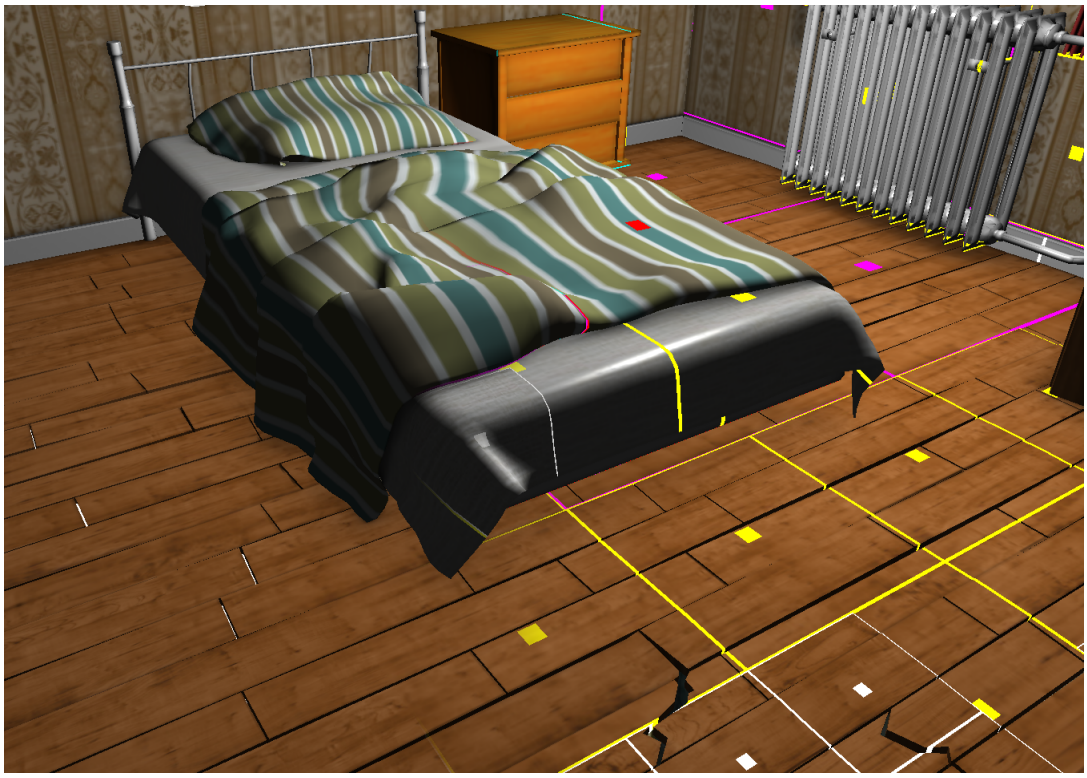
CHALMERS



# Virtual Texturing with WebGL

*Master's Thesis in Computer Science: Algorithms, Logic and Languages*

SVEN ANDERSSON
JHONNY GÖRANSSON

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2011
Master's Thesis 2011:1

**Abstract**

Until recently, achieveing hardware accelerated 3D content on web sites have only been accessible through third party plugins. The new HTML5 standard eliminates this restriction by adding native 3D rendering through the *WebGL* API. This technology brings established desktop applications online, bridging the gap between software platforms.

This thesis describes a successful WebGL implementation of *Virtual Texturing*, a new approach to texture mapping that enables support for theoretically infinite image dimensions. Virtual Texturing is implemented with the help of several elements new to the HTML5 specification, such as Web Workers and Web Sockets. This thesis provides an introduction to these features, as well as an overview of supported WebGL features in modern web browsers. Results show that Virtual Texturing in WebGL is a viable approach for websites with high resource demands, and that browsers are rapidly catching up to meet the demands of complex 3D rendering.

This thesis successfully implements Virtual Texturing in WebGL, to study the restrictions of developing complex 3D content in a web environment. Results show that Virtual texturing in WebGL is viable, and that WebGL has a bright future. Most browsers have, or will in a near future, support for features needed in complex 3D rendering.

## Acknowledgements

Thanks to our families for their support throughout the development of this thesis.

We would like to acknowledge Meindbender for their support, providing us with locale and benchmarking workstations during the development of the prototype and this thesis. We would also like to thank our supervisor at Chalmers, Ulf Assarsson.

Further appreciation is extended towards the author and contributors of the Three.js 3D WebGL framework, for their support on the work performed with their engine. Three.js was used as the base for the prototype created for this thesis.

Sven Andersson & Jhonny Göransson, Göteborg, November 22 - 2011

# Contents

# 1

# Introduction

## 1.1 Background and Motivation

With the advent of the new HTML 5 standard, web developers are now equipped with powerful tools to create interactive multimedia content directly from a universal API without external plugins or third-party tools. Several new elements have been introduced with the new version, such as the video, audio and canvas elements, which enables support for embedded media components in HTML. While the audio and video components are the most interesting for common media use, the canvas element opens up new possibilities for a richer web experience by giving access to new rendering contexts. The HTML5 specification* identifies these as `2D` and `webgl` respectively. WebGL is a web standard for a low-level 3D graphics based on OpenGL ES that extends the current javascript API with direct GPU hardware interaction, exposing the OpenGL drivers through the HTML5 Canvas object. This means that a developer familiar will OpenGL will be able to dive right into WebGL with little effort.

At the moment, high-end consumer graphics hardware is equipped with limited amount of main memory to hold geometry, textures, shader programs and whatever assets the artist needs in a scene. Virtual Texturing (along with its geometrical relative, Sparse Voxel Octree), aim to circumvent these restrictions for texture resources which enables artists to use extremely large images. The underlying idea of virtual texturing stems from the concept of virtual memory [SGG11], a memory management technique implemented in operating systems to deal with limited physical memory. Virtual memory expands the address space beyond the range of addresses within the physical memory that a given process can utilize, thus enabling a program to use more memory than what

*`http://www.w3.org/TR/html5/`

is available. Virtual memory is divided into blocks of data called pages. When addressing virtually mapped memory, the application is unaware of the location of the pages and can reference these as a continuous block of data regardless. Similarly, a page in a Virtually Textured system consists of a segment of texture data that can reside on either main memory or a storage device locally or distributed over a network. While Virtual Texturing and similar texture streaming techniques are common in desktop applications with heavy resource demands, it is also a good candidate for web applications for several reasons:

- **Reduced load times**

  Before an image can be used by the GPU, it must be downloaded from a web server to main memory and then uploaded from main memory to the GPU. This is a lot of data transfer for large images, which could cause the user to wait for a long period of time before anything is shown on screen. With Virtual Texturing, it is possible to show the image with lower quality while a better version is downloaded.

- **Avoiding file size restrictions**

  Web browsers normally restrict resource file sizes. A large image compressed in a lossless format might not be eligible to load by certain browsers. As Virtual Texturing segments images into smaller blocks, large images can be streamed part by part over time.

- **Less intermediate storage needed**

  When loading large amounts of images on devices with limited main memory, the memory footprint quickly rises, which can cause a browser to become unresponsive or ever crash. Virtual pages are dynamically loaded and unloaded in run-time, and only the active set of pages exist in memory. While Virtual Texturing requires more storage space server-side, the memory usage will be spread out over time instead of when the page is loaded.

- **Support for extremely large textures**

  While a browser can typically rasterize arbitrarily sized images, WebGL can only use textures with dimensions supported by the GPU profile. Virtual Texturing enables textures of theoretically unlimited size.

- **Less wasted pixels**

  Depending on texture mapping layout, a texture map for an object might contain a lot of unused areas. When batching textures together in a single unified texture map, this area can be covered by other objects, effectively minimizing wasted space.

- **Fewer state changes**

  Using a Virtual Texture for most, or even all, geometry in the scene will dramatically reduce state changes in the rendering loop, since the Virtual Texture only

need to be bound once per frame.

There exist several texture techniques that deal with these restrictions in various ways. *Tiling* is a texture mapping technique in which the texture is repeated continuously across the surface of an object. This technique has low memory costs, and is specifically useful when rendering large, open surfaces with seemingly repetitive patterns. However, in natural environments, such scenery contain almost infinite amounts of unique details, shaped by centuries of varying weather conditions and other factors contributing to decay. *Texture Splatting* [Blo00], a common technique for adding variety and details, diminishes the repetitiveness of large textured areas by blending between two or more images. Virtual Texturing removes these concerns altogheter, as an artist can create a uniquely detailed surface with an extremely large texture map.

Although virtual texturing requires extra overhead in terms of stream maintenance, server-side storage and real-time texture decompression, implementations have shown that the benefits of visual quality as well as fewer state changes overshadow the processing costs. Furthermore, Virtual Texturing is well suited for existing rendering pipelines and lighting solutions in a game engine or a 3D framework.

## 1.2 Problem statement

The goal of this research is to successfully implement Virtual Texturing in WebGL, and to study the restrictions of developing complex 3D content in a web environment. Furthermore, this thesis will conduct a market review of the capabilities of WebGL, and the viability of Virtual Texturing usage in modern web browsers.

The purpose of this thesis is to provide a comprehensive overview of the ideas behind Virtual Texturing and to evaluate this technique in a web environment. Virtual Texturing is a relatively new approach to texture mapping, and a rather unexplored subject in terms of web development.

# 2

# Previous Work

## 2.1 Mipmaps

Williams et al. [Wil83] presented a new principle for texture filtering to reduce the number of texture samples needed to correctly map textured pixels covering multiple texels. A mipmapped texture consist of a pyramidic layout of textures with descending sizes of the original image down to 1 by 1 pixels, Figure 2.1.

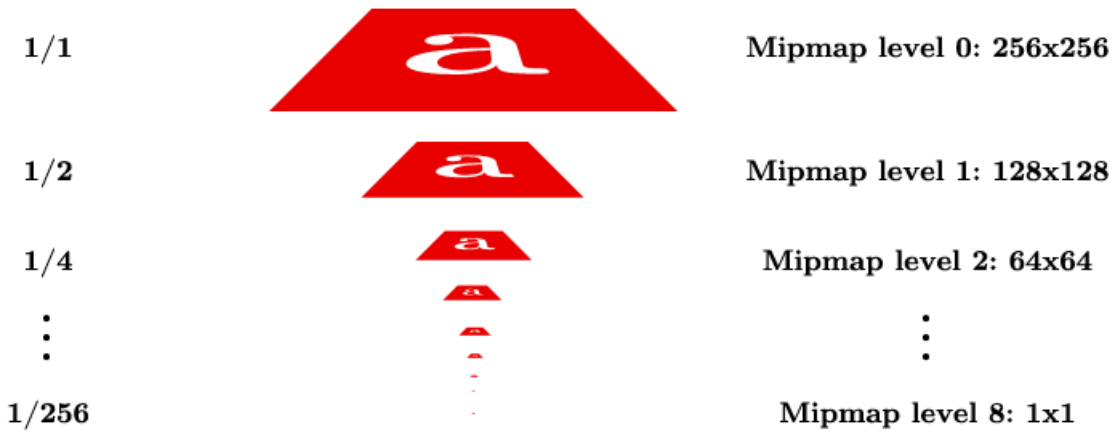



**Figure 2.1:** The pyramidic structure of a mipmap chain.

## 2.2 Clipmaps

Clipmaps [TMJ98] extends on the idea of mipmaps, making it possible to map a larger than otherwise possible texture into memory. This is accomplished by only loading a subset of the higher detailed mipmap levels, called the clipping region. The clipping region is usually positioned in correlation to where the camera is located in the scene, and adjusted according to camera movement, resulting in new parts of the clipmap being streamed in. With this appoarch, geometry close to the camera will strive to have the highest resolution texture compared to geometry further away. This technique works well for landscapes, where the geometry is mapped to a large surface that cannot fit in main memory. The downside of clipmaps is that high resolution samples can only be gathered locally inside the clipmap region.

