Rendering of Streamed Vector Maps on a Mobile Phone

Dan Nilsson

Department of Information Technology
Lund University

Advisors: Mats Cedervall (LTH), Martin Santesson (Wayfinder)

August 7, 2008

Printed in Sweden
E-huset, Lund, 2008

Abstract

This thesis documents the design and implementation of a 3D vector map renderer for mobile phones. It was carried out at Wayfinder Systems, which previously used a limited 3D renderer that is layered upon an old 2D renderer.

An overview of mobile 3D API:s is provided, which focuses on cross platform compatibility and performance. A target API is chosen (OpenGL ES) and an extensible rendering framework is developed. The different steps required for preparation of vector map data for such a framework is documented.

Features such as elevation model support are implemented in order to fully demonstrate the advantages of the new renderer and an overview of other map features that could be added is also provided. An algorithm for splitting up polygon data against a terrain model is presented which should be more effective than standard clipping algorithms.

Finally, the thesis concludes with an evaluation of the new renderer.

Acknowledgments

First, I would like to express my thanks to Martin Santesson at Wayfinder, who has been invaluable when discussing new ideas and concepts for the thesis. I would also like to thank Mats Cedervall at LTH, who has provided great support that made the thesis possible.

Last but not least, I would like to thank my family and friends who provided support and inspiration throughout the writing of the thesis.

Table of Contents

Chapter 1

Introduction

Wayfinders maps are sent to the mobile clients dynamically, which always allows the user to get reasonably up–to–date maps. Since the map data might have to travel over the GPRS network (which has a peak bit rate of 116 kbps), it needs to be heavily compressed.

A proprietary vector map format was developed in order to fulfill this requirement. Instead of sending out images of the maps to the client, the geometric layout is encoded into vector data. Roads and houses are represented as separate polygons instead of pixels. This minimizes space usage better than pixel based formats can do, at the cost of forcing the client software to perform more advanced drawing logic.

Wayfinder currently uses a projection of a flat 2D drawing surface to display 3D. This approach has performance issues and it cannot easily be extended to draw new types of map data (such as terrain and 3D models). This thesis examines ways to overcome these problems by designing an efficient and extensible 3D vector map renderer. The primary goals that should influence the design process are as follows:

- Choose an efficient 3D API that allows support for hardware acceleration
- Create a renderer that can be used across multiple platforms
- Add support for terrain data

- Add support for 3D models

The report begins with a brief introduction of implementation considerations in Chapter 2, including choice of platform and 3D API.

In Chapter 3, the reader will learn about the challenges of preparing data for the new renderer, using the current vector map format as input. The chapter primarily deals with how make the new renderer draw the same flat vector maps that the previous renderer could. The performance of the new renderer compared to the old is evaluated with benchmarks.

Chapter 4 discusses the concept of elevated terrain and how it applies to the rendering of maps. Auxiliary height data is used in conjunction with the vector map to create elevated surfaces. It ends with a brief discussion on how this new data might be distributed to client software.

3D models and how they can be used when rendering maps is discussed in Chapter 5.

The final chapter discusses the results of the thesis along with future work that can be carried out.

Chapter 2

Implementation Considerations

2.1 The requirements

Wayfinder's vector map rendering has been been implemented for several application types on multiple platforms. A goal in this thesis was to investigate the different graphics API:s that are available for these platforms so that a suitable alternative could be chosen as a basis for the new renderer.

API stands for "Application Programming Interface" and it describes code interfaces that are usually used as abstraction layers over lower level functionality. When writing graphics software, using such interfaces enables the programmer to write more general code without working directly against different hardware configurations.

For this thesis, an API for 3D graphics had to be selected. Since the rendering of 3D graphics is computationally demanding, it would be desirable to have a high performance library that can take advantage of hardware acceleration on devices which support it.

To reduce the amount of platform specific code that needs to be written, it would be advantageous if the same API is open and cross platform, meaning that the same code can be compiled for different platforms without changes. This is particularly useful since writing applications for mobile phones usually involves targeting multiple platforms and architectures.

2.2 API abstraction levels

API:s are often implemented on different abstraction levels. Some API:s provide high level abstractions for 3D worlds (using scene graphs, see Figure 2.2) while some only define the most basic drawing routines. Regardless of how they are implemented, the vast majority rely on the same basic primitives. Some higher level API:s only implements a scene graph design and depends on lower level API:s to perform the actual drawing.

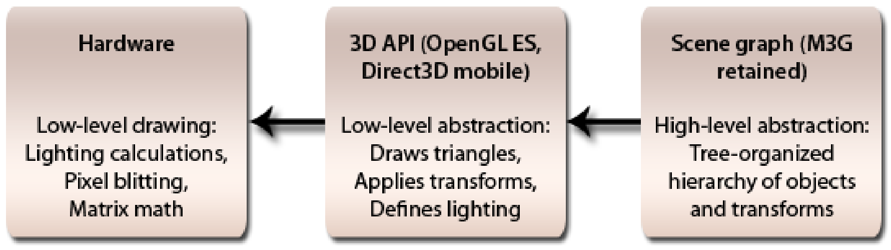
Hardware
Low-level drawing: Lighting calculations, Pixel blitting, Matrix math
3D API (OpenGL ES, Direct3D mobile)
Low-level abstraction: Draws triangles, Applies transforms, Defines lighting
Scene graph (M3G retained)
High-level abstraction: Tree-organized hierarchy of objects and transforms

Figure 2.1: Dependency chain of abstraction layers.

A scene graph stores shapes and transforms (light, object movement etc) as high level objects called nodes. The nodes are organized in a tree structure, allowing groups of transforms to be applied to shape nodes in a stack–like fashion. The structure is very general and can describe a number of different scene types using a single format that can easily be serialized.

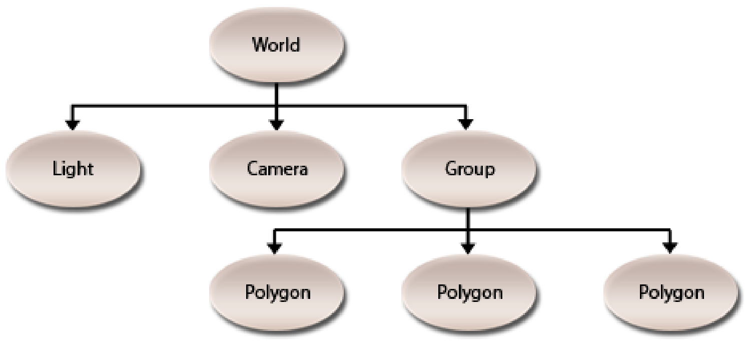

Figure 2.2: Basic layout of a scene graph structure.

2.3 API overview

The purpose of this thesis is not to discuss implementation details tied to specific API:s. It should instead document the difficulties and design considerations of a mobile 3D vector map renderer which are relevant for any 3D API. However, to provide an overview and to justify the choice of development platform, an evaluation of the available API alternatives is presented.

2.3.1 OpenGL ES

OpenGL ES is a scaled down version of desktop OpenGL, primarily designed for embedded devices such as consoles, phones, appliances, and vehicles. It is built from well–defined subsets of desktop OpenGL and adds support for features such as fixed point arithmetic that are relevant for embedded platforms.

