CHALMERS

Fast Text Rendering on Embedded GPUs

Master of Science Thesis in Interaction Design and Technologies

JONATHAN GUSTAFSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, February 2013

Fast Text Rendering on Embedded GPUs

JONATHAN GUSTAFSSON

Examiner: ULF ASSARSSON

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 10 00

[Cover: A device where text is rendered by the GPU]

Department of Computer Science and Engineering
Göteborg, Sweden February 2013

Abstract

Modern mid to high-end mobile phones and tablets all have GPU-rendering capabilities that are more energy efficient and potentially faster at rendering text than the CPU. On desktop computers, GPU acceleration is common in PDF readers and browsers. This is due the great rendering capabilities of the desktop GPU, already proven in games and CAD. The rendering capabilities of device GPUs are far less proven.

This thesis presents a survey containing a number of different devices that were benchmarked iteratively. During each iteration, the bitmap rendering algorithm was tweaked and tuned as new information was discovered.

As shown in the results section of this thesis, modern devices using the final iteration of the rendering algorithm are indeed ready for a move to GPU rendering. It also shows that the future for this solution is bright, as rendering performance on devices will improve according to Moore's law.

Acknowledgements

The work done in support of this thesis was performed at Opera Software in their Göteborg office during the summer of 2012. I would like to thank every one there for a great stay and for giving me feedback whenever I needed it. I especially want to thank Marcus Geelnard for supervising me, giving me feedback and direction during the whole summer. Without him, this thesis would have never come to fruition. Additionally, Ulf Assarsson served as my examiner during this thesis, giving great feedback on the structure and language of the report. Finally, would like to thank Robin Ytterlid, for proofreading and giving feedback on this thesis.

Contents

1

Introduction

The typical utilization of the graphics processing unit (GPU) has been limited to applications that contain computationally heavy features, *e.g.* a high amount of matrix multiplications, shading, and collision detection. Consumer devices, *e.g.* mobile phones, tablets, and TVs, have been showing a clear trend in providing fairly advanced GPUs with support for 3D acceleration. The hardware can also be used for accelerating 2D graphics, instead of doing conventional software rendering on the central processing unit (CPU). There are a number of different issues on devices that are challenging for a developer compared to a typical modern desktop computer. Some of these issues are zooming, differences in screen size, pixel densities, both GPU and CPU processing capabilities, and both vertical and horizontal translation. It is true that there is a high amount of similar challenges when it comes to text rendering on a desktop system as well. However, the amount of processing power on a modern desktop computer will have no problem rendering a screen full of text by using the GPU, which means that the performance related issues are negligible on that platform.

There are a number of software techniques that could be used to accelerate 2D rendering on devices and this thesis presents a study that was conducted to explore some of these techniques.

While software techniques are important factors in developing a fast text renderer, it is equally important to consider the devices that are used to run them. During development of software that aims to be compatible with a high number of different devices, the developers need to consider the fragmentation problem. The different qualities of the different devices can result in cases where the application is simply is not presentable, which is the fragmentation problem. For example, Apple's iOS is used by quite few devices compared to Android, which has a larger flora of devices [1], which means that the problem is smaller for iOS developers. This survey has used a small but representative collection of devices during the benchmark suites in an effort to minimize the potential effects of the fragmentation problem.

1.1 Purpose

The main goal of this thesis is to provide developers with a survey that can provide assistance when deciding if and how GPU-accelerated text rendering can be beneficial to the developers' projects.

1.2 Limitations

The fragmentation of Android devices, that was mentioned in Section 1, causes difficulty when producing a survey as it is impractical and very time consuming to test every unique device. The amount of devices was therefore limited, to allow the thesis to focus more on software techniques, rather than having to test a higher amount of devices.

Curves can be used as an approach to implement text rendering and GPU-accelerated techniques have been presented by Loop and Blinn [2], discussed in 2.2.4. However, curve rendering is outside the scope of the survey, as it requires a high amount of vertices, discussed in Section 2.2.1. It also has a higher amount of computations that has to be done for each glyph. During the initial literature study that preceded the implementation phase, the number of vertices was suspected to be the likely bottleneck when it came to GPU-based rendering on devices. It was this suspicion that excluded curve rendering from this thesis as there were other issues that were prioritized higher.

Nvidia has additionally released their new GPU-based path rendering technique [3], which could have been included. However, there is currently no support for Android or iOS and this made it impossible to investigate this technique any further, in this thesis.

1.3 Method

The process of the work done in support of this thesis required the completion of a number of steps:

- Investigation of prior work in the field, *e.g.* glyph caches, texture atlases, and GPU-assisted curve rendering.
- Evaluation of the known techniques and a conclusion concerning what techniques lend themselves to hardware acceleration.
- Implementation of a benchmark tool that can assess the relevant capabilities of actual hardware, *i.e.* Mobile phones that run Android, to determine feasibility and limits of various techniques. The benchmarking tool has to support various data sets, including Unicode [4] characters.
- Design and implement several rendering algorithms and testing them using the benchmark tool.

1.4 Programming Language and Tools

Android mainly supports applications written in Java, but not exclusively. Android's Native development kit (NDK) [5] allows developers to implement parts of an application using native-code languages, *e.g.* C and C++, and the benchmark tool consisted mainly of C++ code. As performance was key for the benchmark tool, C++ and use of the NDK was chosen over Java. The choice of C++ allowed the development to be carried out on a personal computer running a Linux distribution, where testing and debugging was easier than on a devices. Since Android only supports the OpenGL ES subset of the OpenGL application programming interface (API), OpenGL ES 2.0 was used by the benchmark tool. In Section 2.2, the use of OpenGL ES 2.0 over OpenGL ES 1.1 is motivated further.

