The affordances of quantum art

Russell Huffman - Designer
Paul Kassebaum - Physicist

Agenda

Art and quantum computing introduction
(10 minutes)

Physics primer with Paul Kassebaum
(35 minutes)

Project tutorial with Russell Huffman
(35 minutes)

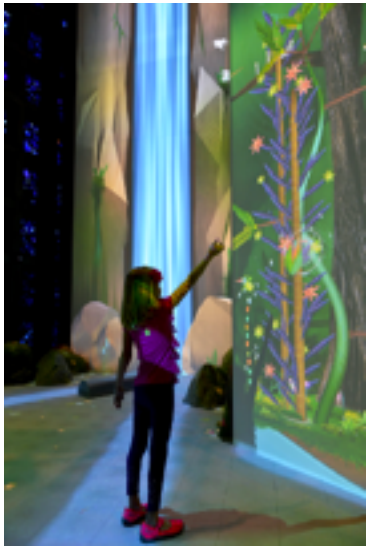
Q&A
(10 minutes)

Every technology and every artistic medium has a set of affordances that make them unique.

Watercolor by Cézanne

Woodcut by Dürer

Photograph by Eisenstaedt



Interactive installation by DESIGN I/O

We are at the beginning of a new technology with new creative opportunities.

We have the opportunity to define a new medium.

What are the creative affordances of quantum computing?

