**EE497 – 3D Interface Technologies Assignment 2**

**Part 1: 3D Rotation**

**Design**

The OpenGLES20Activity class is the first class and it is used to create the app and used to initiate the android app, in this class the sensor manager and sensors that are used was declared.

The class also contains the pause and resume method that are used to control what happens to the app when it is paused.

The OpenGLES20Activity calls the MyGLSurfaceView which is used to render the background and objects displayed in the application.

MyGLSurfaceView then calls the MyGLRenderer this is the class that controls most of the application, it contains the onSurfaceChanged, loadShader, onSurfaceCreated, onDrawFrame and onSensorChanged methods these methods are the main methods that control the display and operation of the app.

The three main functions are onSurfaceCreated, onDrawFrame and onSensorChanged.

onSurfaceCreated Is a method that initates what happens when a new surface is displayed, in my application the surface used is a primitive triangle therefore this method calls the method Triangle which returns a geometric shape which is a triangle.

onSensorChanged, for this application this was the most important method as it used if statements two if statements to find the acceleration and magnetic field of the android phone.

|  |
| --- |
| if(sensor.getType() == Sensor.*TYPE\_ACCELEROMETER*){  accelerometer = sensorEvent.values;  ……  if(sensor.getType() == Sensor.*TYPE\_MAGNETIC\_FIELD*) {  magnetometer = sensorEvent.values;  } if (accelerometer != null && magnetometer != null) {  boolean success = SensorManager.*getRotationMatrix*(rotationMatrix, null, linear\_acceleration, magnetometer);  if(success){  SensorManager.*getOrientation*(rotationMatrix, orientationAngles);  this.azimuth = Math.*toDegrees*(orientationAngles[0]);  this.pitch = Math.*toDegrees*(orientationAngles[1]);  this.roll = Math.*toDegrees*(orientationAngles[2]);  }} |

onDrawFrame is the method that draws what is seen on the user’s screen.

Using the values calculated for pitch and roll the triangle drawn on the screen will vary for the user.

Code example can be seen below

|  |
| --- |
| GLES20.*glClear*(GLES20.*GL\_COLOR\_BUFFER\_BIT*);  // Set the camera position (View matrix) Matrix.*setLookAtM*(mViewMatrix,0,0,0,-5,0f,0f,0f,0f,1.0f,0.0f);  // Create a rotation and translation for the cube Matrix.*setIdentityM*(mRotationMatrix, 0);  // Assign mRotationMatrix a rotation with the seekbar Matrix.*rotateM*(mRotationMatrix,0, (float) this.pitch, 1.0f, 0.0f, 0.0f);  // Combine the model with the view matrix Matrix.*multiplyMM*(mMVPMatrix,0, mViewMatrix,0,mRotationMatrix,0);  Matrix.*rotateM*(mRotationMatrix,0, (float) this.roll,0.0f,1.0f,0.0f);  Matrix.*multiplyMM*(mMVPMatrix,0, mMVPMatrix,0,mRotationMatrix,0); |

**Describe any strange behaviour exhibited by the completed application and explain how this could be eliminated.**

When pitch of +/-90 it is possible drive roll, using rotation matrices as seen above has compensated for the issue.

**Part 2: 3D Translation**

It was possible to remove the gravitational influence on the acceleration by isolating the high and low frequencies in the acceleration and removing them from the original acceleration recorded.

|  |
| --- |
| accelerometer = sensorEvent.values; // Isolate the force of gravity with the low-pass filter. gravity[0] = alpha \* gravity[0] + (1 - alpha) \* sensorEvent.values[0]; gravity[1] = alpha \* gravity[1] + (1 - alpha) \* sensorEvent.values[1]; gravity[2] = alpha \* gravity[2] + (1 - alpha) \* sensorEvent.values[2]; // Remove the gravity contribution with the high-pass filter. linear\_acceleration[0] = sensorEvent.values[0] - gravity[0]; linear\_acceleration[1] = sensorEvent.values[1] - gravity[1]; linear\_acceleration[2] = sensorEvent.values[2] - gravity[2]; |

• Use dead reckoning to convert the acceleration samples into velocity samples and use these velocity samples to calculate an approximate value for the device’s position (along the x, y and z axes)

• Finally, use calculated position values to translate the shape in the x, y and z directions. Note that the first few values (~100) will contain errors, so these should be ignored.

