EE497 – 3D INTERFACE TECHNOLOGIES

ASSIGNMENT 2

CIARAN O'DONNELL

STUDENT NO: 15414048

DATE: 18TH/APRIL/2019

Introduction:

The goal of this assignment is to implement a sensor driven 3D controller to interact with 3D graphics. The controller will be android based and the 3D graphics that it will interact with will be drawn on the android device using androids OpenGL ES capabilities.

The assignment will be implemented and tested within the Android Studio IDE. This will allow for the java code to be written and tested using a single software through the Android Studios device emulators.

Part 0: Preparation

To begin this assignment the first step is to prepare a 3D scene to be manipulated by the controller. This scene is created with an Open GL environment. The steps to create this scene are described in the android developer training resources found at: https://developer.android.com/training/graphics/opengl/index.html.

The first four lessons must be completed to create the 3D scene that will be manipulated by the 3D controller.

Lesson 1 Build an OpenGL ES environment:

Lesson one describes the task of creating a Open GL ES environment on which to deploy the 3D scene. In this case the OpenGL ES environment is created via creating an android activity that controls the environment and creating a GLSurfaceView object and a renderer class to draw to the GLSurfaceView.

Lesson 2 Define shapes:

The next lesson was to simply define the shapes to be drawn by the renderer. In this case a simple square and triangle are defined. These shapes are defined by way of initializing the coordinates of each vertex, and then connecting the vertices in a counter clockwise direction for each triangle face.

Lesson 3 Draw shapes:

In order for the shapes to be drawn the Vertex Shader and Fragment Shader must be programmed and a program object created to store them. The vertex shader describes to the renderer how to draw the vertices of the object and the fragment shader describes how to render the surface of the shape.

Lesson 4 Apply projection and camera views

On completion of lesson three the app can display a triangle shape to the screen however the shape is warped and changes based on the screens dimensions. In order to solve this the shape should be transformed through a projection transformation to adjust the coordinates based on the GLSurfaceView, and again through a Camera View transformation to adjust the coordinates based on a virtual camera position, this lesson describes that process.

Within this lesson it is important that the Matrix.setLookAtM is set to:
(mViewMatrix, 0, 0, 0, -5, 0f, 0f, 0f, 0f, 1f, 0f);
To avoid clipping later in the assignment.

With this initial preparation complete the 3D controller may now be implemented to interact with this 3D scene.

Part 1 3D Rotation

Accelerometer Sensor

The first step in implementing the 3D controller is the integration of the accelerometer. This is done in android via a sensor manager object. This object controls sensors and sensor listener events. The accelerometer object must also be created. Then a sensor listener can be created in the sensor manager listening to the accelerometer sensor.

```
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

mSensorManager.registerListener(sensorListener, mAccelerometer,
SensorManager.SENSOR_DELAY_NORMAL);
```

With the sensor listener created the sensor event methods must be implemented. This includes an onSensorChanged() method as well as an onAccuracyChanged() method. In this case the accuracy method is not needed and left empty however the onSensorChanged method is used to update the accelerometer values at each sensor read.

```
public void onSensorChanged(SensorEvent event) {

    if(event.sensor.getType() == Sensor.TYPE_ACCELEROMETER){
        accel[0] = event.values[0];//x
        accel[1] = event.values[1];//y
        accel[2] = event.values[2];//z
    }

}

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {

}
```

Magnetic Field Sensor

Once the accelerometer sensor is implemented the next step is to also implement the magnetic field sensor to improve the accuracy of 3D orientation calculations later. The magnetic field sensor is integrated in much the same way as the accelerometer. The sensor object is created and a listener added to the same sensor manager as before. The listener methods can then be added to to update the magnetic field values. With a check done for which sensor is updating on each run of the method.

```
mMagnetic = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);

mSensorManager.registerListener(sensorListener, mMagnetic,
SensorManager.SENSOR_DELAY_NORMAL);
```

```
public void onSensorChanged(SensorEvent event) {

    if(event.sensor.getType() == Sensor.TYPE_ACCELEROMETER){
```

```
        accel[0] = event.values[0];//x
        accel[1] = event.values[1];//y
        accel[2] = event.values[2];//z
    }

    else if(event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD){
        mag[0] = event.values[0];
        mag[1] = event.values[1];
        mag[2] = event.values[2];
    }
}
```

Orientation Calculation

Once the data is received from the accelerometer and magnetic field sensor it is possible to calculate the devices orientation. This is done via a sensor manager function that calculates a rotation matrix from the accelerometer and magnetic field data and then populates a float array with the orientation data.

```
public static float[] updateOrientation(){
    SensorManager.getRotationMatrix(rotationMatrix, null, accel, mag);
    SensorManager.getOrientation(rotationMatrix, mOrientationAngles);

    return mOrientationAngles;
}
```

The mOrientationAngles object in this case is a float[3] array with the azimuth, pitch and roll value stored as mOrientationAngles[0], mOrientationAngles[1], and mOrientationAngles[2].

