Introducing Threaded Compositor (http://blog.ryumiel.net/articles/introducing-threaded-compositor/)

Date Tue 02 December 2014 Tags WebKit (http://blog.ryumiel.net/tag/webkit/) / WebKitGTK+ (http://blog.ryumiel.net/tag/webkitgtk/) / AcceleratedCompositing (http://blog.ryumiel.net/tag/acceleratedcompositing/) / Igalia (http://blog.ryumiel.net/tag/igalia/)

Since CSS Animation appeared, the graphics performance of WebKit has become an important issue, especially in embedded systems world. Many users (including me) want fancy and stunning user interfaces, even in a small handset device which have very restricted CPU and GPU.

Luckily, I could work through the years to improve the graphics performance of WebKit (http://webkit.org). And with the support from colleagues in Igalia (http://www.igalia.com), I just finished to make the Threaded Compositor for WebKitGTK+ (http://webkitgtk.org) as a review-ready state.

The Threaded Compositor focused to improve the performance of WebKitGTK+ for CSS Animation using dedicated thread for compositing. But not only CSS Animation, it also provides a way to improve the performance of scrolling, scaling, rendering of canvas and video element.

Before diving into the detail, it would be good to watch a video which introducing the Threaded Compositor.

Click here if a video does not appear above this text. (http://youtu.be/PJSsEiNasow)

The example used in this video can be accessed from here (http://ryumiel.github.io/spread). I modified the original version (http://github.com/MathiasPaumgarten/spread) to run automatically for benchmarking purpose.

Also, the current implementation of Threaded Compositor is available on the github (http://github.com/ryumiel/webkit-experimental/tree/threaded-compositor). It is based on the WebKit r176538.

Motivation

The main-thread is where everything gets executed, including layout, JavaScript execution, and other heavy operations. Thus, running accelerated compositing in the main-thread severely limits responsiveness and rendering performance. By having a separate thread for compositing, we can bring a significant performance improvement in scrolling, zooming, and rendering.

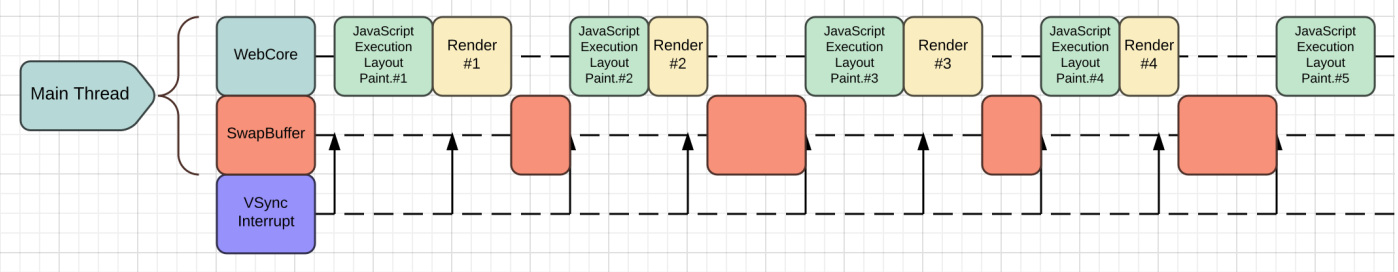
Currently, several ports have already implemented a way to perform compositing off the main-thread. Coordinate Graphics, which is used by Qt and EFL, runs accelerated compositing in UI Process, and Chromium has implemented Compositor Thread that runs off the main render thread.

Threaded Compositor is a threaded model of Coordinated Graphics. Unlike Coordinated Graphics, it composites contents in the dedicated thread of Web Process and doesn't use complicated inter-process shareable textures. Because of that, it is much easier to implement other multi-threaded rendering techniques which is covered at Compositing the Media and the Canvas.

