attributes and is responsible for running the game itself. Both server and client executables have this class, and only one is running its methods at a time. A constantly running Java Timer is responsible for updating the game every “tic”, a unit of game time equivalent to 18 milliseconds.

**Attributes:**

* **private Timer** gameTimer
  + Timer responsible for updating the game every Tic. This includes updating monster AI, player movement, level scripts, and player bot AI. Essentially, this is the game clock in both single- and multi-player modes.
* **public Boolean** isSinglePlayer
  + True if singleplayer (running locally), false if coop (running server-side)
* **public ArrayList<Entity>** entityList
  + List of every Entity active in the game at the moment.
* **public Queue<Entity>** newEntityQueue
  + Entities queued for addition to the game world. This must be done in between game tics, or concurrent modification errors can occur, which is why they are queued before they are added.
* **public Queue<Entity>** deleteEntityQueue
  + Similarly, these are Entities queued for removal from the game world, and these operations must also be done between tics.
* **public LinkedList<EntityState>** stateList
  + The list of all animation frames used by every Entity. These are loaded from ENTITIES scripts at runtime, from the resource .WAD and any add-ons.
* **public Map<String, EntitySpawner>** entityTable
  + The EntitySpawner object is used whenever an Entity of that type needs to be added to the game, either at map load, on projectile launch, or a scripted spawn event. The String is the name of the Entity to be spawned.
* **public Map<Integer, EntitySpawner>** mapIDTable
  + The mapID is a number ID used to identify Entities in level data. This ID can also be used to request a specific EntitySpawner.
* **public Map<Integer, LevelData>** levels
  + A map of each level’s data to that level’s number
* **public ArrayList<TileAction>** effectList
  + List of scipted effects that can be placed on map tiles (I.e. when you cross into a certain room/area)
* **public LevelData** currentLevel
  + The currently loaded level
* **public LevelData** nextLevel
  + The next level to be loaded. Used when in the process of switching levels.
* **public Boolean** switchingLevels
  + True if currently switching levels
* **public Map<Integer, Queue<ScriptCommand>>** scripts
  + Map of scripts to script numbers. Scripts are stored as a queue of commands, which are essentially TileActions with optional time delays (measured in tics)
* **public LinkedList<LevelScript>** runningScripts
  + These are scripts running right now. They are updated every tic in the game loop until the script is over. A LevelScript is given a queue of commands from the scripts map, and it runs through each entry until the command queue is empty.
* **public Queue<LevelScript>** newScriptQueue
  + Like Entities, scripts need to be queued for addition to the runningScripts list, or concurrency errors may occur.
* **public Queue<LevelScript>** deleteScriptQueue
  + Likewise, LevelScripts must also be queued for deletion once their command queue is emptied
* **public Server** server
  + The raw KryoNet server object, which can be used to communicate with players and RCON. This is null in single-player.
* **public SpaceServer** spaceServer
  + The wrapper class for the server (see above).
* **public Boolean** goingToNextLevel
  + True if level transistion is occurring. This is used to halt the game loop before setting switchingLevels to true, signalling that the game is not being updated right now.
* **public Integer** ticCounter
  + Total tics elapsed during game
* **public Integer** difficulty
  + Current difficulty level of the game