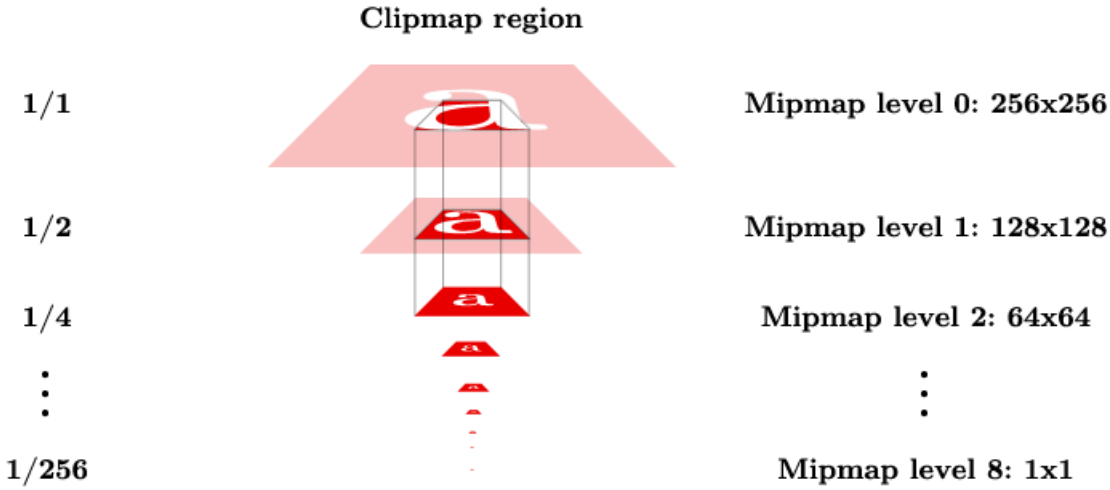



**Figure 2.2:** Clipmap with an example clipping region marked in red. All the mip map levels below 1/2 in this example exist fully in memory.

## 2.3 Texture Streaming

To avoid excessive loading times when increasing the amount of textures and texture sizes in a scene, as well as the GPU memory restriction, a technique called Texture Streaming [vW06] [Has07] is often used. The basic concept of Texture Streaming is to initially only load the lowest mipmap levels of a texture, and stream the higher levels on demand depending on the camera position. To reduce the storage amount needed and the bandwidth requirements of the requested texture, the images are usually compressed and must consequentially be decompressed in real-time, causing varying performance depending on

image format. [vW06] evaluates and describes several different real-time decompression algorithms and formats for rendering textures from large texture databases.

## 2.4 Texture Atlases

A common problem with game development and real-time rendering is reducing render state changes to a minimum [NVI04]. To avoid excessive amounts of state changes, render calls are normally batched together into groups that share the same render state. However, one of the biggest problems with state batching is that several texture changes must still occur to render objects with different textures. One efficient way to decrease the amount of texture changes is to copy textures into one large, unified texture, called a *Texture Atlas*. As several models share the same texture space in an atlas, texture coordinates must be modified so that a texture sampler within a shader program can address the correct sub-rectangle within the texture atlas. While this process can be done manually in modeling software, it is generally easier to incorporate a tool or script that performs this task in the content creation pipeline as models and assets can be produced by different people in different files.

One inherit problem with a texture packing scheme is that bilinear sampling at the border between two subtextures or mipmap levels causes bleeding. This problem exists with Virtual Texturing as well, but can effectively be avoided by adding a border around the texture pages (Page borders are discussed further in 3.9.1) without compromising texture filtering. Another problem inherit to the texture atlas approach, is when a model contains texture coordinates beyond [0,1]. This is usually the case with edge sampling techniques such as clamp-to-edge, repeat and mirror when sampling outside the texture. The different texture modes can still be achieved, but require extra effort in terms of specialized shader programs that produce the same result as a native API that handles sampling outside edges. With this added overhead, the performance might vary irregularly, causing the gain of less state changes to be mitigated by the processing cost of this approach. Another possible workaround to this, NVIDIA mentions, would be to replicate a wrapped texture across the texture atlas in order to simulate wrapping. This technique is more wasteful with space, and restricted by texture size and the amount of wrapping. Virtual Texturing on the other hand, does not suffer from this restriction as one of the major benefits from this technique is that a surface can be uniquely textured down to every single texel that covers it. This means that even though wrapping is not explicitly supported in a Virtual Texturing pipeline, it is not necessary either.

## 2.5 Virtual Textures/Texturing

Virtual Texturing combines ideas of mipmapping, clipmapping, texture streaming and texture atlases to a single complex system that implements the advantage of each tech-

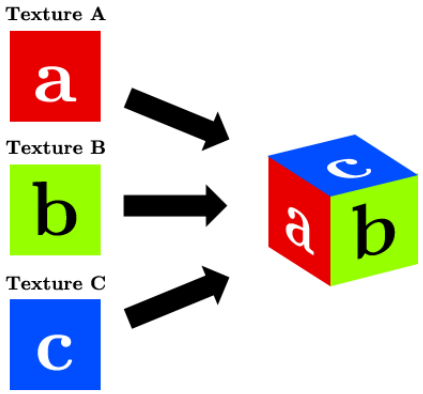



**Figure 2.3:** A cube with three textures, one for each side. To render the cube, three state changes are needed.

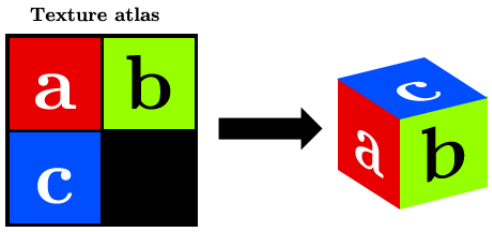



**Figure 2.4:** The cube in Figure 2.3 rendered using a single texture atlas. To render the cube, only one state change is needed since all sides of the cube are mapped with one single texture. The black area in the texture altas indicates wasted texture space. It is generally good practice to minimize wasted space by carefully considering atlas generation algorithms.

nique. Most of the work done in this thesis is based on the excellent work done by Sean Berret on *Sparse Virtual Textures* [Bar08], and Albert Julian Mayer in *Virtual Texturing* [May10].

### 2.5.1 MegaTextures

MegaTextures is an implementation of clipmapping, developed by John Carmack of id Software for the Splash Damage title Enemy Territory: Quake Wars. As with clipmapping, the purpose of MegaTextures was primarily focused on rendering texture mapped outdoor terrain geometry. No official documents exist that describe the in-depth details regarding the MegaTexture implementation, but the gathered pieces of public information is the basis for the Sparse Virtual Texturing developed by Sean Berret [Bar08]. The MegaTexture pipeline was later redesigned and generalized to handle arbitrary geometry as well as terrain, together with the id tech 5 engine for the ID Software titles Rage and the upcoming Doom 4.

### 2.5.2 Sparse Virtual Textures

Sparse Virtual Textures, developed by Sean Berret [Bar08], is based primarily on publicly available information gathered from online discussion boards and email correspondence between Sean Berret and John Carmack on the subject of the MegaTexture implementation. Sparse Virtual Texturing is inspired by the concept of Virtual Memory, an Operating System technique that enables a process to address memory as if it was present in main memory at all times. Similarly, Sparse Virtual Textures enables 3D applications to address every texture, or rather one large texture, as if it had access to the whole texture/all textures inside the graphics memory at all times.

## 2.6 Sparse Voxel Octrees

Sparse Voxel Octrees [LK10] represents geometry as voxels, stored in a hierarchical octree of axis-aligned cubes that encapsulates a volumetric portion of the model. Voxel technology is traditionally used in the field of medicine together with various 3D scanners, such as MRI and x-rays, that generate spatial information from real objects. In order to achieve high quality rendering, an extremely large data set is required which is similar to Virtual Texturing in the sense that the data can thus be streamed in to main memory on a need basis (determined by LOD distance).

Rendering a voxel model is performed by a ray tracing algorithm that performs intersection tests against the nodes of the tree. Laine et al. performed tests of using traditional rasterizing rendering, but concluded that it not only performed worse than ray casting, but also led to inexact solutions when voxels that did not map to one pixel or smaller were used. While Voxel Octrees are best suited for static data, it is still possible to use a mix of raster graphics and voxel traversal depending on application. Voxel technology could potentially be the next evolutionary step in rendering technology, and is an interesting candidate for next generation rendering engines such as ID Tech 6 [Shr08] of ID software.

# 3

# Virtual Texturing

## 3.1 Overview

The Virtual Texturing pipeline is divided into several individual parts that performs a specific task. Each part can be executed out-of-order and infrequently over time, which yields a more stable framerate with comparable visual quality. The process starts with the *Page Determination stage*, where the virtually textured geometry is rendered to a Frame Buffer Object (FBO) and analyzed by a shader program that outputs the active set of pages used within that frame. Since the FBO is only accessible on the GPU, it must be copied to main memory so that the needed pages can be signaled for streaming in later stages of the pipeline.

The page data, i.e. the texture segments, are copied from an external location to a physical texture located in the GPU. This texture is called the *page cache*, and is used by the virtually textured geometry in the final render pass. Because incident segments in the page cache are not necessarily neighbors in the virtual texture, unwanted bleeding can occur across the borders when utilizing hardware filtering. These artifacts are removed

| **This Thesis** | Mayer [May10] | Mittring [MG08] |
|---|---|---|
| Virtual Texture | Physical Texture | Tile Cache |
| Indirect Texture | Pagetable Texture | Indirect Texture |
| Page Determination | Tile Determination | Computing Local LOD |

**Table 3.1:** Different terminology for each Virtual Texturing part in different papers.

by applying a one pixel border to the pages. Borders are explained in more detailed in the filtering section 3.9.1.

After the set of needed pages has been determined, the page data must be compared to the contents of the page cache so that missing pages can be identified. In order to cover as many pixels as possible with new data while still minimizing page requests, this stage makes sure that the most referenced page is prioritized when the streaming begins.

The page streaming is performed asynchronously behind the scenes, which enables the system to maintain full interactivity with the user. When a page is loaded into main memory, the new texture data must be uploaded to the page cache texture. Depending on the current streaming method, the compressed image data is decoded in real-time either by the browser, or by third-party libraries in JavaScript. Three different streaming methods are covered in this thesis; simple streaming (3.5.1), streaming with web workers (3.5.2) and streaming with web sockets (3.5.3). If the cache is full, a page-replacement algorithm is executed which finds a suitable candidate to swap with the new page. When this process is complete, the page cache and page cache texture are updated accordingly.

An overview of the pages and the layout of the Virtual Texture on the web server is displayed in Figure 3.1. An overview of the different parts of the technique client side, is displayed in Figure 3.2.

