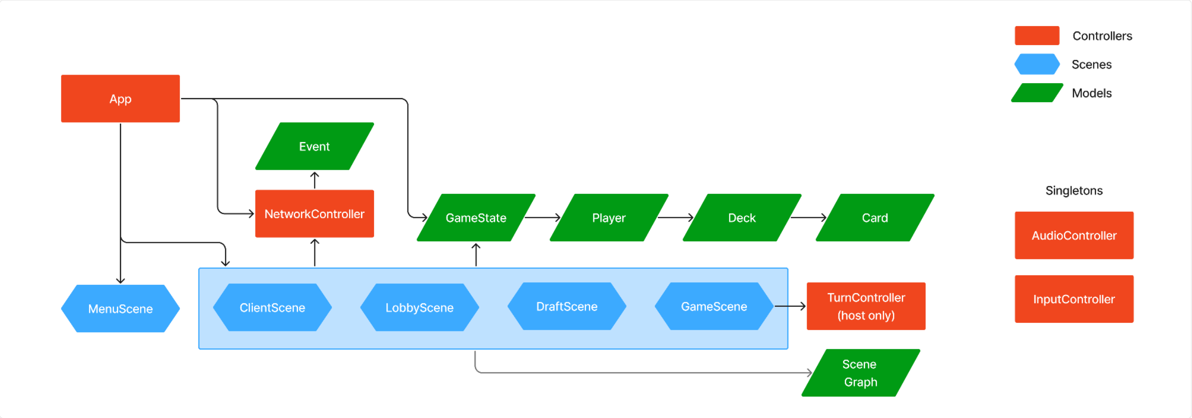
Trigger Happy Architecture Specification

innate studios — Team 8

Amber Min, Caroline Hohner, Elaine Ran, Ireanne Cao, Jacob Seto,
Linda Hu, Luke Leh, Pedro Pontes García, Phoebe An, Shirley Li

Dependency diagram



Models

Card

Description: This class represents a card in a player's hand, deck, or discard pile. A card object has a card type, which is an enumeration.

Justification: This model is not merely an enumeration. It is responsible for returning a card's type, its corresponding asset string, obtaining a string value out of it, and comparing it to other cards with an overloaded equality operator.

Responsibilities	Collaborators
Get card type	-
Get asset string	-
Get string value	-

Responsibilities	Collaborators
Compare with another card	-

Deck

Description: This class represents all the cards that a player owns in a game. It stores the player's current hand pile, discard pile, and draw pile. It is able to shuffle and draw cards.

Justification: This model provides an easy way for the player to interact with their cards and store information on what the player may play in a round.

Responsibilities	Collaborators
Add card to draw pile	Card
Draw four cards from draw pile to hand pile	Card
Discard card from hand pile to discard pile	Card
Discard entire hand pile into discard pile	-
Get number of cards of specific or any type in specific pile or whole deck	Card
Get whether deck is full given initial size	-
Empty whole deck	-

Player

Description: This class represents a player in the game. It stores the player's ammo, lives and their current deck.

Justification: This model provides an easy way for the player to interact with their cards and store information on player ammo, lives, and cards.

Responsibilities	Collaborators
Get/set ammo	-
Get/set lives	-
Set avatar	-
Set name	-
Get UUID	-
Initialize deck	Deck
Get deck information	Deck

GameState

Description: This class represents the state of a game, including a map of all the players in the game, and a specific reference to the client's player.

Justification: This model class is passed between different scenes within the main gameplay in order to transmit general state about the game. It is used as a container class.

Responsibilities	Collaborators
Get/set client player information	Player
Get other player information	Player

Event

Description: This class represents a network event (such as "I'm ready", "I took my turn") sent from a client to the ad-hoc server and vice versa. Events are sent over the network as JSONs.

Justification: This model class is needed to store, serialize and deserialize event data to send it across the network.

Responsibilities	Collaborators
Get event data	-
Store event data from JSON (deserialize)	-
Get serialized JSON from event data	-

Controllers

App

Description: This is the root controller. It creates the scenes and manages transitions between them.

Justification: This controller is the base for all other controllers, so it is essential to the game.