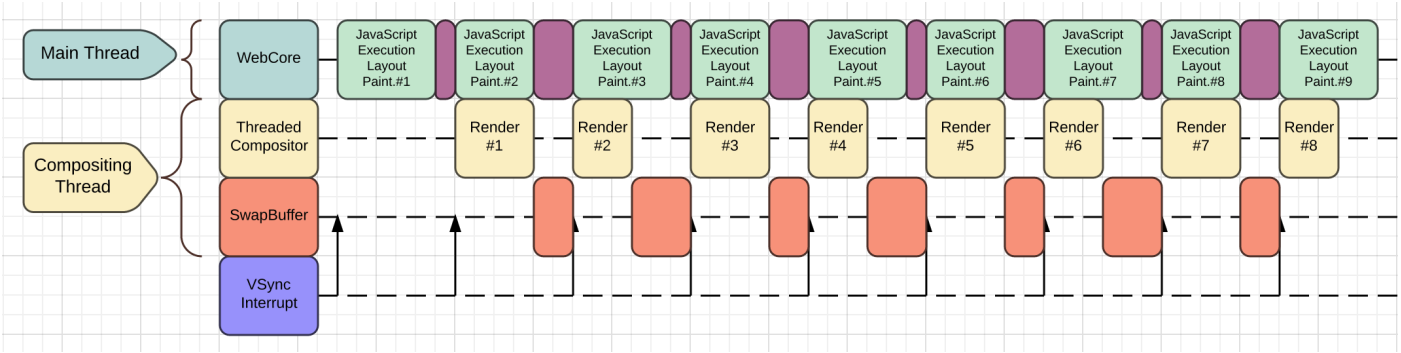
Off-the-main-thread Animation

Compositing a contents using OpenGL[ES] is a basic technique to accelerate CSS animations. However, even we are using GPU accelerated compositing, we can face a V-sync pitfall. (This happens more frequently in the embedded systems.) Most of GPU uses V-sync (Vertical synchronization) to prevent the screen tearing (http://en.wikipedia.org/wiki/Screen_tearing) problem. It may blocks OpenGL's SwapBuffer call at most 16 ms - if the monitor's refresh rate is 60 Hz - until the next vblank interrupt.

As you can see in the below diagram, this problem can waste the main-thread's CPU time.



Again, when the main thread executing OpenGL[ES]'s SwapBuffer call, the driver should wait V-sync interrupt from the DRM/KMS module. The above diagram shows the worst case scenario. The main-thread could render 6 frames in given time if there was no V-sync problem. But with the V-sync, it renders just 4 frames. It can be happened more frequently in embedded environments which have less powerful CPU and GPU.

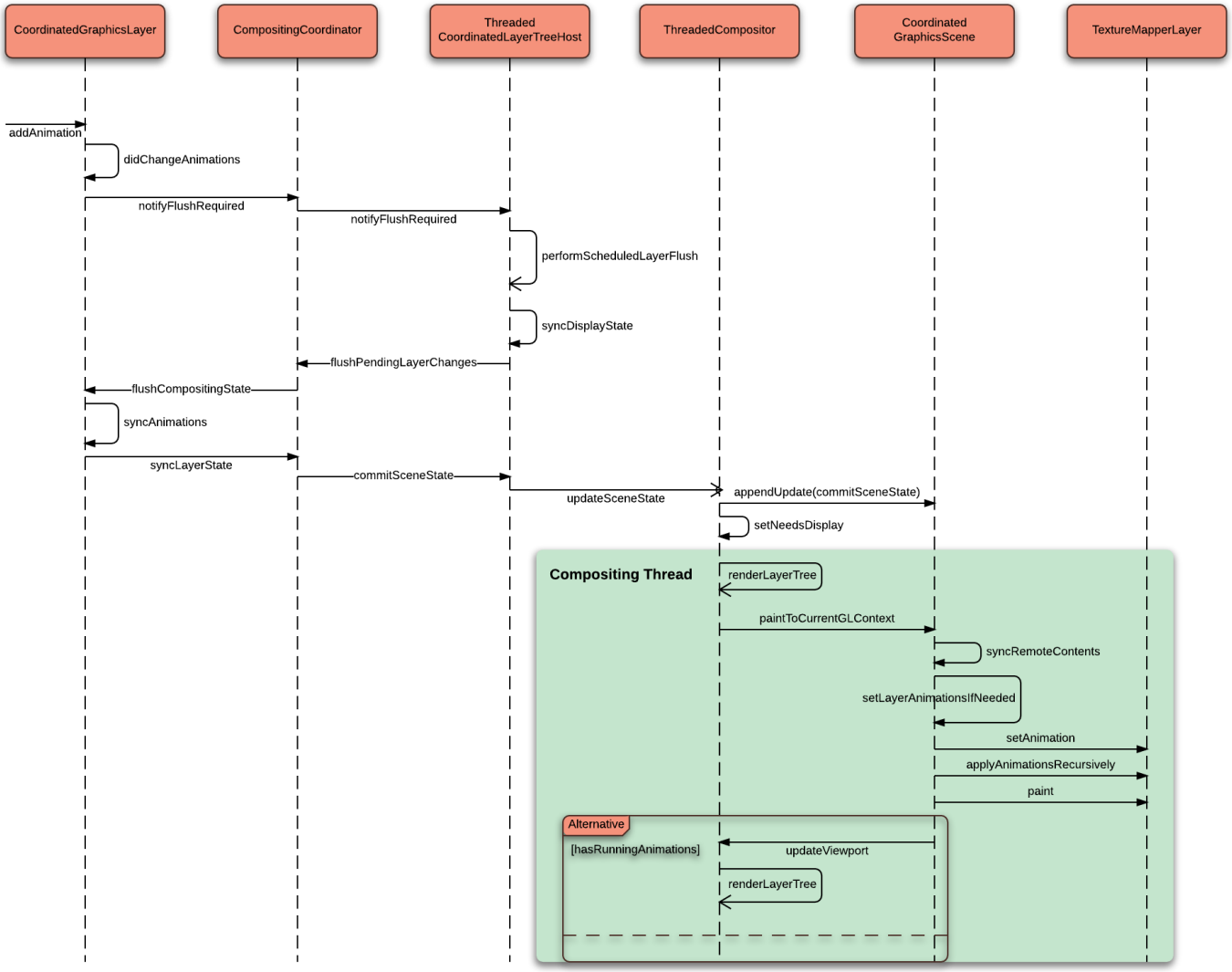


To overcome this problem, we uses a very common technique to use a different thread to update screen. As visualized at above diagram, the compositing thread takes all overheads from the V-sync problem. The main-thread can do other job while the compositing thread handles OpenGL calls. (Purple rectangles in the main-thread indicate inter-thread overheads.)