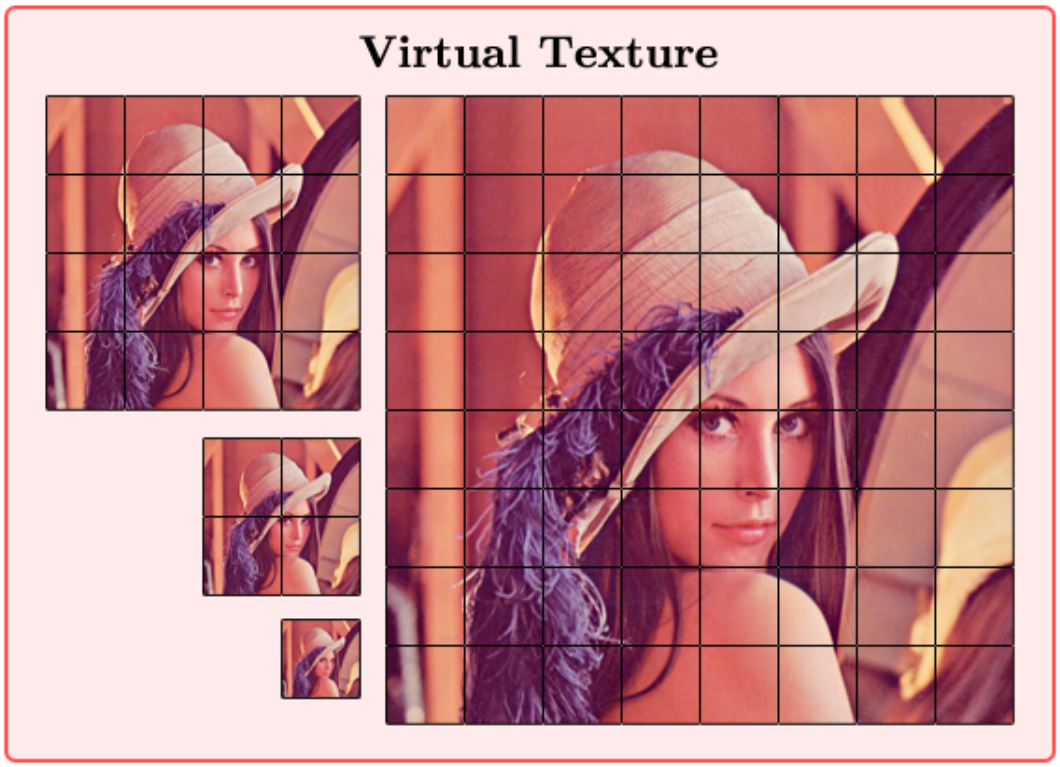



**Figure 3.1:** Virtual Texture overview. Each image block represents one page in memory.

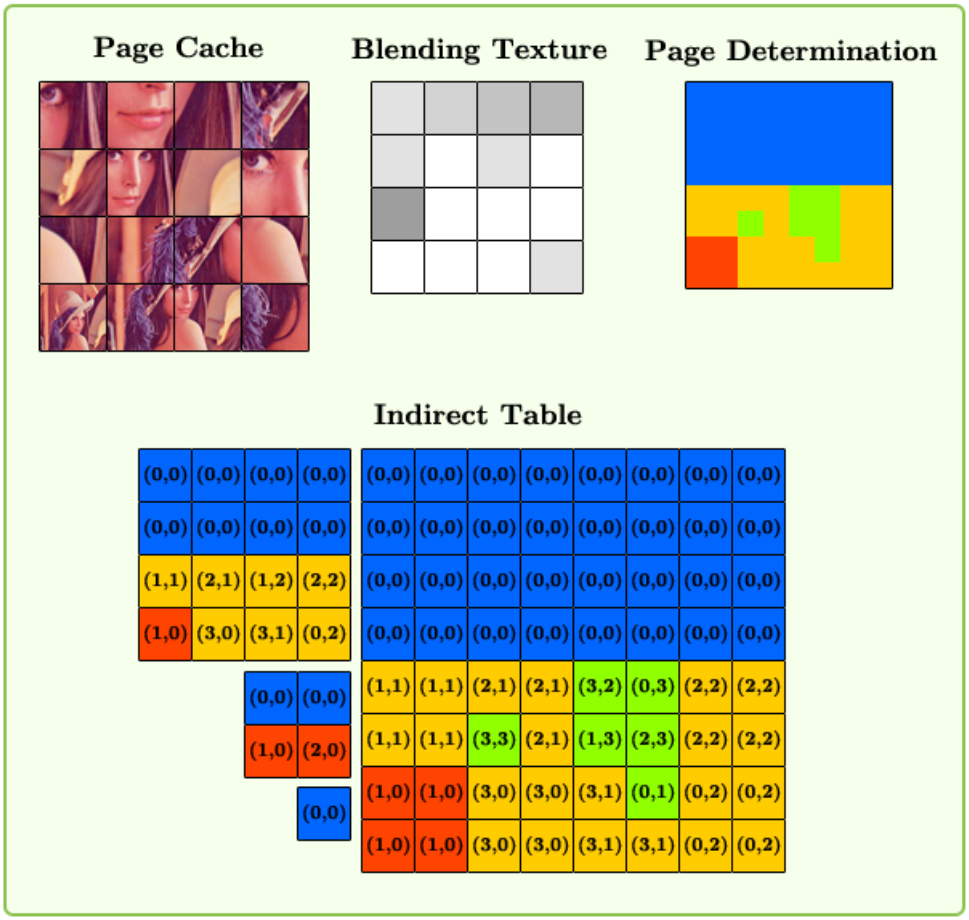



**Figure 3.2:** Browser data and structure overview.

## 3.2 Implementation Details

The prototype was developed using Three.js [Cab10], an open-source 3D framework for WebGL, which supports scene loading (represented in the JSON* file format), management as well as common vector mathematics. The scene used in performance testing, "The Bedroom", was originally modeled by David Vacek and was part of a lighting challenge on Cgsociety in late 2009. The scene was exported in Blender 3D to JSON by using a slightly altered exporting script provided by the Three.js library. A virtual texture generation tool, VTGen, was developed in Python with a mix of common libraries for image manipulation, data encoding and OS functionality.

*JavaScript Object Notation

## 3.3 Pre-Processing

After the textured scene has been properly exported to a JSON file, it must be passed through an offline virtual texture generator that constructs the virtual texture pages from the original texture maps. The virtual texture generator iterates over all the unique textures used in the scene, and packs these into a virtual space defined by the dimension of the specified virtual texture dimensions. The texture packing is implemented with a greedy texture atlas generation algorithm that splits the virtual texture space recursively into a binary tree. Every leaf in the tree ultimately contains one unique texture. When the algorithm has finished executing, the textures in the leaves have been rendered with trilinear filtering onto page-sized squares and saved to compressed image files by using the Python Image Library*. Every mipmap level in the mipmap chain is generated in this process as well, all the way down to the coarsest mip-map (covering the entire Virtual Texture) contained in one single page.

Currently, VTGen assumes that geometry is UV mapped before processing. While this approach is simple and gives perfectly good results, a mesh parameterization [HLS07] [Fen04] scheme that automatically unwraps the geometry and renders the texture maps to the Virtual Texture accordingly, could be a possible future addition. This technique would give more tightly packed virtual textures than pre-mapped UV, since empty texture space around isolated objects could be shared by several objects. This approach is explained in [MG08].

## 3.4 Page Determination

As all pages from the virtual texture cannot fit in the page cache at the same time, it has to be determined which pages needs to be in the cache at a given time, and which of these that get priority in the page request queue. This process can be divided into four separate steps:

- Render Needed Pages
- Read Out Pixel Data
- Process Pixel Data
- Request Pages

There is an expected latency from demand to delivery of page requests, which stems from the combined latency in all steps of the page determination pass as well as the latency added by the different streaming methods. This combined latency will result in pages taking several frames to appear in the page cache from the initial request. It

*http://www.pythonware.com/products/pil/

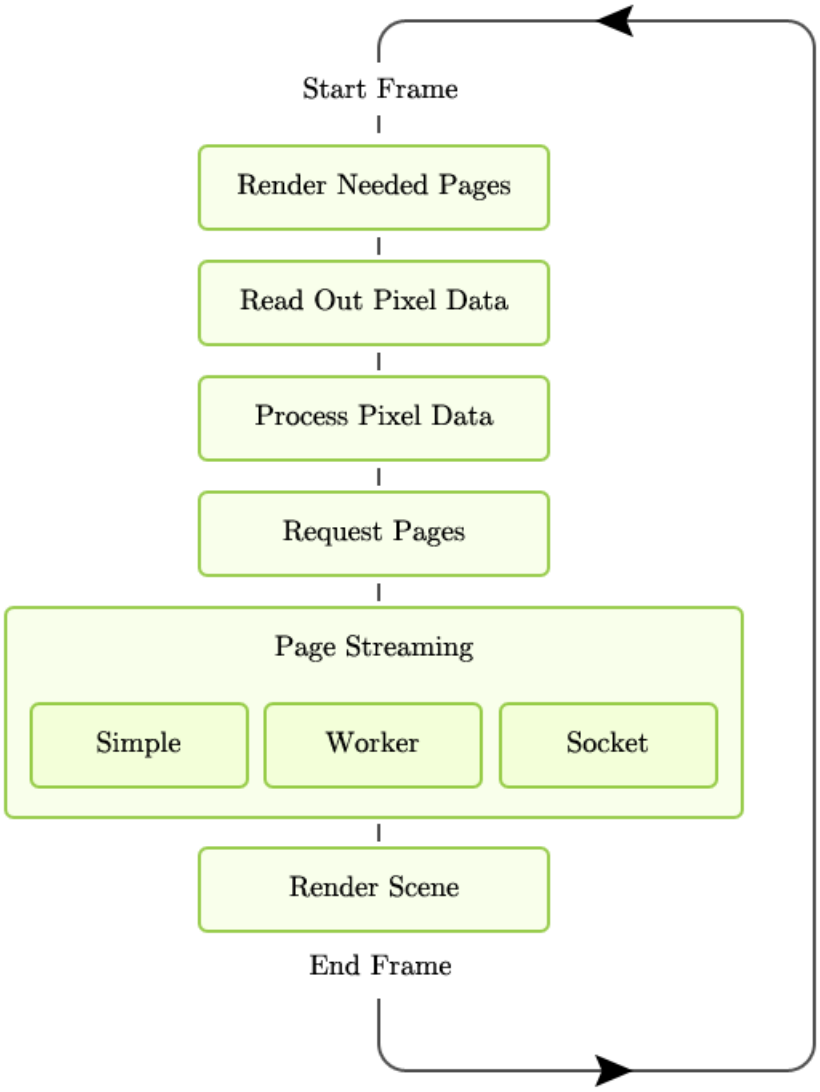



**Figure 3.3:** Workflow of the rendering loop. An important thing to note is that not every step has to be executed each and every frame. For example, the steps can be divided such that only one step is executed each frame. This will however increase the time it takes from determining that a page needs to be streamed in, to where the page is finally streamed and uploaded to the page cache. Careful consideration has to be taken when balancing this, so that the CPU/GPU usage is maximized while still having a stable framerate.

would be beneficial to know the exact pages that are needed in advance, so that they could be requested and streamed before they are visible in the viewport.

An exact page prediction is generally hard to conduct, but can be approximated in scenes where the camera path is either known in advance or restricted to certain areas of

the scene at a given point in time. A realistic scenario of this behavior could be a racing game or an indoor first-person-shooter, where the location of the player is restricted by linear gameplay. In scenes with fast and virtually unbounded movement, designing a preloading scheme is less intuitive due to non predictive user navigation. However, [Neu10] suggests several prediction schemes based on heuristics, that mitigates the effects of this latency in various scenarios. This thesis explores an exact determination algorithm with only a slight modifcation to the page determination rendering by increasing the field of vision (FOV) of the scene camera. This means that since areas around the current camera position are visible in the viewport, more pages can be considered for streaming in this pass. This idea is based on the notion that pages in close proximity of the current camera are likely to be visible in the frame in subsequent frames.

### 3.4.1 Render Needed Pages

The first step of the page determination process is to find out which pages are visible in the current frame. This is accomplished by rendering the scene to an offscreen buffer with a fragment shader that outputs the information that is needed to calculate visible pages in later stages. The rendered per-pixel data consists of an estimation of the mipmap level (similar to how OpenGL performs this task internally) and the texture coordinates of the Virtual Texture for the given pixel.

The size of the render buffer is an important aspect to consider. The preferred solution in this case, would be to use the same size as the view buffer, which would result in a ratio of pixel count to page need count of 1:1. [May10] shows that the minimal buffer size for correct page determination is a 1/8 of the view buffer.

One issue with WebGL development in its current state, is that established OpenGL extensions are not implemented in all WebGL enabled browsers. One such extension is the `GL_OES_standard_derivatives`, which enables the derivative functions `dFdx` and `dFdy` in GLSL shaders. [May10] supplies shader code that utilize this extension for the page determination rendering pass, which is also the approach used in this paper. This extension is currently only available in Google Chrome and the alpha release of Mozilla Firefox (See Table 4.2).

However, by using hardware texture filtering, local derivatives can still be achieved in browsers without extension support. [Fer04] describes how encoding increasing mipmap levels into a small lookup texture can be used to perform these calculations when rendering. Every pixel in each mipmap level of the lookup texture is assigned a number representing its actual level in the mipmap chain. When sampling this texture with hardware filtering, the interpolated value between two mipmaps is retrieved. This floating point value is discretisized into the correct mipmap level by using the GLSL function `floor`. This method was tested in this implementation, and can be used as a fallback method for browsers that lack this support in future versions.

Listing 3.1 show the modified fragment shader code used in this implementation.

