

Coherent GT Performance Best Practices

2.3.0

Contents

| | | |
|----------|--|-----------|
| 1 | Performance guide | 1 |
| 1.1 | Profiling and troubleshooting performance with Coherent GT | 1 |
| 1.1.1 | Is the issue in the UI or the application? | 2 |
| 1.1.2 | Performance audits | 2 |
| 1.1.3 | GPU Memory | 2 |
| 1.1.4 | Debugger overview | 2 |
| 1.1.5 | How to find which operation in the UI is slow? | 5 |
| 1.1.6 | Some of my JavaScript code is slow, what now? | 6 |
| 1.1.7 | My JavaScript animations are slow, what to do? | 6 |
| 1.1.8 | Seems my styles and/or layout is slow, what now? | 6 |
| 1.1.9 | I have too many repaint / I have a huge paint, what now? | 7 |
| 1.1.10 | How can I find if a particular element is causing a slow-down? | 7 |
| 1.2 | Performance best practices | 8 |
| 1.2.1 | JavaScript | 8 |
| 1.2.2 | Drawing | 9 |
| 1.2.3 | Elements | 10 |
| 1.2.4 | Rendering caches | 10 |
| 1.2.5 | Avoiding re-paints with layers | 11 |
| 2 | Appending and removing elements | 15 |
| 3 | On-screen Performance Info | 17 |
| 4 | Hiding/Showing Elements | 19 |
| 5 | Iteration tips | 21 |
| 6 | Layers and how to use them | 23 |
| 7 | Transparent animation - Sprite sheet vs Video | 25 |

Chapter 1

Performance guide

Asynchronous GT

Coherent GT's asynchronous mode defers most calls to the library to another thread. This makes your UI run 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 [this page](#) for more details.

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.

Disable additional styles

Since GT 1.8.5 the default styles used in pages are split in two categories - a "core" subset that is always included and an "additional" one that contains scrollbar styles. The inclusion of "additional" styles is controlled through the `Coherent::UIGT::ViewInfo::EnableAdditionalDefaultStyles` option. The additional styles incur a performance penalty as they apply scrollbar styles on all elements in the page. We advise disabling those styles and manually styling scrollbars in CSS *only* on elements that really will need them.

Do not do:

```
::-webkit-scrollbar { // applied on all elements - BAD
    width: 12px
}
```

Instead do:

```
.MyScrolledElement::-webkit-scrollbar { // applied only on elements with specific class - GOOD
    width: 12px
}
```

In the cases where you don't have control over the page - like when showing content off the Internet, you can still enable the additional styles and scrollbars through `EnableAdditionalDefaultStyles`.

1.1 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 $\sim 1.6\text{ms}$ 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.

1.1.1 Is the 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 Debugger 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.

1.1.2 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 Page Auditing chapter in the main documentation file for information on how to run the audit.

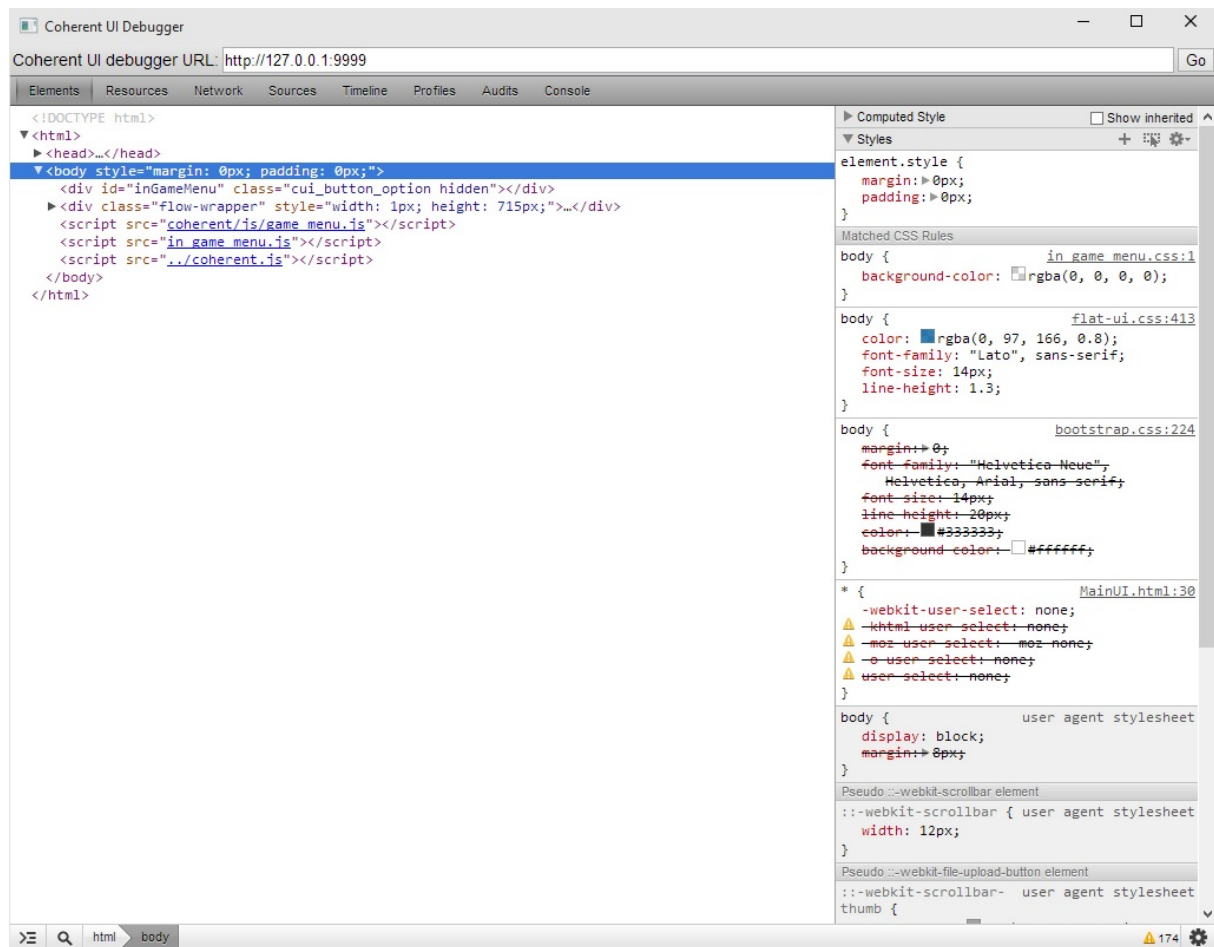
Note: if you are using UE4 the Auditor can be launched via *Coherent GT -> Launch Performance Auditor* in the editor.

