DirectX 9Ex (WPF) dirty rectangle glitches in high GPU-Load situations

Bug description and preliminary analysis

Markus Neff, May 5th, 2021

Background/Scenario

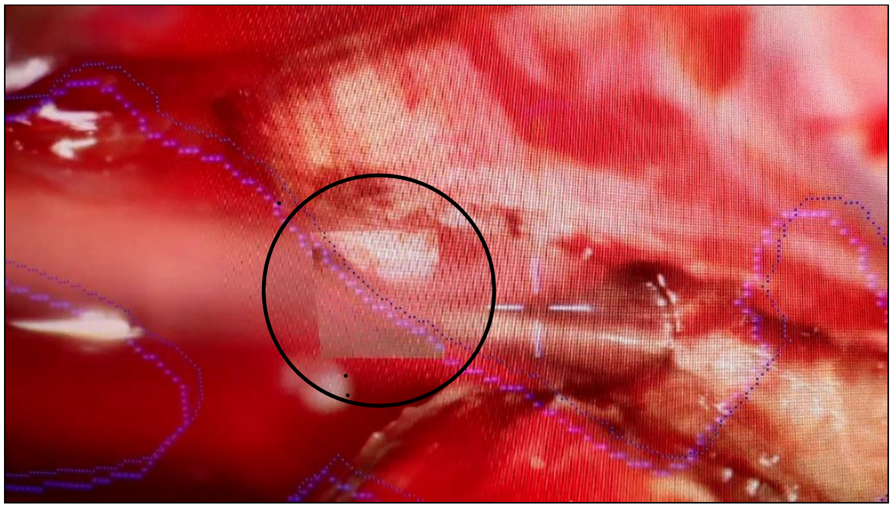
- Brainlab AG develops surgical medical devices with WPF-based medical device software – here is for example a photo of our Brainlab Curve Image Guided Surgery product:

- One of our applications (taken as an example here – multiple applications are actually affected) displays videos of an operating room (OR) microscope connected to our medical device
- Our system is a Windows 10 IoT 2019 (comparable to Win 10 Enterprise 1809) PC system with multiple monitors attached
- On each display, the OR team can open one of our full-screen applications and all of them have 3D views like this one for example:

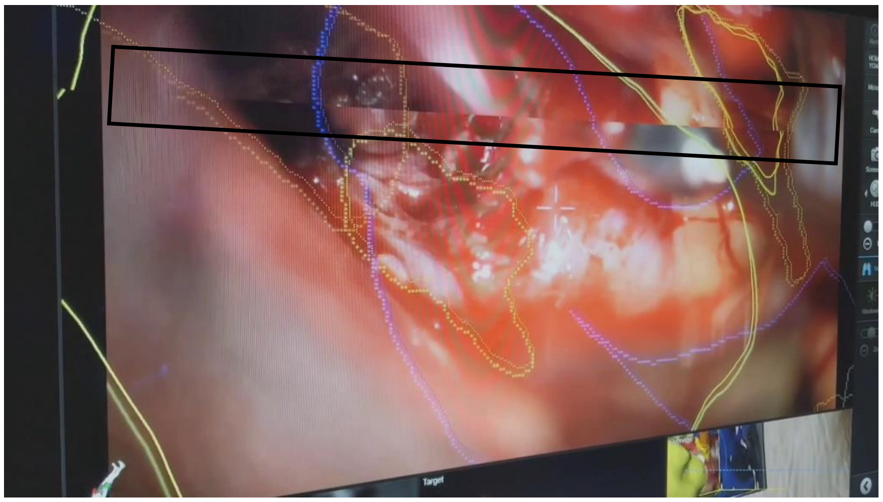
- Even if one of the applications is actively interacted with by e.g. one of the touch screens of the device, the other application on the other display is expected to run with the same viewing performance, as the other ones are concurrently looked at by further personnel in the operating room
- 3D content is rendered into an offscreen D3D11 context and brought into the WPF world by using a D3DImage visual and a technique inspired by and very close to this Microsoft guidance: Surface Sharing Between Windows Graphics APIs - Win32 apps | Microsoft Docs.
  Please note that this information is only intended to complete the picture – reduction to simple reproducers during our analysis showed, that it **does not contribute** to / trigger the problem (see below).
- Normally, the video in the 3D view and the 3D view rendering itself updates and invalidates/dirties the whole D3DImage back-buffer surface, but in some scenarios, we have multiple (D3D)Image visuals on one application screen and there might be WPF content with animations overlayed to the primary (D3D)Image visual, which leads to a situation where the WPF renderer is faced with partial 2D scene updates and proper dirty rectangle handling becomes essential.

