Parallel Distributed Rendering of HTML5 Canvas Elements

Shohei Yokoyama and Hiroshi Ishikawa

Shizuoka University,
3-5-1 Johoku Naka-ku Hamamatsu 432-8011, Japan
{yokoyama,ishikawa}@inf.shizuoka.ac.jp

Abstract. In this paper, we explain the rendering of ultra-high-resolution web content using HTML5 <canvas> elements. Many high-resolution massive datasets have recently been presented on the web. For example, Google Maps provides satellite images of Earth's surface and atmosphere at various resolutions. However, the scope of information that a user can view depends on the number of pixels of the user's computer monitor, irrespective of the data resolution. Therefore, we propose a parallel distributed rendering method for web content using multiple LCD monitors. We demonstrate that our system can draw an 8240 pixel × 4920 pixel HTML5 <canvas> element over a 16-monitor tiled display wall.

Keywords: HTML5, Canvas Element, Parallel Rendering, Tiled Display Wall.

1 Introduction

Recently, massive datasets have become available on the web as information technology has developed. IDC reported that 998 EB of data were generated in 2010 in comparison with 161 EB in 2006 [9]. Such massive datasets are shared and exchanged over the Internet. Data use generally occurs in three stages: data generation, data processing and data visualization. Systems that handle massive datasets on the Internet must achieve high scalability at each of these three stages.

This paper describes a novel technique for realizing real-time, parallel distributed rendering of web content with user interaction. We focus on the <canvas> element, which is one of the newest web technologies. It is part of HTML5 and allows for dynamic scriptable rendering of 2D shapes and bitmap images. Our main contributions are summarized as follows.

- We propose a novel parallel distributed rendering technique using the HTML5 <canvas> element on tiled display walls.
- We demonstrate that the proposed technique achieves high scalability in rendering an ultra-high-resolution (8240 pixel × 4920 pixel) web application.
- We also illustrate a low-cost hardware and middleware for the proposed technique of web-based parallel distributed rendering. To our knowledge, this technique has not been described previously.

S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.): ICWE 2011, LNCS 6757, pp. 331–345, 2011.

A tiled display wall is a technique to build a virtual ultra-high-resolution display comprising multiple display monitors and is used to visualize ultra-high-resolution data. Many studies have investigated high-resolution displays, but most proposals targeted are for scientific and medical visualization. Although the developed techniques perform well, they have a very high cost. Furthermore, developers who work with such displays must have deep knowledge of programming and networking.

In this paper, we propose a novel technique for realizing parallel distributed rendering of the HTML5 <canvas> element on tiled display walls.

The population of web developers is growing at a tremendous pace. Many skilled web developers and programmers are working currently to create display applications. In addition, web browsers are used in various operating systems and apply many standards. Many web services such as Google Maps API are available on the Internet. We propose a method for using a high-resolution web application that is executed on a tiled display wall based on web standards that include web technologies such HTML5, JavaScript and PHP.

The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 explains our tiled display wall environment, which is the basis of this research. Our proposed method of realizing parallel distributed rendering of the HTML5 <canvas> element is described in Section 4. In Section 5, we discuss our experiments and evaluate the results. Finally, Section 6 concludes the paper.

2 Related Works

Parallel distributed rendering of high-resolution images has a long history [5]. The purpose of such research is to develop techniques for efficient rendering of three-dimensional computer graphics. OpenGL made real-time animation of computer graphics possible [2]. Subsequently, Chromium [10] unified these two technologies, the parallel rendering and the OpenGL, and realized real-time rendering of high-resolution images using a parallel distributed method.

The resolution of LCD monitor is insufficient to display high-resolution images. The best commercially available LCD monitor has WQXGA resolution (2560 pixel × 1600 pixel). Therefore, Chromium and other technologies are designed for parallel rendering on a tiled display wall [18].

A tiled display wall is a virtual ultra-high-resolution display consisting of multiple display monitors. Figure 1 illustrates both a tiled display wall and an LCD monitor displaying the same image from Google Maps. The tiled display wall not only produces a large display but also has many pixels. Therefore, it can show high-resolution content.

