Round Viewer

1. Description

The application offers the possibility to view real scenes which were recorded using a cube map method (360 degrees view in all directions) to be viewed on the phone in an interactive manner. The application provides the possibility to view the scene by rotating the phone: i.e. the user rotates the phone to the right, the scene open in the application also rotates to the right, the user rotates the phone upwards, the scene also rotates upwards etc. This functionality is achievable with the help of a gyroscope. If the mobile device does not posses such a sensor, the user will be able to rotate the scene using finger swipes: i.e. manually rotate the scene in a direction by swiping in that direction.

There is a selection of several scenes provided for the user.

2. Implementation

The application consists of two activites. In the first activity, the user is presented with a list of scenes from which he/she can choose to visualize. After choosing a scene, a new activity is opened which contains a view with the interactive image to visualize.

2.1. OpenGL

The implementation is done with the help of OpenGL ES 2.0. OpenGL for Embedded Systems (OpenGL ES or GLES) is a subset of the OpenGL computer graphics rendering API for rendering 2D or 3D graphics such as those used by video games, typically hardware accelerated using a GPU. It is designed for embedded systems like smartphones, tablets, consoles and PDAs.

2.1.1. Cube map

For this application, the OpenGL scene is pretty simple: we have a cube, which has a texture mapped on all of its sizes (the cube mapped photo/scene), which rotates on all axes with a matrix given from the outside (i.e. the rotation of the phone/device). The camera/viewer is positioned in the center of this cube, in its interior, and cannot move. There is no lighting enabled in OpenGL, thus the user will not know that the camera is actually placed in the interior of a cube, because he/she will not be able to distinguish edges or shadows. The term is known as skybox in computer graphics. Rotating the cube at the same time with the rotation of the device will create the desired effect, of the user being able to look at the scene in all directions, just like he would turn his head around.

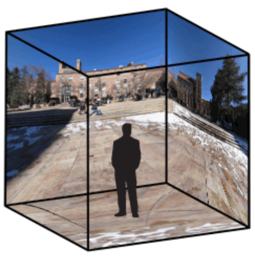

Fig. 1 – depiction of the method

For each face of the cube, a different portion of an image similar to the one in Fig. 2 is applied as texture. As one can easily notice, if we fold the image in Fig. 2 we can easily form a cube. The images are taken in such a way that they are joint perfectly at edges, being seemingly continuous.

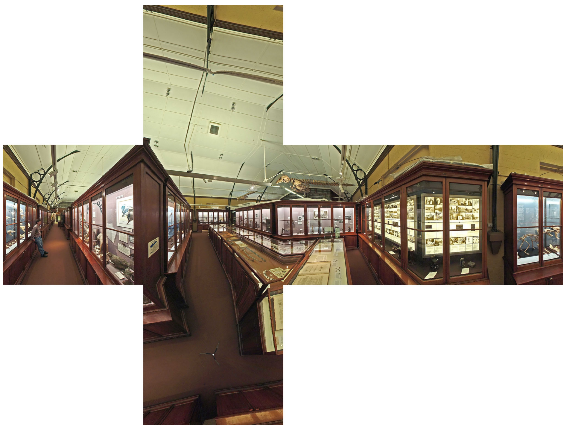

Fig. 2 – image used as texture

2.1.2. Renderer

The Renderer class of the application (see Fig. 3, class diagram) has the role of defining the cube and the UV coordinates for the cube (the mapping between points of the cube in world space coordinates to points from the texture space coordinates), defining its rotation and placing it into the scene (i.e. rendering the scene).

In OpenGL objects are defined by using triangles which assemble more complex objects. Normally, for complex objects the coordinates of the triangles are specified in a separate object file (e.g. .obj) which can be created using specialized 3D modelling software, and then loaded into an OpenGL application. Since for this scene the only object is a cube, there was no need for such thing. The cube has 6 faces (12 triangles), which were defined manually, alongside with the UV mapping.

OpenGL runs native, therefore when passing those numbers the array of vertices needs to be converted into a ByteBuffer. This is because the internal representation of an array in the application might be different from the native representation in memory. The image is also loaded from JPEG resource images.

