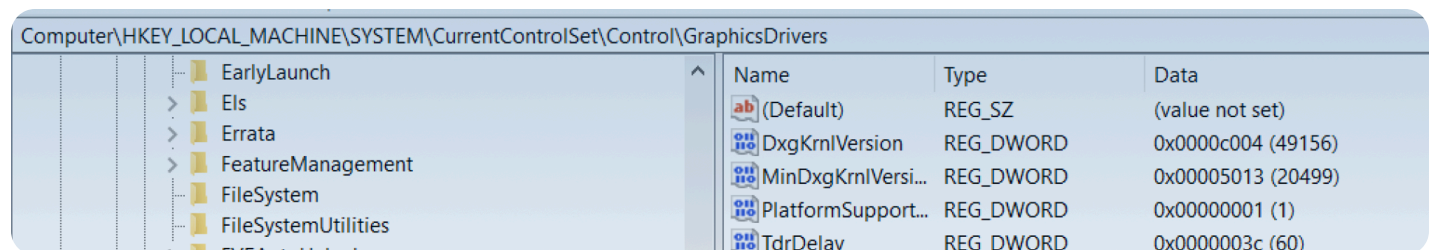


Developer
/ Documentation
/ Unreal Engine ▾
/ Unreal Engine 5.4 Documentation
/ Designing Visuals, Rendering, and Graphics
/ Optimizing and Debugging Projects for Real-Time Rendering
/ Dealing with a GPU Crash

Dealing with a GPU Crash

An overview of investigating, resolving, and reporting GPU Crashes in Unreal Engine.



Name	Type	Data
(Default)	REG_SZ	(value not set)
DxgKrnlVersion	REG_DWORD	0x0000c004 (49156)
MinDxgKrnlVersi...	REG_DWORD	0x00005013 (20499)
PlatformSupport...	REG_DWORD	0x00000001 (1)
TdrDelay	REG_DWORD	0x0000003c (60)

When a crash occurs in Unreal Engine, you may want to start by looking at the callstack generated by the [Crash Reporter](#) and log files that contain information to help in understanding what is happening. However, when a GPU crash occurs, the CPU callstack does not point directly to the cause but just indicates what the CPU was doing when the GPU crash happened. Therefore it provides no actionable information.

The content in this page will guide you through:

- What a GPU crash is?
- How to identify and investigate a GPU crash?
- How to debug a GPU crash from start to finish with various debugging tools?
- How to resolve Timeout Detection and Recovery (TDR) crashes?

What is a GPU Crash?

There are two types of errors which are commonly referred to as GPU crashes:

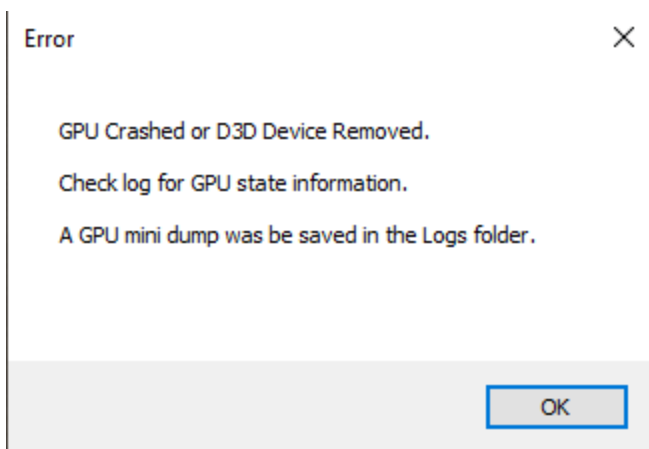
- **GPU timeouts** occur when a draw or dispatch takes too long to execute. Windows has a mechanism called **Timeout Detection and Recovery** (or **TDR**) which resets the GPU if a command takes longer than a certain amount of time (the default is 2 seconds. See

[Understanding and Resolving GPU Timeouts](#) section below). This causes a "device removed" error for all existing graphics contexts across all running applications.

- **GPU page faults** happen when a draw or dispatch tries to read GPU memory which is not available. This is similar to a CPU page fault (also known as access violation or segmentation fault), which happens when a CPU instruction tries to access a memory location which is not mapped to physical RAM.

Recognizing a GPU Crash

When a GPU crash is detected, the engine will terminate and display a dialog such as this:



If a debugger is attached, the error is grabbed before the debugger is shut down. The log will contain an error similar to this:

```
1 LogD3D12RHI: Error: CurrentQueue.Fence.D3DFence->GetCompletedValue() failed
2 at D:\UE5\Main\Engine\Source\Runtime\D3D12RHI\Private\D3D12Submission.cpp:984
3 with error DXGI_ERROR_DEVICE_REMOVED with Reason: DXGI_ERROR_DEVICE_HUNG
```

 Copy full snippet


The failing function and source location can be different occurrences of the crash, even when the underlying cause is the same. The CPU callstack where the error is caught is irrelevant when investigating the cause of a GPU crash since they are reported asynchronously to the application.

It's important to distinguish between device removed errors and other D3D errors because they can be detected at the same location in the CPU code, but the debugging methodology is different. For example, the function `VerifyD3D12Results()` often catches both types of

errors, and the error code must be looked at to determine how to proceed. GPU crashes report the code `DXGI_ERROR_DEVICE_REMOVED`, while other problems report different codes, such as `E_INVALIDARG` or `E_OUTOFMEMORY`.

