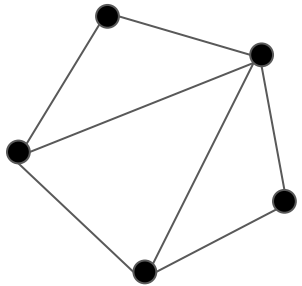
Dave is here to talk to you about

Rendering & Meshes

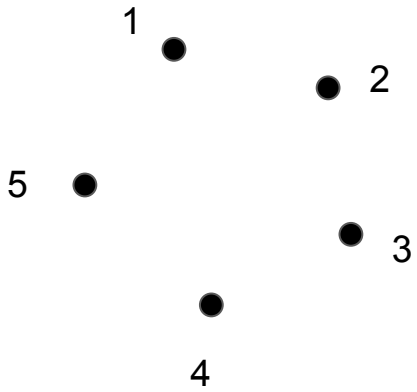
2. How does mesh rendering work, anyway?

Pipeline recap

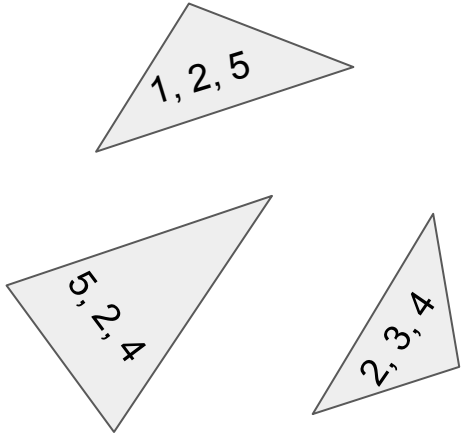
A mesh consists of:



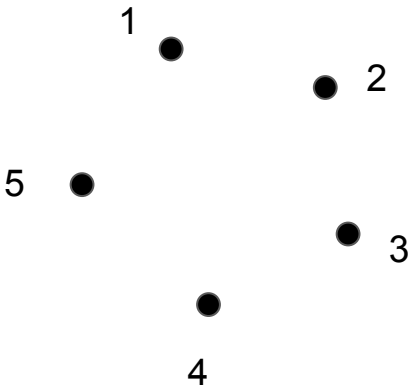
vertex positions



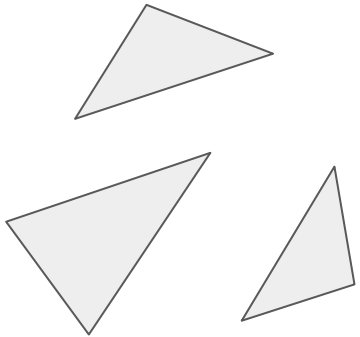
and which vertices make faces



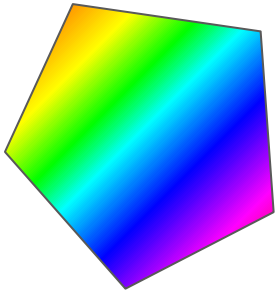
Pipeline recap



Vertex shader computes positions and other properties for each vertex



Properties are interpolated across each face pixel



Fragment shader computes the colour of each pixel on each face

What does the vertex shader do?

- Vertex positions start in **local coordinates**
- The vertex shader translates those into **screen coordinates**
  - We need to scale/rotate/translate these local coordinates into **world coordinates**
  - OpenGL wants x and y in [-1, 1] and maps that to the window automatically. This is "clip space" if you need to google things related to it

`position` `transform * position` `projection * transform * position`

local *world* *screen*

What does the vertex shader do?

- Also, pass any per-vertex info you might need to compute colours in the fragment shader with **out** variables (which become in variables in the fragment shader)

  e.g.

```
in vec2 in_texcoord;
out vec2 texcoord;
void main() {
  texcoord = in_texcoord;
  // ...etc
}
```

What does the fragment shader do?

- Using per-vertex in variables and global shader uniform variables, compute a pixel color

  e.g.

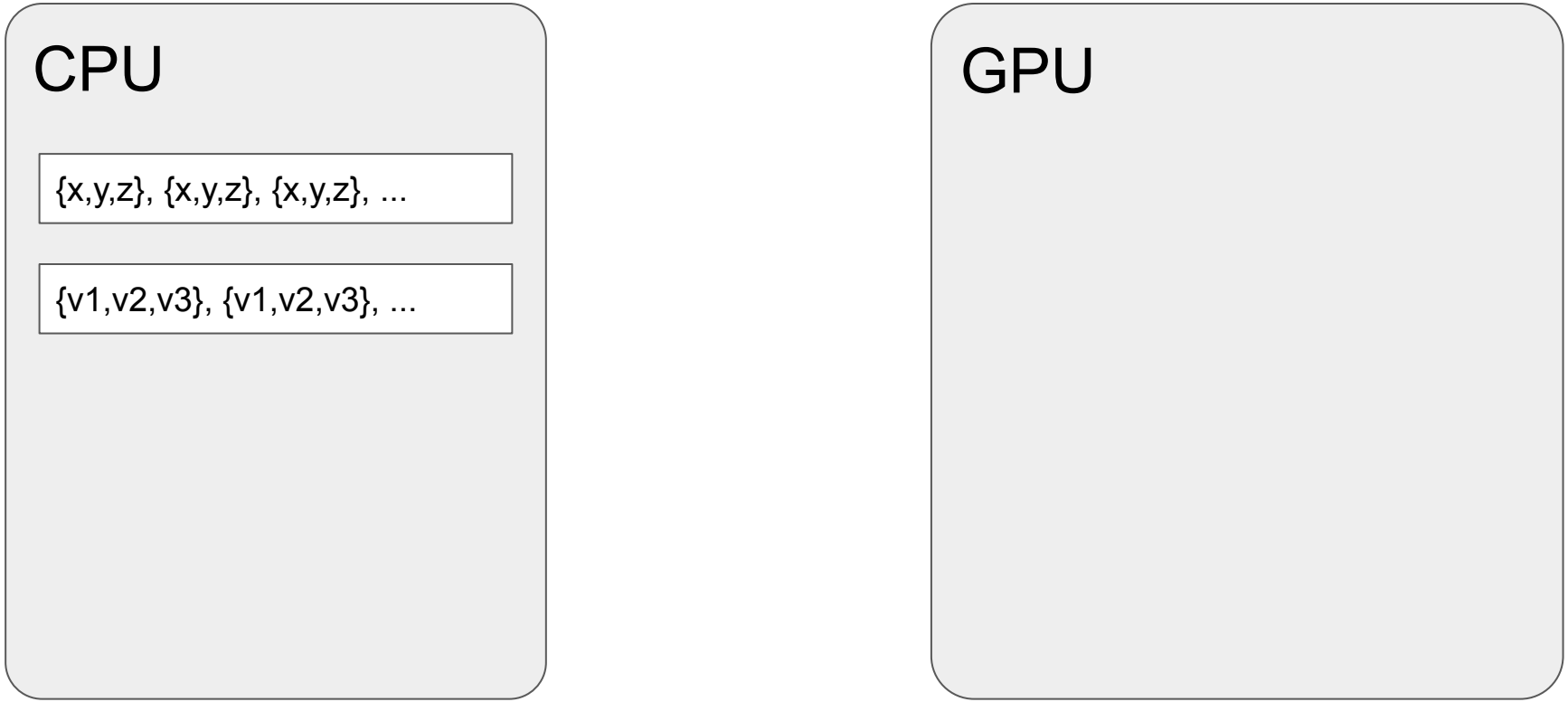
```
in vec2 texcoord;
uniform sampler2D image;
layout(location = 0) out vec4 out_color;

void main() {
  out_color = texture(image, texcoord);
}
```

