2010:009 HIP

BACHELOR'S THESIS

Implementing Virtual Texturing

Martin Andersson

Luleå University of Technology

BSc Programmes in Engineering
BSc programme in Computer Engineering
Department of Skellefteå Campus
Division of Leisure and Entertainment

2010:009 HIP - ISSN: 1404-5494 - ISRN: LTU-HIP-EX--10/009--SE

Implementing Virtual Texturing

Martin Andersson

Luleå University of Technology
Computer Game Development
May 2010

Preface

This project was performed at Agency 9 in Luleå during 10 weeks in the spring term of 2010. The intention was to implement and evaluate a technique for rendering very large textures that does not fit in video memory. This is my bachelor's thesis and the last step in my education at gsCEPT in Skellefteå.

I would like to thank my mentor, Tomas Karlsson, for giving me the opportunity to perform my thesis at Agency 9 and for providing me with a subject that fits my field of interest, graphics programming. I would also like to thank everybody at Agency 9 including Tomas, Tompa, Martin and Lasse for their great company during this project and for using their competence to help me with problems and difficulties that I ran into.

Abstract

With increasing demands for better graphics quality in today's video game industry the game developers are constantly tackling new challenges to satisfy the gamers needs. With limitations on today's hardware the challenge lies in developing different methods to conquer these limitations. When it comes to texturing a 3D object one problem that arises is how we render an object that uses a texture which is too large to fit inside the video memory. This report describes the implementation of a relatively new technique with a concept similar to Virtual Memory known as Virtual Texturing. The result is a real-time application demonstrating a teacup mapped with a texture with the dimensions of 2,097,152 ² pixels.

Sammanfattning

Med ökande krav på bättre grafik i dagens dator- och tv-spelsindustri möter spelutvecklarna ständigt på nya utmaningar för att tillfredsställa spelarnas behov. Med begränsningar på dagens hårdvara ligger utmaningen i att utveckla olika metoder för att överkomma dessa begränsningar. När det kommer till att texturera ett 3D-objekt är ett av problemen som uppkommer hur vi kan rendera ett objekt som använder en textur som är för stor för att få plats i videominnet. Den här rapporten beskriver implementationen av Virtual Texturing som är en relativt ny teknik med ett liknande koncept som hos Virtual Memory. Resultatet är en real-tids applikation som demonstrerar en tekopp mappad med en textur med dimensionerna 2 097 152 ² pixlar.

Contents

1 Introduction

1.1 Background

When a new video game is released the gamers are always demanding better visual quality. The textures are expected to have more detail and higher resolution and the size of the virtual world is also expected to be larger. Higher resolution means more data storage and larger worlds using these textures means that more data needs to be transferred between the data storage device and the display adapter. The transfer needs to happened very fast to create a comfortable gaming experience, preferably 60 times per second.

This tends to create problems on the hardware because it only has a limited amount of storage and a limited bandwidth for transferring data. Even though new graphics cards are released each year with higher memory capacity and higher bandwidth this is not always enough.

When it comes to very large texture we might ask ourselves, how do we render a texture so large it does not fit into video memory? That is the main problem discussed in this thesis. A technique which tries to solve this problem know as Virtual texturing is described and evaluated.

1.2 Purpose

The purpose of this project was to evaluate the technique to find out its strengths and weaknesses. The goal with this project was to create a real-time 3d application implementing Virtual texturing in Java using OpenGL and the GLSL shading language. The application was intended as a demonstration to show the technique in action.

1.3 Delimitations

Considering the limited amount of time I decided not to put down any significant amount of energy on trying to optimize the algorithms, my first priority was to implement a working demonstration as fast as possible. I have also skipped the implementation of padding the edges of the pages to avoid artifacts along the borders of the pages. I have only implemented bilinear filtering with the help of the built in functionality in OpenGL, no trilinear or anisotropic filtering is implemented. Since the goal was intended as a demo I did not put down much energy on the code structure, it ended up with only a few classes with large amounts of functionality hard coded.

2 Theory

Virtual texturing is a technique that takes advantage of the concept of virtualizing data storage very similar to Virtual Memory [1]. We treat our texture as if it is living in video memory when in fact only relevant parts of it are streamed into the video memory from another storage media. These parts are the ones that is needed during rendering. This significantly increases the possible dimensions our texture can have.

In this chapter the theory behind the technique is described in detail and different approaches and solutions to problems are discussed.

2.1 Partitioning of the virtual texture

The first step is to divide our very large texture (from now on called our virtual texture) into smaller equally sized pieces (called pages) that we actually can fit into video memory, see figure 1. The width and height of our virtual texture and these pages should preferably be a power of two to avoid any problems with the graphics driver not supporting non power of two textures. The size of each page is a matter of balancing the pros and cons between having a large or a small size, it really depends on the implementation.

Figure 1: Partitioning the virtual texture into smaller pieces that fits in video memory.

This first step can be done in two ways. The first way is to have the virtual texture divided into pages in a preprocessing step before the program runs and store them on the hard drive as single textures. The other way would be to keep the virtual texture as one whole texture on disk and read out the smaller parts from it during run-time.

