1 Motivation

The idea of this projects is two-fold.
For one I want to showcase a quantum program in action which utilizes the "nature" and properties of qubits to implement several useful game mechanics.
Secondly I would like to demonstrate and provide a playful example to educate people on the very basics of qubits and their behaviour when being manipulated, to illustrate that changing one value will directly impact another value.
The game *Q Shall Not Pass!* is a result of this project and available at GitHub. It aims to visually show you how qubits can be "altered" and manipulated in order to match a given position while discovering that the first impression of randomness isn't so random after all.

2 Outline

The purpose of this project is not to give in-depth explanations how Quantum computing and qubits work, although I will at certain parts briefly explain why and how I use certain specifics of quantum computing to achieve the intended game mechanics.

I will first go over the Vision of the game and what its intention is. Then I will continue explaining possible game mechanics which can be implemented using quantum computing. In accordance with the described mechanics I will give an overview of the game and describe the relevant sections where I used and implemented the described game mechanics. Since the time of this project was limited but there is much more to explore, I will conclude this report with an outlook for future work and how this game can be improved even more.

The project itself contains this report as well as the complete project on GitHub.

3 Game Mechanics

This section will describe the intention of the game, the utilized mechanics and how they were implemented using quantum computing.

3.1 Player and Gate creation

The three variables of each qubit $\langle X \rangle$, $\langle Y \rangle$ and $\langle Z \rangle$ are all in the range of -1 and $+1$ and are constrained to be on the surface or inside the blochsphere. Giving us ultimately the constraint of:

$$\langle X \rangle^2 + \langle Y \rangle^2 + \langle Z \rangle^2 \leq 1.$$

So you cannot have an expectation value where all three of those variables are equal to 1. Since a quantum computer will naturally obey this constraint, we can now think of problems or scenarios we want to map onto these variables.

The original idea of the game was to use the natural contraints of a qubits expectation values to model its proportions.

Specifically I wanted to use the expectation values of each variable to alter a character in a given way.
Since we have this constraint, the character will always have proportional values.
So the initial attempt was to create a character using the $\langle X \rangle$, $\langle Y \rangle$ and $\langle Z \rangle$ expectation values.

In that sense that $\langle Z \rangle$ corresponds to the characters height, $\langle X \rangle$ to it's width and $\langle Y \rangle$ to its race or color. Due to our constraints we will always have a character which is in itself proportional. Although this would result in rather odd looking characters it will still be beneficial and easy to use this mechanic to automatically create seamlessly "randomized" character which are rather proportional in their proporties.
This type of generation uses the expectation values as described above and can of course be applied to an arbitrary level design which could get progressiviely harder, generating new puzzles and different workload that obeys the natural constraint I mentioned in the previous section.
This works without actually having to implement the level generation logic for each level, giving a new experience for each play through. This not only in terms of map generation but again also in variety of player, NPCs or level complexity.

3.2 Continious level progess

As we already discussed in the procedural generation, we can use a Rotation-Gate to adjust a qubit, by an angle of θ in a specific direction/around a specifc axis. Since the state of a qubit can be more visually described by a bloch-sphere, we will use this to illustrate the change of the vector when applying the R_x gate. The application of R_x is essentially the rotation around the x axis by a defined angle θ.
To use this property for our advantage we could for example aim for one of our qubits to be $|1\rangle$ as the intended target state. In order to rotate towards that target we could simply rotate around the x-axis by applying a number of R_x gates to our qubit which is initialized in state $|0\rangle$ until we are at the state $|1\rangle$.
With that in mind we could now design levels that requires you to achieve a number of tasks. For that we could call this qubit our progress-qubit. If our player for instance would need to do 4 tasks per level, we could simply define our operation per tasks equal to $\frac{pi}{4}$, leading to a rotation of $\frac{pi}{4}$ around the x-axis after each successful task, then resulting in a rotation by pi after exactly 4 tasks and transforming $|0\rangle \rightarrow |1\rangle$. These exemplary steps are visualized in figure 1 from left to right.