FreeType [6] is an open source font engine that was used to produce bitmap images of glyphs. The use of FreeType was made with purely practical motivations, since it supports Unicode, TrueType fonts, it has liberal license options [7] and it has been used for several commercial and open source products, *e.g.* Nintendo Wii's Internet browser [8] and Blender [9].

2

Previous Work

Removing bottlenecks in GPU-based rendering is not a new field. Thus, there are a number of techniques that have been developed to counter the issues that these bottlenecks can cause. In this section, a number of these techniques will be presented together with a clarification concerning the different advantages and drawbacks when using CPU-based or GPU-based rendering.

From a text perspective, there are a number of things that are specific to this area of real time rendering. A glyph is not only a character, but also a combination of features, stated in Table 1. This means that the goal, presented in Section 1.3, of supporting the whole Unicode character set is much more complex when all features are taken into account.

2.1 CPU-Based Rendering

Any rendering that does not utilize the GPU has to perform the calculations on the CPU instead. One example where this approach is preferable is when the developer of an application desires to be independent from whatever GPU, if any, is in the system. However, there are real-time rendering situations where the CPU simply is not powerful enough, *e.g.* games in 3D, simulation engines, and computer aided design (CAD). During the recent years, screen resolution on devices have generally grown quite steadily and CPU performance has not grown at a rate that can match the increased

Feature name	Data type
Font	Pointer to FreeType FT_FACE object
RGB color values	One char per color value
Font size	Int
Italic	Bool
Bold	Bool
Underlined	Bool

Table 1: This table of glyph features that were supported in the benchmarking tool that was developed and used to test the different techniques presented in Section 3.

burden on CPU-based rendering that the higher screen resolutions amount to.

Compared to desktop computers, devices have a much higher demand on power efficiency. This is mainly due the limitations of battery capacities [10]. The CPU is additionally much less energy efficient than the GPU [11]. With these points in mind, it would be beneficial to move CPU-based rendering on devices to the GPU. The move would be a advantage both from an energy and user point of view, since it could decrease the frequency of the need to recharge the device.

2.1.1 Font Rasterization for Mobile Devices

In the master thesis by Andersson [12], CPU-based curve rendering on devices was examined. While it was shown that the plane sweep algorithm implemented by Andersson was indeed faster than the FreeType equivalent, it was also shown that the triangle based algorithm did not perform well on the CPU. The triangle based algorithm was influenced by the paper by Loop and Blinn [2], which is briefly discussed in Section 2.2.4. The thesis reflects a focus on memory efficiency that was important at the time that it was written, but is much less critical now. Andersson stated that the future OpenGL ES standard could be used for future research to determine whether or not the triangle based algorithm would be feasible on that platform.

2.2 GPU-Based Rendering

There are several potential motivations as to why a developer should choose to use the GPU for a specific application, as GPUs have come a long way from their gaming roots. Ever since 2006 when Nvidia's CUDA [13] was released, developers have had the option to utilize the GPU in a similar fashion as the CPU, called general-purpose computing on graphics processing units (GPGPU). However, due to the criteria that an application has to be able to be vectorized, *i.e.* that the data can be processed with single instruction, multiple data (SIMD) instructions, GPGPU is hardly a drop-in replacement for the CPU. CUDA is Nvidia exclusive technology and while it is free of charge to use, it requires the developers and users to run the application on Nvidia hardware.

Since 2008 there has been an open standard called OpenCL [14] that is both royalty-free and cross platform, which allows users to use products from any GPU developer that supports OpenCL, *e.g.* ATI, ARM, and Intel [15], and not exclusively Nvidia. Apart from GPGPU applications, the GPU can also be used for other applications, *e.g.* video decoding.

According to Android's developer website [16], support for GPU-based rendering on devices that run Android is very good. In their statistics, based on device visits to Google Play, 90.8% support OpenGL ES 2.0 and

OpenGL 1.1, while 9.2% only support OpenGL ES 1.1. As of Android 4.0 [17], GPU-acceleration on all windows have been enabled by default, for all applications that target Android 4.0 or above. This indicates that the developers of Android see the benefits of GPU-based rendering on devices that run their OS.

2.2.1 Text Objects

The most basic object in text rendering is a glyph object that has a bitmap texture and a quad. This quad can be drawn with either six vertices or in the case of a triangle strip, four vertices. One of the consequences of the use of glyph objects is a high number of vertices, which in turn leads to increased usage of the graphics bus. Another consequence is the high number of draw calls when rendering a typical page of text. In addition, a texture upload for each glyph, each frame, is also a taxing operation on the graphics bus. However, the vertex count and amount of draw calls can be lowered by selecting a more complex text object, *e.g.* words or paragraphs. Using more complex text objects can also decrease the amount of texture uploads if larger segments, than glyphs, of the word or paragraph can be submitted at each upload. Creating these larger segments is more computationally expensive and taxes the CPU more than using glyphs.

2.2.2 Texture Atlases

There are more common bottlenecks that can occur while a developer is creating a real-time rendering application that contains many objects with different characteristics, aside from the ones mentioned in Section 2.2.1. One of these additional bottlenecks is a high amount of draw calls [18]. In the case where each object requires a different texture than the previous object, each object entails a state change which requires a draw call before the new texture can be bound. The amount of draw calls can be decreased by batching objects that share the same rendering state, allowing them to be rendered by the same draw call. The success of the batching strategy depends solely on the amount of objects that can be batched together.