The textures does not have to be stored on the local hard drive, they might be stored on any other media. E.g. DVD/CD, flash memory, another computer on the network. Yet, the hard drive is probably the most efficient place to store the textures. As [2] describes it, requests are fulfilled in constant time and memory requirements are constant.

2.1.1 MIP maps

The concept is meaningless if we don't introduce MIP maps [3]. Without mipmaps we would not be able to render our scene when the entire virtual texture is inside the view frustum because this would mean that all of our pages would have to be inside the video memory and the whole purpose of dividing the virtual texture into smaller pieces is lost.

So what we need to do is create mipmaps that we, like with our virtual texture, divide into smaller pieces (see figure 2). Unlike ordinary mipmaps we do not decrease the original image in size until we end up with a 1x1 pixel image. We stop when the image size is equal to the size of one page. This means we end up with a bit less data compared to ordinary mipmaps which needs 1/3 of extra memory apart from the original image. Generating the mipmaps should preferably be done manually in a preprocessing step.

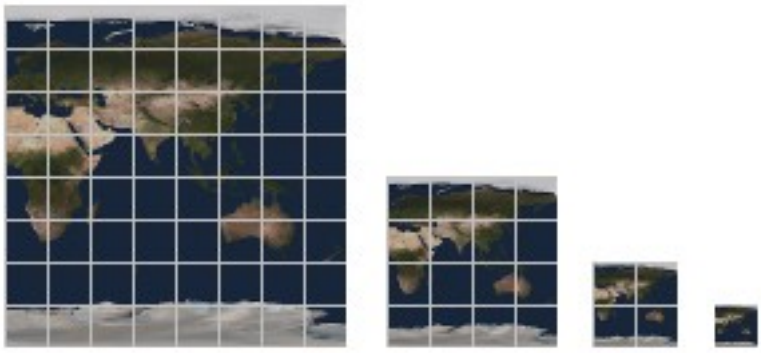

Figure 2: Dividing each mipmap into equally sized pages

2.2 Determining the active pages

The next step is to determine which pages we need to have in video memory (now called active pages) to be able to render what we currently see on the screen (I will use a scene and a snapshot from it as an example throughout this chapter, see figure 3). We also need to take into account what could be visible in the near future, e.g. if the user quickly rotates the camera or if the user is moving fast from one point to another. This would probably mean that a whole different part of the virtual texture is visible and therefore a whole set of new pages needs to be in memory. We can never predict exactly which pages we might be needing in the near future, but we can make some predictions that will give us acceptable results. To know which pages we need for the current frame there are a few different approaches we can take.

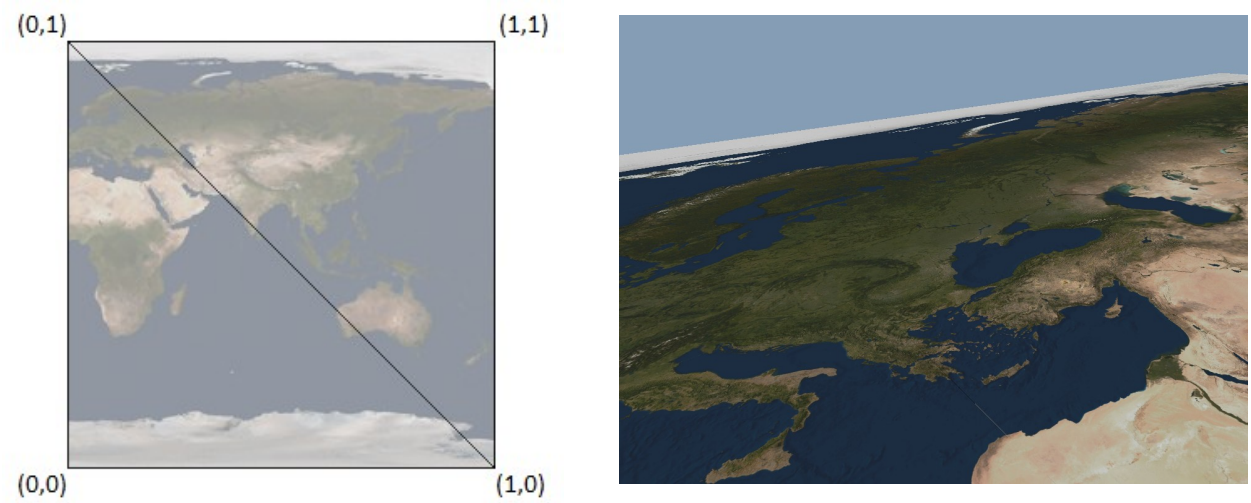

Figure 3: This simple scene contains only two triangles (to the left), the numbers in parenthesis is the virtual texture coordinates (u,v). To the right is a snapshot of the scene from the cameras perspective.