Many proposed approaches are used for building tiled display wall systems. NASA's Hyperwall [15] has a 64-megapixel tiled display wall consisting of 49 monitors (7 horizontal × 7 vertical). LambdaVision uses 55 monitors (11 horizontal × 5 vertical) and builds a 100-megapixel high-resolution tiled display wall system. Renambot et al. who are members of LambdaVision project also

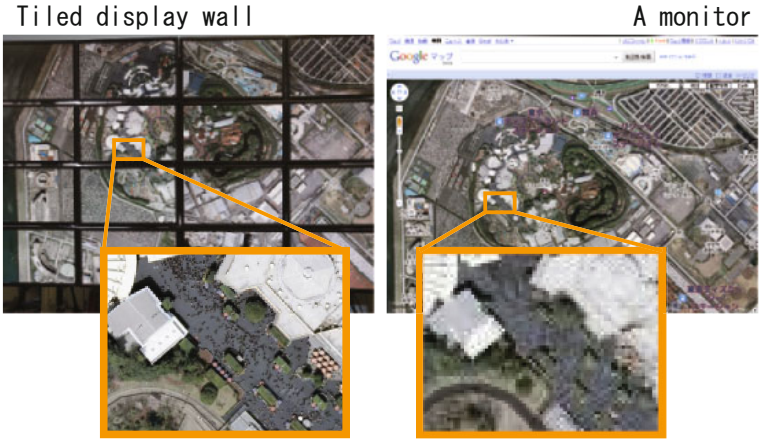


Fig. 1. Resolution of an LCD monitor and a tiled display wall

proposed middleware for tiled display walls called SAGE [14]. HIPerSpace [7], which has a 225-megapixel display, is an extremely large tiled display wall used at the University of California, San Diego. Existing tiled display wall systems were surveyed in detail by Ni et al. [13].

The performance and resolution of these systems are suited for developing applications for scientific visualization of life science data and ultra-high-resolution satellite images. Research issues for improving high-performance scientific computing related to tiled display walls have received more importance than developing consumer applications. Consequently, such applications require the use of expensive high-end machines, complex settings and extensive programming. However, web technologies are growing at a fast rate, ultra-high-resolution satellite images, e.g. Google Maps, are becoming increasingly available via web browsers and are valued by many users.

In other words, high-resolution visualization is no longer for scientists only: ordinary people can access it. Therefore, we built a low-cost tiled display wall consisting of low-end machines and based solely on web technologies. In addition, our tiled display wall system uses only web programming languages. The main purpose of this research is the rendering of <canvas> element of HTML5, which is one of the newest web technologies.

Our proposed system uses HTML5 <canvas> element and JavaScript instead of OpenGL and other graphic APIs for writing applications that produce 2D and 3D computer graphics. That is, multiple <canvas> elements that are displayed on web browsers executed on multiple computers build a virtual high-resolution and large <canvas> element. Research on distributed web interfaces has handled cooperative processing using multiple web browsers [11,19]. However, these studies do not consider whether the monitors are adjacent and do not attempt parallel rendering in the environment of the tiled display wall. Our system realizes cooperative parallel distributed rendering on a tiled display wall.

Applications of the system are implemented using common web technologies including HTML5, JavaScript and server-side scripting. Our tiled display wall middleware provides a single, extensive <canvas> area to developers although it consists of multiple <canvas> elements displayed on distributed machines. This is because the distributed environment is hidden and is realized by the middleware.

3 Web Based Tiled Display Wall

3.1 Hardware and Software Architecture

The software and hardware architecture of our web-based tiled display wall are shown in Figure 2. As shown in the figure, the system consists of multiple computers (`Receivers`), which is part of a tiled display, a computer (`Commander`) as the user interface for web applications on the tiled display wall, and a web server (`Messenger` in the figure) for synchronizing the monitors.

To develop a high-resolution web application on the tiled display wall, developers must implement two PHP programs, commander.php and receiver.php. The commander.php program is accessed from the `Commander`, and it includes a graphical user interface designed and implemented using HTML and JavaScript. The receiver.php program is accessed from the Receiver; it includes high-resolution

