Last week

- We learned about using functions in GLSL
- We created a bunch of functions for drawing squares and circles
- We also looked at how we can draw lines, and how to use expressions to generate patterns

This week

- We will look at how to generate random numbers in a shader
- We will also explore how we use these to make interesting textures
- We will extend this in the lab next week

This week

- We will also look at how vertex shaders work
- We will use vertex shaders to modify geometry
- We will also learn how to share information about geometry between vertex and frag shaders
    - This will allow us to create our own custom lighting that is aware of the shape

RECAP

- You should now feel confident using functions in GLSL to create shapes and patterns

- You should also be able to calculate rotations, translations and scale shapes in 2D using matrices. Don’t be shy…

- If you are still not sure about these things, you need to go back to the lab material in your own time. It will slowly get easier the more you practice

RECAP

- Let’s look at some of the lab work a bit:
- https://mimicproject.com/code/ba6a2bbd-dd43-e661-da51-ec36fccbc0d3
- https://mimicproject.com/code/0e84f212-142e-8cef-b427-fa1d8492f368

NOW

- Before we look at vertex shaders in detail
- We need to learn something new

NOISE

NOISE

- What is noise?
- How can we describe and understand it?
- Where do we experience noise?
- What do we know about noise?

NOISE

- How can computers generate noise?
- Is this something they do naturally?
- How can a computer make better noise?

NOISE

- In computer graphics we use noise all the time
- It’s an essential part of making everything look and feel more natural (why?)
- We use noise in textures, objects, lighting and simulations, including most forms of AI

NOISE

- In order to use noise in our shaders, we need to be able to calculate randomness
- How do we calculate randomness?
- What is ‘random’?

NOISE

- There may well be no such thing as ‘RANDOM’
- When something is random, this just means we can’t detect a pattern.
- This makes the whole problem of noise much easier to grasp
- It also helps us understand the world a lot better!

NOISE

- Computers are not random
- They follow instructions
- Can a computer be random?
  - How?

NOISE

- So how do we generate noise?

- Let’s think about sound
- Where do we hear noise?
- What kind of noise do we hear?

NOISE

- White noise is a signal where there is equal energy in all frequencies.
- This means that every frequency is represented by the signal, and we don’t associate it with any form of order

NOISE

- We can also ‘shape’ noise in different ways
- There are many different types of ‘noise’ distributions, and they have different trade offs
  - Throwing two dice is a good example - this generates a ‘normal’ distribution (you should remember this from school). But to model this, you need a generator.
  - Brown noise, pink noise - 1/f noise can all generate interesting noise - sometimes fractal in nature. 1/f noise is a good model of natural variation.

NOISE

- When you turn on an old analogue radio, what do you hear?
- Noise…
- Lots of frequencies interacting without a pattern

NOISE

- SO..logically, we can use techniques from radio to generate noise signals

- We can use analogue radio modelling to create random number generators!

- These are really great random number gens but they aren’t talked about much.

NOISE

- Two common methods for radio transmission are AM and FM
- AM = Amplitude modulation
- FM = Frequency modulation

NOISE

- Both methods require the following:
  - A carrier (a sine wave)
  - A modulator (a modulating wave)
  - A modulation index (the amplitude of the modulator)

NOISE

- Both the carrier and the modulator have a frequency
- In amplitude modulation, we modulate the amplitude of the carrier by the frequency of the modulator.
- In frequency modulation we modulate the frequency of the carrier by the frequency of the modulator
    - In both cases the modulation index (the modulator's amplitude) have a BIIIIG impact. Little modulations are tiny changes. BIG modulations are BIG changes.

NOISE

- Let's take a look at an example using FM to generate randomness in a shader.

- https://mimicproject.com/code/40f1b8ac-167d-95c5-7f4d-9d129bebb1b1

NOISE

- Now let’s look at methods for 2D noise generation using FM

- https://mimicproject.com/code/b127c4fb-e76f-76e1-09ab-18973a61eedd

- To begin with it might not look noisy but wait..

NOISE

- We can do very interesting things by using the integer or fractional parts of our coordinate systems.

- By using both at the same time, we can have randomness, and order, both at the same time.

