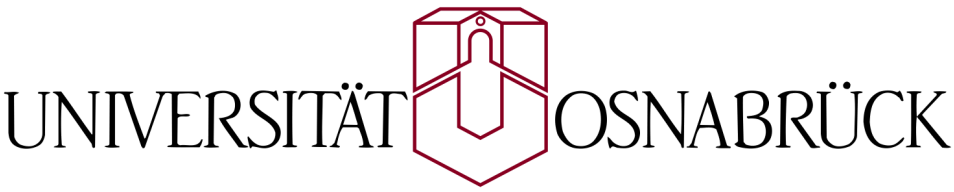


Interactive Visualization of the Earth Surface

with a Screen Resolution Dependent Level of Detail

Master's Thesis

by

Sergey Krutikov

Supervisors:

Prof. Dr. Oliver Vornberger

Jun.Prof. Dr.-Ing Elke Pulvermüller

Department of Mathematics / Computer Science

University of Osnabrück

August 2013

Acknowledgement

I would like to thank everyone who supported me whilst conducting this master's thesis, especially:

- Prof. Dr. Oliver Vornberger for supervising my work.
- my tutor Henning Wenke for professional advices and hints in the area of computer graphics.

Abstract

Interactive visualization of the Earth surface means depicting of the globe in 3D at different viewing angles and from different distances to the viewer. This term typically implies that the level of detail of the Earth surface stays high even if viewed from closer distance. Such a detailed depicting of the Earth is impossible without a very big dataset from the Earth surface. The problem is that this data, due to its size, does not fit entirely into graphics memory. Therefore, a special level of detail technique like clipmaps, virtual texture or (if it is about terrain) geometry clipmaps must be applied in order to make rendering of this data in realtime possible. This thesis is devoted to implementation of such a technique. In course of the work especially the modern features of OpenGL 4.2 like hardware tessellation with the respect to this problem are tested.

Besides, a new approach originated in course of working at this thesis is introduced. It enables to make vertex meshes which are used for terrains especially fine.

Zusammenfassung

Unter der interaktiven Visualisierung der Erdoberfläche versteht man die dreidimensionale Darstellung der Erdkugel aus verschiedenen Blickwinkeln und Entfernungen zum Betrachter. Dieser Begriff schließt typischerweise ein, dass der Detaillierungsgrad der Erdoberfläche sogar beim Betrachten aus kleiner Entfernung hoch bleibt. Solch eine detaillierte Darstellung der Erde ist unmöglich ohne die Nutzung eines umfangreichen Datensatzes mit Daten von der Erdoberfläche. Das Problem ist, dass diese Daten aufgrund derer Größe nicht komplett in den Speicher der Grafikkarte passen. Deshalb muss eine spezielle Level of Detail Technik wie Clipmaps, Virtual Texture oder (wenn es sich um Landschaften handelt) Geometry Clipmaps angewendet werden, um das Rendern der Daten in Echtzeit zu ermöglichen. Diese Arbeit ist der Implementierung von einer solchen Technik gewidmet. Im Laufe dieser Arbeit werden besonderes die modernen Funktionen von OpenGL 4.2, solche wie Hardware Tessellation bezüglich dieses Problems getestet.

Darüber hinaus ist ein neuer im Laufe der Arbeit entstandener Ansatz vorgestellt. Er erlaubt die für die Landschaften benutzen Vertex Meshes besonderes feinkörnig zu machen.

Contents

1. Introduction

Interactive visualization of the Earth surface is a very popular mode in computer graphics implemented by a wide variety of software, both games and virtual globes like Google Earth, NASA World Wind and so on. This term includes the ability of a software to depict the globe at different viewing angles and from different distances to the camera. Users of such software should be enabled to freely rotate the globe, zoom in and out on any spot on the Earth and to tilt the globe to the horizontal view in real time.

Normally, and as far as this thesis is concerned, depicting of the globe involves coloring the globe according to an imagery of the Earth as well as visualizing the terrain elevation details like hills and mountains on it. This thesis is devoted to solving both tasks.

1.1. Motivation

As to the zooming feature of such a software, the wish to enable as high zoom level as possible is just apparent. This way, the user is allowed to look at the Earth surface very closely which grants him more freedom and provides more detailed surface information to him. This implies, of course, that even at high zoom levels the picture stays detailed and clear. This can be managed only via using very detailed data of the Earth surface. In this work the satellite imagery from the NASA Blue Marble Next Generation Project ([BM04]) in which the Earth surface is represented as an RGB image with the pixel resolution 86400×43200 pixel^2 is used. Figure 1.1 visualizes the importance of choosing as detailed data as possible for these purposes: it shows how much the results of rendering at high zoom level are influenced by the level of detail of the chosen data.

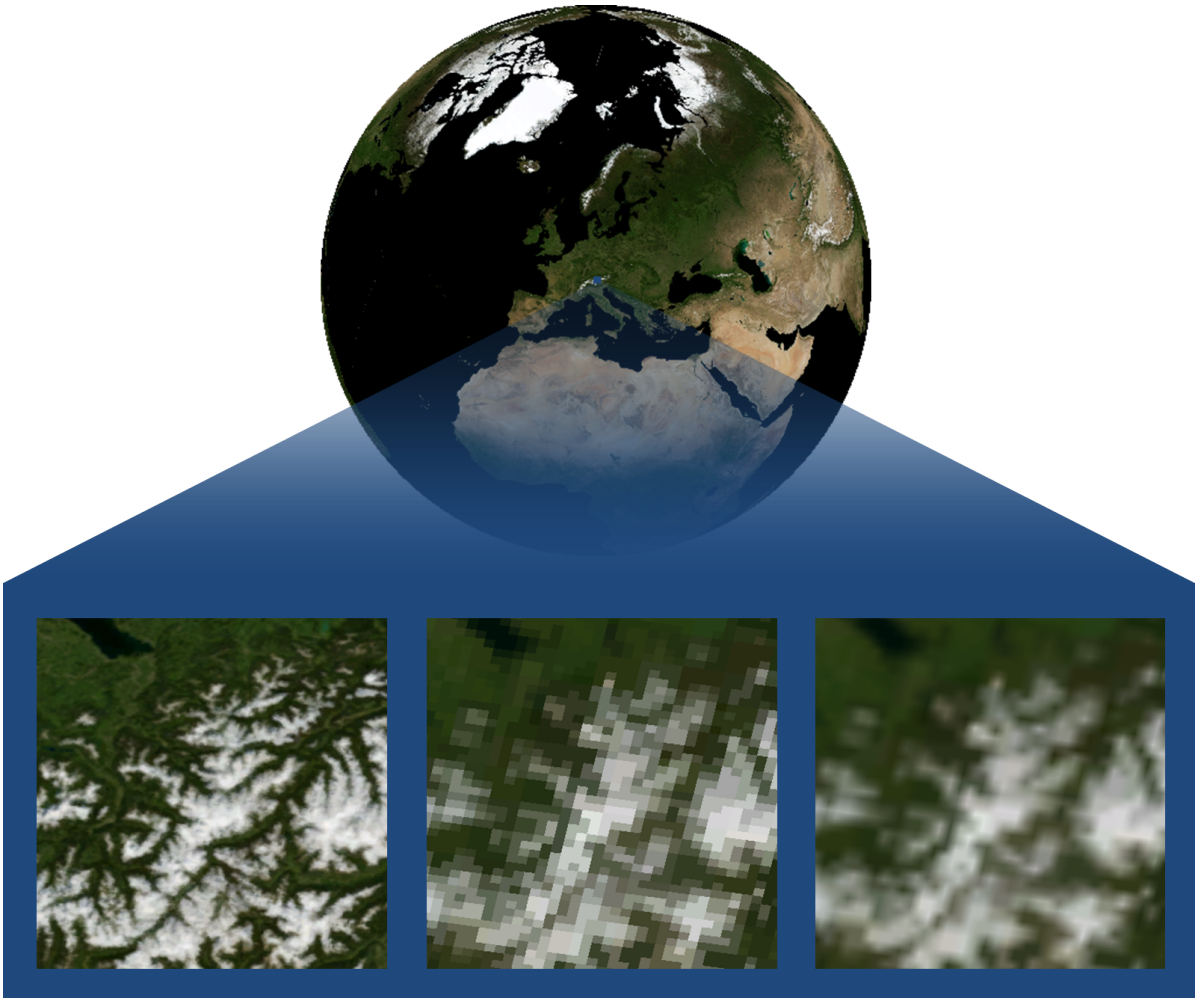


Figure 1.1.: Influence of the degree to which the chosen data is detailed on the result of rendering the globe at high zoom levels. The globe is rendered with the help of OpenGL. The globe shown at top is rendered from a big distance. The bottom three images show a spot in Alps after zooming in: the left image shows the result of this zooming if rendered with the original resolution 86400×43200 pixel^2, the middle image show the result if rendered with the pixel resolution 8640×4320 pixel^2, and the right image is the same as the middle one but with activated linear filtering.

1.2. Problem

High level of detail of the Earth surface images also means very high memory consumption. For example, the NASA Earth surface image with pixel resolution 86400×43200 pixel^2 demands 86400×43200×3B≈10GB of memory because each pixel occupies 3 bytes in memory due to the RGB-format (and this is only color data without elevation information). Even compressed, this data needs a big amount of memory to manage it.

The huge size of the surface images is on the one hand very advantageous for detailed rendering but on the other hand poses a serious problem for 3D graphics systems like OpenGL. Such systems, in order to achieve acceptable performance, render on GPU and therefore require that the images to be rendered are completely in memory of the video card. The problem is that the graphic cards have limited amount of memory and can not completely contain such huge data [1].

Depicting detailed elevation data also causes problems because geometry has to be fine subdivided in order to be able to visualize mountains and hills. In this work the elevation data is also taken from the Blue Marble Next Generation Projection in form of a height map with the same pixel resolution 86400×43200 pixel^2. Making this data to a geometry would require 86400×43200×3×4B≈45MB of graphics memory for rendering it as each point is tree-dimensional and each dimension is usually managed by graphics systems as a 4 bytes float value. Thus the detailed geometry also does not fit entirely in graphics memory.

As it is impossible to render large and detailed surface directly, any software which pretends to be called virtual globe must implement some sophisticated method which at least simulates for the user the presence of the whole Earth in the minutest details even whiteout having it in the graphics memory. In terms of computer graphics such software must implement a level of detail algorithm.

[1] In this work, a graphics card with (only) 1GB video memory was used.

1.3. Purpose

The aim of this thesis is to test how the problem defined in the previous section can be solved by using OpenGL 4.2. Especially the capability of the new features in OpenGL 4.2 (like hardware tessellation) with respect to this problem should be proved. In this manner an alternative method for solving this well known problem is tried out.

As the interactive visualization of the Earth surface implies both texturing the globe as well as visualizing the terrain elevation, this thesis must practically accomplish both tasks:

- As to texturing the Earth with a very large texture, clipmapping ([CM98]) and virtual texture ([ST08]) are the key words used with respect to this problem in the majority of cases. These methods simulate the residence of a large image in the graphics memory whereas only a small part of this image is in fact available in the memory at a time. In this thesis a slightly adopted and a bit simplified version of virtual texture is implemented. Special focus is laid on the new feature of OpenGL introduced in of GLSL 4.2 (OpenGL 4.2) called images ([OP13]).

- Depicting of detailed elevation data causes the same problem as rendering very large textures, as in order to apply detailed elevation data, the mesh of the Earth must be very fine – all methods which only simulate vertex displacement like parallax mapping ([PM01]) are not suitable for this work as looking at the terrain surface at a small angle would reveal the simulation. The most popular approach in this research area is geometry clipmaps ([GC05]) which is an adaptation of simple clipmaps for generating very fine meshes in real time. In this thesis, the new function of OpenGL 4.2, namely the hardware tessellation is used in order to implement (slightly limited) alternative to classical approaches.

Figure 1.2 shows snapshots from the program written in course of working at this thesis and accomplishing both tasks: texturing and applying the elevation data. One detail in these images reveals that a level of detail technique is used here: the mount Popa (the green hill in the middle of the rather yellow area of the national park) is only visible in the snapshot at the bottom whereas all other images do not provide this (detailed) information. Note, that the shown elevations are very exaggerated in comparison with real landscape with the aim to make the surface details more evident.

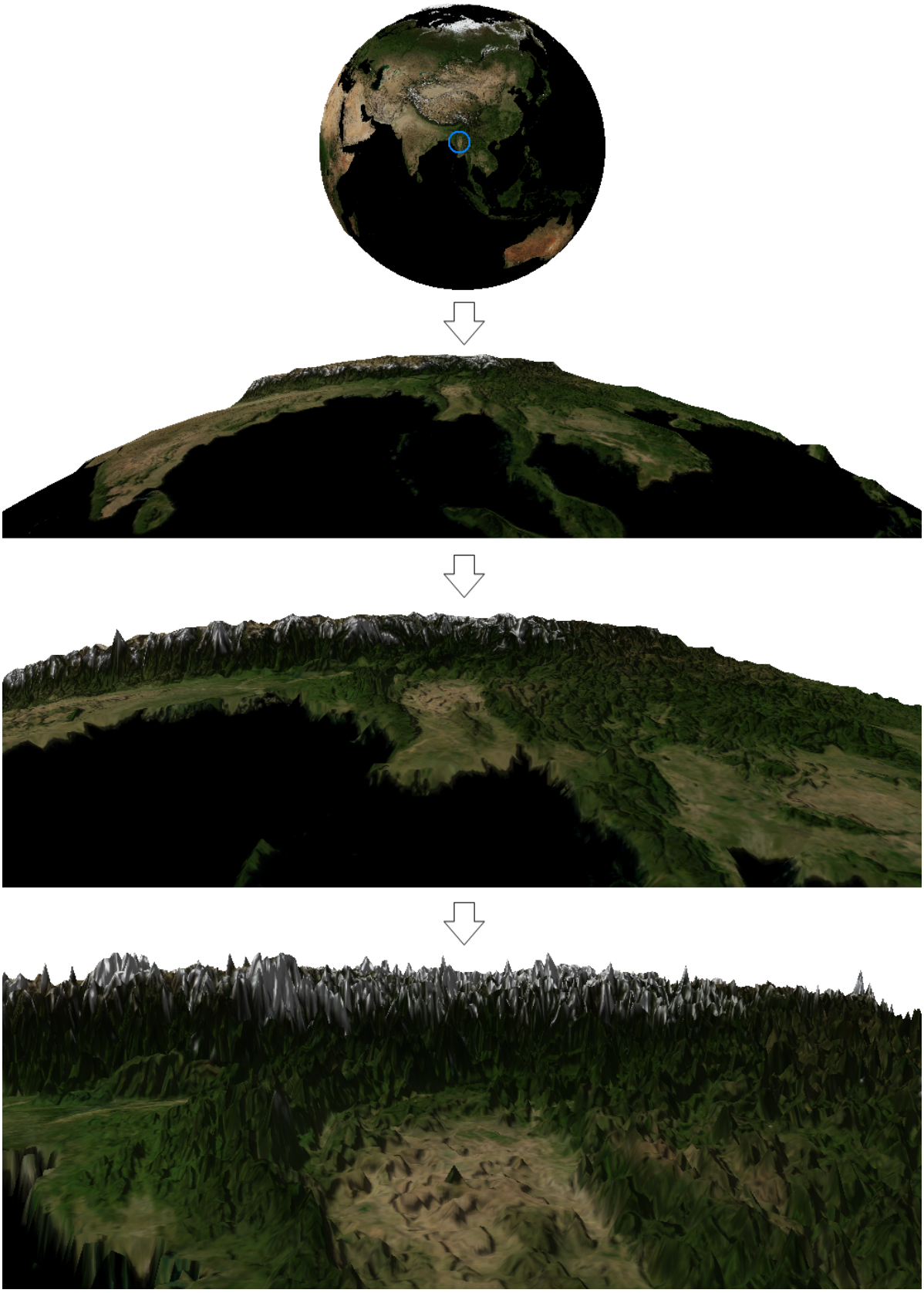


Figure 1.2.: Earth sphere at different zoom levels. The images represent selectively some snapshots of zooming in to Popa Mountain National Park in Burma (marked with the blue circle in the top image). Based on imagery and elevation data from ([BM04]).

1.4. New Approach

Additionally, this thesis discusses a new method of fine subdivision developed in course of this work and absolutely independently from any other approach. This method emulates vertex meshes which consist of as many vertices as many pixels the graphics window contains. This way, the resolution of the geometry is only limited through screen resolution (and of course through the level of detail of the given data). This fact gives the name to this method (and to the subtitle of this thesis): "Screen Resolution Depended Level of Detail" (see chapter 4).

1.5. Task Overview

To sum up, the aim of this thesis is to solve two tasks, texturing the globe and applying the elevation data to it whereas the second task subsumes the tessellation of the geometry. Figure 1.3 tries to visualize the relation between these tasks: for each frame, the software must texture the visible portion of the geometry, tessellate it to needed degree of fineness and apply the elevation data to it.

1.6. Assumed Knowledge

It is assumed that the reader is familiar with basics of computer graphics and OpenGL, at least with OpenGL versions prior to 4.0 as only the functionality of new features emerged in OpenGL since version 4.0 are explained to some extent in this thesis. Mipmapping is also mentioned in the text without any further explanations. The concepts of clipmapping and virtual texture are only shortly explained and geometry clipmapping is practically not discussed at all so further reading may be required for better understanding these topics.