2.2.1 Render to off-screen buffer

One approach would be to render the scene in a first pass where we for each pixel store information about the 3D object it represents (if any) in the color channels. This information is stored in a off-screen buffer and is then analyzed on the CPU to determine which pages are active (see figure 4). The information we need to store is which mipmap level and which page in the virtual texture the pixel is using. The mipmap level is estimated by calculating the derivative of the virtual texture coordinate [4]. When we calculate which page the pixel represent we use our calculated mipmap level and again our virtual texture coordinate in combination with trivial math to get the x and y coordinate for the bottom-left corner where the page is located in the virtual texture space. If the pixel does not represent a 3D surface that uses the virtual texture (e.g. the background or some other object occluding our surface), we need some kind of value telling us it should be ignored when we later analyze it.

Figure 4: This is an example of how the off-screen buffer might look like, containing the necessary information to determine the active pages. The rings going upward is the different mipmap level, and two lines are visible going vertically and horizontally, these are the page borders.

There is a performance issue when dealing with this approach. The fact that we need to analyze each pixel in our off-screen buffer means that a lot of CPU calculations has to be done every single frame. E.g if we have a screen resolution of 1280 x 1024 pixels, that means we need to perform calculations on over 1.3 million pixels every frame and the screen resolution might be even higher than that which means even more calculations.

One solution to this problem is to take advantage of the vertex pipeline in the GPU. If we take our off-screen buffer and treat the pixels as vertices we can render these orthogonally on a texture and then use this texture to figure out which pages are active. I will discuss the implementation of this in chapter 3.2.

Another solution is to render the first pass in a lower resolution than we use in our final pass, as Barrett suggests in his implementation [5]. E.g. if we scale down the resolution 1280 x 1024 by 8 in each dimension we end up with about twenty thousand pixels which is a lot less compared to the original amount. Unfortunately, this solution does introduce another problem. When using a lower resolution we end up losing possibly critical information because we have fewer pixels. Some of the lost pixels might reference a page that we need to use in the final pass. This however is acceptable to a degree, depending on the number of pages we have and how much we scale down the original image.

2.2.2 Triangle-camera distance

A different approach on how to determine which mipmap level to use is to measure the distance between the triangle to be rendered and the camera. The greater the distance, the lower the mipmap level to use. The big problem with this approach is that when the triangle is spanning over a long distance the mipmap level becomes very inaccurate. E.g. if the triangle starts near the camera and ends far away from the camera you end up with to low resolution near the camera (assuming that the distance is measure from the center of the triangle) since only one level is used per triangle. You could get around this problem by splitting up the triangle in smaller pieces, but this would of course mean more geometry data.

2.3 Texture management

Now when we know which pages are required to render the final scene we need a system that handles the loading of the corresponding textures. We need to load the textures asynchronously from our storage media to avoid stalling the application, also known as streaming. This can be done by having another thread doing the loading. The texture needs to be loaded from the hard drive to system memory and finally into video memory, see figure 5 below.

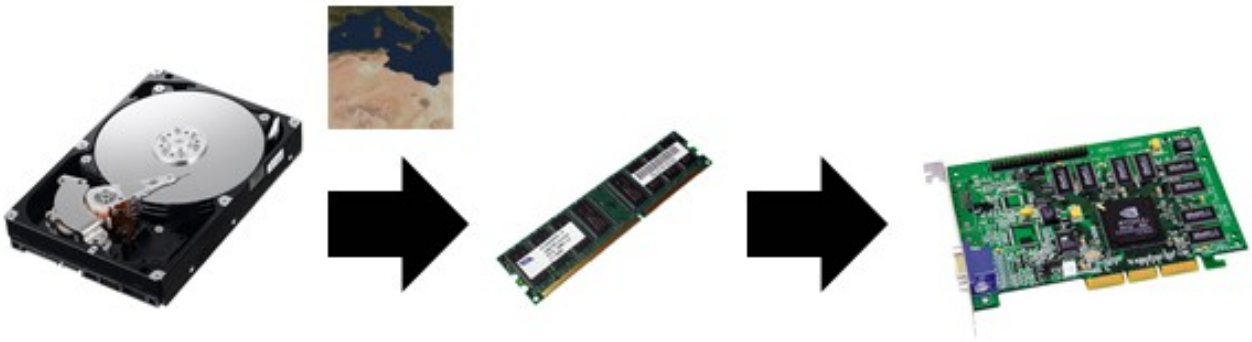

Figure 5: Streaming from the hard drive.

2.3.1 The physical texture

I have previously referenced the place where we store all our pages as the video memory. This is indeed where it is stored but we have to allocate how much memory we want to use. This is done by creating a texture (from now on called the physical texture) that will hold all the pages needed for rendering. This texture is updated every frame with new pages (assuming the viewport has changed and new pages are visible) by packing them tightly together, for an example see figure 6. Where we place the page inside the physical texture does not matter because we will store its position in what we call our page table, described in the next chapter.