This is an example of what is not a GPU crash:

```
1 Fatal error:
   File:D:\UE5\Main\Engine\Source\Runtime\D3D12RHI\Private\D3D12Util.cpp [Line:
   903]
2 Interfaces.CopyCommandList->Close() failed
3 at
   D:\UE5\Main\Engine\Source\Runtime\D3D12RHI\Private\D3D12CommandList.cpp:284
4 with error E_INVALIDARG
5 [Callstack] 0x00007fff2671c838 UnrealEditor-
   D3D12RHI.dll!D3D12RHI::VerifyD3D12Result()
```

 Copy full snippet

This particular crash is caused by sending incorrect arguments to a D3D API call, and it must be debugged in a different way. See the [Understanding and Resolving GPU Timeouts](#) section below.

What are the Possible Causes of a GPU Crash?

There are multiple reasons why GPU crashes occur in practice:

- **Content errors**, such as trying to render an extremely dense mesh with the non-Nanite rendering path, or an infinite loop in a custom HLSL node in a material, or a Niagara GPU system which takes too long to execute under certain conditions.
- Similar errors in **custom rendering code**, for example, a very long loop in a custom pass added to [Render Dependency Graph](#) (RDG).
- Engine bugs causing a built-in pass to take too long to execute, or incorrect resource management causing page faults.
- **GPU driver bugs**.
- **Hardware problems**, such as an overheating GPU.

- **VRAM exhaustion.** Video memory is virtualized in Windows, so it is possible to allocate much more than what is available on the GPU, especially when running multiple applications concurrently. However, moving blocks in and out of VRAM is slow, and can cause GPU timeouts under extreme circumstances.
- **Crashes from other apps.** Similarly to what is explained above, a GPU timeout invalidates all currently running contexts, so if another application causes a crash, it will cause the engine to crash as well.

How Do I Report a GPU Crash to Epic Games?

These are some guidelines to follow in order to provide as much information as possible to the Rendering teams when reporting a GPU crash:

- **Use the latest driver version available for your GPU.**
 - This helps rule out known driver bugs which have already been fixed by the manufacturer. Do not rely on Windows Update or even vendor tools for checking if the drivers are up-to-date. Always check the manufacturer website:
 - AMD: <https://www.amd.com/en/support>
 - Intel: <https://www.intel.com/content/www/us/en/download-center/home.html>
 - Nvidia: <https://www.nvidia.com/en-us/geforce/drivers/>
- **Include full logs with your report.**
 - The logs for the last 10 editor sessions are stored in the `Saved/Logs` subdirectory of your project directory. The log will be the most recent file in there when a crash occurs.
- **Distinguish whether it's a general crash from a GPU crash.**
 - Look through the most recent log for mentions of `DXGI_ERROR_DEVICE_REMOVED` or another error code. Any crash should be reported, but for the purpose of triaging, it's important to identify GPU crashes from other types of issues from the start.
- **Include debug logs for reproducible GPU crashes.**
 - If you can reproduce the crash, even somewhat reliably, add the command line argument `-gpucrashdebugging` and trigger the crash to happen. This will add useful information for identifying the problem to the engine log when the flag is present.
 - If an **Aftermath dump file** is generated, this is also useful to include in any report. These files are only generated when running on an NVIDIA GPU. Look inside the log

for a line that says "Writing Aftermath dump to: [PATH]" to find out if and where the dump file was written.

- **Include as much information as possible about how to reproduce the crash.**
 - Include steps, or actions you took to produce the crash. Additionally, you can help our engineers investigate and solve these crashes best if you can isolate the problem to a simple, self-contained project, or ones of Epic's own sample projects included with Unreal Engine.

Debugging a GPU Crash Workflow

GPU drivers have become very complex, which means that bugs and issues are more common.

If you're experiencing a GPU crash on your own machine, start by verifying you're running the latest available drivers from the GPU manufacturer. If you are investigating a GPU crash that happened on a different machine, the engine log will show the driver version and date, like the following example:

```
1 LogD3D12RHI: Found D3D12 adapter 0: NVIDIA GeForce RTX 3080 (VendorId: 10de,
DeviceId: 2206, SubSysId: 38901462, Revision: 00a1)
2 LogD3D12RHI: Max supported Feature Level 12_2, shader model 6.7, binding tier
3, wave ops supported, atomic64 supported
3 LogD3D12RHI: Adapter has 10067MB of dedicated video memory, 0MB of dedicated
system memory, and 130990MB of shared system memory, 3 output[s]
4 LogD3D12RHI: Driver Version: 546.17 (internal:31.0.15.4617, unified:546.17)
5 LogD3D12RHI: Driver Date: 11-9-2023
```

 Copy full snippet

After looking over the log, you'll want to identify the reason listed in the error. Note that this is separate from the error code, such as `DXGI_ERROR_DEVICE_REMOVED`. If the reason is `DXGI_ERROR_DEVICE_RESET`, this is likely a crash caused by a different application, so there is nothing to do besides making sure that you're not running other 3D apps at the same time as Unreal Engine. Reason codes are not completely reliable, but in most cases this is still valuable to check.

If the log and reason have not ruled out other causes, you can start to suspect a content or engine bug causing the crash. Depending on the build and runtime configuration, some level

of GPU crash debug logging may already be enabled. Look at the end of the log after the "device removed error" is reported to see if debug logging is included.