Another approach to reduce draw calls is to store all textures in one or several texture atlases, which allows more efficient batching as many objects now potentially share the same rendering state. An inherent flaw with texture atlases is a phenomenon called bleeding, which occurs during bilinear sampling near the edges of subtextures in the atlas, as shown in Figure 1. To solve the bleeding problem, a 1 texel border around each texture can be added to ensure that no neighboring texture is sampled.

Pre-created Atlases In a text rendering situation where the whole Unicode character set is supported, it is not feasible to create textures for each and every character in the Unicode standard. Nor would it be possible

Figure 1: Illustration of the texture bleeding phenomenon, which is the result of bilinear sampling at the boarder between two neighboring subtextures in a texture atlas. This sampling results in the edge of the glyph having been affected by the neighboring subtextures.

to create enough texture atlases to contain them all since there are slightly over 100,000 characters [19] to support. In addition, support for different font features, *e.g.* sizes, colors, and italics, makes the pre-creation strategy even less appealing. However, in a case where the number of glyphs that will be used is well known and limited within acceptable bounds, pre-creation is a viable option that should be considered due to its simplicity. As mentioned in Section 2.2.1, all text objects can potentially be rendered in one draw call if the texture atlas contains all of them.

Completely forgoing any preparation or data structures and simply creating a new texture for each glyph, each frame, is a naïve approach that was discussed in Section 2.2.1. One approach that can be used to allow dynamic support for any part of the Unicode character set is caching, which can be implemented with texture atlases.

Caching with Atlases Caching is an old computer science topic [20] and has been well established over the years. In a situation where there is a limited space to store data in an efficient manner, a rule must be set in place to manage the additions and removal of data in that limited space. The Least Recently Used (LRU) algorithm is one of the most common caching algorithms and it takes advantage of temporal locality, *i.e.* data used recently will probably be used again in the near future. However, LRU is not optimal when there is a number of unique elements that are used cyclically, *e.g.* once per frame. In that case, the caching algorithm will replace elements that have not been used for a long time, which are exactly the elements in the cache that are most likely to be used in the near future. It would be much better to remove elements that have been used recently, as they will be used next time the cycle has completed one iteration. This caching algorithm is called Most Recently Used (MRU) [21]. As written text, most often, does not fulfill the uniqueness criteria of the cyclical example, the LRU is a simple and fitting caching algorithm for text. Several more complex caching

algorithms have been developed and Dumont *et al.* [22] has presented a texture caching algorithm which takes camera perception into account to decide which Mipmap [23] level should be used.

A texture cache is achieved by filling a texture atlas and by replacing existing objects with new objects according to a caching algorithm, when the texture is full [24]. Caching with atlases requires a more complex data structure, generally requiring more CPU computations and higher memory usage compared to the use of the naïve approach or the pre-created texture atlas, discussed in the two previous sub sections. However, it is able to support the use of Unicode and it has the potential to be able to render all text objects in one draw call.

In a texture cache implementation, discussed in Section 2.2.2, the bleeding phenomenon affects the visual appearance in a more drastic way than merely having the edges of a texture slightly polluted by the neighboring subtexture in the texture atlas. Each neighboring subtexture of a glyph's texture can be replaced in a texture cache, which results in a new bleeding effect that pollutes the glyph's texture. A new bleeding effect can potentially be created each frame, resulting in a flicking effect.

2.2.3 Signed Distance Fields

Green, has presented a paper [25] on signed distance fields that can be used to create glyphs. Compared to using bitmaps, Green's solution has two important advantages: zooming can be performed without aliasing caused by low resolution, the performance is equal or close to regular texture mapping, and there is also lower memory consumption for the storage of the glyphs. Green also introduces a number of effects that can be achieved with this approach through the use of programmable shading, *e.g.* outlining, soft edges, and drop shadows.

This technique requires a conversion of bitmap textures into a signed distance field representation in a preprocessing step. The preprocessing is done in the following steps:

- Supply a high resolution binary texture where each texel is classified as either in or out.
- For each output texel, determine if the texel is in or out and compute the 2D distance to the nearest texel of the opposite state by searching the local neighborhood.

According to Green, the execution time of the second step is negligible due to the limited distance range which may be stored in an 8-bit alpha channel. However, taking the limited computing powers of a general device into account, the execution time for the whole algorithms might prove to be significant. In the first step, the algorithm requires a binary texture

that contains boundary information of the glyph. However, Green does not specify how this texture is generated. As this thesis requires support for the whole Unicode character set, it is imperative that this step can be performed without any manual intervention and in a timely fashion.

2.2.4 Curves

In the paper [2] by Loop and Blinn a GPU-accelerated curve rendering algorithm is presented. By using a pixel shader, a specific pixel is decided to be inside or outside of a Bézier curve and is shaded accordingly. This Bézier curve is described by a set of texture coordinates that are attached to the vertices of quadratic and cubic curve control points. The paper has a whole subsection dedicated to font rendering, which utilized Delaunay triangulation to go from curves to triangles.

While the method in this paper utilizes the GPU for rendering it does require CPU preprocessing of the curve data. The authors note they expect reduced performance when rendering dynamic geometry, due to the extensive CPU involvement. However, Kokojima *et al.* [26] has improved the method by using a stencil buffer and transparency multisampling and by removing the Delaunay triangulation and subdivision of overlapping triangles, which were the main preprocessing steps on the CPU. The improvement yielded a ten time performance increase.

