# Coherent GT Performance Best Practices

1.4.0

# Contents

**1 Profiling and troubleshooting performance with Coherent GT 1**

1.1 Is an issue in the UI or the application? . . . . . 1
1.2 Asynchronous GT . . . . . 2
1.3 Performance audits . . . . . 2
1.4 Inspector overview . . . . . 2
1.5 How to find which operation in the UI is slow? . . . . . 4
1.6 Some of my JavaScript code is slow, what now? . . . . . 5
1.7 My JavaScript animations are slow, what to do? . . . . . 5
1.8 Seems my styles and/or layout is slow, what now? . . . . . 5
1.9 I have too many repaint / I have a huge paint, what now? . . . . . 6
1.10 How can I find if a particular element is causing a slow-down? . . . . . 6
1.10.1 Performance best practices . . . . . 7
1.11 JavaScript . . . . . 7
1.12 Drawing . . . . . 8
1.13 Elements . . . . . 8
1.14 Avoiding re-paints with layers . . . . . 9
1.15 Improving your JavaScript with Google Closure compiler . . . . . 11
1.16 Consider asynchronous mode . . . . . 12

# Chapter 1

# Profiling and troubleshooting performance with Coherent GT

Coherent GT includes powerful tools and APIs to help developers measure its performance impact on the application and eventually optimize the UI content. Coherent GT aims to occupy a maximum of 10% of the frame budget, which for a 60 FPS title equals to ∼1.6ms per-frame.

If you feel that Coherent GT is taking more time than the per-frame budget you've allotted to it, or simply want to squeeze more cycles out of it, this guide will show you what to look for and how to optimize your UI.

*Note* that the numbers in these guide are for reference. They depend on the platform and the architecture of the application you use Coherent GT in. Coherent GT is a product that improves constantly with every version and the performance profile changes accordingly.

Coherent Labs offers a *Developer access program* that directly connects users to experts from Coherent Labs. We can provide profiling, auditing and ideas in order to get the best out of Coherent GT. Please contact support or your account manager for further information.

## 1.1 Is an issue in the UI or the application?

The first step is to determine if a performance issue is due to the UI or something else in the application. The easiest way to see this is look for the Coherent GT performance warnings. In development builds (if not disabled in the ViewInfo or SystemSettings), Coherent GT will emit performance warnings when it detects that something is not performing as expected. The performance warnings are printed in the UI log or in Unreal Engine 4 are directly printed on-screen in the Unreal Editor.

Each performance warning will tell you where it is happening and give a clue where to look for when optimizing. Take for instance the warning related to JS execution times:

"Coherent GT: Performance warning on View 0 JS execution time is high (values is in ms). Threshold value: 0.75 Detected value: 1.6 URL: coui://MyTestUI/uiresources/hud.html [You can customize this warning]"

This means that there is something in the JS code that is taking too much time. You can use the Coherent GT Inspector to check what is executed in JavaScript. If there are no performance warnings, it is very unlikely that the issue is related to Coherent GT. The threshold values of the warnings are customizable, so you can tune them for your own frame-time budget. Note that the warnings don't cover the GPU time required to draw the UI. To profile GPU time please use advanced GPU debugging tools like the Visual Studio Graphics Analyser, NVidia NSight, AMD GPU Profiler, Intel GPA, Renderdoc etc.

## 1.2 Asynchronous GT

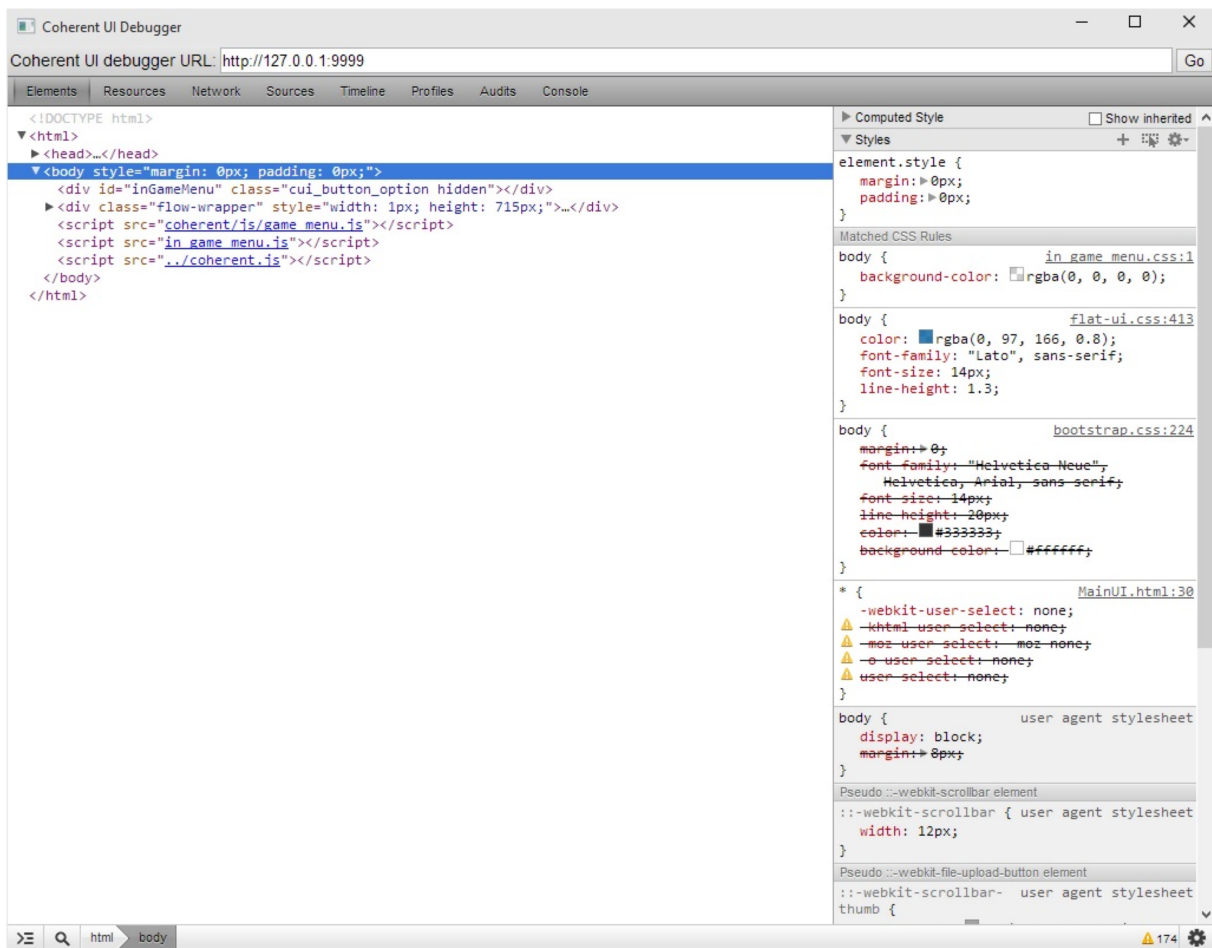
As of Coherent GT 1.2 you can run all JavaScript, style, layout and rendering command recording in a worker thread. This effectively removes all UI overhead from the main thread of the application. You should always strive to have good performance with the async API also. The Advance() and Layout() calls will stall if the previous calls haven't finished on the worker thread in order not to desync the UI and the application. If the UI work is more than the whole frame of the game, it will stall.

