Operating System Compositor and Hardware Usage to Enhance Graphical Performance in Web Runtimes

Antti Peuhkurinen[1(✉)], Andrey Fedorov[1], and Kari Systä[2]

[1] Huawei Technologies Oy, Helsinki, Finland
{antti.peuhkurinen,andrey.fedorov}@huawei.com
[2] Tampere University of Technology, Tampere, Finland
kari.systa@tut.fi

Abstract. Web runtimes are an essential part of the modern operating systems and their role will further grow in the future.Many web runtime implementations need to support multiple platforms and the design choices are driven by portability instead of optimized use of the underlying hardware.Thus, the implementations do not fully utilize the GPU and other graphics hardware.The consequence is reduced performance and increased power consumption. In this paper, we describe a way to improve the graphical performance of Chromium web runtime dramatically. In addition, the implementation aspects are discussed.

Keywords: Graphics · Web runtimes · Performance

1 Introduction

Performance of the graphics rendering in web runtimes becomes more important when the web applications get more visual and dynamic. Many design decisions in current web runtimes aim at portability and are done before the current enablers like graphical processing units were common. In addition, some of the design decisions have been made poorly in the very beginning in the web runtimes. Examples include the design of the graphical scene graph and lack of using shared buffers between the processes.

In this paper we describe design and prototype of a web runtime that can use the latest hardware enablers to achieve maximum performance. The proposed design is suitable also to other web runtime implementations. Details of texture compression algorithms, driver level buffer update synchronization, display hardware, hardware compositor and graphics processing unit internal design are beyond the scope of this paper.

The rest of this paper is structured as follows. Section 2 discusses background of this work, including a typical architecture designs in mobile terminals and an introduction to the prototype we are implementing. Section 3 describes the principles of the prototype we have implemented, together with an overview

A. Bozzon et al. (Eds.): ICWE 2016, LNCS 9671, pp. 381–388, 2016.
DOI: 10.1007/978-3-319-38791-8_23

of the implementation. Section 4 discusses lessons we have learned during the testing of the implemented prototype. Section 5 concludes the paper with some final remarks.

2 Background

Figure 1 we show an example from common graphics pipeline where output of two applications are rendered by the *operating system compositor* to the device screen. With an operating system compositor we mean the lowest level graphical compositor that combines graphics and visuals from several running applications to a single display. For example, in a mobile device the status bar, user interface of the running applications and pop-up notifications are most probably output of different processes. Initially the applications draw to their own frame buffers. The application's frame buffers are then moved through a ring buffer to the compositor process. Ring buffering is used to create double or triple buffering. With multiple buffers the application can draw to a frame buffer simultaneously with a compositor reading another frame buffer from same ring buffer without blocking happening. This ensures fast throughput.

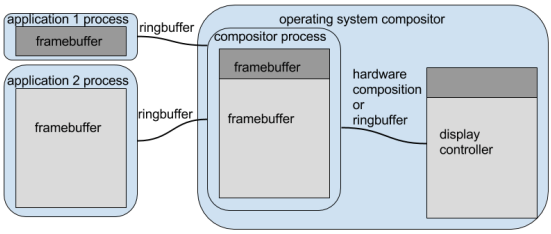

Fig. 1. Common graphics pipeline from application to display controller

The compositor process is responsible of moving the application surface data to the display controller. The connection between a compositor and a display controller is done either with a straight hardware composition in the display controller or with a second ring buffer between the compositor and the display controller. In simplest case the content of a frame buffer can be directly copied to a certain area of a display frame buffer. Ring buffer is used in cases where the compositor process needs to process the application frame buffer and enable simultaneous and non-blocking operation of the display controller.

In our earlier research, we have created a prototype of a 3D volume manager which serves in a similar function as a traditional window manager and operating system compositor but works in a 3D context [1–3]. One of the key contributions of that research was the graphics protocol between the applications and the operating system compositor.

