Laboratory 3
3D Rendering

The Vertex Transformation Pipeline

Remember that OpenGL ES objects are a collection of points in 3D space; that is, their location is defined by three values. These values are joined together to form faces, which are flat surfaces that look remarkably like triangles. The triangles are then joined together to form objects or pieces of objects.

To get a series of numbers that form vertices, other numbers that form colors, and still other numbers that combine the vertices and colors on the screen, it is necessary to tell the system about its graphic environment. Such things as the location of the viewpoint, the window (or viewport) that will receive the image, aspect ratios, and other things that are needed to complete the 3D circuit. More specifically, we will cover OpenGL's coordinates, how they relate to the frustum, how objects are clipped or culled from the scene, and the drawing to your device's display.

OpenGL Coordinates

OpenGL 3D coordinates, we have a system using *Cartesian* coordinates, the standard of expressing locations in space. Typically, for screen coordinates on 2D displays, the origin is in the upper-left corner with +X going right and +Y going down. However, OpenGL has the origin in the lower-left corner, with +Y going up. But now we add a third dimension, expressed as Z. In this case, +Z is pointing toward you, as shown in Figure 1.

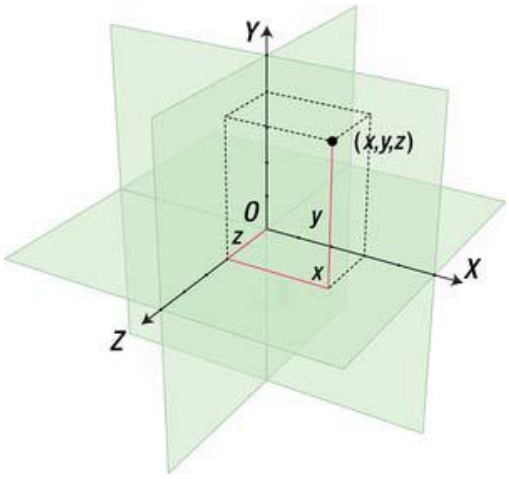

Figure 1: OpenGL ES 3D Cartesian coordinate system

In fact, we have several kinds of coordinate systems, or *spaces*, in OpenGL, with each space being transformed to the next:

- *Object* space, which is relative to each of your objects.
- Camera, or *eye*, space, local to your viewpoint.
- Projection, or *clip*, space, which is the 2D screen or viewport that displays the final image.
- *Tangent* space, used for more advanced effects such as bump-mapping, which will be covered in later laboratories.
- Normalized device coordinates (NDCs), which express the xyz values *normalized* from -1 to 1. That is, the value (or set of values) is normalized such that it fits inside a cube 2 units on a side.
- Windows, or *screen*, coordinates, which are the final locations of your scene when displayed in the actual screen.

Naturally the previous can be expressed in pipeline form, as shown in Figure 2.

