ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
COMPUTER SCIENCE - MASTER

Visualisation of high-resolution 3D images on a web application

Student:
Matthieu Rudelle

Supervisor:
Loïc Baboulaz

June 2015

Abstract

The web is a complicated world where users are demanding and where small optimizations can make a big difference. Users want the information fast, but they also want the best quality. In terms of images, there is a well documented solution to that; Virtual Texturing. But when it comes to browsers and web application very few documentation exist and they lack technical details. This leads to my project which consists in the implementation of a Virtual Texturing engine for a web application. And this paper documents the details of my journey.

CONTENTS

1 Introduction

During Summer 2014 I had the opportunity to work for the eFacsimile project on the development of a web-application for visualization of 3D artworks. The application features a gallery and an artwork viewer. The latter allows the user to change the light condition (orientation, intensity, specularity, temperature...) on the painting and see how this one reacts according to different light conditions, thus recreating a realistic representation of the item as it would appear when exposed in a particular environment. The app also represent the relief of the artwork in 3D.

The project was very exciting but we shortly faced a big issue. In order to make the painting as realistic as possible, we had to increase the resolution of the texture, which means more texture to fit into memory and also more data to send over the network before the client can actually see your item.

To solve this problem, I proposed to use Virtual Texturing (will be refereed as VT for the rest of this paper), but I didn't have enough time to develop it during my internship. So Loïc Baboulaz proposed me to work on it during a semester project. Turns out it involved a lot of work and architecture design but the result is surprisingly smooth!

VT solves the problem of high resolution textures, but similar techniques exist for geometry with dense vertex meshes, they created adaptive meshes and we will shortly talk about them in this report.

The idea behind VT is that the Level Of Details (LOD) required to render a scene is not uniform, indeed, parts of the texture far from the viewer require a much less resolution than a part that is close-by. Thus we can partition the texture space into small cells of different resolution depending on how they lie on the screen. The idea here is to have cells of fixed dimensions —in pixels— while the space taken by the cell —in the scene— is variable. By doing so we reduced the amount of pixels to be stored and rendered without altering the visual quality.

We will structure this rapport in two parts. The first gives a final overview of how the current implementation is working along with several technical details on this final implementation. Then the second part will detail my journey toward the implementation of this project, I will thus speak of the challenges and issues that I solved, along with the very interesting constraints I had to cope with, mostly due to the large size of data I handled.

2 Related Work

Virtual Texturing (VT) is now commonly used in game engine like id Tech 5 used in the game *Rage*[1], it allows to efficiently render huge textures —often combined with clip-map[6, section 4]— while keeping the GPU load fairly low.

Though, very little work has been documented and available online about porting this technique to Web Browsers and most of them use facilitating libraries such as Three.js[2], but it didn't suit our needs, because, due to the nature of the texture, we need an ad-hoc solution to use Polynomial Texture Mapping (PTM) textures[3]. However I based part of my architectural design on the Master Thesis of Sven Andersson and Jhonny Göransson [11] that discuss more the performance of different implementations rather than technical details on the implementation.

I also largely inspired my work and research on the following work and documentation:

- Picking in WebGL from *WebGL Beginner's Guide* [4, chap. 8]
- Holger Dammertz's blog on *sparse Virtual Texturing* [6]
- id Software's conference on *id Tech 5 challenges* [7]
- Mayer's thorough thesis on Virtual Texturing [1]

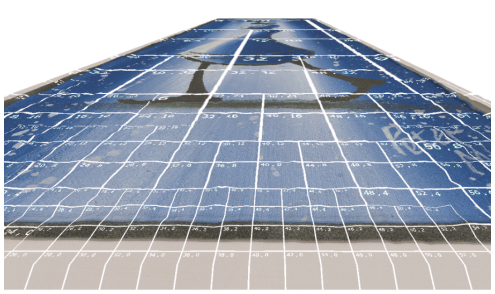
When doing researches on adaptative meshes, I mostly worked on papers discussing *BDAM: batch dynamic adaptative meshes* [10] and other implementations such as *ROAM: Real-time Optimally Adapting Meshes* [9].

3 How it Works

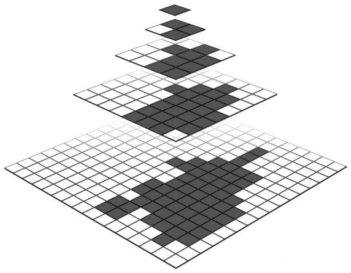
As explained above, the main idea is to adapt the Level of Details (LOD) according to the location of the pixel in the view. On fig. 3.1a we can see how the texture space is partitionned into cells and how they roughly all occupy the same surface on the screen. In this project we partition the texture space into cells of size 256×256 pixels. Cells can have various resolution level, the last one –level 8– features the best resolution, while the unique cell at level 0 contains the entire texture with a very low resolution. Each level is exactly half the size of the following one, figure 3.1b represent this layered structure.