1

Superposition

2

Entanglement

3

Interference

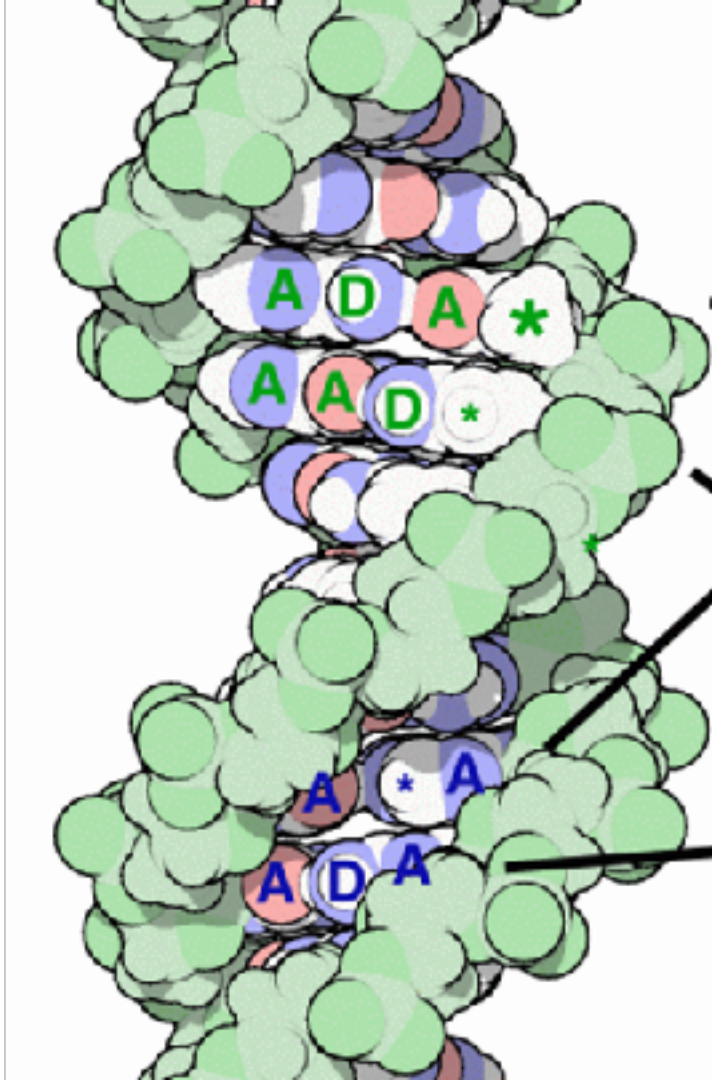
4

Noise

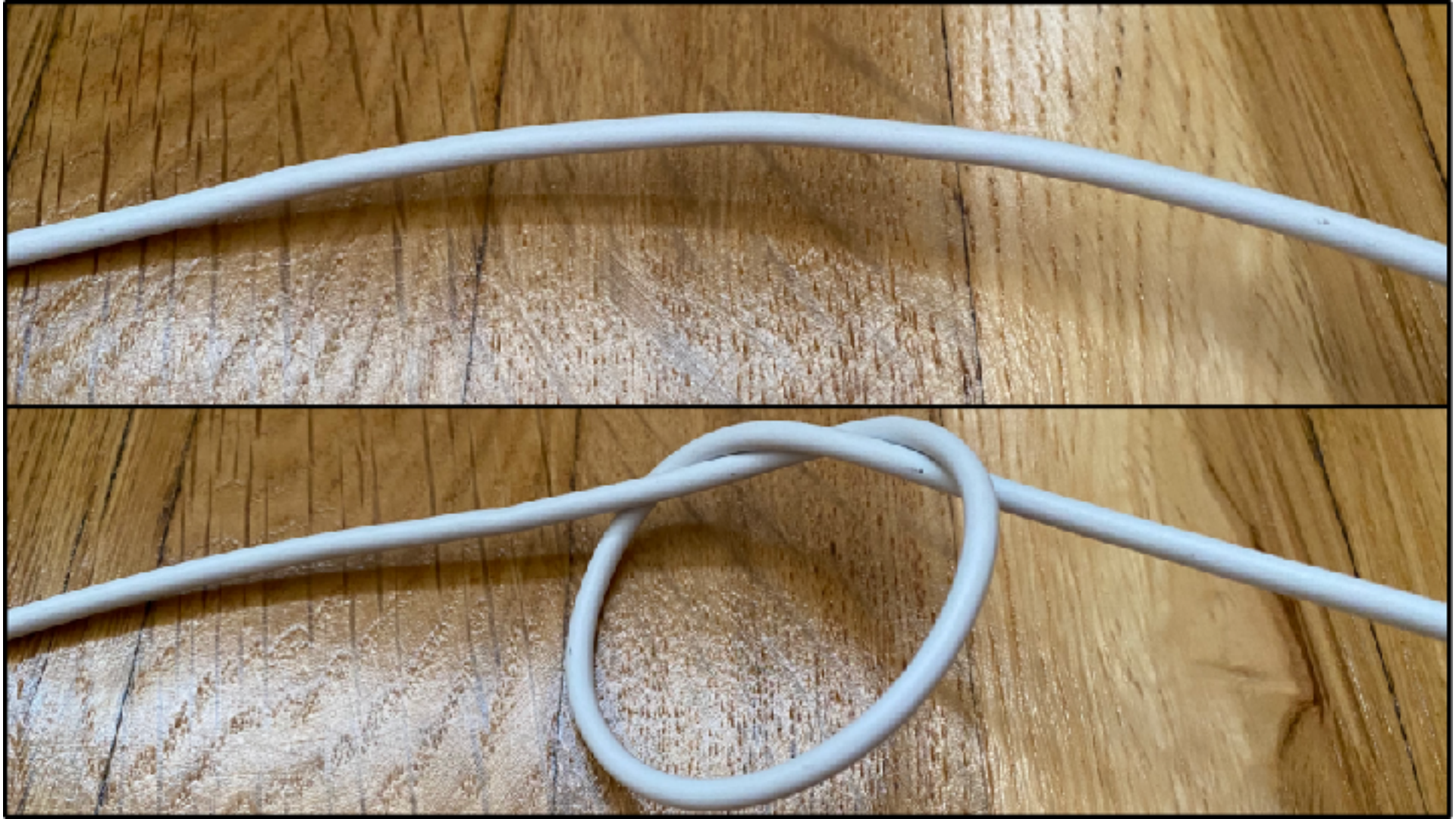
Intro to quantum computing for artists

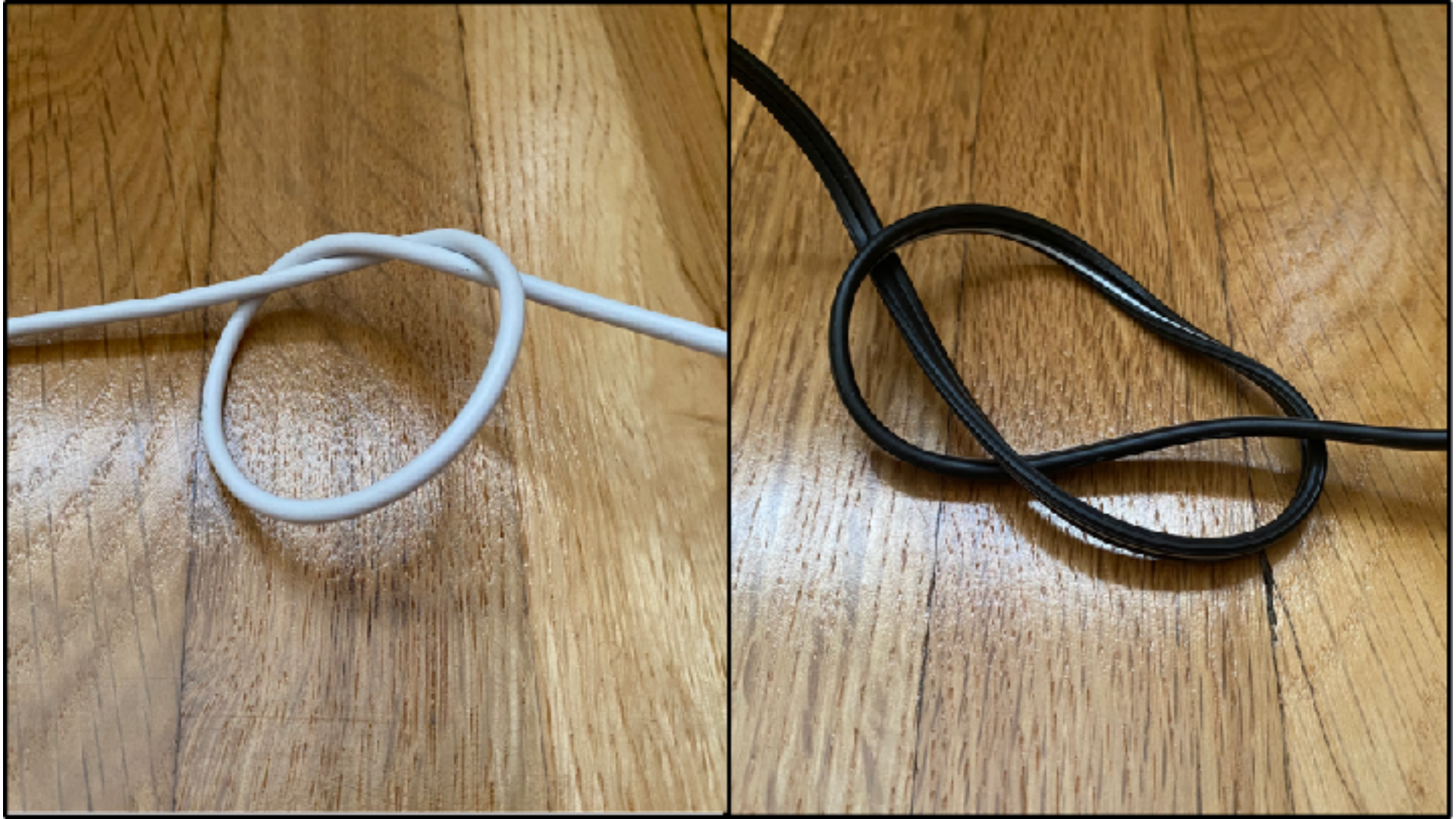
Dr. Paul Kassebaum
Physicist

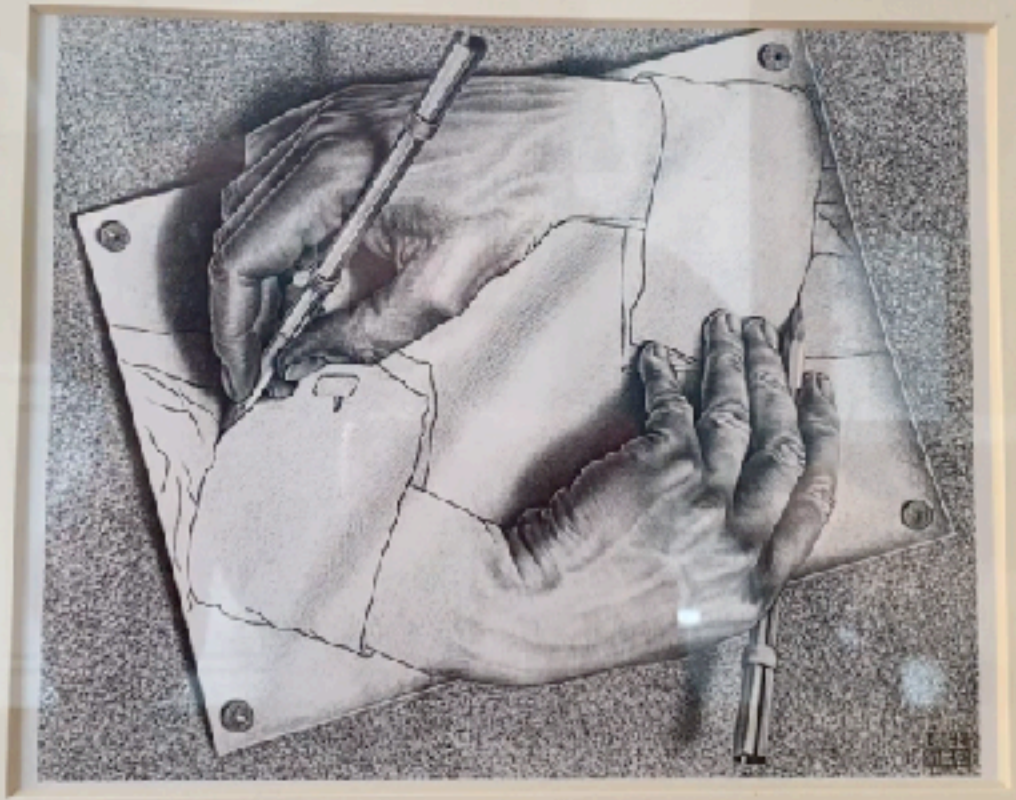
linkedin.com/in/paulkassebaum

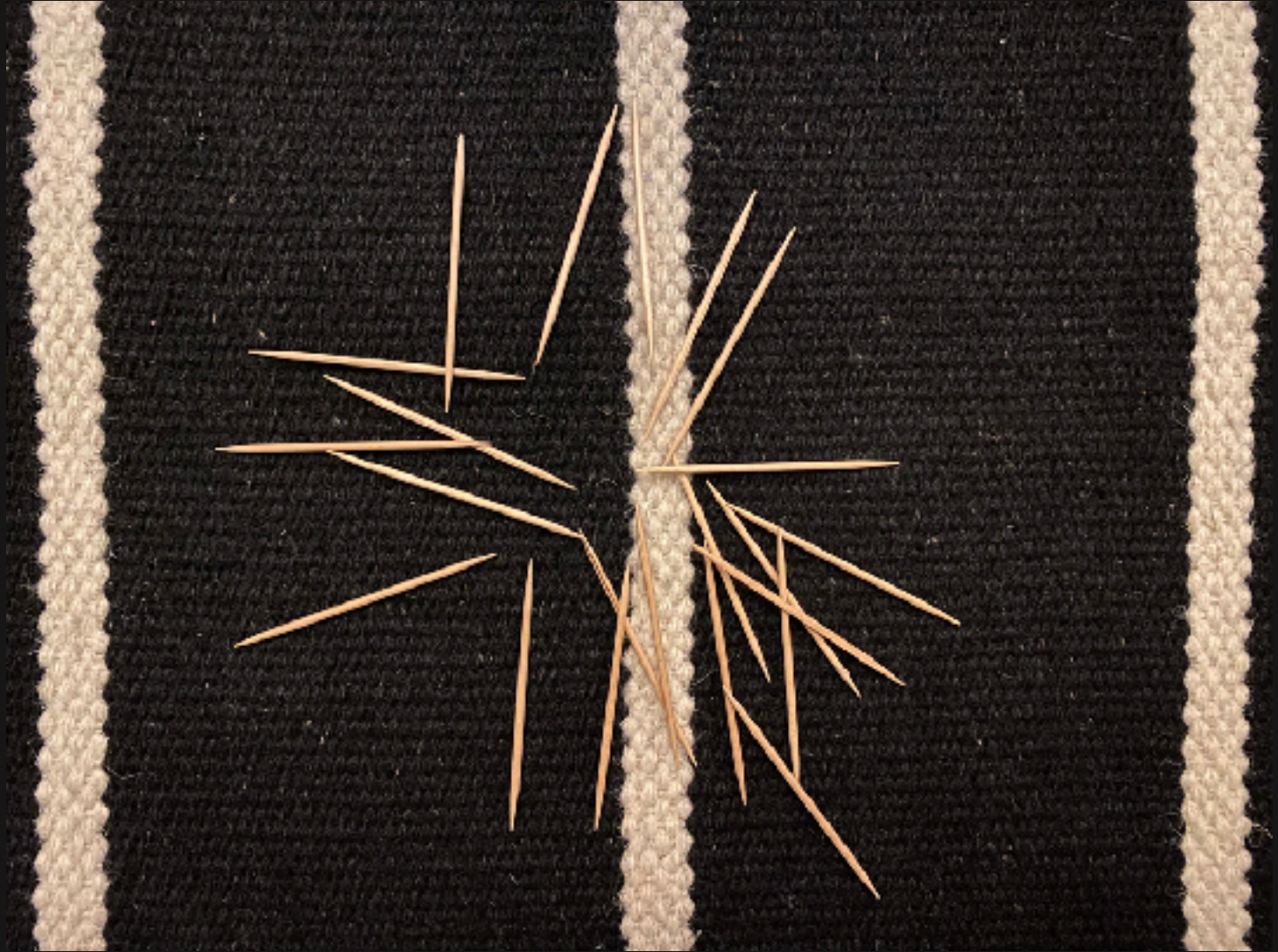

A D A *
A A D *
A * A
A D A

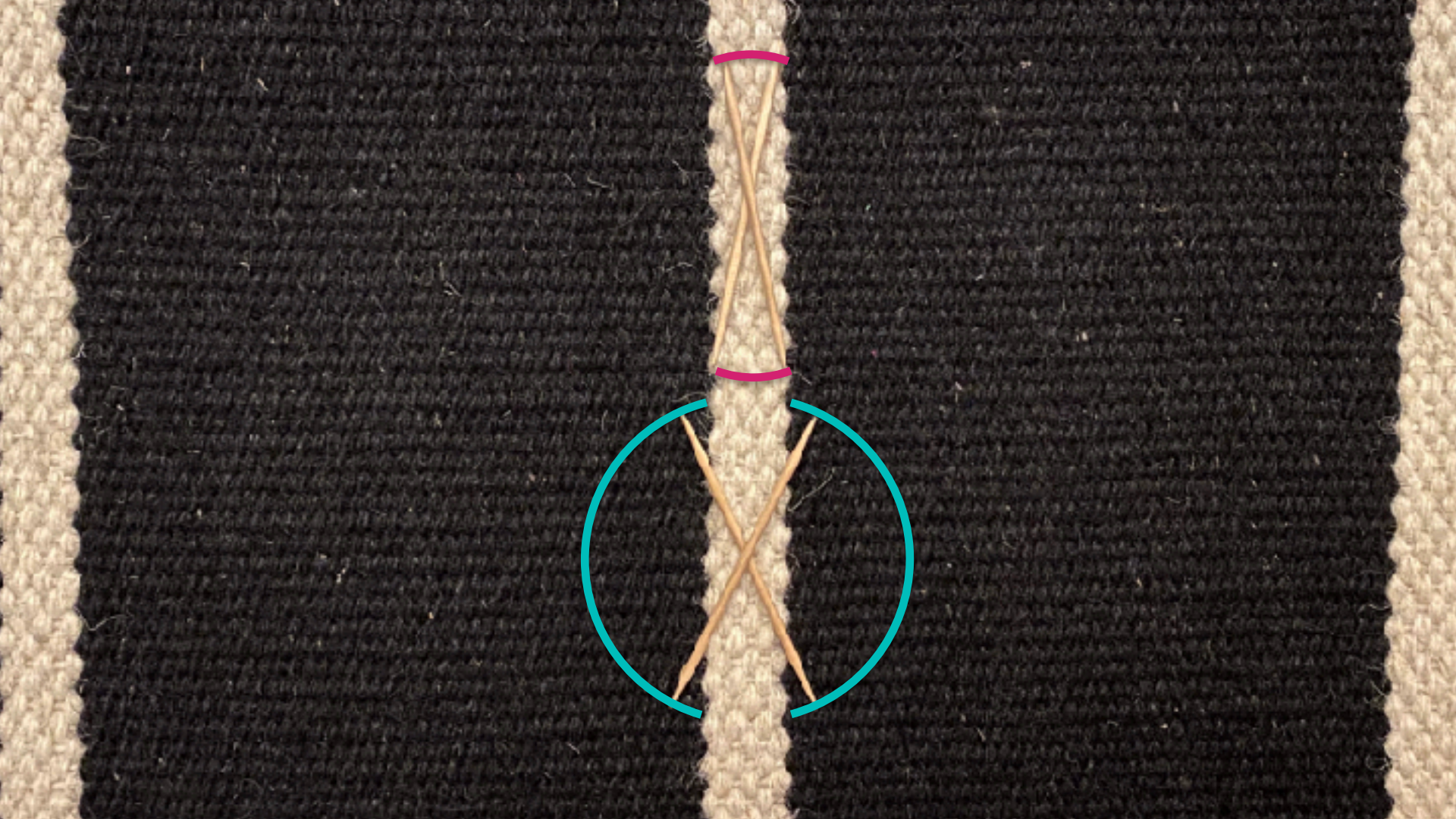

Q
Q

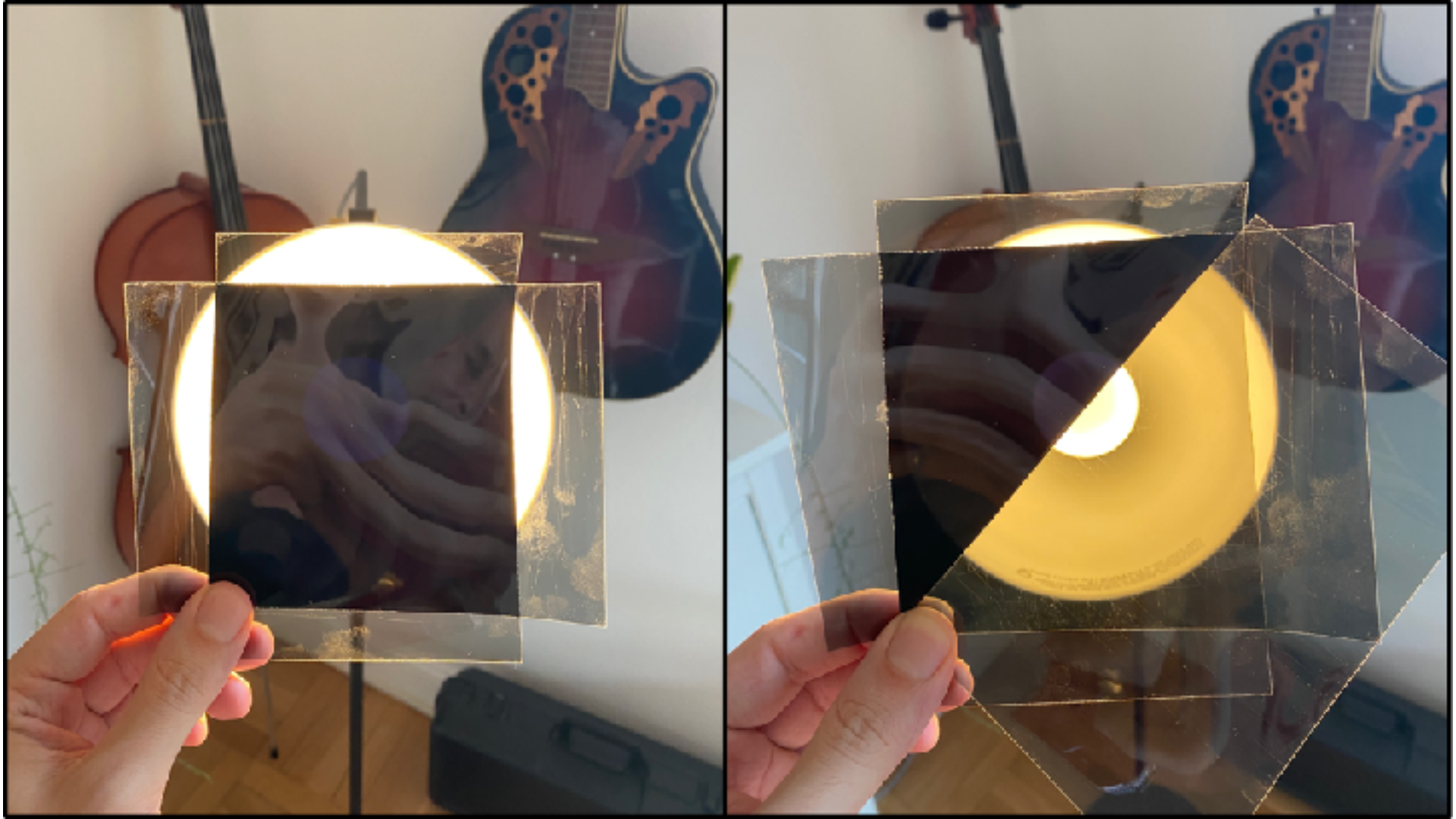










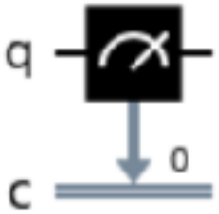


Properties of quantum information

1. Not all information attributes of a quantum information medium are distinguishable.
2. Undetectability of sharpness
3. Quantum information cannot be cloned
4. Pairs of observables not simultaneously preparable or measurable
5. Unpredictability of deterministic processes
6. Irreducible perturbation of one observable caused by measuring another
7. Consistency of consecutive measurements of a non-sharp observable
8. Quantization
9. Coherence and locally inaccessible information


