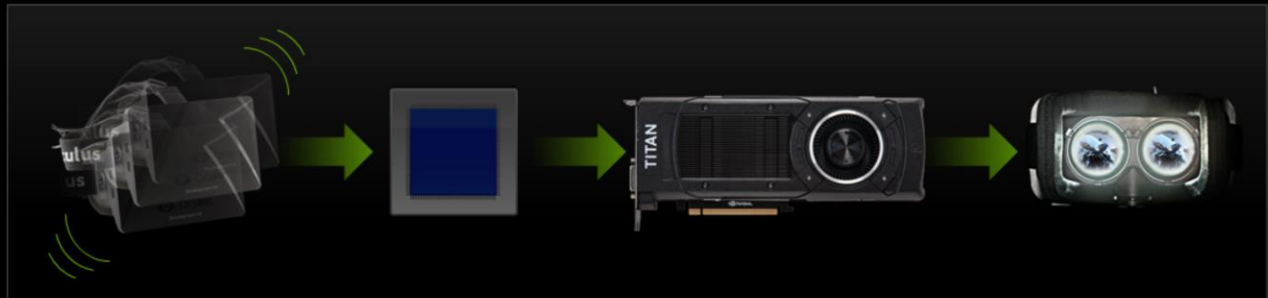
GAMEWORKS VR

Nathan Reed — Developer Technology Engineer, NVIDIA

NVIDIA.

LATENCY



Motion to photons in ≤ 20 ms

gameworks.nvidia.com

Virtual reality is extremely demanding with respect to rendering performance. Both the Oculus Rift and HTC Vive headsets require 90 Hz, a very high framerate compared to gaming on a screen. We also need to hit this framerate while maintaining low latency between head motion and display updates.

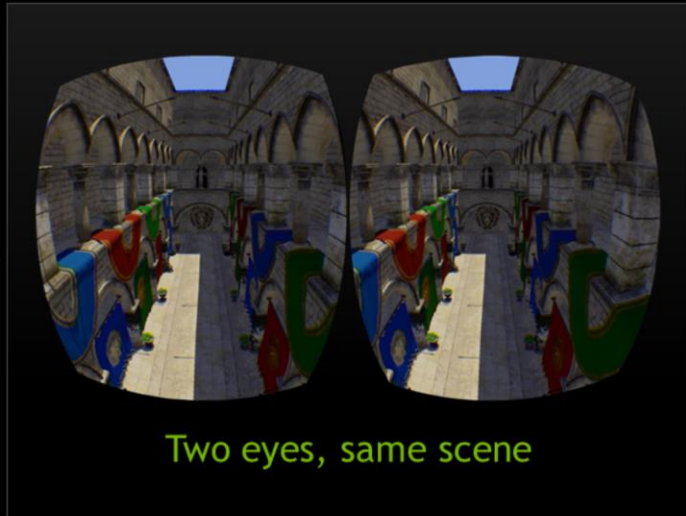
Mike Abrash at Oculus has done some research on this, and his results show that the motion-to-photons latency should be at most 20 milliseconds to ensure that the experience is comfortable for players.

The problem is that we have a long pipeline, where input has to first be processed and a new frame submitted from the CPU, then the image has to be rendered by the GPU, and finally scanned out to the display. Each of these steps adds latency.

Traditional real-time rendering pipelines have not been optimized to minimize latency, so this goal requires us to change how we think about rendering to some extent.

NVIDIA.

STEREO RENDERING



gameworks.nvidia.com

The other thing that makes VR rendering performance a challenge is that we have to render the scene twice, now, to achieve the stereo eye views that give VR worlds a sense of depth. Today, this tends to approximately double the amount of work that has to be done to render each frame, both on the CPU and the GPU, and that clearly comes with a steep performance cost.

However, the key fact about stereo rendering is that both eyes are looking at the same scene. They see essentially the same objects, from almost the same viewpoint. And we ought to be able to find ways to exploit that commonality to reduce the rendering cost of generating these stereo views.