1.1.3 GPU Memory

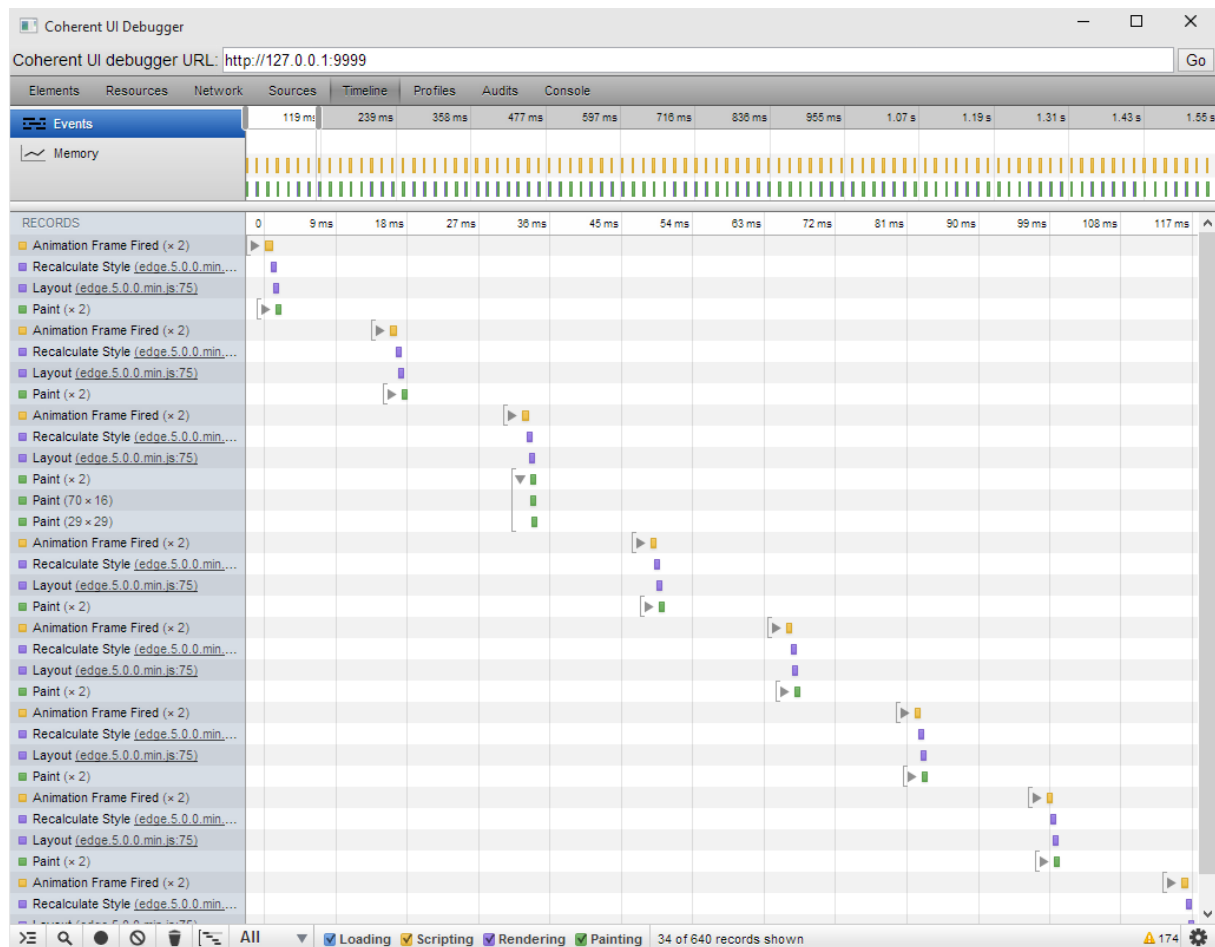
Coherent GT supports naming GPU texture resources with debug information. The `Coherent::UIGT::SystemSettings::SetRenderingResourcesDebugNames` flag will force GT to send to the renderer backend names of GPU resources. In the supplied DirectX 11 backend, the `SetPrivateData` method is used to annotate the debug names and they will be visible in tools like the Visual Studio Graphics Debugger and PIX. Developers can extend the functionality by changing the implementation of the `RendererBackend::SetDebugName` family of methods.

1.1.4 Debugger overview

When deeper information is needed when profiling, we can use the bundled Coherent GT Debugger. The Debugger 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 Debugger 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 Debugger, please refer to the "UIDebugging" chapter of the Documentation. In Unreal Engine 4, just use the Coherent GT Menu and click "Launch Debugger".



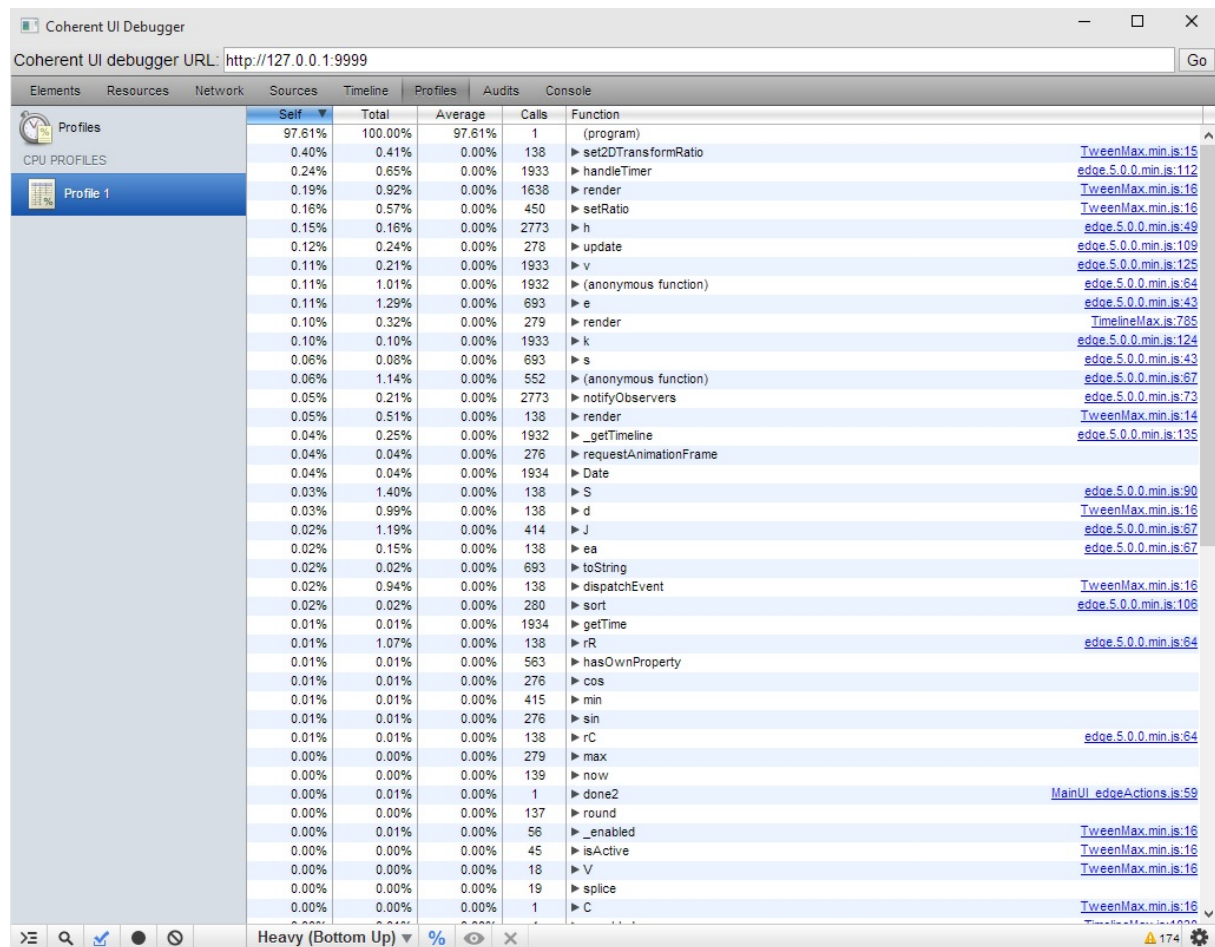
This image shows the "Elements" tab of the Debugger. It can be used for live-editing all CSS properties. If you think that a bottleneck is rooted in style/layout or painting, you can delete all elements in the UI and see what impact that will have on frame-rate. Press "Ctrl-Z" to undo the delete and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



