Q-PACMAN Manual

Ugo Louche

September 8, 2016

Abstract

A quick overview of the math and mechanics behind Q-Pacman. Probably highly incomplete.

Contents

1 An incomplete introduction on Quantum Programming

The aim of this section is to give a concise insight into how Quantum Programming is involved in Q-Pacman. More precisely, we will detail in what sense Q-Pacman can be seen as a Quantum Game and describe how the inner mechanics of the game work. Nevertheless, we would like to emphasis that in the end, there is no proper Quantum Computing involved in the sense that, obviously, no modern home computer has the ability to perform operations on quantum bit register. The program we propose here merely *emulate* the behavior of a quantum calculator on a few bits and as such act *as if* a few bits were actually of quantum nature although this is not the case.

Moreover, the present document is by no means a proper tutorial to Quantum programming and therefore we will only go over the notions of Quantum Programming that are relevant to the matter at hand. In other words, the present document is at time overly simplified or inaccurate with respect to Quantum programming, especially with respect to the field that is no interaction Q-Pacman's game logic.

For a proper tutorial on quantum programming, the interested reader may refer to [1] which is incidentally an inspirational source for this Q-Pacman.

1.1 A formalism for classic computing

Roughly speaking, a *regular* bit—that is a bit used by one's everyday computer—can be thought as a 2- dimensional vector v living in a vector space of basis $\{\mathbf{u}_1; \mathbf{u}_2\}$ such that $v = \lambda_1\mathbf{u}_1 + \lambda_2\mathbf{u}_2$ where $\lambda_i \in \{0; 1\}$ for $i = 1, 2$ and $\lambda_1 + \lambda_2 = 1$. In plain English, this means that $\mathbf{v}$ is either equal to $\mathbf{u}_1$ or $\mathbf{u}_2$ and we thus fall back to the classic semantic of a single bit, that is a unit of information being either 0 or 1; thus $\mathbf{u}_1$ and $\mathbf{u}_2$ naturally correspond these two base states and we will use $|0\rangle$ and $|1\rangle$ as a shorthand for $\mathbf{u}_1$ and $\mathbf{u}_2$.

Similarly, a two-bit register can be seen as a 4- dimensional vector in a vector space of base $|00\rangle, |01\rangle, |10\rangle |11\rangle$, which is nothing more than the tensor product of the 1- bit base by itself. The same applies to larger register and in all generality, a n- bit register can be seen a 2^n- dimensional vector which can only be equal to one of the 2^n basis vectors, thus only 2^n values can be represented from a n- bit register.

1.2 Going to the Quantum World

Now, if we were limited to classic computing, the above formalism is arguably an overly complex model of classic computation. Indeed, mapping bits to vectors only to restrict their value to those of the basis' vectors seems a peculiar way to introduce an expressive model only to constrain it in such a way that all its expressiveness is lost.

The relevance of this model though lies in that it is an intuitive and simple enough framework to reason about Quantum Computing. Namely, a *Quantum* bit is a 2- dimensional vector $\mathbf{v}$ living in a vector space of basis $\{|0\rangle; |1\rangle\}$ where $\mathbf{v} \doteq \lambda_1 |0\rangle + \lambda_2 |1\rangle$ such that for $i = 1, 2$ $\lambda_i \in \mathbb{C}$ and $\lambda_1^2 + \lambda_2^2 = 1$. In other words, v is now a vector of complex coefficients living on the surface of the sphere of unit

Euclidean norm. And this is how much more expressive Quantum Computing is with respect to classic Computing.

Namely, we may remember that a single bit could only encode 2 state of information, 0 or 1; comparatively, a quantum bit can encode any (complex) combination of these two base state. In a nutshell, to keep track of a single bit, one only has to memorize which basis vector it is mapped to ($|0\rangle$ or $|1\rangle$) where to keep track of a Quantum Bit, one has to memorize the value of both λ_1 and λ_2 which are complex numbers. Concretely, this means that in order to simulate a quantum register of n bits, one has to keep track of 2^n complex values, each one corresponding to one of the 2^n basis states possible, those being the 2^n possible combination of n regular bits.