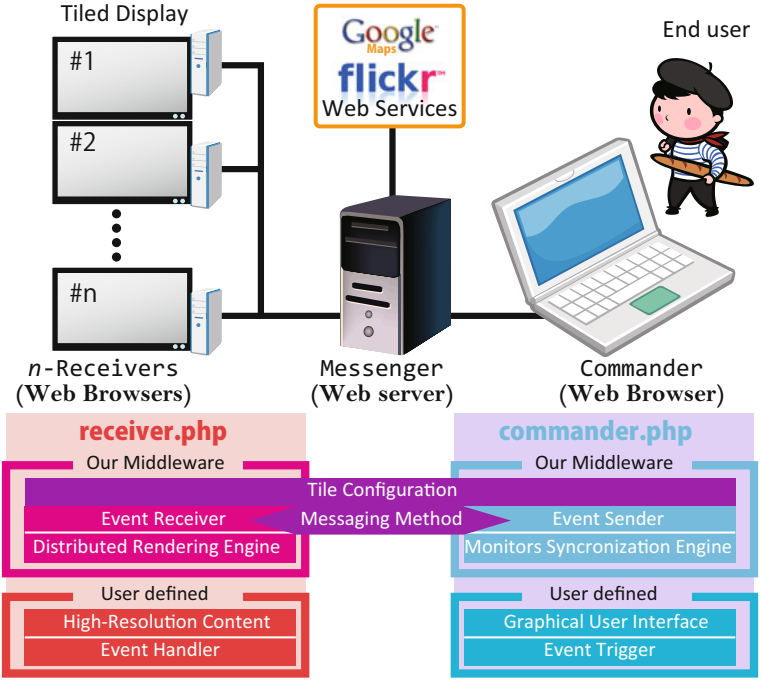


Fig. 2. Hardware and software architecture

Fig. 3. Tiled display wall

content designed using HTML and JavaScript. In addition, both PHP programs import our runtime library, which is written in JavaScript and provides a messaging method, and a configuration for the tiled display wall.

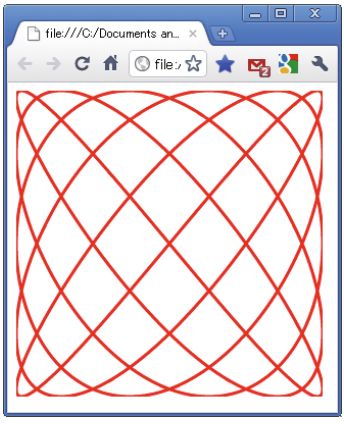
Both commander.php and receiver.php are stored on the web server (`Messenger`). Therefore, the first step in executing an application is to download commander.php and receiver.php on the `Commander` and the `Receivers`, respectively, from the `Messenger` via their web browsers.

Figure 3 shows our 16-monitor tiled display wall testbed, which uses this architecture. The tiled display wall consists of 16 full-HD (1920 pixel × 1080 pixel) LCD monitors and 16 "nettop" PCs that have Atom (Intel Corp.) processors and act as `Commander`s. Our display differs from existing tiled display walls in that the rendering engine is built on a web browser. All the `Receivers` display a web browser in a kiosk mode, a full-screen mode with no toolbars. For example, Internet Explorer can be launched in kiosk mode by using -k as a command line argument. The `Receivers` are essentially computers with simple factory settings because a web browser is pre-installed on a new computer; no extra program is necessary to build the proposed system.

The `Commander` is also a web browser. For this testbed, we select Wii (Nintendo), which has an Opera web browser (Opera Software ASA) as the `Commander`. Wii is a home video game console; an intuitive control, Wii remote, is available with it. The `Messenger` is a web server; we use Apache2 HTTP daemon on the Fedora Linux distribution (Red Hat Inc.).

3.2 Messaging

A user uses the `Commander` to control a tiled display wall application. In our testbed, the user holds a Wii remote. As the rendering is done on the `Receivers`, the user's interactions must be sent from the `Commander` to all the `Receivers`. Our middleware's messaging method is used for this purpose. Event triggers can be defined in commander.php and the event handler can be defined in receiver.php.




```
01:<html><body>
02: <canvas width=300 height=300 id="c0">
03: </canvas>
04: <script type="text/javascript">
05:  var canvas
06:   = document.getElementById('c0');
07:  /* Width and height of canvas */
08:  var width  = 300;    var height = 300;
09:  var ctx = canvas.getContext('2d');
10:  ctx.lineWidth = 3;  ctx.strokeStyle="#f00";
11:  ctx.beginPath();
12:  width/=2; height/=2;
13:  for(j=0;j<=2*Math.PI;j+=Math.PI/500){
14:    x=Math.sin(5*j) + 1;
15:    y=Math.sin(6*j) + 1;
16:    ctx.lineTo( x * width, y * height);
17:  }
18:  ctx.closePath();
19:  ctx.stroke();
20: </script>
21:</body></html>
```

Fig. 4. JavaScript code for rendering a Lissajous curve

The event consists of destinations, an identifier and options (arguments). Our middleware sends the message from the `Commander` to the `Receivers` specified as the destination. Our tiled display wall system is described in detail in our previous work [3,20].

4 Distributed Parallel Rendering

4.1 HTML5 <canvas>

The <canvas> element is part of HTML5 and allows for dynamic, scriptable rendering of 2D shapes and bitmap images. It is the first graphic API for pure HTML content, and it is expected to increase the expressiveness of web content. Figure 4 shows the JavaScript code for drawing a Lissajous curve and a screenshot of a web browser showing the Lissajous curve rendered on the <canvas> element. In the figure, the drawing line is defined between the beginPath method and the stroke method. The filled shape is rendered using a fill method instead of the stroke method. Other drawing methods include the bezierCurveTo method for drawing a bezier curve, the arcTo method for drawing an arc and the drawImage method for displaying image files such as .jpeg, .png and .gif. The colour and width of the line are varied by changing the properties of the context object (line 10).

