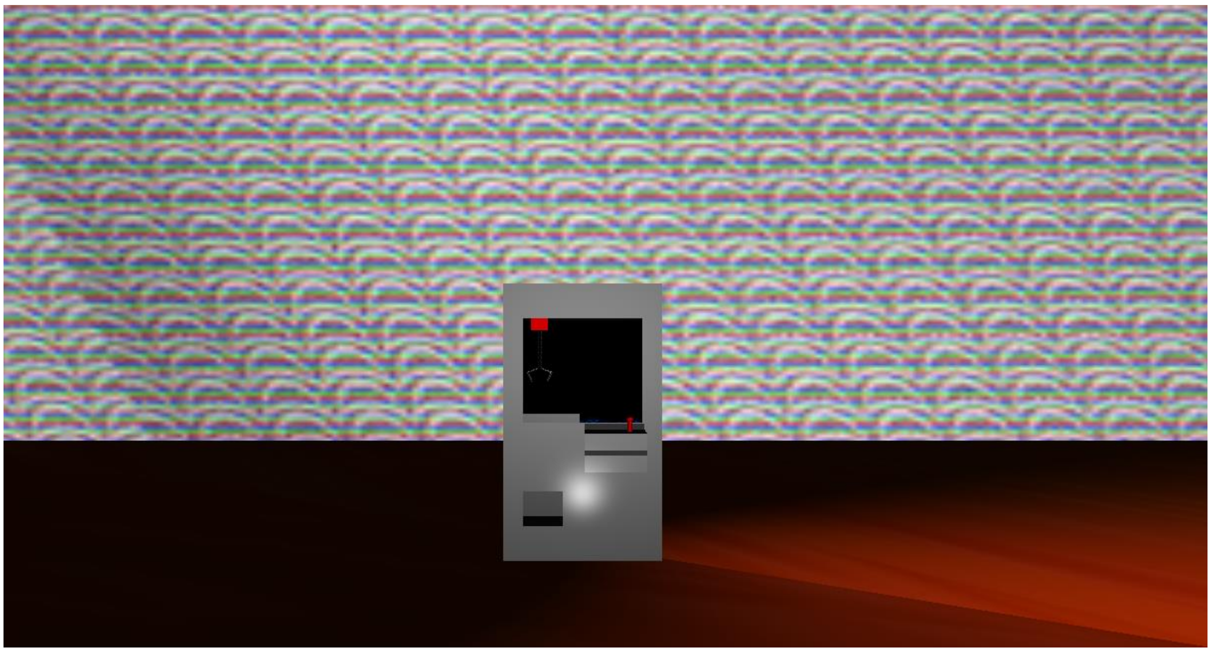
Graphics Final Report

Darragh Hurley – 16319250

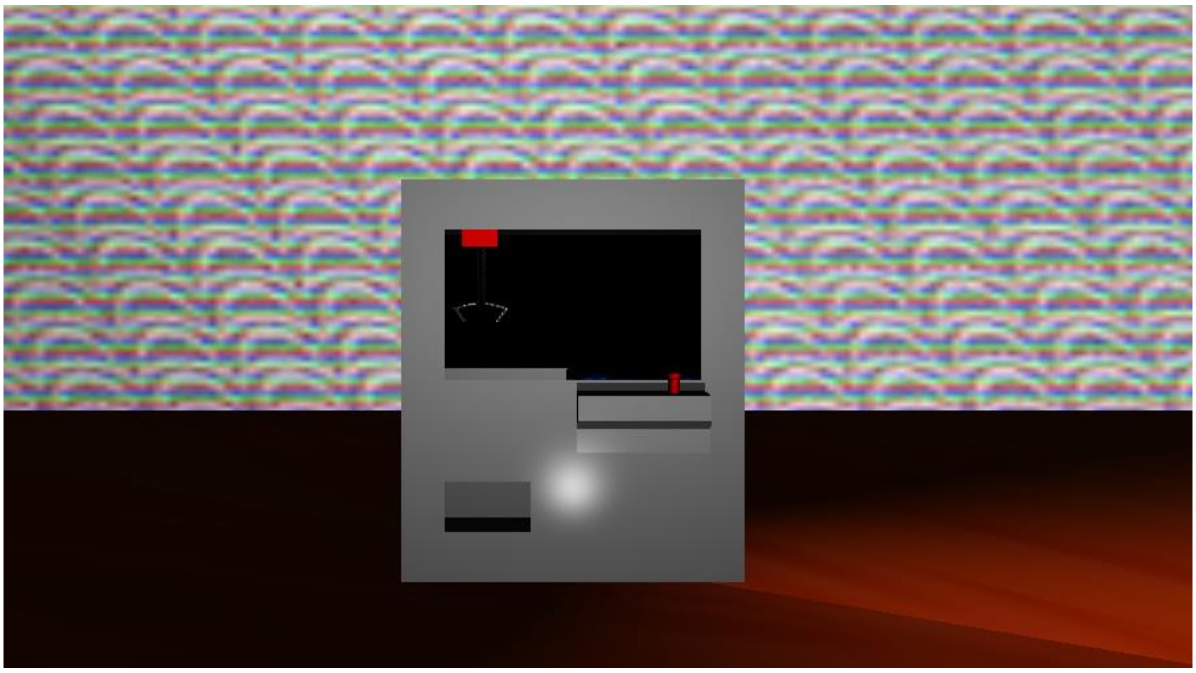
User interaction and camera-control:

My project was to create a claw machine within opengl that is fully functional.