1.3 A Story of Probability

Strong of this convenient representation, we may nonetheless ask ourselves what is the semantic behind seeing quantum bits as complex vectors. Remember that in the classic case, the semantic is obvious: if $\lambda_1 = 1$ then the bit is 0, otherwise it is 1. In the quantum case, the quantum bit can be represented by any arbitrary combination of value for λ_1 and λ_2 (as long as the Euclidean norm is 1). The intuition behind that is that a quantum bit is in a superposed state: it is at the same time partly 0 and partly 1 and the value of λ_1 and λ_2 are measures of how these two states are mixed. However, another interesting property of quantum object is that these superposition of states collapse when the Quantum bit is actually measured. That is to say, when measure, a quantum bit will be either 1 or 0 and never both, however, if the process is repeated multiple times, the measurement result may actually differ. Once again, since λ_1 and λ_2 characterize how the two basis states are mixed, they also allow one to predict the likelihood of observing 0 or 1 when a measure occurs.

Formally, given a vector $\mathbf{v}$ representing a quantum bit as defined above, the probability to observe $\mathbf{v} = |0\rangle$ (resp. $\mathbf{v} = |1\rangle$) is $|\lambda_1|^2$ (resp. $|\lambda_2|$). And from the norm constraint, we can see that the probability of observing $|0\rangle$ or $|1\rangle$ is equal to $\|\mathbf{v}\|_2^2 = 1$.

1.4 Introducing Quantum Operators

Now that the basics of how quantum data are represented and behave we may iterate a step further and introduce the concept of computation on quantum bits. Simply put, we may see quantum operations as linear transformation defined over the complex vectors representing quantum bits. Indeed, note that the vector representation we have exposed so far fully accounts for the effects of the Quantum nature of quantum bits, hence we are left with plain ordinary algebra. The only constraint here is that, given a linear operator $\mathbf{U}$ we have to ensure that the result of $\mathbf{U} \times \mathbf{v}$ still carries the quantum semantic discussed above. In other words, $\mathbf{U} \times \mathbf{v}$ must be of unit norm. More specifically, we enforce that any quantum operator is reversible, that is, any operator $\mathbf{U}$ is unitary ($\mathbf{U} \times \bar{\mathbf{U}}^T = \mathbf{I}$)[1].

[1] where $\mathbf{I}$ is the identity matrix and $\bar{\mathbf{U}}^T$ the conjugate transposes of $\mathbf{U}$

1.4.1 A few examples of Quantum Operators

Without going too much into details, we may give a couple of examples of quantum operators. A more complete list will be given at a later point when discussing game play elements.

One of the most fundamental operators we may describe is the Hadamar operator:

$$\mathbf{H}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The particularity of the Hadamar transform is that it map the basis vectors $|0\rangle$ and $|1\rangle$[2] to an equally probable superposition of $|0\rangle$ and $|1\rangle$. Moreover, applying a second time $\mathbf{H}$ maps the result back to the initial input since $\mathbf{H} = \bar{\mathbf{H}^T}$). Note that the Hadamar transform is easily generalized to n Quantum bit by taking its tensor product, as such.

$$\mathbf{H}_2 = \mathbf{H}_1 \otimes \mathbf{H}_1 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

Interestingly enough, the hadamar transform carries it semantic on multiple Q-bits register and applying $\mathbf{H}_2$ on any of the 4 basis vector of 2 Q-bits will result in an equally probable mixture of $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$.