We apply the <canvas> element and graphic API to parallel distributed rendering on tiled display walls. A <canvas> element generally cannot be placed over an entire tiled display wall because the wall is a distributed system consisting of more than one computer. However, our proposed method virtualizes multiple <canvas> elements as a single huge high-resolution <canvas> element over an entire tiled display wall. This method is described in the next section.

receiver.php

```
01:<html><head>
02:<?php echo $WallDispray_LIBLARY; ?>
03: <script type="text/javascript">
04: Ext.onReady(function(){
05:   receiver
06:     = new Wdm.Receiver(tileConfigure);
07:   receiver.start();
08: });</script>
09:</head></html>
```

commander.php

```
01:<html><head>
02:<?php echo $WallDispray_LIBLARY; ?>
03: <script type="text/javascript">
04: Ext.onReady(function(){
05:   var commander
06:     = new Wdm.Commander(tileConfigure);
07:   commander.start();
08:   /* Width and height of canvas */
09:   var width  = 4000;    var height = 4000;
10:   var canvas = commander.createCanvas(
11:     {"top"   :300,    "left" :900,
12:      "height":height,"width":width},"px");
13:   var ctx = canvas.getContext('2d');
14:   ctx.lineWidth = 30;  ctx.strokeStyle="#00f";
15:   ctx.beginPath();
16:   width/=2; height/=2; var x,y;
17:   for(j=0;j<=2*Math.PI;j+=Math.PI/500){
18:     x=Math.sin(5*j) + 1;
19:     y=Math.sin(6*j) + 1;
20:     ctx.lineTo(x*width,y*height);
21:   }
22:   ctx.closePath();
23:   ctx.stroke();
24: });</script>
25:</head></html>
```

Fig. 5. JavaScript code for distributed rendering of a Lissajous curve

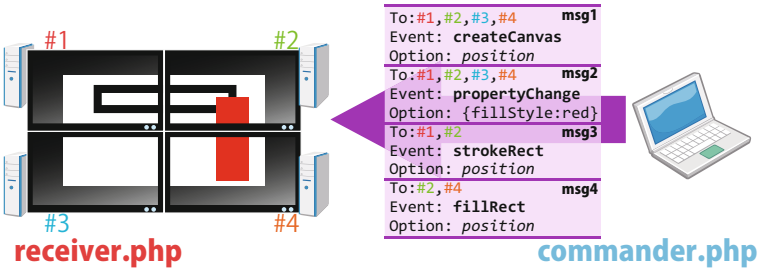


Fig. 6. Messages for drawing two rectangles on a tiled display wall

4.2 Rendering from commander.php

The code for distributed rendering of a <canvas> element is written in either commander.php or receiver.php. First, we describe the method of rendering from commander.php.

Figure 5 is an example of code that draws a 4000 pixel × 4000 pixel Lissajous curve on a tiled display wall. As shown here, the code that renders shapes and images on the tiled display wall is the same as that shown in Figure 4. However, the Canvas object (line 10) and the Context object (line 13) are our wrappers for a native object of the <canvas> element. If a method, e.g. lineTo (line 20), of the Context wrapper is called, our middleware sends the event to the `Receivers` via the messaging mechanism.

Although we use a complex system consisting of many computers, a developer can write code for a 4000 pixel × 4000 pixel <canvas> element without any knowledge of networking or a distributed environment. Our middleware delivers code appropriately to the corresponding `Receivers` on the tiled monitors.

receiver.php

```
01:<html>
02:<?php echo $JAVASCRIPT_LIBLARY; ?>
03: <script type="text/javascript">
04:   Ext.onReady(function(){
05:     receiver = new Wdm.Receiver(configure);
06:     var LissajousDrawer = Ext.extend(Wdm.Drawer,{
07:       constructor:function() {
08:         LissajousDrawer.superclass.constructor.apply(this, arguments);
09:         this.c=5;
10:       },
11:       draw:function(ctx){
12:         var width  = this.width;
13:         var height = this.height;
14:         ctx.lineWidth = 30;
15:         ctx.strokeStyle="#00f";
16:         ctx.beginPath();
17:         width/=2; height/=2; var x,y;
18:         for(j=0;j<=2*Math.PI;j+=Math.PI/500){
19:           x=Math.sin(((this.c++)%15)*j) + 1;
20:           y=Math.sin(((this.c++)%15)*j) + 1;
21:           ctx.lineTo(x*width,y*height);
22:         }
23:         ctx.closePath();
24:         ctx.stroke();
25:       },
26:       runner_mode:"sleep",
27:       runner_timing:1000
28:     }); /*draw() is called every 1000 msec*/
29:     receiver.setDrawer("CURVE",LissajousDrawer);
30:     receiver.start();
31:   });
32: </script>
33:</head></html>
```

commander.php

```
01:<html><head>
02:<?=$JAVASCRIPT_LIBLARY?>
03: <script type="text/javascript">
04:   Ext.onReady(function(){
05:     var commander
06:       = new Wdm.Commander(configure);
07:     commander.start();
08:     var canvas = commander.createCanvas(
09:       {"top"   :300,    "left" :900,
10:        "height":4000,"width":4000},"px",
11:        "CURVE");
12:     });
13: </script>
14:</head></html>
```