Problem Description

- With high GPU-load, dirty-rect handling in WPF-based applications gets out of sync with window content update
- This triggers situations in which e.g. a bigger update of a visual is first clipped to the update rectangle of another smaller visual / control and just shows in one of the subsequent frames:



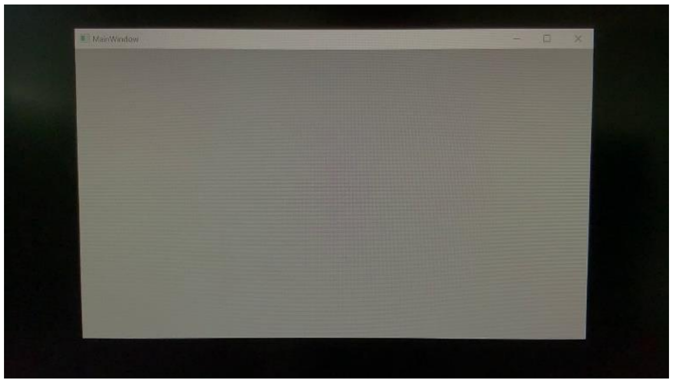
- Another variation of the problem is perceived as flickering similar to a tearing artifact – this is most likely because the update rectangle is the result of WPF combining several scattered partial content updates:



- Two videos showing the effect in real-world scenarios are included as files “UpdateRect_Prob_Real_Product_1.mp4” and “UpdateRect_Prob_Real_Product_2.mp4”.
- In a simple WPF reproducer (included as Visual Studio project “WPFSimple”), it is enough to have two overlayed rectangles, one bigger one in the back, from time to time (e.g. 10 Hz) changing its color (e.g. from black to gray and vice versa), and a smaller (barely visible) overlayed red one, having a very low, animated (modulated) alpha value (which is tried to be updated every frame by the WPF animation mechanism => 60 Hz).

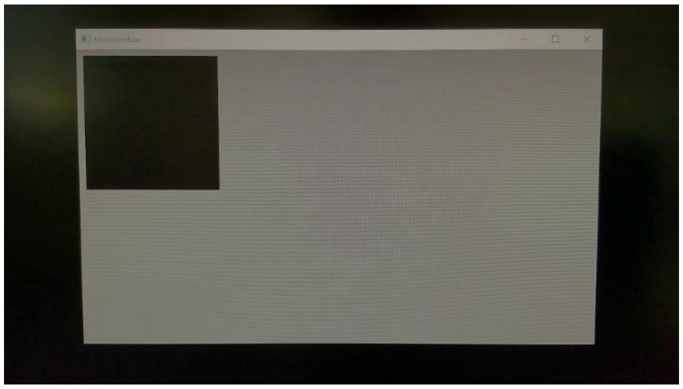
Expected result

- As soon as color of the background rectangle changes, full rectangle with new color is visible
- The overlayed rectangle is barely visible => not expected to alter the overall screen-appearance
- If in parallel, no other GPU intensive application is running, we see the expected result:



Observed result

- If in parallel, another 3D-heavy application is running (e.g. another process with a window on the other screen – such as the included Visual Studio project “D3D11_GPULoad”) and has the mouse focus, the following can be observed
- After a flip of the background rectangle color, the new color sometimes gets visible first in the bounds (update rectangle) of the small barely visible overlayed rectangle:



- In one of the next frames, the update of the rest of the background rectangle with its new color becomes fully visible
- Videos of the running application (without and with parallel GPU load) are included as “WPFSimple_Expected.MOV” and “WPFSimple_Problem_Visible.MOV”.