Another interesting 2- bit gate is that swap gate which, as its name implies, swaps the coefficient of $|01\rangle$ and $|10\rangle$ without changing those of $|00\rangle$ and $|11\rangle$. In other words, it swaps the left bit with the right bit, thus mapping 11 and 00 to themselves and 10 to 01 and 10 to 01.

$$\mathbf{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

1.5 A Last Remark

Before closing this section, we would like to emphasize an important limitation of the quantum model that led to some of the design choices we made. By forcing operators to be reversible, we exclude any operator that may loose information along the way. For instance, imagine an operator that take2 Q-bit and collapses them into $|00\rangle$. Such operator would be defined as

$$\mathbf{U} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Such operator is not valid because it is not unitary: its inverse is undefined and the operation is thus not reversible. It is a known result that any program can be made reversible by the addition of ancillary bits acting as a memory of previous state, like a road map of what to do in order to reverse the computation. While

[2]By this we mean the vector $\mathbf{v}$ such that either $\lambda_1 = 1$ or $\lambda_2 = 1$

turning a non-reversible program into one in classic computing can be easily done at the expense of memory, doing so in quantum computing is prohibitive in our case. Indeed, emulating a quantum system means that each new Q-bit introduced double the number of complex value to track. The computational and memory cost of doing so with ancillary bits thus grows exponentially with the number of ancillary bits and quickly become beyond the computational capabilities of our computer. Thus, we have to limit ourselves in our design to naturally reversible operations.

2 Game Play Elements

Before going into the details of how quantum programming interfaces with regular game play elements, we will have to give a broader vision of the elements of the game. There should not be anything really new here as everything is pretty standard and the purpose of this section is merely to state plainly how the game works.

2.1 A brief overview of the game

2.1.1 The Board

The game takes place on a 20 by 32 board of 25 pixel cells. The board thus act as a grid and entities are located by their position on the board rather than their position on screen. The reason behind using a discrete board with such a large cell comes directly from the quantum nature of the game and the need for having a small (a few bits at most) and efficient way to store entities' position in game. Despite the smooth entities movement, the game is thus discrete by nature, which may explain some graphically perplexing collision events and triggers at times.

2.1.2 Entities

Every element in the board that can interact with the player is an entity (*classicEntity* in the source code). Such entity comes in three variations: ghosts, gum and SuperGum. Of those, only one can move, the ghosts. The other two entities are immobile.

3 Quantum Game Play

This section will describe the main data structure that lies behind the quantum nature of the player. Note that this data structure is also shared by other entities in the sense that it is a convenient way to store the state of any entity; The difference being that in the case of the player, this data structure is stored in Quantum bits.

3.1 Internal State

3.1.1 What are internal states

Internal states are the underlying data structure specifying the basic properties of every entity in the game: its position, its direction, and its super state. In a nutshell, an internal state is stored on sixteen bits $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ where:

- The entity's position on the x axis is stored in bits 0 to 4
- The entity's position on the y axis is stored in bits 5 to 9
- The entity's orientation is stored in bits 10 to 11
- The entity's super flag is stored in bit 12
- The bits 13 to 15 have no particular use for now.

Also note that the value0xFFFF is freely used as an *undefined* state for practical reasons. We may also note that since the board has a width of 32 cells and height of 20 not all possible value for the bits 5 to 9 are used. Thus not all configuration of internal state correspond to proper position on the board. Finally, an entity's orientation is one of the four following values: *NORTH, EAST, WEST, SOUTH*; these values are respectively coded by $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. In particular, *NORTH* and *SOUTH* are both obtained from one another by applying a NOT gate, whereas one can go from/to *EAST* to/from *WEST* through either a NOT or SWAP gate.

Once again, note that internal states are commonly used for any kind of entity. Although the player's internal state is special in the sense that it is stored in a (emulated) quantum register.

3.1.2 Why Size Matters

Inputs in the game affect the player's orientation. That is the bits 10 and 11. Nonetheless, the 14 other bits also need to be quantum since a superposition of directions will lead to different locations once the player entity will perform its regular move forward. Incidentally, local effects such as eating a Super Gum are applied only to some instances of the player entity. In the end, the whole Internal State register is quantum, hence the importance to keep its size as small as possible. Indeed, the game internally tracks 2^{16} complex number, each one corresponding to a possible Internal State configuration. Note that the code take advantage from the fact that the last three bits are unused and some operation simply ignore the last three bits (especially when a loop over every instance is required) thus in practice only a quarter of the tracked complex values are actually used in the game.

