Flying Unicorn: Developing a Game for a Quantum Computer

Kory Becker
kbecker@primaryobjects.com

Abstract

What is it like to create a game for a quantum computer? With its ability to perform calculations and processing in a distinctly different way than classical computers, quantum computing has the potential for becoming the next revolution in information technology. Flying Unicorn is a game developed for a quantum computer. It is designed to explore the properties of superposition and uncertainty.

Keywords Quantum computing, quantum physics, quantum mechanics, quantum, qubit, emerging technology, programming, Qiskit, IBMQ, IBM Q Experience

1. Introduction

Quantum computing is a technology that uses the properties of quantum mechanics, including superposition and entanglement, in order to perform computations significantly faster than traditional computers, which are based upon transistors [8]. While classical computers utilize bits holding a single value of zero or one, a quantum computer encodes information in quantum bits, or qubits, which represent the value of both zero and one simultaneously [2]. With this unique difference, a quantum computer is able to execute computational calculations in an entirely new way, exceeding the time complexity performance of classical algorithms in areas of search and encryption [14]. The potential exists to apply this technology to a large range of applications, including databases, mathematical calculations, machine learning, and games.

1.1 Quantum Computing

Quantum computing takes advantage of the properties of quantum mechanics in order to represent bit information and perform calculations. Rather than measuring the voltage across a transistor for indicating a value of low or high, a quantum computer uses a measurement, such as spin on an electron or photon, resulting in a calculation of zero, one, or a probability between, also called superposition [3].

Representation of Data While a classical computer can represent one bit of information per bit, a quantum computer can represent two bits of information per bit [16] (also called a qubit). Similarly, two qubits can represent four bits of information. By effect, while a classical computer can represent n-bits of information during a single CPU cycle, a quantum computer can process 2^n bits of information per CPU cycle [17]. This has an exponential effect in the amount of data that can be processed. For example, 50 qubits can represent 2^{50} bits. To put this value into context, consider that 1 petabyte is equal to 10^{15} bytes. A quantum computer with nearly 50 qubits can represent this magnitude of data in a single calculation. If a social network stores data for 2 billion records, within the range of 500 petabytes of data, a quantum computer with 60 qubits could represent this amount of information.

This paper makes the following technical contributions:

1. We present Flying Unicorn, a game developed for a quantum computer using the Quantum Information Science Kit framework, demonstrating the usage of quantum computing concepts.
2. We present some observations, including differences between error rates and execution speed of a quantum simulator running on a classical computer compared to a quantum computer running at IBM Q Experience.
3. We provide demonstrations of quantum techniques for random number generation and Grover's search algorithm in an easily understood format of implementation within a game.

2. Methodology

Flying Unicorn [1] is a text-based game developed using Python and the Quantum Information Science Kit [11] (Qiskit). An example is shown in Figure 1. It was designed to demonstrate the usage of quantum computing's properties, including qubit measurement, superposition, and uncertainty. In the game, the player controls a unicorn in an attempt to fly up into a castle. The game allows real-time execution on a quantum simulator or on a physical quantum computer hosted by IBM Q Experience [10] (IBMQ).

```
Your unicorn, Pixel Twilight, is ready for flight!
Use the keyboard to fly up/down on a quantum computer,
as you ascend your way into the castle.

====================
-[ Altitude 0 feet ]-
Pixel Twilight is waiting for you on the ground.
[up,down,quit]: u
You soar into the sky.
Running on the simulator.
{'0': 944, '1': 56}

====================
-[ Altitude 56 feet ]-
Pixel Twilight is floating gently above the ground.
[up,down,quit]:
```

Figure 1. Excerpt from Flying Unicorn after one turn. The player has reached an altitude of 56 feet.

2.1 The Design of Flying Unicorn

Flying Unicorn's game-play is turn-based, where the player can choose an action at each round. Depending on the state of the player, a corresponding message is rendered on the display. As the player transitions to various altitudes throughout the game, the player's status message is adjusted respectively. Flying Unicorn relies on a traditional game loop in order to allow the player to choose an action in each round. However, unlike a traditional game, which maintains player state information via variables based upon random access memory, Flying Unicorn maintains state via measurement of a qubit.