In this project the user has complete control over all aspects of the scene. The models can be rotated, uniformly scaled and non-uniformly scaled by key-press. The rotation of the model is controlled by the arrow keys and the scaling of the object is controlled by specific letters (i.e. x scales up along the x axis).

Scaled along the x-axis:



The main interaction that the user has with the scene is the ability to move around the scene using the camera. The camera is also controlled via keypresses. The camera can move forward, backward, left, right, up and down within the bounds of the skybox. The user also has control over the angle at which the camera is pointed. The camera can look right, look left, look up and look down using keypresses. An orthographic projection of the machine can also be achieved via keypress. As a consistent scene environment I made a floor for the claw machine to be on top of and used a skybox implementation to build an arcade type room.

Look around keys for camera:

```
	if (key == 'w') {
		rotate_camUD = rotate_camUD - 3;
	}
	if (key == 's') {
		rotate_camUD = rotate_camUD + 3;
	}
	if (key == 'a') {
		rotate_camLR = rotate_camLR - 3;
	}
	if (key == 'd') {
		rotate_camLR = rotate_camLR + 3;
	}
```

```
Movement Keys (within the skybox):
```

```
	if (key == 'i') {
		if (cameraY == cubeSize - 1)
		{
			cameraY = cubeSize - 1;
		}
		else cameraY += 0.5f;
	}
	if (key == 'k') {
		if (cameraY == 1.f)
		{
			cameraY = 1.f;
		}
		else cameraY -= 0.5f;
	}

	if (key == 'j') {
		if (cameraX == -(cubeSize - 1))
		{
			cameraX = -(cubeSize - 1);
		}
		else cameraX -= 0.5f;
	}
	if (key == 'l') {
		if (cameraX == cubeSize - 1)
		{
			cameraX = cubeSize - 1;
		}
		else cameraX += 0.5f;
	}

	if (button == 3) //scroll to move forward/backward
	{
		if (cameraZ == cubeSize-1)
		{
			cameraZ = cubeSize-1;
		}
		else cameraZ += 0.5f;
	}
	if (button == 4)
	{

		if (cameraZ == -(cubeSize-1))
		{
			cameraZ = -(cubeSize-1);
		}
		else cameraZ -= 0.5f;
	}
```

Hierarchical animated object:

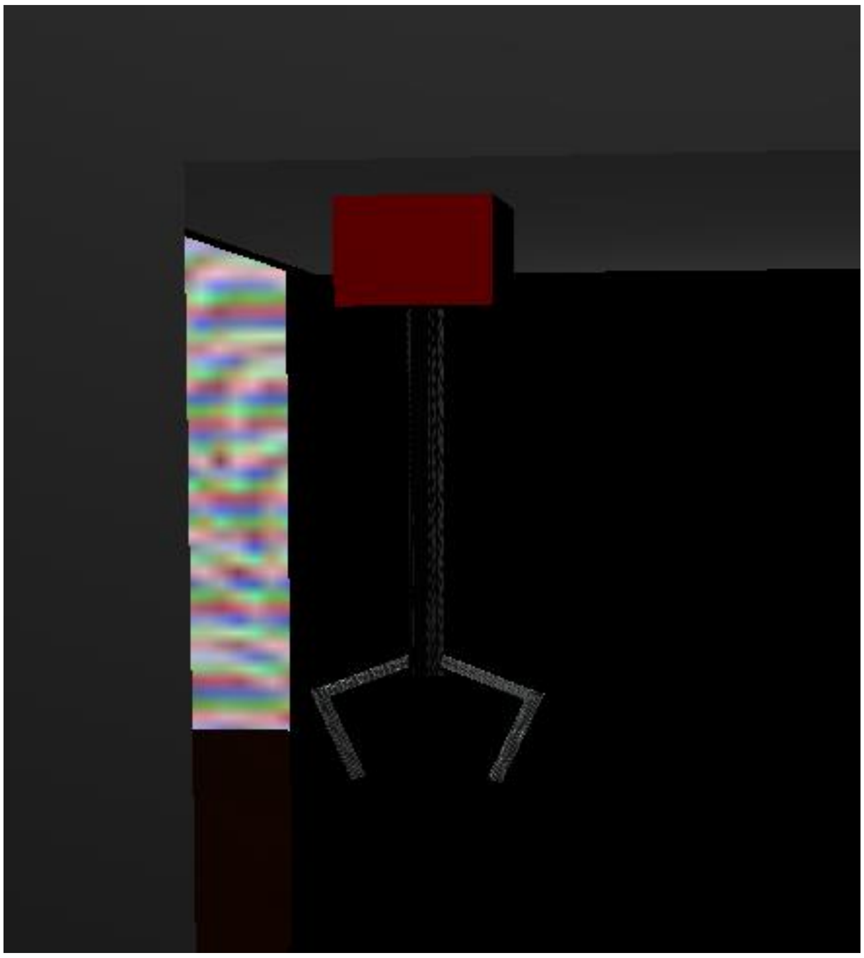
The main hierarchical animated object in this project is the claw within the machine. The claw as a whole is made from a base model, a rope model and 4 claw models. The base is a child model to the machine and can move along its x-axis and z-axis only within the bounds of the machine. The rope is a child model to the base as it must move everywhere that the base moves. The rope can extend by moving up and down the y-axis only. Two of the claws are child models to the rope as again they must move when the base/rope moves. The other two claws are child models to those claws. These claws are limited to rotating along the z-axis. To make an object become a child model you simply multiply it by the object which is its parent model as shown below:

```
mat4 rope = identity_mat4();

	rope = translate(rope, vec3(ropeX, ropeY, ropeZ));
	// Apply the root matrix to the child matrix
	rope = base * rope;
```