For GPU crashes with DirectX 11 (DX11), most GPU crashes are not actionable. Page faults are impossible, in theory, because there's no explicit memory management. If a page fault does occur, it's due to a driver bug. But, the type of crash hasn't surfaced to the application. If the crash happened on an NVIDIA GPU and Aftermath is enabled, you might get an Aftermath dump file that can be opened in the NVIDIA Nsight Graphics developer tool to see which pass crashed. For other GPU vendors, there's no way to debug GPU crashes, which means there's no actionable way for us to debug them unless they can be reproduced reliably. In that case, the debug methodology is to disable passes until we narrow down which one is the culprit, and then try to understand how that particular pass can cause the GPU to hang.


DirectX 12 (DX12) provides several sources of debug information. The first is the **RHI breadcrumbs**. This is where the engine uses the `WriteBufferImmediate` API to track active commands on the GPU. This system prints something like the log below when a crash occurs:

```
1 [GPUBreadCrumb] Last tracked GPU operations:
2 [GPUBreadCrumb] 3D Queue 0
3 Breadcrumbs: > Frame 1979 [Active]
4 Breadcrumbs: | BufferPoolCopyOps [Finished]
5 Breadcrumbs: | TexturePoolCopyOps [Finished]
6 Breadcrumbs: | WorldTick [Finished]
7 Breadcrumbs: | SendAllEndOfFrameUpdates [Finished]
8 Breadcrumbs: > GPUDebugCrash_DirectQueue_Hang [Active]
9 Breadcrumbs: > ClearGPUMessageBuffer [Active]
10 Breadcrumbs: VirtualTextureClear [Not started]
11 Breadcrumbs: ShaderPrint::UploadParameters [Not started]
12 Breadcrumbs: UpdateDistanceFieldAtlas [Not started]
13 Breadcrumbs: Scene [Not started]
14 Breadcrumbs: EnqueueCopy(GPUMessageManager.MessageBuffer) [Not started]
15 Breadcrumbs: AccessModePass[Graphics] (Textures: 14, Buffers: 4) [Not
    started]
16 Breadcrumbs: CanvasBatchedElements [Not started]
17 Breadcrumbs: SlateUI Title = QAGame - Unreal Editor [Not started]
```

 Copy full snippet

This particular log shows two passes / shaders running on the GPU. They are marked with the `[Active]` tag for `GPUDebugCrash_DirectQueue_Hang` and `ClearGPUMessageBuffer`. These

shaders need to be investigated to determine which one caused the hang.

 In this particular case, the problem is the `GPUDebugCrash_DirectQueue_Hang` pass. It was added by the console command `GPUDebugCrash hang`, which is used to trigger an intentional crash in order to verify that the debug code works.

The next source is **DRED**, which is [Microsoft's Device Removed Extended Data API](#). It uses a similar mechanism to the RHI breadcrumbs, but it can also report the page fault address when that type of error occurs. The DRED information is not hierarchical, it just contains a list of events in the active command list, like the example below:

```
1 DRED: Last tracked GPU operations:
2 DRED: Commandlist "FD3D12CommandList (GPU 0)" on CommandQueue "3D Queue (GPU 0)", 4 completed of 59
3 Op: 0, Unknown Op
4 Op: 1, BeginEvent [EditorSelectionOutlines]
5 Op: 2, BeginEvent [OutlineDepth 1679x799]
6 Op: 3, BeginEvent [EditorSelectionDepth] - LAST COMPLETED
7 Op: 4, ClearDepthStencilView
8 Op: 5, EndEvent
9 Op: 6, BeginEvent [DrawOutlineBorder]
10 Op: 7, DrawInstanced
11 Op: 8, DrawInstanced
12 Op: 9, DrawInstanced
13 Op: 10, DrawInstanced
14 Op: 11, EndEvent
```

 Copy full snippet

The `LAST COMPLETED` marker indicates the last command which finished on the GPU. Anything after that might be currently running and should be investigated. There's no immediate way to distinguish between commands which are active and commands which haven't started yet, unlike RHI breadcrumbs.

DRED has two modes: **lightweight** and **full**. Lightweight mode only uses markers sent by the engine, which is the same source of information RHI breadcrumbs use. Full mode automatically inserts a marker for every rendering command, such as Draw, Dispatch, Barrier, and so on, while also including the engine markers sent by the engine. Full mode provides

more information but has a significantly larger performance impact on the GPU. Lightweight mode is usually sufficient for tracking down problems.

When the console variable `D3D12.TrackAllAllocations` is enabled, the RHI keeps track of the address ranges for active resources, as well as freed resources over the last 100 frames. DRED also stores similar information (regardless of this console variable). When a page fault occurs, the fault address is compared against this list of ranges, so that we can report which resource was being accessed at the time. Below is an example output from RHI breadcrumbs:

```
1 PageFault: PageFault at VA GPUAddress "0x674000000"
2 PageFault: Last completed frame ID: -1 (cached: 4331) - Current frame ID:
  4332
3 PageFault: Logging all resource enabled: No
4 PageFault: Found 1 active tracked resources in 16.00 MB range of page fault
  address
5 GPU Address: [0x674000000 .. 0x6F135FFF8] - Size: 2100690936 bytes, 2003.38
  MB - Distance to page fault: 0 bytes, 0.00 MB - Transient: 0 - Name:
  Shadow.Virtual.VisibleInstances - Desc: Buffer 2100690936 bytes
6 PageFault: Found 0 active heaps containing page fault address
7 PageFault: Found 0 released resources containing the page fault address
  during last 100 frames
```

 Copy full snippet

In the case of this log, the resource was active, so possible causes include:

- It was evicted because it was not marked correctly for the residency manager before it was used.
- It's a tiled resource and we didn't map pages to it.

