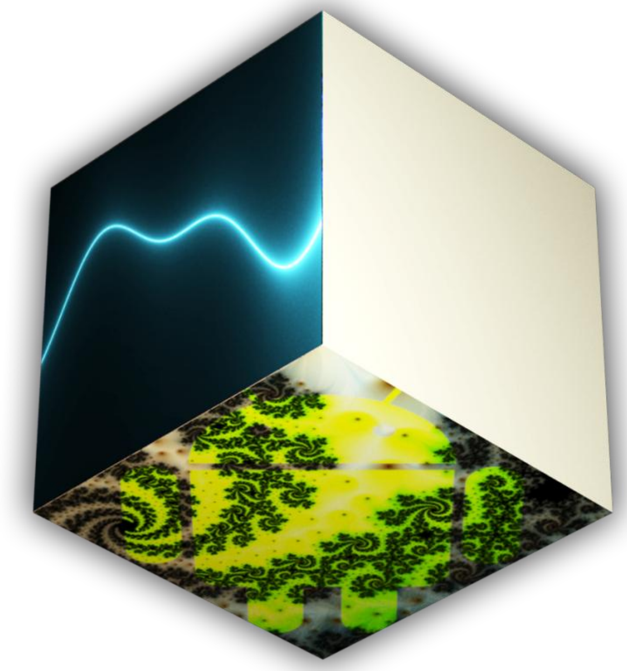


ProtoShade

Fragment Shader Design Tool for the OpenGL ES Shading Language

Major Assignment

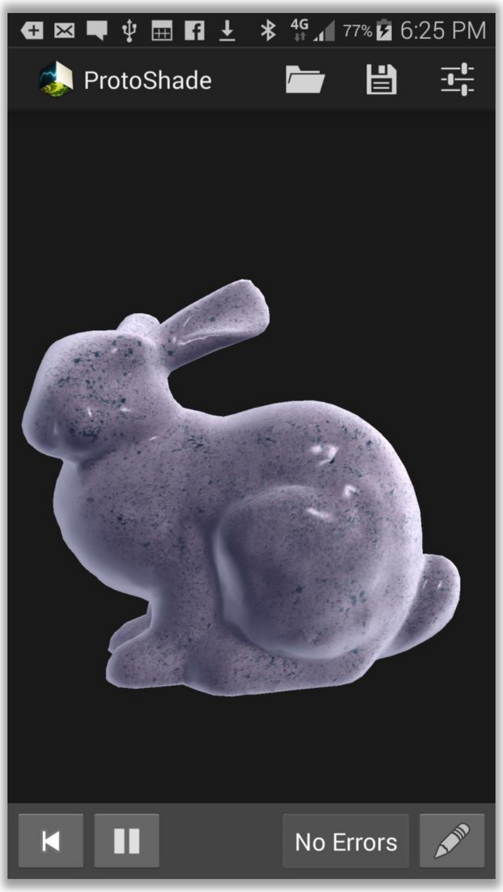
For Mobile Applications Development (**31285**)

By 11688025

Deinyon Davies, Spring 2014, for UTS School of Software

Application Overview

ProtoShade is a fragment (pixel) shader prototyping and design utility for the OpenGL ES Shading Language, with an emphasis on rasterizer special effects design, to differentiate from existing "viewport-sized quadrangle" fragment shader design utilities.



The user might use the application to prototype lighting algorithms, or to design unique special effects which are projected onto a three-dimensional model. The program supplies **four** example shader programs, and **five** models.

The project is composed of **30** Java classes, of which some contain inner classes, and are segmented into **6** packages (including the application package). The codebase contains a **custom graphics engine** and Object-Oriented OpenGL abstractions, a simple **linear algebra library**, a **Wavefront OBJ Model Parser**, and various Android-specific classes for shader storage, resource loading with progress feedback, and various forms of multi-threaded communication (including Broadcast Listeners and queued Runnables).