“Differential Diagnosis” – exclusion of certain components as root

- Problem happens with older and newer .NET/WPF versions
  - .NET 4.6.1 without and with latest Microsoft quality and security rollups
  - .NET 4.8 without and with latest Microsoft quality and security rollups
  - .NET Core 5.0 with latest Microsoft quality and security rollups

  => Our conclusion: not related to a specific bug in a specific .NET/WPF version

- Problem happens on older and newer Windows version with and without quality / security rollups/updates
  - Win 10 Enterprise 20H2 with latest (as of time of writing) quality / security updates
  - Win 10 IoT 2019 (equiv. version 1809) without and with latest (as of time of writing) quality / security updates
  - Win 10 IoT 1607 (with unknown patch status)
  - Win 7 Ultimate with latest quality / security updates (dated Jan 2020 as back then official support for this version was stopped by Microsoft)

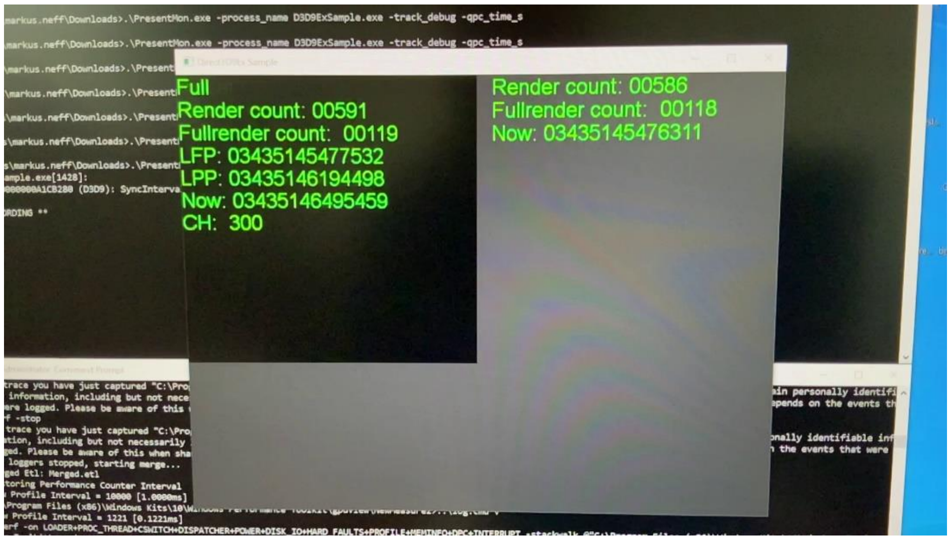
  => Our current conclusion: independent of a specific windows version

- Problem happens on different systems with different graphics cards and different drivers / driver versions
  - Dell Laptop System with Win 10 20H2 and Intel HD 630 + NVIDIA Quadro M1200
    - Newest drivers as of Apr. 2021:
      - Intel: 27.20.100.9316 (from Feb/18/2021)
      - NVIDIA: 27.21.14.6192 (from Mar/11/2021) => 461.92
  - Intel NUC 10 System with only one GPU – Intel UHD Graphics of Intel Core i7-19710U with driver 27.20.100.9316 (from Feb/18/2021)
  - Desktop Workstation with only one NVIDIA M2000 GPU with driver version 452.77
  - Lenovo Mobile Workstation with Windows 7 and Quadro 2000M GPU and driver version 377.83

  => Our current conclusion: not an issue of a specific driver version and/or GPU

Preliminary Analysis Results

- Problem can also be observed with a simple D3D 9Ex based application (included as Visual Studio Solution “D3D9Ex_DirtyRectProoblem”) that uses same presentation mechanism as WPF and “simulates” the rendering as done by WPF:
    - Update frame once every VSync interval
    - Render a sub-rectangle every frame
    - Render full surface in first frame and then every fifth frame
    - Swapchain
        - SwapEffect = D3DSWAPEFFECT_COPY
        - BackBufferCount = 1
        - PresentationInterval = D3DPRESENT_INTERVAL_ONE
    - IDirect3DDevice9::Present
        - pSourceRect and pDestRect used to express partial surface updates



```
markus.neff\Downloads>.\PresentMon.exe -process_name D3D9ExSample.exe -track_debug -qpc_time_s
markus.neff\Downloads>.\PresentMon.exe -process_name D3D9ExSample.exe -track_debug -qpc_time_s
\markus.neff\Downloads>.\Present
\markus.neff\Downloads>.\Present
\markus.neff\Downloads>.\Present
s\markus.neff\Downloads>.\Present
s\markus.neff\Downloads>.\Present
ample.exe[1428]:
000000A1CB280 (D3D9): SyncInterva

RDING **
```

```
Full
Render count: 00591
Fullrender count:  00119
LFP: 03435145477532
LPP: 03435146194498
Now: 03435146495459
CH:  300
```