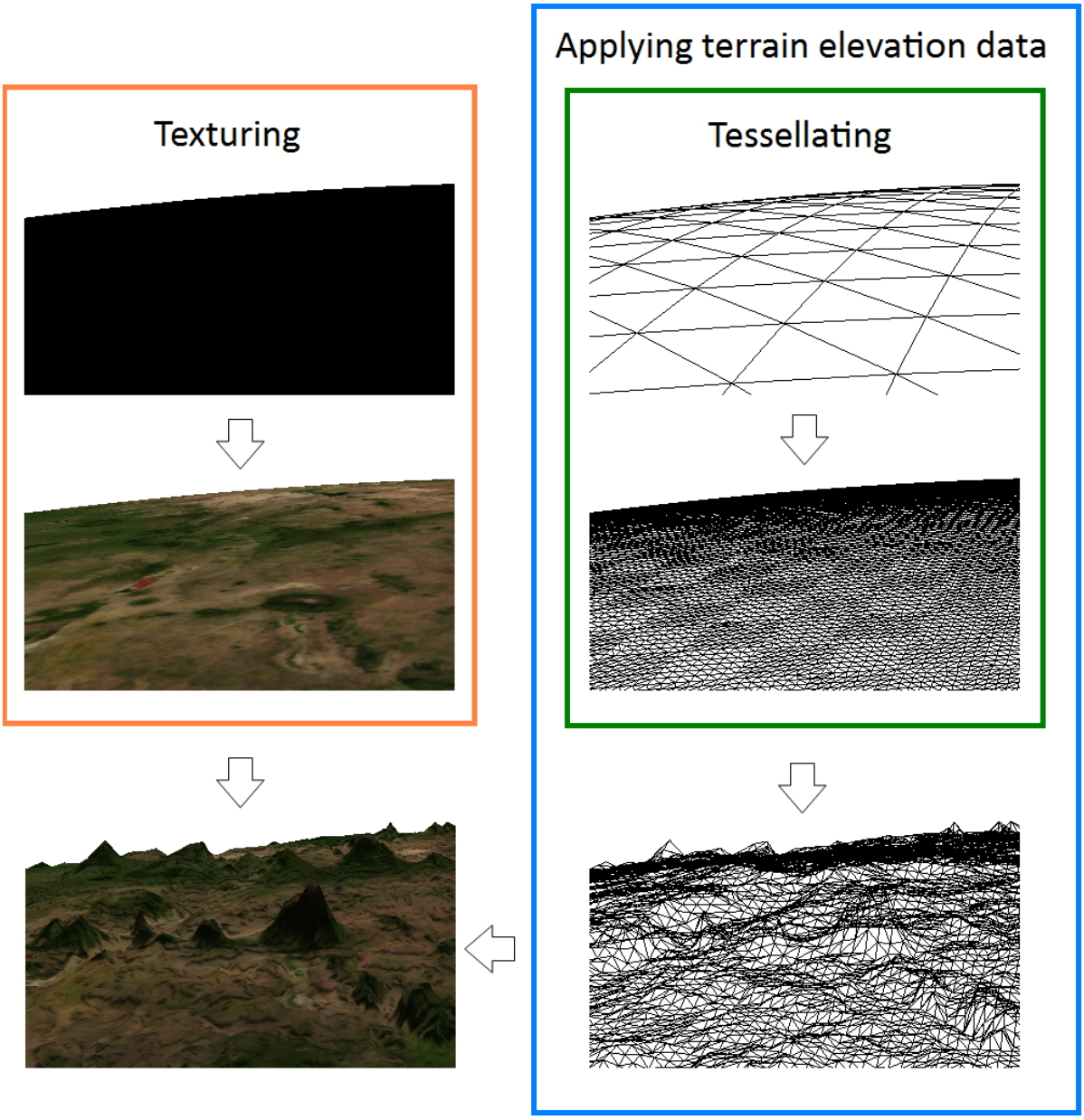


Figure 1.3.: Overview of the tasks in this thesis based on a example. For each frame the visible portion of geometry must be textured, tessellated and the elevation data must be applied to the tessellated result. Based on imagery and elevation data from ([BM04]).

1.7. Structure

In order to make this thesis as understandable as possible, instead of starting directly with the discussion of the visualization of the whole globe, it begins with explanation of how a simple quad consisting of four vertices can be textured with a big texture, then tessellation of a plane mesh is discussed and only at the end, based on this knowledge, the rendering of the Earth is presented.

On the whole, this thesis is divided into the following chapters:

1. **Introduction**
   is this very chapter.

2. **Managing Big Textures**
   explains with the help of a simple example how to simulate texturing a quad consisting of four vertices with a texture which size exceeds the memory of the video card.

3. **Visualizing Detailed Meshes**
   explains the basics of hardware tessellation in OpenGL on a simple plane mesh and shows how this technique can help to visualize detailed elevation data.

4. **Screen Resolution Dependent Tessellation**
   introduces the solution to the problem of large vertex meshes invented in course of working at this thesis.

5. **Rendering Earth Surface**
   explains finally how to apply all the knowledge from previous chapters to the actual task of this thesis, that is visualization of the Earth surface at high level of detail.

6. **Conclusion**
   discusses advantages and disadvantages of methods used in this workand tries to predict the future of these methods.

Appendix **A. Data** discusses shortly the data used for this thesis and how this data can be prepared and optimized.

2. Managing Big Textures

The problem of rendering huge textures using limited amount of graphics memory is not new and has been researched for years and many works have been devoted to solving it. This section discusses shortly the two most popular solutions to this problem existing nowadays as well as how the problem is solved in this work.

2.1. Related Work

Among all the existing approaches in this research area, two approaches deserve special attention as they are reputed to be classical ones and serve as a basis for most other researches. In first place it is so called clipmapping [CM98], firstly introduced in 1996 and serving as a reference for many works for more then a decade after its first publication. Second approach is called virtual texture (or "MegaTexture" if it is about id Software) and is the most modern approach researched among others by such well-known game developers as Crytek [AV08] or id Software [ID09].

These methods have in common that they try to imitate the residence of a large texture in graphics memory whereas only a small part of this texture is really available in memory at a time.

2.1.1. Clipmapping

Clipmapping is actually an extended version of the classical mipmapping ([MM83]), adopted for large textures which do not fit in memory of the video card.

Based on mipmapping, the clipmapping is the technique of displaying a texture at

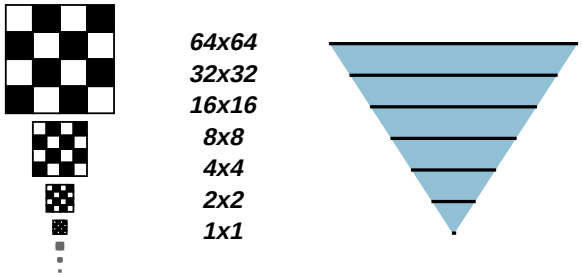


Figure 2.1.: Mipmap pyramid and its side view (from [CM98]).

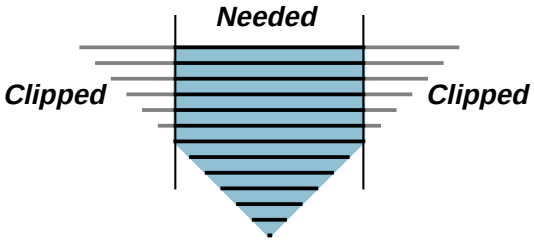


Figure 2.2.: Clipmap region within a mipmap pyramid (from [CM98]).

different levels of detail optionally including antialiasing ([MM83]). Like the original mipmapping, it uses the so called mipmap pyramid, a set of textures, including the original texture itself for maximal detail and its reduced copies for lower detail as shown in figure 2.1. These textures are called mipmaps. The difference to the classical mipmapping is, that the textures in this pyramid which largely overdimension the current screen resolution are clipped to some fixed size (see figure 2.2) in such a way that they become as small as possible but nevertheless can completely cover the screen. A sophisticated updating mechanism reloads then, based on viewer's position, dynamically the clipped textures when the camera moves in order to simulate the motion over a large texture.

Initially, this method demanded a specialized hardware, but with the advent of programmable GPUs some implementations also for common video cards were purposed (e.g. [NV07]).

The main bottleneck of the classical clipmapping is the incapacity of managing not flat objects which makes it inapplicable, for example, for rendering textured spheres.

2.1.2. Virtual Texture

Due to the shortcoming of clipmapping, namely the lack of ability to render not plane landscapes, an alternative method is needed for texturing arbitrary geometry with large textures.

One of the first publications in this direction might be [UT04]. In this paper, the main idea of virtual texturing is discussed. The second mentioning of this technique occurs in the interview with the id Software developer John Carmack ([MT06]) in which he reveals his ideas on this topic. This interview sparks interest in the theme but no concrete details about implementation is published until Sean Barrett presents his system at Game Developers Conference in 2008 and makes an implementation of it public ([ST08]). His work becomes a main reference for following works.

The idea of virtual texturing originates from the concept of virtual memory by operation systems. Similar to the concept of virtual memory in modern operating systems which makes it possible to access a large address space exceeding the actual size of the physical memory, the concept of virtual texture enables emulation of a big texture residing on a video card though the video memory would not be enough for it.

Whereas virtual memory extends the physical memory by storing data to the hard drive and loading needed parts of it on demand, the virtual texture extends the video memory of a video card by random-access memory and even (if RAM does not suffice) by hard drive.

The method for managing large textures implemented in course of this work might be called virtual texture as it meets the requirements defined above.

2.2. Basic Concept

The aim of this section is to show the functioning of the concept of virtual texture by means of an example. In this example the assumption is made that the video card can only hold a texture with the pixel resolution 80×80 pixel^2 but the texture that is needed to be rendered has the pixel resolution 320×320 pixel^2. As texture a smiley image shown in figure 2.3 is taken.

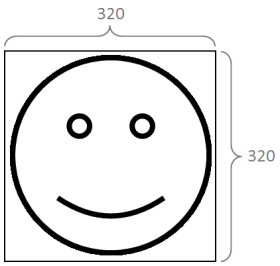


Figure 2.3.: Sample texture.

Thus, the problem to be solved is to emulate the residence of the texture with the pixel resolution 320×320 pixel^2 in the graphics memory whereas in effect available memory can only manage a texture with the pixel resolution 80×80 pixel^2 .

2.2.1. Mipmap Pyramid

As mentioned above, clipmapping and all similar techniques (like virtual texturing) use the mipmap pyramid. In the mipmap pyramid the n-th texture has half the width and half the height comparing to the $(n-1)$-th texture (assuming that the biggest texture has number 0). Concerning the smiley example, the reduced copies of the original image with the pixel resolution 320×320 pixel^2 must have pixel resolutions 160×160, 80×80 and so on. Though it is possible to reduce the original texture further, it is no need to do this as, according to the given example, textures with the pixel resolution 80×80 pixel^2 already fit in memory of the video card. Thus, the resulting mipmap pyramid consists of three textures which are shown in figure 2.4.

2.2.2. Main Idea

The idea of virtual texturing is to use the fact that the smaller visible area of a geometry becomes, the smaller texture is needed to cover it. According to this thought, if only a small part of a geometry is visible, the initial low pixel resolution texture can be replaced with a small part of texture with a better pixel resolution. Figure 2.5 visualizes this concept by means of an example: in the case that the quad or at least its mass is entirely visible in the window the lowest quality mipmap (which completely fits into graphics memory) should be rendered directly (top line in figure) and if only a small part of the

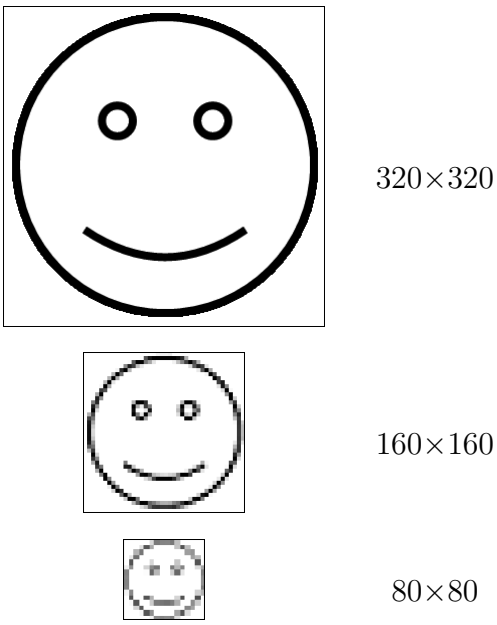


Figure 2.4.: Mipmap pyramid of the sample texture from figsure 2.3. (The differences in quality are exaggerated in order to make them more clear throughout this chapter.)

quad is visible the corresponding part of a texture with higher pixel resolution can be taken (bottom line in figure).

Generally, the rule for texturing a quad using concept of virtual texture comprises following steps:

1. find out which part of the quad is currently visible and determine how big this part is;
2. based on the size of the visible part choose the appropriate mipmap as well as which part of this mipmap is to use;
3. render the chosen texture section.

For didactic reasons, the first step is at first skipped in the following explanations and appears only towards the end of the chapter as it is the most extensive one. The following sections discuss steps 2 and 3 with the assumption that the step 1 is done.

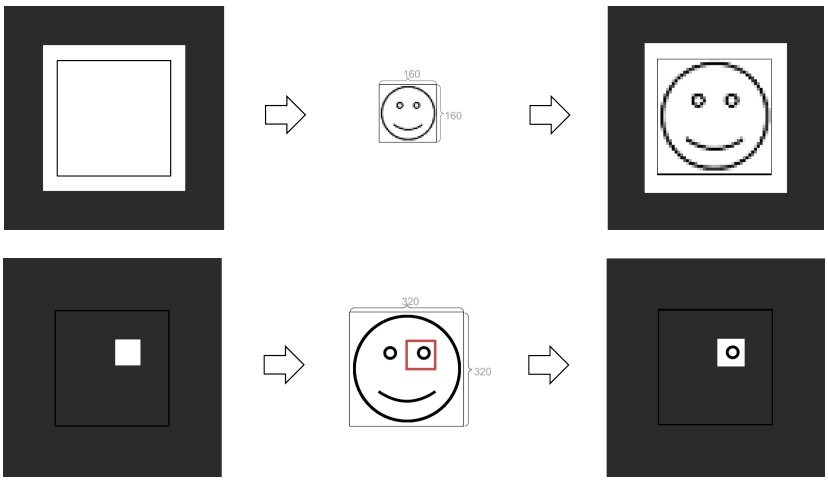


Figure 2.5.: Usage of the textures with different pixel resolutions depending on the size of the visible area of the quad: if the quad is visible entirely (above), the worst quality texture is chosen and if only a small part of the quad is visible (below), a texture with higher pixel resolution can be used.

2.3. Determining the Appropriate Texture Section

In order to determine which part of the quad is visible, it must be considered as consisting of smaller quads called **tiles**. The geometry itself do not need to be subdivided but the program should manage it as an array of small quads. Based on the size (measured in tiles) of this visible part, the appropriate mipmap and the suitable section of this texture can be chosen.

2.3.1. Subdividing in tiles

The number of tiles is not arbitrary but depends on the number of levels in the mipmap pyramid. If n is the number of levels and $\texttt{WIDTH}_{\texttt{tiles}}$ and $\texttt{HEIGHT}_{\texttt{tiles}}$ denote the width and height of the quad measured in tiles, the following formula can be used for determining the number of tiles:

$$\texttt{WIDTH}_{\texttt{tiles}} = \texttt{HEIGHT}_{\texttt{tiles}} = 2^{n-1} \tag{2.1}$$

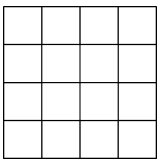


Figure 2.6.: Quad subdivided in 16 equal parts.

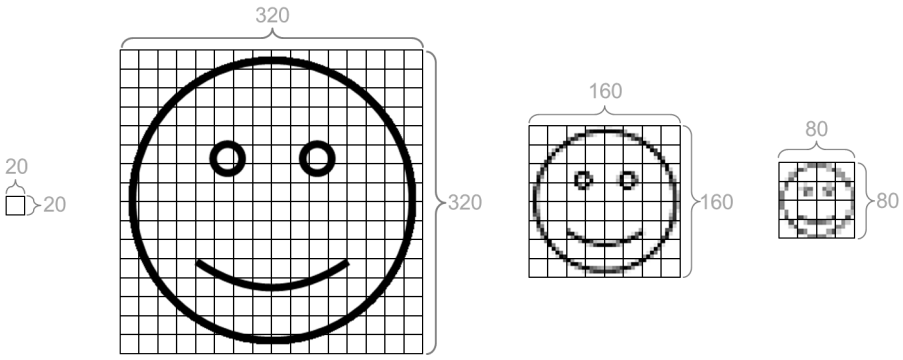


Figure 2.7.: All textures of the mipmap pyramid from figure 2.4 subdivided in tiles of the pixel resolution 20×20 pixel^2.

In the given example, the quad should be therefore subdivided in $2^2 \times 2^2 = 4 \times 4$ tiles as shown in figure 2.6.

The same subdivision must happen to the lowest resolution mipmap. Consequently, in the smiley example the the smallest texture must be subdivided in tiles with the pixel resolution 20×20 pixel^2. All other textures in the mipmap pyramid should be subdivided in tiles of the same size as well. Figure 2.7 shows all these subdivisions according to the smiley example.

2.3.2. Determining the Appropriate Mipmap

The subdivision of the quad in tiles makes it possible to determine which texture in the mipmap pyramid and which part of this texture is to take so that the choice is optimal. Firstly, a square area (called in the following **visible area**) of tiles which completely includes all the visible tiles, must be found and based on the size of this area an appropriate decision must be made. Measured in tiles, width and height of this area must be from the set $\left\{2^k \mid k \in \mathbb{N}_0\right\}$.

