CHALMERS

Virtual Geometry Textures

Implications of combining Virtual Texturing with geometry textures

Master of Science Thesis

HENRIK SCHULZE NILSSON

Compouter science department
Division of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, February 2014

Virtual Geometry Images
Implications of combining Virtual Texturing with geometry textures

HENRIK SCHULZE NILSSON

Examiner: Ulf Assarsson
Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden February 2014

Acknowledgements

Sven Andersson, Jhonny Göransson, Ulf Assarsson – Mindary, Sweden: Porcupine Engine©
CRS4 and ISTI-CNR, Italy: Adaptive Quad Patches demo

Abstract

This thesis investigates the implications and expected performance of a Virtual Geometry Textures system, defined as a system that uses images to stream and render geometry similarly to Virtual Texturing. Most added knowledge has been obtained by investigating a fuse of two 3D engines, aimed at acquiring each of the desired properties of Virtual Texturing. The thesis provides insights on implications and obstacles discovered during that investigation. It also suggests possible solutions and further optimizations that can be made based on these insights. Further, a survey is presented which lead to Adaptive Quad Patches being the choice of system for further the investigation. It renders the geometry as subdivided patches with vertex-displacement maps. The investigation showed that Virtual Texturing is not yet beneficial when added to this type of patch based rendering in WebGL since (1.) patch seams must be pre-computed on the CPU, and (2.) rendering of pre-tessellated geometry patches is currently not effective enough to justify the use of a page cache.

Table of Contents

1 Introduction

In this chapter an overview will be given of the thesis and its main contributions. It provides a background for this work and the rationale behind it. After that, the problem formulation is given followed by the contributions and limitations of this thesis. It ends with an outline of the rest of the chapters of this thesis.

1.1 Background

The web has been under rapid development ever since it started become widely available in the early nineties. As the hardware and software improved, the bandwidth and the complexity of the delivered content increased. The first intents of rendering 3D content in the browser were taken as early as in 1994. The newly founded Web3D consortium developed a platform independent language called VRML (Virtual Reality Modeling Language) for defining 3D content in the browser, but due to lack of browser support without plugins it was discontinued.

Its successor, X3D (Extensible 3D Graphics), an XML based file format, will not be included in HTML5 [BI10] but still strives, in competition with other formats, to become the 3D standard of the Web, circumventing the need for browser plugins through its proposed syntax model X3DOM in combination with WebGL and JavaScript [X3D11].

WebGL 1.0 was introduced in 2011 and is based on OpenGL ES 2.0 which is aimed at embedded systems. It takes a different approach than e.g. X3D and uses the HTML5 standardized canvas element to render. No plugins are needed and an API for direct access to the powerful GPU through JavaScript in the browser is provided [BI10].

WebGL is already supported by most major browsers and, similar to audio/video and photo, 3D content creation is finally becoming cheaper in terms of time, user skills and economic investment. Consequently, 3D graphics over the Internet is expected to attract a lot of additional attention in the near future [GMR+12, CLCN10].

The ability to view 3D online in a browser, referred to as Web3D, introduces many new possibilities such as online 3D museums or 3D web shops, but also creates new demands on technology. As 3D scanning tools improve and 3D model mesh quality are refined more bandwidth, memory and processing power are needed to display them [GMR+12].

There are many additional approaches to tackle the problem of limited hardware. A vast amount of research has been dedicated mesh compression. MPEG4 is a reference work in that field [JPP08]. Mesh simplification is another large field, concerning different level of detail, often involving progressive meshes [Hop96]. Furthermore, different out-of-core techniques have been developed to enable loading of only the needed parts of meshes that are too large to be fully loaded in memory at an instance of time. Several techniques are often combined to achieve even better performance and render even more complex 3D scenes.

1.1.1 Mesh representations

Although some paradigm shifting techniques for representing geometry meshes are currently being investigated, techniques such as ray traced Sparse Voxel Octrees still need more research and would most likely also need better hardware adaption to compete with conventional methods [Wil12]. Today, models are almost exclusively rendered by polygon rasterization and represented as polygon meshes that approximate the original shape of the model [May10].

Polygonal meshes

For distribution of polygonal meshes most work has been dedicated to compression. The MPEG-4 standard includes some of the state of the art techniques for that [JPP08]. Other approaches use view-dependent LOD based on techniques similar to [XV96, LE97]. These methods normally use CPU to compute the visible polygons and range from fine-grained refinement at triangle level that minimizes triangle amount, to coarser refinement at some cluster level that is less CPU intensive. [HSH10] introduced non-trivial data structures to enable computation of the visible polygons on the GPU. Though compact, [GMR+12] point out that decoding of such data structures introduces potential problems in a script-based web implementation.

Geometry Images

Geometry Images as presented by Huges Hoppe [GGH02a] consists of two-dimensional arrays of vertex position values (x, y, z) defining a mesh by the implicit regular-grid structure of the array. The geometry can be represented in the images both as regularly sampled charts [PCK04] allowing implicit parameterization, or irregular [SWG+03] resulting in less parameterization distortion but more complicated level of detail from mipmaps. Triangular-chart patches together with a spectral clustering method for feature detection can lead to a ten-fold improvement in fidelity compared to quad-chart geometry images

[FKY+10]. Depending on how the geometric structure is represented in the image, different compression methods can be borrowed from the field of image processing. [CLCN10] uses content aware image resizing saving more of detailed surfaces and less where there is less detail. [LHC07] uses JPEG2K for compression and delivery. There are also many alternative ways of storing geometry in images which will be covered in subsequent chapters of this thesis.

1.2 Rationale

As was stated in the previous section and similar to audio/video and photo, 3D content creation is finally becoming cheaper in terms of time, user skills and economic investment [GMR+12]. WebGL-enabled web browsers, leads to 3D graphics over the Internet being expected to attract a lot of additional attention as 3D scanning tools are becoming commodity components and an ever increasing complexity and file size of 3D models are seen [GMR+12, CLCN10].

Much research has been aimed towards online geometry transmission techniques, but most of them represent geometry as polygonal meshes [CLCN10]. Bridging the gap between image and geometry processing as research fields, representing a 3D model by a geometry image bring some benefits to compactness, ease to render, transmission and storage compared to many traditional mesh representation formats [CLCN10, HCW+09].

[LHC07], [GMR+12] and [HCW+09] all use geometry images aiming towards Web3D and underline the importance of having a progressive mesh streaming mechanism to decrease the waiting time. Still, progressiveness is not enough for rendering models exceeding the capacity of the GPU video memory. Several out-of-core techniques have been developed for conventional geometry but almost all of those that address geometry as images have been restricted to terrain "or at least planar scenes" [May10]. [NPC07] addressed any topology for geometry images but aimed towards offline environments. They achieved excellent results combining a number of innovations but did not describe their out-of-core paging algorithm thoroughly. Yet, a paging algorithm aimed towards WebGL would have to take other parameters and limitations into account.

Virtual Texturing elegantly enables out-of-core paging for textures and has been successfully implemented for WebGL by [AG12]. However, as geometry textures are processed differently in the GPU, investigating the advantages of a similar out-of-core paging algorithm to Virtual

Texturing for large online mesh models represented as geometry images, hereby referred to as Virtual Geometry Textures, remain an interesting, non-trivial and unexplored research subject.

The rationale for conducting this thesis can be summarized as the combination of the following three observations:

- Virtual Texturing provide an elegant out-of-core paging system for textures and has been successfully implemented for Web3D by [AG12].
- Geometry can be represented as images, which in many cases can be advantageous to conventional geometry representations [GMR+12, CLCN10, HCW+09].
- Combining Geometry Images and Virtual Texturing would be interesting since a similar system, although aimed towards an offline environment, has already been developed by [NPC07], achieving great results. If proven feasible for the web, similar web based systems might benefit from this combination.

1.2.1 Definition of "Virtual Geometry Textures" used in this thesis

Throughout this thesis the name Virtual Geometry Textures (VGT) refers to Virtual Texturing symbiotically combined with Geometry Textures, which, in its turn refers to any technique which involves streaming and rendering of geometry data stored in textures. The aim is not to limit this thesis to the use of any certain class of geometry textures, but rather to find the one that seems most suited for being used in combination with Virtual Texturing.

1.3 Problem formulation

Based on the rationale for and definition of Virtual Geometry Textures given in the previous section the main purpose of this study is to clarify the potential performance and implications of Virtual Geometry Textures for WebGL as a further development of previous similar methods to stream 3D progressively online.