In order to create this data structure in a preprocessing phase, a script crops and scales the original image into tiles of 256×256 pixels in 9 folders –corresponding to the 9 different

[1] `http://mrelusive.com/publications/presentations/2010_gtc/GTC_2010_Virtual_Textures.pdf`
[2] `https://github.com/elfrank/virtual-texturing`
[3] `http://www.hpl.hp.com/research/ptm/papers/ptm.pdf`



(a) How VT tiles lay on the canvas

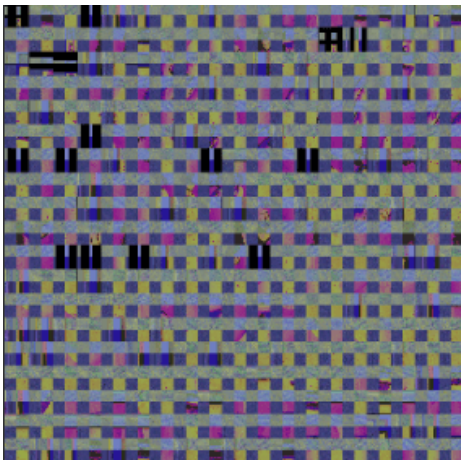


(b) Example of the layered architecture

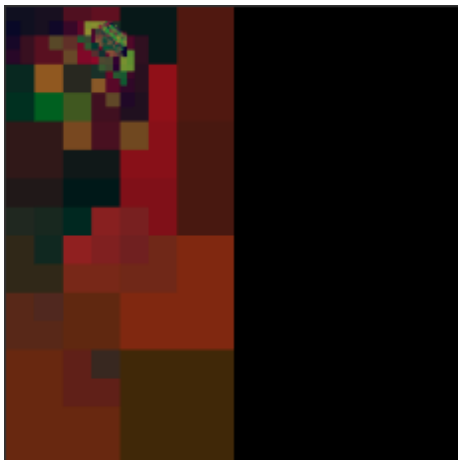
Figure 3.1: visualization of the tilling system

LOD–, they are then stored on the server to be efficiently served like static files. The script uses ImageMagick[4] and is tailored to handle images that do not fit in memory –generally several GB–. When used on texture of size 65536 × 65536 pixels, which is the maximum handled by this engine (further explanation on this in section 4.1), it generates up to 87381 tiles which represents roughly 10GB in PNG images. (section 4.2 gives more detail about this script)

On render time, the client's browser holds a list of the tiles and LOD it needs to render a frame. When new cells are needed they are allocated in a **Page Cache** –that acts as a buffer to store cells– and passed to a WebWorker to be downloaded from the server. Once downloaded the new tile is written to the Page Cache and added to the **Indirection Table** –that is used to resolve the position of a tile in the cache–. Those two texture are then regularly forwarded to WebGL in order to update the view.



(a) Example of a page cache



(b) Example of an indirection table

Figure 3.2: Only two textures are used for rendering

[4] `http://www.imagemagick.org`

The Page Cache and the Indirection Table are the only textures used to render the model. Fig. 3.2a shows an example of page cache filled with 19×19 tiles, and fig. 3.2b shows a typical indirection table where we can see the position of the viewer (close to the painting) and the direction of its look toward the bottom left. On the latter, x and y coordinates are represented with the red and green components, while the mip map level is given by the blue component. The coordinate translation for each pixel is performed in real-time, directly on the computer's GPU (performed by the shaders) using the following process:

- We need to render the texture at position $coord_{TX}$ in the texture space.
- We fetch $color_{ID}$ the color in the indirectionTable at position $coord_{TX}$.
- The red, green and blue coordinate of $color_{ID}$ are respectivly the x and y coordinates of the cell in the PageCache along with the LOD of this cell.
- We find the coordinate in the page cache $coord_{PC}$ as a function of $color_{ID}$.
- We return $color_{PC}$ the color found in the Page Cache at position $coord_{PC}$

The process is actually simplified here, as a cell in the PC is partitioned in 4 smaller textures, but we provide more details in section 4.6.

In order to keep the list of cells needed to render the scene, WebGL paints a third texture, the **Feedback Buffer**, that is rendered offscreen (not used for scene rendering) which is assigned a particular shader. This shader uses the x and y derivatives to compute the LOD requested by the pixel, then it stores the x coordinate, y coordinate and LOD as RGB components of the pixel. This texture is then read by the CPU which travels every pixel and outputs the coordinates and LOD of cells needed to render the current view. As reading every pixel is very slow on CPU, we use a feedback buffer that is 1/8th the size of the screen. This ended up to be fast enough to be traveled on render time.