Moreover, the Threaded Compositor accelerates CSS Animations, also.

Because all of the layer properties (including GraphicsLayerAnimation) are passed to the compositing thread, it can make available to run CSS animations on the compositing thread without interrupting the main-thread. When CoordinatedGraphicsScene finishes to render a scene, it will check is there any running animations in the its layer tree. If there are active one, it sets a timer to render a scene itself.

Attached sequence diagram can be helpful to understand this idea.

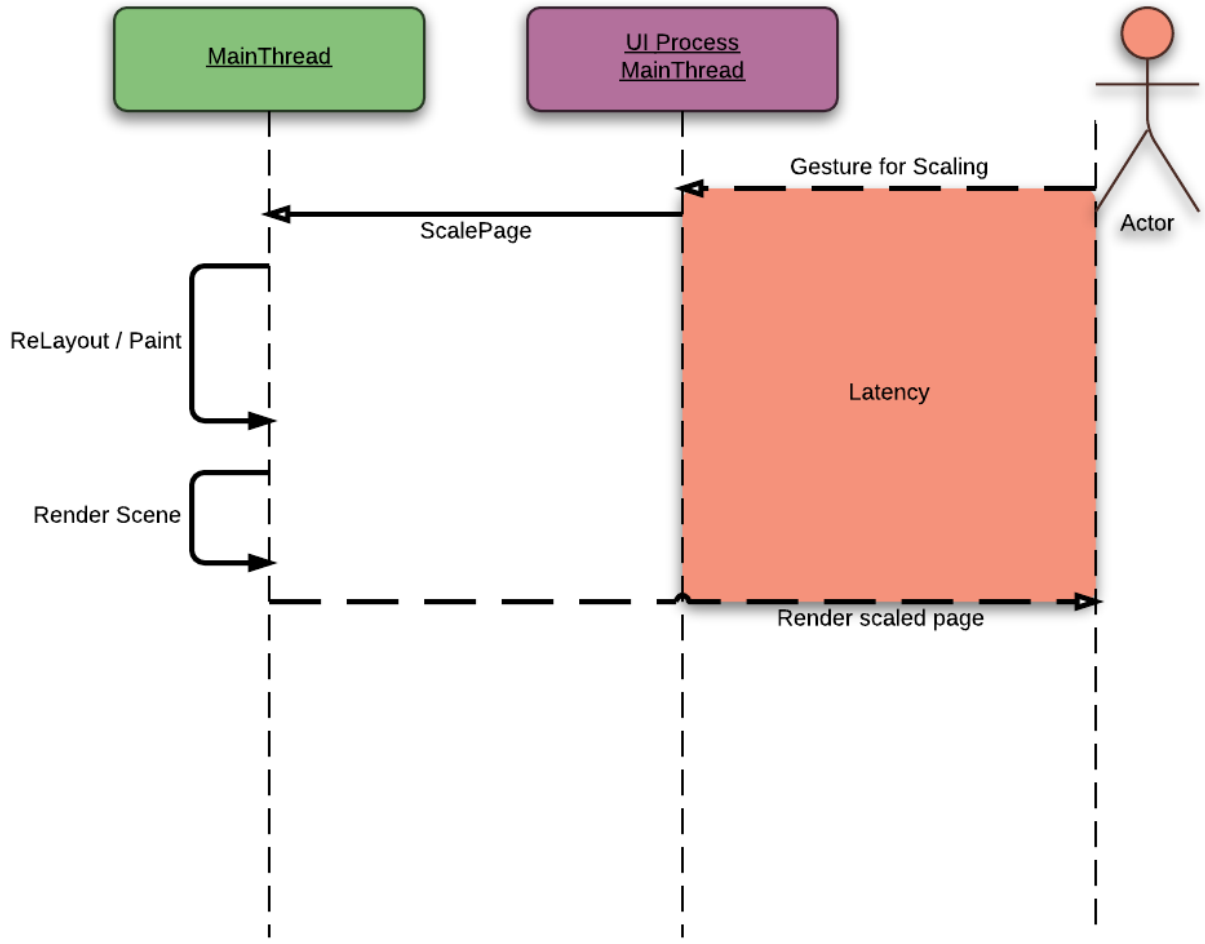


(http://blog.ryumiel.net/images/articles/Threaded_Compositor_Design_Update_Animation.png)