Compiling shaders

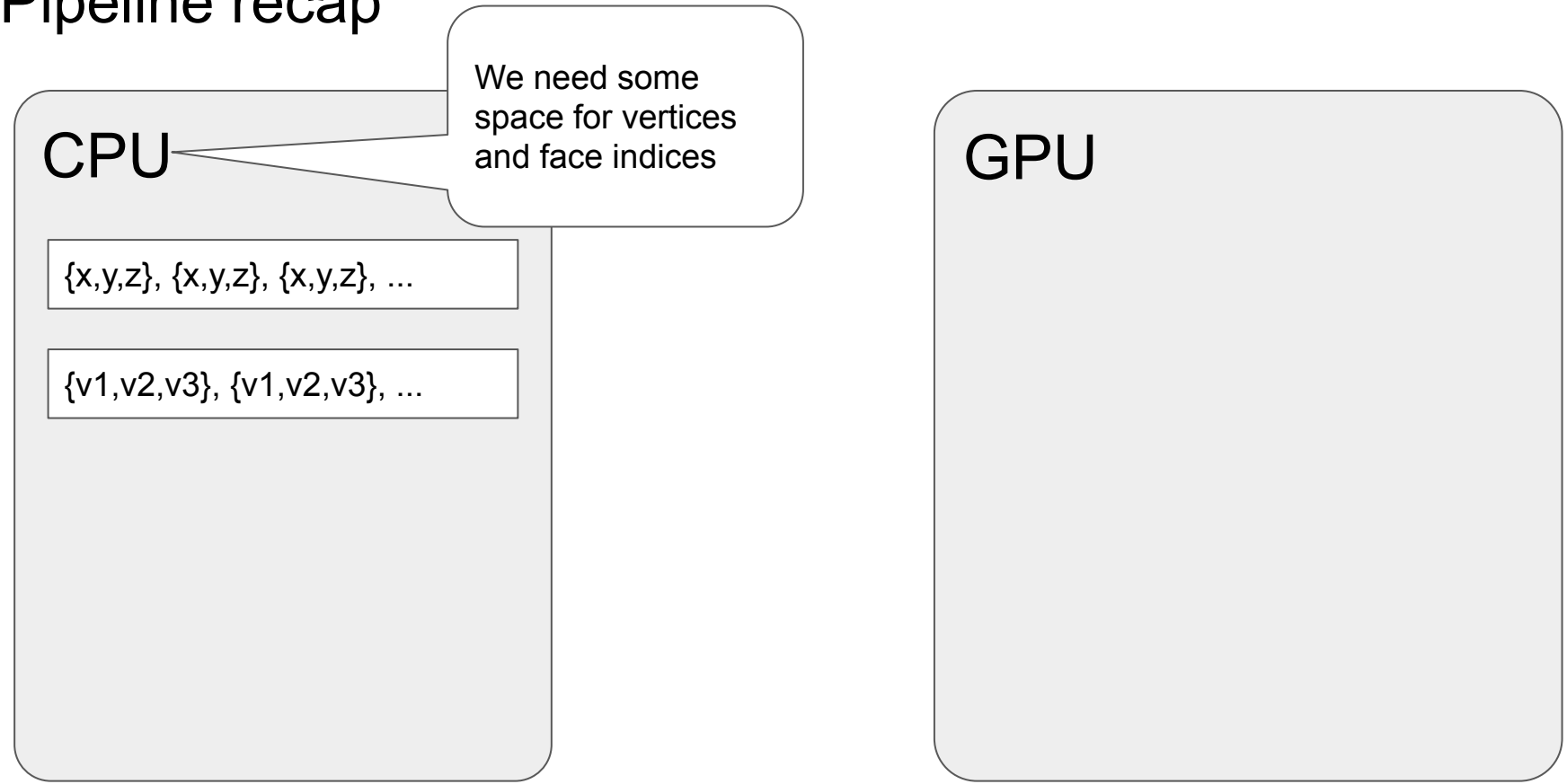
- Shaders get compiled at *runtime*, not when our C++ gets compiled
- Starting with a string for each shader, we:
  - Give the string to OpenGL with **glShaderSource()**
  - Tell OpenGL to compile the shader with **glCompileShader()**
  - Create a program for the vertex + fragment shader with **glCreateProgram()**
  - Attach both shaders to the program with **glAttachShader()**
  - Link it all together with **glLinkProgram()**

How do we get mesh info to the shaders?