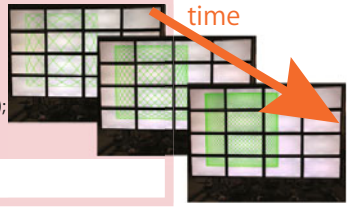


Fig. 7. JavaScript code for parallel distributed rendering of a Lissajous curve

Figure 6 shows examples of messages that are sent and received between the `Commander` and the `Receivers`. These messages create a <canvas> element and draw two rectangles, one black and one red. The black rectangle is drawn on monitors #1 and #2; therefore, the message is sent to the #1 and #2 receiver.php files. The red rectangle is drawn on monitors #2 and #4; therefore, the message is sent to the #2 and #4 receiver.php files. Thus, only monitors that must display part of a shape receive the message containing the draw event. This mechanism reduces network traffic between the `Commander` and the `Receivers`.

Because commander.php also shows a graphical user interface and handles event listeners that respond to user actions, a dynamic drawing based on user actions is suitable for this method. This method is a distributed rendering but not a parallel rendering because the drawing commands are centralized in commander.php. In addition, the latency between the `Commander` and the `Receivers` cannot be bypassed, particularly for animations. For this purpose, the rendering command is written in receiver.php. The screen shown in Figure 6 is drawn using these methods.

4.3 Rendering from receiver.php

The drawing commands written in receiver.php are delivered to all `Receivers` and executed in parallel. Figure 7 shows an example code for drawing different

Lissajous curves every second. The receiver.php file is downloaded by all the `Receivers`, but drawing commands that do not include the area of the relevant monitor are ignored.

The LissajousDrawer class, which extends Wdm.Drawer, is defined in receiver.php in Figure 7. LissajousDrawer has a constructor, a draw method and some properties. The draw method is called constantly, and the runner timing property (line 26 in receiver.php) is the sleep time between the method calls. The size and position of the <canvas> element are defined in commander.php (lines 09,10). The createCanvas method's third argument, CURVE, is an identifier of a Drawer instance. The identifier also appears in the argument of the setDrawer method (line 29 in commander.php). The Canvas object in commander.php and the Drawer object in receiver.php are bound together by the identifier.

5 Performance Evaluation

5.1 Photomosaic

To test our method, we implemented a complex rendering application consisting of a high-resolution photomosaic renderer and measured the parallel distributed rendering performance.

The photomosaic technique transforms an input image into a rectangular grid of small images. One of the earliest concepts of the photomosaic was presented by Salvador Dali [6]. His painting "Gala Contemplating the Mediterranean Sea which at Twenty Meters becomes a Portrait of Abraham Lincoln (Homage to Rothko)" looks like a portrait of Lincoln, but a nude of his wife Gala and a small portrait of Lincoln actually appear in the painting. This is the best-known image made from many other images. Robert Silvers, a student at MIT Media Lab, proposed a computer-aided photomosaic [16]. One of his masterpieces is a portrait of Mickey Mouse composed from many scenes from Walt Disney's movies [17].

The algorithm for generating a computer-aided photomosaic is as follows:

1. Divide an input image into small rectangular areas.
2. Search for images similar to each area from an image database.
3. Replace the areas with the similar images.

That is, the principal part of the algorithm consists of repeated searches for similar images. Blasi et al. proposed an efficient photomosaic generation method based on approximate nearest neighbour searching Blasi et al. [4,8] They showed that a 1024 pixel $\times$ 768 pixel photomosaic is created in about 32 s using a database of 1417 images of 10 pixel $\times$10 pixel on a personal computer that has an Athlon XP-M 1800+ CPU and 192 MB of RAM. This speed is sufficient to be used for a personal computer monitor, but the resolution of computer monitors is insufficient to render photomosaics. A high-quality photomosaic requires a high-resolution canvas; consequently, the photomosaic technique is often used for printed matter. However, by using the proposed method, a high-quality photomosaic having resolution sufficient for printing can be rendered dynamically