Metaphor by
@ciralouise

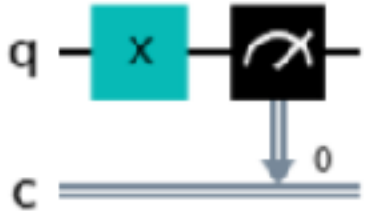
```
1 qc = QuantumCircuit(1,1)
2 qc.measure(0,0)
3 qc.draw()
```




```
1 sim = AerSimulator()
2 sim.run(qc).result().get_counts()
```

```
{'0': 1024}
```

```
1 qc = QuantumCircuit(1,1)
2 qc.x(0)
3 qc.measure(0,0)
4 qc.draw()
```




```
1 sim.run(qc).result().get_counts()
```

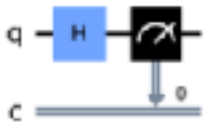
```
{'1': 1024}
```

```
num_bits = 10
qc = QuantumCircuit(1,num_bits)
for c in range(num_bits):
    qc.x(0)
    qc.measure(0,c)
qc.draw()
```

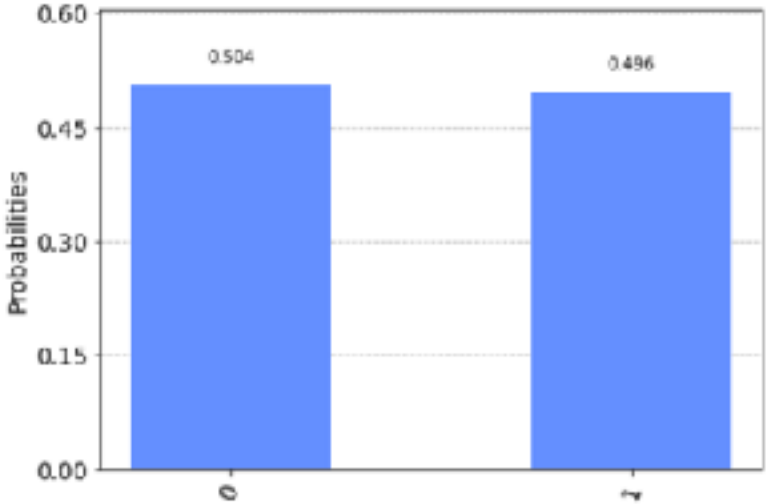

```
sim.run(qc).result().get_counts()
```

```
{'0101010101': 1024}
```

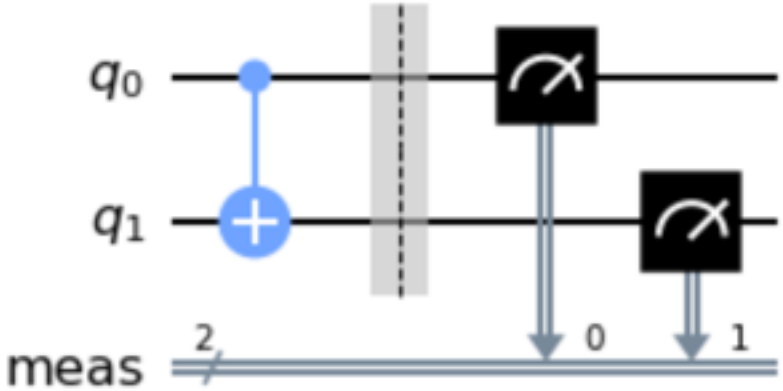
```
qc = QuantumCircuit(1,1)
qc.h(0)
qc.measure(0,0)
qc.draw()
```



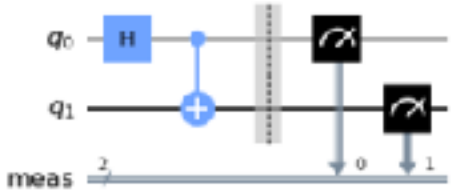

```
counts = sim.run(qc,shots=2048).result().get_counts()
plot_histogram(counts)
```



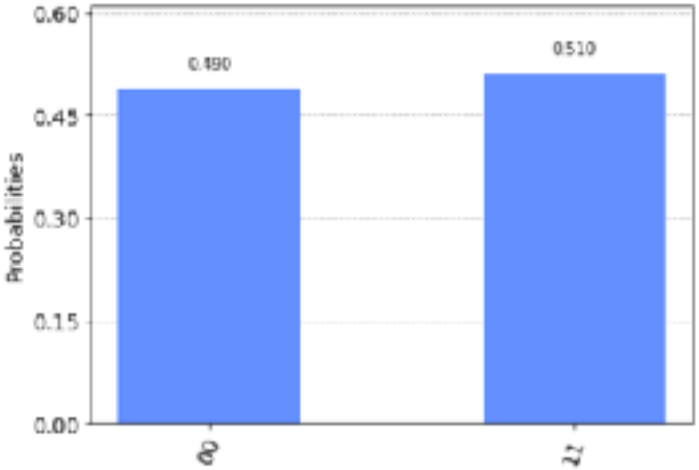
```
1 qc = QuantumCircuit(2)
2 qc.cx(0,1)
3 qc.measure_all()
4 qc.draw()
```



```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.measure_all()
qc.draw()
```




```
counts = sim.run(qc).result().get_counts()
plot_histogram(counts)
```



Intro to quantum computing for artists

Dr. Paul Kassebaum
Physicist

linkedin.com/in/paulkassebaum

Quantum computing art tutorial

Russell Huffman
Designer

Qiskit Slack workspace
@Russell

Medium
https://medium.com/@jrussellhuffman

Instagram
https://www.instagram.com/jrussellhuffman/

website
http://www.jrussellhuffman.com/

Tutorial overview

This tutorial will walk through creating a quantum art composition with the workflow I use. This (rather specific) workflow may or may not be the right workflow for everyone, but hopefully it will serve as a good starting place and foundation.

1

Simulate a quantum circuit (Processing + MicroQiskit)

2

Draw something with it (Processing)

3

Switch to real hardware (Qiskit)

4

Export composition (Processing)

My inspiration

Rothko worked with fundamental shapes, but was willing to let them be imperfect. To me, this makes for a great starting place for making art with a quantum computer.

Quantum computing itself is in its infancy. I want to create art that uses quantum fundamentals despite the imperfections of the technology.

“No. 22 (REDS)” by Mark Rothko Courtesy the Donald B. Marron Family Collection, Acquavella Galleries, Gagosian and PaceGallery

Rothko inspired quantum art

Idea: Create a simple artwork with simple rules drawing inspiration from Rothko

The quantum algorithm
One of the simplest quantum circuits that shows superposition and entanglement, called “The Bell State”.

Interpreting the data into art
The value from the Bell uses 2 qubits with 2 possible outputs, which are used to draw the composition.

Rendered image of composition on campus

Data interpretation can be a creative act, which opens up the doors for us creative types to work in highly technical fields like quantum computing.

Why Processing?

Pros:
1. Large community
2. Very widely supported
3. Thorough documentation
4. Can export to SVG

Cons:
1. Requires programming knowledge
2. Not a lot of boilerplate help

Other options:
Blender
Unity
Touch Designer
P5.js

Why Qiskit?

Pros:

1. Large active community
2. cloud and local support
3. Thorough documentation and educational material
4. Connects to real quantum systems

Cons:

1. none (duh)

Summary in pseudo-code

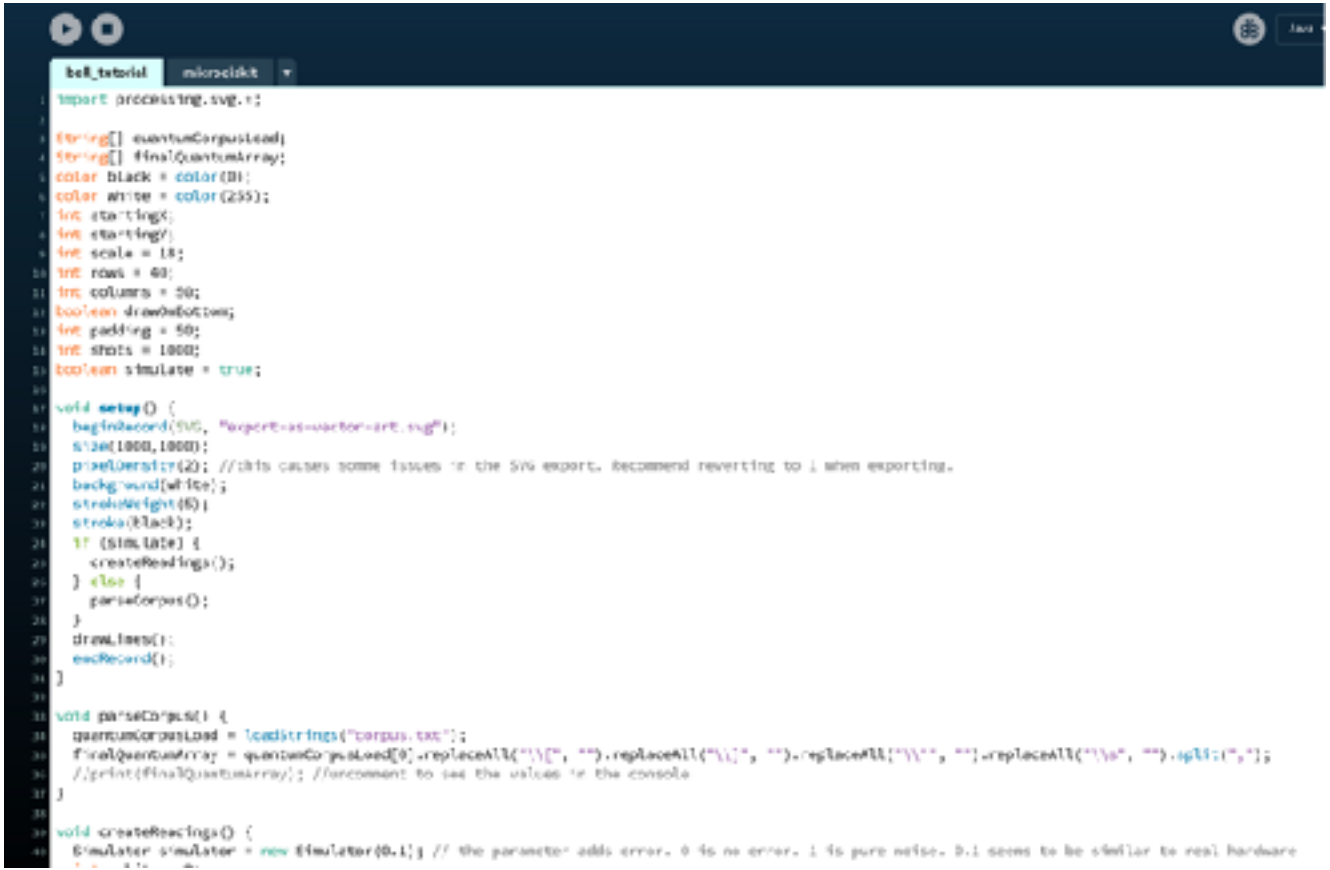
1. Open Processing.
    1. Go either the “starter_project” or “bell_tutorial” folder
    2. Open the .pde (processing) file of the same name
2. Create the bell circuit
    1. Simulate the circuit
    2. Store it’s data to use in the composition
3. Draw the composition with the data
    1. loop through all the shots from the quantum data
    2. For each shot, store the first and second values separately
        1. Determine the location of line: Use the first value to determine if it should draw on the top or bottom of the composition
        2. Determine the direction of line: Use the second value to determine if it should draw a horizontal or vertical line
    3. draw the line, based on the location and direction
4. Preview the composition
5. Switch from simulated data to real data
6. Export the composition

Open processing

Go to the “bell_tutorial” folder

Open the .pde (processing) file of the same name.

You will see a pre-populated project file. I encourage everyone go to https://processing.org/ to learn more about Processing overall.



Create the bell circuit

Scroll down to `createReadings()`. This function is used to call the microqiskit functionality. The syntax is meant to mimic standard qiskit syntax.

