CSCI 265 Product Design

Team name: We Be Daves

Project/product name: See a Neevle, Hear a Neevle

Contact person

- Dave Narealdave, nareal@somewhere.ca

Table of Contents

List of Figures

H. Logical ERD

I. The grand design

1. Known issues/omissions

In this section we list any currently-known errors, omissions, or other problems with the rest of this design document.

1. The various module diagrams have undergone several changes since the original grand design diagram was produced, but the grand design diagram has not yet been updated to reflect those.
2. The requirements description of healing items implies the items heal some players but damage others, but never give clear rules on which is which. The assumption made in this document is that they heal players of the same energy base and harm those of different bases, but this needs to be confirmed and updated in the requirements.
3. Upon reaching an experience threshold, the requirements don't specify precisely how/when the player is supposed to be presented with the choice of which upgrade level they wish to apply.
4. In this document the lost-connection and game-over screens have been combined, which doesn't match the requirements. One or the other needs to be revised (team decision pending).
5. For the NPC guide creature in puzzle zone 2, the requirements provide nowhere near enough detail on the creature behaviour nor how/when it unlocks the appropriate gate. The description in this guide is based on best-guesses as to the desired behaviour.
6. In section 7.2.4, the specific list of ids for the possible items and ids for the nature of the change to be applied have not yet been developed.
7. In the ERD of section 7.4, 0-or-more cardinalities were not showing up correctly so are temporarily represented as 1-or-more.
8. In section 7, several of the items depicted in the ERD of 7.4 are not explicitly covered in the descriptions yet (overall game data, map zone data, and sound data).
9. The detailed object model for section 9.2 has yet to be developed.
10. The different sections of the document have not yet been extensively cross-checked for consistency, nor have they been carefully cross-checked against the requirements document, so there are almost certainly some contradictions between different sections and some possible requirements omissions.
11. There is some confusion over whether we'll need one thread/task per connection, currently it is believed we can use one thread per socket and (host side) pool multiple connections on the socket.

2. Product overview

In this document we discuss the design of the game **See a Neevle, Hear a Neevle** (SaNHaN hereafter). While the game details are discussed in the Requirements document, here we introduce the core features from an overall design perspective. Note that some of these features are expected to have particularly far-reaching impacts on our overall design, and these will be discussed in greater detail in the subsequent section (Core design influences).

The vision for this game is to have one player setting up and 'hosting' the game on their machine and a number

(currently 5) of other players joining by running (their copies of) the game on their own machines and providing some suitable set of connection choices/information. The players are divided into two teams, dropped into a game world (viewed top-down as a 2D map) in which each player controls their own character: moving about the map, interacting 'in real time' with one another and with other creatures, items, terrain, puzzles, and obstacles.

There are a number of game-world features and mechanics whose design (and eventual implementation) are expected to be relatively straight-forward, if detailed. While not exhaustive, the following list provides a rundown of the key points:

- storage/updates/visuals for each player's connection information (ip, port, etc)
- storage/updates/visuals for each player's in game character (team, name, symbol, colour, stats, location, etc)
- storage/updates/visuals for each NPC creature and interactable item
- storage/updates/visuals for the overall game map, the active zone(s), and each player's viewable window
- detection of player actions (movement, interactions, chat, projectiles)
- collision detection (between characters, projectiles, items, obstacles, etc)
- combat resolution (contact between energy types, projectiles, etc)
- sound representation (on-map and in subtitle windows)
- control of all NPC creature actions in the active map zone(s)
- controlling/updating the behaviour of all puzzles/triggers in the active map zone(s)
- handling of the level-up system for players
- setup to host/join a game and detecting/handling of lost/dropped connections
- detection/handling of game-over situations (won and lost)
- handling of all the game menu systems, options, and navigation (main menu, host game, join game, main gameplay screen, help menu, options menu, game-over screen, connection-lost screen)

The aspects of the game that are expected to be most problematic from an overall design perspective are that the game involves multiple players connecting with one another from their own machines and interacting with each other and the game world 'in real time'. This means the design must carefully plan its network connections and communications, and must ensure that all actions and events are quickly detected and handled, and that updates to the game world happen simultaneously across all players' games. These are non-trivial problems in game design, and are discussed specifically in the next section (Core design influences).

3. Core design influences

There are three core facets of the system we discuss here:

- an overall design methodology,
- the design implications of a multiplayer/networked game,
- the design implications surrounding parallel tasks or threads.

In terms of an overall design methodology, we have chosen a top-down object-oriented approach. Both aspects are well understood and widely used and accepted, and any major programming language or game engine we may choose to use for implementation will almost certainly provide good programming support for these choices.

As mentioned previously, two of the biggest overall design problems the developers are faced with for SaNHaN stem from the following two features:

- it is a multiplayer game with each player running their own copy (i.e. not local coi-op), with one 'host' player creating a session and the other players joining it,
- each of the players is controlling their own character simultaneously: moving about the map and interacting with each other and the other elements of the game world in real time *but in a shared game world*, so the actions and effects should be echoed across all the players' games simultaneously.

With respect to the first problem above, the design has to choose a networking methodology. After a bit of research, the team has found there are two main methodologies used in multiplayer gaming: connection-based and connectionless.

The connection-based (TCP) methodology involves each client player establishing a fixed connection with the host for the duration of the game. Each message sent across the connection is guaranteed to arrive intact (as long as the connection is maintained) and all messages are received in the same order as they were sent across the connection. This provides a more coherent communication pattern than will be seen in the connectionless version, but involves more network overhead to set up and maintain the connection.

The connectionless (UDP) methodology has no fixed connection between the host and clients, instead each machine simply sends messages to others as desired. This involves lower overhead from a network perspective, but there is no guarantee that messages will arrive in the order they were sent, or even that they will arrive at all. It is then up to the product (the game in this case) to somehow detect and handle cases where messages didn't arrive or arrived out of order.

Fortunately, both methodologies appear to be well supported by all of the common programming languages (e.g. Python, C++, Java, etc) and game development platforms (e.g. Unity, UnReal), so the team's design choice here will not overly restrict later implementation choices.

The team has agreed that, based on our lack of experience in the area, dealing with the coordination/recovery mechanisms we would need in a connectionless system might complicate our design and implementation to the point where we could no longer get a working product completed this term. As a result, *a connection-based methodology has been chosen*. The implication for this is that our design needs to account for dropped connections. If a client player loses their connection with the host then our decision is that the player will simply be treated by the host as dissipated (aka dead), while if the host drops (loses their connections with everyone) then effectively the game ends for everyone. A later version of the product could consider some kind of recovery mechanism that allows a different player to take over as host, or a means of keeping the game going with the dropped player(s) controlled as NPCs.

With respect to the second listed problem, the design has to establish a scheme for detecting player actions and game world events as quickly as possible, and ensuring that the information is shared across all the players' machines as quickly as possible. Any failures here could result in the game appearing to lag or appear choppy, or (even worse) to have the game world to become out-of-synch across the different players' machines. An example of the latter would be if player X shoots at player Y at almost the exact same time as player Y attempts to dodge, and (say due to synchronization issues) on player X's machine it looks like their shot hit player Y but on the other players' machines it looks like player Y dodged out of the way in time.

To address this, our design puts a heavy emphasis on having one single machine determine the impact/results of game actions (the host machine), but to have multiple parallel processes running on each machine with a careful plan for communicating events and updates between them. Having such parallel processes coordinate with one another without ever blocking or interfering with one another is a challenging but well-researched problem in gaming (and other areas), and we have attempted to pattern our design in a way consistent with that widely described in the field.

Fortunately, as with the connection methodology, the most likely choices of programming languages and game

