

An Exploratory Study in Quantum Acceleration of Ray Tracing

Copyright 2015 Eric Johnston

Draft 1 (August 7, 2015)

Updated August 22, 2015

Fine Print: This document is not finished, but you're welcome to read it anyway. It has not been peer-reviewed, or even thoroughly researched. These are my own notes and speculations, with shiny pictures added. If you have questions, comments, or really good jokes, please contact me at ej@machinelevel.com

Overview

Ray tracing is a form of computer graphics in which rays are projected out into a scene, to probe it for color and lighting values. These values are used to determine the color of each pixel in the resulting image. The quality of the resulting image, and also the cost of the computation, increases with the number of rays emitted. Ray tracing is used commercially in the motion picture industry, which is motivated to reduce cost. For a high-budget movie running at 24 frames per second, a single frame can require multiple hours to render. Even a 50% reduction in this time would result in a substantial business motivation.

[todo: ray tracing diagram]

In this informal whitepaper, I'll explore two approaches to speeding this up using a quantum computation device. A simple but functional running simulation will be constructed, as a proof of concept.

Note: As of 2015, commercial ray tracing applications work with gigabytes of data per scene. Currently, gigabyte systems are not only unavailable but staggeringly remote, rendering this approach impractical. That's the point; when the devices are ready to handle more complexity, we'll be ready to make use of it.

A live functional sim of this whitepaper's examples is here: http://qc.machinelevel.com/qgray_examples/qgray_v1.html

The Goals of This Exploration

Goal #1 is simple: Use a quantum computer to **speed up ray tracing** in a practical way. The cost of ray tracing an image is the cost of tracing a single ray through the scene, multiplied by the number of rays. Design and simulate a device which can actually do this, and then scale it up and produce a photonic etching pattern for producing the device.

Goal #2 will be to provide a simple way to trade speed for quality. Essentially, ray tracing has this built-in already. By tracing fewer rays, the render is sped up and the quality is reduced. Whatever we do to speed it up, we want to preserve this ability.

Goal #3 will be to add in motion blur, for free. In classical ray tracing systems, motion blur is typically achieved by sampling the scene at various times as well as positions, but this adds to the computational cost.

The naïve approach for anyone who's heard of multiverse computation is to say *"That's easy! Do a Hadamard transform of all of the initial ray's variables, run a single trace using that superposed omni-ray, and voilà! All possible rays have been traced. Next problem."*

This actually works, **however** it's not useful. You'll end up with a combination of all possible rays, but when you read it, it will simply collapse into a single ray. Quantum algorithms don't return *more* information per query than classical ones, just *different* information. So if we can't trace one ray and end up with a complete image full of pixels, what can we do?

Approach 1: Use Supersampling and Quantum Sum

The high-quality ray tracing world counts on using many millions or billions of rays to produce an image, but it's not practical to *store* all of those results. What's done normally is to project many rays for each pixel of the image, and simply store the sum (or an average) of the results.

If you're doing seat-of-the-pants quantum computation experiments, you should be saying "Aha! That's a computation which uses many rays, but returns the same amount of information as a single ray. We can work with that."

We'll set up a superposition ray trace for all of the samples in a single *pixel* of our image, run the test, and then try to get either the sum or the average out of the QC, possibly using a modification of Grover's search.

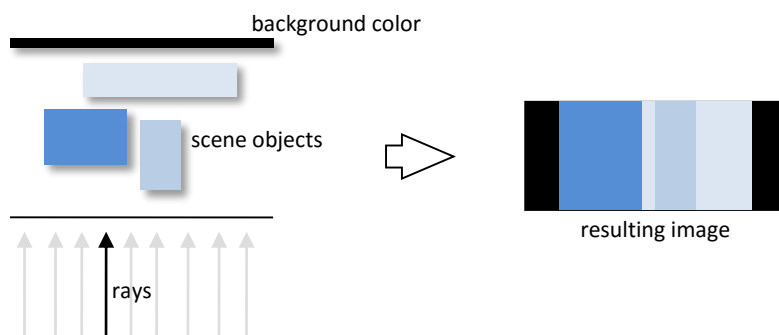
Approach #2: Scene Objects

Another thing we could do is to superpose the scene objects, maybe even storing all triangles in the scene as a superposition of a single triangle, and try to figure out which one each ray hit. That seems reasonable, as we're still only asking for a single hit, so the quantity of information is preserved.

This actually won't be as effective as Approach #1, because modern ray tracing uses all sorts of clever algorithms such as BVH's (Bounding Volume Hierarchies) to keep the computation time down as the scene complexity rises. Still, we should keep this in mind, because it can possibly be combined with other solutions at a later time, and *any* collision structure (including BVH's) might be able to benefit from this.

Step 1: A Simple QC Ray Tracing Framework

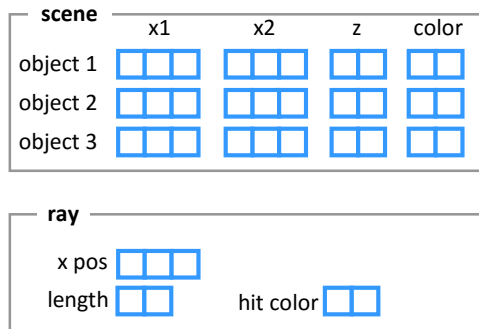
Enough planning, let's build something. For this simulator to work, we're going to need to keep it *super simple*, but we also want it to be able to scale up. Consider the following scene:



To keep the data and logic simple, this is an orthogonal (non-perspective) view, the objects are all rectangular, and the 'colors' are just grayscale intensity values. This scene is also two-dimensional, resulting in a one-dimensional projected image. All of these concessions can be fixed later on, as we have access to more complex quantum computers.

We're going to use [qcengine](#) for the construction and simulation (more info is available through that link).

We'll set up the following quantum data structures:



That seems to be the data we need, and it's only 37 qubits.

Here's our scene setup data:

```
var x_bits = 3;           // The number of bits in the x coordinate
var z_bits = 2;           // The number of bits in the z (depth) coordinate
var color_bits = 2;       // The number of bits in an object's color
var num_objects = 3;      // How many objects we're going to use
var background_color = 0; // The color to use when no object is hit by a ray

// It's okay to have too many object defs; we'll only use num_objects of them
var object_defs = [ {x1: 1, x2: 3, z: 0, color: 2},
                    {x1: 2, x2: 5, z: 1, color: 3},
                    {x1: 4, x2: 6, z: 3, color: 1} ];

var ray = {};
var scene = {};
var x_sign_bitmask = 1 << (x_bits - 1); // We'll use this to do "if x < 0" conditions.
```

...and here's the code to build the machine from the starting data...

```
function allocate_all()
{
  var bits_per_ray = x_bits + z_bits + color_bits;
  var bits_per_object = 2 * x_bits + z_bits + color_bits;
  var total_qubits = bits_per_ray + num_objects * bits_per_object;
  qc.reset(total_qubits);
  qc.print('Using ' + total_qubits + ' qubits.\n' +
          'RAM est: ' + Math.pow(2, total_qubits + 3 - 20) + ' MB.\n');

  ray.x = qint.new(x_bits, 'ray.x');
  ray.length = qint.new(z_bits, 'ray.length');
  ray.color = qint.new(color_bits, 'ray.color');
  scene.objects = [];
  for (var i = 0; i < num_objects; ++i)
  {
    var obj = {};
    obj.x1 = qint.new(x_bits, 'obj['+i+'].x1');
    obj.x2 = qint.new(x_bits, 'obj['+i+'].x2');
    obj.z = qint.new(z_bits, 'obj['+i+'].z');
    obj.color = qint.new(color_bits, 'obj['+i+'].color');
    scene.objects.push(obj);
  }
}

function init_scene()
{
  for (var i = 0; i < num_objects; ++i)
  {
    var obj = scene.objects[i];
    var def = object_defs[i];
    obj.x1.write(def.x1);
    obj.x2.write(def.x2);
    obj.z.write(def.z);
    obj.color.write(def.color);
  }
}
```

When it's run, we print out the scene to check it producing this:

```
Using 37 qubits.  
RAM est: 1048576 MB.  
  
....111. }  
.....  } scene volume  
..3333.. }  
.222.... }  
_!_____ } ray
```

So there's our simple scene. Three objects, and at double precision it's going to take 1 *terabyte* of RAM to simulate. In 2015, that's pretty expensive. Luckily, we have the option of scaling it down further, for example by reducing the object count.

Quick fun diversion

Or, we also have the option of scaling it up. Just because we can't run a quantum sim doesn't mean we can't design the machine to run it for real. What happens if we go to 32 bits for *x* and *z*, and use 24-bit colors for each object? That's fine.

```
Using 448 qubits.  
RAM est: 5.54533938824163e+129 MB.
```

...so to simulate this QC on a classical computer, we need something like 10^{123} terabytes. Even if you're reading this a thousand years in the future (3015), that's going to be expensive, considering there are something like 10^{79} atoms in the observable universe. You'll be better to use a real 448-qubit QC. Those are likely very cheap by now.

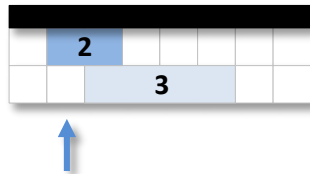
Here's the very first pass of our actual ray tracing function:

```
function ray_trace()  
{  
  for (var i = 0; i < num_objects; ++i)  
  {  
    var obj = scene.objects[i];  
  
    // The ray hits if ray.x >= obj.x1 and ray.x <= obj2.  
  
    // ...so, if obj.x1 - ray.x - 1 is negative  
    obj.x1.subtract(ray.x);  
    obj.x1.subtract(1);  
    // ...and, if ray.x - obj.x2 - 1 is negative  
    ray.x.subtract(obj.x2);  
    ray.x.subtract(1);  
  
    // if there's a hit, exchange the color  
    var condition_mask = qintMask([obj.x1, x_sign_bitmask,  
                                   ray.x, x_sign_bitmask]);  
    ray.color.exchange(obj.color, -1, condition_mask);  
  
    // Then restore the ray  
    ray.x.add(obj.x2);  
    ray.x.add(1);  
  }  
}
```

Note that this function destroys the scene data each time a ray is traced. Also, we're completely ignoring the z component and the ray length, so this simple case will only work if the objects are sorted by depth. That's all okay, this is just a first pass.

And since we're ignoring depth, let's remove the depth-related qubits. Also, let's start even simpler, with just two objects.

Here's a diagram of this scene...

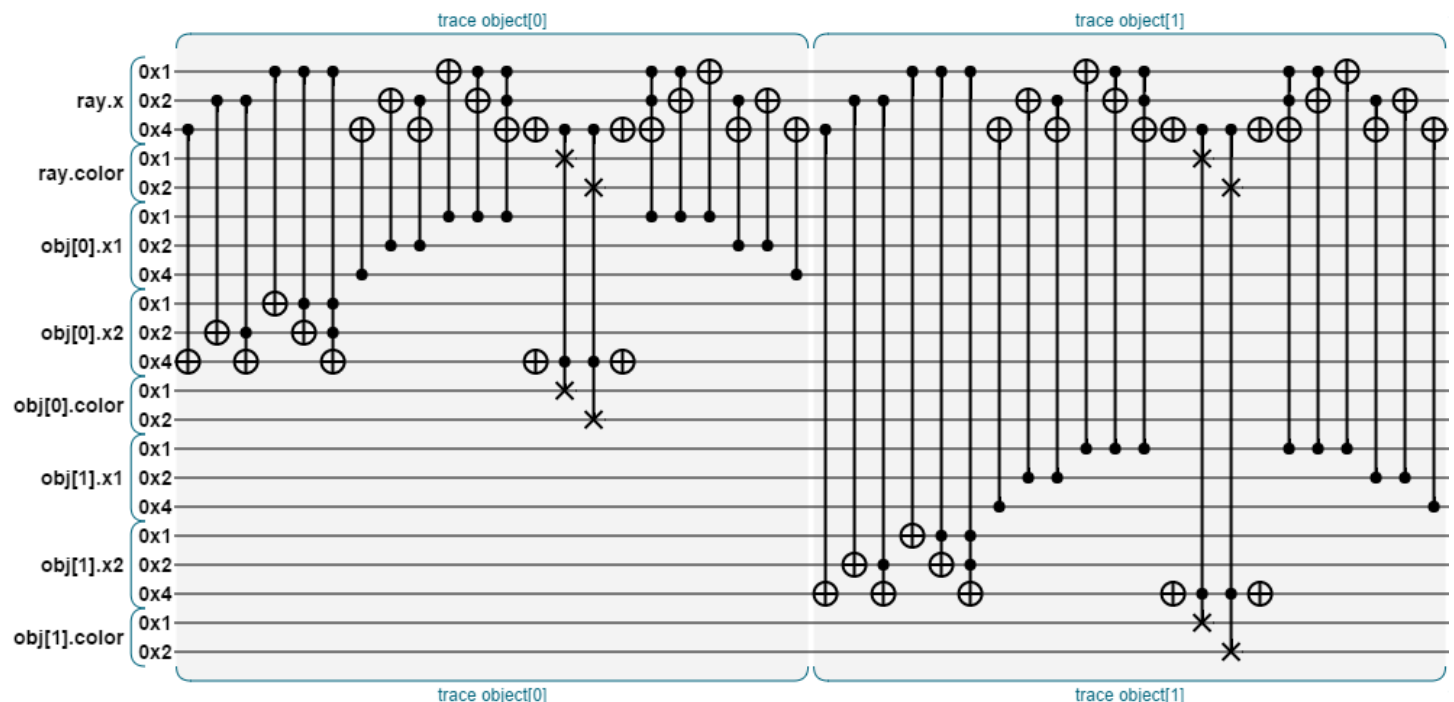


Here's our setup, ready to run:

```
Using 21 qubits.
RAM est: 16 MB.

.222.... } scene volume
..3333.. }
_!_____ } ray
```

Sixteen megabytes is quick and easy. We can start here, get it working, and then expand back out. That source code produces this QC circuit for a 2-object scene:



So let's let it run, just to trace one ray. Here's what we'll do:

```
allocate_all();
init_scene();
ray.x.write(1);
ray.color.write(background_color);
```

```
ray_trace();
```

...and here's what we get:

```
Using 21 qubits.  
RAM est: 16 MB.  
output color is 2  
  
(Finished in 4.384 seconds.)
```

Success! The color in position $x=1$ is actually 2. We've just done something simple and unexciting, but it worked. A single sample isn't really enough to verify correct operation, so let's do a brute-force "trace every single ray possible" and make sure it comes out okay. Note that this is what we're trying to *avoid* doing in the future, but we can use it to establish the "ground truth" values that our finished system should match.

```
function do_brute_force_traces()  
{  
  var ray_count = 1 << x_bits;  
  for (var i = 0; i < ray_count; ++i)  
  {  
    init_scene();  
    ray.x.write(i);  
    ray.color.write(background_color);  
    ray_trace();  
  }  
}
```

```
Using 21 qubits.  
RAM est: 16 MB.  
  
.222....  
..3333..  
_!_____  
  
results: 0 2 3 3 3 3 0 0  
sum: 14  
average: 1.75  
  
(Finished in 28.336 seconds.)
```

The x coordinate has three bits, so there are eight (2^3) possible rays to trace. The results all look correct, so if we were creating an image it would look perfect. That's not what we're doing. We're rendering a single pixel, using multiple rays for supersampling. There's no need to store (or even display) all of the results. We just need the sum, or the average. In this case, our sum is 14, and the average is 1.75. If our final solution comes up with either of these answers, we're set.

