

Algorithmic
Art
Workbook #8
3D glasses
and AI



Table of Contents

Table of Contents	2
MIT Licence	6
Workbook #8 Quick Content Summary	7
Module G Unit #1 3D glasses	9
Sketch G1.1 index.html file	10
Sketch G1.2 a box	11
Sketch G1.3 the anaglyph	13
Sketch G1.4 the scene	14
Sketch G1.5 a tunnel	16
Sketch G1.6 pushing it back	18
Sketch G1.7 parallel lines	20
Sketch G1.8 translation	22
Sketch G1.9 rotate	24
Sketch G1.10 moving depth	26
Sketch G1.11 lots of them	28
Sketch G1.12 a continuation	30
Sketch G1.13 adding 3D	32
Sketch G1.14 text	34
Sketch G1.15 the font	36
Sketch G1.16 WEBGL	38
Sketch G1.17 the anaglyph	40
Sketch G1.18 Dodecahedron	42
Sketch G1.19 face the pentagon	44
Sketch G1.20 resizing	46
Sketch G1.21 draw the dodecahedron	48
Sketch G1.22 rotate	51
Sketch G1.23 the anaglyph	54
Module G Unit #2 typography	58
Sketch G2.1 upload font	60
Sketch G2.2 display text	61
Sketch G2.3 breath of life	63
Sketch G2.4 dashed outline	65
Sketch G2.5 what's the point	67
Sketch G2.6 lines, letters and arrays	69
Sketch G2.7 getting the point	71
Sketch G2.8 alpha	73
Sketch G2.9 3D text	75
Sketch G2.10 geometry	76
Sketch G2.11 let there be lights	78

Sketch G2.12 steering behaviour	80
Sketch G2.13 pt points	81
Sketch G2.14 vehicles	83
Sketch G2.15 steer	85
Sketch G2.16 good behaviour	88
Module H Unit #1 introduction to AI	93
What is ml5.js?	94
Key elements of ml5.js	95
BodyPose	96
HandPose	97
FaceMesh	98
ImageClassifier	99
SoundClassifier	100
ml5 NeuralNetwork	101
Module H Unit #2 handPose pretrained AI	104
The index.html file	105
Sketch H2.1 our starting sketch	106
Sketch H2.2 video	107
Sketch H2.3 putting you on the canvas	108
Sketch H2.4 flipping the video	109
Sketch H2.5 the handPose model	110
Sketch H2.6 start detecting	111
Sketch H2.7 callback	112
Sketch H2.8 an array of hands	114
Sketch H2.9 the keypoints	115
Sketch H2.10 mirroring the yellow dots	117
Sketch H2.11 reference keypoints	119
Sketch H2.12 index and thumb	121
Sketch H2.13 circle size	123
Sketch H2.14 adding some coloured color	126
Sketch H2.15 checking the distance	128
Sketch H2.16 slimmed down sketch	130
Sketch H2.17 many hands make light work	131
Sketch H2.18 the vertex	133
Module H Unit #3 handPose emoji	136
Sketch H3.1 index.html	137
Sketch H3.2 starting sketch	138
Sketch H3.3 hand of hands	139
Sketch H3.4 emitters	141
Sketch H3.5 particles	143
Sketch H3.6 movement	145

Sketch H3.7 let's go running	148
Sketch H3.8 is it alive?	152
Sketch H3.9 tweak	157
Module H Unit #4 faceMesh	162
Sketch H4.1 index.html	163
Sketch H4.2 our starting sketch	164
Sketch H4.3 points of view	165
Sketch H4.4 step back	167
Sketch H4.5 drawing outer lips	169
Sketch H4.6 and the inner lips	172
Sketch H4.7 lip sync	175
Sketch H4.8 now the nose	177
Blowing Bubbles	179
Sketch H4.9 adding the emitter	180
Sketch H4.10 adding the particle	183
Sketch H4.11 now the bubbles	187
Sketch H4.12 negative gravity	192
Sketch H4.13 back to basics	197
Sketch H4.14 triangles	198
Sketch H4.15 showing the triangles	200
Sketch H4.16 adding a mask	202
Sketch H4.17 texture mapping	205
Sketch H4.18 mapping uv coordinates	207



MIT Licence

Copyright ©2026 Warren George

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Brief summary:

Permissions (what you can do)

Commercial use
Modification
Distribution
Private use

Limitations (what is not covered)

Liability
Warranty



Workbook #8 Quick Content Summary

This is where you can have some fun with a range of applications. The first unit is making 3D more 3D, and by that I mean with 3D glasses. You will need some for that unit; they are very inexpensive. You can get cardboard or plastic frame ones, but what you do need are the type that have red and blue lenses.

The second unit is about using a new feature available in version 2.x in p5.js, where you can do some interesting stuff with text, both 2D and 3D. For that, you will need to download a font that can be used in this fashion.

The final units are where you play with AI. Not generative AI such as with Large Language Models or Diffusion models, but with pretrained models that you can use in your browser, which pick up features of your hand or face.

Algorithmic Art

Module G

Unit #1

3D glasses





Module G Unit #1 3D glasses

For this unit, we will need a pair of 3D glasses. They are very cheap to buy online and usually come in packs of ten (if you need that many). They have a red lens (LEFT) and a blue lens (RIGHT); that is the sort you will need for this unit. The frames can be made of card, but I recommend getting the plastic frames as they are more robust and work well.

Figure 1





Sketch G1.1 index.html file

You may well have gone to see films like Avatar in 3D, where the images leapt out at you. We can do the same with p5.js and 3D shapes. You are not going to create an Avatar-quality scene, but you can at least have fun seeing the 3D shapes come to life. To create a 3D image in the sense of it jumping out of the screen, we can take a little shortcut by using a library that someone has very kindly created to do all the donkey work.

We will need to use a library for this unit called p5.anaglyph, and it needs to go in the index.html file. The script is as follows:

```
<script src="https://cdn.jsdelivr.net/gh/jdeboi/p5.anaglyph@main/dist/p5.anaglyph.min.js"></script>
```

You add it the usual way through the `index.html` file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.3.2/lib/p5.js"></script>
  <script src="https://cdn.jsdelivr.net/gh/jdeboi/p5.anaglyph@main/dist/p5.anaglyph.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```

Notes

Just add it in the usual way.



Sketch G1.2 a box

Starting sketch, to get things rolling, we create a simple, rotating, 3D box.

```
let angle = 0

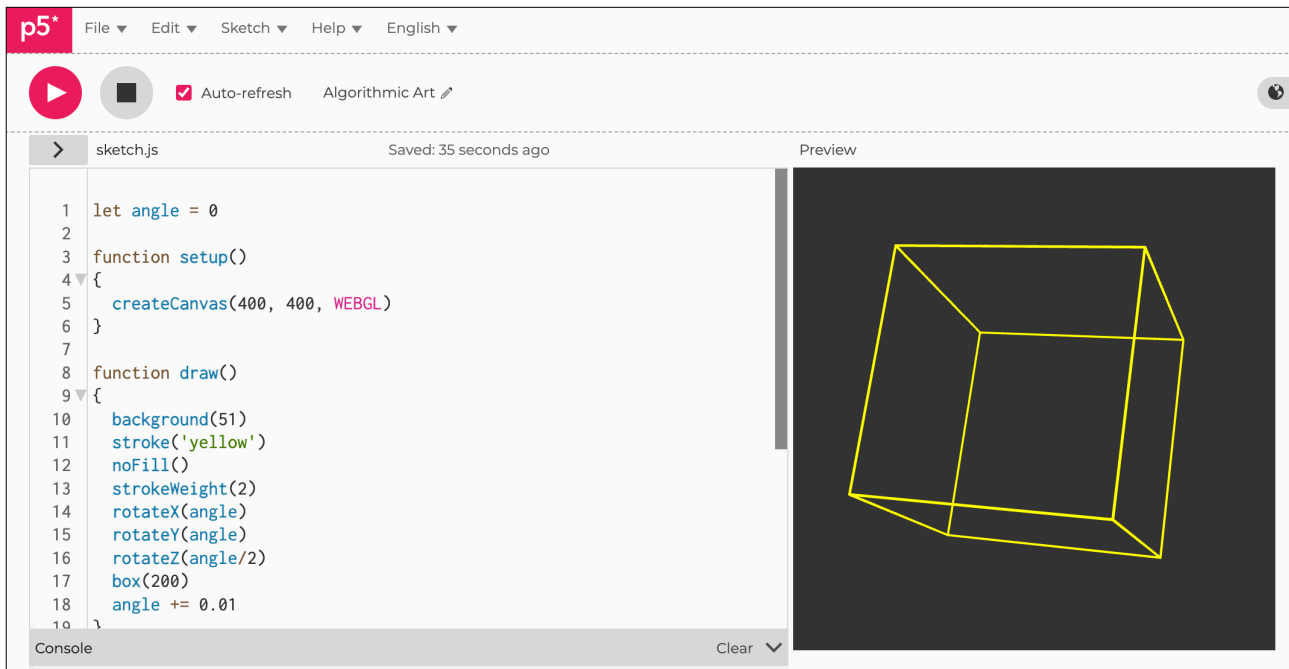
function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  background(51)
  stroke('yellow')
  noFill()
  strokeWeight(2)
  rotateX(angle)
  rotateY(angle)
  rotateZ(angle)
  box(200)
  angle += 0.01
}
```

Notes

We have a lovely rotating box. Not properly 3D enough yet.

Figure G1.2





Sketch G1.3 the anaglyph

Adding the Anaglyph library to the sketch.

```
let angle = 0
let anaglyph

function setup()
{
  createCanvas(400, 400, WEBGL)
  anaglyph = createAnaglyph(this)
}

function draw()
{
  background(51)
  stroke('yellow')
  noFill()
  strokeWeight(2)
  rotateX(angle)
  rotateY(angle)
  rotateZ(angle)
  box(200)
  angle += 0.01
}
```

Notes

Nothing changes yet.

Code Explanation

```
anaglyph = createAnaglyph(this)
```

Part of the library that allows us to use an object



Sketch G1.4 the scene

We need to create a separate function that will take in all the values we have currently. This will be what we call our scene. The scene will hold the 3D effect for when we use the 3D glasses.

We need to create another function to handle the new scene, which will inherit all the features of the anaglyph library. We cannot do this in draw alone. Here we need to identify the new scene as we draw the cube in its new guise. We can use any name (within reason), but I plumped for **graphic**.

! Notice we have transferred nearly all the code from **draw()** into this new function. Now it is time to put on your 3D glasses.

```
let angle = 0
let anaglyph

function setup()
{
  createCanvas(400, 400, WEBGL)
  anaglyph = createAnaglyph(this)
}

function draw()
{
  anaglyph.draw(scene)
  angle += 0.01
}

function scene(graphic)
{
  graphic.background(51)
  graphic.stroke('yellow')
  graphic.noFill()
  graphic.strokeWeight(2)
  graphic.rotateX(angle)
  graphic.rotateY(angle)
  graphic.rotateZ(angle)
  graphic.box(200)
}
```

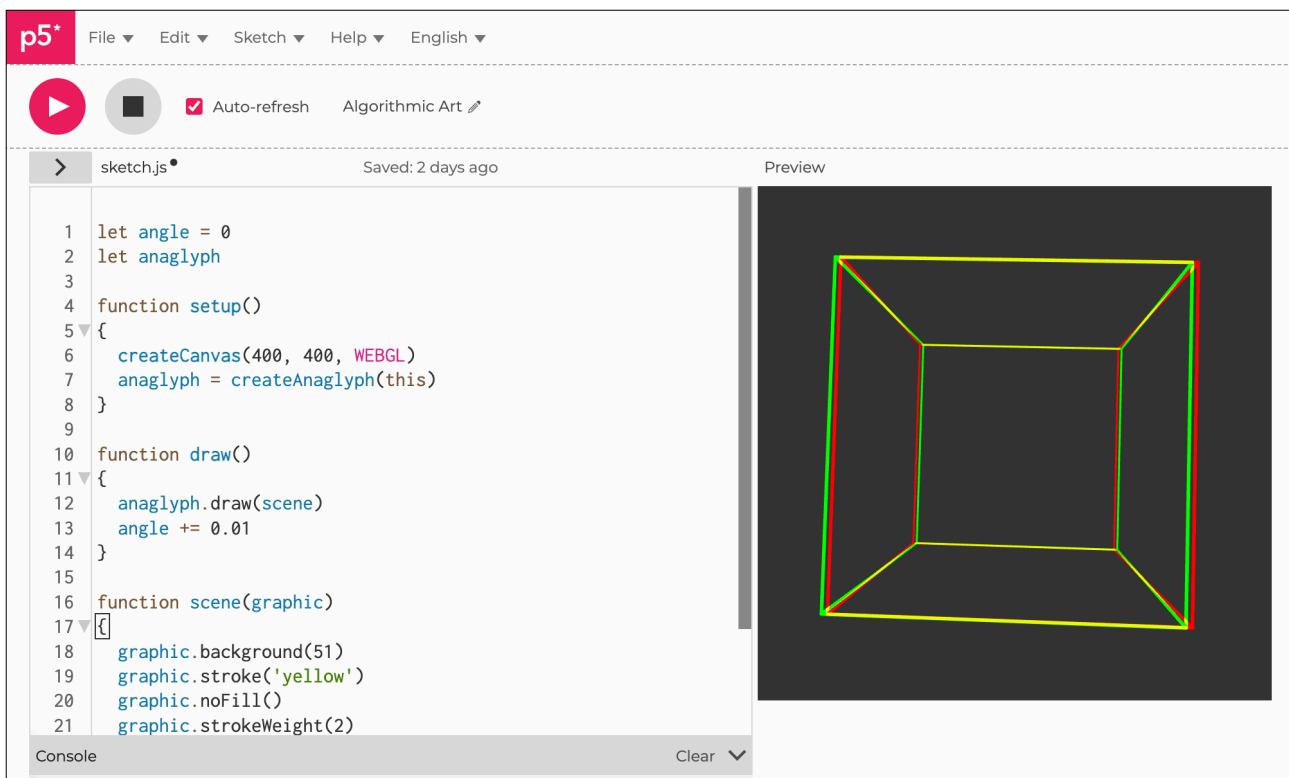
Notes

As you don your glasses, you will see the cube spring into life. You could include the angle being incremented in the `scene()` function, but it seems to spin twice as fast, so I have left it in the `draw()` function.

Challenges

1. Try other shapes.
2. How would you have more than one box?

Figure G1.4





Sketch G1.5 a tunnel

! This is our starting sketch

We are going to create a tunnel using the anaglyph effect. This will use 2D shapes to create a 3D effect, but all will be revealed. We want a square with the dimension as a variable, `boxWidth`. This allows us to play with different values, but this is key anyway as we will be moving forwards over time.

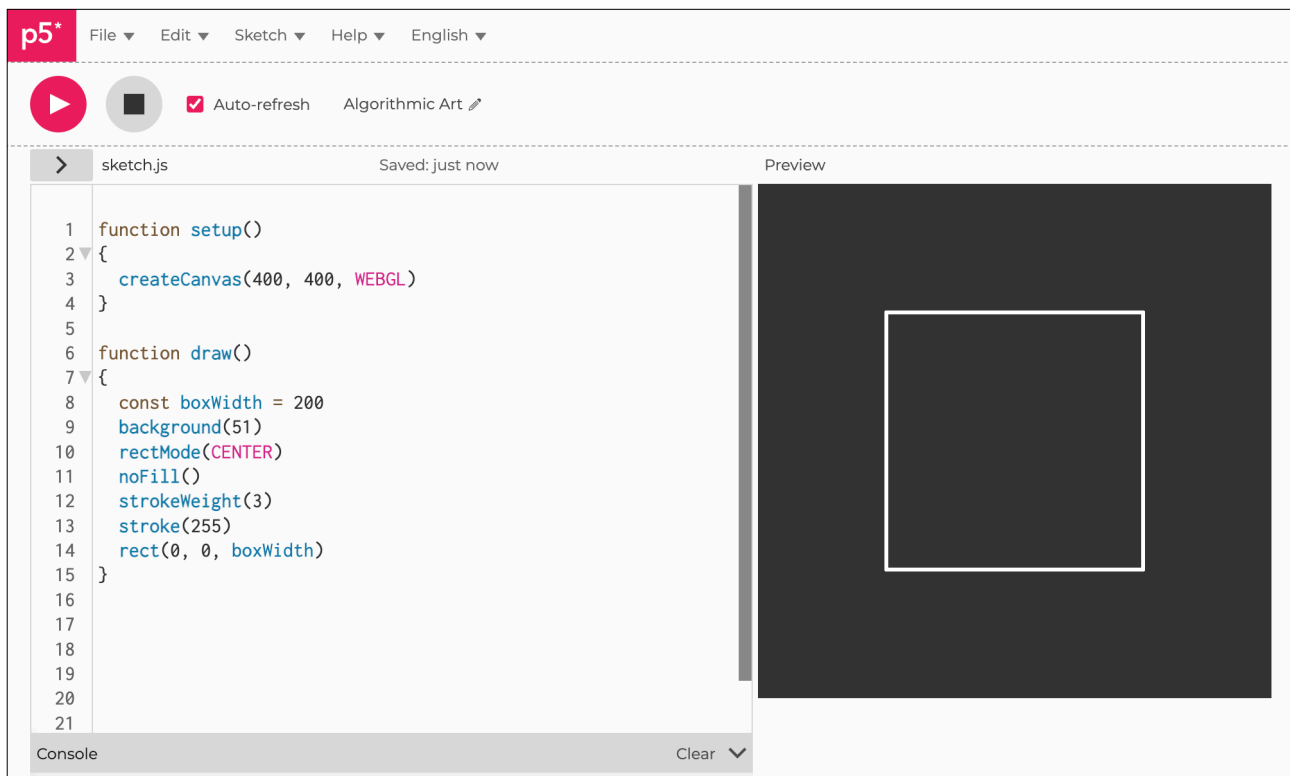
```
function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)
  rect(0, 0, boxWidth)
}
```

Notes

Using white for the lines just has a better effect later on.

Figure G1.5





Sketch G1.6 pushing it back

We want to push the square further back. We translate the square using all three elements: the **x**, **y**, and **z** components. This makes the square seem smaller.

```
function setup()
{
  createCanvas(400, 400, WEBGL)
}

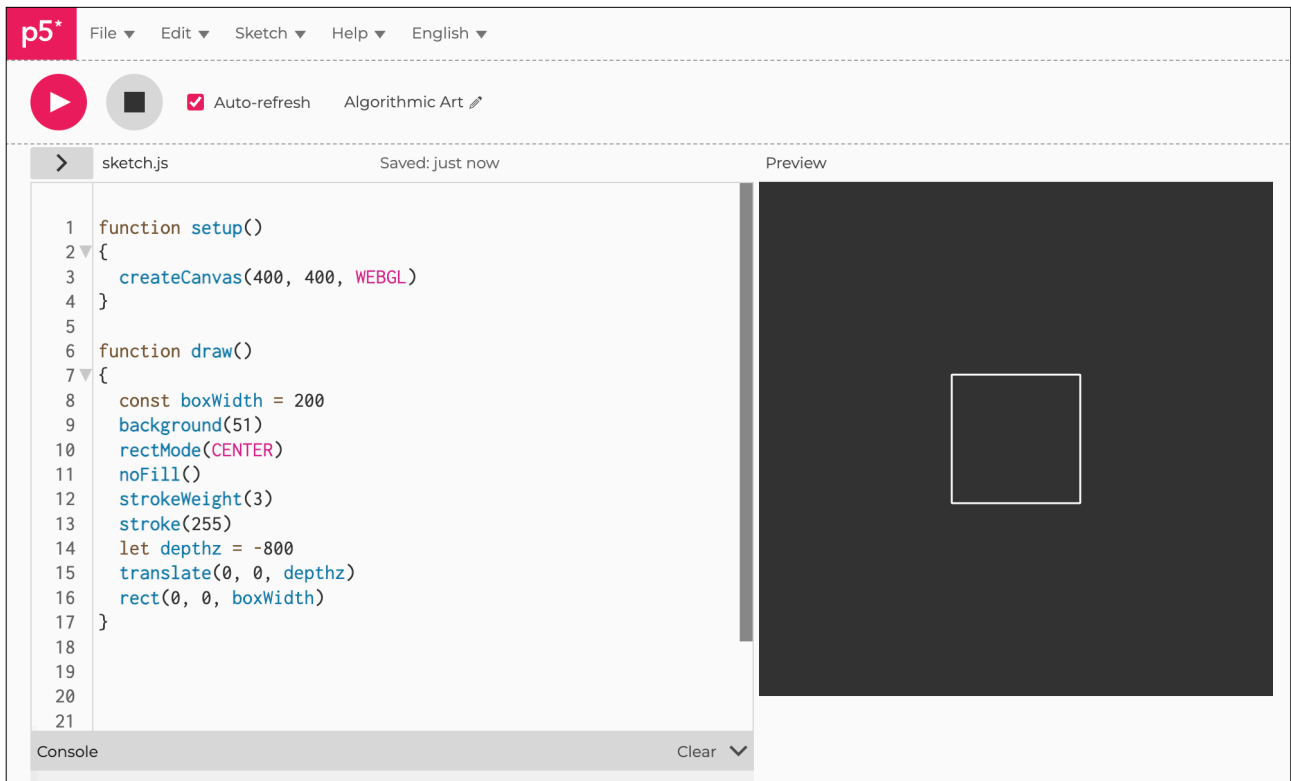
function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)

  let depthz = -800
  translate(0, 0, depthz)
  rect(0, 0, boxWidth)
}
```

Notes

The same-size square just further away in the **z** direction.

Figure G1.6





Sketch G1.7 parallel lines

We will be moving the square forwards shortly, but for the moment, we want to create the effect of a cube. So, we want the parallel lines that the square will follow. These are two squares. Think of them as where the walls meet the ceiling.

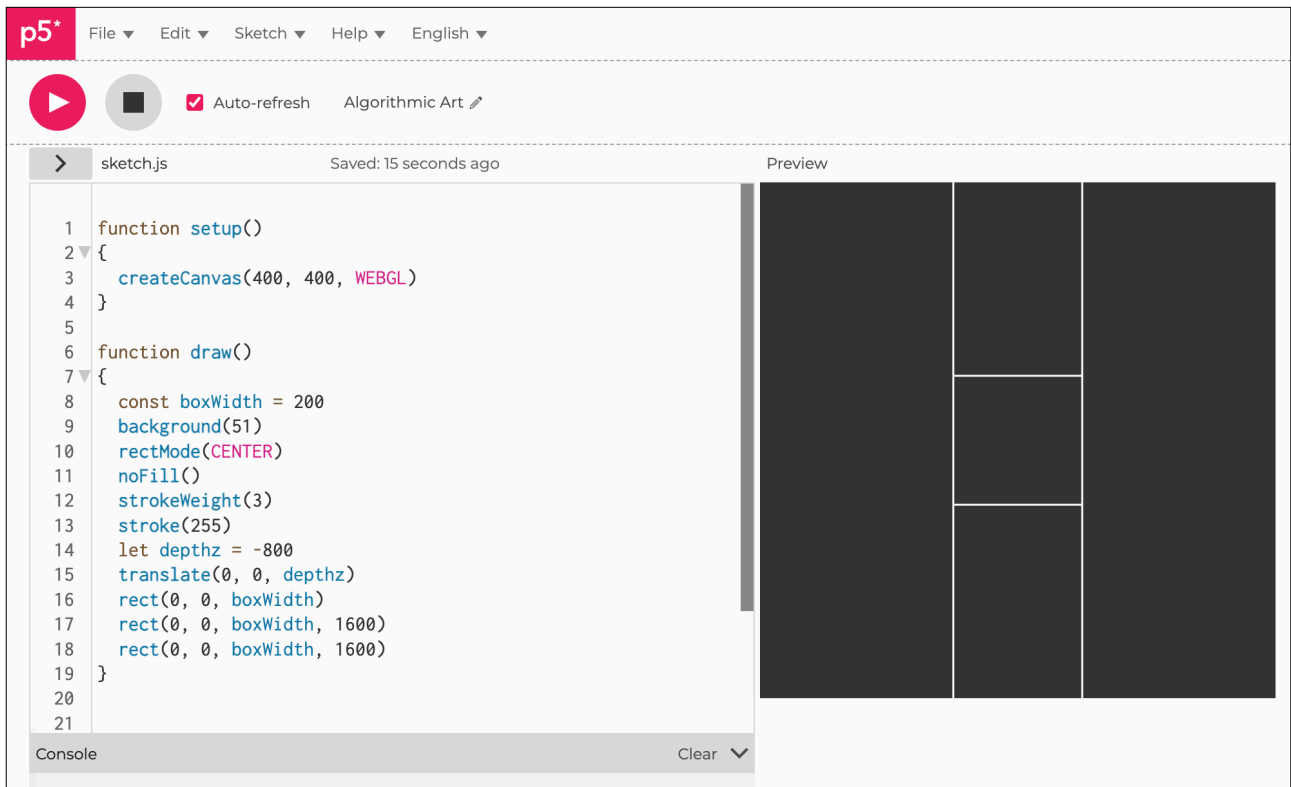
```
function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)
  let depthz = -800
  translate(0, 0, depthz)
  rect(0, 0, boxWidth)
  rect(0, 0, boxWidth, 1600)
  rect(0, 0, boxWidth, 1600)
}
```

Notes

This looks very uninspiring. All will become clear, honestly.

Figure G1.7





Sketch G1.8 translation

We will translate these last two rectangles. We will also add in some `push()` and `pop()` otherwise all the translations will be compounded. We want to keep them separate for when we move the square.

```
function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)

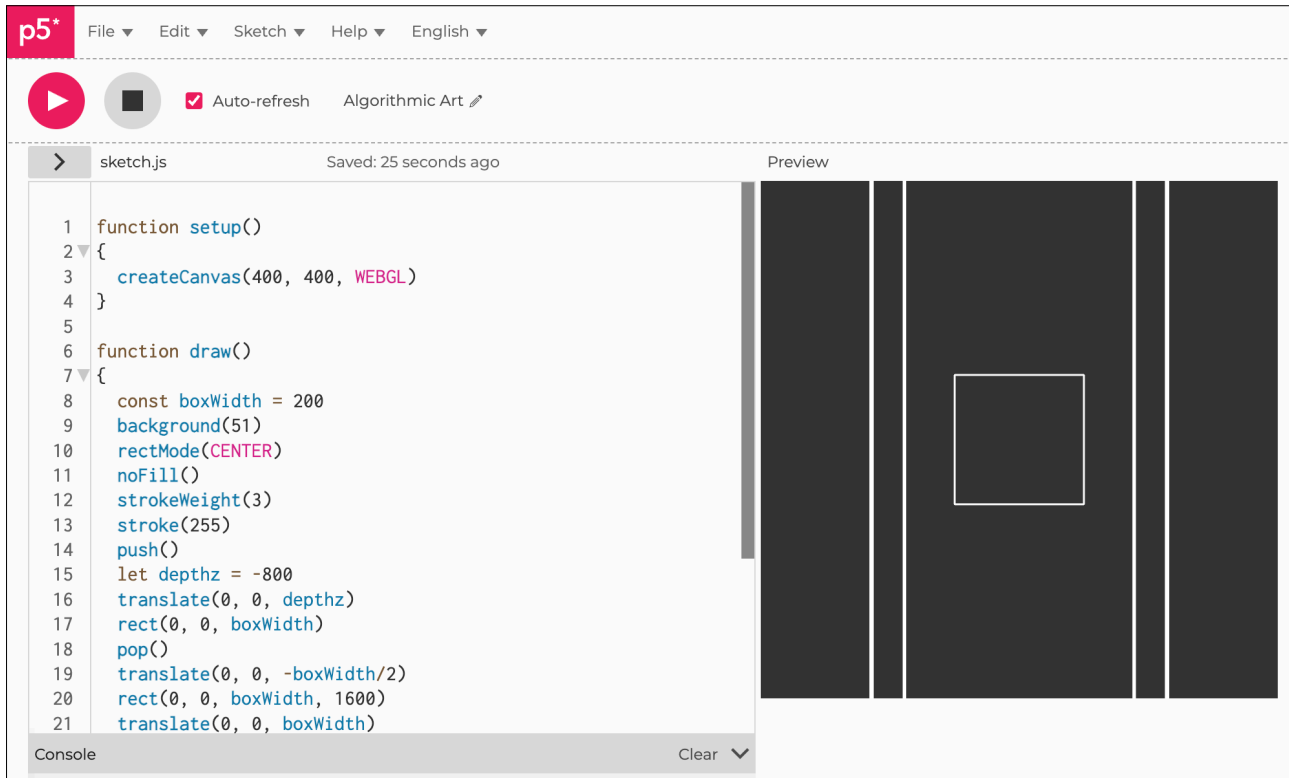
  push()
  let depthz = -800
  translate(0, 0, depthz)
  rect(0, 0, boxWidth)

  pop()
  translate(0, 0, -boxWidth/2)
  rect(0, 0, boxWidth, 1600)
  translate(0, 0, boxWidth)
  rect(0, 0, boxWidth, 1600)
}
```

Notes

Still looks unpromising!

Figure G1.8





Sketch G1.9 rotate

Now see what happens when we rotate the two rectangles.

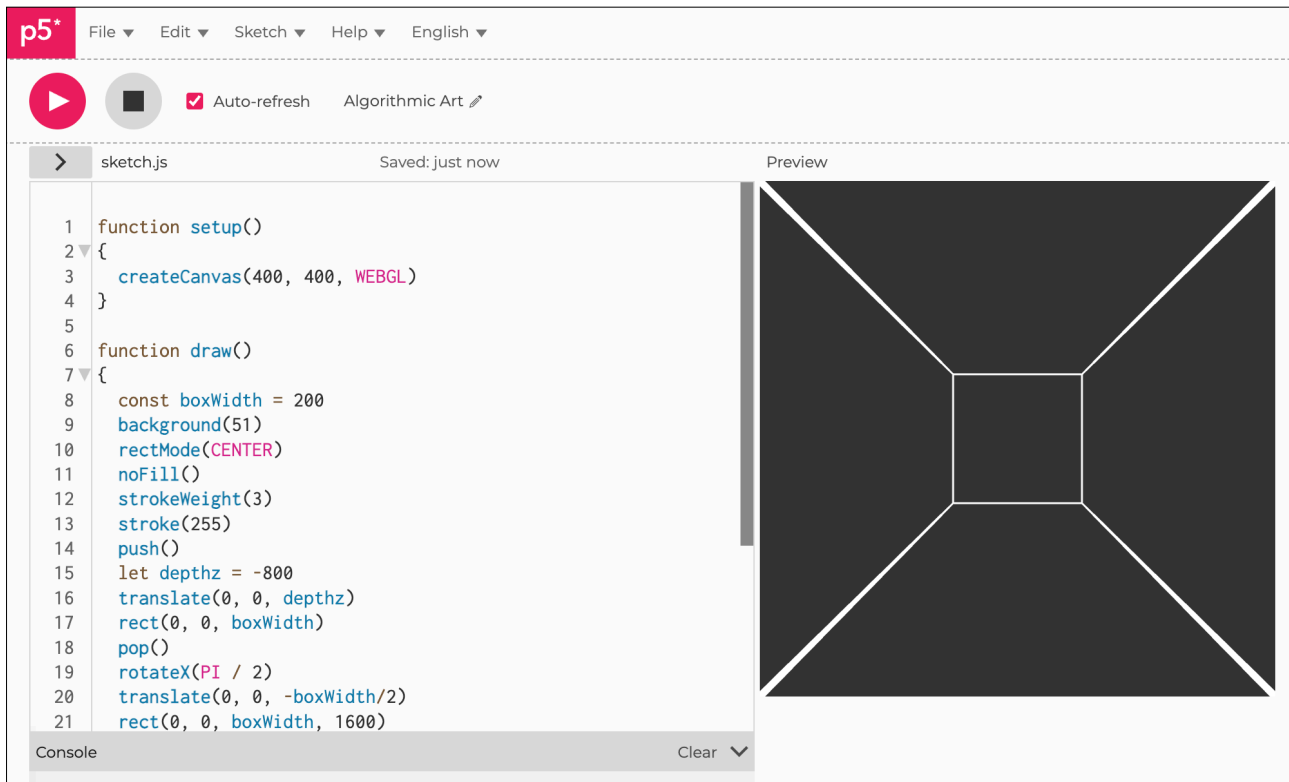
```
function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)
  push()
  let depthz = -800
  translate(0, 0, depthz)
  rect(0, 0, boxWidth)
  pop()
  rotateX(PI / 2)
  translate(0, 0, -boxWidth/2)
  rect(0, 0, boxWidth, 1600)
  translate(0, 0, boxWidth)
  rect(0, 0, boxWidth, 1600)
}
```

Notes

It now looks like the square has edges radiating out from it towards you. **PI** is the value of **180°** in degrees.