In the section “add circuit here”, there are two lines of code to represent the hadamard gate and the cnot gate.

```
void createReadings() {
  Simulator simulator = new Simulator(0.1); // the parameter adds error. 0 is no error. 1 is pure noise. 0.1 seems to be similar to real hardware
  int qubits = 2;

  QuantumCircuit phiPlus = new QuantumCircuit(qubits, qubits); //the first parameter is number of quantum registers (qubits). the second is classica

  // ---------------------------------------------------
  //add circuit here
  phiPlus.h(0);
  phiPlus.cx(0, 1);


  // ---------------------------------------------------

  //measure entire circuit:
  for (int i = 0; i < qubits; i++) {
    phiPlus.measure(i, i); //measure all the qubits
  }

  //drop everything into the same array as real hardware data
  List<String> measurements = new ArrayList();
  for (int i = 0; i < shots; i++) {
    Map<String,Integer> counts = (Map)simulator.simulate(phiPlus, 1, "counts");
    String firstKey = counts.keySet().iterator().next();
    measurements.add(firstKey);
  }
  finalQuantumArray = measurements.toArray(new String[0]);
  //println(finalQuantumArray);
}
```

Measure the circuit

All quantum circuits must be measured to be useful. The measurement operation should be the same for every circuit.

```
void createBellStates() {
  Simulator simulator = new Simulator(0.1); // the parameter adds error. 0 is no error. 1 is pure noise. 0.1 seems to be similar to real hardware
  int qubits = 2;

  QuantumCircuit phiPlus = new QuantumCircuit(qubits, qubits); //the first parameter is number of quantum registers (qubits). the second is classical

  // ---------------------------------------------------
  //add circuit here
  phiPlus.h(0);
  phiPlus.cx(0, 1);


  // ---------------------------------------------------

  //measure entire circuit:
  for (int i = 0; i < qubits; i++) {
    phiPlus.measure(i, i); //measure all the qubits
  }

  //drop everything into the same array as real hardware data
  List<String> measurements = new ArrayList();
  for (int i = 0; i < shots; i++) {
    Map<String,Integer> counts = (Map)simulator.simulate(phiPlus, 1, "counts");
    String firstKey = counts.keySet().iterator().next();
    measurements.add(firstKey);
  }
  finalQuantumArray = measurements.toArray(new String[0]);
  //println(finalQuantumArray);
}
```

Simulate the circuit

The following code is used to run the circuit using MicroQiskit.

```
void createBellStates() {
  Simulator simulator = new Simulator(0.1); // the parameter adds error. 0 is no error. 1 is pure noise. 0.1 seems to be similar to real hardware
  int qubits = 2;

  QuantumCircuit phiPlus = new QuantumCircuit(qubits, qubits); //the first parameter is number of quantum registers (qbits). the second is classica

  // ---------------------------------------------------
  //add circuit here
  phiPlus.h(0);
  phiPlus.cx(0, 1);


  // ---------------------------------------------------

  //measure entire circuit:
  for (int i = 0; i < qubits; i++) {
    phiPlus.measure(i, i); //measure all the qubits
  }

  //drop everything into the same array as real hardware data
  List<String> measurements = new ArrayList();
  for (int i = 0; i < shots; i++) {
    Map<String,Integer> counts = (Map)simulator.simulate(phiPlus, 1, "counts");
    String firstKey = counts.keySet().iterator().next();
    measurements.add(firstKey);
  }
  finalQuantumArray = measurements.toArray(new String[0]);
  //println(finalQuantumArray);
}
```

Store circuit data in an array

all the of the data from the quantum circuit is finally stored in `finalQuantumArray`, which we will use throughout the rest of the tutorial

```
void createMeasings() {
  Simulator simulator = new Simulator(0.1); // the parameter adds error. 0 is no error. 1 is pure noise. 0.1 seems to be similar to real hardware
  int qubits = 2;

  QuantumCircuit phiPlus = new QuantumCircuit(qubits, qubits); //the first parameter is number of quantum registers (qbits). the second is classica

  // ---------------------------------------------------
  //add circuit here
  phiPlus.h(0);
  phiPlus.cx(0, 1);


  // ---------------------------------------------------

  //measure entire circuit:
  for (int i = 0; i < qubits; i++) {
    phiPlus.measure(i, i); //measure all the qubits
  }

  //drop everything into the same array as real hardware data
  List<String> measurements = new ArrayList();
  for (int i = 0; i < shots; i++) {
    Map<String,Integer> counts = (Map)simulator.simulate(phiPlus, 1, "counts");
    String firstKey = counts.keySet().iterator().next();
    measurements.add(firstKey);
  }
  finalQuantumArray = measurements.toArray(new String[0]);
  //println(finalQuantumArray);
}
```

Loop through data

The entire content of the function is contained in a for loop the length of the quantum data array.

We will be going through every entry in that array drawing a line to the canvas based on each entry.


```
void drawLines() {
  for (int i = 0; i < finalQuantumArray.length; i = i+1) {

    //store 0 and 1 as characters
    int binary0='0';
    char b0 = (char)binary0;
    int binary1='1';
    char b1 = (char)binary1;

    // get the first qubit character
    char qb0 = finalQuantumArray[i].charAt(0);
    // get the second qubit character
    char qb1 = finalQuantumArray[i].charAt(1);



    //decide which canvas to draw the line on
    if (qb0 == b0) {
      drawOnBottom = false;
    } else {
      drawOnBottom = true;
    }

    //draw either a horizontal or vertical line
    if (qb1 == b0) {
      drawHorizontalLine(i, drawOnBottom);
    } else {
      drawVerticalLine(i, drawOnBottom);
    }
  }
}
```

For each shot, store the first and second values separately

Before we can do anything with the data, we need to store each value independently so that we can take actions on them independently.

```
void drawLines() {
  for (int i = 0; i < finalQuantumArray.length; i = i+1) {

    //store 0 and 1 as characters
    int binary0='0';
    char b0 = (char)binary0;
    int binary1='1';
    char b1 = (char)binary1;

    // get the first qubit character
    char qb0 = finalQuantumArray[i].charAt(0);
    // get the second qubit character
    char qb1 = finalQuantumArray[i].charAt(1);


    //decide which canvas to draw the line on
    if (qb0 == b0) {
      drawOnBottom = false;
    } else {
      drawOnBottom = true;
    }

    //draw either a horizontal or vertical line
    if (qb1 == b0) {
      drawHorizontalLine(i, drawOnBottom);
    } else {
      drawVerticalLine(i, drawOnBottom);
    }
  }
}
```

Determine the location of line

If the left qubit's value is measured as 0, draw it on the top half of the composition. If the left qubit's value is measured as 1, draw it on the bottom half of the composition.

For now, this value is just stored in a boolean called `drawOnBottom`, which can either be true or false

```
void drawLines() {
  for (int i = 0; i < finalQuantumArray.length; i = i+1) {

    //store 0 and 1 as characters
    int binary0='0';
    char b0 = (char)binary0;
    int binary1='1';
    char b1 = (char)binary1;

    // get the first qubit character
    char qb0 = finalQuantumArray[i].charAt(0);
    // get the second qubit character
    char qb1 = finalQuantumArray[i].charAt(1);


    //decide which canvas to draw the line on
    if (qb0 == b0) {
      drawOnBottom = false;
    } else {
      drawOnBottom = true;
    }

    //draw either a horizontal or vertical line
    if (qb1 == b0) {
      drawHorizontalLine(i, drawOnBottom);
    } else {
      drawVerticalLine(i, drawOnBottom);
    }
  }
}
```

Determine the direction of line

If the right qubit's value is measured as 0, draw a horizontal line. If the right qubit's value is measured as 1, draw a horizontal line.

Instead of storing this value in a boolean like the prior step, we will go ahead and call a function to draw the line, which we will create in the next step.

```
void drawLines() {
  for (int i = 0; i < finalQuantumArray.length; i = i+1) {

    //store 0 and 1 as characters
    int binary0='0';
    char b0 = (char)binary0;
    int binary1='1';
    char b1 = (char)binary1;

    // get the first qubit character
    char qb0 = finalQuantumArray[i].charAt(0);
    // get the second qubit character
    char qb1 = finalQuantumArray[i].charAt(1);



    //decide which canvas to draw the line on
    if (qb0 == b0) {
      drawOnBottom = false;
    } else {
      drawOnBottom = true;
    }

    //draw either a horizontal or vertical line
    if (qb1 == b0) {
      drawHorizontalLine(i, drawOnBottom);
    } else {
      drawVerticalLine(i, drawOnBottom);
    }
  }
}
```

Draw a horizontal line

Drawing a line in processing is a matter of specifying a starting location with x and y coordinates, and an ending location with x and y coordinates. Once provided, Processing will draw the line between the coordinates.

```
void drawHorizontalLine(int i, boolean drawBottom) {

  double ycoordinate = Math.floor(i/(columns));
  double xcoordinate = i - (ycoordinate*columns);

  startingX = (int) xcoordinate * scale + (scale/5) + padding; //start 20 percent in so each line is mi
  startingY = (int) ycoordinate * scale + (scale/2); //start 50% in

  startingY += scale/4 * 10 + padding; //adding a gap from the top

  if (drawBottom) {
    startingY += rows * scale / 2; //start halfway down
    startingY += scale/2 * 10; //adding a gap between the two
  }

  line(startingX,startingY, startingX + scale - (scale/(5/2)), startingY); //make the length 20 percent
}
```

Draw a horizontal line

Each line should be drawn next to the previous line, unless it is the last item in the row, in which case it should be rendered on the next line down.

The global variable `columns` at the top of the page file is used here.

```
void drawHorizontalLine(int i, boolean drawBottom) {

  double ycoordinate = Math.floor(i/(columns));
  double xcoordinate = i - (ycoordinate*columns);

  startingX = (int) xcoordinate * scale + (scale/5) + padding; //start 20 percent in so each line is mi
  startingY = (int) ycoordinate * scale + (scale/2); //start 50% in

  startingY += scale/4 * 10 + padding; //adding a gap from the top

  if (drawBottom) {
    startingY += rows * scale / 2; //start halfway down
    startingY += scale/2 * 10; //adding a gap between the two
  }

  line(startingX,startingY, startingX + scale - (scale/(5/2)), startingY); //make the length 20 percent
}
```

Draw a horizontal line

The function then looks at the drawBottom parameter to determine if the line should be drawn halfway down the page or not.


```
void drawHorizontalLine(int i, boolean drawBottom) {

  double ycoordinate = Math.floor(i/(columns));
  double xcoordinate = i - (ycoordinate*columns);

  startingX = (int) xcoordinate * scale + (scale/5) + padding; //start 20 percent in so each line is mi
  startingY = (int) ycoordinate * scale + (scale/2); //start 50% in

  startingY += scale/4 * 10 + padding; //adding a gap from the top

  if (drawBottom) {
    startingY += rows * scale / 2; //start halfway down
    startingY += scale/2 * 10; //adding a gap between the two
  }

  line(startingX,startingY, startingX + scale - (scale/(5/2)), startingY); //make the length 20 percent
}
```

Draw a horizontal line

Finally, once all the positioning is correct, the function draws the horizontal line.

Note: There are lots of small mathematical tweaks to the position of the line in this function. These aren't a part of the core algorithm itself, but part of the artistry of determining the balance, spacing, and overall affect of the composition.

```
void drawHorizontalLine(int i, boolean drawBottom) {

  double ycoordinate = Math.floor(i/(columns));
  double xcoordinate = i - (ycoordinate*columns);

  startingX = (int) xcoordinate * scale + (scale/5) + padding; //start 20 percent in so each line is mi
  startingY = (int) ycoordinate * scale + (scale/2); //start 50% in

  startingY += scale/4 * 10 + padding; //adding a gap from the top

  if (drawBottom) {
    startingY += rows * scale / 2; //start halfway down
    startingY += scale/2 * 10; //adding a gap between the two
  }

  line(startingX,startingY, startingX + scale - (scale/(5/2)), startingY); //make the length 20 percent
}
```

Draw a vertical line

The vertical line function is fundamentally the same as the horizontal line, except that it draws the line vertically instead of horizontally.

The only changes necessary are flipping the values on the starting and ending coordinates in comparison to the horizontal line function.