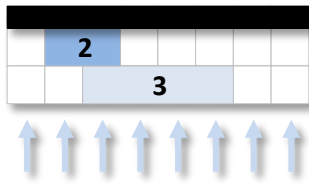
Now we've got a working (if simple) ray tracing program. I've actually spotted a bug in the code (you may have as well), but since the answers are correct (and easy to double-check as we go), let's keep focus and push this all the way through before refining the program.

Step 2: Finding a Quantum Advantage

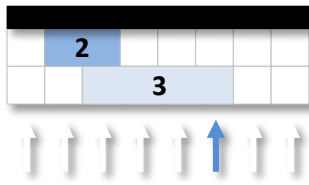
Why go through all the trouble of building a quantum circuit instead of a classical computer program? We *can* actually trace every possible ray at once, like this:

```
allocate_all();  
init_scene();  
ray.x.write(0);  
ray.x.hadamard();  
ray.color.write(background_color);  
ray_trace();
```

Previously, we were setting `x` to a specific value. Using the `hadamard()` call, we put `ray.x` into an even superposition of all possible values (so for 3 bits it will contain all values 0 – 7 in evenly distributed probability.)



When we run the program just once, the code will trace ***all possible rays*** at the same time, on the same hardware. We can perform this action, but once we've done it, we need to read an answer. Doing that collapses the quantum state, and we're left with a randomly-selected answer.

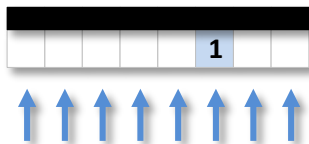


The answer is correct, but the rest of the information is lost. It's no better than if we had rolled some dice to choose which ray to shoot, and then run it through a classical program.

Step 2a. Intro to Grover: Quick and Easy

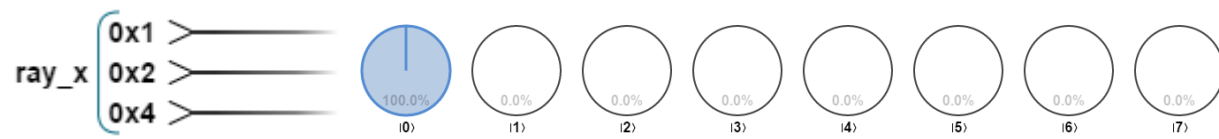
If you're reading this, you've probably heard of Grover's search algorithm, but you might not know the exact mechanics of it. In an case, here's a run-through of what it is and how it might apply here.

Considering the following simple scene:



In this case, there's only one ray which will return a 1, and all others will return 0. Suppose we want to find out *which* ray is the winner, without taking the time to trace all of the rays. That's what Grover's algorithm is for. Here we go.

First, we need an **even simpler** ray tracing system, with one key change. Let's take a look at the input state, initialized to zero. Three qubits is going to be eight states, but each state has a magnitude and phase, so we can visualize it in circle-graph notation, like this:

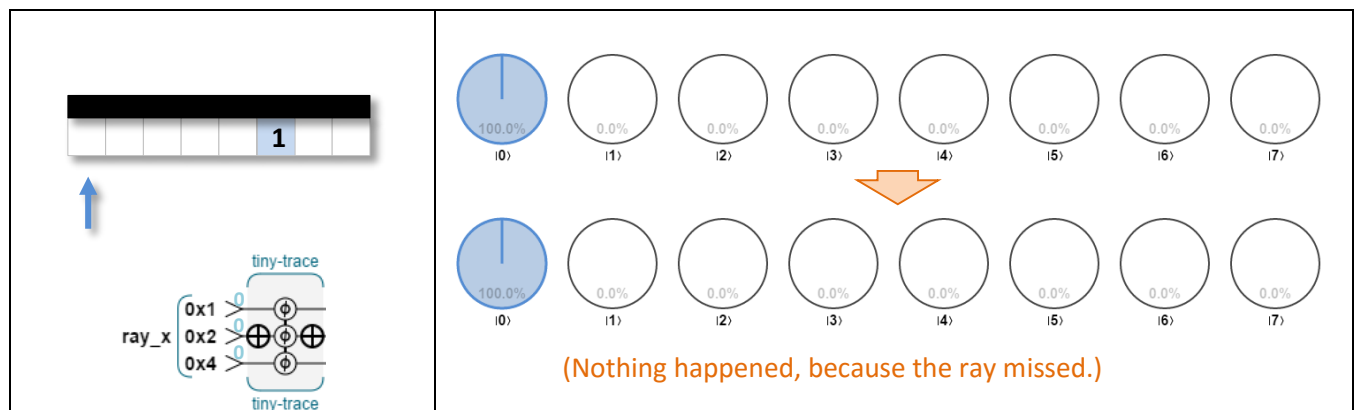


That's it. Just the ray position. Now we'll write a program which says "if the ray position is equal to the object position, rotate the phase by 180 degrees." It turns out that's easy; if the object is in position 5 (for example), then we just take whatever the input state is, and rotate circle 5 by 180 degrees. Here's the program which accomplishes this in qcengine, along with the circuit it produces for an object in position 5.

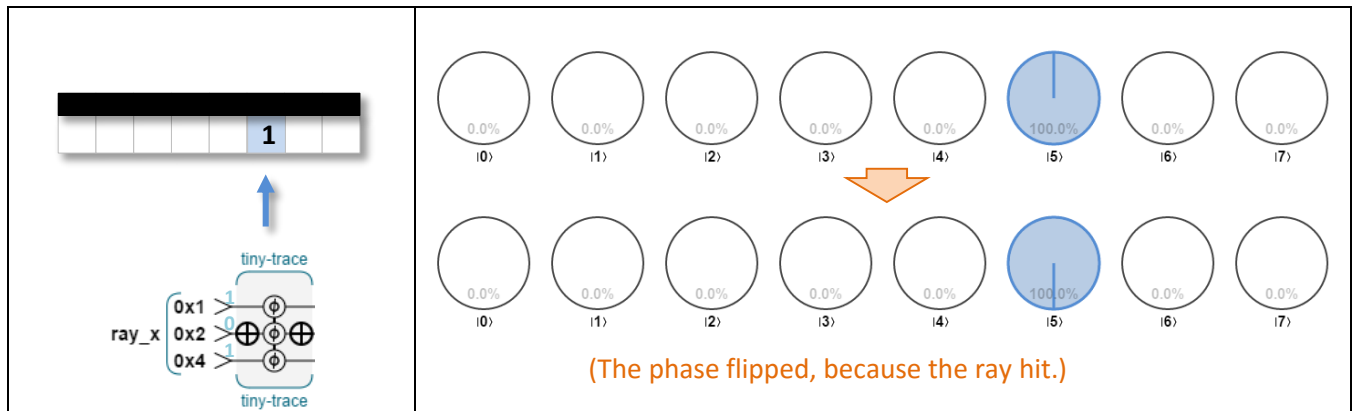
<pre>function tiny_trace(x, obj_pos) { x.not(~obj_pos); x.phaseShift(180); x.not(~obj_pos); }</pre>	
---	--

That really is as far down as we're going to strip our "ray tracing" system without a risk of being laughed at; it's requiring some imagination to keep taking it seriously. Here's the effect of this program on a few different inputs:

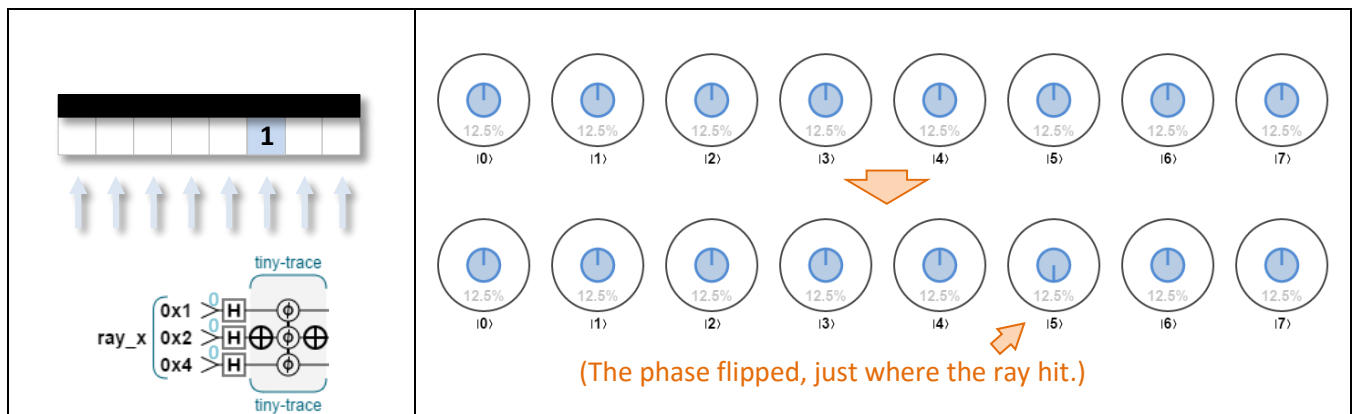
Case A: ray_x = 0 (The ray misses the object.)



Case B: ray_x = 5 (The ray hits the object)



Case C: ray_x is a superposition of all possible values (this contains one hits and several misses)



A program called “Rotate circle #5” is about as simple as it gets, but already this system does two interesting things a digital logic circuit can’t do:

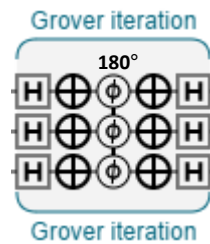
1. We don’t need output bits! The result is stored in the phase of the input bits. This is really interesting, as we’re *storing additional hidden information in the same register*, without requiring any more storage space.
2. We can easily calculate all possible outcomes, with just one run of the program.

...and also, there are some quantum-quirks:

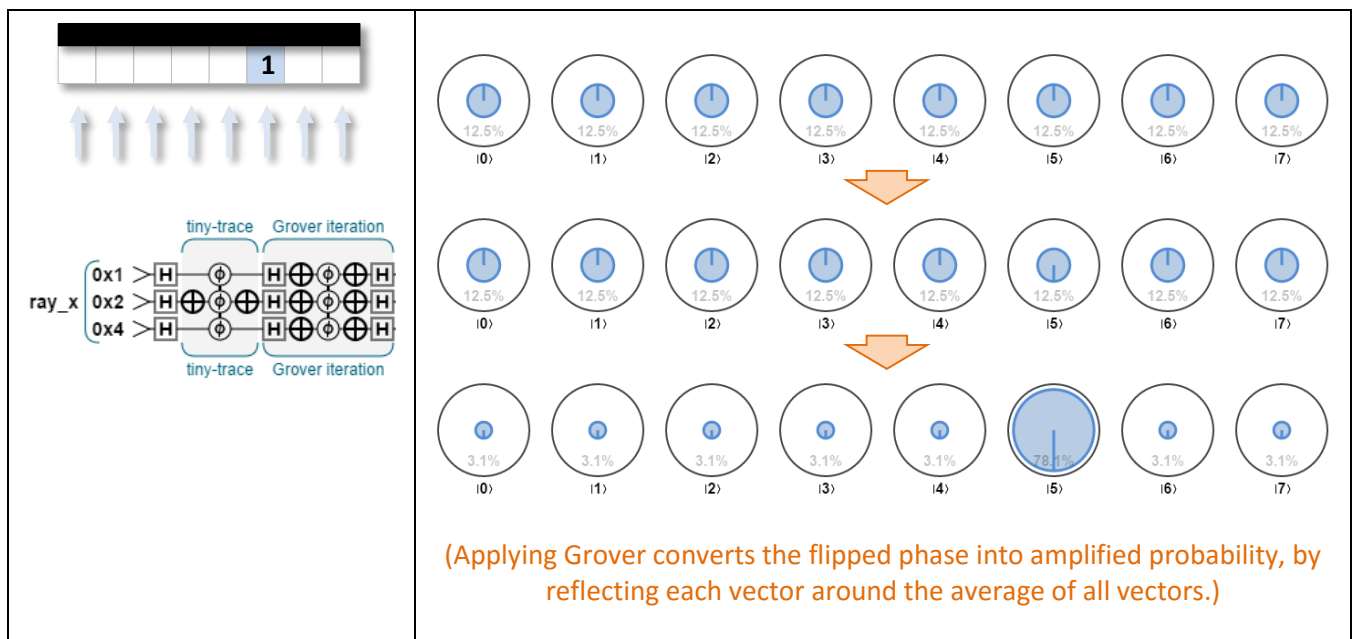
1. We don’t have output bits! If we read the register, we simply get the input value, with no indication of whether or not the ray hit. The “additional hidden information” is hidden a little too well.
2. In the superposition case, reading the register produces a random number, with no useful aspect whatsoever.

Here’s where the Grover iteration comes in. What we’ll do is flip all of our 2D vectors around the average value of all of them, so that the one that’s out of phase will stand out as the most probable value to read. The program is actually easy.

```
function groverIteration(x)
{
  qc.codeLabel('Grover iteration');
  x.hadamard();
  x.not();
  x.phaseShift(180);
  x.not();
  x.hadamard();
}
```

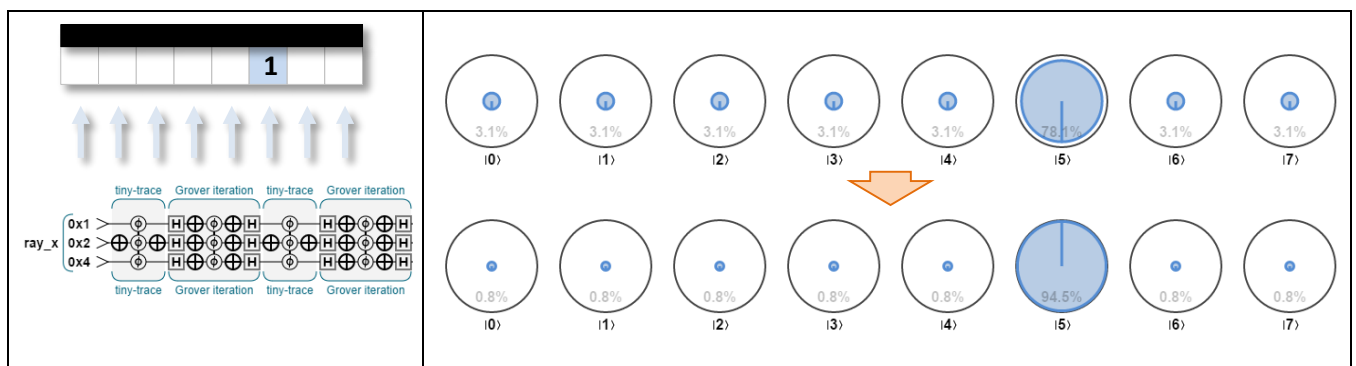


Now let's apply this to our superposition-output state and see what happens.