The most important performance-related feature of the Debugger 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 event 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.



| Self | Total | Average | Calls | Function |
|--------|---------|---------|-------|-------------------------|
| 97.61% | 100.00% | 97.61% | 1 | (program) |
| 0.40% | 0.41% | 0.00% | 138 | ▶ set2DTransformRatio |
| 0.24% | 0.65% | 0.00% | 1933 | ▶ handleTimer |
| 0.19% | 0.92% | 0.00% | 1638 | ▶ render |
| 0.16% | 0.57% | 0.00% | 450 | ▶ setRatio |
| 0.15% | 0.16% | 0.00% | 2773 | ▶ h |
| 0.12% | 0.24% | 0.00% | 278 | ▶ update |
| 0.11% | 0.21% | 0.00% | 1933 | ▶ v |
| 0.11% | 1.01% | 0.00% | 1932 | ▶ (anonymous function) |
| 0.11% | 1.29% | 0.00% | 693 | ▶ e |
| 0.10% | 0.32% | 0.00% | 279 | ▶ render |
| 0.10% | 0.10% | 0.00% | 1933 | ▶ k |
| 0.08% | 0.08% | 0.00% | 693 | ▶ s |
| 0.08% | 1.14% | 0.00% | 552 | ▶ (anonymous function) |
| 0.05% | 0.21% | 0.00% | 2773 | ▶ notifyObservers |
| 0.05% | 0.51% | 0.00% | 138 | ▶ render |
| 0.04% | 0.25% | 0.00% | 1932 | ▶ _getTimeline |
| 0.04% | 0.04% | 0.00% | 276 | ▶ requestAnimationFrame |
| 0.04% | 0.04% | 0.00% | 1934 | ▶ Date |
| 0.03% | 1.40% | 0.00% | 138 | ▶ S |
| 0.03% | 0.99% | 0.00% | 138 | ▶ d |
| 0.02% | 1.19% | 0.00% | 414 | ▶ J |
| 0.02% | 0.15% | 0.00% | 138 | ▶ ea |
| 0.02% | 0.02% | 0.00% | 693 | ▶ toString |
| 0.02% | 0.94% | 0.00% | 138 | ▶ dispatchEvent |
| 0.02% | 0.02% | 0.00% | 280 | ▶ sort |
| 0.01% | 0.01% | 0.00% | 1934 | ▶ getTime |
| 0.01% | 1.07% | 0.00% | 138 | ▶ rR |
| 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 |
| 0.00% | 0.00% | 0.00% | 279 | ▶ max |
| 0.00% | 0.00% | 0.00% | 139 | ▶ now |
| 0.00% | 0.01% | 0.00% | 1 | ▶ done2 |
| 0.00% | 0.00% | 0.00% | 137 | ▶ round |
| 0.00% | 0.01% | 0.00% | 56 | ▶ _enabled |
| 0.00% | 0.00% | 0.00% | 45 | ▶ isActive |
| 0.00% | 0.00% | 0.00% | 18 | ▶ V |
| 0.00% | 0.00% | 0.00% | 19 | ▶ splice |
| 0.00% | 0.00% | 0.00% | 1 | ▶ C |

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.

1.1.5 How to find which operation in the UI is slow?

Start the Debugger 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.

1.1.6 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 JavaScript 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 ones you don't use, don't just hide them. Consider moving simple animations to CSS animations.
- 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.

1.1.7 My JavaScript animations are slow, what to do?

Some libraries animate through 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+).

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.

1.1.8 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 Debugger console 'document.getElementsByTagName("*')'.

1.1.9 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 Debugger, clicking the *gear* (bottom-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 its 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.

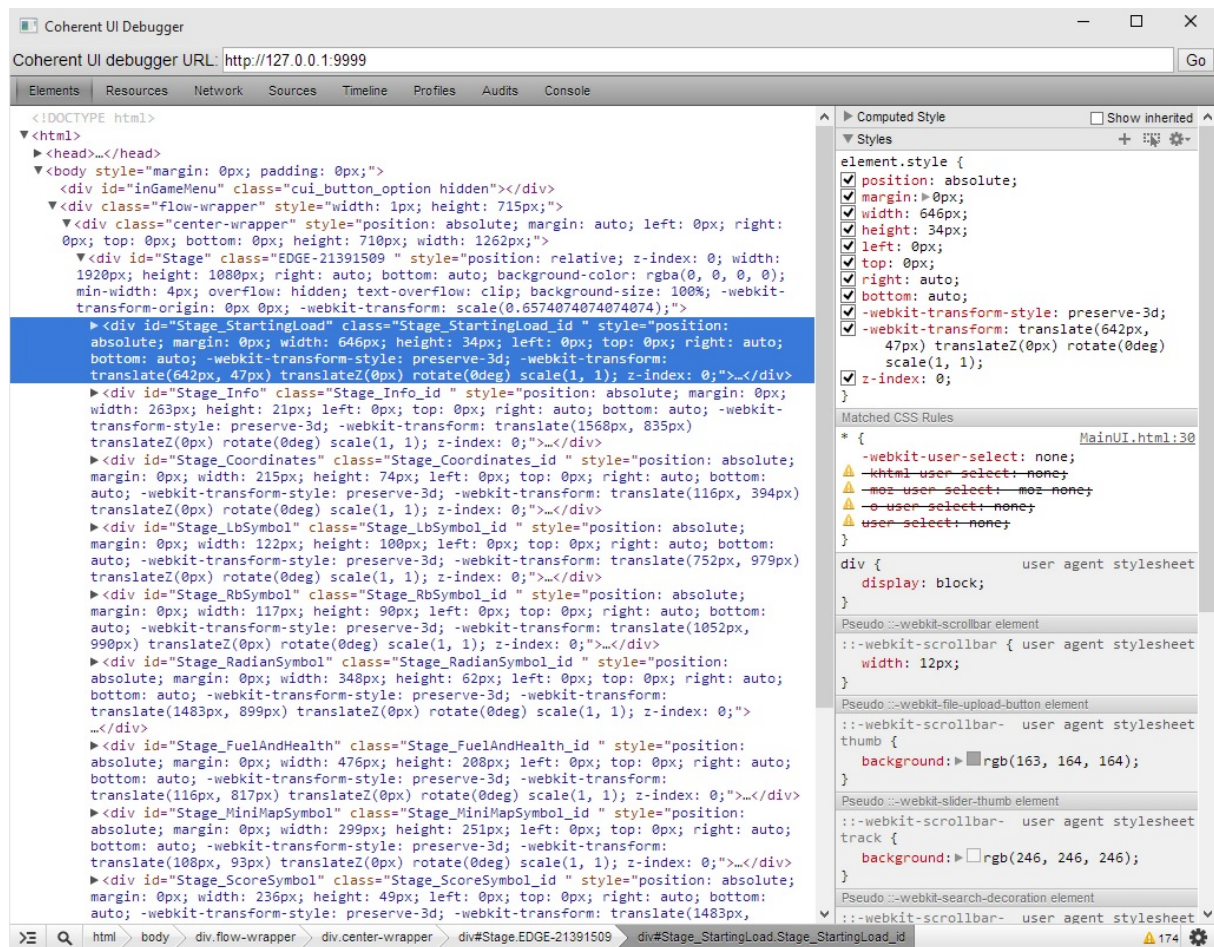
Please consult the section on layers below in this document.