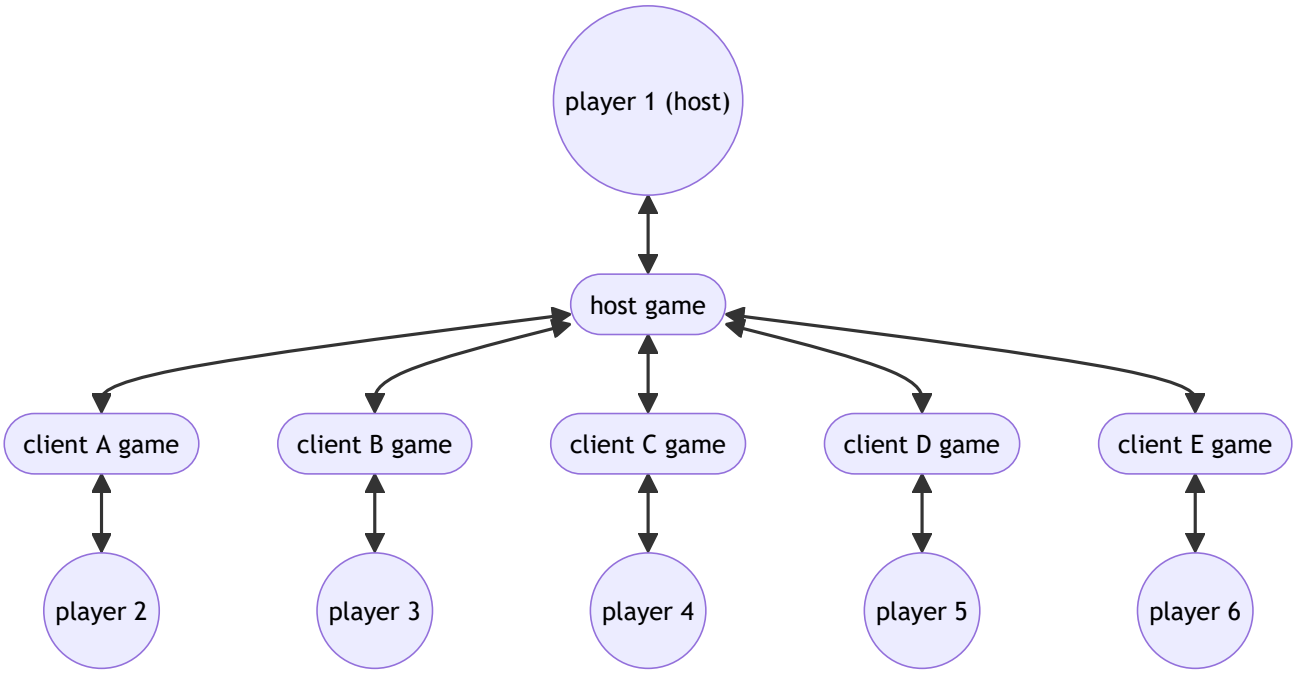
engines all provide mechanisms for event detection and for coordinating processes running in parallel. Threads are supported in Python, C++, Java, Unity, and UnReal, and both Unity and UnReal also provide some thread alternatives (e.g. tasks and runnables). This means that again our high level design can proceed without (too much) fear that we will be overly inhibiting our later implementation choices.

4. System context

Based on a six player game with player 1 acting as host and players 2-6 having joined as 'clients', the diagram below shows the six players each running their own copy of the game and the communication/interaction channels between them.

The interactions between the game instances and the players, as well as the interactions between the host and clients, will be discussed in the next section and depicted in the architectural design diagram (DFD 1).

Context diagram



In the next section (5 Architectural design) we will dive into the actual decomposition of our system into a coordinated collection of modules, with detailed discussions of each individual module providedin section 6. Details on the data design of the system will then be covered in section 7.

5. Architectural design

Our preliminary system design decomposes the overall system into four core modules:

- the game control module (Controller)
- the sender module (Sender)
- the receiver module (Receiver)
- the UIX module (UIX)

In this section we outline the key functionality associated with each module and how the four modules interact with each other, with the player, and with the network connection to the other players. Much more detailed discussion of each individual module is provided in section 6 of this document.

The game control module

For the host, this module is responsible for handling the actual in-play game logic, including:

- responding to user commands (mouse clicks, keyboard input, etc), which it obtains through requests to the queue of received messages (maintained by the receiver module),
- initial configuration of the game settings and requesting the relevant connections,
- controlling the NPC characters currently active,
- updating all stored information for characters, NPCs, puzzles, and items based on the current actions (and interactions) of the players and NPCs,
- determining when the game has ended (either due to failed connections or to one team winning or both teams losing) and calculating/providing final end-of-game information to be used in the final display screens.

For the client players, on the other hand, this module largely updates the local game data based on the information sent to it from the host.

Its communication with each of the other components is as listed below:

- **UIX module:** the controller tells the UIX when it is time to switch to the regular gameplay display and when it is time to switch to the game over display, and during regular gameplay it sends a steady stream of display updates.
- **Sender module:** as host, the controller sends a steady stream of messages that need to be sent to all the players, with updates on the state of the game (including the initial setup, configuration information, gameplay updates, and eventual endgame updates). As a client player, on the other hand, the controller should only need to send a message to the host if an anomaly has been found (e.g. if messages received from the host contain a logical contradiction with the local game state).
- **Receiver module:** the controller receives the sequence of incoming player action messages from the other player machines (if this is the host machine) or game update messages from the host (if this is a client player).

The sender module

As host, this module is responsible for setting up (and eventually shutting down) the necessary connections to support sending data to each of the other players. (At the time of writing it is believed it will need a seperate connection for each other player.) As client, the module is responsible for handling the client side of the connection setup (and shutdown) for sending messages to the host. The module maintains a queue of all messages waiting to be sent, removing messages from the queue one at a time (in the order they were queued) and sending them to the relevant recipients. The module will also be responsible for notifying the controller if a connection is dropped unexpectedly (e.g. by enqueuing a suitable message into the receiver's message queue).

Aside from the messages actually sent to other player machines over the established connections, the bulk of the sender's communications with other modules consists of the controller or UIX modules providing the sender with messages to be sent out (with such messages being queued until it is their turn to be sent).

There is also an extra form of possible outgoing communication from the sender module, described as follows. When

the sender is running on the host and receives a game update message to be sent to all players, it sends the message to the five client players normally, but for the local (host) player it enqueues the message directly into the receiver module message queue. (This is done as a simpler alternative to the pointless exercise of having the host attempt to set up a connection with itself.)

The Receiver module

Similar to the sender module, this module on the host side is responsible for setting up, maintaining, and shutting down the connections to be used to receive messages from the client players (and the client side receiver module will do similarly for receiving messages from the host). The module maintains a queue of all messages that have been received, allowing the controller to request/retrieve them one at a time from the queue in the order they were received. The module will also be responsible for notifying the controller if a connection is dropped unexpectedly (e.g. by enqueuing a suitable message into the queue).

Aside from the messages actually received from other player/host machines over the established connections, the bulk of the receiver's communication with other modules consists of:

- the controller requesting/retrieving messages from the receivers queue,
- the sender enqueuing local-only messages into the receivers queue.

The UIX module

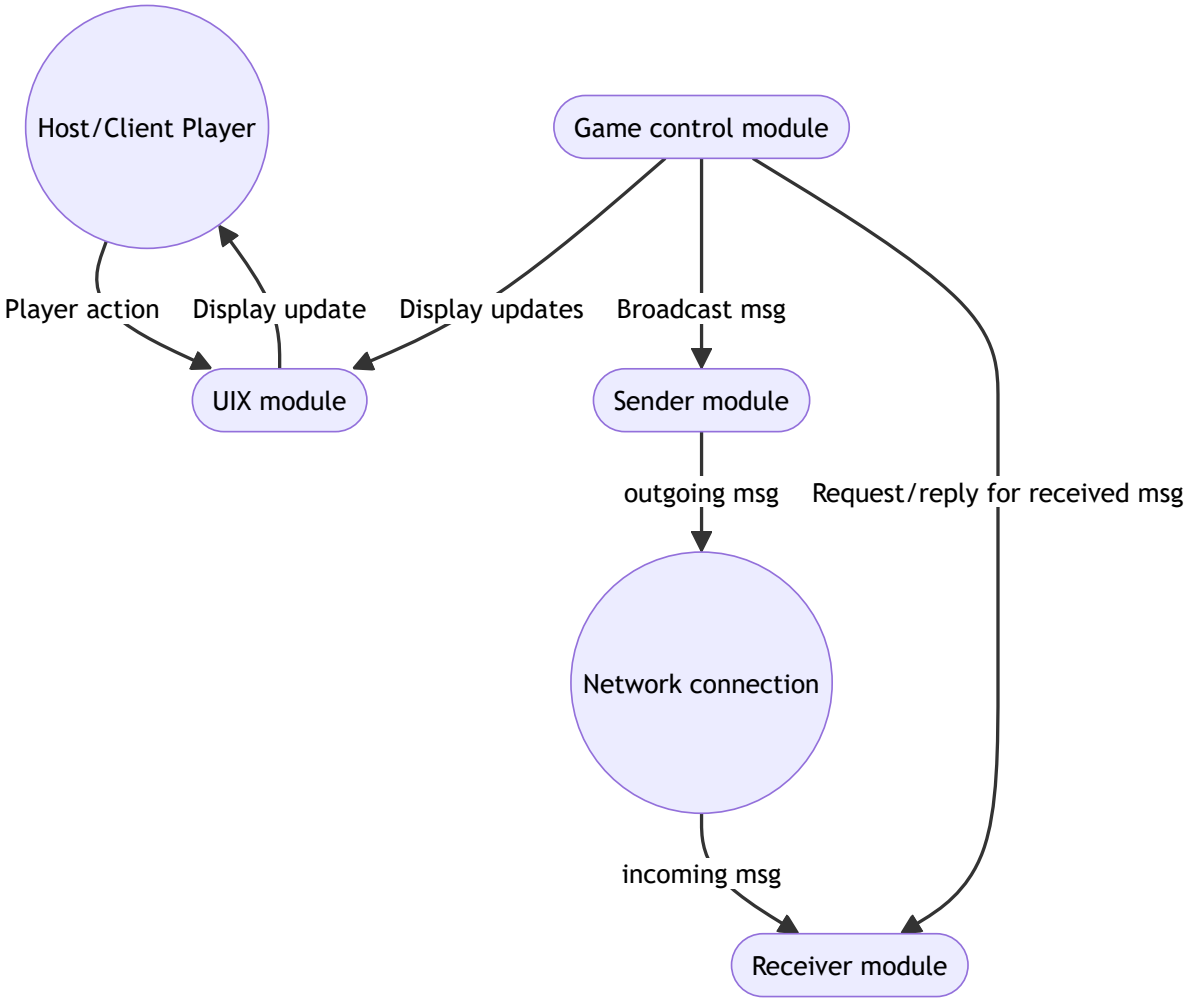
This module is responsible for handling all interactions that occur directly with the local player: all input capture (mouse, keyboard, touch screen, microphone, etc) and all output (sound and visual). All input must be captured as quickly as feasible, processed locally (e.g. for error detection and determining the nature of the input), and then suitable action command messages constructed to be sent to the host (with the sender module figuring out how to handle the situation if this player *is* the host). All output updates are conveyed to the UIX module as messages from the controller, and are placed in queue of display updates to be applied one at a time.