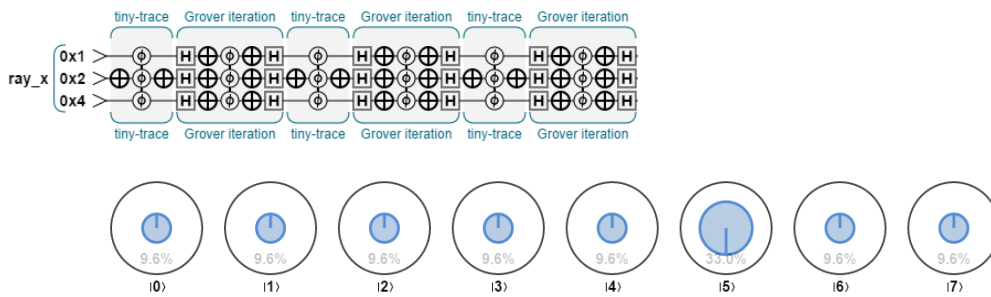


That's useful! All rays were tested, and the magnitude of our "hit" jumped to almost 80%. That means if we're asking "Which ray hit the target?" and we read this register now, there's an 80% chance of getting the right answer, after running the trace just once. That's already better than we could do classically.

Now, though, let's do another trace-and-Grover, and see where we end up.

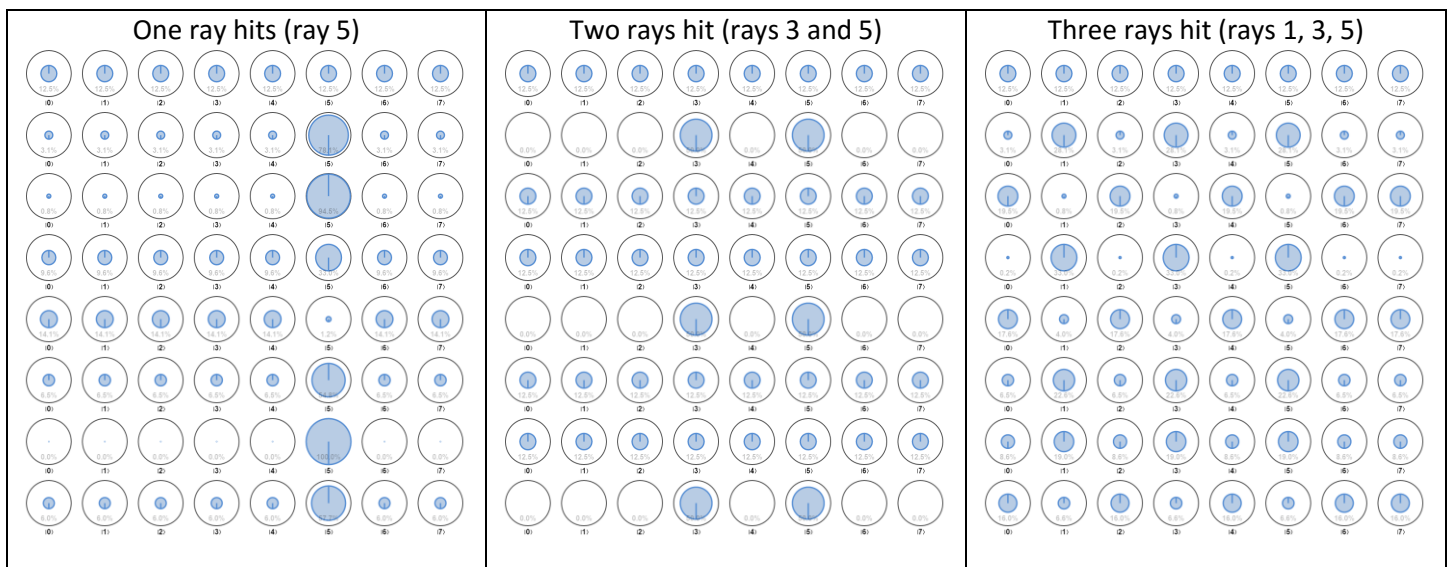


Even better! Now after just two runs, we've got a 94.5% chance of getting the correct answer. Heck, let's do it again!

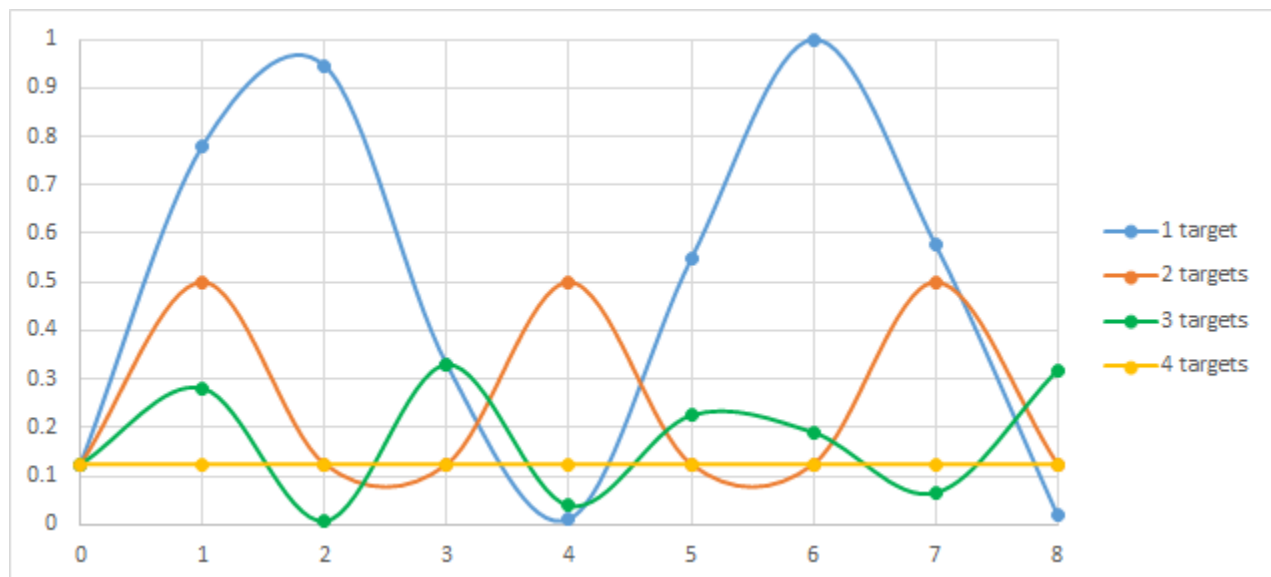


Uh oh, we've lost some awesomeness. Now we've only got a 33% chance of getting the right answer. Here's what's happening: Grover iterations cause a **periodic** amplification. If you keep applying them, they go through cycles, and the period of the cycle depends on how many possible input values exist, as well as how many solutions there are. Our case has exactly one solution, so the useful period is the square root of the number of inputs. We're using 8 input values (3 bits), so we get the best answer after about $\sqrt{8} = 2.8$ iterations. As we saw, 2 is the one which worked out best.

The period also depends on the *number* of correct solutions. So if we have two objects

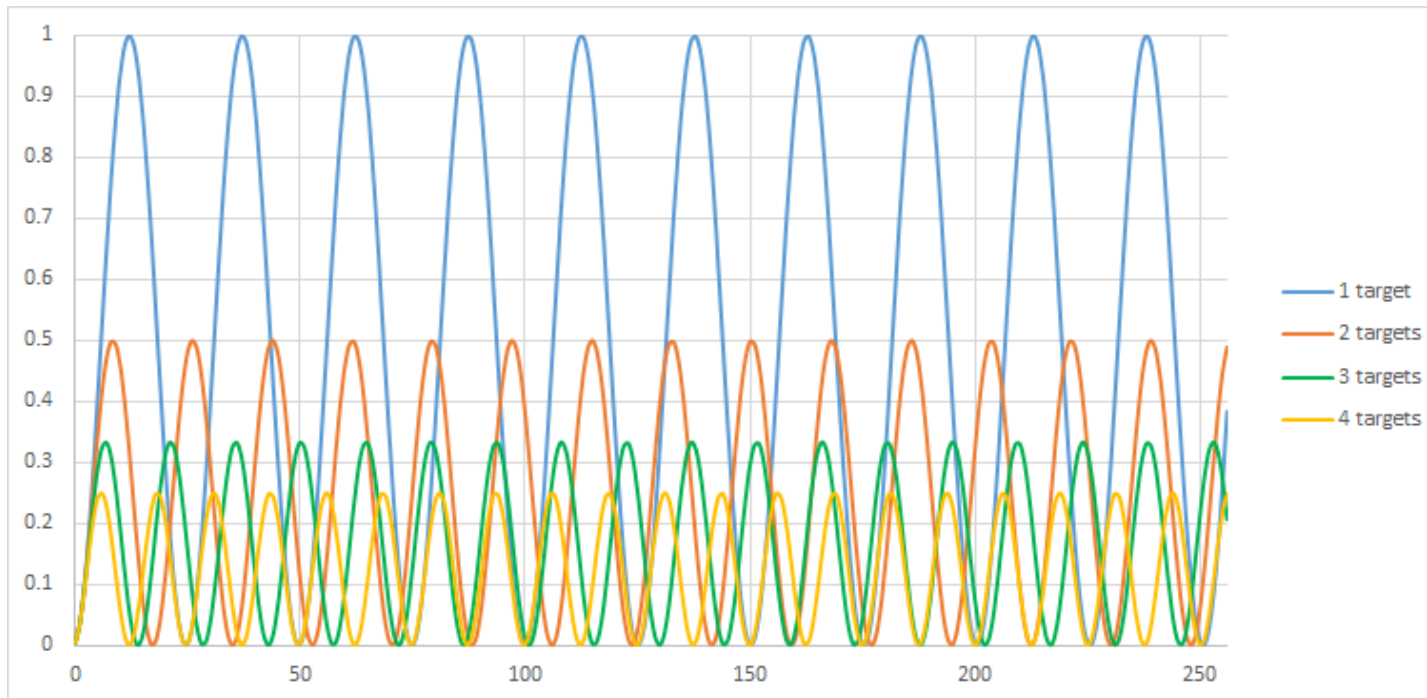


Looking at the "correct answer" target probability, here's how it changes when there are 1, 2, 3 and 4 correct targets:



Notice that as the possible number of hits goes up, the frequency of this cycle goes up and the amplitude goes down. When half of all possible values are valid targets, it drops to a flat line, because there are the same number of vectors in each direction.

Eight values is pretty rough for looking at frequencies, so here's the exact same chart, but running the simulation with 256 values (8-bit value for ray_x):



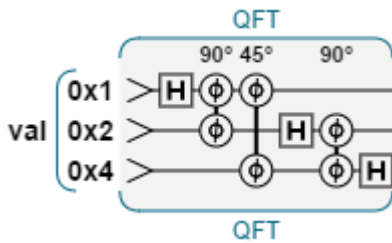
If we were just doing a basic Grover search, we'd stop here and say "Make sure you do the right number of iterations." In fact, in this case the 1-target wave peaks at 12 iterations. So if we *know* there's only one target, and want to know which one it is, we can find it after 12 ray tests (instead of 256), with 99.9947% probability. That's the magic of the Grover Search.

In this case, that's not the question we want answered. Still, these frequencies are actually useful. For our super-sampled pixel, instead of asking which ray hit an object, we'd love to know **how many** rays hit objects, and we're willing to not care which rays they are.

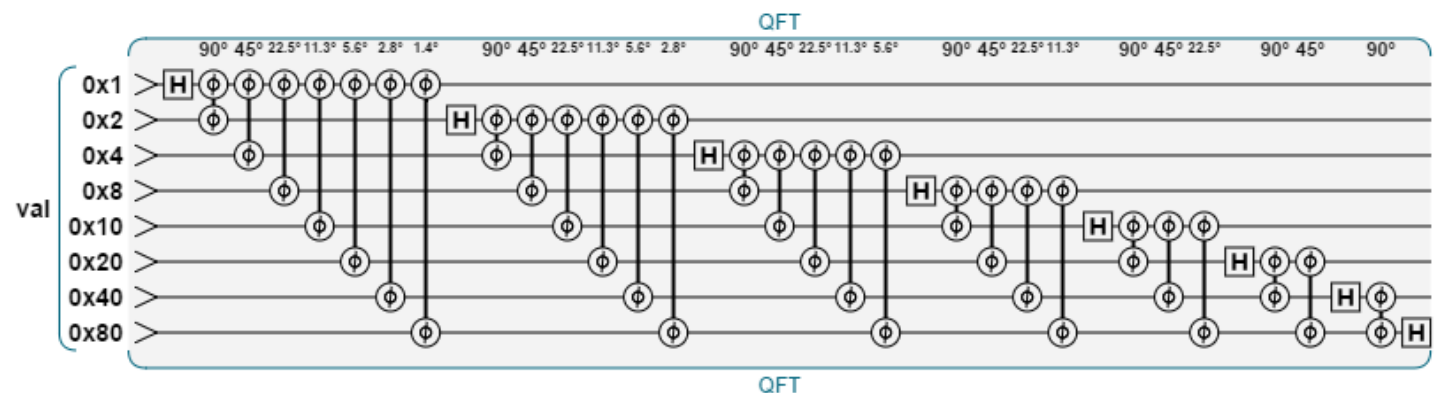
Step 2b. Intro to QFT

To detect different frequencies (in music and other signal processing), some kind of an FFT is useful. The quantum version of an FFT is called a QFT. If you're guessing it's made of the same three gates as everything else we've done so far, you're right.

