# Monday

## 6:30-7:30 Read project brief

## 7:30-8:15 Research what Linux is

Linux is an operating system that began development in 1960 as UNIX.

## 8:15-9:30 Research how to build for Linux

You can run a Linux subsystem on Windows without emulator or VM. You can use this as a server to build and debug on. I’m not sure if you can run it on the subsystem but you should be able to. You can also use the Raspberry Pi to build on it, but that would be slow(er). I will probably move the built files to the Raspberry Pi and run it on.

## 10:30-12:00 Install Linux subsystem to compile the project

It’s not working, getting many errors like rsync and something not found. Will probably compile on the Raspberry Pi.

## 12:30-15:30 Set up the framework

## 15:30-18:00 Get models drawn on the screen

I’ve been researching the project. To get models on screen, I have to implement the model draw and load logic. Brian is telling us to first look at using tinyobj (or something like that). It apparently is used to load wavefront object files. I will research this parser (if that is the correct term) and after that look into how OpenGL does rendering to implement the drawing logic of the model.

## 19:00-19:15 Research TinyObjLoader

I'll have to read into the wavefront file specification. Right now, I'm not sure how meshes are rendered so I will research that.

# Tuesday

## 8:35-12:15 Fix Raspberry Pi’s internet connection

I just decided to reflash the OS. You should not try and enter an IP address in the wifi section. It broke my Pi.

During reinstall it got stuck on updating. I’m totally hating the Pi rn. I’ve already been trying to fix it for 2 and a half hours because it is sooo slow. My Pi kept crashing when I unplugged the mouse. I don’t understand why it would.

## 13:00-13:15 Research TinyObjectLoader to import object files

I’ll first research the OBJ format.

## 13:15-14:15 Implement LoadandConvert

After reading the docs of tinyobjloader, all data for vertices are put inside an attribute object. There is a shapes object which contains indices to the attribute object. We use the shapes object to index all the data.

I don’t know what data is required by OpenGL ES so I can’t really do much with the data

## 15:45-17:30 Research how OpenGL ES draws meshes

* Learned what shaders are and do. How to load and apply them.
* Learned how to draw a basic triangle, though much of it remains a bit unclear, I do understand that we need to load all the vertices and use that as vertexArray.

Honestly, the tutorial that I followed didn’t help that much. It didn’t really discussed what was being changed. I have not spended my time that efficient today.

## 17:30-18:00 Learn how OpenGL renders meshes

* OpenGL initialization is two faces. The first is creating an OpenGL context. A context is everything that OpenGL is and is very platform and language-binding specific. You are advised to use a window toolkit like GLFW for this. The next step is to load all the functions. For some reason, OpenGL doesn’t take the “just include a header file and be over it” approach. But you have (or are advised) to use a loading library like GLEW.

# Wednesday

## 6:55-8:55 Read how to render with OpenGL

Graphics pipeline

* Get vertices
* Use vertex shader on vertices
* Create shape with primitive hint (primitive assembly)
* Use geometry shader on shape
* Rasterize final shape
* Do clipping
* Use fragment shader on shape
* Do alpha test and blending

I understand the reason behind this order, but I don't understand why alpha tests and blending is done after fragment shader. Is this to make it easier to calculate colors when shapes have an alpha value?

OpenGL works with NDC ranging from -1 to 1 on all axises.

Fragments are data that OpenGL needs to calculate a pixels color.

Primitive assembly seems to interpret the vertices counterclockwise.

**Vertex input**

Create memory on GPU for vertex data. Tell gpu how to interpret vertex data. Specify how to send vertex data to GPU. Tell how vertex shader should interpret all the vertices.

The memory is managed with VBO, vertex buffer objects. This allows us to send large batches of vertices to the GPU allowing shaders to access the vertices fast.

VBO has type GL\_ARRAY\_BUFFER.

When copying user data to the VBO (do this with glBufferData), we have 3 different ways of storing it:

* GL\_STATIC\_DRAW: the data will most likely not change at all or very rarely.
* GL\_DYNAMIC\_DRAW: the data is likely to change a lot.
* GL\_STREAM\_DRAW: the data will change every time it is drawn.

**Vertex shader creation**

Shaders are written in GLSL

#version 330 core  
layout (location = 0) in vec3 aPos;  
  
void main()  
{  
 gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);  
}

gl\_Position is an keyword that we can/must give a value.

We first take the version and specify that we are using core.

Layout location will be explained later.

We specify vertex attribute aPos and forward it. Normally positions of the vertices are not in NDC.

We create a shader with glCreateShader, it returns ID.

Shader compiling

Shaders are compiled during runtime (driver dependant?) So we need to write code to compile a shader.

We call glShaderSource to compile which for some reason the last parameter stays null. Second parameter is amount of source codes were passing (can be more than one?).

To check for errors during compilation, run glGetShaderiv which returns state of last compilation. If there was an error, run glGetShaderInfoLog to get error message.

**Fragment shader creation**

#version 330 core  
out vec4 FragColor;  
  
void main()  
{  
 FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);  
}

Fragment shaders only require one output value that we can specify with the out keyword (why not an keyword like gl\_Position here?).

**Shader program**

The shader program links outputs of shaders with other inputs of shaders. This is where linker errors can happen of in/out mismatches.

Call glCreateProgram to get programID.

Call glAttachShader to add shaders to the program. I think order is managed under the hood.