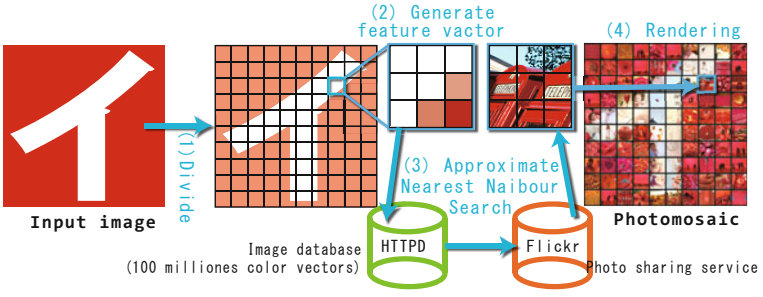


Fig. 8. Photomosaic generation

on a high-resolution tiled display wall. This is why we selected photomosaic generation as a benchmark of the proposed system.

The other important problem of photomosaic generation is the database size of the small images. The quality of a photomosaic is influenced not only by the number of pixels but also by the size of the database. This problem is solved by using the web because many photo-sharing websites share billions of images. In this section, we measure the performance of the entire system and assess the effectiveness of the proposed method using photomosaic generation based on photos on Flickr [1].

5.2 Setting

The benchmark application creates photomosaic images using large image sets from Flickr, a public image-hosting service. We crawled pictures from Flickr, divided each of them into nine sections and extracted the average colour of each section. We used one million colour vectors for this experiment and stored them in a database. A colour vector consists of red, green and blue levels, saturation and brightness of the nine subsections; therefore, one small image has a 45-dimensional vector. The specifications of the machines are listed in Table 1.

Figure 8 illustrates the photomosaic generation algorithm. The input photograph is subdivided, and each subsection is compared with the colour vectors of the image database. This process yields a Flickr URL for the closest match. That is, a list of URLs is generated for a given image, the images are downloaded from Flickr, and the photomosaic is drawn on a tiled display wall. Every `Receiver` searches and draws in parallel (see Figure 11 in the Appendix).

Table 1. Machine specifications

`Messenger`		Receiver	
M/B	Asus P6T7 WS SuperComputer	Model	Acer Aspire Revo
CPU	Intel Core i7 975EE (Quad Core)	CPU	Intel Atom Processor
Memory	12GB	Memory	2GB
OS	Linux (Fedora12)	OS	Windows Vista

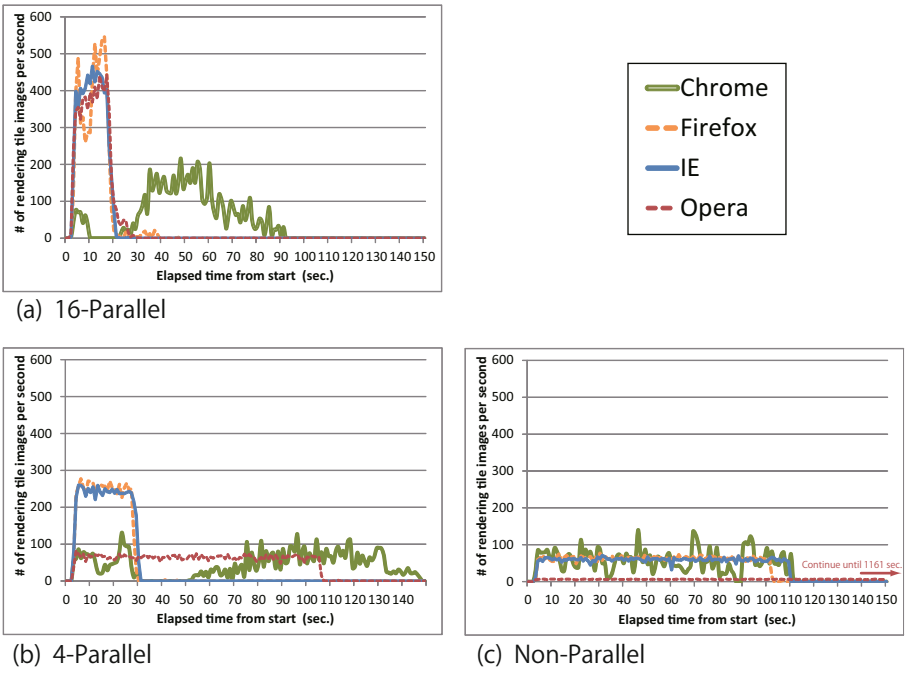


Fig. 9. Throughput of parallel distributed <canvas> rendering

Our testbed has a 8240 pixel × 4920 pixel display consisting of 16 monitors. We used one million Flickr thumbnail images of 75 pixel × 75 pixel resolution and measured the execution time to create and draw an 8240 pixel × 4920 pixel photomosaic using 16 monitors, four monitors and one monitor. When four monitors and one monitor were used, the photomosaic was drawn into a scrollable area because it was larger than the screen. We also used the ANN library [12] to search for similar images. The web browser on the `Receivers` is Chrome 9.0, Firefox 3.6.13, Opera 11.01 or Internet Explore 9 Beta. The results are the mean values of five executions.