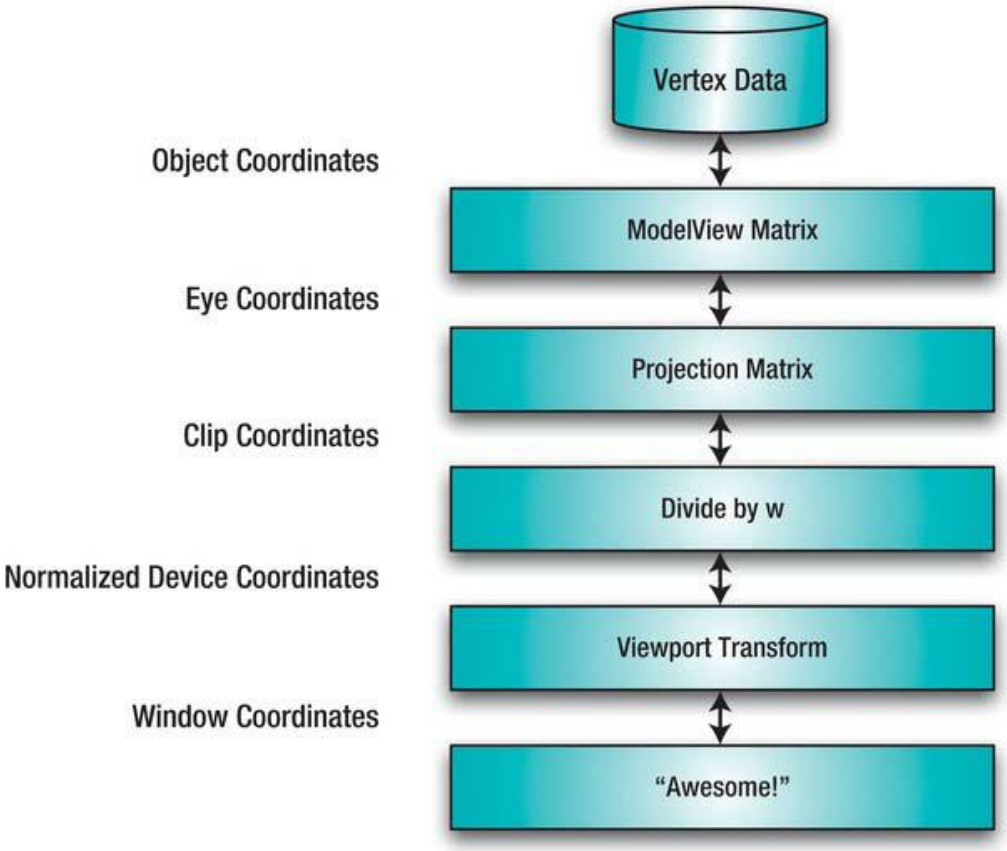

Figure 2: Vertex transformation pipeline

Object, eye, and clip space are the three you usually have to worry about. For example, object coordinates are generated with a local origin and then moved and rotated to *eye space*. If you have a bunch of airplanes for a combat game, for example, each will have their own local origin. You should be able to move the planes to any part of your world by moving, or *translating*, just the origin and letting the rest of the geometry follow along. At this point, the visibility of objects is tested against the viewing frustum, which is the volume of space that defines what the virtual camera can actually see. If they lay outside the frustum, they are considered invisible and so are clipped, or culled out, so that no further operations are done on them. Much of the work in graphics engine design focuses on the clipping part of the engine,

so as to dump as many of the objects as early as possible to yield faster and more efficient systems.

And finally, after all of that, the screen-oriented portions of OpenGL are ready to convert, or *project*, the remaining objects. And those objects are your planes, zeppelins, missiles, trucks on the road, ships at sea, and anything else you include in your application.

OpenGL doesn't really define anything as "world space". However, the eye coordinates are the next best thing in that you can define everything in relation to your location.

Eye Coordinates

There is no magical eyepoint object in OpenGL. So, instead of moving your eyepoint, you move all of the objects in relation to your eyepoint. In eyepoint relative coordinates, instead of moving away from an object, the object, in effect, is moving away from you. Imagine you are making a video of a car rushing by. Under OpenGL, the car would be standing still; you and everything around you would be moving by it. This is done largely with the `glTranslate` and `glRotate` calls, as you will see later. It is at this point where OpenGL's modelview matrix comes into play. The ModelView matrix handles the basic 3D transformations (as opposed to the Projection matrix, which *projects* the 3D view onto the 2D space of your screen, or the Texture matrix, which helps apply images to your object). You will refer to it frequently.

Viewing Frustum and the Projection Matrix

In geometry, a *frustum* is that portion of (typically) a pyramid or cone that results after being cut by two parallel planes. In graphics, the viewing frustum defines the portion of the world that our virtual camera can actually see, as shown in Figure 3.