**Methods:**

* **public void start()**
  + Begins the game and starts the MIDI music system, either locally (in single-player) or by sending a MIDI data packet to clients (in multi-player)
* **public void stop()**
  + Stops and removes the game timer, effectively halting game logic
* **public void gameTick()**
  + Precondition: GameLogic timer has been started (only the timer can call this method)
  + Updates all of the moving parts on the map- mostly Entities, but also level scripts.
  + Iterates through the entityList and calls decrementTics on each Entity, updating their AI logic by one tic.
  + If the entity is a player, it either moves the player according to the player’s current input, or updates bot AI if the player is disconnected (meaning that the player Entity is AI controlled)
  + If the player bot’s target is null for whatever reason, (NullPointerException) it simply stops the bot’s AI for the tic, and the bot will continue in the next tic with its new acquired target
  + If, for whatever reason, the game tries to access a player with an Id that does not exist (IndexOutOfBoundsException- idToPlayerNum list), then the game will skip updating that player
  + Next, all level scripts are updated. For each script, if the executed command was the last in the script, queue script for removal.
  + After the Entities and scripts are updated, the game adds or removes any level scripts and Entities queued for removal or addition. By only adding or removing Entites and scripts after their respective lists are iterated through, we avoid ConcurrentModificationExceptions
  + Next, if the game is multiplayer, relavent game rendering information is sent to all clients, so their screen can draw the action
  + Finally, if a level change event has been activated, the the game prepares to transition levels.
  + Otherwise, the next game tic is scheduled on the timer. To ensure the game runs at a constant rate, the execution time of the method itself is subtracted from the tic duration.
  + If the timer has been stopped (IllegalStateException when trying to schedule), then the exception is caught and next tic is simply not scheduled.
* **public void loadEntities(LevelData level, Boolean editor)**
  + Precondition: level is a LevelData object representing a valid, playable game level, which has been read from a LEVEL# lump in a .WAD file.
  + Clears current entityList and loads new Entities generated from the level’s list of LevelObjects.
  + If in level editor mode, load all Entities from all difficulty levels, and all players.
  + Else, load only entities from current difficulty, and as many players as are currently connected (or just one, in single-player)
  + If the Entity type specified by the LevelObject is unknown, catch the exception and simply do not spawn the object.
  + Postcondition: all Entities on the map for a given difficulty level have been loa, and the game is ready for action.
* **public void loadLevels(Array<WadFile> wads)**
  + Precondition: user has at least one .WAD (resource.wad) loaded, plus any add-ons.
  + Iterates through every wad file, looks for levels (LEVEL#), and adds them to the GameLogic’s level map, mapped to the level number specified in the lump name (I.e. LEVEL9 is mapped to 9). If more than one .WAD has a level in a given slot, the level in the last .WAD loaded occupies the slot.
  + If constructing a new LevelData from a .WAD’s level data fails for whatever reason (IOException), skip loading that level
  + Postcondition: all game levels are loaded and can be accessed on demand
* **public void changeLevel(LevelData level)**
  + Precondition: level is a LevelData object representing a valid, playable game level, which has been read from a LEVEL# lump in a .WAD file.
  + Replaces GameLogic’s currentLevel, updates MIDI system with new level’s music, and calls loadEntities using the given level.
  + When finished, gameTick calls are scheduled as normal on the timer.
  + Postcondition: the game is now continued with a new level.
* **public PlayerPawn getPlayer(Integer tag)**
  + Simply returns the player with the given player number. If that player doesn’t exist, return null.
  + If a ConcurrentModificationException is caught (I.e. the entityList is being iterated through), simply call the method again, until the entityList is free)
* **public void readyChangeLevel(LevelData newLevelData)**
  + Precondition: newLevelData is a valid level loaded from a .WAD
  + Simply prepare the GameLogic for a level switch at the end of the next tic by signalling that a level change event has occurred
  + Postcondition: on the next tic, the game timer will stop and level transition will occur (see changeLevel)
* **public ArrayList<Entity> entitiesInsideView(Connection c)**
  + Precondition: game is in multiplayer mode, game loop is running, and c is a PlayerConnection object representing a connected player client who needs to draw the game scene
  + Using clientside camera information received from the client, check which Entities currently active in game are actually visible from the player’s perspective.
  + Also, always add the player’s own PlayerPawn entity no matter what.
  + If a NullPointerException occurs, send the client’s Pcatch it and be sure to at least send the client’s own PlayerPawn
  + Postcondition: the client receives a list of Entities to draw, while minimizing packet size on the server’s side.
* **public void playerServerSound(String name)**
  + Simply sends a packet to any connected player clients instructing them to play the given sound
* **public void alertMonsters(Entity soundSource)**
  + Wakes up any monsters in a given radius and sets their target to the soundSource Entity (unless the monster is in ambush mode)
* **public void setPlayerBotTarget(PlayerPawn p)**
  + Calculates the closest monster or player for an AI-controlled player to follow

**Client (SpaceClient):**

The client, represented by the SpaceClient class, receives regular updates from the game server and is responsible for updating the player’s game and keeping it in sync with the server. The client receives a specific, small amount of information from the server needed to run the game- mostly instructions on what parts of the game world to draw from the client’s perspective- and sends only the player input from each player.