In other words, this thesis sets out to answer the question: *What are the implications and potential performance of a VGT system for the web?*

To answer this question, an introduction to Virtual Texturing and techniques using geometry stored as textures is given in the Chapter 3, followed by a survey of three interesting such techniques in Chapter 4, an analysis of this problem formulation and a method for how the question should be answered in Chapter 5. The outcome is then presented in Chapter 6.

1.4 Contributions

This thesis is aimed at any software engineer interested in-, and is supposedly about to implement, a system for streaming large geometries and textures progressively. It sets out at investigating whether it is beneficial to stream geometry in textures together with Virtual Texturing in a WebGL environment. It aims to be an interesting contribution to Web3D by providing insights to an alternative out-of-core technique for loading large geometry meshes progressively. It provides a survey of three methods using textures to render geometry, and discusses their respective suitability for the proposed purpose. The third method is then evaluated deeper and investigated. An elaboration and discussion of implications, as well as pitfalls encountered during the investigation is presented.

In addition, Chapter 3 provides an introduction to Virtual Texturing and the relevant investigated geometry carrying formats. It should be of use to anyone who wants an introduction to the topic.

1.5 Limitations

The thesis investigates whether VGT is at all feasible rather than implementing a complete prototype. Several preprocessing techniques would have to be linked together to realize a test implementation from which it would be possible to measure and extract hard data. This thesis does not cover these since, depending on the choice of technique, different preprocessing steps will have to be combined to achieve suitable geometry image formats.

Two bottlenecks were found that prevents VGT from being implemented with full performance using the current state of the WebGL standard. Because of this, the VGT is system is said to be currently not feasible, but since that is based on this particular instance of the problem, that combines two particular 3D engines, it is naturally hard to leave any guaranties that a VGT system could not be effectively implemented. However, at least some of these insights should be of more general use.

Because of the bottlenecks found, any further investigation that would have to be made having alleviated these was left out.

1.6 Outline

The outline of the remainder of this thesis looks as follows:

Chapter 2 – Related Work: Briefly describes the related works of most interest.

Chapter 3 – Theory: Gives the theory of the main concepts treated in this thesis; Virtual Texturing, and per-vertex displacement mapping.

Chapter 4 – Survey: Presents a survey of three techniques for storing and rendering geometry as images. This survey serves as background and motivation for choosing Adaptive Quad Patches as the technique to be used for further investigation in Chapter 6.

Chapter 5 – Problem analysis: Analyses thesis question further and presents a method for answering it.

Chapter 6 – Virtual Geometry Textures: System Investigation – Investigates and elaborates on the implications of combining Virtual Texturing with Adaptive Quad Patches (AQP). This is done by elaborating on how suited the architectural properties of the essential features of Virtual Texturing would be to be added to AQP.

Chapter 7 – Future Work: Covers the two points considered most important to for future investigation and suggests solutions to them.

Chapter 8 – Conclusion: Summarizes the main points of the thesis and what has been achieved.

2 Related work

There are several works related to this thesis, many of them already mentioned in Chapter 1. This chapter further puts the thesis into its context. In the next chapter, a deeper description will follow for the theory of the main works for which the possibilities of a VGT system have been investigated. Other related works will be referenced throughout the paper as suited.

2.1 Online streaming of geometry through images

A number of progressive streaming techniques have been developed for the web enabling progressive geometry streaming and progressive refinement of level of detail (LOD). [SC12] implements fine-grained progressiveness at vertex level by streaming a reversed order of mesh splits achieved from the simplification of a mesh. [LCD13] achieves good results using a progressive decompression algorithm with several LODs but suffers from an intensive use of the CPU. A comprehensive work covering many aspects of general progressive streaming online is [Wei10].

Relatively few approaches so far use textures to deliver geometry online. Adaptive Quad Patches (AQP) [GMR+12] uses geometry vertex displacement textures to enable GPU rasterization and let the networking and LOD computing component rely on already existing and optimized libraries for compression and streaming of images. Their LOD is computed at a much coarser level as quadratic patches.

While [GMR+12] does have the capability to load geometry in textures progressively, it does not make use of the GPU to evaluate which geometry textures are needed. Nor does it use a cache texture or an indirection table to address to address larger textures and cache these on the GPU. These are features of Virtual Texturing, an out-of-core technique proposed originally as “MegaTexture” by John Carmack [May10]. It has been extensively evaluated and investigated by [May10] and elegantly solves out-of-core paging for textures and has been successfully implemented for Web3D by [AG12]. [NPC07] is the only work found using geometry images combined with an out-of-core algorithm. Although investigated in an offline environment [FKY+10] refer to [NPC07] stating that a similar out-of-core algorithm could be a useful extension. This thesis investigates the combination of a similar patch based system, where the geometry is delivered as textures, and the out-of-core algorithm Virtual Texturing. The next chapter will cover the theory behind Virtual Texturing as well as the other techniques needed for understanding the rest of this thesis.

3 Theory

In this chapter the theory of the main concepts treated in this thesis is presented. To begin with, Virtual Texturing is explained, followed by a recap of displacement mapping which is related to most geometry texturing techniques.

3.1 Virtual Texturing

This section contains a brief recap Virtual Texturing. While Virtual Texturing has its variations, generally the one referred to in this thesis is the implementation by [AG12].

Virtual Texturing is a rather advanced system combining several techniques to alleviate many of the difficulties commonly related to texturing. As shown by [AG12] Virtual Texturing also has many advantages in online applications. Virtual Texturing combines the advantages of the following, briefly explained, techniques:

- Mipmaping [Wil83] is a technique used for increasing efficiency in texture filtering. A mipmap is a pre-calculated "pyramid" of a texture image containing copies of itself at different sizes, one for each different level of detail. At each level the size is reduced to ¼ of the level above, resulting in ~33% total extra space needed. Using a mipmap, the image can be sampled directly at a sufficient level of detail, or at a blending distance between two levels. This reduces the number of samples needed.
- Clipmapping [TMJ98] is a technique for loading only needed parts of a mipmap. This is more memory effective and crucial for cases when the mipmap texture is too large to fit into memory – as in the case of Virtual Texturing, where the texture atlas used is commonly very large.
- Texture atlases [Wlo05] are used for performance reasons to reduce render state changes by placing all textures in one large texture. This way no extra calls have to be made to the GL for binding other textures when switching between the objects that are currently drawn.
- Texture streaming [VW06] is used to load the lowest resolution mipmap level first and later stream higher resolutions on demand. As streaming is often inefficient at higher data rates the textures are normally compressed during streaming.

[AG12] combined these techniques and achieved Virtual Texturing in WebGL basing the implementation mostly on Sparse Virtual Textures by Sean Barret [Bar08] . This led to a system having the characteristics described in the following sub-sections.

3.1.1 Pre-processing step

A pre-processing step generates the virtual texture from the textures of all objects in the scene. The result is a large clipmap comprising of thousands of equally sized page images on disk, stored in directories by mipmap level and systematically named for easy access and streaming.

3.1.2 Page determination

The page-determination step is used to evaluate which texture pages need to be fetched and loaded. This step renders a pre-frame, reads out pixel data, processes pixel data and requests needed pages. The pre-frame rendering uses a specialized fragment shader that calculates uv coordinates, mipmap level, and alpha blending information which it stores in the RGBA color channels for every pixel.

During this step the pixels are rendered to a frame buffer object instead of the screen buffer, and are then read back to memory so that they can be analyzed for a correct page fetch. The pixel read back stalls the GPU. Therefore it is important to reduce read back time as much as possible. This is done by minimizing the frame buffer. As shown by [May10], its minimal size can be to 1/8 of the view buffer while still giving a correct page determination. Also, the read back is not necessarily done for every frame. Depending on camera movement for example, it can be done more or less frequently.

3.1.3 Page streaming

Page streaming is used for downloading needed pages asynchronously through Web Workers (extensive tests was performed by [AG12] comparing page streaming performance of loading through Web Workers, Web Sockets, and simple Javascript). When using Web Workers for fetching images, only raw image data can be passed back to the main thread. Luckily, data can be loaded directly into WebGL without creating any Javascript image objects. [AG12] realized that this was important since image objects will normally be cached by the browser, causing the memory to fill up and eventually a page crash due to the thousands of page requests that are done in Virtual Texturing.

3.1.4 Page cache

The page cache is a large texture that holds the active texture pages on the GPU and is regionally updated every time a new page is loaded. It reduces the number of state switches needed by the renderer by being bound only once and being continuously updated with pages uploaded by texSubImage2D. It is mipmapped but contains pages from any of all mipmap levels. The cache has a CPU side representation: an array of JavaScript objects is kept and updated containing state information about each page currently in the cache. The state information comprises of the cell position in the cache, page index, mipmap information and a counter, indicating the last frame in which the page was referenced. When the cache is full it uses a pseudo least-recently-used (LRU) page replacement scheme, placing the new page at the position of the first found page that has the highest current frame count.