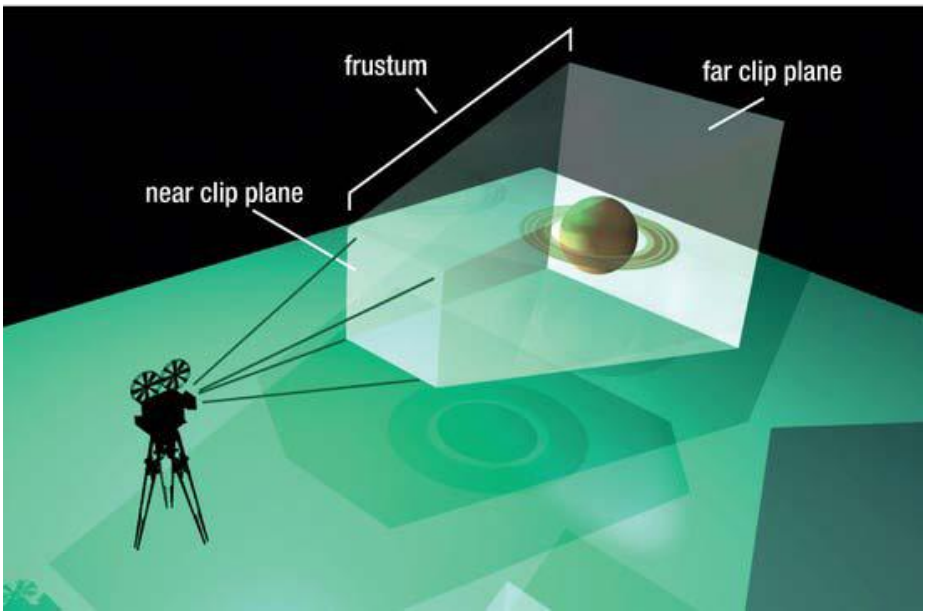

Figure 3: Viewing frustum

Unlike a number of things in OpenGL, the definition of the viewing frustum is very straightforward and follows the conceptual figures closely by simply defining a volume, the viewing pyramid, in space. Any objects that are whole or in part within the frustum may eventually find their way to the screen as long as it is not obscured by anything closer.

The frustum also is used to specify your field-of-view (FOV), like your camera's wide-angle vs. telephoto lens. The larger the angle that the side planes form when compared to the center

axis (that is, how they fan out), the larger the FOV. And a larger FOV will allow more of your world to be visible but can also result in lower frame rates.

Up to this point, the translations and rotations use the ModelView matrix, easily set using the call `gl.glMatrixMode(GL_MODELVIEW);`. But now at this stage of the rendering pipeline, you will define and work with the Projection matrix. It is a surprisingly compact means of doing a lot of operations.

The final steps to convert the transformed vertices to a 2D image are as follows:

- A 3D point inside the frustum is mapped to a normalized cube to convert the XYZ values to NDC. NDC stands for normalized device coordinates, which is an intermediate system describing the coordinate space that lays inside the frustum and is resolution independent. This is useful when it comes to mapping each vertex and each object to your device's screen, no matter what size or how many pixels it has, be it a phone, tablet, or something new with completely different screen dimensions. Once you have this form, the coordinates have "moved" but still retain their relative relationships with each other. And of course, in NDC, they now fall into values between -1 and 1. Note that internally the Z value is flipped. Now –Z is coming toward you, while +Z is going away, but thankfully that great unpleasantness is all hidden.

- These new NDCs are then mapped to the screen, taking into account the screen's aspect ratio and the "distance" the vertices are from the screen as specified by the near clipping plane. As a result, the further things are, the smaller they are. Most of the mathematics is used for little more than determining the proportions of this or that within the frustum.

The previous steps describe *perspective projection*, which is the way we normally view the world. That is, the further things are, the smaller they appear. When those inherent distortions are removed, we get *orthographic projection*. At that point, no matter how far an object is, it still displays the same size. Orthographic renderings are typically used in mechanical drawings when any perspective distortion would corrupt the intent of the original artwork.

You will often need to directly address which matrix you are dealing with. The call to `gl.glMatrixMode()` is used to specify the current matrix, which all subsequent operations apply to until changed. Forgetting which matrix is the current one is an easy error to make.

The `Cube` Class

Like in the previous laboratory, create an Android project named `BouncyCube` in the `cg.bouncycube` package, with an empty activity `BouncyCubeActivity`, and no xml layout generated.

The first class we should add to our project is the `Cube` class.

The `Cube` class will consist of a constructor and a `draw` method.

The Constructor

The first thing we need to do in the constructor is to define the vertices. Here, since we are describing a 3D cube, it is necessary to specify `x`, `y` and `z` coordinates.

```
float vertices[] =
{
    -1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, -1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f,
```

```
        -1.0f, 1.0f, -1.0f,
        1.0f, 1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f, -1.0f
    };
```

And as you can see, the cube is two units on a side.