Fig. 3.3 depicts the overall architecture of the VT engine, green blocks represent classes I implemented during my previous internship but that were modified to work with the new VT engine, while redish and orange blocks are parts I implemented from scratch for this project. The numbered arrows follows the typical path from the first render to the time needed cells are printed on the screen. Red arrows follow the path of a typical VT workflow. The `Painting_Viewer` is in charge of periodically order the rendering of the scene, the order is given to `VT_Scheduler` that handles both rendering the scene and instantiate a new VT loop. Indeed, at each iteration it will trigger a on-screen render in `PTM_GL` and will, sometimes –only when the previous VT loop has ended–, render the feedback buffer to discover new tiles to request.

Still on figure 3.3, redish boxes represent **Web Workers**[5], they run on a different threads than the rest of JavaScript in order not to block the render loop while doing costly operations,

[5] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`

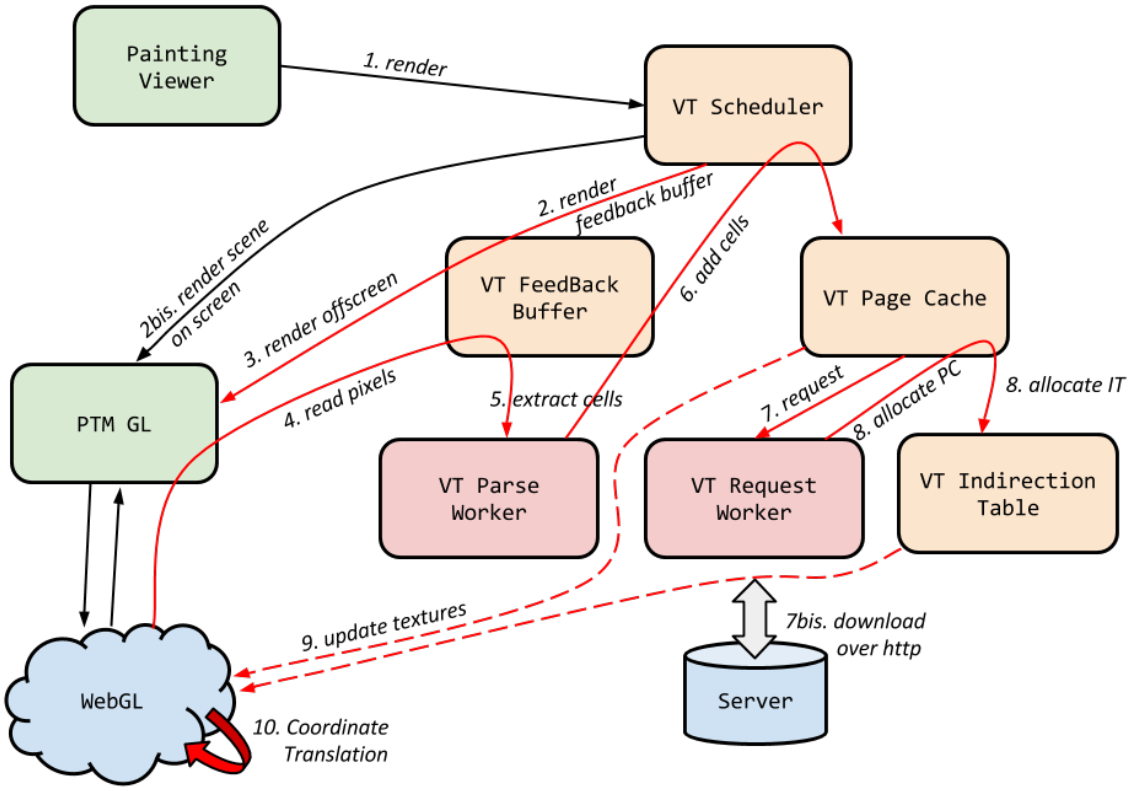


Figure 3.3: Global architecture of the code

in the manner of a background job.

Finally, the `VT_Cell` class is not present on this graph and represent a Cell by its location and mip map level. It contains helpers methods such as url resolver to find the path to this cell on the server.

In terms of programming language, most of the project is implemented in JavaScript using WebGL[6], Web Workers[7], Canvas[8], Underscore.js[9] and XMLHttpRequest[10] APIs. The tenth stage on fig. 3.3 is performed in GLSL [5], which is an adaptation of C for GL shaders. However, the preprocessing –partition textures into cells in 9 different layers– is handled by a Shell Script utilizing ImageMagick to manipulate the images. Finally, the server-side is handled by Django[11], leveraging Python, but very little work was done on the framework to adapt the existing solution to VT.

[6] `http://webglfundamentals.org/`
[7] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`
[8] `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial`
[9] `http://underscorejs.org/`
[10] `https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest`
[11] `https://www.djangoproject.com/`

This is a quite simplified view of the engine, and many problem raised, for instance due to the huge size of images we handle, the complexity of textures needed to render the PTM model and so on. In the following section we will see more technical aspects that made this project more complicated but also much more interesting!