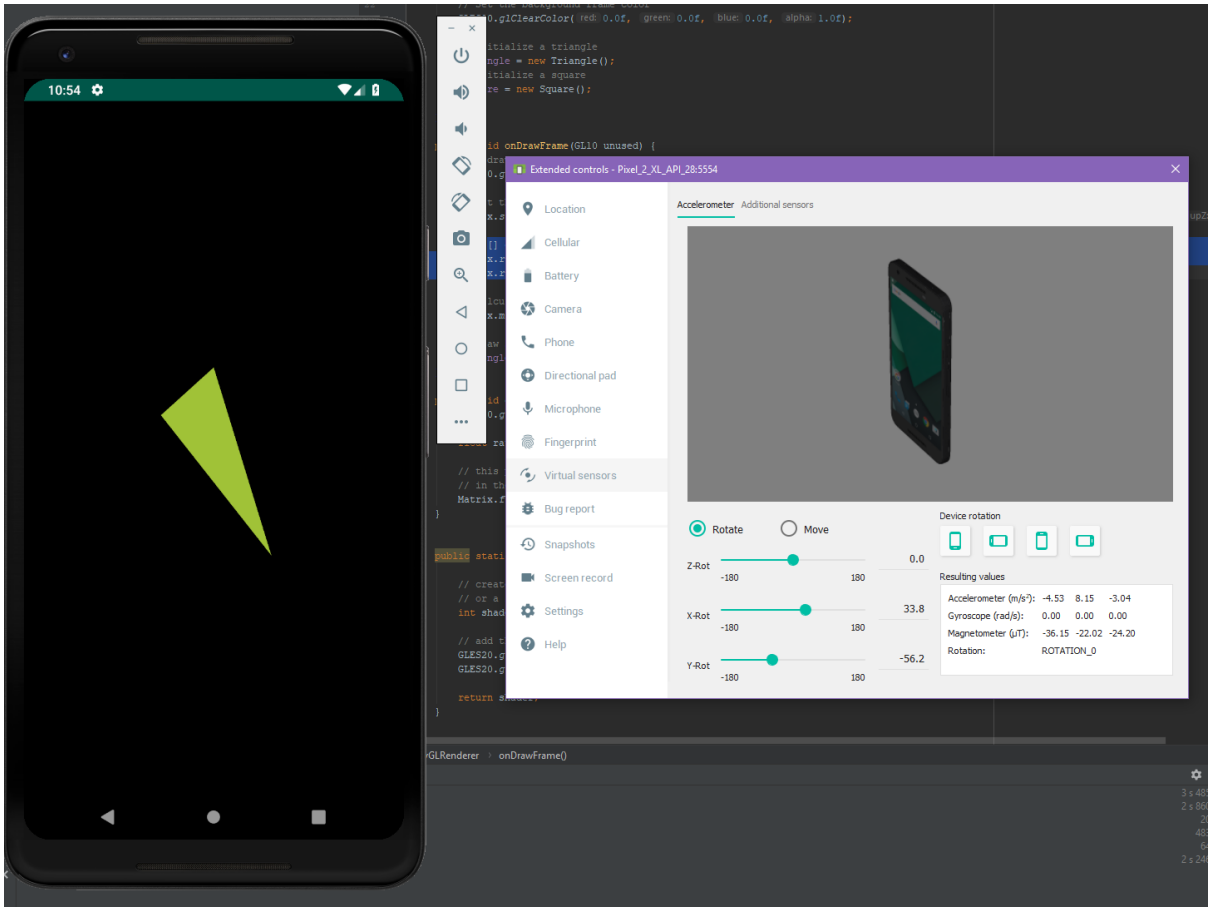
Control 3D Scene

With the orientation of the device calculated it is time to use this data to control the rotation of the 3D scene. This is done within the renderer object via rotating the view matrix of the 3D scene. First the orientation data is updated within the Renderer on draw method. Then with this data the view matrix is rotated about the X axis by the pitch of the device. Then the view is rotated about the y axis by the roll value of the device.