Listing 3.1: Page Determination pass fragment shader

```
#extension GL_OES_standard_derivatives : enable

/*
 * Page Determination fragment shader, based on code by
 * Albert Julian Mayer presented in Virtual Texturing (2011),
 * which in turn is based upon Barretts SVT demo shader.
 */

const float readback_reduction_shift = 2.0;
const float vt_dimension_pages = 128.0;
const float vt_dimension = 32768.0;

const float mip_bias = 0.0;

varying vec2 vUv;

// analytically calculates the mipmap level similar to what OpenGL
    does
float mipmapLevel(vec2 uv, float textureSize)
{
  vec2 dx = dFdx(uv * textureSize);
  vec2 dy = dFdy(uv * textureSize);
  float d = max( dot(dx, dx), dot(dy, dy) );

  return 0.5 * log2(d) // explanation: 0.5*log(x) = log(sqrt(x))
         + mip_bias - readback_reduction_shift;
}

void main()
{
  gl_FragColor.gb = floor(vUv.xy * 255.0) / 255.0;
  float miplvl = clamp(8.0 - mipmapLevel(vUv.xy, vt_dimension), 0.0,
      8.0);
  gl_FragColor.r = miplvl / 255.0;
  gl_FragColor.a = 1.0;
}
```

### 3.4.2 Read Out Pixel Data

The next step is to transfer the rendered pixels from the GPU to main memory. In the standard OpenGL API, there are two ways of doing this: `glReadPixels` and `glGetTexImage`. [May10] performed benchmarking of both methods, which showed that using `glReadPixels` on a FBO is the fastest method. No such comparison can currently be conducted with the WebGL API, as `readPixels` is the only function supported of these two according to the specification [Gro11b].

Since the rest of the application is halted until all of the data has been copied to main memory, the pixel read out step is one of the most critical performance bottlenecks

in the page determination pass. It is preferred to do memory download asynchronously, so that the CPU can continue executing operations while the download occurs. This can be accomplished by using Pixel Buffer Objects (PBO), which is an extension to OpenGL making it possible for both asynchronous upload and download to and from the GPU. However, as PBOs are not supported by OpenGL ES / WebGL, other optimizations had to be evaluated.

To reduce the time it takes to read out the image data, a minimum amount of read out data is preferred. In the current implementation, only three components are needed to reference a page in the shader, which fits perfectly for image formats with three color channels such as `RGB`. However, the current specification of WebGL only allows a texture read back with four channels (the `RGBA` format). This indicates a data waste of 25%, which could lead to a measurable increase in read out latency. Hopefully this will be addressed in future versions of the WebGL specification.

### 3.4.3 Process Pixel Data

When the pixel data has successfully been read back to main memory, the information in each pixel need to be converted into actual page indexes. Listing 3.2 displays a simplified version of the pixel to list conversion used in this prototype.

**Listing 3.2: PD pass list conversion**

```
// Loop all pixels in the page determination texture
var needed_pages = {};

for (var i = 0; i < pixels.length; i+=4) {
  var indx_mip = pixels[i];          // R = Mipmap level
  var indx_x = pixels[i+1] / 256.0; // G = s texture coordinate
  var indx_y = pixels[i+2] / 256.0; // B = t texture coordinate
                                     // A = unused

  indx_x = Math.floor(indx_x * __mipSizes[indx_mip]);
  indx_y = Math.floor(indx_y * __mipSizes[indx_mip]);

  [...]

  // calculate VT page index from pixel data
  var index = __mipOffsets[indx_mip] + indx_y * __mipSizes[indx_mip] +
      indx_x;

  if (needed_pages[index] != undefined)
  {
    // increase hit count for the page
    needed_pages[index].hit += 1;
  }
  else
  {
    // first time we needed this page,
```

```
      // set hit count to 1
      needed_pages[index] = {};
      needed_pages[index].hit = 1;

      needed_pages[index].miplvl = indx_mip;
      needed_pages[index].x = indx_x;
      needed_pages[index].y = indx_y;
    }

    [...]
  }

  /*
   * needed_pages now contains a list of pages needed.
   *
   * We still need to check which of these already exist
   * in the page cache.
   */
```

Furthermore, it is also required to identify if the needed pages already exist in the page cache in order to avoid redundant page requests. This would be possible to do in the same loop that converts the pixel information to a list, or in a separate loop operating just on the resulting list. Performing this check for every pixel in the conversion loop would result in pages of the same index to be checked multiple times each frame. Instead it would be better to only do this check on the resulting list, since this list is much smaller in size than the copy of pixel data gathered from the read out step.

This method is a straight forward solution, and depending on the amount of pixels that needs to be converted, might vary in processing time. [May10] presents an alternative way of processing the pixel data into a needed pages list by using the OpenCL* framework. This approach would decrease the amount of data needed to be read back to the CPU, and would help decrease the transfer time.

While a web-based equivalent of the OpenCL framework, WebCL, is currently under development by the Khronos Group, Nokia† and Samsung‡ has already released prototype extensions for the Firefox browser and the WebKit layout engine. Even if it is possible to utilize these libraries to implement a WebCL version of the OpenCL approach, the need for third-party plugins is undesirable in the current state.

### 3.4.4 Requests Pages

When a list of needed pages has been created, a loop iterates through the list and requests the page with the highest reference count. The page is requested via one of the three

*Open Computing Language

†http://webcl.nokiaresearch.com/

‡http://code.google.com/p/webcl/

streaming methods presented below (see 3.5). The selected page is removed from the list of needed pages so that it is ignored the next time a stream candidate is selected.

Depending on the requirement of request-to-delivery and need for smoother updates of visual quality, more sophisticated page selections can be used. One addition featured in this paper, is the option to force loading of the mipmap parents of a page before the actually needed page was loaded. This resulted in a smoother quality stepping, but did not fully eliminate "pop in" artifacts*.

## 3.5 Page Streaming

When a new page candidate has been selected for a request, the current page streaming method deals with it thereafter. In a desktop setting, these requests would be handled by low level calls that load the pages from a local medium as hard drives, optical disks or even main memory. In WebGL, the image data of the virtual texture is located on a remote web server and will have to be streamed over a computer network. Asynchronous loading is a crucial feature to avoid major stalls in the render loop. Ideally, the streaming would be performed in a separate thread and processor, utilizing multiprocessing functionality. While concurrency is easily achievable in modern operating system APIs, developing for a web platform is fairly restricted in this area but still possible with the help of HTML5 Web Workers (see 3.5.2).

### 3.5.1 Simple Page Streaming

JavaScript browser APIs provides asynchronous remote file loading capabilities through the `Image` object, that loads the images over a network connection transparently. Image objects have the advantage that they can be passed directly to WebGL for easy texture creation, utilizing fast internal image decompression in the web browser. The main disadvantage of standard image loading via JavaScript is that is not executed concurrently which causes stalls in the render loop, leading to lower framerate.

### 3.5.2 Streaming using Web Workers

*Web Workers*† is a feature in HTML5, that allows JavaScript code to be executed concurrently in the browser. However, Web Workers does not have direct access to the DOM‡

*A "pop in" is a visual phenomena that the user experience when a geometry/texture noticeably changes level of detail.

†`http://dev.w3.org/html5/workers/`

‡Stands for Document Object Model and is a platform independent way of interacting with the web page data in JavaScript.

Tree, and supports only a limited amount of functionality compared to the main execution script. Communication between the main thread and web workers is performed by event-driven message passing. Web Workers does not share memory with other threads, which means that messages must be serialized at both sides of the communication channel. Depending on browser, the data in a message can contain strings, structured data such as JSON dictionaries or binary data arrays.

Loading an image with a Web Worker requires the raw image data to be passed back to the main thread once loading is done, since Web Workers cannot create `Image` objects by themselves. However, to create Image objects from raw image data, the data has to be encoded with *base64* using the Data URI Scheme*, see 3.5.4. The object would have to be allocated in the main thread explicitly, using the data streamed via the Web Worker. The image data, if decompressed manually in JavaScript, can also be loaded directly into WebGL. This eliminate the need to create Image objects on the main thread when the data has arrived.

The latter was implemented in this thesis by using third party libraries to decompress JPEG and PNG encoded files inside the Web Worker script, before passing the data back to the main thread. The option involving base64 encoded files was also evaluated. A performance comparison can be examined in Figure 4.11 of Section 4.4.

### 3.5.3 Streaming using Web Sockets

The *Web Socket*† interface enables data transfer by opening a single bi-directional communication channel between client and server without the use of a standard HTTP connection. This means that a server can feed any data through this channel, bypassing the HTTP overhead with each page request. The Web Socket protocol, currently at draft 17 [Gro11a], is under constant development and is not uniformly supported by all web browsers‡.

The socket is initiated with an authentication handshake, that validates the safety and correctness of the proposed connection. Initially, the client sends a data header that contains information about the host, protocol version and a security key. On the server side, the data fields in the header is parsed in several steps to ensure the validity of the client. When the server has finished this task, a response message with a modified key based on the security key found in the data header is sent back to the client, finalizing the connection.

In the last stages of the prototype development, a successful implementation of the Web Socket protocol was integrated into the Virtual Texturing page request module. For this purpose, a special purpose web socket server was built to service the page data

*Stands for Uniform Resource Identifier, and is a way to represent and identify web page resources on the internet

†`http://dev.w3.org/html5/websockets/`

‡An unofficial list of web socket, as well as other web features, can be found at `http://caniuse.com`

through the connection. While increasing the complexity of the solution as well as the need for access to a dedicated machine to run the server software on, the advantages of the socket approach are beneficial. First of all, removing the need for several HTTP connections enables more data throughput between client an server, which ultimately leads to more serviced page requests and better overall quality. Secondly, a special purpose server enables more creative ways of data management server-side. One such feature tested in this prototype was naive page caching functionality, that stored referenced pages in memory. This data is shared between threads server side, and helps reduce the request-to-arrival time for pages often referenced. Non-web based Virtual Texturing approaches normally stores the entire texture sequentially in a binary blob, which is not preferable memory-wise in a normal web server. A web socket server on the other hand, could manage the Virtual Texture similarly, enabling functionality such as transmitting large image blocks containing several pages or full/partial mipmap chains.

### 3.5.4 Base64 Encoded Images

To create `Image` objects dynamically using image data, the data has to be encoded as a base64 encoded string using the aforementioned Data URI Scheme. This makes it possible to stream pages in Web Workers, transmitting the base64 encoded image data to the main thread where it can be used to create `Image` objects on the fly. The benefits of this approach is that threaded page loading is achieved without custom image decompression.

The base64 streaming method requires that each page in the Virtual Texture is base64 encoded before loading begins. This can be performed offline in the preprocessing tool VTGen, or in a separate script when all the pages have been stored. The drackbacks of using base64 encoded images is that they require not only 30% more disk space but also need one more decode pass from string to binary data, which could possibly significantly effect rendering framerate since it is done in the main thread.

### 3.5.5 Manual PNG and JPEG Decompression

One problem encountered when utilizing JavaScript `Image` objects, was that the memory footprint steadily increased during the lifetime of the session and ultimately caused a page crash. Due to the nature of Virtual Texturing system, thousands of page requests will occur during run-time, and as web browsers normally cache `Image` objects, main memory is bound to fill up eventually. To deal with this problem, the image data would have to bypass browser image handling and instead be processed and loaded directly into WebGL with JavaScript only.

Decompression was also done in the Web Worker as soon as the compressed image was streamed, and the decompressed data was sent back to the main thread where it could be loaded into textures directly.

## 3.6 Page Cache

The page cache is a large texture that contain the set of pages that have been streamed from a storage unit. When rendering, the page cache texture is only bound once for all virtually textured object in the scene, effectively reducing the amount of state switches that normally occur with individually textured objects. The cache texture is sampled with point sampling and contains no mipmaps. Each cache cell is updated by calling `subTexImage2D` on the texture region that should be updated with new image data.

When the cache is full, a pseudo least-recently-used (LRU) page replacement scheme is executed that finds a suitable candidate to swap with an incoming page. The cache is represented in main memory by an array of JavaScript objects for every page. Every object in this structure contains state information for each page, such as cell position in the cache, page index, mipmap information and a counter that indicates the last frame a page was referenced. The page replacement algorithm iterates over these pages in the cache and compares their frame reference to current frame counter and discards the first candidate that fails this comparison. The next time the page cache is full and a new page needs to be inserted, the page loop will start from the last position and continue iterating, wrapping around to the beginning when the end has been reached. For every page that is either removed or added from and to the cache, the indirection table and the indirection texture must be updated accordingly.