## 1.3 Performance audits

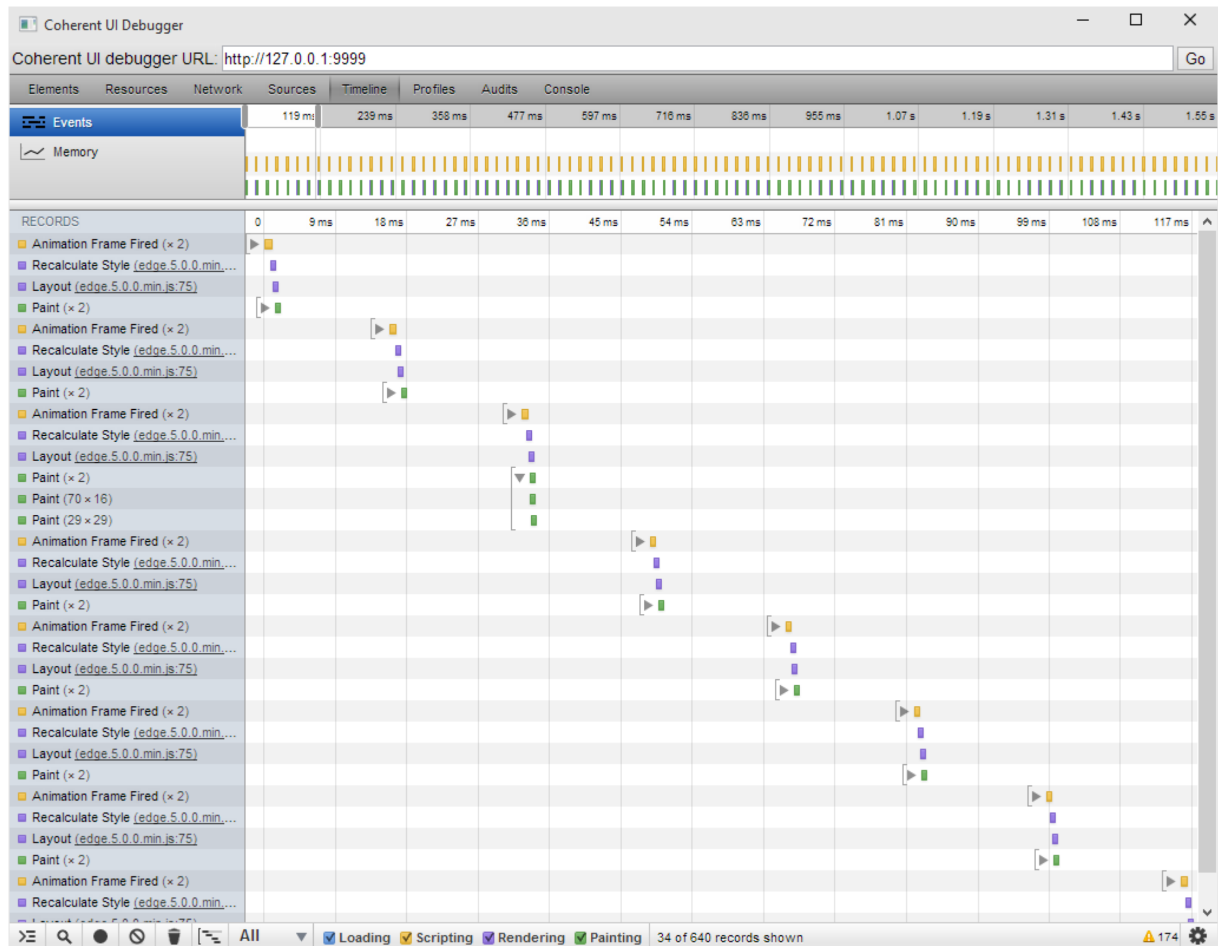
If you have determined that there is a GT operation that is taking too long, you can run an automated "Performance audit" that will help pinpointing sub-optimal elements or code. Coherent GT can automatically analyse your current UI and report what might be optimized. Please run the automated "Performance audit" and re-check the performance after you've followed the instructions provided by it. See the Rendering chapter in the main documentation file for information on how to run the audit. In the Unreal Engine 4 Editor just use the Coherent GT menu.