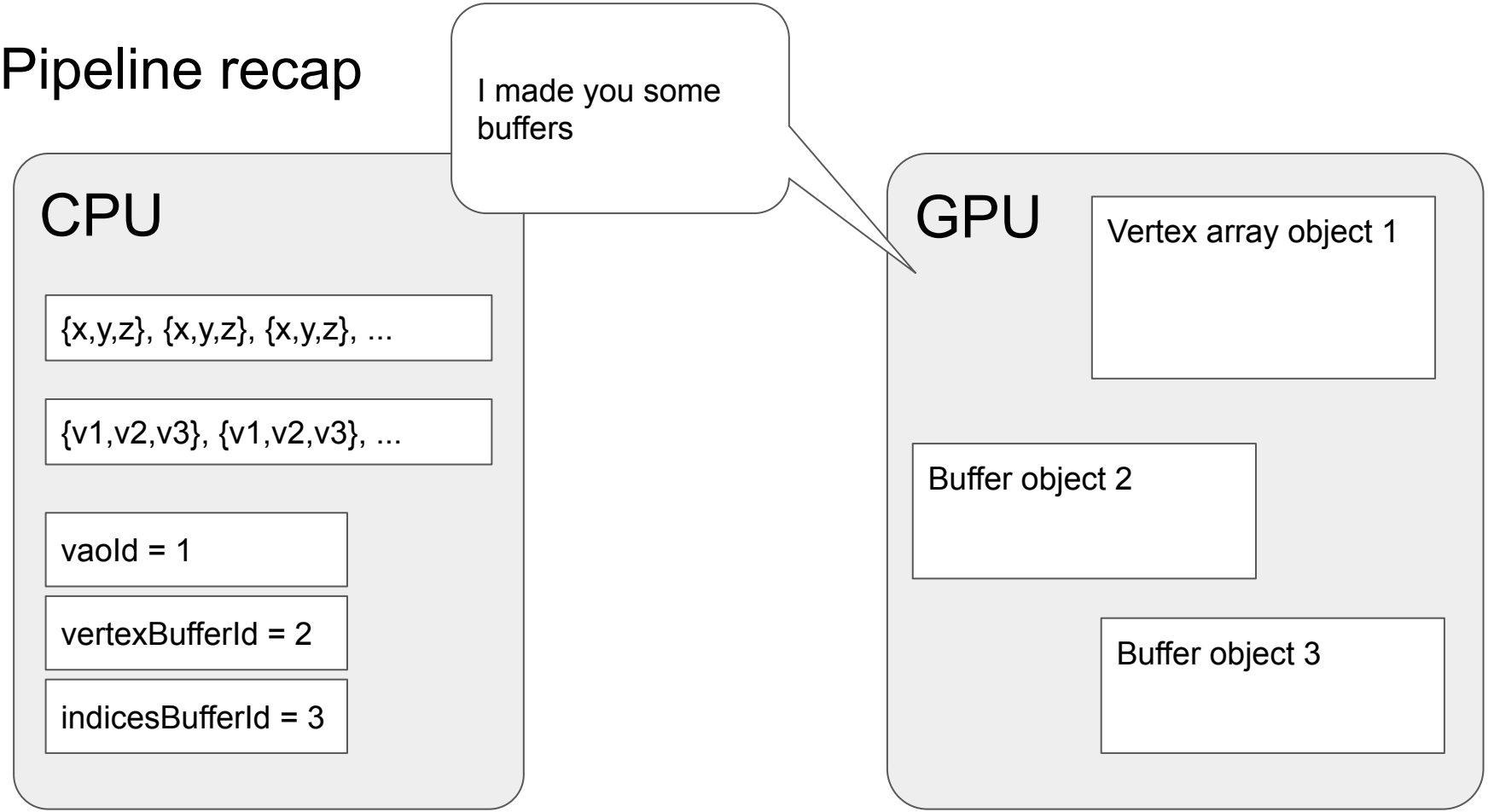
Pipeline recap



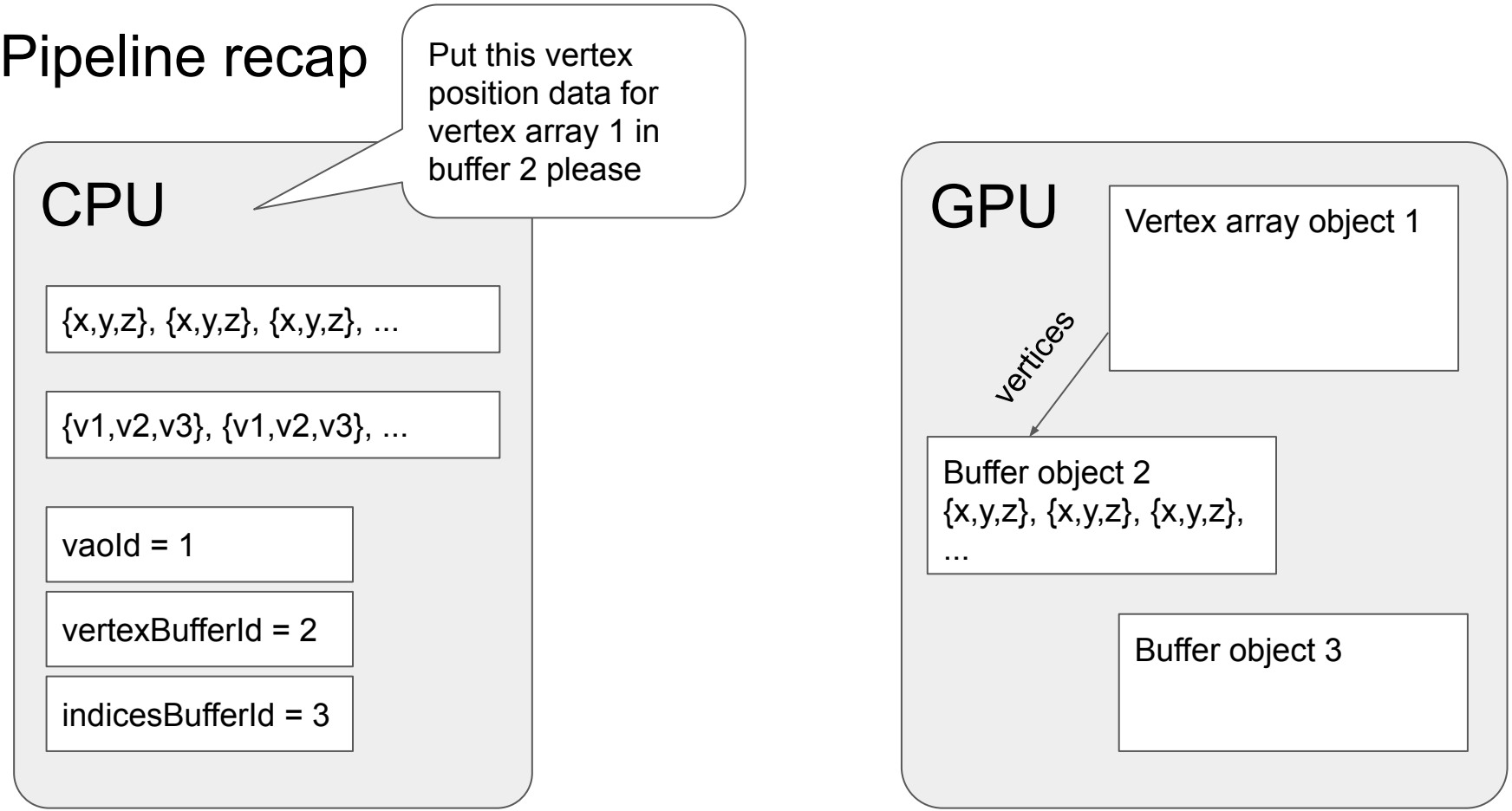
Pipeline recap



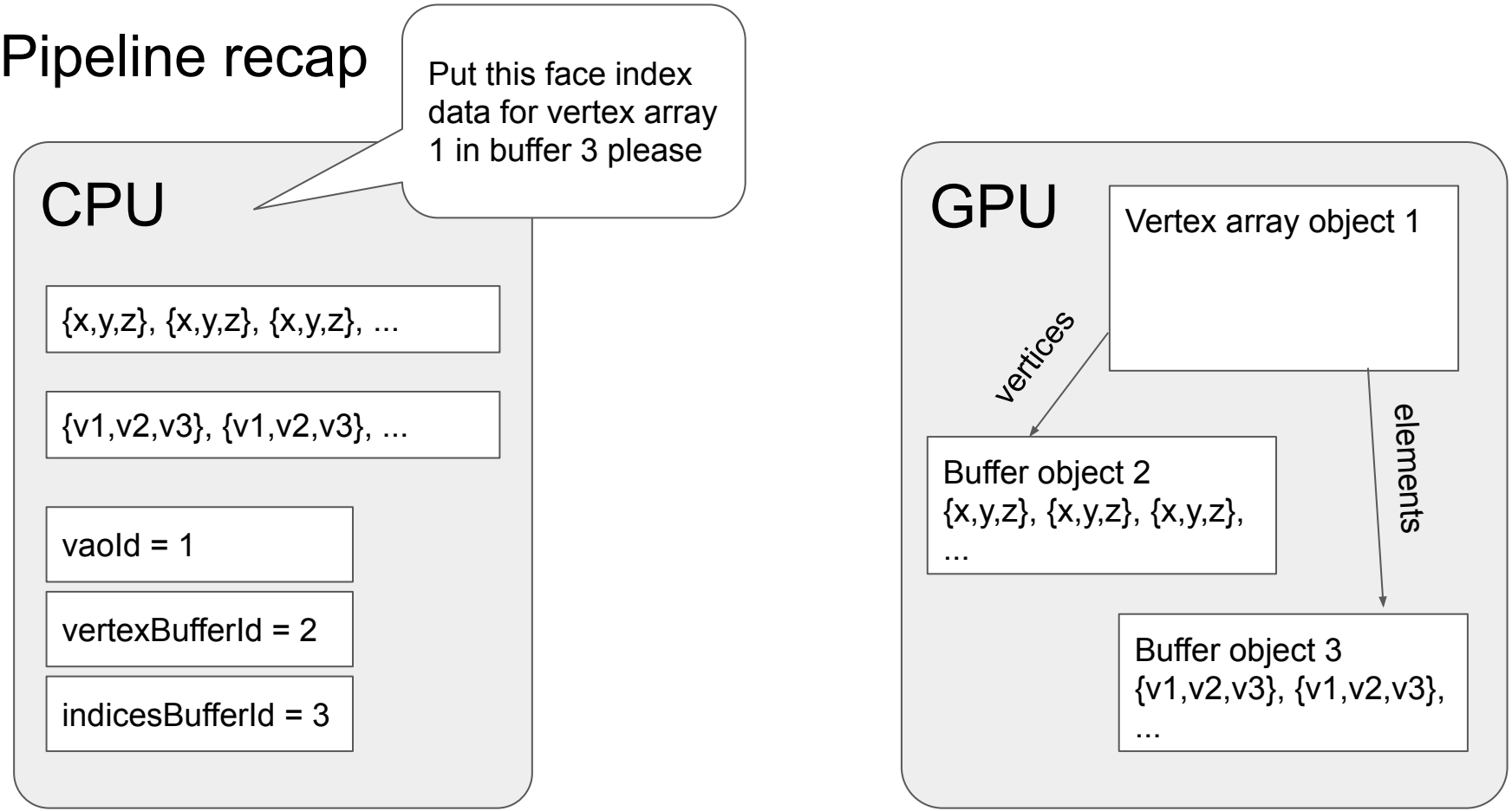
Pipeline recap



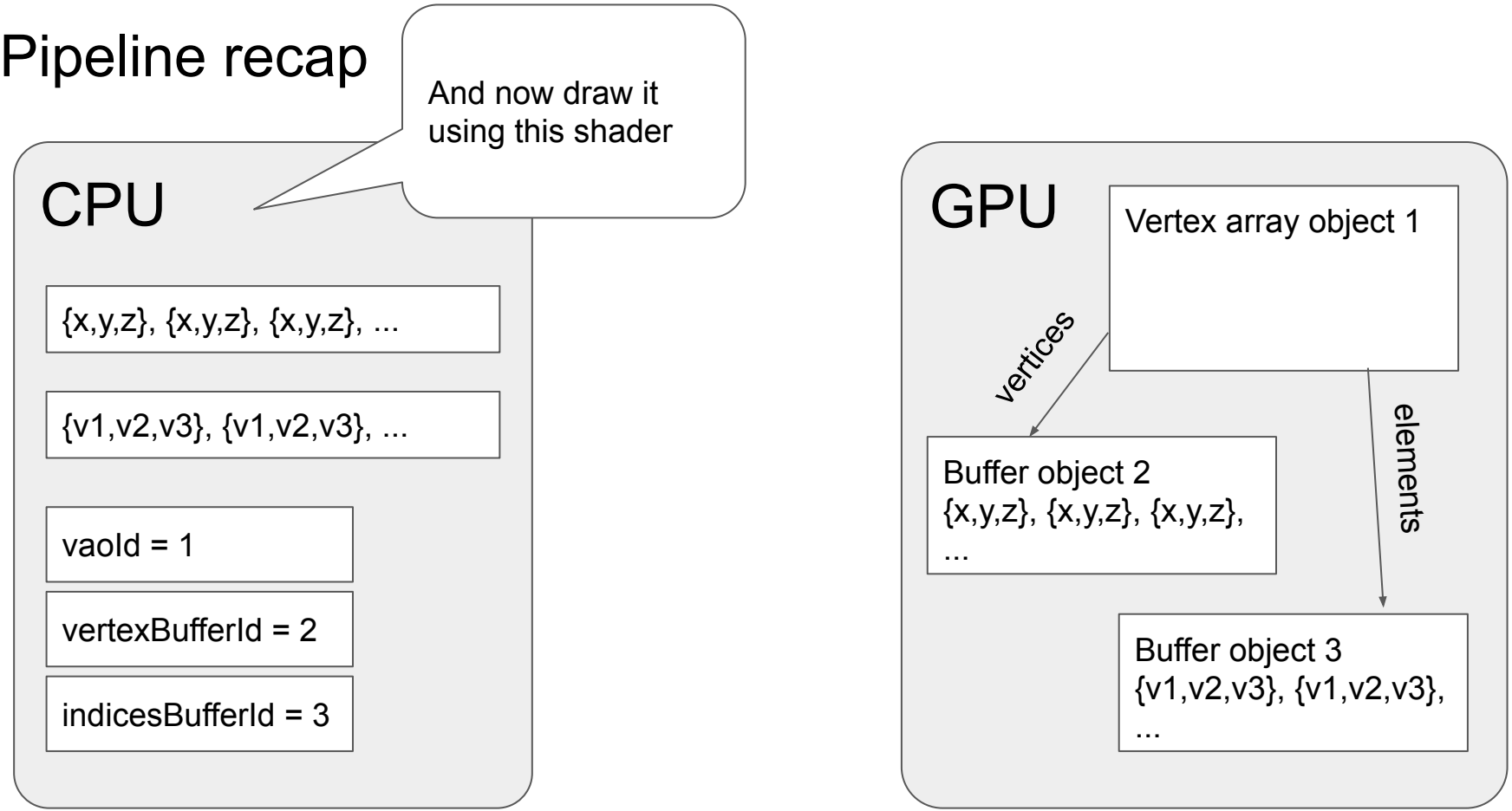
Pipeline recap



Pipeline recap