Responsibilities	Collaborators
Load assets for game	-
Initialize network connection	NetworkController
Update loading progress	LoadingScene
Update and render active scenes	HostScene, ClientScene, MenuScene, GameScene, DraftScene, WinScene
Transition between scenes	HostScene, ClientScene, MenuScene, GameScene, DraftScene, LoadingScene, WinScene

NetworkController

Description: This singleton class is responsible for connecting hosts and clients. It receives and sends messages to hosts and clients. It also handles serialization and deserialization of packages into Events.

Justification: This class is needed to manage networking using a centralized controller.

Responsibilities	Collaborators
Establish connections with server	-
Send outgoing events	-
Receive incoming events	-
Check and update network status	-
Serialize and deserialize network events	Event

TurnController

Description: This class is responsible for moderating rounds. It acts as the ad-hoc server by synchronizing round flow and resolving turn outcomes.

Justification: This class is needed to move the game through each turn and ensure each player's local state is consistent with the rest of the party.

Responsibilities	Collaborators
Collect turn messages from players	NetworkController
Resolve and report turn outcomes to players	NetworkController
Send information about disconnected players	NetworkController

AudioController

Description: This class loads and plays all audio for the game.

Justification: This controller centralizes all the audio and gives extra functionality such as fade-in/fade-out effects to apply to the audio played.

Responsibilities	Collaborators
Play audio	-
Apply audio effects	-
Load all audio in the game	-
Fade in/out music	-

InputController

Description: This singleton class detects and registers player inputs, including taps and drags.

Justification: This controller is necessary to abstract complex actions like drag-and-drop for cards or decks, swipes for character scrolling, or direct access to taps in complex layouts to avoid overlapping listeners.

Responsibilities	Collaborators
Activate and track states for all input devices	-
Get player tap events	-
Get player drag-and-drop events	-
Get player swipe events	-

Scenes

MenuScene

Description: This class is responsible for displaying our game's main menu, which allows players to host and join games.

Justification: This controller is important for handling the state of hazards in the game, including spawn rate, spawn time, and whether hazards are to be destroyed.

Responsibilities	Collaborators
Initialize scene UI elements	-
Set that host button was pressed	-
Get if player should switch to HostScene	-
Get if player should switch to ClientScene	-
Play audio	AudioController

HostScene

Description: This class is responsible for initializing network controllers, and provides an interface for hosting the game. It also provides the UI elements to make game modifications.

Justification:

Responsibilities	Collaborators
Initialize and manage the scene's UI elements (buttons, text fields, labels	-
Get whether player should move to DraftScene	-
Get whether player should move to	-

Responsibilities	Collaborators
MenuScene	
Display game ID and players data	NetworkController
Play audio	AudioController

ClientScene

Description: This class is responsible for initializing network controllers and provides an interface to join a game. It also provides the UI elements for players to choose characters.

Justification:

Responsibilities	Collaborators
Initialize and manage the scene's UI elements (buttons, text fields, labels)	-
Get whether player should move to DraftScene	NetworkController
Get whether player should move to MenuScene	-
Display game ID and players data	NetworkController
Play audio	AudioController

DraftScene

Description: This class is responsible for displaying the deck construction scene.

Justification: This class is necessary to represent the deck construction phase before a game.

Responsibilities	**Collaborators**
Initialize and manage the scene's UI elements (buttons, text fields, labels)	-
Initialize the player’s deck	GameState
Get if player should move to GameScene	-
Get if player should move to MenuScene	-
Play audio	AudioController

GameScene

Description: This class is responsible for displaying the main gameplay scene. It sends player actions to the turn controller, and displays the turn resolutions as well.

Justification: This class is necessary to represent the main gameplay loop. Necessary for sending player information.