Example output from DRED:

```
1 DRED: PageFault at VA GPUAddress "0x2322884000"
2 DRED: Active objects with VA ranges that match the faulting VA:
3 DRED: Recent freed objects with VA ranges that match the faulting VA:
4 Name: Editor.SelectionOutline (Type: Resource)
```

 Copy full snippet

In this case, the fault is inside of a recently freed resource. The problem is that the resource was released while there was still pending GPU work using it.

Finally, we can extract information from Aftermath when the crash occurs on an NVIDIA GPU. It reports the type of fault (Timeout or PageFault), as well as the pass which triggered the error. The extracted information from Aftermath is hierarchical in this case — similar to RHI breadcrumbs — it shows the path in the scene pass hierarchy:

```
1 [Aftermath] Status: Timeout
2 [Aftermath] Scanning 5 command lists for dumps
3 [Aftermath] Begin GPU Stack Dump
4 [Aftermath] 0: Frame 1456
5 [Aftermath] 1: Scene
6 [Aftermath] 2: RayTracingReflections 0
7 [Aftermath] 3: MaterialSort SortSize=4096 NumElements=24231936
8 [Aftermath] End GPU Stack Dump
9 Aftermath: Writing Aftermath dump to:
   ../../../QAGame/Saved/Logs/UEAftermathD3D12.nv-gpudmp
```

 Copy full snippet

Aftermath produces a GPU crash dump file with DX11, which can be opened in Nsight. Nsight can show the location in the source where the problem occurred if the shader that caused the problem had debug information enabled.

These debugging features are controlled by the following command line arguments and console variables:

Debugging Feature	Command-line Flag	Console Variable
RHI Breadcrumbs	<code>-gpubreadcrumbs</code>	<code>r.GPUCrashDebugging.Breadcrumbs</code>
Full DRED	<code>-dred</code>	<code>r.D3D12.DRED</code>
Lightweight DRED	<code>-lightdred</code>	<code>r.D3D12.LightweightDRED</code>
Aftermath	<code>-nvaftermath</code>	<code>r.GPUCrashDebugging.Aftermath</code>
Everything ON	<code>-gpucrashdebugging</code>	<code>r.GPUCrashDebugging=1</code>
Everything OFF	<code>-nogpucrashdebugging</code>	—



The per-feature command-line flags accept an optional =0 or =1 argument to explicitly specify the state of the feature. Using the flag alone enables the feature, so it's equivalent to "=1." Command line arguments take priority over console variables.

The flags to turn everything on or off can be combined with other flags in order to include or exclude specific features. For example, using `-gpucrashdebugging -gpubreadcrumbs=0` enables everything except RHI breadcrumbs, while `-nogpucrashdebugging -dred` enables only full DRED while disabling everything else.

Consider the following performance impacts when using the debugging features above:

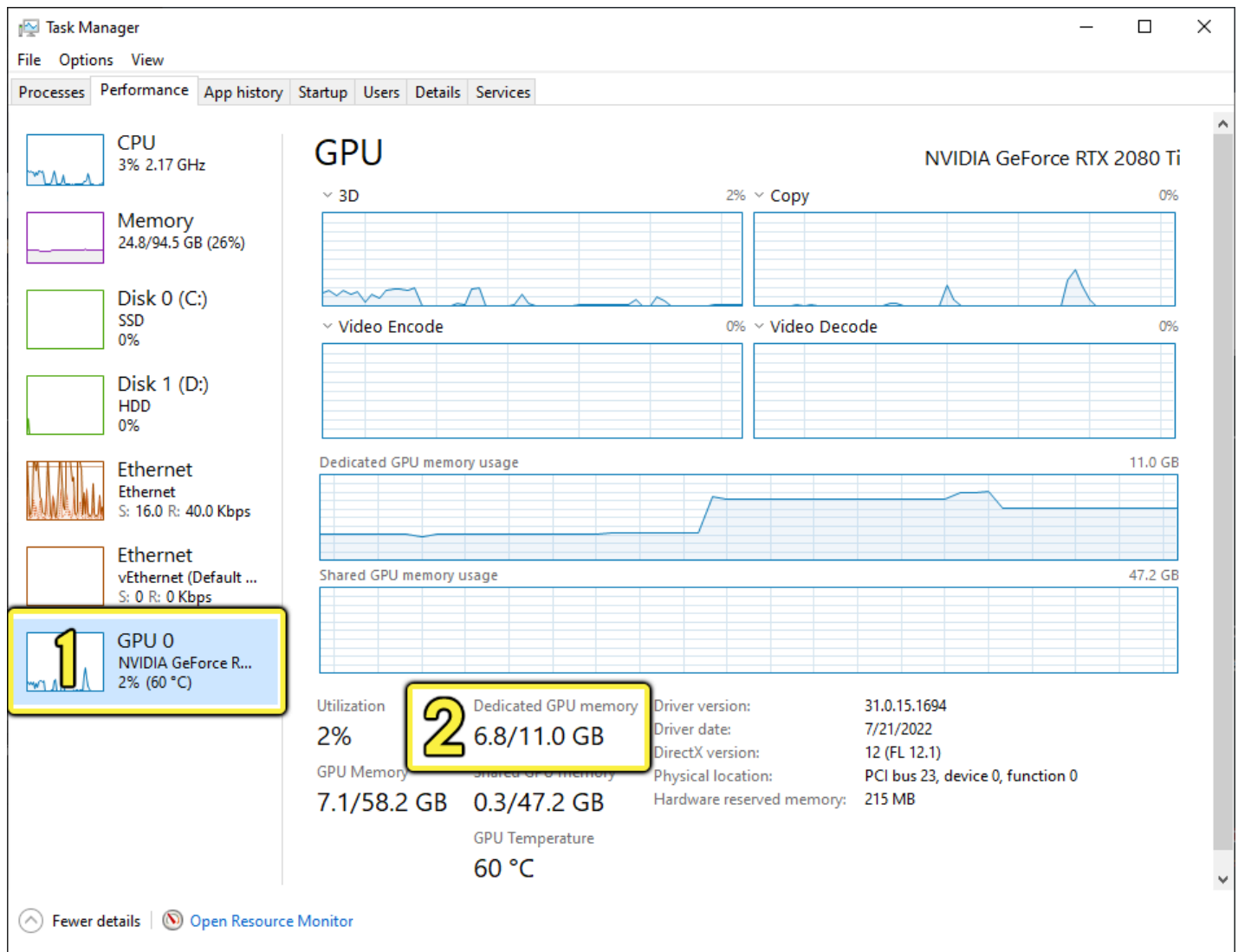
- RHI breadcrumbs and lightweight DRED can have non-negligible GPU performance cost, so the engine default settings only enable them in **Debug** and **Development** builds.
- Full DRED has a significant impact on performance and defaults are set to off in all build configurations.
- Aftermath doesn't have a performance cost and is always enabled on supported GPUs.