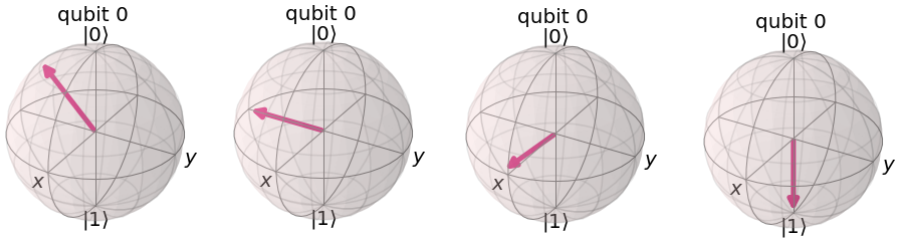

Figure 1: Progress of a state visualized on a bloch sphere by a rotation of $\frac{pi}{4}$ around the x-axis

Of course this behaviour can be altered in any way, creating the possibility to do larger tasks leading to a higher rotation and several small tasks applying only small rotations, all with the same intention to tranform your state of $|0\rangle$ towards $|1\rangle$. Along the same logic we could rotate backwards when the player did something that was not beneficial towards reaching the goal, more or less like applying a penalty for an action.
Now in order to obtain an automatic level progress system as described above, without hard coding

each progression step, we can define two-qubits: one serving as control-qubit/progress-qubit basically tracking the progress of your tasks to reach your goal and a target-qubit which is monitored and triggers the actual level progression.
We start by initializing these two qubits as $|00\rangle$. Each time the player fulfils a task the first qubit is rotated around the x-axis by a predefined θ which is given as:

$$\theta = \frac{pi}{\#tasks}$$

After each step a $CNOT/C_x$ is applied onto these two qubits using the first one as the control and the second one as the target, to check if the level conditions are already met.
This is resulting in a state flip of the target bit if our control bit was $|1\rangle$, meaning the $CNOT$ would apply a logical NOT to the target bit and flip its state from $|0\rangle$ to $|1\rangle$.
Or in other words it will apply an X-Gate on the second qubit if the first one is $|1\rangle$. Since the X-Gate is a special case of an R_x-Gate which basically rotates the qubit around its x-axis by an angle of θ what in the case of an X-Gate is a rotation by π.
Consequently doing nothing when the control is $|0\rangle$ and therefore not progressing forward

$$CNOT|00\rangle$$

$$|00\rangle$$

And actually tranforming the target when it is $|1\rangle$ to then automatically proceed to the next level

$$CNOT|10\rangle$$

$$|11\rangle$$

This type of mechanic can of course be applied to a number of different games. For instance it would also be possible to create a game to find a specific number of clues, for example 4 out of 7, and then automatically progressing to the next stage since applying $\theta = \frac{pi}{4}$ would do exactly that. This all using the nature of qubits without actually having to specify specifics on how a level needs to be solved and when the winning condition is reached.

4 Game Overview

Since the code of the game goes further than pure quantum logic I will only go through the relevant parts, where I actually used quantum computing as explained in the section Game Mechanics.

The game itself is quite simple, you start as a small black arrow which proportions are automatically generated using the expectation values of the variable $\langle X\rangle$, $\langle Y\rangle$ and $\langle Z\rangle$. The game essentially provides you with two tasks per level. One is adjusting your players proportions, which are respectivily the expectation values of the three variables in order to match the proportions of the gate which is also generated automatically at each level. The second task is to gather snacks along the way before you actually can pass through the gate. If you either forgot to collect all snacks or you did not match the gates proportions sufficiently, you'll lose health and the level is restarted, otherwise you will proceed to the next level.

4.1 Map Generation

In order to draw the map a QuantumCircuit is generated at the beginning of the game containing one qubit. This logic essentially uses the normalized global positional x- and y-coordinates of the entire game screen to generate three rotation angles for each axis. These three generated rotations are subsequently applied to the single qubit which is then simulated using microqiskit to retrieve a probability.
This probability is mapped onto a list of terrain items to procedurally generate the underlying map, causing a generation of a coherent map (This logic was not implemented by myself but used as is, from the explanation and examples inside the Qisge tutorial). The map generation is only executed when the game starts or when a new level is generated. But since this is done using the positional x- and y- coordinates that the player has already traveled, it suggests to the user a sense of a continious and coherent map to discover, with connected and related sort of biomes.

4.2 Player and Gate generation

The same algorithm is applied to the player generation.