|  |
| --- |
| float timechange = System.*currentTimeMillis*()-lastTime;  lastTime = System.*currentTimeMillis*();  if (count>150){  Newvelocity[0] = oldVelocity[0] + (linear\_acceleration[0]\*timechange);  Newvelocity[1] = oldVelocity[1] + (linear\_acceleration[1]\*timechange);  Newvelocity[2] = oldVelocity[2] + (linear\_acceleration[2]\*timechange);  distance[0] = Newvelocity[0] \* timechange;  distance[1] = Newvelocity[1] \* timechange;  distance[2] = Newvelocity[2] \* timechange;  x = distance[0]/100000f;  y = -distance[1]/100000f;  z = distance[2]/100000f;;}  count++; } |

The above code uses dead reckoning to convert the acceleration samples into velocity samples an if statement is used to ignore the first 150 vales as it said the first 100 values may contain some errors.

OpenGLES20Acitivity.java

|  |
| --- |
| package com.example.mradw.androidassignment2;  import android.app.Activity; import android.hardware.Sensor; import android.hardware.SensorManager; import android.os.Bundle;  public class OpenGLES20Activity extends Activity {   private MyGLSurfaceView mGLView;  private Sensor accelerometerSensor;  private Sensor magneticSensor;  private SensorManager SM;   @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);   // Create a GLSurfaceView instance and set it as the ContentView for this Activity.  mGLView = new MyGLSurfaceView(this);  setContentView(mGLView);   // Create our Sensor Manager  //manages used sensors  SM = (SensorManager) getSystemService(*SENSOR\_SERVICE*);  // Accelerometer  accelerometerSensor = SM.getDefaultSensor(Sensor.*TYPE\_ACCELEROMETER*);  // Magnetic  magneticSensor = SM.getDefaultSensor(Sensor.*TYPE\_MAGNETIC\_FIELD*);  }   @Override  protected void onPause() {  // The following call pauses the rendering thread.  super.onPause();  mGLView.onPause();  SM.unregisterListener(mGLView.getmRenderer().getSensorEventListener());  }   @Override  protected void onResume() {  // resumes a paused rendering thread.  super.onResume();  mGLView.onResume();  // Register sensor listener  SM.registerListener(mGLView.getmRenderer().getSensorEventListener(), accelerometerSensor, SensorManager.*SENSOR\_DELAY\_NORMAL*);  SM.registerListener(mGLView.getmRenderer().getSensorEventListener(), magneticSensor, SensorManager.*SENSOR\_DELAY\_NORMAL*);  }  } |

MyGLSurfaceView.java

|  |
| --- |
| package com.example.mradw.androidassignment2;  import android.content.Context; import android.opengl.GLSurfaceView;  public class MyGLSurfaceView extends GLSurfaceView {  private final MyGLRenderer mRenderer;   public MyGLSurfaceView(Context context){  super(context);   // Create an OpenGL ES 2.0 context  setEGLContextClientVersion(2);   mRenderer = new MyGLRenderer();   // Set the Renderer for drawing on the GLSurfaceView  setRenderer(mRenderer);  }   public MyGLRenderer getmRenderer() {  return mRenderer;  } } |