Responsibilities	**Collaborators**
Initialize game, including scene graph and core controllers	TurnController
Update player data and timer	NetworkController
Get if player should move to WinScene	-
Get if player should move to MenuScene	-

Responsibilities	Collaborators
Play audio	AudioController
Detect gestures to interact with UI	InputController

WinScene

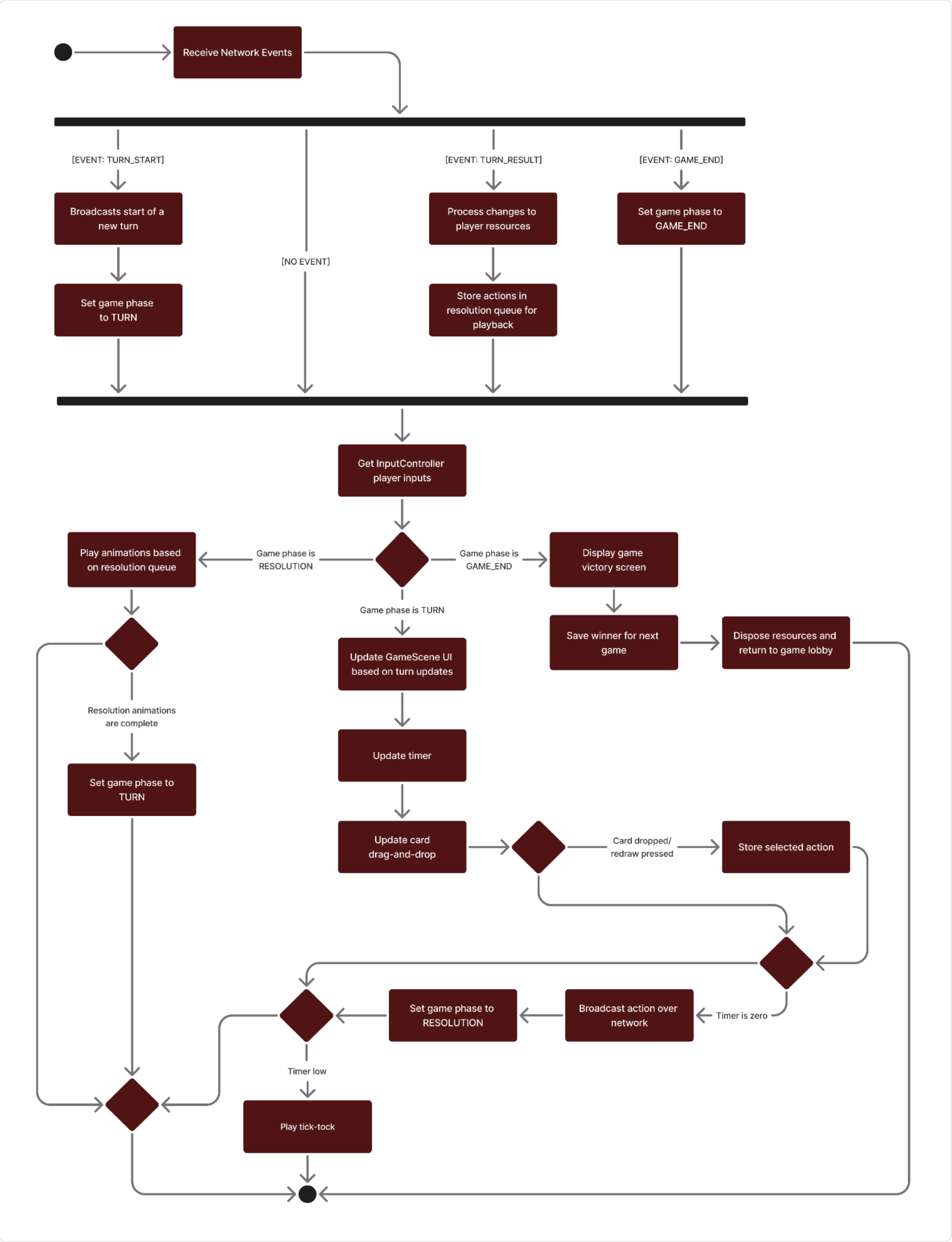
Description: This class is responsible for displaying the win scene.

Justification: This class is necessary to represent the scene associated with the end of a game.