Generating a Player Name The initial stage of game play begins with the generation and assignment of a randomly generated name. The name will be used within the status messages as the player transitions through the game. A player name consists of two parts, first and last name, which are selected from a predetermined list of name fragments. In order to select the fragments to compose the name, two randomly generated numbers are selected through a quantum process. This is described in the "Quantum Random Number Generation" section.

Representing Player State Player state is represented by a qubit which indicates the current altitude of the player's unicorn, as shown in Figure 2. The state is used for selecting the status message displayed to the user, in addition to determining the goal state of the game. At each turn, the player chooses an action to move up or down, which adjusts the player's altitude accordingly. To create a random effect to the amount of change in altitude that the player receives, the spin state of a qubit is utilized on a range of zero to one. A value of zero corresponds to the player's starting state, and a value of one is the goal state.

```
unicorn = QuantumRegister(1)
```

Figure 2. Representing player state as a qubit in the Qiskit framework.

Partial Qubit Inversion By inverting a qubit from its ground state of zero, and using a partial inversion, we can gradually adjust the resulting qubit measurement to correspond to how close the player is to the goal.

Upon applying an invert gate (x), the qubit's state changes from zero to one. To apply a fraction of an invert, we utilize the partial NOT $u3$ gate, which places the qubit into superposition and applies a partial inversion operation, forcing the qubit value closer towards one. The closer the player is to the goal, the larger the fraction of the partial inversion operator that we perform on the qubit. If the player has reached the goal state, we apply a full inversion operation on the qubit to move its value from zero to one, resulting in conclusion of the game.

To determine the amount of inversion to apply to the qubit, we add the player's current altitude with a modifier for an action of up or down and divide the result by the goal altitude to obtain a percentage. This percentage indicates the amount of inversion to apply to the qubit. The calculation is shown in Figure 3.

Finally, we perform multiple measurements of the qubit during each turn's program execution and record the number of times the qubit is measured as a value of one. The measurement directly corresponds to the fraction of inversion that we apply to the qubit. Since the nature of quantum computing contains uncertainty and error rates due to decoherence in measurements, this process has the effect of adding randomness to each turn's action. When the resulting qubit's measurement plus the modifier for the player's action reaches or exceeds the goal altitude, the player wins the game.

```
frac = (altitude + modifier) / goal
if frac >= 1:
  program.x(unicorn)
elif frac > 0:
  program.u3(frac * math.pi, 0.0, 0.0, unicorn)
```

Figure 3. Applying a partial NOT gate to the player's state.

2.2 Superposition of Player State

Player state corresponds directly to the measurement of a qubit, with values ranging from zero to one. The player begins at a ground state of 0 and wins the game when the number of measurement outcomes for 1 exceed the winning threshold. This threshold is set to the total number of measurements (1024), corresponding to 100% of all outcomes in a program execution on the simulator, and slightly less on IBMQ, to account for error.

At each state in-between the ground and winning states, the qubit is placed in superposition using the $u3$ partial inversion gate. This results in the qubit having a linear combination of the values of 0 and 1. When a measurement occurs, superposition of the qubit collapses [15] [18], resulting in an outcome value of 0 or 1. Using ket notation, these two values are described by the computational basis states $|0\rangle$ and $|1\rangle$. For example, if the player has reached an altitude of 25% to the goal, after performing $1,000$ measurements of the qubit, we see 250 outcomes for 1 and 750 outcomes for 0 (corresponding to a qubit inverted by 25%). We leverage the number of outcomes holding the value of 1 to represent the current altitude for the player.