Figure 2 shows the new pipeline that was enabled by our earlier work. In our pipeline application can have multiple shared buffers in use without need of an

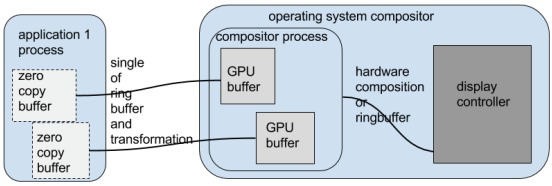

Fig. 2. Single pass compositor and zero copy GPU buffers

intermediate ring buffering. In addition, application can map it's buffer straightly to specific area on display. In this paper, we call a buffer which had a copying removed between processes as a *zero copy buffer*. The zero copy means in this context that the application does not need to copy the buffer to the compositor, but instead the compositor can use the same shared buffer straightly. In addition, the compositor buffers can be mapped to be exact part of a display which can relax the graphics pipeline even more. When a buffer is mapped to a certain display area we say that the buffer is using a *hardware layer*.

In the research reported in this paper we wanted to study feasibility of this protocol with an existing web runtime. One suitable web runtime candidate for the research was the Chromium web runtime. Chromium has a multi-process architecture and moves graphical data between it's processes. The Chromium's development roadmap did not have any similar work planned so we saw this as a good area to continue our research [6].

To make the Chromium to use our graphics pipeline, several technological aspects related to the operating system compositor required some special considerations. To begin with, an attention must be paid to management of graphics drawing within the web runtime. We need to control the textures and their life time, format and transformation. Moreover, when creating the textures we need to control the amount of needed buffers and use of hardware layers. All these elements are addressed in the following.

Figure 3 depicts typical processes running in Chromium. In the internal graphics processing of Chromium it is normal to copy graphical data from a process to another. In addition, Chromium composites most of it's own graphical data to a surface given by the operating system compositor. Chromium treats this as the main interface to adapt to different operating systems and hardware. In some custom cases, like video playing, Chromium can use platform specific adaptations for better performance.

3 Implementation

3.1 Graphics Protocol

In our earlier prototype applications were able to move their graphical scene to the operating system compositor. Instead of each application drawing themselves to single rectangular frame buffers, the applications moved their textures

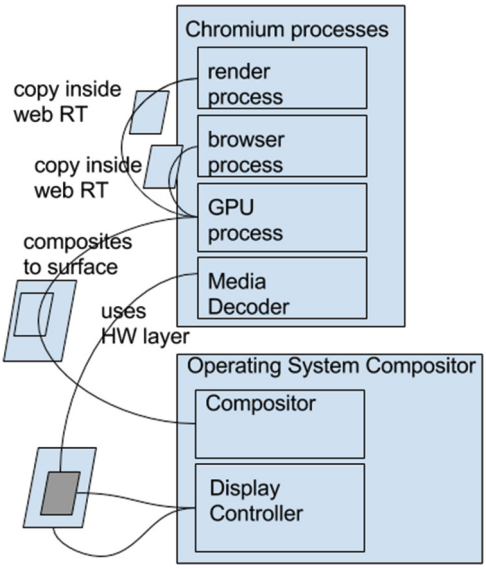

Fig. 3. Default Chromium graphics stack

and texture transformation data to the compositor which then composites the graphical scene to a final screen buffer [2]. The benefits from this kind of protocol are the possibility to use directly lower level graphics buffers and the enablers for more hardware dependent and multi-application (or multi-process) drawing optimization. In addition, when application scene changes it is possible to move only the changes instead of drawing the whole surface again for the compositor. For example in case of animation, the graphic libraries can send the transformation data that is being applied to a texture. This way only the needed transformation information is sent to the compositor instead of first drawing a texture with the transformation being applied to it and then sending this whole new texture to the compositor.

For the transfer of the graphical data over the process boundaries we have defined four types of objects: (1) a render object combining the other three object types, (2) a transformation matrix that defines the transformation of the render object in application space, (3) a mesh that defines the vertex data, and (4) a texture which is a bitmap.