All this content (coordinates, texture coordinates, texture) is fed to the shaders. An OpenGL shader is a program which runs in the graphics pipeline to achieve a certain task in the pipe between scene space and image on screen. There are 2 shaders defined for this application: a vertex shader and a fragment shader. The first one applies the Model-View-Projection matrix multiplication on each vertex of the object, while the second one applies the loaded texture according to the UV coordinates specified. They are coded using GLSL (GL Shading Language).

2.2. Rotation

The rotation is achieved via input from a *rotation vector* sensor. It is not physically a sensor, but it gives the rotation of the device on all axes, as a rotation matrix. To achieve this, it uses input from the gyroscope and magnetometer. The magnetometer is needed because the Y axis is tangential to the ground at the device's current location and points toward the geomagnetic North Pole, while X points approximately East.

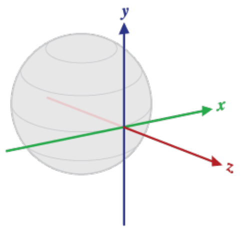

Fig. 3 – Coordinate system used by the rotation vector sensor

The matrix of rotation given by the rotation vector sensor can be used as is into the Renderer to rotate the cube accordingly.

If the device does not posses such sensors, then the SensorManager will notify this by returning false on registerListener(). If this is the case, the touchscreen control is enabled, in order to allow the interactivity with the application. The touch control is done via swipe gestures in all directions. There is a custom GLSurfaceView in the application, called TouchGLSurfaceView, which listens for touch events and creates a rotation matrix according to the movement of the finger on the screen. For this, the difference between the position when the finger touched the screen and the position when the finger was raised from the screen is calculated. Those values are used as tetha (angle) values in the Rx(tetha) and Ry(tetha) from the followin formulae:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 4 – rotation matrices about x, y and z axis

A rotation matrix is generated, similar to the one generated by the rotation vector sensor, and then passed further to the Renderer to rotate the cube. The rotation matrix is obtained by doing Rx*Ry, with angle values taken from the swipe gesture movement, as explained.

2.3. Components

The application design can be seen in the class diagram in fig. 5. Some of the classes were already discussed. The WelcomeActivity presents a list of scenes from which the user can choose. MainActivity contains only one TouchGLSurfaceView, which presents a scene rendered by the Renderer. The Renderer uses the TextureLoader and RawResourceReader to load the texture image and the shader codes respectively. MainActivity also uses SensorManager to register a rotation vector sensor, if available on the device. If such sensor is not present, touch screen control is enabled in TouchGLSurfaceView and the view starts registering user swipes on screen to generate rotation.