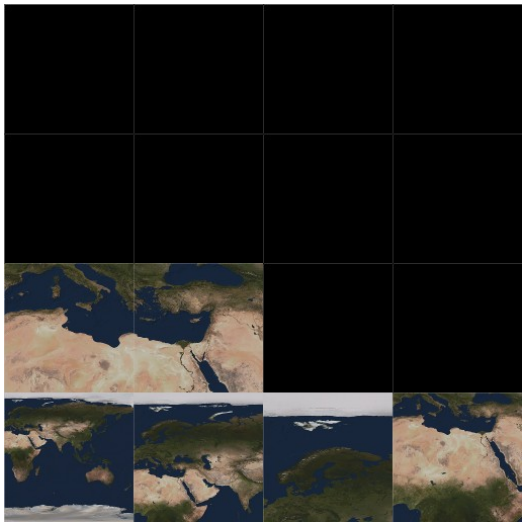

Figure 6: Active pages packed into the physical texture.

The dimensions of our physical texture depends on the implementation, it is a matter of balancing quality with performance and limited resources. In the most trivial implementation the size could be equal to the size of one page. This could mean that only the lowest mipmap always would be rendered because no other pages would fit inside the physical texture. If quality is what we want this is clearly not the way to go. In the perfect implementation we would like to have the physical texture large enough to make room for the maximal number of pages. Unfortunately, it is probably impossible to know exactly what this maximal number is because of all the possible geometrical structures our mesh can have. Fortunately we can get away with estimating this number.

The problem here is what would happened when the physical texture is full. We would have to either overwrite some page or just stop filling in any more pages. The preferred way around this is to have a LRU cache [6] that keeps track of which pages can be discarded and overwrite the most unused pages.

2.3.2 Filtering artifacts

One big problem with this technique arises when we use texture filtering on the physical texture. The problem is that when the interpolation is happening at the borders between adjacent pages, pixels from two or more different pages will be blended together, and because the pages is placed in no specific way inside the physical texture the neighboring pixels at the borders has no relation to each other. This will cause artifacts later on in the rendering process when we are sampling from the edge of a page. You can clearly see a seam in figure 3 right beneath Greece.

To get around this problem we can add a padding around the borders of each page. This padding consists of texel from the adjacent pages in the virtual texture. We also need to contract the sampling one texel on each side of the page or else the padding will have no effect because we will still be interpolating from some other page. The padding needs to be increased as we go down to a lower mipmap level.

2.4 The page table

This physical texture is now containing all our texels required to render the final scene. The question now is how do we know where inside this texture we want to sample from.

This is where the page table comes in handy. The page table will translate our virtual texture coordinate into a texture coordinate in our physical texture. It will tell us exactly where inside the physical texture our texel is located. The page table is a texture with the same number of pixels as the number of pages in the virtual texture. Hence, each pixel in the page table corresponds to a page in the virtual texture.

When we place our page in the physical texture we also want to store its position in the page table, i.e. the bottom-left corner of the page as x and y coordinates of where is it located inside the physical texture (ranging from 0 to 1, where (0,0) is the bottom-left corner and (1,1) is the top-right corner of the texture). We also want to store the mipmap level the page belongs to. This information is later used to compute the place where we will sample from.

Because we want access to the page table from the GPU pipeline we want to represent the page tables as textures which will be living in video memory. We will be needing one page table for each mipmap level. The size of each page table will be equal to the number of pages in each mipmap level. E.g. for the coarsest mipmap level we will be using a 1x1 pixel texture, next level 4x4 pixels, next 16x16 pixels etc. We call these texels our page table entries. This means that each texel in the page table has a corresponding page from the virtual texture.

After all the page tables for each mipmap level has been filled with the information about the active pages these are combined into a final page table that has the same number of entries as the page table corresponding to the highest mipmap level. This final page table will be used later during rendering. When combining the page tables we can think of it as if all the page tables have the same dimensions and that we are putting them on top of each other with the lowest mipmap level (the page table that has 1 texel) at the bottom and with the highest mipmap level at the top, see figure 7.