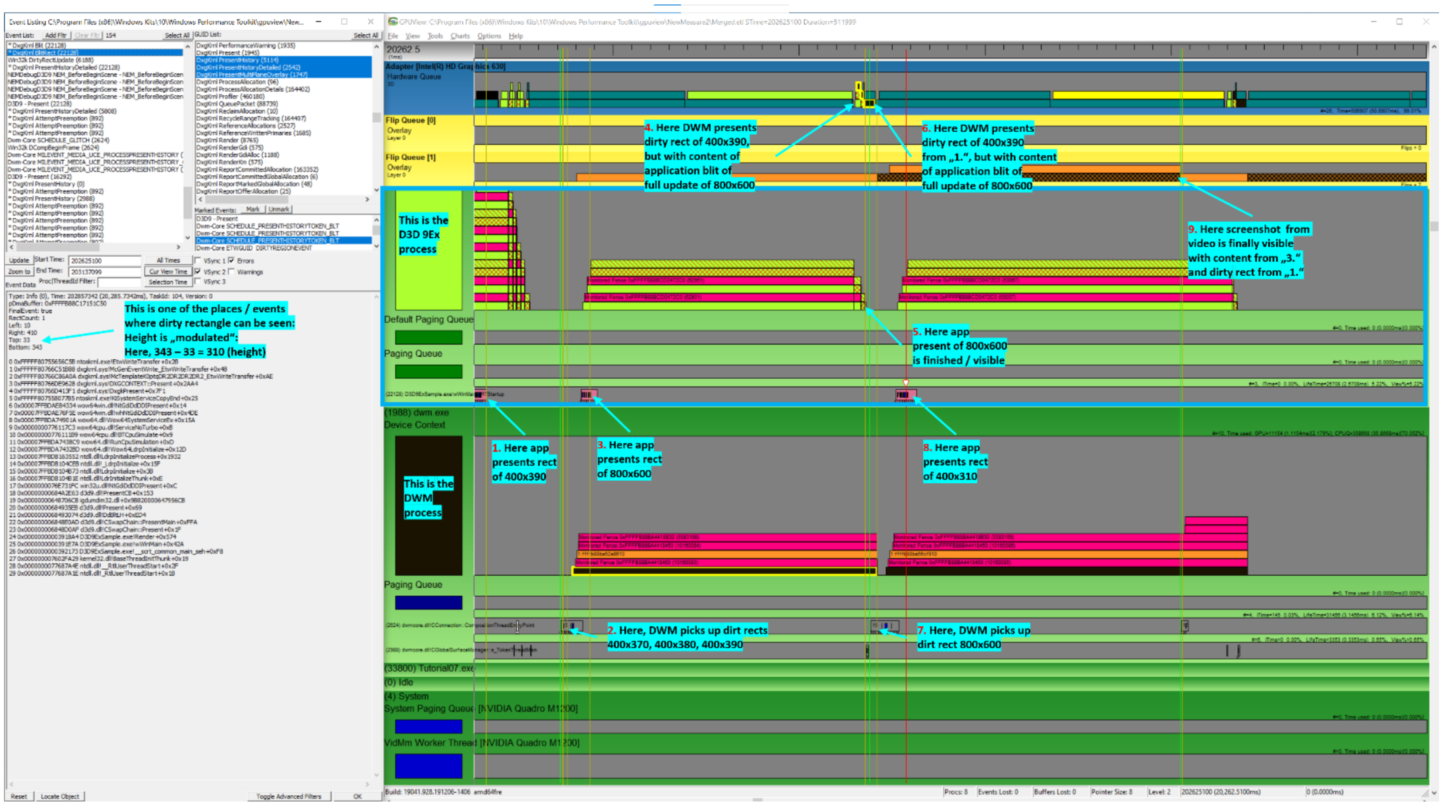
```
Render count: 00586
Fullrender count:  00118
Now: 03435145476311
```

```
trace you have just captured "C:\Pro
 information, including but not nece
ere logged. Please be aware of this
f -stop
 trace you have just captured "C:\Pro
ation, including but not necessarily
ged. Please be aware of this when sha
 loggers stopped, starting merge...
ged Etl: Merged.etl
toring Performance Counter Interval
 Profile Interval = 10000 [1.0000ms]
\Program Files (x86)\Windows Kits\10\W
 Profile Interval = 1221 [0.1221ms]
erf -on LOADER+PROC_THREAD+CSWITCH+DISPATCHER+POWER+DISK_IO+HARD_FAULTS+PROFILE+MEMINFO+DPC+INTERRUPT -stackwalk
```

```
ain personally identifi
epends on the events th

onally identifiable inf
 the events that were
```