Scrolling and Scaling

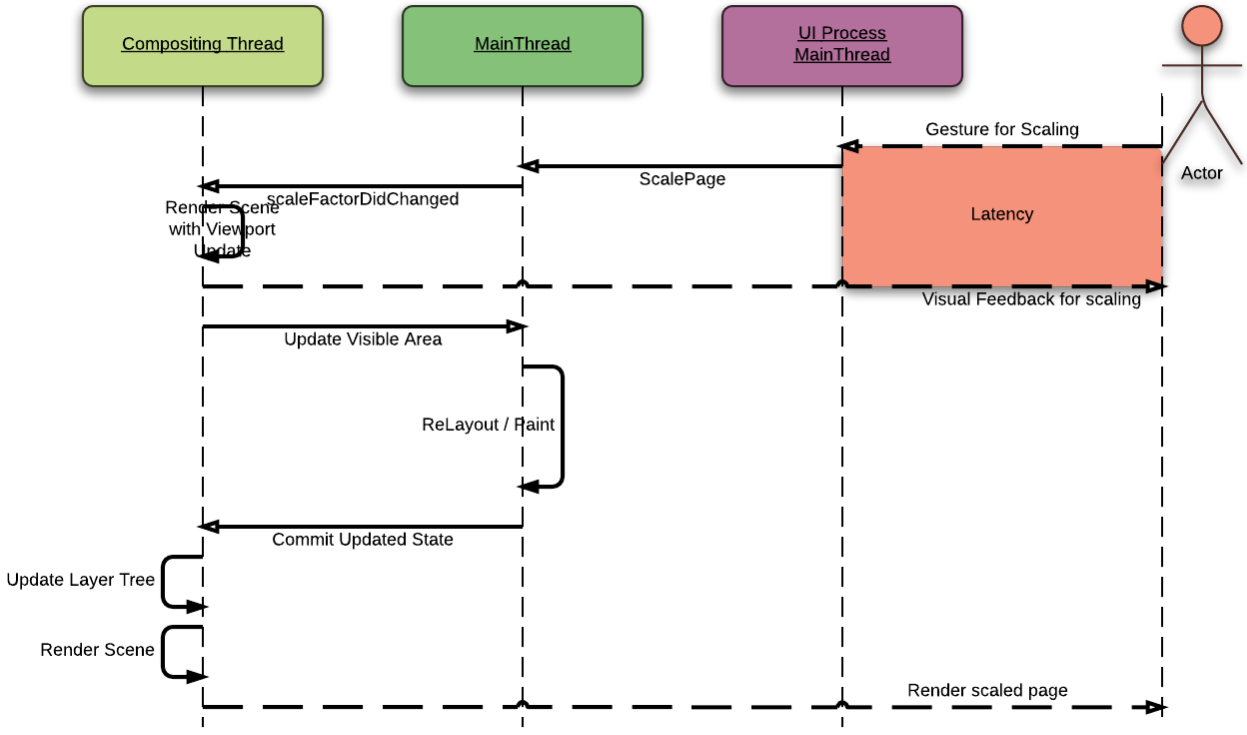
Scrolling and Scaling are also expensive job especially in the embedded device.

When we are scaling a web page, WebCore needs to update whole layout of its web page. As all of us knows, it is really expensive operation in the embedded platform.

