PMC Week 6

Introduction to GLSL Fragment Shaders

RECAP

- We've learned how to draw shapes in 2D and 3D using functions / algorithms.
- We've learned how to generate and modify 3D vertices, set their colours, their normals, set up lighting, and do transformations.
- We've also learned how to apply textures, and map them to geometry
- We've all had a chance to build our own 3D sandbox.

Shaders

- This week we will start to understand the basics of GLSL, what it is, how it works, and why it's useful.
- Then we will start exploring the fragment shader.
    - Specifically, we will be drawing basic shapes in ways that we should now be used to.

Things to keep in mind

- Position (x,y, or θ,r)
- Size / Scale
- Difference
- Distance

GLSL - Basic Concepts

- Open Graphics Library
- C-type language
- You can use if statements, for loops, #define etc.
    - #define can be very useful in GLSL

GLSL - Basic Concepts

- GLSL has LOADS of cool built in functions, including lots of mathematical functions that are useful for doing computer graphics. Most of these should be familiar to you.

- It also has a number of built-in types. Some of them will be familiar, others won't. So we need to look at these.

GLSL - Basic Concepts

- There are other types of Shader Language
- HLSL for example
- But GLSL works on the most systems
    - So it’s what you should learn first. HLSL is not much different

GLSL - Basic Concepts

- GLSL compiles at runtime
- This is because different graphics cards interpret GLSL differently, so you can’t distribute a runnable binary.
- This means that you need to distribute the source code with your application

GLSL - Basic Concepts

- The shader has to be loaded in your main application, just like any other asset

- If you want to interact with your shader, you need to use special variables that you pass into the shader from your main program

- You also need to do this for any textures (images) that you want to do shader operations with

GLSL - Basic Concepts

- GLSL is VERY fast
- Much faster than any other approach you have used before
- This is because of the GPU architecture that GLSL runs on

GLSL - Basic Concepts

- Computations on the GPU are run ***in Parallel***
- For example, let's consider a '**Fragment'** shader
- The screen is divided into fragments (sections)
- Each fragment runs the fragment shader program ***simultaneously***
  - ***A fragment can be a single pixel…***

GLSL - Basic Concepts

- This means that all the drawing techniques you have learned cannot be executed in the same way
- In fact, you need to learn an entirely new way of thinking about graphics programming
- But once you've got the basics, you will see that it is much more powerful and efficient.

GLSL - Basic Concepts

- The two most important variables in a fragment shader are as follows:

gl_FragCoord

//gl_FragCoord is the current 2d fragment coordinate

gl_FragColor

//gl_Fragcolor is the current Fragment's **3d or 4d colour in normalised units.**

GLSL Types

- GLSL has lots of built in types.
- Some of them might be familiar and work in a similar way
  - float, int, bool, array, struct;
- Others are completely new.Also they are ***AWESOME***
- vec2, vec3, vec4, ivec2, ivec3, ivec4, bvec2, bvec3, bevc4
- mat2, mat3, mat4

Types: floats, Ints, bool

- In general, ***most graphics cards can't compile shaders that use integer literals***
- This means that if you use a number, it should always be a float.

```
//right

distance *=2.0;

//WRONG

distance *=2; // This integer literal is going to break you
```

GLSL vectors

- vec2 is a 2D vector
- You can define a vec2 as follows:
  vec2(0.5,0.5);
- One great thing about vecs is that you can do math operations directly on them.

```
vec2 output = vec2(0.5,0.5) * 10.;

vec2 output = vec2(0.5,0.5) + 20.;
```

GLSL vectors

- You can access elements of a vector using .x, .y, .z

```
  vec2 output = vec2(0.5,0.25) * 10.;

// output.x now equals 5.0

// output.y now equals 2.5
```

vec3, vec4

- vec3 is a 3D vector
- vec4 is a 4D vector
- In GLSL, we usually define colours using 3D and 4D vectors (we also use them for lots of other stuff)
- You can specify colours in either 3D or 4D

```
gl_FragColor = vec4(1.0,1.0,1.0,1.0);

// The above sets the colour of the current fragment
```

GLSL Vectors

- Another great thing about GLSL vectors is that you can initialise all the cells in a vector with a single number

```
vec3 myVec = vec3(0.5); //myVec = (0.5,0.5,0.5)
```

- ***Yet another*** awesome thing about GLSL vectors is that you can initialise them with vectors of different sizes.

```
vec3 3dvec = vec3(0.5,0.5,0.5);

vec2 myVec = vec2(3dvec);

vec2 myVec2 = vec2(3dvec.x, 3dvec.y);
```

GLSL vectors

- Yet another awesome thing is that you can do cellwise vector operations without having to write loops.

```
vec2 myVec1 = vec2(2.,3.);

vec2 myVec2 = vec2(10.,10);

vec2 myVec3 = myVec1 * myVec2;

//myVec3 is now (20.,30.)
```

GLSL vectors

- The final awesome thing is that you can select and combine elements of a vector using x,y,z,w notation and do maths with them.