## 1.4 Inspector overview

When deeper information is needed when profiling, we can use the bundled Coherent GT Inspector. The Inspector is available in the package and can be used to connect to a live UI, debug UI, profile and check all elements within the DOM. The Inspector uses the same UI as the WebKit Inspector, which is a well-known tool to all web developers. On a guide how to start the Inspector, please refer to the “UIDebugging” chapter of the Documentation. In Unreal Engine 4, just use the Coherent GT Menu and click “Launch Inspector”.



This image shows the "Elements" tab of the Inspector. It can be used for live-editing all CSS properties. If you think that a bottleneck is rooted in style/layout or painting, you can delete all elements in the UI and see what impact that will have on frame-rate. Press "Ctrl-Z" to undo the delete and restore your UI.




The most important performance-related feature of the Inspector is the Timeline. It shows exact timings of all operations taking place within the UI. Just click the "Record" button (the grey dot in the bottom-left) and collect as much data as needed.

Important events include:

- Event - outside events like mouse over, clicks, keyboard events etc.
- Timer fired - when a timer in JS fires
- Request animation frame fired - what a requestAnimationFrame handler is called
- Recalculate style - a CSS style recalculation caused by some event or JavaScript code
- Layout - how much time it took to re-layout part of the page. The vent includes how many elements needed layout and how many DOM elements will be touched.
- Time Start/End - events triggered in native code via the Binding API
- Paint - parts of the View that get re-painted. Note that only the CPU time is recorded, not the actual time it took the GPU to perform the draw actions. Each event also highlights the parts of the screen that have been re-painted.

Coherent UI Debugger

Coherent UI debugger URL: http://127.0.0.1:9999 Go

Elements Resources Network Sources Timeline Profiles Audits Console

Profiles

CPU PROFILES

Profile 1

| Self | Total | Average | Calls | Function | |
|---|---|---|---|---|---|
| 97.61% | 100.00% | 97.61% | 1 | (program) | |
| 0.40% | 0.41% | 0.00% | 138 | ▶ set2DTransformRatio | TweenMax.min.js:15 |
| 0.24% | 0.65% | 0.00% | 1933 | ▶ handleTimer | edge.5.0.0.min.js:112 |
| 0.19% | 0.92% | 0.00% | 1638 | ▶ render | TweenMax.min.js:16 |
| 0.16% | 0.57% | 0.00% | 450 | ▶ setRatio | TweenMax.min.js:16 |
| 0.15% | 0.16% | 0.00% | 2773 | ▶ h | edge.5.0.0.min.js:49 |
| 0.12% | 0.24% | 0.00% | 278 | ▶ update | edge.5.0.0.min.js:109 |
| 0.11% | 0.21% | 0.00% | 1933 | ▶ v | edge.5.0.0.min.js:125 |
| 0.11% | 1.01% | 0.00% | 1932 | ▶ (anonymous function) | edge.5.0.0.min.js:64 |
| 0.11% | 1.29% | 0.00% | 693 | ▶ e | edge.5.0.0.min.js:43 |
| 0.10% | 0.32% | 0.00% | 279 | ▶ render | TimelineMax.js:785 |
| 0.10% | 0.10% | 0.00% | 1933 | ▶ k | edge.5.0.0.min.js:124 |
| 0.06% | 0.08% | 0.00% | 693 | ▶ s | edge.5.0.0.min.js:43 |
| 0.06% | 1.14% | 0.00% | 552 | ▶ (anonymous function) | edge.5.0.0.min.js:67 |
| 0.05% | 0.21% | 0.00% | 2773 | ▶ notifyObservers | edge.5.0.0.min.js:73 |
| 0.05% | 0.51% | 0.00% | 138 | ▶ render | TweenMax.min.js:14 |
| 0.04% | 0.25% | 0.00% | 1932 | ▶ _getTimeline | edge.5.0.0.min.js:135 |
| 0.04% | 0.04% | 0.00% | 276 | ▶ requestAnimationFrame | |
| 0.04% | 0.04% | 0.00% | 1934 | ▶ Date | |
| 0.03% | 1.40% | 0.00% | 138 | ▶ S | edge.5.0.0.min.js:90 |
| 0.03% | 0.99% | 0.00% | 138 | ▶ d | TweenMax.min.js:16 |
| 0.02% | 1.19% | 0.00% | 414 | ▶ J | edge.5.0.0.min.js:67 |
| 0.02% | 0.15% | 0.00% | 138 | ▶ ea | edge.5.0.0.min.js:67 |
| 0.02% | 0.02% | 0.00% | 693 | ▶ toString | |
| 0.02% | 0.94% | 0.00% | 138 | ▶ dispatchEvent | TweenMax.min.js:16 |
| 0.02% | 0.02% | 0.00% | 280 | ▶ sort | edge.5.0.0.min.js:106 |
| 0.01% | 0.01% | 0.00% | 1934 | ▶ getTime | |
| 0.01% | 1.07% | 0.00% | 138 | ▶ rR | edge.5.0.0.min.js:64 |
| 0.01% | 0.01% | 0.00% | 563 | ▶ hasOwnProperty | |
| 0.01% | 0.01% | 0.00% | 276 | ▶ cos | |
| 0.01% | 0.01% | 0.00% | 415 | ▶ min | |
| 0.01% | 0.01% | 0.00% | 276 | ▶ sin | |
| 0.01% | 0.01% | 0.00% | 138 | ▶ rC | edge.5.0.0.min.js:64 |
| 0.00% | 0.00% | 0.00% | 279 | ▶ max | |
| 0.00% | 0.00% | 0.00% | 139 | ▶ now | |
| 0.00% | 0.01% | 0.00% | 1 | ▶ done2 | MainUI_edgeActions.js:59 |
| 0.00% | 0.00% | 0.00% | 137 | ▶ round | |
| 0.00% | 0.01% | 0.00% | 56 | ▶ _enabled | TweenMax.min.js:16 |
| 0.00% | 0.00% | 0.00% | 45 | ▶ isActive | TweenMax.min.js:16 |
| 0.00% | 0.00% | 0.00% | 18 | ▶ V | TweenMax.min.js:16 |
| 0.00% | 0.00% | 0.00% | 19 | ▶ splice | |
| 0.00% | 0.00% | 0.00% | 1 | ▶ C | TweenMax.min.js:16 |