3.1.5 Indirection table

After each page replacement an indirection table is updated accordingly. It is used in the fragment shader as a translator between uv coordinates virtual textures and their locations in the page cache allowing the Virtual Texturing fragment shader to address individual texture fragments from the physical page cache as if they resided locally in the GPU memory. The indirection table is uploaded as a texture to the fragment shader where its data is used to translate the virtual texture's uv coordinates to physical cache texture uv coordinates. Since it is normally the case that the virtual texture used is larger than the cache texture (otherwise the cache texture would per definition redundant), and due to the latency in page loading, eventually some page faults will occur. When that happens, the translation gives the physical cache coordinates to a lower resolution mipmap level of the requested page. At least the lowest quality mipmap level will always be available in the cache.

3.2 Per-vertex displacement mapping

Displacement mapping [SKU08] is a collecting term for techniques mapping a texture to geometry in some way to enhance the appearance of its geometrical surface. These techniques are can be classified as *per-vertex* and *per-pixel displacement mapping*. Depending on whether they are performed in the vertex shader or fragment shader respectively.

Common per-pixel techniques are normal maps, bump-maps and parallax mapping. All these are handled in the fragment shader stage of the rendering pipeline and can only give an illusion of a displacement, but no real geometry displacement is performed. Thus, flat model silhouettes will reveal the illusion.

The other class; per-vertex displacement mapping is discussed extensively in this thesis and will hereafter simply be referred to as displacement mapping.

One such displacement mapping technique commonly used is height mapping, in which a surface of vertexes is displaced up or down along the surface normal according to scalar values sampled from a gray scale displacement map. This kind of displacement is commonly used for terrain, but can also be implemented for general objects.

Another per-vertex technique is vector displacement which instead uses a color texture as displacement map storing displacements as vectors, making it possible to displace the vertices in any direction – even overlapping each other [Hil13].

One problem with displacement maps is the intense use of the vertex shader, which must invest an equal amount of processing for every vertex, while generally suffering from far less performance than the fragment shader. Another problem is that vertex shaders have longer texture access times (Vertex Texture Fetch).

3.3 Instanced tessellation

Performing real displacements to geometry requires its surface to be sufficiently tessellated. In recent hardware this is done a fixed pipeline step called tessellator. A tessellator takes as input a geometric model at a certain polygon density, subdivides its faces and outputs it with a higher polygon density. Unlike the vertex shader, the tessellator *creates* vertices so that subdivisions can be effectively calculated in a fully dynamic manner directly on the GPU. This can save a lot of bandwidth and memory and enables view-dependent continuous level of detail (LOD) to be computed on the fly [mso]. However, this feature is not likely to be supported by WebGL in the near future since it has only started appear in advanced hardware. So to achieve real-time tessellation some other approach is needed.

One obvious way to gain more displacement points is by using subdivision surfaces which are popular for representing curved surfaces or refining geometry and are usually combined with displacement mapping to add surface details.

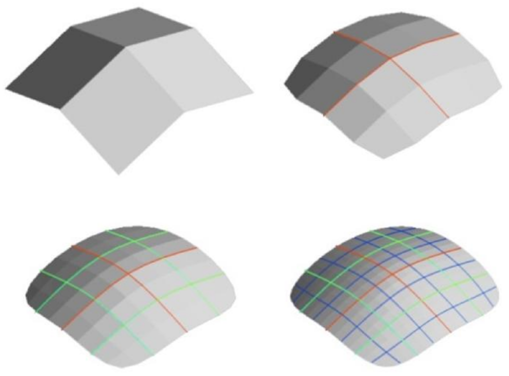

Figure 3.1: Catmull-Clark subdivided surface.

The most popular method is the Catmull-Clark subdivision scheme. It follows a set of simple rules to subdivide a mesh and can be performed adaptively or uniformly. An adaptive subdivision scheme use more processing power but places vertices smarter. A uniform subdivision scheme is faster but places vertices isometrically over the mesh. This leads to some regions being under-tessellated and others over-tessellated. While under-tessellation makes polygons visible, over-tessellation can severely affect rendering performance [Bun05].

[DRS09] presents an instanced tessellation approach for subdivision surfaces which after some modifications can be used in WebGL as it does not rely on hardware support for tessellation. This approach pre-calculates a set of tessellated triangle patches of different LOD, which are uploaded once to the GPU and rendered at multiple positions using instancing. Adaptive Quad Patches, explained in the survey in the next chapter, uses a modified version of this approach to render square patches instead of triangles. One downside is that it does not use *hardware*-instancing as this is not widely supported in WebGL yet. This method will be explained in greater detail in the next section.

4 Survey

Given an online Virtual Texturing system, this chapter presents a survey of three alternative techniques for storing its geometry in images. They are described and discussed here from the aspects of their compatibility with Virtual Texturing. Even though they were finally not selected for further investigation in this thesis they are still interesting enough to be included. For a deeper study of each of these techniques, refer to their respective sources.

4.1 Geometry Images

This section describes the "original definition" of Geometry Images introduced by [GGH02a] in 2002. Since then, many variations have been developed building on that same idea.

Rendering traditional polygonal meshes is an inherently complex task in regards to collecting and grouping together incoherent connectivity information and attribute data that 3D models consist of. Primitives' vertices must be accessed by hardware in random order, and their associated texture coordinates must be fetched. Next, the pixels of each primitive's corresponding area in the texture domain must be fetched [HCW+09]. While the costs of these steps are mitigated through vertex and page caches, it has been noted that "the fact that this pipeline has been made efficient is a remarkable engineering feat" [GGH02b].

Geometry Images is a fundamentally different shape representation that solves several of the problems inherent to polygonal meshes. They consist of two-dimensional arrays of vertex position values $[x, y, z]$ defining a mesh by the implicit regular-grid structure of the array. Neighboring pixels in the image are stored as neighboring nodes in the mesh. This makes them completely regular and with connectivity information being represented implicitly by the pixels in the image. Model textures and normal maps can be coded in the same fashion and use the same parameterization [HCW+09].

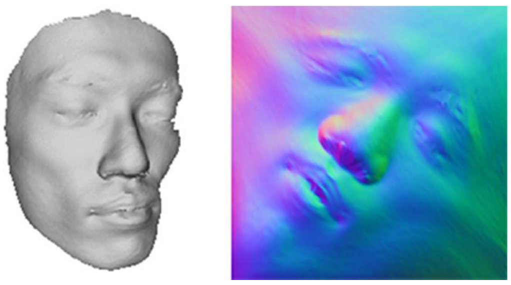
Figure 4.1: Face converted to GIM.

4.1.1 Geometry Image creation

The creation process involves many advanced algorithms but can be made fully automatic. Geometry images are created by re-sampling a mesh onto a flat square domain - an image. The goal is to re-sample the mesh as evenly as possible to avoid artifacts and performance waste due to some regions being under-, or oversampled.

The first step involved in creating a geometry image introduces a cut in the mesh, changing its topology so that it can be mapped to a flat surface [GGH02a]. Thus, if the mesh has higher genus than 0, i.e. it has more than 0 holes; the cut must be introduced in such a way that these holes are cut open before the resulting mesh can be mapped to a flat surface.

The second step involves iterating a cut-improvement algorithm that introduces additional cuts to improve the quality of the parameterization and minimize stretch. Extremities in meshes produce areas of tightly packed primitives in the flat surface domain, which leads to undersampling. By cutting through these extremes, a more isometric parameterization is achieved and undersampling reduced. Several different algorithms can be chosen for this step.

The third step maps the mesh to the square domain. The cuts are found at borders of the square, each being related to one other corresponding "cut-side" (resulting from duplication of vertices and edges in each cut) having the same length at the border to ensure a correct mapping at rendering time to avoid cracks.

Having transformed the mesh into a 2D square domain, it is overlaid by a uniform sampling grid used for mapping sampling positions in space $[x, y, z]$ of the original mesh to pixels $[r, g, b]$ in the geometry image. Vertices and attributes, such as colors and normals can then be sampled together, resulting in an implicit parameterization that eliminates the need for indexes and texture coordinates. It is also possible to sample for example the normal map to a higher resolution since it tends to contain more details. *uv* coordinates for it can then be calculated directly in the shader program. The renderer then simply traverses the geometry image in a raster-scan order by spanning each 2x2 quad of samples from the image with two triangles [GGH02a].