GAMEWORKS VR

SDK for VR headset and game developers

MULTIRES SHADING

Increase performance via an innovative new way to render for VR

VR SLI

Scale performance with multiple GPUs

CONTEXT PRIORITY

Minimize head tracking latency with asynchronous time warp

DIRECT MODE

Plug and play compatibility from GPU to HMD

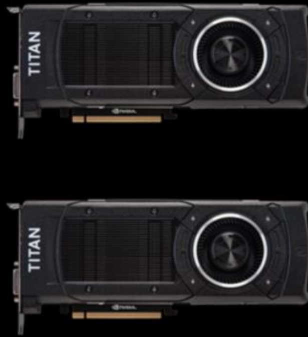
FRONT BUFFER RENDERING

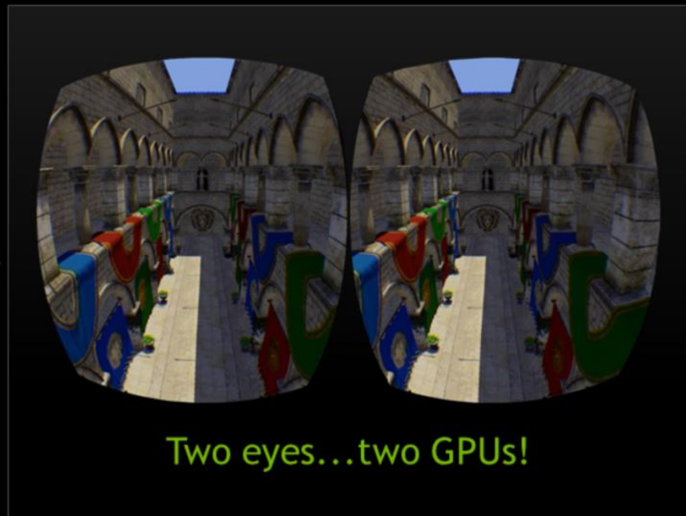
Reduce latency by rendering directly to the front buffer

gameworks.nvidia.com

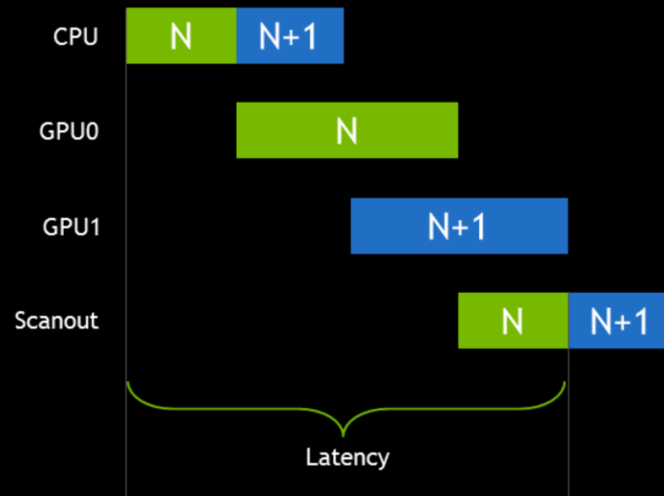
And that's where GameWorks VR comes in. GameWorks VR refers to a package of hardware and software technologies NVIDIA has built to attack those two problems I just mentioned — to reduce latency, and accelerate stereo rendering performance -- ultimately to improve the VR experience. It has a variety of components, which we'll go through in this talk.

VR SLI



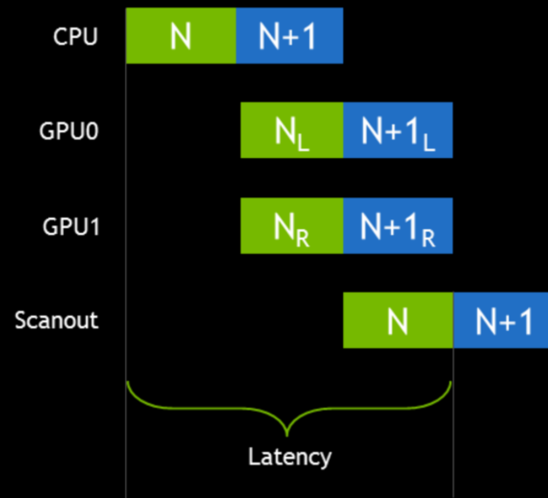


Given that the two stereo views are independent of each other, it's intuitively obvious that you can parallelize the rendering of them across two GPUs to get a massive improvement in performance.