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.

1.1.10 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 Debugger 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 Debugger (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.

1.2 Performance best practices

Coherent GT allows for easy performance testing through the built-in Debugger 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 `Coherent::UITG::SystemSettings` or per-View in the `Coherent::UITG::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 Debugger 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.

1.2.1 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 using "Element.style" to change the style of an element whenever possible. This syntax modifies the

"inline" style of the element and prevents the library from caching and sharing styles between elements. Prefer adding/removing CSS classes when possible.

- 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.
- Prefer the Coherent Editor to other HTML editing tools. The Coherent Editor has been designed with performance in mind.
- 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 less efficient than CSS 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 Debugger 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 CPU cycles. You can control the animation frame-rate through the `Coherent::UIGT::ViewInfo::← AnimationFrameDefer` member. Set the defer value to the largest possible that looks correct.
- Avoid using "Element.getBoundingClientRect()" as it can cause re-styling in the JavaScript call-stack.
- In CSS avoid using complex selectors like ":nth-child()". The selectors are significantly more costly to evaluate and disable some internal optimizations when calculating styles. You can always substitute them with manually set CSS classes.

1.2.2 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 `Coherent::UIGT::View::ShowPaintRects` method or using the Debugger, 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.
- Prefer hiding elements that you don't want to draw. Setting "opacity:0;" is NOT a good way of hiding elements as they will still be laid-out and styles. Prefer "visibility:none" and when possible - even better "display:none". Elements with "display:none" have a much smaller performance impact during styling and layout compared to visible elements.

1.2.3 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.

1.2.4 Rendering caches

Coherent GT employs several caches to accelerate the rendering of the pages. Currently there are 5 caches whose size and content can be controlled by the user:

- Font cache - all loaded fonts are kept in-memory. You can clear the font cache with `Coherent::UIGT::UISystem::ClearFontCache()`. If a font needs to be re-used, it'll be re-loaded. Use this in case you transition from a state where you know some fonts will not be used again. Note that invoking `Coherent::UIGT::UISystem::ClearFontCache()` will also destroy all the text atlas textures through the graphics backend. Glyphs that need to be shown after that will be rasterized again in a new texture.

The following 4 caches are rendering-oriented and their current state can be queried and modified with the `Coherent::UIGT::ViewRenderer::GetCacheStats`, `Coherent::UIGT::ViewRenderer::SetCacheStats`; `Coherent::UIGT::ViewRenderer::ClearCaches` methods.

- Shadows cache - blur effects are costly, so Coherent GT caches textures with already created blurs (shadows).
- Effects cache - some costly effects are cached in textures and re-used when possible.
- Filters cache - CSS 3 filtered images are cached up-to a certain size.
- Paths cache - Tessellated paths are cached in order to save CPU time on tessellations when identical paths are used multiple times.

It might be beneficial to profile and see if some cache is becoming a bottleneck. Usually due to the progressive rendering of the UI there shouldn't be issues with the default sizes, but for instance having dozens of special effects (shadows, blurs, gradients) that are animated *simultaneously* might require larger caches in a frame sequence.

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 raster images or SVG. Especially if animated these tiny divs will put pressure on the layout engine.
 - Vector elements with stroked outlines increase draw calls.
 - Prefer sharp edges on vector elements when clipping instead of rounded ones. Rounded corners increase area to draw but incur a performance penalty when used for clipping. Clip with alpha images when possible.
 - Shadows and blurs are costly. They impose a strict ordering that breaks Draw Call batching and require complex shaders for producing the effects.
-