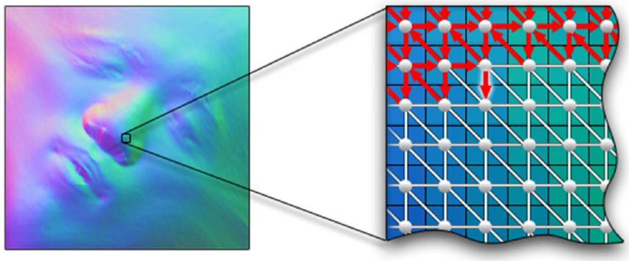
Figure 4.2: Illustration of how GIMs are processed into vertex triangles.

Nothing is stated in [GGH02a] about how *vertices* for the geometry images are sent to the graphics hardware for rendering. But the implementation of the library OpenGI (Open Geomertry Images), created by Christian Rau [Rau11], which was investigated for this thesis, creates a geometry mesh patch of vertices that matches the size of the geometry image. It is sent to the GPU and the geometry image is rendered to its vertices using vertex texture fetch in a simple vertex shader [GGH02a].

4.1.2 Evaluation

Thanks to the regularity of how the geometric structure is represented in the images, different processing and compression methods can be borrowed from the field of image processing [GGH02a]. [CLCN10] uses content aware image resizing saving more of detailed surfaces and less where there is less detail. [LHC07] compresses geometry images using JPEG2K. [HCW+09] combines spectral analysis with geometry images to achieve more visually pleasing shapes at high compression rates.

Geometry Images can be trivially up- and downsampled to achieve different level of detail representations in mipmapping, although straightforward lossy compression may introduce visible seams. In [GGH02a], this problem is eliminated by encoding a cut fusing into a small data sideband. The geometry can be represented in the images both as regularly sampled charts [PCK04] allowing implicit parameterization, or irregular [SWG+03] resulting in less parameterization distortion but more complicated level of detail from mipmaps. Triangular-chart patches together with a spectral clustering method for feature detection can lead to a ten-fold improvement in fidelity compared to quad-chart geometry images [FKY+10].

Geometry Images that encodes a full model per image may be less suited for Virtual Texturing. This is because Virtual Texturing uses a page cache to load desired pages from a virtual texture progressively, demanding more flexibility than the original Geometry Image method. Currently, pages used in the virtual texture clipmap must be of the same size. Thus, geometry models of different sizes must be up or downsampled accordingly, inevitably leading to some models being coarser than others.

Splitting Geometry Images into patches could be an interesting solution this, and could also save memory by providing a cage for each patch. This would reduce the need for precision as vertices store positions relative to the cage. Otherwise, Geometry Images generally need at least 12 bits per channel [BJFS12] which is more than the 8 bits being the current limitation in all common web browsers.

4.2 Sequential Image geometry

In addition to the pre-processing steps for Geometry Images being extremely complicated, also, not all types of meshes can be handled without modifications. To avoid these problems, X3DOM utilizes what they call Sequential Image Geometry (SIG) [BJFS12] which simply codes vertex data, *as-is*, in a fixed sequential order to texture [r, g, b]-channels.

Image files are used to carry the data but merely as containers storing unlinked vertex data. Vertex- and normal coordinates are simply coded into texture channels. Mesh coordinates are linearly normalized in accordance with their respective bounding boxes, converting all values to the interval [0:1]. Bounding boxes are kept directly in the HTML / X3D document and can be used for culling calculations to avoid loading of unnecessary data.

Noting that all common image formats supported by web browsers today are limited to 8 bits per color channel and that this gives 32 bits per texel in RGBA textures, SIG takes the approach of splitting the encoding of normals, texture coordinates, and positions into separate images.

While 2D texture coordinates and normals can fit in one 32 bit RGBA texture each, vertex *xyz*-coordinates need more space and are instead coded in several RGB textures in accordance with a coarse-to-finer precision strategy: The first texture contains the first 8 higher significance bits of each respective RGB channel. The next texture contains the next 8 respective significant bits and so on.

A valid model can be rendered already from the first image, and later be refined to any level of bit depth by streaming more geometry precision through additional textures. This can be used as a kind of level of detail (LOD) technique. By using the bounding boxes which are stored in HTML, the LOD can be calculated to prioritize geometry precision for meshes that, for example, are closer to the camera. Note that this LOD technique does not involve any change in the number of vertices that has to be drawn but solely concerns the precision at which they are drawn.

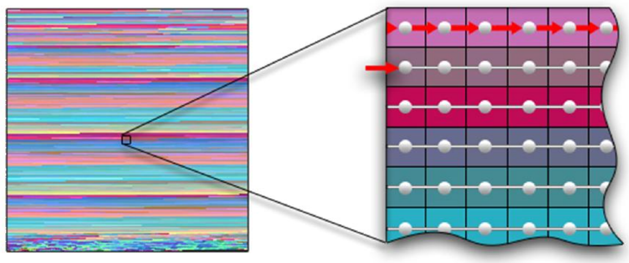
Figure 4.3: Illustration of how SIG displacement textures are processed into vertex strips.

Geometry patches are created on the CPU using a simple triangle striping algorithm. Vertices are displaced in a simple vertex shader according to positions sampled from the Image Geometry textures.

4.2.1 Evaluation

SIGs are advantageous in that complicated pre-processing steps are avoided and rendering made simple and effective. As seen in Listing 1 below, the final position is simply sampled from the texture and adjusted to the bounding box.

```
uniform sampler2D IG_coordinateTexture;
…
vec3 pos = texture2D( IG_coordinateTexture, IG_texCoord ).rgb;
pos = pos * (IG_bboxMax - IG_bboxMin) + IG_bboxMin;
…
gl_Position = modelViewProjectionMatrix * vec4(pos, 1.0);
```

Listing 1: SIG's vertex shader samples the vertex positions directly from the texture.

SIGs main disadvantage, however, is that simply packing vertex data in images results in pixel values with almost no local coherence.

Therefore SIGs loses many interesting 2D image compression possibilities, such as the ability for up and down sampling of images to higher or lower resolutions such as used in mipmapping. Although the ability of having different LODs is possible through the way vertex coordinates are handled in the images as explained above, one misses the desirable feature of using other than lossless image compression.

Another disadvantage with this method is that it does not include any handling of seams. Cracks between patches easily appear if only one of the 8 bit precision images is streamed. At least two images for each patch, or a vertex precision of 16 bits per channel, are generally needed to avoid these cracks showing up. Still, in the demo of Happy Buddha, it appears that this is not enough.

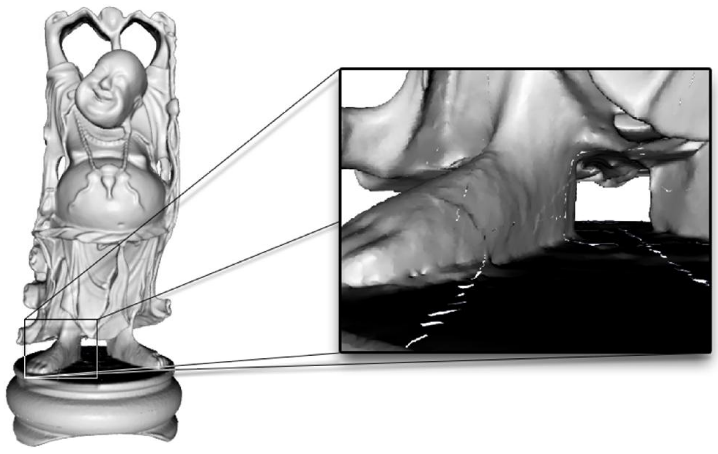

Figure 4.4: No handling of seams may result in visible seams. Here using in 8 bits per channel.

4.3 Adaptive Quad Patches

Adaptive Quad Patches (AQP) has been chosen for this thesis as the system to combine with Virtual Texturing. It uses a complete automatic pipeline for converting, compactly store, effectively stream, and render geometry models. It takes the approach of splitting the geometry into quadratic patches which can then be streamed and rendered at different fidelity depending on the view and demand of detail.

AQP has been developed for both OpenGL and WebGL so it is worth noting that this thesis will only cover the WebGL version, which has some disadvantages arising from the gap of features between these standards. Although the WebGL version performance is lower, it makes better relative use of for example the streaming facilities.

Motivation for use of AQP in this thesis

AQP builds on many of the techniques similar to those considered in the survey and seems to find a good balance between these. Although AQP is a unique system, the fact that it makes use of many existing techniques increases the generality of this thesis. The main reasons for using it in this thesis are as follows below.