4 Challenges & Issues

4.1 Feedback Buffer

This part was quite complicated to get passed. It was my project's first step but yet it was a blocking one. Indeed I had to discover in details the inner workings of WebGL, I needed WebGL to draw the same scene with two different shaders and in two different contexts –on screen and off screen–, I also needed to read the off-screen buffer from the CPU to be parsed.

The theory about how to compute the mip map level is rather simple, the shader is given the texture coordinate and is able to compute it's derivative thanks to WebGL's extension: `GL_OES_standard_derivatives`. Given $v \in [0,1]^2$ the 2-D vector representing the texture coordinate and $t_{size} \in \mathbf{N}^2$ the 2-D vector representing the texture's dimension in pixel, we compute the mip map level[11, p. 14] as:

$$MipMapLVL = log\left(max\left(\left|\frac{\partial(v \circ t_{size})}{\partial x}\right|, \left|\frac{\partial(v \circ t_{size})}{\partial y}\right|\right)\right) + mip\_bias \tag{4.1}$$

Here `mip_bias` is determined empirically as a trade-of between visual quality and memory usage.

While this value is stored in the blue component of the pixel, red and green are occupied by the x and y coordinate of the bottom left corner of the cell according to it's size. Thus, at level 8, cells are numbered $(0.0), (0,1), (0,2)...(0,255)...(255,255)$ while at level 7, they are numbered $(0.0), (0,2), (0,4)...(0,254)...(254,254)$. No matter the resolution of a cell, it will always be refereed to by the coordinate of the leftmost, lowest cell from level 8 that fit into it's span. We had the idea from id Software's conference[7, p.11] and it allows to keep consistent indexing among every part of the process.

We can see above that we are restricted to 256×256 addressable tiles at the highest resolution level. This is because, in WebGL, color components are stored on 8 bits each**??**, which limits us to a $[0,255]$ range. However it is possible to bypass such limitation for instance by using the 4 unused bits of the blue component or even using a second off-screen buffer but this is not in the scope of this project. Moreover, by using tiles of 256×256 px we can already address up to 4.2 Billion pixels texture.

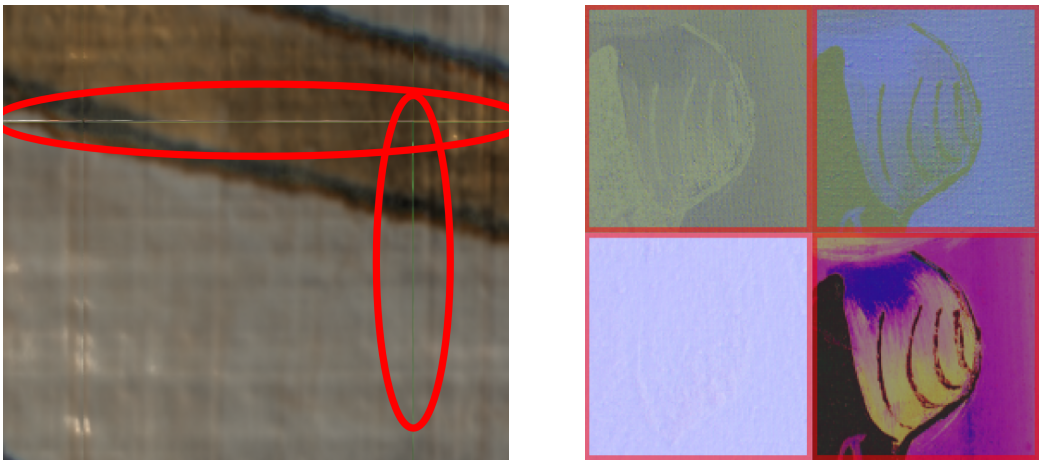
4.2 TILLING SCRIPT

The idea behind this part is very simple: *"Starting from level 8, cut the texture into tiles of* 256×256*px, scaling down the texture by* 50% *between each level, until we reach level 0"*. But it ended up to be much more complicated and it took me more time than expected to tackle it.

In the first place I started to search for programs or APIs that would do the job efficiently in a generic manner, but, because of the project's nature, I had to pay attention to the following details:

- We need to handle various image sizes, even though they are limited to 65536px on each side.
- Given that we render images using PTM**??** we need to ship every parameter –4 sub-images– with each tile to ensure consistency while rendering.
- In order to avoid texture bleeding like on fig. 4.1a cells should have a border of 2px on each side –leaving transparent pixels for cells on the image sides.
- It must be easy to resolve the path to a cell by using it's level and location

Thus a typical cell is square of size $2*(128+2*2) = 264$px and looks roughly like fig. 4.1b. Where borders have been highlighted in red and the 4 sub-images are clearly visible. The chosen format for the output folder is `<mipMapLvl>/vt_tile_<tilePosX>_<tilePosY>.png`.