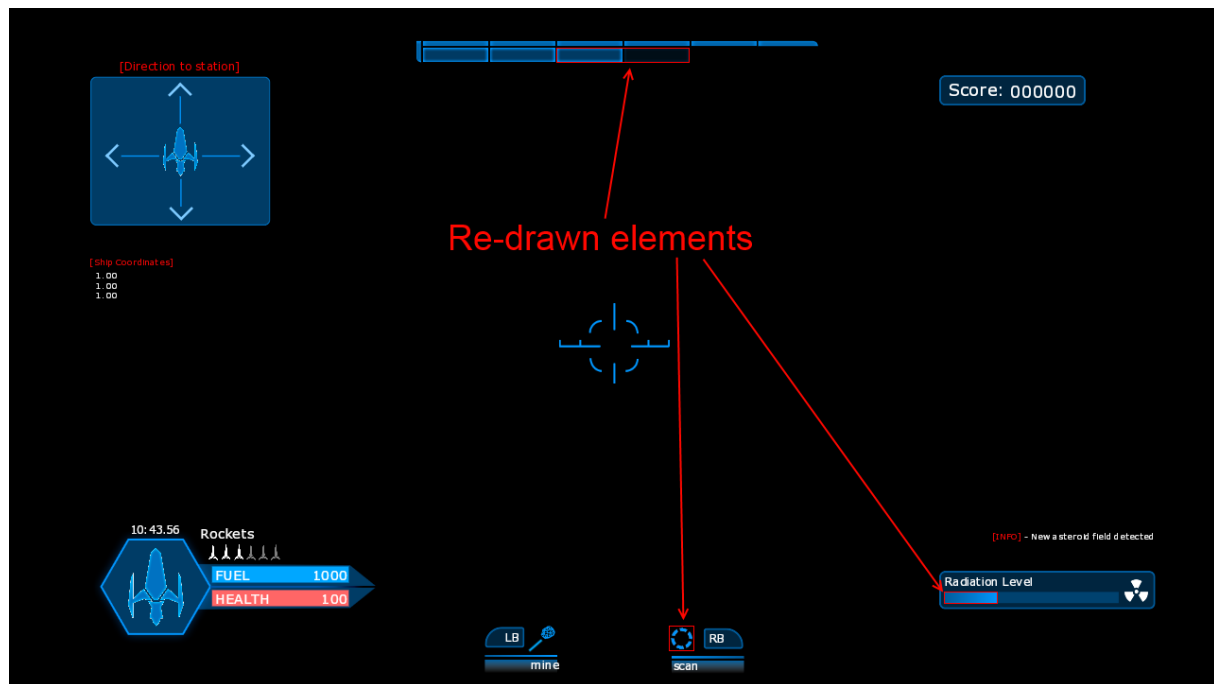


Figure 1.1: Visualizing redrawn parts in a HUD

1.2.5 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)`.

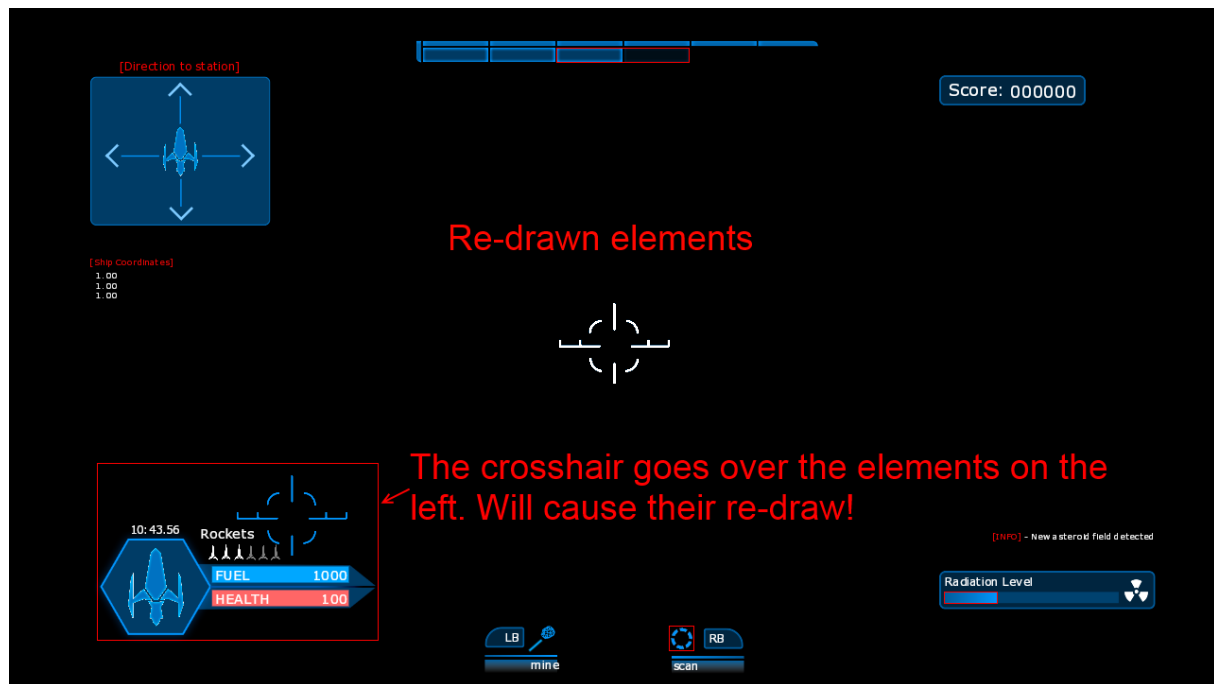


Figure 1.2: The animated crosshair causes redraws of HUD elements

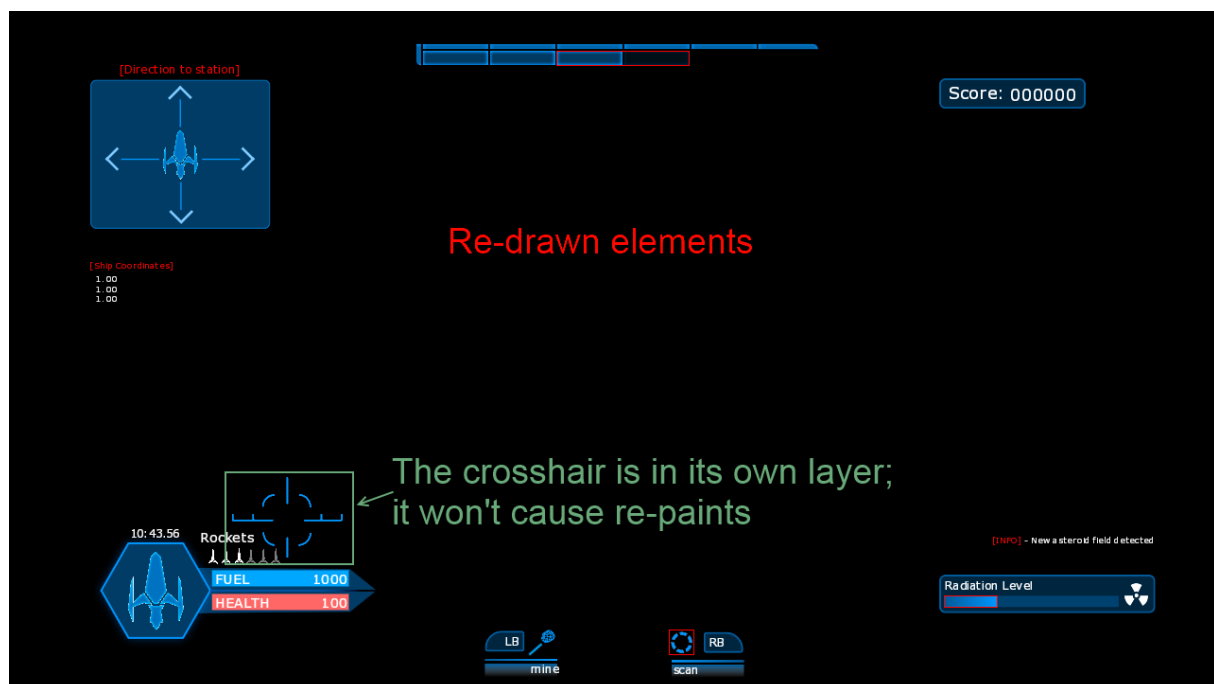


Figure 1.3: The crosshair has its own layer. Now it doesn't cause re-paints to elements it intersects.

