Terrain Rendering on Mobile Devices using Index-buffer switching

BACHELORTHESES 2

Student Christian Granitzer, Fhs32234
Adviser DI Robert Praxmarer

Salzburg, 01 May 2013
Affirmation

Hereby I confirm that I wrote this thesis independently and on my own, as well as that I have not made use of any other resources or means than those indicated. Furthermore I confirm that I have cited all text that originates from any of those resources.

This work was not submitted or released in the same or a similar way to any other board of examiners, neither in the home country nor abroad.

......................................................... .........................................................
Date                                                                                   Signature
Abstract

First-surname: Christian Granitzer
Institution: Salzburg University of Applied Sciences
Degree program: Multimedia Technology
Title: Terrain Rendering on Mobile Devices using Index-buffer switching
Adviser: DI Robert Praxmarer

Key words: Terrain, Level-of-Detail, Rendering, Mobile, OpenGL ES 2.0, Index-buffer switching, crack elimination

Visualizing large terrains in the field of computer-graphics has always been of huge interest. With graphics hardware and CPU processing power having moved on for desktop computers, more complex data structures and scene management could be realized. However, with the emergence of Smartphones and mobile devices we are once more facing restrictions, regarding CPU and CPU to GPU bandwidth. Therefore this work is dedicated to the topic of terrain rendering on mobile platforms. It is structured in two major parts, one of a theoretical and one of a practical nature. The theoretical part of this thesis gives an overview about the topic terrain, and its representation in correlation with computer graphics. It furthermore explains common principles that overcome the problem of rendering large terrains by reducing the overall triangle count by means of Level-of-Detail techniques, and the alleviation of popping effects through blending and morphing. The sections on ROAM and Geometrical Mip-Mapping will illustrate which methods do and do not come into consideration for being used in a mobile application. The practical part outlines prerequisites for mobile application development and singularities of OpenGL ES 2.0, as well as a specification of the device that was used in conjunction with the prototype. It furthermore gives an overview about the prototypes base structure and its functionality. The main part of the practical approach aims to illustrate the rendering of a 512x512 terrain, and its Level-of-Detail management, realized by index-buffer switching. At the end a visual solution for the problem of crack occurrence is provided which works entirely in the vertex shader stage.
Acknowledgements

I’d like to thank Michael, my brain, and everyone else who supported me in writing this thesis.
# Table of Contents

1. **Introduction** 3  
   1.1. Motivation 3  
   1.2. Scientific Question 5  
   1.3. Methods 5  

2. **Terrain** 6  

3. **Level of Detail** 7  
   3.1. Discrete Geometry LODs 8  
   3.2. Blend LODs 8  
   3.3. Alpha LODs 8  
   3.4. CLODs and Geomorph LODs 9  
   3.5. LOD Selection 9  
   3.6. Difference to arbitrary 3D-Models 10  

4. **Terrain simplification** 10  
   4.1. Real-time Optimally Adapting Meshes 11  
      4.1.1. Triangle Bintree 11  
      4.1.2. Split-Queue 12  
      4.1.3. Merge-Queue 13  
   4.2. Geometrical Mip-Mapping 14  
      4.2.1. Texture Mip-Mapping 14  
      4.2.2. Basic algorithm 15  

5. **Discussion** 15  

6. **Practical Approach** 16  
   6.1. Prerequisites 16  
      6.1.1. Lifecycle callbacks 16  
      6.1.2. OpenGL ES 2.0 Programmable Pipeline 17  

1. Introduction

The following chapter gives an overview about the relevance of the topic associated with this thesis. Furthermore it will define the scientific question which outlines the field of research and introduces the methods for answering it.

1.1. Motivation

Mobile devices like smartphones and tablets gain more and more importance in the game industry. According to a study from Strategy Analytics the number of smartphone users worldwide exceeded the one-million mark in the third quarter of 2012. (www.businesswire.com, 2013). Furthermore ABIresearch estimated the number of sold tablets worldwide to be 200 million, and that this number is about to increase over the next 5 years and reach 1.2 milliards in 2018. (www.abiresearch.com, 2013). At the Games Developer Conference 2012, Tim Sweeney, CEO of Epic Games pointed out how fast tablets and smartphones graphics capabilities increase and step by step get closer to Playstation 3 and Xbox 360 level and that Epic's most profitable game wasn't the console game Gears of War but the mobile game Infinity Blade instead. (www.gamasutra.com, 2012). An advantage compared to conventional gaming devices like desktop PC, Consoles or handhelds is amongst others, the intergenerational component of mobile devices. In barely any of the first categories the acceptance level in the over-fifty-years segment of the population is as high as with smartphones and tablets. The intention of Nintendo’s Wii console was to broaden the target audience from teenagers and technophile adults to the whole family, especially older people. This, by the Wii only partially reached intention, was fully achieved by mobile devices. The general necessity of the mobile phone which evolved over the last decade, and the cross-generational acceptance of this technology, eased the step towards toleration of innovations and reforms that have arisen with the emergence of smartphones and tablets. Moreover the complexity of applications and games available on those devices up to now, is being kept at a low level, which again results in a lower inhibition level for their usage compared to games in the mid-core or hard-core section.
Figure 1: Smartphone owners that use their devices for playing games (www.thinkwithgoogle.com, 2013)

The evaluation of the two years 2011 and 2012 highlight a general upward trend in the usage of smartphones in conjunction with games. Countries with no, or only evaluation data of one year, were excluded from the graph. Due to a lot of people making use of the big variety of games offered, it is foreseeable that demands on mobile games will change in the next years. Even though there is a tremendous amount of 2D games compared to 3D games at the moment, there are various 3D titles available and there will be a lot more of them in the future, once the 2D game market is saturated. Similarly as it is with desktop PC’s, demands on in-game-graphics for mobile games get higher and higher and terrains get bigger and more detailed. The only limit is the devices hardware, limiting the complexity of scenes by a maximum number of polygons which can be rendered while maintaining a consistent frame rate. In order to decrease the number of polygons within a scene, so called Level-of-Detail techniques and mesh simplification methods are used. Another problem with mobile devices is that the current standard of the graphics API, OpenGL ES 2.0, does not implement a geometry shader stage in order to dynamically tessellate terrain on the graphics processing unit. According to the Khronos Group (www.khronos.org, 2013), which is the consortium behind the OpenGL API, the successor will neither implement a geometry shader stage and if we consider that even though the standard OpenGL ES 3.0 is specified already but that at
the moment there is not even a device implementing this standard, one can assume that it will take quite some time to tackle this restriction. The most desired solution for dynamic Level-of-Detail on mobile devices would be one with a preferably low CPU overhead and a static number of polygons that get drawn by the graphics API in every frame. Dynamic tessellation on the graphics processing unit is unconvertible due to the lack of a geometry shader stage, which de facto requires the CPU to perform this work instead. Due to the up-to-date topic and three dimensional mobile applications gaining more and more importance, this work aims to answer the following scientific question.

1.2. Scientific Question

Can dynamic CPU based Level-of-Detail improve the performance of terrain rendering with OpenGL ES 2.0 on the Mobistel Cynus T2?

1.3. Methods

This work consists of two major parts, one of a theoretical nature, where concepts for terrain rendering are introduced and discussed, and a second one that tries to put some of the principles of the first one into practice. The prototype will be documented in an accumulative manner, meaning that it will not only present the result of the end state but rather the way to the reached point. This approach asserts that ideas that didn’t work out as intended are pointed out as well as those that worked. For the practical part I will make use of the Android Development Tools, short ADT, which provides all the components to get started for developing an Android Application. Eclipse is used as Integrated Development Environment and Java as programming language.
2. Terrain

We refer to terrain as the characteristics that define a landscape and its appearance. The earth’s surface for instance can be considered as terrain, though its shape may be fictitious as well. In the field of computer graphics, terrain may even be only a certain portion of a much bigger area. Details like trees, rocks, or buildings may be omitted as well, with the outcome that the raw surface morphology already meets the definition of terrain. In its simplest form terrain remains static, where in real life it is dynamic. This can be illustrated for instance by weather conditions, where falling rain may increase the water gauge and cause terrain portions near the water to be covered, thus reducing the amount of land. Yet another example may be continental drifts, which occur through the movement of tectonic plates, resulting in a separation of parts of landmass.