3

Implementations

During the course of the study, a number of techniques were developed, tested and enhanced, in an iterative fashion. The coming subsections will each contain a motivation for the design choices that were made and a description of the methods while their results from the benchmark tests are presented in Section 4.

3.1 Base

In order to create a base line for unoptimized rendering performance but otherwise functional text renderer and a foundation that could be easily improved upon, an approach similar to what is described in Section 2.2.1 was developed. This naïve approach was simply called *Base* and consisted of three main steps.

The first step of Base was to parse the data set, containing information about the text that should be rendered, *e.g.* position, color, font, and the string of text, and storing the information in memory. The data set used in this thesis consisted of text where a group of few words were guaranteed to have the same glyph features, but where these groups had no real coherence amongst themselves. This leads to a situation where the Base approach could have no demands on the ordering of the data, requiring it to be more flexible.

In the second step, FreeType is initialized together with a number of FreeType objects, called faces. These faces enable the creation of glyph bitmaps and they were created by looping through each group, creating a face if that group's combination of glyph features had not yet have a face that supported them. Lastly, the rendering step is entered and all groups are rendered, except the groups that are culled away due to view frustum culling. As mentioned in Section 2.2.1, the rendering step itself constitutes of a texture upload and draw call per glyph.

The data included in each draw call is the position of the vertices, the texture coordinates, and the red, green, blue (RGB) values of the text. Base, compared to the other implementations that are presented in this thesis, has one extra benefit: it can use triangle strips [27]. However, this implementation suffers from the small batch problem [28] which states the importance

of reducing the number of draw calls due to the overhead associated with each draw call.

3.2 Glyph Cache

As the Base implementation was intentionally unoptimized, the next step in the iterative design process was to lower the amount of draw calls to an amount that was acceptable. The aim was inspired by Nvidia's paper on texture atlases [18] which explains the importance of batching draw calls in order to increase frame rate. Building on Base, the *Glyph cache* implementation actually consisted of a number of caches, one for each FreeType face. These caches consisted of information about all the glyphs that are currently residing in the cache, *e.g.* texture coordinates on the atlas and screen space coordinates, and a texture atlas where the glyphs' bitmap was stored.

The rendering step has a few additional steps, compared to Base. One of these steps is cache swapping, which occurs whenever a text group that is to be rendered does not have the same face as the previous group, requiring another texture to be bound. Before changing cache, a draw call is passed, rendering all previously unrendered glyphs.

Before a glyph is rendered, it is checked against the current cache and added to it, if it is not yet included. An addition of a glyph into a cache includes a texture upload. The LRU algorithm is used when the cache is full and glyphs have to be removed to make room for new additions, illustrated in Algorithm 1. As shown in Figure 2, each glyph was given the same amount of space on the texture, making bitmap substitutions easier as a new bitmap would always be able to replace an old bitmap. This partitioning of the texture space simplified the implementation of the caching algorithm while consuming more texture space. However, this simplification scales well with this implementation as the bitmaps that occupy more space on the texture is usually fewer than the amount of bitmaps that occupy less space. For example, there are fewer but larger glyphs in headlines while the body matter of a text has a higher amount but smaller glyphs. Unlike Base however, the glyphs are not rendered one by one but rather put in an array which is flushed during a texture swap, lowering the amount of draw calls dramatically.

Data: glyph G
if *cache is full* **then**
| replace least recently used glyph with G
else
| place glyph in Cache
end
Increment the cache's age counter
return *reference to G's new position*

Algorithm 1: The glyph insertion algorithm of Glyph cache

Figure 2: Illustration of Glyph cache texture space partitioning

With the Glyph cache implementation, sorting is an option to minimize the number of cache swaps, and therefore also the number of draw calls. However, this requires a pre-processing step which, as stated in the previous subsection, is outside of the scope of the design of these algorithms.

3.3 Global Cache

As described in Section 4, the draw call minimization of the Glyph cache implementation was not sufficient and further steps had to be taken to minimize the amount of draw calls. The strategy in this implementation was to push the previous texture atlas scheme to its extreme and implement a cache that that contains all glyphs, regardless of their glyph features. The *Global cache* strategy would, however, demand that previous texture space partitioning scheme in Glyph cache, would no longer be feasible. The reason for this is that typical text has few large glyphs and a high amount of smaller glyphs and the resulting partitioning would have resulted in a high amount of wasted texture space, as shown in Figure 3.

Building on the Glyph cache, this technique has a few key differences. As mentioned before, there is only one cache and thus one texture, which supports the possibility to render all glyphs in one draw call. This results in no cache swapping. However, the insertion of new glyphs and the replacement of existing glyphs is also far more complex. Pseudo code that illustrates the algorithm that completes these replacement actions are illustrated in Algorithm 2.