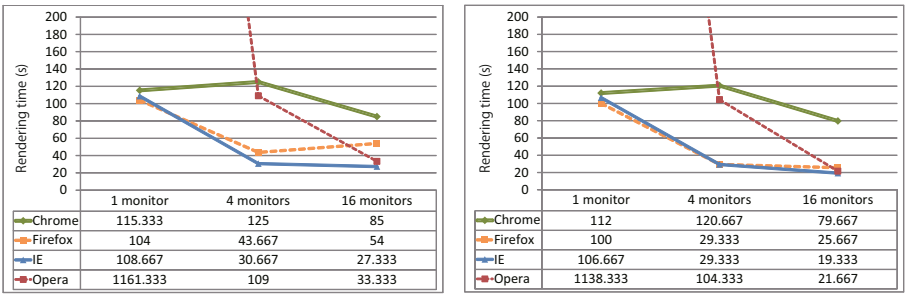
5.3 Result

Photomosaic rendering requires approximately 650 nearest neighbour searches in a short period. A few time-outs occasionally occur when 6500 thumbnail images are downloaded from Flickr. In this case, the application can send a re-send request to Flickr, but we ignore the time-outs in this experiment because the purpose is to estimate the performance of the proposed system, not error recovery. Table 2 shows the mean number of error images and thumbnail images downloaded from Flickr.

The number of thumbnails of each condition take different values. This is because when nearest neighbour searches occasionally return the same thumbnails, the web browser loads the image from cache memory.

Table 2. Average number of drawing tile images and HTTP errors

	Chrome		Firefox		IE		Opera	
	errors	images	errors	images	errors	images	errors	images
1 monitor	0.0	6350.0	2.7	6324.0	1.7	6346.0	1.3	7303.7
4 monitors	0.7	6457.0	3.3	6329.0	1.7	6352.7	1.0	6657.7
16 monitors	2.3	6620.3	25.0	6401.3	3.7	6515.7	1.0	6498.7



(a) Until 100% rendering (b) Until 98% rendering

Fig. 10. Rendering time

The results of this experiment are presented in Figure 9. Figure 9(a) uses all 16 monitors of the tiled display wall. All web browsers except Google Chrome drew more than 400 thumbnail images per second at their peak performance. In the case of four monitors, the peak performance of Firefox and Internet Explorer was approximately 250 images per second. In the case of one monitor (non-parallel conditions), the peak performance of Firefox and Internet Explorer was about 80 images per second. These results suggest that the throughput of parallel distributed <canvas> rendering is proportional to the degree of parallelism.

The rendering time is presented in Figure 10(a). The result includes an unexpected delay in HTTP response from Flickr. Therefore, we also measured the time to draw 98% of all the thumbnails [Figure 10(b)]. The fastest case was 27 s to create a photomosaic in Internet Explorer. In Firefox, the photomosaic was created in 54 s, but the time required to draw 98% of the thumbnails shown in Figure 10(b) is 25 s, which is similar to the other times. Opera shows the same trend, but the result of nonparallel operation is worse than the others. Only Chrome shows a different trend; it is slowest in the case of four parallel monitors. We think the reason is that Chrome has a high speed of operation. This is because a web browser on a `Receiver` sends many HTTP requests for the nearest neighbour search on the `Messenger`, exceeding its capacity and occupying all the reserved ports. Consequently, the other `Receivers` wait for completion of the drawing by the Receiver.

Finally, we present the memory consumption of Internet Explorer and Firefox in Table 3. The values represent the difference between the memory used before and after photomosaic rendering.

Table 3. Memory consumption

	Firefox	IE
1 monitor	67.8	232.7
4 monitors	35.9	41.8
16 monitors	17.9	8.4
		(MB)

Memory consumption might be proportional to the size of the <canvas> element. This is because the memory consumption of 16 parallel monitors is less than that of the others. The result shows this trend. These experiments clarify that a web-based tiled display wall achieves effective parallel distributed rendering of <canvas> elements.

6 Conclusion

As described in this paper, we propose a method of parallel distributed rendering of the HTML5 <canvas> element. We also describe the design of a web-based tiled display wall system. The experimental results show that the proposed system is highly efficient and scalable. High-resolution web content realized by this research will bring about qualitative changes in Internet applications.