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 Debugger 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 Debugger 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.
-

Chapter 2

Appending and removing elements

Some tips and techniques on how to minimize the performance impact when appending children using `.appendChild`. Appending many children all at once may cause longer frames, but by using `.requestAnimationFrame` this can be avoided. There is no predefined number of how many elements can be appended at once - profiling is highly recommended for each case in each scene.

Factors:

- How many elements will be appended?
- How many nested elements the appended element has?
- What styles are applied - box shadows, filters, transforms?
- Is the UI running in-game or not?

Appending children without `.requestAnimationFrame`:

Visually this may be the fastest way to create and show elements on the scene but this does not come without a cost. Creating many elements at once can cause frame bottleneck which is not a desired effect. If those elements have box shadows or any filters, the frame will be prolonged. Also when too many elements are created with images which load for the first time in the scene, those images will visualize at the end of the frame. For some images you may even consider preloading them in advance (Loading UI textures in advance (preload textures)).

In this example 20 elements are requested to be appended in one call. Each of them contains 21 more elements inside. This brings a total of 440 elements.

You can see from the Timeline that this operation takes more than 4 milliseconds.

Appending children with rAF:

This is the lightest way to create and append children with minimal performance impact. It doesn't matter how many elements should be created. Basically the `window.requestAnimationFrame` method is responsible for calling a function before the next Paint. The downside is that if many elements are requested to be appended with rAF, their visualization on the screen may be noticeable. To counter this effect, more than one element can be requested to be appended on a single call.

Same 20 elements are appended, but each one is created separately.

Frame-by-frame this is the lightest way when creating and displaying elements on the screen.

Appending children with rAF on chunks

Depending on the case, this technique can entirely remove the noticeable effect when creating a list of elements with rAF. The number of elements created at each `.requestAnimationFrame` should be adapted according to the weight of the scene.

In this example rAF is used, but instead of creating 1 element on each request, 3 are created.

You can see that the time taken to append and visualize the created elements on the screen is shorter than if appended one by one on each call.

Summary - appending children

The lightest and recommended way to create and append children is to split the workload over several frames using `.requestAnimationFrame`. Appending them in one call can lead to unnecessary and unwanted prolonged frame.

Note

When creating elements, always remember to change/update the content inside that element before appending it. The order should be: create -> update -> append. If the order is create -> append -> update there can be unnecessary Layouts, Paints and recalculating of styles.

Removing elements (no rAF)

This section of the guide about removing elements emphasizes on the case where many elements are removed. Removing elements is a much lighter process than creating them.

Removing many children at once is doing less work than when using `.requestAnimationFrame` for each element. One or a few times there is a need for a Layout and Paints, depending if the elements are removed in a single call or on portions using rAF.

Single call example with the same 440 element.

Removing elements with `.removeChild` (rAF)

As opposed to appending elements with `.requestAnimationFrame`, removing them this way is heavier. Creating and visualizing an element on the screen is always heavier than removing it. So each time an element is removed from the list, an update to the Layout is needed. Paint is also done each time although if these elements are layers, the Paint is minimal (layers will be discussed later in this guide).

Removing elements with rAF on chunks

This is the middle ground. In extreme case - if many elements need to be removed, they can be removed on portions using rAF. Lets say 50 elements need to be removed, but removing them at once can lead to a longer frame. Removing them by 5 or 10 will distribute the load in separate frames.

Summary - removing children

It is perfectly acceptable to remove children in one go. The catch is if too many elements need to be removed, partitioning the removal on portions will be better. "Too many" is defined by the weight of the scene. Profiling is the key to define the correct number.

Note

Do not empty a list of elements with `container.innerHTML = ""`, with `.textContent` or with `Node.innerText`. Although they do the job, this is not the proper way to remove children and these methods won't gain any performance. This is not their intended purpose.

Chapter 3

On-screen Performance Info

GT has a built-in profiler that displays information about how much time GT spends in its major subsystems such as scripting and rendering.

Enable on-screen performance metrics in the GTPlayer from the panel at the top or by pressing CTRL + P. To enable it programatically, call `Coherent::UIGT::View::EnableOnScreenPerformanceInfo`. You'll be displayed with several counters displaying various information. The counters are split in CPU and GPU information and their descriptions can be found below.

Main thread counters

1. System advance - the time the system took to advance internal timers (including CSS animation timers), download and process resources (CSS / JS / image / font files) and execute JS scripts. Equivalent to the time spent in `Coherent::UIGT::UISystem::Advance`. This timer is affected by the number and complexity of CSS animations and the number and size of the resources to be processed as well as whether they are local or downloaded via HTTP(S).
2. JS timers - the time the view took to execute any pending JS timers caused by calls to `requestAnimationFrame`, `setTimeout` or `setInterval`. Equivalent to the time spent in `Coherent::UIGT::View::ExecuteJSTimers`.
3. Styling and layout - the time the view took to compute what styles affect each element and where to place the element in accordance to the HTML standard. Equivalent to the time spent in `Coherent::UIGT::View::UpdateStylesAndLayout`. This timer is affected by the number HTML elements that have changed since last frame and by the number and complexity of all loaded CSS rules.
4. Record rendering commands - the time the view took to issue rendering commands for each element. Equivalent to the time spent in `Coherent::UIGT::View::RecordRenderingCommands`. Timer is affected by the number of elements that need to be repainted since last frame and the number of layers that need to be composited.
5. Layers count - the number of layers GT currently has created. See [Layers and how to use them](#) for more info.

Rendering thread counters

Note

Most of the information here is too low level for designers / JS programmers and if you are amongst them you probably only need to look at the previous section.

In the 'Record rendering commands' step GT creates a list of high-level (frontend) commands such as 'Draw a red circle' or 'Clip this image'. These commands are translated to low-level (backend) commands such as 'Bind this shader' or 'Updated vertex buffer' within `Coherent::UIGT::ViewRenderer::Paint` and later executed inside `renoir::RendererBackend::ExecuteRendering`. Usually these operations are executed on the rendering thread.

1. Frontend command count - the number of frontend commands GT has generated for the current frame.

2. Backend command count - the number of backend commands generated from the current frame.
 3. Backend draw commands - the number of draw calls for the current frames.
 4. Backend clear commands - the number of clear render target calls for the current frame.
 5. Backend set RT commands - the number of commands for setting render target, effectively the number of render target changes for the frame.
 6. ExecuteBuffers - the time required for processing all frontend commands and execution of backend commands.
 - (a) Frontend commands - the time it took to translate the frontend commands to backend commands.
 - (b) Backend execution - the time it took to execute the backend commands on the graphics API.
 7. GPU time - the time it took to paint the *last* frame on the GPU. This stat is currently only visible on DX11.
-

Chapter 4

Hiding/Showing Elements

Hiding and showing elements should be done with `visibility: hidden/visible` or with `display: none/block` (or `inline-block`, `flex`, etc.). One of the two can be used, but there are cases in which one is better than the other. In most cases using `display` is the better solution for hiding elements because the document will be rendered as if those elements didn't exist in it. In cases where animation is applied to an element, `visibility/display` should be applied after the animation ends.

