Project in EDAN35 High Performance Computer Graphics

Alexander Najafi * Erik Karlén†

Lund University
Sweden

Abstract

To create a 3D graphics application for a mobile device one has to face a number of challenges, especially since the hardware of a mobile device is significantly worse than a laptop or desktop computer. This project focused on implementing algorithms for OpenGL ES to create a labyrinth game for Android. The project includes algorithms for handling the accelerometer, doing collision detection and creating a fire effect using a particle system.

1 Introduction

Graphics has for a long time only been run on computers and gaming devices. In the last few years there has been an increasing interest in writing graphics applications for mobile devices. As the GPU chips in mobile devices gets better and faster, more advanced and larger graphics applications can be written. The Open GL API used for computers has its child, OpenGL ES. OpenGL ES is a graphics API designated for embedded systems such as in mobile phones. The API gets more and more advanced for every year that passes, and it is slowly going towards the standard OpenGL API.

Gaming has moved from the handheld consoles such as the Gameboy or PSP onto the phones. Creating 3D games for mobile devices is a huge industry and is growing every day. The area is very interesting and demanding in terms of optimization and the variety of devices. There are allot of challenges that has to be faced, that does not exist on desktops or consoles such as performance and battery life. On the other hand the mobile devices gives you allot of freedom to play around with many different sensors and a big high resolution touch screen. This project was chosen because of the challenges but also because of the opportunity to do something different and to make the best use of the accelerometer.

2 Algorithms and application

2.1 Mobile device

The game is a simple labyrinth game where the objective is to get the ball to the goal by steering it through a labyrinth while making sure not to fall down one of the holes. To steer the ball, the accelerometer in the phone was used. The accelerometer reads as expected, the acceleration of the mobile device in three dimensions. If the device is tilted, data for that can be retrieved using Android API calls. The data that is retrieved from the accelerometer is used to compute a directional vector of in which direction the device is tilted. The more the device is tilted, the longer the vector is. This vector is then used to set the acceleration of the ball in two dimensions, and not as you might have expected, the velocity. This gives the game a more natural feeling of actually tilting a board and making a ball move around.

Since the game is targeting a mobile device there had to be many restrictions on especially memory, CPU and GPU usage. Since the Android operating system does not give a single application much memory to play around with, large and many textures are out of the picture when developing mobile games. The labyrinth game runs on under 30 mb of memory and is only using three different textures, there is only a single light source and the shading techniques has to be kept short and optimized. The application loads all the texture when it starts and is then resuing them as it needs the them. This makes the memory usage quite stable and is only varying a little bit. See figure 1 for memory usage.

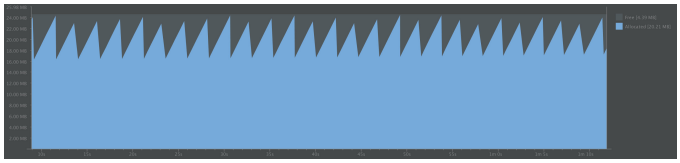

Figure 1: Memory usage when runing the application.

2.2 Particle system

The objective of the game is to get the burning ball to the goal where there is water that puts the fire out. To get the look of fire a particle system was implemented. The particles are emitted from the ball and are slowly dying out as their lifetime gets bigger.

When a particle is emitted from the source, it gets its parameters set from a random function. It gets a direction, a velocity, a position (small distance from the emitter), its age set to 0, a lifetime and some delay. All the particles in the scene are initialized with randomly generated parameters and passed to the shader as the application starts. For every frame the particles age are increased by a stepsize, if the age is greater than the lifesize, the particle is killed. To save memory, the particles that reaches its lifetime, gets its age set to 0. This way the number of particles stay constant and no memory is wasted in creating new particles all the time. The delay parameter is used to not make all the particles set of from the emitter at the same time. A particle is only emitted from the source if the delay time is passed.

Each particle in the particle system is created by the built in OpenGL feature, glpoint. This way, the GPU handles all the geometry for the particles and relieves the CPU for some heavy work. The particles positions are computed in the vertex shader. The position of each particle depends on its last position p, its velocity v, the age a, the delay time d and a static gravity vector that describes in what direction the fire should move g. The gravity vector is in the labyrinth game fairly small and has a small positive Z-value. The movement of the ball makes the look of the fire actually going out from the ball in whichever direction the ball moves. The position of the ball is calculated with

$$newPosition = p + v * (a - d) + 0.5 * g * (a - d)^2 \quad (1)$$

The position calculated is multiplied with the model view matrix and returned from the vertex shader.

As the particle grows older, the size of the particle decreases. An age factor is calculated and multiplied with a constant (the labyrinth game used 120 as a size factor) that describes the size of the particles as

$$size = (1.0 - ((a - d)/lifetime)) * 120 \quad (2)$$

*e-mail: elt12ana@student.lu.se
†e-mail: tfy12eka@student.lu.se

By using the built in features of OpenGL for particles, the built in, standard variable gl_pointsize is set to the size calculated above.

The job of the fragment shader in the particle system is to calculate each particles color. This is used using a base color mixed with a texture. The texture used is simply shown in figure 2 and is used to make the particles look faded and a bit cloudy.

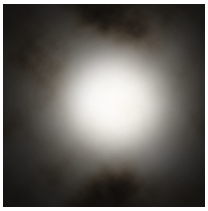

Figure 2: Texture used for the fire effect

The texture coordinates used for sampling is taken from the built in variable gl_PointCoord that contains the coordinate of a fragment within the particle. Too keep the base color in the particle, only the red channel of the sampled color from the texture is used. This, together with a calculated alpha factor are used to form the output of the particle shader.