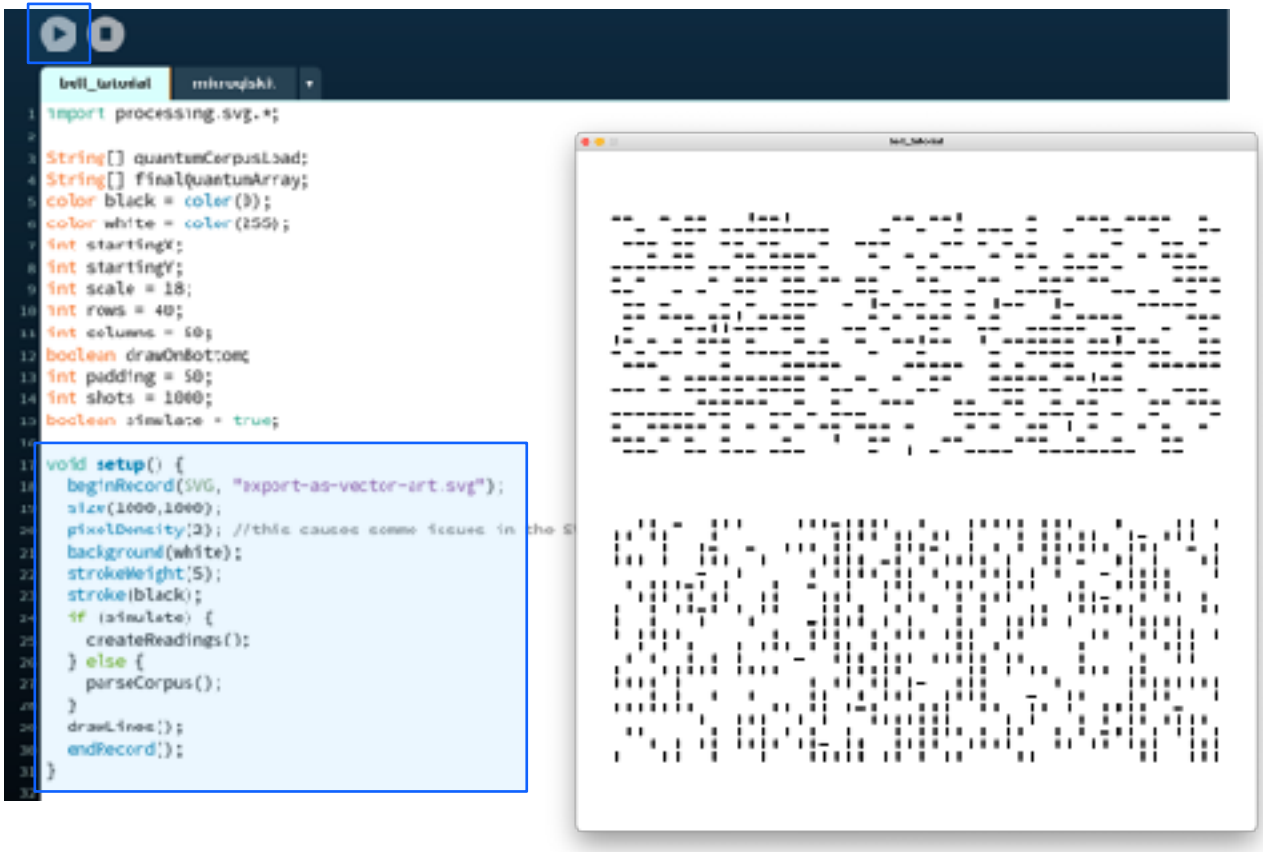
```
void drawVerticalLine(int i, boolean drawBottom) {
  double ycoordinate = Math.floor(i/(columns));
  double xcoordinate = i - (ycoordinate*columns);

  startingX = (int) xcoordinate * scale + (scale/2) + padding; //make the length 20 percent short, whi
  startingY = (int) ycoordinate * scale + (scale/5);

  startingY += scale/4 * 10 + padding; //adding a gap from the top

  if (drawBottom) {
    startingY += rows * scale / 2; //start halfway down
    startingY += scale/2 * 10; //adding a gap between the two
  }

  line(startingX,startingY, startingX, startingY + scale - (scale/(5/2)));//make the length 20 percent
}
```

Preview the composition

With all the code in place, we can now preview the composition by clicking the “play” button in the top right corner of the interface.

Everything in the `setup()` function get’s run when the play button is pressed.

A separate window will open with the composition



Switch to real quantum data

Open the Jupyter Notebook titled “Bell tutorial.ipynb”. This can be worked on locally or via Quantum Lab via https://lab.quantum-computing.ibm.com

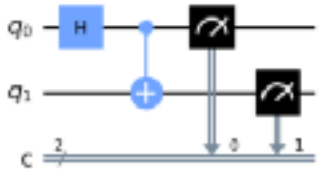
If working from Quantum Lab, all the necessary imports should be provided automatically. Otherwise, add them.

[10]
```
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ, execute
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

```
ibmqfactory.load_account:WARNING:2021-08-24 11:02:27,714: Credentials are already in use. The existing account in the session
will be replaced.
```

[11]
```
circuit = QuantumCircuit(2, 2)

circuit.h(0)
circuit.cx(0, 1)
circuit.measure([0,1], [0,1])

circuit.draw()
```

[11]



Switch to real quantum data

Recreate the same circuit that was originally created in Processing with MicroQiskit.

As shown here, the circuit uses 2 qubits, applies a hadamard gate to the first, and a cnot connecting the first to the second qubit.

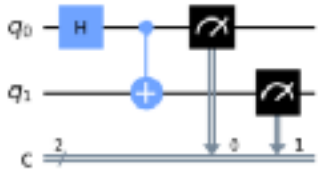
The qubits are then measured and the circuit is drawn. (Drawing the circuit is optional)

[10]:
```
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ, execute
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

```
ibmqfactory.load_account:WARNING:2021-08-24 11:02:27,714: Credentials are already in use. The existing account in the session
will be replaced.
```

[11]:
```
circuit = QuantumCircuit(2, 2)

circuit.h(0)
circuit.cx(0, 1)
circuit.measure([0,1], [0,1])

circuit.draw()
```

[11]:



Switch to real quantum data

Select a quantum system to run the circuit on. You should have several to pick from, which are listed under the “systems” tab under “quantum services”

https://quantum-computing.ibm.com/services?services=systems

IBM **Quantum**

```
[13]
#pick a device to run on
backend = provider.get_backend('ibmq_bogota')
# backend = Aer.get_backend('qasm_simulator')

# Execute the circuit on the backend
job = execute(circuit, backend, shots=1000, memory=True)

# Grab results from the job
result = job.result()

#get individual shots
memory = result.get_memory()

# need an array to grab all the results data
outputArray = []

#convert results to int and drop into array
for x in range(0, 1000):
    outputArray.append(memory[x])

print(outputArray)
```

```
[ ]
%%capture cap --no-stdout
print(outputArray)
```

```
[ ]
with open('corpus.txt', 'w') as f:
    f.write(cap.stdout)
```

Execute the job and see individual shots

We will be using the individual shots from the quantum system. By default, we see an aggregate of the results. To see the individual shots, we will be accessing the memory from the results.

IBM **Quantum**

```
[13]: #pick a device to run on
backend = provider.get_backend('ibmq_bogota')
# backend = Aer.get_backend('qasm_simulator')

# Execute the circuit on the backend
job = execute(circuit, backend, shots=1000, memory=True)

# Grab results from the job
result = job.result()

#get individual shots
memory = result.get_memory()

# need an array to grab all the results data
outputArray = []

#convert results to int and drop into array
for x in range(0, 1000):
    outputArray.append(memory[x])

print(outputArray)
```

```
[ ]: %%capture cap --no-stdout
print(outputArray)
```

```
[ ]: with open('corpus.txt', 'w') as f:
    f.write(cap.stdout)
```

Store the data

We will next store all of the data from the results in a simple array that can be easily imported into Processing.

```
[13]: #pick a device to run on
backend = provider.get_backend('ibmq_bogota')
# backend = Aer.get_backend('qasm_simulator')

# Execute the circuit on the backend
job = execute(circuit, backend, shots=1000, memory=True)

# Grab results from the job
result = job.result()

#get individual shots
memory = result.get_memory()

# need an array to grab all the results data
outputArray = []

#convert results to int and drop into array
for x in range(0, 1000):
    outputArray.append(memory[x])

print(outputArray)
```

```
[ ]: %%capture cap --no-stdout
print(outputArray)
```

```
[ ]: with open('corpus.txt', 'w') as f:
    f.write(cap.stdout)
```

Export the data

Run these two code cells to export the data as a text file. If working in Quantum Lab, the .txt file will listed to the left among the other files.

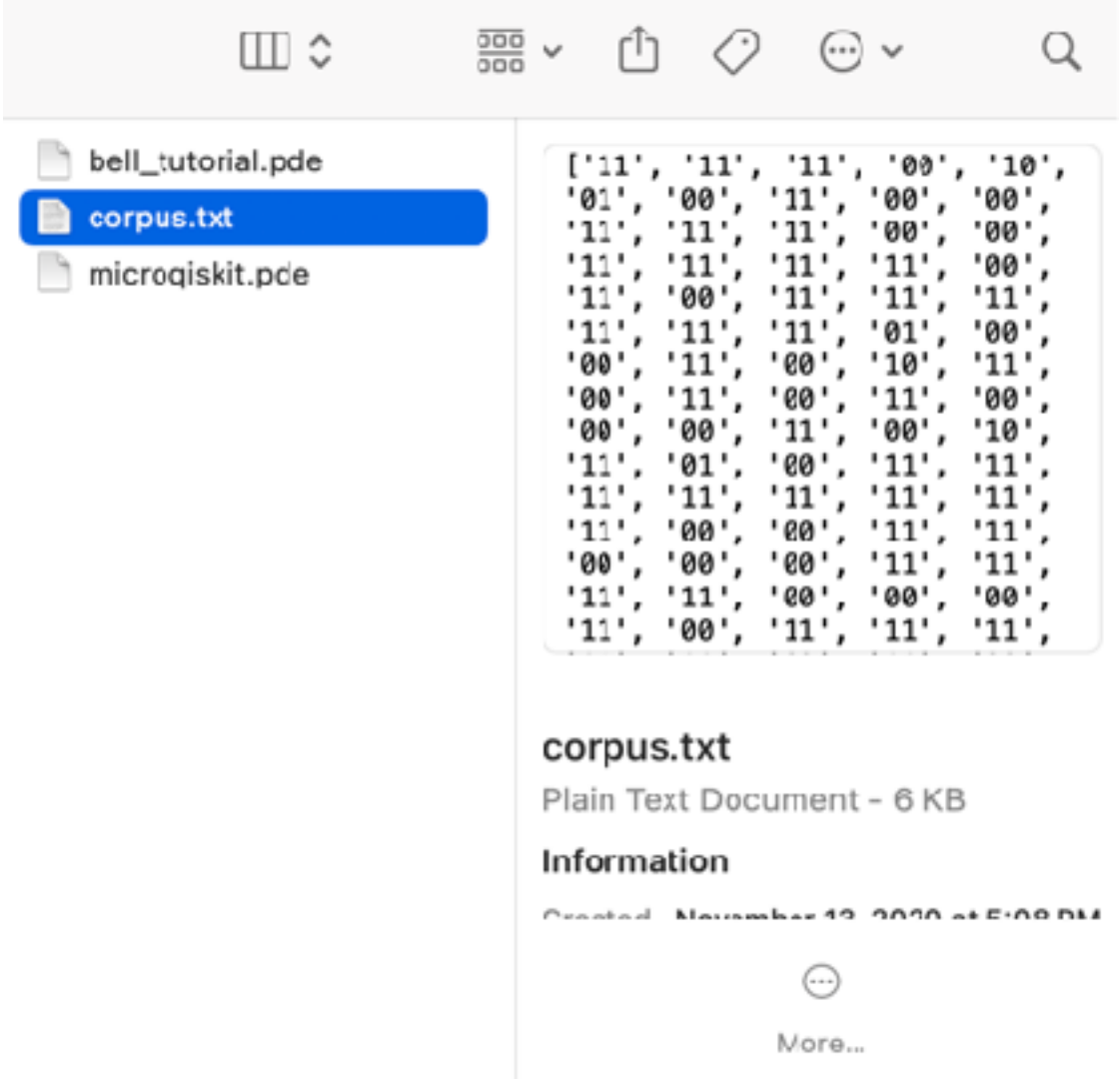
```
[13]: #pick a device to run on
backend = provider.get_backend('ibmq_bogota')
# backend = Aer.get_backend('qasm_simulator')

# Execute the circuit on the backend
job = execute(circuit, backend, shots=1000, memory=True)

# Grab results from the job
result = job.result()

#get individual shots
memory = result.get_memory()

# need an array to grab all the results data
outputArray = []

#convert results to int and drop into array
for x in range(0, 1000):
    outputArray.append(memory[x])

print(outputArray)
```

```
[ ]: %%capture cap --no-stdout
print(outputArray)
```

```
[ ]: with open('corpus.txt', 'w') as f:
    f.write(cap.stdout)
```