The UIX has two forms of communication:

- when it has detected (and checked) a user input as some kind of action request it then enqueues a suitable message in the sender's message queue,
- the controller enters new display update messages into its display updates queue.

Core architectural design diagram

The four core modules, their interaction with the player, and their interaction with other players through a network connection are depicted below.



6. Module descriptions

As outlined in section 5, there are four core interacting modules: Sender, Receiver, Controller, and UIX. Each of those is discussed in detail in sections 6.1 through 6.4 below, with further details on the data design and manipulation covered afterwards in section 7.

6.1 Sender: the message sending module

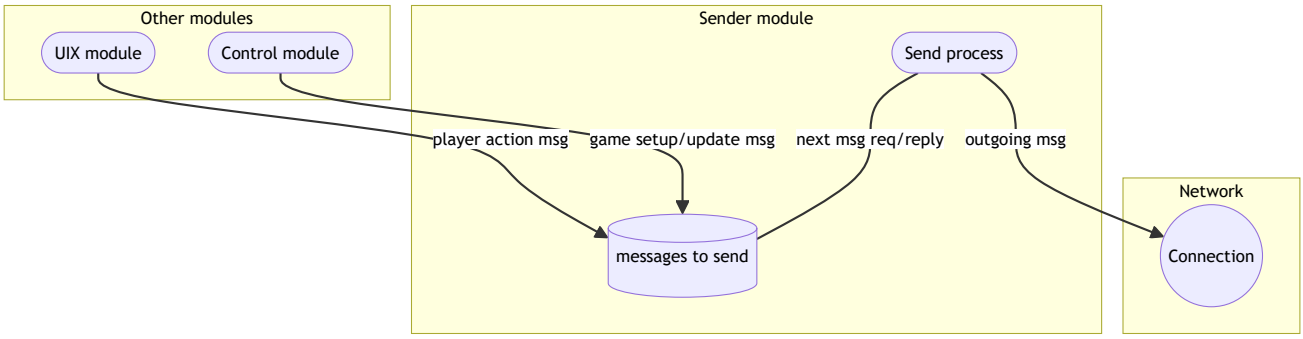
The sender module is responsible for all message transmissions that must go from the local machine to another player. Currently that comes in four forms:

- as host: sending to a single specific other player,
- as host: sending to all players,
- as host: sending to the local player (i.e. the host as a player) and bypassing the normal network connection handling,
- as client: sending to the host.

The messages to be sent can come from either the UIX module (for player action messages to be communicated to the host game logic) or from the control module (for game updates to be communicated to all players).

The game components and actions are depicted in the diagram below then discussed in sections 6.1.1 and 6.1.2.

Sender module design diagram



6.1.1 The send process

The send process itself must take messages from the queue of messages to be sent (taking the messages in the order they were enqueued) and transmit them to one or more other players.

As a connection-based strategy was chosen for our multiplayer design, the sender will also need to set up, maintain, and eventually shut down the relevant connections.

As a thread/task-based strategy was also chosen as a means for effective parallel handling of events and communication, the sender must also set up a separate thread for this activity.

Since the host sender thread corresponds with the client receiver thread, and the client sender thread corresponds with the host receiver thread, we have mapped out the action sequences for both sides of both threads below.

Anticipated action sequence on host side

Note that, because the host send and client receive operate are so tightly interwoven, the actions of both parties are outlined below, foreshadowing what is happening in the receive process.

1. Host sets up thread/task and socket/address for host sends, listens for connections
    i. Accepts incoming connections from clients
        a. stores necessary client connection info
    ii. Can begin checking queue of messages to be sent
        a. dequeue next available message (if any) or pause briefly (if none)
        b. message will have indicator of intended target (specific player or all)
        c. send to intended target using appropriate connection data
2. Host sets up thread/task and socket/address for host receives, listens for connections
    i. Accepts incoming connections from clients
        a. stores necessary client connection info
    ii. Can begin monitoring connections for incoming messages
        a. enqueues messages in received queue

Anticipated action sequence on client side

1. Client sets up socket/address for client receives
    i. Tries to establish connection with host send socket (timeout on failure?)

ii. Can begin monitoring connection for incoming messages
    a. enqueue messages in received queue

2. Client sets up socket/address for client sends
    i. Tries to establish connection with host receive socket (timeout on failure?)
    ii. Can begin checking queue of messages to be sent to host
        a. dequeue next available message (if any)
        b. send to host using appropriate connection data

6.1.2 The message storage queue

Detailed data discussion is provided in section 7, but each message to be sent contains an id for the intended target, a message type (indicating the nature of the action to be performed, and a data portion whose composition depends upon the message type. (Specific details on the composition of the messages are provided in section 7 of this document.) The queue needs core queue operations: enqueue, dequeue, isempty, and needs to be thread-safe (i.e. there is no possibility of queue corruption when different threads try to perform operations on the queue simultaneously).

The send process will need to check the intended recipient (all or a specific id) and broadcast appropriately, but is not concerned with the other message fields.

Note that the sender process is the only one that dequeues from this queue, while messages may be enqueued by either the control or the uix modules.

Send and receive will use a loop to check if there is a message to be handled (and handle if so), but to ensure this does not overload the processing power of the host/client machine a short pause will be invoked (a fixed number of milliseconds) each time the loop detects there is currently no message to be handled.

6.2 Receiver: the message receiving module

The receiver module is responsible for capturing all messages sent to the local machine from other players. On the host machine this means capturing all messages from all client players, while on the client player machines this means capturing all messages from the host. It must then store all the received messages in a form accessible to the controller module, which will extract and deal with them one at a time, in the order they were received.

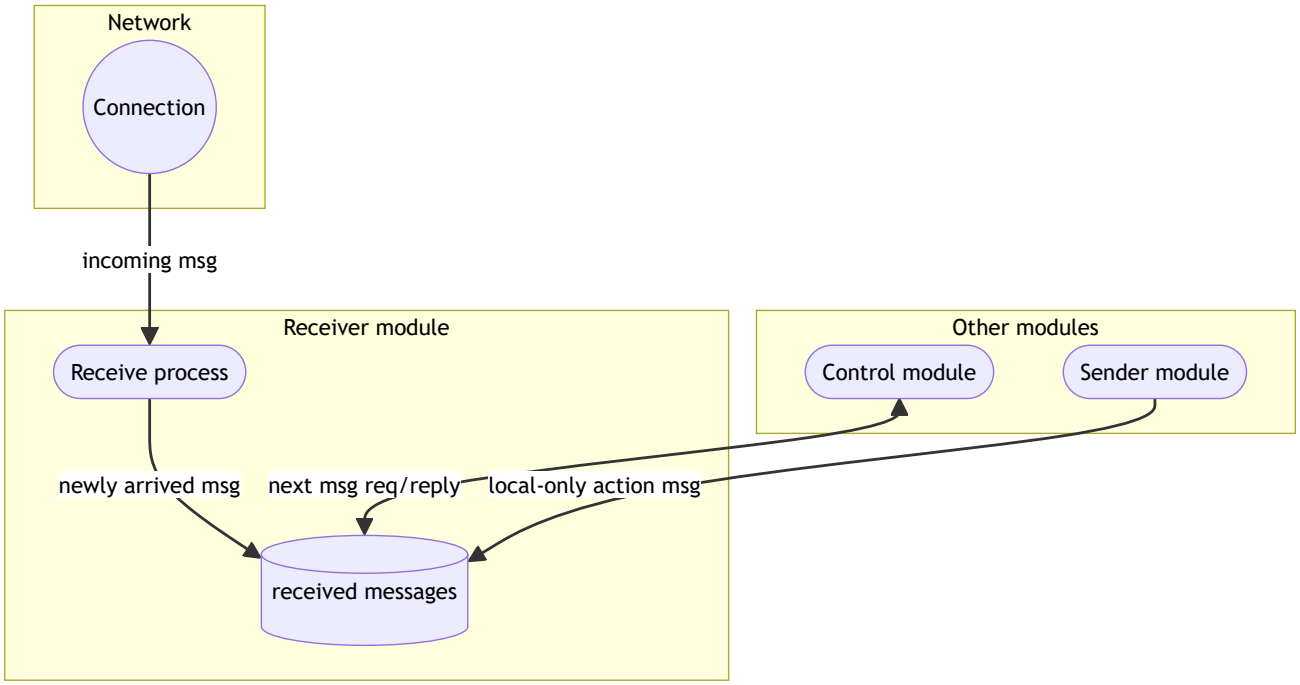
To do so, the receiver module must:

- set up and maintain a queue of messages received,
- set up and monitor connections with either the host or the other players,
- capture incoming messages and rewrite/enqueue them in the queue of messages received.