This work has inspired several ideas:

- We implements WebSocket messaging between the `Commander` and the `Receivers`.
- We will attempt to improve the latency of distributed frame synchronisation and the timing of the draw method call of the Drawer object in receiver.php.

References

1. Flickr from yahoo, `http://flickr.com/`
2. OpenGL, `http://www.opengl.org/`
3. Wall display in mosaic, `http://shohei.yokoyama.ac/Wall_Display_in_Mosaic/en`
4. Blasi, G.D., Petralia, M.: Fast Photomosaic. In: Poster Proceedings of ACM/WSCG 2005, Citeseer (2005)
5. Crockett, T.: An Introduction to Parallel Rendering. Parallel Computing 23(7), 819–843 (1997)
6. Dali, S.: Gala contemplating the Mediterranean Sea, which at twenty meters becomes the portrait of Abraham Lincoln (1976)
7. DeFanti, T.A., Leigh, J., Renambot, L., Jeong, B., Verlo, A., Long, L., Brown, M., Sandin, D.J., Vishwanath, V., Liu, Q., Katz, M.J., Papadopoulos, P., Keefe, J.P., Hidley, G.R., Dawe, G.L., Kaufman, I., Glogowski, B., Doerr, K.-U., Singh, R., Girado, J., Schulze, J.P., Kuester, F., Smarr, L.: The optiportal, a scalable visualization, storage, and computing interface device for the optiputer. Future Generation Computer Systems, The International Journal of Grid Computing and eScience 25(2), 114–123 (2009)

8. di Blasi, G., Gallo, G., Petrali, M.P.: Smart ideas for photomosaic rendering. In: Eurographics Italian Chapter Conference 2006, pp. 267–272 (2006)
9. Fantz, J.F., Reinsel, D., Chute, C., Schlichting, W., McArthur, J., Minton, S., Xheneti, I., Tonoheva, A., Manfrediz, A.: The expanding digital universe. In: An IDC White Paper - Sponsored by EMC (2007)
10. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: A stream-processing framework for interactive rendering on clusters. In: ACM SIGGRAPH 2002, pp. 693–702 (2002)
11. Melchior, J., Grolaux, D., Vanderdonckt, J., Roy, P.V.: A toolkit for peer-to-peer distributed user interfaces: Concepts, implementation, and applications. In: Proceedings of The ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2009), pp. 69–78 (2009)
12. Mount, D.M., Arya, S.: Ann: A library for approximate nearest neighbor searching, `http://www.cs.umd.edu/~mount/ANN/`
13. Ni, T., Schmidt, G.S., Staadt, O.G., Livingston, M.A., Ball, R., May, R.: A survey on large high-resolution display technologies, techniques, and applications. In: Virtual Reality Conference, 2006, pp. 223–236 (2006)
14. Renambot, L., Rao, A., Singh, R., Byungil, J., Krishnaprasad, N., Vishwanath, V., Vaidya, C., Nicholas, S., Spale, A., Charles, Z., Gideon, G., Leigh, J., Johnson, A.: Sage: the scalable adaptive graphics environment. In: Workshop on Advanced Collaborative Environments, WACE 2004 (2004)
15. Sandstrom, T.A., Henze, C., Levit, C.: The hyperwall. In: Proceedings of International Conference on Coordinated and Multiple Views in Exploratory Visualization, pp. 124–133 (2003)
16. Silvers, R.: Photomosaics: Putting Pictures in Their Place. PhD thesis, Massachusetts Institute of Technology (1996)
17. Silvers, R., Tieman, R.: Disney's Photomosaics. Hyperion (1998)
18. Staadt, O., Walker, J., Nuber, C., Hamann, B.: A Survey and Performance Analysis of Software Platforms for Interactive Cluster-based Multi-screen Rendering. In: ACM SIGGRAPH ASIA 2008 Courses, pp. 1–10. ACM, New York (2008)
19. Vandervelpen, C., Vanderhulst, G., Luyten, K., Coninx, K.: Light-weight distributed web interfaces: Preparing the web for heterogeneous environments. In: Lowe, D.G., Gaedke, M. (eds.) ICWE 2005. LNCS, vol. 3579, pp. 197–202. Springer, Heidelberg (2005)
20. Yokoyama, S., Ishikawa, H.: Creating Decomposable Web Applications On High-Resolution Tiled Display Walls. In: Proceedings of the IADIS International Conference on WWW/Internet, pp. 151–158 (2010)

Appendix: Parallel Distributed Photomosaic Rendering

Fig. 11. Parallel distributed photomosaic rendering in 20 s. A movie version is available to the public at http://www.youtube.com/watch?v=ECZ03giPlXE.