The application presents an innovative user interface, including a **custom Surface View**, a **custom Alert Dialog**, a dynamic **Adapter Alert Dialog**, a SQLite-Database-powered List Activity, and a Preference Activity.

Using ProtoShade

The program launches a Phong and Rim Shading example shader. To edit the currently visible shader, press . The buttons at the lower-right reset, play and pause the shader uniform variable "time", which may be used within your shader to create animations. The "No Errors" button will display the number of compiler errors (if any), and will become enabled when there are errors. Note that changes to the shader source code are automatically compiled and displayed.

Use the three Action Bar buttons to **Load**, **Save**, and change **Preferences**. The *Load Shader* Activity displays a list of previously saved shaders, as well as pre-installed sample shaders. The *Save Shader* Dialog displays a thumbnail-size render of the current shader, projected onto the model. The *Application Preferences* Activity provides the ability to select the desired model, and to enable or disable model rotation while the Code Editor is activated.

Development Log

Early Demonstration & Linear Algebra Library

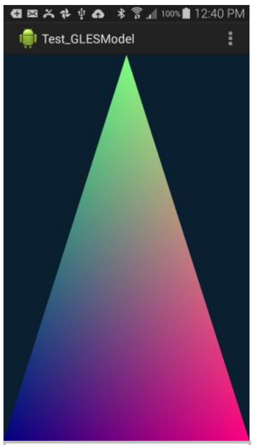


Figure 1.0 | The first OpenGL ES demonstration program.

The project began on the *20th of September* as an OpenGL ES graphics demonstration, with the goal of creating an application that would display a two-dimensional Gouraud shaded triangle, with an arbitrary GLSL (OpenGL Shading Language) shader, in a highly modular and abstracted framework, or "game engine". See **Figure 1.0** for a screenshot of the completed program.

It was obvious that the program required a linear algebra mathematics library with basic support for vector and matrix operations, and optimally, computer-graphics matrix operations, including camera-based coordinate system matrix definition. **Four** Java linear algebra libraries were considered:

- **Vecmath**: The Javax vector package
- **la4j**: The "Linear Algebra for Java" library
- **JAMA**: The "Java Matrix Package"
- **EJML**: The "Efficient Java Matrix Library"

Of the four libraries, the Javax **Vecmath** package was determined as most suited to light-weight vector-matrix computation without the equation solving overhead of other libraries. However, the Vecmath library still did not entirely meet the requirements for some computer graphics operations, leading to the development of a simple **custom linear algebra library**.

The custom linear algebra library supports four-by-four matrix, 2D vector and 3D vector storage and operations, exposing both static and object methods for binary operations (multiplication, addition, …) to compensate for Java's lack of operator overloading support. The Matrix class supports two computer-graphics functions for reference frame construction, including perspective projection (see **Figure 1.1**), and "look-at' coordinate system construction (see **Figure 1.2)**.

$$M_{perspective} = \begin{bmatrix} \frac{\cot\left(\frac{1}{2}\theta\right)}{\frac{width}{height}} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{1}{2}\theta\right) & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 \times far \times near}{far - near} \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Figure 1.1 | Perspective Projection matrix, which transforms geometry into a viewing frustum, and prepares the W component for screen-space division. θ is the viewport "Field of View" (FOV) angle, *width* and *height* are the viewport dimensions, and *far* and *near* are the frustum's Z-clipping-planes.

$$V_{up} = |(0,1,0)|$$
$$N = \left|V_{target} - V_{position}\right|$$
$$T = N \times V_{up}$$
$$B = T \times N$$
$$M_{lookat} = \begin{bmatrix} T_x & T_y & T_z & 0 \\ B_x & B_y & B_z & 0 \\ N_x & \boldsymbol{N}_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & -V_{position_x} \\ 0 & 1 & 0 & -V_{position_y} \\ 0 & 0 & 1 & -V_{position_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 1.2 | Computation of the "Look-at" matrix, which orients the coordinate system so that the viewer at *Vposition* faces *Vtarget*, by constructing a coordinate system from the camera's direction.