The probability for a qubit outcome to result in a value of 0 or 1 corresponds to the amplitude coefficient applied to each state [13]. The game applies this coefficient through the $u3$ partial inversion gate. Using the prior example of a player at 25% to the goal, we can apply an amplitude of 0.5 to the 1 state and 0.87 to the 0 state ($0.87\,|0\rangle$ and $0.5\,|1\rangle$). The sum of the squares of the amplitudes will always equal 1 ($0.87^2+0.5^2 = 1$), which we leverage as a percentage within the game to represent the altitude for the player with respect to the goal. In this manner, by applying the appropriate amplitude coefficient to the computational basis state for $|1\rangle$, we can obtain the desired altitude for the player. In this example, $0.5^2 = 0.25 * 100 = 25\%$, as shown in Figure 4.

As the player flies upward towards the goal, we increase the amplitude coefficient for the $|1\rangle$ state, respectively increasing the number of outcomes resulting in a value of 1, and thus, the resulting altitude of the player with respect to the goal. As the player's altitude changes, so too does the player's state, in accordance with the predefined list of status messages, as shown in Figure 5.

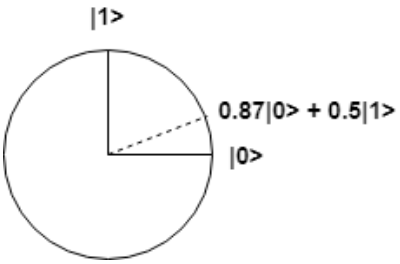

Figure 4. Representing player altitude at 25%. An amplitude coefficient of 0.87 is applied to the $|0\rangle$ state and 0.5 to the $|1\rangle$ state, resulting in a 25% probability ($0.5^2 * 100$) of a computational basis outcome of $|1\rangle$.

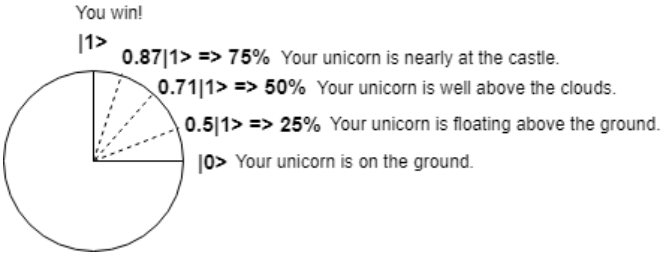

Figure 5. Selecting the player status message from the measurement of a qubit. The amplitude applied to the $|1\rangle$ state is created through partial inversion. Probability is calculated by squaring the amplitude ($0.71^2 = 0.5 * 100 = 50\%$).

2.3 Quantum Random Number Generation

As part of the process for selecting a random name for the player's unicorn, a quantum random number generation process is utilized, based upon the uncertainty of a qubit in superposition. While the default random number generator on a Unix computer is a pseudo-random generator based upon a seed-based sequence invoked by the rand method [5], quantum random number generation is based upon the random properties of quantum mechanics, leveraged by the uncertainty of a qubit in superposition [7].

A qubit is placed into superposition by executing the Hadamard gate [6]. This results in a qubit having equal probability for a measurement of zero, one, or a value in-between.

Several methods were analyzed for producing a random number within the 20 qubit limitation of IBMQ. Each method exposed trade-offs with respect to memory, performance, and allocation limits, as listed in Table 1.

One Qubit A naive approach for generating a random integer is to utilize a single qubit in superposition and measure its resulting value. A single qubit would correspond to one bit in an integer value. Since one qubit has an equal probability of resulting in a value of 0 or 1, the maximum integer value of one qubit is 1. Likewise, the maximum value for 2 qubits is a binary value of 11 or a decimal value of 3. This approach is simple to implement. However, as the maximum integer grows larger, so too do the number of required qubits. The naive approach requires n qubits to represent a maximum integer of $2^n - 1$.

The approach generates one bit in a random number for each execution of the quantum program. Since only a single qubit is used, this method can generate an infinitely large integer. IBMQ offers up to 20 qubits for processing on their publicly offered machines [9]. This approach remains well within these constraints. However, each represented bit within the integer requires the allocation and measurement of a qubit, resulting in multiple executions of the program (once for each qubit). This results in poor performance on the quantum simulator and on the physical quantum computer at IBMQ, where wait times often exist prior to executing a program.