Import data into Processing

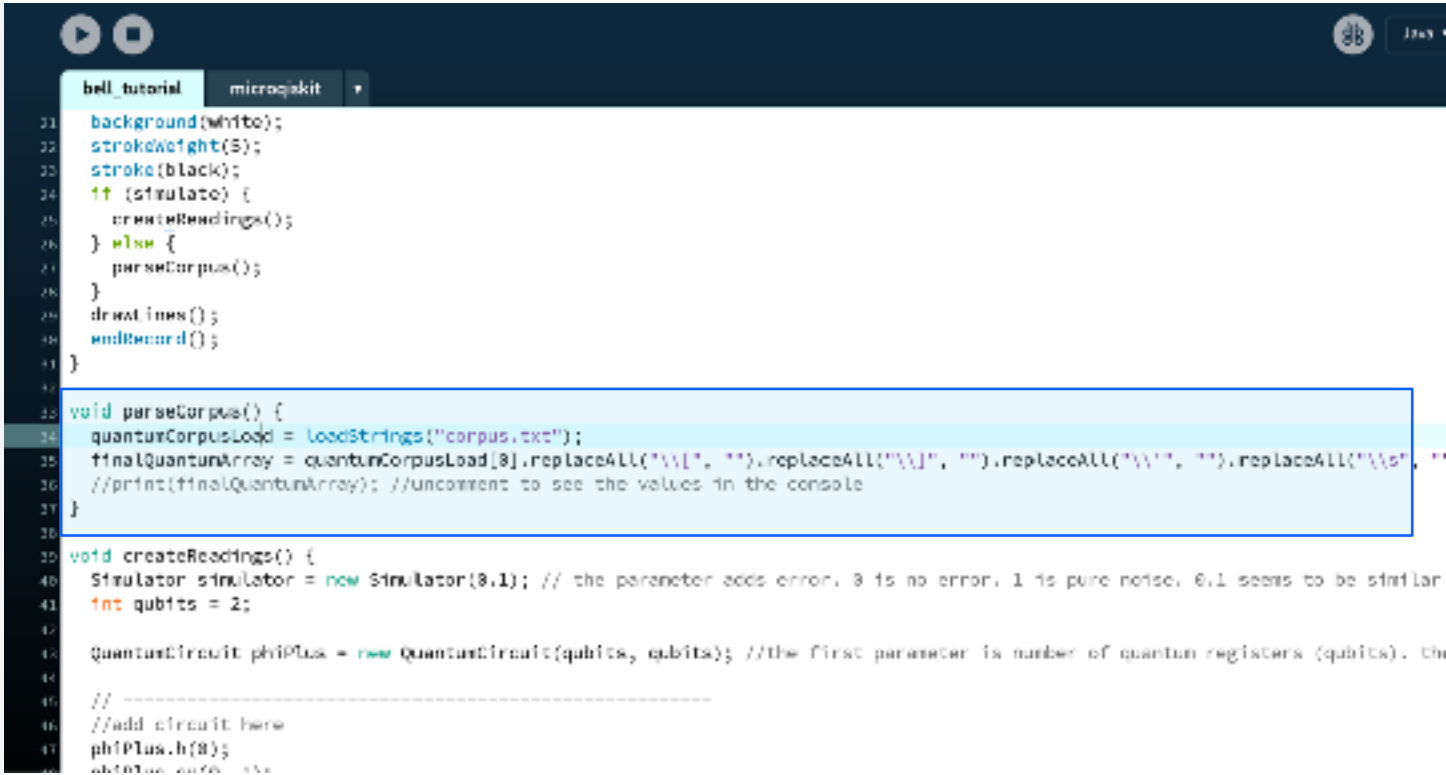
Add the “corpus.txt” file to the Processing project root file. It should be next to the “bell_tutorial.pde” file and the “microqiskit.pde” file.



Reference the new text file in processing

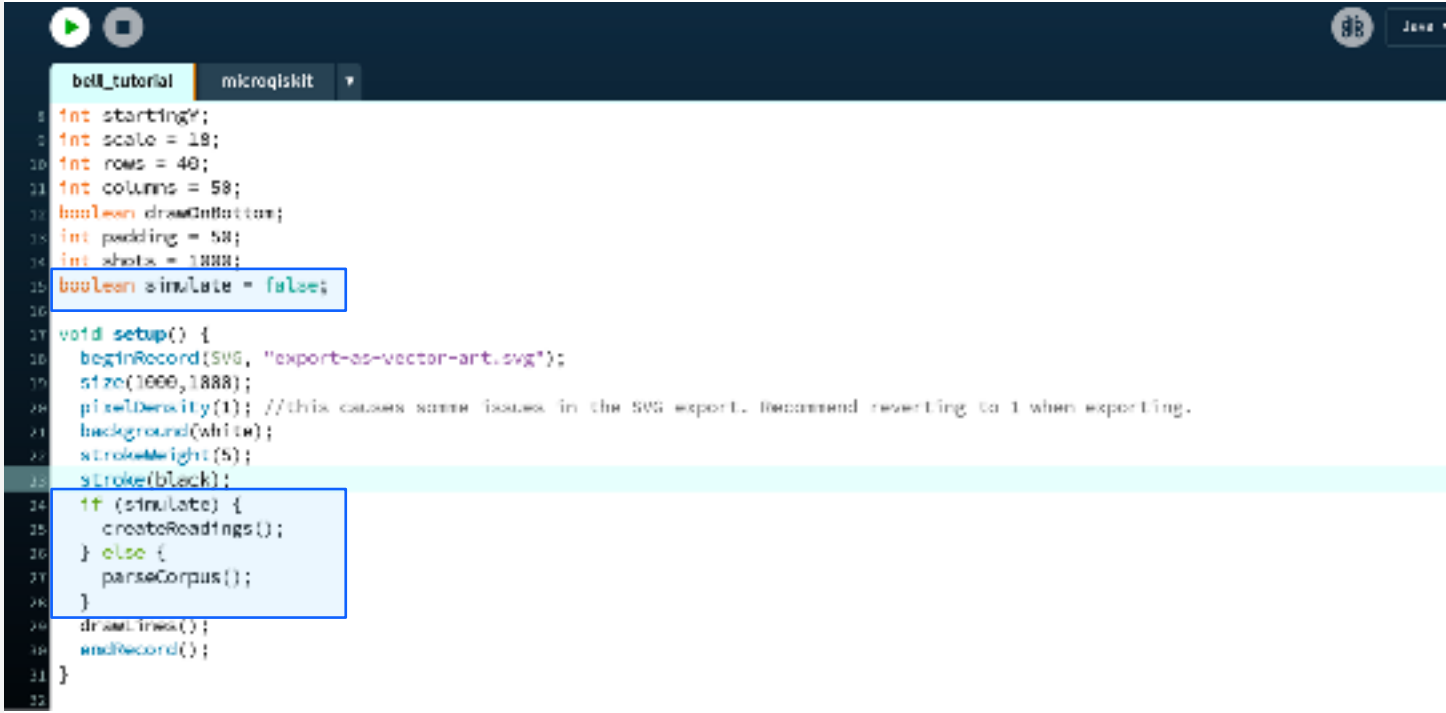
The text file is loaded into the Processing sketch via the `parseCorpus()` function.

The text file is loaded with `quantumCorpusLoad` and parsed into it's final location `finalQuantumArray`



Switch from simulator to real data

Switch the boolean called `simulate` from `true` to `false`. This will tell our sketch to create the composition with the real quantum data instead of the simulated data.

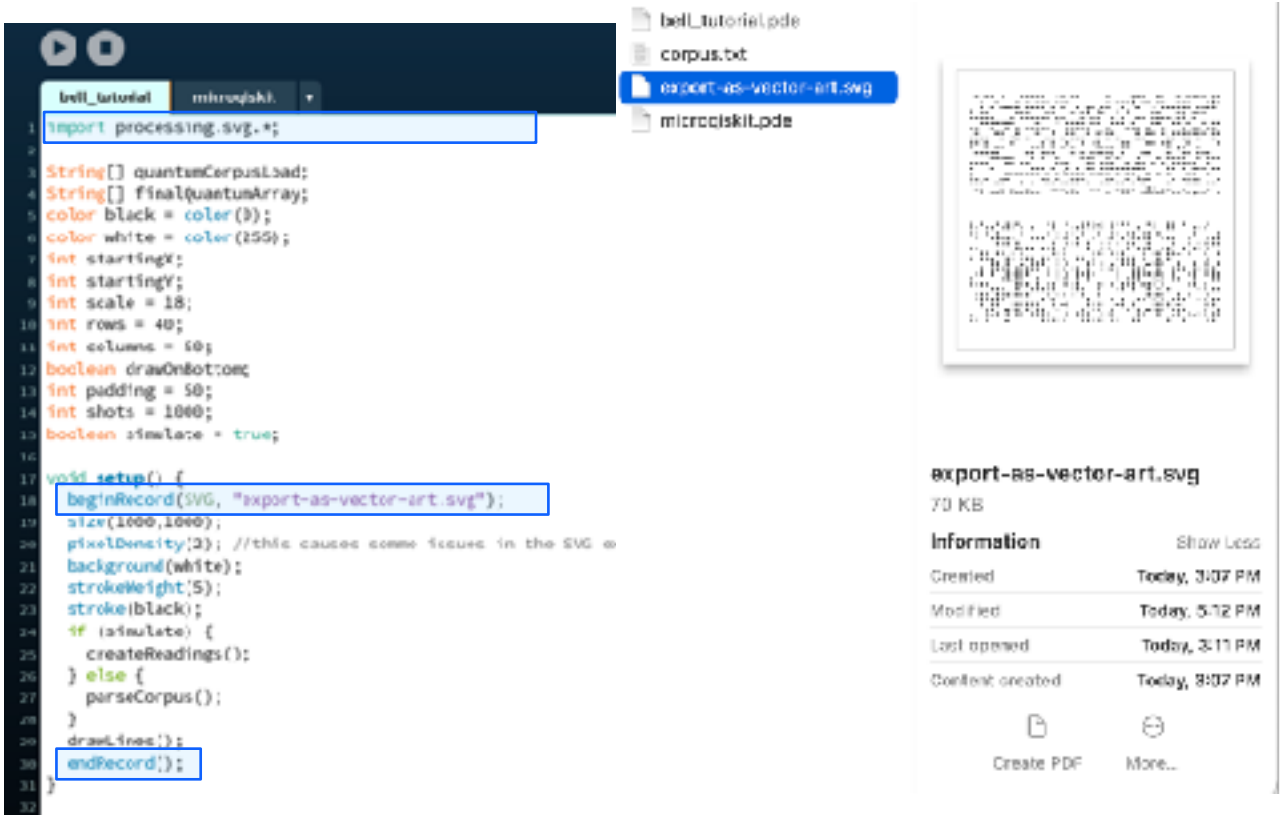


Export the composition

In most cases we will want to be able to work with the final composition for the purposes of printing, sharing, etc.

This is done in processing by importing the svg exporting capability and calling it in the `setup()` function.

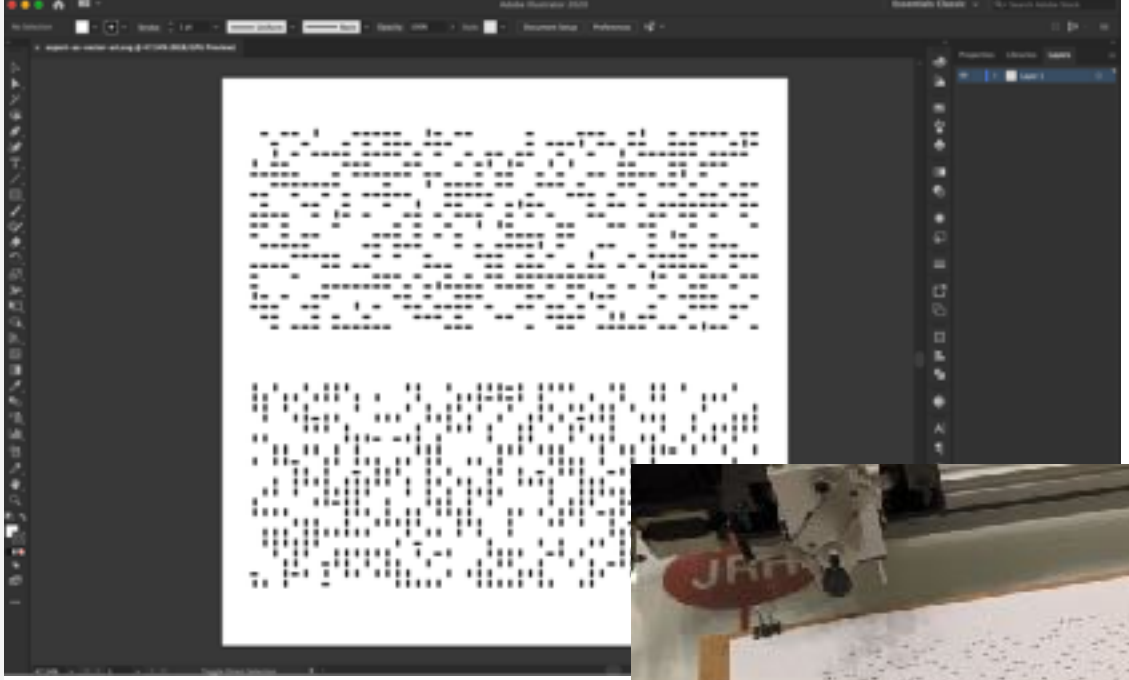
An SVG will be saved in the project root folder with the name given to it in the `beginRecord()` function.