- Apart from patching, its approach to store geometry is similar to vector displacement maps which are increasingly supported in 3D modeling software such as ZBrush.
- It implements instanced tessellation using subdivided meshes which has been proposed by for instance in [Tat08] as a substitute for hardware tessellation when such are not available.
- It is a fully functioning 3D engine allowing for side-by-side comparison with a Virtual Texturing engine.
- It is well suited for meshes that define closed objects with large components (without many finer topological details). These are normally features of scanned 3D objects, which is likely one of the more interesting targets for web 3D in the near future.

Patch based meshes

As mentioned in the AQP paper, different approaches to using patch based rendering such a [HSH10] have proven very effective in terms of rendering speed. But since these approaches often require coding of non-trivial data structures and techniques for decompression, they could be problematic to implement in a script-based web environment. Instead, AQP adapts and makes use of Semi-Uniform Adaptive Tessellation [DRS09] pre-calculating mesh grids at different level of detail on the CPU side. These patches of vertices are then uploaded to the GPU before being instanced and displaced by a displacement texture.

An interesting property of AQP is that *all* geometry is stored in textures. Images are generally easy to handle in a web environment, and textures can, unlike geometry buffers, be partially updated without having to re-bind them in the GL-context using "texSubImage2D". Compared to geometry images which are normally concerned with re-parameterizing the whole mesh into one image, AQP achieves a tighter texture packing by re-parameterizing the mesh into multiple smaller squares based on the mesh topology. The splitting of the mesh into smaller squares instead of larger irregular charts such as in [KLS03] or [PH03] leads to simpler handling of chart boundaries without need for specialized transition functions between patches.

4.3.1 Adaptive quad patch creation

The square patches are achieved by first minifying the mesh so that it becomes a coarse grained root mesh. This base mesh is then then re-parameterized. The re-parameterization takes the approach of [PTC10] replacing each triangle in the mesh by three edges originating from the triangle center point where a new vertex is created (see figure).

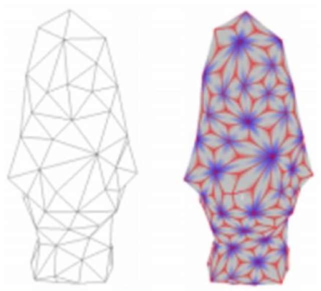

Figure 4.6: Simplified mesh and quadratic patches re-parameterization pattern.

Geometry displacement images are then created by sampling values from an overlay of the original mesh with respect to each patch in the root mesh. A tightly packed geometry texture atlas is then created by storing these images side by side. Also, color textures and normal maps atlases are created similarly. The amount of patches in the texture atlas can be set to a maximum value to bring down the download sizes. 512 patches is the maximum value used in the AQP implementation. If more patches are created, these will be stored in another texture atlas.

The texture atlas is mipmapped to create different levels of detail (LOD). When downsampling is made for the texture atlas, for the coarser mipmap layers, inner samples and border samples are handled differently. While inner samples are simply the average of 4 samples from the finer mipmap level, border samples are averaged only on the 2 samples which are part of the boundary, creating continuity across patch borders. For the same reason, corner samples use pure sub-sampling and stay unchanged through all mip layers.

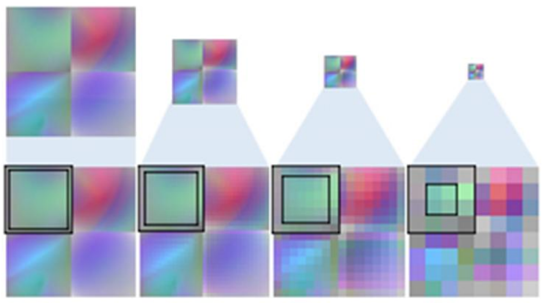

Figure 4.7: Patch border pixels are kept down the mipmap chain to ensure continuity across borders.

The corners of the root mesh are stored as raw vertex positions. Similar to the rest of the mesh these are stored in image format, but as absolute positions using 16 bits, which is double to the bit precision normally supported by images in web browsers. Therefore two images are used, one containing the high- and the other containing the low precision bits of the coordinates. Also, two more images are used storing individual displacement *quantization ranges* for each vertex in the mesh. These ranges are used to avoid discretization artifacts by adapting the quantization level of each vertex's displacement.

For a more detailed description of the pre-processing steps refer to the original paper *Adaptive Quad Patches: an Adaptive Regular Structure for Web Distribution and Adaptive Rendering of 3D Models*.

4.3.2 Rendering adaptive quad patches

Pre-tessellated geometry grids (patches) are created at the initialization of the engine, one per each mipmap level of the texture tiles. Texture mipmap levels are downloaded and streamed asynchronously on demand using Web Workers.

As illustrated in Figure 4.8, the following steps are taken when drawing a model using the AQP engine:

1. *Commit new tiles* - new tiles are uploaded to the textures already bound to the GPU using WebGL call texSubImage2D.
2. *Edges LOD Evaluation (ELE)* - the screen projected length of the edges of each patch is evaluated and a LOD is set for every edge. The patch LOD is then set as the maximum of these LODs. This step affects the performance and will be elaborated on more deeply in Chapter 6.
3. *Update edges LOD texture* - simply updates the edge LODs texture using texSubImage2D.

4. *Draw patches* - draws every patch after having set the correct sampler textures and uniforms. Each patch is drawn in a separated draw call with uniforms and samplers updated in between.

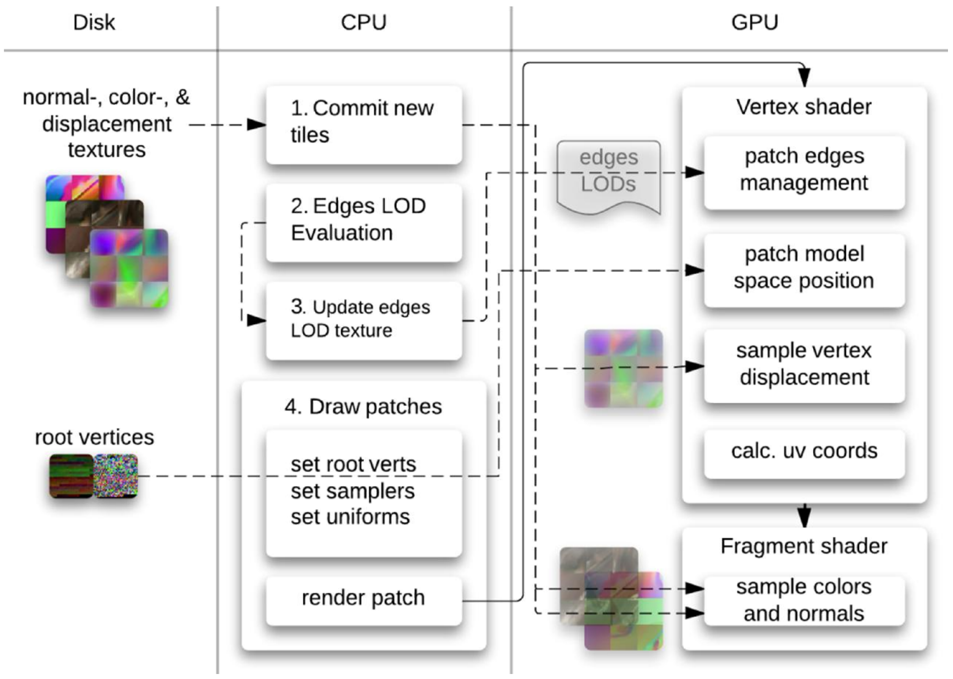

Figure 4.8 Control flow of the rendering steps in the AQP engine. Dotted lines showing data flow.

The most relevant parts of this system will be covered more deeply in the elaborations.

5 Problem analysis

The problem description stated a question that is the main purpose of this thesis to answer:

"What are the implications and potential performance of a VGT system for the web? "

The definition used for Virtual Geometry Textures in this thesis states that the aim is *not* to use any certain class of geometry images but rather to use the one most sufficient to combine with Virtual Texturing. Having presented the theory and survey, one should now have enough background to imagine how the VGT system can be realized. The question can then be reformulated into a less general form that can constitute an instance of this problem. If VGT in this formulation is replaced by a system where the essential features of Virtual Texturing come into use also for its geometry rendering, then that would be a valid instance of this problem. The survey pointed at Adaptive Quad Patches (AQP) as a system that seemed most suited for this purpose, and was selected for further investigation.

5.1 Method

A demo of the AQP system has been achieved for this thesis from CRS4, as well as the Virtual Texturing engine Porcupine Engine© developed at Mindary. Thus a deeper investigation of the systems at hand is made in an attempt to fuse them and realize a VGT system. Each essential part of the Virtual Texturing technique is studied to gain understanding of how it could be incorporated with AQP to stream and render large meshes through textures.