There are a few different versions, the most common ones in use today is 1.0 and 1.1. A new version (2.0) is currently being introduced, which will be more oriented towards programmable graphics hardware (primarily focusing on shader programming).

The Android SDK from the Open Handset Alliance Project will support OpenGL ES 1.0. Apple's iPhone SDK uses 1.1 and Symbian platforms (UIQ and S60) supports 1.0 in earlier devices and 1.1 in later.

For Java platforms there is the JSR 239 specification, which defines a direct binding to OpenGL ES 1.0. It is supported on the Sony Ericsson Java platform, beginning with version JP-8. There is also the M3G API with wider support that is also tied to OpenGL ES, which will be discussed later.

2.3.2 Direct3D Mobile

Like OpenGL ES, Direct3D Mobile is a scaled down version of its desktop counterpart. Like OpenGL ES it has a feature set oriented towards mobile programming with features like fixed point arithmetic.

It is made by Microsoft and is currently only supported on Windows Mobile platforms. It is a proprietary API and suffers from a lack of cross platform compatibility. Support for the technology was introduced with the Windows Mobile 5.0 platform in 2005.

2.3.3 M3G - Mobile 3D Graphics

M3G is an API for the Java 2 Micro Edition platform. It is not based on desktop implementations such as Java 3D, instead focusing entirely on mobile devices. It was specified by the Java Community Process, which is a formalized process that allows interested parties to be involved in defining future versions and features of the Java platform.

Unlike the previously mentioned API:s it has no support for fixed point arithmetic, relying only on float and integer primitives to perform math operations. Even though the API lacks direct support of fixed point arithmetic, its specification mandates that it should be possible to implement it on mobile devices without hardware support for floating point operations.

M3G has two modes of operation, retained and immediate mode. The retained mode supports a scene graph API while the the immediate mode only supports low level drawing.

The M3G standard mandates that the immediate mode should be compatible with OpenGL ES and that the retained mode should be built entirely on top if it. This enables M3G to be quickly implemented on new platforms using existing OpenGL ES libraries. For more information regarding the relationship between M3G and OpenGL ES [13].

2.3.4 Mascot Capsule

Mascot Capsule was the first commercially deployed mobile 3D API, predating standardized API:s which came later. It was developed by the Japanese company HI Corp and first released in 2001. Initially supporting only event–driven control of skeletally animated characters using orthographic projection and z–sorted polygons, it was later extended with a generic and robust feature set combined with a lower level API.

Subsequent versions have added more features and in 2004 an implementation using M3G for low–level drawing was made a part of their fourth version. The low level drawing is either implemented as a proprietary software rasterizer or with OpenGL ES, depending on the target configuration.

2.4 The platforms and conclusion

As previously mentioned, most API:s are similar in feature sets (all are adequate enough to perform the rendering required by this thesis). Perhaps the rendering speeds of the API:s are different, and that is usually relevant, but that can depend on many factors and investigating that is beyond the scope of the thesis.

The two stand–alone alternatives are Direct3D Mobile and OpenGL ES. The other API:s are all layered upon either software rasterizers or OpenGL ES. These derived API:s are all tied in some way to OpenGL ES. This makes a compelling case for using OpenGL ES—if the features can be implemented in bare OpenGL ES, they can be implemented using the derived API:s. This is guaranteed by their documentation.

Even if OpenGL ES and Direct3D Mobile are not directly comparable, they are similar in many aspects. It is likely that a renderer with a modular structure can support both API:s. This is due to the fact that much of the code consists of preparation of input data to the renderer, which can be shared across platforms.

Only M3G supports a scene graph structure. If there was a wider selection of API:s that implemented this level of abstraction, or if M3G had a larger number of supported platforms, the use of a scene graph structure would been a factor when deciding which API to use.

It was ultimately decided that the Symbian platform should be used as an initial target platform, which has excellent support for OpenGL ES. The implementation language of choice for this platform is C++.

As previously stated, all of the API:s that were considered had adequate technical support for the implementation. If that would have been the only factor, it could just as well have been implemented on a Java platform with support for M3G/OpenGL ES or on a Microsoft platform with Direct3D Mobile.

Chapter 3

Drawing Flat Maps

The map data is organized into tiles with separate layers for different parts of the geometry. One layer might contain building outlines and another road networks. The drawing order is explicitly defined and the map is essentially made up of overlapping polygons.

There are two polygon types. They both share common meta attributes, such as color and names.

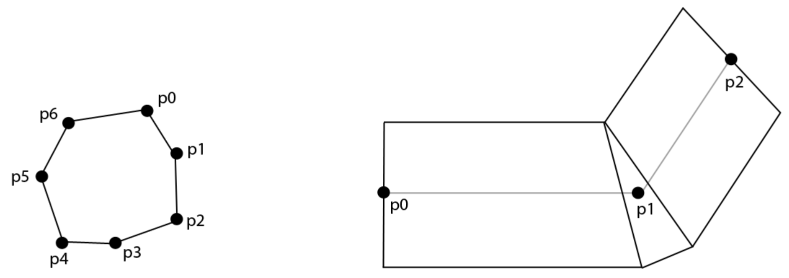

Figure 3.1: Polygons and polylines

- Polygon - series of vertices (polygon point) that defines the hull of a closed polygon.
- Polyline - series of points that defines the center of a line curve (roads, railways etc.)

3.1 Visualizing data

The previous renderer was first designed to draw flat top–down maps. It used native 2D drawing API:s for each platform that often could draw the polygons directly. Instead of creating a new renderer for its new 3D mode, Wayfinder opted to add a layer upon the old renderer. This layer perspective transforms the vertices of the polygon shapes before the 2D map renderer draws them (see Figure 3.2). In terms of implementation work this was a time saving approach since only a transformation layer was added to an existing design.

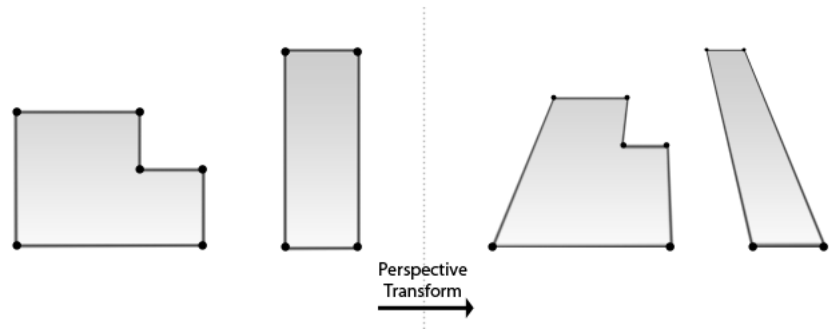

Figure 3.2: The transformation of vertices that was added to the old renderer in order to achieve a 3D mode.

However, each new platform requires a mapping between its 2D drawing API:s and the rendering code. Furthermore the transformation pass will have to be performed whenever the 3D mode is active.

When using a 3D API such as OpenGL ES, the transformations pass is performed by the API. This reduces code maintenance and also allows these transformations to take place on 3D hardware when available, making them significantly faster. With the correct perspective set, OpenGL ES can also be used to draw the old top–down maps.

