Interactive graphics application for Android

Felix Olsson*

Lund University
Sweden

Abstract

OpenGL es is currently the most widely deployed graphics api. This paper describes the the creation of interactive graphical application and some of its features, such as post-processing effects. This project is build for android using openGL es 2.0 and 3.0.

1 Introduction

The goal of this project was to create an interactive graphics application for android from the ground up, in the form of a functional game and a framework that could easily be expanded upon, utilizing openGL es 2.0 and 3.0.

The game is viewed in androids *landscape mode* Figure 1 and consists of a controllable spaceship, a number of starts and some asteroids.

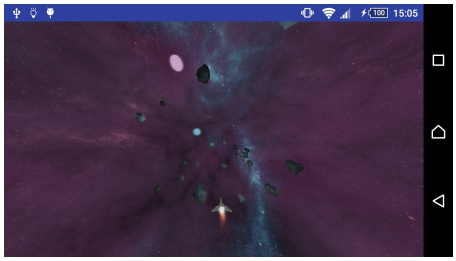

Figure 1: This is what the game looks like.

An interactive product generally needs lighting, otherwise it will look static, for that aspect the project contains a simplistic lighting shader.

It is possible to generate vertices, texture coordinates and normals for simple objects such as a cube. However for more complex objects it can be very time consuming or even impossible to create such with an algorithm, this is where a model loading application can be helpful by importing and loading models made in a model creation software into memory.

Interactivity through touch is a corner stone of smart-phones devices therefore the project contains a function to map the point touched on the screen to coordinates in the openGL application. The project also includes user feedback in the form of vibrations.

Modular post-processing effects, such as blurr effects, are implemented with the use of kernels, and fire effects are implemented with the use of a particle system, this in order to increase the visual appeal of the application.

2 Application

The simple lighting shader is based around the phong shading model with added attenuation for increased realism. In this implementation the models diffuse and specular textures are taken into consideration as well as the lights attributes such as diffuse light and ambiance. Attenuation is a way of emulating the effect of a fading point light.

*e-mail: dat12fol@student.lu.se

$$L_{distance} = \frac{1}{a_{const} + a_{lin} * dist + a_{exp} * dist * dist}$$

For desired effect the following values where chosen $a_{const} = 1.0, a_{lin} = 0.014, a_{exp} = 0.0007$[1]

The model loading class reads from a .obj file, analyzing each line and in accordance with the file structure for an .obj file and loads data into a series of buffers[2]. Currently the model loader supports vertex coordinates, texture coordinates and normals in the .obj format[3] and the file must be written with triangulated faces.

Touch controls are implemented with the help of androids on touch listener[4] the coordinates returned from the touch event are mapped in screen coordinates, in order to get usable coordinates for openGL the values must be normalized by dividing by the width and height and invert the Y-axis . Since the coordinates only describe a point in 2D space, a ray is created from the point on the near end of the frustum to the far end of the frustum. The depth is calculated as the intersection point of the ray and a given plane [Brothaler 2013a].

The touch controls are used in several aspects of the game. If the user presses in the vicinity of the spaceship a laser object will spawn, traveling in a straight line from the spaceship. If the user instead presses further away from the spaceship, the spaceship will travel to that location every frame by a distance defined by linear interpolation[5] between the start position and the destination.

In order to apply post-processing effects a *framebuffer* is bound during run-time, the *framebuffer* contains two textures. By specifying to which of the textures a shader should send its output to and what values, and then applying an image-transformation algorithm to the textures, post-processing effects can be achieved. The image-transformation algorithm implemented in the current iteration of the project is a Gaussian-blur approximation kernel[6] [Powell],

$$\begin{bmatrix} 1.0/256.0 & 4.0/256.0 & 6.0/256.0 & 4.0/256.0 & 1.0/256.0 \\ 4.0/256.0 & 16.0/256.0 & 24.0/256.0 & 16.0/256.0 & 4.0/256.0 \\ 6.0/256.0 & 24.0/256.0 & 36.0/256.0 & 24.0/256.0 & 6.0/256.0 \\ 4.0/256.0 & 16.0/256.0 & 24.0/256.0 & 16.0/256.0 & 4.0/256.0 \\ 1.0/256.0 & 4.0/256.0 & 6.0/256.0 & 4.0/256.0 & 1.0/256.0 \end{bmatrix}$$

this kernel is only applied to the lights sources, in order to achieve a simple implementation of bloom. The two textures are afterward

[1] http://www.ogre3d.org/tikiwiki/tiki-index.php?page=-Point+Light+Attenuation

[2] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/

[3] http://www.fileformat.info/format/wavefrontobj/egff.htm

[4] https://developer.android.com/reference/android/view/View.OnTouchListener.html

[5] $https://en.wikipedia.org/wiki/Linear_interpolation$

[6] $https://en.wikipedia.org/wiki/Kernel_(image_processing)$

blended together and applied to an plane covering the screen, with the default *framebuffer* bound [de Vries 2015].