Figure G1.9





Sketch G1.10 moving depth

This bit can get a little complicated. We want to move the square forwards. We have a count to help advance the square incrementally.

```
let count = 0

function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)
  push()
  let depthz = count - 800
  translate(0, 0, depthz)
  rect(0, 0, boxWidth)
  count += 2
  pop()
  rotateX(PI / 2)
  translate(0, 0, -boxWidth/2)
  rect(0, 0, boxWidth, 1600)
  translate(0, 0, boxWidth)
  rect(0, 0, boxWidth, 1600)
}
```

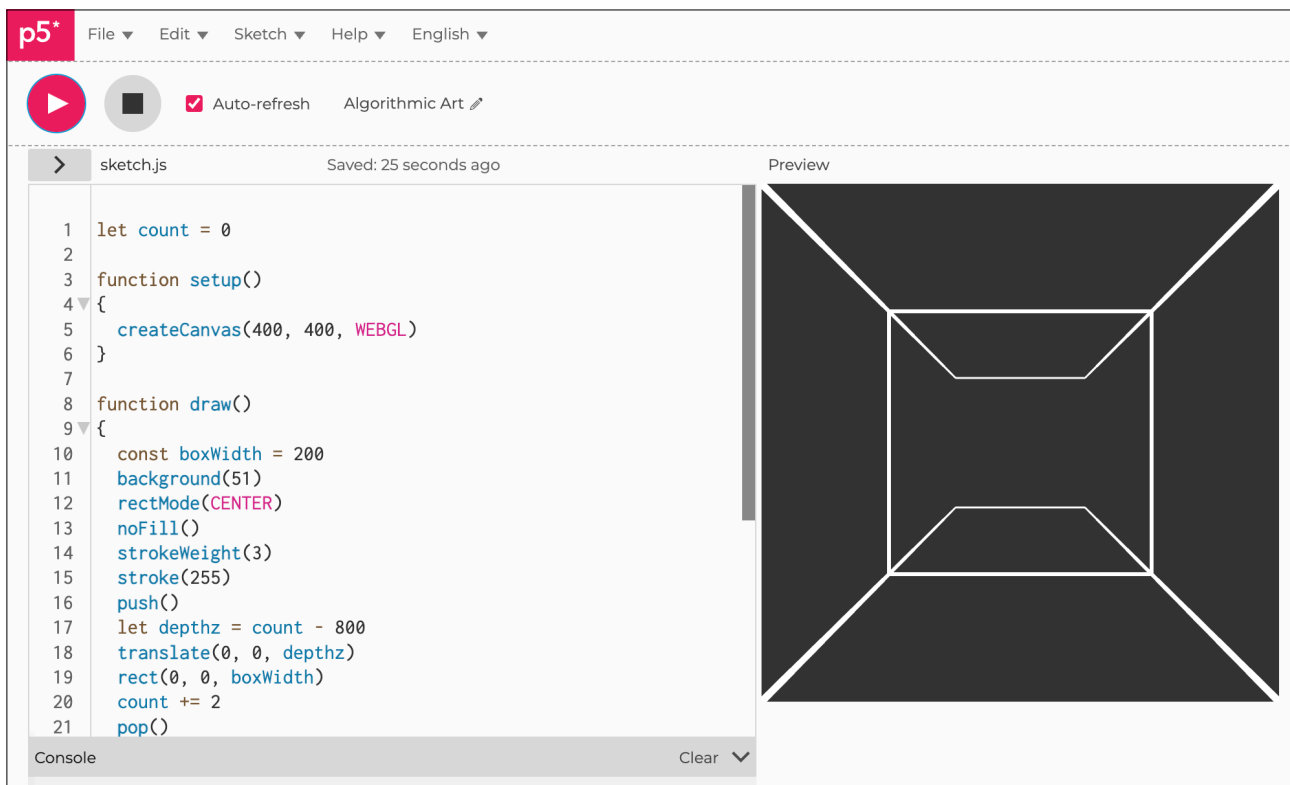
Notes

You should see the square slowly moving towards you. We start with the count on zero and the depth at **-800**, and the count increasing so that the depth eventually becomes zero.

Challenge

Try some other values (other than **800**) to see what is going on.

Figure G1.10





Sketch G1.11 lots of them

We want lots of squares all moving together, spaced out in intervals. We will create **30** squares. We will slow them down a little bit by changing the count. The distance between the squares is **100**, so each square is drawn **100** from the next one and then moves them forward at a slow speed to **0.05**.

```
let count = 0

function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)

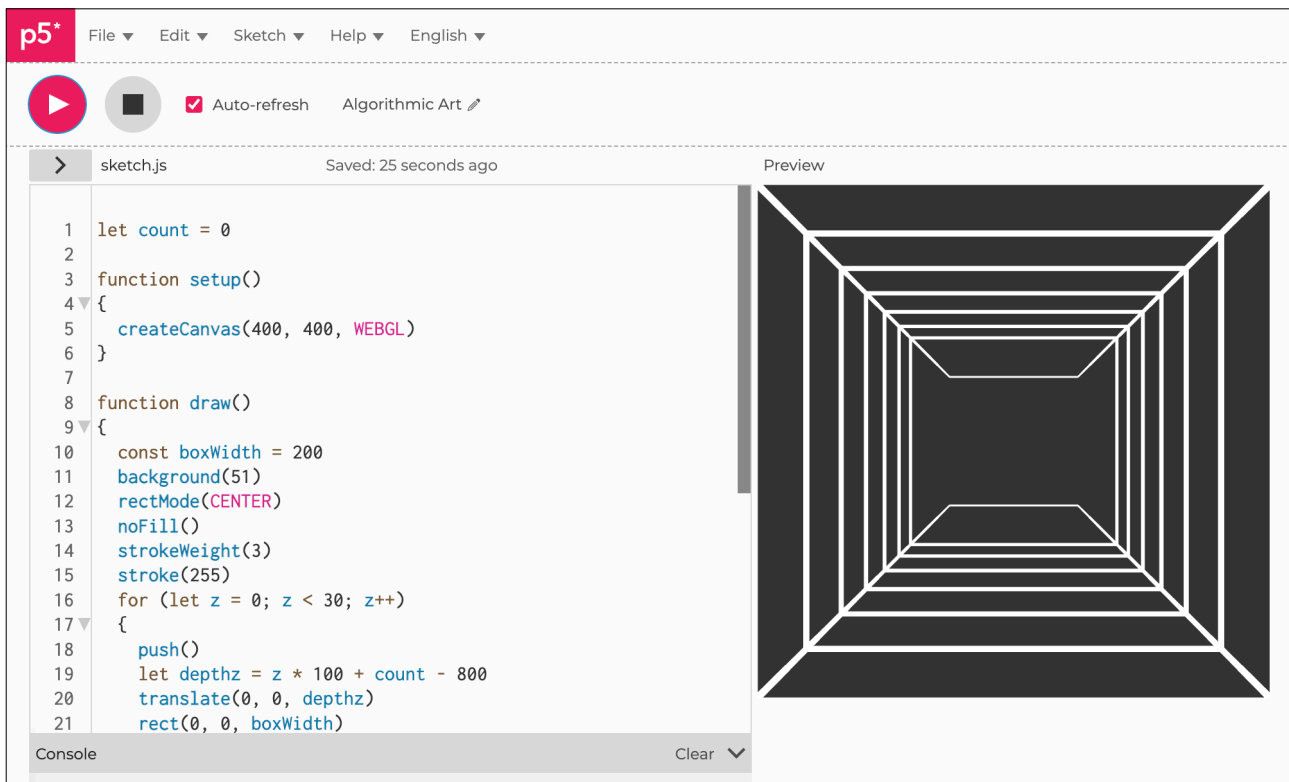
  for (let z = 0; z < 30; z++)
  {
    push()
    let depthz = z * 100 + count - 800
    translate(0, 0, depthz)
    rect(0, 0, boxWidth)
    count += 0.05
    pop()
  }

  rotateX(PI / 2)
  translate(0, 0, -boxWidth/2)
  rect(0, 0, boxWidth, 1600)
  translate(0, 0, boxWidth)
  rect(0, 0, boxWidth, 1600)
}
```

Notes

This is great except for one tiny detail: we eventually run out of squares. We need to find a way to keep drawing them continuously.

Figure G1.11





Sketch G1.12 a continuation

The next step is to redraw them. To do that, we need to reset the count back to zero once a certain number is reached. In this case, we will pick a number such as **2000** (because I know it works). Then subtract **800** to push it back to the right place.

```
let count = 0

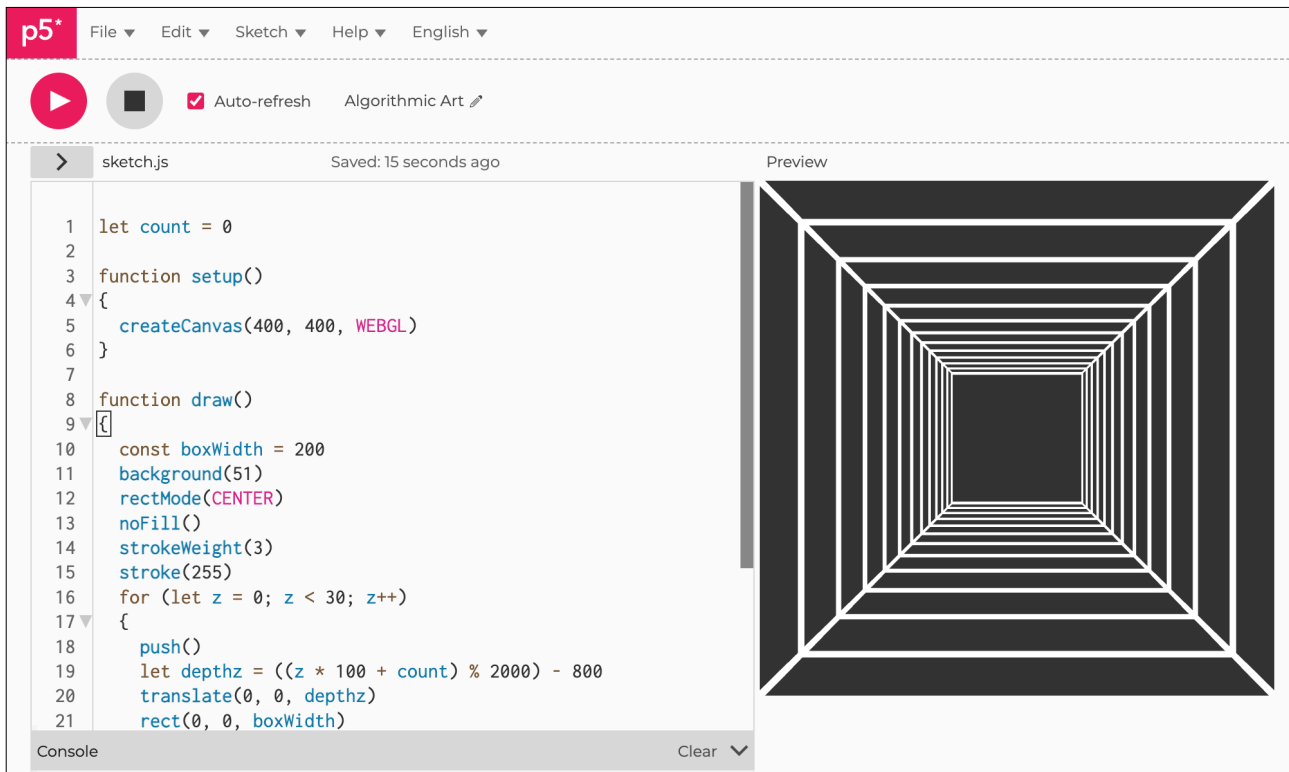
function setup()
{
  createCanvas(400, 400, WEBGL)
}

function draw()
{
  const boxWidth = 200
  background(51)
  rectMode(CENTER)
  noFill()
  strokeWeight(3)
  stroke(255)
  for (let z = 0; z < 30; z++)
  {
    push()
    let depthz = ((z * 100 + count) % 2000) - 800
    translate(0, 0, depthz)
    rect(0, 0, boxWidth)
    count += 0.05
    pop()
  }
  rotateX(PI / 2)
  translate(0, 0, -boxWidth/2)
  rect(0, 0, boxWidth, 1600)
  translate(0, 0, boxWidth)
  rect(0, 0, boxWidth, 1600)
}
```

Notes

You now have a continuous tunnel. But it is not 3D yet.

Figure G1.12





Sketch G1.13 adding 3D

We introduce the anaglyph bits and pieces.

```
let count = 0
let anaglyph

function setup()
{
  createCanvas(400, 400, WEBGL)
  anaglyph = createAnaglyph(this)
}

function draw()
{
  anaglyph.draw(scene)
}

function scene(graphics)
{
  const boxWidth = 200
  graphics.background(51)
  graphics.rectMode(CENTER)
  graphics.noFill()
  graphics.strokeWeight(3)
  graphics.stroke(255)
  for (let z = 0; z < 30; z++)
  {
    graphics.push()
    let depthz = ((z * 100 + count) % 2000) - 800
    graphics.translate(0, 0, depthz)
    graphics.rect(0, 0, boxWidth)
    count += 0.05
    graphics.pop()
  }
  graphics.rotateX(PI / 2)
  graphics.translate(0, 0, -boxWidth/2)
  graphics.rect(0, 0, boxWidth, 1600)
  graphics.translate(0, 0, boxWidth)
```

```
graphics.rect(0, 0, boxWidth, 1600)
```

```
}
```

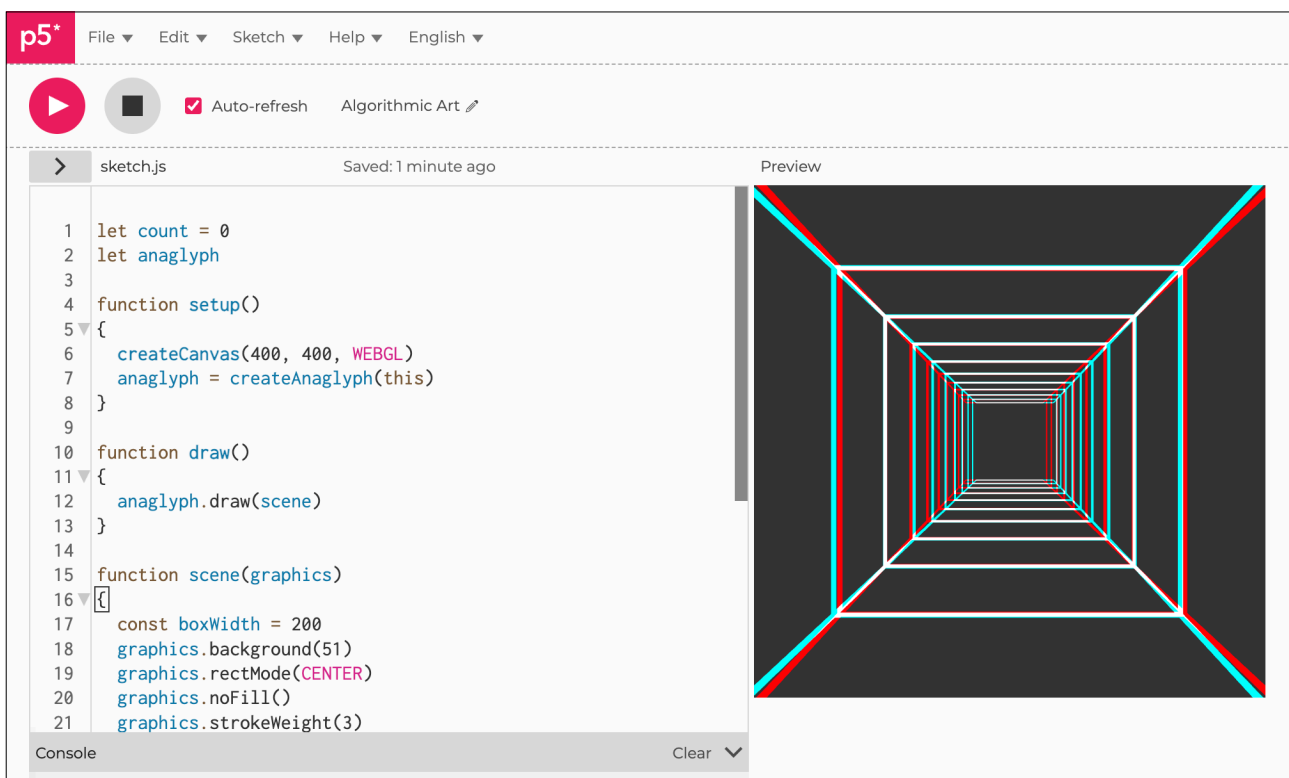
Notes

This is just an example to play with. There are lots of elements to this which you could transfer onto some other project. Using white lines, `stroke(255)` for better effect.

Challenge

Could you do the same thing with a triangle?

Figure G1.13





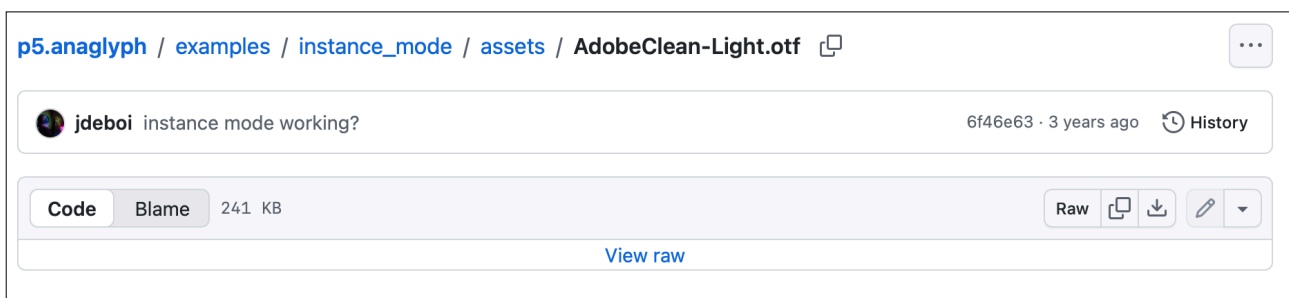
Sketch G1.14 text

We need to use a special kind of text file. We can't use just ordinary text with **WEBGL**. Go to the link below (I should have the link available on the website page). Download it and check where it is downloaded to:

https://github.com/jdeboi/p5.anaglyph/blob/main/examples/instance_mode/assets/AdobeClean-Light.otf

Download the file by clicking on the download button on the far right-hand side.

Figure 2: font download on GitHub



Upload the text file and add it to the list of files in the usual way.

! This is our starting sketch.

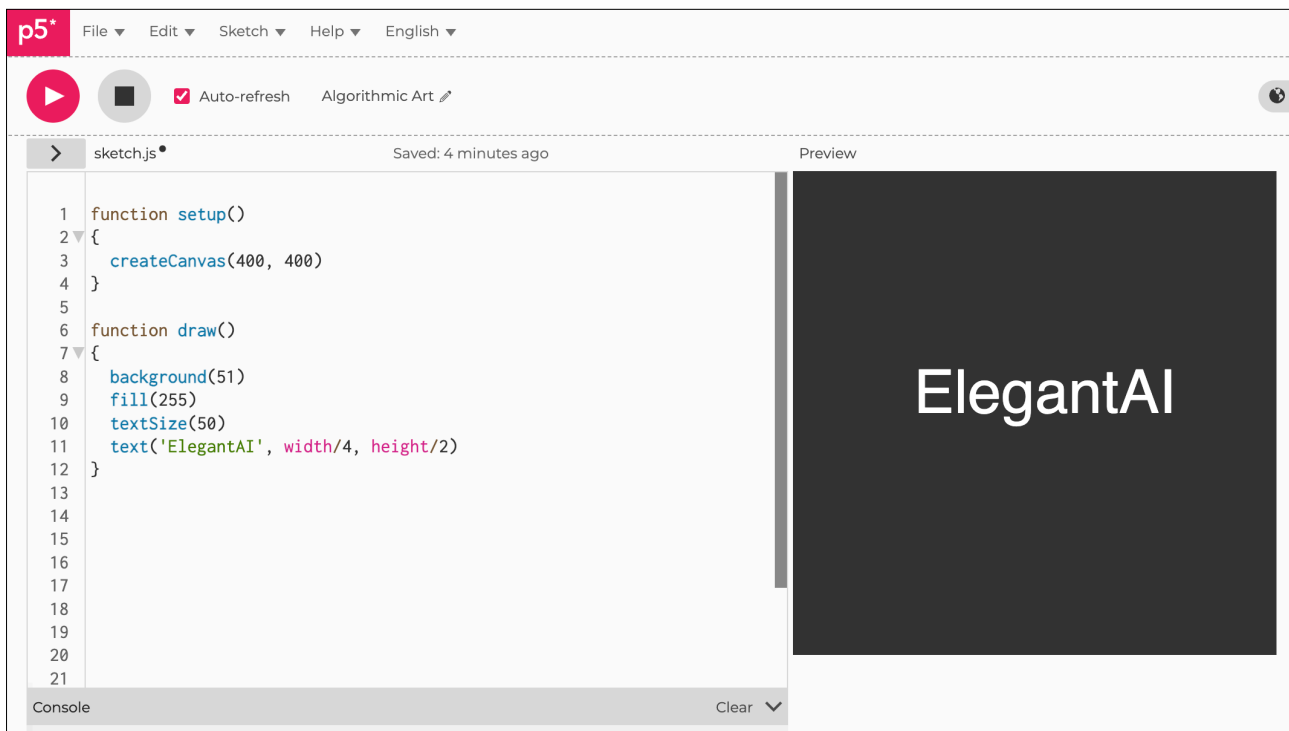
```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(51)
  fill(255)
  textSize(50)
  text('ElegantAI', width/4, height/2)
}
```

Notes

This gives us a bog-standard text on canvas.

Figure G1.14





Sketch G1.15 the font

We can change the font. To upload it before we start using it, we need to use asynchronous and await. Make sure the name in the `loadFont()` function matches the exact name of the font file.

```
let font

async function setup()
{
  createCanvas(400, 400)
  font = await loadFont('AdobeClean-Light.otf')
}

function draw()
{
  background(51)
  fill(255)
  textSize(50)
  textFont(font)
  text('ElegantAI', width/4, height/2)
}
```

Notes

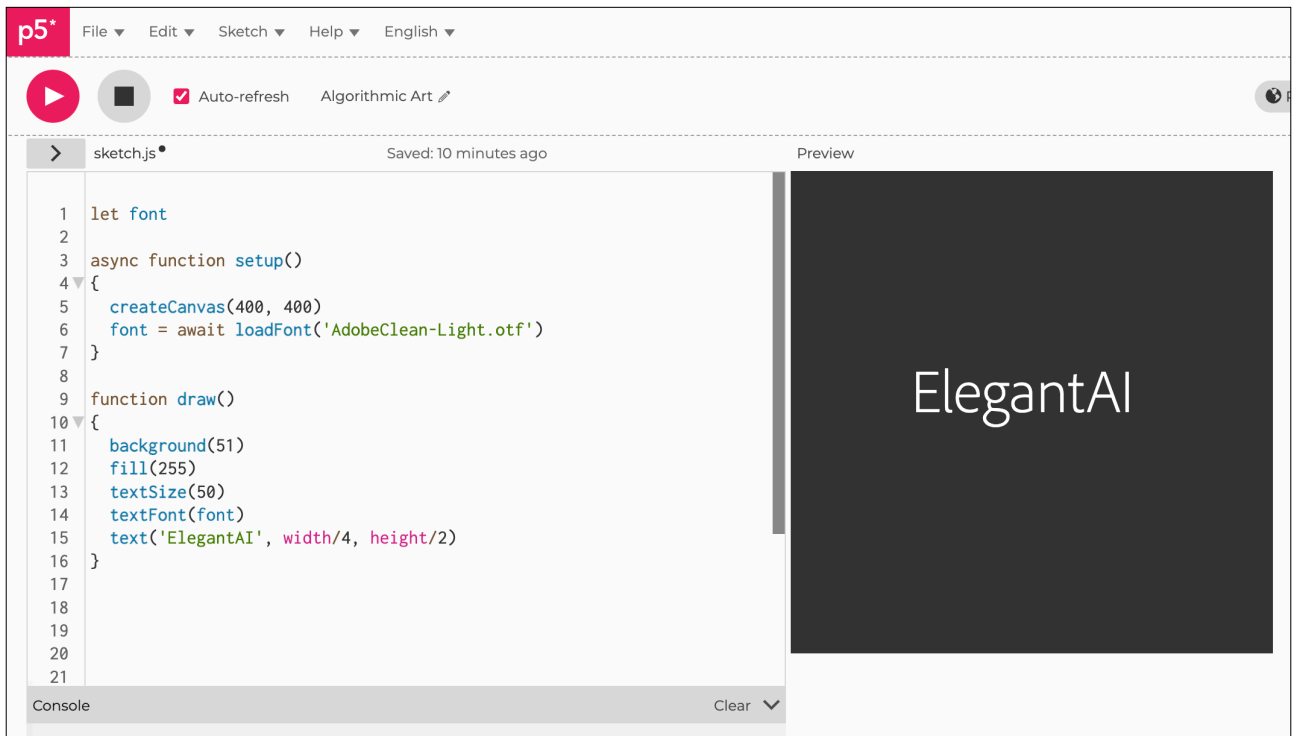
The font looks very similar. We need this extension to be able to use **WEBGL** and therefore anaglyph.

Challenge

You could try downloading a more interesting font off the internet, but they must be **.otf** or **.ttf** format.

Code Explanation

Figure G1.15





Sketch G1.16 WEBGL

Add the **WEBGL** and we can now make it rotate about an axis.

```
let font
let angle = 0

async function setup()
{
  createCanvas(400, 400, WEBGL)
  font = await loadFont('AdobeClean-Light.otf')
}

function draw()
{
  background(51)
  fill(255)
  textSize(50)
  textFont(font)
  rotateY(angle)
  text('ElegantAI', 0, 0)
  angle += 0.01
}
```

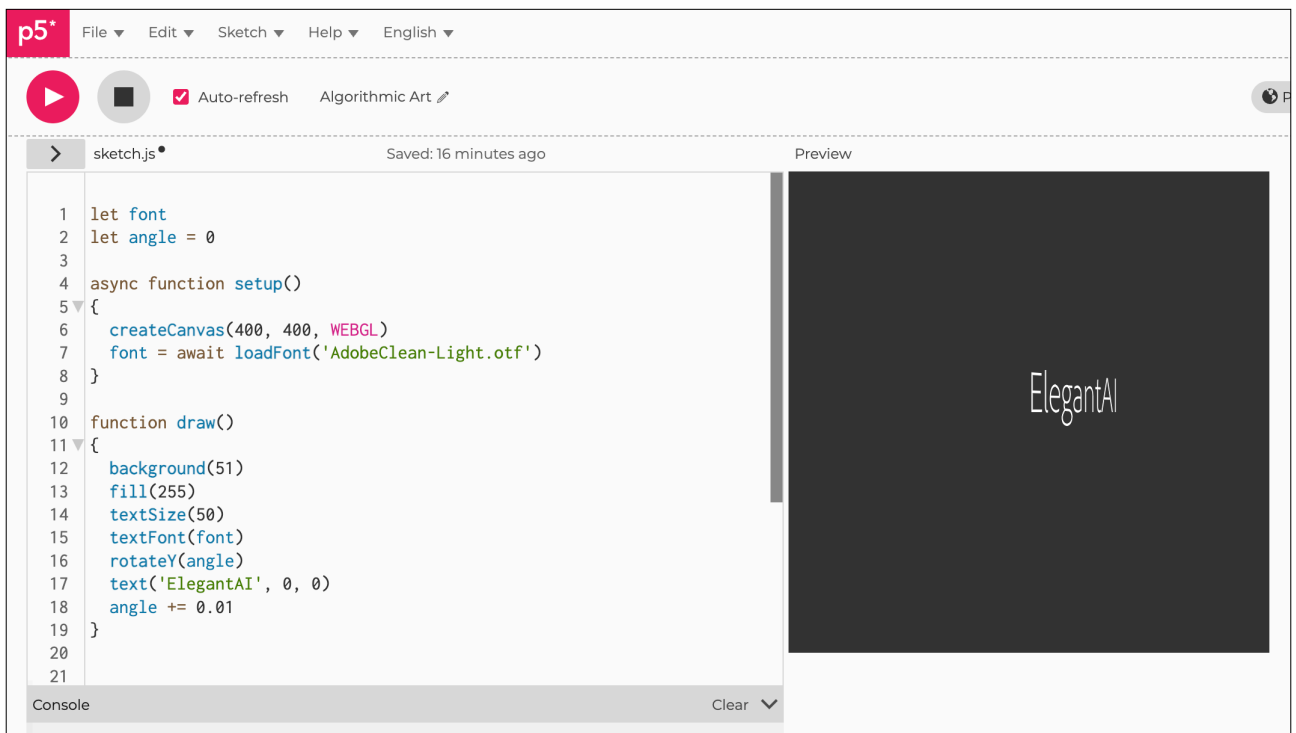
Notes

The text should be rotating slowly. Remember that we set the co-ordinates to **0, 0**.

Challenge

Can you rotate it about the centre of the text?