## 3.7 Indirection Table

The indirection table is an array representation of all the available pages in the virtual texture hierarchy. Each element represents one page with information on where to find the corresponding page in the page cache. If a specific page is missing from the page cache, the indirection table element instead points to a page in the page cache with a lower mipmap level as a fallback mechanism. This means that a surface with pages that are not availible in the page cache texture yet, will be mapped with a lower quality page. Therefore, the page that covers the entire Virtual Texture is always availible in the page cache so that no unmapped surfaces are rendered.

As soon as a new page has been successfully streamed in to the page cache, the indirection table updates the affected elements. Each element stores x, y and mip map level information where the page can be found in the page cache. The table has to be uploaded into GPU memory so that the shader program used in the final render pass can use it to find the location of a specific page in the page cache. The indirect table is managed in such a way that its elements map directly to RGB values and can thus be copied to GPU memory in a single call.

## 3.8 Blending Texture

When new pages are available in the page cache, the user might experience a sudden change in texture quality, commonly known as a ”pop in”. This sudden change in quality could be distracting for the user, and should be minimized as much as possible. There are ways to minimize the amount of quality difference with each change, covered in 3.4.4, by making sure the parent (i.e. the page that covers this page, but in a higher mip map level) is loaded before each requested page. However, this will only decrease the amount of change, a pop in can still be noticed. To avoid a very noticable pop in, each new page can be gradually blended with the current page in the page cache. One way to do this, presented in [vW09], is to upscale a previously used page as soon as a better page is a available, and then continuously update the page cache with blended data of the new and previous pages. This could possibly lead to high bandwidth usage to the GPU, depending on implementation. An alternative way is to do the blending manually in the final texturing shader*. A blending value for each entry is stored in the indirect texture/table corresponding to a blend amount for each page in the page cache. These values are subsequentally updated in each frame, until they are fully blended in the page cache.

In this paper, a modification of the second alternative that uses a separate texture with blending information for each page in the page cache is implemented. This reduces the number of entries that needs to be updated each time a page blend value changes, compared to updating each affected entry in the indirect texture.

**Figure 3.4:** A simple example of a new page with better quality being gradually blended in over lower quality pages. With blending disabled, the intermediate states between the first and last would be skipped, and the newly streamed page would appear to ”pop in”.

## 3.9 Filtering

### 3.9.1 Bilinear

To get good looking filtering on the rendered scene, using the page cache as texture source, at least bilinear filtering (`LINEAR` inside OpenGL/WebGL API) can and should

*i.e. the shader that uses the page cache and indirect table to texture everything in the scene.

be enabled. However, since pages in the page cache very rarely will be ordered close to their original adjacent pages in the physical texture, filtering errors will occur on page edges, causing page bleeding.

To overcome artifacts caused by page bleeding, [Bar08] proposes that a small one or two pixel border with pixels from adjacent pages should be added to each page. This feature requires more data for each page, which will increase the page size. To maintain power-of-two dimensions, the pre-processing tool performs a slight downsample on each page to support this added border. It is important to notice that the shader that samples the page cache has to compensate for the added border.

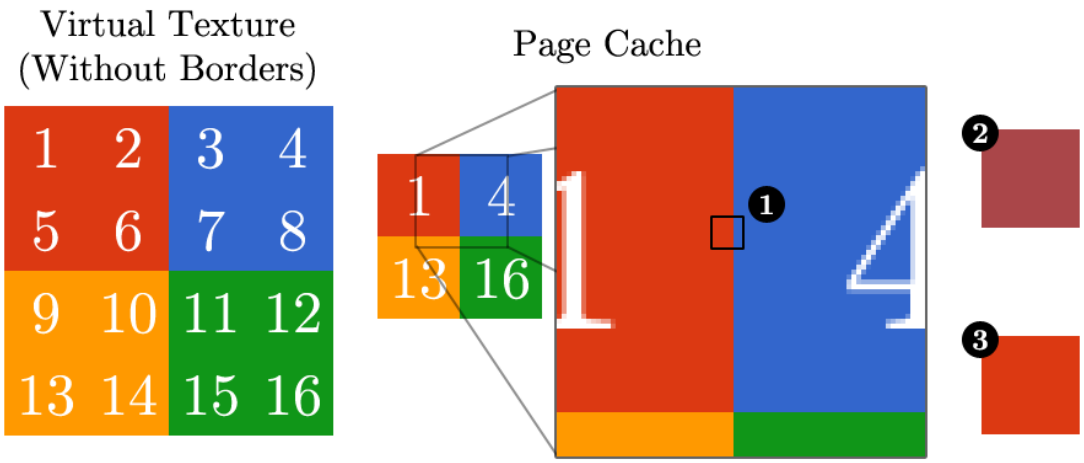



**Figure 3.5:** Example of a Virtual Texture without page borders, and where pages in the page cache are positioned in such a way to demonstrate color bleeding between pages. 1) A bilinear sample inside the page cache, on the edge of page 1. 2) The incorrect resulting color sample, due to color bleeding from page 4. 3) The correct expected color.

### 3.9.2 Trilinear

With only bilinear filtering of the page cache, filtering artifacts can still occur where the sampling changes from one mip map level to another. Ideally, the standard mechanisms in WebGL for trilinear filtering would be used to perform this task, as in the bilinear filtering case. However, this would require rebuilding the mip map chain every time the page cache is updated. But, as [May10] points out, since the page cache already contains pages from different mip map levels, a full mip map chain is not needed, only one extra level. This thesis explores a second trilinear approach, where filtering is accomplished inside the shader, using only the pages available in the page cache (i.e. without any extra mip map levels added). This is performed by figuring out which pages (of different mip map levels) that are ”needed”, and then interpolating them as closely as possible to the WebGL trilinear filtering. To get an interpolation value between these two mip map levels, a trilinear filtered sample is taken from a look-up-texture (Figure 3.6) with

different values for each mip map level. Two different entries from the indirect table and their corresponding page cache are then linearly mixed with the previously gathered interpolation value.

**Figure 3.6:** Content of a mip map look-up-texture. Sampling from the texture with trilinear filtering will provide interpolated values between the different shades of gray. This value is used as a bias value when mixing between two pages of different mipmap levels, enabling trilinear filtering.

### 3.9.3 Filtering in the Virtual Texture

The selected filtering method for generating the different mip map levels in the virtual texture has great influence on the final visual quality in the scene. This thesis generates the Virtual Texture pages by utilizing trilinear filtering from the standard OpenGL API. However, since generating the physical texture is done in a pre-processing step, more sophisticated ways of performing downscale filtering could be implemented.

## 3.10 Texture compression

A common practice in desktop usage of the standard OpenGL implementation, is to use a special group of texture compression algorithms called DXT* which can reduce the size to a ratio of 4:1 for images with alpha channel or 6:1 for images with three color channels. While the compression ratio is quite low for applications that require large amounts of storage space, the benefits of DXT is that it is supported directly by modern graphics hardware, and no decompression has to be done on by the CPU.

Unfortunately, WebGL does not have any support for loading images compressed in DXT in its current version. It is still possible to transfer compressed images in such formats from the server and decompress them in JavaScript before uploading to GPU memory. However, decompressing such formats before uploading them to the GPU would defeat the purpose, as the data is then uncompressed in the GPU memory and would take as much space as any other uncompressed texture.

*Also known as S3 Texture Compression (or S3TC for short)

## 3.11 Texture sizes

### 3.11.1 Virtual Texture size

Theoretically, a virtual texture is unrestricted in size. Practically, on the other hand, factors exist that affects the maximum size of the Virtual Texture, which in turn regulates the size of the pages.

One of the most crucial aspects regarding texture size is the format of the page determination read out. Since the page determination needs to be able to represent the index of every available page in the virtual texture, stored in the RGB channels, the format and type of the page determination texture plays a big part on performance. The most frequently used format for internal storage of textures is `unsigned byte`, which would give each channel a maximum of 256 different values. With this in mind, storing `x` and `y` coordinates in two channels each, the page determination can index up to 256 by 256 pages at lowest mipmap level. It would be possible to index more than 256 values in each channel if a floating point texture is used instead. This type of texture is only available in browsers with the WebGL extension `OES_texture_float` enabled. However, due to the added storage size when using floating point textures (64 bits for FP16, 128 for FP32 and 32 for UINT), the read out latency increases.

The execution of this module can also be distributed over several frames, performed by only copying parts of the buffer at a given frame. While this feature reduces the data amount to transfer between GPU and main memory during each read out, profiling this stage in the prototype showed a constant latency regardless of the amount of the data that was copied (See 4.10).

Using a maximum of $256^2$ pages (in the highest mip map level) in the virtual texture gives the actual size as $virtualtexture_{size} = 256 * page_{size}$ (assuming the use of an `unsigned byte` texture).

### 3.11.2 Indirect texture size

The indirect table and the virtual texture has a correlation in size, since every entry in the indirect table and texture will reference a corresponding page in the virtual texture. The size of the indirect table texture will be exactly the amount of pages possible to index, i.e. 256 by 256 pixels if an `unsigned byte` texture is used.

### 3.11.3 Page Cache size

The page cache texture could be any size between the size of a single page and the maximum amount of page cache cells the indirect table can index. If using a `unsigned byte` texture, the maximum page cache would be the same as the size of the virtual

texture. Naturally this would not be possible, as the purpose of virtual texturing is to overcome the memory restrictions posed by the GPU hardware.

The page cache size can be calculated as $pagecache_{size} = page_{size} * 256$ (assuming the use of `unsigned byte` texture), as long as it's equal or below the maximum texture size the GPU accepts.

### 3.11.4 Page size

The page size of the Virtual Texture can vary from as small as one by one pixels up to the size of the maximum texture dimension supported by the GPU. The page size must be carefully selected before generating the texture pages, as it heavily impacts the performance of the streaming in the real-time application. There are several arguments that can be considered when determining page sizes:

- Small Pages
  - Less sensitive to fast camera movement
    In applications with a dynamic camera, its likely that requested pages are loaded into the page cache at a time where they are no longer visible in the scene, wasting execution time and space in the page cache on redundant pages. As the ratio between serviced page requests and page sizes increases with smaller page sizes, more visible pages can be stored in the page cache.
  - Increased total page count
    When decreasing the page size, one more level is needed in the mipmap generation which produces four times as many pages as the previous level. With this in mind, the required storage space for the page data increases rapidly with smaller page sizes and, depending on server access and storage availability, could potentially be a restricting factor.
  - Reduces wasted space
    With pre-UV unwrapped geometry, Texture maps of individual objects will most likely contain areas that are not mapped to any geometry. This will cause wasted pixels in all the mipmapped levels of the texture, and more importantly in the page cache of the Virtual Texture. With smaller page sizes, a higher number of pages are more likely to contain such empty areas, and can be completely excluded from the page generation.
- Large Pages
  - More data per page
    Evidently, more data has to be streamed and decoded per page in relation to page sizes, which increases execution per page request.
  - More page swaps
    The amount of pages one page cache can store is directly related to the size

of one page, using large pages reduces the amount of pages that fit into the page cache. The lower number of pages the page cache can hold increases the chance it will have to replace older pages in the cache with newer ones, even if the replaced pages are currently in use.

- Less overhead
  Large pages require less pages to map a textured surface with the corresponding section in the virtual texture.
- Less space needed
  As previously mentioned, the size of the complete virtual texture and its mipmap chain is directly dependent on the size of the dimension of one page.

To gain the most performance out of Virtual Texturing, finding a good balance between these components is very much dependent on the needs of the application. The most common dimensions for page sizes range from $64^2$ to $256^2$. Benchmark data gathered from this prototype shows how these factors affect the various parts of the pipeline (see Chapter 4).