These default settings can be changed for each project.

Resolving GPU Out-Of-Memory Issues

If the GPU runs out of memory, it could potentially cause a crash. It largely depends on the RHI being used, some are more resilient than others and in the case of an OOM event, they may get slow instead of dying.

To understand why an out of memory crash may be occurring, start with the **Windows Task Manager** and use the **Performance** tab. Here, you can select your GPU (1) and see its available memory and how much it is currently consuming (2).



Windows Tasks Manager displaying the current stats for the GPU that include its available memory and current amount consumed.

With your project open and running, you can see how much GPU memory is being consumed versus what is available. If you are close to the available memory limit, it is most likely the problem causing the crash. In this case try the following:

- Close other programs that may be consuming large amounts of GPU memory.
- Simplify the scene using lower resolution textures, lower resolution meshes, culling to reduce objects in the scene, and so on.
- Use a lower screen resolution.
 - While working in the editor, you can use the Level Viewport **Screen Percentage** to render at a lower resolution.
- While working in the editor, if you have multiple viewports open, close all but one.
- Avoid disabling primary features like Niagara or Ray Tracing.
 - Bypassing these components changes many things, which could lead to invalid conclusions as to the cause of the GPU crash.

Understanding and Resolving GPU Timeouts

When the CPU sends a command to the GPU for computing something, the CPU sets a timer to count how much time the GPU needs to complete the operation. If the CPU detects the operation is taking too much time (by default, it is two seconds in Windows), it resets the driver causing a GPU crash to occur. This is called a TDR event (or Timeout Detection and Recovery).

Ideally, the engine should never send the GPU such an amount of work that triggers a TDR event. Instead, the engine should be able to split the task into smaller chunks so that TDR is avoided. In order to avoid these types of events, you can increase the amount of time it takes for a timeout to occur by editing the Windows Registry (see steps below for [How to Resolve TDR Events](#)).

TDR Events with Hardware Ray Tracing

[Hardware Ray Tracing](#) is particularly costly and is more likely to trigger TDR events when it is enabled. Some expensive ray tracing passes (such as Ray Tracing Global Illumination at very large resolutions) could take a long time to render and could trigger a TDR event.

The most expensive ray tracing passes (global illumination, and reflections) provide a way to render the passes in tiles instead of a single pass using the following console variables:

- `r.RayTracing.GlobalIllumination.RenderTileSize`
- `r.RayTracing.Reflections.RenderTileSize`

When the tile size of a pass is greater than 0, these passes are rendered N x N pixel tiles, where each tile is submitted as a separate GPU command buffer. This allows high quality rendering without triggering timeout detection.

How to Resolve TDR Events

One way of avoiding TDR events is to increase the amount of time it takes for Windows to trigger one by editing the Windows registry keys. In this guide, you are going to create two new registry keys: **TdrDelay** and **TdrDiDelay**.

- `TdrDelay` sets a timeout threshold. It is the number of seconds the GPU delays the preempt request from the GPU scheduler that handles processing and memory (VRAM).

- `TdrDdiDelay` sets the amount of time the operating system (OS) allows threads to leave the driver. After that time has elapsed, a timeout delay failure occurs.