Figure 2: Terrain (www.outerra.com, 2013)

The above figure shows a rendering of terrain from the outerra project. It is under development since 2008 and renders high quality terrain, offers seamless transitioning from outer space to the planet’s surface and unlimited visibility regarding view distance.

As the data, defining the geometry of the terrain has to reside somehow in memory in order to be displayed on the screen, and due to limitations of memory space, techniques known as Level of detail, or short LOD have been introduced and heavily used in the field of computer graphics.
3. Level of Detail

LOD techniques for terrain have a long history, though it is a specialized area of Level of detail. As Luebke et al (2003) states, it is for example of particular importance to flight simulators which in fact were the first systems that made use of LOD in practical sense. Especially back at the time when computer hardware’s performance was limited, it has been a necessity to optimize the number of graphics primitives, representing a scene in order to display geometry at a constant frame rate. It was common to display fast moving, or objects far away in a lower resolution than near objects as the reduced visual quality of the model often goes unnoticed.

![Multi-resolution instances of 3D object](www.opensg.org, 2013)

The figure above illustrates the same model in different resolutions. Though we can notice the difference by looking at them from the same distance, as they move farther away, we can no longer determine a visible difference. The figure below illustrates this effect.

![60.000 vs. 600 polygon model at different distances](modification, www.opensg.org, 2013)
A common problem when replacing an object with one with a lower or higher resolution is the so called popping effect, which leaves the mark of a sudden appearance or disappearance of geometric details. In order to avoid this phenomenon, blending or morphing is used to transform one model's geometry into another model's geometry, which alleviates the popping effect. Other systems that make use of level of detail techniques are for example terrain based computer games, military mission planning systems or geographic information systems, short GIS.

"In general, LOD algorithms consist of three major parts, namely, generation, selection, and switching. LOD generation is the part where different representations of a model are generated with different detail. [...] The selection mechanism chooses a level of detail model based on some criteria, such as estimated area on the screen. Finally, we need to change from one level of detail to another, and this process is termed LOD switching. (Möller T. et al., 2008). The author furthermore describes several approaches for Level-of-Detail solutions, each having specific strengths and weaknesses. The following sections give an overview about the different techniques used.

3.1. Discrete Geometry LODs

Discrete Geometry LODs use a predefined set of objects, each representing the same model, like the ones in Figure 4. The switch from one Level-of-Detail to another just happens as a certain criterion comes true, thus for one frame it may be a certain LOD and on the next frame another, which causes the above explained popping effect.

3.2. Blend LODs

Blend LODs are linear blends over a short time interval, where the current LOD is replaced with the, for the current frame evaluated, new LOD. First the current LOD is drawn opaquely to the frame buffer. The new LOD is faded in by linearly increasing the objects alpha value until it reaches full opacity. Exactly at this time it is selected as the current LOD and the old LOD is faded out. Transition intervals have to be kept short due to the fact that both LODs are fully opaque at the point when the faded-in LOD becomes the current LOD.

3.3. Alpha LODs

This method avoids popping altogether by simply using alpha blending as the object moves away. If the distance is big enough, which is the point where transparency reaches its maximum, the object doesn't have to be rendered anymore, which is also the only point where a
 performance increase can be expected. The advantage to discrete geometry LOD is the much higher continuity by completely avoiding popping. The disadvantage is that the speedup is only noticeable as the object completely disappears and that the technique relies on sorting the alpha blended objects by depth, in order to ensure correct transparency blends.

3.4. CLODs and Geomorph LODs

Another way of making smooth transitions from one LOD to another can be achieved with edge collapse methods. A collapse happens if an edge shrinks until its two endpoints meet and become one. The slightly simplified model ends up with having one vertex less than the old model which results in a smooth transition if animated over time. This process can easily be reversed by storing a set of edge collapses, that happen as the model is simplified and adding these vertices again as the model is refined. The refinement step is also called vertex split, which, as the name implies, entails creating a new line by adding a new vertex. This scheme is called continuous level of detail, short CLOD. Although it seems appealing to use this scheme according to its seamless transitions, not every model in the simplification chain looks good and it is more complicated to use with polygonal meshes than with static models. This means that if there are a bunch of the same objects in the scene each one has to determine its own specific set of triangles. Geomorphs also provide a way for smoothly transitng between two LODs. It suggests having a certain amount of static models, representing the different LOD levels and when switching to a higher or lower resolution model, interpolating the vertex positions of the old model with those of the new model. During the switching step the vertex number remains static and once the geometry deformation is done, the new model is used to represent the object. The old model is then rejected. The advantage compared to CLODs is that the individual static model can be selected beforehand, resulting in always having a good looking model. Drawbacks are the need of interpolating vertices, which brings about extensive computation and the pretence of an always changing model which may be annoying.

3.5. LOD Selection

Once the different resolution meshes are created, there is still the need of finding the appropriate version to use. There are so called benefit functions that are evaluated for the objects location and the current viewport. Based on a metric they pick the appropriate LOD, where an example may be the distance from the viewport to the object. Möller describes the selection mechanism as follows: “The most detailed LOD has a range from zero to some
user-defined value $r_1$, which means that this LOD is visible when the distance to the object is less than $r_1$. The next LOD has a range from $r_1$ to $r_2$ where $r_2 > r_1$. If the distance to the object is greater than or equal to $r_1$ and less than $r_2$, then this LOD is used, and so on.” (Möller T. et al, 2008).

3.6. Difference to arbitrary 3D-Models

Due to the fact that terrain is usually a vast portion of continuous environment, where at certain points, especially higher ones the user’s field of view reaches far into the distance, a view dependent level of detail solution is mandatory. In contrast to 3D objects that change their resolution as they come closer or move farther away, terrain kind of keeps a certain resolution as the user is more likely moving along the terrains surface.

4. Terrain simplification

Over the past decades different approaches have been introduced to tackle the problem of managing and displaying large terrains at interactive frame rates. In general the simplification process can be assigned to one of two major categories, which are top-down and bottom-up. In “Level of Detail for 3D Graphics” Luebke et al (2003) defines the two categories as follows. “In a top-down algorithm, we normally begin with two or four triangles for the entire region and then progressively add new triangles until the desired resolution is achieved. These techniques are also referred to as subdivision or refinement methods. In contrast a bottom-up algorithm begins with the highest resolution mesh and iteratively removes vertices from the triangulation until the desired level of simplification is gained. These techniques can also be referred to as decimation or simplification methods. […] Bottom-up approaches tend to be able to find the minimal number of triangles required for a given accuracy. However, they necessitate the entire model being available at the first step and therefore have higher memory and computational demands.” (Luebke et al, 2003).

![Figure 5: Top-down vs. bottom-up approach (Level of detail for 3d graphics, 2003)](image)
Furthermore the author points out that a distinction between the preparation time and runtime sense of bottom-up versus top-down has to be made. He states that bottom-up approaches are almost always used during initial hierarchy construction, but at runtime are hybrid in practice, having a top-down quadtree block framework operating in the background.

4.1. Real-time Optimally Adapting Meshes

The Real-time Optimally Adapting Meshes, short ROAM, algorithm was one of the first approaches used for obtaining an optimal triangulated mesh, using view dependent error metrics. Optimally triangulation means that the smallest set of triangles is obtained which properly defines a mesh. A binary tree consisting of an arbitrary number of triangles, in conjunction with two queues, one for splitting operations and one for merging operations, is used as the basic principle. In order to operate on high frame rates the algorithm implements several performance optimizations which will not be depicted in this work, as the focus will rather lie on the basic algorithm.