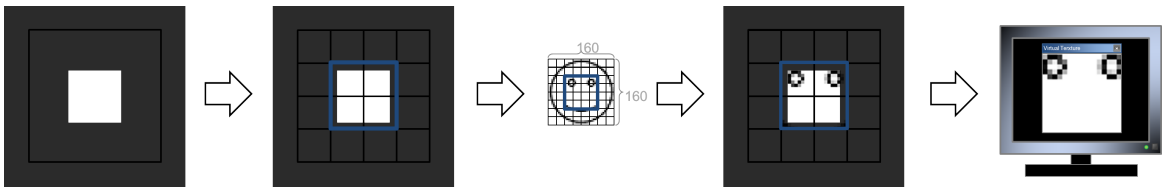


Figure 2.8.: Process of texturing the partly visible quad. The first image shows coarsely which part of the quad is visible in the graphic window; the second one shows more precisely which tiles specifically are visible. In the third image, the according part of the texture is chosen; the fourth image reveals the mapping of this part of the texture to the according (visible) part of the quad. Finally, the fifth image shows how the scene looks like on the screen of the monitor.

If $\texttt{width}_{\texttt{tiles}}$ and $\texttt{height}_{\texttt{tiles}}$ denote the size of this area then the following formula (which is in some kind the inverse of formula (2.1)) can be used to determine the level l in the mipmap pyramid:

$$l = \log_2(\texttt{width}_{\texttt{tiles}}) = \log_2(\texttt{height}_{\texttt{tiles}}) \tag{2.2}$$

2.3.3. Determining the Appropriate Texture Section

After determining the proper texture in the mipmap pyramid the appropriate section in this texture must be taken and mapped to the visible area: this section will always consist of as many texture tiles as many tiles the quad itself contains (in the given example of 4×4 tiles). Figure 2.8 shows an example of how it works: the visible part of the quad already forms a square area with $\texttt{width}_{\texttt{tiles}}$=$\texttt{height}_{\texttt{tiles}}$=2 and therefore the texture of the level $\log_2(2)$=1 (due to formula 2.2) from the mipmap pyramid must be taken and the middle section in this texture consisting of 4×4 tiles must be selected. Figure 2.9 shows another example in which the biggest texture is used.

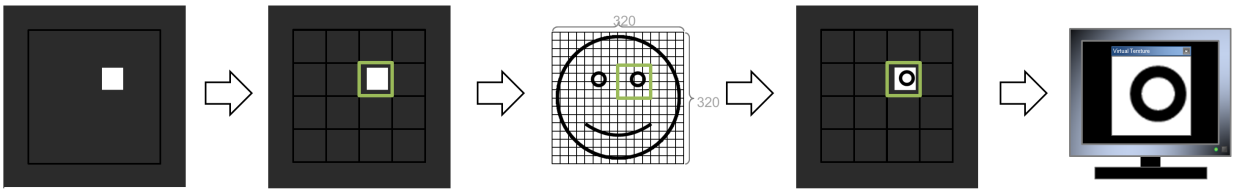


Figure 2.9.: Process of texturing the partly visible quad in the case that only one tile is visible: the biggest texture can be used in this case.

2.4. Rendering Chosen Texture Section

In this section a basic setup for rendering the texture sections (described in the previous section) with the help of OpenGL is presented. Also a problem which occurs with respect to texture coordinates is examined in detail.

2.4.1. Initializing Texture

As explained in the previous section, the size of the texture is for all zoom levels the same, merely the contents may vary. Therefore, the texture can be initialized with a single OpenGL command `glTexImage2D()` or, if a compressed texture (see appendix A, subsection A.3.1) is used, with its special variant `glCompressedTexImage2D()`. Sticking to the example from above, this command in Java with LWJGL can look like

```
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGBA_S3TC_DXT5_EXT,
    80, 80, 0, data);
```

This command initializes a DXT (see appendix A, subsection A.3.1) compressed 2D texture with the pixel resolution 80×80 pixel^2 with some blank data.

2.4.2. Managing Tiles

As in the concept of virtual texturing all textures consists of tiles which are handled independently, the textures must be first subdivided in the tiles as described above. In order to accelerate the starting of the program, the cutting of the texture in tiles can be

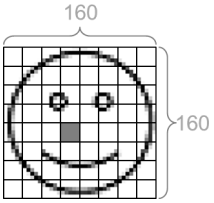


Figure 2.10.: The tile which corresponds to the array `tiles[1][3][4]`.

done in the preprocessing step. For this purposes a small script was written which reads a mipmap pyramid in DDS format (see appendix A, subsection A.3.1) and outputs a file consisting of the texture tiles in a concrete order. With the assumption that the whole mipmap pyramid fits in RAM, this file is read in the main program to a multidimensional byte array. More precisely, the array is 4-dimensional:

```
private static byte tiles[][][][]
    = new byte[NUMBER_OF_LEVELS][/*x*/][/*y*/][/*tile*/];
```

In this array, the first dimension represents the level in the mipmap pyramid, the second and third are used for coordinates of the tile on the texture and the last dimension represents the according tile as a sequence of bytes. In the example of the previous subsection, the array `tiles[1][3][4]` represents the tile in the middle texture of the mipmap pyramid which is marked with gray color in figure 2.10. As the first parameter is 1, the middle texture of the texture stack is chosen (as number 0 represents the biggest texture and number 2 the smallest), the parameter 3 and 4 mean that the fourth tile from left and fifth from above are taken. The contents of this array `tiles[1][3][4]` is a byte representation of this tile.

2.4.3. Rendering Tiles

Though it is possible to render all these small textures separately, a more convenient way is to combine them into one texture and render it with a single call. For this purpose, the OpenGL command `glTexSubImage2D()` or in the case of compressed textures its special variant `glCompressedTexSubImage2D()` can be used. For example in the case described in figure 2.9 the calls can look like

```
for(int y = 4,  y_offset=0;  y < 8;  y++, y_offset += 20){
    for(int x = 8, x_offset = 0; x < 16; x++, x_offset += 20){
```

```
        byte *data = getData(tiles[0][x][y]);
        glCompressedTexSubImage2D(GL_TEXTURE_2D, 0,
                                  x_offset, y_offset,
                                  20, 20,
                                  GL_COMPRESSED_RGBA_S3TC_DXT5_EXT, data);
    }
}
```

Here, according to figure 2.9 the tiles in the green square are read from the array `tiles` and mapped to the texture.

2.4.4. Defining Texture Coordinates

In order to map texture to a geometry correctly, texture coordinates must be defined. Normally texture coordinates are simply defined as vertex attributes for each vertex. But in the case the geometry is not a single quad but a fine vertex mesh, this approach can consume considerable portion of graphics memory. And in the scenario of hardware tessellation (discussed in the next chapter) it is absolutely impossible as the vertices are generated "on the fly". Thus, a more general solution is to calculate texture coordinates also "on the fly" in the shader program.

As in the example the geometry is a simple quad consisting of four vertices, the simplest way to provide them with texture coordinates in the shader program is to use their spatial coordinates. Defining the initial spatial coordinates as shown in figure 2.11 and then in course of rendering adding 1 to the x and y value and then dividing them by 2 results in texture coordinates in figure 2.12. As all spatial transformations happen (as usual) in shader program there is always a possibility to read the initial spatial coordinates (before these transformations) and to use their values in shader program.

2.4.5. Adopting Texture Coordinates

One detail must be kept in mind while rendering textures, namely the need of scaling and shifting texture coordinates in the case of using not ground level textures from the mipmap pyramid, i.e. not the smallest textures which completely fit in memory but a bigger one. For example, in the case depicted in figure 2.8, the resulting (combined)

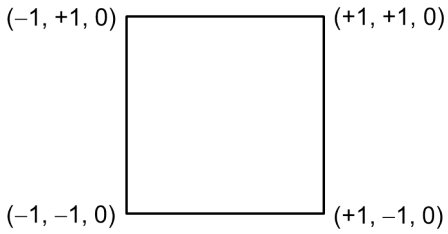


Figure 2.11.: Quad with initial spatial coordinates.

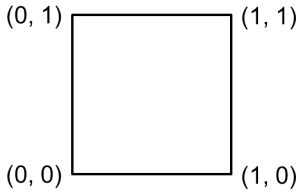


Figure 2.12.: Quad with texture coordinates.

texture is shown in figure 2.13 (left image) and exactly the same looks also the quad to which this texture is mapped if using the standard texture coordinates shown in figure 2.12. The problem is that actually only the middle part of this quad is seen on the monitor (and that is exactly the reason why the middle level texture of the texture pyramid is chosen) and this part is empty as the middle image in figure 2.13 symbolizes. Thus, using the initial texture coordinates leads to the empty graphics windows as shown in the right image in figure 2.13 and not to the wished result in the right image in figure 2.8.

In order to avoid this problem, the initial texture coordinates of the quad (as defined in figure 2.12) shown with respect to the given example in the left image in figure 2.14 must be altered in an appropriate manner. These texture coordinates must be somehow shifted to the middle as the middle image in figure 2.14 symbolizes. But the given geometry

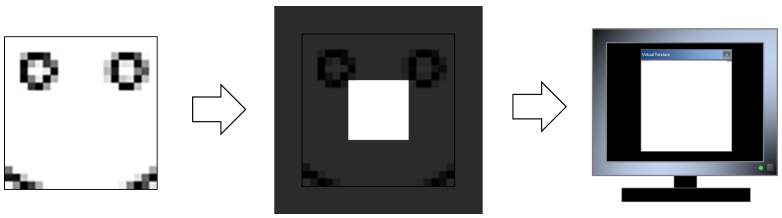


Figure 2.13.: Rendering of the partly seen quad without adopting texture coordinates.

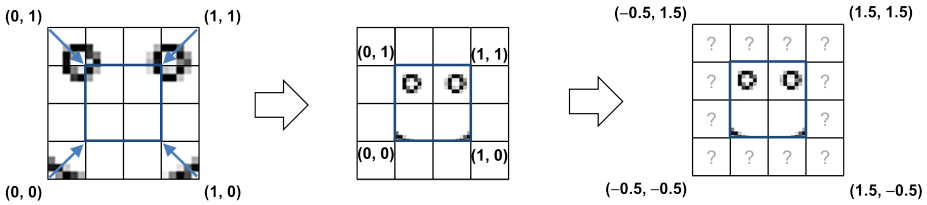


Figure 2.14.: Rendering of the partly seen quad with adopting texture coordinates.

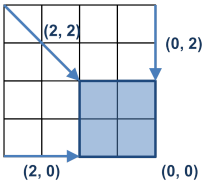


Figure 2.15.: Defining texture section as offsets to the edges of the original texture.

is just a quad consisting of four edge vertices providing no possibility of defining new texture coordinates in the middle of it. As the only vertices which can be altered are the edges of the quad, exactly their texture coordinates must be recalculated in the manner that the texture is resized in needed proportions. The right image in figure 2.14 proposes the texture coordinates of the edges which guarantee the wished behaviour. The tiles marked with question marks stay undefined but there is no need of concerning about them as they are not visible anyway.

As the texture coordinates are completely managed in shader program (see above), the corresponding shader must get the parameters which would allow it to calculate the texture coordinates. There are many ways to pass the needed information to shaders. As the partition of the square in tiles and choosing the appropriate tiles is of a discrete nature, the definition of the needed texture section can be done as the four offsets (measured in tiles) from each edge of the quad. As shown in figure 2.15, four pairs of numbers are enough to define this section. Another possibility is to describe the needed section as the offset from the upper right edge of the original texture as well as width and height (measured in tiles) of this section as figure 2.16 symbolizes. This second variant is chosen in this work.

Denoting the offset from the upper left corner with variables $\texttt{x0}_{\texttt{tiles}}$ and $\texttt{y0}_{\texttt{tiles}}$, the width and height of the visible part of the texture section as $\texttt{width}_{\texttt{tiles}}$ and $\texttt{height}_{\texttt{tiles}}$ and the width and height of the whole quad as $\texttt{WIDTH}_{\texttt{tiles}}$ and $\texttt{HEIGHT}_{\texttt{tiles}}$

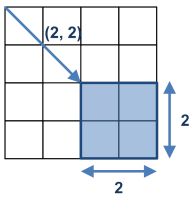


Figure 2.16.: Defining texture section as the offset to the upper right edge of the original texture as well as width and height (measured in tiles) of this section.

(anything measured in tiles), it is possible to recalculate the texture coordinates with the following formula:

$$\begin{aligned} s' &= \frac{\texttt{WIDTH}_{\texttt{tiles}}}{\texttt{width}_{\texttt{tiles}}} \cdot \left(x + \left(1 - \frac{\texttt{width}_{\texttt{tiles}}}{\texttt{WIDTH}_{\texttt{tiles}}} \right) - 2 \cdot \frac{\texttt{x0}_{\texttt{tiles}}}{\texttt{WIDTH}_{\texttt{tiles}}} \right) \\ t' &= \frac{\texttt{HEIGHT}_{\texttt{tiles}}}{\texttt{height}_{\texttt{tiles}}} \cdot \left(y + \left(1 - \frac{\texttt{height}_{\texttt{tiles}}}{\texttt{HEIGHT}_{\texttt{tiles}}} \right) - 2 \cdot \frac{\texttt{y0}_{\texttt{tiles}}}{\texttt{HEIGHT}_{\texttt{tiles}}} \right) \end{aligned} \tag{2.3}$$

The variables x and y are the real (not transformed) geometry coordinates of the vertices with the assumption that the geometry is just a quad from figure 2.11. As mentioned above, this calculation happens in shader program so the initial spatial coordinates must be reserved (with the help of in/out variables) before their transformation (e.g. in the vertex shader) and delivered to the further stages of the OpenGL rendering pipeline unchanged so that the subsequent shaders (e.g. the fragment shader) can access them.

The variables $\texttt{x0}_{\texttt{tiles}}$, $\texttt{y0}_{\texttt{tiles}}$, $\texttt{width}_{\texttt{tiles}}$ and $\texttt{height}_{\texttt{tiles}}$ must be transmitted to the shader program with the help of uniforms, e.g. with two `ivec` variables standing for coordinates of the upper left corner and size of the texture section (again, measured in tiles). $\texttt{WIDTH}_{\texttt{tiles}}$ and $\texttt{HEIGHT}_{\texttt{tiles}}$ are just constants (in the given example they are always equal four) so they can be just hard coded in shaders. The variables s' and t' are not mapped to $[0, 1]$ so, for using them with 2D textures, the following formula must be applied to them:

$$\begin{aligned} s &= 2 \cdot (s' + 1) \\ t &= 2 \cdot (t' + 1) \end{aligned} \tag{2.4}$$

Applying this formula, for example, to the situation depicted in figure 2.14 in fact leads to the texture coordinates shown in this figure on the right.

2.5. Determining the Visible Part of the Quad

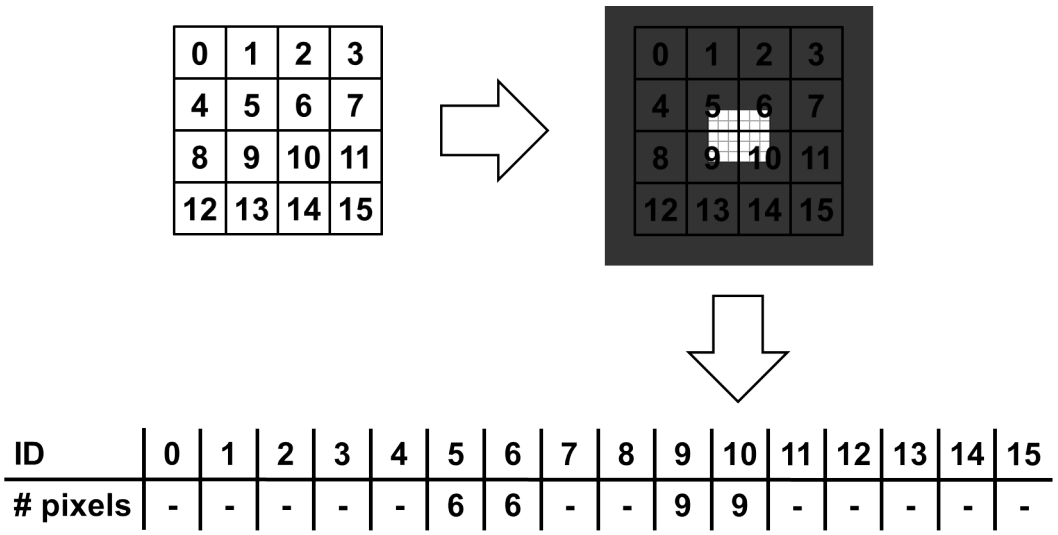
So far, the approach was discussed under the assumption that the needed information concerning the position and size of this section is known and just transmitted from the host program to the shader program. This section explains how the host program can gather this information before passing it to the appropriate shader program for rendering.

There are several approaches in determining the visible portion of a geometry. One classical method is to make all the needed calculation completely on the host side via implementing for example clipping to the frustum in the host program. This approach lacks for preciseness as due to the nature of OpenGL such operations in the rendering pipeline may differ strongly from what own methods do. Sean Barrett ([ST08]) proposes in his video presentation to prerender the complete scene and then to analyze the result of this rendering in order to determine which tiles are currently visible. In this work a similar approach is implemented.