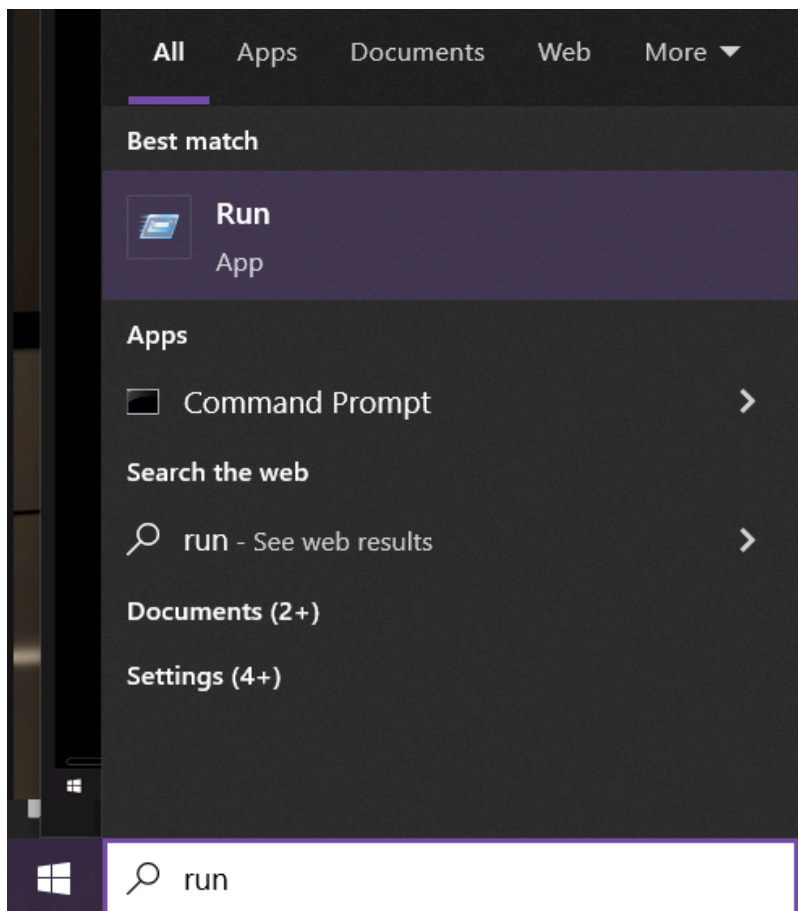
To learn more about registry keys, consult Microsoft's documentation about [Tdr Registry Keys](#).



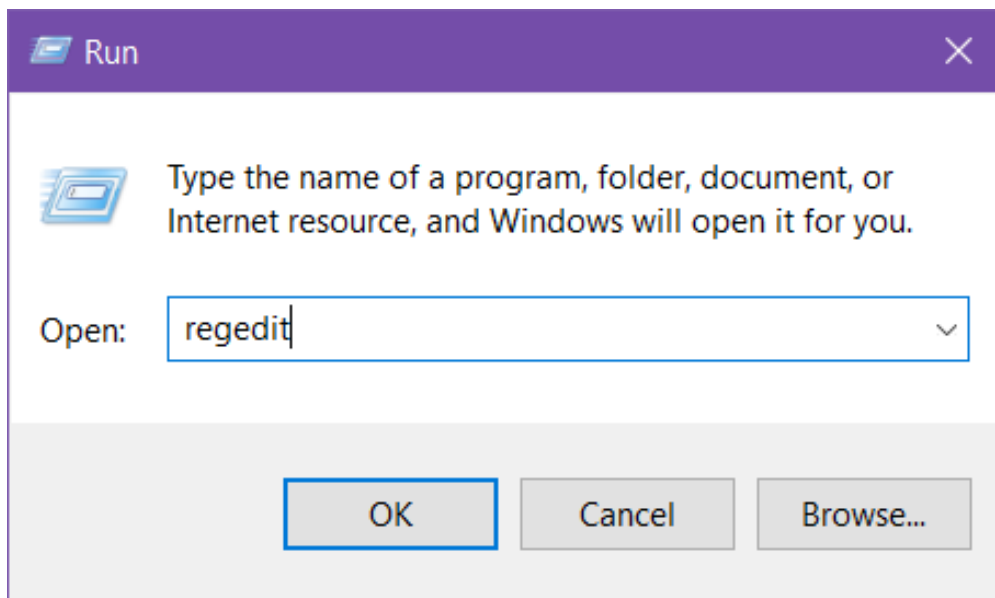
Changing the registry keys on your Windows operating system can have unexpected consequences and require a full reinstallation of Windows. Although adding or editing the registry keys in this tutorial should not result in those consequences, we recommend you backup your system prior to proceeding. Epic Games takes no responsibility for any damage caused to your system by modifying the system registry.

You need to add two registry keys to your graphics drivers. Follow these steps to add the registry keys.