Heavy (Bottom Up) % 174

The "Profiles" tab allows to run JavaScript profiling that will show how much time individual JS functions take. It can be used to quickly find bottlenecks in JS code.

## 1.5 How to find which operation in the UI is slow?

Start the Inspector and capture in the Timeline some frames where you experience sub-optimal performance. Now in the Timeline you can see how much time each event took. There are several usual outcomes from this operation.

- Too many event calls from native code. Calling JavaScript from C++/C#/Blueprints incurs some overhead. It's better to pack multiple pieces of information together and send it in one event to the UI.
- Too much/slow JavaScript code. Use the "Profiles" to check JS code. Check also the next section where we discuss more techniques to make JS code faster.
- Style and Layout is taking a lot of time. Please check the section that in detail explains how to optimize styling and layout.
- Too many/slow paints. Having too many tiny paints or large paints that cover many elements on-screen could be detrimental to performance. Please consult the section about paints that explains how to effectively use layers to eliminate superfluous paints.
- Parse HTML calls. Doing in-line HTML parses (caused by JS) is very slow. You can see the line in code that causes a parse and eliminate it. Some third-party libraries like jQuery tend to do this and their use is discouraged.
- Paints with "Image decode". When a new image is required, it has to be loaded and decoded. This can cause a noticeable stall in the UI. Consider pre-loading images required by a page.

## 1.6 Some of my JavaScript code is slow, what now?

There are several reasons that can cause JS code to under-perform. As a general rule JS code is slower than native code. Try to execute minimal JS to drive the UI and move calculations and logic to native code. Run a profile of the JS code from the "Profiles" tab. This could give you an idea where a bottleneck in code lies.

- Too many TriggerEvent (20+) calls can cause degraded performance. There is a fixed cost to call JS from native code. Pack data and execute less calls to JS. The same applies for calling native code from JS.
- JavaScript causes in-line re-layouts. This is easily recognizable as 'layout' events inside some JS execution.- There are methods in JavaSctipt that cause immediate re-layouts of the page in order to get a property. This is extremely inefficient and can cause severe performance drops. jQuery is a library that often does this in its "css()", "show()", "hide()" and other methods. We discourage using jQuery at all.
- Slow-downs in Edge Animation code. If you use Adobe Edge Animation, you might see slow-downs in its JS code. The major issue is that Edge animates every thing with JavaScript. If you have too many simultaneous animations (10+), the JS performance can be severely affected. Edge *continues* to animate even Stages/- Symbols that are not visible. If you have many dynamic symbols, delete the one you don't use, don't just hide them. Consider moving simple animations to CSS animations or the Coherent Animation library.
- Many calls to jQuery/Angular. Libraries like jQuery were created for web browsers where the performance requirements are much lower compared to a game UI. Avoid using jQuery or Angular. In particular avoid all methods that require in-line re-layouts or html parsing. Never use "html()", "css()", "show()", "hide()". Coherent GT *will* emit warnings if it detects calls to those methods.

## 1.7 My JavaScript animations are slow, what to do?

Some libraries animate though JavaScript code and that can lead to performance degradations. Avoid at all costs jQuery animations.

If you are using Edge Animate, avoid having too many animations running simultaneously (10+). It's best to move animations to the Coherent Animation library. You can still use your symbols and stages.

The best solution is moving to CSS-based animations (http://www.w3schools.com/css/css3_-animations.asp). CSS animations are fully evaluated in C++ and are an order of magnitude faster than JavaScript-based ones. Especially for simple and repeating animations, move them to CSS. You can do this keeping your Edge Animate work-flow and continue to use your stages and symbols.

## 1.8 Seems my styles and/or layout is slow, what now?

First check that a maximum of only one layout and style re-calculation is done per-frame. Some JS libraries force in-line re-layouts that degrade performance.

Try to limit the subset of elements that have to be re-laid out. There are several simple techniques that allow you to do that:

- Place elements absolutely
- Move elements with "translate" instead of "top/left"
- Scale elements with "scale" instead of "width/height".

Limit the DOM elements count. The more elements you have, the slower a full layout will take. as a general rule 150-200 DOM elements are usually more than enough for very complex UI. To see the DOM elements count, run this line in the Inspector console 'document.getElementsByTagName("*")'.

## 1.9 I have too many repaint / I have a huge paint, what now?

Use the Timeline to see which parts of the screen are re-painted. You can see what is getting redrawn each frame by using the Inspector, clicking the *gear* (low-right) icon and checking *Show Paint Rectangles*. Some elements might cause re-draws of parts of the screen by moving over them although they are the only thing that changes.

The best way to reduce paints is to effectively use "Layers". You can promote elements to layers. When an element changes, it will re-draw only it's layer and not other elements that are under or over it. Moving and scaling layers with "-webkit-transform" is essentially a 'free' operation. It will not cause re-layouts or re-paints. The layers will just get composited in a different way.