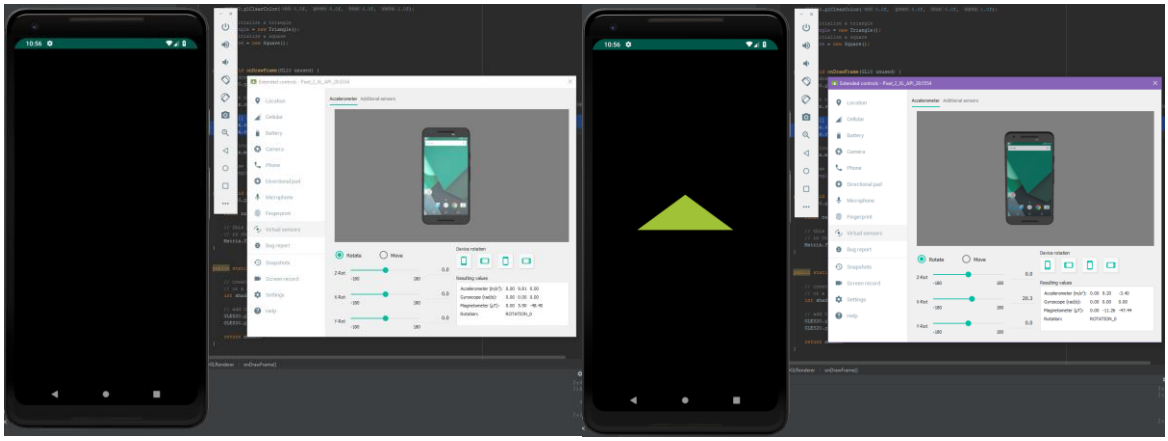
```
float[] orientation = OpenGLES20Activity.updateOrientation();
Matrix.rotateM(viewMatrix, 0, orientation[1]*57.2957795f, 1, 0, 0);//rotate x by
pitch
Matrix.rotateM(viewMatrix, 0, orientation[2]*57.2957795f, 0, 1, 0);//rotate y by
roll
```

The pitch and roll values must be converted from radians to degrees for the rotate method.

Once this step is complete it is possible to run the app and via rotating the device in the emulator the shape can be seen to rotate in response.



One interesting bit of behaviour with this app in its current state is how when the device is in its default position the shape can not be seen due to the fact that it is rotated to be perpendicular to the screen.



This behaviour could be changed via initially rotating the view of the scene 90 degrees about X, or by taking it into account when defining the shape.

Part 2: 3D Translation

The next step is to return to the code completed at part 0, and again build atop of this to control the 3D scene based on the devices 3D position in space. Again, the first step of this process is to implement the accelerometer to retrieve acceleration data.

Accelerometer Sensor

The accelerometer sensor is implemented identically to before using a sensor manager to house the listener and an onSensorChanged method for each update of the sensor.

```
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

mSensorManager.registerListener(sensorListener, mAccelerometer,
SensorManager.SENSOR_DELAY_NORMAL);
```

```
public void onSensorChanged(SensorEvent event) {

    if(event.sensor.getType() == Sensor.TYPE_ACCELEROMETER){
        accel[0] = event.values[0];//x
        accel[1] = event.values[1];//y
        accel[2] = event.values[2];//z
    }

}

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {

}
```

Gravity Filter

Due to the fact that the interest is now with the device's movement and not its orientation gravities effect on the device is no longer of interest. Therefore, it must be filtered out. One way to do this is to use a weighted moving average as a low pass filter to calculate what element is due to gravity and remove it.

```
final float alpha = 0.8f;

gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0];
gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1];
gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2];
accel[0] = event.values[0] - gravity[0];
accel[1] = event.values[1] - gravity[1];
accel[2] = event.values[2] - gravity[2];
```

This code will remove the gravitational element of the data however it is inaccurate at initialization due to the moving average being calculated with no prior gravity data. Therefore, the first 100 readings should be ignored.

```
if(i>100){
    accurate = true;
//Calculation will go here
}
else i++;
```

Calculate Distance

Once the gravitational element on the data is removed the acceleration data left over can be used to calculate the distance the device has moved by way 'dead reckoning'. The velocity is calculated via the previous velocity plus the acceleration*change in time, then the distance is calculated via this new velocity*change in time.

```
long time = new Date().getTime();
float dt = time - prevtime;
prevtime = time;


if(i>100){
    accurate = true;
    xV = (xV + accel[0]*dt);//x velocity
    xD = xV*dt;//x distance

    yV = (yV + accel[1]*dt);//y velocity
    yD = yV*dt;//y distance

    zV = (zV + accel[2]*dt);//z velocity
    zD = zV*dt;//z distance

}
else i++;
```

This code shows how the distances could be calculated via dead reckoning. However, in this case the change in time (dt) was in error and therefore the distance moved was instead calculated using current acceleration only, which while incorrect allows for the assignment to continue.

```
//long time = new Date().getTime();
//float dt = time - prevtime;
//prevtime = time;


if(i>100){
    accurate = true;
    xV = (xV + accel[0]);
    xD = xV;

    yV = (yV + accel[1]);
    yD = yV;

    zV = (zV + accel[2]);
    zD = zV;

}
else i++;
```