- This is powerful - and what you need to do in order to create more natural looking images.

NOISE

- The problem with 1D noise approaches is that they don't calculate noise across both xy dimensions, so you can see lines and repetitions on both axes

- By using the dot product (again), we can get noise values for fragments based on their magnitude

  - dot product takes 2D input and provides 1D output

    - The magnitude when both terms are the same

- If we use it as the input to our FM generator, it produces interesting effects in 2D

NOISE

- Including the one you've been looking at…
- https://mimicproject.com/code/15f43bb8-fecb-1c9e-a47c-8141944f4190

Vertex Shaders

- At the end of the lab, I asked you to take a look at the vertex shader example
- Let’s go through it now.
    - https://mimicproject.com/code/8028d479-ec2f-e8ca-0bc8-891a4669d66f

Vertex Shaders

- In summary
  - We have two interacting shaders:
    - Vertex shader
    - Fragment shader
- Together they are used to create a ‘**MATERIAL**’

Vertex Shaders

- The vertex shader allows us to pass on and modify aspects of openGL meshes
- The final output is gl_Position()
    - This is the equivalent of gl_FragColor()
- gl_Position is a vec4, xyzw
- w is the ‘homogenous’ coordinate
    - We divide xyz by w. It’s very useful…

Materials

- Materials are bound to a specific mesh
    - So, your shaders get bound to a specific mesh - such as your model, plane or sphere - as part of the material
- You can have quite a lot of shaders in memory, but only one is active at any one time - this impacts how you use them

Vertex Shaders

- All your meshes can be processed procedurally without issue inside the draw loop
    - Bind your material to your mesh
    - Draw the mesh
    - Unbind your material
    - Bind a new material to new geometry
    - Draw a new mesh
    - Unbind…..

Uniforms and Vertex Shaders

- Vertex shaders have the same uniforms as fragments shaders
- You should use high precision uniforms when sharing them between frag and vert shaders
- Otherwise your shader code might not run

In practice

- Some programmers just write one big vertex and frag shader pair
- They then specify a variable that sets which part of both shaders should be active for each piece of geometry
- They then bind just that single pair of shaders to a single material for all the geometry
    - Then pass a uniform in to select which part of each shader should be active, then draw the specific geometry, then change the active shader sections, then draw the next bit of geometry…etc.

Varyings

- Vertex shaders can have ‘varying’ variables
- These can be passed to fragment shaders
    - They need to be defined in both the vertex and the fragment shader in order for this to work

IN and OUT

- Varyings are deprecated in openGL 4
- They have been replaced by ‘in’ and ‘out’
- But both methods are actually used all the time
    - Let’s look briefly at the openFrameworks examples

Attributes

- Attributes are the vertex shader **INPUTS**
- Essentially, these are the elements of the
- **VERTEX BUFFER OBJECT (VBO)**
    - The VBO contains the elements of the mesh that you have bound to the material. They get passed to the GPU and can be accessed as vertex shader attributes
- This means you can get all the data streams from the mesh as attributes, modify them, and replace them with new data :-)

- **There are a bunch of built in attributes that relate to the VBO**

gl_Vertex Position (vec4)

gl_Normal Normal (vec4)

gl_Color Primary color of vertex (vec4)

gl_MultiTexCoord0 Texture coordinate of texture unit 0 (vec4)

gl_MultiTexCoord1 Texture coordinate of texture unit 1 (vec4)

gl_MultiTexCoord2 Texture coordinate of texture unit 2 (vec4)

gl_MultiTexCoord3 Texture coordinate of texture unit 3 (vec4)

gl_MultiTexCoord4 Texture coordinate of texture unit 4 (vec4)

gl_MultiTexCoord5 Texture coordinate of texture unit 5 (vec4)

gl_MultiTexCoord6 Texture coordinate of texture unit 6 (vec4)

gl_MultiTexCoord7 Texture coordinate of texture unit 7 (vec4)

gl_FogCoord Fog Coord (float)

In, Out, Attribute, Varying

- in: link to shader from previous stage in pipeline
- out: link out of a shader to next stage in pipeline
- attribute: same as in for vertex shader
- varying: same as out for vertex shader, same as in for fragment shader