1. Type '**run**' into the Windows operating system search bar. Open the **Run** application.

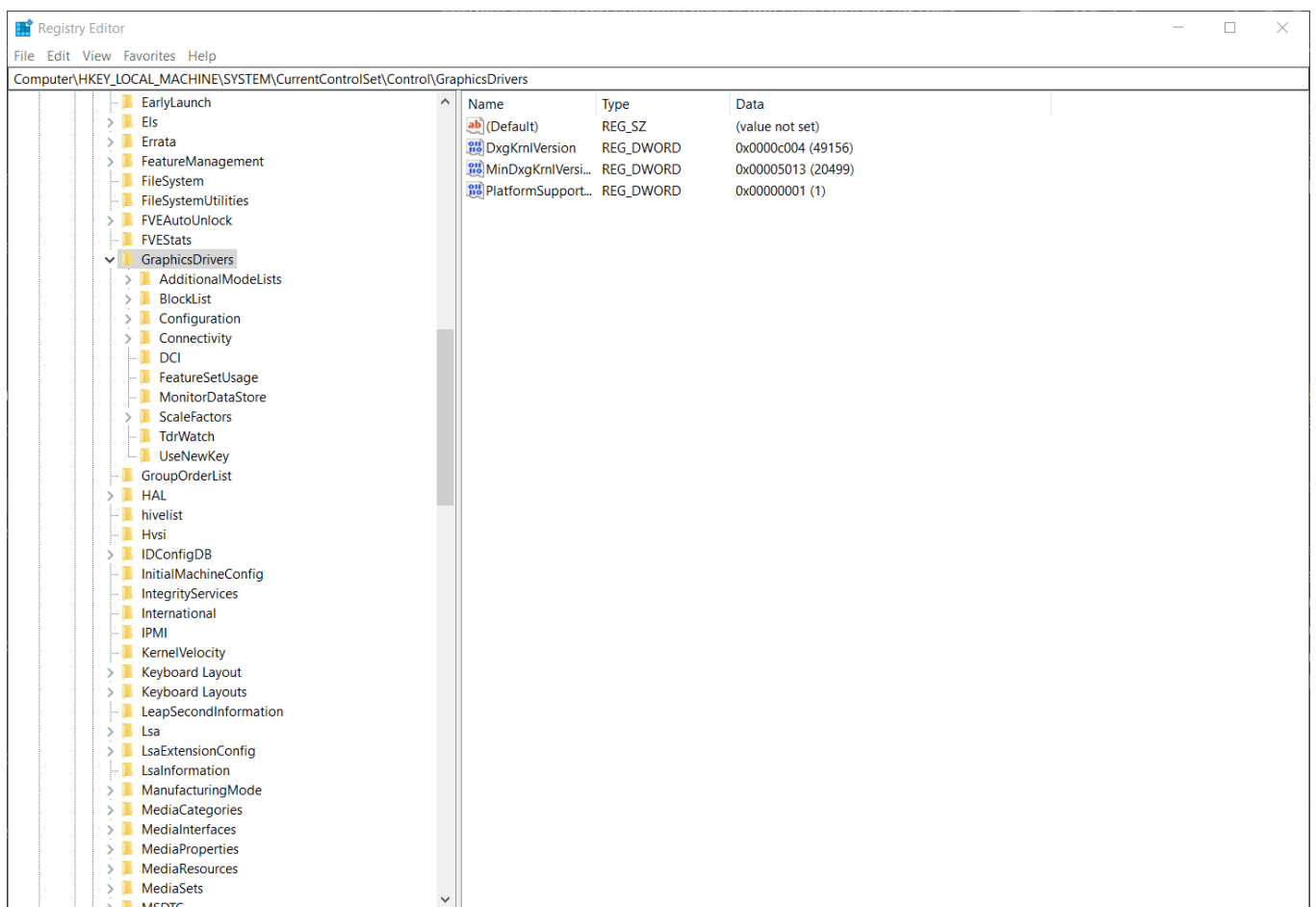


2. In the search field, type '**regedit**'. Click **OK** to open the Registry Edit Tool.



3. Navigate to the **GraphicsDrivers** section of the navigation on the left side of the Registry Edit tool. The location for this is

`Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers`.



Click image for full size.



The registry keys need to be added to the **GraphicsDrivers** folder, not any of its children. Be sure to select the correct folder.

4. The registry key you need is called `TdrDelay`. If this registry key already exists, double-click to edit it. If it does not already exist, right-click in the pane on the right and select **New > DWORD (32-bit) Value**.

Name	Type	Data
(Default)	REG_SZ	(value not set)
DxgKrnlVersion	REG_DWORD	0x0000c004 (49156)
MinDxgKrnlVersi...	REG_DWORD	0x00005013 (20499)
PlatformSupport...	REG_DWORD	0x00000001 (1)