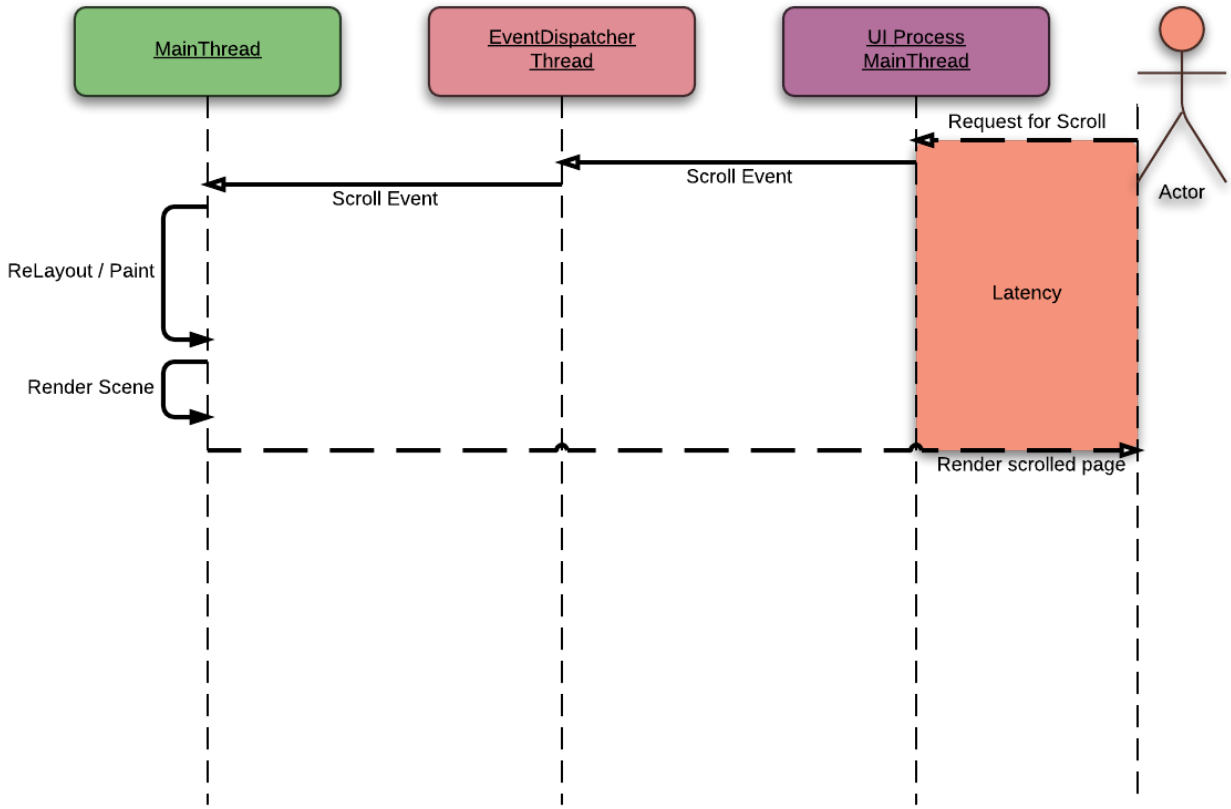


Let's assume that a user tries to scale a web page using a two finger gesture. The WebCore in the Web Process tries to produce scaled web page with a exact layout as soon as possible. However, if it is a embedded device, it can need more than 100 ms - 10 fps.

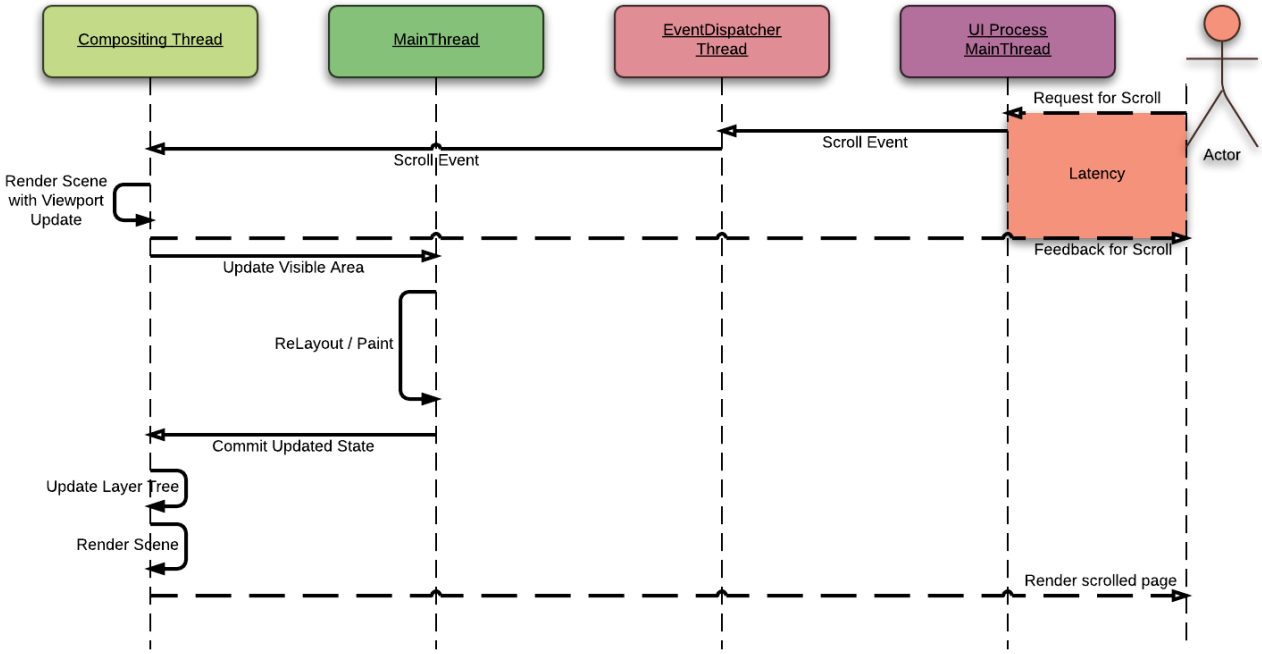
The Threaded Compositor renders a web page with a requested scale using its cached layer tree. By doing like that, we can give a immediate visual feed back to the user. And re-layouted web page will be updated later.



It is similar in the scrolling operation. When we are scrolling the web page, this operation doesn't need a huge re-layout operation. But it needs re-painting and bliting operation to update a web page.