MyGLRenderer

|  |
| --- |
| package com.example.mradw.androidassignment2;  import android.hardware.Sensor; import android.hardware.SensorEvent; import android.hardware.SensorEventListener; import android.hardware.SensorManager; import android.opengl.GLES20; import android.opengl.GLSurfaceView; import android.opengl.Matrix;  import javax.microedition.khronos.opengles.GL10;  public class MyGLRenderer implements GLSurfaceView.Renderer, SensorEventListener {   private Triangle mTriangle;   // mMVPMatrix is an abbreviation for "Model View Projection Matrix"  private float distance[] = new float[3];  private final float[] mMVPMatrix = new float[16];  private final float[] mProjectionMatrix = new float[16];  private final float[] mViewMatrix = new float[16];  private float[] mRotationMatrix = new float[16];  private float[] accelerometer;  private float[] magnetometer;  private int count = 0;  private long lastTime;  private float x = 0.0f, y = 0.0f, z = 0.0f;  private double azimuth = 0.0, pitch = 0.0, roll = 0.0;    @Override  public void onSurfaceChanged(GL10 unused, int width, int height) {  GLES20.*glViewport*(0, 0, width, height);  float ratio = (float) width / height;  // this projection matrix is applied to object coordinates  // in the onDrawFrame() method  Matrix.*frustumM*(mProjectionMatrix, 0, -ratio, ratio, -1, 1, 3, 7);  }   public static int loadShader(int type, String shaderCode){   // create a vertex shader type (GLES20.GL\_VERTEX\_SHADER)  // or a fragment shader type (GLES20.GL\_FRAGMENT\_SHADER)  int shader = GLES20.*glCreateShader*(type);   // add the source code to the shader and compile it  GLES20.*glShaderSource*(shader, shaderCode);  GLES20.*glCompileShader*(shader);   return shader;  }   public void onSurfaceCreated(GL10 unused, javax.microedition.khronos.egl.EGLConfig config) {  // Set the background frame color  GLES20.*glClearColor*(0.0f, 0.0f, 0.0f, 1.0f);   // initialize a square  mTriangle = new Triangle();  }   public void onDrawFrame(GL10 unused) {  GLES20.*glClear*(GLES20.*GL\_COLOR\_BUFFER\_BIT*);   // Set the camera position (View matrix)  Matrix.*setLookAtM*(mViewMatrix,0,0,0,-5,0f,0f,0f,0f,1.0f,0.0f);   // Create a rotation and translation for the cube  Matrix.*setIdentityM*(mRotationMatrix, 0);   // Assign mRotationMatrix a rotation with the seekbar  Matrix.*rotateM*(mRotationMatrix,0, (float) this.pitch, 1.0f, 0.0f, 0.0f);   // Combine the model with the view matrix  Matrix.*multiplyMM*(mMVPMatrix,0, mViewMatrix,0,mRotationMatrix,0);   Matrix.*rotateM*(mRotationMatrix,0, (float) this.roll,0.0f,1.0f,0.0f);   Matrix.*multiplyMM*(mMVPMatrix,0, mMVPMatrix,0,mRotationMatrix,0);   // Calculate the projection and view transformation  Matrix.*multiplyMM*(mMVPMatrix,0, mProjectionMatrix, 0,mMVPMatrix,0);   // Translate shape  Matrix.*translateM*(mMVPMatrix,0,x,y,z);   // draw shape  mTriangle.draw(mMVPMatrix);}    @Override  public void onSensorChanged(SensorEvent sensorEvent){  float[] Newvelocity = new float[3];  float[] oldVelocity = new float[3];  final float alpha = 0.8f;  final float[] gravity = new float[3];  final float[] linear\_acceleration = new float[3];  final float[] rotationMatrix = new float[9];  final float[] orientationAngles = new float[3];   Sensor sensor = sensorEvent.sensor;  if(sensor.getType() == Sensor.*TYPE\_ACCELEROMETER*){  accelerometer = sensorEvent.values;  // Isolate the force of gravity with the low-pass filter.  gravity[0] = alpha \* gravity[0] + (1 - alpha) \* sensorEvent.values[0];  gravity[1] = alpha \* gravity[1] + (1 - alpha) \* sensorEvent.values[1];  gravity[2] = alpha \* gravity[2] + (1 - alpha) \* sensorEvent.values[2];  // Remove the gravity contribution with the high-pass filter.  linear\_acceleration[0] = sensorEvent.values[0] - gravity[0];  linear\_acceleration[1] = sensorEvent.values[1] - gravity[1];  linear\_acceleration[2] = sensorEvent.values[2] - gravity[2];   float timechange = System.*currentTimeMillis*()-lastTime;  lastTime = System.*currentTimeMillis*();  if (count>100){  Newvelocity[0] = oldVelocity[0] + (linear\_acceleration[0]\*timechange);  Newvelocity[1] = oldVelocity[1] + (linear\_acceleration[1]\*timechange);  Newvelocity[2] = oldVelocity[2] + (linear\_acceleration[2]\*timechange);  distance[0] = Newvelocity[0] \* timechange;  distance[1] = Newvelocity[1] \* timechange;  distance[2] = Newvelocity[2] \* timechange;  x = distance[0]/100000f;  y = -distance[1]/100000f;  z = distance[2]/100000f;   oldVelocity = Newvelocity;}  count++;  }   if(sensor.getType() == Sensor.*TYPE\_MAGNETIC\_FIELD*) {  magnetometer = sensorEvent.values;  }  if (accelerometer != null && magnetometer != null) {  boolean success = SensorManager.*getRotationMatrix*(rotationMatrix, null, linear\_acceleration, magnetometer);  if(success){  SensorManager.*getOrientation*(rotationMatrix, orientationAngles);  this.azimuth = Math.*toDegrees*(orientationAngles[0]);  this.pitch = Math.*toDegrees*(orientationAngles[1]);  this.roll = Math.*toDegrees*(orientationAngles[2]);  }  }  }   @Override  public void onAccuracyChanged(Sensor sensor, int i) { }  public SensorEventListener getSensorEventListener(){ return this; } } |