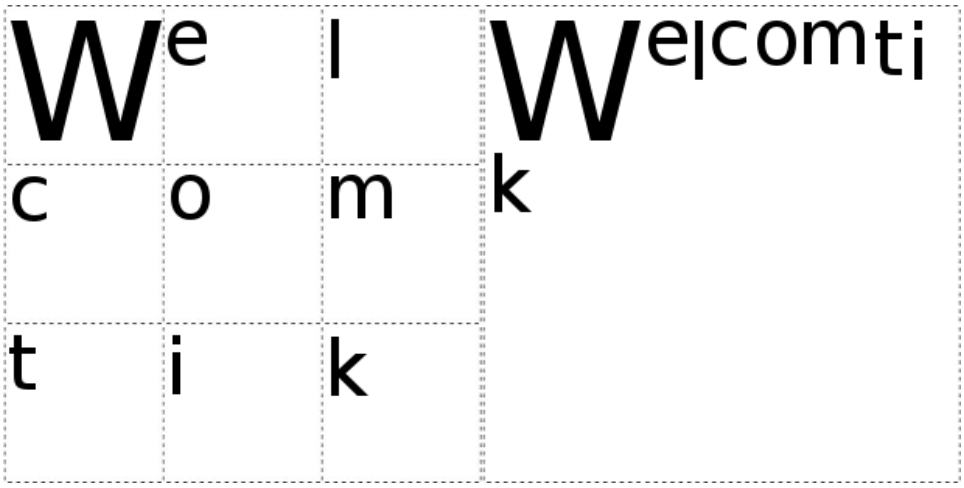

Figure 3: A comparison of the Glyph cache and Global cache texture partitioning schemes where one glyph is large and the rest are small. To the left: *Glyph cache's* solution where there is a high amount of wasted space. To the right: Global cache's solution where the texture space is used more efficiently.

```
Data: glyph G
forall the rows in Cache do
    if G's height <row height then
        if G fits on row then
            add G to row
            break
        end
    end
end
//G cannot fit on any row
if G can replace any glyph then
    replace least recently used glyph that is smaller than G
else if A new row cannot fit in the cache then
    remove the row of the least recently used element
    while G does not fit in the area do
        remove rows until G can fit in the new area
    end
    add new row and G to it
else
    add new row and G to it
end
increment the cache's age counter
return reference to G's new position
```

Algorithm 2: The glyph insertion algorithm of Global cache

Like the Glyph cache technique, this technique also checks if a glyph that is going to be rendered is in the cache and adds it if it is not. The drawback with this solution is that the search is more expensive, compared to Glyph cache as all glyphs are contained in the same glyph structure making the search space larger. This drawback was mitigated by the addition of a search tree data structure that lowered the time complexity of the previous linear search $\mathcal{O}(\mathrm{n})$ to a logarithmic search $\mathcal{O}(\log(\mathrm{n}))$.

4

Results

As mentioned in Section 1.1, the benchmark testing was performed on a number of devices, device specifications in Appendix A, that were chosen to be representative for the flora of devices that use the Android OS. This section will present the results of the benchmarks, for each different technique, where performance is measured in frames per second (FPS). However, before presenting the actual results, each data set will be described and motivated. The results from the Base technique are omitted due to, orders of magnitude, worse performance on all devices.

- **Web Audio Specification** [29]: This data set was chosen for its glyphs with few size or font variations, making it ideal to improve by caching.
- **BBC homepage** [30]: The BBC data contains a high number of size and font variations, increasing draw calls.
- **Chinese Wikipedia** [31]: As Unicode support was important, a Chinese data set was included. This data set has a lower amount of glyphs than the two previous ones and a lower amount of size and font variations than the BBC data set.
- **Japanese Wikipedia** [32]: To include the Japanese writing system, consisting of both Kanji and Kana characters, the Japanese Wikipedia homepage was included.
- **English Wikipedia** [33]: This data set was the first data set that was included during the development process and was also included in the final version of the benchmark tool. This data set has the highest amount of glyphs and draw call generating glyph variations of all the data sets.

4.1 Benchmarks

Each of the test data sets were created to replicate the text on a website. The text was rendered by scaling it in a manner that ensures that the text would fit the width of the screen. Due to the devices not being able to show

Input	Draw calls	Number of vertices
Web Audio Specification	7	5500
BBC homepage	36	7500
Chinese Wikipedia	28	4500
Japanese Wikipedia	24	3700
English Wikipedia	53	10200

Table 2: This table of Glyph cache results shows the amount of draw calls and the average number of vertices per frame, rounded to the nearest hundred, rendered in each.

all content at the same time, the benchmark consisted of the content being translated upwards, initially, with the direction being reversed every time there was no more content to show. This manner of testing was chosen to test the highest load of typical user interaction, as a zoomed-in view would cull away a higher amount of glyphs, as explained in Section 3.1. As all devices that were included in this survey were found to support textures of size 2048 by 2048 texels, this size was chosen for the benchmark to minimize cache misses that might diminish performance. As the caches converged quite quickly, *i.e.* the caches contained all glyphs of the test data, the frame rate that is presented here was sampled over a 10 second period after cache convergence.

During the development process of the benchmark tool and the techniques, several benchmarks were performed to test and improve performance, in accordance with iterative development. The results in the following subsections represent the final versions of the techniques, which are described in Section 3.

4.1.1 Glyph Cache

The performance results, in Figure 4, of *Glyph Cache* confirms a number of interesting things, one of these things was that the maximum frame rate of each device is limited by the refresh rate of its screen, which is different for each device. The Web Audio specification data set causes all devices to hit their maximum frame rate, confirming that it was ideal for caching. Also, the Galaxy Tab 10.1 hits its frame rate limit for all test data, which is due to its powerful Tegra 2 chip. Comparing the benchmarks results with the specifics in Table 2, there is a clear correlation between lower frame rate and a high amount of draw calls, as the Web Audio input data performed better, on all devices, than the Chinese and Japanese input data despite the former having a higher number of vertices. This confirms that the reason for the very poor performance of the Base technique was in part related to the very high amount of draw calls.