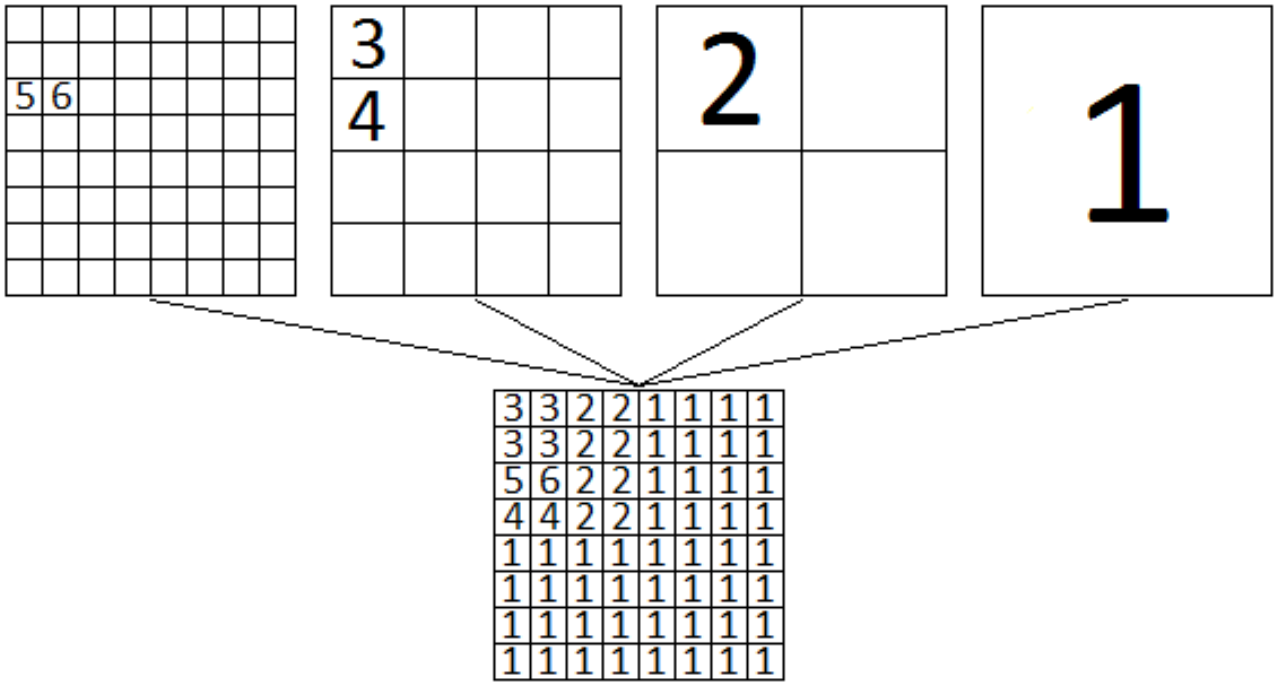

Figure 7: Combining the page tables to produce the final page table. Each number is representing one of the active pages. The numbers are just for the purpose of the illustration, they are not actually stored in the page tables.

2.5 Rendering

Now we have everything we need to be able to render the final scene. We have all our texels packed into the physical texture and we have our page table texture that will tell us where we want to sample from in the physical texture. Both of theses textures are in video memory so that we can access them from the GPU pipeline. In figure 8 below the red dot represents a texel we want to draw, we can follow it through each step till it reaches the screen in the bottom-left picture.

The top-left picture shows a texel that we want to render. The next picture uses the exact same texture coordinate which maps to a page in the physical texture in the bottom-right picture, and finally the bottom-left pictures shows the texel from the camera perspective.

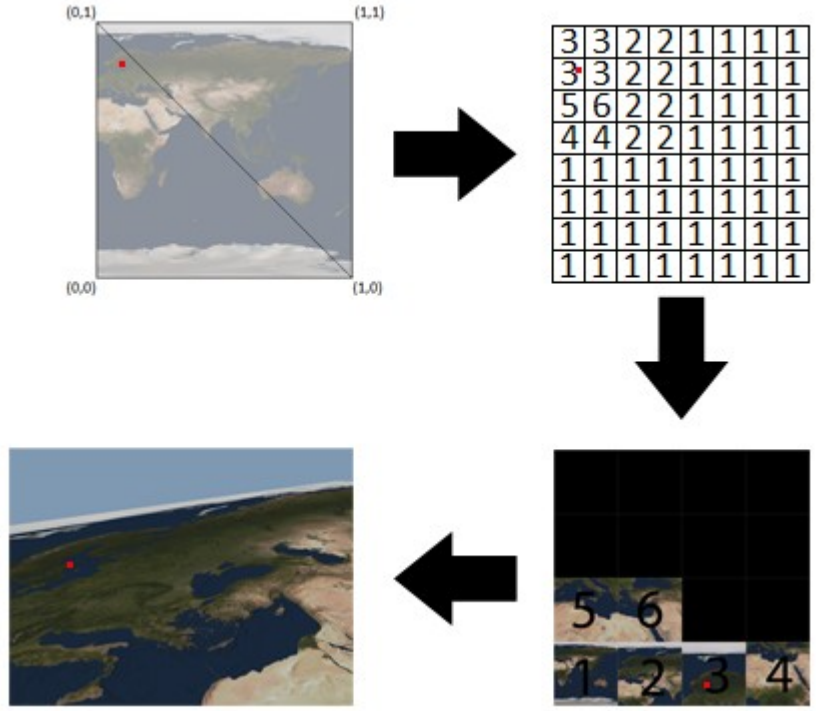

Figure 8: Each step in the rendering process.

3 Implementation

3.1 Pages needed for rendering

The first step of the implementation was to figure out how to know which pages needed to be used for rendering. I decided to use the method described earlier that stores mipmap level and page coordinate in each pixel's color.

I begun my first implementation by writing a fragment shader (see appendix A) that computes the mipmap level and the page index. The page index was a number starting from zero and ending at the number of pages at the specific mipmap level minus one. E.g. for the highest level the index went from zero to 2047, with zero being the bottom-left page and 2047 being the top-right page. In the final implementation I changed this indexing to x and y coordinates instead. This simplified some computations since the indexes ultimately had to be calculated to coordinates which meant more workload on the CPU.