Multiple Qubits An alternative method for number generation is to allocate multiple qubits during initialization, one qubit for each bit to represent in the integer. We then perform a single measurement and execute the program. This method alleviates the wait time delay with executing the program

multiple times on the simulator and IBMQ. Since only a single measurement is being performed, only one bit sequence combination will be returned, corresponding to an integer. However, as the integer grows larger, the number of qubits required within a single program exceeds the limitation of 20 qubits by IBMQ. This results in a maximum integer value of $1,048,575$ represented by 20 bits (or 20 qubits).

In addition to the maximum integer value that can be generated, qubits require additional memory and CPU processing time within the simulator. Multiple qubits can increase program execution time on the simulator and require additional memory for processing, eventually exceeding memory available on the computer.

Probabilistic Measurement The selected approach to random number generation, which takes into account limiting the number of qubits required, as well as requiring only a single program execution, leverages the probabilistic nature of quantum computing [19]. Since each qubit in superposition can simultaneously hold a value of zero and one, all potential combinations of bit values can be represented during a measurement, provided enough measurements are performed to capture all outcomes. Given q qubits, this approach can represent an n-bit integer containing 2^q bits. This allows the representation of a 4 bit integer using 2 qubits measured multiple times with one run of the program.

Consider the combinations possible with 2 qubits in superposition. We can measure the qubits and receive 4 possible outcomes (00, 01, 10, 11). By using a sufficient number of measurements to record counts for each occurrence of the various outcomes, and mapping each outcome to a bit in the integer, we can select a value of 0 or 1 depending on whether the count is greater than the average probability. The average probability is calculated as the number of measurements divided by the number of outcomes (2^q). An example of this process is shown in Figure 6. Using this method, we only require 2 qubits and 1 run of the program to represent a 4 bit integer. Taking into consideration the maximum of 20 qubits allowed on IBMQ, we can generate a maximum integer value comprised of $1,048,576$ bits. This method offers increased execution speed with limited overhead for generating a sufficiently large random number.

Table 1. Quantum random number generation.

Qubits	Measures	Runs	Result
1	1	4	Slow. Max value infinite.
4	1	1	Memory and CPU limitation. Max value $1,048,575$.
2	1000	1	$2^q = n$ bit integers. Max bits $1,048,576$.

2.4 Quantum Search Guessing Game

During each turn, the player may encounter an optional random event, where they have the opportunity to receive a bonus or penalty, depending on whether they are able to guess a secret jewel before a quantum computer player.

```
Qubit measurements:
{'10': 23, '11': 28, '01': 24, '00': 25}
Average probability: 25
Bits: [0, 1, 0, 0]
Integer: 4
```

Figure 6. An example of measuring two qubits to generate all possible outcomes in the generation of a four-bit integer.

Grover's Search Algorithm The guessing game invokes Grover's search, a quantum search algorithm that can locate an element within an unordered list with a high degree of probability, faster than any classical search algorithm [4]. Grover's search can locate an element by using properties of quantum computing within $O(\sqrt{N})$ evaluations. By contrast, a classical algorithm can solve the task in $O(N)$ evaluations by iterating over each combination of input values until the correct sequence is found.

Grover's search algorithm is implemented within the game by using all combinations of 4-bits as the unordered set of 2^n states [12] (16 states, resulting in the possible values $0-15$). An oracle function is implemented that returns 1 for the target random number and 0 for all other states. The algorithm then manipulates 4 qubits and 1 program execution to perform Grover's search. The result of the program returns a list of all combinations of bits, along with outcome counts corresponding to how many times the particular state was measured. The maximum outcome is used as the result of the search algorithm, and thus, used as the guess from the quantum computer opponent, as shown in Figure 7.

Within the game, the secret number is represented in the form of a jewel name (amethyst, sapphire, emerald, and jade) from which the player must guess which jewel is the real one.