Because the Threaded Compositor uses TILED_BACKING_STORE, it can render the scrolled web page with cached tiles with a minimal latency. During ThreadedCompositor renders a web page, the main-thread scrolls "actual" view. Whenever the main-thread finishes "actual" scroll, these changes are collected by CompositingCoordinator. And these changes are passed to the ThreadedCompositor to update the view.



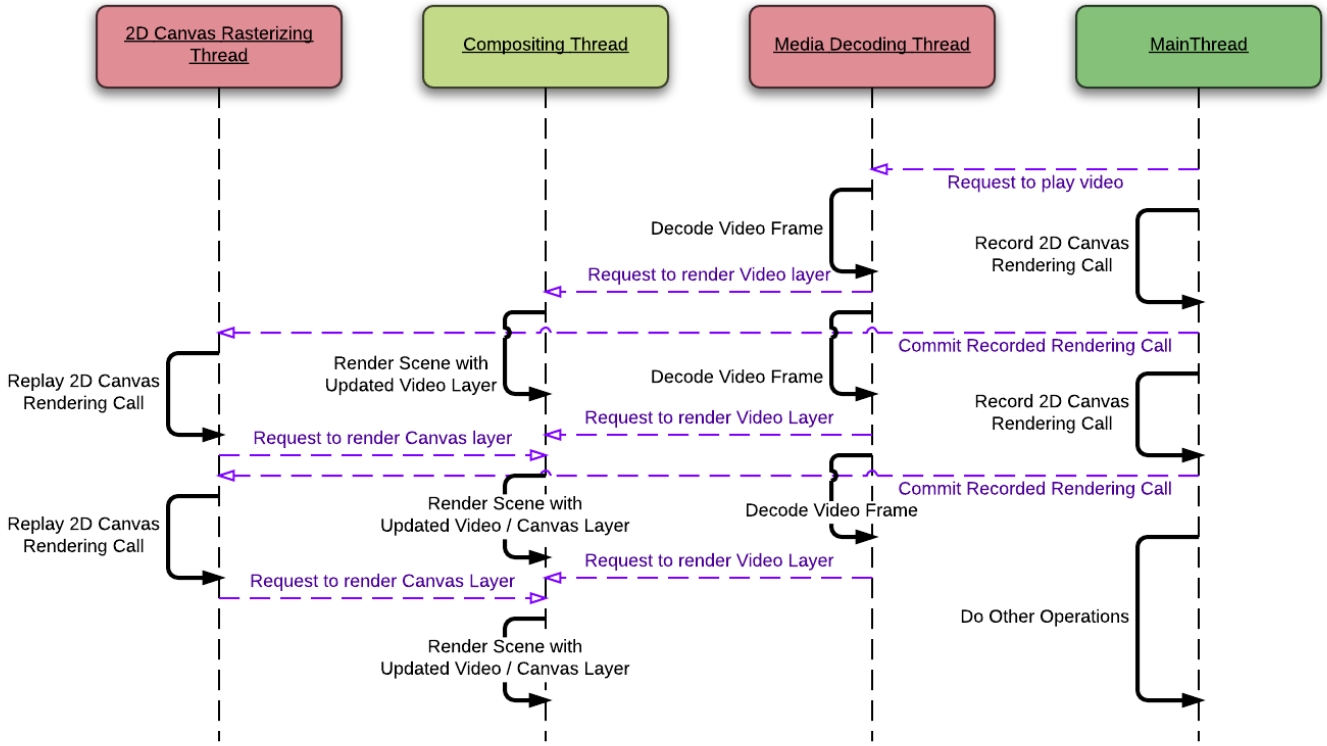
Unfortunately, these optimization is only supported when we are using fixed layout. To support this without fixed layout, we need to refactor TILED_BACKING_STORE in WebKit and implement proper overlay scrollbars in WebKitGTK+.

Compositing the Media and the Canvas

Not only CSS3 Animation and scaling but also the video and canvas can be accelerated by the Threaded Compositor. Because it is little bit out of scope of this post - Because it is not implemented for upstreaming - I'll only describe about it briefly here.

In the TextureMapper, Media (Plugins, Video element, …) and Canvas (WebGL, 2D Canvas) can be rendered by implementing TextureMapperPlatformLayer. Most important interface of the TextureMapperPlatformLayer is TextureMapperPlatformLayerClient::setPlatformLayerNeedsDisplay which requests the compositor to composite the its contents.

If the actual renderer of a Media and a Canvas element runs on off-the-main-thread, it is possible to bypass the main-thread entirely. The renderer can calls TextureMapperPlatformLayerClient::setPlatformLayerNeedsDisplay when it updates its result from its own thread. And compositing thread will composite the result without using the main-thread.



Also, if the target platform supports streams of texture (https://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_stream_consumer_gltexture.txt), we can avoid pipeline hazards when drawing video layers in modern mobile GPUs which uses tile based rendering.