As stated in the limitations section, this thesis set off at investigating implications and potential performance rather than leaving any guarantees that it would result in a complete prototype. If it would have, that implementation would still be very basic, considering the time span of the thesis, and the performance measure would not be very useful since it would be lacking many possible optimizations for its rendering pipe. The aim of this thesis is rather to discuss the architectural choices, bottlenecks and possible solutions and optimizations for it. Further research is then made depending on what is found, to be able to discuss these findings.

6 Virtual Geometry Textures - System investigation

This chapter describes the investigation on the implications of combining Virtual Texturing with Adaptive Quad Patches (AQP). Since there are many potentially useful choices of technique for rendering geometry through images, some of those that are most interesting were discussed in the survey of Chapter 4. Then, following the general solution in the problem in Chapter 5 this chapter will investigate each part of the Virtual Texturing technique alongside with AQP and elaborate whether each feature of Virtual Texturing could be implemented for AQP. This is done by elaborating on architectural properties of the essential features that must be included in such an implementation.

The essential features of Virtual Texturing are discussed in Chapter 3. Of these, the two main features that had to be added to AQP is Page Determination for geometry patches and a Page Cache for a virtual texture of geometry patches.

The approach taken at the beginning of this investigation was to extend Virtual Texturing with patch based rendering such as the one used in AQP. It was soon realized that AQP engine would be better off left mostly unchanged; thus, the approach was reversed to adding Virtual Texturing to AQP. The rest of this chapter describes the two main findings of this investigation.

6.1 Page Determination pass

This section elaborates on whether the page determination pass used in Virtual Texturing by [AG12] should be used to determine what geometry displacement pages should be streamed. It explains why the use of geometry patches currently implies CPU side pre-calculations of LODs, and why that would make the page determination pass used by this Virtual Texturing system redundant. It continues by explaining why VT page determination can be completely replaced by the Edge LOD Evaluation (ELE) step when using a system like AQP, and ends with an investigation of how large meshes can be used without needing an ELE step.

6.1.1 Virtual Texturing page determination

[AG12] uses a page determination pass to determine what pages to fetch and from which level of detail. A “pre-frame” is rendered at low resolution using a special fragment shader. It is rendered to an off-screen Frame Buffer Object and then read-back from the GPU to be analyzed on the CPU.

The pixels rendered by this page determination fragment shader contain coordinate-, LOD-, and blending information coded into the RGBA color channels. OpenGL implementations normally approximate the LOD of each fragment based on the partial derivatives of the primitive's mapping of texture coordinates to window coordinates.

[AG12] used the extension GL_OES_standard_derivatives to access derivative functions `dFdx()/dFdy()` for LOD calculation. However, some browsers did not support this, so for these browsers a fallback was necessary. The workaround for having to calculate the mipmap-level analytically was suggested by [Pha04] and uses a mipmap look-up texture where the pixels in each mipmap-level store the mipmap-level number. Sampling from this texture in a fragment shader gives an interpolated value between the different mipmap layers which can be rounded down (floored) to represent the desired mipmap-level.

6.1.2 Elaboration

Provided is the goal to use geometry stored in textures together with Virtual Texturing. The survey concluded that displacement maps with pre-tessellated surfaces seem most interesting for this. As explained, streaming the geometry through textures allows for progressive updates of its precision, similar to what is being done for textures in Virtual Texturing. However, as will be derived below, using the same page determination method as in [AG12] turns out unnecessary for systems patch based WebGL currently lacks some important features that would make this set-up really effective.

Pre-tessellated patch systems like AQP must adapt the tessellation per patch to avoid over and under tessellation. Normally patches are rendered by separate draw calls and will make up separated patches of primitives. If the vertices of these patches' edges are not aligned perfectly, visible seams or cracks may appear between them. Therefore, especially edge vertices of patches of different LOD need to be handled properly.

Edge Level of detail Evaluation

Vertices are snapped to edges according to a *snap-function* presented in the paper Semi-Uniform Adaptive Patch Tessellation [DRS09]. The information needed for doing this is sampled from the edge LOD texture calculated in the ELE step on the CPU-side. Patch position in space is calculated from merging the higher and lower significant bit values from the root textures. These values are passed as uniforms with each patch.

ELE cannot be done in a vertex shader
AQP uses a vertex-snap function that snaps together the edge vertices between patches of different LODs. A vertex is snapped depending on the different tessellation factors between two adjacent patches. Due to the vertex shader only being able to process one vertex at a time [KBR14], it normally has no information about the neighboring vertices. Thus, to snap an edge vertex correctly, the vertex shader must be provided with this extra information.

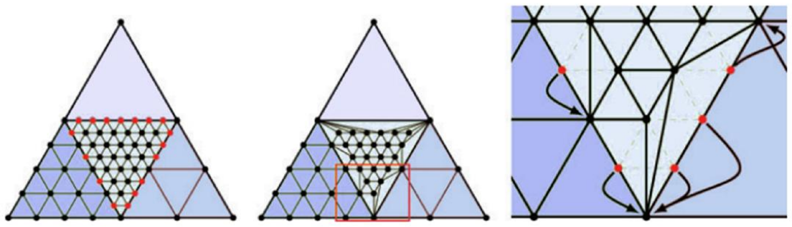
Figure 6.1: The vertex-snap function.

This information is calculated per tile by examining the length of a patch's four edges projected to the screen and selecting the LOD to be as coarse as the coarsest one of these edges. AQP then store the each respective edge's LOD in a texture. This texture is then bound to the GPU so that the vertex shader can sample the correct LOD information depending on which patch it renders (which it will know from a shader variable that is set before each draw call).

ELE replaces VT page determination
Since calculating the patches LODs is a sub step of calculating the edges LODs; for all patches, their needed LODs will be known and easily requested after this step. In AQP there is a 1-to-1 mapping between geometry-, and color pages. Thus, having calculated which geometry pages are needed means that the corresponding color texture pages are already known. This makes the page determination pass superfluous, which means the AQP edge LODs determination step fully replaces the Virtual Texturing page determination pass.

No GPU based ELE alternatives for WebGL
Since the ELE step is based on a calculation using multiple vertices, it cannot be done using a vertex shader. Either it has to be done completely on the CPU side, or ideally, using GPGPU computing such as OpenCL or CUDA. Unfortunately only experimental implementations of GPGPU libraries are available for a general online

setting. While there are more alternatives to how this can be done, none that does not use the CPU or some GPGPU computing was found during the research for this thesis. Seemingly, there is no such solution known to be suited for WebGL.

6.2 Page cache

The cache allows the paging system to store exactly the pages that are needed on the GPU at every instance of time. This enables the use of virtual textures which are larger than would be possible to load into the physical memory. A second advantage of using a page cache is that it can reduce the number of state switches which occur when using multiple textures. The page cache is an essential part of Virtual Texturing and this section elaborates on the implications of adding it as a feature to AQP or similar patch based systems.

6.2.1 Page cache's raison d'etre in general

Here two statements are made, upon which the reasoning in this section is relies. First, it is important to point out that it is only useful to implement a page cache if (1.) the virtual texture representing the scene is larger than the maximum supported texture size, or if (2.) texture switches use a considerable amount of the total rendering time. If (1.) is not true i.e. if the amount of pages *does not* exceed the texture limit, then each streamed pages could simply be stored and addressed directly to a GPU representation of the virtual texture. If (2.) is not true, it is not the bottleneck and very likely not enough reason for implementing a page cache.

6.2.2 Page cache's raison d'etre in AQP

The current AQP implementation does not suffer from (2.) so to know if a page cache would make sense to be implemented for AQP this section investigates if (1.) is true. This is done by calculating the amount of possible patches stored in the page cache implemented by [AG12] and estimating whether this amount of patches is feasible to be rendered at a reasonable frame rate.

The size of the cache texture is only limited by how many pages the indirect table can address and the largest texture size allowed being stored on the GPU video memory by the specific WebGL implementation. The texture size limits are also bound by the WebGL standard which is set with cross-platform compatibility in mind, often to comply with devices having the lowest common denominator GPU processing power, such as mobile phones or tablets. To further reduce the amount of state switches between different textures, preferably a cubemap texture is used as the cache since it is allowed to hold 6 times more texture data (one texture per side of a cube). According to webglstats.com [Boe14] which collects visitor browser statistics from a range of different websites, almost 90 % of the browsers of February 2014 could handle 4096^2 pixel cubemaps and almost 70 % could handle cubemap sizes of up to 8192^2 pixels. About 20 % could handle cubemaps of double that size.