A more serious concern with the old renderer is that it lacks support for new map features. Adding support for features like terrain rendering and 3D models in the old renderer would be comparable to implementing a complete proprietary graphics API. This is undesirable—doing so requires massive code maintenance and work effort. If a 3D API like OpenGL ES is used instead the support for implementing such features is available from the start.

With OpenGL ES the entire rendering engine can be made cross–platform with relatively little effort while benefiting from superior performance.

The preprocessing steps necessary to prepare data for such a rendering engine, capable of drawing the same flat 3D mode that the previous renderer could, are described in the following section.

3.2 Triangulation

3.2.1 Introduction

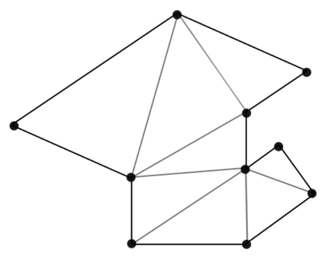

Figure 3.3: A triangulated polygon

Mobile 3D API:s rarely support the direct drawing of flat polygons. Instead they focus on efficient drawing of triangles, which then in turn can be used to build up polygons. This means that all polygons in the input data needs to be converted into sets of triangles, a process which is called triangulation.

The following sections will deal with how to triangulate polygons and polylines.

3.2.2 Triangulating polylines

A polyline consists of a sequence of points and a width value that together defines a line curve polygon. To triangulate these, there is a need for an algorithm that iterates over the points of input data and builds up a polygon of connected triangles. Two such algorithms were constructed (see Figure 3.4), each creating polygons with different properties.

The first relies upon lines that join triangulated quadrilaterals (polygons with four vertices) to build the polygon shape. The line joining two quadrilaterals at each point is perpendicular to the vector $p_1 - p_0$. If n is the number of points in the input set this simple approach will only require $2n$ triangles. However, it has a severe drawback in that the width of the road is not uniform, creating a curve with a skewed appearance.

A better solution is to use a triangle to join the two quadrilaterals, thereby reducing them to trapezoids. The width of the polygon is then constant in the trapezoids, though still not in the triangles. If this skewed property in the triangles needs to be reduced, additional points can be added to make it follow the actual shape better (see Figure 3.5). This could be done by evaluating the angle of the turn and letting turns with larger angles be divided into more triangles.

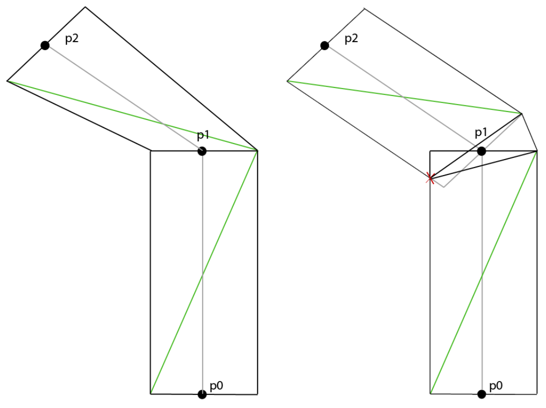

Figure 3.4: Skewed and accurate way of triangulating polylines

The best solution will still be to have more input points near turns. Since the points from the vector data fulfilled this to an acceptable extent, a single triangle with no extra control points was used to join adjacent trapezoids.

A compact way to represent triangles is to use triangle strips. The first triangle is explicitly specified using three vertices, and all subsequent by using only one new point and two old. This means that consecutive triangles share an edge (see Figure 3.7). Triangle strips can be used for polylines. If the algorithm that produces skewed line segments is used, the number of vertices will be reduced to about $2n$ from $6n$. For the accurate algorithm, the number of vertices is about $4n$ compared to $9n$ for the non–strip version.

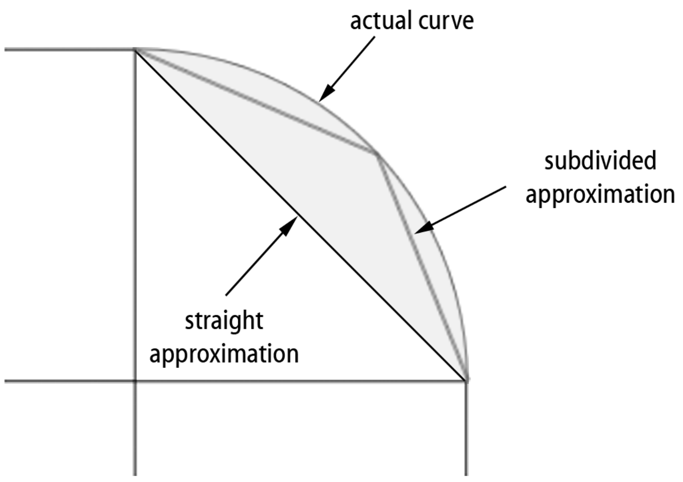

Figure 3.5: How to represent turns in polylines. A straight approximation leaves visible errors in sharp turns. Subdivided approximations reduces the magnitude of this error.

Note that one extra point is needed for the accurate algorithm since the algorithm needs to realign after making a joining triangle. The reason for this is that the following quadrilateral needs to share the upper two vertices of the triangles (see Figure 3.6)

To deal with these kinds of situations, where the next triangle needs different vertices than the previous two or where the order is wrong, a swap operation can be used. In OpenGL this is done by re–adding one of the two previously entered vertices. This creates triangles that are not visible when rendering.

3.2.3 Triangulating polygons

A polygon is stored as a clockwise sequence of vertices that defines its hull. There are a lot of triangulation algorithms for polygons with different time complexities. The simplest algorithm available is probably the "Subtracting Ears" method, which continually creates triangles by removing ears (triangles with two sides on the edge of the polygon and the other one completely inside) from the polygon. It requires that the input polygons are not self-intersecting

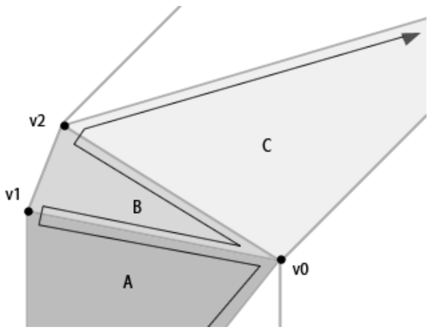

Figure 3.6: Situation that requires a swap operation. Note that after the first triangle A has been closed using vertices $v0$ and $v1$, the algorithm then adds the vertex $v0$ again. This creates a dummy triangle that realigns the output. Finally the second triangle B is created using $v2$, an operation which also specifies the first two vertices of the third triangle C.

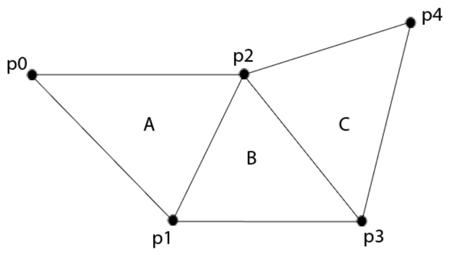

Figure 3.7: 3 triangles (A, B, and C) specified using only 5 vertices

and free of holes, which are conditions that the input polygons do satisfy. Its time complexity is $O(N^2)$, but with a low constant factor. Since the input polygons were guaranteed to consist of a small number of vertices this algorithm was used for the thesis.