Imagine you have a heavy HUD interface with many elements and a crosshair that moves across all the screen. Without layers, when the crosshair moves over other elements of the View they'll have to be redrawn, even if they are completely static. This can be wasteful and reduce performance. An effective solution is to move the crosshair in its own layer. You can do this by adding a dummy transform: `-webkit-transform: rotateX(0deg)`. Now when the crosshair moves over other elements, they will not be redrawn and performance will be improved.

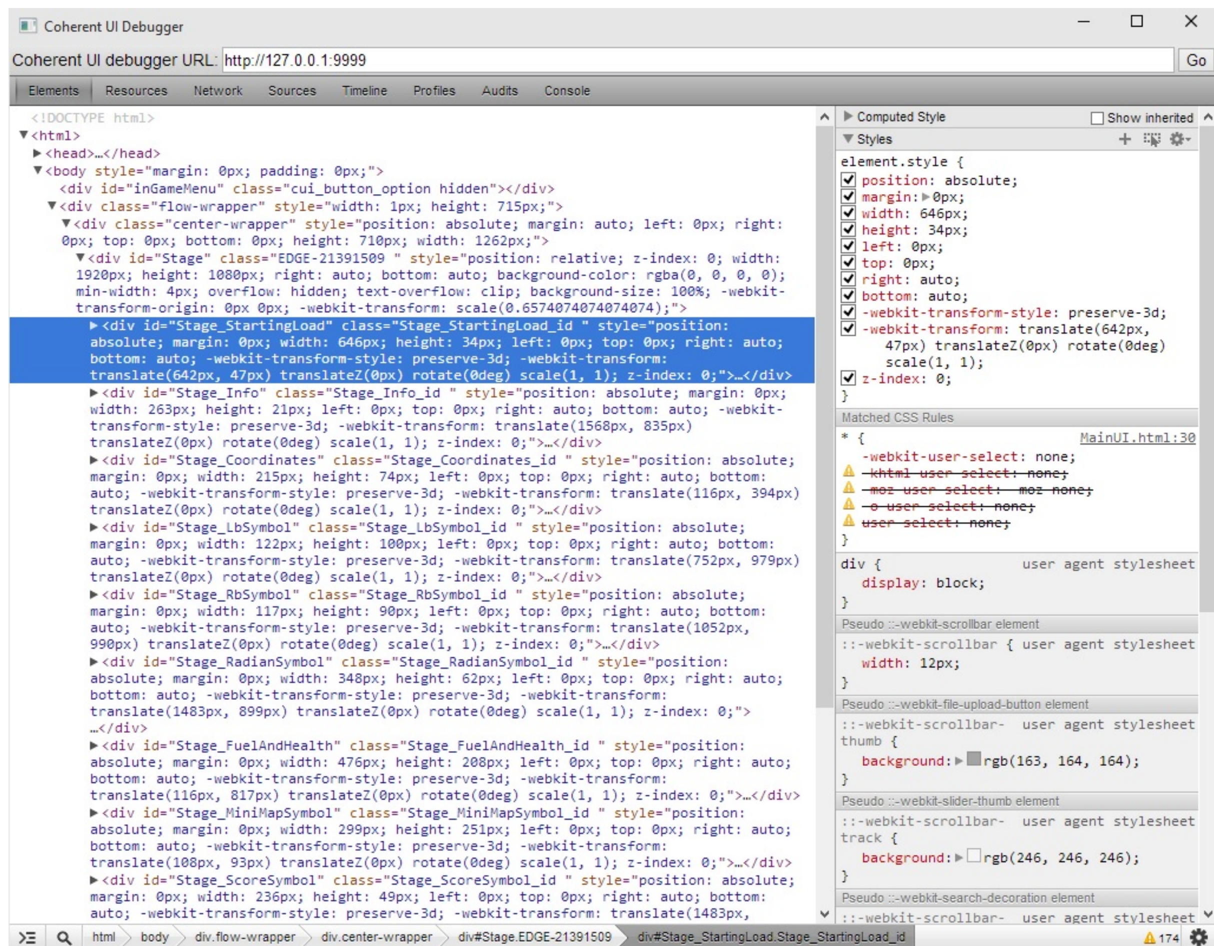
Please consult the section on layers below in this document.

*Note* that many or expensive paints will also affect the GPU time of the rendering as they'll often cause more GPU commands to happen.

## 1.10 How can I find if a particular element is causing a slow-down?

There is a very simple way to check if a particular element or a hierarchy of elements is slow. Look at your frame-time, connect the Inspector and simply delete this element from the "Elements" tab. Use "Ctrl-Z" to undo the operation and get the element back. If you see a significant change in frame-time, there probably is something sub-optimal in the element.

Removing the element might reduce paints or layouts, in this case try to move it to a layer.




In other cases the element itself (or a descendant) might have CSS properties that are costly to evaluate or render. Elements with shadows are slower to render than elements without. Try removing single CSS properties in the Inspector (in the right pane after you've selected the element) and look for an improvement in frame-time. Please consult the following sections for details on properties that are more costly than others.

### 1.10.1 Performance best practices

Coherent GT allows for easy performance testing through the built-in Inspector support. Use the Timeline feature to check the performance of different parts of your UI - JavaScript execution time, layout times and re-paints.

Coherent GT will emit performance warnings in the file log. You can disable those warnings in the `System-Settings` or per-View in the `ViewInfo` structure. You can also change the threshold values that will cause a warning to emit. Warnings help identifying quickly during development that some change has caused a performance drop. You should disable them in shipped products. When you receive a performance warning, you can use the Inspector to profile eventual inefficiencies in the interface and use the information provided in this document to optimize the UI.