[AG12] utilizes a cubemap of size 4096^2 pixels, which means it holds $4096^2 * 6$ pixels in total. The amount of pages that the cache can store is directly related to the page size [AG12]. For the AQP demo used in this thesis the largest patches are relatively small; only 32^2 pixels. If patch sizes are increased in relation to the model, the granularity of the mesh gets coarser, making it more difficult to represent topological details. Larger patches can also introduce stretch artifacts in some regions. The sensitivity heavily depends on the mesh topology, but based on the demo model used in this thesis, it can be assumed that patch sizes in relation to model size should not be increased much more since some regions are already showing stretch artifacts. Thus, to calculate the amount of patches the current patch size settings will be used. This amounts to $\frac{4096^2 * 6}{32^2} = 98304 = 2^{14} * 6$ patches that can be stored locally on the GPU and could be addressed directly without the need for a page cache indirection table. If we consider that each patch will also store its mipmap representations this results in $\frac{3}{4} * 2^{14} * 6 = 73728 = 2^{13} * 9$ unique patches. Also, assuming that only 1 texture cube cache is used to store each attribute texture; geometry, normal, and color, reduces the amount by $1/3$ to $24576 = 2^{13} * 3$ being total amount of patches that a model can contain before a page cache is needed.

Finding the bottleneck

The larger the mesh, the more patches it will contain and each will have to be evaluated to determine its correct edge LOD. Likewise, the patch rendering time will increase linearly by the amount of patches rendered. Thus, the overall performance will be increasingly dependent on the speed of these two steps.

As stated in (1.) earlier, a page cache with indirection is only necessary for textures larger than would fit in the GPU memory. As calculated above, in this setting models can contain up to $3 * 2^{13}$ patches before a page cache is needed since models less than that size would fit fully in GPU memory. In the horse model, the amount of patches is 300. This means that a scene containing $\left\lfloor \frac{3*2^{13}}{300} \right\rfloor = 81$ such models could be rendered without the need for a page cache.

However, as concluded earlier, the total time for the ELE step grows linearly with the amount of patches it must evaluate. Using the profiler tool in Chrome DevTools, the ELE currently takes never less than 2 ms, evaluated on an Intel Core i7 Q 720. Thus, only the ELE step would take at least $81 * 2 = 162$ ms at the point at which the number of patches are so many that a cache is needed. With patch renderings included the frame rate would drop to less than 3 fps, which is far less than acceptable.

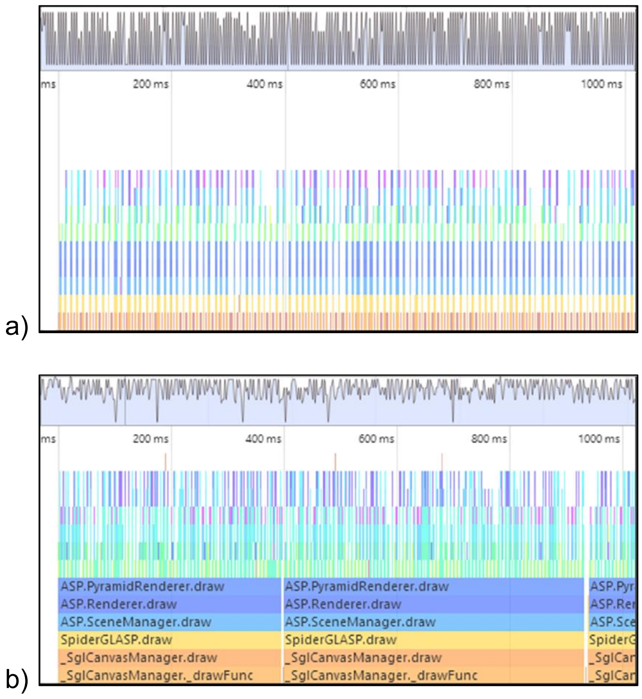

Figure 6.2 showing profiling in Chrome Development Tools.
a) Showing 1000 ms of Javascript functions when rendering the original horse model, and b) showing the same period of time when rendering 81 times more patches (only slightly more than 2 frames are rendered here during the same time period).

It should be noted that culling algorithms could be included, alleviating some processing, but that being very dependent on the scene to be rendered.

The ELE algorithm was originally presented in [DRS09] which did some extensive testing, finding that "*for meshes with more than about 1000 patches, render queue generation appears to be faster on the GPU* *than on the CPU*". However, as noted before, until GPGPU libs such as WebCL become widely supported this will have to be calculated in Javascript on the CPU side which will be considerably slower.

If some lag in the LOD state switches could be acceptable, then a compromise could be to only do ELE at some pre-set interval of frames, but this also stalls the rendering. A better solution would be to move the ELE execution into an asynchronous thread using the Javascript web worker technology. This is should probably be easy and would save much rendering time in terms of ELE.

However, the next possible bottleneck should be investigated before. It concerns the rendering of the large amount of patches. The mesh is rendered using one draw call per patch. Between each patch draw call the 4 corner coordinates of the patch are extracted from the root texture. Also, information about patch x and y coordinates, color-, normal-, and geometry LOD is updated. All this information is passed to the shader program as uniforms which are updated before the draw call is made. This result in a considerable amount of expensive native Javascript calls: uniform updates as well as draw calls.

The renderer's patch drawing performance was profiled using the same test setup; rendering the horse model which consist of 300 patches using a laptop with an Intel Core i7 Q 720 CPU and an AMD Radeon HD 6500M GPU.

It was found that a considerable amount of rendering time was spent on updating uniforms between drawing each patch and that this information most of the times remained unchanged since the last patch was drawn. After inserting a simple test for whether uniforms had been changed, most of the redundant uniform updates could be removed, reducing the complete patch rendering time by almost 20 % when rendering $81 * 300$ patches.

However, the frame rate is still below 10 FPS which is far from acceptable. Therefore, rendering that amount of patches with AQP can be considered unnecessary, which implies that according to (1.); implementing a page cache for AQP is currently not feasible. This temporarily overthrows the idea of implementing Virtual Geometry Textures in WebGL. But having found the bottlenecks, the next chapter discusses how they can be overcome in the future reviving the idea of VGT.

7 Future work

This section covers the two points considered most important for future investigation. Solutions to the two major bottlenecks that were found and discussed in Chapter 6 are presented.

7.1 Hardware instancing of patches

As made clear in the last section, the main bottleneck is the efficiency at which WebGL renders the patches. Thus the most interesting area for future investigation when it comes to AQP with Virtual Texturing is the rendering of patches. Currently, most rendering time is spent on setting uniforms and making draw calls. It would therefore be interesting to investigate whether hardware instancing would make AQP patch rendering fast enough to make use of a Virtual Texturing style texture cache.

While there was no support for hardware instancing in any WebGL implementation at the commencement of this thesis, according to webglstats.com, its support has since raised by over 50 percentages.

By moving the uniforms used in the current implementation into attribute buffers, these can be instanced using a drawElementsInstanced call, eliminating *both* the need to set uniforms between every draw call, *and* the need to make separate draw calls per patch. Most probably this would have high impact on the rendering and hopefully it would make it worthwhile to implement a Virtual Texturing cache as well.

7.2 GPGPU support - WebCL for ELE

As elaborated in Chapter 6 the Edge LOD Evaluation (ELE) step as a bottleneck could probably be much alleviated by moving its execution to a separate thread using WebWorkers. This would result in a lag in the LOD switches. [DRS09] used GPGPU techniques to make this step faster. A Khronos WebCL working group was formed in 2011 to defined Javscript bidnings to the Khronos OpenCL and some test implementations have already been made [Khr14]. Hopefully this will soon become supported in major browsers, as it would likely allow a great speed up of the ELE step and remove the last bottleneck that currently prevents VGT from being appropriate for a web implementation.

8 Conclusion

With the aim to be an interesting contribution to Web3D by providing insights in an alternative way of loading large geometry meshes progressively, this thesis set out to answer the question of what the implications and potential performance of a Virtual Geometry Textures (VGT) system for the web would have. VGT was defined here as Virtual Texturing combined with Geometry Textures, and Geometry Textures was in its turn defined as any technique that involves streaming and rendering of geometry data stored in textures.

Section 1.2 in Chapter 1 summarized the rationale for this investigation and the survey in Chapter 4 showed why this is a non-trivial problem to solve. To answer the thesis question, first, the survey presented and evaluated three potential techniques. Then, an analysis of the problem formulation was made in Chapter 5, and a method for answering the question was presented. The method used was to elaborate on architectural properties of two 3D engines; one implementing Virtual Texturing, and the other Adaptive Quad Patches (AQP) implementing Geometry Textures (as vertex displacement maps). The essential features needed from each of the two techniques were investigated in an attempt to extend AQP with Virtual Texturing to accomplish VGT. However, the attempt showed that two of the essential features of Virtual Texturing were incompatible with AQP.