Display vs Visibility property

In this section we will emphasize on the comparison between the two and answer which and why should be used in a given case.

Display property Pros:

- Hides an element and its children. Document will be rendered as if those elements didn't exist in it which translates to increased performance and responsiveness of the UI.
- `Display: none` is the fastest way to hide elements from the scene performance-wise. Hiding elements with `visibility: hidden` causes a bigger **Paint** than `display:none` because it doesn't exclude the elements from the document, just hides them visually.
- `Display: none` - recommended to be used on element whose animation is **no longer relevant to the scene**. After the same element is displayed, the animation will be **restarted** from the beginning. If `visibility` is used, the element is still rendered and the continuing animation can cause other elements to be redrawn if it goes "over" them.
- `Display: none` gives the option to place another element in the place of the hidden one in case the two elements aren't absolutely positioned.
- It is highly recommended to use `display: none` for hiding entire screens/scenes. Bigger elements have a bigger chance to go over other elements and cause redraws as mentioned earlier.

Note

Best practice when displaying elements is to change the content of the element before displaying it back on the screen. This will ensure no secondary Layouts, Paints and Style recalculations are done. Changing the content of an element while it is hidden with `display: none` - there is a very brief *Recalculate Style* done.

Display property Cons:

Displaying many elements back on the screen will cause drop in performance than when using `visibility: visible` because the elements should take back its place on the screen and recalculate their styles (background color, geometry, etc.). A workaround is to display elements on portions. For example - firstly displaying buttons/icons, then minimap and then a list of items. Making elements to appear in portions will not bottleneck frames.

Visibility property Pros:

- **Visibility:** `hidden` is good for temporarily hiding an element from the scene. Especially if this element has some animation which should continue running for the period it is hidden.
- Hiding an element with **visibility:** `hidden` preserves the element's place and dimensions in the document and doesn't change the position of other elements.

Visibility property Cons:

- Do not use **visibility:** `hidden` to hide entire scenes/screens or very big elements. Even though an element is not visible, it is still in the document flow. Furthermore it still affects other elements and even if it is in its own layer, it can still have diminishing effects. If one element is hidden and another is placed on top of it, this will cause heavier Paints if there is some interaction on it or an animation is applied to any one of them.
- If an element has an animation and it is `hidden`, the animation will still run which will have an impact on the performance. Use **visibility:** `hidden` only if the animation should stay active.
- Do not use **only** **visibility:** `hidden`. More elements in the document flow means less performance.

Summary - display vs visibility

When choosing between `display` or `visibility` property, it must be taken into consideration their intended purpose. In short - `visibility` should be used to hide and show an element which needs to stay in the document flow at all times. Hiding an element will allow an animation to continue running and it can be paused to preserve its current state. `Display: none` should be used to hide/disable no longer needed elements in the context of the scene and increase the responsiveness of the UI.

Chapter 5

Iteration tips

Some tips for smooth workflow and keeping developing in fast pace.

- When the development is in front-end stage, developing it in GT environment is highly recommended.

Run the page in the GT Player (or the Coherent Player if you are using the Unreal Engine 4 Plugin) instead of Chrome or any other browser.

Run the GT Player from the package folder - .bat. Default location and name is set inside the .bat file:

```
SET _Url=coui://uiresources/sos/MainUI.html
SET _GTPlayerFolder=..\Samples\UI\bin\
```

With the Coherent Player (UE4 package), open Command Prompt where the CoherentPlayer.exe is located (*package/CoherentPlayer/CoherentPlayer/Binaries/Win64*). Next, run the player and set the coui root folder and page paths like so: `CoherentPlayer.exe CoUIRoot="D:\package\CoUITTestFPS\Content\" -Page="coui://uiresources/MainUI.html"`. For full screen, press F11 after the scene loads or add `-Fullscreen` in the command line to start directly in full screen.

- Use the debugger. While running the GT Player, open the Debugger and attach it to the page. To open the Debugger, click on the Tools button (top right corner) and click on Debugger.

For useful shortcuts click on the question mark icon on the Tools bar.

If running Coherent Player, in the command line add `-InspectorPort=19999` and Make sure the inspector folder is inside *package/CoherentPlayer/CoherentPlayer/Binaries/Win64* alongside CoherentPlayer.exe.

- Do not enable "Show paint rectangles" and "Show composited layer borders" while profiling. This will add noise to the Recording.

More about the Debugger - UI Debugging and live editing

- While developing the page, make sure **vsync** is set to **1** inside the .bat. Running without vsync puts high load to the CPU to show how fast the page can run in GT.
- Loading external page inside the GT Player - inside the .bat file:

```
SET _Url=http://www.example.com/
```

This is the fastest way to check if something online is running as expected in GT.

Note

filter property works in GT only with prefix `-webkit-filter`.

- Watch for **Performance warnings** while or after you close the GT Player. Open the **CoherentUITGT.log** inside the package folder `\Samples\UI\bin\`.

Chapter 6

Layers and how to use them

The purpose of this section is to discuss where and why to use layers.

Firstly, let's define what **Render Tree** is. The tree consists of visual elements of the document or in other words - their visual representation in it. An "element" in this case is defined as **render object**. Each render object in the Render tree has defined style properties, including its geometry and position. Every time an element changes its geometry or position, it affects other elements too.

For example - if a scene has one wrapper at the top with dropdown menus, opening a menu will touch on other elements which are visually under it, causing those other elements to repaint too. This can be avoided by promoting the wrapper to layer. By doing this when a menu is opened, only the wrapper and its children will be repainted. Making an element a layer creates new texture especially for the content of this element.

Which elements should you promote to layers?

- Big elements which hold other dynamic elements e.g. wrapper with buttons which have some hover effect.
- Elements which position themselves over other elements in the UI, either by some hover effect or with an animation.
- List elements which are relative and close to one another and have some hover effect or animation.

How to use layers appropriately

- A layer should remain a layer at all times.

Preserve the transformation:

```
.element {
    transform: translate3d(0vh, 0vh, 0vh);
}

.element:hover {
    transform: translate3d(0vh, -0.6vh, 0vh);
}
```

If you are using 2D transformations, keep the element as a layer at all times by applying rotateX(0deg):

```
.element {
    transform: rotateX(0deg);
}

.element:hover {
    transform: rotateX(0deg) scale(1.1);
}
```

- Do not make every element a layer. Using too many layers can take up a lot of memory. It can also make the UI less responsive. Promoting elements to layers should always go hand in hand with profiling with the Debugger through the Timeline and using the "Show paint rectangles" feature to see if painting is contained only to the desired element. These are the only ways to see if the decision is beneficial and as expected.