Figure G1.16





Sketch G1.17 the anaglyph

Quite a bit of refactoring, as you might imagine, and then put on your 3D glasses. We are also going to rotate it around the centre of the text.

```
let font
let angle = 0
let anaglyph

async function setup()
{
  createCanvas(400, 400, WEBGL)
  font = await loadFont('AdobeClean-Light.otf')
  anaglyph = createAnaglyph(this)
}

function draw()
{
  angle += 0.01
  anaglyph.draw(scene)
}

function scene(graphics)
{
  graphics.background(51)
  graphics.fill(255)
  graphics.textSize(50)
  graphics.textFont(font)
  graphics.rotateY(angle)
  graphics.text('ElegantAI', -75, 0)
}
```

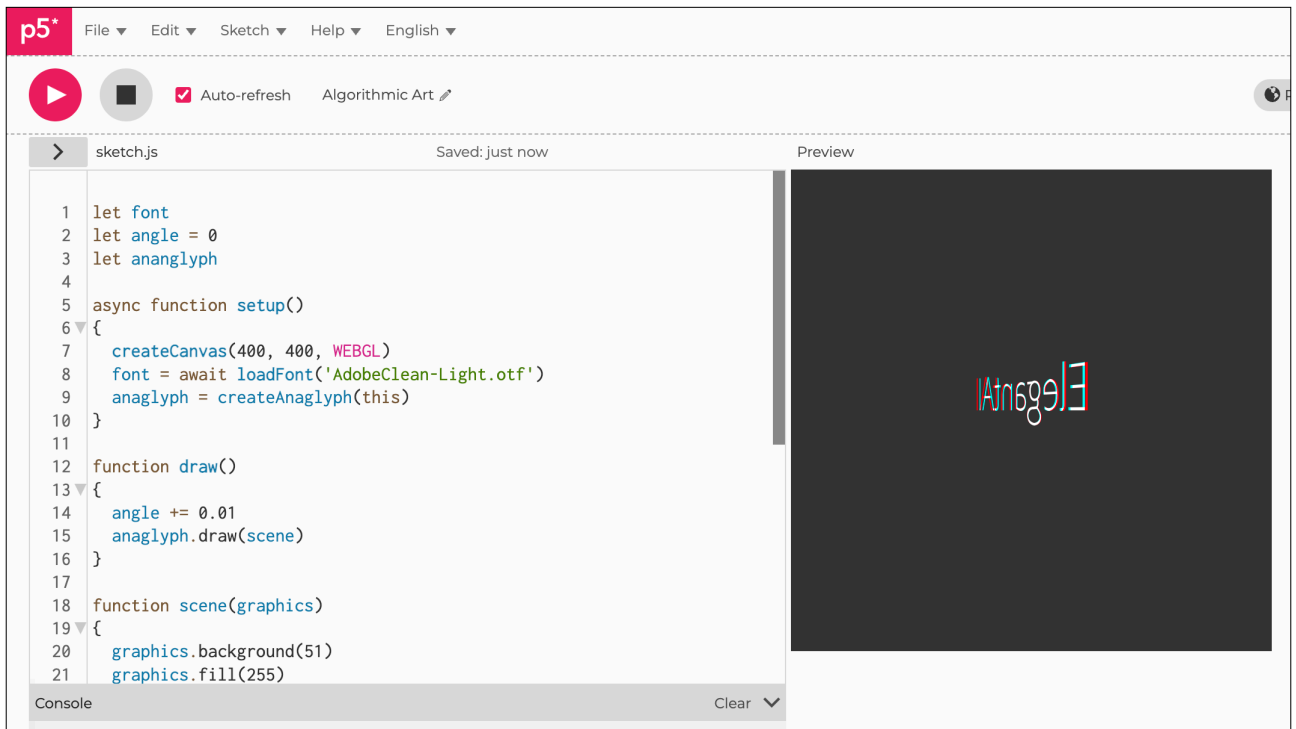
Notes

We rotate it around somewhere in the middle of the text. Also, now it is in 3D.

Challenge

Could you move the text backwards and forward?

Figure G1.17





Sketch G1.18 Dodecahedron

We can also create an irregular (although this is actually regular) shape using the `vertex()` function. In this case, we are going to create a **dodecahedron**, which means that there is going to be a whole lot of copying and pasting.

A **dodecahedron** has **12** faces, where each face is a **pentagon**. The pentagon has **5** edges and **5** vertices. This means that the **dodecahedron** has a total of **20** vertices.

This is a 3D object. We need the vectors of each vertex, which contain three elements (**x**, **y**, **z**). We need an array to hold all these vertices. We will call it `vert[]`. We also use some maths to calculate the other vertices. We use the **golden ratio**; we used something like this in the phyllotaxis example.

```
let vert = []

function setup()
{
  createCanvas(400, 400, WEBGL)
  const A = (1 + sqrt(5)) / 2
  const B = 1 / A
  vert.push(createVector(1, 1, 1))
  vert.push(createVector(1, 1, -1))
  vert.push(createVector(1, -1, 1))
  vert.push(createVector(1, -1, -1))
  vert.push(createVector(-1, 1, 1))
  vert.push(createVector(-1, 1, -1))
  vert.push(createVector(-1, -1, 1))
  vert.push(createVector(-1, -1, -1))
  vert.push(createVector(0, B, A))
  vert.push(createVector(0, B, -A))
  vert.push(createVector(0, -B, A))
  vert.push(createVector(0, -B, -A))
  vert.push(createVector(B, A, 0))
  vert.push(createVector(B, -A, 0))
  vert.push(createVector(-B, A, 0))
  vert.push(createVector(-B, -A, 0))
  vert.push(createVector(A, 0, B))
  vert.push(createVector(A, 0, -B))
  vert.push(createVector(-A, 0, B))
  vert.push(createVector(-A, 0, -B))
}
```



```
}
```

Notes

These are just the vertices. Believe it or not, we need to take these vertices in a very specific order and arrange them around each pentagon face. Admittedly, other shapes might have been easier.

Code Explanation

<code>const A = (1 + sqrt(5)) / 2</code>	The golden ratio approximately: 1.618033988749895
<code>const B = 1 / A</code>	The inverse of the golden ratio is: 0.6180339887498948



Sketch G1.19 face the pentagon

We need to combine the vertices to each face. We have 12 faces, each with 5 vertices. Some of these vertices are shared by 3 faces. The great thing is that someone has done the hard work for us (not me), and we can identify each vertex with an individual index, which will have three elements. We will use this information later.

We need an array to hold these faces, and we will call the array `faces[]`.

```
let vert = []
let faces = []

function setup()
{
  createCanvas(400, 400, WEBGL)
  const A = (1 + sqrt(5)) / 2
  const B = 1 / A
  vert.push(createVector(1, 1, 1))
  vert.push(createVector(1, 1, -1))
  vert.push(createVector(1, -1, 1))
  vert.push(createVector(1, -1, -1))
  vert.push(createVector(-1, 1, 1))
  vert.push(createVector(-1, 1, -1))
  vert.push(createVector(-1, -1, 1))
  vert.push(createVector(-1, -1, -1))
  vert.push(createVector(0, B, A))
  vert.push(createVector(0, B, -A))
  vert.push(createVector(0, -B, A))
  vert.push(createVector(0, -B, -A))
  vert.push(createVector(B, A, 0))
  vert.push(createVector(B, -A, 0))
  vert.push(createVector(-B, A, 0))
  vert.push(createVector(-B, -A, 0))
  vert.push(createVector(A, 0, B))
  vert.push(createVector(A, 0, -B))
  vert.push(createVector(-A, 0, B))
  vert.push(createVector(-A, 0, -B))

  faces.push([0, 16, 2, 10, 8])
  faces.push([0, 8, 4, 14, 12])
```

```
faces.push([16, 17, 1, 12, 0])
faces.push([1, 9, 11, 3, 17])
faces.push([1, 12, 14, 5, 9])
faces.push([2, 13, 15, 6, 10])
faces.push([13, 3, 17, 16, 2])
faces.push([3, 11, 7, 15, 13])
faces.push([4, 8, 10, 6, 18])
faces.push([14, 5, 19, 18, 4])
faces.push([5, 19, 7, 11, 9])
faces.push([15, 7, 19, 18, 6])
```

```
}
```

Notes

To speed up the pasting and copying, I created a line with x's rather than numbers and copied and pasted 12 times. Then I went back through them very carefully, changing the x to the relevant number. A bit tedious but necessary.

Challenge

I am sure you can find a quicker way to do this.



Sketch G1.20 resizing

You may be wondering what this is going to look like as the numbers we have put in seem to be mainly 1's and 0's or thereabouts. We need to increase each vector by 50 so that it is something we will actually be able to see.

```
let vert = []
let faces = []
let r = 50

function setup()
{
  createCanvas(400, 400, WEBGL)
  const A = (1 + sqrt(5)) / 2
  const B = 1 / A
  vert.push(createVector(1, 1, 1))
  vert.push(createVector(1, 1, -1))
  vert.push(createVector(1, -1, 1))
  vert.push(createVector(1, -1, -1))
  vert.push(createVector(-1, 1, 1))
  vert.push(createVector(-1, 1, -1))
  vert.push(createVector(-1, -1, 1))
  vert.push(createVector(-1, -1, -1))
  vert.push(createVector(0, B, A))
  vert.push(createVector(0, B, -A))
  vert.push(createVector(0, -B, A))
  vert.push(createVector(0, -B, -A))
  vert.push(createVector(B, A, 0))
  vert.push(createVector(B, -A, 0))
  vert.push(createVector(-B, A, 0))
  vert.push(createVector(-B, -A, 0))
  vert.push(createVector(A, 0, B))
  vert.push(createVector(A, 0, -B))
  vert.push(createVector(-A, 0, B))
  vert.push(createVector(-A, 0, -B))
  for (let i = 0; i < vert.length; i++)
  {
    vert[i].mult(r)
  }
}
```

```
faces.push([0, 16, 2, 10, 8])
faces.push([0, 8, 4, 14, 12])
faces.push([16, 17, 1, 12, 0])
faces.push([1, 9, 11, 3, 17])
faces.push([1, 12, 14, 5, 9])
faces.push([2, 13, 15, 6, 10])
faces.push([13, 3, 17, 16, 2])
faces.push([3, 11, 7, 15, 13])
faces.push([4, 8, 10, 6, 18])
faces.push([14, 5, 19, 18, 4])
faces.push([5, 19, 7, 11, 9])
faces.push([15, 7, 19, 18, 6])
}
```

Notes

You can use `console.log(vert[2])` to see how that works.

Challenge

You can multiply by a different number if you want.



Sketch G1.21 draw the dodecahedron

We have an array of faces. In each face, we have an array which are the indices for the vertices. Basically, we systematically work through each face and draw out each vertex. Each vertex is a point, and we can join these vertices together.

```
let vert = []
let faces = []
let r = 50

function setup()
{
  createCanvas(400, 400, WEBGL)
  const A = (1 + sqrt(5)) / 2
  const B = 1 / A
  vert.push(createVector(1, 1, 1))
  vert.push(createVector(1, 1, -1))
  vert.push(createVector(1, -1, 1))
  vert.push(createVector(1, -1, -1))
  vert.push(createVector(-1, 1, 1))
  vert.push(createVector(-1, 1, -1))
  vert.push(createVector(-1, -1, 1))
  vert.push(createVector(-1, -1, -1))
  vert.push(createVector(0, B, A))
  vert.push(createVector(0, B, -A))
  vert.push(createVector(0, -B, A))
  vert.push(createVector(0, -B, -A))
  vert.push(createVector(B, A, 0))
  vert.push(createVector(B, -A, 0))
  vert.push(createVector(-B, A, 0))
  vert.push(createVector(-B, -A, 0))
  vert.push(createVector(A, 0, B))
  vert.push(createVector(A, 0, -B))
  vert.push(createVector(-A, 0, B))
  vert.push(createVector(-A, 0, -B))
  for (let i = 0; i < vert.length; i++)
  {
    vert[i].mult(r)
  }
}
```

```

faces.push([0, 16, 2, 10, 8])
faces.push([0, 8, 4, 14, 12])
faces.push([16, 17, 1, 12, 0])
faces.push([1, 9, 11, 3, 17])
faces.push([1, 12, 14, 5, 9])
faces.push([2, 13, 15, 6, 10])
faces.push([13, 3, 17, 16, 2])
faces.push([3, 11, 7, 15, 13])
faces.push([4, 8, 10, 6, 18])
faces.push([14, 5, 19, 18, 4])
faces.push([5, 19, 7, 11, 9])
faces.push([15, 7, 19, 18, 6])
}

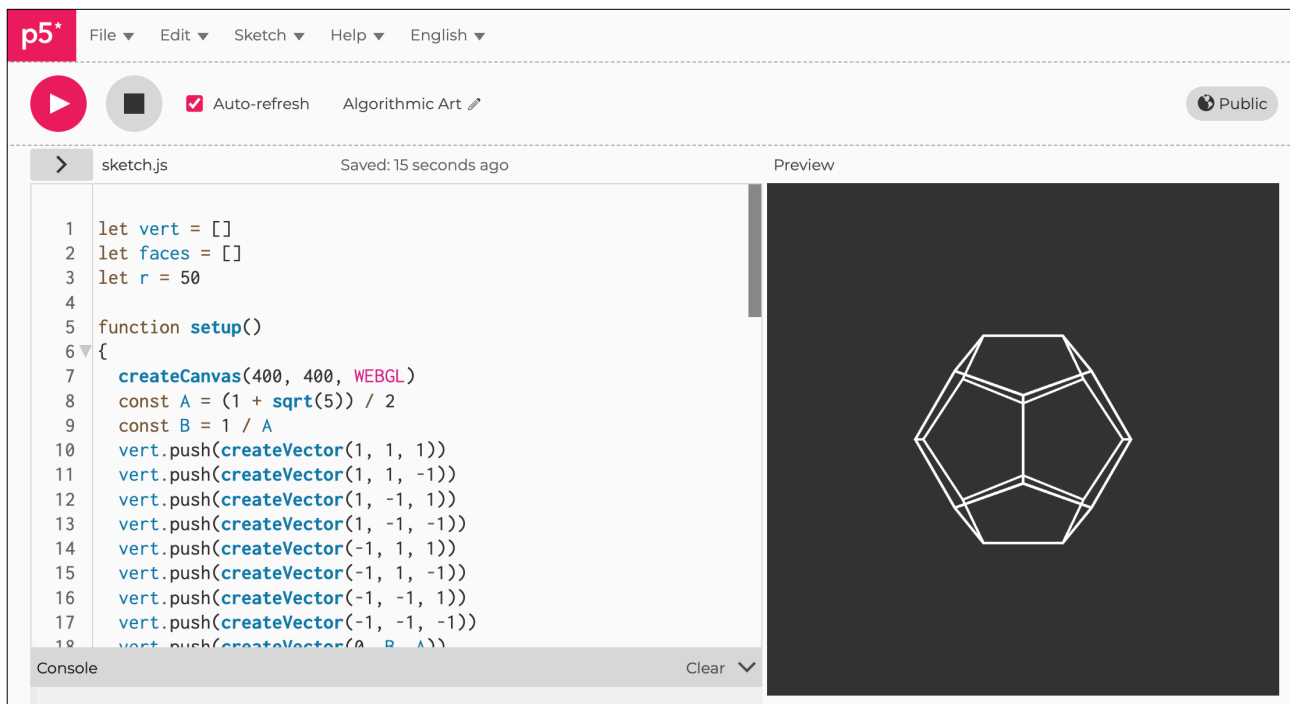
function draw()
{
  background(51)
  stroke(255)
  strokeWeight(2)
  noFill()
  for (let i = 0; i < faces.length; i++)
  {
    beginShape()
    for (let j = 0; j < faces[i].length; j++)
    {
      let v = vert[faces[i][j]]
      vertex(v.x, v.y, v.z)
    }
    endShape(CLOSE)
  }
}

```

Notes

We now have something to look at, a static dodecahedron. All that hard work has paid off.

Figure G1.21





Sketch G1.22 rotate

We can now rotate the dodecahedron.

```
let vert = []
let faces = []
let r = 50
let angle = 0

function setup()
{
  createCanvas(400, 400, WEBGL)
  const A = (1 + sqrt(5)) / 2
  const B = 1 / A
  vert.push(createVector(1, 1, 1))
  vert.push(createVector(1, 1, -1))
  vert.push(createVector(1, -1, 1))
  vert.push(createVector(1, -1, -1))
  vert.push(createVector(-1, 1, 1))
  vert.push(createVector(-1, 1, -1))
  vert.push(createVector(-1, -1, 1))
  vert.push(createVector(-1, -1, -1))
  vert.push(createVector(0, B, A))
  vert.push(createVector(0, B, -A))
  vert.push(createVector(0, -B, A))
  vert.push(createVector(0, -B, -A))
  vert.push(createVector(B, A, 0))
  vert.push(createVector(B, -A, 0))
  vert.push(createVector(-B, A, 0))
  vert.push(createVector(-B, -A, 0))
  vert.push(createVector(A, 0, B))
  vert.push(createVector(A, 0, -B))
  vert.push(createVector(-A, 0, B))
  vert.push(createVector(-A, 0, -B))
  for (let i = 0; i < vert.length; i++)
  {
    vert[i].mult(r)
  }
  faces.push([0, 16, 2, 10, 8])
}
```

```

    faces.push([0, 8, 4, 14, 12])
    faces.push([16, 17, 1, 12, 0])
    faces.push([1, 9, 11, 3, 17])
    faces.push([1, 12, 14, 5, 9])
    faces.push([2, 13, 15, 6, 10])
    faces.push([13, 3, 17, 16, 2])
    faces.push([3, 11, 7, 15, 13])
    faces.push([4, 8, 10, 6, 18])
    faces.push([14, 5, 19, 18, 4])
    faces.push([5, 19, 7, 11, 9])
    faces.push([15, 7, 19, 18, 6])
  }

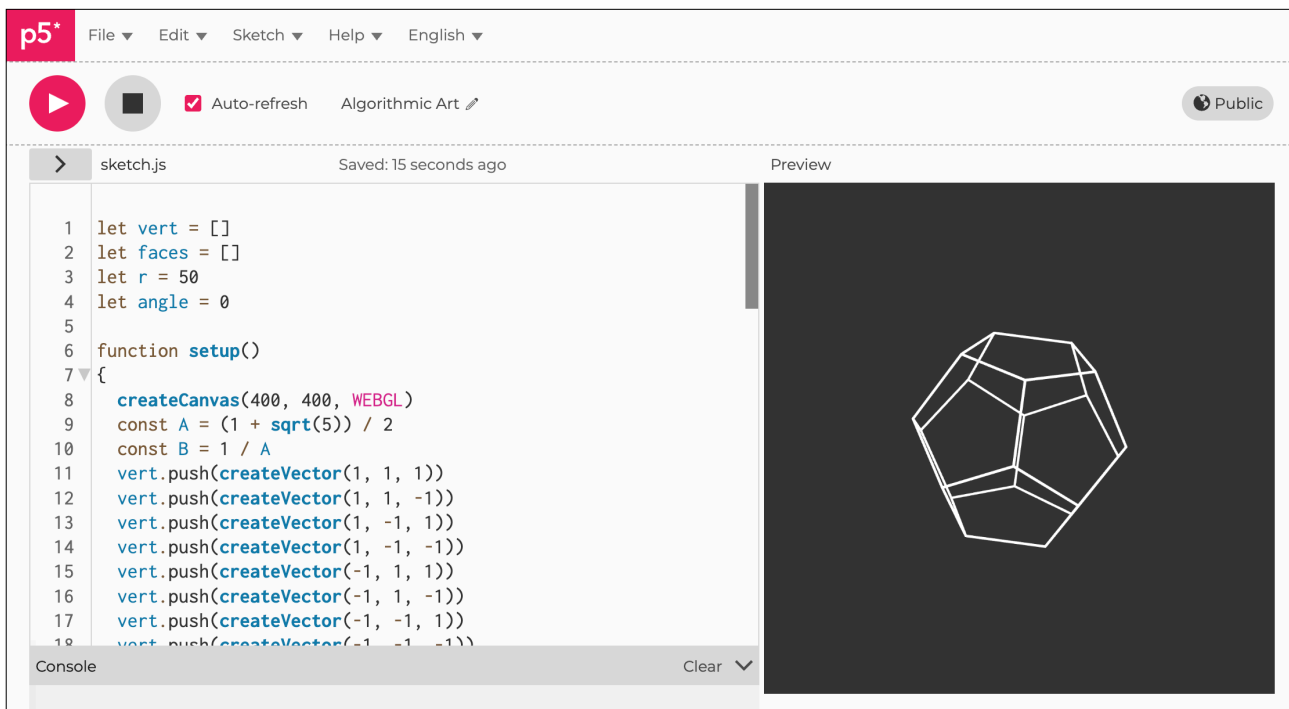
  function draw()
  {
    background(51)
    rotateX(angle)
    rotateY(angle)
    stroke(255)
    strokeWeight(2)
    noFill()
    for (let i = 0; i < faces.length; i++)
    {
      beginShape()
      for (let j = 0; j < faces[i].length; j++)
      {
        let v = vert[faces[i][j]]
        vertex(v.x, v.y, v.z)
      }
      endShape(CLOSE)
    }
    angle += 0.01
  }

```

Notes

A lovely rotating dodecahedron.

Figure G1.22





Sketch G1.23 the anaglyph

Here we bring in the usual changes. One way is to copy most of the `draw()` function and paste it into the new `scene()` function. Then we add the `graphic.` prefix. Time to put your glasses on. I think you will agree it looks great.

```
let vert = []
let faces = []
let r = 50
let angle = 0
let anaglyph

function setup()
{
  createCanvas(400, 400, WEBGL)
  anaglyph = createAnaglyph(this)
  const A = (1 + sqrt(5)) / 2
  const B = 1 / A
  vert.push(createVector(1, 1, 1))
  vert.push(createVector(1, 1, -1))
  vert.push(createVector(1, -1, 1))
  vert.push(createVector(1, -1, -1))
  vert.push(createVector(-1, 1, 1))
  vert.push(createVector(-1, 1, -1))
  vert.push(createVector(-1, -1, 1))
  vert.push(createVector(-1, -1, -1))
  vert.push(createVector(0, B, A))
  vert.push(createVector(0, B, -A))
  vert.push(createVector(0, -B, A))
  vert.push(createVector(0, -B, -A))
  vert.push(createVector(B, A, 0))
  vert.push(createVector(B, -A, 0))
  vert.push(createVector(-B, A, 0))
  vert.push(createVector(-B, -A, 0))
  vert.push(createVector(A, 0, B))
  vert.push(createVector(A, 0, -B))
  vert.push(createVector(-A, 0, B))
  vert.push(createVector(-A, 0, -B))
  for (let i = 0; i < vert.length; i++)
```

```

{
  vert[i].mult(r)
}
faces.push([0, 16, 2, 10, 8])
faces.push([0, 8, 4, 14, 12])
faces.push([16, 17, 1, 12, 0])
faces.push([1, 9, 11, 3, 17])
faces.push([1, 12, 14, 5, 9])
faces.push([2, 13, 15, 6, 10])
faces.push([13, 3, 17, 16, 2])
faces.push([3, 11, 7, 15, 13])
faces.push([4, 8, 10, 6, 18])
faces.push([14, 5, 19, 18, 4])
faces.push([5, 19, 7, 11, 9])
faces.push([15, 7, 19, 18, 6])
}

function draw()
{
  anaglyph.draw(scene)
  angle += 0.01
}

function scene(graphic)
{
  graphic.background(51)
  graphic.rotateX(angle)
  graphic.rotateY(angle)
  graphic.stroke(255)
  graphic.strokeWeight(2)
  graphic.noFill()
  for (let i = 0; i < faces.length; i++)
  {
    graphic.beginShape()
    for (let j = 0; j < faces[i].length; j++)
    {
      let v = vert[faces[i][j]]
      graphic.vertex(v.x, v.y, v.z)
    }
  }
}

```

```

    }
    graphic.endShape(CLOSE)
  }
}

```

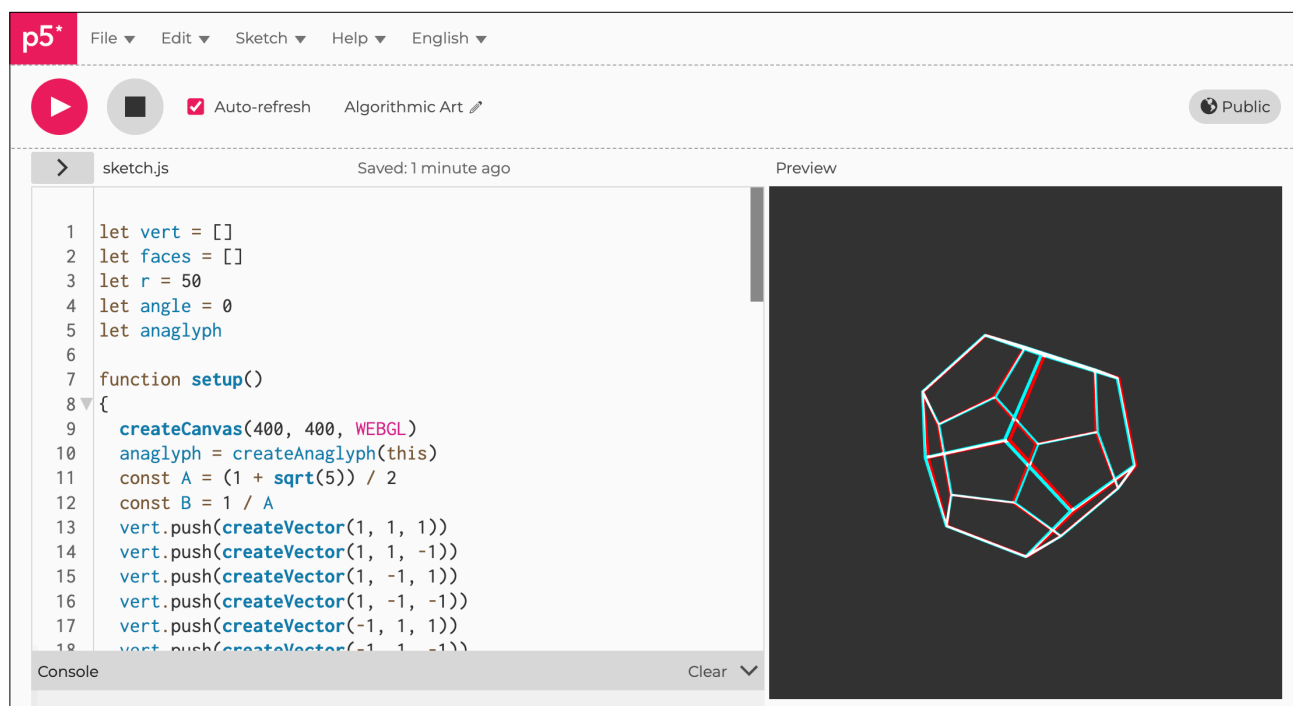
Notes

This shows that you do need to have a good understanding of geometry. I have shown you this to whet your appetite.

Challenge

Try your own designs; maybe start with something simpler to start with.

Figure G1.23



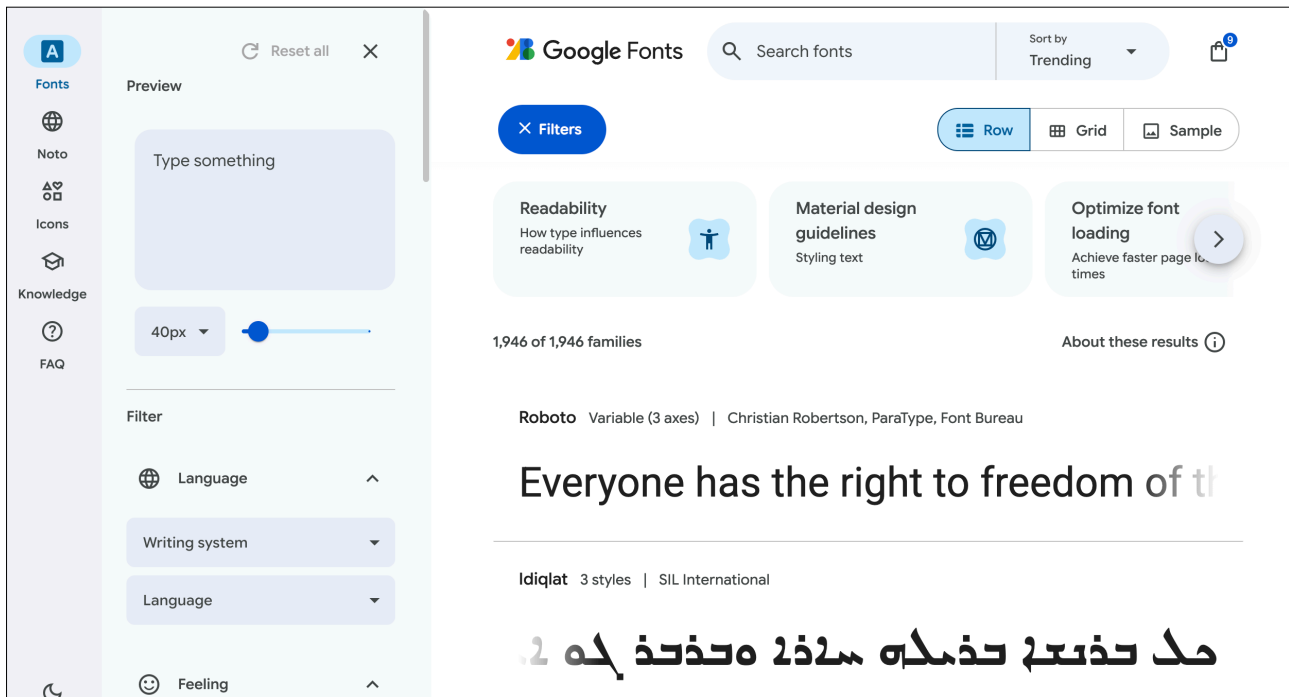
Algorithmic
Art
Module G
Unit #2
typography



Module G Unit #2 typography

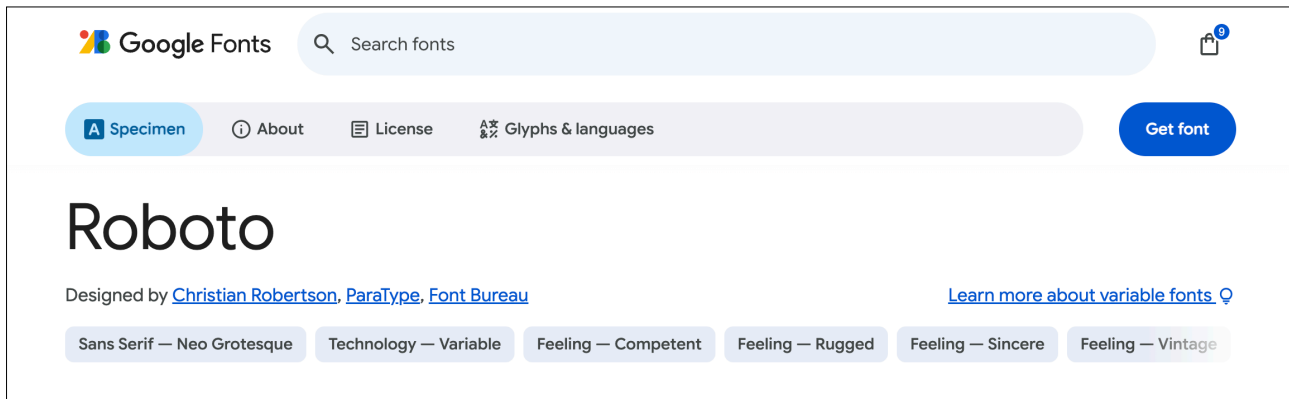
We want to download a font for this unit. The one we want has variable weights. The one I suggest is called **Roboto-VariableFont_width,wght.ttf** which is a bit of a mouthful. To download a **.ttf** font, first go to <https://fonts.google.com/>. There are some much more interesting fonts which I would encourage you to explore, but finding a variable weights one is harder than you might think.