**Attributes:**

* **private Client** masterClient
  + The KryoNet Client object for interacting with the master server
* **private Client** gameClient
  + The KryoNet Client object for interacting with the game server
* **private GameScreen** screen
  + The client's game screen, where the game itself is drawn
* **public ValidLobby** validLobby
  + Object received upon lobby connection attempt, contains a true or false value for validity, and a reason for failure if false
* **private StartMenu** startMenu
  + The client’s main menu

**Methods:**

* **public SpaceClient(GameScreen screen, StartMenu startMenu)**
  + Precondition: Client is running the game and has accessed the co-op mode feature from the menu.
  + Creates the KryoNet client objects for the master server.
  + If master server is not running, simply stop the connection attempt, notify the player, and allow the player to continue as normal
  + No return value, as this is a constructor
* **public void createGameClient(Integer tcpPort)**
  + Precondition: client is connected to the master server and has received either a free port to create a lobby (host) or the port to a lobby that they wish to join (guest)
  + Creates a KryoNet client to interact with a game server. The game client is different from the master client, and the client program maintains a connection to both.
  + There is no connection to a game server with the given port for any reason, cancel the connection gracefully (IOException caught)
* **public void makeLobby()**
  + Precondition: client is connected to the master server, but not any game server yet (user is most likely in the co-op menu)
  + Send a packet to master server instructing it to create a new lobby. Send the file names and SHA hashes of all loaded add-ons so any guests can verify their files against the host’s
  + Postcondition: if the master server has free ports in its port range, a new lobby is created and the calling client is the designated host.
* **public void sendLobbyCode(String lCode)**
  + Precondition: client is connected to master server, but not any game server yet
  + Send lobby code to master server in order to log in to one of the open lobbies
  + Postcondition: if a lobby with the given code is running, connect client to that lobby
* **public void getInput(Boolean[] controls)**
  + Precondition: client is connected to a game server and gameplay is active
  + The controls array represents any input the player can send to the server- the four walking directions and the shoot command (true if button is pushed, false if not)
  + Send these commands to the server, which will use them to manipulate the client’s PlayerPawn avatar (I.e. the user controls their player)
  + Postcondition: the user’s player Entity on the server responds to the clientside input
* **public void getCameraData(CameraData cameradata)**
  + Precondition: cameradata is an object which contains the dimensions and position of the player’s game screen camera. The player is presumably engaged in gameplay on the game server.
  + Send this camera data as a packet to the game server.
  + Postcondition: the server will take this data and respond with a list of Entities for the player to draw. Only the Entities which should be visible in these camera dimensions will be drawn.
* **public Client getGameClient() / getMasterClient()**
  + These methods are just getter methods for the KryoNet client objects
* **public void sendLevels() / sendEntities()**
  + Precondition: client is a lobby host, and is therefore connected to the master and game server. The game server is being prepared.
  + The host sends the LevelData and EntitySpawner objects for each level and entity, respectively, to the server in advance. This way the server does not need copies of the .WAD files
  + Postcondition: the server has access to all necessary levels and entities and can now begin gameplay on demand

**Title Screen(TitleScreen):**

**Attributes:**

* **public static boolean** changeScreen
  + Check if the screen should be updated.
* **public static boolean** update
  + Check if the files need to be loaded.
* **public MyGDxTest** myGDxTest
  + The main game object that runs the application
* **public GameScreen** gameScreen
  + A reference to the client game screen class
* **public LobbyScreen** lobbyScreen
  + A reference to the client’s lobby screen class
* **public static com.badlogic.gdx.scenes.scene2d.ui.Table** mainMenuTable
  + Holds all of the main menu buttons
* **public static com.badlogic.gdx.scenes.scene2d.ui.Table** coopMenuTable
  + Holds all of the coop menu buttons
* **public static com.badlogic.gdx.scenes.scene2d.ui.Table** joinLobbyTable
  + Holds all of the join lobby buttons