2.5.1. Basic Idea

For the purpose of determining the visible tiles, these tiles must be somehow identifiable. The idea is to assign an ID-number to each tile of the quad, for example like in the upper left image in figure 2.17.

These ID-numbers must then be passed through the rendering pipeline to the fragment shader in the prerendering process so that each fragment "knows" to which tile it belongs. Rendering the scene to some kind of buffer makes it possible to read this information from this buffer back later and decide on its basis which tiles are visible. Figure 2.17 shows exemplarily this idea: in the case that zoom level is big enough so that only the middle part of the quad is visible on the screen and with assumption that current pixel resolution is just 6×5 pixel^2 (the subdivision in gray squares of the visible part in the upper right image), the table at the bottom shows which information is available after reading to the buffer, namely the knowledge that the tiles 5, 6, 9 and 10 are visible on the screen.



ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# pixels	-	-	-	-	-	6	6	-	-	9	9	-	-	-	-	-

Figure 2.17.: Detecting visible tiles. In this example the pixel resolution is only 6×5 pixel^2 and the visible fragments are marked with the gray mesh. At the bottom is the table mapping the tiles to the number of pixels on this tile which are visible on the screen. This table can be read from the rendering buffer and help to decide which tiles are visible.

2.5.2. Identifying Tiles in the Shader Program

Whereas the geometry in the given example is just a simple quad consisting of four vertices, a subdivided version of it must be used while prerendering which would allow the shader program to distinguish between tiles. In the case that the quad consists of 16 tiles the geometry must consist of 25 vertices equidistantly spread over the quad as shown in figure 2.18.

There are several ways to assign an ID-number to a tile. A trivial approach is to define ID-numbers as vertex attributes. But since most vertices in the mesh belong to more then one tile, such assignments are not unique: for example, the middle vertex in the upper left image in figure 2.17 connects the tiles 5, 6, 9 and 10 but can not represent all of them simultaneously. Therefore some efforts must be made in order to avoid this problem. One possibility is to duplicate the vertices, so that each vertex belongs to exactly one tile. This means especially that instead of the rendering modes `GL_TRIANGLE_STRIP/GL_QUAD_STRIP` the rendering modes `GL_TRIANGLES/GL_QUADS` must be used while rendering.

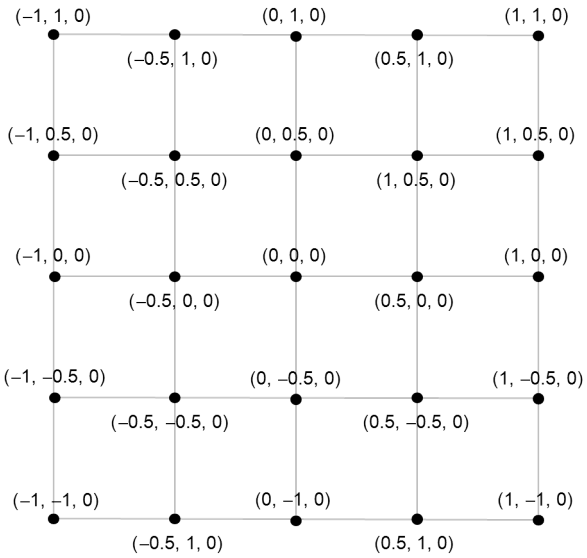


Figure 2.18.: Spatial coordinates of the vertices in the subdivided quad.

Another possibility is to assign some two-dimensional attribute to each vertex (or just use its initial spatial coordinates) and pass them through the rendering pipeline to the fragment shader which should then decide on the basis of these attributes to which tile the current fragment belongs. For example, if the initial spatial coordinates from figure 2.18 are chosen as attributes then reading in the fragment shader the (interpolated) value $(-0.7, 0.7, 0)$ would mean that the current fragment lies on the tile from the first row, second column.

But since OpenGL 4.2 it is no need to implement such methods as the more convenient way is to use the new feature of OpenGL 4 called **tessellation patches** ([OP13]). With this feature, defining the vertex-buffer object with rendering mode `GL_PATCHES` makes it possible to read the ID-number of the tile in the tessellation control shader which is prior to the fragment shader. Thus it is possible to provide each fragment with the ID-number of the tile to which it belongs.

2.5.3. Saving ID-numbers from the Shader Program to a Buffer

So far, in the fragment shader each fragment "knows" to which tile it belongs. If the scene is rendered to some kind of buffer instead of rendering to the screen, this information can then be analyzed in the host program and decision can be made which tiles are visible.

Barrett ([ST08]) proposes to render to the framebuffer, actually using the render-to-texture technique ([OP13]). The contents of such framebuffer is the array of all the pixels of the rendered scene. This framebuffer then can be read in the host program.

Coding the ID-number of the tile to which current fragment belongs as its color, it is possible to get this information in the host program. A big disadvantage of this method is that reading back of the framebuffer is extremely slow, especially taking into account the fact that the whole prerednered scene must be transmitted to the host program for determining a rather small list of visible tiles: rendering the quad subdivided in 16 tiles with the resolution 1024×1024 pixel^2 would require reading of 1048576 framebuffer elements in order to determine which of 16 tiles are visible. One possibility to avoid this problem is to reduce the size of the framebuffer thus reducing the pixel resolution of the prerendered scene compared to the screen resolution. But even reduction of height and width of the framebuffer to 1/4 of the screen size do not completely eliminate the latency though lowering the precision considerably. Another (and better) possibility it to reduce the size of the buffer directly on the GPU thus avoiding the expensive data exchange between GPU and RAM. [VC10] propose to use CUDA ([CU13]) for so called stream compaction ([SC09]) in order to reduce the size of the buffer, and in [VO10] solving the same problem with OpenCL ([OC13]) is discussed.

In this work no stream compaction is used. Instead, the task of compacting the whole scene to a short list of tile visibilities is solved via the new feature of GLSL 4.2 (OpenGL 4.2) called images. Images are just an alternative to samplers with the difference that they allow the programmer to use a buffer (in following called **image buffer**) for random read and write access in any shader.

The idea is not to save the complete scene in a buffer and then to analyze it on the host side. Instead, only a small one-dimensional image buffer consisting of as many elements as many tiles are in the quad should be used. Considering this buffer as array, the position (or index) of each element can be seen as the ID-number of the corresponding tile. The aim is that each fragment (which is a priori visible) writes to the corresponding position thus marking the tile to which it belongs as visible. With this in view, if all elements of the buffer are set to zero before the invocation of the fragment shader, then each fragment in the fragment shader needs only to increment the value of the element in the buffer at the position which is equal to the ID-number of the tile. Incrementing means that the shader reads a value from a position in the buffer, increments it by 1 and

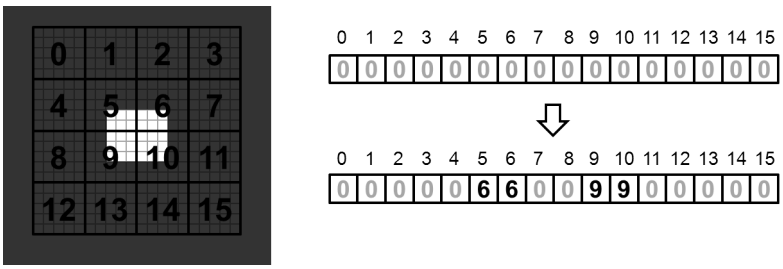


Figure 2.19.: Contents of the image buffer before and after the invocation of the fragment shader based on the situation in figure 2.17: visible fragments (6 fragments belonging to tile "5", 6 fragments from tile "6", 9 fragment from tiles "9" and "10", respectively) increase the values in the image buffer at corresponding positions and invisible ones leave the buffer unaffected.

then writes the result back to the same position. As the not visible fragments do not change the buffer, the not visible tiles preserve the value zero. Thus, the result of the fragment shader is the buffer in which the nonzero elements represent the visible tiles and the indices of such elements in the buffer represent the ID-number of the corresponding visible tiles. In figure 2.19 are the contents of the buffer visualized before and after the fragment shader invocation, based on the situation described in figure 2.17.

The problem of this method is exactly the incrementing of the values in the buffer because, as all visible fragments of a tile write to the same element in the buffer, this writing must be synchronized (e.g. with the help of atomic functions). But actually, as only the visibility of a tile itself is of interest and not the number of visible fragments belonging to it, instead of incrementing the corresponding element in the buffer, some nonzero number (e.g., "1") can be just assigned to this element. This way, a tile is considered visible as soon as at least one fragment belonging to it appears on the screen and this is actually the desired behaviour.

One remark is to be added at this place for the sake of completeness: the contents of the image buffer are not reset to zero after a rendering step but preserve the values from one invocation of a shader program to another. This leads to the problem that the tiles which were visible and then became not visible do not register this fact and stay marked as visible in the buffer. In order to avoid this, the buffer must be reset extra. This can be done in an additional rendering pass or even in the host program but the most optimal way is to implement this reset in the tessellation control shader. This shader is invoked on each tile and, as mentioned above, this shader knows the ID-numbers of

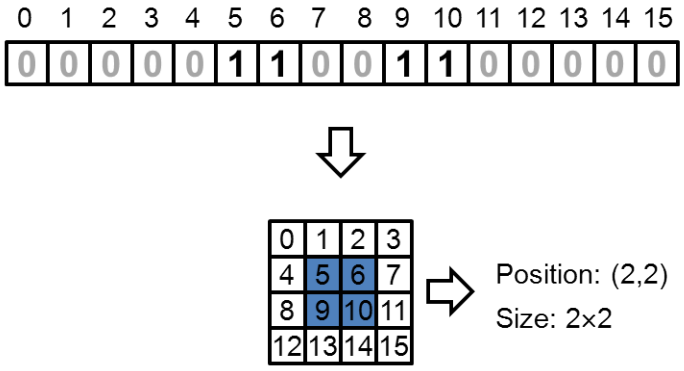


Figure 2.20.: The output buffer of the fragment shader from figure 2.19 (except that the sixes and nines are now ones) results in size and position of the texture selection (measured in tiles) is 2×2 and $(2, 2)$, respectively.

the tiles. So, letting it write zero at the corresponding position leads to the reset of the buffer.

2.5.4. Interpreting Buffer

So far, the result of the prerendering pass is just a one-dimensional buffer (or an array) of tiles in which the visibilities are marked with "1" and zeros symbolize currently not visible tiles. Based on this array, the host program should derive the position and size of the texture section as two two-dimensional parameters introduced in section 2.4 (subsection 2.4.5) and defined there as ($\texttt{width}_{\texttt{tiles}}$, $\texttt{height}_{\texttt{tiles}}$) and ($\texttt{x0}_{\texttt{tiles}}$, $\texttt{y0}_{\texttt{tiles}}$). For example, the analysis of the output array in figure 2.19 (except that the sixes and nines in the buffer are now ones) should obviously lead to the result that size and position of the texture selection (measured in tiles) is 2×2 and $(2, 2)$, respectively (see figure 2.20).

But not all cases are as obvious as the one in figure 2.20. In the case that the tiles "5", "9", "10" and "11" are visible, as visualized in figure 2.21 (left), the whole quad must be considered visible as the middle image in figure 2.21 symbolizes. This conclusion is derived from the fact, that the longest line of tiles in the given case consists of 3 tiles (tiles "9", "10" and "11" form the longest line of visible tiles) and therefore the final texture must cover at least 3×3 tiles. But due to the structure of the mipmap pyramid, in which each texture (except for the smallest one) has two times as big width and height as the next (coarser) texture in the pyramid, the width and height ($\texttt{width}_{\texttt{tiles}}$

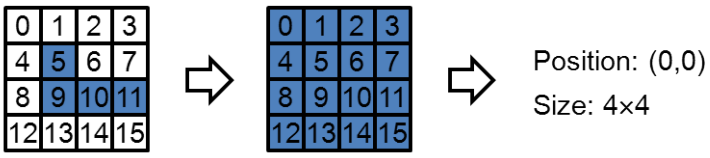


Figure 2.21.: Determining the width and height of the area that needs to be textured based on the visible tiles: the visible tiles "5", "9", "10" and "11" demand that the whole quad must be textured.

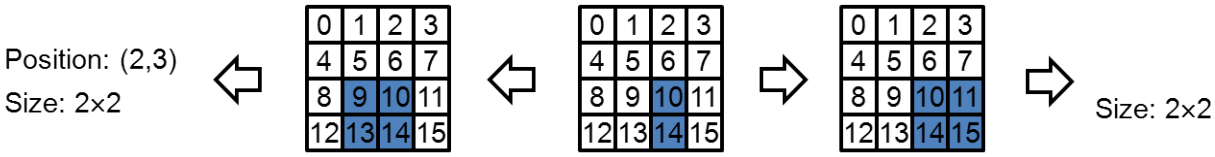


Figure 2.22.: Determining the width and height of the area that needs to be textured based on the visible tiles is not unique: if only the tiles "10" and "14" are visible then either of two areas "9"-"10"-"13"-"14" and "10"-"11"-"14"-"15" can be covered with texture.

and $\texttt{height}_{\texttt{tiles}}$) of visible quad area must be a power of two. Therefore the visible part of the quad must be considered as consisting of 4×4 tiles as the area of 2×2 tiles is not enough for covering the line "9"-"10"-"11". With other words: the whole quad must be textured in this case and hence the worst quality (the smallest) texture of the mipmap pyramid must be used for this. Figure 2.22 delivers another example, which shows that the choice of the visible area is not always unique: if only the tiles "10" and "14" are visible then either of two areas "9"-"10"-"13"-"14" and "10"-"11"-"14"-"15" can be covered with texture resulting in the same visual effect (as the tiles "9", "13", "11" and "15" are not seen on the screen anyway).

Generally speaking, the rule for finding the visible area as defined in section 2.3 can be defined as following: Find a square (consisting of tiles) on the quad which

- includes all the visible tiles,
- has height and width (measured in tiles) which is a power of two,
- has as small area (measured in tiles) as possible and
- is preferably centred around the visible tiles.

Formally, the parameters ($\texttt{width}_{\texttt{tiles}}$, $\texttt{height}_{\texttt{tiles}}$) and $\texttt{x0}_{\texttt{tiles}}$, $\texttt{y0}_{\texttt{tiles}}$) can be calculated using the following formulas:

$$\begin{aligned} \texttt{width}_{\texttt{tiles}} &= \texttt{height}_{\texttt{tiles}} \\ &= \texttt{round2}\left(\max\left\{\left(\texttt{max\_x} - \texttt{min\_x}\right), \left(\texttt{max\_y} - \texttt{min\_y}\right)\right\}\right). \end{aligned} \tag{2.5}$$

$$\begin{aligned} \texttt{x0}_{\texttt{tiles}} &= \texttt{min\_x} - \left\lfloor \frac{\texttt{width}_{\texttt{tiles}} - (\texttt{max\_x} - \texttt{min\_x} + 1)}{2} \right\rfloor \\ \texttt{y0}_{\texttt{tiles}} &= \texttt{min\_y} - \left\lfloor \frac{\texttt{height}_{\texttt{tiles}} - (\texttt{max\_y} - \texttt{min\_y} + 1)}{2} \right\rfloor \end{aligned} \tag{2.6}$$

Formula (2.5) uses the function `round2` which rounds a number to the next highest power of 2. It is defined in formula (2.7). In formula (2.6) the mean value between $\texttt{width}_{\texttt{tiles}}$ and the real size (in tiles) of visible part of the quad is calculated in order to centre the texture around the midpoint of the visible part. Figure 2.23 shows an example of applying this formula to a quad subdivided in 8×8 tiles: visible area, marked in figure with dark blue color becomes the area marked with light red color. The result is not unique as shifting this area one unit upwards would also meet the requirements. But this fact is not really disturbing as the not unique tiles are not visible anyway.

$$\texttt{round2}(x) := \min\left\{n \in \left\{2^k \mid k \in \mathbb{N}_0\right\} \mid n \geq x\right\}. \tag{2.7}$$

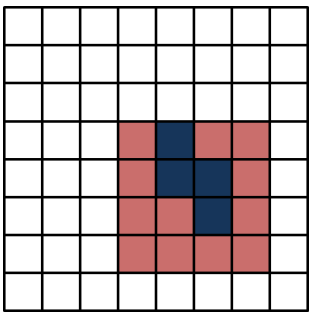


Figure 2.23.: Formula (2.6) tries to centre the texture around the visible part of the quad: applying it to the visible area marked in figure with dark blue color results in the area marked with light red color (including, of course, the dark blue tiles).

2.6. Smooth Switching between Levels

As described above, virtual texture techniques try to map a texture section with the best possible pixel resolution to a geometry so that during zooming, if the system "notices" that a switch from one texture resolution to another is needed, it makes this switch. The problem is that if this switch is made immediately, it becomes apparent to the viewer. This is because the resolutions of these textures differ by a factor of 4 (they are the neighbouring mipmaps in the mipmap pyramid). The problem is, that at a certain point, the virtual texture system makes a "jump" from one resolution to another immediately. Figure 2.24 visualizes this problem based on the situation described in figure 2.8 at the moment of changing the level of detail: it shows that the contents of the screen change considerably.

A solution to the problem is symbolizes in figure 2.25: the program must somehow blend these textures in order to avoid this behaviour.