Colors are defined similarly, but in this case, there are four components for each color: red, green, blue, and alpha (transparency). These map directly to the eight vertices shown earlier, so the first color goes with the first vertex, and so on.

```
    byte maxColor=(byte)255;
    byte colors[] =
    {
        maxColor, 0, 0, maxColor,
        maxColor, 0, 0, maxColor,
        maxColor, 0, 0, maxColor,
        maxColor, 0, 0, maxColor,
        0, 0, 0, maxColor,
        0, 0, 0, maxColor,
        0, 0, 0, maxColor,
        0, 0, 0, maxColor,
    };
```

Figure 4 shows the way the vertices are ordered. Under normal situations, you will never have to define geometry in this fashion. You will likely load your objects from a file stored in one of the standard 3D data formats, such as those used by 3D Studio or Modeler 3D. And considering how complicated such files can be, it is not recommended that you write your own because importers for most of the major formats are available. Notice the various axes: X is going right, Y is going up, and Z is going toward the viewer.

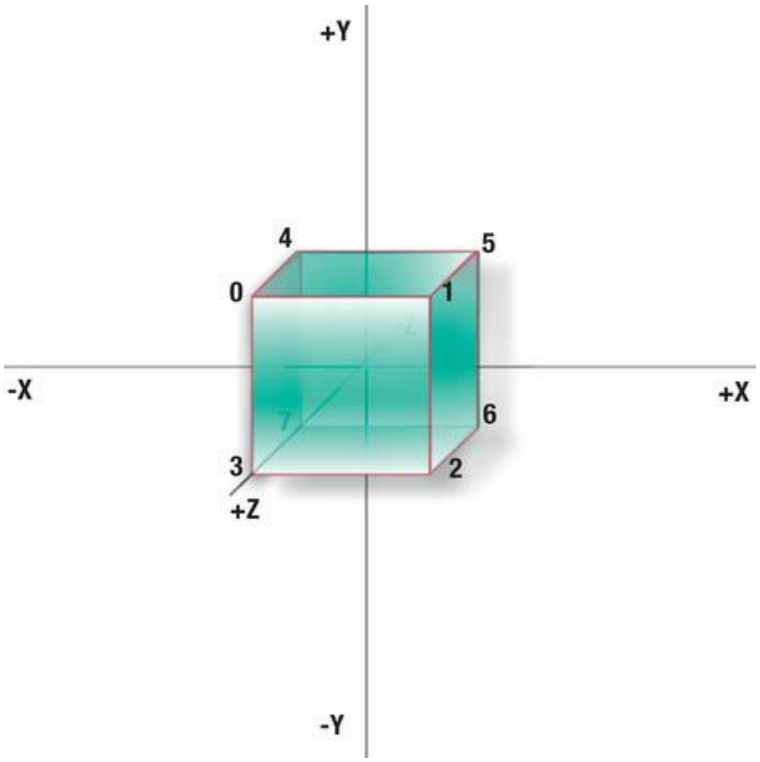

Figure 4: The order of the vertices

Some new data is now needed to tell OpenGL in what order the vertices are to be used. With the square, it was easy to order, or *sequence*, the data by hand so that the four vertices could represent the two triangles. The cube makes this considerably more complicated. We could have defined each of the six faces of the cube by separate vertex arrays, but that wouldn't scale well for more complex objects. And it would be less efficient than having to push six sets of data through the graphics hardware. Keeping all the data in a single array is the most efficient from both a memory and a speed standpoint. So, then, how do we tell OpenGL the layout of the data? In this case, we will use the drawing mode called *triangle fans*, as shown in Figure 5.

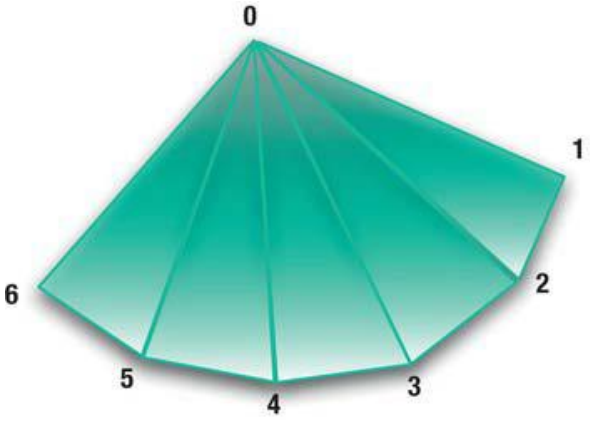

Figure 5: Triangle fan

There are many different ways data can be stored and presented to OpenGL ES. One format may be faster but uses more memory, while another may use less memory but at the cost of a little extra overhead. If you were to import data from one of the 3D files, chances are it is already optimized for one of the approaches, but if you really want to hand-tune the system, you may at some point have to repack the vertices into the format you prefer for your application.