3.2 A few remarks on Invertibility and why there are not any Walls

A major side-effect of using emulating a quantum Internal State for the player is that, as discussed before, all operations on the Internal State must be unitary. That includes the operation resulting from inputs, of course, but also the ones

resulting from automatic game play such as going forward and environmental one such as colliding with another entity (ghost, gum or super gum).

3.2.1 Automatic Game Play and the Absence of Walls

Let us first focus on automatic game play. The rules for player movement are the easiest there is: given a position and a direction, the next cell is deterministically computed such that each couple cell/direction as one and only one predecessor. As such, this is a unitary (thus invertible) process: from a given game state, the game can be rewound to the previous state (as long are inputs are known).

However, this is no longer true as soon as walls are introduced. Introducing walls mean that either the player is automatically redirected or stopped. let suppose the player hit a wall in front of him, without obstacle on his left or right in either case, the player is set in a non-reversible configuration:

- Either the player is stuck in place, and the resulting position can result from the player being already stuck at the previous step, or the player hitting the wall for the first time.
- Either the player is set to an arbitrary direction that allows him to move further, and the resulting position may be both because the player hot the wall, or because he came from the opposite direction.

Both of these cases can be tackled out by introducing some path tracking bits that store the necessary information to reverse the game of one step. However, doing so consistently means tracking the path of each internal State configuration across the entirety of the game, and since this tracking must be stored in quantum bit too, the memory and computational cost of doing so are prohibitive. Hence the lack of walls on the board.

3.2.2 Environmental Game Play

There are a few problems with environmental game play elements in the context of quantum computing and applying collision effects are a local scale is, in itself problematic.

Let us first take the case of Ghost. The immediate, fully local, implementation of the expected game rules would be that a collision with a ghost *kill* the player. That is to say, if the player is in a state where it has, say, a probability of 0.5 to collide with a ghost, it has now a probability of 0.5 to be dead. The problem with such approach is that the operation is not reversible. Therefore, the solution implemented here is that collisions with ghosts damage a global health bar, weighted by the probability of the collision. The health bar in itself being a regular entity that does not follow the rules of quantum programming.

The same also applies to some extent with super Gum. It is not possible (without additional tracking bits) to devise a game play mechanism where eating a Super Gum switch the player into a super state that stops after a while. At the minimum, doing so would require that each operation have an effect to the alternate state, that is, super gum would have an effect on players in a super state, and the timer check on non-super player. In the case of super gum, we chose to drop entirely the time and make super Gum act as switches, thus a super gum both turn a normal pacman into a super pacman (that is, a pacman that can eat ghosts) and a super pacman into a regular one.

The last environmental element left is the case of gums, which are simply treated at a fully global level. Eating a gum grant a number of points weighted by the probability of eating said gum.

3.2.3 A Last Remark on Reversible Interaction

Thinking of game play in terms of reversible operations can sometimes be tricky and counter-intuitive. In the end, it comes down to describing and interaction as matrix operators and checking whether or not it is a unitary matrix. However, doing so may not be so simple when dealing with 16 bits quantum register as the corresponding matrices are of dimension 2^{16} by 2^{16}. Of course, in most cases, a given operation only affects a few bits and can usually be expressed as a much smaller matrix (see section 3.3).

A good intuitive way to think about it though is as the following: if any operation maps two different states into the same one, even partly, it is not reversible. In particular, if an operation maps a given state into another, then it must also map said target state elsewhere and not into itself.

3.3 Input Game Play

The last kind of game play interactions are interactions prompted directly by input. As said before, these interactions only apply to the two direction bits of Internal States. Therefore, we can characterize them as 4 by 4 matrices over a 2- bit quantum register. Reasoning on the full $2^{16} \times 2^{16}$ matrix is the same as reasoning over the 4×4 one as every other bits are mapped to themselves.

4 Gameplay Rules

4.1 Game Over

Game over mechanics are, for once, fairly simple. Hitting a ghost while not in super mode reduces the player's health points by a fixed amount multiplied by the probability of the player being actually hitting the ghosts. Remember that the player is in every position at the same time, so it technically hits all the ghosts8 times at each turn (4 times for each direction, times2 whether it is in super mode or not). The game ends when the player drop to or below 0 health point

4.2 Inputs

The inputs are mapped on the numpad's keys1 to 9.