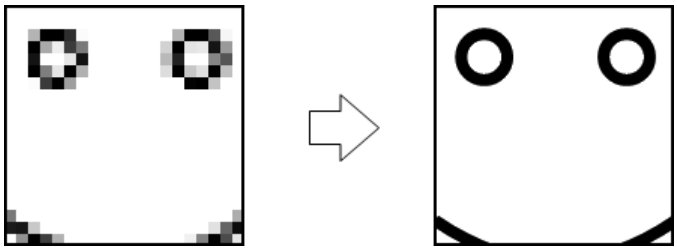


Figure 2.24.: Switching from one level of detail to another. It can be very apparent to the viewer.

Figure 2.25.: Zooming in with blending.

Figure 2.26.: Blending mipmaps when the quad is depicted in perspective.

As the quad can be viewed at different angles, it is also wished that blending occurs in the case of viewing the quad in perspective. In this case, the nearer part of the quad should be textured with a high resolution texture and the farther one with a less detailed one and the middle part should be a result of blending between the both. Figure 2.26 visualizes this idea with the help of the smiley example: the distant part of the texture is less detailed than the near part.

Luckily, there is no need to solve the given problem singly: the solution has been existing for years and is called trilinear filtering (its enhanced version is called anisotropic filtering). The functionality of this concept is based on mipmapping: trilinear filtering (as well as anisotropic filtering) performs linear interpolation not only between texels of a texture but also between mipmaps. This technique is natively supported in OpenGL.

Trilinear filtering assumes the availability of the mipmap pyramid in order to interpolate between its textures. This technique makes only sense if at least two neighboring mipmaps are present in the graphics memory which enables the filter to interpolate between them. In the example in figure 2.25 both textures must be available in the memory if blending is demanded.

With these thoughts in mind, the virtual texture system described in course of this chapter must be slightly extended: besides the mipmap (section) demanded for covering

the currently visible area, the corresponding mipmap section with smaller resolution should be also copied to the graphics memory for the purposes of blending. Exception is, of course, the smallest mipmap which can not be blended but as it is used only in the cases that the geometry is far away from the viewer this does not cause any problems.

3. Visualizing Detailed Meshes

After the discussion of texturing a square plane with a very large texture which does not fit into graphics memory this section deals with rendering of very fine vertex meshes in order to simulate detailed terrain on the Earth surface.

3.1. Overview

The classical approach to manage geometries, which is used normally for rendering geometries consisting of small number of vertices, just copies all the vertices directly to the graphics card. For this work, this approach can not be applied: As mentioned in the introduction chapter, in this work the height map data from the Blue Marble Next Generation Projection with the pixel resolution 86400×43200 pixel^2 is used for getting elevation information. Creating a vertex mesh which completely covers this height map or, in other words, has as many vertices as many pixels there are in the image, would require 86400×43200×3×4B≈45GB of graphics memory for managing it as each point is tree-dimensional and each dimension is managed by OpenGL as a 4 bytes float value. No optimization or compression method can help to compact this data so that it fits into graphics memory. Thus, similar to managing big textures, large and detailed vertex meshes also need to be imitated with the help of some level of detail technique.

Geometry clipmaps ([GC05]) might be considered the most famous approach in this area. The idea of this concept is to adopt the original clipmaps ([CM98]) for vertex meshes.

All methods which only simulate vertex displacement like parallax mapping ([PM01]) are not suitable for this work as looking at the terrain surface at a small angle would reveal the simulation.

Since version 4.2, OpenGL provides a mechanism of subdividing vertex meshes "on the fly" on GPU, simulating this way a more detailed geometry than actually initially available. This technique is called tessellation ([OP13]). This section discusses the usage of this technique for simulating large and detailed vertex meshes.

3.2. Definition

The aim of tessellation is to subdivide a primitive in needed number of primitives. Two kinds of primitives are supported with respect to tessellation in OpenGL, namely quads and triangles. Primitives are subdivided in smaller primitives of the same art: quads are subdivided in quads and triangles in triangles. Figure 3.1 shows an example tessellation of a triangle. As this section deals with the tessellation of a square plane mesh, it suggests itself to use only quad tessellation in the following.

The subdivision through tessellation is not unlimited: the maximal level of tessellation (maximal number of quads in each line of the resulting mesh) is limited to 64 in current implementations of OpenGL. Thus OpenGL is capable of transforming a single quad to a mesh consisting of 64^2=4096 quads as figure 3.2 visualizes. But even with this limitation the huge data of 45GB mentioned above can be reduced to 45GB÷4096=11MB which can be defined and stored on the GPU. This mesh, of course, should be tessellated only partly (for example with a level of detail technique), otherwise a memory overrun is inevitable.

The aim of each level of detail technique based on tessellation is similar to that of virtual texture: whereas virtual texture tries to texture only visible portion of geometry with the texture of the best possible quality, tessellation subdivides only the part of geometry which is close enough to the viewer and the closer it is the bigger tessellation level

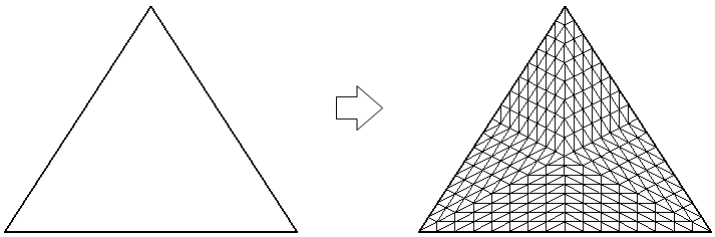


Figure 3.1.: Tessellation of a triangle.

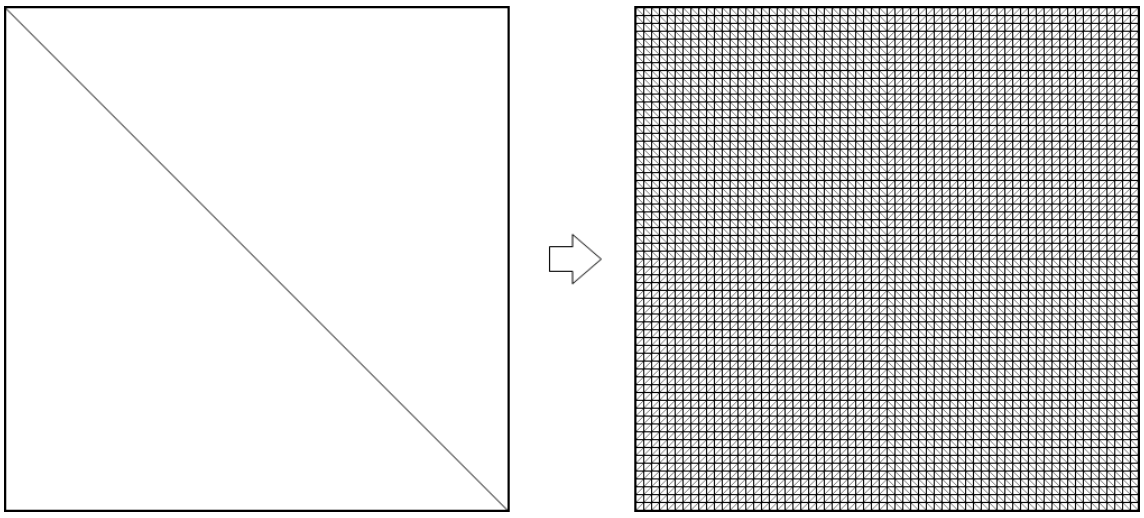


Figure 3.2.: Maximal tessellation of a quad: 1 quad is subdivided into 64^2=4096 quads.

becomes.

Due to the limitation that the maximum tessellation level can be 64, it is not possible to create a really large detailed mesh just based on a single quad. Therefore a quite coarse mesh must be defined and stored to the GPU and then the tessellation should be used in order to make the parts of this mesh which are close enough to the viewer finer.

3.3. Controlling Tessellation

The tessellation algorithm itself is hidden by the OpenGL implementation. All a programmer needs is to define a mesh of vertices and six parameters which determine the level of tessellation. Four of this parameters called “outer level 0”, ..., “outer level 3” affect the subdivision of each quad side. Two of them called “inner level 0” and “inner level 1” determine the level of tessellation in inner area of the quad: “inner level 0” is responsible for subdividing of the inner area horizontally and ”inner level 1” determines its vertical subdivision (see figure 3.3). Each level specifies the number of parts in which a corresponding side or inner area must be subdivided. In following the influence of these parameters on tessellation is discussed with the help of examples. For the sake of clarity only the outer levels of tessellation are explained at first and afterwords the combination of both is shown.

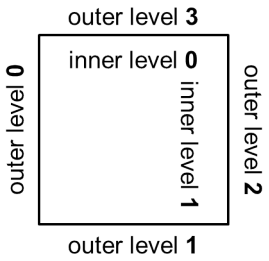


Figure 3.3.: Tessellation scheme.

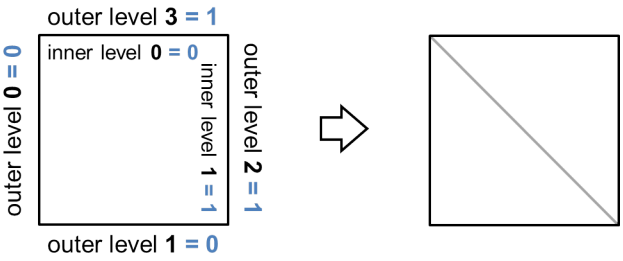


Figure 3.4.: Trivial tessellation with all levels equal 0 or 1: the quad is not subdivided at all.

3.3.1. Outer Levels

Figure 3.4 shows a trivial example of setting outer level to 0 or 1. As subdividing in 0 or 1 parts means no subdivision at all, setting all levels to 0 or 1 leave the quad unchanged. OpenGL merely triangulates the quad which is symbolized with a gray diagonal line in the right image.

Setting outer level to 2 divides the corresponding quad side in half. Figure 3.5 visualizes such a tessellation.

Figure 3.6 shows an example with outer levels set to four different levels.

3.3.2. Inner in Combination with Outer Levels

Figure 3.7 shows an example of how inner and outer levels are combined while tessellating: in the top left image the subdivision markers are determined based on subdivision of the quad vertically and horizontally due to the inner levels; in the top right image the

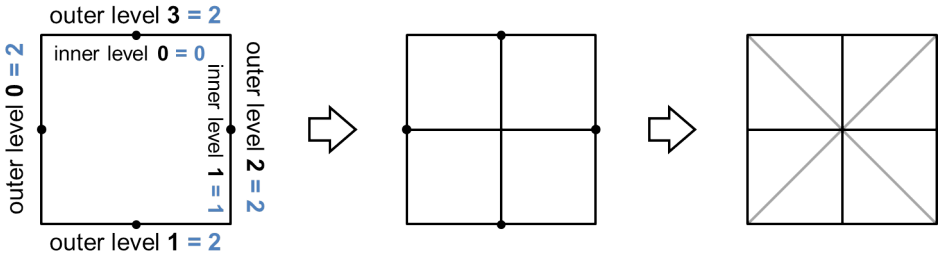


Figure 3.5.: Tessellation with all outer levels set to 2 divides all four sides of the quad in half.

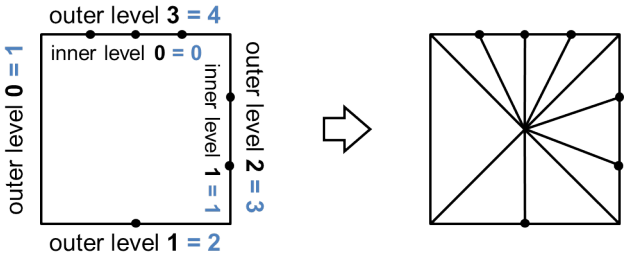


Figure 3.6.: Example of tessellation with different outer levels for each side of the quad.

subdivision markers for each side are determined; in the top middle image the markers are combined.

Setting all levels to 0 brings OpenGL to ignore the corresponding quad. This feature is very advantageous with respect to level of detail as it makes it possible to exclude not visible quads from rendering already in early stages of the rendering pipeline.

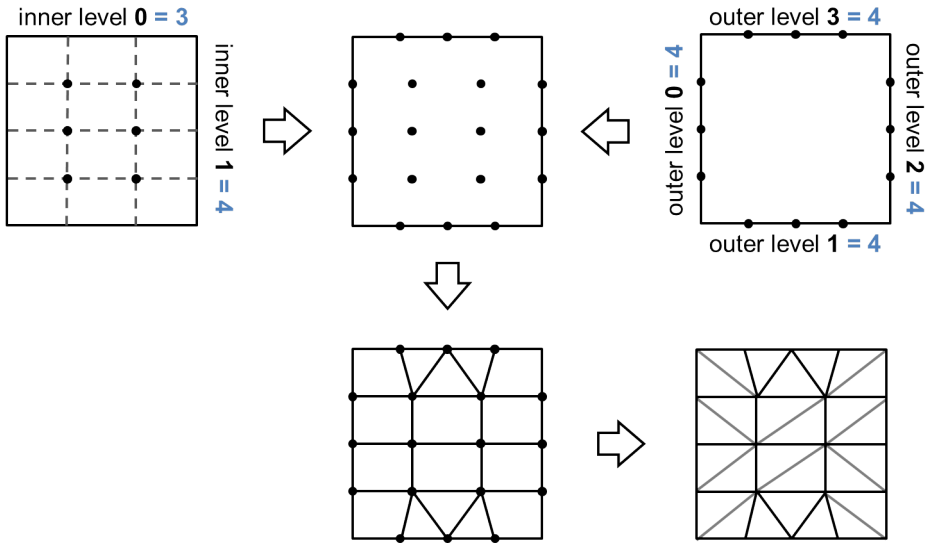


Figure 3.7.: Combination of inner and outer levels: in top left image the subdivision markers are determined based on subdivision of the quad vertically and horizontally due to inner levels; in top right image the subdivision markers for each side are determined; in top middle image the markers are combined.

3.4. Level of Detail via Tessellation

There are different strategies of managing geometrical level of detail of a mesh via tessellation. In this section some of them are discussed with the help of the example mesh shown in figure 3.8.

An obvious way to manage level of detail via tessellation is to use in some manner the distance between the rendered mesh and the camera: the near object to the camera the more it (or a part of it) must be tessellated.

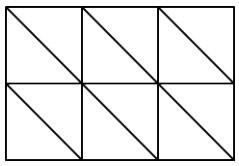


Figure 3.8.: Sample mesh.

3.4.1. Tessellation Based on the Distance between the Camera and the Object

The most naive approach in this respect is just to measure the distance between camera and the whole mesh (e.g. distance to the centre of gravity). Based on this distance a (uniform) proper level of tessellation should be chosen for the whole mesh. But this method is obviously inappropriate for rendering really huge meshes: in the case the mesh comes too close to camera and consequently the tessellation level (of each quad) becomes big, too many vertices are generated. This leads to graphics memory overflow and consequently to crush of the graphics driver. Thus only approaches which tessellate only part of a geometry deserve to be considered for the purposes of this work.

3.4.2. Tessellation Based on the Distance between the Camera and the Tile

An improved method of tessellation which regards partial tessellation of a mesh is based on measuring the distance between the camera and each quad in the mesh. Thus, it is possible to avoid that all quads are tessellated at the same level and the problem of memory overflow does not occur. Figure 3.9 visualizes an example for this method based on the sample mesh from figure 3.8. The camera icon at the bottom in the left image symbolizes that the mesh is viewed from the bottom so that the middle bottom quad in the mesh is the closed to the camera and the top line of quads in the mesh are the farthest. The distance to each quad must be converted to an appropriate tessellation level. There is no sense to specify any formula making such a calculation precisely at this point as it depends to a considerable degree on the scale of the scene – but it is obvious that this tessellation level must be somehow inversely proportional to the distance. In the middle image in figure 3.9 is assumed that the distances are converted to the levels written over the quads: the closest quad has level 3 and the farthest quads have level 1. Actually there are 6 tessellation levels for each quad as mentioned above but, as each quad in this method considered independently from others and get a unique tessellation level, all inner and outer levels are just assigned the same value for each quad, respectively. This results in the tessellation shown in the right image in figure 3.9.

This art of tessellating causes a problem known as T-junctions ([VG11]).

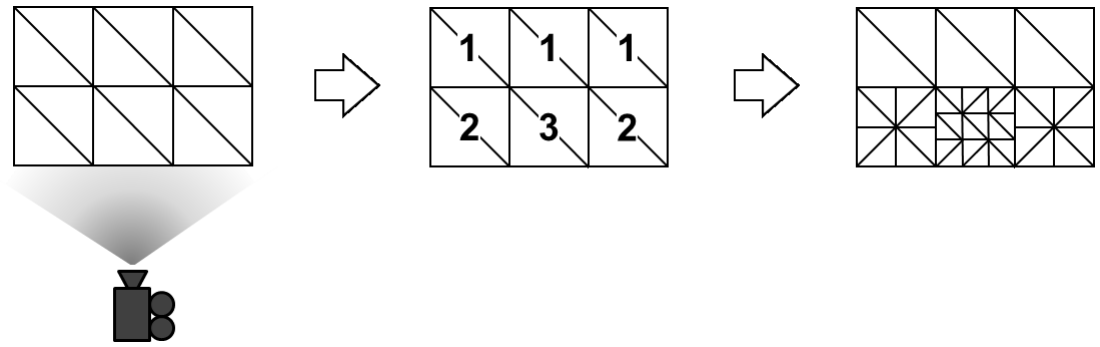


Figure 3.9.: Tessellating based on the distances from each quad to a camera: the left image visualizes that the mesh is viewed from bottom; the middle image shows an exemplary assigned tessellation levels based on the distances to the camera (the closest quad has level 3 and the farthest quads have level 1); the right image shows the resulting tessellation.