(a) Example of texture bleeding

(b) Typical tile with border highlighted

Figure 4.1: Only two textures are used for rendering

Thus, I first considered options that create a full VT architecture like libvips[8] that creates

tiles like DeepZoom[12] or GoogleMaps[13]. They are very efficient at handling huge images but they are useless as we need an ad-hoc solution to ship the 4 parameters in one tile. After considering tools such as Gimp Batch Mode[14] I settled for ImageMagick which is a lightweight and renowned script API to automate image manipulation.

Working with images as big as several GigaBytes was challenging, indeed they do not fit in memory (or at least not on standard computers). For that matter we create `.mpc` files[3], that are uncompressed and memory-mapped for rather "quick" access. The generated image takes roughly 10GB but can be manipulated directly from disk without using any memory. This solution worked well, the original image is cropped into multiple smaller tiles, then it is scaled to half its dimensions and cropped again, etc... But when I was able to work with really bigger images, resizing images became exceptionally long because the `-resize` option is actually scanning the image in two passes –one vertical and one horizontal– and the vertical one is very inefficient in terms of data locality[15]. However, using the option `-scale` instead of `-resize` solved the problem as it works more efficiently on large images[2].

This preprocessing it quite long, mostly because every read is performed from disk and that a lot of images need to be created (lots of overhead). For instance, when working on an image of size 18718×53170 pixels the script took 131 minutes to complete, generating 20585 images summing up to 2.4GB.

4.3 Page Cache

The page cache is an interesting concept, it holds every tiles needed to render a scene, but without any ordering or location specific. It is easy to partition as every cell has the same size, but once it is full we need some sort of rule to replace cells.

One dummy solution would be to replace the older cell (the age being defined as the time –in frames– since it was last rendered) by the new one. But, as the application runs in real time we use an optimized version that produces a roughly similar result but which is much faster; we travel the cache by row and column and if the cell's age is above a certain threshold, we replace it by the new one, and we continue to travel the cache when a newer cell arrives. However, when we loop over the cache, meaning that no cell was old enough, we define the new threshold as:

$$threshold_{new} = \left\lfloor \frac{threshold_{old}}{2} \right\rfloor \tag{4.2}$$

[12] `https://msdn.microsoft.com/en-us/library/cc645050\%28VS.95\%29.aspx`
[13] `https://www.google.ch/maps`
[14] `http://www.gimp.org/tutorials/Basic_Batch/`
[15] `http://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=27671`

We use this formula to fit best the distribution of cell's ages in the page cache and avoid doing useless loops that could create stalls in the animation.

One final optimization is done when adding multiple cells, the threshold is kept unchanged between each cell even though it can be lowered while allocating a new cell. And when a new batch of cells arrive, it is reset to the default value.

In terms of implementation, the page cache is a canvas stored by the class, from which parts are erased and drawn before being forwarded in full to WebGL in order to update the view. An important point to note is that the position $(0,0)$ is constantly occupied by the tile containing the entire texture at the highest resolution. This cell cannot be unallocated except when we flush the cache, in which case this cell is directly reallocated at position $(0,0)$. The level satisfying this condition (one tile contain the entire texture at highest resolution) is computed by:

$$bottom\_level = LEVEL\_MAX - \left\lceil log_2 \left(\frac{max\left(Texture\_Size_x, Texture\_Size_y\right)}{CELL\_SIZE \times 2} \right) \right\rceil \quad (4.3)$$

Where $CELL\_SIZE = 256$ pixels and $LEVEL\_MAX = 8$.

4.4 Cell Request

Cell requesting is a critical process in the path of a typical VT work-flow. And there is two important points to keep in mind while implementing it:

- How to manage parallel requests
- How to order the request in a way that we best reflect the user preferences

Concerning the first part, the initial solution was starting to request a cell as soon as it was discovered, creating a fairly big mess with tons of requests going on at the same time. And blocking completely the animation. To solve this problem a centralized downloader was implemented to handle requests one by one. Then to speed up the "stream" we set a fixed number of request in parallel –we empirically found best performances when using 4 parallel requesters–, all working with the same stack of request, similar to a pool of jobs. This centralization allowed another optimization that consists in canceling a download when the cell is unallocated before it is actually requested. This is fairly frequent and allows us to avoid useless download while prioritizing more recent downloads.

In order to reduce stalls in the animation, the requesters are working on a Web Worker. Thus moving the request processing to a different thread than the one rendering the scene. To download images we limited the computation overhead –compared to using DOM elements[11,

section 3.5]– by using JavaScript's XMLHttpRequest. And, as Web Workers interact only with messaging (similar to actors in Scala) we use blobs[16] available in the new HTML5 API. They allow an easier manipulation of files coming from the network.