# 4

# Results

Since the scene is required to have extremely large textures to utilize the virtual texturing technique, all the objects in the scene are textured individually with color maps of up to $8192^2$ pixels each. This is a perfect example of a scene that normally would not be able to be rendered in real time under most circumstances. Even with all the objects in the scene textured, there was still roughly two thirds space left unused in the Virtual Texture, showing that even larger textures could possibly be used (see Figure 4.1).

Each test case consisted of several consecutive benchmarking runs that took roughly two minutes to execute. During the benchmark runs, the scene camera was animated along a predetermined path, as to ensure a level of consistency between tests. All tests were conducted with a page cache size of $4096^2$, and PNG compressed pages of size $128^2$, unless specified otherwise. Page blending (3.8) was disabled to maximize the amount of pages that was requested, since this would otherwise hinder pages from being requested until other pages had been fully blended in. Google Chrome was chosen as the testing

| | |
|---|---|
| Processor | AMD Opteron 6128 2GHz (2 processors) |
| CPU Memory | 32 GB |
| Graphics Card | NVIDIA GeForce GTX 560 Ti |
| GPU Memory | 1 GB |
| Screen Resolution | 1920 x 1080 |
| Web Browser | Chromium 17.0.914.0 (Developer Build 106280) |

**Table 4.1:** Benchmark Computer

browser since it supports every main feature needed in its current stable release*. The three different streaming methods evaluated are **Simple**, **Worker** and **Socket**. These are explained more in detail in Table 4.3.

Scene
59615 vertices, 87644 faces

Virtual Texture
32768 x 32768 pixels

**Figure 4.1:** Benchmarking scene and its Virtual Texture. Note that only a one third of the available texture space was used.

*Google Chrome 15.0.874.106

| Browser | WebGL 1.0 | Web Workers | Web Sockets | Derivatives Extension | Notes |
|---|---|---|---|---|---|
| Google Chrome 15.0 | ✓ | ✓ | ✓ | ✓ | Supports every feature needed in current stable release. |
| Mozilla Firefox 10* | ✓ | ✓ | ✓ | ✓ | Support for standard derivatives extension has been added to the nightly builds and will be available with the release of Firefox 10.† |
| Opera 12‡ | ✓ | ✓ | ✓ | | Opera's next big release will have WebGL support, while the current stable release does not support it. |
| Safari 5.1 | ✓ | ✓ | ✓ | | Latest version of Safari has WebGL support, but is disabled by default and has to be manually enabled via a menu option. |
| Internet Explorer 10§ | | ✓ | ✓ | | Microsoft researchers have labeled WebGL as a harmful technology in its current state, and are not planning on adding support for their web browser Internet Explorer.[Def11] |

**Table 4.2:** Browser Features

| | |
|---|---|
| **Simple** | Standard/simple page streaming uses the JavaScript calls to load the images dynamically, but the browsers internal functionality to decode the image formats. This is the most typically used method to load images dynamically in HTML/JavaScript. (See 3.5.1) |

*WebGL, Web Workers and Web Sockets are all supported in the stable release version 8, but the standard derivatives extension is only available in alpha releases of Firefox 10. `http://nightly.mozilla.org/`

†`https://bugzilla.mozilla.org/show_bug.cgi?id=684853`

‡Opera's beta/alpha builds are called Opera Next, `http://www.opera.com/browser/next/`

§Preview release 10, `http://ie.microsoft.com/testdrive/`

| | |
|---|---|
| **Worker** | Uses multiple Web Workers and AJAX* to dynamically load the images and then decompress then manually with JPEG/PNG decompression libraries written in JavaScript. The uncompressed image data is then sent back to the main thread were it can be loaded into GPU and texture objects directly. (See 3.5.2) |
| **Socket** | An alternative to the Web Workers implementation. Instead of AJAX calls to load the compressed images (which will create a new HTTP connection for each call), it uses a technique called Web Sockets to establish a persistent connection that is valid throughout whole page visit. (See 3.5.3) |

**Table 4.3:** Streaming Methods

## 4.1 Streaming Methods

The two most important aspects of the performance of a streaming method, is how fast it can handle pages from request to delivery and how much it impacts the FPS. Even if a specific streaming method is extremely fast but has a very low average FPS, the speed of streaming will not matter much if the simulation is lagging due to bad or unstable FPS.

The load time for a page is calculated as the time difference from when a page is initially requested by the page determination, to when it is fully uncompressed and ready to be uploaded into GPU memory. Figures 4.2 and 4.3 show the load times for all streaming methods using different page sizes, PNG and JPEG files respectively.

Looking at these figures, the simple streaming method is clearly the fastest method in all cases. Even between the different page sizes it is always the lowest and has a consistently low increase.

Both Socket and Worker methods have a significantly higher load time, while Socket in most cases have a slightly lower load time. Worker takes the longest time to process a request, and approximately twice as slow at loading pages. However, the average FPS must also be taken into account before ruling out any stream method.

When comparing the JPEG and PNG image formats, JPEG generally takes longer to load than PNG. This might seem odd since JPEG files are much smaller compared to PNG encoded files, smaller image size per page should result in a lower loading time. A closer comparison shows that the JPEG files is marginally faster to the PNG files, but only for Simple streaming. However, both Worker and Socket methods have much higher load times than Simple streaming, even compared to PNG files. One of the reasons that

*`http://www.w3schools.com/ajax/default.asp`

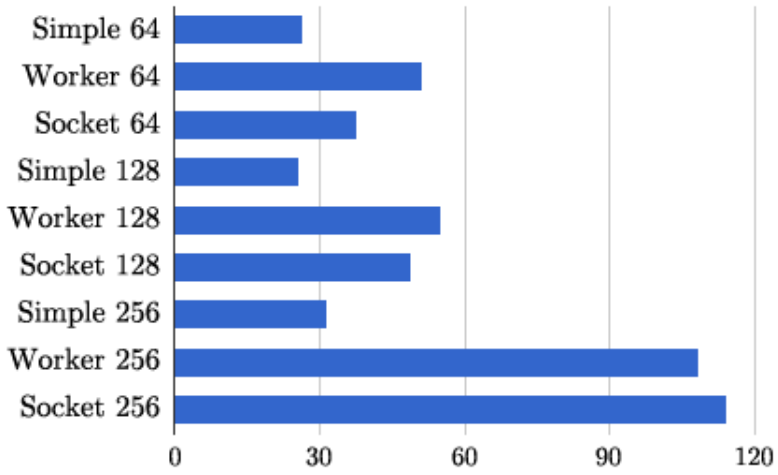



**Figure 4.2:** Average load times (ms) for PNG compressed images, comparing different streaming methods.

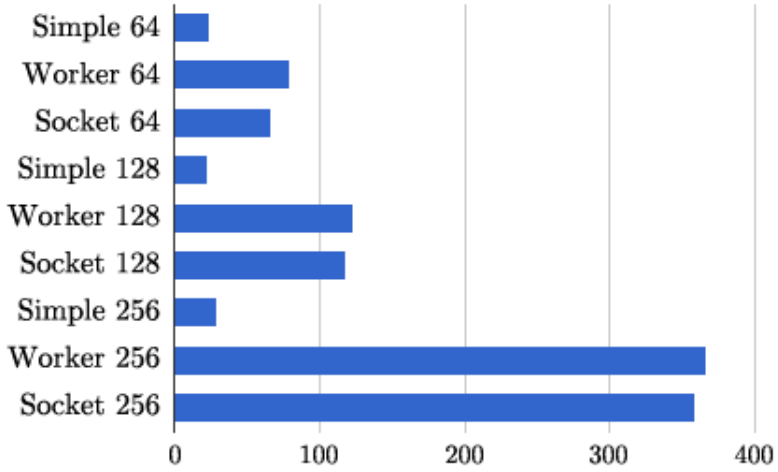



**Figure 4.3:** Average load times (ms) for JPEG compressed images, comparing different streaming methods.

could be the cause of longer streaming for Worker and Socket methods, is the fact that the image is decoded inside the Web Workers using a third party JavaScript library. This is also the case with PNG files, but with library developed by a different third party, which could be the reason why PNG files with Worker/Socket is faster. Switching to an alternative JPEG decoder could possibly speed things up.

Lastly, even with the highest loading times ($256^2$ JPEG pages with Socket streaming), the delay is still only 0.367 seconds for a page request to delivery, which depending

on the application could very well be acceptable.

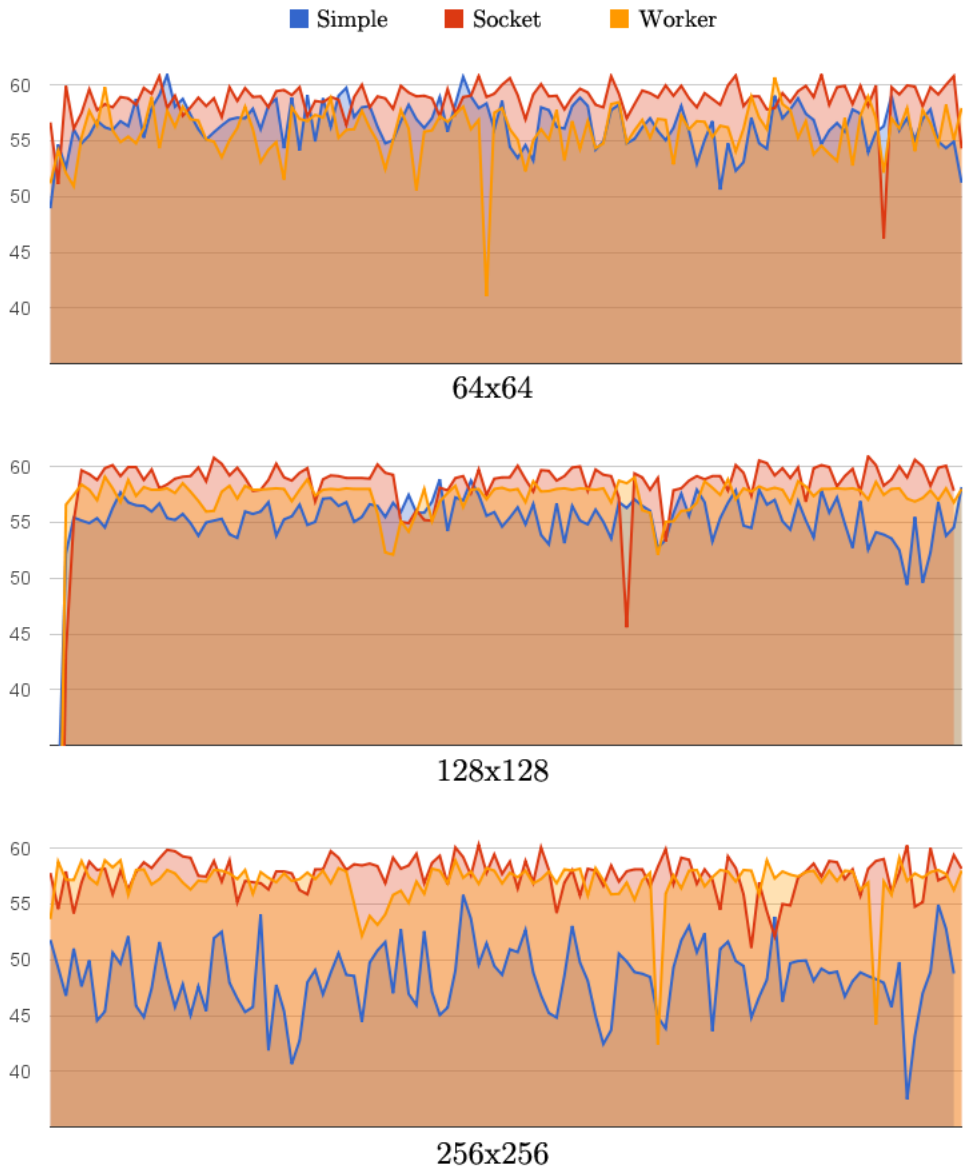



**Figure 4.4:** FPS comparison of different page streaming methods.

Looking at the FPS (Figure 4.4), there is a similar trend where Worker and Socket methods give comparable results. However, the Simple streaming method has a severe dip in FPS compared to the other two, most noticeably for the largest page sizes. One of the reasons for this could be that the page loading and decompression is done in the same thread as the main rendering loop is executed in. Adding to this, is also the fact that the Simple method loads each page faster than other methods and thus will try

to load many more pages per second, and could in turn choke the FPS since it has to decode and upload so many pages each second.