- Deeper investigation based on ETW traces and GPUView suggests that under heavy GPU load the update rectangles as picked up by DWM are not properly synchronized with the actual D3D back-buffer updates – DWM falls a bit behind with the update rectangles:



```
Type: Info (0), Time: 202857342 (20,285.7342ms), TaskId: 104, Version: 0
pDmaBuffer: 0xFFFFB88C17151C50
FinalEvent: true
RectCount: 1
Left: 10
Right: 410
Top: 33
Bottom: 343

0 0xFFFFF80755656C5B ntoskrnl.exe!EtwWriteTransfer+0x2B
1 0xFFFFF80766C51B88 dxgkrnl.sys!McGenEventWrite_EtwWriteTransfer+0x48
2 0xFFFFF80766CB6A0A dxgkrnl.sys!McTemplateK0ptqDR2DR2DR2DR2_EtwWriteTransfer+0xAE
3 0xFFFFF80766DE9628 dxgkrnl.sys!DXGCONTEXT::Present+0x2AA4
4 0xFFFFF80766D413F1 dxgkrnl.sys!DxgkPresent+0x7F1
5 0xFFFFF807558077B5 ntoskrnl.exe!KiSystemServiceCopyEnd+0x25
6 0x00007FFBDAE84334 wow64win.dll!NtGdiDdDDIPresent+0x14
7 0x00007FFBDAE76F5E wow64win.dll!whNtGdiDdDDIPresent+0x4DE
8 0x00007FFBDA74901A wow64.dll!Wow64SystemServiceEx+0x15A
9 0x00000000776117C3 wow64cpu.dll!ServiceNoTurbo+0xB
10 0x00000000776111B9 wow64cpu.dll!BTCpuSimulate+0x9
11 0x00007FFBDA7438C9 wow64.dll!RunCpuSimulation+0xD
12 0x00007FFBDA7432BD wow64.dll!Wow64LdrpInitialize+0x12D
13 0x00007FFBDB163552 ntdll.dll!LdrpInitializeProcess+0x1932
14 0x00007FFBDB104CEB ntdll.dll!_LdrpInitialize+0x15F
15 0x00007FFBDB104B73 ntdll.dll!LdrpInitialize+0x3B
16 0x00007FFBDB104B1E ntdll.dll!LdrInitializeThunk+0xE
17 0x0000000076E731FC win32u.dll!NtGdiDdDDIPresent+0xC
18 0x00000000684A2E63 d3d9.dll!PresentCB+0x153
19 0x00000000648706CB igdumdim32.dll+0x9B82000647956CB
20 0x00000000684935EB d3d9.dll!Present+0x69
21 0x0000000068493074 d3d9.dll!DdBltLH+0xED4
22 0x000000006848E0AD d3d9.dll!CSwapChain::PresentMain+0xFFA
23 0x000000006848D0AF d3d9.dll!CSwapChain::Present+0x1F
24 0x00000000003918A4 D3D9ExSample.exe!Render+0x574
25 0x0000000000391E7A D3D9ExSample.exe!wWinMain+0x42A
26 0x0000000000392173 D3D9ExSample.exe!__scrt_common_main_seh+0xF8
27 0x000000007602FA29 kernel32.dll!BaseThreadInitThunk+0x19
28 0x0000000077687A4E ntdll.dll!__RtlUserThreadStart+0x2F
29 0x0000000077687A1E ntdll.dll!_RtlUserThreadStart+0x1B
```