The gate as well as the player are always regenerated and redrawn when a new level is loaded. The original game idea suggested to use the expectation values to define proportional values for the character, creating a new styled character at every run. Unfortunately since the manipulation of Sprites inside Unity together with Qisge is limited and therefore couldn't be used to dynamically alter the height and the width of a character together with the sprites color, I decided to use the expectation value of $\langle Z \rangle$ as the player´s height and overall size of the sprite, $\langle Y \rangle$ as the rotation-angle by essentially scaling it to a degree between 0 and 360 and $\langle X \rangle$ to the width of the player. Although the width isn't really visualized in the same way as size and rotation, it can still be used together with the remaining gates to rotate around the axis modifying the expectation value and using it as some kind of "helper", for example when using the Haddamard-Gate to map operations in z-axis to x-axis and vice versa.
Also while the map and the gate are generated using the global positions of x- and y-coordinates, the player uses its own x- and y-coordinates which are bound to the visible screen.
This leads to a new generation of player and the opposing gate to match yourself with at each level, giving players the opportunity to discover new challenges with every run without actually having to hard-code each levels specific achievememnts and proporties.

4.3 Player modification

In order to align you self with the gate you want to pass through, you need to modify your player which is essentially represented by a single qubit. This qubit can be altered using rotations around the Z,X and Y axis or by applying a Haddamard gate.
In order to have more flexibility in rotating the vector, the rotation angle: θ, used in the R_x-, R_y- and R_z-Gates is equal to $\frac{2\pi}{8}$

4.4 Level progression using snacks

The idea of the level progression was already described in section 3.2.

This is done using a QuantumCircuit containing two qubits. The number of tasks a player has to fulfil per level is given by the number of snacks to eat which again depends on your current level. The target is always the same. You want to transform your initialized $|00\rangle$ first into $|10\rangle$, by applying a specific number of rotations around the x-axis, which then also triggers the $CNOT$ to ultimately transform the state into $|11\rangle$, resulting in proceeding to the next level.

5 Outlook

Altough some implementations and mechanics are rather constructed, this project still showed different capabilities on how to use quantum computing to implement specific game mechanics which can actually be useful. These could all still be extended even further providing different approaches or extensions to develop game mechanics using quantum computing.

5.0.1 Mapping Color problems to expectation values

Along the idea of proportional character generation by maping the height, width and race of a player onto the three variables $\langle X\rangle$, $\langle Y\rangle$ and $\langle Z\rangle$, we can also could think of a lot of different scenarios for this kind of problems. Since the RGB color space is essentially build out of three color channels, it might not come as a surprise to also use this problem and map it onto our three expectation values. These could be scaled to something between 0 and 255, and then you could again use the nature of a single qubit to serve as an input for a color generation scheme.

5.1 Map generation

In this project I used a random generation around the behaviour of qubits on the base of the paper on procedural map generation and the given example of Qisge.

Although this may seem constructed, we could extend this map generation behaviour using a QuantumBlur effect on top of the current procedural generation. The QuantumBlur effect is used and described on the official GitHub Page, which essentially can generate height maps out of existing quantum circuits and vice versa, generating quantum circuits out of height maps.
Since this is done by enconding an image as a quantum circuit, manipulating it and turning it back into a modified image, we could use this to effectivly taking our current map as input applying a few gates on it to turn it again into a height map. This could serve as an input of using the highest/brightest points of the blurred image to mark certain milestones in the level, to build a maze or a predefined path a player or enemy can take to reach a certain position on the map. Due to the already coherently generated underlying map, which now serves as input for milestones, this could again suggest a slowly progressing level system, which is derived from the underlying map.

5.2 Level progression

The level progression in this game could be made even easier without separetly generating the gate and the player, but instead starting with the inteded state of the gate and then applying several rotations backwards giving you the initial state of a player. Although that would take a lot of fun away trying to puzzle your way towards a solution, it would be more beginner friendly without starting to randomly rotating the gates until it rougly looks the same.

5.3 Player to Gate alligment

In addition to the last point: In the current implementation, the qubits for player and gate are generated separetly and are compared using their specific expectation values. Using a tolerance-threshold this is fairly simple to implement and use in your game.
A more interesting approach would be to actually extend the usage of QuantumGraph and utilize the given capabilities of adjacent qubits in the graph to align the player with the gate, without actually having to compare player and gate proportions.
In that case we again have the opportunity to use the nature of quantum computers to do something useful for us, without actually having to tell them what specific goal to reach, for example trying to reach a somewhat equal relation between the two qubits in order for you to pass the gate.