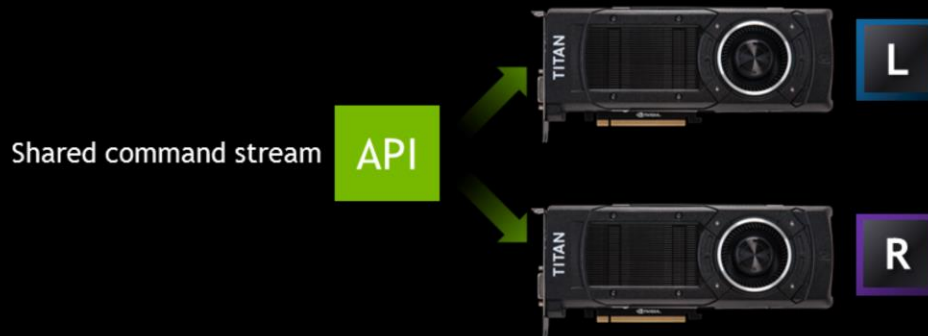


Before we dig into VR SLI, as a quick interlude, let me first explain how ordinary SLI normally works. For years, we've had alternate-frame rendering, or AFR SLI, in which the GPUs trade off whole frames. In the case of two GPUs, one renders the even frames and the other the odd frames. The GPU start times are staggered half a frame apart to try to maintain regular frame delivery to the display.

This works reasonably well to increase framerate relative to a single-GPU system, but it doesn't help with latency. So this isn't the best model for VR.

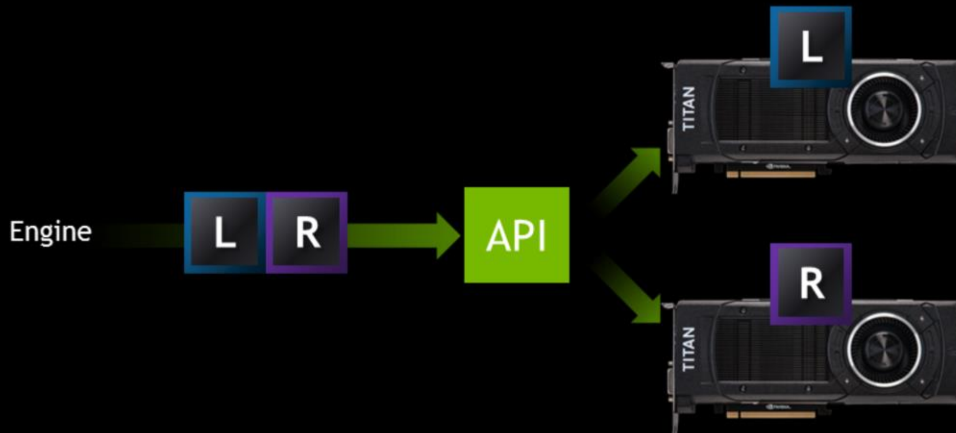


A better way to use two GPUs for VR rendering is to split the work of drawing a single frame across them — namely by rendering each eye on one GPU. This has the nice property that it improves both framerate *and* latency relative to a single-GPU system.



I'll touch on some of the main features of our VR SLI API. The key idea is that the application submits a single command stream that gets broadcast to both GPUs. Having a shared command stream isn't the only way to use the API, but it's the most efficient.

Having a shared command stream means that all the state changes and draw calls that you submit are executed on both GPUs. So you render your scene only once, including all the objects that are visible to both eyes, rather than submitting separate draw calls for each eye.



In order to get a stereo pair out of a single command stream, in our API we have ways to specify a very limited set of state that can differ between the two GPUs: namely, constant buffer bindings, viewports, and scissors.

So, you can prepare one constant buffer that contains the left eye view matrix, and another buffer with the right eye view matrix.

Then, in our API we have a SetConstantBuffers call that takes both the left and right eye constant buffers at once and sends them to the respective GPUs. Similarly, you can set up the GPUs with different viewports and scissor rectangles.

In case broadcasting all the draws to both GPUs is a bit too restrictive, we also have support for GPU affinity masking. This allows you to direct individual draw calls to one GPU or the other, or both.

This is a great way to start out adding VR SLI support to your engine. In case your engine already renders stereo views sequentially, you can switch to parallel stereo rendering very easily by inserting SetGPUMask() calls at the beginning of each eye. Set mask to the first GPU, render the left eye, then set mask to the other GPU and render the right eye.

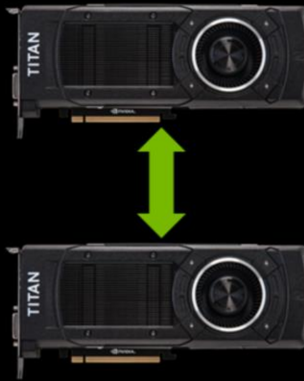
The one downside there is you're still paying for the cost of separate draw calls for each view. If you convert your engine to render the scene once, using the broadcast mode, then you can cut your draw calls in half and therefore cut your CPU load in half. A draw call that goes to both GPUs is no more costly than a draw call that goes to just one.