Then, concerning the ordering of the request. Compared to naive methods using a queue, we actually prioritize recent downloads by using a **stack**. Thus when new cells are discovered by the VT engine they are directly put on top of the list. That makes sense because when the user moves quickly from places to places, what he cares the more about is the parts of the texture located at its last position. Also, combined with the technique seen in the paragraph above, that helps canceling requests of cells that became out of scope before they are requested.

Finally, in order to improve further the user experience, we compute the distance of the cell from the center of the screen and use it to first download cells closer to the middle of the view (usually the viewing position of the user). We also favor downloads of cells with lower resolution. Indeed, they occupy more space in the view and that increases consistency between visible cell's resolution.

The final order is given by:

- Recent requests first (they come in batch so we can further order them)
- Cells with lower resolution first then the more precise ones
- Cells closer to the center of the screen first followed by cells further way

4.5 Indirection Table

At a first glance, the Indirection Table (IT) seems to be easy to implement, we just need to draw squares of a particular color on the space spanned by the by the cell. The color components are actually given by the x and y coordinates along with the LOD. And we need to make sure bigger cells are drawn first so that they do not overlap smaller ones that feature better a resolution.

Problems actually occur when we want to update the cache without having to redraw every tiles because updates in the IT are very frequent. Thus I first tried to implement several optimized algorithms to redraw only a small part of the table at each iteration but it became fairly complicated to handle the 9 layers altogether, and asymptotic times were often exponential. Finally we take advantage of the optimization of the HTML5 canvas API by holding 9 layers on which parts can be drawn and erased linearly. Then, when we need to update the WebGL, we just have to merge the 9 layers on one canvas and forward it to WebGL.

[16] `https://developer.mozilla.org/en/docs/Web/API/Blob`

4.6 Coordinate Translation

This section (like the tile scripting one) is very different from the others in the sense that it is not implemented in JavaScript and thus not easy to debug. Here, Coordinate Translation refers to the process of transforming a coordinate in the texture space into a coordinate –actually 4 for every parameters of PTM– into the Page Cache space. And the entire process is handled by the GPU –which is very efficient at running a large quantity of replicas of a particular process in parallel– to enable real-time animation.

It was very hard to stay focused and keep a clear idea of the meaning of each value in the process. Indeed we are using variables in multiple "coordinate worlds" like color coordinates, texture coordinates, pixel coordinates. Plus the entity to which they refer: the indirection table, the page cache, the original texture ...

However it was very interesting to learn how to use graphic languages in shader, because it offers limitless possibilities [5]. Virtual Texturing yet being a very impressive one! In terms of technical details I found one feature quite powerful: the ability to pass argument as "output variable". Indeed, adding *in, out* or *in out* to the parameter classifier you can change the way a parameter is accessed and thus add parameters as output variables. Using this technique was helpful when creating the function returning the location (in the page cache) of the 4 sub-textures used to render one pixel.

The process to compute the color of one pixel is explained in section 3. However we can distinguish two main steps in the resolving locations from texture coordinates to to coordinates in the page cache :