This shader was bound in a first rendering pass where the scene was rendered (consisting of a quad only, UV-mapped with the virtual texture) to a texture using a frame buffer object. This texture had the exact same dimensions as the frame buffer. This texture now contained the information needed to determine the active pages.

The next step was to read out the information from the texture, this is done from the CPU side with glReadPixels function which takes in the data from the currently bound buffer (which in this case was my frame buffer object bound to my texture) and outputs a Java ByteBuffer with the pixel data. This step was very computationally expensive with the frame rate landing at around 30 fps, especially with higher screen resolutions since more pixels had to be read out, but it was good enough for my first implementation since my main goal was to get everything to work.

I found a way to optimize this step by using a Pixel buffer object instead of a Frame buffer object, though this did not increase the frame rate significantly. In my final implementation I optimized this step by downscaling the texture 8 times which significantly increased the frame rate to around 200-300 fps. Although this downscaling might mean we lose some critical information as described earlier, I did not notice any difference. I also tried to downscale the texture even more, when it was downscaled to about 16 times things started to behave a bit strange, with lower resolution parts popping up here and there, clearly due to pixels being lost in the downscaling.

Now I had a buffer to work with containing all the pixels telling me which page and mipmap level they were using. In the first implementation this buffer was definitely to large to loop through since it contain over 1.3 million pixels when using a resolution of 1280 x 1024, so I had to come up with some way of getting a more reasonable sized buffer that could be iterated at a CPU level.

With help from Tomas a solution was created that took advantage of the vertex pipeline on the GPU. By taking the ByteBuffer containing the pixels and putting it in a PBO using glBufferData I could then use this as input for a vertex shader and treat the pixels as vertices. These vertices were then rendered orthogonally using glDrawArrays onto the frame buffer. Before rendering the vertices the frame buffer dimensions was set using glOrtho. The width of the buffer was set equal to the number of pages in the highest mipmap level and the height was set equal to the number of mipmap levels. In my implementation this meant a dimension of 2048 x 6 pixels. Continuing with the active pages in the earlier example, when rendering the vertices into the frame buffer we end up with a texture looking like figure 9 below. The white dots represents the active pages. The lowest row represents the lowest mipmap level with only one page, in this case one page is active since it has a white dot, i.e. a vertex has been rendered there. The second lowest row represents the second lowest mipmap level etc.

Figure 9: The white texels represents the active pages.

The frame buffer was cleared to a black color and a fragment shader was also used that rendered each fragment white. glReadPixel was then used to transfer the pixel data into a ByteBuffer. This meant I now had a reasonably sized buffer that contained the necessary information needed to know the active pages. When examining this buffer I started at the highest row going left to right and checking if the pixel was white, if so I added it to an array which would end up holding all the active pages. For each row the number of pages decreased meaning that the number of iterations would be equal to the total number of pages, in my case 2730 iterations, which is a reasonably small amount to iterate on the CPU. The buffer is really a one dimensional array, so we have to offset the index with the width times the row when going down to the next row.

One limitation with this solution is that the maximum number of pages that we can use depends on how wide the frame buffer can be. When I begun preparing for the larger virtual texture I soon realized that this would mean a width of 16,777,216 pixels (the number of pages at the highest mipmap level, 4096 ^ 2), which clearly is impossible. This lead me back to the option of scaling down the texture containing information about the active pages, which now seemed like a better way around when using very large virtual textures, even though this was not the most accurate option. Thus, I ended up not using the above solution in my final implementation and instead using the buffer which held all the pixels from the downscaled texture. This buffer now contained 20480 pixels (since my downscaled buffer was 160 x 128) which was an acceptable amount to iterate on. The buffer was then iterated and the active pages was stored in an array containing the page coordinates and the mipmap level. The size of the array was equal to the number of pages the physical texture could hold.

In my first implementation I used a ordinary byte texture to store the active page information. This meant that I was limited to store values between 0 to 255 in each color channel limiting the number of pages to 256 x 256. When I begun testing with my larger virtual texture I realized I had to be able to store values up to 4095. To deal with this I used a 32 bit floating point texture instead (GL_RGB32F_ARB) which is able to store very large numbers.

3.2 Loading the textures

Now when I knew which pages I needed, the next step was to load the corresponding textures from disk. In my first implementation I did this in the same thread as everything else which stalled the program until the loading was finished, this was not acceptable so I had to implement some kind of system that used another thread for the loading. With some help from Martin I had a working class called TextureLoader up and running pretty fast. It uses Java's ExecutorService class which starts a new thread when a new texture needs to be loaded. The thread ends when it has finished loading the texture. The TextureLoader class now contains a texture that is ready to be loaded into video memory and when this texture has been fetched a new texture can be requested for loading.

When a texture is requested for loading and it has not yet been loading into video memory it loops through the lower mipmap levels to see if any texture from a lower level is already loaded into video memory, if so it is used as a fallback until the higher resolution texture has finished loaded. The texture from the coarsest mipmap level is always loaded in video memory and is never removed from it, this is to ensure that there is always a fallback when no other texture has been loaded yet.