3.2 Graphics Protocol Data Formats

Figure 4 visualizes the atomic 3D data objects in our graphics protocol and the objects the following features. (1) The *mesh* supports shading and texture UV mapping. Each vertex has X, Y, Z values for position, X, Y, Z values for normal and U, V values of texture mapping. The prototype presented in this paper use only rectangular meshes. (2) The *textures* are based on rectangular

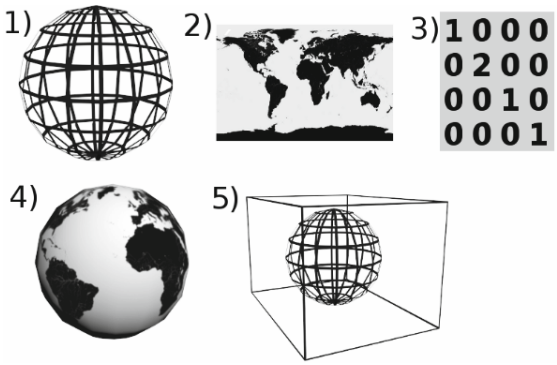

Fig. 4. Graphics protocol data formats

bitmaps. The textures can be in different formats and have single or multiple buffers. Textures with multiple buffers can work as a ring buffer. (3) Our *Transformation Matrix* is a 4×4 matrix that presents a transformation of the render object in the application space. Transformation contains location, rotation and scale. For example the meshes animated with CSS can use transformation matrix to control the animation instead of full using a series of rasterized bitmaps. (4) The *Render Object* connects the above three literals - mesh, texture and transformation matrix - and makes the combination drawable in the compositor. (5) *Application Volume* is the cuboid application space where the render objects of the application can be drawn into. We use a cuboid 3D volume to replace the old rectangular window from the old 2D application paradigm. This is because the graphics stack was originally designed for the augmented and virtual reality 3D application use cases [2].

3.3 Chromium with Enhanced Graphics

Our solution has been depicted in Fig. 5. The main difference to the Chromium's default architecture shown earlier in Fig. 3 is that now the processes of a single Chromium instance can share the buffers between themselves and with the compositor. In addition, Chromium can easily map multiple buffers to be hardware layers. There is no need to composite the website and the Chromium's user interface to a single framebuffer offered by the compositor anymore. This reduces the amount of drawing and buffering needed, thus reducing memory usage and GPU usage.

4 Evaluation

4.1 Power and Performance

The measurements where done by using Huawei P8 device [4]. We tested version 45 of Chromium using our own graphics stack against default P8 Android

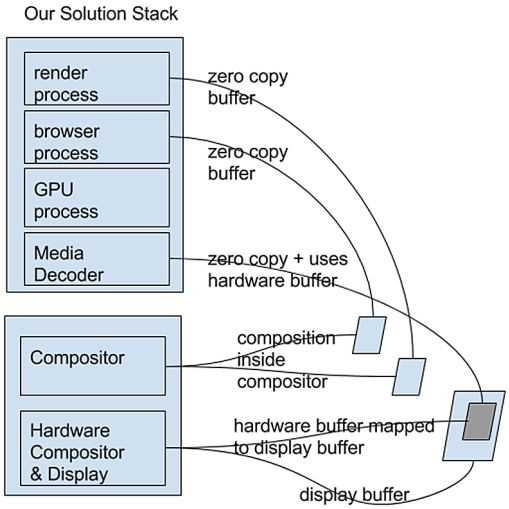

Fig. 5. Chromium with enhanced graphics

Chromium. For the initial tests we have used a Chinese website www.taobao.com as test content [5]. This website has some simple CSS animations, lot of CPU based text bitmap creation and lot of image data. The test was started by loading the page completely, then the page was scrolled down for a length of full screen height for five times waiting 2.0 s between the scrolls and then up five times. We measured scroll times and speeds so that they would as similar as possible in repeated tests. During the testing we measured the CPU, GPU load and clock speeds, memory consumption and total voltage over and amperes drawn from the battery. Agilent 66319D power supply was used as a battery replacement in our testing. Clock speeds and load where measured with sysfs for memory, CPU and GPU. During the tests we also took the screen brightness, device temperature, other processes run and all similar noise factors into account to make the test results more comparable. We call this test case as Taobao test case.