**Methods:**

* **public void show()**
  + Allows another screen to be displayed over the original, making it the current screen without disposing of the original. This allows the original screen to hold onto its resources, removing the need for displays to be rendered more than once, but prevents memory from being freed.
* **public void render​(float delta)**
  + Method called every frame to check and update what is currently being displayed on the screen. LibGDX’s game state manager organizes the presence of these displays allowing different presentations to run behind the scenes or on input from the user or game logic.
* **public void resize​(int width, int height)**
  + Method called when either of the dimensions of the window are changed. Typically, this method is where the update method for the camera and view port are held to scale displays on the current screen to appropriate ratios after they are rendered
* **public void pause()**
  + Method that prevents the screen from creating new frames. It essentially “freezes” the screen’s current state. Typically utilized when the screen is not active or visible. Screens are always paused before they are disposed.
* **public void resume()**
  + Unfreezes a screen in a paused state, allowing it to create new frames once again.
* **public void hide()**
  + Removes screen without disposing of its resources.
* **public void dispose()**
  + Method typically called before a screen terminates, allowing it to release all the resources it is holding and reallocate memory.
* **public void showPopup​(PopupWindow invalid\_lobby)**
  + Shows the error popup menu on screen.
* **private void createMenus()**
  + Creates all of the menus for the game
  + Handles an interruption exception for the submit lobby code button.

**Lobby Screen(LobbyScreen):**

* **public static int**difficultyLevelSelected
  + A static variable that keeps track of the difficulty the user has selected.
* **public boolean**startGame
  + Checks if the host has started the game and updates accordingly.
* **public int**playerNumber
  + Holds the player number that the player has in a multiplayer game.

**Methods:**

* **public void show()**
  + Allows another screen to be displayed over the original, making it the current screen without disposing of the original. This allows the original screen to hold onto its resources, removing the need for displays to be rendered more than once, but prevents memory from being freed.
* **public void render​(float delta)**
  + Method called every frame to check and update what is currently being displayed on the screen. LibGDX’s game state manager organizes the presence of these displays allowing different presentations to run behind the scenes or on input from the user or game logic.
* **public void resize​(int width, int height)**
  + Method called when either of the dimensions of the window are changed. Typically, this method is where the update method for the camera and view port are held to scale displays on the current screen to appropriate ratios after they are rendered
* **public void pause()**
  + Method that prevents the screen from creating new frames. It essentially “freezes” the screen’s current state. Typically utilized when the screen is not active or visible. Screens are always paused before they are disposed.
* **public void resume()**
  + Unfreezes a screen in a paused state, allowing it to create new frames once again.
* **public void hide()**
  + Removes screen without disposing of its resources.
* **public void dispose()**
  + Method typically called before a screen terminates, allowing it to release all the resources it is holding and reallocate memory.
* **public void updatePlayerNumber()**
  + Updates the player number if a new player joins or leaves the lobby.

**Game Screen(GameScreen):**

**Attributes:**

* **public SpaceClient** client
  + A reference to the client object class.
* **public int** playerNumber
  + The player number the clients are assigned to.
* **public boolean** isSinglePlayer
  + Check to see if the game is single-player or multi-player.
* **public int** updatePing
  + A self-made time that updates players ping every 50 intervals.