- Key 1: perform a swap on the two directional bits. Namely, the NORTH and SOUTH direction are untouched while the EAST and WEST direction are reversed.
- Key 2: reverse the current direction, NORTH and SOUTH are reversed as well as EAST and WEST
- Key 3: not mapped yet

- Key 4: perform a 90 degrees left turn, NORTH goes to WEST, WEST to SOUTH and so on.
- Key 5: apply a Hadamar transform. In a nutshell, every direction is split in equal proportion into NORTH, SOUTH EAST, WEST. Although the Hadamar transform is not that simple and there is no better explanation than running the maths from the explicit 4 by 4 Hadamar matrix given previously.
- Key 6: perform a 90 degrees right turn. NORTH goes to EAST, EAST to SOUTH and so on.
- Key 7: Not mapped yet
- Key 8: Not mapped yet
- Key 9: Not mapped yet

Remark that inputs can be combined to form combo that can achieve a higher level control pattern. For instance, the combo5-WAIT-2-WAIT-5 allow to split the player in 4 the fusion it back at its previous location.

More refined combo can be found from experimentation, matrix calculus or a subtle mix of the two.

4.3 Ghost AI

Ghosts are fairly simple in their behavior and freely inspired from Pacman's ghost AI behavior, with some twist to make it work with a quantum target. Each ghost has a different personality, and with it comes a different *scatter areas* and chasing strategies. Ghosts have two operating mode: *scatter* and *chase*. During scatter mode, they pick a random point in their scatter area and try to reach it; during chase mode, ghost target one of the player locations and chase him. Scatter and chase mode alternate based on a timer defined by the ghost's personality. In addition, catching a target in chase mode automatically switch the ghosts back to scatter mode, the reverse is not true though and if a ghost reaches its scatter target, it will simply pick a new one.

The four ghosts personality are:

- Hunter: Always focus the most probable player location in chase mode, scatter in the bottom right area of the game board. Chase for 30 turns and scatter for 10 turns.
- Lazy: Always focus on the closest player location (with probability greater than 0, obviously), scatter in the upper left area of the game board. Chase for 10 turns and scatter for 30.
- Random: In chase mode, pick a random player instance (with probability greater than 0, obviously) and chase it. Scatter in the bottom left area of the game board. Chase and scatter for 10 turns
- Mimic: Pick a random personality in chase mode, and go after the corresponding target, scatter in the upper right area of the game board. Uses the scatter and chase timer of the last personality it has mimicked.

4.3.1 Closing Remarks on Ghosts AI

A few things should be noted about ghosts. Picking a target is a very costly task because the player is technically everywhere at the same time, with different probability. As such, picking a target involves in most cases going through the 2^16 possible player's internal state and look for the closest/most/probable/random one. Once a target has been selected, ghosts will simulate the input in order to track their target without having to go through the entire computation again. In case of the Hadamar input, where the target split, the ghost will typically select one of the splatted players at random. Therefore, targets are computed only when a ghost enter chase mode.

Also, note that ghosts may catch players they may not be chasing.

4.4 Re-spawn and integrity

4.4.1 Integrity explained

Every classic entity (i.e, ghosts, gum and super gum) has an integrity internal field that goes from 1 to 0 where 1 means the entity is active and 0 mean it's not. An inactive entity does not interact with the player nor move (in case of ghosts).

In a regular game, integrity would not be necessary as an object is either active or not. However, in our case, given that the player is quantum, entities may be in a state where they have *probably* collided with the player. This is what integrity captures. When an entity interacts with the player, it loses some of its integrity equal to the probability of the player actually being here. This can be seen as the entity start to become transparent. When an entity's integrity is set to 0 it disappears and its re-spawn timer start.

4.4.2 Re-spawn

Each entity class has a different re-spawn timer. Once the timer is spent, its integrity is basically reset at 1 and the entity is again a part of the active game.

References

[1] Benoît Valiron. Quantum computation: a tutorial. *New Generation Computing*, 30(4):271–296, 2012.