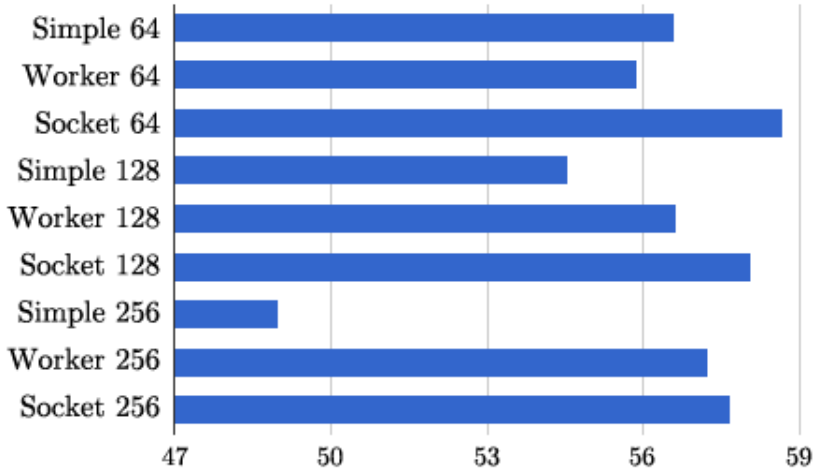



**Figure 4.5:** Average FPS count for different streaming methods and page sizes, of **JPEG** format.

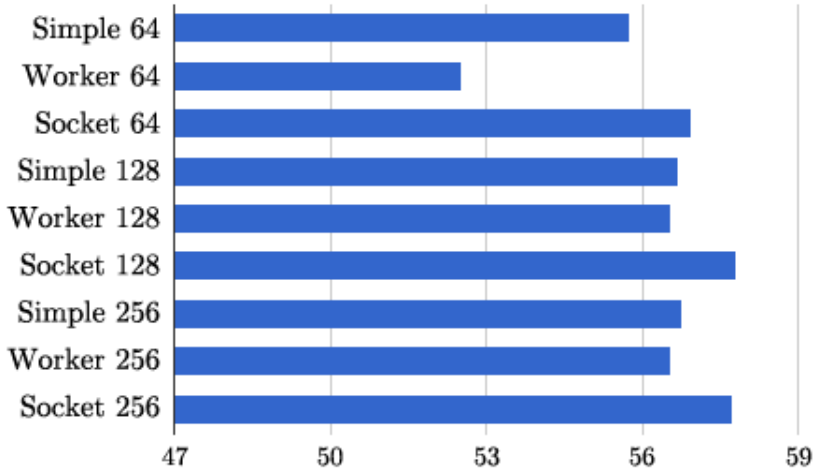



**Figure 4.6:** Average FPS count for different streaming methods and page sizes, of **PNG** format.

## 4.2 Page and Cache Sizes

When comparing page misses, using different page sizes, results show that large pages result in lower page misses. This is expected, since large pages cover a greater area

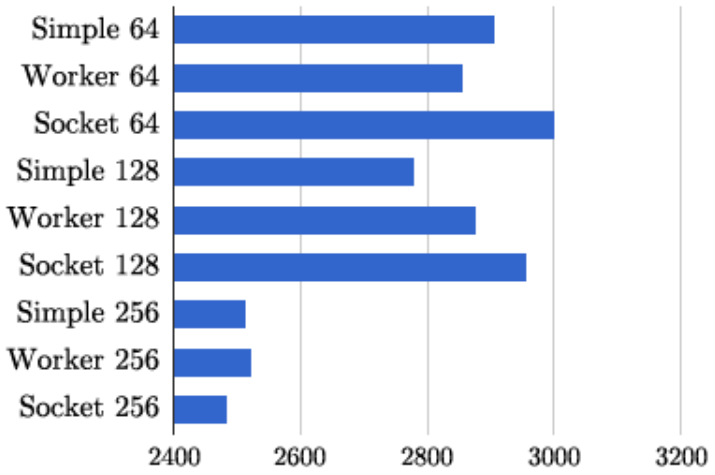



**Figure 4.7:** Average page misses for different page sizes and streaming methods.

compared to small pages. Comparably, to cover the same area of a $256^2$ page with $64^2$ pages, 16 pages are required.

However, depending on the application, streaming more pages to cover a large area might be a good thing. For example, if the scene has a fast moving camera, the required set of pages will change more often compared to a slow moving camera. A small page size would mean that the most crucial pages would have time to be streamed in, whereas bigger pages with longer request-to-arrival times might arrive to late to be of use. This could happen in the case that a user has moved the camera from the area where the requested page is no longer visible in the viewport. However, this behavior is highly dependent on the implementation and could sometimes be acceptable. For example, in the case of an FPS game, it is likely that the now redundant page could soon be referenced again.

Depending on how much memory and the maximum texture size the client has support for, maximizing the page cache size is generally the best approach in terms of page misses. Ideally, the page cache would be the same size as the whole Virtual Texture. However, depending on the scene and application, most of the frequently used pages could fit into the page cache, and eventually the number of page requests per second could approach zero if a large enough page cache is used. In the test scene (4.1), this can be shown in Figure 4.8 where the two biggest page caches show a decline in page misses over time.

Page and Virtual Texture sizes have the biggest impact on indirect table updates, since they directly correlate to the size of the indirect texture size. A larger indirect texture means that updates/changes higher up in the mipmap levels has to recursively update more levels and a larger data set compared to a smaller indirect texture. The

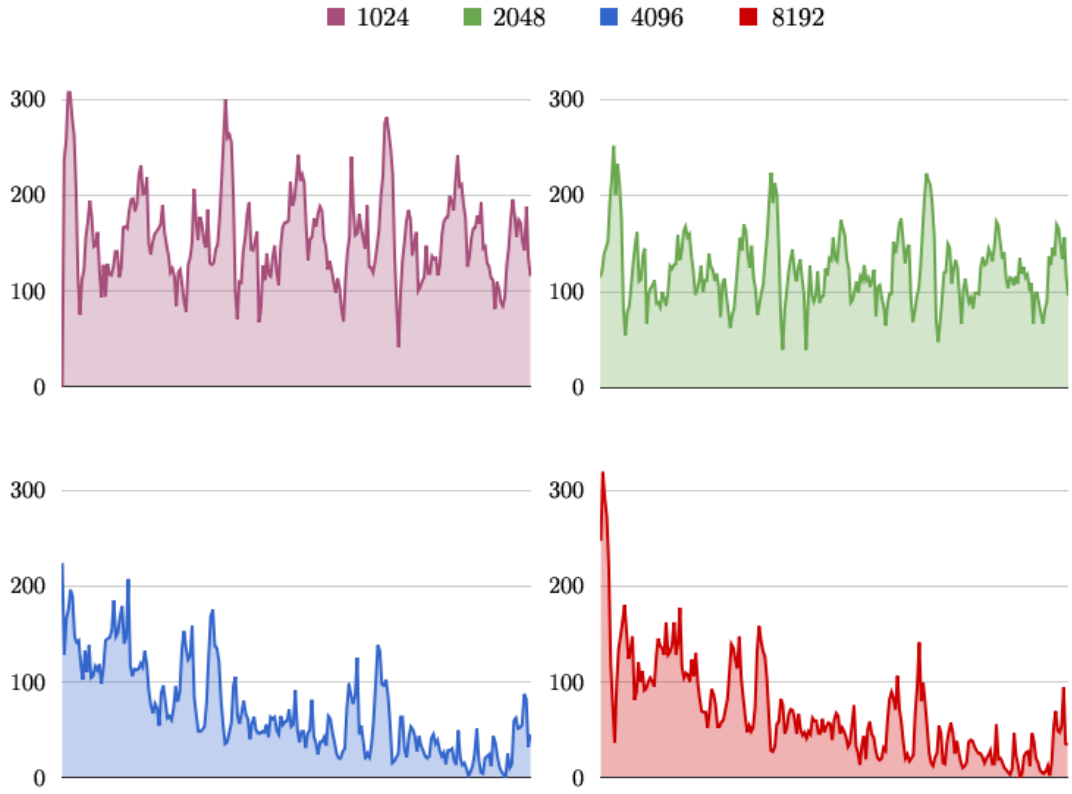



**Figure 4.8:** Missing pages (i.e. pages that need to be streamed in) over time, comparing different page cache sizes.

increase of data that needs to be sent to the GPU each update could possibly add to the total update time. Figure 4.9 shows a large difference in average update time for page sizes of 128 and 64. To keep a stable 60 FPS, each frame cannot take longer than 16.6 ms to complete. This means that an indirect texture update taking almost 12 ms will leave 4 ms for other essential parts to execute (i.e. page determination, page cache update and scene rendering).

## 4.3 Page Determination

Preferably the size of the page determination texture would be the same size as the viewport, giving a 1:1 correlation of pages needed to pixels visible. However, for higher viewport sizes, the amount of data needed to be downloaded from the GPU is too high to do in real time. This can be seen in Figure 4.10 where the largest tested size of $1024^2$ has the lowest amount of overwrites but provides read out times of almost 100ms. However,

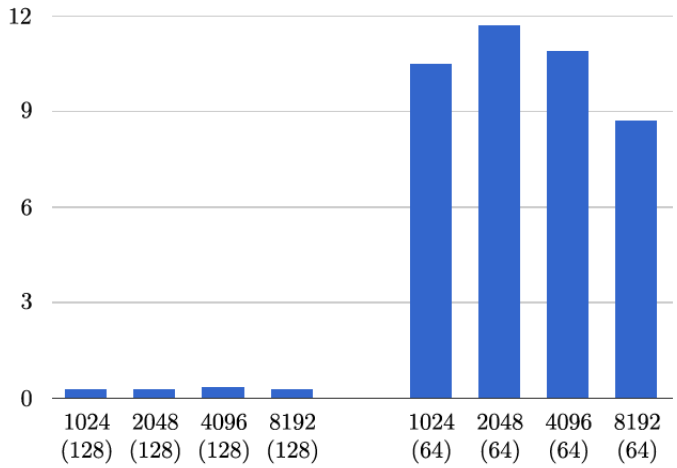



**Figure 4.9:** Average indirect table update time (ms) for different page cache and page sizes.

as mentioned in Section 3.4.1, a small page determination size still produces satisfiable results, down to a 1/8 of the viewport size. Looking at the lowest size in Figure 4.10, which is roughly a 1/8 of the viewport size in the tests, read out latency was measured at 17ms. This is still a bit too high to keep a steady FPS of 60, and could result in FPS drops and ultimately be distracting for the user.

## 4.4 Misc Optimizations

Instead of using JavaScript libraries to decode images during streaming with Web Workers, native image decoding can still be achieved by loading the image data as a base64 encoded string. Figure 4.11 compares two test runs, one with base64 encoded images streamed via Web Workers and decoded in the main thread, and the other with simple streaming (which performed worst in terms of FPS compared to Socket and normal Worker methods, see Figure 4.4). This shows that moving the loading to a Web Worker, while letting the browser handle the decoding can be worse than letting the browser take care of both natively. Base64 images take up 30% more space and causes message passing overhead in terms of data copy and serialization between the worker and main thread. These factor influences the stream latency negatively, and could be the reason why this approach gives poor results.