In addition to the warnings, Coherent GT can also audit HTML pages and check for suboptimal CSS and HTML usage. See the Rendering chapter in the main documentation file for information how to use them.

## 1.11 JavaScript

- Avoid having too much logic in JavaScript. Although the GT JS interpreter is fast, code ran through it is still slower than native code. Prefer JS for simple logic and code as much as possible in native.
- Avoid doing too many calls to/from JS in a single frame. Crossing the boundary between JS and C++ involves

some overhead. Prefer packing more information in bigger events.

- Avoid setting state in JS with redundant state. For example if you set the health of a player, don't send an event each frame with the health but just send it when the health has actually changes in native code. This will save redundant JS executions and probably some repaints.
- Prefer CSS3 animations over JavaScript ones when possible. CSS3 animations are evaluated in C++ and no JS code has to be executed for them.
- When animating with JavaScript prefer the Coherent Animation library. It is has very powerful features and great performance.
- When using Adobe Edge Animate avoid having too many animations active at the same time. Adobe are constantly improving the performance of their timeline animations but they are still slower than the ones created with the Coherent Animation library. You can improve performance by moving animations to the Coherent Animation library. Edge supports seamlessly adding external code/animations.
- Profile third-party libraries that you use. Some of them generate code that is not optimal.
- Avoid using jQuery's `show()`, `hide()`, `css()` functions - use directly the JS element's methods.
- jQuery is slow for DOM access. On events that happen very often prefer using the built-in JS methods `getElementBy*`.
- Avoid using jQuery's `html()` and `text()` functions - use directly `textContent`.
- Cache selected elements in functions or members. Avoid re-selecting elements.
- Extensively use the Inspector to profile JavaScript code. You can use the Timeline and the *Profile*->*Collect JavaScript CPU Profile* facilities.
- You can set the frame-rate at which `requestAnimationFrame` is called. Usually this can be set to a value that is much less than the frame-rate of the game. Calling the animation callbacks too often will waste C-PU cycles. You can control the animation frame-rate through the `ViewInfo::AnimationFrameDefer` member. Set the defer value to the largest possible that looks correct.

## 1.12 Drawing

- A key factor to Coherent GT's high performance is the fact that it doesn't always redraw elements. You can see what is getting redrawn each frame by calling the `View::ShowPaintRects` method or using the Inspector, clicking the *gear* (low-right) icon and checking *Show Paint Rectangles*.
- Avoid moving or changing styles of element that are not drawn. In some cases this might cause redundant re-paints of parts of the UI. Use the *Show Paint Rectangles* debug feature to make sure that only the correct parts are redrawn.
- Avoid changes that cause layout recalculations - prefer absolute positioned elements when possible.

## 1.13 Elements

Coherent GT draws the UI using the GPU. It emits Draw Calls to the driver that perform the actual rasterization. Minimizing the Draw Call count is important in order to improve CPU performance. Coherent GT will try to minimize the necessary Draw Calls. Coherent GT can draw both raster images and vector elements. Both have advantages and drawbacks. Raster images are usually faster to draw and require less draw calls but consume memory and are not resolution independent.

**Vector elements** (HTML elements like divs, SVGs) are resolution-independent and will look great under any scaling. Depending on the element they might be more costly to draw. Complex SVG images might require many Draw Calls.

Note that for elements that reside in parts of the screen that don't get re-painted the following guidelines might not apply as they will be drawn just once. Elements that are never redrawn are effectively free per-frame. However if

another element moves on-top of a static one - both will get re-drawn. Use the *Show Paint Rectangles* to inspect the parts of the screen that get re-painted in your UI.

- Some elements are more costly to draw than others and will require more Draw Calls.
- Avoid tiny `divs`. Creating small UI elements from `divs` is an overkill. Prefer rasters images or SVG. Especially if animated these tiny `divs` will put pressure on the layout engine.
- Vector elements with stroked outlines increase draw calls. Each stroke adds a draw-call.
- Prefer sharp edges on vector elements instead of rounded ones. Rounded corners increase draw calls. Consider raster images.
- Shadows and blurs are very costly. They impose a strict ordering that breaks Draw Call batching and require complex shaders for producing the effects.