Here's some qcengine code for a QFT, and the circuit for the 3-bit version:

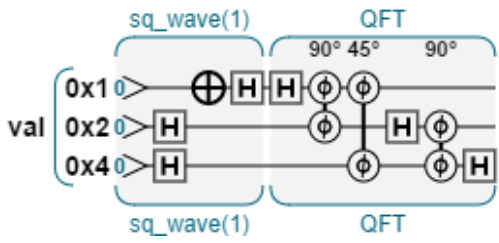
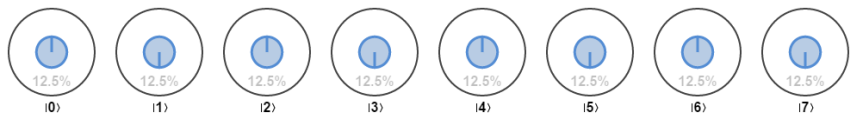

<pre>function QFT(x) { qc.codeLabel('QFT'); for (bit1 = 0; bit1 < x.numBits; ++bit1) { var mask1 = 1 << bit1; var theta = 90.0; x.hadamard(mask1); for (bit2 = bit1 + 1; bit2 < x.numBits; ++bit2) { var mask2 = 1 << bit2; x.phaseShift(theta, mask1 + mask2); theta *= 0.5; } } }</pre>	 <p>The diagram shows a 3-bit QFT circuit. Three input lines are labeled 0x1, 0x2, and 0x4. The circuit consists of Hadamard (H) gates and phase shift gates (circles with phi). Above the gates, the angles 90°, 45°, and 90° are indicated. The entire circuit is labeled 'QFT' at the top and bottom.</p> <p><i>Is the use of degrees instead of radians freaking you out yet? I kind of like it.</i></p>
---	--

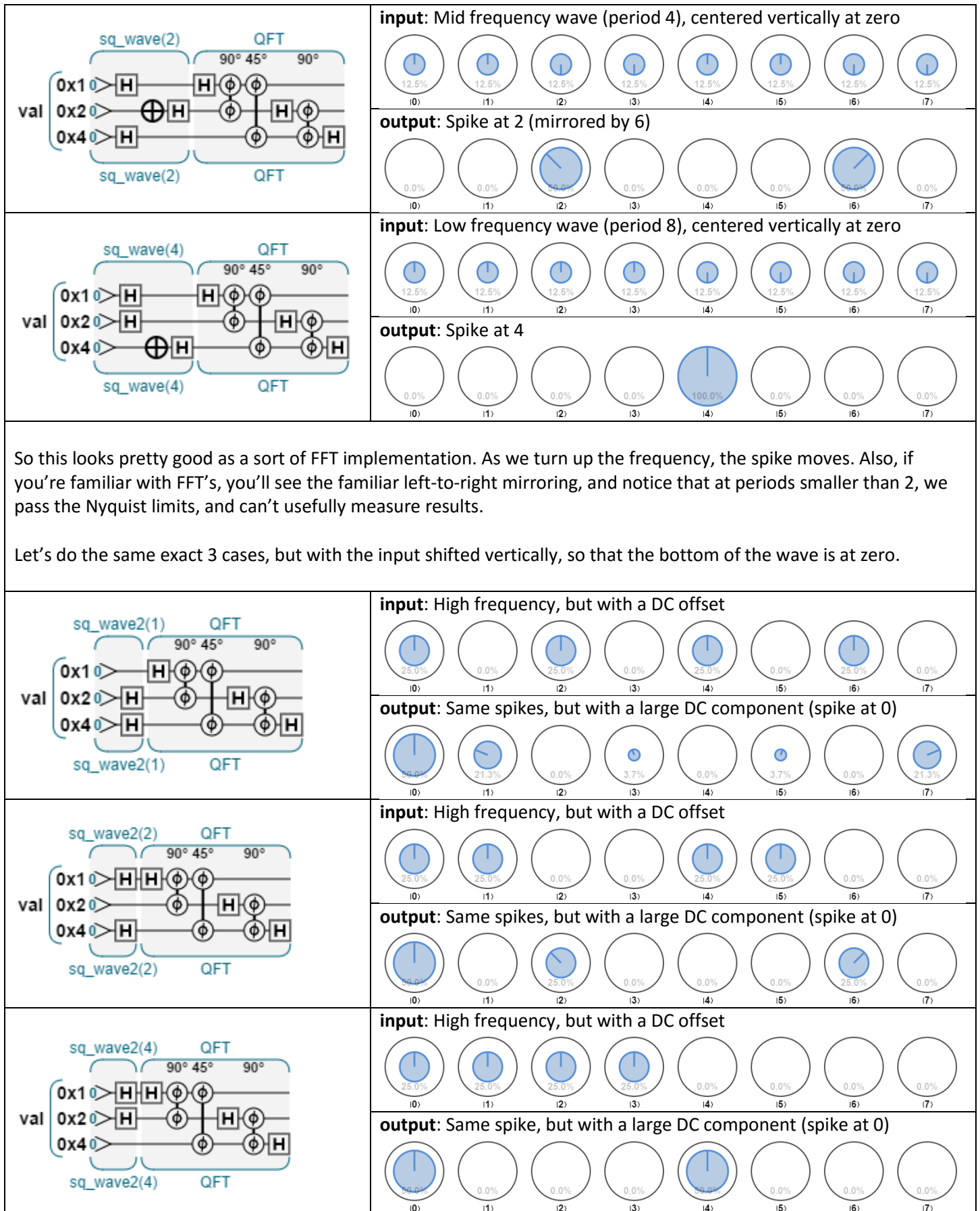
...and just because it's interesting, here's what the code builds for an 8-bit QFT:



To see what it does, let's run some tests. To test a typical FFT function, one thing we can do is throw waves of different frequencies at it, and check for spikes in the output. Then, if we offset the same waves vertically (introducing a DC component) we should see a spike at one end.

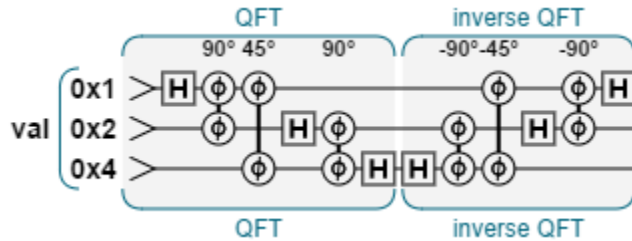
Let's give it a shot. We'll stick to the 3-bit for this.

 <p>The diagram shows a 3-bit QFT circuit. The input lines are labeled 0x1, 0x2, and 0x4. The circuit consists of Hadamard (H) gates and phase shift gates. Above the gates, the angles 90°, 45°, and 90° are indicated. The entire circuit is labeled 'QFT' at the top and bottom. The input lines are also labeled 'sq_wave(1)'.</p>	<p>input: High frequency wave (period 2), centered vertically at zero</p>  <p>Eight circular plots labeled i0) through i7) showing a high-frequency wave. Each plot has a value of 12.5%.</p> <p>output: Spikes at 1 and 3 (mirrored by 7 and 5)</p>  <p>Eight circular plots labeled i0) through i7) showing the output. The values are: i0) 0.0%, i1) 42.7%, i2) 0.0%, i3) 7.3%, i4) 0.0%, i5) 7.3%, i6) 0.0%, i7) 42.7%.</p>
---	---



There are more tests we could do, but this looks like it's performing reasonable well as a QFT, or a quantum analogue of an FFT.

One last thing... in case we need an *inverse* QFT, that's pretty easy. As long as we don't read or write, every quantum operation we're using is unitary and reversible. So to subtract two numbers (for example), you only need to run addition gates backwards. That makes an inverse-QFT trivial to construct:



Running this on any input should (and does) give back the original input.

Step 3. Intro to Counting

Using a Quantum Counting algorithm, we can essentially take a Fourier transform of the Grover amplifications, in order to discover how many hits there are.

Clearly this is valuable to us, as what we're after is the sum (or average) of the hit values in our super-sampled pixel. These two scenes should return exactly the same result:



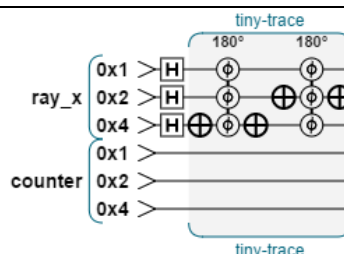
What we'd like is to get the number "2" from both of these, with a high degree of confidence, and *without* running 8 separate traces. That's the dream.

From the previous section, we've got waves with frequencies which depend only on the number of hits. If we can detect these frequencies (using something like an FFT) maybe we can tease out the information we really want.

We should have everything we need to give this a shot. Let's (literally) just put them together and see where we end up.

First, we'll modify our phase-flipping `tiny_trace()` function so it'll flip more than one value. Then, we'll also attach a 3-bit "counter" but not use it just yet. So this is a 6-bit quantum register, which means we'll have 64 possible values, each with magnitude and phase. Time to test this step.

```
function tiny_trace(x, objects)
{
  qc.codeLabel('tiny-trace');
  for (var i = 0; i < objects.length;
  ++i)
  {
    x.not(~objects[i]);
    x.phaseShift(180);
    x.not(~objects[i]);
  }
}
```



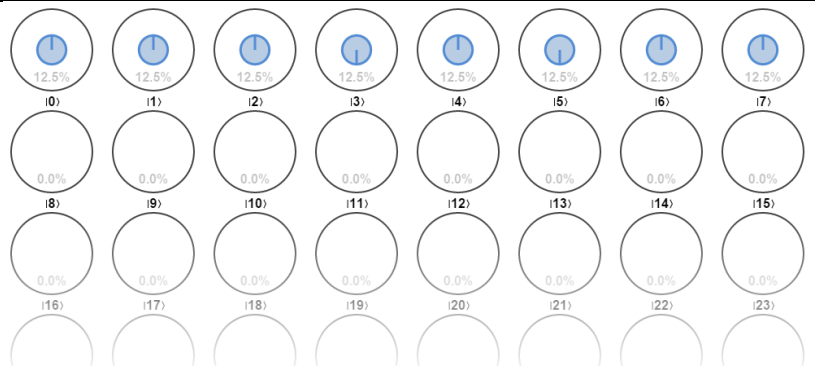
```

var x_bits = 3; // number of bits in
the x coordinate
var p_bits = 3; // number of bits in
our counting precision

qc.reset(x_bits + p_bits);
var ray_x = qint.new(x_bits, 'ray_x');
var counter =
    qint.new(x_bits, 'counter');
var object_positions = [3, 5];

qc.write(0);
ray_x.hadamard();
tiny_trace(ray_x, object_positions);

```



Success! We have objects in position 3 and 5, and our `tiny_trace()` found them and flipped the appropriate phases (3 and 5) in our quantum state. The higher values are unused, because we've left the **counter** bits at zero. Next, just to see it working, we can apply our Grover iteration.

```

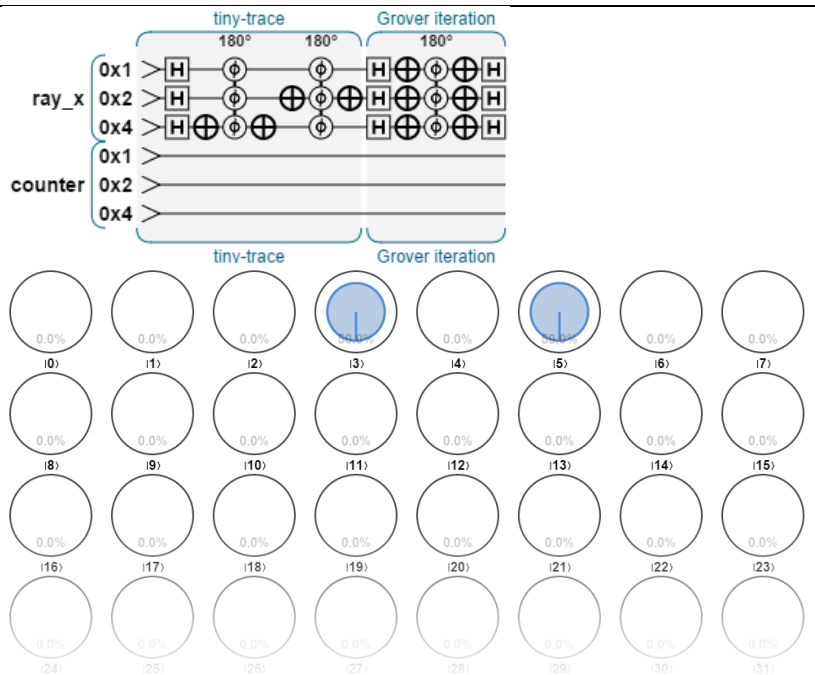
function tiny_trace(x, objects)
{
    qc.codeLabel('tiny-trace');
    for (var i = 0; i < objects.length;
    ++i)
    {
        x.not(~objects[i]);
        x.phaseShift(180);
        x.not(~objects[i]);
    }
}

var x_bits = 3; // number of bits in
the x coordinate
var p_bits = 3; // number of bits in
our counting precision

qc.reset(x_bits + p_bits);
var ray_x = qint.new(x_bits, 'ray_x');
var counter =
    qint.new(x_bits, 'counter');
var object_positions = [3, 5];

qc.write(0);
ray_x.hadamard();
tiny_trace(ray_x, object_positions);
grover_iteration(ray_x);

```

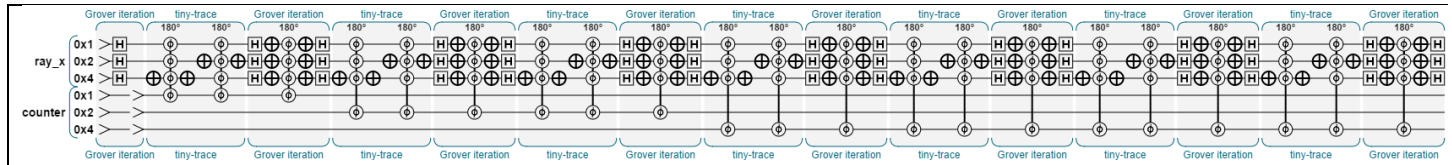


So far so good. So now we should get a periodic result based on the number of iterations.

So what is **counter** for? To start off, that's going to count how many times we do the detection and Grover iteration. A count value of 2 means we do it 2 times, so with this mechanism we can do it between 0 and 7 times. Let's code this up and test.

Here's how it works:

- If the 1 bit is set in counter, run `tiny_trace()` and `grover_iteration()` once.
- If the 2 bit is set in counter, run `tiny_trace()` and `grover_iteration()` twice.
- If the 4 bit is set in counter, run `tiny_trace()` and `grover_iteration()` four times.



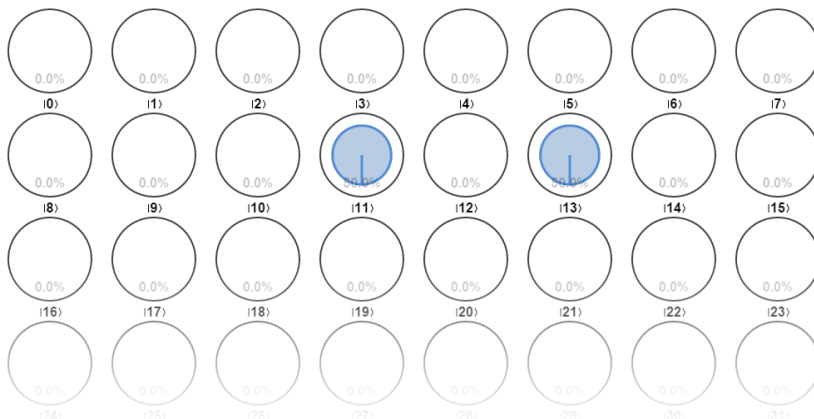
```
qc.write(0);
ray_x.hadamard();
counter.write(1);

for (var i = 0; i < p_bits; ++i)
{
    var iter = 1 << i;
    var condition = qintMask([counter, iter]);
    for (var j = 0; j < iter; ++j)
    {
        tiny_trace(ray_x, object_positions,
            condition);
        grover_iteration(ray_x, condition);
    }
}
```

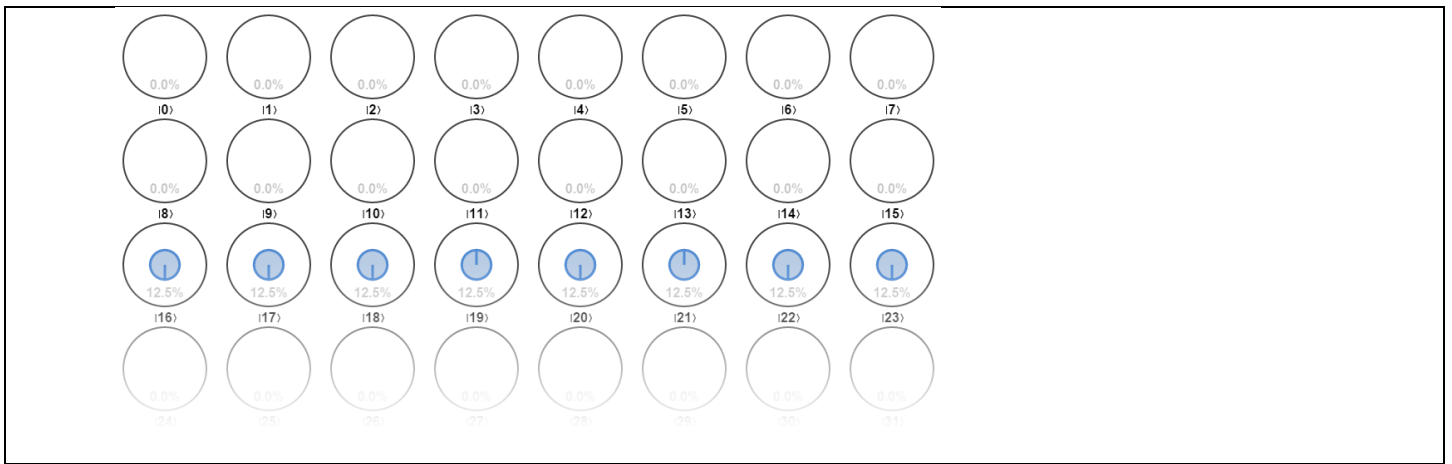
```
function tiny_trace(x, objects, condition)
{
    qc.codeLabel('tiny-trace');
    for (var i = 0; i < objects.length; ++i)
    {
        x.not(~objects[i]);
        x.phaseShift(180, ~0, condition);
        x.not(~objects[i]);
    }
}

function grover_iteration(x, condition)
{
    qc.codeLabel('Grover iteration');
    x.hadamard();
    x.not();
    x.phaseShift(180, ~0, condition);
    x.not();
    x.hadamard();
}
```

If we set counter to 1 and run this, we can see that the 1 row holds the result of one iteration.