Figure 1: Google fonts



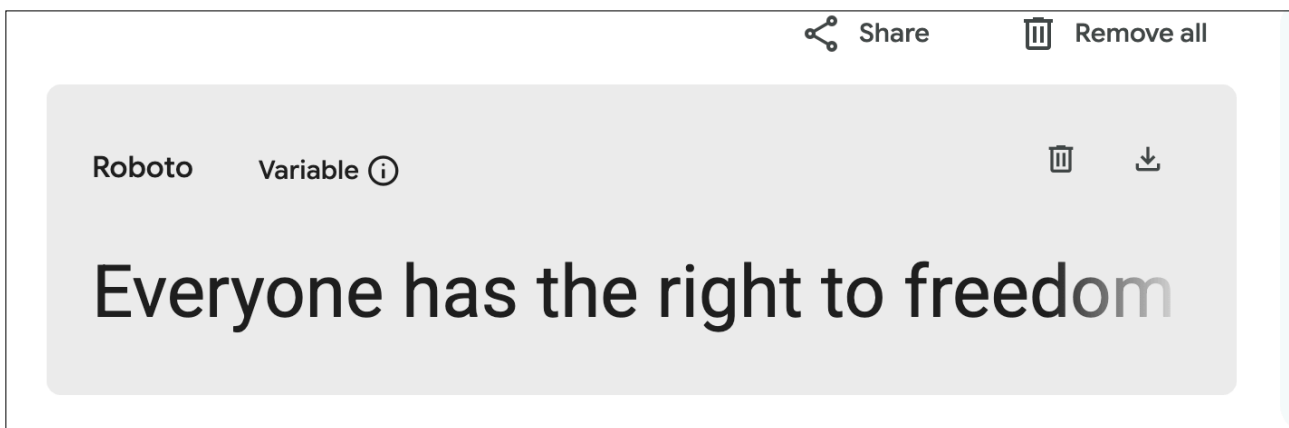
Next, select the family for **Roboto** and click on the **Get font** button (top right).

Figure 2: Roboto family of fonts



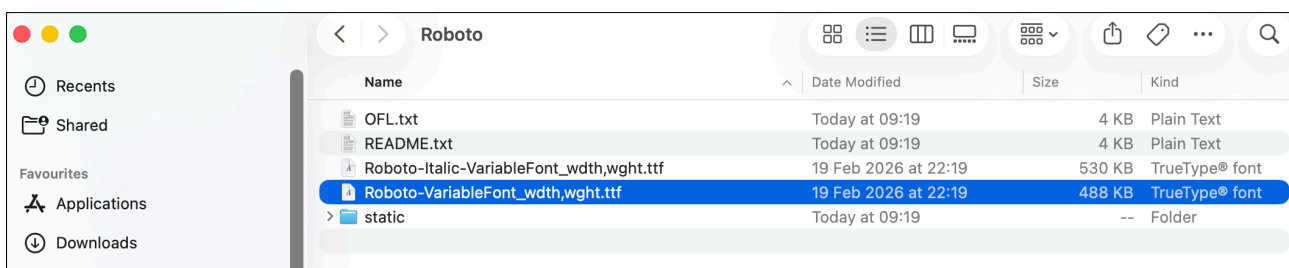
We get a few fonts from this family. Click on the download button.

Figure 3: download button



This will still give you numerous fonts. The one you want to find is the **Roboto-VariableFont_wdth,wght.ttf**. Depending on your computer (and operating system), it will be in a place where downloads are saved. You don't need to do anything with it except find where it is in a folder labelled **Roboto**.

Figure 4: downloaded font somewhere

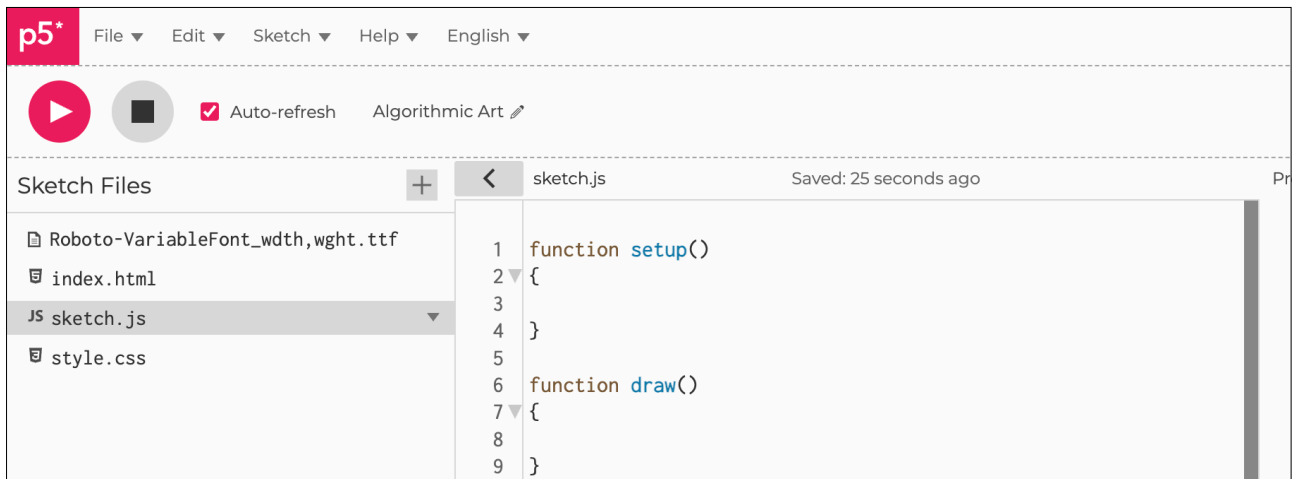




Sketch G2.1 upload font

This is something you should be familiar with. Go to the list of files for your sketch and click on the upload file tab. Click Browse and find the font file and upload it.

Figure G2.1





Sketch G2.2 display text

! This is our starting sketch.

It will print the text to the canvas. We use `async` and `await` to make sure we have uploaded the font before we use it.

```
let font

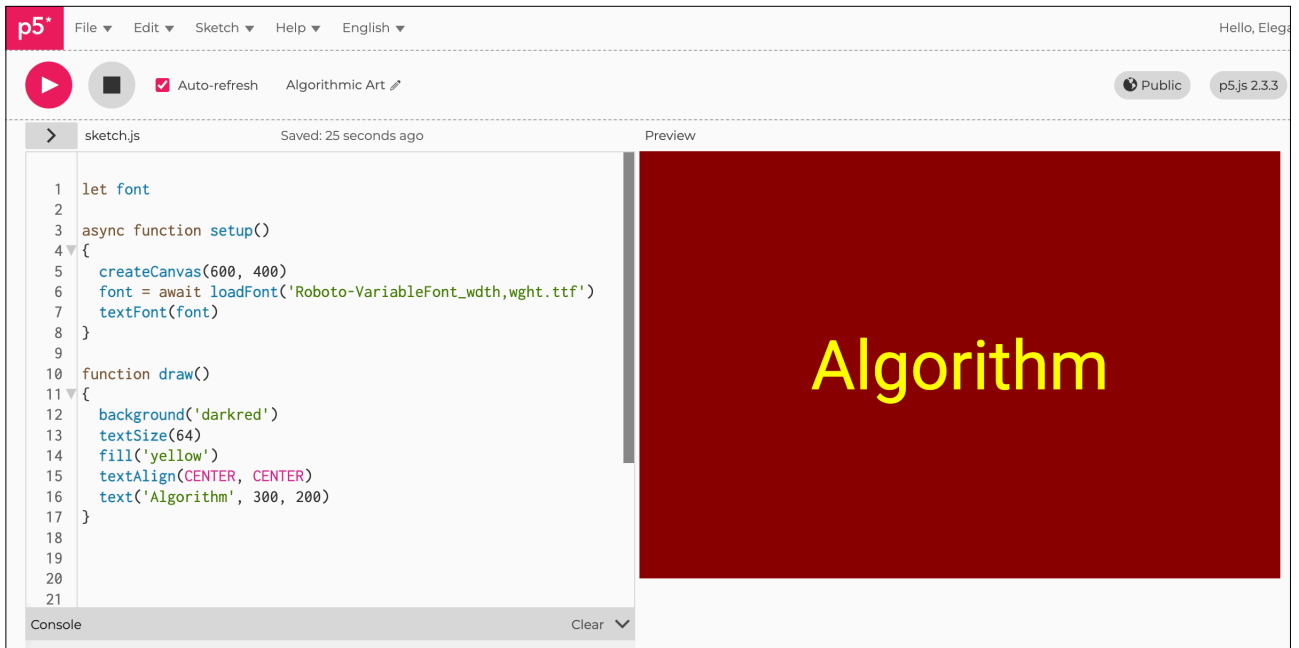
async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
}

function draw()
{
  background('darkred')
  textSize(64)
  fill('yellow')
  textAlign(CENTER, CENTER)
  text('Algorithm', 300, 200)
}
```

Notes

This will also serve to check that the font is working.

Figure G2.2





Sketch G2.3 breath of life

Now we can breathe some life into the text. We will get it to expand and contract as a sine wave. The sine wave runs between **-1** and **+1**; we will map this to the weights (thickness of the font).

```
let font
let wt
let angle = 0

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
}

function draw()
{
  background('darkred')
  wt = map(sin(angle), -1, 1, 100, 900)
  textWeight(wt)
  textSize(64)
  fill('yellow')
  textAlign(CENTER, CENTER)
  text('Algorithm', 300, 200)
  angle += 0.05
}
```

Notes

We increment the angle by **0.05**. This oscillates the weight from **100** to **900**, back to **100**, and so on.

Figure G2.3b





Sketch G2.4 dashed outline

We will create some dashed lines around the outline of the text. We will do this by getting the points that track the outline (not all of them, just spaced out) and then drawing a small rectangle in their place. We will also need to rotate these rectangles as they go round the corners of the letter shapes.

! This is our starting sketch

```
let font

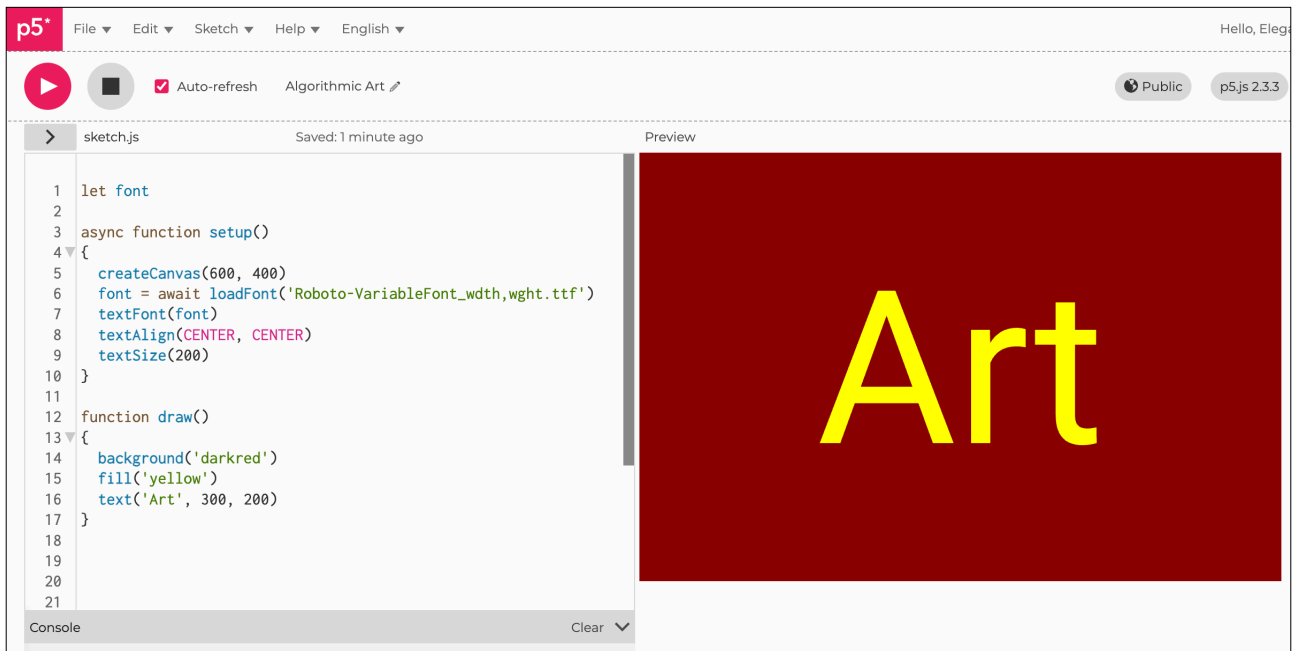
async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wdth,wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
}

function draw()
{
  background('darkred')
  fill('yellow')
  text('Art', 300, 200)
}
```

Notes

We will remove: `text('Art', 300, 200)` shortly. Notice also that it is much shorter text and larger for a good reason.

Figure G2.4





Sketch G2.5 what's the point

We can use some clever functions in p5.js version 2.x. We want to get the points in the text. We want to hold all the points in a variable called `allPoints`. We don't want all the points but just a sample of them; for that, we have the option `sampleFactor`, which we will set to `0.1`.

! Note that we have deleted the text line of code in the `draw()` function.

```
let font
let allPoints

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  allPoints = font.textToContours('Art', 300, 200, options)
}

function draw()
{
  background('darkred')
  fill('yellow')
}
```

Notes

We now have all (some) of the points. Each point is an array of four elements: `x`, `y`, `angle`, and `alpha`. We next need to get those data points and draw them.

Challenge

Change the `sampleFactor` (later on) to see the difference it makes.

Figure G2.5





Sketch G2.6 lines, letters and arrays

The length of the `allPoints` array is `7`. This is because the letters are made up of segments. For instance, if you had the letter `0`, it would count for two arrays, the inner and outer circle. The `A` has three, the `r` has two, and the `t` has two. This is just the way the font structures the letters.

We take each segment and pull out the points in that array and create another array called `letterPoints`. This will have a much larger number of elements. We can then use that data to draw the points of each letter, creating a dashed line effect. There are other effects you could create.

```
let font
let allPoints
let letterPoints

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wdth,wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  allPoints = font.textToContours('Art', 300, 200, options)
}

function draw()
{
  background('darkred')
  for (let i = 0; i < allPoints.length; i++)
  {
    letterPoints = allPoints[i]
    for (let j = 0; j < letterPoints.length; j++)
    {
      fill('yellow')
    }
  }
}
```

Notes

Just taking it one step at a time. Nothing to see yet except a sea of dark red.



Sketch G2.7 getting the point

We use another variable that can pull out and reference individual elements for each point, such as the **x**, **y**, and **angle** components, which we will need to draw the dashed line. This means it will follow the curve of the letter (if there is one). We call this variable **pt** (point for short).

```
let font
let allPoints
let letterPoints
let pt

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  allPoints = font.textToContours('Art', 300, 200, options)
}

function draw()
{
  background('darkred')
  for (let i = 0; i < allPoints.length; i++)
  {
    letterPoints = allPoints[i]
    for (let j = 0; j < letterPoints.length; j++)
    {
      pt = letterPoints[j]
      noStroke()
      fill('yellow')
      rectMode(CENTER)
      push()
      translate(pt.x, pt.y)
      rotate(pt.angle)
      rect(0, 0, 5, 2.5)
```

```

    pop()
  }
}
}

```

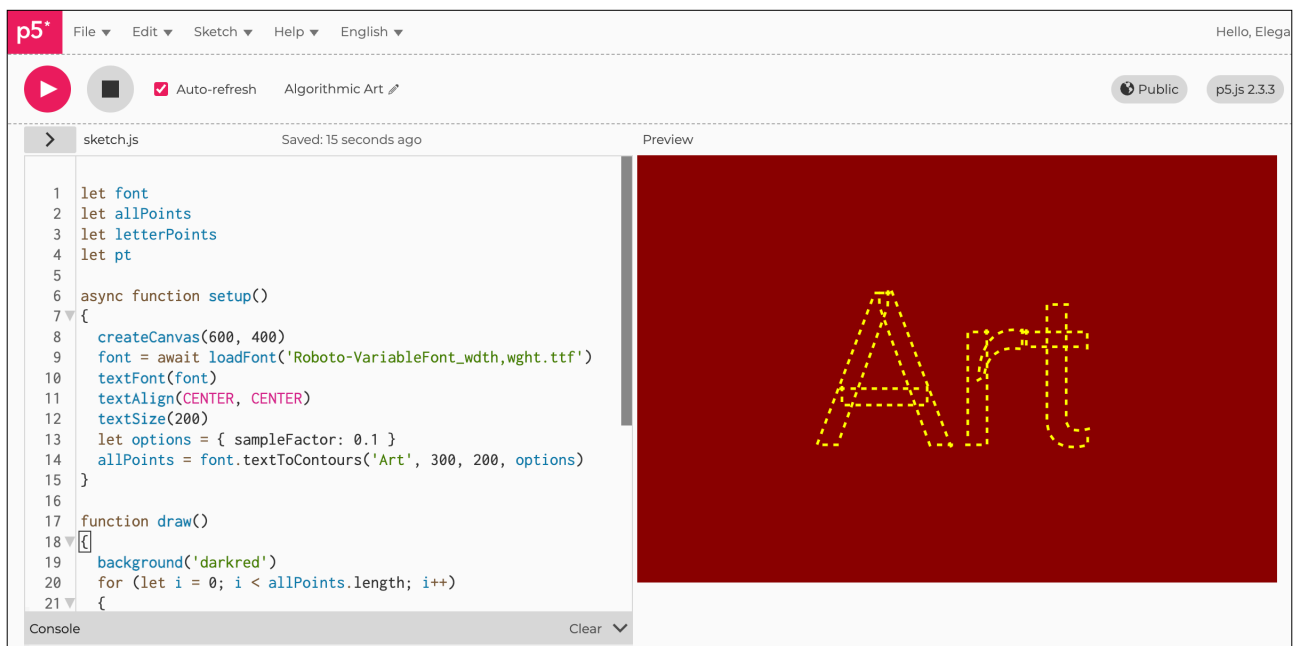
Notes

We have our points as dashes.

Challenge

Try other shapes

Figure G2.7





Sketch G2.8 alpha

Adding in one more feature, which you may or may not find interesting. It is the fourth element, the **alpha**. We have to use the **RGB** values for yellow before we can add the **alpha**.

```
let font
let allPoints
let letterPoints
let pt

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  allPoints = font.textToContours('Art', 300, 200, options)
}

function draw()
{
  background('darkred')
  for (let i = 0; i < allPoints.length; i++)
  {
    letterPoints = allPoints[i]
    for (let j = 0; j < letterPoints.length; j++)
    {
      pt = letterPoints[j]
      noStroke()
      fill(255, 255, 0, pt.alpha * 100)
      rectMode(CENTER)
      push()
      translate(pt.x, pt.y)
      rotate(pt.angle)
      rect(0, 0, 5, 2.5)
      pop()
    }
  }
}
```

```
}  
}  
}
```

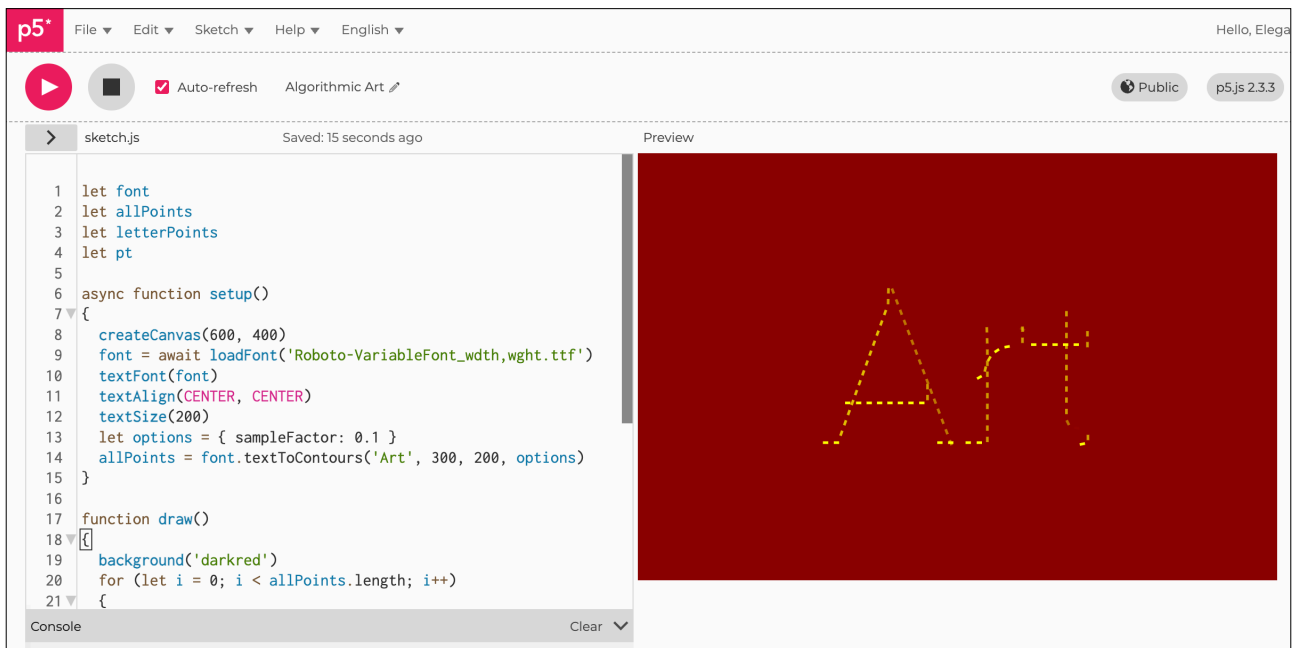
Notes

Shows that you can manipulate a shadow effect. We multiply by **100** to make it visible.

Challenges

1. Play around with the variables and see what you can make.
2. Could you use them to make it 3D (glasses)?

Figure G2.8





Sketch G2.9 3D text

Using the same font, we can extrude the text and make it 3D.

! Our starting sketch, main difference highlighted. I would recommend using version **2.0.5** for this example and for the next one. In the images, I have used the latest version, which does still work a bit.

```
let font

async function setup()
{
  createCanvas(600, 400, WEBGL)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { extrude: 50, sampleFactor: 0.1 }
}

function draw()
{
  background('darkred')
}
```

Notes

Just a red canvas.



Sketch G2.10 geometry

Instead of getting the contours of the text, now we get the geometry using the `textToModel()` function.

! You will get an error message (ignore it). If it bothers you, use a previous version of p5.js (still version 2.x), such as 2.0.5 (my recommendation).

```
let font
let textGeom

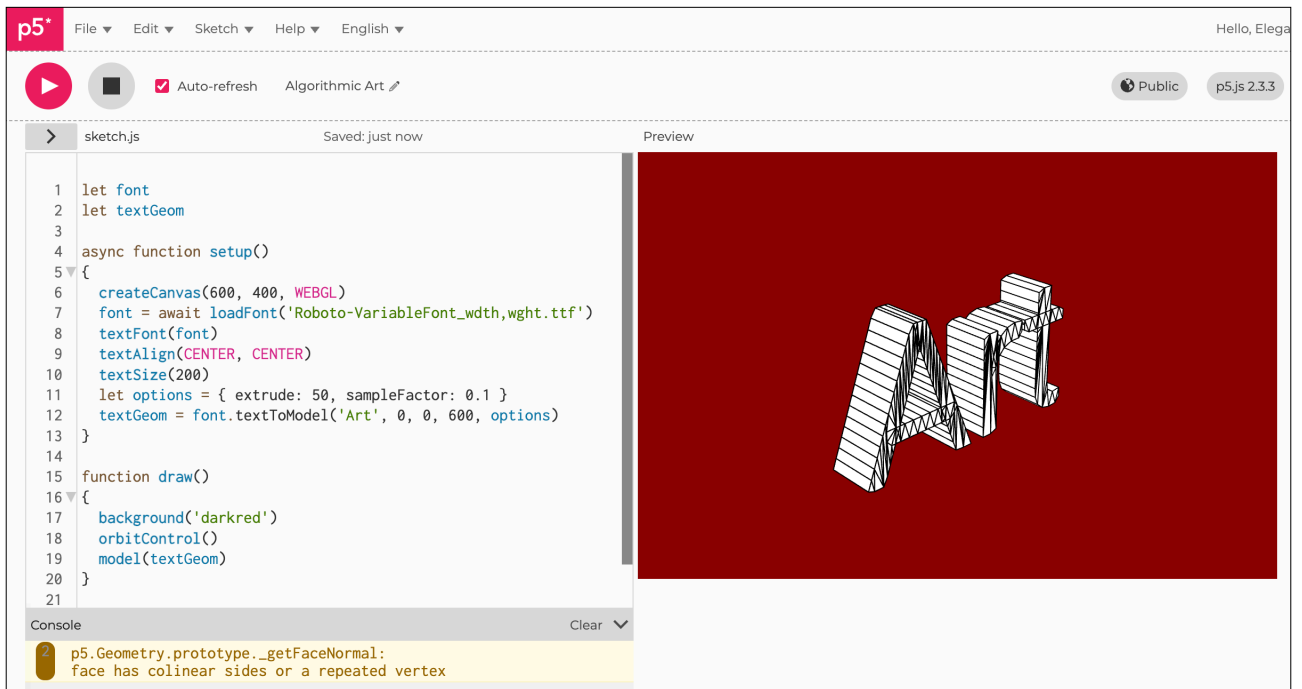
async function setup()
{
  createCanvas(600, 400, WEBGL)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { extrude: 50, sampleFactor: 0.1 }
  textGeom = font.textToModel('Art', 0, 0, 600, options)
}

function draw()
{
  background('darkred')
  orbitControl()
  model(textGeom)
}
```

Notes

We also add the orbital control function so we can rotate it with the mouse.

Figure G2.10





Sketch G2.11 let there be lights

Remove the lines and add some light.

```
let font
let textGeom

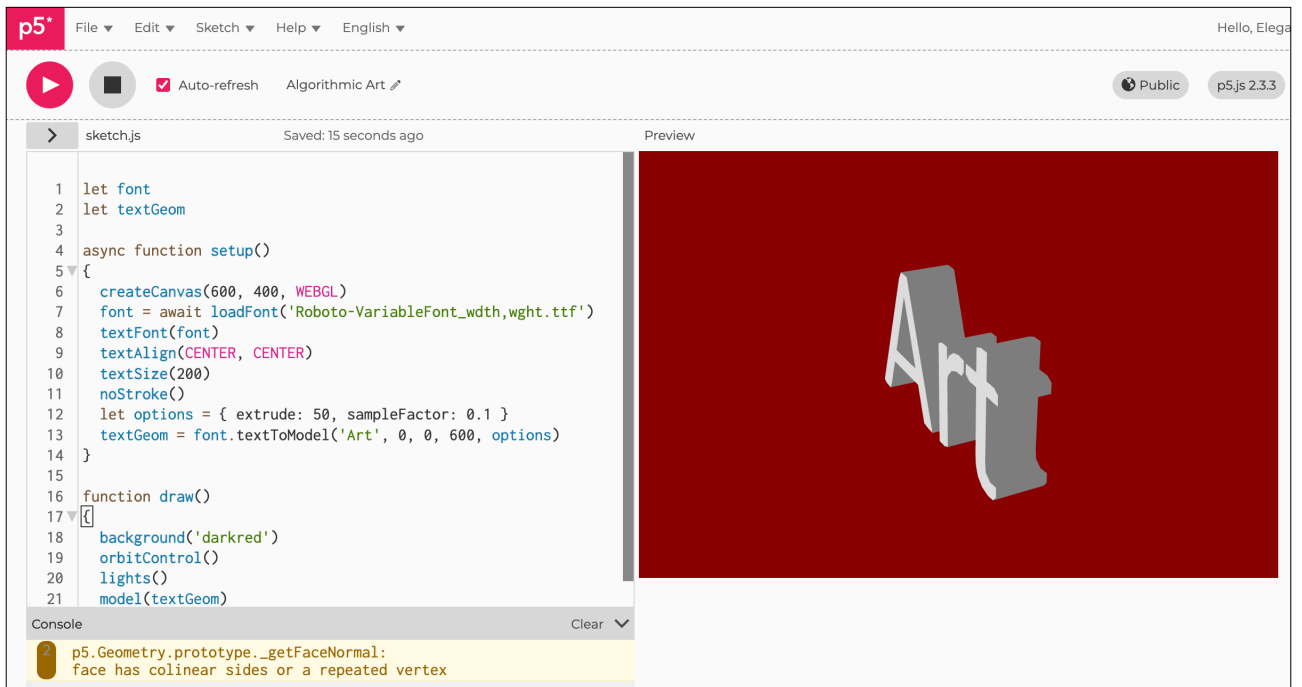
async function setup()
{
  createCanvas(600, 400, WEBGL)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  noStroke()
  let options = { extrude: 50, sampleFactor: 0.1 }
  textGeom = font.textToModel('Art', 0, 0, 600, options)
}

function draw()
{
  background('darkred')
  orbitControl()
  lights()
  model(textGeom)
}
```

Notes

You will still get the error message, more of a warning than a problem.

Figure G2.11





Sketch G2.12 steering behaviour

We can use the points on the contours of the text to move them from some random position back to where they belong. The effect is for the text to materialise as the points swarm in. You get the picture, I hope.

! Very important. Please change the latest version from **2.2.3** to version **2.0.5** for this next example. You will get lots of horrible warning messages, which will be some bug in the current (my latest) version. It may well be sorted by the time you are coding this.

! This is our starting sketch using the same font.

```
let font
let points

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wdth,wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  points = font.textToContours('Happy', 300, 200, options)
}

function draw()
{
  background('darkred')
}
```

Notes

Nothing to see yet



Sketch G2.13 pt points

Getting all the points (well, most of them). Just to make sure everything is working OK. Comment out the background in `draw()` and let's draw the points in setup for now. We will be changing how we draw them later.

```
let font
let points
let pt

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  points = font.textToContours('Happy', 300, 200, options)
  for (let i = 0; i < points.length; i++)
  {
    pt = points[i]
    for (let j = 0; j < pt.length; j++)
    {
      stroke(0)
      strokeWeight(5)
      point(pt[j].x, pt[j].y)
    }
  }
}

function draw()
{
  // background('darkred')
}
```

Notes

We can see the points for the word **Happy**.