When a texture is requested for loading its parents all the way up to the root in the mipmap tree is always set to active and requested for loading. This is a good way to cope with the prediction problem discussed earlier as it is more likely that a texture from a fairly high mipmap level already is loaded in video memory to be used as a fallback until the higher texture is ready. The downside with this is of course that more textures needs to be loaded and more video memory is used.

3.3 Page tables

The page tables was created as byte textures with glTexImage2D. In the earlier iterations of my implementation I also used a ByteBuffer for each page table to be updated with the active pages information and then I used glTexSubImage2D to fill the page table texture with the data from the ByteBuffer. This worked fine until I started working with the larger virtual texture. With this texture now containing over 16 million pixels at the finest mipmap level and with a page table requiring the same number of entries this lead to greatly decreasing performance when I called the glTexSubImage2D function, so I had to come up with another solution for this.

Once again I took advantage of the GPU vertex pipeline. Very similar to the solution described earlier when I rendered the pages as vertices to an off-screen texture I now rendered the active pages as vertices into its corresponding page table texture but this time a also needed to store information about the page coordinates in the physical texture and the mipmap level. For this I made use of the vertex colors storing the x and y coordinate in the red and blue channel and the mipmap level in the green channel, just like the method for the first step when determining the active pages.

3.3.1 Merging page tables

When each page table had been filled with the necessary information I had to combine these in some way to create a final page table with all the page tables from each mipmap level. Since all the page tables were stored as textures I decided to combine these using a shader that took them as input and then render these to another texture (see appendix B). I used this shader in a loop that begun with the first and coarsest mipmap level together with the level above. These two textures were then compared and if the texel in the higher mipmap level had a non black color this was then rendered to the output texture. In the next iteration the texture output from the last loop was used together with the next mipmap level (level 3), and the same comparison was done. This process was repeated until the final mipmap level was reached producing a final texture that had all the page table textures combined (see figure 7).

3.4 Rendering

The rendering was done using a shader that took the physical texture and the final page table texture as input (see appendix C). The page table texture was first used to calculate the coordinates in the physical texture that points to the bottom-left corner of the page. The offset within the page is then calculated and added together with this coordinate. This points the final texel that we want to render.

A problem with the pages being offset was encountered in the earlier implementations. The offset increased more and more after each page. This was due to a round of error in the shader and was solved by first using the floor function to take the float value down to a whole number and then 0.5 was added.

4 Discussion

I decided pretty early to use a virtual texture that was already split into pages, but I had in mind I would try out the method of reading out parts from a single large texture later on. At first I used a texture with size 65536 x 32768 pixels divided into 2048 pages each with size 1024 x 1024 pixels at the highest mipmap level. The following lower levels had 512, 128, 32, 8 and finally 2 pages at the lowest level, thus a total of 2730 pages. This was a satellite image of the earth that I found on the internet [7].

After my first implementation the guys at Agency9 suggested to try using an even larger texture to find out the limitation of how large the texture could be. They supplied me with satellite image data of Sweden. These images had almost the same file structure as the earth texture I was already using which made it easy for me to try it out. The difference was that this image had 15 mipmap levels instead of 6, each page had a size of 512 x 512 pixels, and they had the same number of pages both vertically and horizontally. Thus, at the highest mipmap level this meant a total image size of 8,388,608 ^ 2 pixels which would be more than enough to try out the limitations.

I soon realized that this would require a texture with size 16384 ^ 2 pixels to store the page table data, which would be stored in video memory. This would not work because the maximal size that a texture can have in OpenGL was limited to 4096 on my computer. Thus, the limit of my virtual texture would be 4096 ^ 2 or 2,097,152 ^ 2 pixels. You could probably go around this limitation if you wanted to have an even larger texture by dividing the page table into several smaller page tables or by using larger page sizes which would reduce the number of pages.

In my final implementation the size of the physical texture was set to 6 x 6 pages or 3072 x 3072 pixels. This was big enough to hold all active pages when they where at the most (although there might be some circumstance where it becomes full). Thus, I did not care to implement any kind of LRU system to deal with the problem when the physical texture becomes full. When this happens the texture requested to be loaded is discarded.

In one of the earlier implementations I did not use a loop to iterate through the page table texture, instead I took all the textures as input to the shader. This meant that my maximum number of mipmap levels was limited to the maximum number of textures that the shader could take as input. Since my larger texture had 15 mipmap levels and since most graphics cards only supports 8 textures this solution was later replaced with the one described earlier.

I also tested the method of keeping the virtual texture as a single texture on disk. To do this I used the Java class ImageInputStream to make the image ready for reading. The class ImageReadParam was then used to set the image region to read from, using the function setSourceRegion. Finally the BufferedImage function read was used to read out the image data. Unfortunately this whole process was extremely slow and since this has to be done every time a new texture needs to be loaded I quickly decided to abandon it.

4.1 Conclusion

Even though there are some problems with Virtual texturing I think it is a very interesting technique for rendering very large textures. Since many of the new games that are released have huge outdoor environments that needs large textures I think this technique will be seen a lot in future games.