Responsibilities	Collaborators
Initialize and manage the scene's UI elements (buttons, text fields, labels)	-
Display the information related to the end of a game	-
Play audio	AudioController

Game Activity Diagram

Activity Diagram


Receive Network Events
[EVENT: TURN_START]
[EVENT: TURN_RESULT]
[EVENT: GAME_END]
Broadcasts start of a new turn
Process changes to player resources
Set game phase to GAME_END
[NO EVENT]
Set game phase to TURN
Store actions in resolution queue for playback
Get InputController player inputs
Play animations based on resolution queue
Game phase is RESOLUTION
Game phase is GAME_END
Display game victory screen
Game phase is TURN
Save winner for next game
Dispose resources and return to game lobby
Update GameScene UI based on turn updates
Resolution animations are complete
Update timer
Set game phase to TURN
Update card drag-and-drop
Card dropped/ redraw pressed
Store selected action
Set game phase to RESOLUTION
Broadcast action over network
Timer is zero
Timer low
Play tick-tock

Data representation model

Decks

The decks are stored in a JSON file `jsons/decks.json` in the assets directory. This file has the following structure:

```
{
   "pickpocket": [
      "SHOOT",
      "SHOOT",
      "SHOOT",
      "SHOOT",
      "SHOOT",
      "REFLECT",
      "STEAL",
      "STEAL",
      "STEAL",
      "STEAL",
      "STEAL",
      "STEAL",
   ],
   ...
}
```

Networking Protocol

For our networking, we have several different messages sent over the network. These messages are serialized into JSONs for transmission and deserialized into event structs. We will have two types of messages:

- Client to host: this type of message is only sent to the host.
- Host to clients: this type of message is sent to all clients (including the host)

We have seven types of events:

1. GAME_START - enter game state from lobby
2. TURN_READY - player reports ready for turn to host
3. TURN_START - host gives player turn timer
4. TURN_ACTION - player reports turn to host
5. TURN_RESULT - host sends playback to players
6. GAME_QUIT - host or player quits, response depends on sender

7. GAME_END - natural game end
8. PLAYER_DROPPED - remove this player's uid as they have left the game

And also nine types of actions:

1. NO_ACTION
2. RELOAD
3. SHOOT
4. REFLECT
5. STEAL
6. SPLIT_SHOT
7. BARREL
8. HELPING_HAND
9. REDRAW

Each event must be associated with a JSON value `data`, as well as a string representing the person who sent the event, and an address type representing the type of the person (Client or Host). The particular form of `data` depends on the type of event being sent.

For TURN_START, the host sends:

```
{
   "timer": 10
}
```

Where “timer” is set to some predetermined length of the turn. In this example, each turn is ten seconds. The host sends this to every client, so the clients can initialize their own timers.

For TURN_ACTION, the clients must send what action they chose to the host. The `data` needs to include the action taken by the player, the target of this action (if necessary) and the current state of the player. The `data` associated with this event will have this general structure. This is sent from clients to the host.

```
{
   "action": 0,
   "target": "36ebe11e-0eb7-4ecd-8db6-d87b7580c6e0",
   "state": {
       "ammo": 1,
       "lives": 4,
       "cards": 2
   }
}
```

For TURN_RESULT, the host sends the correct sequence of actions taken and updated player states to the clients. Each client will update their game states corresponding to the data in the event. The `data` associated with this event will have this general structure.

```
{
   "actions":
   [
       {
           "to": "",
           "from": "36ebe11e-0eb7-4ecd-8db6-d87b7580c6e0",
           "action": 1
       },
       {
           "to": "36ebe11e-0eb7-4ecd-8db6-d87b7580c6e0",
           "from": "848ef7be-0c4d-40ac-ad67-657b43317b3b",
           "action": 2
       }
   ],
   "states": {
       "848ef7be-0c4d-40ac-ad67-657b43317b3b": {
           "ammo": 0,
           "lives": 3,
           "cards": 2
       },
       "36ebe11e-0eb7-4ecd-8db6-d87b7580c6e0": {
           "ammo": 1,
           "lives": 2,
           "cards": 2
       }
   }

}
```

For all other events, we send an event with `data` that is empty.

Application Restoration Plan

When the app is suspended, Trigger Happy runs in the background with muted audio. The system saves the game state and network details while pinging the host to maintain connectivity. Other players see a visual cue on the disconnected player's portrait.

If a player disconnects due to a crash or onShutdown(), a 2-turn countdown begins. Reconnecting within this period resumes normal gameplay; failure results in removal. Since matches are short, rejoining mid-game isn't allowed. During the countdown, the player takes no actions. Afterward, they can start a new game with the previous party.

When the application triggers onResume(), upon resumption of the application, audio is restored and connectivity is verified to synchronize the player's state with the ongoing game session.