Finally, the test program implements a simple per-vertex Gouraud GLSL ES shader, which performs bilinear interpolation across the triangle's three vertex colours, where the three colours are the normalised vertex coordinates.

The triangle rotates in 3D space by multiplying each of its vertices by an arbitrary Y-rotation matrix within the GLSL vertex shader program.

Perspective-Correct Texture Sampling Demonstration

The project requires a mechanism for parsing various texture image data from a standard Android or filesystem resource into a unified uncompressed format for use with the OpenGL ES texture sampling system. Moreover, each vertex must now support an additional two-dimensional vector, known as the U/V or 'texture' coordinates, where necessary (ie, when the model has a texture).

Vertices are now defined by their own types, in place of single Vector objects. Vertex types are defined in a polymorphic fashion:

- **VertexPosition**
  (3D Position vector (P) only)
  - **VertexPositionNormalTexture**
    (Above, plus: 3D normal vector (N) and 2D U/V coordinate (TC))
    - **VertexPositionNormalTextureTangent**
      (Above, plus: Two additional 3D tangent-space vectors (T & B)

The tangent-space vectors are not yet implemented, though they shall be used to create lighting effects such as Normal Mapping.

A 2D texture class was created (Texture2D) which is used to load and store OpenGL ES texture resources as Java objects. Currently, the Texture2D class parses Android drawable resources to OpenGL ES textures.

The Vector classes provide standard geometric vector operations, as well as methods for exposing the lower-level storage operations for OpenGL Vertex Buffer creation.

Wavefront OBJ Mesh Parser & Phong Vertex Specularity Shader

In order to display arbitrary 3D meshes, it was necessary to select an existing 3D model format, and implement a custom 3D model parser. Influenced by personal experience, extensive documentation, and widespread popularity, the Wavefront OBJ model format was selected. See **Figure 2.0** for a screenshot of the completed test program.

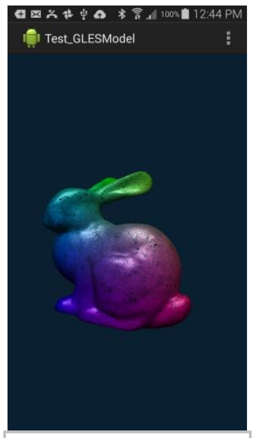


Figure 2.0 | Phong Specularity & Gouraud Interpolation

The Wavefront OBJ file format provides vertex and face declarations in plain ASCII text, where each line is prefixed by the type of vertex or face declaration that the line contains. The **OBJModel** class uses a BufferedReader to read each line from the OBJ file into one of five ArrayLists with the type of **OBJLine**.

The **OBJLine** class and its subclasses store one line from the OBJ file as a string, and its associated data which has been parsed to the appropriate format. A **CoordinateLine** stores a single coordinate (point, vector or texture coordinate), whereas a **FaceLine** defines the linkage of three vertices to form one triangle.

The OBJ model parser requires that all faces are triangulated in order to be stored and drawn correctly. Additionally, the program shall not support group segregation, nor will it support OBJ MTL materials, for these features are not applicable to the application.

After having successfully parsed the OBJ model, vertex data is transferred into a **Vertex Buffer Object**, which is a generic class whose generic type extends **VertexPosition**. The vertex buffer is supplied to the model's **Shader**, which constructs GLSL array attributes from the buffer. Additionally, the **Model** class stores the world transform matrix for the geometry, which is multiplied by the camera's projection matrix, and is supplied to the shader as the "MV-Matrix".

Per-Fragment Lighting & Project Re-Structure

On the *20th of October*, the project was re-structured from the original project proposal; an asteroid shooter game. The re-structure was the result of several unforseen delays, which resulted in approximately three weeks of lost development time. Subsequently, the interface was subject to a major re-design. See **Figure 3.0** for a screenshot of the new interface design prototype.