The particle effects[Brothaler 2013b] used in the application are implemented with the use of a particle system containing attributes such as position, speed, spawn time and direction for every particle. A particle spawner uses this container to spawn the desired amount of particles. The particles are spawned in a sphere[7][8] around a desired point and travel in a direction with some variance with a semi-random speed. The life of a particle is defined by when the particle will be re-spawned, however using the difference between the particle start time and the current time and semi-random values from a noise texture divided by the color of the particles, the particles will disappear after some time, since the particles are additively blended with the environment. The color of the particles is determined by a custom made texture[9].

To alleviate the loading times an android *dialog* loading [10] screen is show at the start of the application, disappearing when the openGL has loaded its resources.

The goal of the game is to not collide with the asteroids, which continually spawn in front of the spaceship. In order to avoid this the player has two options, either shoot them down or avoid the asteroids. Those two aspects are calculated every frame, first the distances from the spaceship to the asteroids are calculated, if the player should be too close, the phone vibrates[11] and a life is subtracted. The life total is displayed with the use of custom android toast message[12], displaying *hearts* to illustrate the remaining life total Figure 2. Secondly the distance and the direction of the potential lasers from the asteroid are calculated. If the laser are in the vicinity of an asteroid, that asteroid is re-spawned and in its place a particle explosion is displayed.

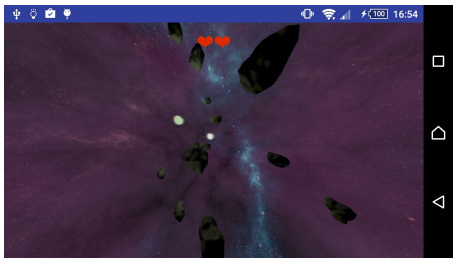

Figure 2: Toast message displayed after getting hit.

3 Results

The application loads in about thirty seconds, however this is increased substantially when there are several complex models loaded into the application. This performance decrease is most likely the result of the naive and unoptimized model loading class.

During runtime the framerate is stable, despite the use of a *framebuffer* and post-processing kernel image-transformation. The kernels which calculate the value of a pixel by acquiring values from nearby pixel, this operation results in several extra calculations per pixel but does not hamper performance.

[7] $https://en.wikipedia.org/wiki/Spherical_coordinate_system$
[8] http://www.3dgep.com/simulating-particle-effects-using-opengl/
[9] $http://sol.gfxile.net/firey/$
[10] https://developer.android.com/guide/topics/ui/dialogs.html
[11] https://developer.android.com/reference/android/os/Vibrator.html
[12] https://developer.android.com/guide/topics/ui/notifiers/toasts.html

Figure 4 depicts the post-processing effects for bloom on the lights sources. Figure 5 a sharpening kernel applied the a texture holding everything except the light sources and Figure 6 depicts the same texture but with a edge detection kernel applied. The post-processing effects are very modular and changing from one kernel to another is simply the matter of defining the matrix in the shader drawn to the overlay. The textures captured during runtime could also be applied to another objects, to archive for example a mirror effect.

The particle can be used to great effects to create different visual effects, as can be seen in the project such as fire for the engines or explosions Figure 3. It is simply a matter of changing the variables such as: the number of particles, the speed variance of each particle and the direction variance. The noise texture could also be exchanged for another texture and thereby altering the color falloff.

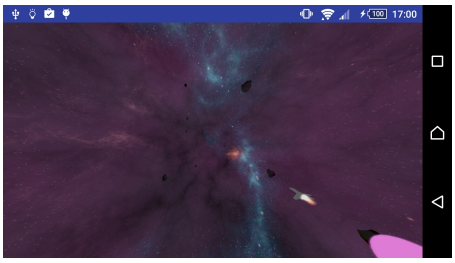

Figure 3: An explosion particle effect .

The touch controls can also be used to great effect for 3D movement, its simply the matter of defining at what depth the developer wants to calculate the intersection.

4 Discussion

An improvement that could be made for most of the objects in the application are proper use of Vertex Buffer Objects(VBO), in current implementation most objects are bound into several VBO's, instead of one utilizing an offset, this was due to the simpler implementation of using several VBO's. As of openGL 3.0 VAO can also be bound, however most of the application was first written in openGL 2.0 and later upgraded, some of the shaders are also written using the GLSL syntax used by openGL es 2.0.

I was unable to incorporate directional lighting due to limitations of my phone, when i tried my phone crashed several times due to a faulty memory read error, no matter how the implementation was done.

For more visually appealing appearance the blur post-processing effect could have used a proper implementation of Gaussian blur instead of an approximation, if the phone could have been able to handle the increased performance demands.