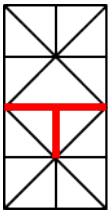


Figure 3.10.: An example of a T-junction: the middle horizontal line is the common border of both quads and the lower quad subdivides it in two parts whereas the upper one does not subdivide it at all.

3.4.3. T-junctions

T-junctions occur when a line belonging to two different quads is subdivided in one quad not in as many parts as in the other one or, in terms of OpenGL, the corresponding outer levels of two adjacent quads are not equal. An example of such a T-junction is shown in figure 3.10 in which the T-junction is marked with bold red lines: here the middle horizontal line is the common border of both quads and the lower quad subdivides it in two parts whereas the upper one does not subdivide it at all. Exactly this T shape shown in figure 3.10 is responsible for the name of this phenomenon. Figure 3.12 shows all spots in tessellated mesh from figure 3.9 on which T-junctions occur.

The negative effect of T-junctions becomes apparent if a height map is applied to the mesh which results in displacement of vertices: moving of a vertex causing T-junction

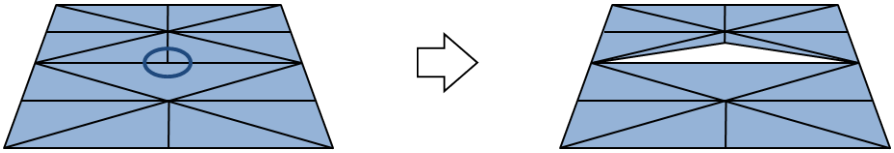


Figure 3.11.: An example of how T-junctions can cause cracks in the mesh. In the left image the problem vertex is marked and the right image shows the crack which appears as result of setting the vertex in the middle higher then the rest of the plane.

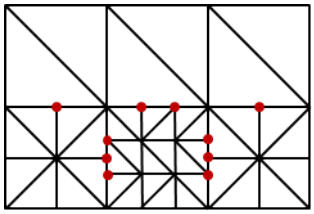


Figure 3.12.: All T-junctions in the tessellated mesh from from figure 3.9 (marked with red bold dots).

leads to a crack. Figure 3.11 shows this effect on the quads from figure 3.10 (flipped vertically and in perspective): in left image the problem vertex is marked and the right image shows the crack which appears as result of setting this vertex higher then the rest of the plane.

Figure 3.12 shows all T-junctions which occur in figure 3.9.

3.4.4. Tessellation Based on the Distance between the Camera and Tile Sides

The simplest solution of this problem is to tessellate sides of all quads in such a manner that for each two adjacent quads the common[1] edge is tessellated at the same level quasi "from both sides". Figure 3.13 visualizes this solution applied to the case in figure 3.11: after setting the tessellation level for the upper edge of the lower quad to 2 no crack appears if moving the middle point higher.

As OpenGL process all quads of a mesh independently there is no plausible possibility to

[1]The term "common" is not quite appropriate as OpenGL manages the quads independently and two quads do not share any edges but nevertheless the described edges lie in each other so that it can be assumed that they behave the same.

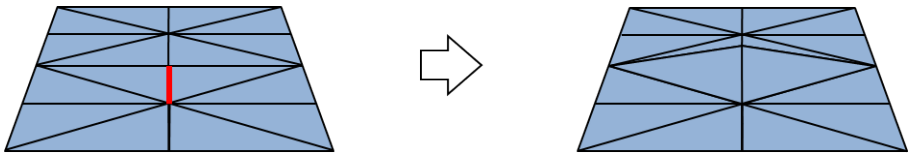


Figure 3.13.: Solving the problem of T-junction from figure 3.11 via tessellating the edge between both quads consistently at the same level "from each side".

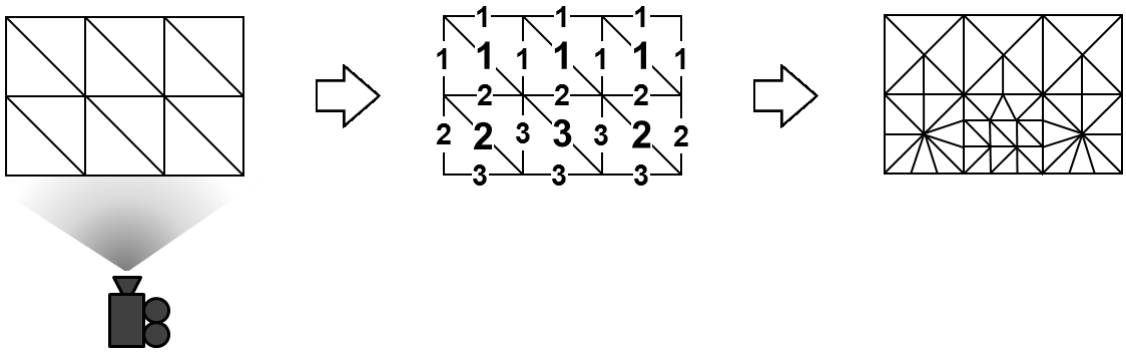


Figure 3.14.: Determining tessellation levels for each edge helps to avoid T-junctions.

analyze for each quad the adjacent quads in order to choose the appropriate tessellation level for all its sides: if a quad is processed, there is no guaranty that all its neighbors have already been processed. But as all the edges lies practically in each other it makes sense to target not the quads but the edges themselves in order to chose appropriate tessellation level for them. It can be done, for example, by calculating distance from camera to midpoints of edges. The middle image in figure 3.14 shows an example of different outer tessellation levels (denoted with smaller numbers on edges) based on distances to corresponding edges. The right image shows the resulting mesh in which no T-junction occurs. It should be noted that the distances from the camera to quads (denoted with bigger numbers in the middle of each quad in the middle image in figure 3.14) must be also considered for determining the inner tessellation levels.

3.4.5. Popping

The last proposed solution is suitable for static scenes but has a little negative aspect if the mesh changes its position with regard to the camera dynamically. The problem is that if quad changes its distance to camera the changes of the tessellation levels may cause the effect called popping ([VG11]). Figures 3.15 and 3.16 try to explain this

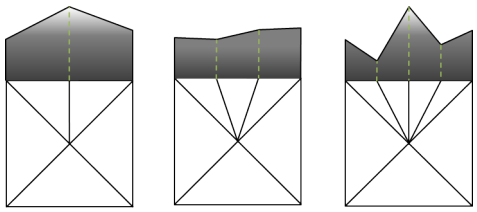


Figure 3.15.: Tessellation of a quad edge at levels 2, 3 and 4 and side view representation of the rendered height map applied to the vertices on this edge.

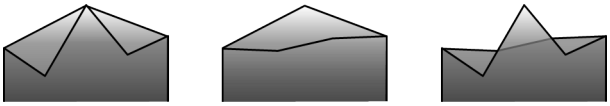


Figure 3.16.: All 3 possible transitions between all tessellation levels in figure 3.15.

phenomenon. Figure 3.15 shows a quad with three different tessellation levels (namely 2, 3 and 4) for its top edge and a schematic representations (side views) of these edges after applying a height map to them. The height map is the same for all 3 variants but due to the different number of vertices on the edge in each case the picture is different. Figure 3.16 shows schematically all 3 possible transitions between all tessellation levels in figure 3.15. The left image visualizes the transition between levels 2 and 4: in this case the two minima of the level 4 are added if changing the tessellation from level 2 to level 4 and removed if changing it in reverse order whereas the maximum stays unchanged. The middle and left images show transition between tessellation level 3 and level 4 as well as between level 3 and 2: these transitions are not smooth as they change the picture completely.

The conclusion from these observations is that only the transition between the tessellation levels 2 and 4 is smooth enough as it only adds or removes details to the vertex mesh. Generally, for avoiding popping, the tessellation levels must be rounded up to the next power of 2. In other words, formula (2.7) must be applied to each tessellation level. In this example, only an edge of a quad was considered but the same is true for all its edges.

Adopting the last consideration to the example in figure 3.14 leads to the mesh in figure 3.17.

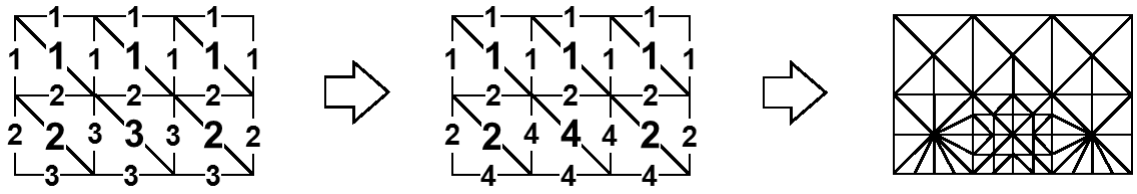


Figure 3.17.: Applying formula (2.7) to each tessellation level makes it robust against popping.

3.5. Correlation between the Pixel Resolution of the Texture and the Mesh Fineness

Although the tessellation method described above is quite reliable and gives appropriate results, it is not completely satisfying in respect to the current work. Optimally, geometric resolution of the mesh should be equal or at least very close to the texture resolution as geometric details (namely height map) is coded in the texture. In other words, there must be as many vertices as many texels are used in the scene in each frame. With regard to the example of smiley texture from the previous section: if the quad or at least its mass is completely seen on the screen and is therefore textured with the texture with the pixel resolution 80×80 pixel^2 then it should be also preferably subdivided in such a manner that it becomes a mesh consisting of 80×80 equal spread vertices. In the case described in figure 2.8 when the middle part of the quad is textured with a section of the middle quality texture from the mipmap pyramid this middle part (and only this part) should be subdivided additionally as the resolution of the texture is higher compared to the situation in which the quad is completely visible. For reasons of clarity and comprehensibility, in the following the smallest texture in the mipmap stack is assumed to be 20×20 (so that the middle one is 40×40). In the left image in figure 3.18 the mesh for the case that the whole quad or at least its mass is visible is shown: it consists of 20×20 small quads as the corresponding texture resolution is 20×20 pixel^2. The middle image shows how the mesh should be tessellated: As only the middle 4 quads are visible, this visible part is textured with the corresponding (middle) part of the middle quality texture (with the resolution 40×40 pixel^2) from the mipmap pyramid – in other words, the middle part of the quad is textured with a texture with the double pixel resolution and therefore the corresponding part of the mesh must consist of double number of quads comparing to the initial mesh. The right image symbolizes that invisible tiles are

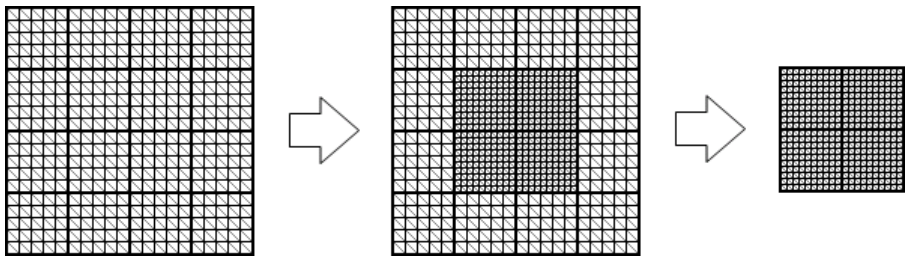


Figure 3.18.: Tessellation of the quad in the situation described in figure 2.8: Only the visible 4 tiles in the middle are tessellated in the manner that the geometrical resolution of this part of the quad is doubled and only these 4 tiles in the middle are rendered.

omitted and only the tessellated 4 tiles in the middle are considered while rendering.

3.6. Geomorphing

Section 3.4.5 explains how to avoid popping caused by tessellation due to the changing tessellation level. But in this work, the elevation data is taken from a height map so that replacing one texture (section) with another in course of virtual texturing also causes popping. The methods which try to solve this problem of popping geometries are called geomorphing.

Though blending two mipmaps as described in the previous chapter (section 2.6) gives a slight improvement, the problem is that the changes in geometry are much more noticeable to the viewer than equivalent color changes.

Next chapter describes a method which tries to avoid this problem with the help of an alternative approach.

4. Screen Resolution Depended Tessellation

Although using of trilinear filtering for height maps as mentioned at the end of the previous chapter helps to provide a smoother geomorphing but the result is not always reliable and some jumping from one resolution to another may be noticeable. On the other hand tessellation in OpenGL is very limited and can not generate geometries with unlimited level of detail. This section introduces a new approach which has arisen in the course of this work and which tries to solve both problems.

4.1. Basic Idea

The objective which is pursued in this section is to generate for each rendered pixel a corresponding vertex so that no height values (from the alpha channel in the texture) are wasted: if the rendered quad consists of 10000 pixels on the screen, then it must also consist of 10000 vertices.

Again, the concept is explained with the help of an example. Figure 4.1 shows an RGB texture in couple with the height map (both left) and an example of applying this data to a quad which is shown in perspective and is textured with the RGB texture and has a hill in the middle derived from the height map (right).

The actual idea is to prerender the scene and write the color and position of each fragment to a buffer (e.g. image buffer described in chapter 2, subsection 2.5.3) and then to apply the data from this buffer to a mesh which covers the whole screen, in other words, a mesh of vertices which consists of as many vertices as there are pixels on the screen. With the screen resolution 1280×1024, for example, the number of vertices

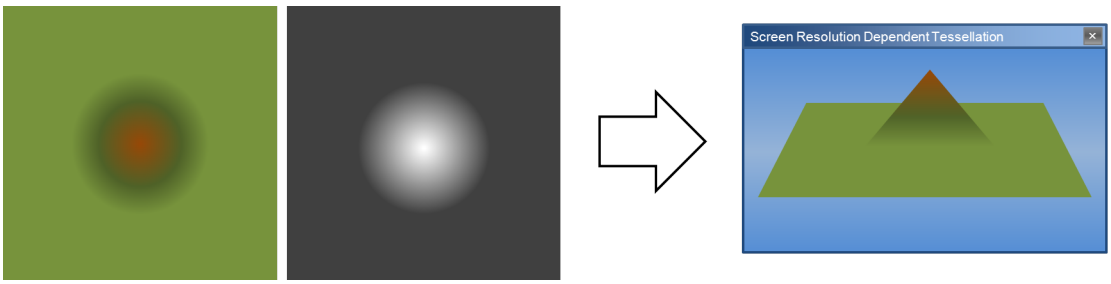


Figure 4.1.: An example of rendering a scene based on RGB data and a height map: on the left are an RGB texture and a height map and on the right is a rendered scene based on both.

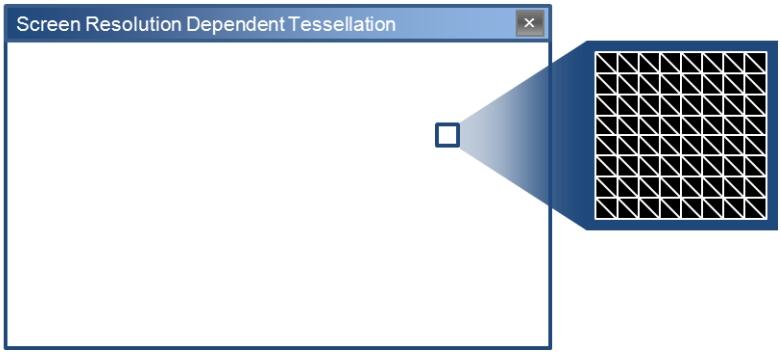


Figure 4.2.: Covering the whole screen with a mesh in which the distance between each two vertices is exactly one pixel on the screen: they are so serried that no holes in the mesh are visible.

amounts to 1310720. Such amounts of data are managed by OpenGL easily and the good news is that this number of vertices stays the same during the execution of the program (as long as the screen resolution stays unchanged) and so the mesh can be initialized once at the beginning of the program.

Figure 4.2 visualizes this setup: if rendering this mesh, the screen of the program window looks just like filled uniformly with a color due to the fact that the mesh is very tight. For the sake of clarity this mesh is refereed in the following as **screen mesh**.

Figure 4.3 visualizes the situation in which only the colors of prerendered scene are applied to the screen mesh: the screen seems to show a single textured quad but in fact, it is completely covered with a colored mesh. The advantage of this approach is obvious: whereas only a single quad is needed to prerender the screen, the final result is a very detailed (in fact maximally detailed) mesh. Figure 4.4 makes the whole operation

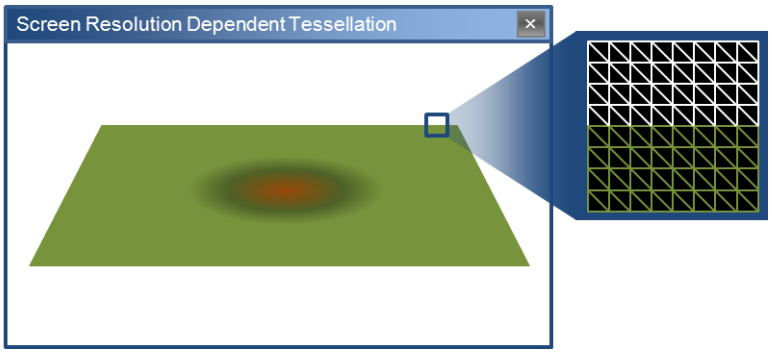


Figure 4.3.: Screen mesh after applying colors from prerendered scene to it.