Offsetting the Quantum Advantage When run on the Qiskit simulator, the computer has a high degree of probability of guessing the secret jewel correctly in a single round, which makes the guessing game more difficult for a human player. For this reason, the player is offered a hint that the random number (i.e., jewel) is within a range of 4 values (refer to Figure 7). This gives the player a 25% chance of guessing correctly. The player is given this advantage even though the computer is searching against the entire range of values. The quantum simulator often guesses correct in just one round, which demonstrates the success of the quantum search algorithm. However, the error rate when running on IBMQ often prevents a successful result from being found.

3. Results

Flying Unicorn demonstrates the representation of game state on a quantum computer. Through the manipulation of a qubit by inversion, the game state is able to be determined

```
Round 1. Which unicorn jewel is the real one?
[amethyst,sapphire,emerald,jade]: sapphire
You guessed wrong.
Measurements after 1 iteration of Grover search:
{'0111': 6, '1111': 3, '0010': 4, '1100': 3,
'1001': 3, '0000': 4, '0101': 2, '1011': 43,
'1000': 2, '0011': 6, '0100': 4, '1110': 3,
'1010': 5, '0110': 3, '1101': 7, '0001': 2}
Maximum outcome: 1011 (11)
The mischievous cloud guesses emerald.
```

Figure 7. An example of running 4-qubit Grover's search, as implemented within the jewel guessing mini game. The secret key 1011 (decimal value 11) is correctly found as the maximum outcome (43). When grouped by four, this results in the secret key $emerald$.

based upon a classical computing calculation with the resulting altitude based upon the measurement counts of the qubit. The game also demonstrates an implementation of a quantum search algorithm that gives a computer opponent the ability to determine a randomly selected number within a single program execution.

3.1 Classical vs. Quantum Computing

Two distinct differences were noted while developing the game using quantum computing, as compared to a traditional computing platform. These are discussed below.

Error Rate The rate of error when performing quantum calculations needed to be taken into consideration during game development. While the error rate was used as a feature of the game in order to add a random factor to the altitude change at each turn, differences between the simulator (which operates at a lower error rate) and IBMQ can greatly affect game play. Specifically, when determining if the player has reached the goal, the increased error rate from IBMQ could occasionally result in the player receiving too low of an altitude gain to be able to complete the game. That is, a full inversion of the qubit via the x gate, would result in a measurement of counts in the 1 computational basis of less than 100% of the total measurements, in which case, the goal threshold would never be reached. As an example, upon applying inversion on a qubit beginning in the 0 state, we would expect a result of $1024/1024$ outcomes having a value of 1, but would instead receive a result of $970/1024$ outcomes with a value of 1 and $54/1024$ outcomes with a value of 0, undershooting the goal state. An error rate buffer was utilized to lower the threshold for the player to reach the goal, compared to that used on the simulator (total number of measurements), as shown in Figure 8. The selected error buffer value of 75 was chosen as a general range in measurement errors from IBMQ, allowing the player to reach the goal once their altitude reaches or exceeds the total number of measurements minus the buffer ($1024 - 75 = 949$).

Additionally, success rates for using Grover's search algorithm in the guessing game differed greatly between the simulator, which could often solve the search in one program execution, and IBMQ, which would often be unable to arrive at a solution.

Execution Speed A significant difference with execution speed was demonstrated between running the game on the simulator versus a physical quantum computer provided by IBMQ as listed in Table 2. Each round of the game requires a quantum program execution, which entails a wait period depending upon the number of preexisting jobs queued. While the Qiskit quantum computer simulator executed programs within a mean of 0.13 seconds, IBMQ demonstrated a longer delay, executing programs within a mean of 79.84 seconds. Execution times differed depending upon traffic on the IBMQ platform, as requests are queued in the order that they arrive. In effect, running a game on IBMQ, particularly one that may rely on real-time graphics, actions, or multiplayer activity, is not yet feasible on publicly available hardware. However, for less time-sensitive tasks, such as cryptography key generation and low frequency algorithms, the latency in response time may be less of a concern.

```
# Buffer to account for measurement differences.
errorBuffer = (75 if device == 'real' else 0)

# Max altitude for the player to reach the goal.
goal = 1024 - errorBuffer

# Number of measurements per program run.
shots = goal + errorBuffer
```

Figure 8. Offsetting the goal threshold to account for measurement differences between the simulator and IBMQ.