- It is enough to promote a parent element to layer even if some of its children also have an animation. Layers don't allow the Paint to spread "outside" of it. An exception case is when the element has too many nested elements in which case another layer can be considered to be added inside the parent element. Make sure to profile.

List of properties that trigger layer creation in GT:

1. All Filters `-webkit-filter`.
2. `backface-visibility`
3. Transformations:
 - `translate3d`
 - `scale3d`
 - `rotateX`
 - `rotateY`
 - `rotate3d`
 - `perspective` (transform property only)
 - `matrix3d`
 - `translateZ`

Note

Transform: `translateZ(0)` don't create layers.

Tip: Use pagination instead of making long one page scrollable list with items. This will lighten the page load by making the non-active pages hidden with `display: none` along with their children. If the design doesn't allow it, make sure only the elements in the visible area of the screen are displayed. Consider using virtual lists and lazy-loading approaches for large lists.

Chapter 7

Transparent animation - Sprite sheet vs Video

How to integrate each one into the scene and performance overview.

Sprite sheet

When creating the sprite sheet, make sure not to leave empty spaces anywhere i.e. each row must be filled with frames to the end.

Note

Textures up to 16,384 by 16,384 pixels are allowed. Keep in mind that this imposes a limitation of the total duration (number of frames) of the effect you want to create through the sprite sheet. With transparent video this limitation does not exist.

The sprite sheet used for this sample has 12 columns and 8 rows and its dimensions are 6000x2248 pixels.

[Open spritesheet](#)

Element style, animation and keyframes:

```
.element {
  width: 100%;
  height: 100%;
  background-image: url("spritesheet.png");
  background-size: 1200% 800%;
  animation: playX 0.504s steps(12) infinite, playY 4.032s steps(8) infinite;
}

@-webkit-keyframes playX {
  0% { background-position-x: 0%; }
  100% { background-position-x: -1200%; }
}

@-webkit-keyframes playY {
  0% { background-position-y: 0%; }
  100% { background-position-y: -800%; }
}
```

In short, each row is displayed for half a second and for that half a second each and every frame is displayed. Imagine a typewriter which instead of printing it is reading.

The sprite sheet element should be inside another element which controls the dimensions. Every frame in the sprite sheet has 500x281 pixels so the controlling element will have that dimensions.

playX animation

The "steps" represent the number of individual frames in a row/column.

The playX animation is for the the animation from left to right and its duration (0.504s) is actually the steps (12) multiplied by (42ms - the time between the keyframes). In this case the time between the frames (42ms) is actually 24 frames multiplied by 42ms = 1008ms.

In another example - if there are 25 frames per second in the animation, the time between the frames will be 1000ms / 25 frames = 40ms.

playY animation

The **playY** animation is for the animation from top to bottom and its duration (4.032) is actually the duration of the playX animation (0.504) multiplied by the number of steps in the PlayY animation (8).

Keyframes

One frame is one "element" part (100%) from the sprite sheet. So 12 columns times 100% = 1200%. For the playY keyframe - 8 * 100% = 800%. Moving by 100% each animation step.

Transparent video

The same effect can be achieved with transparent video using FFmpeg to join a sequence of transparent .PNG images in video.

Here is how to do it - Video Support

In short:

```
ffmpeg -i sequence/image-%05d.png -c:v libvpx -pix_fmt yuva420p -metadata␣:s:v:0 alpha_mode="1" -auto-alt-ref 0 -s 500x281 -b:v 1M -r 24 output.webm
```

- The %05 in the given path after the -i parameters means that the source images have 5 digits in this scope i.e. image-00000.png, image-00001.png, image-00002.png and etc.
- If you have **intervals** in the names of the images just wrap the path in quotes -> -i "sequence/image %05d.png".
- To set custom **resolution** add for example "-s 500x281". If no resolution is specified the output resolution is the same as the source images.
- To set custom **bit rate** add "-b:v 1M".
- To set **frame rate** add "-r 24".

Further readings: [This guide](#) explains and provides some common options for creating video with libvpx. For all options visit the official [Documentation page](#).

Video integration

```
<video class="video-wrap">
  <source id="myVideo" src="output.webm" type="video/webm">
</video>
```

Apply the dimensions to the video element:

```
.video-wrap {
  width: 26vw;
  height: 14.6vw;
}
```

Performance testing

Use the following techniques to check the CPU time and memory used.

Using GT Player (CPU Time)

Disable vsync (using the `-vsync 0` command argument) and while your page is running pay attention to the **Av. dt.** field in the window's title.

Using Unreal Engine 4 (CPU Time)

If running GT in UE4, the test can be performed with any game (e.g. CoherentPlayer, our FPS sample or your own).

With the game running, hit the tilde button (~) to open the UE4 console and execute `stat CoherentGT` to see the performance data.

See our UE4 docs for more information about supported console commands - <http://coherent-labs.com/Documentation/UnrealEngine4-GT/>

Memory usage

To see how much memory GT is taking, run the GTPlayer with `-track_alloc 1`. The window's title will now display the amount of RAM / VRAM taken.

Note

Attaching the Inspector to the scene takes up memory too so keep it detached while measuring memory.

Sprite sheet animation

Test is performed with one animated sprite sheet on the page.

PNG, 6000x2248px.

CPU (GTPlayer): 0.15ms - 0.17ms

Total CPU (UE4): **0.2ms - 0.22ms**

Total RAM: **12MB**

Total VRAM: **60.1MB**

9 sprite sheets (same one used) animations running simultaneously

CPU (GTPlayer): 0.23ms - 0.24ms

Total CPU (UE4): **0.24ms - 0.28ms**. Not much of CPU time taken to run 8 more sprite sheet animations.

Total VRAM: **61.1MB**. Adding nine elements using the same sprite sheet takes up virtually the same amount of memory.

Video

CPU (GTPlayer): 0.3ms - 0.32ms

Total CPU (UE4): **0.58ms - 0.82ms**

Total RAM: ****~20MB****

Total VRAM: ****~20MB****

Note

More videos running on the page consume more processing power. Higher bitrate videos are also demanding more CPU power.

Additional test using 3 different sprite sheets

PNG, 3 x 6000x2248px.

CPU (GTPlayer): 0.21ms - 0.22ms

Total CPU (UE4): **0.21ms - 0.24ms**

Total VRAM: **168MB**

Sprite sheet vs Video Summary

Sprite sheet animation is very light. It needs less CPU time while running than video. Adding more sprite sheets doesn't affect the Total CPU that much. Every single video demands more CPU power. Videos use mainly CPU memory, sprite sheets use up mainly video memory.

Strive to keep the dimensions of the sprite sheet squarer. The provided sprite sheet transforms into 96 frames animation for 4 seconds which is 24 fps. Do not make larger sprite sheets if you don't have to. Keep it as small as possible. Making 60 fps animation may not look that much better than 24 fps one.