New >

Key

String Value

Binary Value

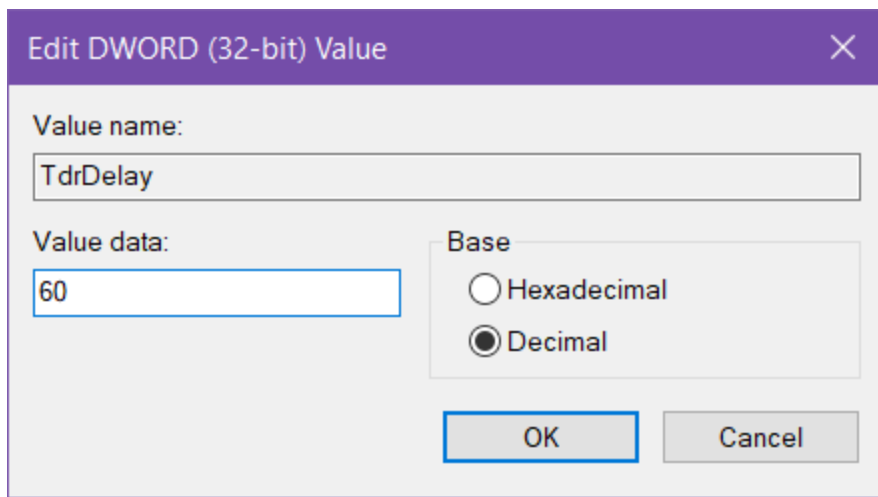
DWORD (32-bit) Value

QWORD (64-bit) Value

Multi-String Value

Expandable String Value

5. Set the **Base** to **Decimal**. Set the **Value** of TdrDelay to **60**. Click **OK** to finish.



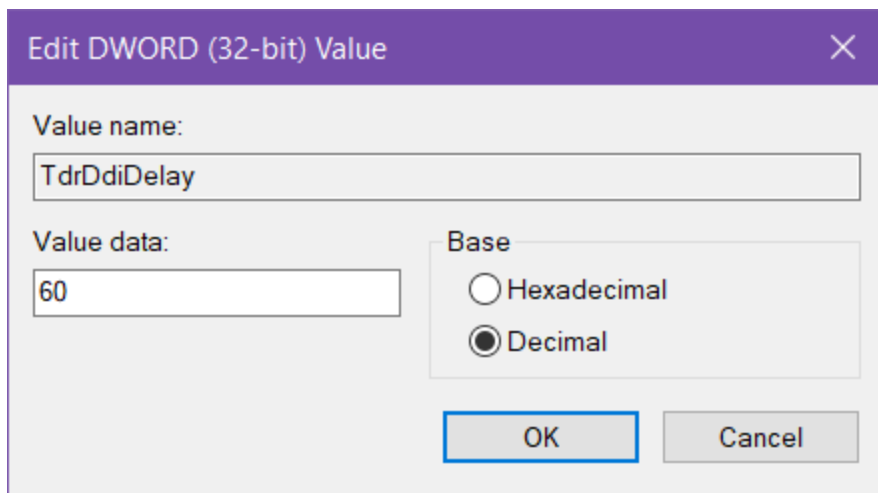
Value name:
TdrDelay

Value data:
60

Base
☐ Hexadecimal
☒ Decimal

OK Cancel

6. You need a second registry key called `TdrDdiDelay`. If this registry already exists, double-click to edit it. If it does not already exist, right-click in the right hand pane and select **New > DWORD (32-bit) Value** to create it.
7. Set the **Base** to **Decimal**. Set the **Value** of `TdrDdiDelay` to **60**. Click **OK** to finish.



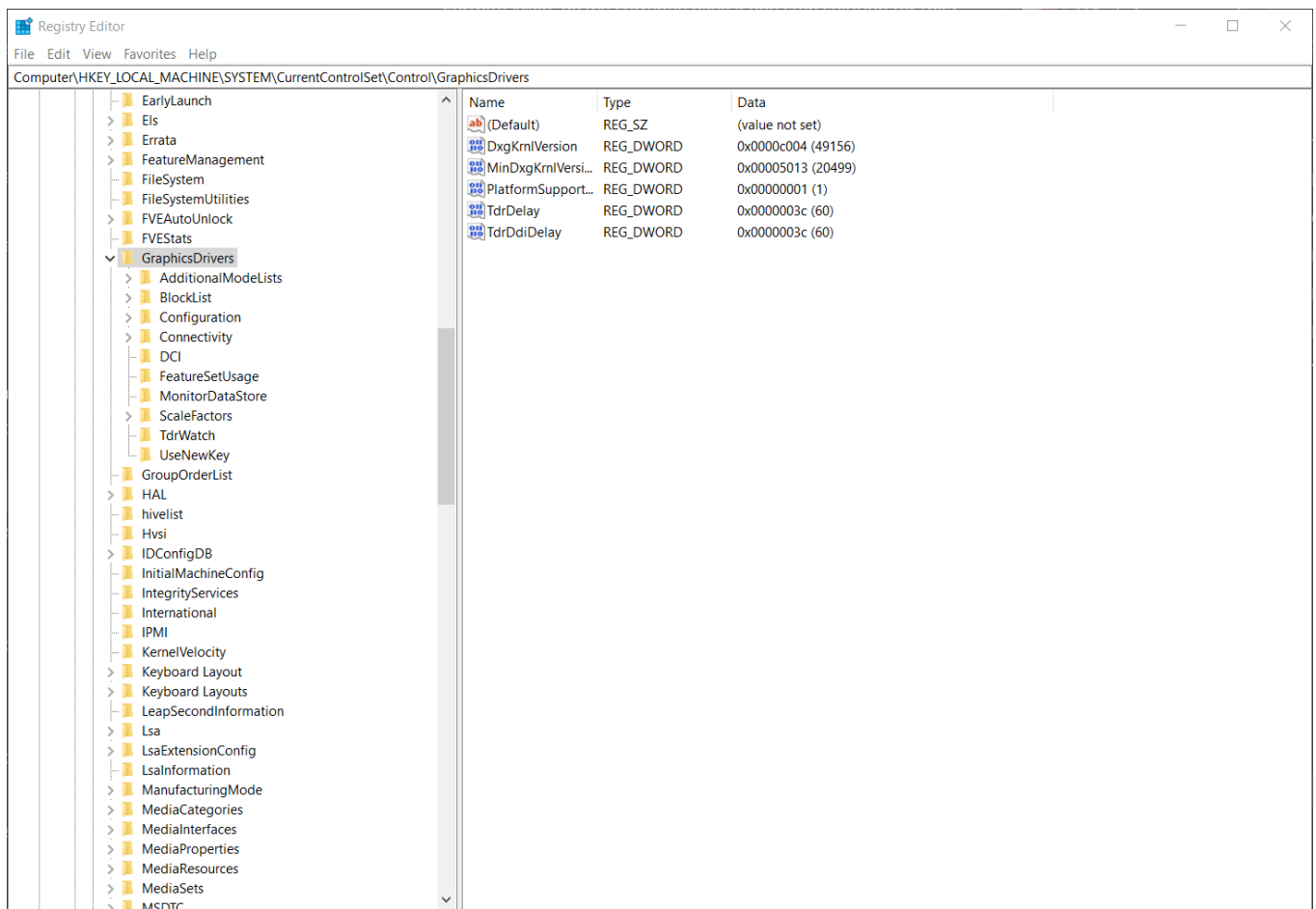
Value name:
TdrDdiDelay

Value data:
60

Base
☐ Hexadecimal
☒ Decimal

OK Cancel

8. Your registry should now include both `TdrDelay` and `TdrDdiDelay`.



Click image for full size.

9. Close the Registry Editor.

10. Restart your computer for these changes to take effect.

By adding these registry keys, Windows will now wait for 60 seconds prior to determining that the application has taken too long for its process.

Although this is a good way to curb GPU crashes based on rendering, this will not resolve all crashes. If you try to process too much data at once, the GPU may time out regardless of how long you set the timeout delay. This solution is only designed to give your graphics card a little extra time.