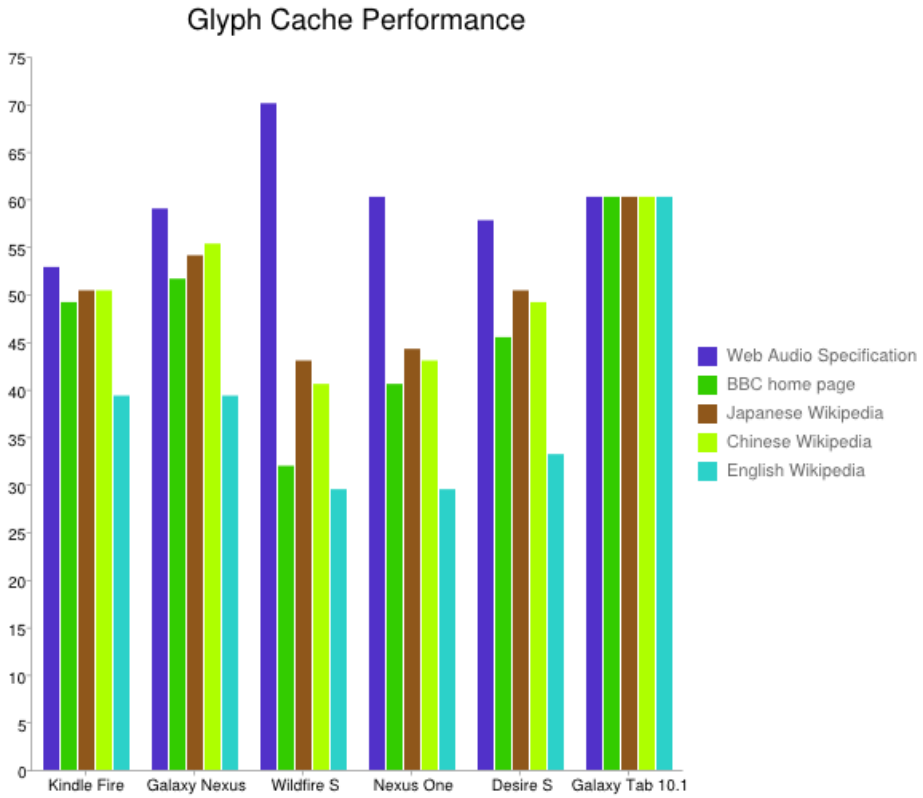

Figure 4: Illustration of Glyph cache benchmark results where the performance was measured in FPS, the results are grouped depending on the device that was benchmarked, and each result is a bar in the graph.

4.1.2 Global Cache

The change in rendering and caching techniques and the addition of a search tree, as mentioned in Section 3.3, resulted in a drastic performance improvement. The number of draw calls were reduced to 1 each frame, as shown in Table 3. In Figure 5 the benchmark results of the final version of the technique is illustrated. The benchmarks resulted in all, but the HTC Wildfire S, hitting their refresh rate limit on every dataset. After minimizing the number of draw calls, there is still a correlation between the frame rate and the mean number of vertices in the results of the Wildfire S. While the results for the HTC Wildfire S is not perfect, the frame rate was increased by almost twice as much, compared to the Glyph cache technique.

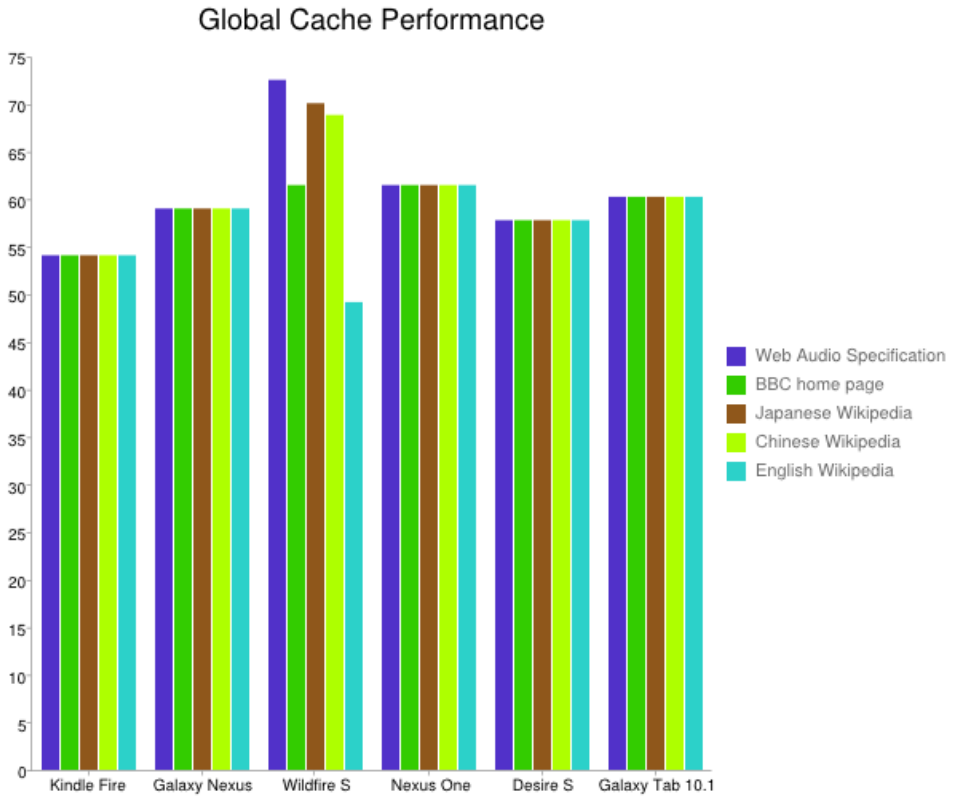

Figure 5: Illustration of Global cache benchmark results where the performance was measured in FPS, the results are grouped depending on the device that was benchmarked, and each result is a bar in the graph.