4.1.1. Triangle Bintree

Duchaineau et Al describes the triangle bintree as follows: “The root triangle, $T = (v_a, v_0, v_1)$ is defined to be a right-isosceles triangle at the coarsest level of subdivision, $l = 0$. At the next finest level $l = 1$, the children of the root are defined by splitting the root along an edge formed from its apex vertex $v_a$ to the midpoint $v_c$ of its base edge ($v_0$, $v_1$). The left child of $T$ is $T_0 = (v_c, v_a, v_0)$, while the right child of $T$ is $T_1 = (v_c, v_1, v_a)$. The rest of the
triangle bintree is defined by recursively repeating this splitting process.” (Duchaineau et al, 1997). The author furthermore defines the bintree structure to consist of neighbouring triangles, where each triangle has a base neighbour, which shares the edge along v0 and v1, a left neighbour, which shares the edge v0 and va, and a right neighbour which shares the edge v1 and va. He points out that as a matter of fact the neighbours of a triangle can either be from the same bintree level as the triangle itself, or from the next finer level for left and right neighbours and the next coarser level for the base neighbour. If the base triangle is from the same level as its base neighbour they form a so called diamond.

![Diagram of Diamond and Split Diamond](image)

**Figure 7: Diamond on the left side, split diamond on the right side (Duchaineau, 1997)**

When a diamond is split, a new vertex is introduced at its centre. The main triangle and its base neighbour is split and replaced by their children T0, T1, TB0 and TB1 as illustrated in the figure above. If the triangle has no base neighbour only the triangle itself is split. Merges can only be carried out for triangulated diamonds. In order to maintain a temporal continuity, added vertices can be morphed over a time interval, instead of immediately setting its position when splitting a triangle. The same logic can be applied for merges. Triangles cannot be split if the base neighbour is from a coarser level than the triangle itself, and therefore the base neighbour has to be split first, which may result in additional recursive splits until the desired level is reached. This procedure is called a forced split.

### 4.1.2. Split-Queue

As the triangulations are built top-down the author assigns priorities to each triangle in the triangulation chain. Every triangle in the current triangulation is added to the split-queue. While the current triangulation is inaccurate, the highest priority triangles are force-split. Afterwards the split-queue is updated by removing the triangles contained in the current triangulation. Furthermore any triangles resulting from forced recursive splits are removed from the queue and new triangles are added to the queue instead. This ensures that the
triangle priorities are updated in an accurate manner, thus a subsequent triangulation has always lower priority triangles than prior triangulations.

4.1.3. Merge-Queue

The section above, that described the bintree structure, depicted that only diamonds can be merged. Therefore a second queue, called merge-queue is introduced that contains all diamonds for the current triangulation. The diamonds priority stems from the triangle with the highest priority that is contained in the diamond. While the current triangulation is not accurate and the desired number of triangles is not achieved respectively, or the maximum split priority is higher than the minimum merge priority, either the split-queue or the merge queue is processed and updated. A merge is executed if there are too many triangles in the current triangulation or if the triangulation is accurate. Therefore the diamond with the lowest priority is merged and the queues are updated by removing all merged children from the split-queue and adding all merged parents to the split-queue instead. The merged diamond is then removed from the merge-queue and all new mergeable diamonds are added. If there are too less triangles in the current triangulation, a split is executed, where the same steps as explained in the split-queue section above, are carried out. Additionally any diamonds whose children were split are removed from the merge-queue and all new mergeable diamonds are added instead. By applying this algorithm an optimal mesh is produced for the current frame which has the same priority as if the top-down approach had been performed on the base mesh. (Duchaineau et Al, 1997).
4.2. Geometrical Mip-Mapping

As the ROAM algorithm is quite CPU extensive it may not be applicable for any mobile devices with low CPU processing power. Another solution for rendering large terrain data by decreasing the overall triangle count is Geometrical Mip-Mapping. Boer W. H. introduces in his paper an algorithm based on the principles of Texture Mip-Mapping which leads back to Williams Lance’s Pyramidal Parametrics, see Lance W., 1983.

4.2.1. Texture Mip-Mapping

The idea of Texture Mip-Mapping is to use detailed, high resolution textures for objects near to the camera and textures with lower resolution and less details for fast moving, far away objects.

![Mip-Map levels for a texture](http://game-art.co.uk/agtec/html/mipmaps.html, 2013)

Each subsequent mip-map level is half of the size of its predecessor, and therefore the size can theoretically be sampled down to a minimum of 1 pixel. However, a power of two size is recommended due to the lack of support for non-power-of-two, also referred to as NPOT, textures of some graphics libraries.
4.2.2. Basic algorithm

The author makes use of a grid of vertices with a fixed distance between each vertex. Afterwards he samples a 8-bit gray-value bitmap, which has the same dimensions as the vertex grid, from which he assigns the colour value to the z-value of the corresponding vertex in the grid. He completely builds up the vertex data structure in a pre-processing step. Furthermore the author uses a quad-tree structure to manage multiple terrain tiles within the scene. The quad-trees root terrain tile and all subsequent tiles are enclosed by a 3-D bounding box, where the parent always encloses its children’s bounding boxes. He obtains the children by scaling down the original terrain block, thus creating multiple Levels-of-Detail, which are selected based on an intersection between the camera and the tiles bounding boxes. However he does not point out how he scales down the terrain blocks. A common method would be to create vertex buffers with different resolutions and to select the appropriate one at runtime. Another method would be to share index buffers with different resolutions among all terrain tiles and simply switch the index buffer as soon as the transition criterion comes true. In this scenario the vertex buffer can remain static on the graphics card. The second method will be illustrated in the practical approach below.

5. Discussion

Based on the theoretical disquisition on different principles for terrain representation, as well as simplification and transition methods, this section aims to define the requirements for the practical prototype. As the prototype will have to run on a mobile device, we opt for a solution with minimal CPU overhead and a low CPU to GPU bandwidth usage. The prototype will implement some of the principles of the Geometrical Mip-Mapping section, for example the data representation based on a regular grid, and usage of a heightmap for defining the terrains surface. Rather than calculating the vertex heights in the pre-processing step it will outsource the sampling of the textures height-values to the GPU, by performing a Vertex-Texture-Fetch, short VTF in order to displace the individual vertices. The Level-of-Detail selection will be carried out by the terrain tiles themselves, deciding when to use which LOD, in order to avoid an additional data structure on top which manages the scene. Furthermore the prototype will implement a modified version of the skirt algorithm, which will be performed entirely in the vertex shader stage, in order to overcome the problem of crack occurrence.
6. Practical Approach

6.1. Prerequisites

The following section describes some prerequisites necessary for developing Android Applications with OpenGL ES 2.0. The descriptions are based on large parts of the official Android Development Tools documentation (developer.android.com)

6.1.1. Lifecycle callbacks

Every Android App calls a set of lifecycle callbacks during its lifetime based on actions the user takes. Starting an app results in moving step by step up in the lifecycle pyramid with the top point being the moment the app is actively being used, whereas leaving the app results in moving back down, providing the option to switch to another app and continuing at this state later on, or destroying the application. The lifecycle callbacks are as follows: onCreate, onStart, onResume, onPause, onStop, onRestart and onDestroy. Immediately after starting an app onCreate is called quickly after that onStart followed by onResume is called. This is the point where the app is actively running. OnPause is called quickly followed by onStop if the user leaves the app. If then the user decides to redisplay the app, onRestart, followed by onStart and onResume is called and the app is once again actively running and the user can interact with it.

Figure 9: Android basic lifecycle (commons.wikimedia.org, 2013)
6.1.2. OpenGL ES 2.0 Programmable Pipeline

On the official website www.khronos.org the consortium behind OpenGL the rendering pipeline of OpenGL ES 2.0 is described as follows:

“OpenGL ES 2.0 combines a version of the OpenGL Shading Language for programming vertex and fragment shaders that has been adapted for embedded platforms, together with a streamlined API from OpenGL ES 1.1 that has removed any fixed functionality that can be easily replaced by shader programs, to minimize the cost and power consumption of advanced programmable graphics subsystems.” (www.khronos.org, 2013) The following figure shows the rendering pipelines stages in detail.