Figure G2.13





Sketch G2.14 vehicles

We want these points to move, and so we will designate them as vehicles. For that, we need a **vehicle class**. This is the basics of vectors: movement of particles. There isn't room in this unit to cover each in detail. The **maxspeed** and **maxforce** variables are there to stop the particles going berserk.

Just in case you are unfamiliar, **pos** is position, **vel** is velocity, **acc** is acceleration. We have set the starting position randomly along the left-hand edge. Remove the commented-out lines of code. We will need to apply a force to the vehicles and eventually steer them to their allotted places.

```
let font
let points
let pt

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wdth,wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  points = font.textToContours('Happy', 300, 200, options)
  for (let i = 0; i < points.length; i++)
  {
    pt = points[i]
    for (let j = 0; j < pt.length; j++)
    {
      // stroke(0)
      // strokeWeight(5)
      // point(pt[j].x, pt[j].y)
    }
  }
}

function draw()
{
  background('darkred')
```

```

}

class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(0, random(height))
    this.vel = createVector(random(1), random(1))
    this.acc = createVector(0, 0)
    this.maxspeed = 10
    this.maxforce = 1
  }

  show()
  {
    stroke('yellow')
    strokeWeight(5)
    point(this.pos.x, this.pos.y)
  }

  move()
  {
    this.pos.add(this.vel)
    this.vel.add(this.acc)
    this.acc.set(0, 0)
  }

  applyForce(f)
  {
    this.acc.add(f)
  }
}

```

Notes

Remember to change the version to **2.0.5** (others do work as well). There is nothing to see yet.



Sketch G2.15 steer

We want to steer the vehicle towards the **target**, which is the final position. We want it to arrive at the target once it has been steered. We have also added these vehicles to an array of vehicles that we want to steer and move to their desired **target** spot.

```
let font
let points
let pt
let vehicle
let vehicles = []

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  points = font.textToContours('Happy', 300, 200, options)
  for (let i = 0; i < points.length; i++)
  {
    pt = points[i]
    for (let j = 0; j < pt.length; j++)
    {
      vehicle = new Vehicle(pt[j].x, pt[j].y)
      vehicles.push(vehicle)
    }
  }
}

function draw()
{
  background('darkred')
}

class Vehicle
{
```

```
constructor(x, y)
{
  this.pos = createVector(0, random(height))
  this.vel = createVector(random(1), random(1))
  this.acc = createVector(0, 0)
  this.maxspeed = 10
  this.maxforce = 1
}
```

```
show()
{
  stroke('yellow')
  strokeWeight(5)
  point(this.pos.x, this.pos.y)
}
```

```
move()
{
  this.pos.add(this.vel)
  this.vel.add(this.acc)
  this.acc.set(0, 0)
}
```

```
applyForce(f)
{
  this.acc.add(f)
}
```

```
arrive(target)
{
  let desired = p5.Vector.sub(target, this.pos)
  let d = desired.mag()
  if (d < 100)
  {
    let mg = map(d, 0, 100, 0, this.maxspeed)
    desired.setMag(mg)
  }
  else
```

```
{  
  desired.setMag(this.maxspeed)  
}  
let steer = p5.Vector.sub(desired, this.vel)  
steer.limit(this.maxforce)  
this.applyForce(steer)  
}  
}
```

Notes

Getting our ducks in a row. I will be exploring vectors and forces in more detail in a forthcoming tutorial series or workbooks about physics simulation. Nothing yet to see.



Sketch G2.16 good behaviour

We need to bring all this together and get the vehicles moving by calling them in the `draw()` function. The `behaviours()` function moves each vehicle to its allotted target position from its random starting point.

```
let font
let points
let pt
let vehicle
let vehicles = []

async function setup()
{
  createCanvas(600, 400)
  font = await loadFont('Roboto-VariableFont_wght.ttf')
  textFont(font)
  textAlign(CENTER, CENTER)
  textSize(200)
  let options = { sampleFactor: 0.1 }
  points = font.textToContours('Happy', 300, 200, options)
  for (let i = 0; i < points.length; i++)
  {
    pt = points[i]
    for (let j = 0; j < pt.length; j++)
    {
      vehicle = new Vehicle(pt[j].x, pt[j].y)
      vehicles.push(vehicle)
    }
  }
}

function draw()
{
  background('darkred')
  for (let i = 0; i < vehicles.length; i++)
  {
    let v = vehicles[i]
    v.behaviours()
  }
}
```

```

    v.move()
    v.show()
}

class Vehicle
{
    constructor(x, y)
    {
        this.pos = createVector(0, random(height))
        this.vel = createVector(random(1), random(1))
        this.acc = createVector(0, 0)
        this.target = createVector(x, y)
        this.maxspeed = 10
        this.maxforce = 1
    }

    show()
    {
        stroke('yellow')
        strokeWeight(5)
        point(this.pos.x, this.pos.y)
    }

    move()
    {
        this.pos.add(this.vel)
        this.vel.add(this.acc)
        this.acc.set(0, 0)
    }

    applyForce(f)
    {
        this.acc.add(f)
    }

    arrive(target)
    {

```

```

    let desired = p5.Vector.sub(target, this.pos)
    let d = desired.mag()
    if (d < 100)
    {
        let mg = map(d, 0, 100, 0, this.maxspeed)
        desired.setMag(mg)
    }
    else
    {
        desired.setMag(this.maxspeed)
    }
    let steer = p5.Vector.sub(desired, this.vel)
    steer.limit(this.maxforce)
    this.applyForce(steer)
}

```

```

behaviours()
{
    let arrive = this.arrive(this.target)
    this.applyForce(arrive)
}
}

```

Notes

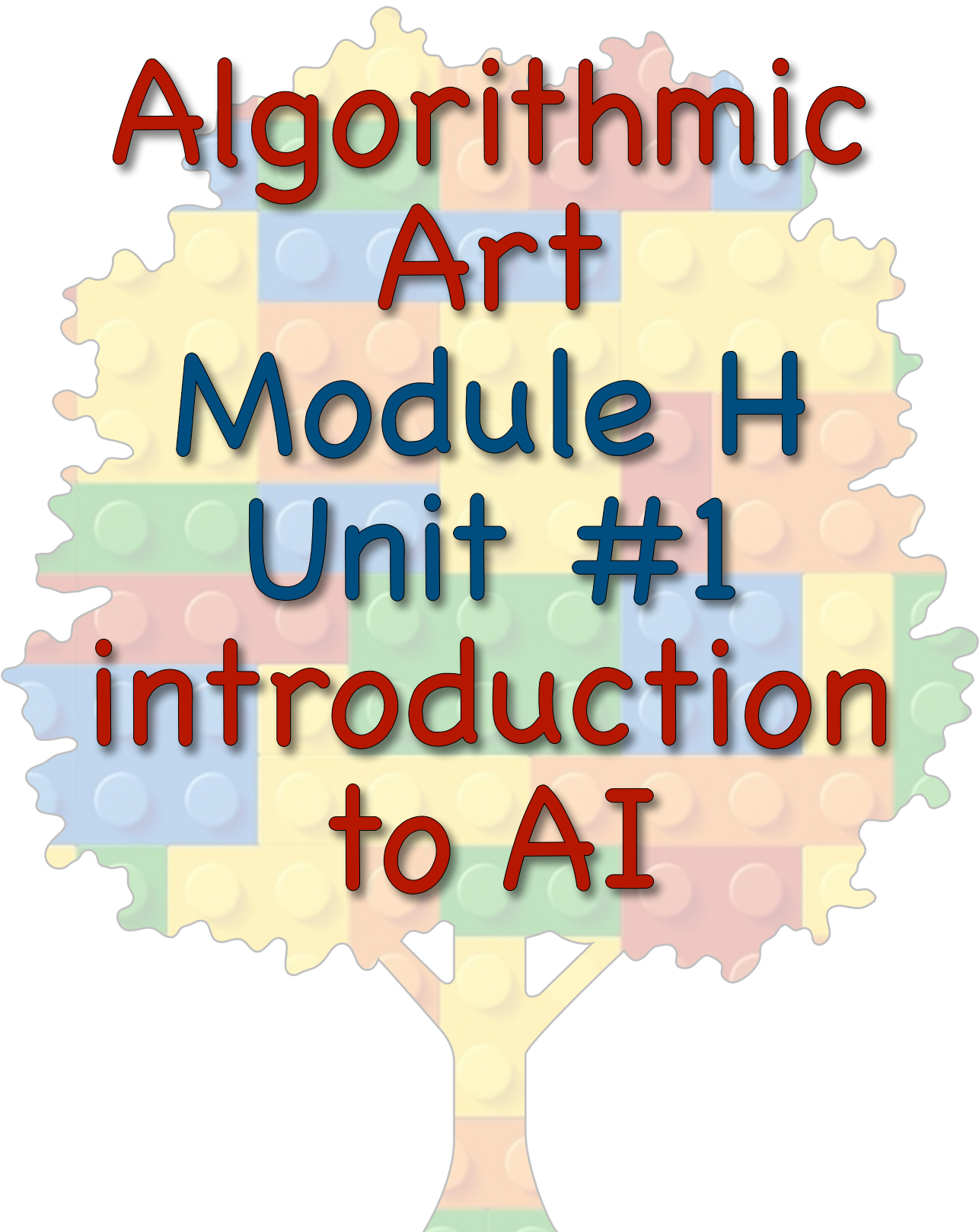
The vehicles (points) should start on the left-hand side edge and rush across the canvas to find their target spot.

Challenges

1. Try other random starting points for effect.
2. Try different max speeds and max force.
3. Add colours.
4. 3D glasses?

Figure G2.16





Algorithmic Art Module H Unit #1 introduction to AI



Module H Unit #1 introduction to AI

Before we jump into using AI to create art, it is worth spending a little bit of time understanding what we are using and where it came from. I have a whole series of workbooks on AI if you are interested after exploring these units.

If you think AI and arts, you may be thinking of Generative AI which uses images it has scraped off the internet to help you create an image from a prompt. That is certainly one use of AI but that is not what we are going to be doing in the next module.

The other workbooks I mentioned take you step by step through the process of building an AI model (virtually from scratch), adding your own data, and training it. There you will get a deeper understanding of how AI works and what tools you have to use and apply it.

This module is all about using pretrained models which have a specific application. They have already been trained on data. They are not LLMs (such as ChatGPT), there are no prompts. You will be coding it using p5.js through a library called ml5.js. Where the ML is short for Machine Learning and the 5 is a reference to p5.js.

Our AI is going to work through p5.js and will work in the browser. Nothing to download and certainly nothing to pay for (although you can donate). It is a simple line of code in the index.html file as you did with anaglyph.js. So it couldn't be easier in that respect. You don't have to understand the inner workings of AI to use this, but you do need to be able to code, and that is why we are doing it, one small step at a time.

This will open up many possibilities of an interactive approach to Algorithmic Art. Where you go with this is up to you. In the next few pages, I will offer a brief outline of what to expect.



What is ml5.js?

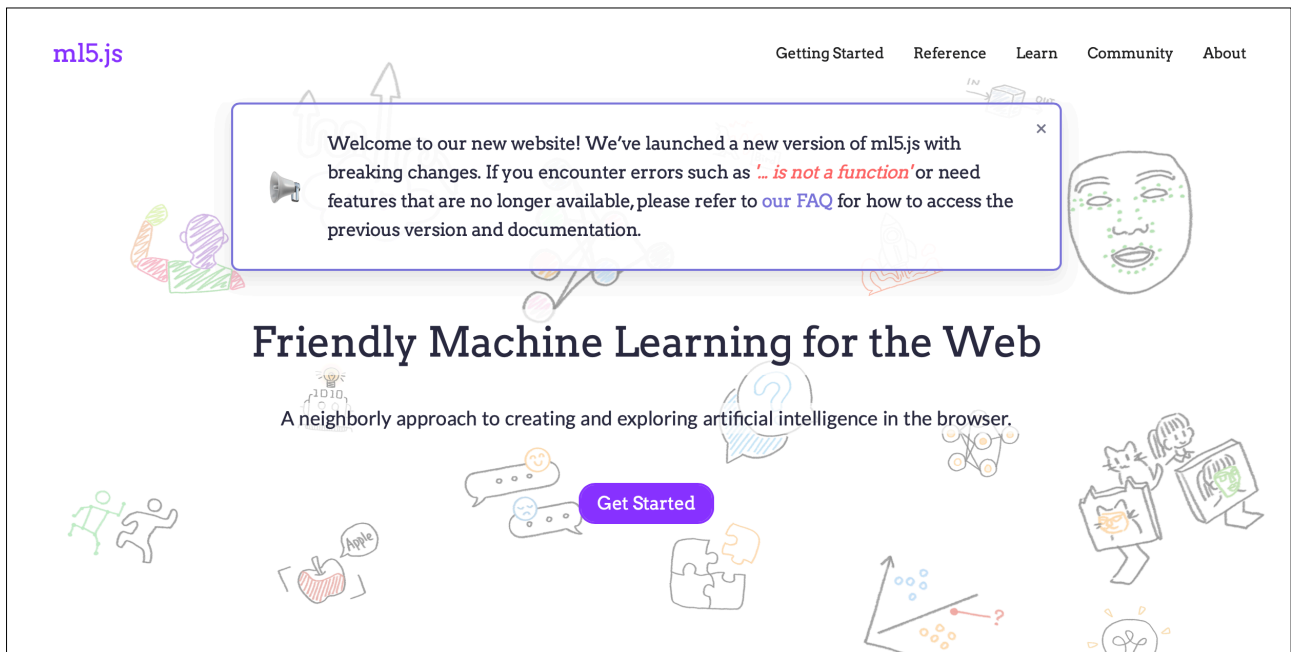
The ml5.js is a library that makes using Machine Learning (AI) accessible to mere mortals like you and me. It is built on **TensorFlow.js**, which in turn is built on **TensorFlow**, which is used in Machine Learning with the coding language of **Python**.

The Processing Foundation has a website where you can look at ml5.js in more detail. The ml5.js library has recently gone through a revision, and so there are, in effect, two websites. I will link you to the new one, which is pretty obvious.

The website is **www.ml5js.org**.

The home page looks like this (see Fig.1):

Figure 1: home page



But before you click on the Get Started button, just scroll down a bit and see what is there. Notice it gives a reference to the old version of ml5.js. It is only important if you find some old code that is based on that version, but we won't be at this stage.



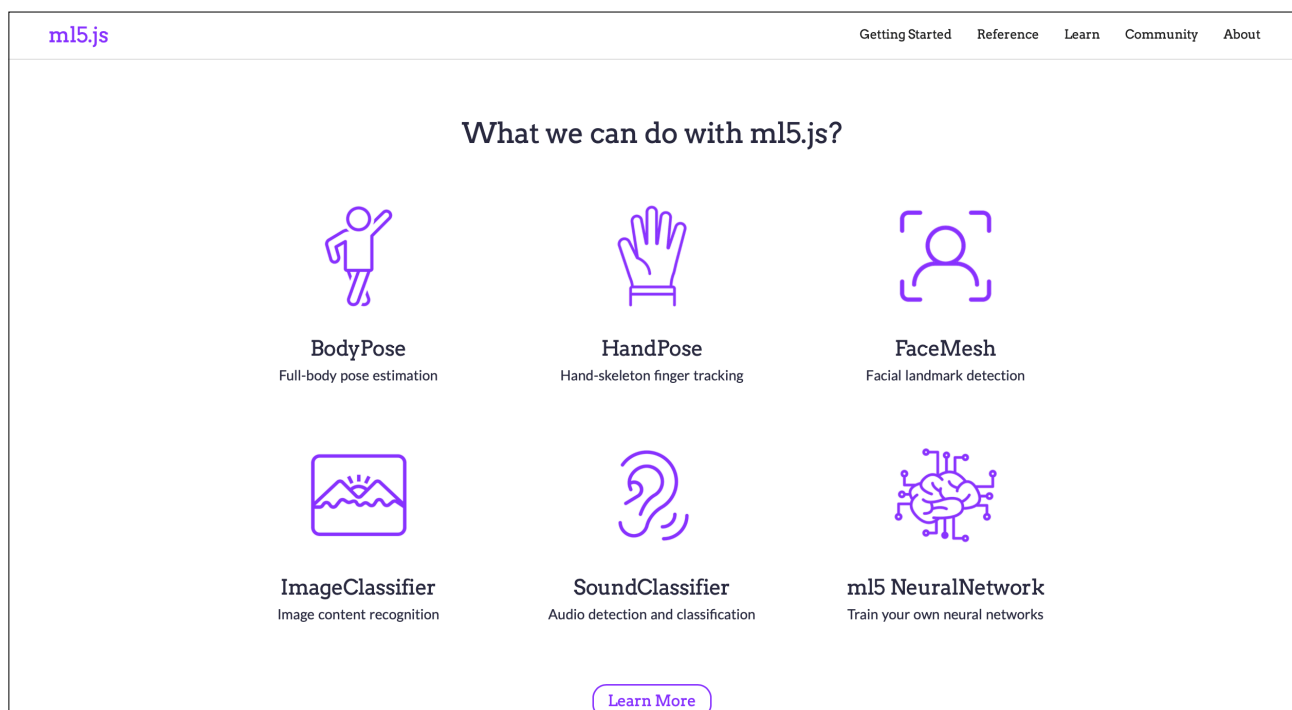
Key elements of ml5.js

As you scroll down, you can see the next window with some graphics (see Fig.2). Here are some examples of the pretrained models I mentioned:

BodyPose
HandPose
FaceMesh
ImageClassifier
SoundClassifier

There is a sixth one called ml5 NeuralNetwork, which is where you can build your own model, train it, save it, and use it. If you are interested, then head over to my AI workbooks. I will briefly go through each of these next.

Figure 2: models

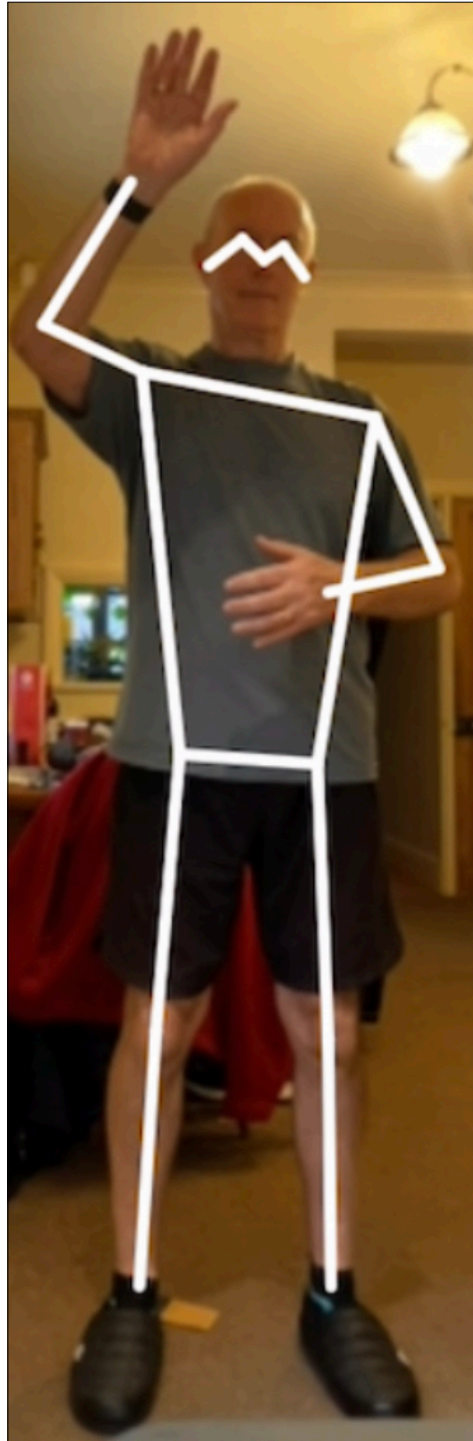




BodyPose

This requires a webcam of some sort pointed at you. When you run this pretrained model, it picks up the key body parts as points, for instance, shoulder, elbow, knee, ankle, and so on. This works in real time, and as you move around, it follows your movement.

Figure 3: BodyPose





HandPose

Here we have a HandPose pretrained model that maps out the major points on one or both hands. We use this model in our next unit as it is easiest to use sat down in front of your webcam.

Figure 4: HandPose

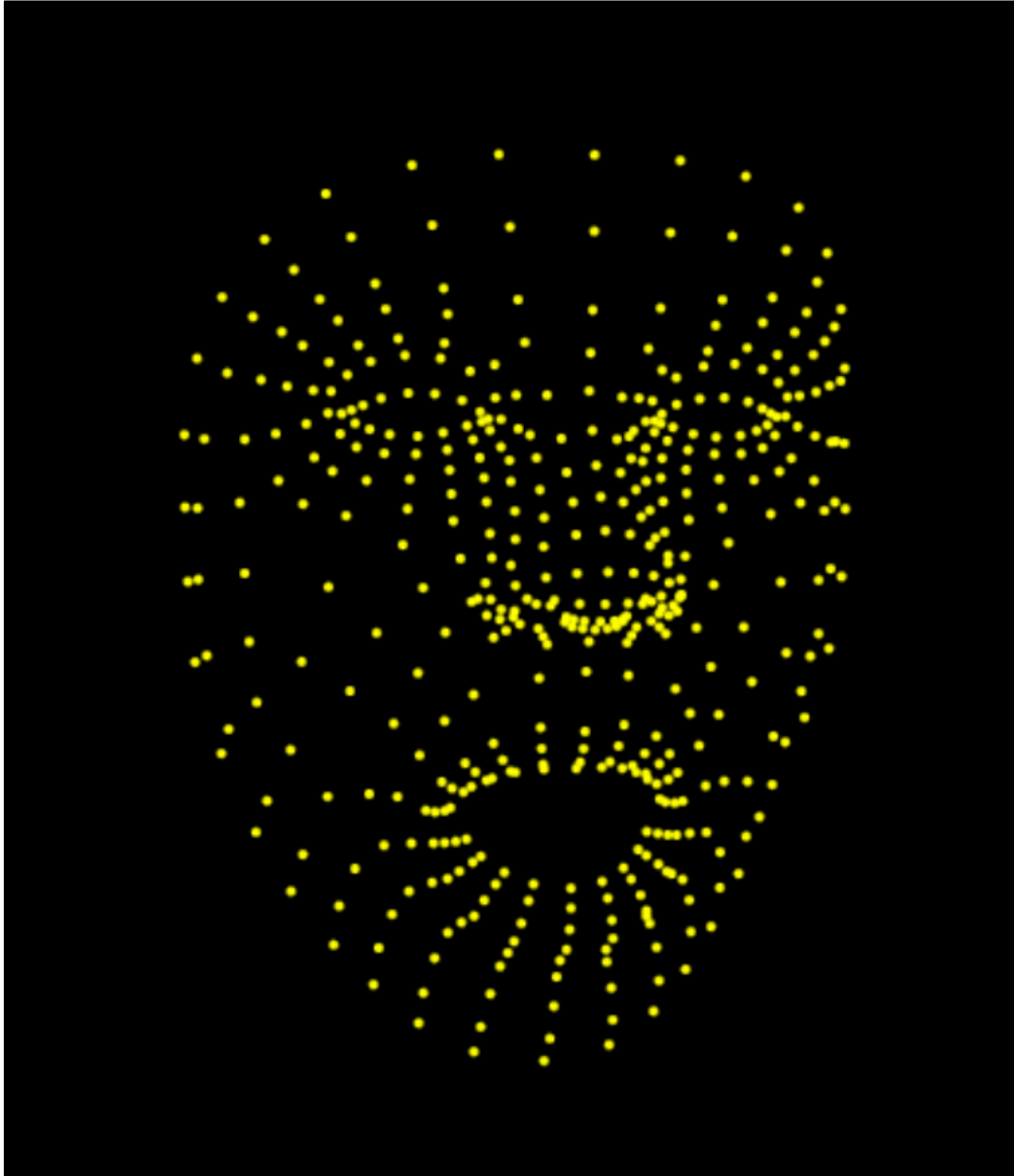




FaceMesh

This model maps lots of points on your face around the eyes, the nose, and mouth. You can use that to add on to your face a mask of your choosing.

Figure 5: FaceMesh





ImageClassifier

This can take an image or a video and classify what it can see. It gives you the confidence score for its prediction. It can only use data it has trained on, so you might find it provides some funny predictions. Here is the example on the website.

Figure 6: imageClassifier





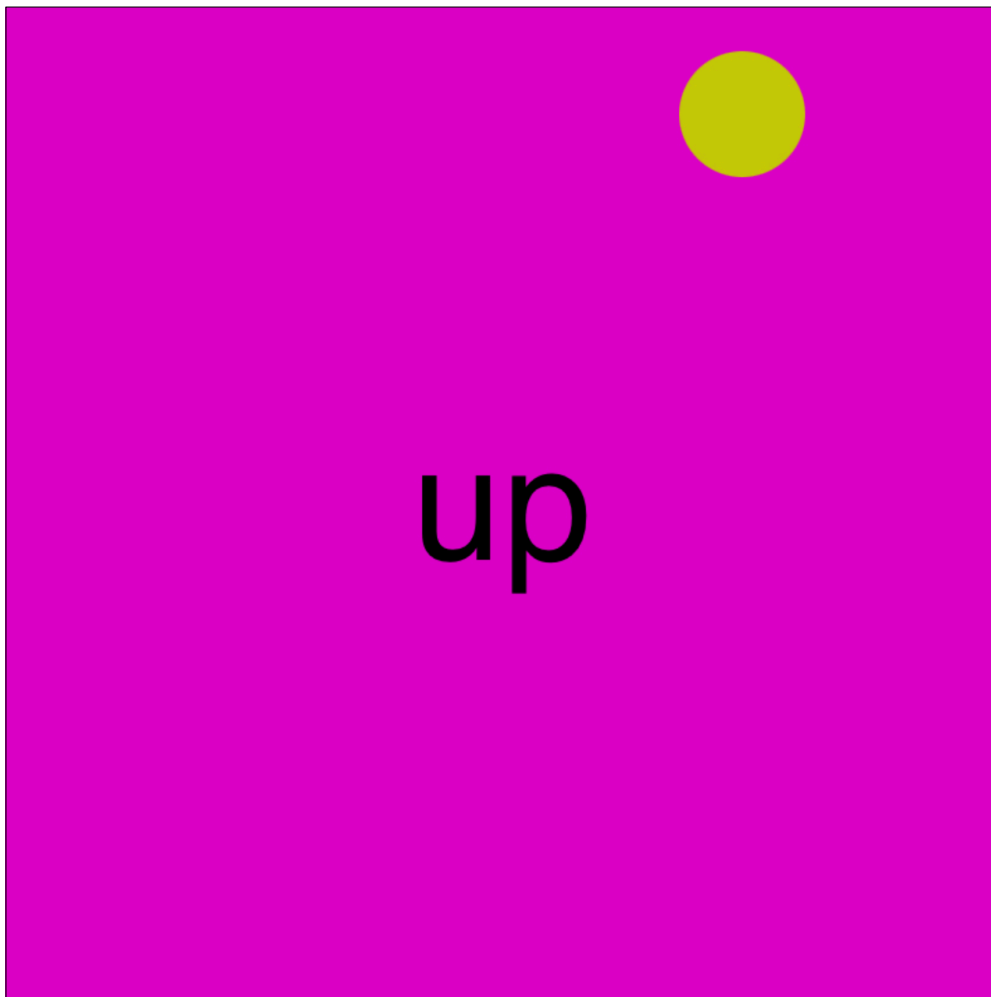
SoundClassifier

This model is trained on recognising spoken words. There are 18 words, and they are as follows:

Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine, Up, Down, Left, Right, Go, Stop, Yes, and No,

You can control or affect some sketch you have written as you speak into the microphone. It only knows those 18 words, but it's something to play around with. In Fig.7 I move a circle around the canvas with my voice, with mixed results.

Figure 7: soundClassifier





ml5 NeuralNetwork

This can be anything you have enough data for. In the AI workbooks, there are plenty of examples to work through to give you a feeling for how it works. They are simple examples that don't need a neural network but demonstrate how they work and how you can build one.

You won't be building one in these series of workbooks, but you may come across an idea to use one creatively, so I would recommend working through the AI workbooks as it may give you some ideas.

A neural network is nothing magical; it is simply an algorithm based on some relatively simple maths, much of which is to do with calculus and matrix multiplication. So if you did that at school, you are most of the way there and not far from being able to build your own from scratch.

A quick explanation is that you have inputs; it could be just some data, it could be an image of a dog or a cat. The data passes through the neural network, which has a series of weights (these are random) and then through a series of nodes which collect the data multiplied by the weights, to the output which has the correct result.

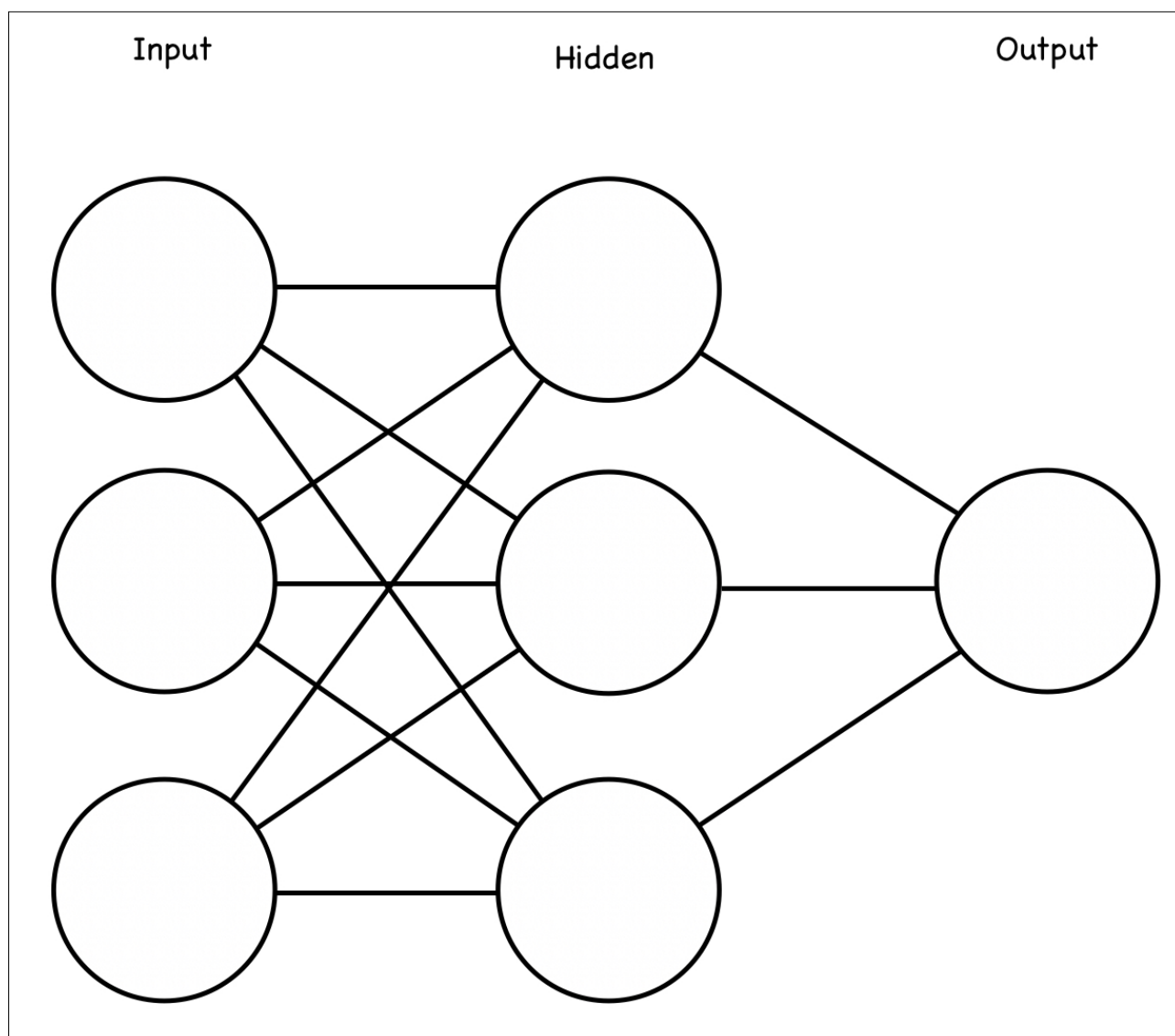
Because the neural network has effectively guessed what the correct result is, there will be an error. This error is passed back through the neural network, and those weights are tweaked slightly, and it has another go at guessing the correct result. It repeats this until there is no or very little error. This is how a model is trained.