If it should prove to be a bottleneck in the future, there are asymptotically faster algorithms (though none as simple). By splitting up a polygon into monotone sub polygons which then can be triangulated in linear time, an $O(nlogn)$ algorithm is attained [6]. Bernard Chazelle showed in 1991 that any simple polygon could be triangulated in linear time [2], though the algorithm proposed was very complex. Simpler algorithms are still being looked for [1].

3.2.4 Format discussion

A relevant question is whether or not these triangulations should take place on the server or on the client side. The placement of processing is a trade–off between bandwidth consumption and client performance. In the current implementation all polygons are sent in the format described in the beginning of the chapter.

For the polylines, the current representation requires significantly less bandwidth than a pretriangulated alternative. If the accurate algorithm is used before sending the vertices at least 4 times as many vertices are sent. This is due to the fact that the algorithm generates the vertices of the polygon using an exact mathematical description where the input is a smaller set of interior center points.

There are no bandwidth advantages for pretriangulated polygons either. An optimal triangle strip representation could theoretically only require as little as n vertices to represent the polygon. Some polygons need swaps however (see Section 3.2.2). Finding the optimal strip representation for a polygon is an NP–complete problem [5], though there are algorithms that gives approximate results in linear time [5].

The rendering time on the client could be improved if advanced triangle strip algorithms (which outputs smaller triangle strip representations) are performed offline on the server. This is because triangle strip representations consume less bandwidth when sent to the graphics hardware.

In the end, the reduction of processing time and the improved rendering times on the client comes at the cost of higher bandwidth usage. If acceptable triangulations (in terms of processing and rendering time) can take place on the client side the issue of lower bandwidth usage becomes the dividing factor in favor of a client–side solution.

Furthermore, a more serious concern will arise when the concept of terrain is introduced (see Chapter 4). When dealing with terrain certain assumptions that were valid in a flat world are no longer valid and additional processing steps will be introduced. If these steps would be performed on the server, two separate sets of triangulated polygon data would have to be kept and sent separately to the client.

If it is feasible to perform triangulation and splitting (see Chapter 4) on the client side it will be possible to support two separate modes of display (flat and terrain) using the same data. The bandwidth usage will also be the lowest possible. For these reasons, it was decided that the client–side approach should be focused on in the thesis.

3.2.5 Result

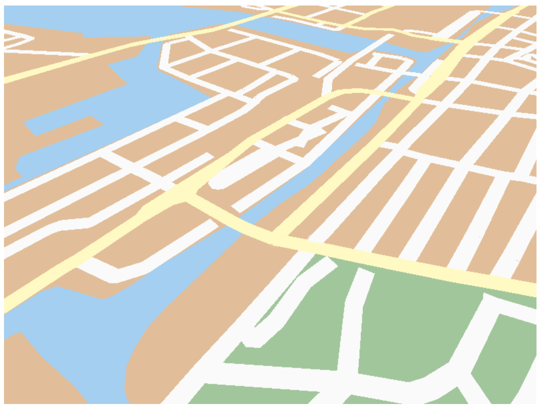

Figure 3.8: OpenGL ES rendering of the city of Malmo in Sweden. Both polygon and polyline triangulation working.

The implementing of the techniques described in this chapter was performed without major difficulties. A renderer was developed which was as capable as the previous. The new design approach meant that even an early unoptimized renderer outperformed the previous by a large margin (see Table 3.2.5).

The OpenGL ES rendering code was separated into specific modules, which meant that the rendering engine was not explicitly tied to the 3D API. This makes it possible to implement support for different 3D API:s on platforms that lacks support for OpenGL ES. The only requirement is that they are able to use the same triangulated data that is generated in the platform independent preprocessing step.

Renderer	Frame time	FPS	Improvement
Old	367.13 ms	2.75	1.0
New with AA	33.03ms	30.3	11.1
New without AA	18.05 ms	55.6	20.3

Table 3.1: Comparison between the old renderer and the new on a Sony Ericsson P1i device. The new renderer was benchmarked both with and without AA (anti–aliasing).

Chapter 4

Terrain Data

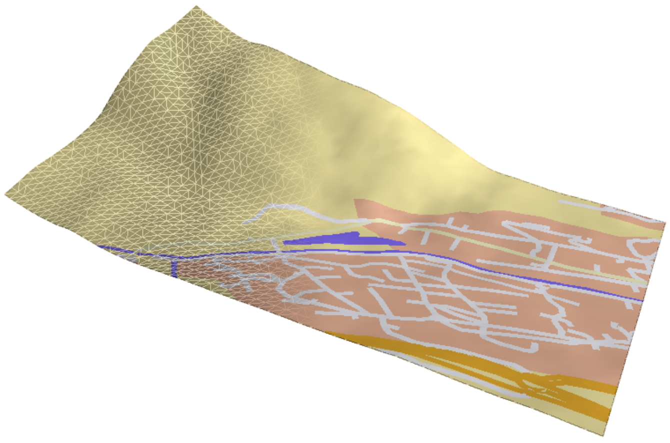

Figure 4.1: Terrain rendering of Chamonix in eastern France, overlaid with the height grid that was used to elevate the terrain. Made using actual output from renderer.

4.1 Format

Today, map suppliers offer world–wide digital elevation level coverage. The data used for this thesis comes in two primary formats. The first is a grid (a matrix of height values) and the second is in the form of polyhyps. Polyhyps are oriented towards traditional top–down maps and are therefore not suited for 3D terrain modeling (since they cannot easily be used to create polygons).

For this thesis the basic grid format was used to create the terrain. There are other formats that are derived from the sample values of this grid model that will be discussed at the end of this chapter.

4.2 Creating the terrain model

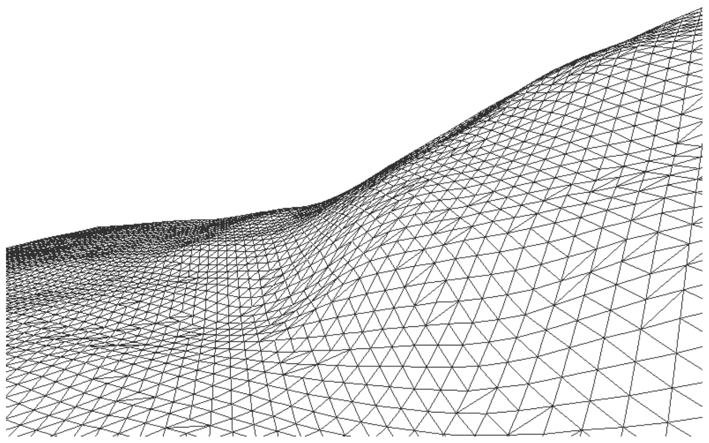

Figure 4.2: Terrain model

The grid is used to build up a terrain model by stitching together triangles. A pair of triangles makes up a square which connects four elevation value points. As Figure 4.2 shows, there are two triangle layouts that can make up a square depending on how the diagonal is placed.

Both layouts are approximations. If the sample points were spaced with smaller intervals, the correct alternative could be determined. To obtain a fair approximation, the average value of the four corner points is calculated.

Then the midpoint of the two possible dividing lines is calculated. The line with the midpoint that is closest to the average point gets chosen.

If an even better approximation is desired, values from close height points can be added to the midpoint calculation. If instead the implementation favors speed over correctness, the step could be summarily skipped by always using one of the layouts.