Besides triangle fans, you will find other ways data can be stored or represented, called *modes*.

- Points and lines specify just that: points and lines. OpenGL ES can render your vertices as merely points of definable sizes or can render lines between the points to show the wireframe version. Use `GL_POINTS` and `GL_LINES`, respectively.

- Line strips, `GL_LINE_STRIP`, are a way for OpenGL to draw a series of lines in one shot, while line loops, `GL_LINE_LOOP`, are like line strips but will always connect the first and last vertices together.

- Triangles, triangle strips, and triangle fans round out the list of OpenGL ES primitives: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, and `GL_TRIANGLE_FAN`. OpenGL itself can handle additional modes such as quads (faces with four vertices/sides), quad strips, and polygons.

When using these low-level objects, you may recall that in Laboratory 2, there was an *index*, or *connectivity*, *array* to tell which vertices matched which triangle. When defining the triangle arrays, called `tTan1` and `tTan2` below, you use a similar approach except all sets of indices start with the same vertex. So, for example, the first three numbers in the array `tTan1` are 1, 0, and 3. That means the first triangle is made up of vertices 1, 0, and 3, in that order. And so, back in the array vertices, vertex 1 is located at x=1.0f, y=1.0f, and z=1.0f. Vertex 0 is the point at x=-1.0f, y=1.0f, and z=1.0f, while the third corner of our triangle is located at x=-1.0, y=-1.0, and z=1.0. The upside is that this makes it a lot easier to create the datasets because

the actual order is now irrelevant, while the downside is that it uses up a little more memory to store the additional information.

The cube can be divided up into two different triangle fans, which is why there are two index arrays. The first incorporates the front, right, and top faces, while the second incorporates the back, bottom, and left faces, as shown in Figure 6.

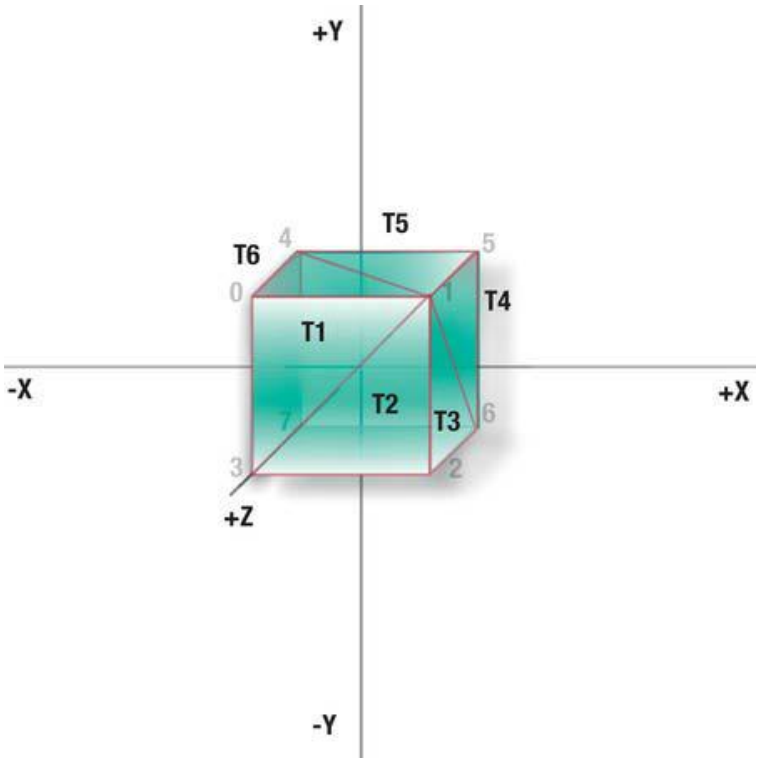

Figure 6: Triangle fan

```
byte tFan1[] =
{
    1,0,3,
    1,3,2,
    1,2,6,
    1,6,5,
    1,5,4,
    1,4,0
};
byte tFan2[] =
{
    7,4,5,
    7,5,6,
    7,6,2,
    7,2,3,
    7,3,0,
    7,0,4
};
```