This is one of the places / events where dirty rectangle can be seen: Height is „modulated“: Here, 343 – 33 = 310 (height)

This is the D3D 9Ex process

This is the DWM process

1. Here app presents rect of 400x390

2. Here, DWM picks up dirt rects 400x370, 400x380, 400x390

3. Here app presents rect of 800x600

4. Here DWM presents dirty rect of 400x390, but with content of application blit of full update of 800x600

5. Here app present of 800x600 is finished / visible

6. Here DWM presents dirty rect of 400x390 from „1.“, but with content of application blit of full update of 800x600

7. Here, DWM picks up dirt rect 800x600

8. Here app presents rect of 400x310

9. Here screenshot from video is finally visible with content from „3.“ and dirty rect from „1.“

  - In the trace one can see that DWM asks for dirty rectangles (present history) at the beginning of preparation (“2.”) for the new screen surface update but before the “flip” copy operation is executed on the GPU (“6.”), further window back-buffer updates are executed by the GPU (“4.”) but not considered in the DWM flip operation anymore
  - The full ETW trace for GPUView is included as “D3D9Ex_UpdateRect_Problem.etl”
  - The full video of the effect is included as “UpdateRect_Prob_D3D9Ex_Repro.MOV”
- “Differential Diagnosis”
  - Does <u>**not**</u> happen with D3D11 (see example TODO) with “SwapEffect = DXGI_SWAP_EFFECT_ SEQUENTIAL” and “IDXGISwapChain1::Present1” with dirty rectangles in the “DXGI_PRESENT_PARAMETERS” structure
    - Looking at a GPUView trace, it seems as if the GPU-based BitBlit copy that is used due to swap effect “DXGI_SWAP_EFFECT_SEQUENTIAL” is always copying the full back-buffer surface independently of what is handed in via the “DXGI_PRESENT_PARAMETERS” structure, which suppresses/prevents the effect from becoming visible
  - Does <u>**not**</u> happen with D3D11 and “DXGI_SWAP_EFFECT_**FLIP**_SEQUENTIAL” because there, multiple complete back-buffer surfaces of composed windows are shared with DWM and handed over for direct composition from them without a copy to an intermediate overall shared surface, where GPU blitting of the backbuffer content needs to be somehow synchronized with copying parts of the overall intermediate surface to the screen surface
  - Sample applications included: “D3D11_Swap_Sequential” and “D3D11_Swap_**Flip**_Sequential”.
  - The full ETW trace for GPUView is included as “D3D11_Flip_Sequential_Working.etl”
- Our current conclusion:
  - Either there is a bug in DWM / DXG system
  - Or D3D 9Ex is not suitable as a base rendering technology for partial window/surface updates in WPF

Questions

- Does Microsoft consider this as a bug in DWM/D3D 9Ex or in the WPF rendering system (because it uses D3D 9Ex which maybe cannot guarantee proper handling/synchronization of partial rendering surface updates) and why so?
- Can the bug be fixed in a future patch of Win 10 IoT 2019 (and also Win 10 LTSB 1607) ?
- Can the bug be fixed in a .NET/WPF 4.8 patch release application end possible?
- Is a workaround on WPF application end possible?
- Is a workaround on Direct3D 9Ex possible that keeps the possibility to use update rectangles (not always updating / copying the full surface)?
- In heavy GPU-load situations, we notice that there must be some priority mechanism in the DX rendering system which makes the (process with the) focused window get much higher priority than the non-focused one
    - It seems to not (only) be related to a dynamic priority boost of the current foreground CPU process / thread as even with manually setting the currently not foreground process to a base priority of "Realtime" does not really change the behavior
    - That priority mechanism is counter-productive for our use case as we have multiple applications on multiple screens that are equally important and shall get same rendering priority independently of what application is currently interacted with directly
    - Is there a way to deactivate or influence that priority mechanism?