5 Conclusion

Implementing everything from the ground up took more time than i imagined, even simple things such as loading in a texture or creating a cubemap takes some effort. I tried to make some abstractions for object transformation and a class for the camera, but i am sure there are some better and more effective way to handle such operation than my implementation. I'm quite happy with the result even though it is not exactly everything i set out to do. I'm especially satisfied with how the post-processing effects turned out, as soon as functionality for gathering shader output to textures was implemented, the post-processing effects was easy and fun to work with.

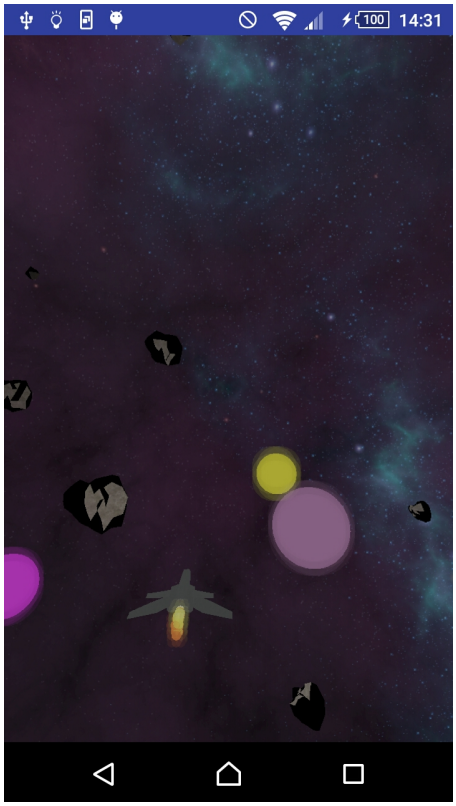

Figure 4: The blur approximation kernel applied to the light sources.

References

BROTHALER, K. 2013. OpenGL ES 2 for Android A Quick-Start Guide. The Pragmatic Programmers, LLC, 165–186.

BROTHALER, K. 2013. OpenGL ES 2 for Android A Quick-Start Guide. The Pragmatic Programmers, LLC, 191–214.

DE VRIES, J. 2015. Learn OpenGL . 240–253.

POWELL, V. Image Kernels Explained Visually. *Explained Visually*.

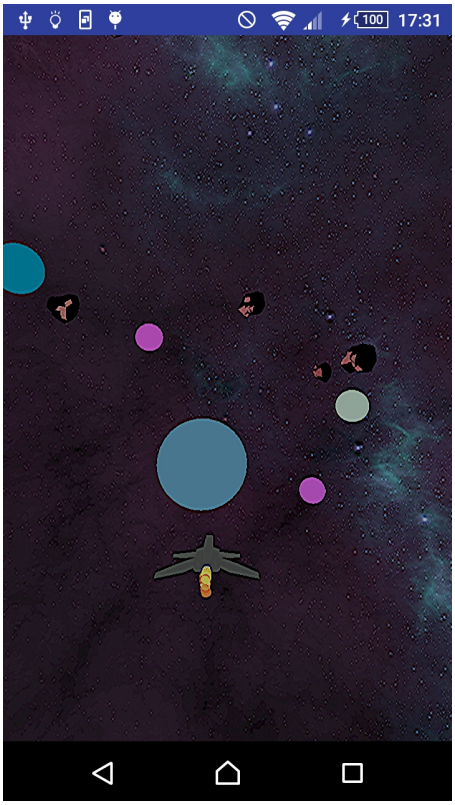

Figure 5: A sharpen kernel applied to the texture containing everything except the light sources.

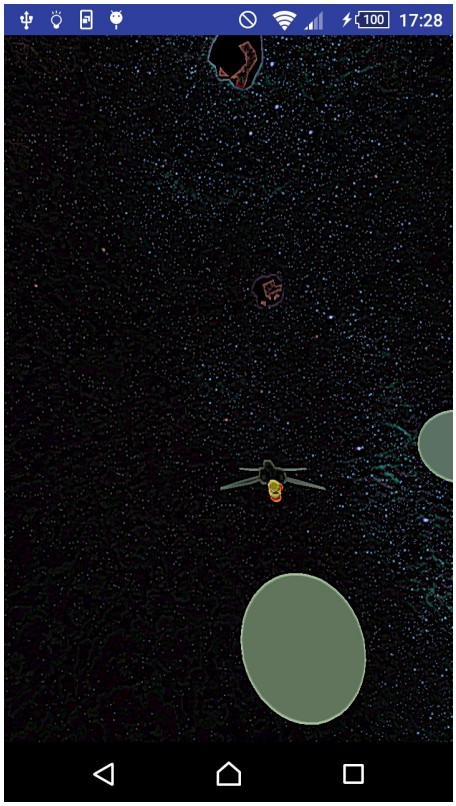

Figure 6: The sharpen kernel is exchanged for an edge detection kernel.