![OpenGL ES 2.0 Programmable pipeline](www.khronos.org/opengles/2_X/, 2013)

6.1.3. Vertex Buffer Objects

Vertex Buffer Objects short VBO’s are used to increase render performance by avoiding the overhead of resending vertex data to the graphics card each frame. As the name implies a VBO can be used for data associated with a vertex, which is not only the position but also, its normal or texture coordinate and colour. The fundamental idea is, to let the data reside in video device memory instead of system memory, in order to render data directly. Though VBO’s can be highly efficient, there are some cases where it may be better not to use them. In the whitepaper regarding VBO’s the Standard Performance Evaluation Corporation short SPEC points out: “While vertex buffer objects offer great potential in the efficiency of providing data to the GPU, they are often highly inefficient when coupled with operations that require CPU processing. As a result, feedback and selection may not perform well when
combined with vertex buffer objects.” (www.spec.org, 2013). This basically means that as long as data stays how it is and doesn’t need to be updated, VBO’s can increase performance whereas with data that changes performance may suffer.

6.1.4. Shader

“A Shader is a program designed to run on some stage of a graphics processor. Its purpose is to execute one of the programmable stages of the rendering pipeline.” (OpenGL wiki, 2013). Those programmable stages are signalized with the orange rectangles surrounding the vertex shader stage and fragment shader stage in figure 10. Every stage defines a set of inputs and outputs and has some built-in variables where the output from one stage may be used as input for the subsequent stage. OpenGL ES 2.0 uses Shaders written in the OpenGL Shading Language, short GLSL which are based on a C-like syntax. Each shader defines a main entry point after which certain operations may be executed.

6.1.5. Vertex Shader

A vertex shader is, as the name implies, executed once per vertex. This vertex is composed of a series of vertex attributes (normal, colour, position, texture coordinate). During the vertex shader stage this vertex is processed and assigned to the built-in variable gl_Position at the end. The mapping for vertex shaders is 1:1, meaning for every vertex entering the vertex shader stage one vertex has to leave the stage through the vertex output stream. The sample below shows a minimal vertex shader that simply passes its input position to the next shader stage.

```c
//vertex shader
attribute vec4 a_VertexPosition;

void main(void)
{
   gl_Position = a_VertexPosition;
}
```

6.1.6. Fragment Shader

A fragment shader is executed once per pixel after the primitive passed the rasterizer stage. Each fragment has a certain position on the screen or framebuffer. Furthermore it has a set of per vertex output attributes, which were passed in from the previous shader stage and have been interpolated over the surface of the primitive. The most important
built-in output variable gl_FragColor defines the final colour of the processed fragment. This shader stage is often used for so called filters. Filters operate either on a single pixel or on a pixel area, which require a second fragment shader pass with the last frames output used as texture for the current frame. The filter then operates on this texture to calculate a new color for the current shader pass. The following sample shows a minimal fragment shader that outputs a white colour value for each fragment processed.

```cpp
//fragment shader
void main(void)
{
    gl_FragColor = vec4(1.0);
}
```

6.2. Device

![Smartphone](http://cdn.pollin.de)

**Figure 11:** Smartphone used for prototype (http://cdn.pollin.de, 2013)

The device used is a Mobistel Cynus T2 with Android 4.1.1 Jelly Bean.

- **Processor:** Dual Core 2 x 1GHz
- **Ram:** 512 MB
- **Display:** 5 inch, Resolution 854 x 480
- **GPU:** PowerVR SGX531
6.3. Base structure of the prototype

The following section introduces the base structure of the prototype. No frameworks were used. The prototype is named YGGDRASIL, this is only pointed out in order to avoid confusion, as the name might appear on some screenshots taken from the mobile device or the desktop computer.

![Resource manager UML diagram](image)

Figure 12: Resource manager UML diagram (own survey, 2013)

6.3.1. Resource Manager

The resource manager is implemented as a Singleton and serves as interface to obtain relevant information about the current app-state. It holds a reference to the app's context, which provides global information about the application, e.g. resources, folder structures and assets like shader, models and textures. In addition the resource manager offers access to instances of a Timer, the main Camera, the input system and a Logger. This was aimed for this solution in order to avoid endless reference pass-through in constructors and method calls, instead the resource manager is accessible from everywhere through a static declared function-call named getInstance(), which returns the reference to the resource...
manager object if an instance is already available, else the resource manager gets instantiated beforehand.

6.3.2. Camera

The camera provides access to its view matrix and projection matrix as well as methods for setting and updating them which basically call the OpenGL ES 2.0 functions Matrix.frustumM and Matrix.setLookAtM. Furthermore the camera has a position, an up-vector and a target position to look at. The figure below illustrates the setup.

![Camera setup in scene (own survey, 2013)](image)

Matrix.setLookAtM takes the view matrix, which is defined as a float array consisting of 16 elements, an integer offset at which position in the matrix the result matrix will be stored, as well as the position, the target point and the up-vector of the camera. The result matrix will be calculated from the input and holds the information illustrated in the figure below.
As we have to take track of our position in world space, the camera class provides methods for obtaining the current position of the camera. Conveniently we can obtain the position easily by inverting the view matrix. After this the position can be obtained from the last column of the inverted matrix like illustrated below.

```
public Vec3f getCameraPositionWorld()
{
    Matrix.invertM(m_InvertedView, 0, m_ViewMatrix, 0);
    Vec3f camPos = new Vec3f(m_InvertedView[12], m_InvertedView[13], m_InvertedView[14]);
    return camPos;
}
```

The same applies to the forward vector, which can be obtained from the last but one column of the inverted view matrix by inverting the signs of each component. This fact makes movement through the scene pretty easy, as everything we have to do is, translate the view matrix based on the users input. The navigation through the scene is up to now limited to following actions.

- Pinch-to-zoom

Moves the camera along its forward vector, either in positive or in negative direction. If the distance between the fingers increases, which corresponds to a zoom-in gesture the camera is moved forward, if the distance between the two fingers decrease the camera is moved backward.
• Swipe

Controls the rotation of the camera around its Y-axis. Swiping from left to right rotate the camera to the left, swiping in the opposite direction rotates it to the right. Swiping up and down, against what might be expected, does not affect the rotation around its X-Axis, it actually just moves the target point down or up. This way we don’t have to recalculate the up-vector and it can therefore always point in direction of the positive Y-Axis. This might be unusual but is sufficient for our implementation as the forward vector will still be influenced by changing the target point, and therefore allow position changes in Y-coordinates. The following figure illustrates the movement based on some screenshots.

![Figure 15: Gesture based camera movement (own survey, 2013)](image)

The method `update(float dx, float dy, float zoom)` handles rotation and position changes. The arguments to this method are the touch input of the user. Specifically the update function gets the touch position changes in vertical and horizontal direction as well as the changes of the zoom factor. The exact calculation will be explained in the section describ-
ing the input system. Based on these deltas the update method calls a pitch(float dy) and yaw(float dx) method which recalculates the target point like illustrated below.

```java
public void Yaw(float dx) {
    yawAngle += dx;
    m_Position = getCameraPositionWorld();
    float x = (float)Math.cos((double)yawAngle);
    float z = (float)Math.sin((double)yawAngle);

    //Simply add the cameraposition to the computed targetpoint if cam is not located
    //at the center of the scene.
    m_Target = new Vec3f(m_Position.getX() + x,
                         m_Target.getY(),
                         m_Position.getZ() + z);
    setCamera();
}
```

Yaw and Pitch follow a simple rule, that is, recalculate the new target point by adding a new direction vector based on the current angle to the current camera position. As yawing the camera does not influence the target position in Y it can be left as it is at the moment. Pitch works in a similar way but instead of recalculating the x and z value it recalculates the y and z value of the new target point, leaving the X position as it is. The following picture should clarify this.

![Figure 16: Yaw and Pitch (own survey, 2013)](image_url)

As the figure above shows, the pitch angle has been limited to a scope of 110 degrees.
### 6.3.3. Logger

The logger class can be used to write debug and status messages to a text file, which is stored on the external SD card of the mobile phone, given that an SD card is available. The following figure shows how the output could look like.