Now, just by setting count to 2, we get a new result, in a new row! Recall that with 8 positions and 2 objects, the second iteration gets us back to the original state.

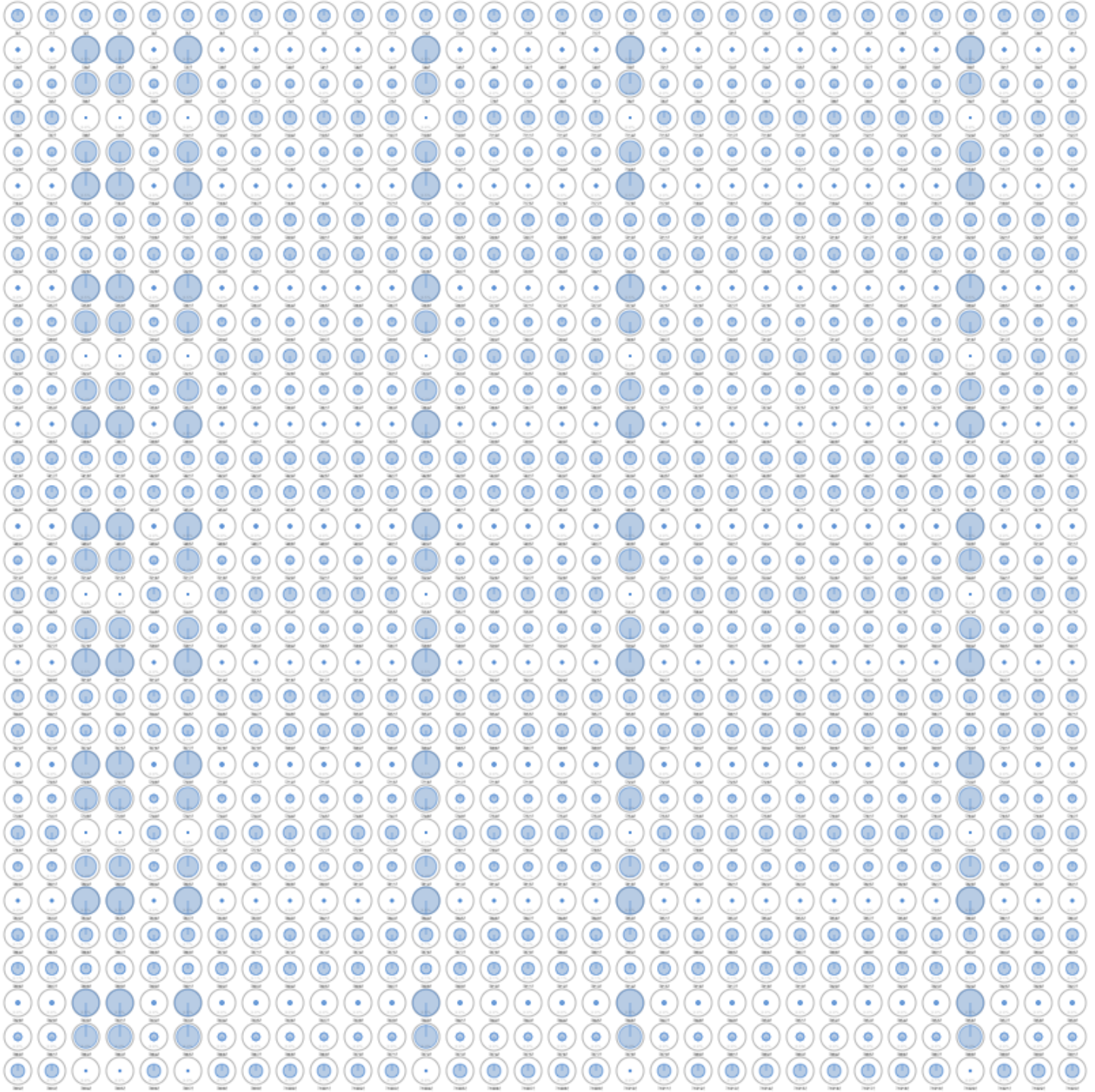


You know what's coming next. We're going to use quantum awesomeness. By setting counter to all possible values, we can fill all of these rows at once! No extra computation, same program. Just pop a Hadamard on counter before running it.

The Beautiful Part

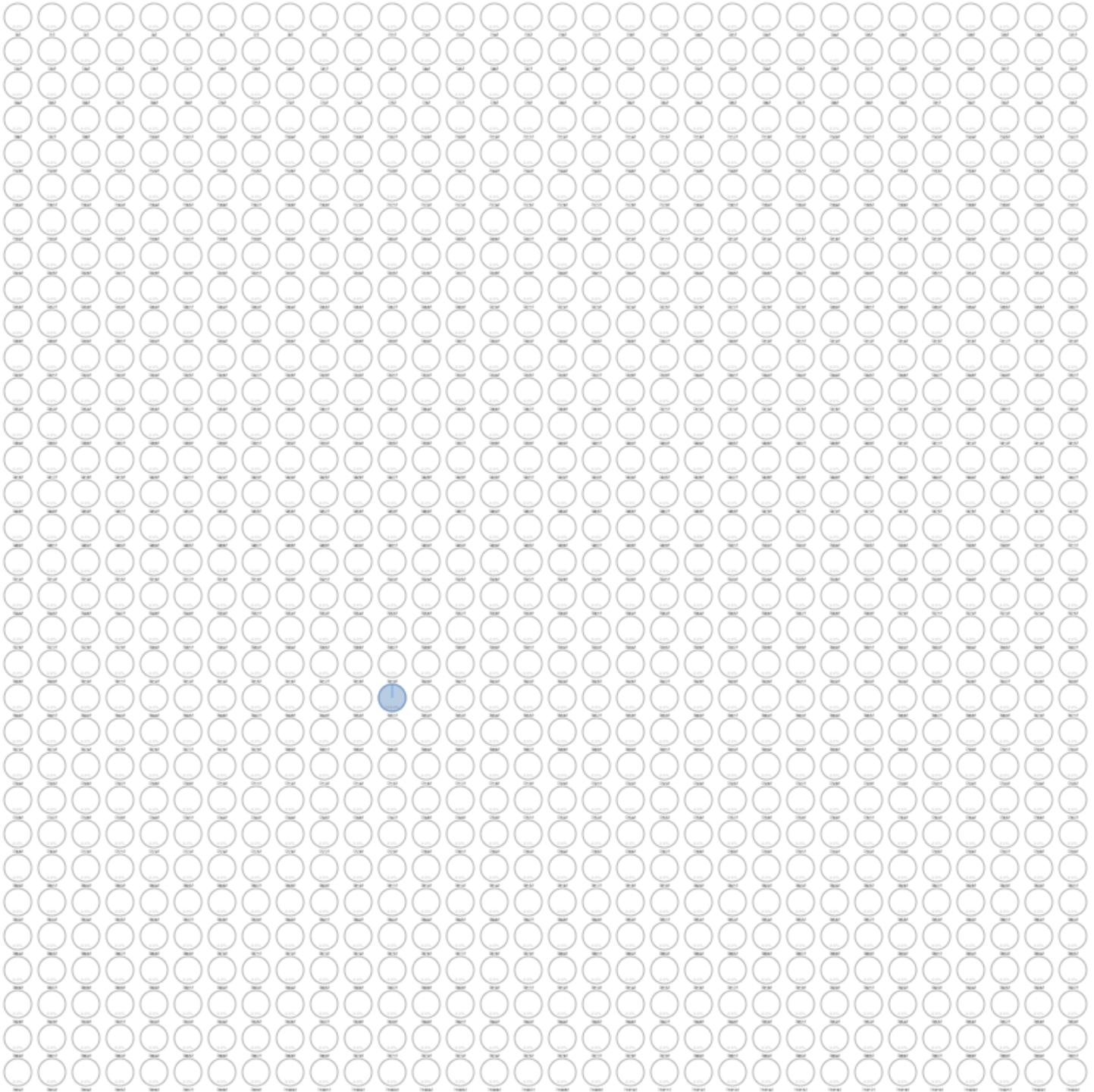
Let's take just the steps we've done so far, use 5-bits each for ray and counter, add some objects, and wander aimlessly.

If you wear glasses, take them off. Otherwise, just relax your eyes, and maybe try standing different distances away from the image, or adjusting the zoom on your screen. Focus is not important, but seeing the whole image is. Let your eyes wander. If you can, try to let each eye wander on its own. You're not supposed to do that, so do it.



In one sense, this is a 2,048-dimensional object ($2 \times 32 \times 32$), which your 2-dimensional retina can see all at once, seeking patterns naturally. In another sense, this is just a 10-bit number, but you're seeing everything that might happen, at a glance. You can see each object the ray *might* have hit, how many objects *could* be hit, and the ripples produced when we Grover-iterate over it.

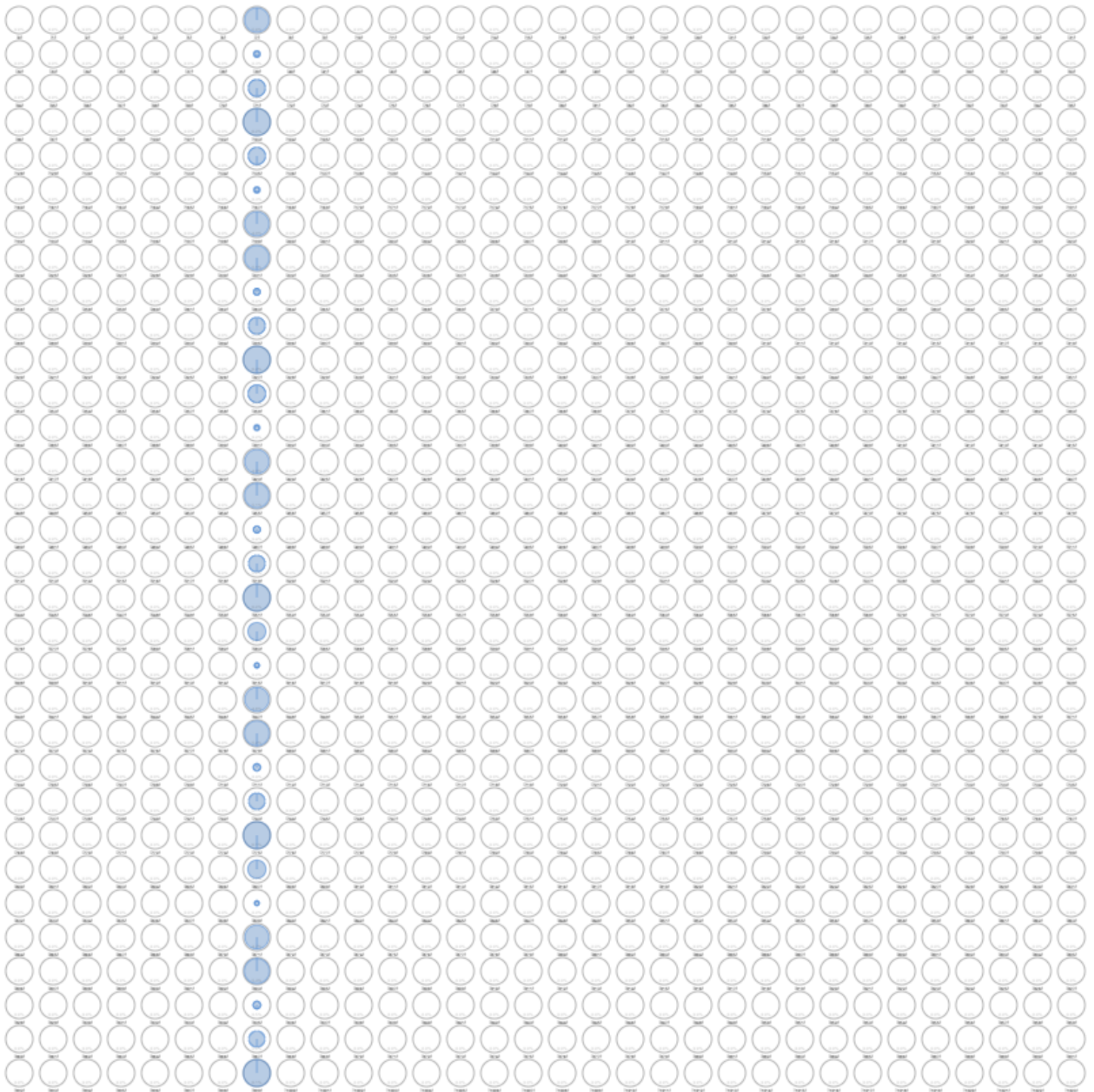
The moment we *read* this register, the snowflake melts into a drop. All of the things we *might* have experienced resolve into the one thing we actually *do*. We're left with a single dot, reporting just the sane digital logic of a 10-bit number.



Hopefully this one chosen number remembers its dream, and whispers some truth about the pattern it was a part of before we looked so closely. Can we tell which object was hit? Or how many there were? Hard to say. That's QC.

Moving Along

Instead of reading the whole thing, suppose we read only the ray position, and leave the counter alone. In that case, we resolve the state down to a single ray out of the 32 we might have launched.