Once the colors, vertices, and connectivity arrays have been created, we may have to convert the internal Java formats of the values given above to those that OpenGL can understand, as shown below. This mainly ensures that the ordering of the bytes is right; otherwise, depending on the hardware, they might be in reverse order.

```
ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
vbb.order(ByteOrder.nativeOrder());
mFVertexBuffer = vbb.asFloatBuffer();
mFVertexBuffer.put(vertices);
mFVertexBuffer.position(0);
mColorBuffer = ByteBuffer.allocateDirect(colors.length);
mColorBuffer.put(colors);
mColorBuffer.position(0);
mTFan1 = ByteBuffer.allocateDirect(tFan1.length);
mTFan1.put(tFan1);
mTFan1.position(0);
mTFan2 = ByteBuffer.allocateDirect(tFan2.length);
mTFan2.put(tFan2);
mTFan2.position(0);
```

The `draw` Method

The `draw(GL10 gl)` method is called by `CubeRenderer.drawFrame()`, discussed later.

The method contains calls to `glVertexPointer` and `glColorPointer` analogue to the ones in the previous laboratory.

Lastly, the call to `glDrawElements` from the `BouncySquare` is replaced by two calls, corresponding to the two triangle fans:

```
gl.glDrawElements( gl.GL_TRIANGLE_FAN, 6 * 3, gl.GL_UNSIGNED_BYTE, mTFan1);
gl.glDrawElements( gl.GL_TRIANGLE_FAN, 6 * 3, gl.GL_UNSIGNED_BYTE, mTFan2);
```

The `CubeRenderer` Class

Now our cube needs a driver and a way to display itself on the screen. Create a new class called `CubeRenderer` which implements the `GLSurfaceView.Renderer` interface.

The Constructor

In the constructor, the `Cube` object is allocated and cached. Thus, we need to add a private field to the `CubeRenderer` class

```
private Cube mCube;
```

and initialize it in the constructor

```
mCube = new Cube();
```

The `onDrawFrame` method

The `onDrawFrame(GL10 gl)` method is the identical to the one from the `SquareRenderer` class, with two exceptions:

- The call to `glTranslatef` is now

```
gl.glTranslatef(0.0f, (float) Math.sin(mTransY), -7.0f);
```

- Let us add some more interesting animation to the scene. We are going to be spinning the cube slowly besides bouncing it up and down. For this, we use the `private float mAngle` field to store the current rotation angle. The first call defines a rotation about the Y axis, and the second call defines a rotation about the X axis. At each new call to the `onDrawFrame` method, the angle is increased by 0.4.

  ```
  gl.glRotatef(mAngle, 0.0f, 1.0f, 0.0f);
  gl.glRotatef(mAngle, 1.0f, 0.0f, 0.0f);
  mAngle += 0.4;
  ```

  The above code should be placed after the call to `glTranslatef`, because the order of transformations is actually applied from last to first, meaning that we first have a rotation about the X axis, then a rotation about the Y axis, and then the translation that moves the cube up and down. Note also that the application of transformations is not commutative, meaning that translation after rotation and rotation after translation will not yield the same result.

The `onSurfaceChanged` method

The `onSurfaceChanged(GL10 gl, int width, int height)` is used to set up the viewing *frustum*, which is the volume of space that defines what you can actually see.

The calls to `glViewport`, `glMatrixMode` and `glLoadIdentity` remain identical with the ones from the `SquareRenderer` class.

Now you can set the actual viewing frustum. There is somewhat more information here so as to handle varying display sizes as well as making the code a little more easily understood. First, we define the frustum with a given FOV, making a little more intuitive when choosing values. The field of 30 degrees is converted to radians as the Java Math libraries require, whereas OpenGL sticks to degrees. The aspect ratio is based on the width divided by the height. So if the width is 1024x768, the aspect ratio would be 1.33. This helps ensure that the proportions of the image scale properly. Otherwise, were the view not to take care of the aspect ratio, its objects would look squashed. Then, we define the values of the `zNear` and `zFar` planes. The next line has the duty of calculating a size value needed to specify the left/right and bottom/top limits of the viewing volume, as shown in Figure . This can be thought of as your virtual window into the 3D space. With the center of the screen being the origin, you need to go from `-size` to `+size` in both dimensions. That is why the field is divided by two—so for a 60-degree field, the window will go from -30 degrees to +30 degrees. Multiplying size by `zNear` merely adds a scaling hint. Finally, divide the bottom/top limits by the aspect ratio to ensure your square will really be a square. Now we can feed those numbers into `glFrustum`, followed by resetting the current matrix back to `GL_MODELVIEW`.

```
float fieldOfView = 30.0f/57.3f;
float aspectRatio = (float)width/(float)height;
float zNear = 0.1f;
float zFar = 1000;
float size = zNear * (float)(Math.tan((double)(fieldOfView/2.0f)));
gl.glFrustumf(-size, size, -size/aspectRatio, size/aspectRatio, zNear, zFar);
gl.glMatrixMode(GL10.GL_MODELVIEW);
```

The `onSurfaceCreated` Method

The last method of the `CubeRenderer` class, namely `onSurfaceCreated(GL10 gl, EGLConfig config)`is identical to the one from the previous Laboratory.