Using a varying to create a light

- We can get the normals out of the vertex buffer and use them to set the colour of the fragment shader at any position

- This is super easy and you already know how to do it

- But this time we will do it in pure 3D shader language from the vertex buffer…

Using a varying to create a light

- https://mimicproject.com/code/3ef97bc0-86b7-cced-40b7-3210a62a5475

RECAP

- You should now feel like you understand the idea of how to generate lighting effects using normals
- You should also be feeling confident that you can generate noise using simple approaches
- You should have some idea why noise is useful

Using a varying to create a light

- We can get the normals out of the vertex buffer and use them to set the colour of the fragment shader at any position

- This is super easy and you already know how to do it

- But this time we will do it in pure 3D shader language from the vertex buffer…

RECAP- VERTEX SHADERS

- In summary
  - We have two interacting shaders:
    - Vertex shader
    - Fragment shader
- Together they are used to create a ‘**MATERIAL**’

RECAP- VERTEX SHADERS

- The vertex shader allows us to pass on and modify aspects of openGL meshes
- The final output is gl_Position()
    - This is the equivalent of gl_FragColor()
- gl_Position is a vec4, xyzw
- w is the ‘homogenous’ coordinate
    - We divide xyz by w. It’s very useful…

RECAP - Materials

- Materials are bound to a specific mesh
    - So, your shaders get bound to a specific mesh - such as your model, plane or sphere - as part of the material
    - You can have quite a lot of shaders in memory, but only one is active at any one time - this impacts how you use them

RECAP = Uniforms

- Vertex shaders have the same uniforms as fragments shaders
- You should use high precision uniforms when sharing them between frag and vert shaders
- Otherwise your shader code might not run

RECAP - Varyings

- Vertex shaders can have ‘varying’ variables
- These can be passed to fragment shaders
    - They need to be defined in both the vertex and the fragment shader in order for this to work

RECAP - IN and OUT

- Varyings are deprecated in openGL 4
- They have been replaced by ‘in’ and ‘out’
- Both methods are actually used all the time
- In is picking up an input, outs need to be picked up by an output.
    - The end of the chain needs to be gl_Position and gl_FragCoord.

In, Out, Attribute, Varying

- in: link to shader from previous stage in pipeline
- out: link out of a shader to next stage in pipeline
- attribute: same as in for vertex shader
- varying: same as out for vertex shader, same as in for fragment shader

Attributes

- Attributes are the vertex shader **INPUTS**
- Essentially, these are the elements of the
- **VERTEX BUFFER OBJECT (VBO)**
    - The VBO contains the elements of the mesh that you have bound to the material. They get passed to the GPU and can be accessed as vertex shader attributes
- This means you can get all the data streams from the mesh as attributes, modify them, and replace them with new data :-)

- **VERTEX ATTRIBUTES (default ones)**

gl_Vertex Position (vec4)

gl_Normal Normal (vec4)

gl_Color Primary color of vertex (vec4)

gl_MultiTexCoord0 Texture coordinate of texture unit 0 (vec4)

gl_MultiTexCoord1 Texture coordinate of texture unit 1 (vec4)

gl_MultiTexCoord2 Texture coordinate of texture unit 2 (vec4)

gl_MultiTexCoord3 Texture coordinate of texture unit 3 (vec4)

gl_MultiTexCoord4 Texture coordinate of texture unit 4 (vec4)

gl_MultiTexCoord5 Texture coordinate of texture unit 5 (vec4)

gl_MultiTexCoord6 Texture coordinate of texture unit 6 (vec4)

gl_MultiTexCoord7 Texture coordinate of texture unit 7 (vec4)

gl_FogCoord Fog Coord (float)

Dynamic Geometry

- Vertex shaders are fantastic for creating dynamic geometry
- They are fast, and none of the computation is done on the CPU - it’s all GPU
- This allows us to create beautiful, powerful dynamic scenes very quickly

Dynamic Geometry

- IN order to modify the positions, we need to take the default position attribute, modify it, then store it in a new vector
- Then we have to pass this new vector into gl_Position
- This will place the vertex somewhere else. But we also need to get the projection and modelView right.
    - Luckily, both three.js and openFrameworks make that part very easy

Basic Example

- https://mimicproject.com/code/70a1462e-ff8b-381e-3389-536e8f40ca8c

Dynamic Geometry

- But in order to see this geometry properly, we need to recompute the normals - otherwise we can't light it properly.

