Quantum Battleships Design Document
Laura Beaulieu

1). General gameplay rules

QB is a two person competitive game in which the goal of each player is to destroy the opposing player's fleet. Each player has a fleet of warships. Each ship has a number of hull modules which can either be dead or alive. Some hull modules have turrets with which to attack opposing ships. If the hull module to which a turret is attached to is destroyed, then the turret can not be used. Each player's fleet is composed of the following ships:

- 1 Corvette, which has one hull module and one turret
- 1 Cruiser, which has two hull modules and one turret on the front module
- 1 Battleship, which has three hull modules and two turrets, one on the front module and the second on the rear module.

Each turn, each player simultaneously chooses to make a number of shots on each on the opposing player's vessels. The number of shots a player can take per turn corresponds to the number of active turrets each player has. When each player finalizes their decisions, a quantum circuit is activated that simulates what happens to the ships, such that four hits on a single ship guarantees at least one module is destroyed at random. Due to the quantum nature of the simulation, a module may not be permanently dead. However the number of dead modules per ship never decreases, so a dead module may be randomly revived at the cost of a living module on the same ship. Therefore once all modules on a ship are dead, that ship is permanently out of action for the rest of the match.

The game is over when at most one player of the two players has their ships still alive while at least one other player has all of their ships eliminated (note: win condition is worded weirdly in order for a single winner to be declared if a version with more than two players is implemented).

2). Backend implementation

Game logic is handled by two scripts. One of which is written in GDscript for the godot game engine. This script manages the logistics of user inputs, game states, menus, score keeping, updating graphical representations, etc.

The second script is written in python. This is so that the IBM QISKIT system can be used for quantum computation, either as a local simulator or via cloud computing on real systems connected to the IBM QISKIT network. Simulation rules go as follows:

- Each hull module is assign a qubit such that |0> corresponds to an alive state and |1> corresponds to a dead state
- Let $|m(n)_j>$ represent the qubit corresponding to the j'th hull module of this ship of size n.
    - Let m(n) simply notate the set of all qubits corresponding to hulls on that ship
    - (notation breaks down if there are multiple ships of the same size, which is fine since each ship per team is of a unique size).
- Let f(m(n), s) be a unitary operator on m(n) such that it returns the entangled state for m(n) after it has been shot an additional s times.
- At the beginning of the simulation for each team, each m(n) is initialized to a bell state that corresponds to having the number of destroyed modules that it began the turn with
    - For the corvette, this is simply |0> if it is alive, not simulated if dead
    - For the cruiser, this is the bell state: $|\psi^{01}> = \frac{1}{\sqrt{2}}(|01> + |10>)$ if one module is dead. Otherwise |00> if alive, not simulated if dead.
    - For the battleship, |000> if fully alive, not simulated if dead Otherwise:
        - $|W_3> = \frac{1}{\sqrt{3}}(|001> + |010> + |100>)$ if one module is dead
        - $|\text{-}W_3> = \frac{1}{\sqrt{3}}(|011> + |101> + |011>)$ if two modules are dead
- *f(m(n), s) is applied to m(1), m(2), and m(3) such that a new quantum state is generated with the following constraints for choice of f:*
    - *f(m(n), 0) is the identity transformation*
    - *f(m(1), 4) = |1> if* $m(1)_1 = |0>$
    - *f(m(1), 2)* $= \frac{1}{\sqrt{2}}(|0> + |1>)$
    - *f(|00>, 4) = |11>*
    - *f(|00>, 4) =* $|\psi^{01}>$
    - *f(*$|\psi^{01}>$*, 4) = |11>*
    - *f(|000>, 4) = |W>*
    - *f(|W>, 4) = |-W>*
    - *f(|-W>, 4) = |111>*
    - *f(m(n), s) =* $f(m(n), 1)^s = f \circ f \circ \ldots \circ f(m(n), 1)$
    - *Exact formulation for f(m(n), s) is a work in progress*
- All of this is actually not necessary. Just get new health = old health - times shot, then apply relevant state
- A measurement is then made on all m(n), noting how many modules were killed and which ones. This data is then transferred from the python script to the GDscript for game logic. The specificity of each module affects only which turrets are able to fire, while the number of modules killed per ship affects the initialization for the next turn.

- A check is made for if all ships are fully dead. If not, then user input is taken again and another simulation is done until a player is eliminated.

3). Frontend implementation

Another script is used to to procedurally generate 3D models for each ships from an array of premade 3D models for each module of a ship. Additionally, a custom shader graph is used to generate an animated texture corresponding to if a module is dead or alive. (See: GIF file of prototype demonstrating textures being applied to each ship). The textures on each module can be updated to a slower, dimmer version of the animated texture if a module is killed as part of the match. (completed)

4). User interface elements

- An array of sprites on the stop of the screen represent the number of shots left for a player to take in their turn. A sprite will be crossed out if a shot can not be taken. (finished)
- Main menu will have an options button for changing graphical settings and starting new matches. Additionally a text box will be provided in order to input an API key for the IBM QISKIT system which will be stored locally in memory and not to disk unlike most settings for security reasons. (todo)
- A dev console to help with debugging which will appear upon pressing the ` key (WIP)
- In match prompts to tell users which keyboard presses correspond to which actions (todo)

5). Strategy

A proper markov chain analysis can not be done at the current time due to a lack of a definitive definition for the shot function f(m(n), s). However, a preliminary analysis can be done analyzing the probabilities if we assume a player will always shoot four shots at a single target. These calculations also assume completely noiseless quantum computers

- 4 shots on a fully intact cruiser yields a ½ chance of knocking out a turret
- 4 shots on a fully intact battleship yields a ⅔ chance of knocking out a turret
- 4 shots on a battleship with ⅔ health remaining yields a ⅔ chance of having 1 turret knocked out, and a ⅓ chance of 2 turrets knocked out for an average expected value of 4/3 turrets knocked out after two turns.
- 4 shots on a fully intact corvettes yields a 100% chance of knocking out a turret along with the entire ship permanently.

Therefore it would seem that the most rational first move would be to knock out the corvette given the previously mentioned limitations.