Note that there is also a special case where the sender module can directly enqueue a message, bypassing the normal send/receive sequence, when the sender and receiver are both the local host.

The diagram below models the core parts of the receiver module and the other program components it interacts with.

Receiver module design diagram



6.2.1 The receive process

As the client/host send/receive processes effectively mirror one another, the four combinations have been discussed together in section 6.1.1 (the send process).

6.2.2 The message storage queue

The message storage queue is identical to that described for the sender (section 6.1.2) except that the player id field is used by the host to indicate who the message came from, rather than who the message was sent to.

Note that messages are usually entered into the queue by the receive process but there is an exception on the host side for the sender process to directly enqueue copies of broadcast messages (so the message that would be 'broadcast' to the host player bypasses the need for network involvement). The control module processes are the only ones that dequeue from this queue.

6.3 Controller: the game control module

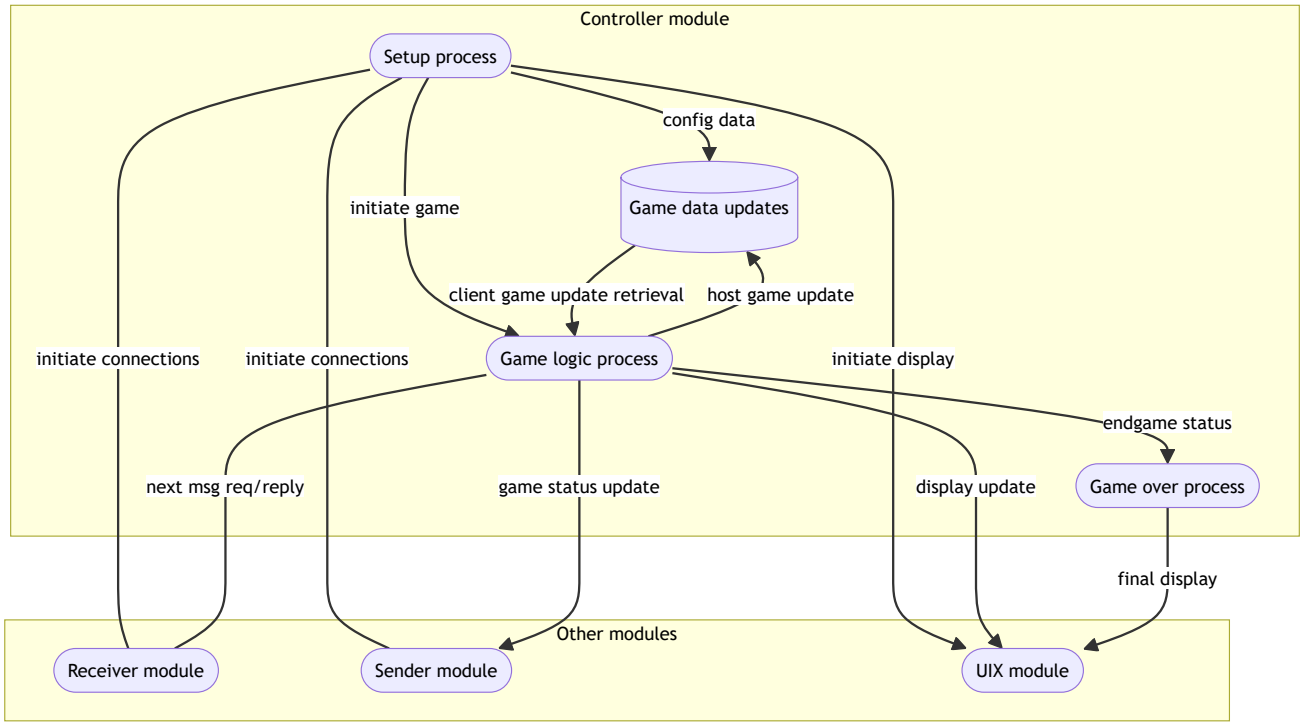
The game control module is responsible for tracking the current state of every character, creature, and object in the game, applying player's action commands, controling the actions of all NPCs, determining the results of every interaction (combat, movement, collisions, puzzles, etc), recognizing the team transitions from area to area, and recognizing and responding appropriately when players die or drop, and when teams have eventually won or lost.

To achieve this, the controller is modeled as three core processes (setup, game logic, and game-over) and a central storage of game data (though the latter is likely to be implemented as a collection of objects, as discussed in section 7 of this document).

The diagram below depicts the interactions of these three processes, the data store, and the external modules used to send messages to other players, receive messages from other players, and update the local user displays.

Sections 6.3.1 through 6.3.4 then go through and discuss each of the controller module components in greater detail.

Controller module design diagram



6.3.1 The setup process

The setup process is meant to handle everything prior to the point where gameplay actually begins. This means it has to handle

- the player's initial navigation through the options, help, and host/join menus,
- creating the necessary threads/tasks and message queues for capturing user input, updating the user display, updating the game logic/state, sending messages to players, and receiving messages from players
- set the necessary configuration data to allow the game to begin
- ensure all the necessary setup information has been broadcast to all players.

The anticipated action sequence below attempts to capture each of these steps in a practical series of steps.

Anticipated Action Sequence

1. Obtain host/client configuration choices
    i. establish the default settings
    ii. player may cycle through help/options menus multiple times
    iii. use the established settings when player selects host/join
2. Wait for all players to join successfully
3. Establish necessary game threads/tasks for
    i. input process
    ii. output process
    iii. game logic process
    iv. sending process (one thread per connection)

v. receiving process (one thread per connection)

4. Establish shared message queues
   i. messages to be sent
   ii. messages received
   iii. pending display updates
   iv. pending game updates (might become internal to game logic process)
5. Establish teams, energy-type assignments
   i. assign players to team/energy
   ii. broadcast team composition to all players
   iii. initialize game data update queue
6. Finish all necessary game data setup/configuration
   i. generate the map object
   ii. generate all PC, NPC, item, terrain, and puzzle objects with their starting stats and place them on the map
7. Invoke game logic process

6.3.2 The game logic process

The game logic process is responsible for the moment-by-moment control of the game itself. All the core actions and events are resolved here. The action sequence below outlines the expected series of steps, though some of those steps are themselves quite involved and are discussed in greater depth in subsequent subsections.

Anticipated Action Sequence

1. Trigger the start of game play (construct trigger message for all players and enqueue for sending)
2. Repeat until game-over detected
   i. Increment the game time clock (one tick)
   ii. Check for next message (if any) in the received messages queue
      a. if a message is found then
         a. apply any game data updates as a consequence
         b. generate and enqueue any resulting display update messages
         c. generate and enqueue any resulting send messages to host or clients
   iii. Determine NPC actions (if any) then
      a. apply any game data updates as a consequence
      b. generate and enqueue any resulting display update messages
      c. generate and enqueue any resulting send messages to host or clients
   iv. Resolve all PC/NPC/projectile movement, then
      a. detect collisions for all moved objects
      b. apply any game data updates as a consequence
      c. generate and enqueue any resulting display update messages
      d. generate and enqueue any resulting send messages to host or clients
   v. Check for game-over (either go back to top of loop or go on to step 3)
3. Invoke game-over process

Greater detail on the following aspects of the sequence is provided as follows:

- updating of the game data and determining messages to be sent is covered in 6.3.2.1

- controlling NPC actions is covered in 6.3.2.2
- detection of game-over is covered in 6.3.2.3

6.3.2.1 Game data updates and messages to be sent

Here we attempt to discuss all the events/actions the game should recognize and respond to other than determining what actions NPC characters should attempt next (that is covered in 6.3.2.2 below). At the moment this is provided as a categorized collection of possible circumstances, how to identify them, and the expected results/responses. It is entirely possible that a clearer organizational structure for this information might be developed for a future revision of this document.