![YGGDRASIL.txt](image)

```
2013-03-28 15:01:40  Created IndexBuffer Level 1
2013-03-28 15:01:40  Number of Indices: 162360
2013-03-28 15:01:40  Created IndexBuffer Level 2
2013-03-28 15:01:40  Number of Indices: 40344
2013-03-28 15:01:40  Created IndexBuffer Level 3
2013-03-28 15:01:40  Number of Indices: 10086
2013-03-28 15:01:40  Created Terrain
2013-03-28 15:01:41  ===>onCreate is called
2013-03-28 15:01:41  ===>onStart is called
2013-03-28 15:01:41  ===>onResume is called
2013-03-28 15:01:41  Successfully loaded Shader displacement_mapping.vert
2013-03-28 15:01:41  Successfully loaded Shader displacement_mapping.frag
2013-03-28 15:01:41  Successfully compiled Shader displacement_mapping.vert
2013-03-28 15:01:41  Successfully compiled Shader displacement_mapping.frag
2013-03-28 15:01:41  Successfully linked Shaders
2013-03-28 15:01:41  Successfully loaded Texture eroded.jpg
2013-03-28 15:01:42  ===>onPause is called
2013-03-28 15:01:42  ===>onStop is called
```

*Figure 17: Log file created by Logger Class (own survey, 2013)*

Besides checking for availability of a SD card and checking permissions to write to it, the Logger class provides a method called `message` which takes a string as argument. As there can be multiple instances of loggers in multiple threads, writing to different files or the same file, we have to take race conditions into account. Additionally to the string that will be written to the file, a time stamp is put in front of the message. The function looks as follows.
public synchronized void message(String content) {
    // only write something if we have the permission and
    // external storage is available
    if (m_ExternalStorageAvailable && m_ExternalStorageWriteable) {
        String dateTimeStamp = m_timer.getDateTimeStamp();
        m_OutputWriter.println(dateTimeStamp + " " + content);
        m_OutputWriter.flush();
    }
}

The time stamp is obtained from the timer class.
6.3.4. Input

The input class is used to handle user input, up to now only swipe and pinch-to-zoom gestures are handled. The main logic of the input class lies within the onTouchEvent(MotionEvent e) method. The method takes a motion event as argument, which is forwarded from another class in MainActivity.

```java
public boolean onTouchEvent(MotionEvent e) {
    float x = e.getX();
    float y = e.getY();

    switch(e.getAction() & MotionEvent.ACTION_MASK) {
        case MotionEvent.ACTION_DOWN :
            swipeIsEnabled = true;
            break;
        case MotionEvent.ACTION_POINTER_DOWN :
            distance = distBetweenFingers(e);
            swipeIsEnabled = false;
            break;
        case MotionEvent.ACTION_MOVE :
            if(e.getPointerCount() == 1 && swipeIsEnabled) {
                float dx = x - m_PreviousX;
                float dy = y - m_PreviousY;
                setDx(dx);
                setDy(dy);
                m_PreviousX = x;
                m_PreviousY = y;
            }
            else if(e.getPointerCount() == 2) {
                float newDist = distBetweenFingers(e);
                float delta = distance / newDist;
                distance = newDist;
                setZoom(1.0f - delta);
            }
            break;
    }
    return true;
}
```

At the beginning we start by obtaining the X- and Y-coordinates for the first pointer index. This is used for calculating the distance the pointer moved, for example for a simple swipe gesture. Next we have to check the actual event that happened. ACTION_DOWN will be...
triggered always, even if the user intends to place two fingers at the same time on the display, the chance of one finger touching the display earlier is very high. This case is meant for detecting a touch event for one finger, so we set swipeEnabled to true. If in the next moments a second finger touches the display, case ACTION_POINTER_DOWN will be triggered, which is meant for detecting multiple touches and swiping will be disabled as a consequence. The last case will be triggered either when moving one or multiple finger over the display. Hence we have to distinguish between two possible options, which is the reason for the check above. The calculation for swiping is quite straightforward as it simply subtracts the X-coordinate obtained at the beginning from the current X-coordinate and assigns it to the respective member variable. The same procedure takes effect for the Y-coordinate. In order to calculate the distance change between two fingers we simply obtain the distance at the moment the second finger touched the display and store its value. In the ACTION_MOVE case we calculate the new distance and divide old by new distance to obtain a value bigger than 1.0 if the distance between the two fingers decreased, or less than 1.0 if the distance got bigger. The result is set as new zoom-factor.

Figure 18: Calculation of distance between fingers (commons.wikimedia.org, modified, 2013)
6.3.5. Main Activity

Serves as the main entry point to the app once launched by the user. According to the activity lifecycle it has to override a subset of the lifecycle callback methods, at least those relevant for the app to work. For more information on lifecycle callbacks see the respective section from prerequisites. The main activity assigns the app’s context to the resource manager as discussed in resource manager section above. Furthermore it sets up the logger.
and instantiates a CustomGLSurfaceView object. The CustomGLSurfaceView extends GLSurfaceView and instantiates and registers the renderer. In order to handle touch input it overrides onTouchEvent(MotionEvent e) from the base class. As described in Input.java we just forward the event to the input system and let it handle the logic.

6.3.6. CGRenderer

This class extends GLSurfaceView.Renderer and has again a set of callback functions to override which are onSurfaceCreated, onSurfaceChanged and onDrawFrame. The renderer runs in a separate thread and is decoupled from the user interface thread.

- onSurfaceCreated(GL10 gl, EGLConfig config)

onSurfaceCreated gets called when the renderer thread starts and whenever the EGL context is lost, which typically happens if the device awakes from sleep mode. In this section the camera is set up shaders are loaded, compiled and linked and textures are loaded as well.

- onSurfaceChanged(GL10 gl, int width, int height)

onSurfaceChanged is called after onSurfaceCreated finished, and gets called every time the surface size changes. A typical change of surface size happens when the orientation of the device changes, therefore we can update the projection matrix in this method.

- onDrawFrame(GL10 gl)

This method is responsible for drawing the current frame. The logic is simple, clear the screen and depth buffer, update the camera, and draw models.
### 6.3.7. Shader

An object of the shader class has to be instantiated by every model which wants to be drawn. Among other things the class implements logic to load vertex shader and fragment shader located inside the assets/shaders folder.

```java
protected String loadShaderFromFile(String name) {
    String shadercode = null;

    InputStream iS = m_AssetManager.open("shaders/" + name);
    InputStreamReader iR = new InputStreamReader(iS);
    BufferedReader bR = new BufferedReader(iR);

    StringBuilder stringBuilder = new StringBuilder();
    String nextLine;
    while ((nextLine = bR.readLine()) != null) {
        stringBuilder.append(nextLine);
    }

    String msg = "Successfully loaded Shader " + name;
    m_ResourceManager.getLogger().message(msg);

    bR.close();
    shadercode = stringBuilder.toString();

    if (shadercode.length() == 0) {
        String msg = "Failed to load Shader " + name;
        m_ResourceManager.getLogger().message(msg);
    }

    return shadercode;
}
```

Loading shaders can happen already at a time before the GLSurface is created, because it doesn’t need any OpenGL ES 2.0 specific API calls. However the method compileShader() does. It takes the type of the shader represented as integer, the name of the shader, and the respective shader source code as arguments.
private int compileShader(final int shaderType,
    final String shaderName,
    final String shaderSource)
{
    int shaderHandle = GLES20.glCreateShader(shaderType);
    if (shaderHandle != 0)
    {
        GLES20.glShaderSource(shaderHandle, shaderSource);
        GLES20.glCompileShader(shaderHandle);
        final int[] compileStatus = new int[1];
        GLES20.glGetShaderiv(shaderHandle,
            GLES20.GL_COMPILE_STATUS, compileStatus, 0);
        if (compileStatus[0] == 0)
        {
            String msg = "Syntax Error in Shadercode "
                + shaderName
                + ": " + shaderSource;
            mResourceManager.getLogger().message(msg);
            GLES20.glDeleteShader(shaderHandle);
            shaderHandle = 0;
        } else 
        {
            String msg = "Successfully compiled Shader "
                + shaderName;
            mResourceManager.getLogger().message(msg);
        }
    } else 
    {
        String msg = "OpenGL ES module is not available yet";
        mResourceManager.getLogger().message(msg);
    }
    return shaderHandle;
}

In the case that this method somehow gets called before the GLSurfaceView is created, for instance in the constructor of the renderer, this will cause the app to crash due to the first OpenGL ES API call in the line

    int shaderHandle = GLES20.glCreateShader(shaderType);

which tries to call the not initialized OpenGL module.
The final step involves linking the programs, through `createAndLink()`. `GLES20.glCreateProgram()` returns a positive integer index as handle to a successfully created empty program. We then can attach shader objects with `GLES20.glAttachShader()`. `GLES20.glBindAttribLocation` lets us assign a vertex attribute to a specific index. After successfully linking the program with `GLES20.glLinkProgram()` these indexes remain fix and can be queried later on through calls to `GLES20.glGetAttribLocation()`. At last we verify that linking the program was successful by calling `GLES20.glGetProgramiv()` and if not delete the created program handle with `GLES20.glDeleteProgram()`.