**Methods:**

* **public void show()**
  + Allows another screen to be displayed over the original, making it the current screen without disposing of the original. This allows the original screen to hold onto its resources, removing the need for displays to be rendered more than once, but prevents memory from being freed.
* **public void render​(float delta)**
  + Method called every frame to check and update what is currently being displayed on the screen. LibGDX’s game state manager organizes the presence of these displays allowing different presentations to run behind the scenes or on input from the user or game logic.
  + Handles a ConcurrentModificationException and NullPointerException when drawing items to the screen by ignoring it to not crash the client.
* **public void resize​(int width, int height)**
  + Method called when either of the dimensions of the window are changed. Typically, this method is where the update method for the camera and view port are held to scale displays on the current screen to appropriate ratios after they are rendered
* **public void pause()**
  + Method that prevents the screen from creating new frames. It essentially “freezes” the screen’s current state. Typically utilized when the screen is not active or visible. Screens are always paused before they are disposed.
* **public void resume()**
  + Unfreezes a screen in a paused state, allowing it to create new frames once again.
* **public void hide()**
  + Removes screen without disposing of its resources.
  + Handles a NullPointerException when stopping the Game Logic by ignoring it to avoid crash.
* **public void dispose()**
  + Method typically called before a screen terminates, allowing it to release all the resources it is holding and reallocate memory.
* **public void getAngle​(boolean isSinglePlayer)**
  + Find the angle that the user’s mouse is currently pointing at and find the in-game coordinates.
* **public float getAngle()**
  + Return the angle the user is facing.
* **public void setRenderData​(Network.RenderData object)**
  + Sets the render data to allow the user to load it.
* **public void setClientData​(Network.ClientData object)**
  + Sets the client data to receive information about other users.
* **public void setServerDetails​(Network.ServerDetails object)**
  + Sets the server details so the client can join the lobby.
* **public void addChatWindow()**
  + Adds a chat window to the screen.
* **public void addChatToWindow​(Network.ChatMessage chat)**
  + Adds chat message to the chat window.
* **public void setPing​(int returnTripTime)**
  + Sets the return trip time data to the ping variable.
* **private void drawMiniMap()**
  + Draws the minimap on the screen for the client.

**LevelEditor (EditorScreen):**

The level editor was formerly a separate application but has since been merged into the main game executable. The editor allows users to create new levels and save them into files which can then be loaded into the game proper, including multiplayer games. The level editor is essentially a version of the game with no game loop, and where the level author can add and remove Entities and level geometry on the fly.

**Attributes:**

* **public WadFile** file
  + Object representing the .WAD file which will contain the work-in-progerss map
* **public File** soloFile
  + If the level is being stored in a standalone text file, this File object is used instead of the .WAD file object
* **public Array<WadFile>** resources
  + Other .WADs besides the one containing the level, used to provide other resources. Usually the default resource.wad goes here, and add-on levels use its assets.
* **public LevelData** level
  + The level being modified, as a LevelData object
* **public Integer** levelnum
  + The level number of the current level
* **public Boolean** windowOpen
  + True if a popup window is open. This is used to block functionality until the popup is closed.
* **public Boolean** fullBright
  + Allows user to render in fullbright mode, viewing the level as if everything were max brightness. Useful for editing dark areas.
* **public CopiedTileData** copiedTileData
  + A level tile which has been copied using CTRL-C, and can be pasted later.
* **public CopitedThingData** copiedThingData
  + An Entity which has been copied using CTRL-C and can be pasted later.
* **private Boolean** dragging
  + True if player is moving an Entity around with the mouse
* **private LevelObject** dragThing
  + The object in question which is being dragged with the mouse
* **private Boolean** selecting
  + True if player is making a group selection with the mouse
* **private Boolean** selectThing
  + True if the player is selecting things. False if selecting tiles.
* **private Float** selectionX1/selectionY1/selectionX2/selectionY2
  + These four values represent the selected area when a player is making a group selection.
* **private Array<LevelObject>** selectedObjs **/ Array<LevelTile>** selectedTiles
  + These arrays represent the entities or level tiles contained within the selection area