Performance

This performance comparison was presented in Linux.Conf.AU 2013. It is based on pretty old implementation but it shows meaningful performance improvement compare to plain TextureMapper method. I could not regenerate this result using my current laptop because it is too fast to make stressed condition. I hope I can share updated the performance comparison using the embedded device soon.

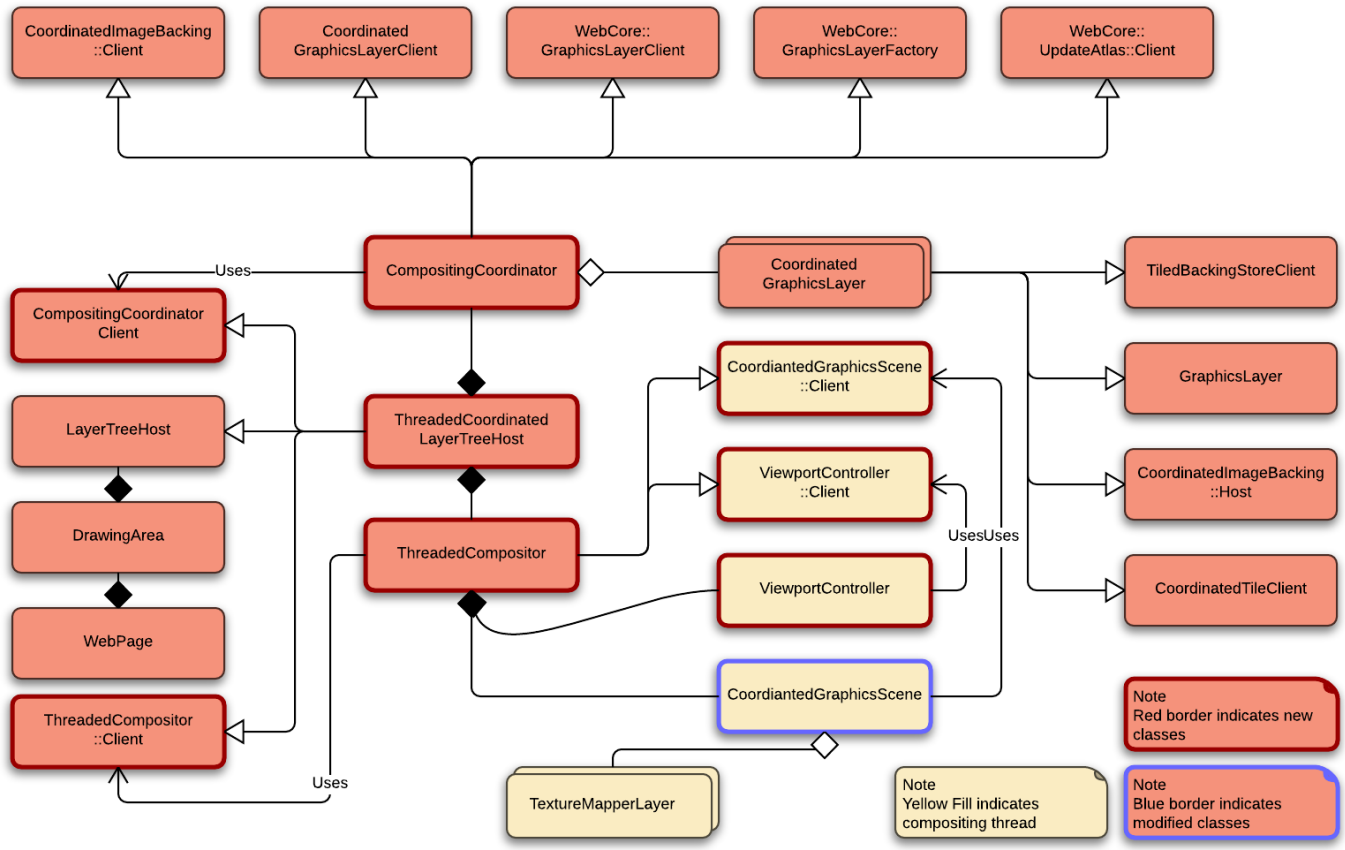
- Test Cases [*]
  - leaves n
    - Modified the famous accelerated compositing example, WebKit Leaves (https://www.webkit.org/blog-files/leaves/), to draw n leaves.
  - 3DCube n
    - n cubes rotating using CSS3 animation.
- Test Environment
  - Prototype of Threaded Compositor Implementation based on r134883
  - WebKit2 Gtk+ r140386
  - Intel Core i5-2400 3.10Ghz, Geforce GTS450, Ubuntu 12.04 x86_64

Tests	ThreadedCompositor (fps)	WebKit2 Gtk+ (fps)	Improved %
leaves500 (http://black.company100.com/test/TC/leaves500NoDelay/)	58.96	35.86	64.42
leaves1000 (http://black.company100.com/test/TC/leaves1000NoDelay/)	58.98	25.88	127.90
leaves2000 (http://black.company100.com/test/TC/leaves2000NoDelay/)	32.98	18.04	82.82
3DCube360 (http://black.company100.com/test/TC/3DCube360NoDelay/)	57.27	32.09	78.47
3DCube640 (http://black.company100.com/test/TC/3DCube640NoDelay/)	33.64	23.52	43.03

[*] These tests are made by Company 100 (http://company100.com)

Class Diagram

I made this diagram to help my understanding during the implementation, but It would be good to share to help others who are interested in it.