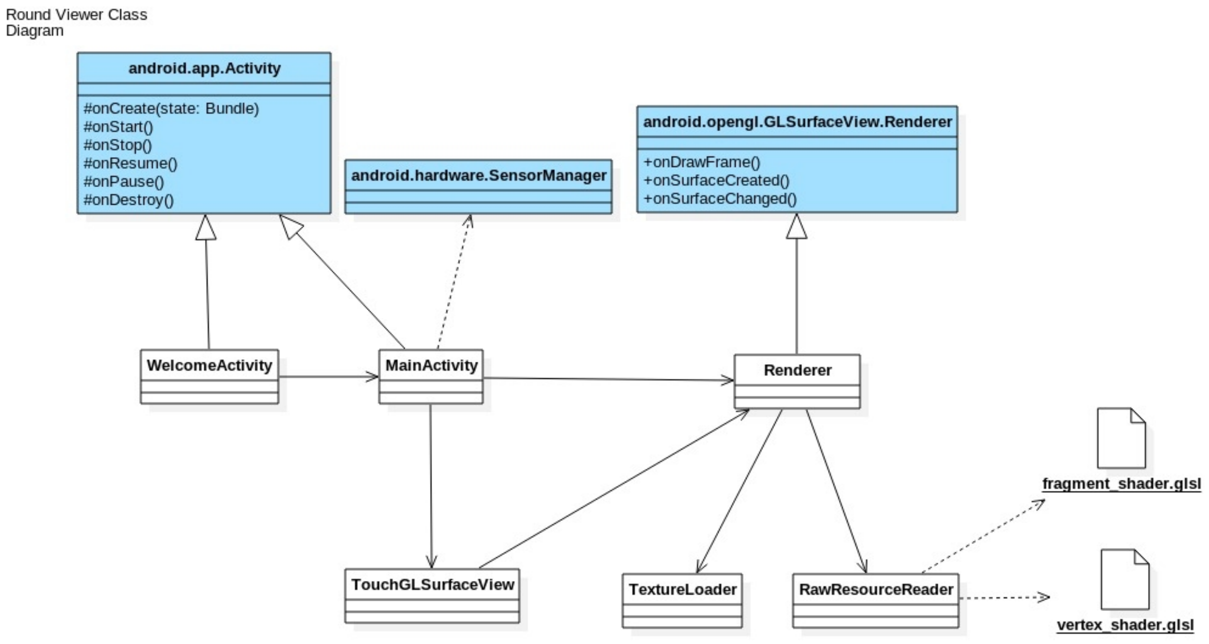

Fig. 5 – Round Viewer class diagram

2.4. Testing

The application was tested using two physical devices: a Motorola Moto G 2013 and a Sony Xperia Z3 Compact, both running Android 5.0. The first device does not have a gyroscope, and hence rotation vector input cannot be read and touch screen is enabled. The second one does have the sensors required to get the rotation of the device. On both of the devices the application behaved as expected.

Only verification and validation testing was used for this project. Due to the nature of the application, unit testing does not apply.

3. Screenshots

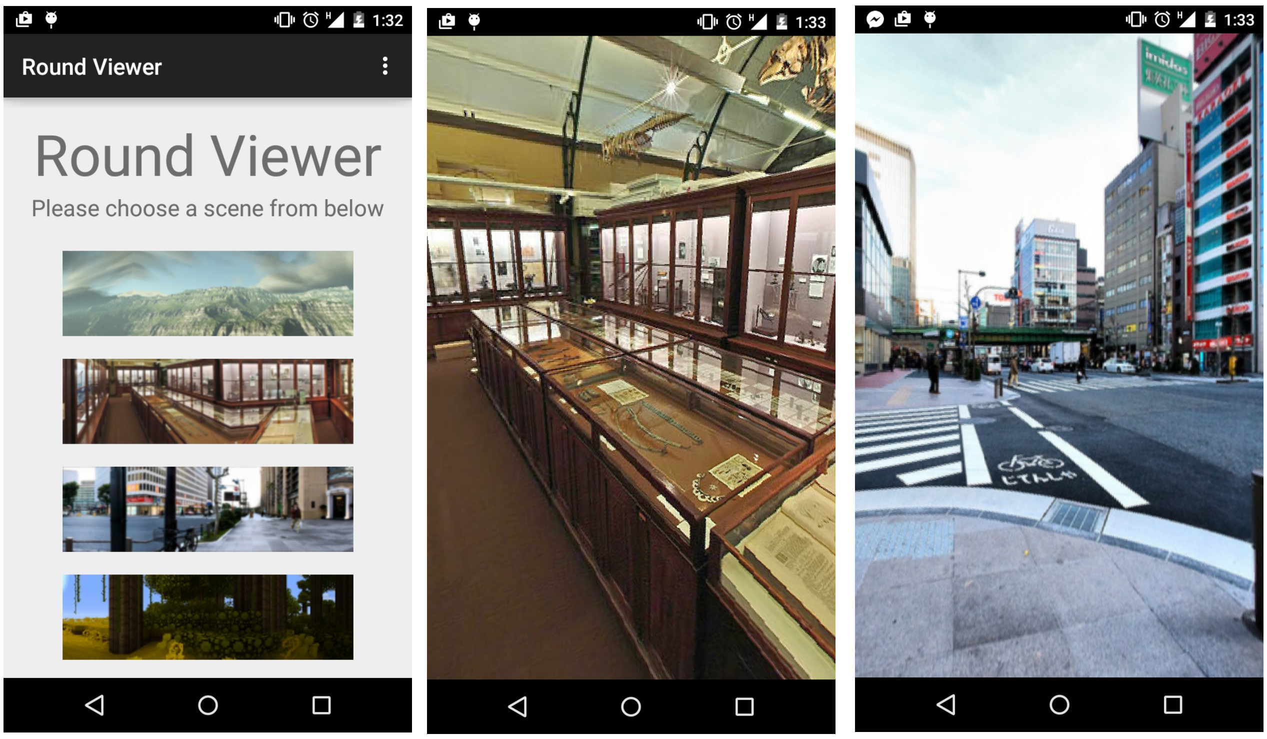

Fig. 6 – Screenshots of Round Viewer