So, a great integration strategy is to use affinity masking to quickly bring up VR SLI in your engine, then incrementally convert the engine to broadcast, piece by piece, to improve CPU efficiency.

By the way, all of this extends to as many GPUs as you have, not just two. So you can use affinity masking to explicitly control how work gets divided across 4 or 8 GPUs, as well.

VR SLI

Cross-GPU data copies, via PCIe



gameworks.nvidia.com

And of course we need to be able to transfer data between GPUs. For instance, after we've rendering both halves of the stereo pair, we have to get them back onto a single GPU to output to the display. So we have an API call that copies a texture or a buffer between two specified GPUs, using the PCI Express bus.

One point worth noting here is that the PCI Express bus is actually kind of slow. PCIe2.0 x16 only gives you 8 GB/sec of bandwidth, which isn't that much, and it means that transferring an eye view will require about a millisecond. That's a significant fraction of your frame time at 90 Hz, so that's something to keep in mind.

To help work around that problem, our API supports asynchronous copies. The copy can be kicked off and done in the background while the GPU does some other rendering work, and the GPU can later wait for the copy to finish using fences. So you have the opportunity to hide the PCIe latency behind some other work.

VR SLI PERFORMANCE SCALING

- Up to the app to decide how to use GPUs
  - Needs engine integration
- Scaling depends on the app
- Duplicating work → less scaling
  - Shadow maps
  - GPU particles, physics sims

gameworks.nvidia.com

So, VR SLI is really an API for explicit control of multiple GPUs. It gives you the ability to distribute the workload however you want, over as many GPUs as you have. However, that also means it's up to the app to decide how to use the GPUs. VR SLI doesn't work automatically, or even mostly automatically, like AFR SLI — it requires work on the engine side to integrate it.

Moreover, performance scaling is going to depend on the app. One potential scaling issue is work that gets duplicated on both GPUs. For example, both GPUs will need to render the shadow maps for the scene. That's technically a bit inefficient, because the two GPUs will be repeating the same work, generating exactly the same shadow maps. GPU particle systems and GPU physics sims are another couple of cases where you'll probably end up repeating the same work on both GPUs.

The scaling we see will depend on what proportion of the frame is taken up by this kind of duplicated work. If there's very little duplicated work, you'll get close to 2x scaling, but if there's a lot you'll get lower scaling.

DEVELOPER GUIDANCE

- Teach your engine to render both views at once
- Currently:

```
for (each view)
  find_objects();
  for (each object)
    update_constants();
    render();
```

gameworks.nvidia.com

We’ve talked about how VR SLI operates at a high level, but as a game developer, what should you take away from this?

The most important thing is that to be ready to use VR SLI efficiently, you don’t want to process individual views one at a time, sequentially. Instead, you want to teach your engine to render both the views in a stereo pair at once.

Currently, most engines are doing something like this pseudocode snippet here at a high level. They’re iterating over views as an outer loop. For each view, they build a list of visible objects, then they calculate the appropriate constants for those objects and generate the rendering commands for them. As mentioned earlier, this can work with VR SLI using GPU affinity masking, but it’s not going to be the most efficient approach.

NVIDIA.

DEVELOPER GUIDANCE

- Where you want to end up:

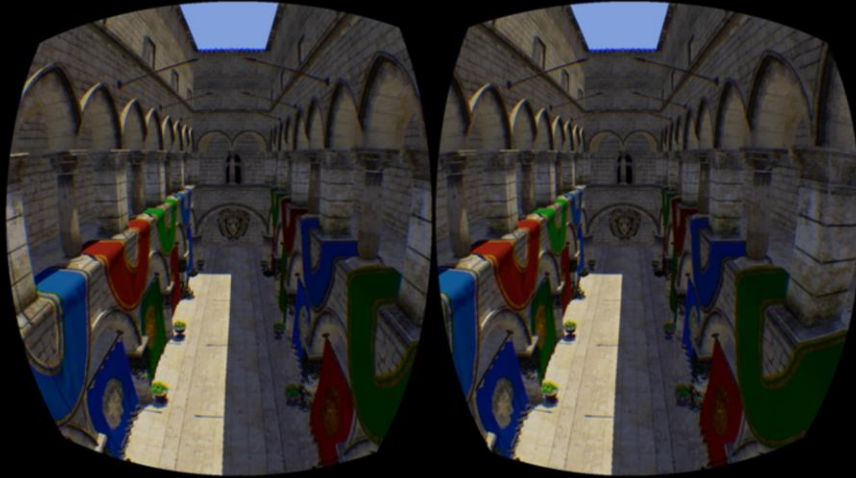
```
find_objects();
for (each object)
  for (each view)
    update_constants();
  render();
```