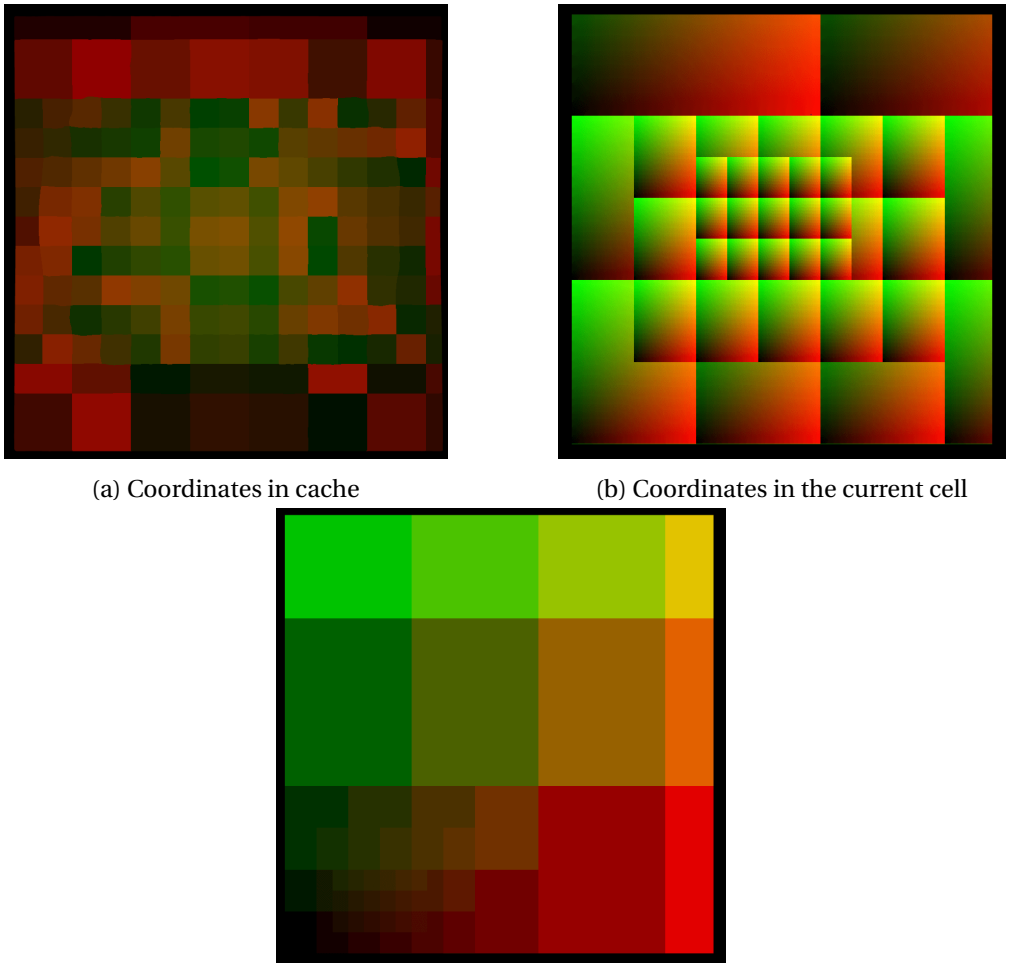
- Computing the origin (in the page cache coordinates) of the cell corresponding to a particular location
- Computing the coordinates in this cell for this particular location.

Those two steps are represented in fig. 4.2a and fig. 4.2b. While fig. 4.2c is the origin of the current tile and is used to compute 4.2b. In those 3 graphs x and y coordinates are represented by red and green components.

4.7 Real-Time Rendering

This part of the project was very interesting because this is where you come to breaking down exactly what your engine does and how long each stage takes. This is very important to understand the main bottlenecks and blocking functions of a program. And the result is quite surprising as the optimization really gave smooth animations.

In the previous section we talked about optimization like moving parts of the algorithm



(a) Coordinates in cache

(b) Coordinates in the current cell

(c) Origin of the current cell

Figure 4.2: Steps used to resolve the translation into Page Cache coordinates

to background threads –called web workers in JavaScript– or performing operations in a distributed manner. Sections 4.4 and 4.5 gives already plenty of details about how the code was optimized to give real-time animations. So here we will talk about the remaining optimization and glitch we fixed to get a smooth animation. But first, in order to track optimization glitches and costly methods we used the Google Chrome Profiler that gives a meaningful insight on what is doing the browser at any time. Example output of the feature can be seen in fig. 4.3.

In this example we see that a lot of time is lost drawing the page cache. This is due to the page cache being drawn into a smaller on-screen canvas quite frequently. It was used to see the content of the cache at any time but was fairly costly so this is now an opt-in option for developers.

This kind of graph also helped finding a mistake where the indirection table and the page

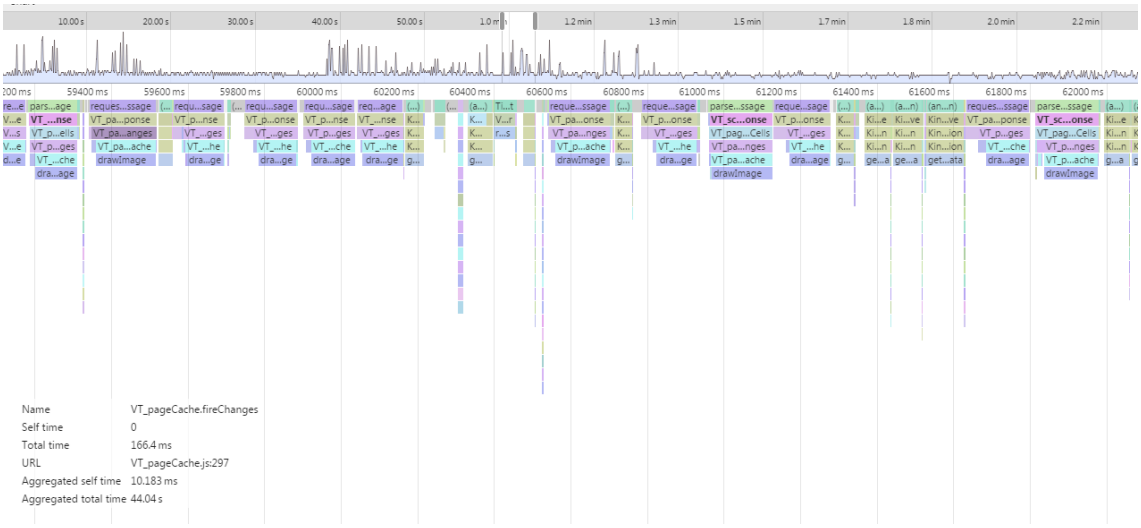


Figure 4.3: Profiling example of the main thread

cache were redrawn every time we were unallocating a new cell. Thus batch of cell were slowing down the render loop, creating stalls when many new cells where discovered. Now the problem is solved and the cache (and indirection table) is forwarded to WebGL only once per batch.