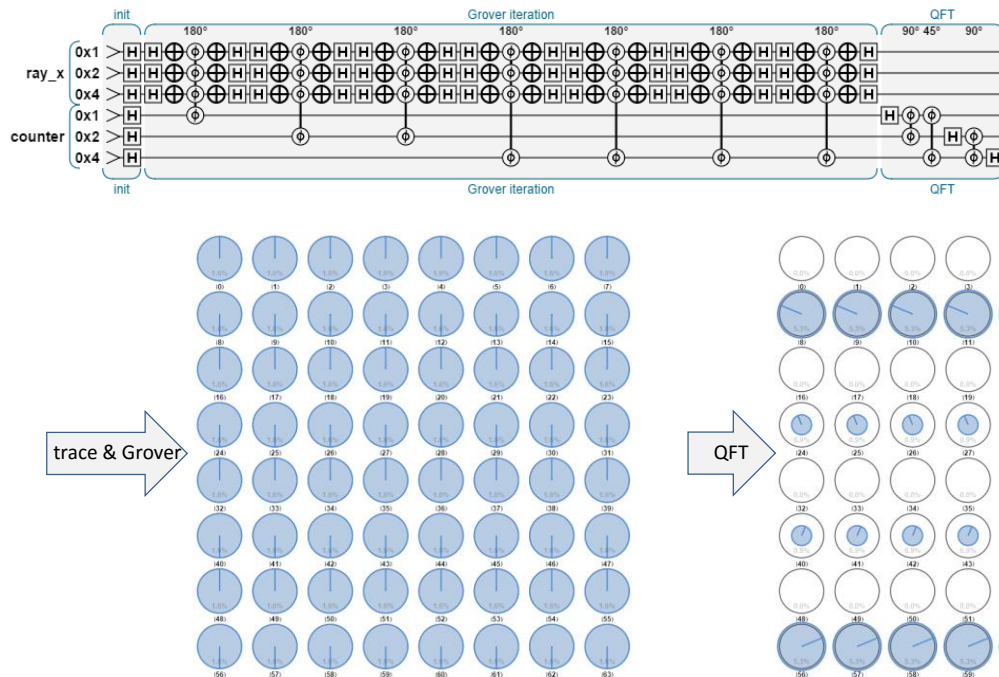
In this case, the ray position resolved to 7. Did this ray hit or miss? Hard to tell, though I think the fact that the second row is smaller than the first means it's a miss. What we do have, however, is a wave *frequency* which may tell us how many objects were there. Time for the QFT.

Just Try It Already!

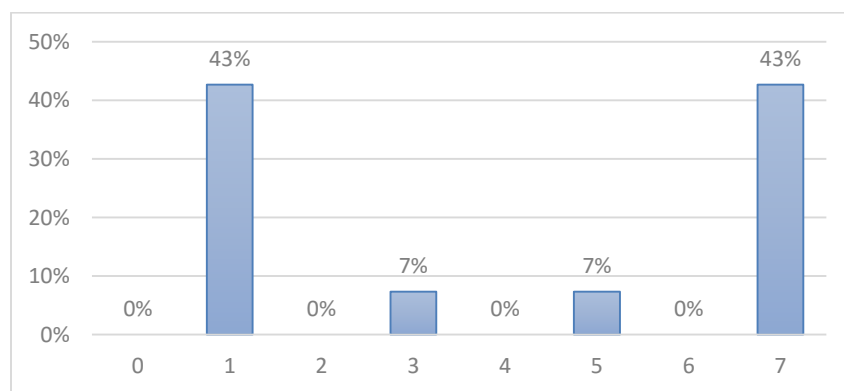
Here's the theory: We can run some trace-and-Grover iterations, producing ripples where the frequency depends on the number of hits. Then, we can QFT the result, and then read **that**, and the spikes of highest probability will be the values we read, and from that we can determine how many hits there were.

It sounds plausible, but will need to be tested. Let's shrink back down to our 3-bit model to see what happens.

First, we'll run the "no targets" case.

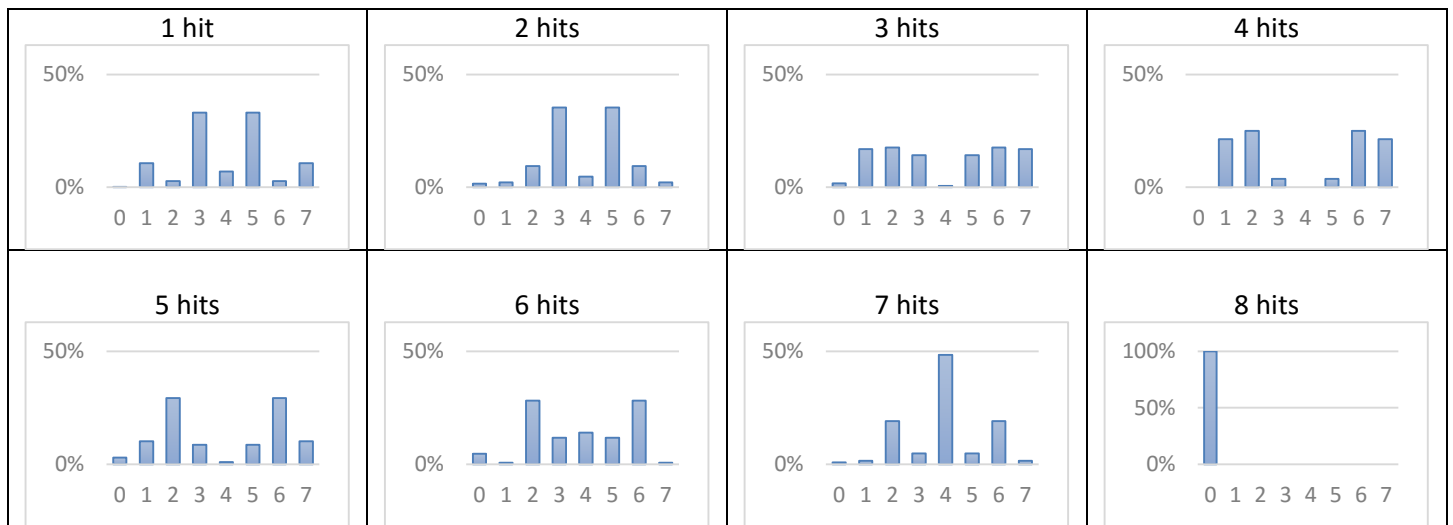


It looks like if we read the counter register after the QFT, we'll most likely get a 1 or a 7. Here are the probabilities:

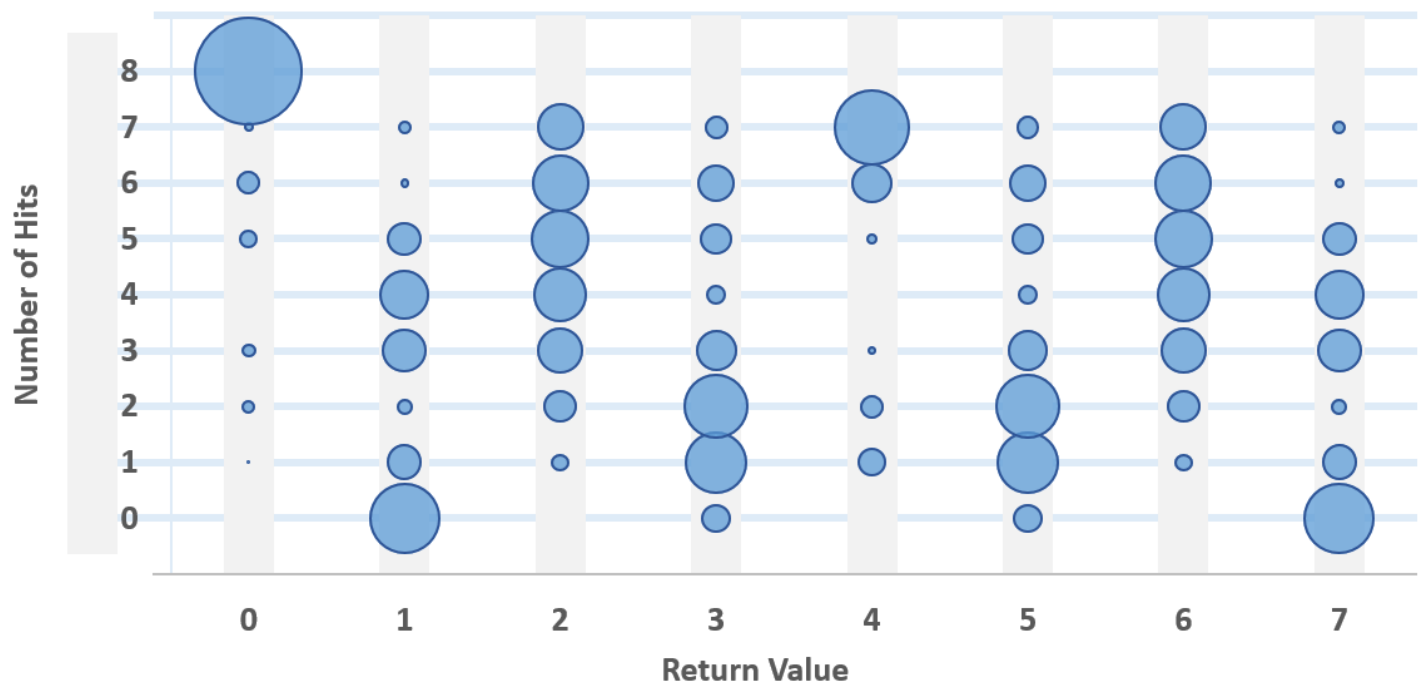


Yep, 86% chance we'll get a 1 or a 7. So now we can state boldly and with no actual confidence whatsoever that "Any time this program returns 1 or 7, there were no hits!"

Clearly we've got a problem with information density. We've used two of our possible return values, and we've still got eight more possibilities we want returned (each possible number of hits.) Never mind that, let's see what happens with other hit numbers.

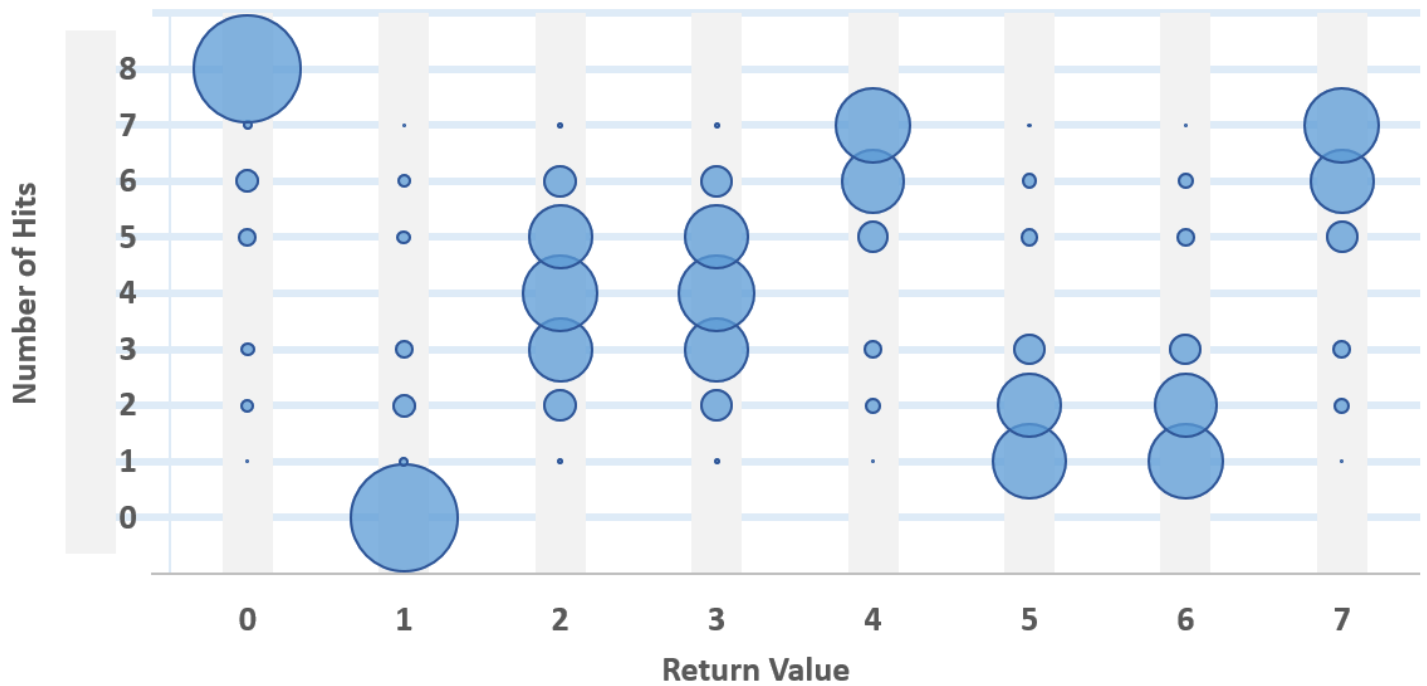


Yeah, we've got a problem. The good news is if it's zero we're pretty positive we had 8 hits. If it's a 4, we've *probably* got 7 hits, but maybe 6. Other than that, we're guessing. The 1-hit and 2-hit cases are nearly identical. Here they are, all stacked up:



That's a mess, and we're pretty much hosed. For most return values, we pretty much have no idea how many hits there were.

Before giving up, let's try exactly the same thing, but with an inverse QFT.

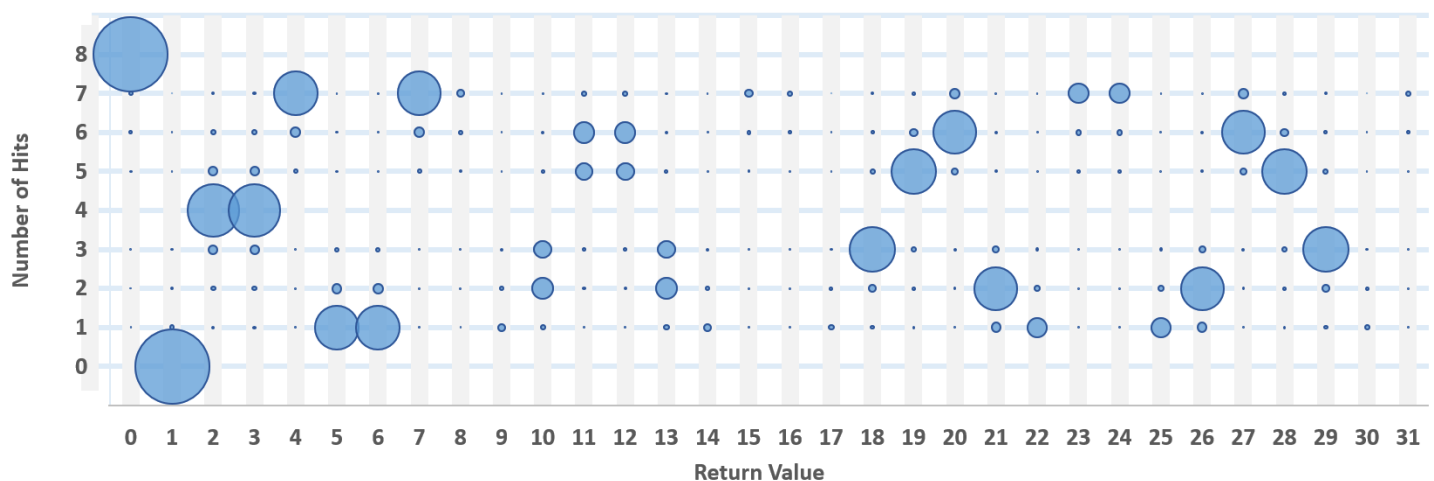


This looks more useful for sure. We still have an information density issue, but things are much better grouped. Why? Not sure. Stay on target. Here's what we know:

- If this program returns 0, then we've got 8 hits.
- If this program returns 1, then we've got 0 hits.
- If this program returns 2 or 3, then we've probably (about 93%) got 3, 4, or 5 hits.
- If this program returns 4 or 7, then we've probably (about 84%) got 6 or 7 hits.
- If this program returns 5 or 6, then we've probably (about 84%) got 1 or 2 hits.