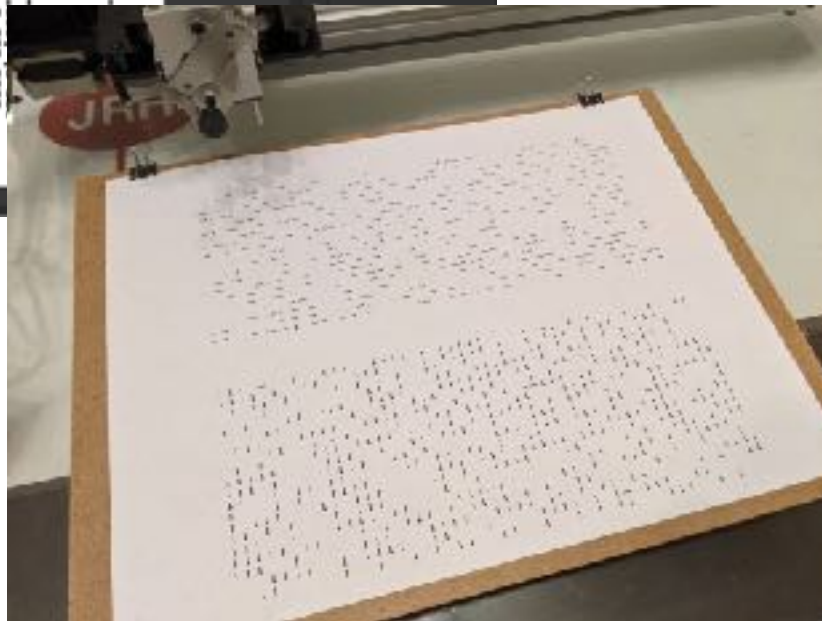


Open in design software

The exported SVG can then be open and manipulated in any design software that can be edit vector graphics, such as Adobe Illustrator or Inkscape.

I recommend Adobe Illustrator for preparing the composition and I recommend Inkscape for pen plotting. Ultimately, it is up to you what you want to do with you generative quantum art composition.





IBM **Quantum**

Conclusion

Why make art with a quantum computer?

Any new technology has the capacity for creative expression and has unique affordances that enable new types of art.

We are here today to explore and discover how those affordances can be expressed creatively.

What does “quantum” look like?

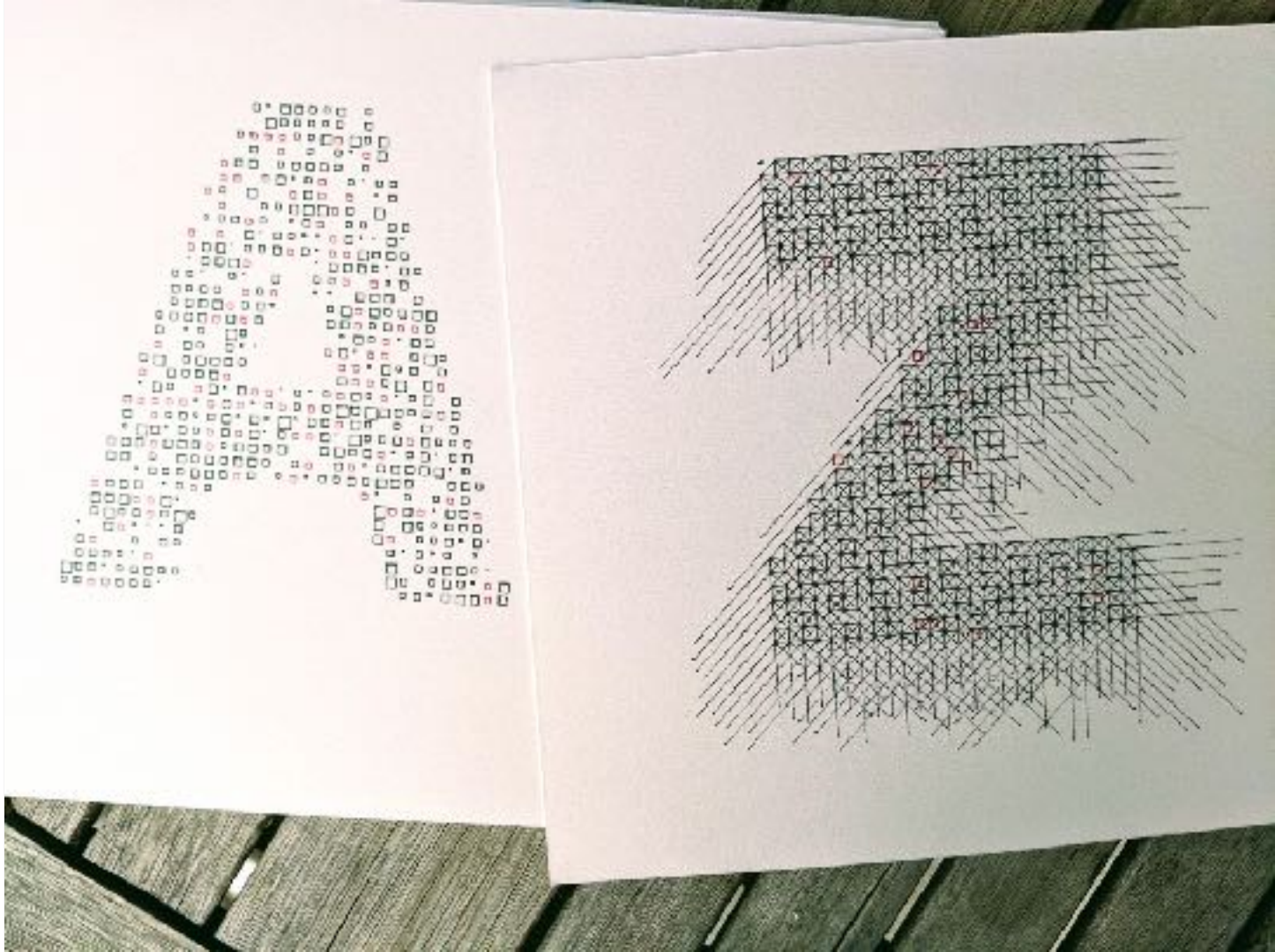
Show and tell

Rothko inspired composition #1
A composition using data from the bell state circuit to create a piece inspired by Rothko's Color Field works.

Rendered image of composition on campus

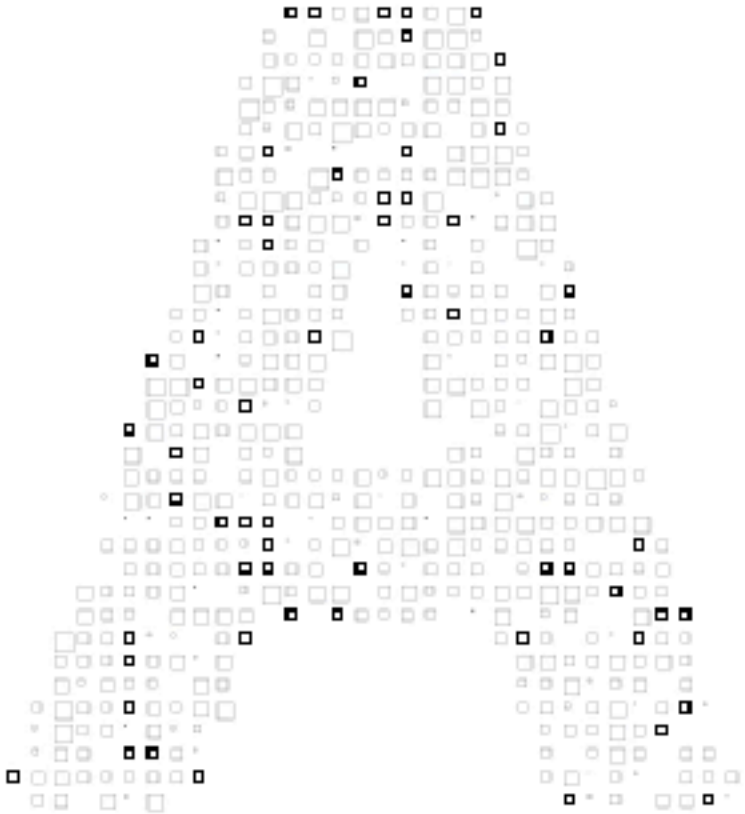
Show and tell

Quantum Alphabet
A typographic series using the Bernstein Vazirani Algorithm to store a letter's data as a quantum state, which is used to render the final composition.



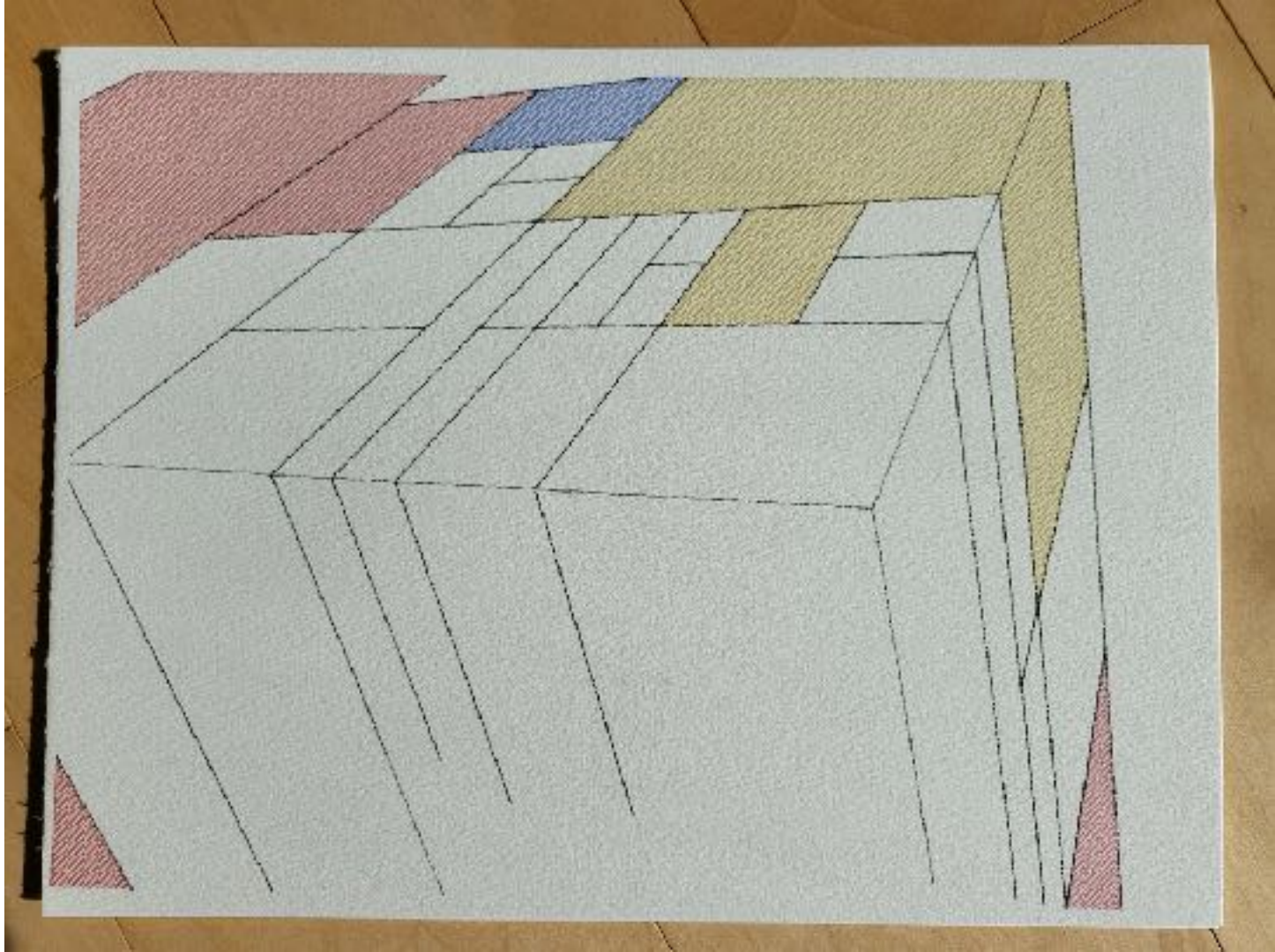
Show and tell

Quantum Alphabet
A typographic series using the Bernstein Vazirani Algorithm to store a letter’s data as a quantum state, which is used to render the final composition.