gameworks.nvidia.com

Where you want to end up to make good use of VR SLI is more like this. First, you want to build a single list of visible objects that is the union of all the objects visible from both eyes. Then, when you calculate the constants for an object, you build both the left eye and right eye constants. Those per-view constants should be stored together somehow, so they can be submitted together to the API. Finally, you'll submit the rendering commands just once for both views.

So, with VR SLI, rendering a stereo pair of views is really almost like rendering a single view. All you have to do differently is calculate and submit a few extra matrices in your constant buffers, to set up the appropriate transforms for the left and right eye views.

MULTI-RESOLUTION SHADING



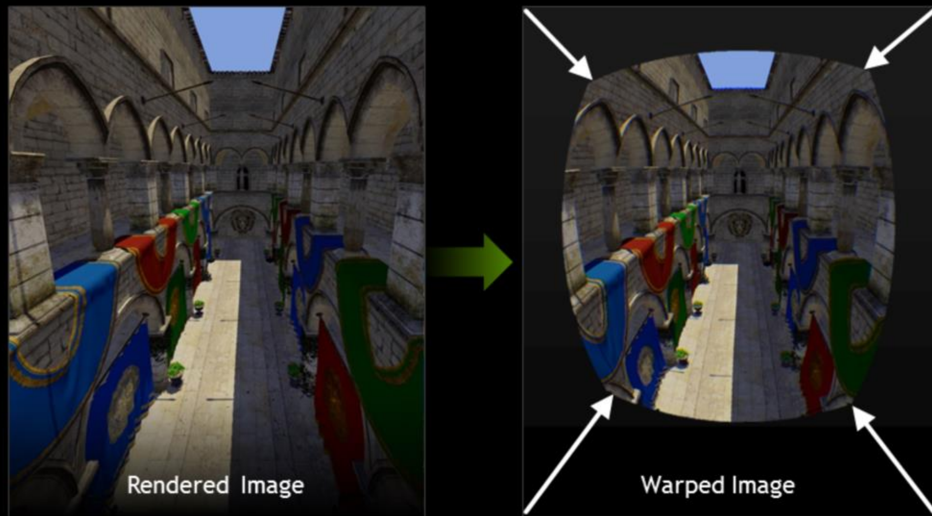
gameworks.nvidia.com

Our next topic is multi-resolution shading.

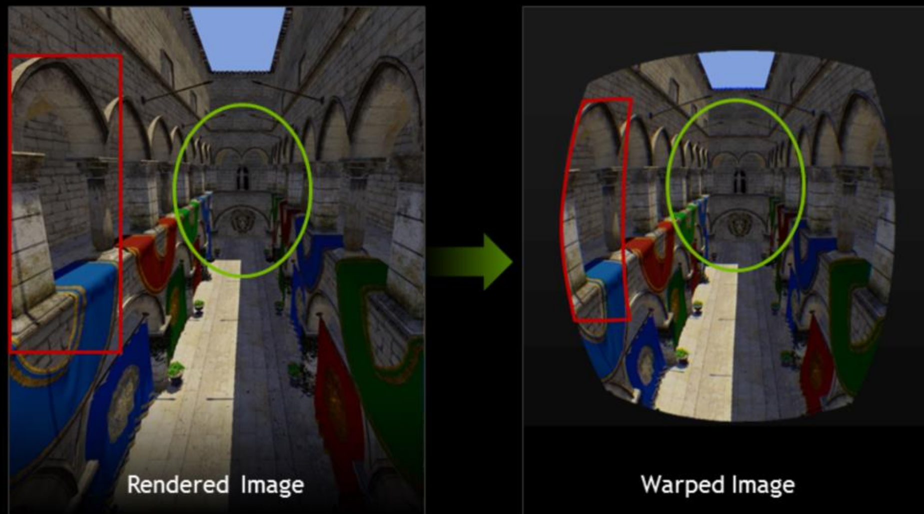
The basic problem that we're trying to solve is illustrated here. The image we present on a VR headset has to be warped to counteract the optical effects of the lenses.