Input	Draw calls	Number of vertices
Web Audio Specification	1	5500
BBC homepage	1	7500
Chinese Wikipedia	1	4500
Japanese Wikipedia	1	3700
English Wikipedia	1	10200

Table 3: This table of Global cache results shows the amount of draw calls and the average number of vertices, rounded of to the nearest hundred, rendered in each.

5

Discussion

This thesis has shown that the device market is ready for a transition to GPU-based rendering. Even the most low-end device that was benchmarked almost hit the 50 FPS mark. This transition would, as discussed in Section 2.1, potentially lower the energy consumption of any application.

The early benchmarks of the Glyph cache technique showed that the main bottleneck did not lie with the number of vertices but rather the amount of draw calls. This was the main motivation as to why the *Global Cache* technique was designed and implemented instead of working on trying to minimize the number of vertices using methods that were described in Section 2.2.1. This realization was surprising since the vertex count was expected to be the main bottleneck that had to be resolved.

Due to memory limitations on low-end devices in this thesis, the size of the test data was limited to allow the benchmark tool to be run at all. The largest possible test data size was used on all devices during the test suite to allow as similar test situations as possible on all devices.

The minimization of the number of draw calls was key in finding an efficient technique that performs well on a majority of devices. However, the HTC Wildfire S never hit its refresh rate limit for all input data. This shows that GPU-based rendering may not be viable on all devices. Indeed, low-end devices with small screens may be more suitable for CPU-based rendering instead, as discussed in Section 2.1. As devices will continue to evolve according to Moore's law, the future outlook of GPU-based rendering looks good as newer devices will have better GPU performance.

Prior to development of the Base technique, FreeType-gl [34] was investigated and it had one important issue that conflicted with this thesis' goals, as presented in Section 1.3. This conflict [35] was found on the project's issue tracker and it concerned the support of Asian languages. While the project may have been improved by now, it was clear that it did not suite this thesis purposes at that moment.

When the positive results of the Global cache technique was confirmed, additional work was done on the benchmark tool to add more functionality to it. These functionalities consisted of adding color information to the glyphs and adding images to the test data. These images were added to create an even more realistic test suite for the render data from websites. Due to time

limitations, these additions were not subject to extensive benchmarking and were therefore omitted from this thesis. However, it showed that it would have been possible to include images into the Global cache and still render the whole screen in one draw call, as long as the texture is large enough.

6

Future Work

One point that would have been interesting to investigate further is the cache techniques. The implementations in connection to this thesis all used a LRU scheme, as it was thought be fit well with the task. However, there are a number of caching algorithms that might work ever better, *e.g.* MRU or an algorithm that utilizes linguistic facts, to make the caching more efficient. It would have also been interesting to investigate cache miss related performance hits in a situation where a user moves from one website to another. However, this was down prioritized and focus was put on developing a fast rendering technique instead.

One of the first papers that was examined during the course of this thesis was the paper on signed distance fields, briefly discussed in Section 2.2.3, by Green. While it served as an inspiration, it was never implemented. The reason for this was that neither minimizing the use of texture space nor implementing zooming was prioritized. However, supplying a continuous high-quality text zoom feature could be an important goal in any end-product and it is here the paper by Green can serve a larger purpose than it did in this thesis.

Global cache was optimized with a search tree to improve performance but before this was implemented, a hash table was considered. As the search tree performed really well, it was deemed unnecessary to focus on optimizing the glyph search further, but it can be a future point of optimization if searching evolves into a bottleneck again in the future.

The choice of font engine could be investigated further, but due to cache convergence it had little impact on rendering performance. There was no need to second guess this decision during development of the benchmark tool as there were other issues that hindered performance.

An emerging market right now is TVs that run an operating system that allow the user to connect to the Internet and receive content from media servers, similarly to modern mobile phones and tablets. It would have been interesting to see a text rendering survey on different TVs, as done in this thesis.

I would like to end this thesis by repeating some words of Andersson, the author of the master thesis discussed in Section 2.1.1. Now that it has been

confirmed that bitmap based font rendering on the GPUs of devices works well and that it is a technique that will likely scale well with the future device hardware developments, I leave the examination of GPU-based curve rendering on devices for future research.

I would love to see the results.

References

[1] OpenSignal, Inc, "Android Fragmentation Visualized." http://opensignalmaps.com/reports/fragmentation.php, 2012. [Online; accessed 17-October-2012].

[2] C. Loop and J. Blinn, "Resolution independent curve rendering using programmable graphics hardware," in *ACM Transactions on Graphics (TOG)*, vol. 24, pp. 1000–1009, ACM, 2005.

[3] Mark J. Kilgard, "An Introduction to NV_path_rendering." http://developer.download.nvidia.com/assets/gamedev/files/An_Introduction_to_NV_path_rendering.pdf, 2011. [Online; accessed 18-December-2012].

[4] The Unicode Consortium, "What is Unicode?." http://www.unicode.org/standard/WhatIsUnicode.html, 2011. [Online; accessed 14-October-2012].

[5] Android development team, "Android NDK — Android developers." http://developer.android.com/tools/sdk/ndk/index.html, 2012. [Online; accessed 12-October-2012].

[6] The FreeType Project, "The FreeType Project." http://freetype.org/, 2012. [Online; accessed 28-October-2012].

[7] The FreeType Project, "FreeType licenses." http://www.freetype.org/license.html, 2012. [Online; accessed 17-October-2012].