**Methods:**

* **public EditorScreen()**
  + Simple constructor for the LibGDX screen, starts up camera and sprite/shape renderers.
  + No return value, as it is a constructor.
* **private void checkControls()**
  + Precondition: Level is loaded and open in the editor
  + Checks mouse controls and runs the left/middle/right click methods if any are pressed.
  + If the mouse has been released after a selection, finish making selection.
* **private void checkShortcuts()**
  + Precondition: Level is loaded and open in the editor
  + Checks shortcut key combinations, generally CTRL and/or SHIFT plus a letter key. If a shortcut has been activated, run that shortcut.
* **private void moveCamera()**
  + Precondition: Level is loaded and open in the editor
  + Check for movement keys (WASD or up/down/left/right) and move the camera if one is pressed.
* **private void editTilePrompt() or Integer tilex, Integer tiley)**
  + Precondition: Level is loaded and open in the editor
  + Open pop-up editor for tile at given x and y, or from the selected tiles list if there is an active selection (no-args version).
  + If there is no tile at (tilex, tiley), create one.
* **private void editThingPrompt(LevelObject obj, Entity e) or (Entity e)**
  + Precondition: Level is loaded and open in the editor
  + Open pop-up editor for the given level object or the list of selected level objects
* **public void openFilePrompt()**
  + Open file selection prompt to pick new .WAD or .LMP (single level) to open and edit. Also allow creation of new files.
* **public void openLevelPrompt()**
  + Open level selection prompt to pick a level within a .WAD or add a new level to the .WAD. This prompt is skipped if a single-level file is being edited.
* **public void loadLevel() or loadNewLevel(String name, Integer level)**
  + Loads the selected level from the .WAD or .LMP (loadLevel) or creates a new LevelData from scratch and loads that instead (loadNewLevel).
* **private Boolean checkForNoTextures()**
  + Precondition: a file has been selected for editing.
  + Makes sure that there is at least one texture loaded in the opened .WAD or the selected resource .WADs. If not, cancel level loading and make player choose resources with textures.
* **private void pasteTile(Integer tilex, Integer tiley) or pasteThing(Float x, Float y)**
  + Precondition: Level is open and a tile or thing is presumably copied into a copiedTileData or copiedThingData object.
  + At the given coordinates, create a new tile or thing using the information provided by copiedTileData/copiedThingData.
  + If the those objects are null, simply do not paste anything.
* **private Boolean isCtrlPressed() or isShiftPressed()**
  + Simply checks whether left or right CTRL/SHIFT are pressed, to make other methods neater.
* **private void leftClick(Float x, Float y, Integer tilex, Integer tiley)**
  + Precondition: Level is open in the editor, and the editor has detected that the player has clicked the left mouse button.
  + If the mouse was just pressed, either begin dragging a thing (if CTRL is not held) or begin making a selection (if CTRL is held). CTRL alone means select tiles, while CTRL-SHIFT means select objects.
* **private void middleClick (Float x, Float y, Integer tilex, Integer tiley)**
  + Precondition: Level is open in the editor, and the editor has detected that the player has clicked the middle mouse button.
  + If the mouse is over an entity, remove the entity. Otherwise, remove the tile that the mouse is over, if there is one.
* **private void rightClick(Float x, Float y, Integer tilex, Integer tiley)**
  + Precondition: Level is open in the editor, and the editor has detected that the player has clicked the right mouse button.
  + If there is an active selection, show the proper edit prompt for that selection
  + Otherwise, if shift is held, create a new entity and show an edit prompt for it
  + Otherwise, if an entity is present underneath the cursor, show an edit prompt for that entity
  + Otherwise, if there is a tile, show its edit prompt. If the cursor is over completely blank space, do nothing.
* **private void makeSelection()**
  + Precondition: Player has just finished a group selection (by letting go of left-mouse button)
  + Calculate the bounding box using the selection\*\* variables (see attributes) as coordinates.
  + Find all tiles or things (depending on selectThing, see attributes) in that area and add them to the list of selected tiles/things
  + Postcondition: the selected map tiles or entities will be edited in batch upon right click, until the selection is cleared.
* **private void drawSelectionBox()**
  + Precondition: Player has selected, or is in the process of selecting, a group of tiles or entities
  + Using the starting point as one corner and the mouse position (or ending point, if selection is finished) as the other, box which represents the selection area.
  + Draw in blue if selecting tiles or green if selecting entities