In this image, everything looks curved and distorted, but when viewed through the lenses, the viewer perceives an undistorted image.

The trouble is that GPUs can't natively render into a distorted view like this – it would make triangle rasterization vastly more complicated.



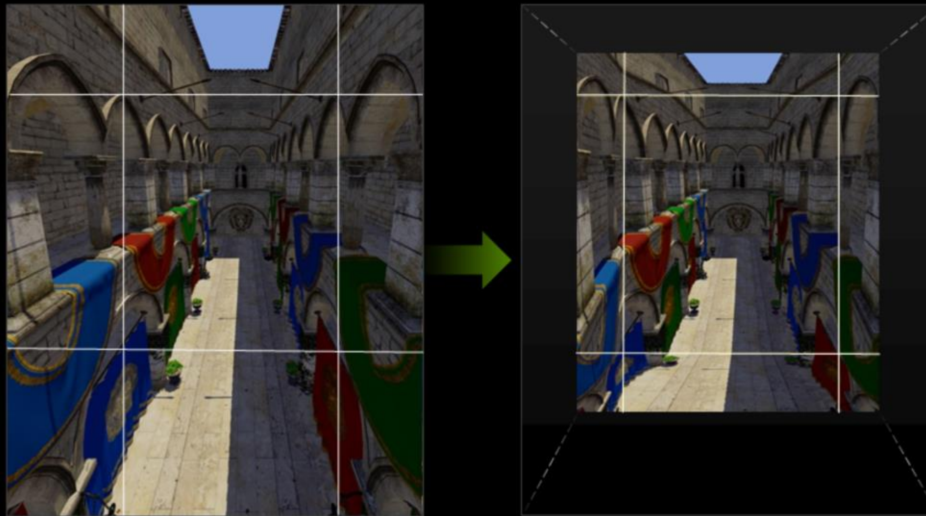
Current VR platforms all solve this problem by first rendering a normal image (left) and then doing a postprocessing pass that resamples the image to the distorted view (right).



If you look at what happens during that distortion pass, you find that while the center of the image stays the same, the edges are getting squashed quite a bit.

This means we're over-shading the edges of the image. We're generating lots of pixels that are never making it out to the display – they're just getting thrown away during the distortion pass. That's wasted work and it slows you down.

MULTI-RESOLUTION SHADING

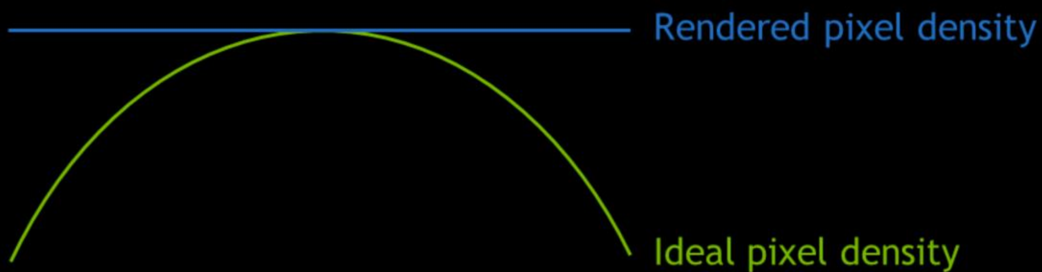


gameworks.nvidia.com

The idea of multi-resolution shading is to split the image up into multiple viewports – here, a 3x3 grid of them.

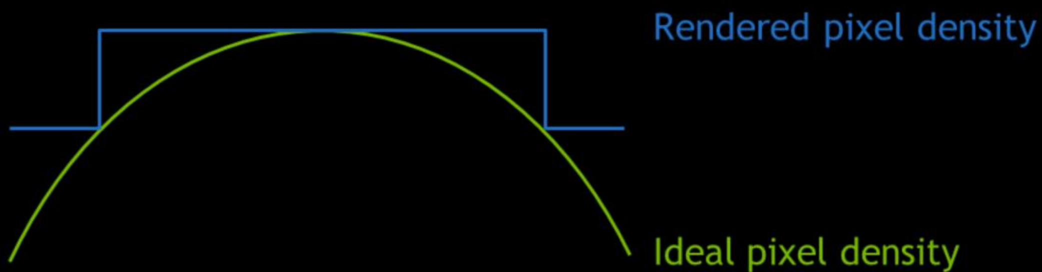
We keep the center viewport the same size, but scale down all the ones around the edges.