Right. So with a high probability of success, we can determine how many hits there were, to within 1 or 2.

We'd like to be more accurate than that, so let's try to solve our information density problem by throwing more bits at counter. Instead of 3 bits (trying to use 8 return values to resolve 9 possibilities), let's go crazy and use 5 bits. Wild. Here's what we get:



Success! Information density issue solved. For most values (certainly the likely ones), we can peg the exact number of hits, usually with 95% accuracy. If the program returns a 3, we know there were 4 hits, just by looking it up in a table we can make later.

Have you spotted the problem yet? Yup. With counter at 5 bits, we're running 31 trace+Grover iterations to fill out the waves.

- Classical (old) method: **100% accuracy**, running the trace **8 times**
- Quantum (new awesome) method: **95% accuracy** running the trace **31 times**

Hm. That's not great, but at least it's working.

Wave Exploration

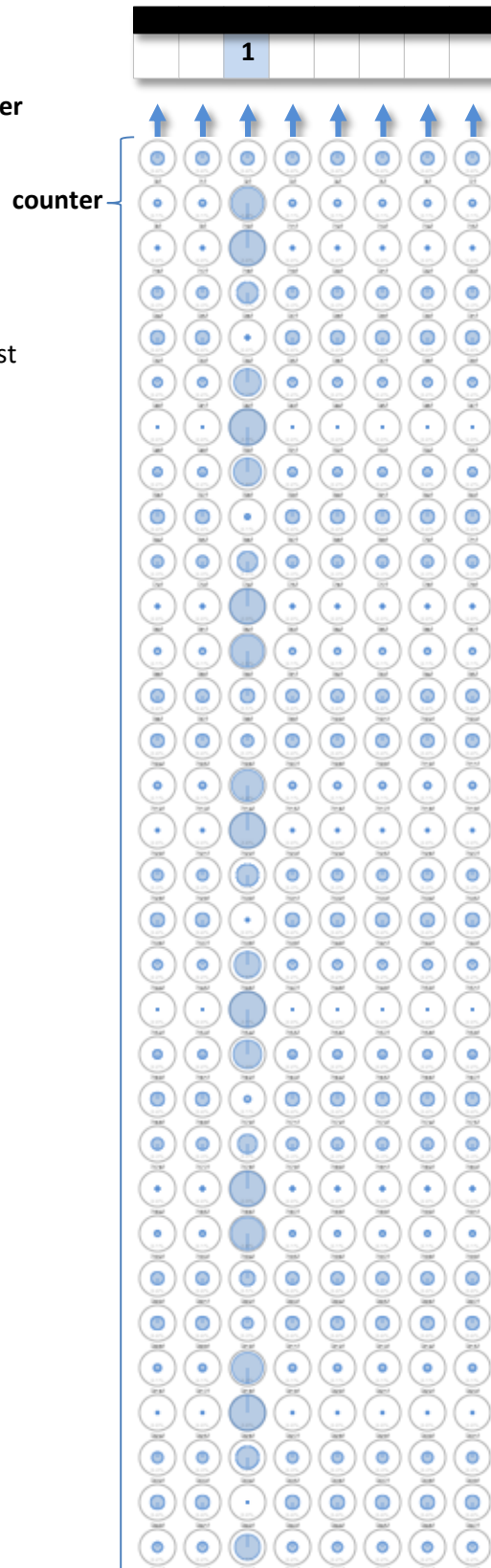
Our new problem boils down to this:

In order to be able to resolve individual hit counts, we want **counter** to be large. But each row requires a trace+Grover iteration to generate it. We'd like to find a way to keep **counter** large, while running fewer traces.

All we really need is the frequency of the wave, so after a certain point, repeating the wave isn't *really* adding any information. We know the frequency's not going to change.

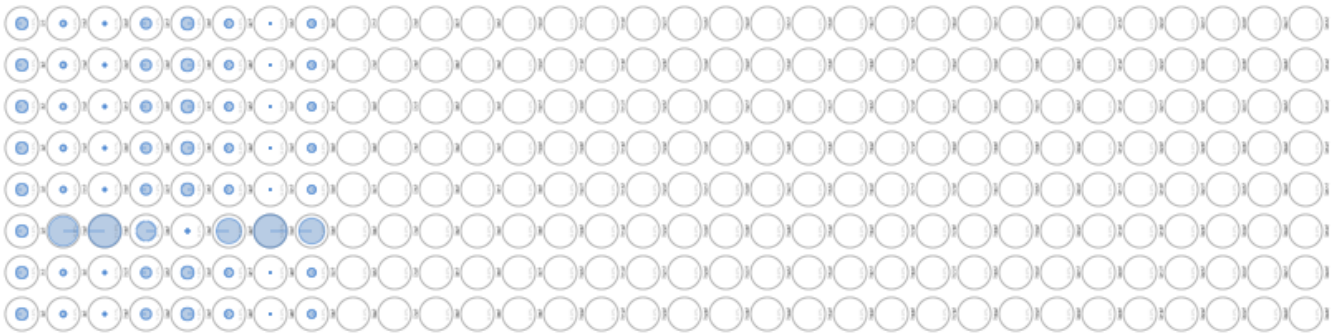
Maybe we can generate a couple of waves (or maybe even just part of one), and "fill" the rest with something which is cheap to generate and won't add too much noise to our results.

Let's try a few options.

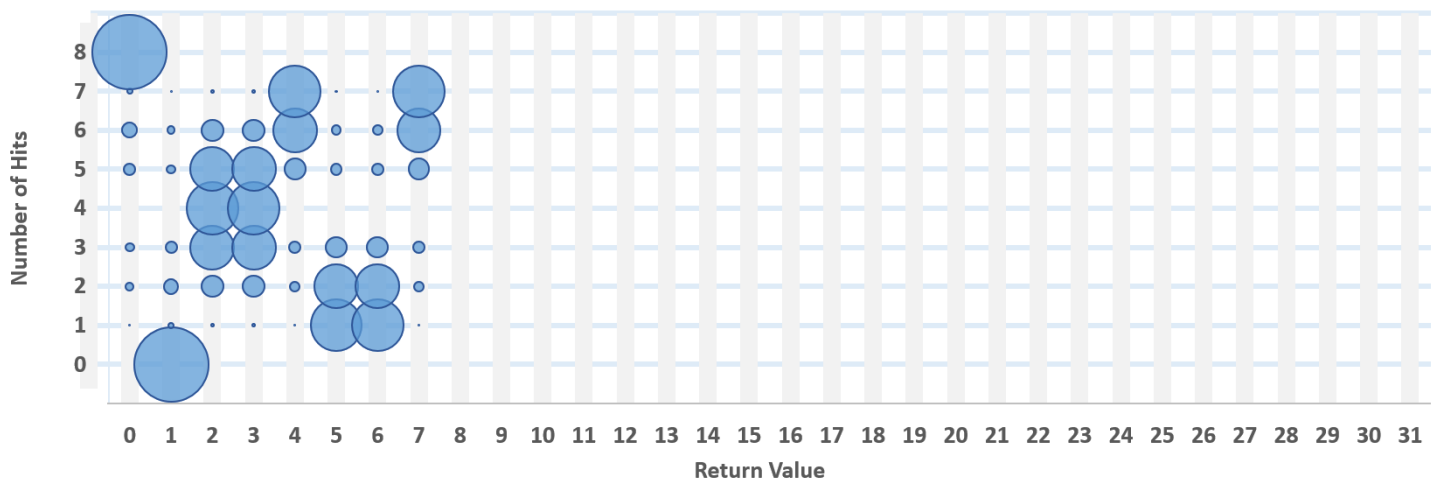


Zero-fill method:

Here, we only run 7 traces, and leave all of the other entries at zero.

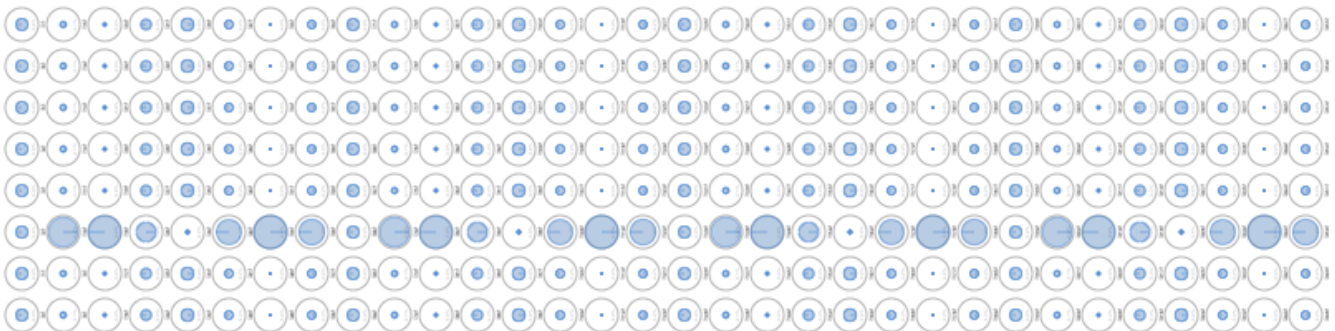


It works, but we end up with the same answer we get by just using counter=8 anyway, so that's no help.

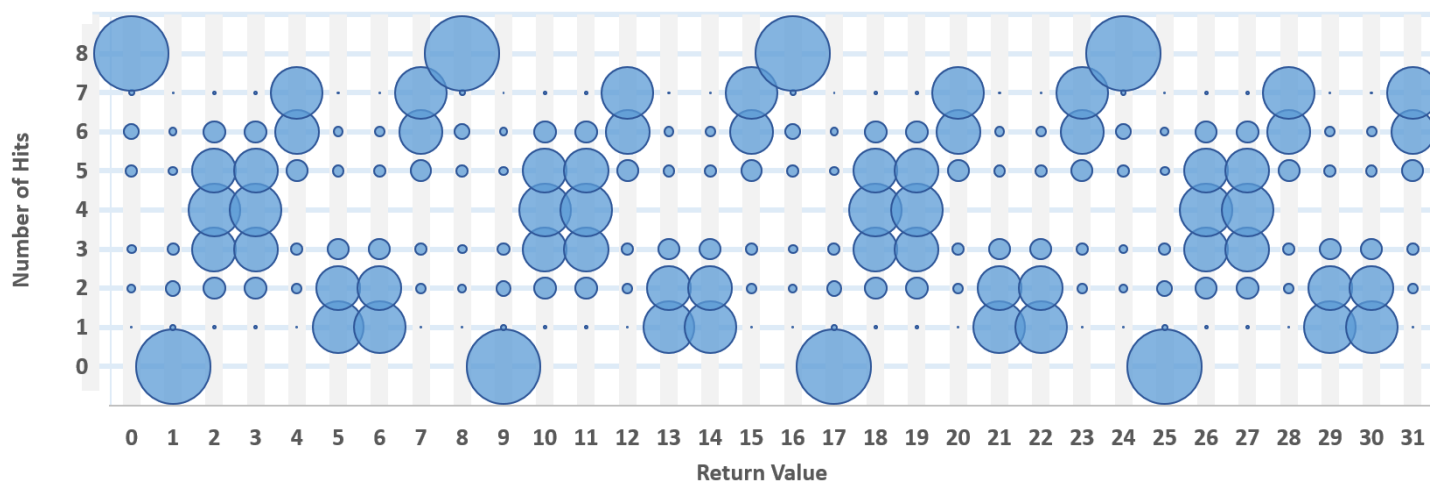


Repeat-fill method:

Here, we still only fill the first 8, but do it in such a way that the answer is replicated into the rest of the slots.

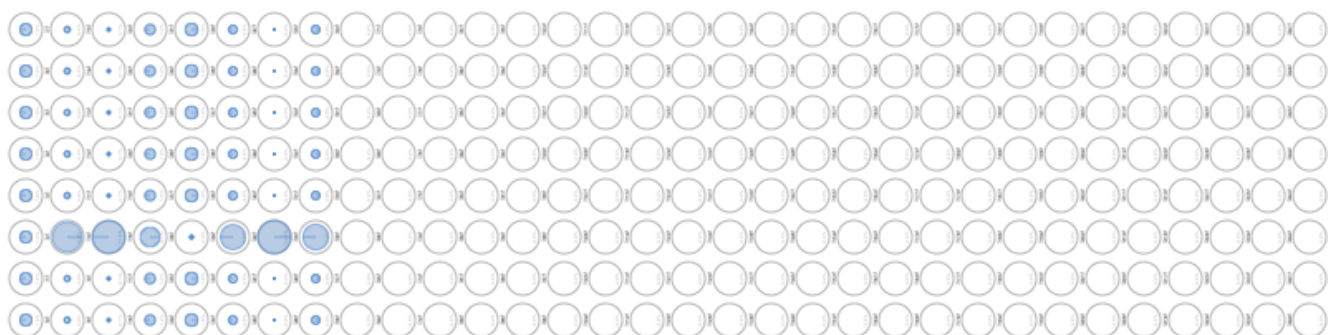


This works too, but it's no help, because we end up with an answer which is replicated 4 times, and has the same problems as the original.

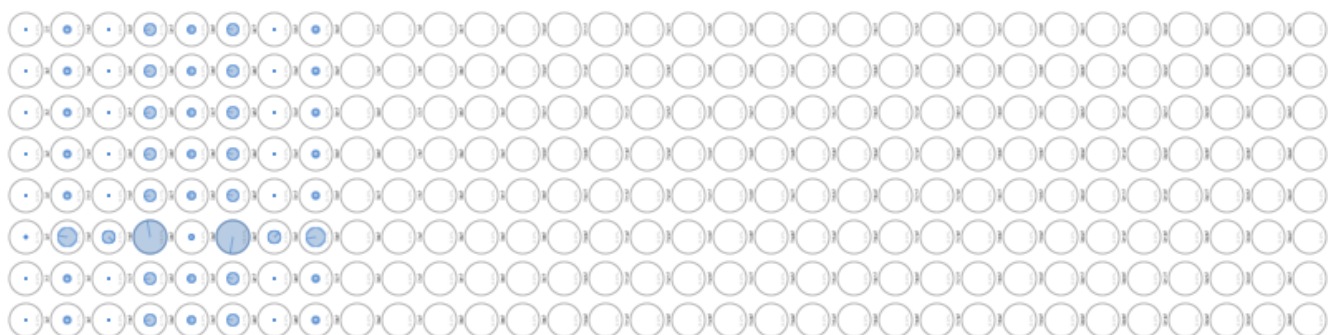


QFT-expand method:

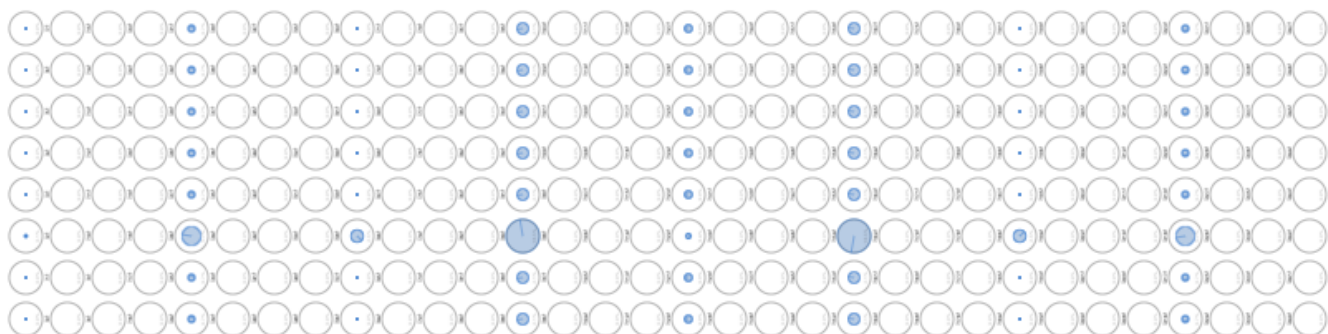
Here's a tricky one. First, fill 8 slots...