As the particle moves further from the source and gets older, the color of the particles changes to make a more realistic fire look. The older a particle is, the more red is weight into the base color. The base color, texture color and the returned blended color for each pixel is returned from the fragment shader as

```
vec4 baseColor = vec4(vColor.x, 1.0-(vColor.y
-gDiff)*0.55, vColor.z, 1.0);

texColor = texture2D(texture, texCoords);

gl_FragColor = vec4(texColor.x, texColor.x,
texColor.x, texColor.x * alphaFactor) *
baseColour;
```

where $texColor$ is the sampled color from the texture and $gDif$ is the age of the particle.

2.3 Collision detection

To make the ball interact with the environment two things had to be done. First it had to be determined if the ball was close enough to something for it to interact with it and second it had to be decided what should happen to the ball. Determining if the ball was close enough to a hole to fall in it was easy. All it required was testing the following

$$|\mathbf{b} - \mathbf{h}| < r_h \tag{3}$$

where $\mathbf{b}$ is the center point of the ball, $\mathbf{h}$ is the center point of the hole and r_h is the hole radius. If this was the case the control of the ball was removed from the user and its position was interpolated to the middle of the hole over a few frames.

The case with the boxes was a little harder than for the holes and required quite a bit of math. For each side of the box it's checked whether the line through it intersects with the circle and where this intersection occurs. The intersection points can be described by $\mathbf{P} = \mathbf{E} + t \cdot \mathbf{d}$ where $\mathbf{E}$ is one edge point of the line segment, t is a varying variable and $\mathbf{d}$ is the vector from the first edge to the other of the segment. With a bunch of math it is possible to show that the two intersection points (assuming they exist) can be calculated as

$$t_1 = -2\mathbf{f} \cdot \mathbf{d} - \sqrt{4|\mathbf{f}|^2 - 4|\mathbf{d}|(|\mathbf{f}| - r^2))}/(2|\mathbf{d}|) \tag{4}$$

$$t_2 = -2\mathbf{f} \cdot \mathbf{d} - \sqrt{4|\mathbf{f}|^2 - 4|\mathbf{d}|(|\mathbf{f}| - r^2))}/(2|\mathbf{d}|) \tag{5}$$

where $\mathbf{f} = \mathbf{E} - \mathbf{C}$ and $\mathbf{C}$ is the center of the ball. Since the discriminants of t_1 and t_2 are negative if the ball isn't colliding with the line it's first necessary to see if they are negative. If they are then at least one of them needs to be between 0 and 1 for the ball to collide with the segment, so that is checked next. This test is then done in each frame for each side of every box. If a collision is detected with a box the ball's speed is changed and its translation in that frame is set so that it doesn't move. One test is done in the x-direction and another in the y-direction, so if the ball is colliding in the x-direction then the speed in only the x-direction is multiplied by -0.3 to make it bounce. Also, if the ball is colliding with a corner only one of t_1 and t_2 will be between 0 and 1 and both the speed in the x- and y-direction are changed.

3 Results

A few screenshots of the final game are shown in figures 3 - 4. The ball is shown in the top corner moving down with the fire effect following. A bunch of boxes are placed in a labyrinth manner with lots of holes to make the game more difficult. In the top left the goal is shown with its water texture.

The performance of the game is decent but can quickly suffer if the number of particles are increased too much. For example, on a Sony Xperia Z5 it runs well with up to 2000 particles, but with the 2 years older Samsung Galaxy S4 the number of particles has to be lowered to around 1000.

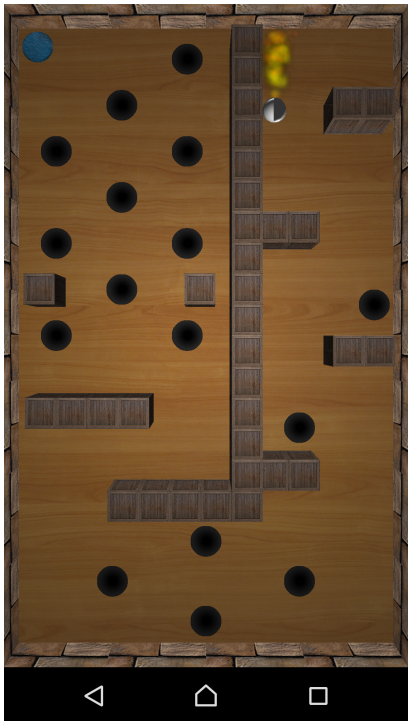

Figure 3: A screenshot of the game.

4 Discussion

While the game mostly works well, it does suffer from a few bugs. The first and most obvious from the screenshot is the fact that the ball is rendered incorrectly. Only half the ball is correctly shown with its Phong shader while the other half is the wrong colour, giving it an odd look. Much time was spent trying to fix this by, for example, rewriting the sphere class, looking at the coordinates and normals and so on. No matter what was done it never worked. However, it is suspected that the problem has to do with the transforma-

Figure 4: Another screenshot of the game.

tions of the vertices. When the ball is placed in the camera's front or rear plane it is rendered correctly.

Another prominent bug in the game, which can not be seen from the screenshots is the ball getting stuck or bouning incorrectly at the corners of boxes. This occurs because the bouncing mechanics aren't correctly implemented and need to be changed. Lots of different implementations were tried but none of them were correct. With more time this bug could probably be fixed, but unfortunately the time was too short.

5 Conclusion

The project of creating the labyrinth game was mostly successful. The game works as expected and the implemented algorithms works and looks good. The time limit of the project was a major factor of not being able to create more detailed graphics and to fix all the bugs.