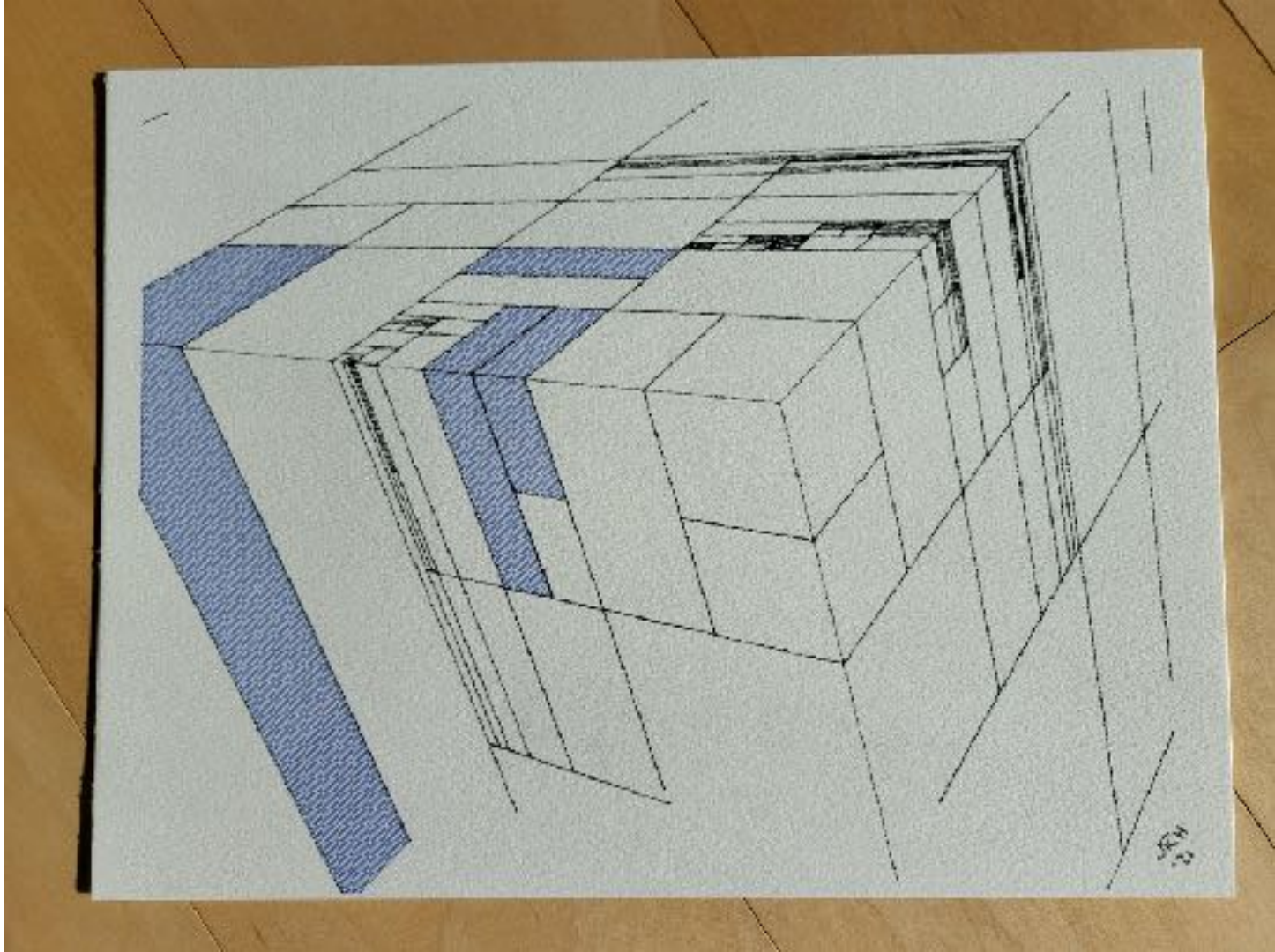
Show and tell

Mondrian’s Cube
A series inspired by Piet Mondrian’s style and philosophy towards abstraction that uses the quantum GHZ state.



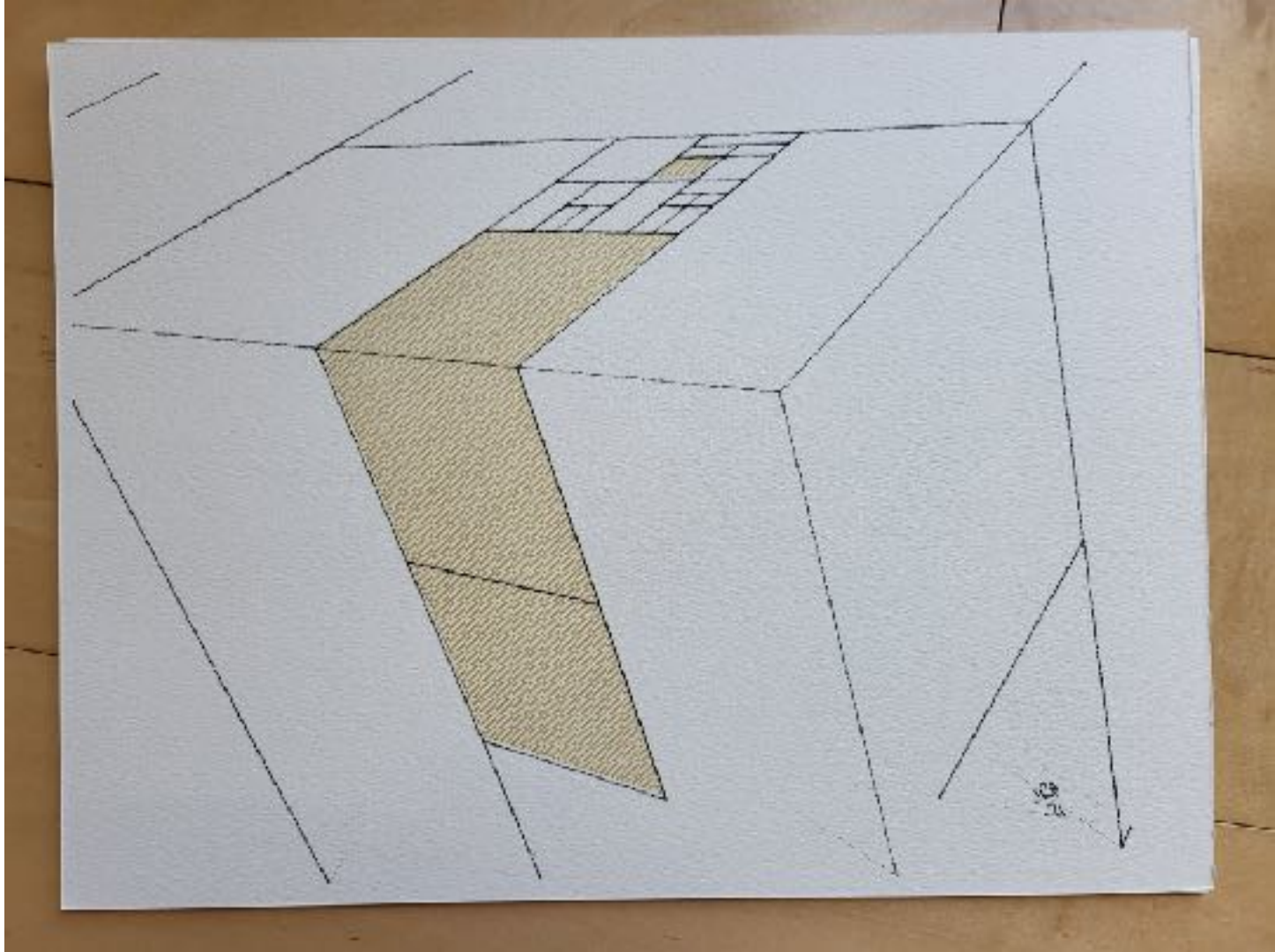
Show and tell

Mondrian’s Cube
A series inspired by Piet Mondrian’s style and philosophy towards abstraction that uses the quantum GHZ state.



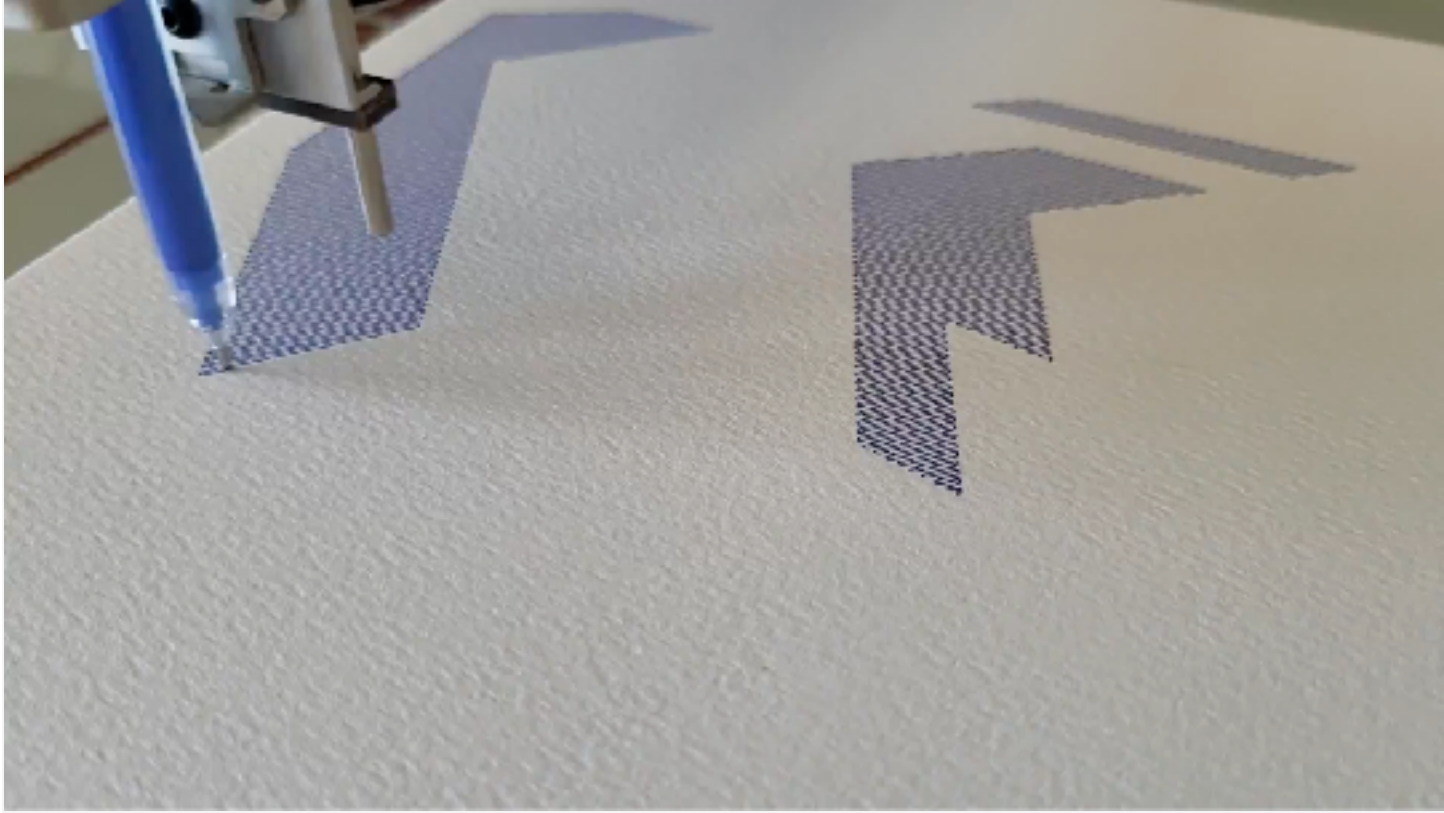
Show and tell

Mondrian’s Cube
A series inspired by Piet Mondrian’s style and philosophy towards abstraction that uses the quantum GHZ state.



Show and tell

Mondrian’s Cube
A series inspired by Piet Mondrian’s style and philosophy towards abstraction that uses the quantum GHZ state.



Your challenge this weekend

Create an artwork that uses quantum computing

The art should

1. Reference at least 1 quantum computing concept (such as superposition)
2. Have a meaningful connection between a concept in quantum computing and your artwork
3. Use a real quantum system (not a simulator)

Do your best, have fun, try this new technology, and be creative.

Properties of quantum information

1. Not all information attributes of a quantum information medium are distinguishable.
2. Undetectability of sharpness
3. Quantum information cannot be cloned
4. Pairs of observables not simultaneously preparable or measurable
5. Unpredictability of deterministic processes
6. Irreducible perturbation of one observable caused by measuring another
7. Consistency of consecutive measurements of a non-sharp observable
8. Quantization
9. Coherence and locally inaccessible information

Resources

Using Qiskit on the cloud
https://quantum-computing.ibm.com/

Install Qiskit locally (optional)
https://qiskit.org/

Install processing
https://processing.org/

Learn quantum computing
https://qiskit.org/textbook-beta/

IBM **Quantum**