List of actions and response

- **Player movement** (WASD)
  - a WASD action sets the direction the player is facing and, if there is no wall in the way, the player progesses in that direction a distance based on their fixed speed (if there is a wall in the way, should a 'bump' sound be generated?)
  - the new location/direction needs to be sent to all players
- **Player speech** (K)
  - the sound text, volume, and location needs to be sent to all players in range
- **Player quit** (or lost connection)
  - the player is treated as dead (energy level 0)
  - the updated player status needs to be sent to all players
- **Player options menu** (ESC)
  - this only has effect for the local display, a message is sent directly to the local display-update queue
- **Player projectile fire**
  - if the player lacks sufficient energy to fire then nothing happens, otherwise proceed with the remaining steps
  - deduct the cost of firing the projectile
  - create the projectile and set its stats (location, direction, speed, energy level, display info)
  - the projectile information needs to be sent to all players
- **COLLISIONS** (collision-detection scheme needed)
  - between character/NPC and projectile:
    - the character/NPC takes damage based on the energy type, character/NPC resistance, and projectile strength. If the damage reduces the character/NPC to critical levels then they become incapacitated, if reduced to zero they are dead.
    - the updated projectile and character/NPC information needs to be broadcast to all players
  - between character and puzzle trigger/gate
    - if the character is of the correct energy base (for the trigger type) and the trigger was not already activated then activate the trigger, possibly opening the gate (depends on specific puzzle, see the puzzle data section)
    - if the character is not of the same energy type as the trigger/base then the character takes damage and may be reduced to critical or dead status
    - any changes to the trigger, gate, or character status need to be sent to all players
  - with zone entry/exit trigger
    - on zone entry, if the zone was not already active then activate the zone, add the player to the list of

players active in the zone, and activate all NPCs within the zone
        - on zone exit, if this was the only player currently in the zone then deactivate the zone, remove the player form the list of players active in the zone, and deactivate all the NPCs within the zone
        - the updated zone/creature activity status needs to be broadcast to all players
        - for zones that are restricted to one team only, the gate appears as a passageway to the permitted team and as an impassable wall to the other team. In the case of the final boss zone, the gate initially appears as open passageway to both teams, but as soon as any player from one team reaches the gate it switches to an impassable wall for the other team
    - *if it is decided that players cannot re-enter a zone once their entire team has departed it then the gate-as-impassable-wall can be set for both teams*
    - with creature or other player
        - if they are of different energy bases then both parties take damage based on their attack and resistance values, reducing their current energy level and possibly dropping them to critical or dead status
        - the updated status of both parties needs to be broadcast to all players
    - with item
        - if the item is an xp item then the player is immediately granted the xp boost (may reach an xp threshold, see below)
        - if the item is a healing item of a matching energy base then the player is immediately granted the current energy boost (up to their maximum energy level)
        - if the item is a healing item of a different energy base then the player is damaged based on the energy level of the item and the player's resistance, possibly reducing the player to critical or dead status
        - the item is removed from the map afterward
        - the updated status of the player and the item needs to be broadcast to all parties
    - with wall
        - no action required (cannot move into the same space as a wall) unless the requirements are updated to specify this creates a 'bump' sound of some volume level
- **Sound generated (by player, NPC, or item)**
    - the text for the sound needs to be looked up and the sound location, volume, and text needs to be sent to each player
- **Experience threshold reached**
    - the player needs to be given the opportunity to choose their next upgrade type, but the requirements are not clear on when/how this should happen
    - one of the player's stats will change as a result of the upgrade, this needs to be broadcast to all players

6.3.2.2 NPC action updates and messages to be sent

In this section we specifically focus on the types of NPCs the game currently supports, and how the game controller should determine their next action during each update cycle.

There are currently four types of supported NPC:

- radiant mounds,
- radiant drifts,
- the energy tsar,
- the NPC puzzle guide (from puzzle zone 2).

Radiant mounds are stationary and appear in puzzle zones 1A/1B, combat zone 1, and the final boss zone. Their generation stats and locations are provided in the requirements document. These are stationary, so the only action taken is to have each of them (when their map zone is active) fire one projectile in each direction each second.

Radiant drifts move but do not fire projectiles. Their generation stats and locations are provided in the requirements document. Their movement pattern (when their map zone is active) is always to move towards the closest player, which will have to be determined/updated by the controller.

As radiant drifts are not particularly clever, they do not consider terrain when determining which player to pursue nor what is the best path to take.

The identification of the closest player operates as follows:

- Since the drifts only move NSEW, their distance from a player (ignoring intervening obstacles) is the vertical distance from the player plus the horizontal distance, i.e. | playerX - driftX | + | playerY - driftY |.
- The distance is calculated for all players in the current map zone (obtained from the map zone's list of players) and the nearest player is chosen (ties are broken randomly).

The pursuit of the closest player operates as follows:

1. If such a spot exists, the drift attempts to move to an adjacent unoccupied space that is closer to the player than the drift's current position, with ties broken randomly.
2. Otherwise, if such a spot exists, the drift attempts to move to an open but occupied adjacent space that is closer to the player, with ties broken randomly…
3. Otherwise, if such a spot exists, the drift attempts to move into an adjacent unoccupied space that is no further from the player than its current location, with ties broken randomly.
4. Otherwise, if such a spot exists, if the drift attempts to move into an adjacent unoccupied space that is no further from the player than its current location, with ties broken randomly.
5. Otherwise, if such a spot exists, the drift moves to any random unoccupied adjacent section of space…
6. Otherwise, the drift moves to any random adjacent space.

The **energy tsar** (generation stats and location provided in the requirements document) moves like a radiant drift but fires projectiles like a radiant mound.

The requirements for the **NPC guide** in puzzle zone 2 do not provide anywhere near enough detail to dictate the creature behaviour, so the following is tentative until the requirements are refined:

- When a player of the same energy base reaches (touches?) the guide then it will begin moving, following a fixed path to the matching gate which is unlocked when the creature is adjacent to it (an invisible puzzle trigger can be used for this purpose).
- The creature's generation stats and location are specified in the requirements document.

6.3.2.3 Game-over detection

There are two ways for the game to be over:

- every player on both teams is dead (energy level 0 or quit/lost connection), or
- a player has reached The Great Core.
  Under any other circumstances the game is not yet over.

6.3.3 The game over process

This process is, as one would expect from the name, meant to address the steps necessary once it has already been determined that the game has ended: either one team has won (reached The Great Core first) and the other lost, or both teams have lost.

The steps to be taken, and information needed, for this final display are outlined below.

Anticipated Action Sequence

1. Determine the won/loss status of each team (did they get the core or not).
2. Determine the survival status of each player (are they dead or not).
3. Determine the xp level and upgrade status for each player (lookup from the player object).
4. Construct and enqueue display update messages for each player based on the results of 1-3.
5. Wait for the player to either close the game or request to be taken back to the initial setup to potentially create/join a new game.

6.3.4 The game data update queue

This queue is meant to store messages regarding updates that need to be applied to the current game state. This allows the game logic in the host controller to determine what is happening in the game and communicate the results/updates to all players as update messages. The controller logic for the client players simply extracts messages from this queue and applies the specified change.

6.4 UIX: the user interaction module

The role of the UIX module is to handle all direct interactions with the player: capturing any player inputs (mouse, keyboard, microphone, etc) and controlling all means of conveying information to the player (display screen, speakers, haptic feedback if that were to be implemented, etc).

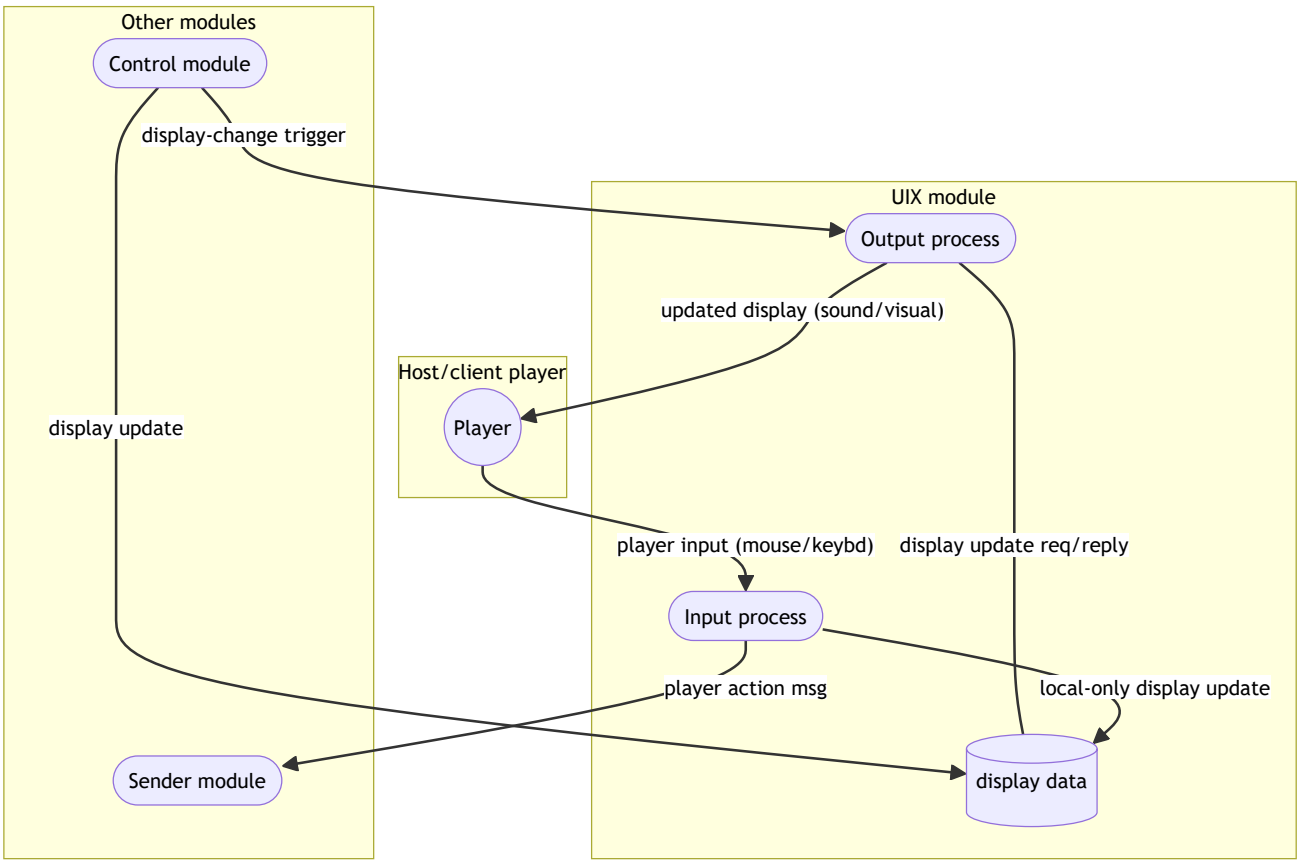
For captured player input, the UIX is responsible for basic error checking and then enqueuing the input as player action messages which can later be sent to the host for processing.