Fig.8 shows a simple diagram of what is going on inside a simple neural network. There are lots of layers and nodes (or neurons), and all it is doing is lots and lots of simple calculations. When you use the Large Language Models such as ChatGPT, they have billions of these nodes (neurons), hence why they are housed in data centres.

We use much smaller ones that we can run in our browser. The simplest is called a perceptron, which is effectively one node (neuron) with, say, two inputs and one output. This is such a vast topic, and for the most part, you do not need to know how it works in the same way you do not need to know how the internal workings of the engine work in your car, but you do need to know when and where to put water, fuel, electricity, or air into the car to keep it moving.

For now, you can just use it. That is complex enough. Enjoy and have a go at seeing what you can create.

Figure 8: diagram of a simple neural network



Algorithmic Art

Module H

Unit #2

pretrained
handPose

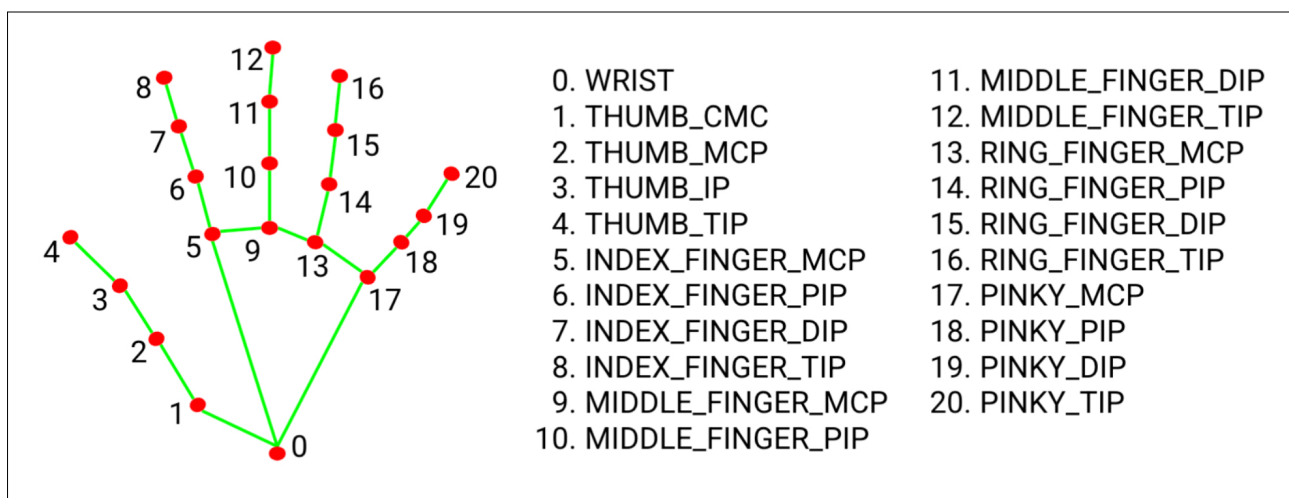


Module H Unit #2 handPose pretrained AI

The **handPose** model is another pre-trained model that is part of the ml5.js library. We can get all the data points of your hands and use some of them to control the size and position of a circle.

The **handPose** model can detect **21** keypoints on a hand (see **figure 1** below). It can detect both hands at the same time and determine the left from the right hand. Also, the keypoints are given as **3D** co-ordinates, but we will focus on the **2D** co-ordinates for now.

Figure 1: all 21 keypoints for handPose





The index.html file

Adding the ml5.js file to the index.html file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch H2.1 our starting sketch

The process is similar to the previous unit, but we will still go through it step by step.

```
function setup()
{
  createCanvas(640, 480)
}

function draw()
{
  background(220)
}
```




Sketch H2.2 video

Getting the video feed from the webcam.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO)
}

function draw()
{
  background(220)
}
```

Notes

You should get a grey canvas and underneath it, a larger video image of you.



Sketch H2.3 putting you on the canvas

We hide the video stream and use the `image()` function. Notice that we include the width and height, not entirely necessary in this case, but if you wanted to match the canvas dimensions, this is the way to go.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

Now you should have a video image of yourself where the canvas was, but if you lift up your hand, it is not mirrored.



Sketch H2.4 flipping the video

We can flip the video using a function in p5.js.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

You will now have a mirror image from the camera.



Sketch H2.5 the handPose model

There is a pre-trained model within ml5.js, and all we need to do is call it. We will preload it before we try to use it; otherwise, we may get error messages.

```
let video
let handPose

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

Nothing will happen when you hold your hand up to the camera because we haven't connected the two together.



Sketch H2.6 start detecting

We use a function called `detectStart()`, and this will connect the model to the video feed.

```
let video
let handPose

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

! If you run this now, you will get an error message. It will tell you that you need a callback function. This callback function will hold the data.



Sketch H2.7 callback

Let's create a callback function called `gotHands()` and `console.log()` the **results** as we did before.

```
let video
let handPose

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}

function gotHands(results)
{
  console.log(results)
}
```

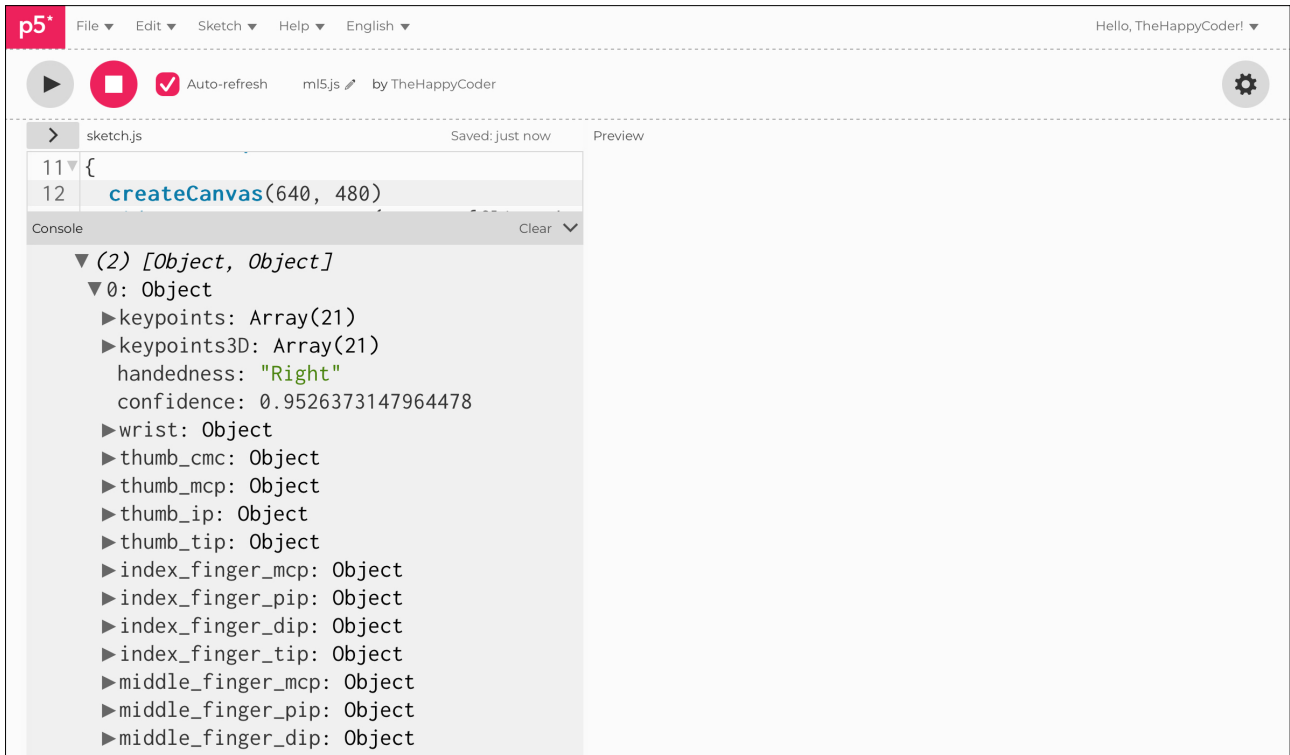
Notes

You will get an empty array when it doesn't see any hands, one array object when you hold up one hand, and two array objects when it can see two hands. If you pause the programme, you can examine the content of the arrays. You will notice how many points there are and the labelling of your fingers, plus other parts of your hand. You have **21 keypoints** for each hand.

Challenge

Highly recommend that you explore the results in the console.

Figure H2.7





Sketch H2.8 an array of hands

We need to store this data and access the array of your hands.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}

function gotHands(results)
{
  hands = results
}
```

Notes

We create an empty array to put the data in and fill it with the results in the callback function.



Sketch H2.9 the keypoints

We want to cycle through all the **keypoints** in the hands array and draw them. For this, we use a nested loop.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  for (let i = 0; i < hands.length; i++)
  {
    for (let j = 0; j < hands[i].keypoints.length; j++)
    {
      let keypoint = hands[i].keypoints[j]
      fill('yellow')
      circle(keypoint.x, keypoint.y, 10)
    }
  }
}

function gotHands(results)
{
  hands = results
}
```

Notes

Notice that the yellow dots are not mirrored!



Sketch H2.10 mirroring the yellow dots

We can add the **flipped** is **true**. Also, we will remove the video of you and replace it with a dark red background.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

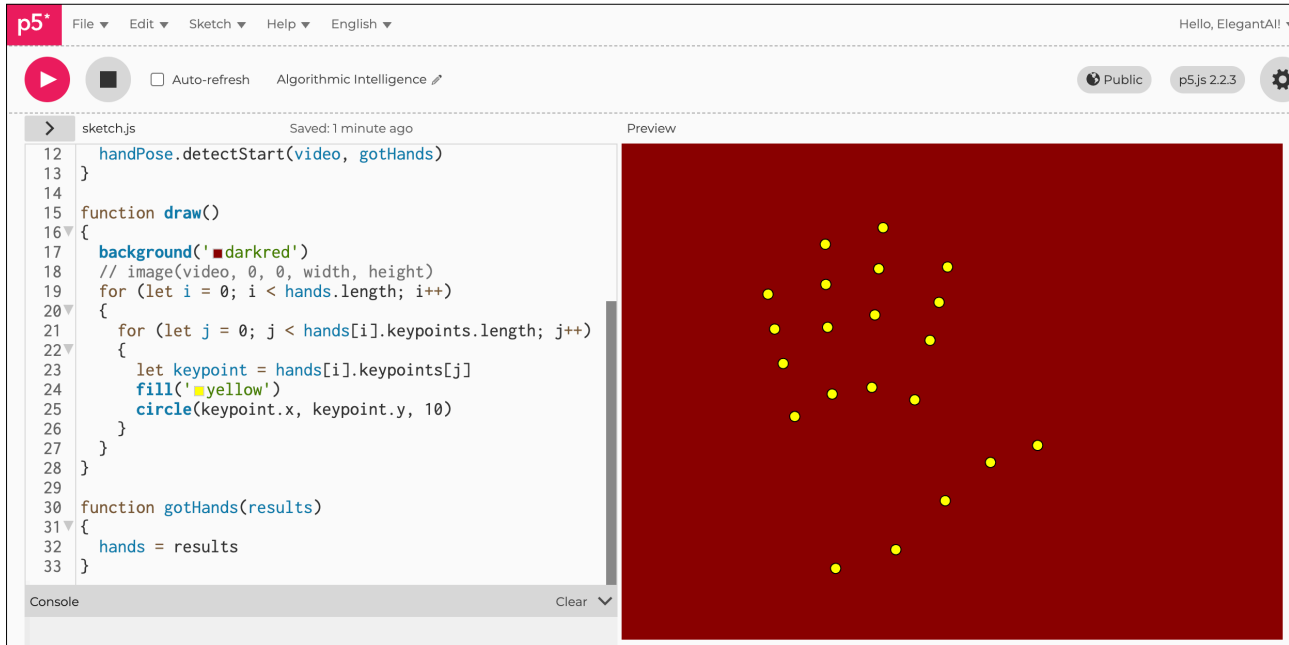
function draw()
{
  background('darkred')
  // image(video, 0, 0, width, height)
  for (let i = 0; i < hands.length; i++)
  {
    for (let j = 0; j < hands[i].keypoints.length; j++)
    {
      let keypoint = hands[i].keypoints[j]
      fill('yellow')
      circle(keypoint.x, keypoint.y, 10)
    }
  }
}

function gotHands(results)
{
  hands = results
}
```

Notes

The points are mirrored on both hands.

Figure H2.10





Sketch H2.11 reference keypoints

Instead of points, let's get the index reference for each point, because then we can use specific **keypoints**.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  // image(video, 0, 0, width, height)
  for (let i = 0; i < hands.length; i++)
  {
    for (let j = 0; j < hands[i].keypoints.length; j++)
    {
      let keypoint = hands[i].keypoints[j]
      fill('yellow')
      // circle(keypoint.x, keypoint.y, 10)
      text(i, keypoint.x -10, keypoint.y)
      text(j, keypoint.x, keypoint.y)
    }
  }
}

function gotHands(results)
{
  hands = results
}
```

```
}
```

Notes

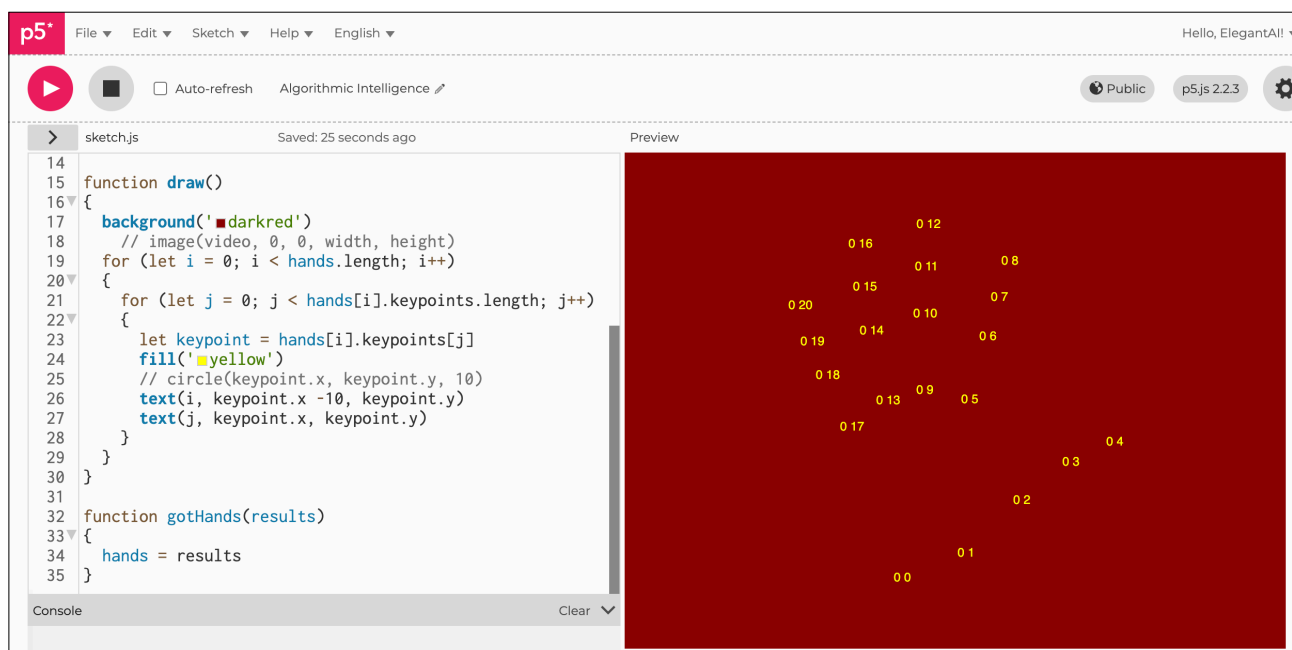
You will notice that when you hold up two hands, you get a **1** for the left hand followed by the reference number on that hand, and a **0** for the right hand followed by the corresponding reference number on the other hand.

We want the index finger and thumb (below are the reference numbers):

Index finger: keypoint **8**

Thumb: keypoint **4**

Figure H2.11





Sketch H2.12 index and thumb

Just to prove we have the right keypoint reference, we can write the words onto the canvas. We are only interested in one hand at a time. We can remove the commented-out nested loops.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  // image(video, 0, 0, width, height)
  if (hands.length > 0)
  {
    fill('white')
    noStroke()
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
    text('index', index.x, index.y)
    text('thumb', thumb.x, thumb.y)
  }
  // for (let i = 0; i < hands.length; i++)
  // {
  //   for (let j = 0; j < hands[i].keypoints.length; j++)
  //   {
  //     let keypoint = hands[i].keypoints[j]
  //     fill('yellow')
```

```
//      // circle(keypoint.x, keypoint.y, 10)
//      text(i, keypoint.x -10, keypoint.y)
//      text(j, keypoint.x, keypoint.y)
//    }
//  }
```

```
}
```

```
function gotHands(results)
```

```
{
```

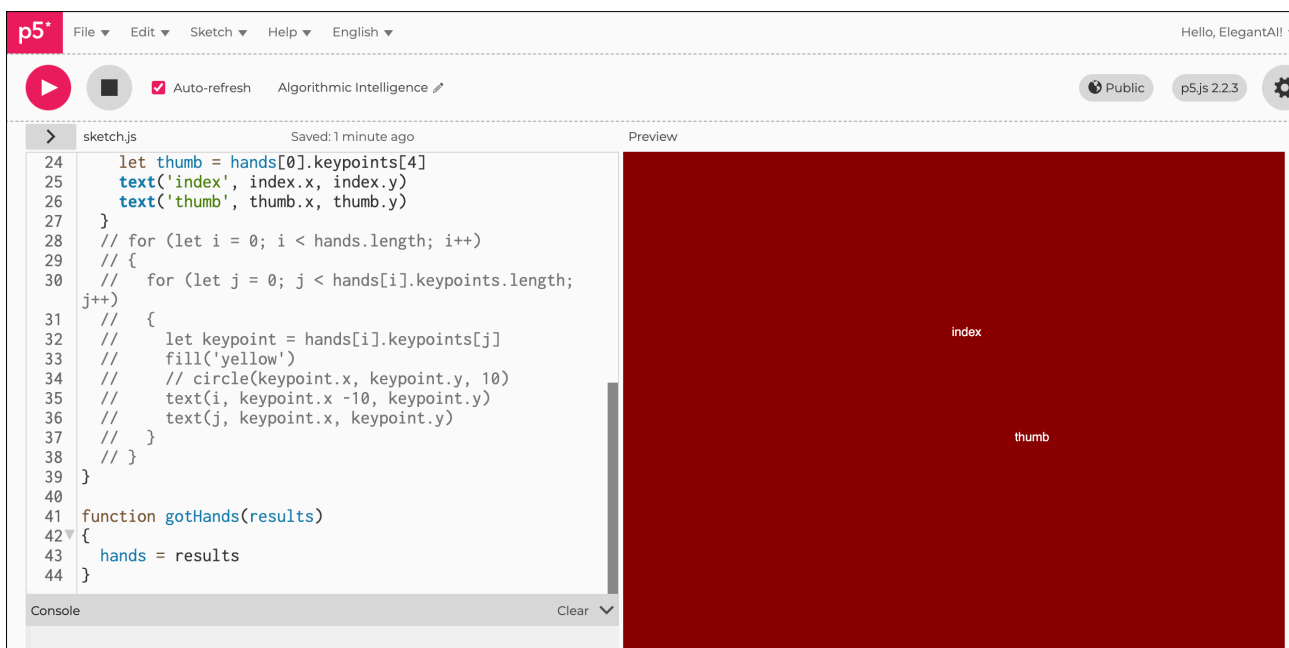
```
  hands = results
```

```
}
```

Notes

You should have the index finger and thumb correctly identified using either hand.

Figure H2.12





Sketch H2.13 circle size

Instead of writing text, we will draw a circle that changes depending on the distance (**d**) between those two fingers.

! We have completely removed the nested loops as we don't need that information anymore.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  // image(video, 0, 0, width, height)
  if (hands.length > 0)
  {
    // fill('black')
    noFill()
    strokeWeight(5)
    stroke('yellow')
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
    let d = dist(index.x, index.y, thumb.x, thumb.y)
    circle(index.x, index.y, d)
    // text('index', index.x, index.y)
    // text('thumb', thumb.x, thumb.y)
  }
}
```

```
function gotHands(results)
{
  hands = results
}
```

Notes

What you get is the circle changing size as you open and close your thumb and index finger. However, I have left the centre on the index finger for now. We want it halfway between the index finger and the thumb. You may not really be able to see it with the image commented out, but the circle is entered around the index finger.

Challenges

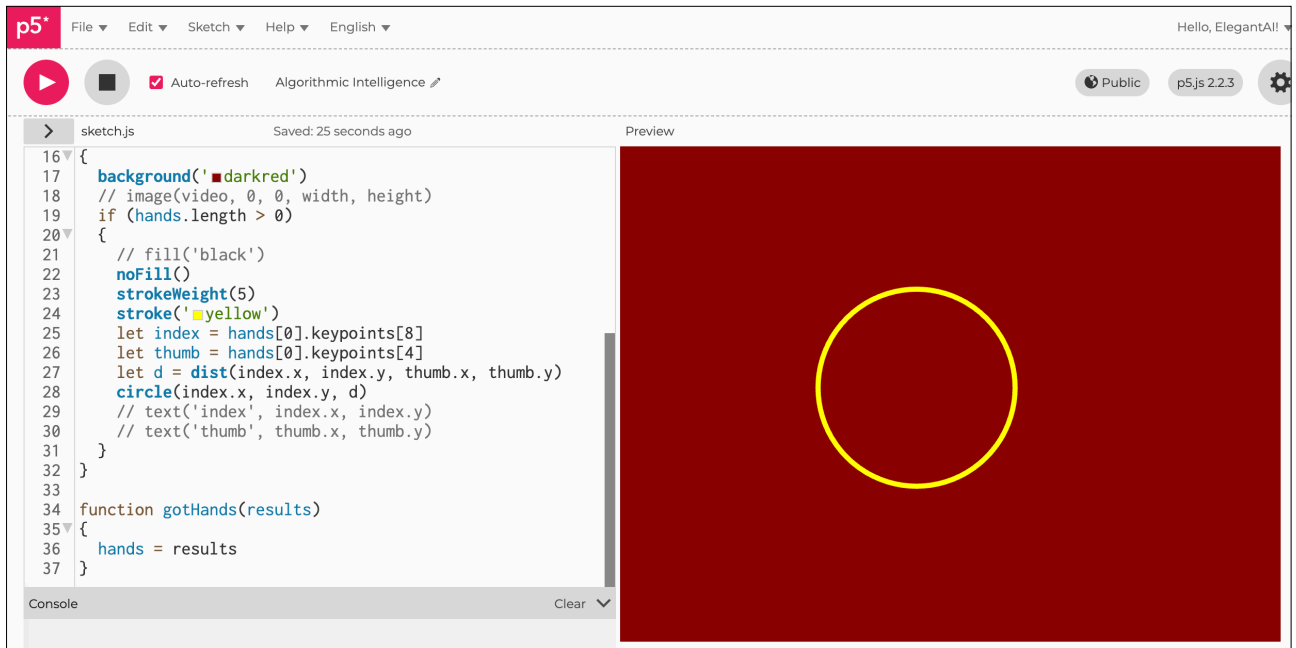
1. Uncomment the line of code to show the video feed.
2. You could try other fingers.

Code Explanation

```
let d = dist(index.x, index.y,
thumb.x, thumb.y)
```

Measures the distance between the x, y co-ordinates of the index finger with the x, y co-ordinates of the thumb

Figure H2.13





Sketch H2.14 adding some coloured color

Now we have the centre of the circle between our thumb and index finger.

```
let video
let handPose
let hands = []
let colour

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  colour = color(255, 255, 0)
}

function draw()
{
  background(220)
  if (hands.length > 0)
  {
    noFill()
    strokeWeight(5)
    stroke(colour)
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
    let d = dist(index.x, index.y, thumb.x, thumb.y)
    circle(index.x, index.y, d)
  }
}

function gotHands(results)
{
  hands = results
}
```

Notes

Just make a few tweaks, which doesn't do much as of yet. The colour variable holds the `color()` function values. We can change these at will.

Challenge

How about drawing a square, rectangle, or ellipse?

Figure H2.14





Sketch H2.15 checking the distance

Now that we have a distance between the thumb and index finger, we can change the colour of the circle to a random colour. Every time you close your index finger and thumb, the circle should change colour.

```
let video
let handPose
let hands = []
let colour

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  colour = color(255, 255, 0)
}

function draw()
{
  background(220)
  if (hands.length > 0)
  {
    noFill()
    strokeWeight(5)
    stroke(colour)
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
    let d = dist(index.x, index.y, thumb.x, thumb.y)
    circle(index.x, index.y, d)

    if (d < 25)
    {
      colour = color(random(255), random(255), random(255))
    }
  }
  else
```

```

    {
      stroke(colour)
    }
  }
}

function gotHands(results)
{
  hands = results
}

```

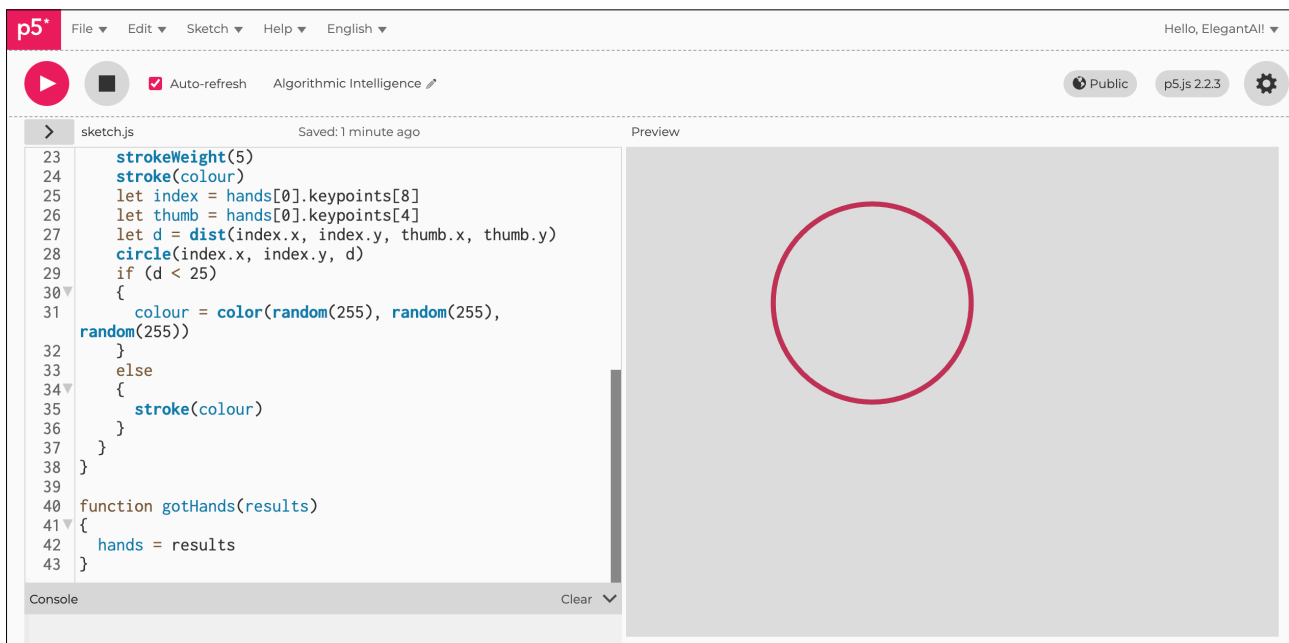
Notes

The reason why we have an `else{ }` condition is so that it settles on one random colour; otherwise, it would cycle through all the random colours, flickering continuously.

Challenge

Change the circle fill colour.

Figure H2.15





Sketch H2.16 slimmed down sketch

! Get rid of everything except what is left below and change the highlighted code.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  if (hands.length > 0)
  {
    noFill()
    strokeWeight(2)
    stroke('yellow')
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
  }
}

function gotHands(results)
{
  hands = results
}
```

Notes

We will see nothing yet. Just check for errors.