4.3 Choosing a 2D adaptation

Now that the terrain model is well defined, the next task is to adapt the vector data so that it can be laid over the terrain. There are two approaches:

1. Make textures from vector data and wrap it over the terrain
2. Adapt the vector data so that it follows the terrain

The first approach has the benefit of using fewer triangles at the cost of maintaining and generating texture sets. A drawback is that the resolution of the textures is lower than that of the vector layout, which is for all intents and purposes infinite. To approximate this multiple textures with different level of detail would have to be used.

However, the second approach has a fair number of problems that need to be addressed as well. Both solutions will require more triangles compared to what they would with a flat terrain, though the adapted vector map solution will require more. This is in part due to the fact that the polygons will need to be split up in order to fit the underlying terrain. The number of triangles produced depends on the grid spacing (see Section 4.8).

If the number of triangles can be kept at a reasonable level for rendering, and if the preprocessing step can be done in acceptable time, then the main difference between the two is the resolution of the polygons. Ideally a fair comparison requires two reference implementations, but since there were time constraints the adaptation of the vector maps was the solution that was focused on in the thesis.

4.4 Adapting vector data

4.4.1 Problem description

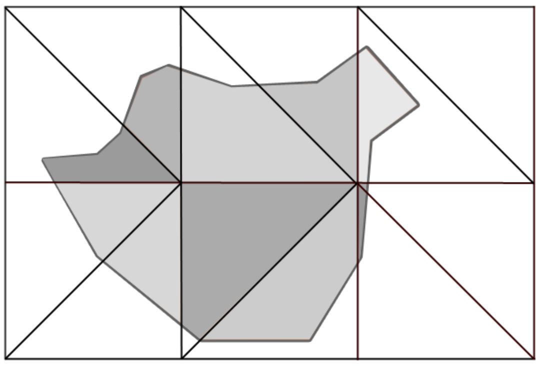

Figure 4.3: Polygon splitting against the terrain triangles.

All of the polygons will need to be split against the triangles of the terrain grid which will create smaller sub polygons. This is to make sure that the shape of the polygon follows that of the terrain.

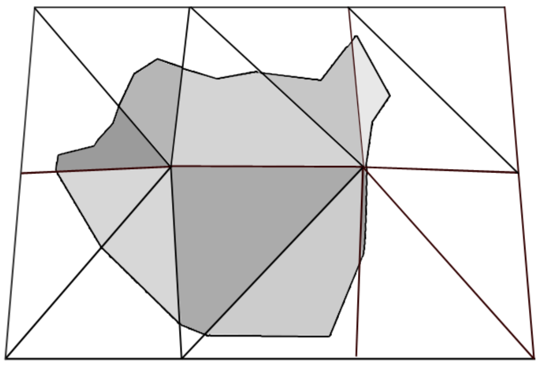

Figure 4.4: Split polygons, now adapted to the terrain triangles heights.

After the polygons have been split against the triangles, they are triangulated. Height values are computed using barycentric interpolation for all vertices in the resulting triangles, thus adapting the triangles to follow the grid terrain (see Figure 4.4).

An alternative approach that was considered triangulates the polygons first and then those triangles are split against the terrain triangles. After the split the resulting polygons will have to be triangulated once again. The advantage of this method is that the splitting is between shapes that are guaranteed to be convex which enables simpler and faster splitting algorithms such as Sutherland-Hodgman [17] to be used. The disadvantage is the two triangulation steps and that the resulting triangulation can consist of a larger number of triangles. Therefore this method was not implemented in the thesis.

If lighting is to be used, normals have to be calculated for all vertices.

4.4.2 Reference implementation

The more common term for the algorithms used in splitting is "clipping", and it illustrates what they are usually employed for. A standard problem is clipping polygons against the viewport of a program, thus reducing the number of drawing operations necessary. However, splitting is a more appropriate term for the problems presented in this thesis because no data should have to be clipped away (discarded).

The outline of a simple and general algorithm that can be used to adapt a polygon to a terrain model is as follows:

1. Find bounding box of polygon
2. Split the polygon against all the triangles in the area of the bounding box

There are several different algorithms which can be used in the second step. The simple variants such as Sutherland-Hodgman creates a single polygon with possibly degenerate edges when concave input polygons are used, which will cause the triangulation process to fail or output unnecessary triangles. The more complex algorithms such as Weiler–Atherton [19], Vatti [18], and Greiner–Hormann [7] all avoid this, with differing complexity.

If a simple algorithm such as Sutherland-Hodgman is used, the resulting triangulation of the polygon will contain unnecessary triangles for re–entrant polygons (concave).

To gain an understanding of the nature of the problem a reference implementation was implemented first based on the described outline. As long as the splitting algorithm is correct and produces non–degenerate polygons, the output should be the same for all of these algorithms. The reference implementation uses a variation of the Vatti algorithm, implemented by the GPC library [12], as a splitter.

This implementation produced correct results, though the preprocessing time was quite slow. Due to time constraints, a better alternative was not implemented in the time frame of the thesis. In the following section a customized algorithm is proposed which should give better performance.

Another possibility that should be investigated is the use of a fast and simple algorithm such as Sutherland-Hodgman. Its output for re–entrant polygons was first thought to be unacceptable for later processing stages (triangulation and light processing (see Section 4.6), but more thorough investigation hinted at the possibility of this being false. Perhaps some additional preprocessing step can be performed (perturbing certain points) that will allow the triangulation functions to work. Unfortunately there was not enough time to implement a test solution.

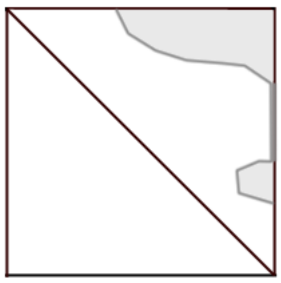

Figure 4.5: A single degenerate polygon.

If the output from such an algorithm could in fact be used, the main difference between it and the other class of algorithms (apart from the likely shorter execution time) would be that it generates a larger number of triangles. This is because some polygons are degenerate (see Figure 4.5), requiring extra triangles for the coincident edges. Even if this solution should prove more feasible, it would still have been necessary to implement a reference solution using the other class of algorithms to determine so.

4.4.3 Suggested algorithm

Description

The primary disadvantage with using the classic clipping algorithms is that they do not exploit the nature of the height grid. The clipping polygons are just a disconnected set of triangles that each requires a separate call to the clipping algorithm. A large number of vertices may have to be discarded in each call. A more clever approach would traverse the polygon once and split it against all the grid triangles it crosses in one pass.

Moving through the grid

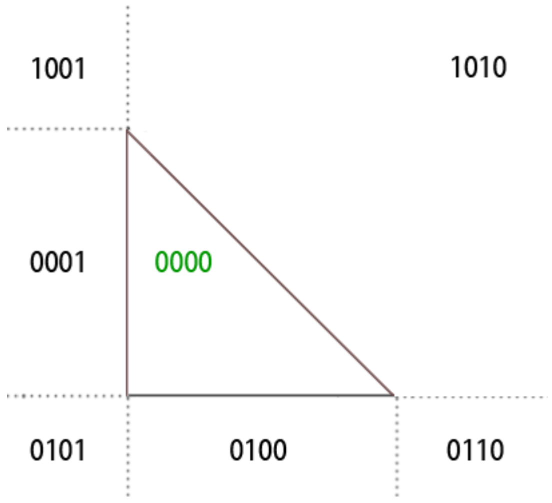

Figure 4.6: Shows the outcodes for a triangle. The 1010 area is above and to the right of the triangle.