Call glLinkProgram to link the shaders of the program, after all shaders are attached.

glGetProgramiv and log is used to check link status and errors.

To use this new program, call glUseProgram with the ID.

Now delete the shaders with glDeleteShader because they are in the program now.

**Linking vertex buffer attributes**

OpenGL does not know how to interpret the VBO. So we specify it.

Vertex buffer data is stored as follows:

* The position data is stored as 32-bit (4 byte) floating point values.
* Each position is composed of 3 of those values.
* There is no space (or other values) between each set of 3 values. The values are tightly packed in the array.
* The first value in the data is at the beginning of the buffer.

We use glVertexAtrribPointer to specify how each vertex attribute should be interpreted.

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  
glEnableVertexAttribArray(0)

Parameter list:

* Because position is in layout 0, we use 0 here (not sure what and why this is, so will research it more in the future).
* The amount of components in vertex. 3 because it only has position (the VBO)
* Type of the components.
* If it has to be normalized (for colors I guess, we use coordinates).
* The size of the stride, can leave 0 to let OpenGL decide (I think), because VBD is tightly packed with no random data, we can also define it.
* Offset of where to start in buffer. You may be able to put more meshes in one VBO.

This is applies on the last glBindBuffer.

Now we can call glEnableVertexAttribArray with the location. I don't understand this and will research it more.

**Drawing**

This is how drawing looks:

// 0. copy our vertices array in a buffer for OpenGL to use  
glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);  
// 1. then set the vertex attributes pointers  
glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  
glEnableVertexAttribArray(0);   
// 2. use our shader program when we want to render an object  
glUseProgram(shaderProgram);  
// 3. now draw the object   
someOpenGLFunctionThatDrawsOurTriangle();

**Vertex Array Object**

A VAO binds all vertex attribute calls (I don't know what vertex attributes are, will research).

By binding the VAO ,we can easily draw another object by just binding another VAO.

Because the vertexAttribPointer points to the latest bount VBO, it always points to the right VBO (wow that’s pretty clever).

Thus changing our draw call to just binding the program and VAO and then someDrawFunction.

**The OpenGL draw**

Bind the VAO and programs.

glDrawArrays take 3 parameters:

* Primitive type
* Starting index of vertex array
* Amount of vertices

**Element Buffer Objects**

To save on duplicate vertices, EBOs contain indexes to vertices.

The bind is the same as a VBO.

The VAO stores all the glBindBuffer calls. So the VBO and EBO will be bound when binding the VAO.

## 9:30-11:15 Follow workshop regarding block C

## 12:00-14:30 Fill out the learning log

## 14:30-15:00 Reformat work log

## 15:00-16:00 OpenGLES 2.0

We use APIs because GPU hardware keeps changing. It is also manufacturer dependent.

Shaders:

Attribute is a variable (from CPU?)

Uniform is a constant from CPU (doesn’t change for the lifetime of the shader? Or draw call?)

Varying is passed from one shader to the next shader.

Out is the modern version of varying. With in(modern version of in?) you take a value from a shader?

## 17:30-17:55 Research OpenGL rendering a bit more

Vertex attributes are things like position, normal, etc.

The vertexAttribPointer points to the currently bound VBO. When calling this, we have to also enable the vertex attribute. Say that position attrib is located at the beginning of a vertex and is positional data, we specify the vertexAttribPointer with 0, that is the pos of position. We need 3 components so 3 is next. We use float, not normalized.

## 17:55-18:30 Implement .obj loading and rendering of spaceship

# Thursday

## 6:40-7:30 research how to compile for Linux without raspberry pi

I keep coming back to CMake or WSL. Both of these seem a bit rough to just get into and I don't want to risk building a project on a tool I don't understand, but not being able to test my code while traveling, and I travel for 3 hours, is quite a huge hit on progression.

Either way, I've read the basics of CMake and WSL, I will probably try WSL once more because swapping targets for compilation seems really easy. With CMake, I probably have to copy the files manually to the Raspberry Pi which seems like a hassle. I will look into configuring the WSL once more.

I don’t think this can be done. Initially, I wanted to use GLFW so that I can compile for Windows and Linux, but the Pi uses EGL. I don’t know what this is so I probably won’t be able to run it on my laptop.

## 7:30-8:05 Look into what a toolchain is

Toolchain is the tools that are used to get source code to executable. Mostly this is the compiler and linker. Having researched this further, I could use a cross compiler to speed up development on my system. I don’t like the idea of adding the libraries onto the remote system instead of the project. I will probably set up my own project because of this.

## 8:45-10:00 Created own project for Linux

## 10:00-14:15 Render spaceship on Pi

I misunderstood the use of glVertexAttribPointer. I thought that you could supply your own index but the index is received from a function called getAttribLocation. You have to use this value as index for the glVertexAttribPointer.

## 14:15-16:30 Help peers with rendering

## 16:30-18:00 Fill out work log

## 19:00-19:15 Set up dependencies for own project

# Friday

## 13:15-0:00 Set up dependencies for project

<https://github.com/raspberrypi/firmware>

Originally, I wanted to set up the libraries myself but apparently, the libraries for the Pi are already on it in this directory. Because every Pi should have them, I won’t copy them in the project but rather make use of the ones on the Pi.

I can’t get the libraries to copy to the raspberry pi, so I will use the local files on it

# Sunday

## 15:55-17:45 Fill out learning log