For updating the player display, the UIX relies on a queue of display update messages that is being filled by the control module. These messages tell the UIX what form of modification is to be made to the display (display in this sense including sound and other output mechanisms).

There is also the possibility for the UIX to recognize certain specific player inputs as having a strictly local effect (such as hitting the ESC key to bring up the options menu), in which case the input process can enqueue the requested action directly into the display updates queue.

The elements of the UIX module and their interactions with other program components are depicted in the diagram below, then discussed in greater detail throughout the rest of section 6.4.

UIX module design diagram



6.4.1 The input process

The input process is focused on capturing user commands in whatever forms are supported, e.g. keyboard, mouse, touch screen, microphone, etc. It needs to detect the user action, validate it, form a message describing the action, and putting the message in a queue for processing: either the sender's queue (usually), or the display updates queue (for strictly local updates such as bringing up the options menu).

Anticipated Action Sequence

As with the sender/receiver processes, the input process continually checks for new user input and either processes it (if found) or pauses briefly (a fixed number of milliseconds) otherwise. Thus the following cycle is repeated endlessly until the game ends:

1. Check for input then perform either 2 (if none found) or 3 (otherwise)
2. Pause for N milliseconds then go to step 1
3. Process the new input
    i. check for validity, proceed if valid
    ii. identify the nature of the action and encode an action message
    iii. if the action is strictly local then enqueue it in the local display update messages, otherwise enqueue it in the sender's queue of messages to be sent

6.4.2 The output process

The output process is responsible for taking display update messages from the display data update queue and

making the appropriate revisions to the local display. Those actions can include:

- switching from the current menu or screen to another (specifying the new target screen or menu),
- adjusting the position of the player's display window on the larger map (e.g. due to player movement),
- adjusting the colour, symbol, or visibility of an object on the map,
- adjusting the position of an object on the map (including moving it to/from the hidden portion of the map),
- adjusting the content of the subtitle window,
- adjusting the player energy bar.

The representation of the data, and how they can be adjusted, is discussed in section 7 of this document.

Anticipated Action Sequence

Much like the input process, the output process uses a check-and-pause scheme for continually checking if there are updates to be applied (i.e. requesting the next available item from the display data update queue) and then making suitable modifications to the system display.

Some aspects of manipulating the local display will depend heavily on the programming language or game engine ultimately chosen for implementation, but the general sequence of events is as follows:

1. Check for the next available update message then perform either 2 (if none found) or 3 (otherwise)
2. Pause for N milliseconds then go to step 1
3. Process the new update
    i. identify the nature of the action
    ii. revise the display content
    iii. refresh/redraw the display (or relevant portions thereof)

6.4.3 The display data update queue

The game logic process for the local player identifies updates that necessitate changing what the player sees on screen. The information related to the update is encapsulated in a data message (much as with the send/receive message queues earlier) and enqueued in the display data update queue for retrieval and use by the output process. The specific content of the display update messages is discussed in detail in section 7 of this document.

6.4.4 Menus and navigation map

There are seven core menu/gameplay screens the player may encounter, of which:

- three are only encountered at the beginning of a game (Main menu, Host, Join),
- one is only encountered at the end of a game (Game over, also used for lost connections),
- one is the main gameplay screen,
- two can be reached at any time: the Options menu (accessible from the main menu or by pressing ESC during gameplay), and the Help menu (accessible from each of the other menus and indirectly through the Options menu during gameplay).

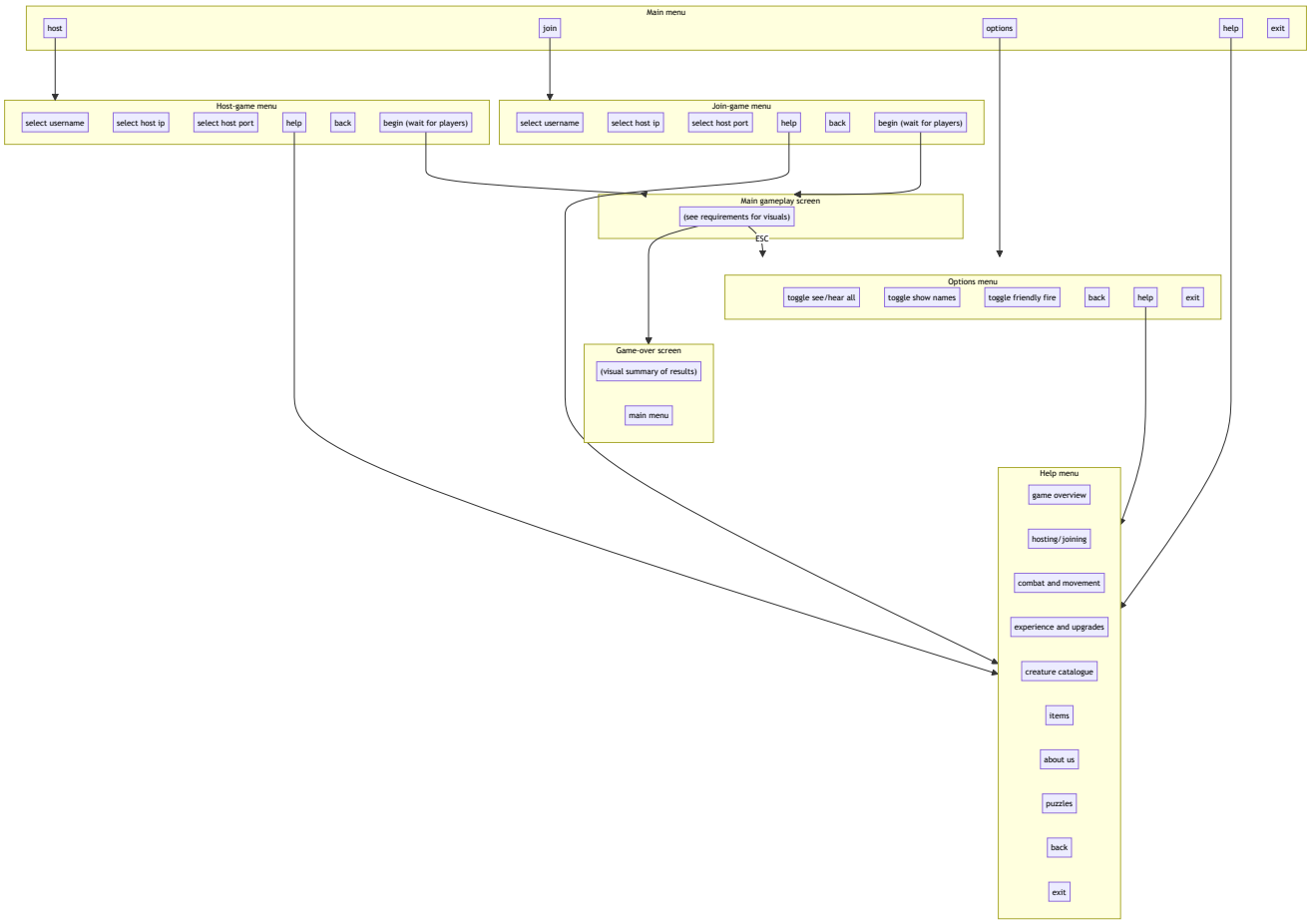
The seven screens are listed below, followed by a navigation map showing the links between them.

- Main menu: the starting screen for the game with menu options that each take the player to another core screen: Host, Join, Options, Help, Exit (ends the game)

- Host game: allows the user to set their username, IP, and port, and gives the user options for the Help menu, Back (return to main menu), or Begin (start game, waits for other players)
- Join game: allows the user to set their username, host IP and port, and gives options for the Help menu, Back, or Begin
- Help screen: provides the user with options to request help in eight different topic areas, or go Back to the previous menu, or exit the game. The eight help areas are listed below, the specific text content is provided in the Requirements document.
    - Game overview
    - Hosting/joining
    - Combat and movement
    - Experience and upgrades
    - Creature catalogue
    - Items
    - Puzzles
    - About us

Menu navigation map

- *the back option returns you to your previous screen*
- *the help options each take you to a static text screen for the relevant topic*



7. Data design

In this section we attempt to organize and describe all the information elements manipulated by the game system.