Sketch H2.17 many hands make light work

We need to make sure we detect both hands for this. One hand is `hands[0]` and the other hand is `hands[1]`.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  if (hands[0] && hands[1])
  {
    noFill()
    strokeWeight(2)
    stroke('yellow')
    let indexL = hands[0].keypoints[8]
    let thumbL = hands[0].keypoints[4]
    let indexR = hands[1].keypoints[8]
    let thumbR = hands[1].keypoints[4]
  }
}

function gotHands(results)
{
  hands = results
}
```

Notes

We are getting the thumb and index finger of each hand. Still nothing to see here.

Code Explanation

```
if (hands[0] && hands[1])
```

Checks to see if it can see both hands



Sketch H2.18 the vertex

Now we can draw the vertex points for our irregular shape. When it detects both hands, it joins up all the points. Move your fingers around to see the effect. Just keep both hands in view.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  if (hands[0] && hands[1])
  {
    noFill()
    strokeWeight(2)
    stroke('yellow')
    let indexL = hands[0].keypoints[8]
    let thumbL = hands[0].keypoints[4]
    let indexR = hands[1].keypoints[8]
    let thumbR = hands[1].keypoints[4]

    beginShape()
    vertex(indexL.x, indexL.y)
    vertex(indexR.x, indexR.y)
    vertex(thumbR.x, thumbR.y)
    vertex(thumbL.x, thumbL.y)
    endShape(CLOSE)
  }
}
```

```

}

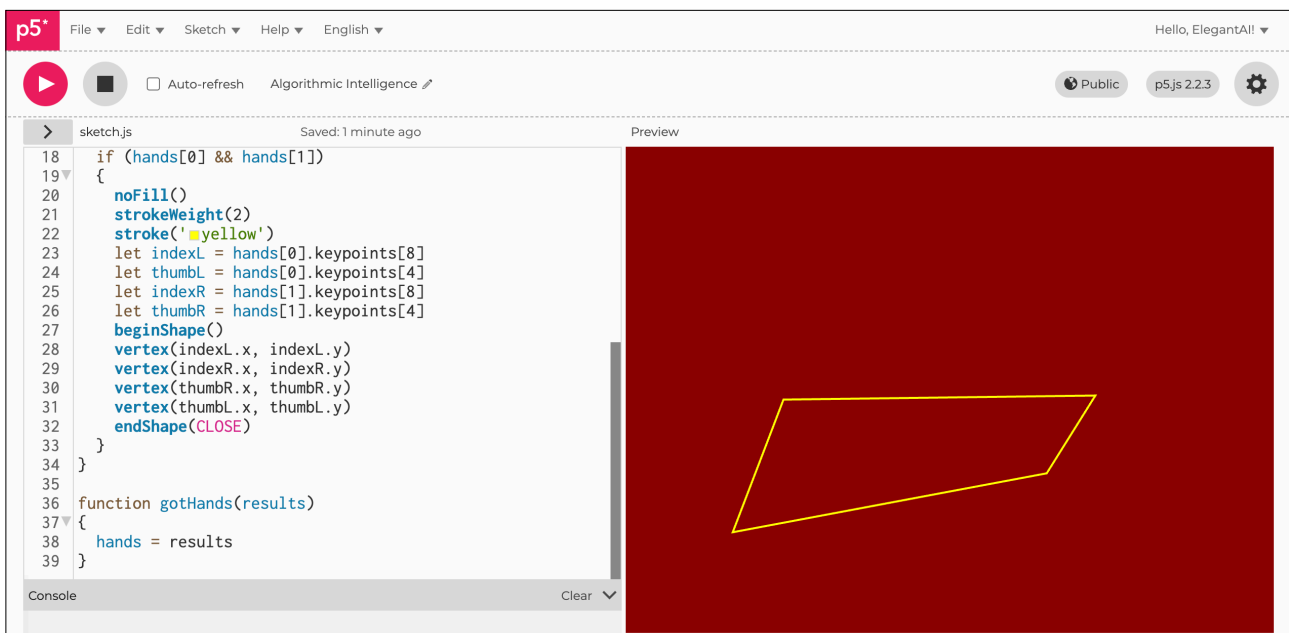
function gotHands(results)
{
  hands = results
}

```

Challenges

1. What shapes could you create with all the fingers?
2. Change the fill colour when you bring your fingers together.

Figure H2.18



Algorithmic Art

Module H

Unit #3

pretrained
handPose
emoji



Module H Unit #3 handPose emoji

This is using the same pre-trained model, but instead of points, we want to emit an emoji as a particle. This makes it a little bit more complicated, but not much. We are going to create particles, which we have covered briefly before, but I will be covering them in more depth in the Physics Simulation series of workbooks.



Sketch H3.1 index.html

As before, make sure you have the ml5.js library line of code included. You will also note that we (at the time of writing) are on version **2.2.3** of p5.js.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.3/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch H3.2 starting sketch

Pulling the key code from the previous unit, we have the starting point. If you want to see (check it is working), then comment out the background and uncomment the `image()` function.

```
let video
let handPose
let hands = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
}

function gotHands(results)
{
  hands = results
}
```




Sketch H3.3 hand of hands

We want to get all the data points of either (or both) hands. To do that, we create a separate variable called `hand` and copy the array called `hands`, which is full of all the data points (**keypoints**). We start with our conditional expression to see if there are any **keypoints** detected. Then, we loop through all the **keypoints** in the `hand` array.

```
let video
let handPose
let hands = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
  if (hands.length > 0)
  {
    for (let hand of hands)
    {
      for (let i = 0; i < hand.keypoints.length; i++)
      {
        let keypoint = hand.keypoints[i]
        console.log(hand.keypoints.length)
      }
    }
  }
}

function gotHands(results)
```

```
{  
  hands = results  
}
```

Notes

This does not do anything yet; it is just adding another piece to the jigsaw. The console log will check to see how many datapoints (**keypoints**) it has detected. There should be **21**.



Sketch H3.4 emitters

We have our hands array, we need to make those datapoints become **emitters** rather than just a point or circle. We introduce an emitter class along with an array. In other words, we want the origins of the **emitters** where the **keypoints** of your hand are.

```
let video
let handPose
let hands = []
let emitters = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  for (let i = 0; i < 21; i++)
  {
    emitters.push(new Emitter(width/2, height/2))
  }
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
  if (hands.length > 0)
  {
    for (let hand of hands)
    {
      for (let i = 0; i < hand.keypoints.length; i++)
      {
        let keypoint = hand.keypoints[i]
        let emitter = emitters[0]
        emitter.origin.x = keypoint.x
        emitter.origin.y = keypoint.y
      }
    }
  }
}
```

```
        console.log(emitter.origin.x)
    }
}
}

function gotHands(results)
{
    hands = results
}

class Emitter
{
    constructor(x, y)
    {
        this.origin = createVector(x, y)
    }
}
```

Notes

We initially fill the emitter array with all the **x**, **y** values set to **width/2**, **height/2**; it doesn't really matter, we could just set them all to zero. There is nothing to see except the **x** values in the console. After this, you can get rid of the console log.



Sketch H3.5 particles

We need some particles to play with. So we need a particle class and a particle. The particle will be an emoji which you can easily get from most modern keyboards; otherwise, just use `point()` or `circle()`. I have used a flower emoji from Apple.

```
let video
let handPose
let hands = []
let emitters = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  for (let i = 0; i < 21; i++)
  {
    emitters.push(new Emitter(width/2, height/2))
  }
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
  if (hands.length > 0)
  {
    for (let hand of hands)
    {
      for (let i = 0; i < hand.keypoints.length; i++)
      {
        let keypoint = hand.keypoints[i]
        let emitter = emitters[0]
        emitter.origin.x = keypoint.x
        emitter.origin.y = keypoint.y
```

```

        // console.log(emitter.origin.x)
    }
}
}

function gotHands(results)
{
    hands = results
}

class Emitter
{
    constructor(x, y)
    {
        this.origin = createVector(x, y)
    }
}

class Particle
{
    constructor(x, y)
    {
        this.position = createVector(x, y)
        this.icon = '🌻'
    }

    show()
    {
        textSize(24)
        textAlign(CENTER)
        noStroke()
        text(this.icon, this.position.x, this.position.y)
    }
}

```

Notes

Nothing to see yet except a nice red canvas.



Sketch H3.6 movement

As with all emitters, there is movement; you want the particles to go somewhere. They could float off, fall to the ground, or fly in all directions. We are going to do the latter, so we need to give those particles a velocity and acceleration; we have already given them a location (**position**).

```
let video
let handPose
let hands = []
let emitters = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  for (let i = 0; i < 21; i++)
  {
    emitters.push(new Emitter(width/2, height/2))
  }
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
  if (hands.length > 0)
  {
    for (let hand of hands)
    {
      for (let i = 0; i < hand.keypoints.length; i++)
      {
        let keypoint = hand.keypoints[i]
        let emitter = emitters[0]
        emitter.origin.x = keypoint.x
```

```

        emitter.origin.y = keypoint.y
        emitter.addParticle()
    }
}
}

function gotHands(results)
{
    hands = results
}

class Emitter
{
    constructor(x, y)
    {
        this.origin = createVector(x, y)
        this.particles = []
    }

    addParticle()
    {
        this.particles.push(new Particle(this.origin.x, this.origin.y))
    }
}

class Particle
{
    constructor(x, y)
    {
        this.position = createVector(x, y)
        this.acceleration = createVector(0, 0)
        this.velocity = p5.Vector.random2D()

        this.icon = '🌻'
    }

    show()
    {

```



```
    textSize(24)
    textAlign(CENTER)
    noStroke()
    text(this.icon, this.position.x, this.position.y)
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
  }
}
```

Notes

There is so much to unpack here, although nothing is seen yet. For a deeper understanding of vectors, movement, etc., it will be worth digging into the physics workbooks.

Challenge

If you comment out a line of code such as (later on): `this.acceleration.mult(0)`. You will see what difference it makes and why we multiply the acceleration by zero on each iteration.



Sketch H3.7 let's go running

Most of the pieces are in place. We now want to run everything, so we will be creating functions inside both classes called `run()`. This will produce a flurry of particles when it sees your hand.

```
let video
let handPose
let hands = []
let emitters = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDE0, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  for (let i = 0; i < 21; i++)
  {
    emitters.push(new Emitter(width/2, height/2))
  }
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
  if (hands.length > 0)
  {
    for (let hand of hands)
    {
      for (let i = 0; i < hand.keypoints.length; i++)
      {
        let keypoint = hand.keypoints[i]
        let emitter = emitters[0]
        emitter.origin.x = keypoint.x
        emitter.origin.y = keypoint.y
      }
    }
  }
}
```

```

        emitter.addParticle()
    }
}

for (let emitter of emitters)
{
    emitter.run()
}

}

function gotHands(results)
{
    hands = results
}

class Emitter
{
    constructor(x, y)
    {
        this.origin = createVector(x, y)
        this.particles = []
    }

    addParticle()
    {
        this.particles.push(new Particle(this.origin.x, this.origin.y))
    }

    run()
    {
        for (let i = this.particles.length - 1; i > 0; i--)
        {
            this.particles[i].run()
        }
    }
}

class Particle

```

```

{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.acceleration = createVector(0, 0)
    this.velocity = p5.Vector.random2D()
    this.icon = '🌻'
  }

  run()
  {
    this.show()
    this.move()
  }

  show()
  {
    textSize(24)
    textAlign(CENTER)
    noStroke()
    text(this.icon, this.position.x, this.position.y)
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
  }
}

```

Notes

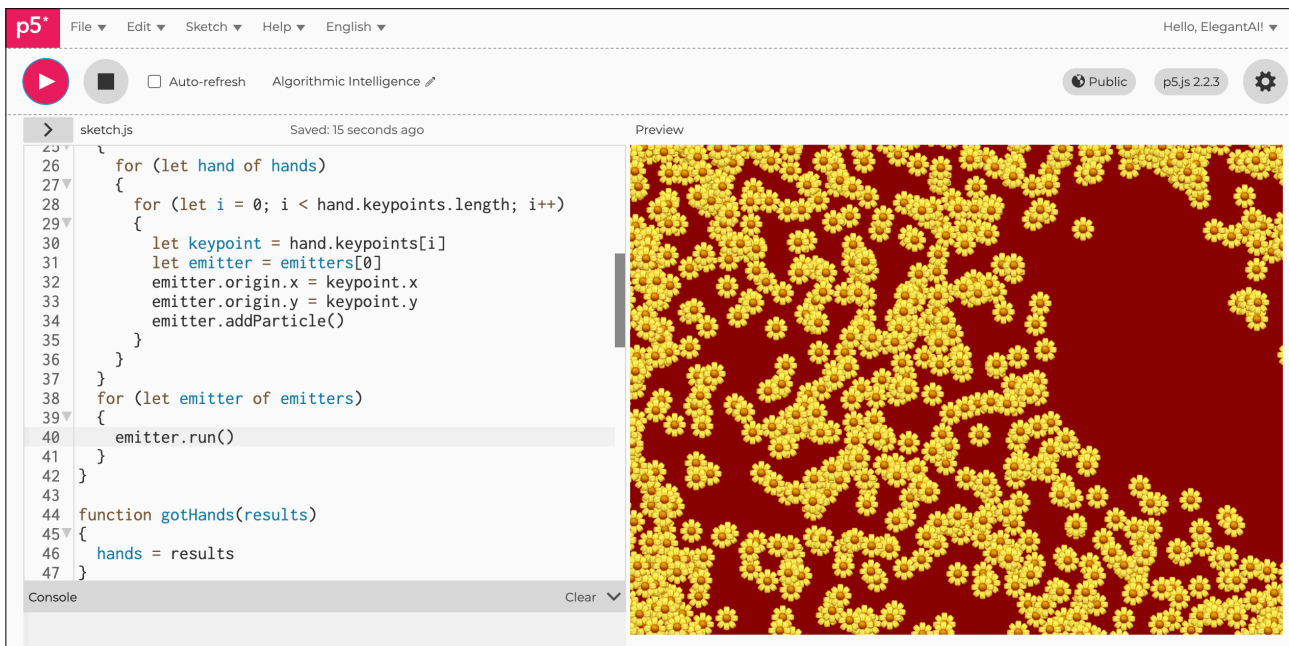
In the particle class, we do something which might seem a bit mystifying. We have a **for()** loop that works backwards through the particles array. There is a reason for this, which will become apparent shortly.

Code Explanation

```
for (let i = this.particles.length - 1; i > 0; i--)
```

This starts with the end of the array and works backwards through it

Figure H3.7





Sketch H3.8 is it alive?

As you can see, it produces many particles, and if they aren't culled, they will slow down the sketch very quickly. So we need a lifespan for the particle so that it doesn't hang around forever. Hence, the reverse `for()` loop in the particle class. We need a function and a variable to handle this.

```
let video
let handPose
let hands = []
let emitters = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  for (let i = 0; i < 21; i++)
  {
    emitters.push(new Emitter(width/2, height/2))
  }
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
  if (hands.length > 0)
  {
    for (let hand of hands)
    {
      for (let i = 0; i < hand.keypoints.length; i++)
      {
        let keypoint = hand.keypoints[i]
        let emitter = emitters[0]
        emitter.origin.x = keypoint.x
```

```

        emitter.origin.y = keypoint.y
        emitter.addParticle()
    }
}
}
for (let emitter of emitters)
{
    emitter.run()
}
}

function gotHands(results)
{
    hands = results
}

class Emitter
{
    constructor(x, y)
    {
        this.origin = createVector(x, y)
        this.particles = []
    }

    addParticle()
    {
        this.particles.push(new Particle(this.origin.x, this.origin.y))
    }

    run()
    {
        for (let i = this.particles.length - 1; i > 0; i--)
        {
            this.particles[i].run()
            if (this.particles[i].expired())
            {
                this.particles.splice(i, 1)
            }
        }
    }
}

```

```

    }
  }
}

class Particle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.acceleration = createVector(0, 0)
    this.velocity = p5.Vector.random2D()
    this.icon = '🌻'
    this.lifespan = 255
  }

  run()
  {
    this.show()
    this.move()
  }

  show()
  {
    textSize(24)
    textAlign(CENTER)
    noStroke()
    text(this.icon, this.position.x, this.position.y)
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
    this.lifespan -= 12
  }

  expired()

```



```

{
  return this.lifespan < 0
}
}

```

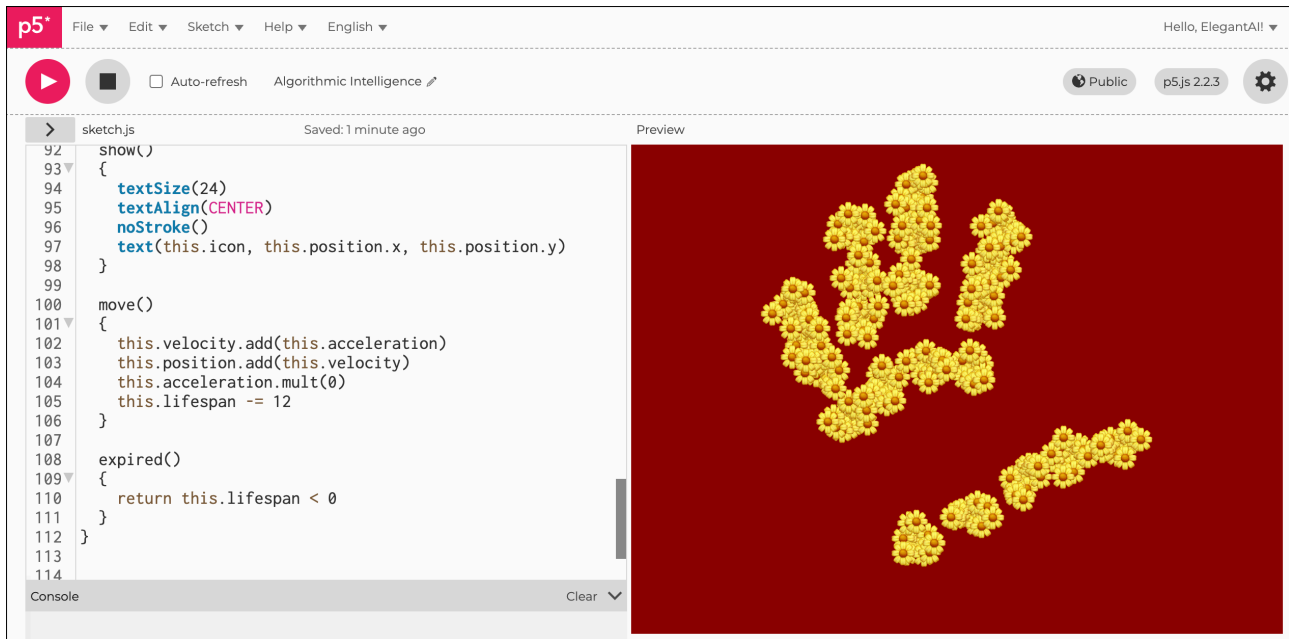
Notes

This improves the performance of the sketch no end.

Code Explanation

<code>if (this.particles[i].expired())</code>	Checks each particle to see if it has expired
<code>this.particles.splice(i, 1)</code>	When that particle has expired we remove it from the array
<code>this.lifespan = 255</code>	Our starting lifespan, the reason we use 255 is so that we can gently remove it rather than just suddenly eliminating it.
<code>this.lifespan -= 12</code>	Every iteration we reduce the lifespan by 12
<code>return this.lifespan < 0</code>	Think of this as an if statement, if the lifespan is less than zero, then return the expired() function as true for each particle.

Figure H3.8





Sketch H3.9 tweak

One final tweak to make it look a lot less sudden: we can reduce the alpha by how much lifespan it has before it is eliminated.

```
let video
let handPose
let hands = []
let emitters = []

async function setup()
{
  createCanvas(640, 480)
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
  for (let i = 0; i < 21; i++)
  {
    emitters.push(new Emitter(width/2, height/2))
  }
}

function draw()
{
  background('darkred')
  // image(video, 0, 0)
  if (hands.length > 0)
  {
    for (let hand of hands)
    {
      for (let i = 0; i < hand.keypoints.length; i++)
      {
        let keypoint = hand.keypoints[i]
        let emitter = emitters[0]
        emitter.origin.x = keypoint.x
        emitter.origin.y = keypoint.y
        emitter.addParticle()
```

```

    }
  }
}
for (let emitter of emitters)
{
  emitter.run()
}
}

function gotHands(results)
{
  hands = results
}

class Emitter
{
  constructor(x, y)
  {
    this.origin = createVector(x, y)
    this.particles = []
  }

  addParticle()
  {
    this.particles.push(new Particle(this.origin.x, this.origin.y))
  }

  run()
  {
    for (let i = this.particles.length - 1; i > 0; i--)
    {
      this.particles[i].run()
      if (this.particles[i].expired())
      {
        this.particles.splice(i, 1)
      }
    }
  }
}

```

```

}

class Particle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.acceleration = createVector(0, 0)
    this.velocity = p5.Vector.random2D()

    this.icon = '🌻'
    this.lifespan = 255
  }

  run()
  {
    this.show()
    this.move()
  }

  show()
  {
    textSize(24)
    textAlign(CENTER)
    noStroke()
    fill(255, this.lifespan)
    text(this.icon, this.position.x, this.position.y)
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
    this.lifespan -= 12
  }

  expired()
  {

```

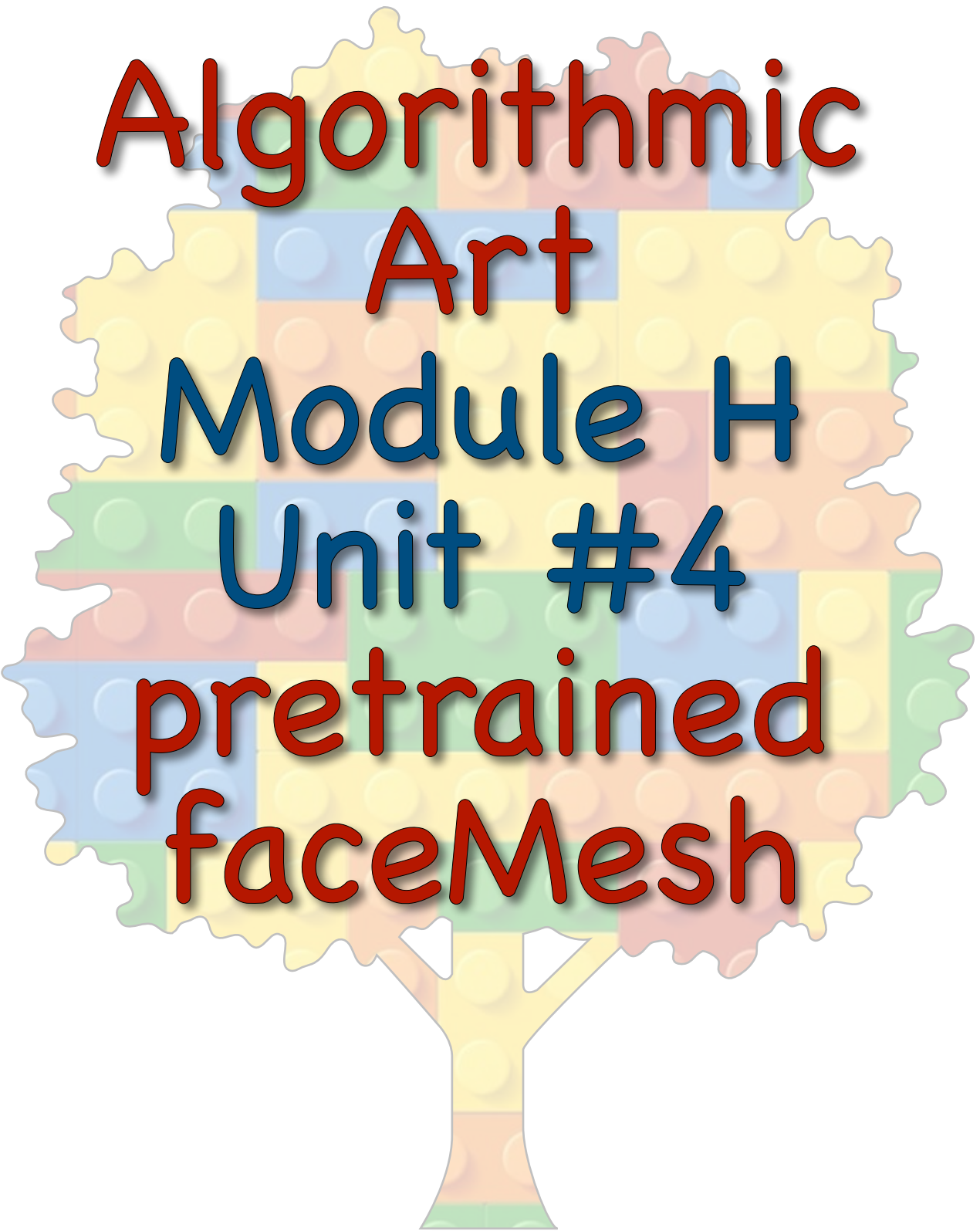
```
    return this.lifespan < 0
  }
}
```

Notes

A far more pleasing result, I think you will agree.

Figure H3.9





Algorithmic
Art

Module H

Unit #4

pretrained
faceMesh



Module H Unit #4 faceMesh

Using the faceMesh pre-trained model, we can create some fun stuff with your face. We will be adding an image to your face, so upload one to your sketch. In my case, I have uploaded an image of some clouds from Unsplash and superimposed it onto my face.



Sketch H4.1 index.html

Just a quick reminder to have the ml5.js library listed in the index.html file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.3/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```

Notes

Notice I am using p5.js version **2.2.3**. I recommend that you use that version too, if not already the default.



Sketch H4.2 our starting sketch

This is very similar to our hand sketch.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}
```

Notes

Nothing to see yet.



Sketch H4.3 points of view

We want to draw a pixel `point()` at each data or `keypoint`. The video will be commented out. If you want to see your face and the points on your face, then just uncomment it.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

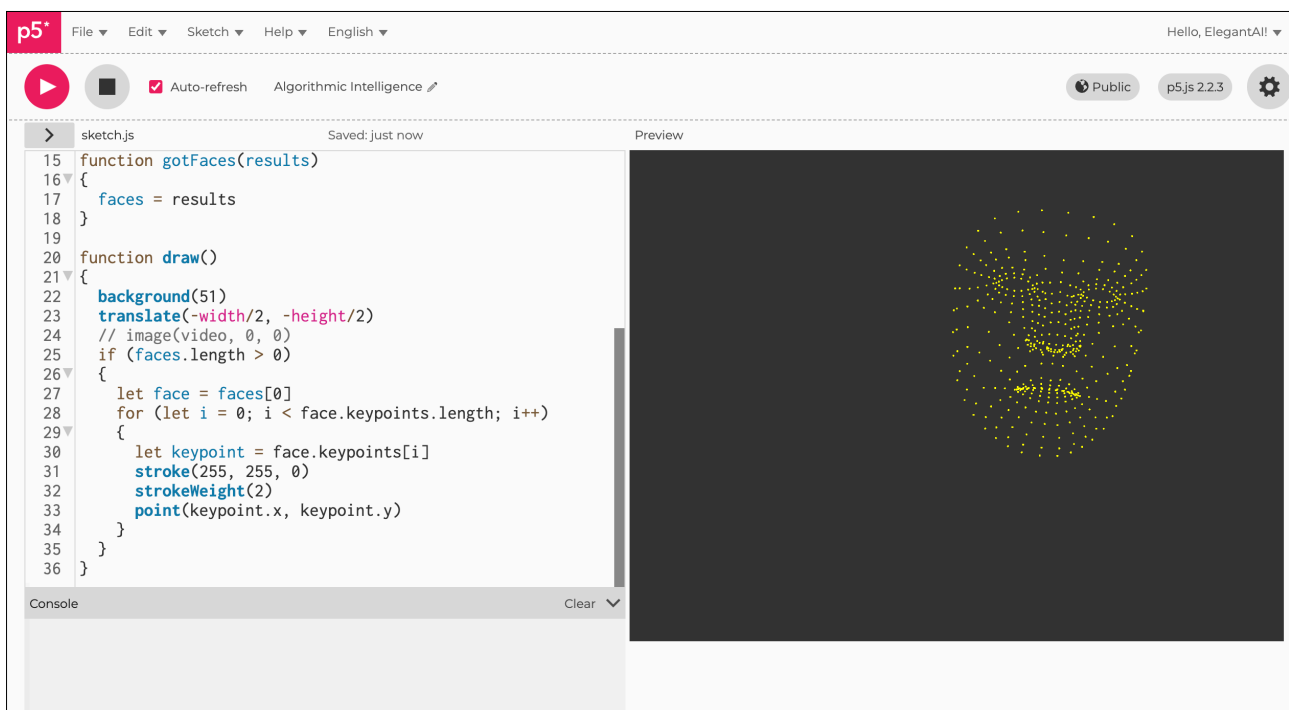
function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    for (let i = 0; i < face.keypoints.length; i++)
    {
      let keypoint = face.keypoints[i]
      stroke(255, 255, 0)
      strokeWeight(2)
      point(keypoint.x, keypoint.y)
    }
  }
}
```

```
}  
  
}  
  
}
```

Notes

I have run through this very quickly as it is so similar to the handPose models. There is plenty of interesting and fun stuff we can do with this model.

Figure H4.3





Sketch H4.4 step back

Remove the bulk of the code in draw. We want to draw the lips. So we need all the points that refer to the inner and outer lips in order. We add this as an array for each set.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
  }
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]

let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]
```

Notes

Somebody has very kindly created the lips arrays for us. I didn't do it. Whoever did that, thank you.



Sketch H4.5 drawing outer lips

Using the outer lips array, we cycle through them and draw the vertex at that data point.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
      let index = lipsExterior[i]
      let keypoint = face.keypoints[index]
      stroke('red')
      strokeWeight(2)
```

```
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
}
}
```



```
let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]
let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]
```

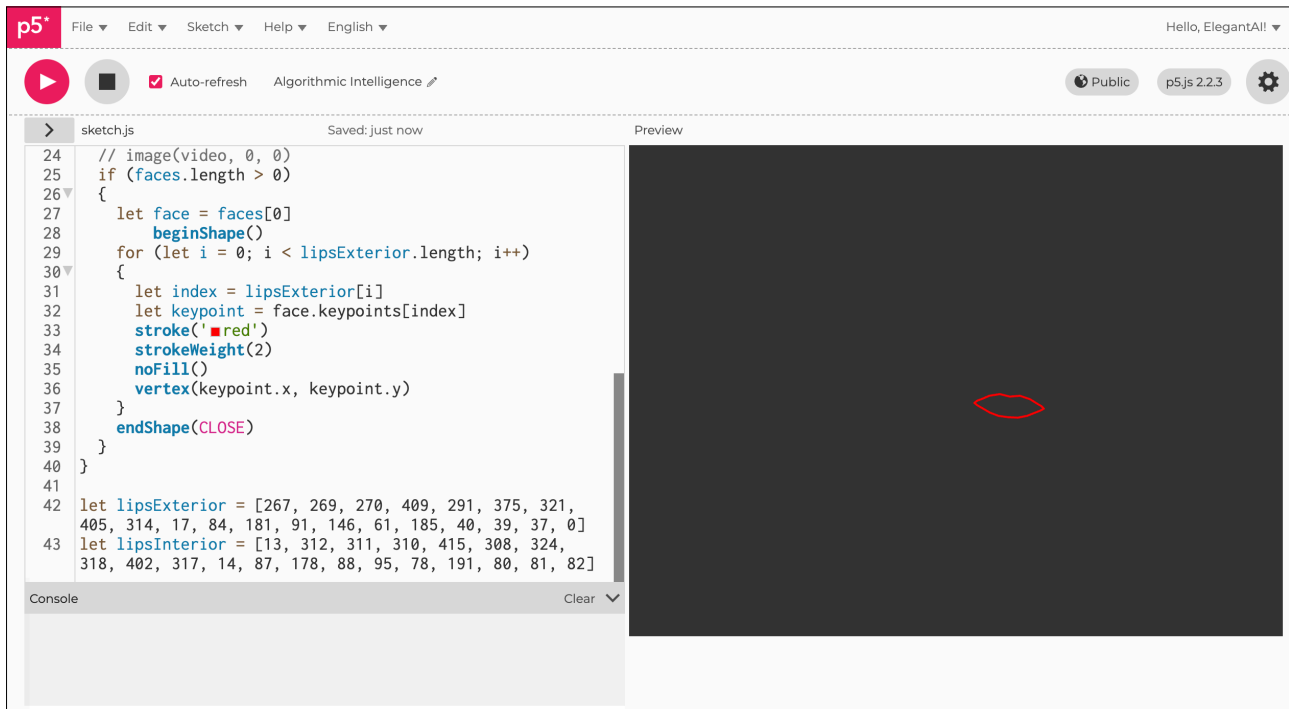
Notes

You should have your lips displayed and moving in sync.