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);

pos.xw = vec2(5.0, 6.0); // pos = (5.0, 2.0, 3.0, 6.0)

pos.wx = vec2(7.0, 8.0); // pos = (8.0, 2.0, 3.0, 7.0)

//SWIZZLE ! wzyx yxzw xzxz
```

Other types of vectors

- You can use bools as you might expect.
- You can also have boolean vectors

bvec3(true,false,true);

- And Int vectors

ivec4(1,1,1,1);

GLSL Matrices

- As you know, matrices are very handy for doing rotations, as well as consistently computing scaling and translation
- You can define 2 * 2, 3 * 3 and 4 * 4 matrices
- mat2, mat3, mat4

GLSL Matrices

- GLSL matrices default to **COLUMN MAJOR**
- This means the are defined column after column

```
mat4 myMat = mat4(1.0, 0.0, 0.0, 0.0,  // 1st column
                  0.0, 1.0, 0.0, 0.0,  // 2nd column
                  0.0, 0.0, 1.0, 0.0,  // 3rd column
                  0.0, 0.0, 0.0, 1.0); // 4th column
```

GLSL Matrices

- Here's an example of how to define a rotation matrix

```
mat2 rotation = mat2( cos(angle), sin(angle),-sin(angle), cos(angle));
```

- You can simply multiply any 2D vector by this mat2 to rotate that vector

GLSL Matrices

- Finally, you can access and set elements of a matrix as follows

```
mat4 m;

m[1] = vec4(2.0); // sets the second column to all 2.0

m[0][0] = 1.0; // sets the upper left element to 1.0

m[2][3] = 2.0; // sets the 4th element of the third column to 2.0
```

Uniforms

- A uniform is a variable of ***any type***. that is passed into the shader from your main program

- The uniform in the shader **must have** the same name as **you defined** in your main program

- Common uniforms include the **screen size** / **resolution x,y**, the **mouse x,y** positions, and a **time** variable (which you can use to compute animations).

Uniforms and textures

- Images can be passed into fragment shaders and accessed as uniforms with the sampler2D type
- In order to do this, you need to set up the texture input in your main program (we're not doing this today)
- You would then set up the texture like this:

```
uniform sampler2D myTexture;
```

Frag Shaders: Drawing Shapes

- Let’s look at some basic fragshaders
- We’ll start off with the basics
- Then we’ll make some shapes.

Frag Shaders: Basics

```
// Start by setting the precision!
precision mediump float;

// You always have to do this first.
```

Frag Shaders: Basics

```
// Now we specify some basic uniforms

uniform vec2 resolution;

uniform vec2 mouse;

uniform float time;

// These are all already defined in our main
program. We are just picking them up here
```

Frag Shaders: Basics

```
void main() {
    gl_FragColor = vec4(abs(sin(time)),0.0,0.0,1.0);
}

// We could try loads of different math operators
```

Frag Shaders: Basics

```
void main() {
    vec2 colour = gl_FragCoord.xy/resolution;

    gl_FragColor = vec4(colour.x,colour.y,0.0,1.0);

}
```

Frag Shaders: Shapes

- So how do we go about drawing a simple image?

- We'll begin with drawing a circle, as this is much easier than you might think

- There are a number of different ways of doing it, but the principles are basically the same

Frag Shaders: Shapes

- The first thing we want to do is figure out where we are going to draw our circle.
- Let's draw it in the middle. We can create a vector that contains this point:

```
vec2 pos = resolution.xy/2.0;
```

Frag Shaders: Shapes

- Now all we have to do is get the distance from the centre to the current fragment (pixel), and then use this to determine a colour
- We can do this with the distance function

```
colour = distance(gl_FragCoord.xy,pos);
```

Frag Shaders: Shapes

- We can literally just use this colour to see something that looks a bit like a circle

- But it’s not a circle. What is it?

Frag Shaders: Shapes

- It's a ***DISTANCE FIELD***
- Distance fields are the primary method for generating shapes in fragment shaders.
- This is because the only thing the shader really knows is where it is on the screen
- So, all programs we write need to use more or less **JUST** this information to work out what colour they should be.
- You can work this out by setting up distance fields for all the objects in your scene, and then making decisions about colours based on these distances.

Frag Shaders: Shapes

- In order to turn the distance field into a circle, we just need to set the colour only if the distance is over a certain amount
- This amount is going to be the RADIUS of our circle.
- There are a few ways of doing this

Frag Shaders: Shapes

if (colour > 1.) colour = 1.;

- Or we can use the *step* function

```
colour = step(colour,1.);

// What is the step function doing?

// What does the value 1.0 represent?
```

Frag Shaders: Shapes

- Another method is to get the squared distance.
- This is a really cool method that makes use of the dot product
- It is the product of the Euclidean magnitudes of the two vectors and the cosine of the angle between them.

Frag Shaders: Shapes

- We can use the built-in GLSL dot function
- It’s really fast!!

```
// This calculates the difference between the centre of our circle
to the current pixel - B-A.
    vec2 pos = gl_FragCoord.xy - resolution / 2.0 ;

    // This gets the squared distance using the dot product

    float dist_squared = dot(pos, pos);
```

Frag Shaders: Shapes

- Other interesting functions
- Interpolate a boundary

```
smoothstep(start,end,input);

//You will literally use this all the time..
```

Some other important techniques

- **Polar coordinates**
  - You can use polar coordinates to calculate pixel colours
  - This lets you make polar shapes just like we have done, only the shapes are already filled

Some other important techniques

- Using **modulo**
- This allows us to divide up the coordinate space
- This means we can draw lots of version of the same thing.
- This also works in 3D

Rotations

- You can also use matrices as I mentioned…