Table 2. Execution speed seconds of a simulator vs. IBMQ.

Type	Runs	Min	Mean	Max
simulator	175	0.06	0.13	0.25
ibmqx4	94	53.81	58.22	90.23
ibmq_16_melbourne	33	59.87	141.43	627.30
IBMQ	127	53.81	79.84	627.30

3.2 Conclusion

In summary, quantum computing is an emerging technology, showing promise of significant speed improvements across a range of computational algorithms.

We've demonstrated the usage of quantum computing within a game, including the implementation of superposition, random number generation, and quantum search. Although the physical quantum computing platform at IBMQ differed from the simulator, specifically with regard to measurement error and execution speed, the technology shows promise of allowing the design of algorithms that can take

advantage of quantum properties, greatly reducing processing time when compared to the existing ones of today.

As quantum computing matures and hardware becomes readily available to consumers, the design of software based upon quantum algorithms will become more practical for implementation, exponentially exceeding the performance of classical computing.

References

[1] K. Becker. Flying Unicorn, 2019. URL `https://github.com/primaryobjects/flying-unicorn`.

[2] M. Brooks. *Quantum Computing and Communications*. Springer-Verlag, 1999. ISBN 9781852330910.

[3] I. Buluta, S. Ashhab, and F. Nori. Natural and artificial atoms for quantum computation. *Reports on Progress in Physics*, 74 (10):104401, 2011.

[4] J. Cui and H. Fan. Correlations in the grover search. *Journal of Physics A: Mathematical and Theoretical*, 43(4):045305, 2010.

[5] V. Du Preez, M. Johnson, A. Leist, and K. Hawick. Performance and quality of random number generators. In *International Conference on Foundations of Computer Science (FCS'11)*, pages 16–21. CSREA, 2011.

[6] G. Heald. An analysis of quantum algorithms, measurement and probability, 2019.

[7] M. Herrero-Collantes and J. C. Garcia-Escartin. Quantum random number generators. *Reviews of Modern Physics*, 89 (1):015004, 2017.

[8] M. Humphrey, P. V. Pancella, and N. Berrah. *Quantum Physics*. Alpha Books, 2015. ISBN 9781615643622.

[9] IBM. IBM Q Devices, 2019. URL `https://www.research.ibm.com/ibm-q/technology/devices/`.

[10] IBM. IBM Q Experience, 2019. URL `https://quantumexperience.ng.bluemix.net/`.

[11] IBM. Quantum Information Science Kit, 2019. URL `https://qiskit.org/`.

[12] S. P. Karlsson, Vera B. 4-qubit grover's algorithm implemented for the ibmqx5 architecture, 2018. URL `http://kth.diva-portal.org/smash/get/diva2:1214481/FULLTEXT01.pdf`.

[13] A. Matuschak and M. A. Nielsen. Quantum Computing for the Very Curious, 2019. URL `https://quantum.country/qcvc`.

[14] A. Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

[15] M. Nagy and S. G. Akl. Quantum measurements and universal computation. *International Journal of Unconventional Computing*, 2(1):73, 2006.

[16] E. Rieffel and W. Polak. An introduction to quantum computing for non-physicists. *ACM Computing Surveys (CSUR)*, 32 (3):300–335, 2000.

[17] D. Rothman. *Artificial Intelligence By Example*. Packt Publishing, 2018. ISBN 9781788990547.

[18] C. Seife. Teaching qubits new tricks. *Science*, 309(5732): 238–238, 2005.

[19] V. Silva. *Practical Quantum Computing for Developers: Programming Quantum Rigs in the Cloud using Python*. Apress, 2018. ISBN 9781484242186.