Challenges

1. Change the colour
2. Add your face

Figure H4.5





Sketch H4.6 and the inner lips

The same again. This time cycle through the inner lips array, drawing the vertex.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
      let index = lipsExterior[i]
      let keypoint = face.keypoints[index]
      stroke('red')
      strokeWeight(2)
      noFill()
```

```

    vertex(keypoint.x, keypoint.y)
  }
endShape(CLOSE)

beginShape()
for (let i = 0; i < lipsInterior.length; i++)
{
  let index = lipsInterior[i]
  let keypoint = face.keypoints[index]
  stroke('white')
  strokeWeight(2)
  noFill()
  vertex(keypoint.x, keypoint.y)
}
endShape(CLOSE)

}
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]

let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]

```

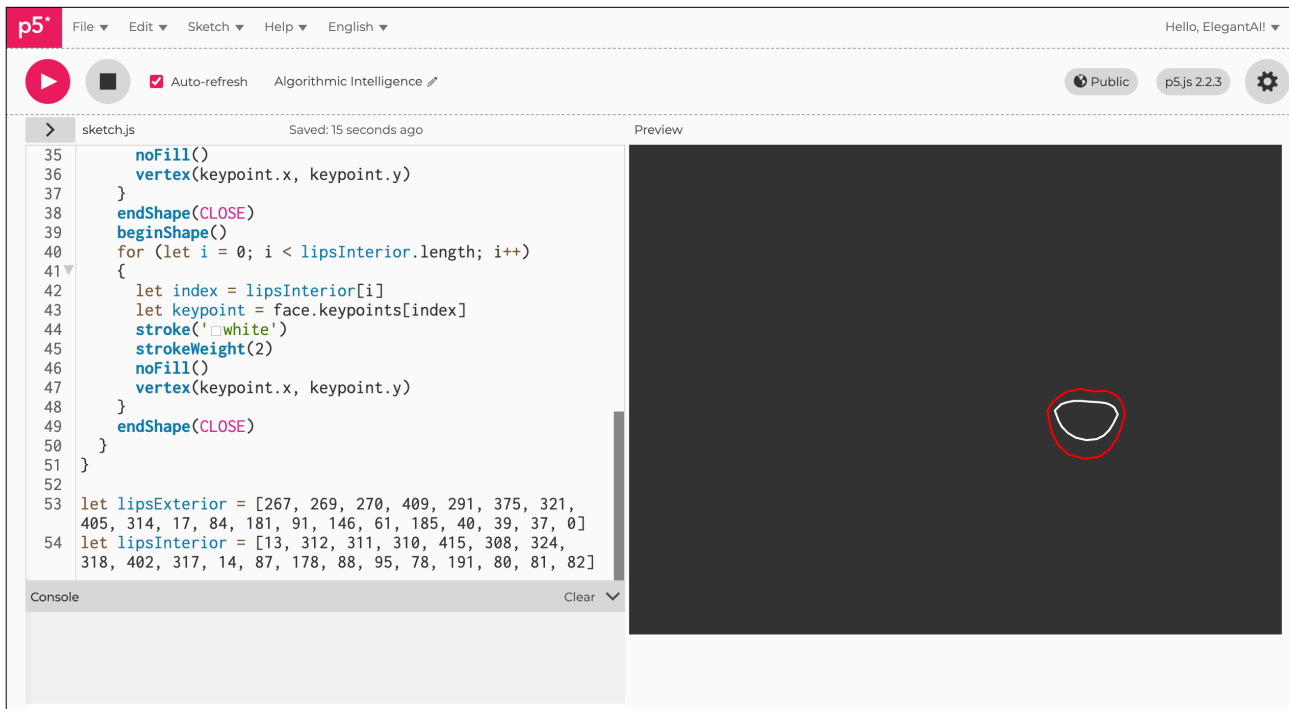
Notes

Should now have two sets of lips moving in sync.

Challenge

Can you think of a way to fill in between the inner and outer lips?

Figure H4.6





Sketch H4.7 lip sync

We are going to add a nose (simple circle) that expands when you open and close your mouth. Why not? We need to work out when you are opening and closing your mouth.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
      let index = lipsExterior[i]
      let keypoint = face.keypoints[index]
      stroke('red')
      strokeWeight(2)
```

```

    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  beginShape()
  for (let i = 0; i < lipsInterior.length; i++)
  {
    let index = lipsInterior[i]
    let keypoint = face.keypoints[index]
    stroke('white')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)

  let a = face.keypoints[13]
  let b = face.keypoints[14]
  let d = dist(a.x, a.y, b.x, b.y)
}
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]
let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]

```

Notes

Working out the distance between point 13 and 14 for the interior lips.



Sketch H4.8 now the nose

We are going to add a new nose, nothing glamorous.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
      let index = lipsExterior[i]
      let keypoint = face.keypoints[index]
      stroke('red')
      strokeWeight(2)
      noFill()
```

```

    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  beginShape()
  for (let i = 0; i < lipsInterior.length; i++)
  {
    let index = lipsInterior[i]
    let keypoint = face.keypoints[index]
    stroke('white')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  let a = face.keypoints[13]
  let b = face.keypoints[14]
  let d = dist(a.x, a.y, b.x, b.y)
  noStroke()
  let nose = face.keypoints[19]
  fill('yellow')
  circle(nose.x, nose.y, d)
}
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]
let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]

```

Notes

Looks even better with the image of you being displayed.

Challenge

Could be a clown nose.



Blowing Bubbles

We are now going to blow bubbles through our mouths. Rather, we are going to emit bubbles from our mouths when we open our mouths. This has many similarities to the emoji emitter in the previous unit. We will need an emitter class and a particle class.



Sketch H4.9 adding the emitter

! Here we remove the nose.

We want to have bubbles emit from your mouth as you open and close your mouth. Similar to the emoji emitter with handPose. So the emitter class is almost identical, except that the particle now has a **d** value for the diameter.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
      let index = lipsExterior[i]
      let keypoint = face.keypoints[index]
```

```

    stroke('red')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  beginShape()
  for (let i = 0; i < lipsInterior.length; i++)
  {
    let index = lipsInterior[i]
    let keypoint = face.keypoints[index]
    stroke('white')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  let a = face.keypoints[13]
  let b = face.keypoints[14]
  let d = dist(a.x, a.y, b.x, b.y)
  // noStroke()
  // let nose = face.keypoints[19]
  // fill('yellow')
  // circle(nose.x, nose.y, d)
}
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]
let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]

class Emitter
{
  constructor(x, y)
  {
    this.origin = createVector(x, y)
    this.particles = []
  }
}

```

```
addParticle(x, y, d)
{
  this.particles.push(new Particle(x, y, d))
}

run()
{
  for (let i = this.particles.length - 1; i >= 0; i--)
  {
    this.particles[i].run()
    if (this.particles[i].expired())
    {
      this.particles.splice(i, 1)
    }
  }
}
```

Notes

The emitter references the particle.



Sketch H4.10 adding the particle

Again, this is very similar to the previous unit, so I'm not going to go through all of it in detail. We will be adding gravity as a force shortly.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
      let index = lipsExterior[i]
      let keypoint = face.keypoints[index]
      stroke('red')
      strokeWeight(2)
```

```

    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  beginShape()
  for (let i = 0; i < lipsInterior.length; i++)
  {
    let index = lipsInterior[i]
    let keypoint = face.keypoints[index]
    stroke('white')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  let a = face.keypoints[13]
  let b = face.keypoints[14]
  let d = dist(a.x, a.y, b.x, b.y)
}
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]
let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]

class Emitter
{
  constructor(x, y)
  {
    this.origin = createVector(x, y)
    this.particles = []
  }

  addParticle(x, y, d)
  {
    this.particles.push(new Particle(x, y, d))
  }
}

```

```

run()
{
  for (let i = this.particles.length - 1; i >= 0; i--)
  {
    this.particles[i].run()
    if (this.particles[i].expired())
    {
      this.particles.splice(i, 1)
    }
  }
}
}

```

```

class Particle
{
  constructor(x, y, d)
  {
    this.position = createVector(x, y)
    this.acceleration = createVector(0, 0)
    this.velocity = createVector(random(-2, 2), random(-2, 2))
    this.lifespan = 255
    this.r = d * 0.5
  }

  run()
  {
    this.move()
    this.show()
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.position.add(this.velocity)
    this.lifespan -= 2
    this.acceleration.mult(0)
  }
}

```

```
show()  
{  
  stroke(255, this.lifespan)  
  fill(255, this.lifespan * 0.5)  
  strokeWeight(4)  
  circle(this.position.x, this.position.y, this.r * 2)  
}  
  
expired()  
{  
  return this.lifespan < 0  
}  
}
```

Notes

The particle in this case is a simple circle. The fill is relative to half its lifespan.

Challenge

Use an emoji instead of the particle.



Sketch H4.11 now the bubbles

We need an emitter variable and create an emitter at some place initially. Then we calculate the halfway point for the x and y values for each bubble as your mouth opens, closes, and moves. The bubble only emits if the mouth is open more than 20 pixels. To have it slow down a bit, we use `random(1)` so that it emits just 75% of the time.

```
let video
let faceMesh
let faces = []
let emitter

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
  emitter = new Emitter(300, 300)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
```

```

    let index = lipsExterior[i]
    let keypoint = face.keypoints[index]
    stroke('red')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  beginShape()
  for (let i = 0; i < lipsInterior.length; i++)
  {
    let index = lipsInterior[i]
    let keypoint = face.keypoints[index]
    stroke('white')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  let a = face.keypoints[13]
  let b = face.keypoints[14]
  let d = dist(a.x, a.y, b.x, b.y)
  let x = (a.x + b.x) * 0.5
  let y = (a.y + b.y) * 0.5
  if (d > 20 && random(1) < 0.25)
  {
    emitter.addParticle(x, y, d)
  }
}
emitter.run()
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]
let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]

class Emitter
{

```

```

constructor(x, y)
{
  this.origin = createVector(x, y)
  this.particles = []
}

addParticle(x, y, d)
{
  this.particles.push(new Particle(x, y, d))
}

run()
{
  for (let i = this.particles.length - 1; i >= 0; i--)
  {
    this.particles[i].run()
    if (this.particles[i].expired())
    {
      this.particles.splice(i, 1)
    }
  }
}

class Particle
{
  constructor(x, y, d)
  {
    this.position = createVector(x, y)
    this.acceleration = createVector(0, 0)
    this.velocity = createVector(random(-2, 2), random(-2, 2))
    this.lifespan = 255
    this.r = d * 0.5
  }

  run()
  {
    this.move()
  }
}

```

```

    this.show()
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.position.add(this.velocity)
    this.lifespan -= 2
    this.acceleration.mult(0)
  }

  show()
  {
    stroke(255, this.lifespan)
    fill(255, this.lifespan * 0.5)
    strokeWeight(4)
    circle(this.position.x, this.position.y, this.r * 2)
  }

  expired()
  {
    return this.lifespan < 0
  }
}

```

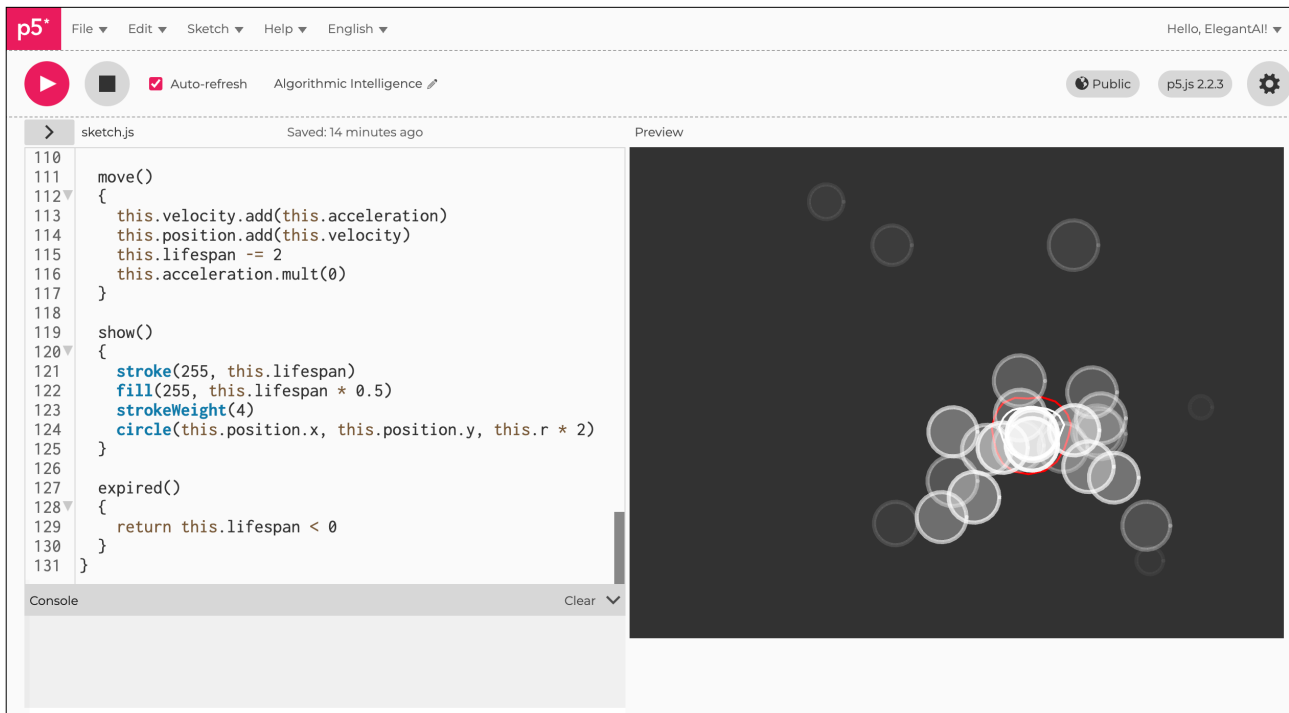
Notes

When the condition is met, we add a particle to the emitter and run the emitter. You should get bubbles coming out of your mouth. Notice that they go in all directions.

Challenge

You could do this without drawing the lips for a better effect.

Figure H4.11





Sketch H4.12 negative gravity

It would be better if the bubbles rose upwards. So let us add some negative gravity!

```
let video
let faceMesh
let faces = []
let emitter

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
  emitter = new Emitter(300, 300)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape()
    for (let i = 0; i < lipsExterior.length; i++)
    {
      let index = lipsExterior[i]
      let keypoint = face.keypoints[index]
      stroke('red')
```

```

    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  beginShape()
  for (let i = 0; i < lipsInterior.length; i++)
  {
    let index = lipsInterior[i]
    let keypoint = face.keypoints[index]
    stroke('white')
    strokeWeight(2)
    noFill()
    vertex(keypoint.x, keypoint.y)
  }
  endShape(CLOSE)
  let a = face.keypoints[13]
  let b = face.keypoints[14]
  let d = dist(a.x, a.y, b.x, b.y)
  let x = (a.x + b.x) * 0.5
  let y = (a.y + b.y) * 0.5
  if (d > 20 && random(1) < 0.25)
  {
    emitter.addParticle(x, y, d)
  }
}
emitter.run()
}

let lipsExterior = [267, 269, 270, 409, 291, 375, 321, 405, 314, 17, 84, 181,
91, 146, 61, 185, 40, 39, 37, 0]
let lipsInterior = [13, 312, 311, 310, 415, 308, 324, 318, 402, 317, 14, 87,
178, 88, 95, 78, 191, 80, 81, 82]

class Emitter
{
  constructor(x, y)
  {
    this.origin = createVector(x, y)
  }
}

```

```

    this.particles = []
  }

  addParticle(x, y, d)
  {
    this.particles.push(new Particle(x, y, d))
  }

  run()
  {
    for (let i = this.particles.length - 1; i >= 0; i--)
    {
      this.particles[i].run()
      if (this.particles[i].expired())
      {
        this.particles.splice(i, 1)
      }
    }
  }
}

class Particle
{
  constructor(x, y, d)
  {
    this.position = createVector(x, y)
    this.acceleration = createVector(0, 0)
    this.velocity = createVector(random(-2, 2), random(-2, 2))
    this.lifespan = 255
    this.r = d * 0.5
  }

  run()
  {
    let gravity = createVector(0, -0.05)
    this.applyForce(gravity)
    this.move()
    this.show()
  }
}

```



```

}

applyForce(force)
{
  this.acceleration.add(force)
}

move()
{
  this.velocity.add(this.acceleration)
  this.position.add(this.velocity)
  this.lifespan -= 2
  this.acceleration.mult(0)
}

show()
{
  stroke(255, this.lifespan)
  fill(255, this.lifespan * 0.5)
  strokeWeight(4)
  circle(this.position.x, this.position.y, this.r * 2)
}

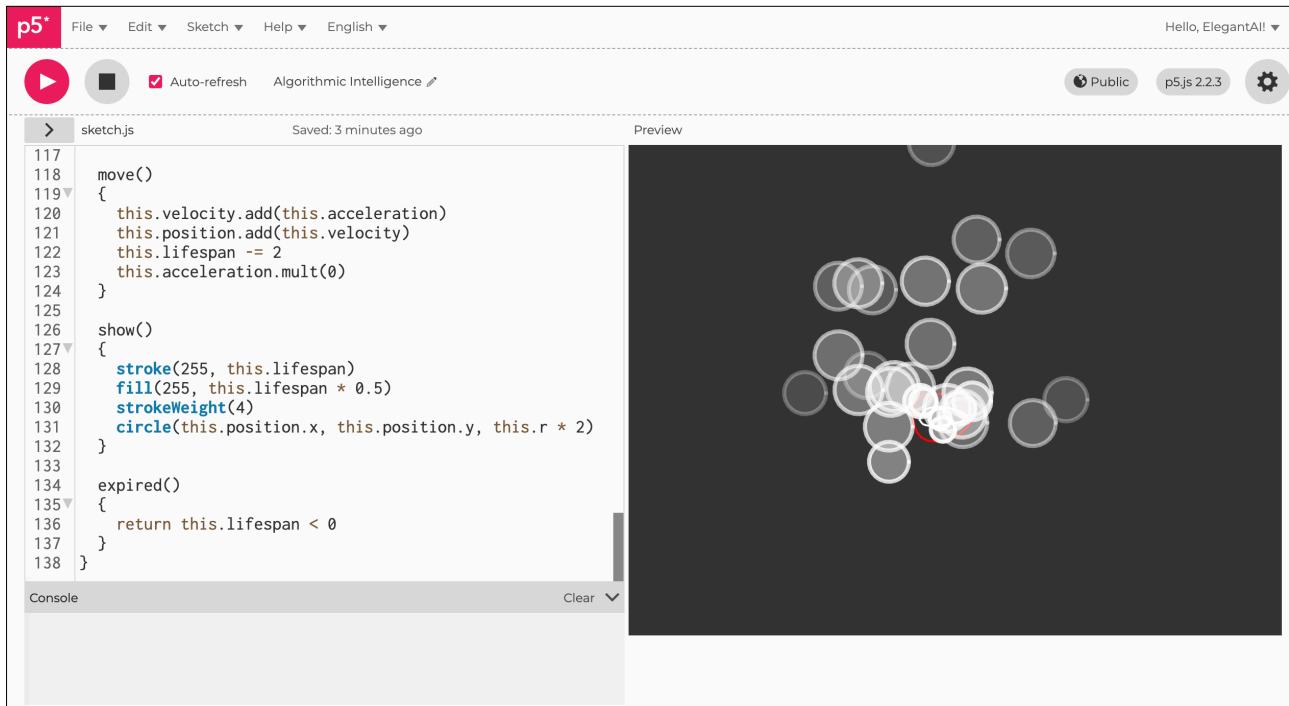
expired()
{
  return this.lifespan < 0
}
}

```

Notes

Looks really nice now.

Figure H4.12





Sketch H4.13 back to basics

Remove everything except leave what is shown below.

```
let video
let faceMesh
let faces = []

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {

  }
}
```

Notes

Just a blank canvas.



Sketch H4.14 triangles

One of the features of faceMesh is that it has a built-in function to get the triangles that join all the data points on the face. We create a variable called triangles to hold that data (console log to see it). There are three data points for each triangle. There are **852** triangles with a total of **2,556** data points.

```
let video
let faceMesh
let faces = []
let triangles

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
  triangles = faceMesh.getTriangles()
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
  }
}
```

Notes

You might see it running slower in certain circumstances.



Sketch H4.15 showing the triangles

There is a lot happening here. The triangles hold the array of arrays (2556 data points) for each triangle. We cycle through each triangle, pull out the data points (a, b, c). We use `beginShape(TRIANGLES)` to draw these triangles as `vertex()`.

```
let video
let faceMesh
let faces = []
let triangles

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
  triangles = faceMesh.getTriangles()
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    beginShape(TRIANGLES)
    for (let i = 0; i < triangles.length; i++)
    {
      let tri = triangles[i]
```

```

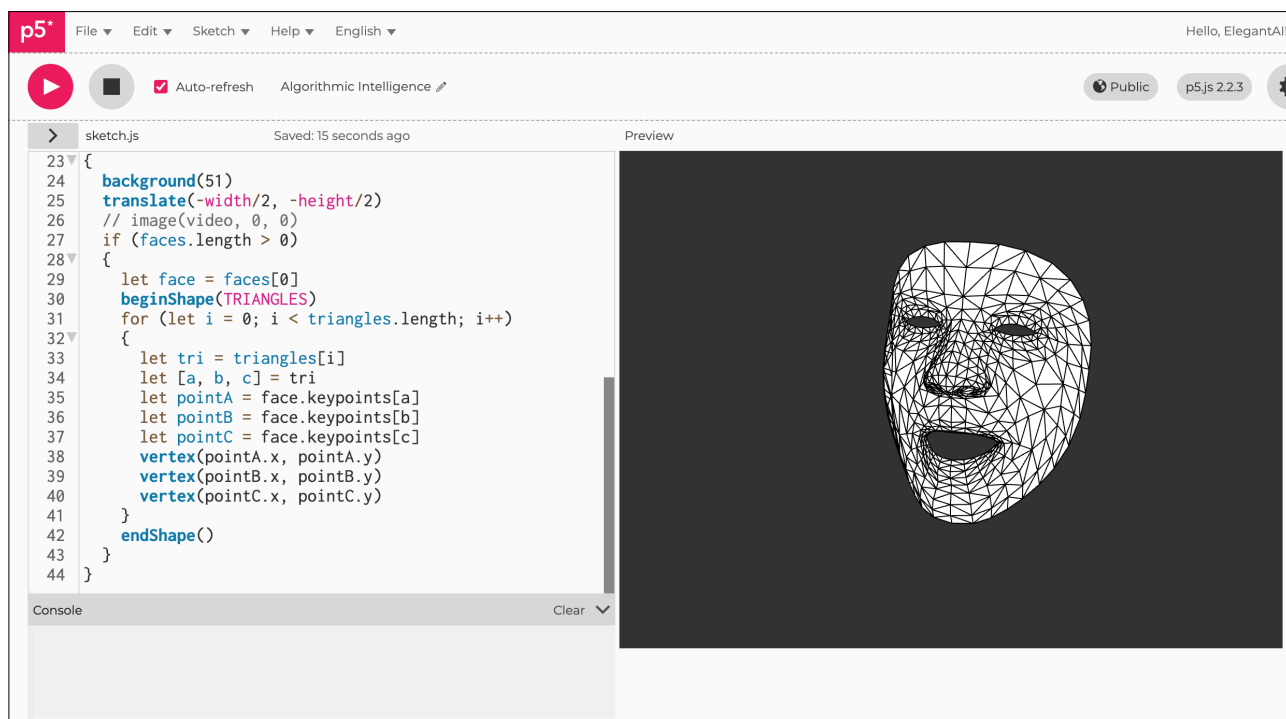
    let [a, b, c] = tri
    let pointA = face.keypoints[a]
    let pointB = face.keypoints[b]
    let pointC = face.keypoints[c]
    vertex(pointA.x, pointA.y)
    vertex(pointB.x, pointB.y)
    vertex(pointC.x, pointC.y)
  }
endShape()
}
}

```

Challenge

If you remove the triangle lines using `noStroke()`.

Figure H4.15





Sketch H4.16 adding a mask

! Upload an image and rename it in the file and sketch accordingly. I have uploaded a cloud image as shown in the fig below.

We are going to use an image; any image will do. In my case, I have downloaded an image of some clouds from [Unsplash](#), but you can use any image or your own. The mask has co-ordinates x, y, z , but these need to be mapped onto the image. The image is 2D, and the co-ordinates are called u and v , hence (u, v) . We create a variable to hold the image (`img`) and the u, v co-ordinates (`uvcoords`). There is a function in `faceMesh` to get those co-ordinates, called `getUVCoords()`.

```
let video
let faceMesh
let faces = []
let triangles

let uvcoords
let img

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  img = await loadImage("clouds.jpeg")
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
  triangles = faceMesh.getTriangles()
  uvcoords = faceMesh.getUVCoords()
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
```

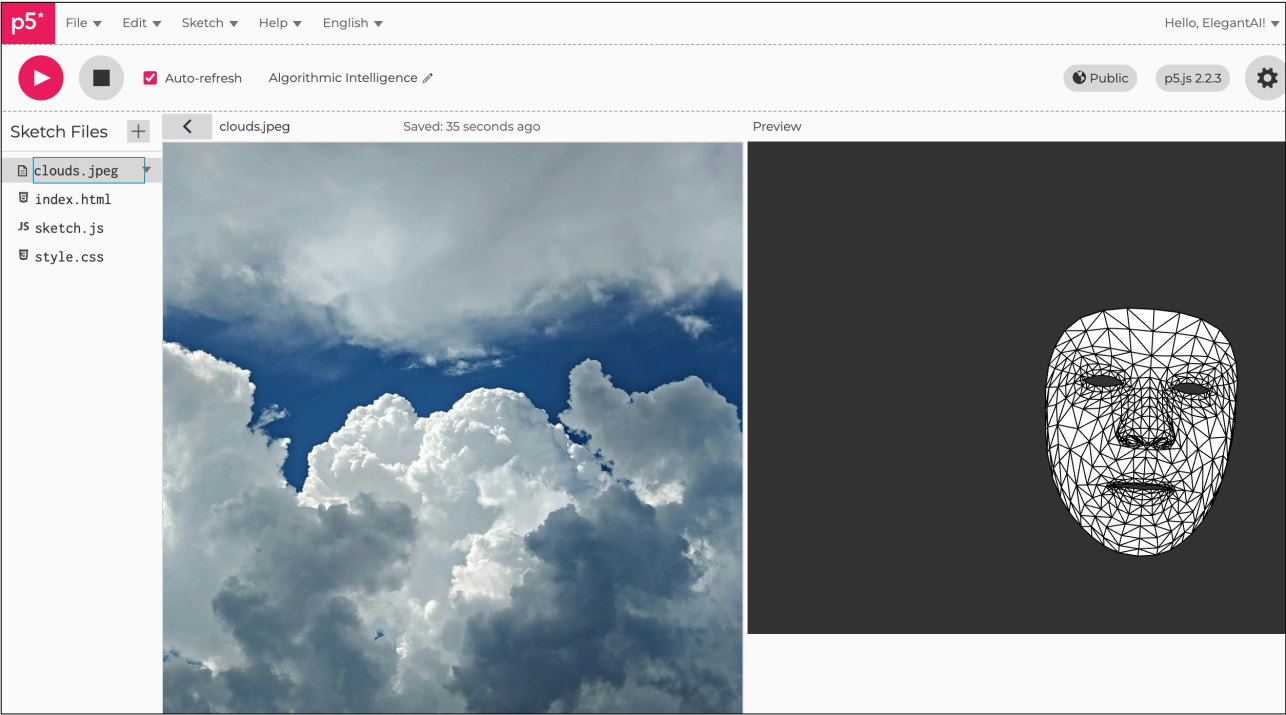


```
translate(-width/2, -height/2)
// image(video, 0, 0)
if (faces.length > 0)
{
  let face = faces[0]
  beginShape(TRIANGLES)
  for (let i = 0; i < triangles.length; i++)
  {
    let tri = triangles[i]
    let [a, b, c] = tri
    let pointA = face.keypoints[a]
    let pointB = face.keypoints[b]
    let pointC = face.keypoints[c]
    vertex(pointA.x, pointA.y)
    vertex(pointB.x, pointB.y)
    vertex(pointC.x, pointC.y)
  }
  endShape()
}
}
```

Notes

This is just the beginning and is straightforward. You can use any image of anything; I have just used clouds for effect.

Figure H4.16





Sketch H4.17 texture mapping

We now apply texture mapping to the detected face mesh.

```
let video
let faceMesh
let faces = []
let triangles
let uvcoords
let img

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  img = await loadImage("clouds.jpeg")
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
  triangles = faceMesh.getTriangles()
  uvcoords = faceMesh.getUVCoords()
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    texture(img)
    textureMode(NORMAL)
```

```
noStroke()
beginShape(TRIANGLES)
for (let i = 0; i < triangles.length; i++)
{
  let tri = triangles[i]
  let [a, b, c] = tri
  let pointA = face.keypoints[a]
  let pointB = face.keypoints[b]
  let pointC = face.keypoints[c]
  vertex(pointA.x, pointA.y)
  vertex(pointB.x, pointB.y)
  vertex(pointC.x, pointC.y)
}
endShape()
}
```

Notes

You get a distorted image of the triangles, which is because we haven't finished yet.



Sketch H4.18 mapping uv coordinates

Retrieve the corresponding **UV** coordinates for texture mapping and then define the triangle with both position (**x**, **y**) and **UV** texture coordinates.

```
let video
let faceMesh
let faces = []
let triangles
let uvcoords
let img

async function setup()
{
  createCanvas(640, 480, WEBGL)
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({ flipped: true })
  img = await loadImage("clouds.jpeg")
  video = createCapture(VIDEO, { flipped: true })
  video.hide()
  faceMesh.detectStart(video, gotFaces)
  triangles = faceMesh.getTriangles()
  uvcoords = faceMesh.getUVCoords()
}

function gotFaces(results)
{
  faces = results
}

function draw()
{
  background(51)
  translate(-width/2, -height/2)
  // image(video, 0, 0)
  if (faces.length > 0)
  {
    let face = faces[0]
    texture(img)
```

```
textureMode(NORMAL)
noStroke()
beginShape(TRIANGLES)
for (let i = 0; i < triangles.length; i++)
{
  let tri = triangles[i]
  let [a, b, c] = tri
  let pointA = face.keypoints[a]
  let pointB = face.keypoints[b]
  let pointC = face.keypoints[c]

  let uvA = uvcoords[a]
  let uvB = uvcoords[b]
  let uvC = uvcoords[c]
  vertex(pointA.x, pointA.y, uvA[0], uvA[1])
  vertex(pointB.x, pointB.y, uvB[0], uvB[1])
  vertex(pointC.x, pointC.y, uvC[0], uvC[1])
}
endShape()
}
```

Notes

We have a cloud mask.

Figure H4.18