This better approximates the warped image that we want to eventually generate, but without so many wasted pixels. And because we shade fewer pixels, we can render faster. Depending on how aggressive you want to be with scaling down the edges, you can save anywhere from 25% to 50% of the pixels. That translates into a 1.3x to 2x pixel shading speedup.



Another way to look at what's going on here is as a graph of pixel density aross the image.

The green line represents the ideal pixel density needed for the final warped image. With standard rendering, we're taking the maximum density – which occurs near the center – and rendering the whole image at that high density.

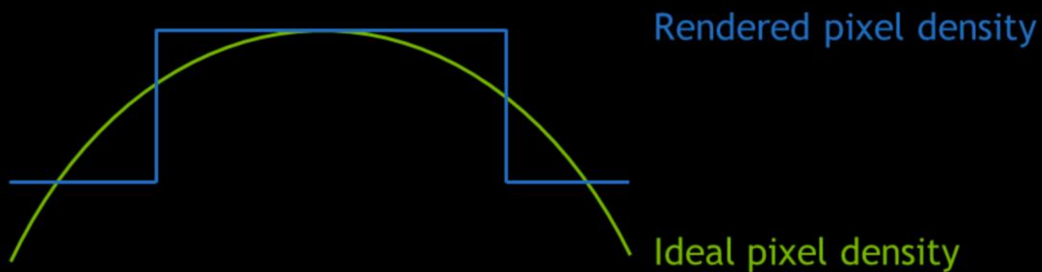


Multi-resolution rendering allows us to reduce the resolution at the edges, to more closely approximate the ideal density while never dropping below it. In other words, we never have less than one rendered pixel per display pixel, anywhere in the image.

This setting lets us save about 25% of the pixels, which is equivalent to 1.3x improvement in pixel shading performance.

AGGRESSIVE MULTI-RES

50% pixels saved = 2x pixel shading speedup

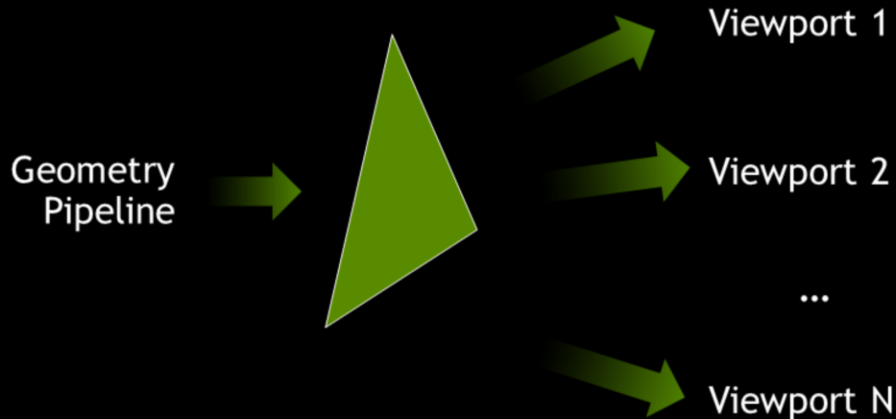


gameworks.nvidia.com

It’s also possible to go to a more aggressive setting, where we reduce the resolution in the edges of the image even further. This requires some care, as it could visibly affect image quality, depending on your scene. But in many scenes, even an aggressive setting like this may be almost unnoticeable – and it lets you save 50% of the pixels, which translates into a 2x pixel shading speedup.

FAST VIEWPORT BROADCAST

Maxwell multi-projection



gameworks.nvidia.com

The key thing that makes this technique a performance win is the multi-projection hardware we have on NVIDIA’s Maxwell architecture – in other words, the GeForce GTX 900 series and Titan X.

Ordinarily, replicating all scene geometry to a number of viewports would be prohibitively expensive. There are a variety of ways you can do it, such as resubmitting draw calls, instancing, and geometry shader expansion – but all of those will add enough overhead to eat up any gains you got from reducing the pixel count.

With Maxwell, we have the ability to very efficiently broadcast the geometry to many viewports in hardware, while only running the GPU geometry pipeline once per eye.