The `compileAndLinkShader()` method combines the `loadShader`, `compileShader`, and `createAndLink` method calls that were explained above. First the shader code gets loaded from the files, in the second step we compile these shaders which will succeed if we have no syntax errors in our shader code. In the next step we create parameters which will be bound to specific indices within the `createAndLink()` method. If everything went well this method returns the program handle and assigns it to the shader handle.

The last method that the shader class provides is used to load textures. `GLES20.glGenTextures` is called for retrieving an unique texture name, the texture does not have a dimensionality yet, but rather assume to have the dimensionality of the texture it is first bound with `GLES20.glBindTexture`. Next the magnifying and minifying parameters for sampling textures are set with `GLES20.glTexParameter()`. Finally `GLUtils.texImage2D()` defines the texture and determines the internal format and type automatically.
6.4. Creating Terrain

As the terrain and how it is represented is not the focus of this work, but rather the efficient rendering itself, we go for a very simple form of terrain. The terrain will be created from a height map. Due to its nature a height map terrain cannot contain overhangs or holes, which means it is more or less a landscape with valleys and hills. By looking at this landscape from up in the air we can interpret the height maps dimensions as the landscapes length and width. The following figure should illustrate this.

![Figure 20: Terrain Height map and plane vertex buffer (own survey, 2013)](image)

The basic idea now is, to build a vertex grid which defines the geometry of this landscape in the form of a plane. The height values, at the time of creating this plane, will be zero for each vertex in the vertex grid. In the vertex shader stage we will then assign the z-value, by means of vertex texture fetching (VTF), for each vertex grid point according to the respective point on the height map. In this specific case we will interpret a white pixel as a high point, and a black pixel as a deep point in three-dimensional space. In order to showcase the setup of the terrain we expand our basic structure and create a class named Terrain Mesh that extends the Model mesh class and simply defines its own vertex buffer. If we look at the figures above, we want to define a new terrain by letting it know which length and width it has, and let the class constructor do the rest of the work.
6.4.1. Building the vertex buffer

Setting up the plane vertex buffer is as simple as looping over length and width and pushing each vertex in our buffer.

```cpp
for( int y = 0; y <= length; y++)
{
    for( int x = 0; x <= width; x++)
    {
        int idx = (((width * 3) * y) + x * 3);
        if(y > 0)
            idx += y * 3;
        //since every vertex consists of 3 components
        vertexBuffer[idx] = -width/2 + (x * 1.0f);
        vertexBuffer[idx + 1] = -length/2 + (y * 1.0f);
        vertexBuffer[idx + 2] = 0.0f;
    }
}
```

By defining our dimensions of the terrain we actually refer to quads rather than vertex points, therefore we set the condition for the loop to end, to be equal width and height. We build our vertex buffer from bottom up having 0,0 as the lower left and width+1, height+1 as the upper right vertex point. In order to have the origin set to the middle of the plane, we subtract half of the width and height for the corresponding components. Next we have to set up the texture buffer.

6.4.2. Building the texture buffer

```cpp
for( int y = 0; y <= length; y++)
{
    for( int x = 0; x <= width; x++)
    {
        int idx = (((width * 2) * y) + x * 2);
        if(y > 0)
            idx += 2 * y;
        //since every texture coordinate has 2 components
        texbuffer[idx] = x * (1.0f/width);
        texbuffer[idx + 1] = 1.0f - (y * (1.0f/length));
    }
}
```

Again we loop over width and height to populate our texture coordinate buffer. This time we define our coordinate system to be 0,1 for the lower left and 1,0 for the upper right vertex point. The reason for this is, loading a texture with OpenGL ES 2.0 will read the pixel values
from bottom up, thus causing the texture to be flipped if we don’t take this into account. All that’s left to do is writing the shaders that perform the Vertex-Texture-Fetch.

**6.4.3. Vertex Texture Fetching**

```glsl
uniform mat4 u_MVMMatrix;
uniform mat4 u_MVPMatrix;
uniform vec3 u_LightPos;
uniform sampler2D u_MainTex;

attribute vec4 a_Position;
attribute vec3 a_Normal;
attribute vec2 a_Texcoord;

varying vec2 v_TexcoordFrag;
const float stretchfactor = 5.0;

void main()
{
    vec4 color;
    vec4 newVertexPos;

    color = texture2D(u_MainTex, a_Texcoord);

    newVertexPos = vec4(0.0, 0.0, color.r * stretchfactor, 0.0) + a_Position;
    gl_Position = u_MVPMatrix * newVertexPos;
    gl_PointSize = 2.0;
    v_TexcoordFrag = a_Texcoord;
}
```

The vertex shader samples the texture colour values into a variable named `color`. Afterwards the original vertex position is added to a new point that simply defines the z value based on the red channel of the sampled pixel. With this method we could also use an arbitrarily coloured image as input. The fragment shader just calculates the final fragment colour by sampling the texture. The figure below shows the result up to now.
As we normally do not intend to simply draw points but triangles instead, we need to change the draw mode to GL_TRIANGLES instead of GL_POINTS. We have furthermore to either change the vertex buffer or introduce an index buffer in order to correctly render our terrain. The figure below illustrates the problem.

![Figure 22: Vertex buffer (own survey, 2013)](image)

As we can see our vertices reside in the given order in memory. If we intend to draw this data as triangles, OpenGL would assume that three consecutive points form a triangle. By looking at the figure we can see the corresponding points would be 0, 1, 2 instead of 0, 1, 4. We can change this by redefining our vertex and texture coordinates layout. The figure below illustrates that the changes brought about the expected outcome.
The drawback of this implementation is the memory consumption overhead, that results from pushing the same vertices more than once into our vertex buffer. In order to avoid this we need to introduce an index buffer.

### 6.4.4. Building the index buffer

The index buffer defines the order of the vertices which define a primitive, in our case a triangle.

![Index buffer diagram](image)

Figure 24: Index buffer (own survey, 2013)

By looking again at the figure that illustrates our vertex data layout, we can define a triangle also, by telling OpenGL which three points compose a triangle. Instead of duplicating vertices we create a new buffer consisting of indices that define our primitives. For the case above our index buffer would hold the values 0, 4, 1, 4, 1, 5, 1, 2, 5 and so on.
for(int y = 0; y < length; y++)
{
    for(int x = 0; x < width; x++)
    {
        int idx = ((width * 6) * y) + x * 6;
        int start = ((width * y) + x);
        if(y > 0)
            start += y * 1;

        indexBufferLV1[idx] = (short) start;
        indexBufferLV1[idx + 1] = (short) (start + 1);
        indexBufferLV1[idx + 2] = (short) (start + width + 1);
        indexBufferLV1[idx + 3] = (short) (start + width + 1);
        indexBufferLV1[idx + 4] = (short) (start + 1);
        indexBufferLV1[idx + 5] = (short) (start + width + 2);
    }
}

In order to draw indexed data we need to call GLES20.glDrawElements() instead of GLES20.glDrawArrays() and provide the index buffer and number of indices as an attribute for the draw call. The result is the same as before, only this time we render solid triangles.