Figure 6 shows the initial test results from the Taobao test case. The test results got are averages over the time and ten test runs. The default Chromium measurements are placed on the left side and our solution's measurements are placed on the right side in each twin bar. The measurements show that the CPU usage is about the same 30 % load with all of the four cores running at 1200 MHz with both of the tested solutions. The GPU load is slightly smaller with our solutions. The GPU clock speed was between 280–480 MHz with our solution and 280–680 MHz with the default Chromium solution. The memory usage is about 20 % smaller with our solution. The power consumption is 17 % less with our solution compared to the default Chromium solution. In the drawing performance we achieved constant 60 frames per second with both of the solutions. It is possible to show heavier content with our system than with the default Android browser still achieving the constant maxed frame rate. This is because our system uses the hardware resources - mainly memory and GPU - more efficiently.

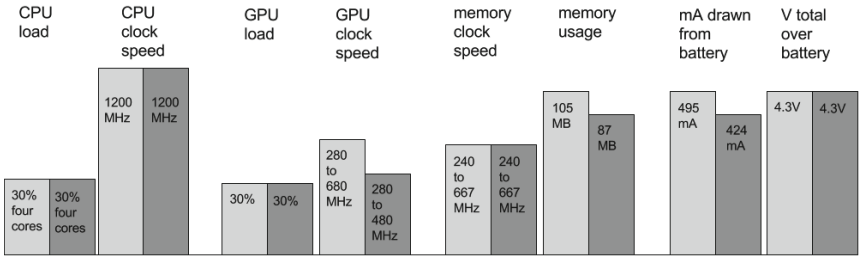

Fig. 6. Test results

4.2 Portability

We believe that our design could be easily ported also to other web runtimes and operating systems. Most portable component is the underlying compositor technology. To enable the web runtime side enhancements operating system compositor must have a similar flexible interface to access the hardware capabilities and to move the application graphical scene to the compositor side. The implementation of the compositor is done top of standard interfaces found from Android compatible hardware. This makes the compositor code portable to other up-to-date platforms. However, the portability of our concrete browser related code is Chromium specific but the general idea behind the changes is easy to adopt also in other web runtimes than Chromium. When porting the technology to other web runtimes it is essential to understand where the web runtime is allocating buffers and how it is managing them internally and between processes.

4.3 Robustness

Robustness of the implementation is being tested currently. It seems that we do not have any major flaws in the design and implementation. We have discovered some small corner cases during the implementation and fixed them.

4.4 Security

When using a GPU it is always possible to introduce flaws in security when multiple processes access the GPU memory at the same time like shown for example by Lee et al. [8]. Our design has not removed this threat completely but it makes it smaller. This is because the web runtime processes are using the GPU much more less than in the earlier web runtime implementation. This means that most of the GPU usage takes place in the operating system compositor. In the operating system compositor we have a better control from the GPU usage because the compositor is one of the vendor controlled system applications. This isolation enhances the security and makes the overall security better.

5 Conclusions

For the success of the web application paradigm the performance and power usage improvements in the web runtime graphics processing are essential. In this paper, we have presented a novel operating system compositor which enables web runtime to use the hardware capabilities more efficiently. With this system we have measured clear power saving and optimized resource usage in a normal web runtime usage scenario. To create a production ready system more detail testing is needed to find out possible corner cases needing more polishing. We will continue this work to enhance the graphical performance of the Chromium browser even more. Next we focus to test the system overall robustness and do some more performance improvements at the same time.

References

1. Peuhkurinen, A., Mikkonen, T., Terho, M.: Using RDF Data as Basis for 3D Window Management in Mobile Devices. MobiWIS, Niagara Falls (2011)
2. Peuhkurinen, A., Mikkonen, T.: Three-dimensional volume managers replacing window managers in augmented reality application paradigm. Poster presented at Mobilesoft, Florence (2015)
3. Peuhkurinen, A.: Method for Displaying a 3D Scene on a Screen. US Patent US20140313197
4. Huawei P8 Mobile Phone. https://en.wikipedia.org/wiki/Huawei_P8. Accessed 13 Feb 2016
5. Taobao Website. https://www.taobao.com/. Accessed 15 Feb 2016
6. Chromium Graphics Roadmap. https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome/gpu-architecture-roadmap. Accessed 13 Feb 2016
7. Android Graphics Components. https://source.android.com/devices/graphics/#android_graphics_components. Accessed 14 Feb 2016
8. Lee, S., Kim, W., Kim, J., Kim, J.: Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In: IEEE Symposium on Security and Privacy, San Jose (2014)