Currently we have the information divided into three broad categories:

- player data (information about the player, outside the actual gameplay),
- message queues and message data (for send queues, receive queues, and display-update queues),
- in-game data (by **far** the largest and most complex category).

The three information groups are covered below in sections 7.1, 7.2, and 7.3 respectively, then the information is represented in the entity relationship diagram (erd) of section 7.4.

7.1 Player data

Player data refers to information outside the actual gameplay data:

- the player's chosen username
- the player's game connections information:
  - the host ip/port
  - socket/address information for each send connection and each receive connection
  - the connection status for each connection (tentatively: pending, active, disconnected)
- a reference to the object/storage containing the player's in-game data (see 7.3.1)

This player data would largely be configured during the setup process, but the connection status would be updated as needed during game-play.

7.2 The message queues and message content

To keep track of generated information/updates that are still awaiting some form of processing, the game uses three message queues on each player's system:

- the queue of messages waiting to be sent,
- the queue of received messages (waiting to be processed),
- the queue of display-update messages (waiting to be applied).
- the queue of game-update messages (waiting to be applied).

The message queues themselves would need to be thread-safe, with methods to construct, destruct, enqueue a message (at the back), dequeue a message (from the front), and check if the queue is empty. Fortunately queues of a similar nature are well understood and typically well supported by most major languages and game engines, and should not require extensive development by our team.

While there are three message queues (messages to be sent, messages received, display-update messages), it is expected that the information contained in the send and receive messages will be identical so we have encapsulated those into a 'send/receive' message type below. With respect to the content of messages sent/received, however, messages can either convey display-update information or player-action information. As a result, we have three distinct message types: send/receive, display-update, player-action. Each of these is discussed below (in sections

7.2.1, 7.2.2, and 7.2.3, respectively).

7.2.1 Send/receive message structure

Send/receive messages need to specify the following:

- the sender
- the intended receiver (possibly with special values to indicate ‘all’ or ‘local’)
- the content type (display-update, player-action, or game-update)
- the actual content (details vary depending on update type)

The design rationale for this structure is that it allows the send/receive message structure to be designed, implemented, and maintained independently of the design/structure of the other two message types.

7.2.2 Display-update message structure

The display-updates may turn out to be somewhat dependent on the language/engine chosen for the game implementation, but it is believed the following are the types of updates that need to be represented and the information that needs to be conveyed to the UIX module to carry them out:

- move the display window
  - specified by identifying where the upper-left corner of the display window should be positioned relative to the global map
- move an object on the map
  - specified by the global id for the object to be moved (we’re assuming all in-game objects will be given unique ids) and the new x,y coordinates for the object
- change an object’s display properties
  - this needs to specify the id of the object to be updated, the property to be changed, and the new value
  - display properties are assumed to include the item’s symbol, color, intensity, visible/not
- add an item to the map
  - this needs to specify the item id and the x,y coordinates for the placement location (see discussion below regarding ‘off map’ placement)
- remove an item from the map
  - this needs to specify the item id and the (off-map) x,y coordinates for the new placement location (see discussion below regarding ‘off map’ placement)

The display-update message structure will thus likely be implemented as:

- a type indicator (which of those five kinds of updates does the message represent)
- a target object id (which specific item’s property is changing)
- a set of data fields, one or two of which contain the new value(s) for this specific field (since different update types alter different numbers and types of value):
  - two integer fields
  - a character field
  - a text field.

With respect to adding/removing items from the map, the team is currently planning to add a large unreachable area to the map where objects can be placed when not actually in play. Thus all objects could be generated during setup

(and set as inactive), then when an item needs to spawn in it can simply be moved from this hidden area to the designated map location. Similarly when an object is destroyed, or needs to despawn, it can simply be moved to this hidden area and marked as inactive.

7.2.3 Player action message structure

Player-action messages represent a received player-input indicating an action they want to perform. The actions themselves are (so far) relatively straightforward:

- move and face up/left/down/right (WASD)
- bring up the options menu (ESC)
- fire a projectile in the direction the player is facing (SPACE)
- speak (K and the line of entered message text)

For all but the speak option, only the command type is needed, but for the speak option a text field of some form is also needed.

7.2.4 Game-update message structure

A game-update message conveys a single status update (determined by the host-side controller logic) to the client players where *their* controller can use it to update the local game state (thus ensuring that only the host-side controller makes decisions about what is taking place in the game and what the true current state of the game is).

The message must thus identify the item whose state is changing and the nature of the change. The relevant items and their associated data components are listed in sections 7.3.1 through 7.3.7. The kinds of data values to be assigned are generally integer, character, or string, so it is expected that the messages will contain:

- an id field to specify which particular game item is being updated,
- an id field specifying the nature of the change,
- plus several data fields, one of which will contain the new updated value for the designated item/field
    - an integer field,
    - a character field,
    - a text field.

The lists of item ids and nature-of-change ids have yet to be established, and are expected to be lengthy and to evolve as development progresses.

7.3 Game logic data

The game logic data needs to represent all the information about every character, creature, item, and puzzle-related object on the map, as well as the actual map content and current game state. As mentioned earlier, this will be by far the most complex data collection to model in a way that is easy to use and maintain effectively.

Currently we have the data compartmentalized into seven categories:

- player character data,
- NPC character data,
- projectile data,
- item data,

- puzzle data,
- display data,
- map data.

At the overall game level we will need some form of reference to each of the above collections, e.g.

- a list of player references,
- a list of NPC references,
- a list of projectile references,
- a list of item references (possibly divided into healing and xp awards),
- a list of puzzle item references (possibly divided into triggers and gates),
- a list of display references (one per player),
- a reference to the map data.

Each category is discussed individually (in sections 7.3.1 through 7.3.7 below), then modeled as an entity-relationship diagram (section 7.4).

7.3.1 Player character data

The information to be maintained for each player character is as follows:

- unique id
- team
- display symbol, colour, intensity
- energy base
- current x,y coordinates and map zone
- current facing (NSEW)
- current and maximum energy levels
- replenishment rate
- resistence level
- attack level
- current upgrade level for energy, attack, resistance, replenishment
- experience points

7.3.2 NPC data

The information to be maintained for each NPC character is very similar to that of the player characters, thus only the differences are listed here.

The NPC creatures need three additional fields:

- the AI control pattern to be used (currently we need control patterns for shambling mounds, radiant mounts, the energy tsar, and the NPC guides for puzzle zone 2)
- is it currently active or inactive
- sound text and volume (does it make noise, what kind, how loud)

The experience points field is also used somewhat differently: for NPCs this designates the experience point value of

the creature if/when players defeat it.

7.3.3 Projectile data

The information needed for projectiles is somewhat simpler than that needed for characters/NPCs:

- unique id
- display symbol, colour, intensity
- energy base
- current x,y coordinates and map zone
- current facing (NSEW)
- energy damage amount (if the projectile strikes something)
- speed (this may not be needed if a fixed speed is used for all in-game projectiles)

7.3.4 Item data

Currently only two types of items are supported: healing items and xp award items.

Healing items need the following information:

- unique id
- current x,y coordinates and map zone
- energy base
- healing amount
- display symbol, colour, intensity

XP award items need the following:

- unique id
- current x,y coordinates and map zone
- xp award amount
- display symbol, colour, intensity

7.3.5 Puzzle data

The current puzzles (aside from the mazes, which are purely a map/terrain issue) involve the triggers to open a gate, the gates themselves, and special NPC guides for the zone 2 puzzles. The triggers and gates can each be regarded much like items types, with their required properties described below.

Trigger properties include:

- unique id
- current x,y location and map zone
- currently active/inactive
- energy base
- display color, symbol, intensity
- trigger type (triggered by same energy base, energy base that can’t hear the trigger, energy base that can’t see the trigger, any energy base)

- sound text and volume
- has it been triggered yet

Gate properties include:

- unique id
- current x,y location and map zone
- currently open/not
- currently visible/invisible/appears as wall
- energy base
- display color, symbol, intensity
- which trigger(s) is it attached to
- is it opened by any one of those triggers or does it wait until all have been triggered
- sound text and volume

7.3.6 Display data

The display data covers which portion of the global map is currently shown on a player's gameplay screen. Most of that is dictated by what is visible (for the player's energy base and current game state) in that portion of the map.

The specific display properties that need to be stored are:

- the height and width of the display (what amount of the global map can be shown at once)
- the current global map x,y coordinates of the upper left corner of the display

7.3.7 Map data

The design of the map data is still somewhat under debate by the team, but the current thought is as discussed below.

The fixed terrain for the map (walls and passages) is represented as some flavour of 2D array, each array position corresponding to a square on the map that is either open space (passage) or blocked terrain (wall).

The current map in use has seven distinct zones (puzzle zones 1A/1B, 2A/2B, combat zones 1/2, final boss zone), each of which can be active or inactive and each of which is activated when any player has passed a trigger point to enter the zone (puzzle triggers can be adapted for this) and deactivated when all players have left the zone.