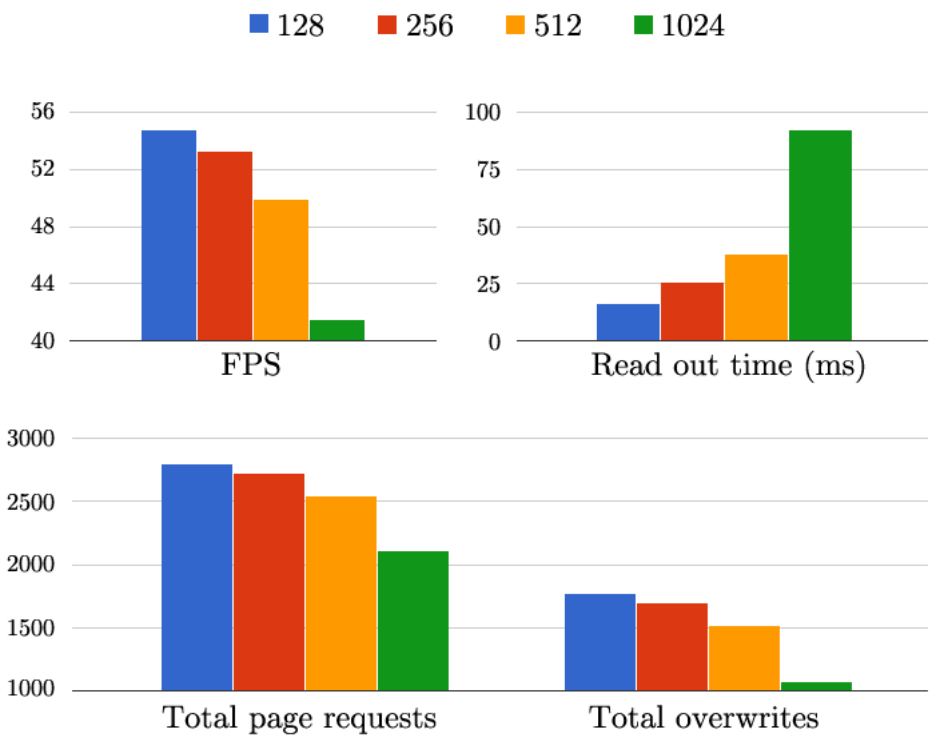



**Figure 4.10:** Comparing different page determination texture sizes.

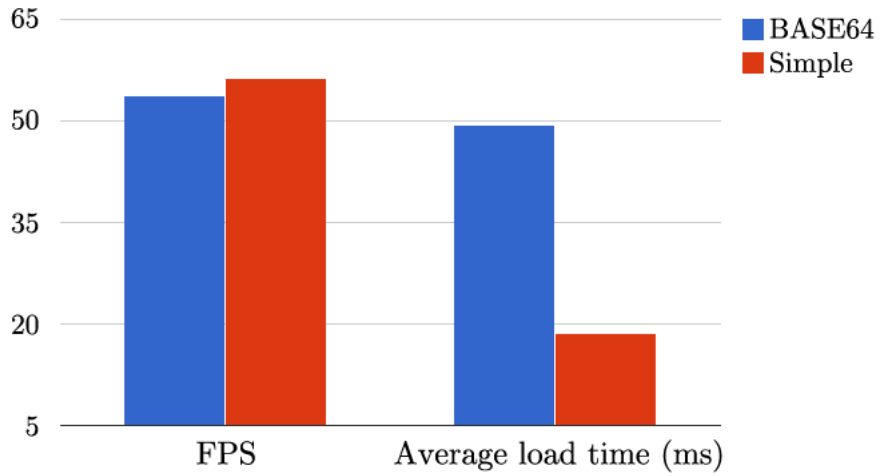



**Figure 4.11:** Comparing streaming of Base64 encoded PNG (via Web Workers) to the simple method streaming PNG compressed files.

# 5

# Discussion

## 5.1 Browsers

Virtual Texturing using WebGL is a viable alternative to texture mapping, and can be used in a wide range of applications. Further advancements in WebGL, Virtual Texturing and browser support might even decrease the number of required features for the technique to work, as well as boosting performance of page streaming and decompression. Furthermore, WebGL support for browsers is definitely maturing, considering that necessary features in the WebGL and HTML5 specifications have been continously added throughout development of this thesis. Almost all browsers have support for the concepts described in this paper in upcoming alpha and beta versions. The main reasons holding back a stable platform independent implementation of Virtual Textures in WebGL, is mostly related to the speed of JavaScript engines and specific WebGL related commands between browsers.

Google Chrome is the only browser that, in its latest public release, has support for every feature that is needed in each streaming method presented in this thesis even though certain WebGL specific functions such as `readPixels` and `texSubImage2D` have been shown to perform worse in Chrome compared to other browsers*. One specific problem encountered with using Firefox, is that Firefox performs premultiplied alpha when executing texture updates, even for textures where an alpha channel is present. As a result, ugly texture artifacts occur when rendering, and must be handled explicitly when developing for the Firefox browser. Fortunately, the artifacts produced with premultiplied alpha have been reported to the web development team at Mozilla†, which

*`http://jsperf.com/webgl-teximage2d-vs-texsubimage2d/7`

†`https://bugzilla.mozilla.org/show_bug.cgi?id=698169`

shows a good example of the current state of the web browser market where rapid development cycles and a good consumer relationship are key factors for successful integration of new web technology. The rest of the market is not far behind, with most future alpha and beta versions supporting all needed features. The only browser lagging behind is Internet Explorer, which lacks support for WebGL and will most likely not support it in the nearest future. Third-party workarounds exist, such as IEWebGL* and Chrome Frame† , that enables WebGL support for Internet Explorer. However, this defeates the purpose of WbeGL as a cross-platform plugin free technology, and is not considered as a target platform for future development of a Virtual Texturing pipeline.

## 5.2 Streaming Methods

### 5.2.1 Simple

Simple streaming is trivially the best supported streaming method since it utilizes the native image loading capabilities in JavaScript that has been around for a long time. This is also the method that has proven to give the fastest loading times (see Figure 4.2 and Figure 4.3). However, the disadvantages with simple streaming is that it gives the worst average FPS and supports only a limited amount of image formats.

### 5.2.2 Worker

A relatively new technique available in most modern browsers except Internet Explorer. The results of using the Web Worker streaming method show slower page loading speeds, partially due to relying on decompressing image data using JavaScript instead of the browser, but a much better average FPS than the simple method. Furthermore, Web Workers provide general purpose functionality for things other than just page loading. For example, the pixel processing in the page determination (Section 3.4.3) can be executed in a web worker, which unburdens the main thread.

### 5.2.3 Socket

Web Sockets is a new technique for data transfer in web pages, partially supported by most browsers. The Web Socket streaming method is shown to have the marginally best average FPS (Figure 4.4), and only a slightly better loading time (Figure 4.2) than the worker method. However, socket streaming have only been tested with one Web Socket server written in Python specifically for this thesis, faster page loading might be achieved with a more advanced and optimized server.

*`http://iewebgl.com/`

†`http://code.google.com/chrome/chromeframe/`

## 5.3 Image Formats and Page Sizes

The choice of page size is highly dependent on the application and user navigation. Scenes with rapid moving cameras could benefit of smaller page sizes, as discussed in Section 3.11.4. But as seen in Figure 4.9, smaller page sizes can lead to lower performance when updating the indirect table. This is something that might change in the future with better performance on specific WebGL functions.

When considering image formats, the choice depends on wether the application needs lossless compression with a higher bandwidth demand but with a better visual quality, or lossy compression with a lower bandwidth demand for source images but a lower visual quality. The decompression times of JPEG and PNG seem equal when using the internal decompressions the browser supplies, but differ greatly when using the JavaScript implementations of the image decompressors. For the sake of performance, it would be most beneficial to access native browser decompression functions directly from a Worker thread instead of unpacking them explicitly in JavaScript routines. An alternative solution would be if WebGL could accept compressed images directly without the need of `Image` objects.

# 6

# Future Work

## 6.1 WebCL

[May10] presents a different way of performing the page determination in order to increase performance of this pass. Instead of generating a list of needed pages on the CPU side in the *Process Pixel Data* 3.4.3 step, the list generation is executed directly in the GPU. This is accomplished by utilizing relatively new features that makes it possible to perform more general calculations on the GPU via the OpenCL API. Currently, no official version of a web-based equivalent has been released, but the Khronos Group confirmed in March 2011 that a API definition of the WebCL* interface is under development. However, two prototype implementations, developed by Nokia† and Samsung‡ respectively, are already available as browser extensions, but due to the non-matureness of their API, they were not evaluated in this paper. Implementation WebCL-based list conversion will be looked at in the future when the upcoming official specification has been released.

## 6.2 Texture Compression

Very few compressed image formats were investigated in this paper. Bandwidth usage per requested page is a critical performance issue, both internally from CPU to GPU as well as over the network connection. The current streaming methods in the Virtual Texture pipeline only support the two most common image formats JPEG and PNG, but

*http://www.khronos.org/webcl/
†http://webcl.nokiaresearch.com/
‡http://code.google.com/p/webcl/

very well be extended and tested with other techniques with better compression ratios or decompression performance.

## 6.3 Object Multi-Texturing

In the last stages of development, several test scenes included objects that referenced multiple textures used in rendering that all map to the same UV coordinates, such as normal, specular and environment maps. In the current implementation, these special maps cannot be trivially be merged and stored into virtual texture pages, and requires either more intelligent page design or separate virtual textures and page caches.

An additional problem when using several textures per object, is that the current implementation of page determination can only handle one virtual texture coordinate per read out pixel. A proposed approach dealing with this limitation, was to alternate the needed texture coordinates in two or more different page determination passes. The page determination is currently performed once every 1/16th frame. A secondary page determination pass could be executed in between these passes, which calculates the coordinates for special maps. However, this could result in a delay for pages that are considered more important, such as normal map pages for example. Another approach to this problem would be to alternate the page determination pixels, such that every even pixel outputs the standard texture coords while the odd pixels output the environment map texture coords.

# Bibliography


[Bar08] Sean Barrett. Sparse virtual textures. http://www.silverspaceship.com/src/svt/, 2008.

[Blo00] Charles Bloom. Terrain texture compositing by blending in the framebuffer (aka "splatting" textures). http://www.cbloom.com/3d/techdocs/splatting.txt, nov 2000.

[Cab10] Ricardo Cabello. Three.js. https://github.com/mrdoob/three.js/, 2010.

[Def11] Microsoft Security Research & Defense. Webgl considered harmful. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx, 2011.

[Fen04] Wei-Wen Feng. Notes on mesh parametrization. http://mgarland.org/class/geom04/material/param-notes.pdf, 2004.

[Fer04] Randima Fernando. Gpu gems: Programming techniques, tips and tricks for real-time graphics, 2004.

[Gro11a] IETF HyBi Working Group. The websocket protocol draft-ietf-hybi-thewebsocketprotocol-17. http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-00, 2011.

[Gro11b] The Khronos Group. Webgl specification. http://www.khronos.org/registry/webgl/specs/latest/, 2011.

[Has07] Al Hastings. Presentation: Texture streaming - everything you care to know and more. http://www.insomniacgames.com/tech/articles/1107/files/texture_streaming.pdf, 2007. Insomniac Games.

[HLS07] Kai Hormann, Bruno Lévy, and Alla Sheffer. Mesh parameterization: Theory and practice, 2007.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings*

*of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 55–63, New York, NY, USA, 2010. ACM.

[May10] Albert Julian Mayer. Virtual texturing. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, October 2010.

[MG08] Martin Mittring and Crytek GmbH. Advanced virtual texture topics. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 23–51, New York, NY, USA, 2008. ACM.

[Neu10] Andreas Neu. Virtual texturing, May 2010.

[NVI04] NVIDIA. Improve batching using texture atlases. `http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/BatchingViaTextureAtlases/AtlasCreationTool/Docs/Batching_Via_Texture_Atlases.pdf`, 2004.

[SGG11] Abraham Silberschatz, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*. Wiley, 2011.

[Shr08] Ryan Shrout. John carmack on id tech 6, ray tracing, consoles, physics and more. `http://www.pcper.com/reviews/Graphics-Cards/John-Carmack-id-Tech-6-Ray-Tracing-Consoles-Physics-and-more?aid=532`, mar 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*, SIGGRAPH '98, pages 151–158, New York, NY, USA, 1998. ACM.

[vW06] J.M.P. van Waveren. Real-time texture streaming & decompression. `http://software.intel.com/file/17248/`, 2006. id Software.

[vW09] J.M.P. van Waveren. id tech 5 challenges - from texture virtualization to massive parallelization. `http://s09.idav.ucdavis.edu/talks/05JP_id_Tech_5_Challenges.pdf`, 2009. id Software.

[Wil83] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17:1–11, July 1983.