The `BouncyCubeActivity` Class

Finally, the `BouncyCubeActivity` class is identical to the `BouncySquareActivity` class, except the fact that we use the `CubeRenderer` instead of the `SquareRenderer`.

Now compile and run. You should see something that looks like Figure .

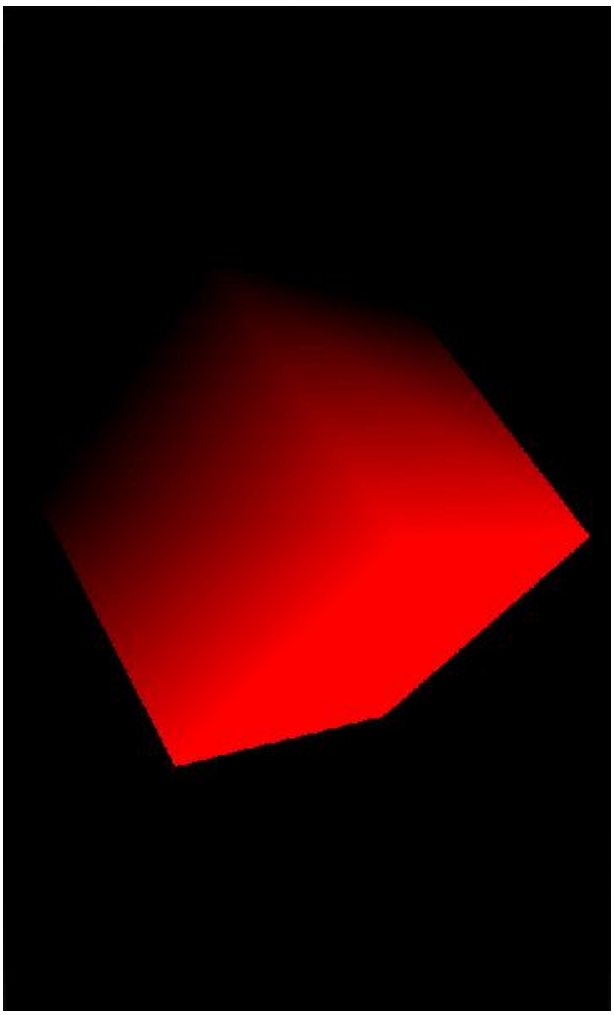

Figure 7: Triangle fan

Assignment

- Change the order the two rotations and the translation is applied to the cube.
- One final transformation command to be aware of is `glScale`, used for resizing the model along all three axes. Let's say you need to double the height of the cube. You would use the line `gl.glScalef(1,2,1);`. Remember that the height is aligned with the Y-axis, while width and depth are X and Z, which we don't want to touch. Where would you put the line to ensure that the geometry of the cube is the only thing affected? Try all possibilities.

- Change the far clipping plane in the frustum by changing the value of `zFar` from 1000 down to 6.0.
- Reset `zFar` to 1000, and set `zNear` from 0.1 to 6.0.
- Change the `z` value in `gl.glTranslatef` to -20.
- Change `fieldOfView=10` degrees from 30 degrees.
- The call `gl.glEnable(GL_CULL_FACE);` enables backface culling, which means that the faces on the rear of an object won't be drawn, because they would never be seen anyway. It works best for convex objects and primitives such as spheres or cubes. The system calculates the face normals for each triangle, which serve as a means to tell whether a face is aimed toward us or away. By default, face windings are counterclockwise. So if a CCW face is aimed toward us, it will be rendered while all others would be culled out. You can change this behavior in two different ways should your data be nonstandard. You can specify that front-facing triangles have clockwise ordering or that culling dumps the front faces instead of the rear ones. To see this in action, add the following line to the `onSurfaceCreated` method: `gl.glCullFace(GL10.GL_FRONT);`.