Firstly, the page determination method used in Virtual Texturing by [AG12] was found to be completely replaceable (but not the other way around) with the corresponding method used in AQP. The AQP engine must do its edge LOD evaluations (ELE) to avoid cracks between the patches in the mesh, but for this calculation each required page must be calculated as well as the desired LOD for it. No uv coordinates are needed in AQP since geometry-, and texture pages are directly linked to each other. This means that knowing which geometry pages to use in a frame, implies knowing what texture pages should be loaded as well.

Secondly, considering that a page cache, which is one of Virtual Texturing's essential features, would be of little use if textures never exceeded the GPU texture size limit, the patch rendering efficiency was examined. After this elaboration it could be concluded that the patch rendering method, using separate draw calls per patch, was too ineffective at rendering the amount of patches that would exceed this limit. Using a patch based system like AQP; the bottle neck is currently not the texture limit but the efficiency at which these patches can be rendered.

The main purpose was not to implement VGT but rather to investigate whether it was at all feasible. Since two major bottlenecks were found owing to the current state of the WebGL standard, instead solutions to these were presented in Chapter 7 and suggested as future work that can potentially solve these problems.

Bibliography

[AG12] Sven Andersson and JHONNY Göransson. Virtual texturing with webgl. 2012.

[Bar08] Sean Barret. Sparse virtual textures. 2008. [Online; accessed 1-February-2014].

[BI10] Daniel Büchele and Simon Ismair. 3d graphics in the browser using webgl. 2010.

[BJFS12] Johannes Behr, Yvonne Jung, Tobias Franke, and Timo Sturm. Using images and explicit binary container for efficient and incremental delivery of declarative 3d scenes on the web. In *Proceedings of the 17th International Conference on 3D Web Technology*, pages 17–25. ACM, 2012.

[Boe14] Florian Boesch. Webgl stats. 2014. [Online; accessed 1-February-2014].

[Bun05] Michael Bunnell. Adaptive tessellation of subdivision surfaces with displacement mapping. *GPU Gems*, 2:109–122, 2005.

[CLCN10] Shu-Fan Wang Yi-Ling Chen, Chen-Kuo Chiang Shang-Hong Lai, Bing-Yu Chen, and Tomoyuki Nishita. Content-aware geometry image resizing. *Proceedings of Computer Graphics International 2010 (CGI'10)*, 2010.

[DRS09] Christopher Dyken, Martin Reimers, and Johan Seland. Semi-uniform adaptive patch tessellation. In *Computer graphics forum*, volume 28, pages 2255–2263. Wiley Online Library, 2009.

[FKY+10] Wei-Wen Feng, Byung-Uck Kim, Yizhou Yu, Liang Peng, and John Hart. Feature-preserving triangular geometry images for level-of-detail representation of static and skinned meshes. *ACM Transactions on Graphics (TOG)*, 29(2):11, 2010.

[GGH02a] Xianfeng Gu, Steven J Gortler, and Hugues Hoppe. Geometry images. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 355–361. ACM, 2002.

[GGH02b] Xianfeng Gu, Steven J Gortler, and Hugues Hoppe.

Geometry images. 2002. [Online, Microsoft Powerpoint; accessed 1-February-2014].

[GMR+12] Enrico Gobbetti, Fabio Marton, Marcos Balsa Rodriguez, Fabio Ganovelli, and Marco Di Benedetto. Adaptive quad patches: an adaptive regular structure for web distribution and adaptive rendering of 3d models. In *Proceedings of the 17th International Conference on 3D Web Technology*, pages 9–16. ACM, 2012.

[HCW+09] Ying He, Boon-Seng Chew, Dayong Wang, Chu-Hong Hoi, and Lap-Pui Chau. Streaming 3d meshes using spectral geometry images. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 431–440. ACM, 2009.

[Hil13] Karl Hillesland. Vector displacement. *GPU Pro 4: Advanced Rendering Techniques*, 4:69, 2013.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108. ACM, 1996.

[HSH10] Liang Hu, Pedro V Sander, and Hugues Hoppe. Parallel view-dependent level-of-detail control. *Visualization and Computer Graphics, IEEE Transactions on*, 16(5):718–728, 2010.

[JPP08] Blagica Jovanova, Marius Preda, and Françoise Preteux. Mpeg-4 part 25: A generic model for 3d graphics compression. In *3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video, 2008*, pages 101–104. IEEE, 2008.

[KBR14] John Kessenich, Dave Baldwin, and Rost Randi. Opengl shading language. 2014. [Online; https://www.opengl.org /documentation/glsl/ accessed 1-February-2014].

[Khr14] Khronos Group. Webcl. 2014. [Online; accessed 1-February-2014].

[KLS03] Andrei Khodakovsky, Nathan Litke, and Peter Schröder. Globally smooth parameterizations with low distortion. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 350–357. ACM, 2003.

[LCD13] Guillaume Lavoue, Laurent Chevalier, and Florent Dupont. Streaming compressed 3d data on the web using javascript and webgl. In *ACM International Conference on 3D Web Technology (Web3D), San Sebastian, Spain*, 2013.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 199–208. ACM Press/Addison-Wesley Publishing Co., 1997.

[LHC07] Nein-Hsien Lin, Ting-Hao Huang, and Bing-Yu Chen. 3d model streaming based on jpeg 2000. *Consumer Electronics, IEEE Transactions on*, 53(1):182–190, 2007.

[May10] Albert Julian Mayer. Virtual texturing. *Institute of Computer Graphics and Algorithms, Vienna University of Technology*, 14, 2010.

[mso] Tessellation overview. [Online; http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx accessed 1-February-2014].

[NPC07] Krzysztof Niski, Budirijanto Purnomo, and Jonathan Cohen. Multi-grained level of detail using a hierarchical seamless texture atlas. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 153–160. ACM, 2007.

[PCK04] Budirijanto Purnomo, Jonathan D Cohen, and Subodh Kumar. Seamless texture atlases. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 65–74. ACM, 2004.

[PH03] Emil Praun and Hugues Hoppe. Spherical parametrization and remeshing. *ACM Transactions on Graphics (TOG)*, 22(3):340–349, 2003.

[Pha04] Matt Pharr. Fast filter width estimates with texture maps. *GPU Gems, Randima Fernando*, pages 357–364, 2004.

[PTC10] Nico Pietroni, Marco Tarini, and Paolo Cignoni. Almost isometric mesh parameterization through abstract domains. *Visualization and Computer Graphics, IEEE*

Transactions on, 16(4):621–635, 2010.

[Rau11] Christian Rau. Opengi. 2011. [Online; accessed 1-February-2014].

[SC12] Bartosz Sawicki and Bartosz Chaber. 3d mesh viewer using html5 technology. *Przeglad Elektrotechniczny (Electrical Review), ISSN*, pages 0033–2097, 2012.

[SKU08] Laszlo Szirmay-Kalos and Tamas Umenhoffer. Displacement mapping on the gpu - state of the art. In *Computer Graphics Forum*, volume 27, pages 1567–1592. Wiley Online Library, 2008.

[SWG$^+$03] Pedro V Sander, Zoe J Wood, Steven J Gortler, John Snyder, and Hugues Hoppe. Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155. Eurographics Association, 2003.

[Tat08] Andrei Tatarinov. Instanced tessellation in directx10. In *Game Developers Conference*, volume 8. sn, 2008.

[TMJ98] Christopher C Tanner, Christopher J Migdal, and Michael T Jones. The clipmap: a virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158. ACM, 1998.

[VW06] JMP Van Waveren. Real-time texture streaming & decompression. *Id Software,(November 2006)*, 2006.

[Wei10] Cheng Wei. Streaming of high-resolution progressive meshes over the internet. *ACM SIGMultimedia Records*, 2(2):4–5, 2010.

[Wil83] Lance Williams. Pyramidal parametrics. In *ACM Siggraph Computer Graphics*, volume 17, pages 1–11. ACM, 1983.

[Wil12] Audun Wilhelmsen. *Efficient Ray Tracing of Sparse Voxel Octrees on an FPGA*. PhD thesis, Norwegian University of Science and Technology, 2012.

[Wlo05] Matthias Wloka. Improved batching via texture atlases. *Shader X3: Advanced Rendering with DirectX and*

OpenGL, pages 155–167, 2005.

[X3D11] X3D Working Group. X3dom proposal, 2011.

[XV96] Julie C Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Visualization'96. Proceedings.*, pages 327–334. IEEE, 1996.