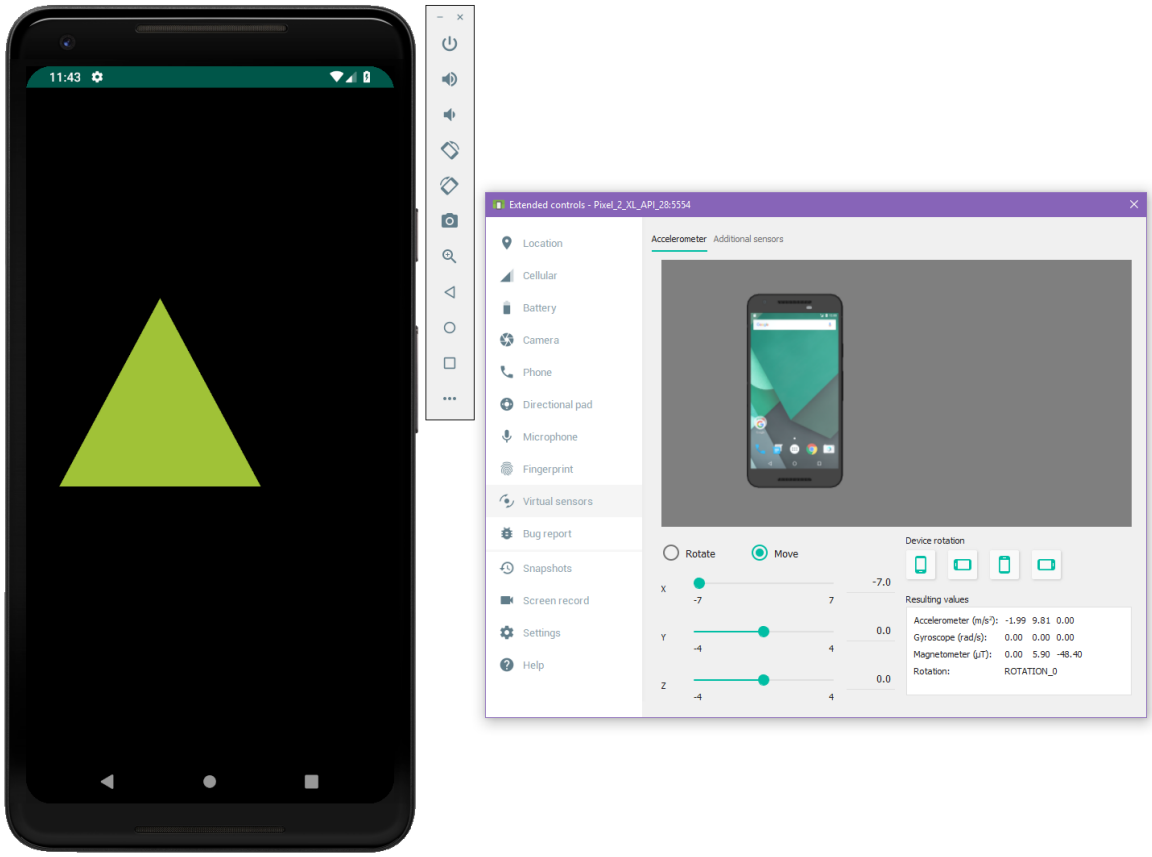
Translation of 3D Scene

With a distance variable calculated from the accelerometer data (in this case a somewhat inaccurate one) it is now possible to use this data to translate the 3D scene. The scene is again translated via a transformation of the view matrix. The matrix is translated via the x axis distance y axis distance and z axis distance. The distances in this case were made public variables but a better solution would be to have a return function for them.

```
if(OpenGLES20Activity.accurate){//only happens once gravity filtered
    Matrix.translateM(viewMatrix, 0, -OpenGLES20Activity.xD/25,
OpenGLES20Activity.yD/25, -OpenGLES20Activity.zD/25);
}//devided by 25 to slow movement so is visable
```

The distance it is translated by is divided by 25 as to slow the distance moved in order to stop the shape simply moving out of the view port.

With this complete it is possible to view the app in operation,



An interesting behaviour with this however is that after each movement the shape returns to its original position. This is due to how the accelerometer senses deceleration. As acceleration is a measurement in change in velocity over time, deceleration is actually just acceleration in the opposite direction back to a velocity of 0. Therefore, the accelerometer senses this as acceleration in the opposite direction and moves the shape in that direction back to zero. This is due to the fact that the app has no data for how long the device was at each velocity due to the problems with calculating a time variable earlier.