[8] Nintendo, "Wii Operations Manual - Channels and Settings." http://www.nintendo.com.au/support/files/Wii_Manuals/WiiOperationsManualChannelsAndSettings.pdf, 2008. [Online; accessed 21-October-2012].

[9] Blender, "Blender Dependencies." http://wiki.blender.org/index.php?title=Dev:2.5/Doc/Building_Blender/Linux/scons/options&action=history, 2012. [Online; accessed 21-October-2012].

[10] Megan Geuss, "Why Your Smartphone Battery Sucks," *PC world*, 2011. [Online; accessed 21-October-2012].

[11] Nvidia, "Doing More with Less of a Scarce Resource." http://www.nvidia.com/object/gcr-energy-efficiency.html, 2012. [Online; accessed 21-October-2012].

[12] M. Andersson, "Font Rasterization for Mobile Devices," 2008.

[13] Nvidia, "What is CUDA." http://developer.nvidia.com/cuda/what-cuda, 2012. [Online; accessed 25-October-2012].

[14] Dale Roe, "OpenCL gets touted in Texas," *Macworld*, 2008. [Online; accessed 25-October-2012].

[15] The Khronos Group, "Conformant Products." http://www.khronos.org/conformance/adopters/conformant-products#topencl, 2008. [Online; accessed 25-October-2012].

[16] Android development team, "Platform Versions." http://developer.android.com/about/dashboards/index.html, 2012. [Online; accessed 25-October-2012].

[17] Android development team, "Platform Versions." http://developer.android.com/about/versions/android-4.0.html, 2012. [Online; accessed 25-October-2012].

[18] Nvidia, "Improve Batching Using Texture Atlases." http://developer.download.nvidia.com/devzone//devcenter/tools/files/Texture_Atlas_Whitepaper.pdf, 2004. [Online; accessed 25-October-2012].

[19] The Unicode Consortium, "The Unicode Standard: A Technical Introduction." http://www.unicode.org/standard/principles.html, 2012. [Online; accessed 25-October-2012].

[20] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[21] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, M. Tan, *et al.*, "Semantic data caching and replacement," in *Proceedings of the international conference on Very Large Data Bases*, pp. 330–341, INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1996.

[22] R. Dumont, F. Pellacini, J. Ferwerda, *et al.*, "A perceptually-based texture caching algorithm for hardware-based rendering," in *Proceedings of 2001 Eurographics Workshop on Rendering*, pp. 249–256, 2001.

[23] L. Williams, "Pyramidal parametrics," in *ACM Siggraph Computer Graphics*, vol. 17, pp. 1–11, ACM, 1983.

[24] Jonathan Blow, "Implementing a texture Caching System," *Game Developer*, vol. 5, pp. 46–56, April 1998.

[25] C. Green, "Improved alpha-tested magnification for vector textures and special effects," in *ACM SIGGRAPH 2007 courses*, pp. 9–18, ACM, 2007.

[26] Y. Kokojima, K. Sugita, T. Saito, and T. Takemoto, "Resolution independent rendering of deformable vector objects using graphics hardware," in *ACM SIGGRAPH 2006 Sketches*, p. 118, ACM, 2006.

[27] T. Akenine-Möller, E. Haines, and N. Hoffman, "Real-time rendering 3rd edition," pp. 549–552, Natick, MA, USA: A. K. Peters, Ltd., 2008.

[28] A. Rege, "Optimization for Directx 9 Graphics." In Game Developers Conference, March 2004.

[29] The World Wide Web Consortium, "Web Audio API." `http://www.w3.org/TR/webaudio/`, 2012. [Online; accessed 17-November-2012].

[30] BBC, "BBC - Homepage." `http://www.bbc.com/`, 2012. [Online; accessed 18-November-2012].

[31] Wikimedia Foundation, "Wikipedia." `http://zh.wikipedia.org/wiki/Main_Page`, 2012. [Online; accessed 18-November-2012].

[32] Wikimedia Foundation, "Wikipedia." `http://ja.wikipedia.org/wiki/Main_Page`, 2012. [Online; accessed 18-November-2012].

[33] Wikimedia Foundation, "Wikipedia, the free encyclopedia." `http://en.wikipedia.org/wiki/Main_Page`, 2012. [Online; accessed 18-November-2012].

[34] Nicolas Rougier, "freetype-gl - OpenGL text using one vertex buffer, one texture and freetype." `http://code.google.com/p/freetype-gl/`, 2012. [Online; accessed 25-November-2012].

[35] Nicolas Rougier, "Issue 9: Improvement for Asian language." `http://code.google.com/p/freetype-gl/issues/detail?id=9`, 2012. [Online; accessed 25-November-2012].

A

Devices' specs

The devices in Table 4 were included in the benchmark suites for this thesis. The table also includes the hardware details, *e.g.* system on a chip (SoC) model and screen resolution. These details were chosen because they were relevant to the development and utilization of the benchmark tool.

Model	Developer	SoC model	Screen resolution
Kindle Fire	Amazon.com	Texas Instruments OMAP 4430	1024x600
Galaxy Nexus	Google	Texas Instruments OMAP 4460	1280x720
Wildfire S	HTC	Qualcomm MSM7227	480x320
Nexus One	Google	Qualcomm QSD8250	800x480
Desire S	HTC	Qualcomm MSM8225	800x480
Galaxy Tab 10.1	Samsung	Nvidia Tegra 250 T20	1280x800

Table 4: The hardware specifications for the devices included in the benchmark suites of this thesis.