Pipeline recap



Pipeline recap

Here you go

CPU

{x,y,z}, {x,y,z}, {x,y,z}, ...

{v1,v2,v3}, {v1,v2,v3}, ...

vaoId = 1

vertexBufferId = 2

indicesBufferId = 3

GPU

Vertex array object 1

vertices

elements

Buffer object 2
{x,y,z}, {x,y,z}, {x,y,z},
...

Buffer object 3
{v1,v2,v3}, {v1,v2,v3},
...

Where does the vertex info come from, anyway?

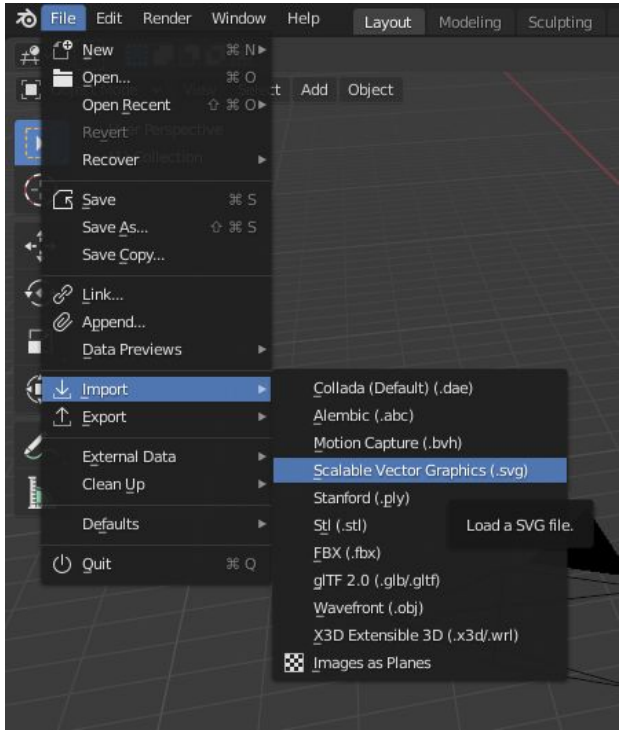
- Hard-coded (e.g. if you just need a square)
- Dynamically generated
- Imported from modelling software