One other important optimization we implemented was to move the feedback buffer processing into a Web Worker that is taking an array of float as input message and emits a list of the cells –with distance to the center of the screen– to be displayed. This actually played an important role in the smoothness of the app.

5 Next Steps

Despite my motivation and eagerness to cover the entire problem of virtual texturing during my project, I was unable to solve one particular issue. From the beginning I planned to implement an adaptive meshes algorithm but apart from studying the subject for more than a week, I was unable to implement a working solution. Indeed the problem is tough, many papers have been written on this and one particularly interested me: Real-time Optimally Adapting Meshes [9]. Several explanations of the algorithm as well as implementations are available online but I would like to port it on JavaScript and complete my Virtual Texturing engine.

Another point I did not have time to cover is to handle cache thrashing[17] which occurs when more cells are required to render a scene than the space available in cache. This can easily

[17] `http://en.wikipedia.org/wiki/Thrashing_\%28computer_science\%29`

occur in mobile devices where the internal memory is quite low. The solution to this problem is to create a system to detect such problems and lower the resolution of the view to keep the entire scene in the page cache.

Finally, some optimization still needs to be performed on the app. It does not suffer from important stalls anymore but is still very heavy in terms of GPU and CPU usage. Thus more analyzing and optimization could further increase the usability of the app, notably on devices with low resources.

6 Conclusion

Virtual texturing is a really vast and complete technology where –I had the opportunity to realize– many skills are required and rigorous architecture design is not optional. I personally had a great time discovering the inner workings of a virtual texture engine along with implementing and tweaking my own version. I believe the scope of this project is really encouraging to take initiatives in the implementation and architecture design of the app, while being able to assess resulting progressions quite easily.

The learning outcome of my project is not only technical skills about virtual texturing or JavaScript's hidden treasures but more importantly the confidence to make decisions based on researches and personal experiences without which the result would maybe not be as impressive as it is now.

Concerning the application itself, results are encouraging, the app works quite smoothly and handles fairly big images. It is close to ready –if not actually ready– to be used in production by actual users. Though some works still needs to be done in order to finalize the optimization and allow more precise geometry.

References

[1] Albert Julian Mayer, *Virtual Texturing*, 2010, URL: `http://www.cg.tuwien.ac.at/research/publications/2010/Mayer-2010-VT/Mayer-2010-VT-Thesis.pdf`.

[2] Anthony Thyssen, Image Magick, *ImageMagick v6 Examples – Resize or Scaling (General Techniques)*, 2012, URL: `http://www.imagemagick.org/Usage/resize/`.

[3] Anthony Thyssen, Image Magick, *ImageMagick v6 Examples – Image File Handling – mpc*, 2013, URL: `http://www.imagemagick.org/Usage/files/#mpc`.

[4] Brandon Jones, Diego Cantor *WebGL Beginner's Guide*, 2012, URL: `https:`

//www.safaribooksonline.com/library/view/webgl-beginners-guide/9781849691727/.

[5] David J. Eck, GraphicsNotes 2013, *Section 19: The Shader Language for WebGL*, 2013, URL: http://math.hws.edu/eck/cs424/notes2013/19_GLSL.html.

[6] Holger Dammertz, *Sparse Virtual Texturing*, 2012, URL: http://holger.dammertz.org/stuff/notes_VirtualTexturing.html.

[7] J.M.P. van Waveren, *id Tech 5 Challenges*, SIGGRAPH 2009, URL: http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf.

[8] John Cupitt *Making DeepZoom, Zoomify and Google Maps image pyramids with vips*, 2013, URL: http://libvips.blogspot.co.uk/2013/03/making-deepzoom-zoomify-and-google-maps.html.

[9] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Millery, Charles Aldrich, Mark B. Mineev-Weinstein *ROAMing Terrain: Real-time Optimally Adapting Meshes*, 1997, URL: https://graphics.llnl.gov/ROAM/roam.pdf.

[10] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, R. Scopigno, *BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization*, 2010, URL: http://www.crs4.it/vic/data/papers/eg2003-bdam.pdf.

[11] Sven Andersson, Jhonny Göransson, *Virtual Texturing with WebGL*, 2012, URL: http://publications.lib.chalmers.se/records/fulltext/155126.pdf.