The joystick and monkey heads are also children models to the machine. The joystick points in the last direction that the claw was moved by rotating along the x-axis or z-axis. The monkey heads were child models to the machine so that they would also rotate and scale with all of the objects even thought in reality they would not be connected to the claw machine.

Texture mapping:

In this project I applied textures to three of models and 6 textures to the skybox that I implemented. The models that I applied textures to are the floor model, the rope model and the claw models. I textured the floor with a red copper texture, I textured the rope with a dark black metallic texture and I applied a lighter, shinier texture to the claws. In order to implement textures to my models I had to create a new vertex and fragment shader that could deal with reading in and outputting textures.



Fragment Shader:

```
gl_FragColor = texture2D(texture, texCoords)
```

Vertex shader:

```
texCoords = vec2(vertex_texture.x,vertex_texture.y);
```

To change between textures when displaying the scene the BIND_TEXTURE() function was used. The textures were first loaded in the initialiser function and were then bound before being drawn.

To load in the textures, I downloaded the SOIL library to read in the image files and display them as textures within the scene.

```
textures[0] = SOIL_load_OGL_texture(
        "Textures/metallic.png",
        SOIL_LOAD_RGBA,
        SOIL_CREATE_NEW_ID,
        SOIL_FLAG_INVERT_Y
);
textures[1] = SOIL_load_OGL_texture(
        "Textures/metal.jpg",
        SOIL_LOAD_RGBA,
        SOIL_CREATE_NEW_ID,
        SOIL_FLAG_INVERT_Y
);
textures[2] = SOIL_load_OGL_texture(
        "Textures/floor.jpg",
        SOIL_LOAD_RGBA,
        SOIL_CREATE_NEW_ID,
        SOIL_FLAG_INVERT_Y
);
```

Phong illumination model:

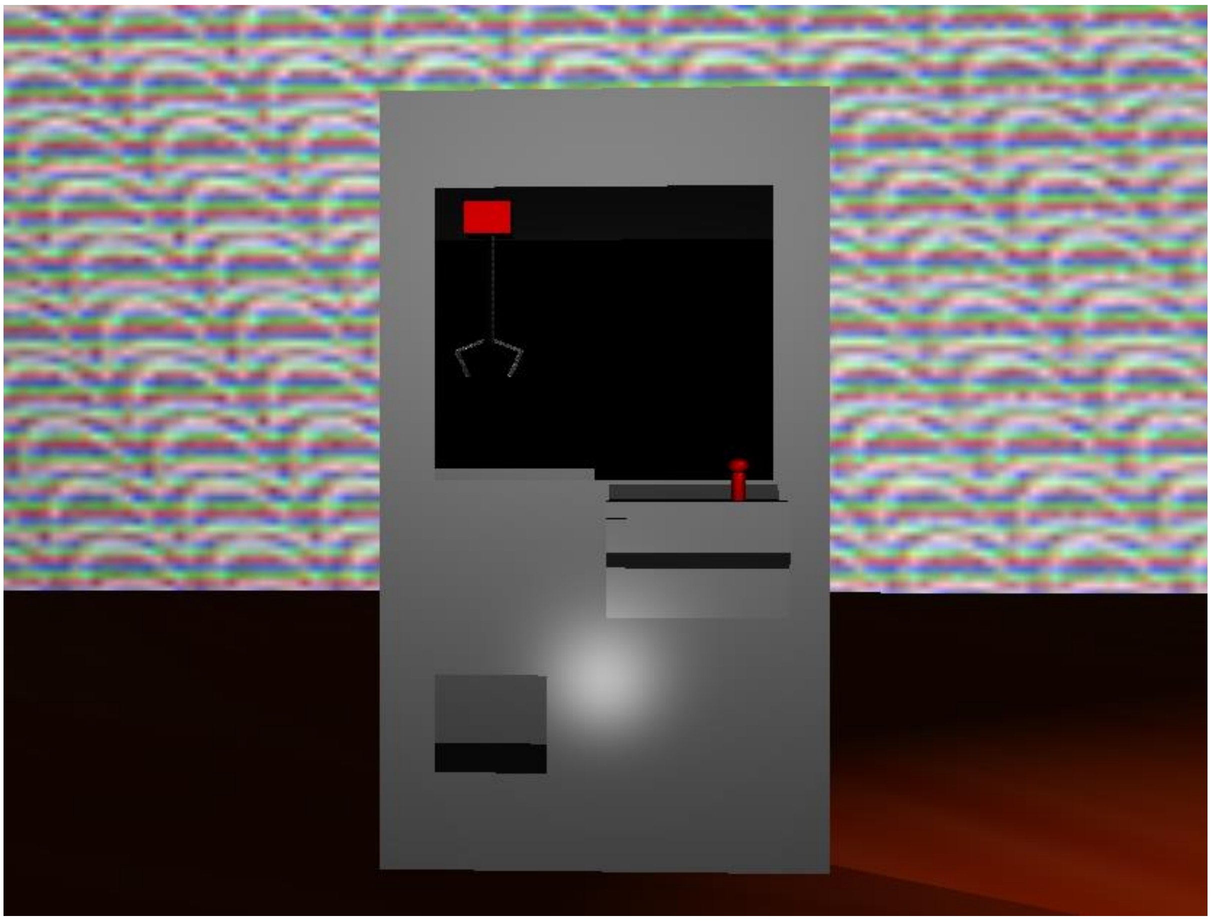
To implement a phong illumination model I had to create an ambient, diffuse and specular lighting arrangement. The main altercations done to have a functioning phong illumination model are in the fragment shaders. Ambient light is used to show the difference of lighting when a light sorce is not shining on the object. Diffuse lighting is to gradually brighten up the object as it is being introduced into the path of the light source. Specular lighting is used to show the shininess of an object by having the spotlight reflecting off of the object.

Fragment Shader code:

```
//ambient light

vec3 ambientLight = vec3(0.1f,0.1f,0.1f);


//diffuse

vec3 posToLightDirVec = normalize(vs_position - lightPosition);

vec3 diffuseColor = vec3(1.0f,1.0f,1.0f);

float diffuse = clamp(dot(posToLightDirVec, vs_normal),0,1);

vec3 diffuseFinal = diffuseColor*diffuse;


//specular
```

```
        vec3 lightToPosDirVec = normalize(lightPosition- vs_position);

        vec3 reflectDirVec = normalize(reflect(lightToPosDirVec, normalize(vs_normal)));

        vec3 posToViewDirVec = normalize(vs_position - cameraPosition);

        float specularConstant = pow(max(dot(posToViewDirVec, reflectDirVec),0),30);

        vec3 specularFinal = vec3(1.f,1.f,1.f) * specularConstant;


        gl_FragColor = vec4 (LightIntensity, 1.0) * (vec4(ambientLight,1.0f)+(vec4(diffuseFinal,1.f) +
(vec4(specularFinal,1.f))));
```

Specular lighting:



Extra features: (Skybox)

I created a skybox in order to build a surrounding room within the scene. The texture that I applied to it was meant to represent an arcade background. Although I understand that this is not necessarily what a skybox entails, I have commented code that can be used to show a city skyline at night that shows the correct functionality of the skybox.

Originally I attempted to load this in using the SOIL library but was having difficulty getting textures to display on the skybox, so I therefore included the stbi header file to help resolve that issue. In the initialise function you set your image parameters (i.e what you want to be displayed on the skybox) and run the loadCubemap function.

Extra Feature: (Making my own models)

Using blender I created all of the models for this project (excluding the monkey heads being used as the toys within the machine). Blender allows you to create a new mesh (e.g. a plane, a sphere) that can then be scaled and altered to represent a specific object. The models that I created include the machine, a joystick, a base for the claw, a rope for the claw, the claws and a floor. Some of these models have textures and others have their own independent shader (so that they have their own colour). In order to display each object a mesh must first be generated and then can be drawn using a shader function.

Youtube Link: https://youtu.be/r99Z_yw7aIQ