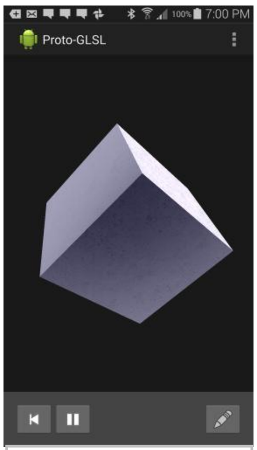


Figure 3.0 | New Shader & Interface Design

The interface design required that the renderer provide its display buffer to a custom Surface View in order to be drawn in-line with other interface elements.

It was now necessary to carefully define the project scope, so as to produce a reliable and well-designed application by the project submission date.

Inter-Thread Communication

For various operations in the application, it is necessary to communicate data between the renderer thread and the User Interface thread, or between a resource loader thread.

In the case of OBJ model loading, an AsyncTask coordinates parsing of the OBJ file, and updates a progress dialog. For messages that are delivered from the User Interface thread to the renderer thread, messages are appended to a queue of Runnables, which are executed by the renderer thread upon each draw. Messages that must be conveyed to the User Interface thread are delivered through Broadcast Intents, and are received by the Main Activity through a custom Broadcast Receiver.

Shader Storage, Compiler Output & Finalization

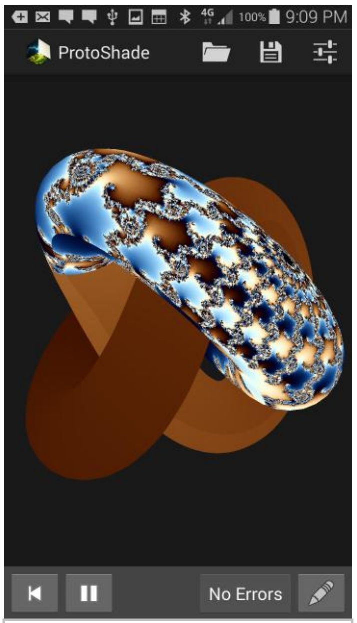


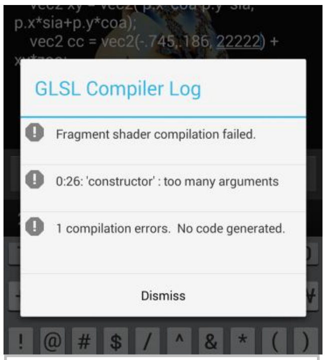
Figure 4.0 | Mandelbrot-Set Fractal Projected Onto a Torus Knot.

On the *27th of October*, the shader storage implementation was completed. Shader metadata (including: 'Title', 'Required' and the path to the source code) is stored within a local SQLite database, whereas the actual shader source code and thumbnail render are stored within the device's internal storage.

The compiler error list is generated by evaluating the result of shader compilation, using OpenGL's glGetShaderiv() function. The program splits the fragment shader output log by a delimiter, and provides each error string to an Alert Dialog through an Array Adapter. The dialog is created on the User Interface thread, after having been called through the Broadcast Receiver.

A Preference Activity allows the user to select one of the supplied OBJ models for rendering. An "onPreferenceChanged" event listener is implemented in order to invalidate the renderer when a new model is selected.

The application was made more robust by testing for bugs. A bug was found where animations would greatly lose precision over time, as the result of the precision of 4-byte floating point numbers. To repair the error, the animation timer is reset when the main activity is resumed.



```
p.x*sia+p.y*coa);
   vec2 cc = vec2(-.745,.186,.22222) +
```

GLSL Compiler Log

```
Fragment shader compilation failed.
0:26: 'constructor' : too many arguments
1 compilation errors.  No code generated.
```

Figure 4.1 | The Compiler Log Dialog.

References

Sellers, G., Wright, R.S. Jr., Haemed, N. 2013, *OpenGL SuperBible*, 6th edn, Pearson Education.