Figure 25: Solid indexed terrain 100x100 (own survey, 2013)
At this point I wanted to measure the performance differences between rendering indexed and non-indexed data. Table 1 contains the results.

<table>
<thead>
<tr>
<th>dimension</th>
<th>Indexed (fps)</th>
<th>Non-indexed (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x32</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>64x64</td>
<td>60</td>
<td>42</td>
</tr>
<tr>
<td>96x96</td>
<td>31</td>
<td>34</td>
</tr>
<tr>
<td>128x128</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>256x256</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1: Indexed vs. non-indexed data rendering performance (own survey, 2013)

The table above shows the frame rates for rendering indexed and non-indexed data at different terrain sizes. While the non-indexed results seem comprehensible the indexed results bear a strange outcome. What is interesting is the significant frame rate drop in the indexed section at the transition from 64x64 to 96x96. I sampled some sizes in between and found an interesting result.

<table>
<thead>
<tr>
<th>dimension</th>
<th>Indexed (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>60</td>
</tr>
<tr>
<td>80x80</td>
<td>58</td>
</tr>
<tr>
<td>88x88</td>
<td>55</td>
</tr>
<tr>
<td>92x92</td>
<td>53</td>
</tr>
<tr>
<td>93x93</td>
<td>32</td>
</tr>
<tr>
<td>94x94</td>
<td>31</td>
</tr>
<tr>
<td>96x96</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 2: In between sampling (own survey, 2013)

The drop occurs exactly when transitioning from 92x92 to 93x93. Nearly 20 frames per second frame rate loss for 372 polygons more. The next thing I was interested in was if my frame rate would be significantly higher if I would render four parts of terrain and therefore stay at 64x64 which would correspond to a 128x128 terrain.
The result is, that compared to drawing one indexed vertex buffer with 128x128 at a frame rate of 22, four indexed vertex buffers with 64x64 terrain portions increase the frame rate by nearly 100 per cent. This phenomenon is most likely caused by level 2 and level 3 cache misses of OpenGL ES. I found that for our scenario a 64x64 terrain patch seems to perfectly fit the caching scheme of OpenGL ES 2.0 and will therefore be used as the default terrain patch size in our prototype.

6.5. Implementing level of detail

A beneficial side effect of relying on an index buffer is that we can quite simple create different levels of detail on the fly when creating the full resolution index buffer.
As the above figure illustrates, when creating a lower resolution terrain mesh, we can achieve this by assigning only the white vertices to our index buffers. So instead of the former order 1, 2, 17, 17, 2, 18 we now assign 1, 3, 33, 33, 3, 35. We sample every second vertex in x direction and skip every second row in y. It is even easier to look up the vertices from the previous index buffer.

```c
int lv1idx = 0;
int lv2idx = 0;
for(int y = 0; y < length; y+=2)
{
    for(int x = 0; x < width; x++)
    {
        if(x%2 == 0)
        {
            lv1idx = x * 6 + (y * width * 6);
            indexBufferLV2[lv2idx] = indexBufferLV1[lv1idx];
            lv2idx++;
            indexBufferLV2[lv2idx] = indexBufferLV1[lv1idx + 1*6 + 1];
            lv2idx++;
            indexBufferLV2[lv2idx] = indexBufferLV1[lv1idx + (width*6) + 2];
            lv2idx++;
            indexBufferLV2[lv2idx] = indexBufferLV2[lv2idx - 1];
            lv2idx++;
            indexBufferLV2[lv2idx] = indexBufferLV2[lv2idx - 2];
            lv2idx++;
            indexBufferLV2[lv2idx] = indexBufferLV1[lv1idx + (width*6) + 11];
            lv2idx++;
        }
    }
}
```

With this method we don’t have to change the vertex buffer at any time. It resides in memory as is and only the index buffer has to be created once and can be switched at rendering time as the camera moves away from the terrain. It is easy to create the next lower level index buffer by applying the same logic to the preceding index buffer level. The same concept can be used for multiple levels of detail, as illustrated by the figure below.
6.6. Index buffer switching

As our terrain will be rendered by stringing together multiple tile portions with the same dimension, we only need to calculate the different index buffer levels once, and share them amongst all tiles. This reduces the pre-processing costs when using a lot of tiles, because all tiles rely on the same index buffers. As illustrated above the preferred terrain tile size is 64x64, which means that this size will be used for the full resolution terrain portion. All subsequent levels of detail will therefore be half in width and height from the preceding tile. Another thing that all tiles have in common is the vertex buffer, so we also create subsequent tiles by passing an already created vertex buffer to the constructor of our terrain mesh class and simply let it be assigned to our mesh’s vertex buffer. If we assume that our terrain is always composed of the same number of tiles, we could also calculate the texture coordinates buffer offline, but for the case of testing different terrain sizes, I opted for dynamically calculating them. This is due to the fact that our terrain portions will all share the same height map and might be placed on an arbitrary position. Yet another possibility would be to shift the calculation of the texture coordinates completely to the vertex shader by declaring uniform variables that define the tile size and location of the tile on the height map. The figure below shows the different resolution meshes for a 64x64 terrain. We created five levels of detail, 64x64, 32x32, 16x16, 8x8 and 4x4, which are switched as the camera moves away from the terrain tile.
When drawing a terrain composed of multiple terrain tiles, we somehow have to decide when to do the transition from one Level-of-Detail to another. An easy method is to make the transition dependent on the distance from the centre of the terrain tile to the camera. I decided to use a radius based criterion, which can be thought of as having multiple bounding spheres that determine the level of detail to be drawn. If the camera enters a certain bounding sphere of a terrain tile, the switch to the next level is carried out. The figure below should clarify this.
Figure 30: Radius based Level of detail for one Terrain Tile (own survey, 2013)

Our update function has to determine which level of detail to select. First we check if our cameras position has changed since the last frame, if not, obviously we don't have to do anything as everything remains as it is. If the camera has moved, we calculate the distance from the camera to the terrain and select the appropriate level of detail.

Vec3f distance =

    new Vec3f(m_TilePosition.getX() - currentCamPos.getX(),
              m_TilePosition.getY() - currentCamPos.getY(),
              m_TilePosition.getZ() - currentCamPos.getZ());

int LOD = (int)(distance.Magnitude() / m_TileRadius);

switch(LOD)
{
    case 0 :  switchIndexBuffer(m_IBufferList.get(0), m_IdxCnts[0]);
              break;
    case 1 :  switchIndexBuffer(m_IBufferList.get(1), m_IdxCnts[1]);
              break;
    case 2 :  switchIndexBuffer(m_IBufferList.get(2), m_IdxCnts[2]);
              break;
    case 3 :  switchIndexBuffer(m_IBufferList.get(3), m_IdxCnts[3]);
              break;
    default : switchIndexBuffer(m_IBufferList.get(4), m_IdxCnts[4]);
              break;
}
The code snippet above illustrates that the distance is calculated from the centre of the terrain tile to the cameras position. Afterwards the appropriate Level-of-Detail is selected by using the calculation of how often the terrain tile radius fits into the distance, which serves as our benefit function. The switchIndexBuffer() function checks if the buffer to be assigned differs from the currently used index buffer. If so, the new index buffer is set to be the current index buffer. The draw method takes then care of rendering the corresponding data.