To find out which grid triangle the polygon crosses, the concept of *outcodes* are used (see Figure 4.6). Normally they are used for clipping rectangles, but they can be used for triangles as well. Essentially, they determine if a point is inside the current grid triangle, and if it is not, which grid triangle to move to in order to eventually reach it.

The reason outcodes are used, instead of perhaps testing for intersection points along all edges of the triangles, is to better exploit the shapes of the triangles. Outcode calculation for two of the edges is trivial since they are orthogonal to the unit vectors of the basis (which is defined by the grid). One is not and requires an intersection test. However if this test passes the point can directly be used in the algorithm.

The algorithm starts by determining which grid triangle the first point is located in. Then outcodes for subsequent points are calculated continuously relative to this triangle. Whenever the outcode is non–zero the point is outside of the current grid triangle, and the algorithm should then use the next grid triangle in the direction of the point. This triangle can be retrieved by exploiting the regularity of the height grid combined with the outcodes.

Marking inner triangles

To fill the triangles which are completely enclosed by the clipping polygon, the algorithm marks the grid triangle immediately to the right of the each grid triangle when traversing the polygon. These right side triangles are also retrieved using outcodes. Since the polygons are clockwise oriented, the interior of the polygon will always be to the right of any given point. These marked triangles are stored and at the end of the algorithm they are expanded until no non–marked triangles can be reached (see Figure 4.7).

Adding entry and exit nodes

The actual splitting is inspired by the method used in the Weiler–Atherton algorithm. As the triangles that the polygon crosses are traversed, entry and exit points are stored in a list of as attributes called nodes (see Figure 4.8) in the triangles. A triangle may have multiple pairs of such points for convex polygons.

These nodes represent points that lay on the edges of the grid triangle. In addition to points where the polygon intersects the grid triangle, the corners of the triangles are also stored in the list. Intersection point nodes are called breakpoint nodes, and can either be points where the polygon enters the grid triangle (`BREAK_BEGIN`) or exits (`BREAK_END`).

As the grid is traversed by moving from grid triangle to grid triangle, `BREAK_BEGIN`

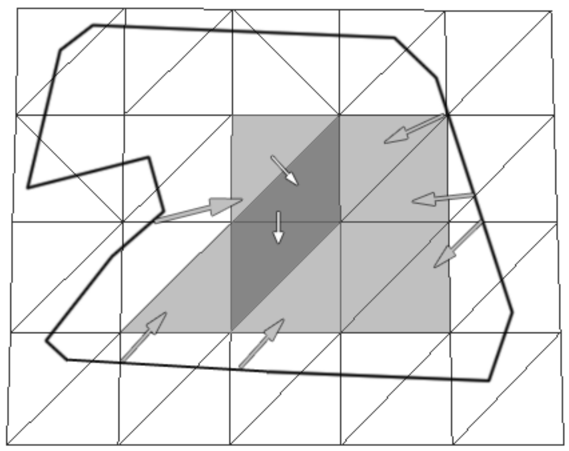

Figure 4.7: Marking of enclosed parts, all to the right of the vertices in the polygon. The darker shaded parts are marked later on in the expansion phase.

and `BREAK_END` pairs are inserted at the points where the polygon intersects the two triangles. The grid triangle that is exited from adds a `BREAK_END` node to its node list and the triangle that is entered adds a `BREAK_BEGIN`. Since the polygon has to close itself, these begin/end nodes always comes in pairs. When a `BREAK_END` is added, it is connected to its `BREAK_BEGIN` partner by the `breakId` variable. The breakpoint nodes also store indices to the points in the polygon that lay between them, inside the grid triangle (via the `outlineIndex` variable). See Figure 4.9 for an illustration.

Creating subpolygons in visited triangles

The insertion of breakpoint nodes and the marking of interior grid triangles is the first phase of the algorithm.

The next phase of the algorithm begins when the polygon is completely traversed. The idea is to create subpolygons using the breakpoint pairs that have been added to the grid triangles. The first part of such a subpolygon is the polygon outline between the two breakpoint pairs. The second consists of a combination of triangle edges and corneres that seals the first part. It is

```
enum Type { CORNER,      /* Triangle corner */
            BREAK_BEGIN, /* Polygon entry point */
            BREAK_END }; /* Polygon exit point */

struct Node {
  Vertex point;       /* Position of the node */
  Type type;          /* Type of node */
  float alpha;        /* 0..1 value representing
                         relative position on
                         triangle borders */
  bool visited;       /* Is the node processed? */
  int breakId;        /* Id of begin/end partner */
  int outlineIndex;   /* Index of first (if begin)
                         or last point (if end)
                         of polygon outline */
 };
```

Figure 4.8: Node layout

possible that other breakpoint pairs lay on the second part and if so they and their outlines are added as part of the subpolygon.

In this phase all of the visited triangles are processed again. First, the corners of the grid triangles are added to their internal node lists which are then sorted based on the alpha values of the nodes. The alpha value is a normalized unit that represents how far along a node is on the the outline of the grid triangle. For example, the corners have 0.00, 0.33 and 0.66 as alpha values (see Figure 4.10).

After the lists have been sorted they are traversed circularly. The construction of a new subpolygon begins when a `BREAK_BEGIN` node is encountered. All of the vertices in the polygon outline are added (using the `outlineIndex` pair) and the current node in the list traversal is changed to the partner `BREAK_END` node.

Now the algorithm will seal the polygon outline by traversing the node list clockwise backwards, along the grid triangles edges, back to the `BREAK_BEGIN`

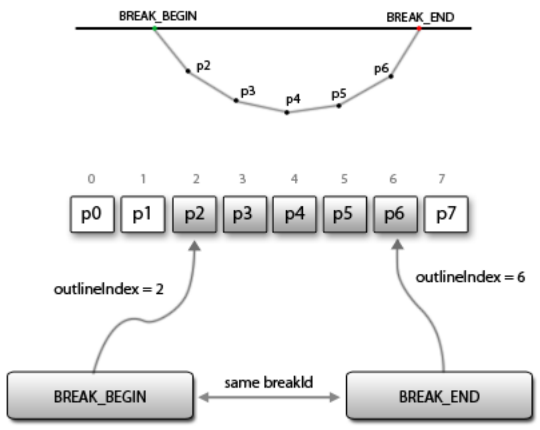

Figure 4.9: Relationship between BREAK_BEGIN and BREAK_END pairs. The part of the polygon between $p2$ and $p6$ is a the connecting line curve between the pair. The pair uses indices to the polygon array to represent this line curve instead of copying its vertices.

node. If it encounters any `BREAK_BEGIN`s that belong to other break pairs (see Figure 4.10 for such an example), it will add their polygon outlines as part of the polygon and yet again adjust the current node according to this pair's `BREAK_END` (see Figure 4.11).

If a corner node is found along the traversal, then its coordinate is added as part of the polygon. When the algorithm reaches the destination (the `BREAK_BEGIN`), a new polygon has been completed. The algorithm then continues to traverse the list, looking for unprocessed `BREAK_BEGIN` nodes. If a new one is detected then the algorithm will once again seal its polygon by clockwise traversal. If no such node can be found the algorithm has finished.