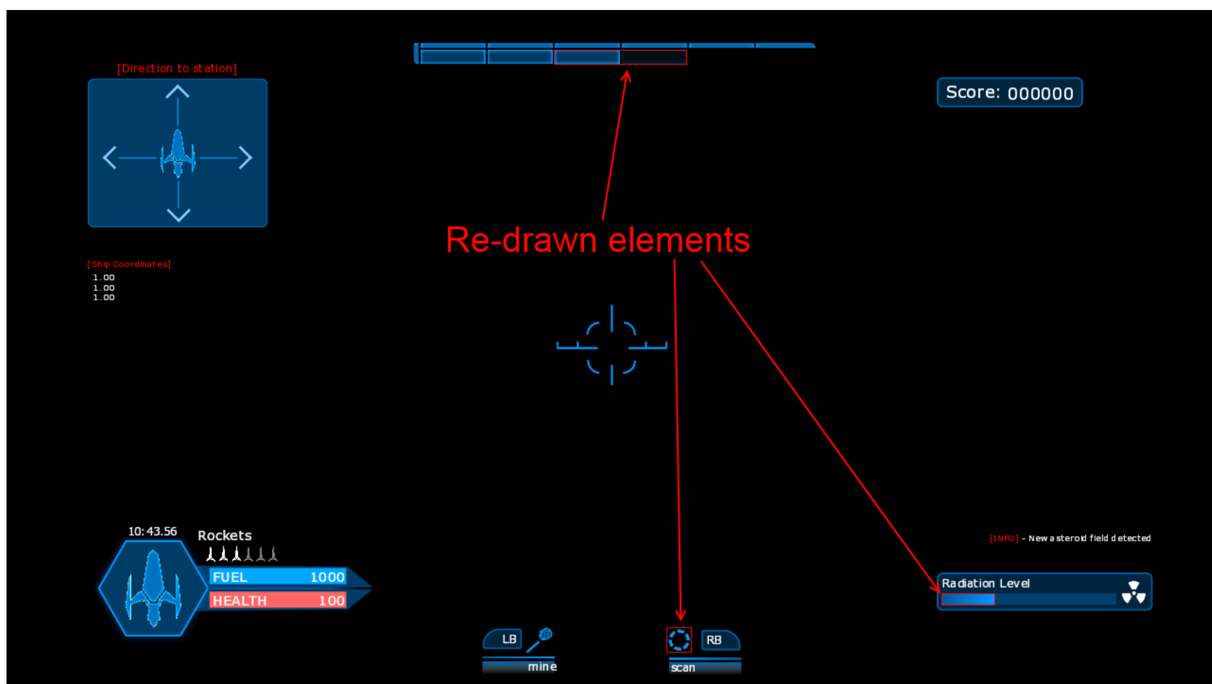



Figure 1.1: Visualizing redrawn parts in a HUD

## 1.14 Avoiding re-paints with layers

Coherent GT can move certain parts of the page in their own layers. When in the page there are layers we say that the page is in "compositing" mode. Layers can be used to avoid re-paints of heavy elements. Every Element that has some sort of CSS3 3D transform gets its own layer. You can also force Elements to become layered with a dummy transform: `-webkit-transform: rotateX(0deg)`.