(http://blog.ryumiel.net/images/articles/Threaded_Compositor_Design_Threaded_Compositor.png)

- CompositingCoordinator

> A class takes the responsibility of managing compositing resources in the main-thread.

- ThreadedCompositor

> This class has a dedicate thread for compositing. It owns ViewportController and CoordinatedGraphicsScene to render scene on the actual surface.

- ThreadedCoordinatedLayerTreeHost

> A concrete class of LayerTreeHost and CompositingCoordinator::Client for CoordinatedGraphics. And it has ThreadedCompositor to use the Threaded Compositor.

- CoordinatedGraphicsScene

> A class which has a complete scene graph and rendering functionality. It synchronizes its own scene graph with a graphics layer tree in compositing coordinator.

- ThreadSafeCoordinatedSurface

> This class implements a surface using ImageBuffer as a backend to use in the Web Process.

- TextureMapperLayer

> It has actual properties to render a layer.

- ViewportController

> This class is responsible to handle scale factor and scrolling position in the compositing thread.

In the main-thread, all of the visual changes of each GraphicsLayer are passed to CompositingCoordinator via CoordinatedGraphicsLayer. And CompositingCoordinator collects state changes from the GraphicsLayer tree when it is requested to do so.

From this point, Threaded Compositor and Coordinated Graphics behaves differently.

In Coordinated Graphics, the collected state changes of the layer tree is encoded and passed to CoordinatedLayerTreeHostProxy in UIProcess. And CoordinatedGraphicsScene applies these state changes to its own TextureMapperLayer tree and renders these on the frame buffer.

But in Threaded Compositor, these states are passed to CoordinatedGraphicsScene which owned by ThreadedCompositor. ThreadedCompositor has its own RunLoop and thread, all of the actual compositing operations are executed in the dedicated compositing thread in Web Process.

Current Status

Most of re-factoring for Threaded Compositor in Coordinated Graphics side was done in the last year.

However, upstreaming of Threaded Compositor was delayed to various issues. I had to simplify the design (which was quite complicate at that time) and resolve various issues Before starting upstreaming process.

In this year, WebKitGTK+ decided to deprecate WebKit1. Because of that it was possible to make much less complicated design. Also, I fixed the Threaded Compositor not to break current behavior of WebKitGTK+ without fixed layout.

I'm happy that I can start upstreaming process of this implementation, from now on.

Things To Do

- Reduce memory uses when updating texture
  - RenderObjects draw its contents to UpdateAtlas to pass bitmaps to the compositing thread. However, The size of each UpdateAtlas is 4MB. It is quite big memory consumption in the embedded device. Still, I couldn't find a way to solve it without using proprietary GPU driver.
- Avoid pipeline hazards in the mobile GPU
  - Modern mobile GPUs uses tile-based [deferred] rendering. These GPUs can render multiple frames (~ 3 frames) concurrently to overcome its low performance. It is well working technique in the game industry which uses static textures.
  - In the WebKit, most of textures are dynamically generated. Whenever we are updating a texture using texSubImage2D, GPU would be stalled because it would be used in the previous rendering call.
  - To avoid this problem, chromium uses producer/consumer model of texture (http://src.chromium.org/viewvc/chrome/trunk/src/gpu/GLES2/extensions/CHROMIUM/CHROMIUM_texture_mailbox.txt)
  - We can create a wrapper for a texture which provides multiple buffering. In this way we can avoid the pipeline hazards.
- Reduce the maintenance burden
  - we need to re-factor Coordinated Graphics to share codes with Apple's UI SIDE COMPOSITING codes. Most of messages are duplicated. So it can be easily done by defining platform specific parts.
- Accelerate the performance of 2D Canvas element and Video element
  - It can be improved (relatively) easily as I described at Compositing the Media and the Canvas.
- Check the performance of Threaded Compositor in the embedded device
  - I hope I can share about it soon.
- Most of all, upstreaming!

Comments

comments powered by Disqus (http://disqus.com)

Social

linkedin (http://kr.linkedin.com/pub/gwang-yoon-hwang/23/45/b10)

github (http://github.com/ryumiel)

Tags

(http://blog.ryumiel.net/tags/)

AcceleratedCompositing (http://blog.ryumiel.net/tag/acceleratedcompositing/)

WebKitGTK+ (http://blog.ryumiel.net/tag/webkitgtk/)

WebKit (http://blog.ryumiel.net/tag/webkit/)

Igalia (http://blog.ryumiel.net/tag/igalia/)