Advantages

An advantage of this algorithm is the reduction of grid triangles that need to be visited. Since it moves through the grid triangles incrementally, it only needs to visit the ones actually touching the polygon. With the previous approach, all grid triangles within the bounding box needed to be visited. Furthermore, the input for each clipping operation is minimal, since vertices outside of the clipping polygon are not part of it.

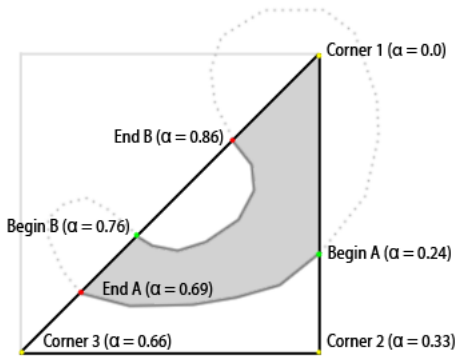

Figure 4.10: Illustrates nodes with their alpha values in a grid triangle node list.

Another advantage is that the number of polygon traversals is not relative to the number of grid triangles it encompasses. This is also due to the fact that it moves through the grid triangles touching the hull of the polygon in one pass.

Finally, due to the known layouts of the grid triangles a tight list representation can be used, with constant alpha–values for the corners.

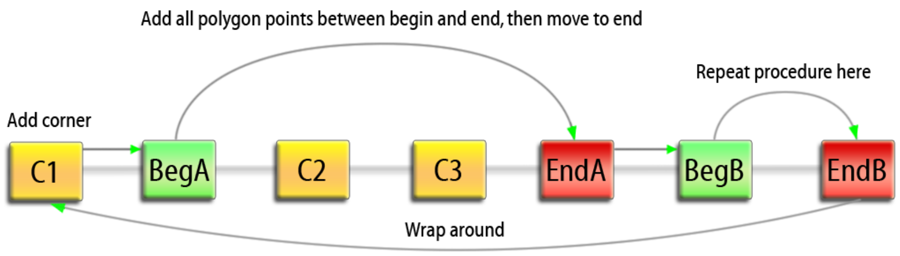

Figure 4.11: Flow of polygon building in algorithm. The boxes are nodes in a circularly traversed list. Green are BREAK_BEGIN, red are BREAK_END, and yellow boxes are corners. Based on Figure 4.10.

Future considerations

There was not enough time at the end of the thesis for creating a sample implementation. As such there is no real way to compare the traditional clipping algorithms with this one. To get an implementation with sufficient processing performance that can be used in the real world will be a challenge, but this algorithm is a first step in overcoming that challenge.

Another area of improvement might be the data type used for the math operations. Most mobile devices have no floating point unit, and a splitting implementation which uses fixed point or integer arithmetic may prove to be substantially faster. The GPC library used in this thesis uses `double` internally.

This algorithm should prove faster than naively using clipping algorithms with a bounding box. However, the fastest approach will always be to use simpler algorithms wherever possible, which is discussed in the following section.

4.5 Shortcuts

There are times when the use of a general algorithm, such as the one described in Section 4.4.3, should be avoided. For example, there are large rectangular polygons that are used to describe land areas. Due to their size, they are likely to cover large amounts of grid triangles.

Since it is trivial to determine if a set of grid triangle is inside one of these rectangles, this set of enclosed grid triangles should be determined before the splitting takes place. For these grid triangles, no actual splitting needs to be performed—the output polygons will be those in set. Also, for these cases no triangulation need to be performed (since the output shapes are all triangles).

Since convex polygons can use simpler and faster splitting methods, they should be redirected to such methods accordingly. This is particularly useful for polylines, which consists of series of very simple convex polygons—triangles and quadrilaterals (or trapezoids).

4.6 Light processing

In order to see the contours of the terrain, lighting needs to be enabled. In this thesis, only ambient and diffuse lighting was used. Diffuse lighting requires normals for each polygon vertex that needs to be calculated. A simple way of calculating the normal for a triangle is to use a normalized vector from the cross product of two triangle edges. However, this will result in very abrupt shading.

It is better to weight the normals of vertices from adjacent triangles.

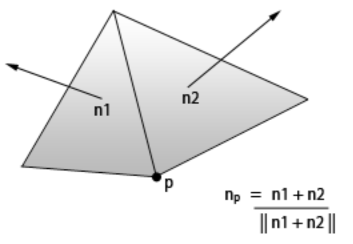

Figure 4.12: Shows how the calculation of normals is performed. The normal of point p is the normalized average of $n1$ and $n2$.

This was how the normals were calculated for this thesis. For a more accurate approximation, the angles of the corners can be used as weights.

4.7 Height offsets

When terrain models are used, the scene needs to be depth sorted. This was not necessary for the flat maps, since all polygons shared the same height value. A problem that arises when depth sorting is enabled is the fact that several overlapping polygons share the same elevation value. This makes it impossible for the 3D API to determine which polygon should be drawn last, causing a flickering appearance. The normals that were calculated in the light processing phase can be used to offset the different layers slightly in order to correct this (see Figure 4.13). ϵ should be the smallest number that causes the 3D API to treat the polygons as being on separate elevation levels (which can be dependent upon the viewport).

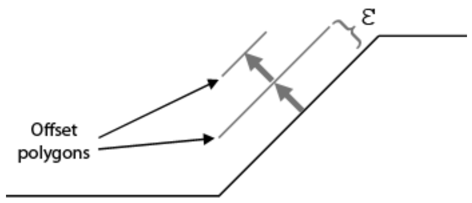

Figure 4.13: Shows how correct drawing order was achieved by offsetting polygons ϵ units along the normal vector. ϵ grossly exaggerated for illustrative purposes.

4.8 Grid spacing

The elevation grid that was used in the thesis has a spacing of 20 meters. The spacing has a significant impact on the triangle count, which influences the rendering and loading times. In order to get acceptable rendering performance, the spacing had to be increased. The number to add for each row/column increment is called the stride. A stride of 1 equals the original spacing of 20 meters, and a stride of k equals a spacing $k * 20$ meters.

As a comparison, the same world with no terrain (flat) has a triangle count of 1662. This indicates (see Table 4.1) that for larger stride values there is not a significant increase in number of triangles. These results are promising, and the renderer managed interactive frame rates at strides as low as 4 or 5.

However, the map loading times are quite long and even effective optimizations will probably not reduce them to such levels that the loading could be done in one pass. For interactive usage scenarios it would be better to process the map incrementally, which the format is suited for.

For example, the large rectangular areas which covers the land could be split and displayed early using the techniques described in Section 4.5. This will give the user an immediate sense of the terrain with further detail being added incrementally.

Frame time (ms)	FPS	Load time (s)	Triangles	Stride
70.71	14.14	4	2292	8
75.97	13.16	5	2474	7
86.89	11.51	6	2934	6
99.12	10.08	8	3437	5
120.38	8.31	11	4237	4
163.05	6.13	16	5969	3
268.25	3.72	31	10116	2
780.60	1.28	100	30518	1

Table 4.1: Map loading and rendering times on a Sony Ericsson P1i device.