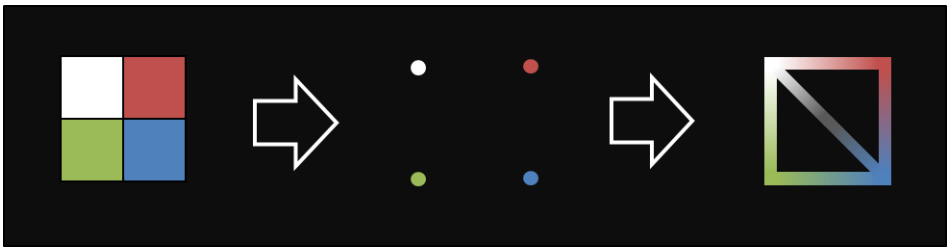


Figure 4.4.: Transforming colors from prerendered scene to the mesh: Each 4 pixels become 4 vertices in the screen mesh.

still clearer. Again, explaining only coloring of the mesh, it shows how four neighboring pixels of the prerendered scene influence the corresponding four vertices of two triangles in the mesh. The colors are interpolated in the mesh but as the vertices are so close this interpolation is not noticeable.

The positions are applied to the screen mesh in the same manner as colors: each vertex of the mesh gets the appropriate position from the buffer and the result is the same geometry as in the prerendering step but very fine tessellated. However a problem occurs if doing so in the case the geometry do not cover the whole scene: the parts of the screen which are not covered by the geometry (the whole white area in figure 4.3) are still represented in the mesh and lead to artifacts in the final scene. Figure 4.5 visualizes this problem: the white colored vertices of the screen mesh get no position from the prerendering process and preserve therefore their initial position which can be far away from the centre of the scene (that is why the corresponding triangles seem to be drawn away to the origin in figure 4.5). To solve this problem, these superfluous triangles should be eliminated.

One question still remains: how the height map should be integrated. The most optimal

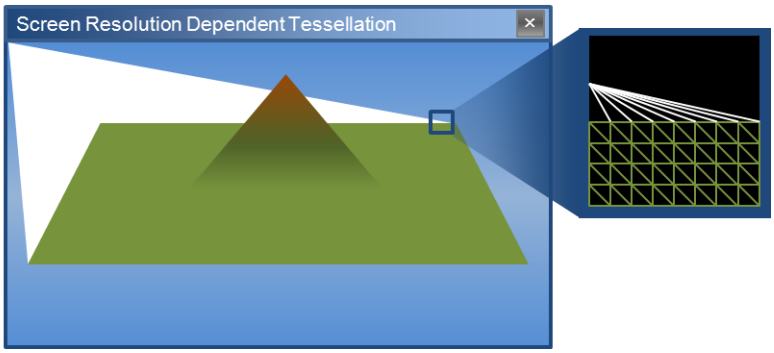


Figure 4.5.: The parts of the screen which do not represent the geometry (but represent background) cause artifacts.

way seems to be applying height map to each pixel in the prerendering stage. An advantage of doing so is that the positions are automatically interpolated.

4.2. Implementation Details

The kernel of this method are the image buffer (referenced in the following as **screen buffer**) which saves the (interpolated) scene and the screen mesh which is just a triangle mesh covering seamlessly the screen. The vertices of this triangle mesh have only position and color as attributes.

The idea is to render the scene using the method from chapter 2 to this screen buffer and then draw the screen mesh assigning the corresponding values from the buffer to each vertex in this mesh. As the screen buffer must save the colors as well as the positions, it actually consists of two buffers. Thus, the rendering step described in chapter 2 becomes a prerendering step and an additional rendering step which renders the screen mesh is added. Besides, one more prerendering steps is added which clears the screen buffer at the beginning (it can be done in the host program as well but it is not so efficient). On the whole the process consists of the following four steps (of course, each step is implemented in a separate shader program):

- clear the screen buffer;
- prerender for determining the visible area;

- prerender the whole scene to the screen buffers;
- render the screen mesh and apply the screen buffers to it;

In the following these steps are discussed.

Clearing the Screen Buffer

This step (and this shader program) is very simple: the screen mesh is rendered with a shader program which only consists of vertex shader and fragment shader. The vertex shader passes the positions through and the fragment shader uses the coordinates of each fragment on the screen (`gl_FragCoord.x`, `gl_FragCoord.y`) as positions in the screen buffers in order to set them to zero. Thus, the screen buffer is reset.

Prerendering for Determining the Visible Area

This step is absolutely the same as in chapter 2.

Prerendering the Whole Scene to the Screen Buffers

This step is almost the same as in chapter 2. The difference is that instead of assigning an output variable in the fragment shader (thus rendering fragments on the screen), this fragment shader make an operation similar to the operation in the first step. The difference is that not a zero is written to each position in the buffers but the (interpolated) values of the fragment, namely its (spatial) position and color.

Rendering the Screen Mesh and Applying the Screen Buffers to It

In this step the screen mesh is rendered to the screen. In this shader program the vertex shader reads the contents of the screen buffer and assigns positions and color from it to each vertex. Thus, after this vertex shader finishes, the vertices are replaced to the right position and a color is assigned to each of them.

The superfluous triangles shown in figure 4.5 can be eliminated in the geometry shader: emitting only the needed triangles solves the problem. Besides, in the geometry shader also per-triangle normals can be calculated. The fragment shader just colors the fragments as usual.

4.3. Results

Figure 4.6 shows an example of applying the described method. The level of detail of the vertex mesh stays always constant, independently of the distance between the Earth surface and the camera. It is only limited through the screen resolution and the level of detail of the used dataset.

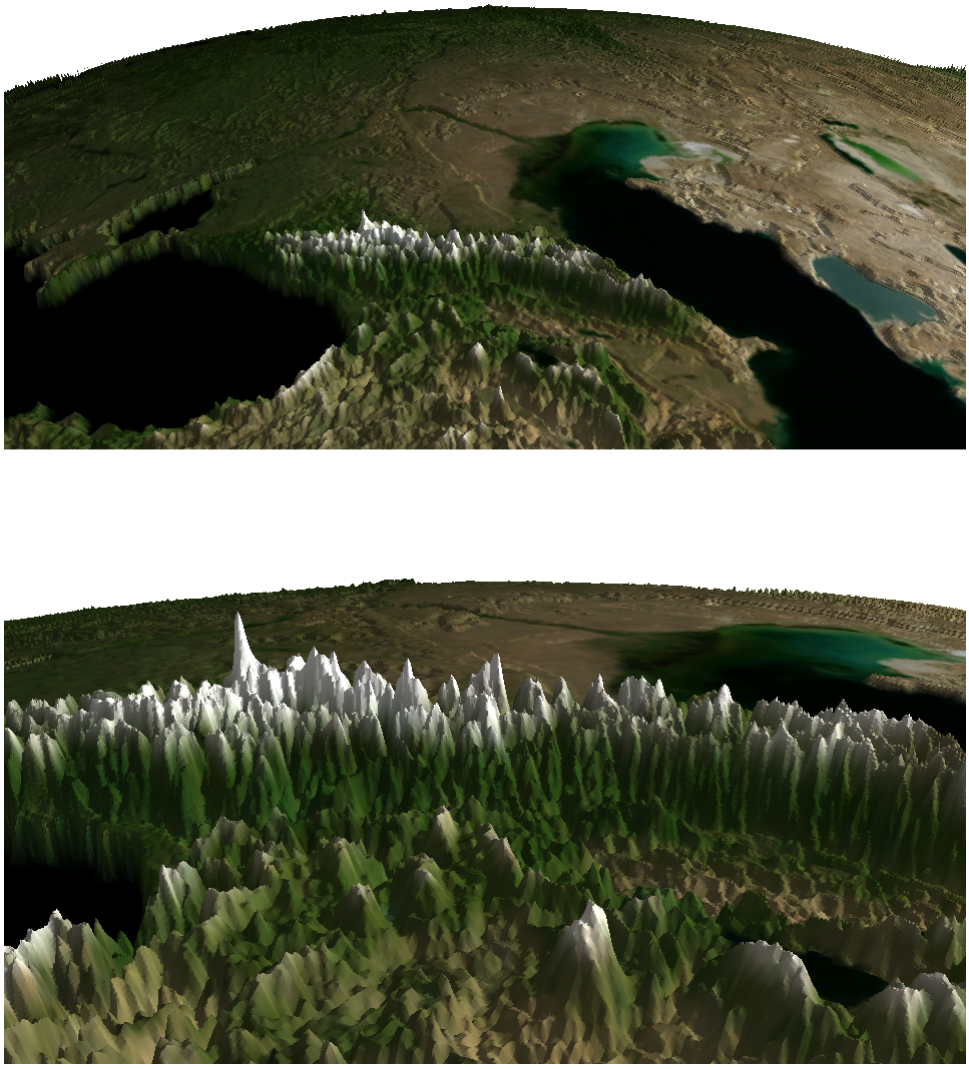


Figure 4.6.: The Caucasus Mountains at two different zoom levels. Independently of the distance to the Earth surface the level of detail of the vertex mesh stays always the same and is only limited through the screen resolution and the level of detail of the used dataset. Based on imagery and elevation data from [BM04].

5. Rendering Earth Surface

So far, the methods are presented which make the simulation of detailed terrain on a quad possible. This chapter explains how to apply these methods to a sphere in order to fulfill the main task of the thesis, namely rendering of the Earth.

5.1. From Plane to Sphere

There are mainly three classical variations of sphere meshes in computer graphics, namely uv-sphere, icosphere and cubesphere which are shown in figure 5.1. A uv-sphere is the result of applying a sine/cosine-formula to the vertices of a plane. The problem of this method is clearly visible in figure 5.1: the vertices are not evenly distributed over the sphere and their density is higher at the poles than at the equator which is especially disadvantageous with respect to depicting elevation data. Icospheres are spheres constructed from the icosahedron (a polyhedron with 20 triangular faces, 30 edges and 12 vertices): its vertices are evenly distributed but the problem is that these spheres must consist of 20×4^n triangles which is very restrictive for the purposes of this thesis and additionally, applying virtual texture to such icospheres is rather difficult. Cubespheres are built from a cube by normalizing all its vertices as figure 5.2 visualizes. They have quit evenly distributed vertices and are easy to manage. These facts are the reason of choosing them in this thesis.

A major disadvantage of using cubesphere is a bit higher vertex density at the spots which correspond to the edges in the initial cube. [CS05] purposes a formula which make the vertex distribution more even. Figure 5.3 shows how the distribution is improved with this formula.

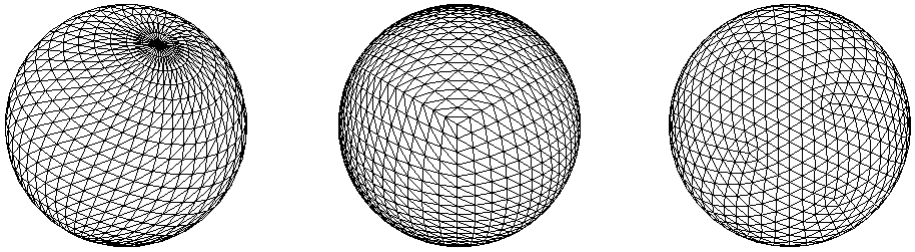


Figure 5.1.: Different kinds of spheres. From left to right: uv-sphere, icosphere and cubesphere. Rendered with OpenGL.

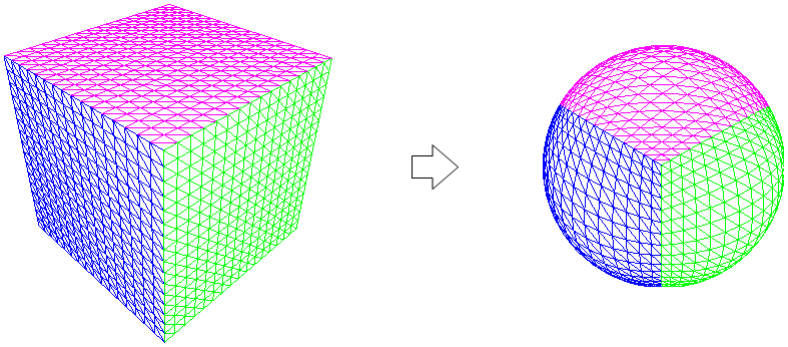


Figure 5.2.: Making cubesphere from cube via normalizing. The sides of the cube are colored for emphasizing the correlation between both objects.

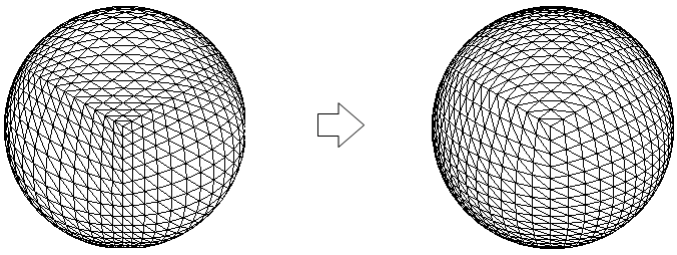


Figure 5.3.: Difference between the usual normalizing and an improved one from [CS05].

In the program, the cubesphere can be defined as a vertex mesh forming a cube and the normalization can be done in the shader program (for example in the vertex shader).

5.2. From Textured Quad to Textured Sphere

Another big advantage of using cubesphere is the possibility to apply cubemaps to them which is discussed in this section. Why the usage of cubemaps is especially advantageous is discussed in more details in Appendix A (subsection A.3.2) – at this place, it is enough to mention that OpenGL supports cubemaps natively.

A cubemap is actually a collection of six square textures which cover the corresponding sides of the cube from which the cubesphere is made. In this regard, the task of texturing a sphere is very similar to the task of texturing quad discussed in chapter 2. In the case of a cubesphere the virtual texture algorithm must be simply applied to each cube side. The only difference are the texture coordinates: in the case of cubespheres they are represented by a three-dimensional vector (x, y, z) which represents the spot on the cube surface which is to be textured.

If a cubesphere is the result of normalizing a cube centred around the origin $(0, 0, 0)$ with the edge length 2 (which is the case in this thesis), then the coordinates of each vertex on its surface are actually also its texture coordinates. Formula 2.3 which adopts texture coordinates for a single quad can be applied for a cubesphere almost without changes: the only thing to be considered is that each face of the cubesphere has another order of coordinates. For example, on the face with midpoint $(0, 0, 1)$ an arbitrary point (x, y, z) has the texture coordinates $(s', t', 1)$ according to formula 2.3.

5.2.1. Virtual Texture

The virtual texture technique described in chapter 2 can be applied to a cubesphere just by applying this technique to each face of it. The only issue occurring while doing this, is the situation in which the parameters $\texttt{width}_{\texttt{tiles}}$ and $\texttt{height}_{\texttt{tiles}}$ on two neighbouring sides are different. This problem should be explained with the help of the following example: Figure 5.4 shows two neighbouring squares sharing a texture. Figure

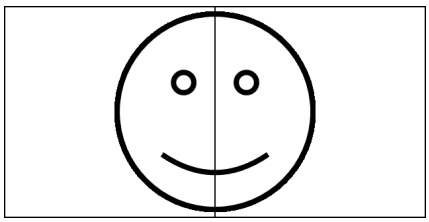


Figure 5.4.: Two squares sharing a texture.

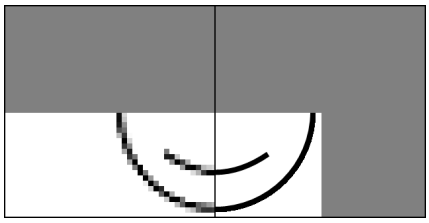


Figure 5.5.: Different $\texttt{width}_{\texttt{tiles}}$ and $\texttt{height}_{\texttt{tiles}}$ of the neighbouring faces cause artifacts on the seam between both.

5.5 shows how the virtual texture algorithm works if the quads a handled independently from each other: In the right quad is only 1/4 of the area of the whole quad is visible and in the left quad is the bottom half of the quad is visible (the not visible parts are marked with gray color). If the quads are considered independently, then, due to the virtual texture algorithm, they are textured with the textures with different pixel resolutions: the left one must be rendered with the smaller texture whereas the right one can be rendered with the texture of the next level in the mipmap pyramid. As it is apparent from the image, this is a considerable visual artifact but even worst is the situation about the tessellation because such neighbouring of differently detailed height maps causes T-junctions.

The only way to avoid this problem is not to handle the faces of a cubesphere independently but calculate the maximal $\texttt{width}_{\texttt{tiles}}$ and $\texttt{height}_{\texttt{tiles}}$ for all faces and then apply these maximal values to each face. This way, the problem in the given example is solved like in figure 5.6.

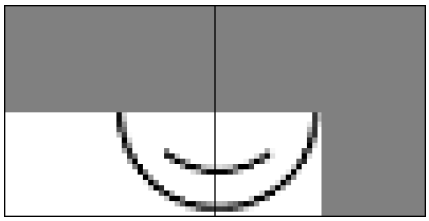


Figure 5.6.: Only choosing the texture with the smallest resolution from all available can solve the problem in figure 5.5.

6. Conclusion

The aim of this work was to visualize the Earth surface as detailed as possible and to test some new OpenGL functions introduced in OpenGL 4.2. In course of this work this task has been principally fulfilled.

The implementation of virtual texture done in this work is comparable to other approaches in this area with respect to ability of rendering really large textures. The new OpenGL 4.2 functionality of "images" turned out to be really helpful in simplifying the work and making the final program more reliable and precise.

Using hardware tessellation in OpenGL is a simple way to simulate a detailed terrain. But this method has a severe limitation, the impossibility of tessellating at a level higher then 64. Thus not any level of detail can be reached using this technique. Though the hardware tessellation is currently very limited for generating really detailed mesh, it is imaginable that in the future this technique will supplant the old approaches like geometry clipmaps.