5 References

[1] http://en.wikipedia.org/wiki/Virtual_memory

[2] Mittring, M. 2008, Advanced Virtual Texture Topics, N. Tatarchuk, Crytek GmbH, 24, 36, 38

[3] Williams, L. 1983, Pyramidal Parametrics, Computer Graphics Laboratory New York Institute of Technology

[4] Shirman, L. & Kamen, Y. 1998, A New Look At Mipmap Level Estimation Techniques, Epson Research and Development, Inc.

[5] Barrett, S. 2008, Sparse Virtual Textures, GDC San Francisco CA, http://silverspaceship.com/src/svt

[6] http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

[7] http://www.celestiamotherlode.net/catalog/earth.php

6 Appendix

A: f_calcpage.glsl

```
uniform int pageCols; // number of pages horizontally (columns)
uniform int pageRows; // number of pages vertically (rows)
uniform int texWidth; // width of the virtual texture
uniform int texHeight; // height of the virtual texture
uniform int maxMipLevel; // the highest mipmap level
uniform int downScale; // how much the screen is downscaled

void main()
{
	// computes the mipmap level, ranging from 0 to maxMipLevel
	vec2 dx = dFdx(gl_TexCoord[0] * texWidth / downScale);
	vec2 dy = dFdy(gl_TexCoord[0] * texHeight / downScale);
	float d = max(dot(dx, dx), dot(dy, dy));
	float miplevel = log2(sqrt(d)) + 0.5;
	miplevel = clamp(miplevel, 0.0, maxMipLevel);
	miplevel = maxMipLevel - floor(miplevel);

	// compute the column number (x) and the row (y) as integers
	int x = gl_TexCoord[0].x * exp2(miplevel);
	int y = gl_TexCoord[0].y * exp2(miplevel);

	x = clamp(x, 0, exp2(miplevel) - 1);
	y = clamp(y, 0, exp2(miplevel) - 1);

	gl_FragColor.r = x;
	gl_FragColor.g = y;
	gl_FragColor.b = miplevel;
}
```

B: f_mergepagetable.glsl

```
uniform sampler2D sampler1;
uniform sampler2D sampler2;

void main()
{
	vec3 final = vec3(0.0, 0.0, 0.0);

	vec3 tex1 = texture2D(sampler1, gl_TexCoord[0].xy).xyz;
	vec3 tex2 = texture2D(sampler2, gl_TexCoord[0].xy).xyz;

	if (tex2 != 0.0)
		final = tex2;
	else if (tex1 != 0.0)
		final = tex1;

	gl_FragColor.xyz = final;
	gl_FragColor.w = 1.0;
}
```

C: f_translate_vtc.glsl

```
// this shader translates the virtual texture coordinate to a physical coordinate

uniform int pageCols;
uniform int pageRows;
uniform int physPageCols;
uniform int physPageRows;
uniform int maxMipLevel;

uniform sampler2D pageTableTex; // the page table texture
uniform sampler2D physicalTex; // the physical texture

varying vec3 V; // view vector
varying vec3 N; // normal vector
varying vec3 L; // light vector


void main()
{
	// get page coordinates (bottom left corner) from the page table texture
	vec3 pageCoord = texture2D(pageTableTex, gl_TexCoord[0].xy);

	// calculate the physical texture coordinates (without the offset within the page)
	// the floor and + 0.5 are used to avoid round off errors
	vec2 physPageTC = vec2(0.0, 0.0);
	physPageTC.x = floor((pageCoord.r * physPageCols) + 0.5);
	physPageTC.x = physPageTC.x / physPageCols;
	physPageTC.y = floor((pageCoord.g * physPageRows) + 0.5);
	physPageTC.y = physPageTC.y / physPageRows;

	// calculate the offset within the page
	vec2 withinPageTC;
	int mipPageCols = exp2(floor((pageCoord.b * maxMipLevel) + 0.5));
	withinPageTC.x = mipPageCols * gl_TexCoord[0].x;
	withinPageTC.y = mipPageCols * gl_TexCoord[0].y;
	withinPageTC = fract(withinPageTC);
	withinPageTC.x = withinPageTC.x / physPageCols;
	withinPageTC.y = withinPageTC.y / physPageRows;

	// add the coords together to get the physical texture coord that
	// corresponds to the virtual coord
	vec2 physTC = physPageTC + withinPageTC;

	// sample from the physical texture
	vec3 texture = texture2D(physicalTex, physTC);


	// finally do some nice phong shading
	V = normalize(V);
	L = normalize(L);

	float dotNL = dot(N, L);

	vec3 diffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse * max(dotNL, 0.0);

	vec3 specular = 0.0;

	if (dotNL > 0.0)
	{
		vec3 R = normalize(reflect(L, N));
		specular = pow(max(dot(R, V), 0.0), 200.0);
	}

	gl_FragColor.xyz = (diffuse + gl_LightModel.ambient.xyz) * texture + specular;
	gl_FragColor.w = 1.0;
}
```