Trinagle.java

|  |
| --- |
| package com.example.mradw.androidassignment2;  import android.opengl.GLES20;  import java.nio.ByteBuffer; import java.nio.ByteOrder; import java.nio.FloatBuffer;  public class Triangle {  private final String vertexShaderCode =  // This matrix member variable provides a hook to manipulate  // the coordinates of the objects that use this vertex shader  "uniform mat4 uMVPMatrix;" +  "attribute vec4 vPosition;" +  "void main() {" +  // the matrix must be included as a modifier of gl\_Position  // Note that the uMVPMatrix factor \*must be first\* in order  // for the matrix multiplication product to be correct.  " gl\_Position = uMVPMatrix \* vPosition;" +  "}";   private final String fragmentShaderCode =  "precision mediump float;" +  "uniform vec4 vColor;" +  "void main() {" +  " gl\_FragColor = vColor;" +  "}";   // Use to access and set the view transformation  private int mMVPMatrixHandle;  private FloatBuffer vertexBuffer;   // number of coordinates per vertex in this array  static final int *COORDS\_PER\_VERTEX* = 3;  static float *triangleCoords*[]={ // in counterclockwise order;  0.0f, 0.622008459f, 0.0f, // top  -0.5f, -0.311004243f, 0.0f, // bottom left  0.5f, -0.311004243f, 0.0f // bottom right  };   // Set color with red, green, blue and alpha (opacity) values  float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };   private final int mProgram;   public Triangle() {  // initialize vertex byte buffer for shape coordinates  ByteBuffer bb = ByteBuffer.*allocateDirect*(  // (number of coordinate values \* 4 bytes per float)  *triangleCoords*.length \* 4  );  bb.order(ByteOrder.*nativeOrder*());   // create a floating point buffer from the ByteBuffer  vertexBuffer = bb.asFloatBuffer();  // add the coordinates to the FloatBuffer  vertexBuffer.put(*triangleCoords*);  // set the buffer to read the first coordinate  vertexBuffer.position(0);   int vertexShader = MyGLRenderer.*loadShader*(GLES20.*GL\_VERTEX\_SHADER*, vertexShaderCode);  int fragmentShader = MyGLRenderer.*loadShader*(GLES20.*GL\_FRAGMENT\_SHADER*, fragmentShaderCode);   // create empty OpenGL ES Program  mProgram = GLES20.*glCreateProgram*();   // add the vertex shader to program  GLES20.*glAttachShader*(mProgram, vertexShader);   // add the fragment shader to program  GLES20.*glAttachShader*(mProgram, fragmentShader);   // create OpenGL ES Program executables  GLES20.*glLinkProgram*(mProgram);  }   private int mPositionHandle;  private int mColorHandle;   private final int vertexCount = *triangleCoords*.length/*COORDS\_PER\_VERTEX*;  private final int vertexStride = *COORDS\_PER\_VERTEX* \* 4; // 4 bytes per vertex   public void draw(float[] mvpMatrix) { // pass in the calculated transformation matrix  // Add program to OpenGL ES environment  GLES20.*glUseProgram*(mProgram);   // get handle to vertex shader's vPosition member  mPositionHandle = GLES20.*glGetAttribLocation*(mProgram, "vPosition");   // Enable a handle to the triangle vertices  GLES20.*glEnableVertexAttribArray*(mPositionHandle);   // Prepare the triangle coordinate data  GLES20.*glVertexAttribPointer*(mPositionHandle, *COORDS\_PER\_VERTEX*, GLES20.*GL\_FLOAT*, false, vertexStride, vertexBuffer);   // get handle to fragment shader's vColor member  mColorHandle = GLES20.*glGetUniformLocation*(mProgram, "vColor");   // Set color for drawing the triangle  GLES20.*glUniform4fv*(mColorHandle, 1, color, 0);   // get handle to shape's transformation matrix  mMVPMatrixHandle = GLES20.*glGetUniformLocation*(mProgram, "uMVPMatrix");   // Pass the projection and view transformation to the shader  GLES20.*glUniformMatrix4fv*(mMVPMatrixHandle, 1, false, mvpMatrix, 0);   // Draw the triangle  GLES20.*glDrawArrays*(GLES20.*GL\_TRIANGLES*, 0,vertexCount);   // Disable vertex array  GLES20.*glDisableVertexAttribArray*(mPositionHandle);  }  } |