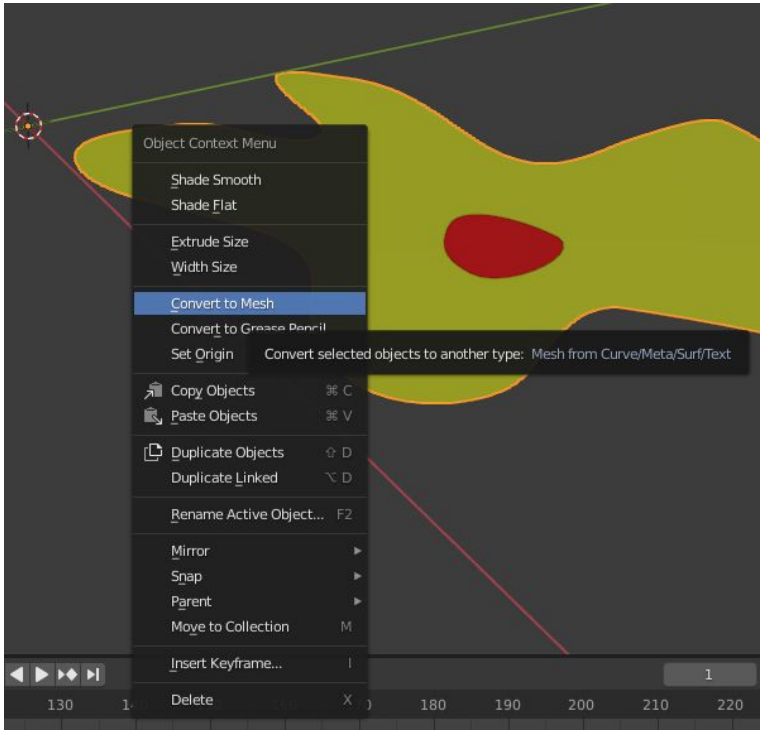
Importing meshes: obj files

```
# vertex positions (and optionally colors) are specified with:
v -0.5 2.0 -0.2 1.0 0.0 0.0
# This will be vertex 1, and the next one will be vertex 2, etc

# texture coordinates are specified with vt:
vt 0.2 0.8
# This will be texture coord 1

# faces are specified as the set of vertex indices around the face:
f 1 2 3
# optionally with texture coordinate indices after a / too:
f 1/1 2/2 3/3
```

Making obj files in Blender

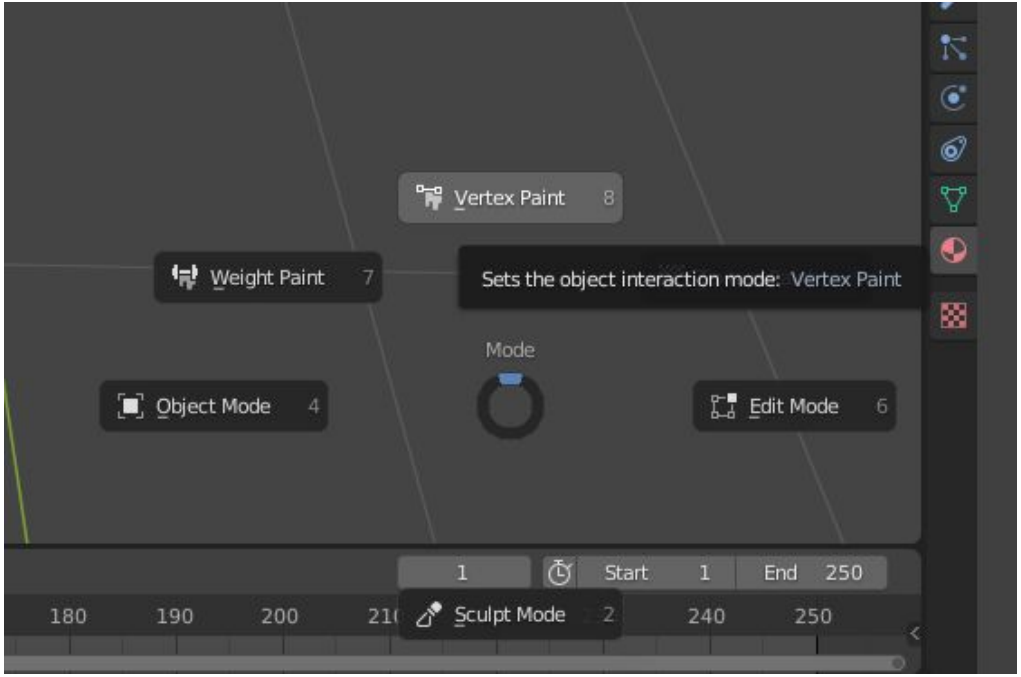
1. Either make a shape in Blender or make an SVG somewhere and import it