Figure 31: Terrain 512x512 level of detail (own survey, 2013)

6.7. Visibility checks

At the moment we are drawing a lot of redundant tiles, in particular the tiles that lie outside the view frustum. We can determine if a tile is inside the cameras view by calculating the dot product of the camera’s reverted forward vector and the vector pointing from the terrain tile to the camera. The result is a scalar specifying the angle between the two vectors.

```
Vec3f forward = m_ResourceManager.getCamera().getForwardVector();
    forward.Normalize();
    forward.Revert();

Vec3f terrainToCam = new Vec3f(
    currentCamPos.getX() - m_TilePosition.getX(),
    currentCamPos.getY() - m_TilePosition.getY(),
    currentCamPos.getZ() - m_TilePosition.getZ());

//patches located around the camera are always visible
```
if(terrainToCam.Magnitude() < 2 * m_Radius)
    return true;

terrainToCam.Normalize();

float facing = Vec3f.Dot(terrainToCam, forward);
if(facing > 0.0f)
    return true;
else
    return false;

The direct comparison shows a slight performance gain. The visibility check is performed once in the update function. If the terrain portion is not visible the update function terminates and the draw function won't be executed.

Figure 32: Terrain 512x512 with visibility check (own survey, 2013)
6.8. Crack elimination

A common problem which arises with Level-of-Detail techniques, are so called cracks. Cracks appear at the border of tiles where a higher resolution mesh meets a lower resolution mesh. There are different techniques that tackle the problem of cracks, most of them directly change the vertex structure, which in our case is not applicable because we don’t want to change our vertex buffer, as it is simply too cost intensive to be calculated on the CPU. We look for a solution that solves the problem on the fly, preferably in the vertex shader as we have to do the Vertex-Texture-Fetch anyway. A visual solution would be placing a skirt around the borders of each tile.

![Figure 33: Cracks at borders of tiles (own survey, 2013)](image1)

![Figure 34: Visual crack elimination by placing skirts around each tile](image2)
We have two possibilities for achieving the desired effect. We could add extra geometry, by defining triangle strips for the tiles borders and keeping track of which LOD is currently active, thus also switching the skirts Level-of-Detail. The drawbacks that arise with this method are an extra geometry overhead and the additional draw calls necessary to render the skirts. The other possibility is to perform the skirting in the vertex shader by repositioning the border vertices of each tile. Therefore we determine in our vertex shader if the current vertex is about to be a border vertex. If so, we tear the vertex’s z value into negative direction. Another step has to be made in order to achieve a correctly appearing result, which is, the last but one vertex has to take the position of the repositioned border vertex. The figure below should illustrate why this is necessary.

Due to the last but one vertex taking the position of the prior border vertex, we artificially stretch our geometry. This results in the error seen in step three. In our current scenario we have one last problem to overcome in order to get the right representation of our terrain tiles. Up to now every vertex in our grid includes important height information, namely the texture coordinate calculated in the pre-processing steps. For our vertex shader skirting algorithm we opt for a vertex layout that maps all the important information, contained in the current 64x64 grid, onto an area of 63x63, thus leaving the border vertices with no information about the terrains representation. In other words, we want to have redundant or no information in our border vertices, as they need to be deformed in the vertex shader. With these steps our crack elimination is complete, and all height information is maintained in
order to provide accurate transition from one tile to another. The results are shown in the figures below.

Figure 36: End result with vertex shader skirting (own survey, 2013)
7. Conclusion

This work illustrated how CPU based Level-of-Detail can improve rendering performance on a mobile device. The vertex structure has been set up by defining a regular grid with fixed spaces between the vertices. We found that terrain tiles with the size 64x64 perfectly fit the caching scheme of OpenGL ES 2.0. By using predefined multilevel index buffers, the prototype introduced a simple method for creating different Levels-of-Detail, while maintaining a constant vertex buffer, which can reside on the GPU. The management of transitions between different LODs is entirely managed by the tiles themselves, which avoids the necessity of an additional data structure on top, in order to save CPU bandwidth. However a quadtree like structure might be able to improve performance by reducing the amount of detailed tiles and therefore the overall triangle count, as the current scheme tends to select tiles with higher than necessary details, for mid and far-away areas, if the camera is close to the terrains surface. Even though visibility checks are performed, that take care of non-visible terrain tiles which are not located within the view frustum, to be rejected, the implementation of horizon-culling could further increase rendering performance for situations where the camera is close to surfaces that tend to occlude a lot of geometry ahead. As the current solution only makes use of Discrete Geometry LODs, this is also a section that surely allows room for improvements. The visual crack elimination performed in the vertex shader provides a satisfactory solution, as it can easily be combined with the mandatory Vertex-Texture-Fetch.
List of Figures

Figure 1: Smartphone owners that use their devices for playing games (www.thinkwithgoogle.com, 2013) 4

Figure 2: Terrain (www.outerra.com, 2013) 6

Figure 3: Multi-resolution instances of 3D object (www.opensg.org, 2013) 7

Figure 4: 60.000 vs. 600 polygon model at different distances (modification, www.opensg.org, 2013) 7

Figure 5: Top-down vs. bottom-up approach (Level of detail for 3d graphics, 2003) 10

Figure 6: Bintree Levels 0 – 5 (Duchaineau, 1997) 11

Figure 7: Diamond on the left side, split diamond on the right side (Duchaineau, 1997) 12

Figure 8: Mip-Map levels for a texture (http://game-art.co.uk/agtec/html/mipmaps.html, 2013) 14

Figure 9: Android basic lifecycle (commons.wikimedia.org, 2013) 16

Figure 10: OpenGL ES 2.0 Programmable pipeline (www.khronos.org/opengles/2_X/, 2013) 17

Figure 11: Smartphone used for prototype (http://cdn.pollin.de, 2013) 19

Figure 12: Resource manager UML diagram (own survey, 2013) 20

Figure 13: Camera setup in scene (own survey, 2013) 21

Figure 14: OpenGL ES 2.0 view matrix (www.songho.ca/opengl/files/gl_anglestoaxes01.png, 2013) 22

Figure 15: Gesture based camera movement (own survey, 2013) 23

Figure 16: Yaw and Pitch (own survey, 2013) 24

Figure 17: Log file created by Logger Class (own survey, 2013) 25

Figure 18: Calculation of distance between fingers (commons.wikimedia.org, modified, 2013) 28

Figure 19: MainActivity UML diagram (own survey, 2013) 29

Figure 20: Terrain Height map and plane vertex buffer (own survey, 2013) 34

Figure 21: Vertex buffer 100 x 100 GL_POINTS with texture coordinates (own survey, 2013) 37

Figure 22: Vertex buffer (own survey, 2013) 37

Figure 23: Vertex buffer 100x100 GL_TRIANGLES (own survey, 2013) 38

Figure 24: Index buffer (own survey, 2013) 38

Figure 25: Solid indexed terrain 100x100 (own survey, 2013) 39

Figure 26: Four draw calls with 64x64 Terrain (own survey, 2013) 41

Figure 27: Index buffer level 2 (own survey, 2013) 41
Figure 28: Index buffer level 3 (own survey, 2013)  43
Figure 29: Index Buffer Level of Detail (own survey, 2013)  44
Figure 30: Radius based Level of detail for one Terrain Tile (own survey, 2013)  45
Figure 31: Terrain 512x512 level of detail (own survey, 2013)  46
Figure 32: Terrain 512x512 with visibility check (own survey, 2013)  47
Figure 33: Cracks at borders of tiles (own survey, 2013)  48
Figure 34: Visual crack elimination by placing skirts around each tile  48
Figure 35: Vertex Shader Skirting (own survey, 2013)  49
Figure 36: End result with vertex shader skirting (own survey, 2013)  50
List of Tables
Table 1: Indexed vs. non-indexed data rendering performance (own survey, 2013) 40
Table 2: In between sampling (own survey, 2013) 40

List of Abbreviations

A
ADT Android Developer Tools
API Application Programming Interface

C
CLOD Continuous Level of Detail
CPU Central Processing Unit

G
GIS Geographic Information System
GLSL OpenGL Shading Language
GPU Graphics Processing Unit

L
LOD Level of Detail

N
NPOT Non Power Of Two

R
ROAM Real-time Optimally Adapting Meshes

S
SPEC Standard Performance Evaluation Corporation

V
VBO Vertex Buffer Object
VTF Vertex Texture Fetch
List of References


http://developer.android.com/


http://www.opengl.org/wiki/Shader

http://www.spec.org/gwpg/gpc.static/vbo_whitepaper.html


Appendix