Chapter 5

3D Models

An original requirement of the thesis was to add support for 3D models in the renderer. Unfortunately there was not time to fully implement this feature, but this chapter will briefly discuss the different approaches that are possible.

5.1 Landmarks

Several map suppliers are now able to deliver textured high resolution 3D models of well–known buildings throughout the world. Except for adding texture support, there is nothing that is inherently more difficult about rendering these models compared to rendering the terrain described in Chapter 4.

The real challenge lies in preparing the texture and model data in such a way so that they can be efficiently rendered on a mobile device. Reducing the amount of vertices, which results in a less refined appearance, is a necessary first step. This is called polyhedral simplification ([14, 10, 4, 15, 11, 8, 9]).

After the model has been simplified its size can be further decreased using algorithms from a research field called geometry compression [3]. When choosing such an algorithm the constrained environments of mobile devices have to be taken into consideration. The most important property will be a fast decoding algorithm. The speed of the model encoding is not as important since it only needs to be performed once, on the server side (the same model can then be sent to all clients).

The textures will also have to be compressed, for which there are also a number of algorithms. One such compression scheme, which is geared towards mobile phones, is called ipackman[16].

5.2 City models

Some map suppliers also provide simpler polygon models of city buildings. These share common textures and are not specified explicitly with regards to polygon vertices. Instead they are built up using attributes such as number of levels, height, roof type, and socket type. This allows for greater compression rates compared to regular landmark models at the cost of lower accuracy (many buildings share textures and some features are invariably lost). Another advantage of using these lower complexity models is that larger areas can be covered using simpler methods.

This effect can also be emulated using the current vector maps. If the polygons which are buildings can be identified, they can be raised an arbitrary amount (see Figure 5.1). This effect is highly inaccurate but can give an added sense of space compared to a flat map. To get more variation, the height could be varied according to distance to city center, building density and city size. Textures could also be used instead of the flat colors of the buildings (and could also be varied with the same factors).

This will also work for terrain maps. In such a context the houses will need a ground socket so that they can follow the shape of the terrain. The roof of the house can still be flat.

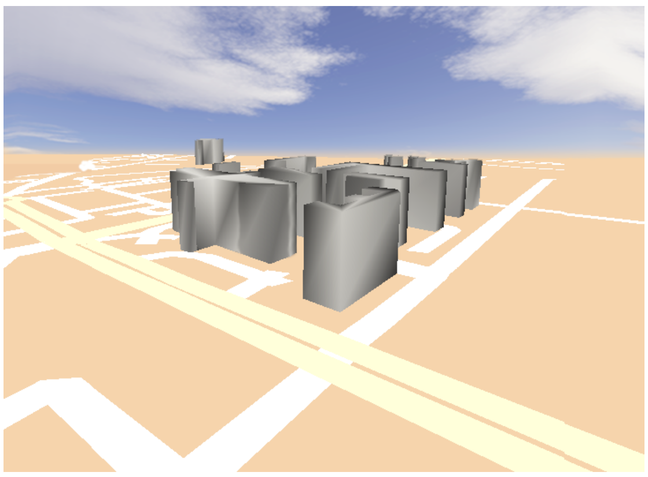

Figure 5.1: Building polygons raised above ground level.

Chapter 6

Conclusions and Future Work

6.1 3D Models

Ultimately the support for 3D models had to be abandoned due to time constraints. Getting the basic rendering working should be achievable in a relatively short time, but creating a framework for a combined server and client solution that can handle preprocessing and compression involves a lot of work. An outline and some reference material for such an implementation is provided in Chapter 5, which can serve as a starting point for future work.

6.2 Extensibility and performance

The renderer was designed to work on multiple platforms while still being efficient. The initial target platform was Symbian OS, and the implementation was carried out on Nokia's S60 framework. In order to make the development cycle faster, an OpenGL ES simulator for Linux was used for prototyping. This also meant that the code had to run on two platforms from the start.

Later on the simulator variant was extended so that it could run on the Windows operating system as well, with relatively minor effort. Mobile ports for the UIQ platform (also Symbian OS) and the iPhone (Apple) followed shortly thereafter and in the end the code runs on three separate mobile platforms and two desktop platforms. This serves as a testament to the portable nature of the rendering architecture.

The performance goal was accomplished as well, achieving rendering times over 20 times faster than the previous versions for flat maps. Note that these benchmarks were performed on a Sony Ericsson P1i, which is not one of the fastest devices on the market today.

6.3 The terrain model

Even though a reference implementation was constructed that supports terrain models a lot of work remains in this area. The process of adapting the vector map to the height grid must be improved, perhaps by using the algorithm suggested in Section 4.4.3. The rendering performance was slower than the flat terrain mode, but the frame rates were still interactive.

An important realization was that the terrain mode and the flat map model have different optimization possibilities, and should be treated with separate profiles. The flat mode does not need depth sorting or lighting, which increases performance considerably. On the other hand, the terrain model may employ occlusion based on the terrain, not rendering parts of the map that is obstructed by terrain outline.

6.4 Conclusion

Most of the goals that were laid out in the first chapter were attained. There was not enough time to thoroughly investigate certain topics, which could be attributed to the broad scope of the subject matter. However, the most important benefits of using the described rendering architecture (speed, extensibility and cross platform support) were all brought to light.

References

[1] Nancy M. Amato, Michael T. Goodrich, and Edgar A. Ramos. Linear-time triangulation of a simple polygon made easier via randomization. In *Symposium on Computational Geometry*, pages 201–212, 2000.

[2] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, 1991.

[3] Michael Deering. Geometry compression. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20, New York, NY, USA, 1995. ACM.

[4] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 173–182, New York, NY, USA, 1995. ACM.

[5] F. Evans, S. Skiena, and A. Varshney. Efficiently generating triangle strips for fast rendering, 1997.

[6] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. Graph.*, 3(2):153–174, 1984.

[7] Günther Greiner and Kai Hormann. Efficient clipping of arbitrary polygons. *ACM Trans. Graph.*, 17(2):71–83, 1998.

[8] André Gueziec. Surface simplification with variable tolerance. In *Second Annual Symposium on Medical Robotics and Computer Assisted Surgery*, 1995.

[9] H. Hoppe. Progressive meshes. In *Computer Graphics (SIGGRAPH'96 Proceedings)*, 1996.

[10] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26, New York, NY, USA, 1993. ACM.

[11] Alan D. Kalvin and Russell H. Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Applications*, 16(3):64–77, 1996.

[12] Alan Murta. General polygon clipper library. `http://www.cs.man.ac.uk/~toby/alan/software/`.

[13] Kari Pulli, Tomi Aarnio, Kimmo Roimela, and Jani Vaarala. Designing graphics programming interfaces for mobile devices. *IEEE Computer Graphics and Applications*, 25(8), 2005.

[14] Jarek Rossignac. Geometric modeling in computer graphics. pages 455–465, 1993.

[15] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *SIGGRAPH Comput. Graph.*, 26(2):65–70, 1992.

[16] Jacob Ström and Tomas Akenine-Möller. ipackman: high-quality, low-complexity texture compression for mobile phones. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 63–70, New York, NY, USA, 2005. ACM.

[17] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, 1974.

[18] Bala R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, 1992.

[19] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 214–222, New York, NY, USA, 1977. ACM.