To accomodate this, we may require a list of map zones, each with a reference to its entry/exit triggers and a field denoting its current state (active/inactive). For game logic processing it may also prove helpful to have lists of references to the players, NPCs, items, and puzzles currently in the zone.

7.4 Logical ERD

In this section we attempt to show the relationships between the core data components (entities) used in the game, as well as the core data fields (attributes) associated with each entity.

The chosen entities in the diagram are listed below, along with a reference to where in chapter 7 there data content is discussed.

- the overall game [T.B.D. discussing missing in section 7]
- the global map and map zones (see 7.3.7)
- the player display windows (see 7.3.6)
- the teams [T.B.D. discussing missing in section 7]
- the player (see 7.1) and their characters (see 7.3.1)
- the NPCs (see 7.3.2)
- the game items (healing and xp awards, see 7.3.3)
- projectiles (see 7.3.3)
- puzzle triggers and gates (see 7.3.5)
- sounds [T.B.D. discussion missing in section 7]
- send/receive messages (see section 7.2.1)
- update messages (for display or game updates, see section 7.2.2 and 7.2.4)
- action messages (for player action commands, see section 7.2.3)

Composite attributes to be used in the E.R.D. below

To reduce the complexity of the diagram, the logical ERD further below makes use of some composite attributes (attributes made up of multiple smaller/simpler attributes):

- location (made up of a character's x,y coordinates and the direction they're facing)
- displayInfo (made up of an object's symbol, colour, and intensity)
- baseStats (made up of current and maximum energy levels, resistance, attack, replenishment, and experience)
- upgradeLevels (max energy, replenishment, attack, resistance)

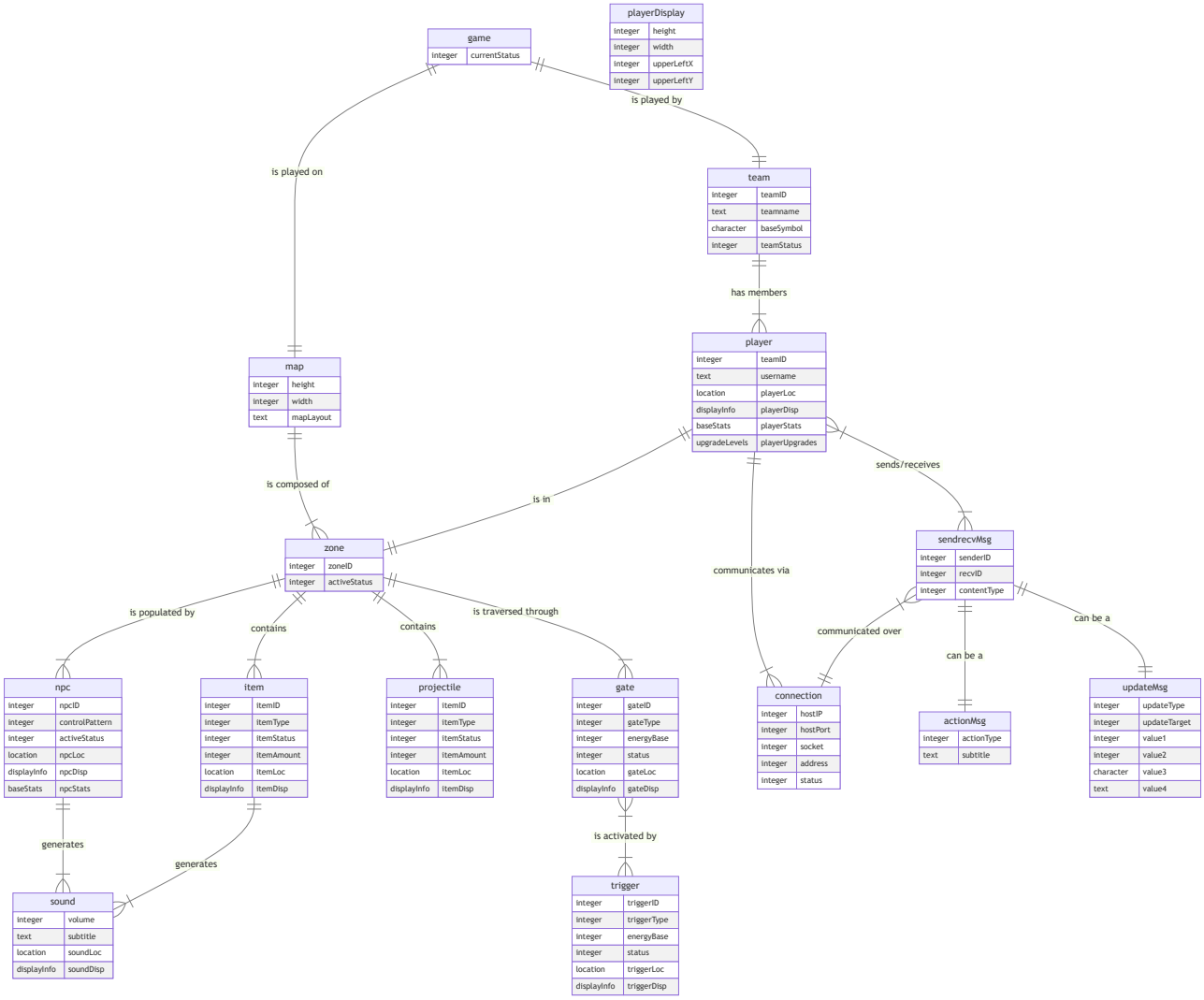
displayInfo	
integer	colour
character	symbol
integer	intensity

baseStats	
integer	currentEnergy
integer	maxEnergy
integer	replenishment
integer	resistance
integer	attack
integer	xp

upgradeLevels	
integer	maxEnergyLevel
integer	replenLevel
integer	resistLevel
integer	attackLevel

location	
integer	xCoord
integer	yCoord
character	direction

Logical ERD



DETAILED DESCRIPTIONS/DISCUSSIONS OF THE ERD ARE STILL TO BE COMPLETED

8. Game state and flow of play

In the current version of the game we have just one map and a relatively linear flow of actions through the game, with the game logic structured to reflect that expected sequence of actions. Ideally this can be restructured so that it is easier to make game progression less linear and more modular, perhaps even encoding the expected sequence as some form of hidden map data.

The expected flow of events is listed below and then shown graphically in the state diagram, with some steps bypassed if one team loses all its players early (through dissipation or dropped connections) and the game ending if both teams lose all their players.

The list of states team A progresses through is given below, team B follows a similar progression but the pace of the two teams is independent (i.e. it is theoretically possible that one team could still be on the first puzzle of the first map zone while the other team is encountering the final boss).

Team A progression sequence:

1. in puzzle zone 1

- working through the first three puzzle gates (one per energy base) in map zone 1
- working through the second three puzzle gates (one per energy base) in map zone 1
- working through the final puzzle gate (all three energy bases) in map zone 1

2. in combat zone 1

- working through the monsters and (potentially) the other team

3. in puzzle zone 2

- working through the maze (with or without the aid of the NPC)

4. in combat zone 2

- working through the monsters and (potentially) the other team

5. in the final boss zone

At each stage the team may either progress or die at the hands of creatures, items, traps, or (in the combat zones) the other team.

The game ends when one team beats the final boss or when all members of both teams have dissipated.

The creatures in each map zone are inactive until a team enters that zone, at which point the creatures become active and the game logic process begins updating their status. When no team is present in a map zone the creatures go back to their inactive status.

9. Transition to physical design

In this section we outline the plans for transitioning from the logical design presented in sections 1-8 to a physical design suitable for implementation. At the time of writing, we have only begun to plan the transition to a physical design: we have chosen our programming language and core relevant language modules and we have identified some of the key implications this will have for design and implementation, but have yet to create a detailed object model.

9.1 Core implementation decisions

The single biggest implementation choice was whether to build our program more-or-less from scratch in a language (like C++, Python, or Java) or to learn and use one of the major existing game engines (like Unity or UnReal). The secondary decision would then be which specific language or engine.

Because we were interested in the challenge of a from-scratch design but wanted a language with good network, thread, and graphics support for relatively inexperienced programmers, we eventually decided on using Python. The language provides good thread support and an existing thread-safe queue data structure, relatively simple processes for setting up and using network connections, the PyTurtle graphics system if we choose to go more advanced than ascii graphics, and an ok class/object system.

9.2 Object model

As discussed in the introductory paragraph for section 9, we have yet to create a detailed object model. The current expectation is that each of the entities described in the ERD (section 7.4) can be directly translated into suitable objects, along with any additional objects to represent the core program components outlined in the system decomposition of section 6. The objective is to complete this section during phase 4 as part of a skeletal/framework implementation.

10. Appendix: the grand design

This diagram combines all the individual module diagrams together into one "grand" scheme, capturing the interactions between key elements of each module. It is intended primarily as a cross-reference tool for the various module diagrams. Detailed discussions of each module and their various components can be found in section 6 of this document.

The grand design diagram