- Once the normals are recomputed we can send them to the fragment shader

- We also need to recompute normals when we decide to rotate or scale the geometry

Dynamic Geometry

- The normals are automatically made available to the vertex shader by default in three.js and openFrameworks.
- This doesn’t always happen - if you are working without any framework or middleware, you will need to set this stuff up manually
- in three.js, the normal for each vertex is held in a vec3 called ‘normal’. You don’t need to declare it.
    - But you would have to if we weren’t using three.js

Dynamic Geometry

- So we need somewhere to store the new position

  vec3 newPosition;

- We can then write the actual position values into the newPosition vector, and change them in some way

  vec3 newPosition = position * something;

Dynamic Geometry

- We also need to create a varying called myNormal
- This should be declared at the top

```
varying vec3 myNormal;
```

- Then we can create a new vec3, called my normals and use this to send the normals to the fragshader

```
myNormal = normal ;
```

Dynamic Geometry

- Once we have the new positions, we can use the normalise function to get the unit vector of the new positions
- As we've understood a few times now, this is the new normal that we can use for the lighting in the fragshader

```
myNormal = normalize(newPosition);
```

Dynamic Geometry

- When we rotate the geometry - or move it in any way, we also need to recompute the normals.

- So how do we rotate all the positions in any case?

- er…

Dynamic Geometry

- Yep we can use a rotation matrix.
- Here’s a rotateX matrix.

mat4 rotateX =
mat4(1,0,0,0,0,cos(angle),sin(angle),0,0,-
sin(angle),cos(angle),0,0,0,0,1);

Dynamic Geometry

- So we can rotate each position vertex as follows:

```
vec4 rotatedPos = rotateX * vec4(newPosition,1.0);
```

And then get new normals for rotatedPos!

Dynamic Geometry

- Let’s take a look

https://mimicproject.com/code/4f968419-a78e-b4d4-bcc5-2722918b25ed

Dynamic Geometry

- We can also scale the geometry using a scaling matrix.
- Notice however, that we rotated the geometry earlier in the code, and scaled later - we should be careful about this.
- Remember - scale, rotate, then translate
- And compute these in reverse order
- TransformedVector = TranslationMatrix * RotationMatrix * ScaleMatrix * OriginalVector;

Dynamic Geometry

- Also, notice that I did something new in the fragshader to improve the lighting.
- I used the max function to clip the low threshold so that it was't dark at the bottom, but instead defaulted to an ambient colour
- You could add any colour, shader idea, whatever that you want!

Dynamic Geometry

- We can create a bump map by adding a texture, or random values to this normal
- As you might remember, this alters the look of the surface, without changing the surface.
- very basic example:

  https://mimicproject.com/code/9c8c6243-0ff3-ebf7-d217-3768dc7c13e1

Dynamic Geoms

- So we can rotate, scale, translate and modify vertices
- We can do this for any geometry that we like
- In the labs, we will do this for a ground plane, and also for an object
    - We will then have our own sandbox
- **But we might need more interesting algorithms to modify the ground plane**

Mandy

- We can easily compute a non-linear geometry in the shader, and use this to create a height map.
- One idea would be to create a mandelbrot in a shader, and use the colours as a z value
- In order to give you an idea how to do this, we're going to look at the Mandelbrot code in the fragshader
  - you would have to compute this in the vert shader in order to modify geometry…

Mandy

First, we compute c

```
vec2 c=(gl_FragCoord.xy-resolution/2.)/resolution.x*4.;
```

Mandy

- Then we iterate any number of times to generate the colour for the shader. Let's take a look in detail at the comments

- Remember, it's z=z^2+c, until the result is over 2.

- The amount of times we have to do this for each fragment, is the value we use to colour the fragment.

Mandy

https://mimicproject.com/code/19644e9b-37e3-20de-6c9a-4bc3a5b47fd2