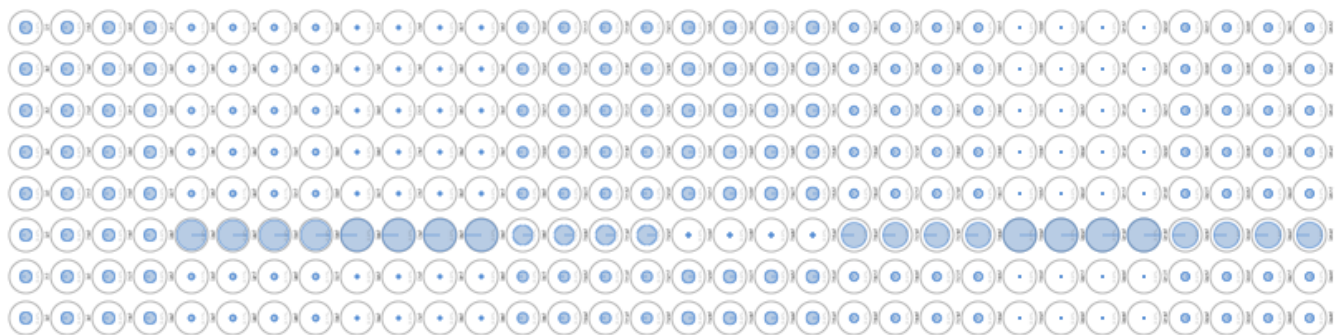
...then do a QFT to get the frequencies...



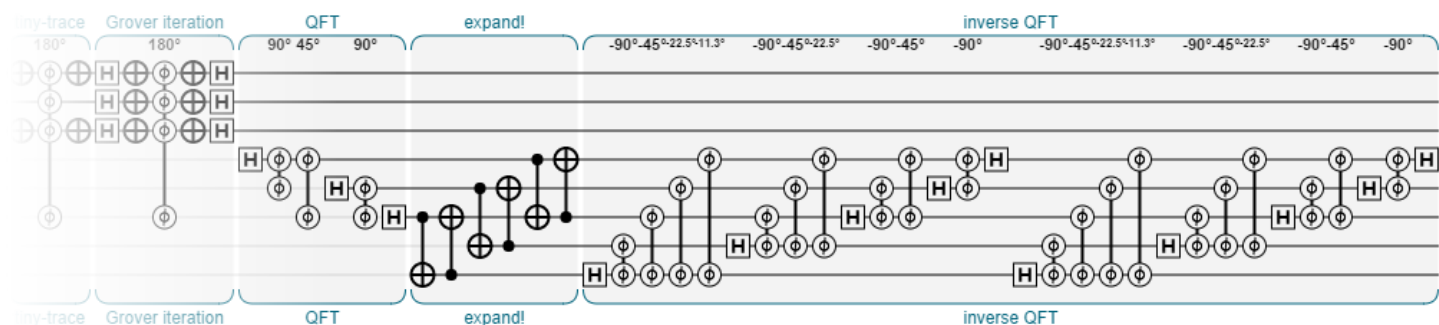
...then use some CNOT gates to space out the frequency spikes...



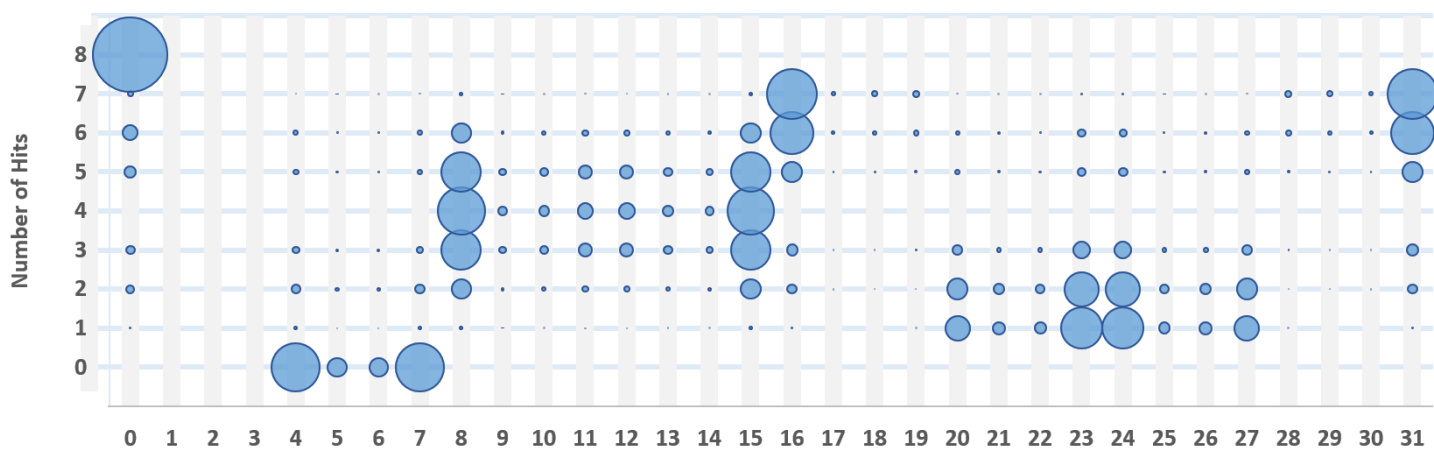
...then inverse-QFT to get a spread-out version of the original waveform...



and then inverse-QFT it *again* like we normally would, and see the answer. That whole “QFT at one resolution and then double-inverse-QFT at a higher resolution” acrobatics ends up looking something like this once the iterations are done:



Spiffy trick, but it’s really no help. We haven’t added any information, so the values are still stuck together, just as much as the originals:



Dang. Okay, time for a new approach.

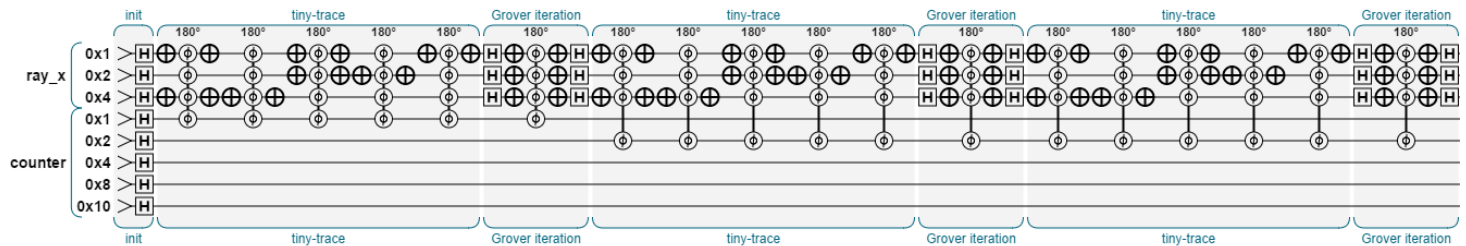
Iteration Cost

There may be a way to fix the information density problem, but first maybe we can step back and take a different approach.

This entire exploration assumes that the cost of a ray trace is substantial, and we want to reduce the number of them that we need to do. **Suppose** there were a way to avoid re-tracing for each trace-and-Grover iteration. Maybe we can trace once, save the answer, and then re-use it for each iteration. Actually, that might be kind of easy.

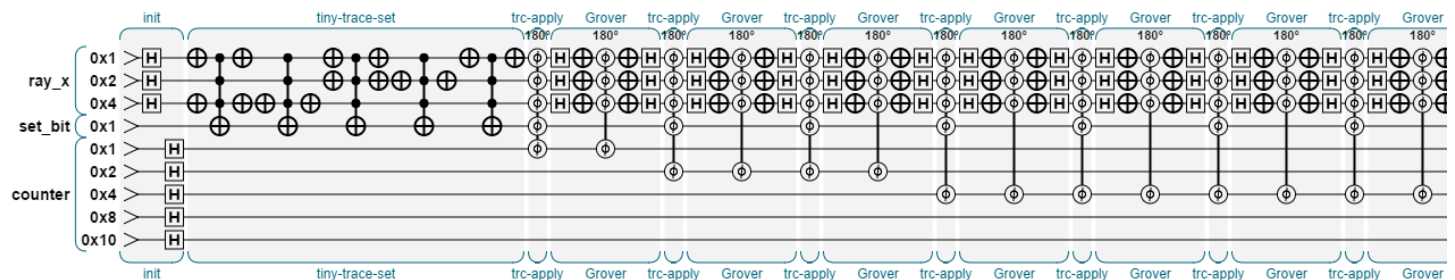
Currently, we're doing something like this:

(And yes, this tiny-trace is non-optimized. That makes it a good example here.)



The tiny_trace() function here is simple (and non-optimized, but it's just an example). The real tracing functions will be much more complex.

Let's add a qubit, and replace tiny_trace() with tiny_trace_set(), which will set the qubit whenever there's a hit. This is closer to our original program anyway.



Now we're talking. If this actually works, then we'll be able to trace just **one** ray, and get all of the information we need, at any resolution we want. Seems too good to be true.

Here's our old tiny_trace:

```
function tiny_trace(x, objects, condition)
{
  qc.codeLabel('tiny-trace');
  for (var i = 0; i < objects.length; ++i)
  {
    x.not(~objects[i]);
    x.phaseShift(180, ~0, condition);
    x.not(~objects[i]);
  }
}
```

Here's the new version (set and apply):

```
function tiny_trace_set(x, set_bit, objects)
{
  var mask = qintMask([x, ~0]);
  for (var i = 0; i < objects.length; ++i)
  {
    x.not(~objects[i]);
    set_bit.cnot(null, ~0, mask);
    x.not(~objects[i]);
  }
}

function tiny_trace_apply(x, set_bit, condition)
{
  var mask = qintMask([set_bit, ~0]);
  mask.orEquals(condition);
  x.phaseShift(180, ~0, mask);
}
```

It's really almost the same function, and the "apply" function we call for each Grover iteration is just one gate.

Turns out this is really hard, and didn't work as planned.

The problem is that you lose the oracle's ability to affect the period.

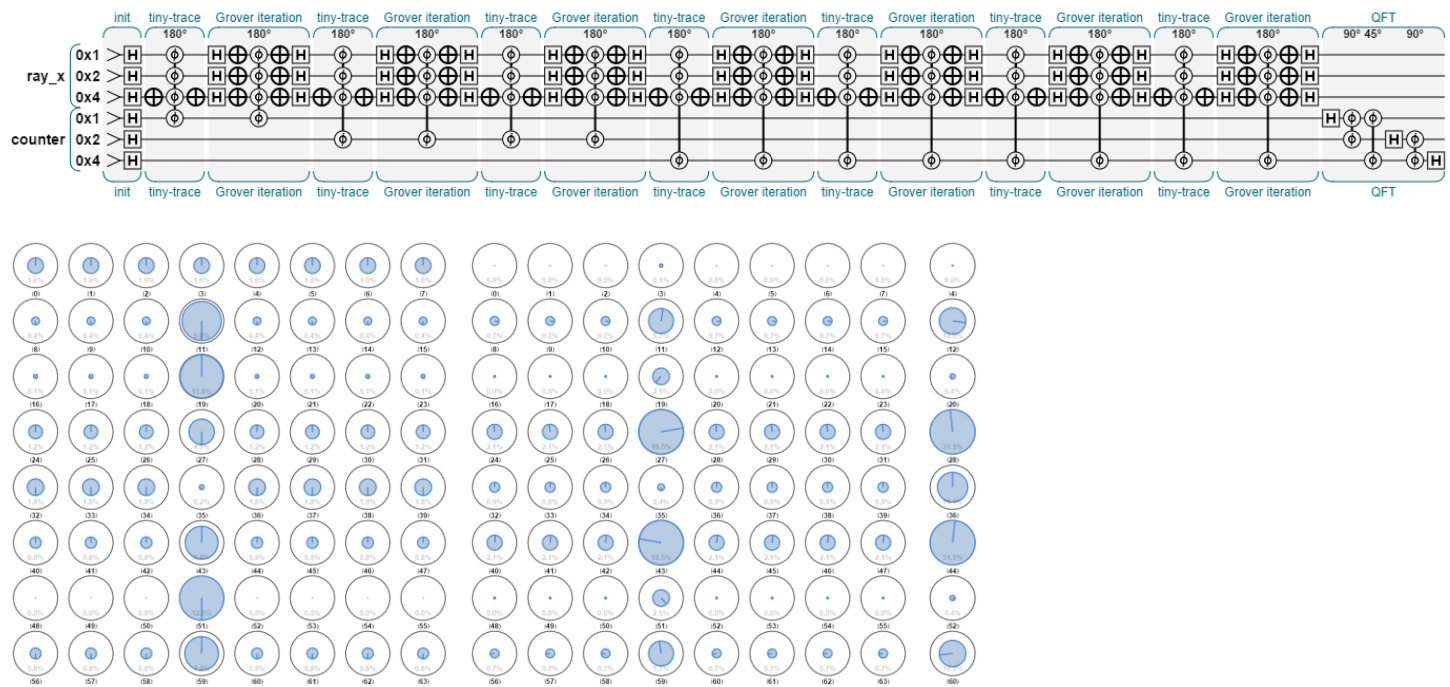
Tried just using an application bit, but that bit's state is messed up after the first application.

Next thing to try: stash/store a series of spikes, and then "re-apply" them with each iteration.

It looks like that may work! Time to re-organise this writeup into a new doc.

Okay, the results are good enough that I've started writing a more formalised version of this. That's not finished (just like this), but it can be found here:

- [Practical Quantum Supersampling for Computer Graphics](#)



[Under construction]

Conclusion

[to do]

Useful References

- [*Silicon Quantum Photonic Circuits for On-chip Qubit Generation, Manipulation and Logic Operations*](#)
- [Minds, Machines, and the Multiverse](#), by Julian Brown
- Elementary Gates for Quantum Computation (1995) [arXiv:quant-ph/9503016v1](#)
- [Quantum Computing for Computer Scientists](#) (Yanofsky and Mannucci), 2008
- IBM QC advances 2/28/2012:
 - <http://ibmquantumcomputing.tumblr.com/>
 - <http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html>
- [TODO: add other papers and books I've found useful]

About the Author

EJ and his muse live in a secret laboratory in San Francisco. Everything else you really need to know is either posted [here](#), or can be obtained by sending an email to ej@machinelevel.com.