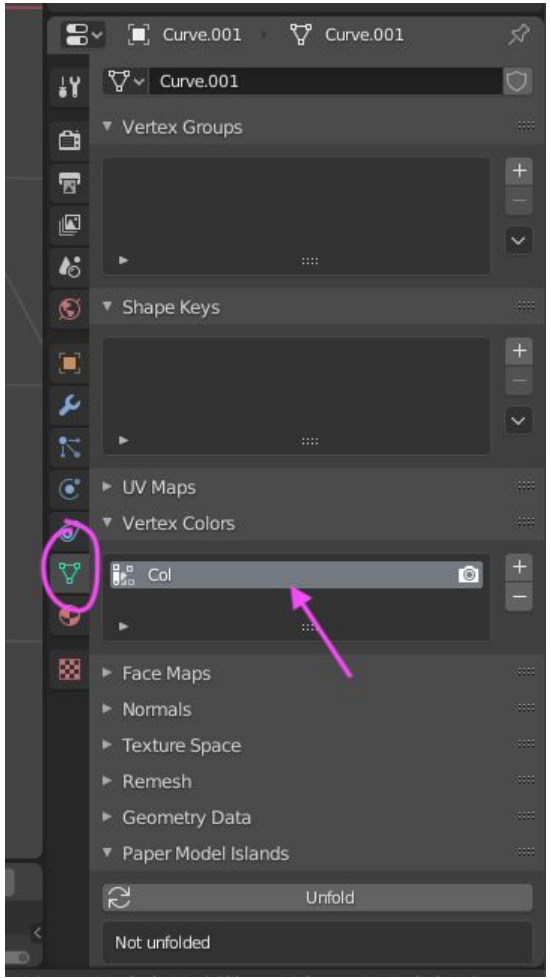
Making obj files in Blender

2. Hit Ctrl-Tab to go into Vertex Paint mode:



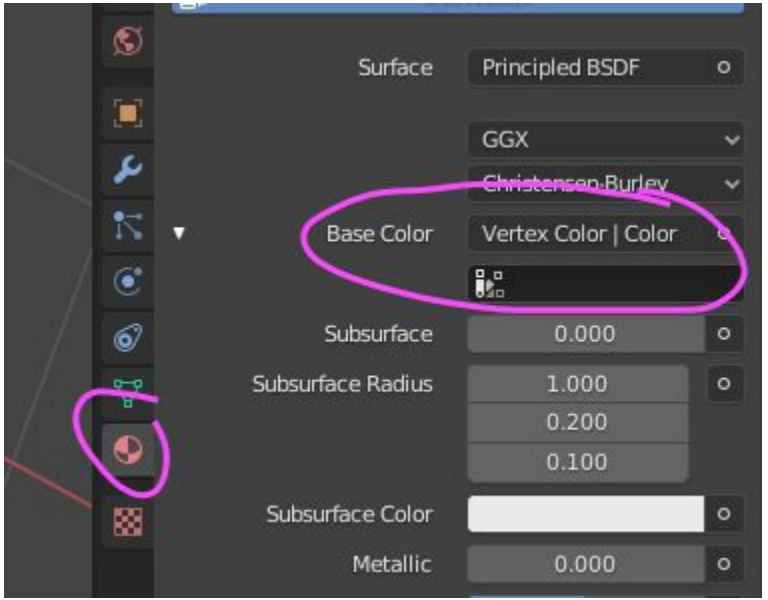
Making obj files in Blender

This creates an empty set of vertex colours for the mesh:



Making obj files in Blender

3. Change the object's material so we can see the colours that we're going to add by setting the base colour to the vertex colours:

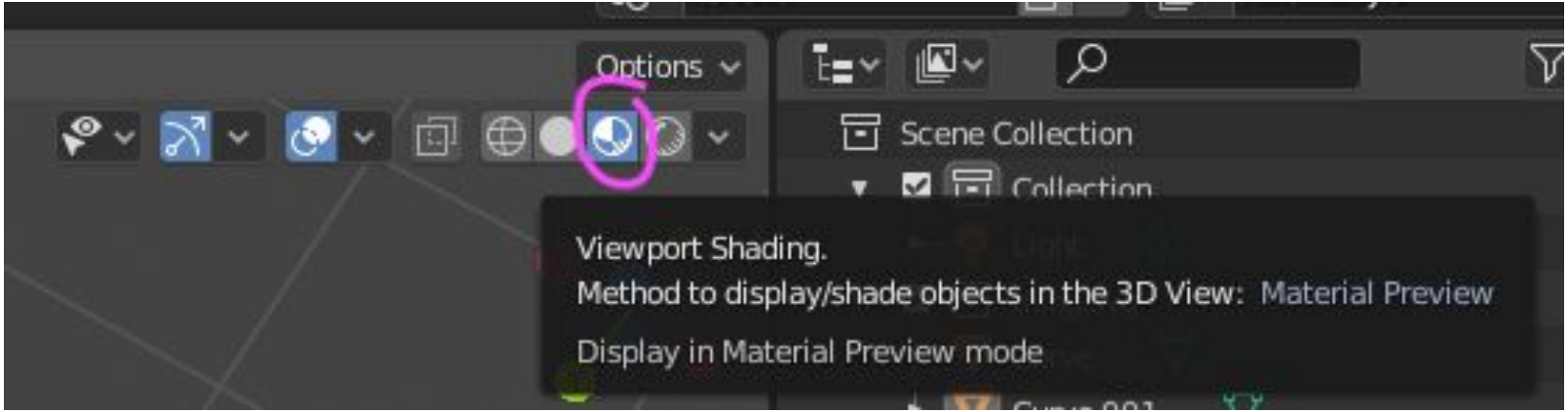


Making obj files in Blender

4. Use Vertex Paint mode to paint the vertices the colours you want

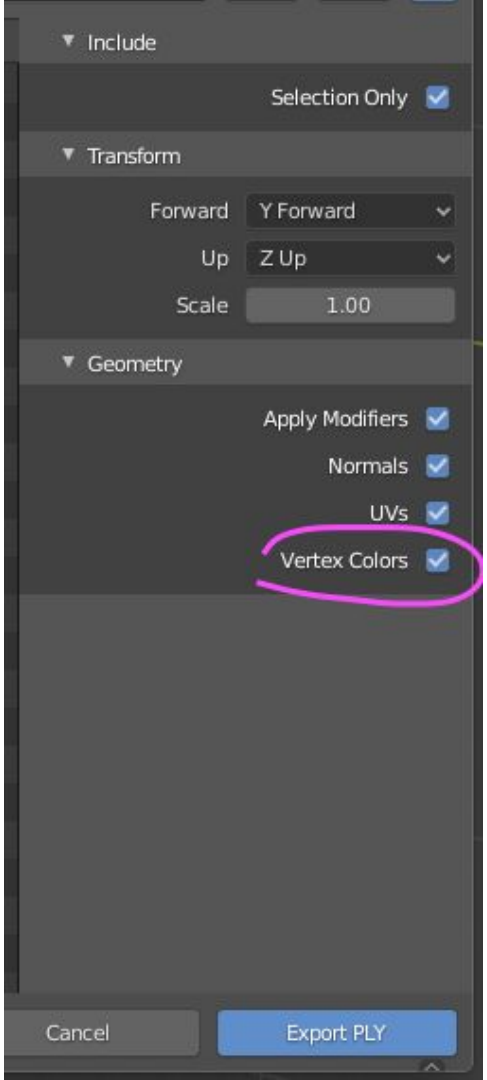
Making obj files in Blender

5. If you ctrl-tab back into Object Mode, use viewport shading to see the colours



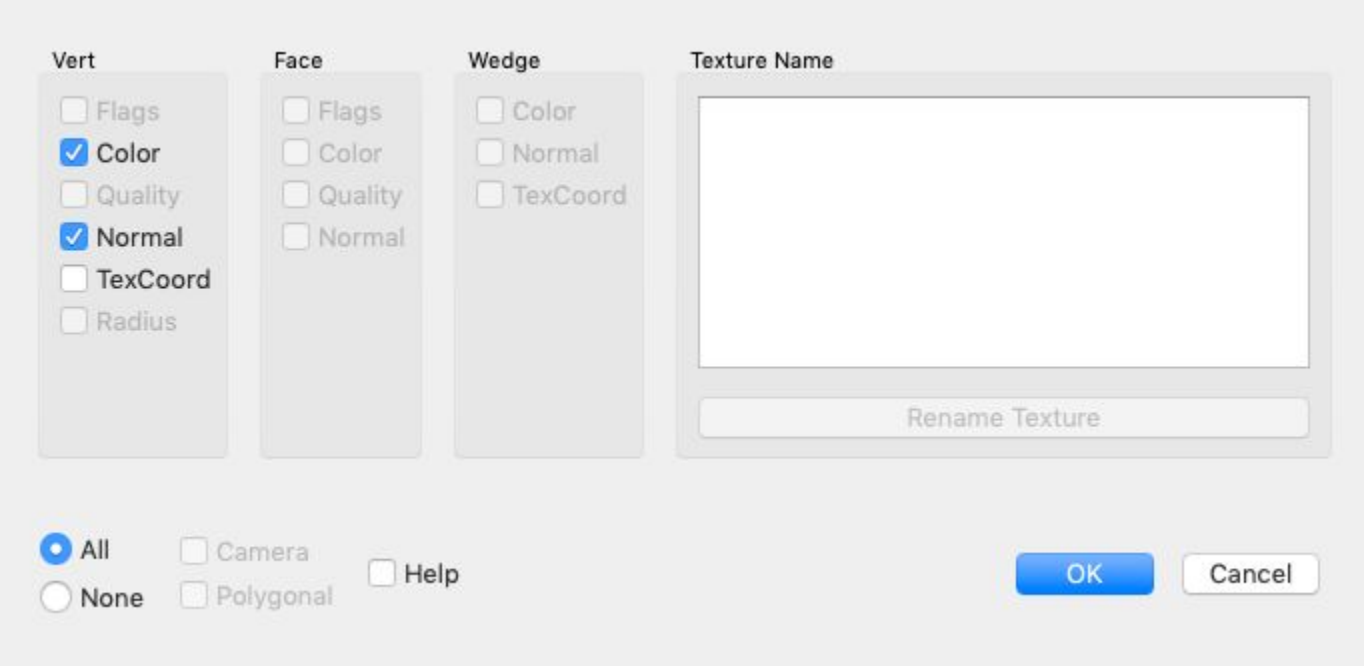
Making obj files in Blender

6. Blender's .obj exporter doesn't actually support vertex colours, but its .ply exporter does! Export a .ply instead:



Making obj files in Blender

7. Convert the .ply to a .obj using **MeshLab** by doing File → Export Mesh As



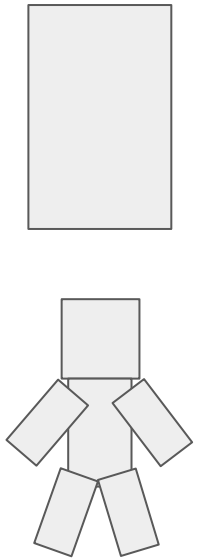
Making obj files in Blender

8. Put that .obj in your data directory and use it in your game!

Another alternative

- Decompose your character into multiple sprites
- The character has one transform, and each body part has its own:

```
// textured.vs.glsl
vec3 pos = projection * transform *
  vec3(in_position.xy, 1.0);

// your_character.vs.glsl
vec3 pos = projection *
  character_transform * part_transform *
  vec3(in_position.xy, 1.0);
```



Depth sorting multiple meshes

Ways of depth sorting

- Using OpenGL's depth buffer
  - OpenGL does it for you!
  - ...but you need to discard totally transparent sprite pixels yourself
  - ...but you still need to draw semi-transparent things in order
- Using the painter's algorithm
  - glDisable(GL_DEPTH_TEST);
  - Draw things in back-to-front order

Painter's algorithm in ECS

- In tiny_ecs.hpp, we have `ComponentContainer::sort(comparisonFunction)`
- `comparisonFunction(a, b)` returns whether a comes before b in the list

e.g.:

```
struct Depth { float depth; }; // or add to Motion
ECS::registry<Depth>.sort([](const Depth& a, const Depth& b) {
  // Higher z → farther away?
  bool should_a_draw_before_b = false; // FIXME
  return should_a_draw_before_b;
});
```

NOTE: removing entities will reorder components, so you need to sort again afterwards!

Questions?