The author's own approach discussed in chapter 4 is capable of rendering very fine meshes with constant fineness which is fully independent of any parameters. It seems to be quite promising because of its small graphics memory consumption with regard to the geometry. It can be especially advantageous for rendering artificial terrains which do not rely on any dataset.

A. Data

The main effort in this work is to represent the Earth sphere as detailed as possible. This implies above all the usage of detailed surface images as well as detailed elevation data for depicting mountains and hills. This section is devoted to describing the data used in this work. The aim is to bring this data to a form which is comfortable for rendering. Most convenient way in this case is to compose all the data in one image as well as to try to reduce the size of the image and bring it to the format which is mostly appropriate for OpenGL.

A.1. Resources

There many resources in the internet with Earth surface data, both free and commercial – quite an extensive overview of such datasets can be found on the web page of the Virtual Terrain Project ([TP11]). But most of them do not cover the entire world or have artifacts or are in such a format which is difficult to adopt for rendering with OpenGL. One of the best free of charge resources which can be found in the internet are the NASA satellite images from the Blue Marble Next Generation Project ([BM04]). In this project, the whole world is represented, among others, as a big rectangular image in uncompressed (raw) RGB format which is the result of a geographic projection, based on equal latitude-longitude grid spacing. This image is available in different resolutions, but the most interesting for the purposes of this work is the highest one, namely the spatial resolution of 16 arc-seconds which approximately corresponds to a distance of 500 meter between two pixels in the image at the equator. This spatial resolution involves the pixel resolution of 86400×43200 pixel^2.

Additionally, the Blue Marble Project Next Generation Project provides not only the image of the Earth surface in form of colored image but also the elevation data in form of

height map of the same pixel resolution 86400×43200 pixel^2. This data is given in form of an uncompressed grayscale image in which the intensity of the gray color is proportional to the height of corresponding area on the Earth surface. Moreover a watermask of the same resolution is provided which allows differentiating all water bodies such as rivers, lakes and oceans from land areas. The oceanmask is also presented which separates ocean from the mainland. These two masks are represented as monochrome images with the same pixel resolution 86400×43200 pixel^2 where black color stands for water and white stands for land.

Besides the impressive detail, this dataset has a big advantage (in comparison with many others) of being cleaned from clouds and snow reflectance as well as being seamless and true-color. Moreover, one world image for each month is available so that a season dependent depicting of the world becomes possible. [BM07]

All these facts contribute to the decision of choosing this dataset in this work.

A.2. Combining Data into an Image

As mentioned in the previous section there is on the one hand an image in RGB format as well as an image with elevation data and an image with the oceanmask (watermask is not used in this work) on the other hand. As OpenGL supports RGBA format, it is advisable to make use of this fact and to code the elevation data in the alpha channel in the first place.

The highest spot on the mainland due to the dataset amounts for 8573 meters and the lowest spot lays 1049 meters under the sea level so that there are overall 9622 discrete height values possible. As the alpha channel can only manage the values between 0 and 255, there is a need of scaling. Making one unit in the alpha channel fro example equivalent to 40 meters scales the interval $[0, 10000]$ (which completely includes all possible heights) down to the interval $[0, 250]$ which is enough for writing the values to the alpha channel.

Thus it is possible to combine color as well as elevation data into one RGBA-image. But there is also a need to incorporate the oceanmask into this image. This can be done, for example, if using alpha channel also for this information, namely reserve 0 for ocean

and the values 1 to 255 for height values on the mainland.

The original heights (`height`) can be mapped to the alpha channel values ($\texttt{height}_{\texttt{alpha}}$) with the help of formula (A.1). This formula maps the heights to the interval $[1, 255]$ (scaling factor is 1 unit $\hat{=}$ 37,8 meters) so that the value 0 stays reserved and can be used for identifying water (0 means "ocean" and > 0 means "mainland"). While rendering, the alpha values must be then scaled back and 1049 must be subdivided from the result in order to decode the real height value back.

$$\texttt{height}_{\texttt{alpha}} = \frac{254 \cdot (\texttt{height} + 1049)}{8573} + 1 \tag{A.1}$$

A.3. Reducing the Size

The huge size of the dataset raises the question whether it is possible to compress or somehow else optimize the data.

A.3.1. Image Compression

There are different ways to compress the image data, both lossless and lossy. All the lossless formats existing nowadays (like PNG) have a big disadvantage with respect to the rendering on GPU, namely the impossibility of running the decoding process multithreaded as the common compression methods extensively use the dependencies between different parts of the image. Contrariwise, many lossy compressions, like JPG-format, can be parallelized and thus used on GPUs as they divide the image in smaller parts and process them independently.

OpenGL natively supports the so called S3 texture compression or S3TC ([TC00]). This compression works similar to the JPEG to the effect that it subdivides the whole image to be compressed in 4×4 blocks of pixels and compresses them independently. This implies one important issue relating to this format, namely both width and height of an image to be compressed must be a multiple of 4.

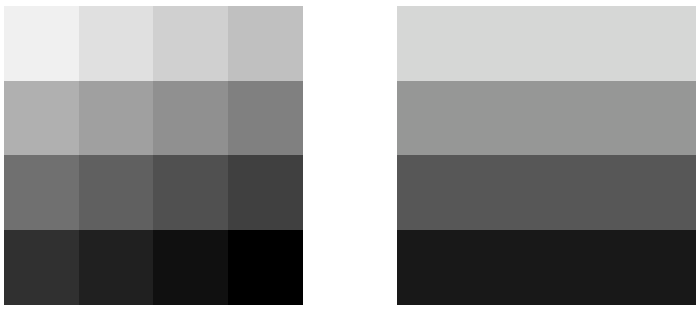


Figure A.1.: A 4×4 block of pixels with **soft** color transitions compressed with the help of S3TC. Left image shows the original pixels and the right one shows the result of compression. (The right image is made with "Paint.NET" [PN11].)

There are on the whole five variations of S3TC (named DXT1 through DXT5) differing primarily in the art of compressing the alpha channel. The most interesting variation for the purposes of this work is the format DXT5 which compresses RGBA image with interpolated alpha.

Contrary to the JPEG, each block of the image is always compressed with the same, constant compression ratio if using S3TC. In the case of DXT5 this ratio is 4:1, i.e., the size of compressed image accounts for only 25% of the original image's size.

As the format is lossy, the reasonable question is how much the image quality is affected by the compression. The method interpolates between edges of each block and for the quality of the compressed picture, it is crucial how hard the color transitions within this block are. Figure A.1 shows the case with soft color transitions in which the lost of quality is evident but acceptable. Contrariwise figure A.2 shows very bad quality of the compressed block due to the very contrast color differences in the original image. But this last example should not be seen as a proof that the method is completely inadmissible, as the color transitions on the image of the Earth surface, which this work deals with, are rather soft and cause no problems with respect to compression which shows figure A.3.

One further big advantage of using the S3 texture compression while rendering despite the reduced size of the image is also acceleration of rendering process as not only the amount of memory on the video card but also the memory bandwidth becomes smaller comparing to the usage of uncompressed images ([TC00]).

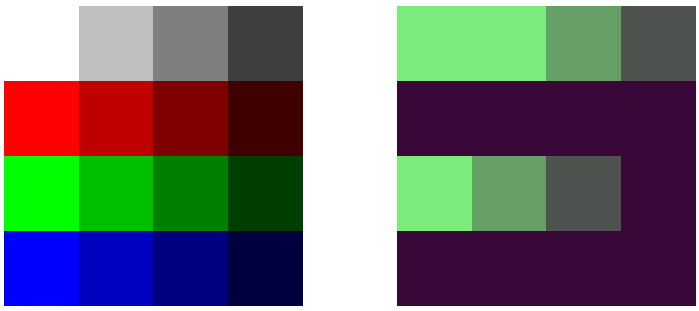


Figure A.2.: A 4×4 block of pixels with **hard** color transitions compressed with the help of S3TC. Left image shows the original pixels and the right one shows the result of compression. (The right image is made with "Paint.NET" [PN11].)

Figure A.3.: Comparison between S3TC-compressed image (left) and uncompressed image (right). It is evident that the images are nearly identical. In the middle is the graphically represented (hardly visible) difference between both. (The image is made with the help of "The Compressonator"[CO08].)

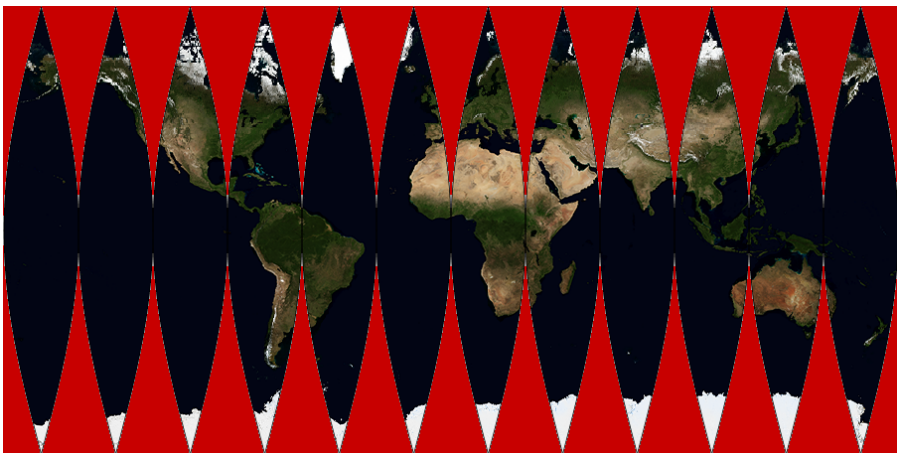


Figure A.4.: Interrupted sinusoidal projection of the world. Red colored areas symbolize the wasted area if using reprojection to a rectangular image. (The image is made with the help of "G.Projector – Global Map Projector"[GP12].)

Due to the compression ratio 4:1 it becomes possible to reduce the original size of the image equal to 15GB to approximately 3.75GB.

A.3.2. Optimal Format

The Earth surface image delivered by NASA is rectangular as the result of a geographic reprojection of the original data which was given in from of sinusoidal projection, like one shown in figure A.4. This form is due to the sphere form of the Earth. It is evident that the considerable part of the surface area, approximately 33% (marked in figure A.4 with red color) is wasted. Thus, it is worth considering another format for representing the Earth surface which manages the area more efficiently.

A more optimal method of representing the Earth surface then just in form of a rectangular image is the so called quadrilateralized spherical cube projection. This projection maps the Earth to a cube and represents the Earth surface as a collection of six equal square areas, namely the faces of this cube. These faces put seamlessly together are shown in figure A.5. Though this projection does not manage to preserve all the 33% of the area wasted by the rectangular representation from above, it nevertheless saves 25% of the image size (the saved area is marked with green color in figure A.5).

According to the given dataset from NASA, using of cubemaps reduces the overall data

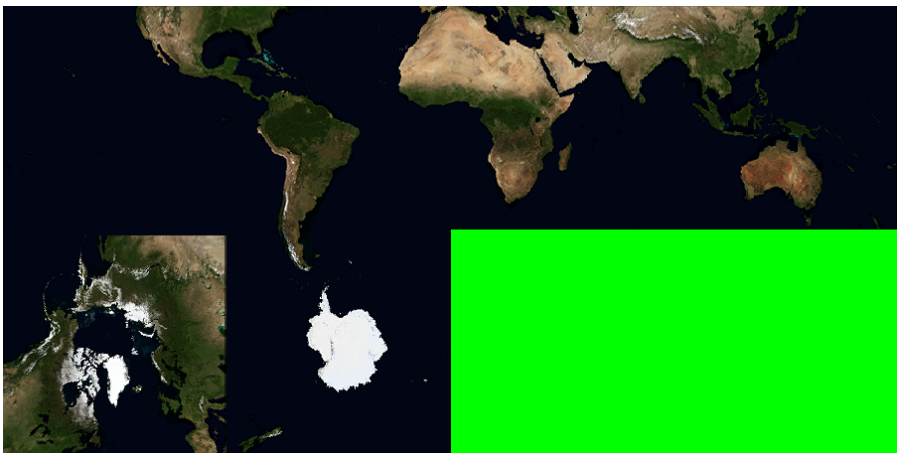


Figure A.5.: Quadrilateralized spherical cube projection of the world. (The image is made with the help of "G.Projector – Global Map Projector"[GP12].)

size from 3.75GB to approximately 2.8GB while using quadrilateralized spherical cube projection. This fact and especially the ability of OpenGL to manage such form of data make this form of projection very preferable.

A.4. Putting All Things Together

Converting the raw data from NASA to the needed compressed and optimized format described above is made in four steps:

- Firstly, the raw color data is combined with the elevation data and the oceanmask with the help of a little Java program.
- Secondly, the quadrilateralized spherical cube projection must be made. There are many programs which can do this kind of projection, e.g., the "NVIDIA Plug-in for Adobe Photoshop" ([DP12]). But most of them can not manage such big images as they try to load the whole input image into memory. One program which is able to do this and used in this work is a small program, simply called "CubeMap" ([CM11]). Its output are six cube faces in raw RGBA format.
- Thirdly, these six faces are converted into PNG format with a small Java program.
- Finally, the faces in PNG format are S3TC compressed with the help of "Paint.NET"

([PN11]). The usual file format of the images compressed with S3TC is DDS (DirectDraw Surface, [TC00]). This format supports amongst others the mipmaps ([MM83]).

Using mipmaps increases of the image's size by approximately 33%. Thus, after all transformations the final size of the data accounts for about 3.7GB.

Bibliography

[TP11] Virtual Terrain Project, 2011
`http://vterrain.org/`

[BM04] NASA
Blue Marble Next Generation Project, 2004
`http://earthobservatory.nasa.gov/Features/BlueMarble/`

[BR04] NASA
Blue Marble Next Generation Project (resources), 2004
`ftp://ftp.cscs.ch/out/stockli/bluemarble/bmng/`
`http://mirrors.arsc.edu/nasa/`

[BM07] Reto Stöckli, Eric Vermote, Nazmi Saleous, Robert Simmon, David Herring
The Blue Marble Next Generation - A true color earth dataset including seasonal dynamics from MODIS
NASA Earth Observatory, 2007

[TC00] Sébastien Dominé
Using Texture Compression in OpenGL
NVIDIA Corporation, 2000

[MM83] Lance Williams
Pyramidal parametrics
Computer Graphics Laboratory, New York Institute of Technology, 1983

[CM98] Christopher C. Tanner, Christopher J. Migdal, Michael T. Jones
The Clipmap: A Virtual Mipmap
Silicon Graphics Computer Systems, 1998

[AV08] Martin Mittring
Advanced Virtual Texture Topics
Advances in Real-Time Rendering in 3D Graphics and Games Course, SIGGRAPH, 2008

[ID09] J.M.P. van Waveren
id Tech 5 Challenges - From Texture Virtualization to Massive Parallelization
id Software, SIGGRAPH, 2009

[NV07] Evgeny Makarov
Clipmaps (NDIDIA White Paper)
NVIDIA Corporation, 2007

[MT06] John Carmack
MegaTexture Q&A
`http://floodyberry.com/carmack/johnc_interview_2006_MegaTexture_QandA.html`, 2006

[UT04] Sylvain Lefebvre, Jérome Darbon, Fabrice Neyret
Unified Texture Management for Arbitrary Meshes
INRIA, 2004

[ST08] Sean Barrett
Sparse Virtual Textures
Game Developers Conference, 2008
`http://silverspaceship.com/src/svt/`

[OC13] KHRONOS GROUP
OpenCL
`http://www.khronos.org/opencl/`, 2013

[CU13] NVIDIA
CUDA
`http://www.nvidia.com/object/cuda_home_new.html`, 2013

[OP13] Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane
OpenGL Programming Guide, Eighth Edition - The Official Guide to

Learning OpenGL, Version 4.3
Addison-Wesley, 2013

[VC10] Charles-Frederik Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle
Accelerating Virtual Texturing Using CUDA
GPU Pro: Advanced Rendering Techniques
A K Peters, 2010

[VO10] Albert Julian Mayer
Virtual Texturing (Diploma Thesis)
Vienna University of Technology - Faculty of Informatics, 2010

[SC09] Markus Billeter, Ola Olsson, Ulf Assarsson
Efficient Stream Compaction on Wide SIMD Many-Core Architectures
HPG '09: Proceedings of the Conferenceon High Performance Graphics 2009

[GC05] Arul Asirvatham, Hugues Hoppe
Terrain Rendering Using GPU-Based Geometry Clipmaps
GPU Gems 2
`http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html`, 2005

[PM01] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, S. Tachi
Detailed Shape Representation with Parallax Mapping
ICAT, 2001

[VG11] Patrick Cozzi, Kevin Ring
3D Engine Design for Virtual Globes
A K Peters/CRC Press, 2011

[CS05] Philip Nowell
Mapping a Cube to a Sphere
`http://mathproofs.blogspot.de/2005/07/mapping-cube-to-sphere.html`, 2005

Used Software

[CO08] AMD
The Compressonator, 2008
http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/the-compressonator/

[GP12] NASA
G.Projector – Global Map Projector, 2012
http://www.giss.nasa.gov/tools/gprojector/

[PN11] dotPDN LLC, Rick Brewster, and contributors
Paint.NET, 2011
http://www.getpaint.net/

[DP12] NVIDIA
NVIDIA Plug-in for Adobe Photoshop, 2012
https://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop/

[CM11] Vladimir Romanyuk
CubeMap, 2011
http://sourceforge.net/projects/cubemap/files/

Proclamation

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Osnabrück, August 13, 2013 .....

Sergey Krutikov