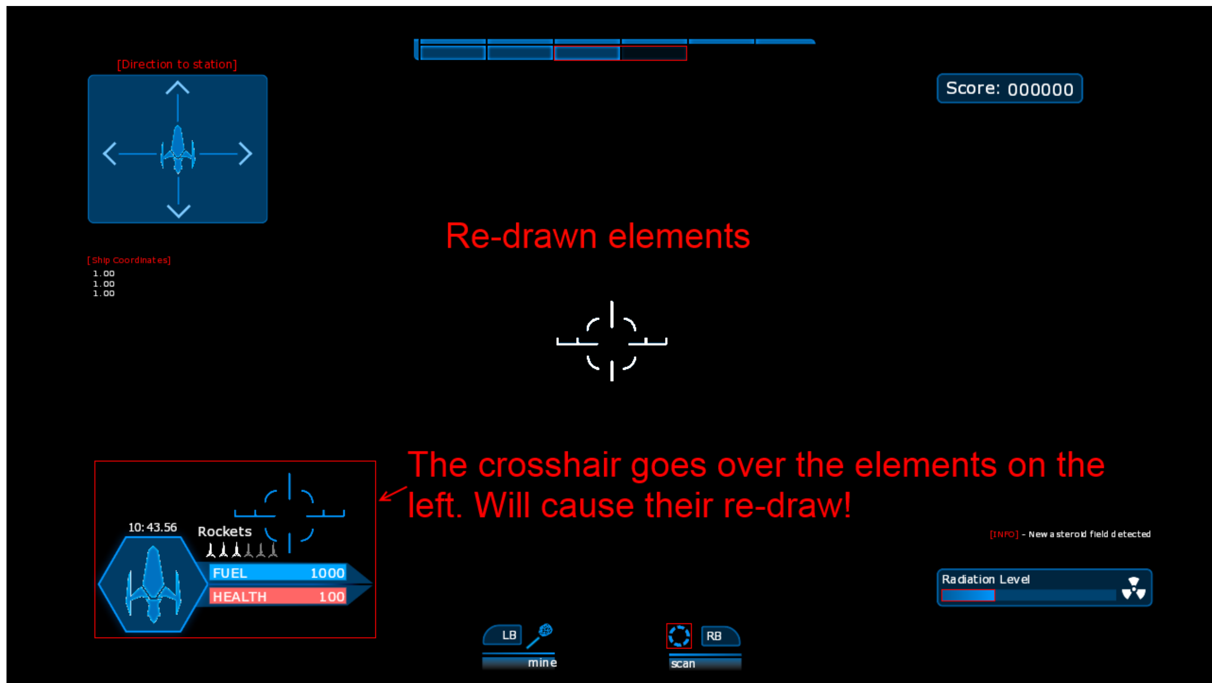



Figure 1.2: The animated crosshair causes redraws of HUD elements

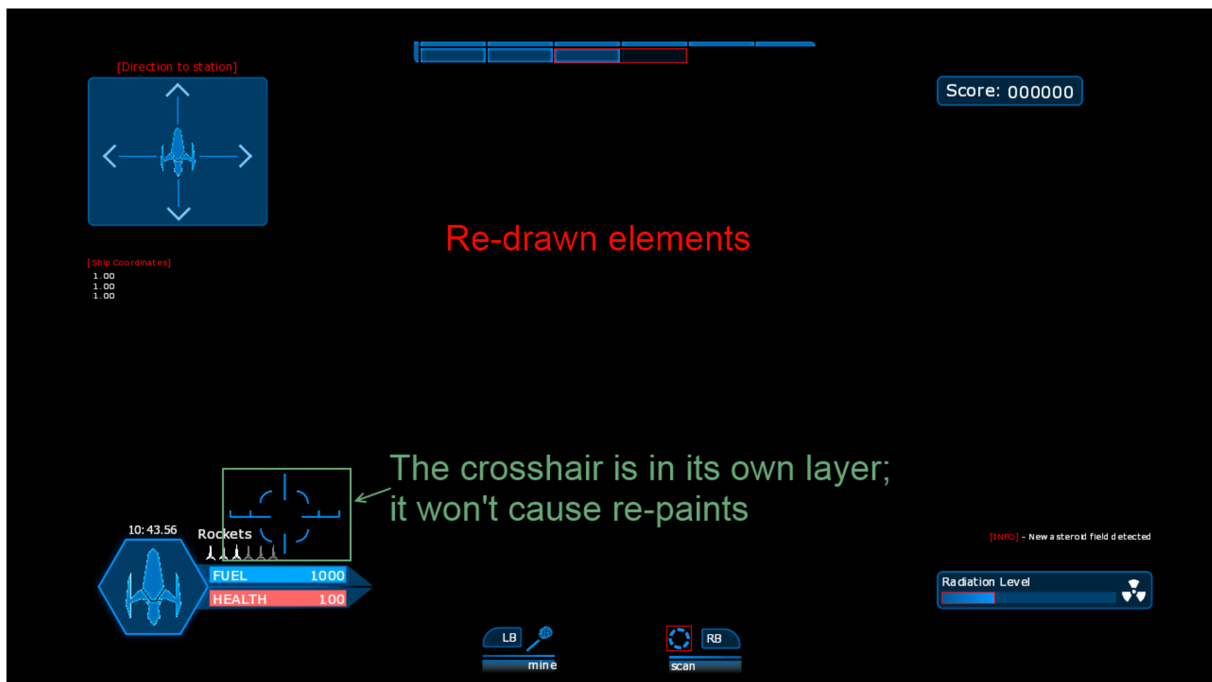



Figure 1.3: The crosshair has its own layer. Now it doesn't cause re-paints to elements it intersects.

**NB: In Coherent GT `-webkit-transform: translateZ(0px)` will *NOT* cause a new layer to be created. Many tools like Adobe Edge Animate abuse this technique and create layers for every element with a dummy translateZ. In the end this hurts performance. Users are encouraged to create their layers with alternative methods like "-webkit-transform: rotateX(0deg)" if needed.**

Imagine you have a heavy HUD interface with many elements and a crosshair that moves across all the screen. Without layers, when the crosshair moves over other elements of the View they'll have to be redrawn, even if they

are completely static. This can be wasteful and reduce performance. An effective solution is to move the crosshair in its own layer. You can do this by adding a dummy transform: `-webkit-transform: rotateX(0deg)`. Now when the crosshair moves over other elements, they will not be redrawn and performance will be improved.

- To make a layer on an element that has no effective transform, use `-webkit-transform: rotateX(0deg)`. Do not use the popular `-webkit-transform: translateZ(0px)`.
- Move to their own layers elements that tend to move over other elements on-screen and thus cause unnecessary re-paints of other elements under/over them.
- Each layer consumes additional GPU memory and has to be re-composited (drawn on the right place on-screen) each frame. Having too many layers can effectively reduce performance. You can use the Inspector to see what parts of the screen are redrawn each frame and where your layers are.
- You can visualize all layers in the page. In the Inspector click the *gear* (low-right) icon and check *Show composited layer borders*. When composited layers are enabled you'll also see a number over each layer - it shows the re-paint count of the layer.
- Using layers to avoid redraw is a known technique in HTML development. Additional information can be found here: http://www.html5rocks.com/en/tutorials/speed/high-performance-animations/.
- Avoid re-sizing layers. When a layer is re-sized, a new texture has to be created and all the content of the layer re-drawn. It is better to have a bigger layer and move elements within it, instead of having a smaller layer that is tightly packed around some elements and gets constantly re-sized as they move.
- The Root layer (usually the document body with elements that are not themselves in layers) will clip any element that goes outside of it. It will always have the size of the viewport. This is *not* true for other layers. As they might be scaled/transformed/rotated, there is no way of knowing their final size and screen position until the final composition. These layers can have arbitrary sizes and elements that go beyond the screen will still be drawn if they are part of a layer. Developers should be careful not to position elements outside the viewport in layers that might degrade performance. Such elements should be disabled or their positions clamped.

## 1.15 Improving your JavaScript with Google Closure compiler

**NB: Not all JavaScript code will benefit from the Closure compiler.**

Google Closure compiler (https://developers.google.com/closure/compiler/) is a tool for making JavaScript download and run faster. You can use it to reduce the size of your JavaScript, check for errors and improve the overall performance of your UI.

There are three optimization levels:

- The WHITESPACE_ONLY compilation level - removes comments from your code and also removes line breaks, unnecessary spaces, extraneous punctuation (such as parentheses and semicolons), and other white-space.
- The SIMPLE_OPTIMIZATIONS compilation level - performs the same white-space and comment removal as WHITESPACE_ONLY, but it also performs optimizations within expressions and functions, including renaming local variables and function parameters to shorter names. Renaming variables to shorter names makes code significantly smaller. Because the SIMPLE_OPTIMIZATIONS level renames only symbols that are local to functions, it does not interfere with the interaction between the compiled JavaScript and other JavaScript.
- The ADVANCED_OPTIMIZATIONS compilation level - performs the same transformations as SIMPLE_OPTIMIZATIONS, but adds a variety of more aggressive global transformations to achieve the highest compression of all three levels. The ADVANCED_OPTIMIZATIONS level compresses JavaScript well beyond what is possible with other tools.

Make sure to check the error tab as it can reveal all sorts of programming errors that are not parse errors such as: using undeclared variable, unreachable code, re-assigning constants, etc. To improve your code you can also use code quality tools such as Google Closure Linter, jsLint and jsHint.

## 1.16 Consider asynchronous mode

Coherent GT's asynchronous mode defers most calls to the library to another thread. This makes your UI is run for with almost no overhead to your main loop. You need to make sure you understand the consequences of multithreading as this can introduce race conditions. Refer to the DetailedGuides/AsyncMode section of the main docs for more details.