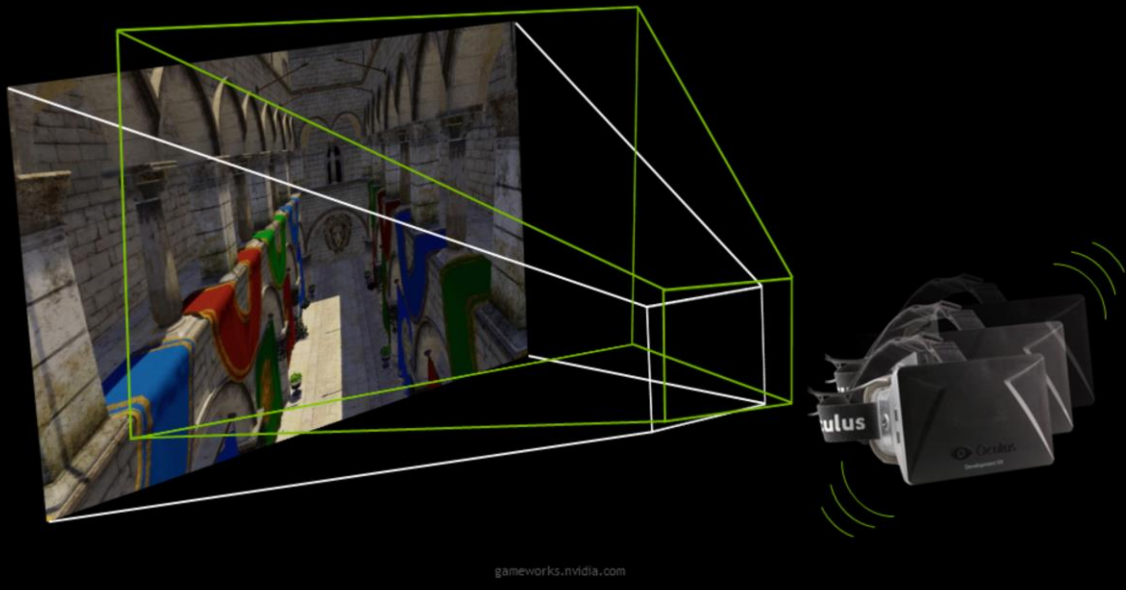
CONTEXT PRIORITY

- Enable VR platform vendors to implement asynchronous timewarp
- Via GPU preemption

gameworks.nvidia.com

Context priority is a low-level feature that NVIDIA provides to enable VR platform vendors to implement asynchronous timewarp. (Note that we don't implement async timewarp ourselves – we just provide the tools for the VR platform to do it.)

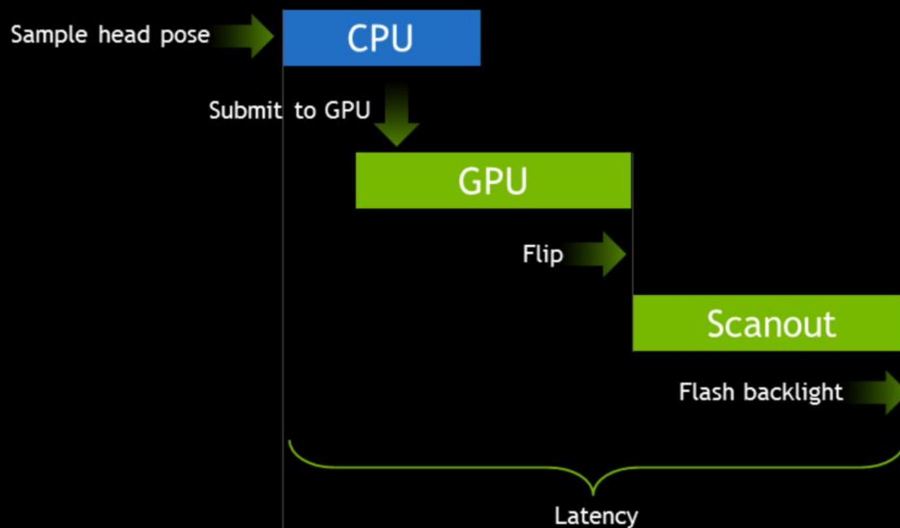
The way we do this is by enabling GPU preemption using a high-priority graphics context.



Before I can explain what asynchronous timewarp is, first I have to talk about ordinary timewarp.

Timewarp is a feature implemented in the Oculus SDK that allows us to render an image and then perform a postprocess on that rendered image to adjust it for changes in head motion during rendering. Suppose I've rendered an image, like you see here, using some head pose. But by the time I'm finished rendering, my head has moved and is now looking a slightly different direction. Timewarp is a way to shift the image, as a purely image-space operation, to compensate for that. If my head goes to the right, it shifts the image to the left and so on.

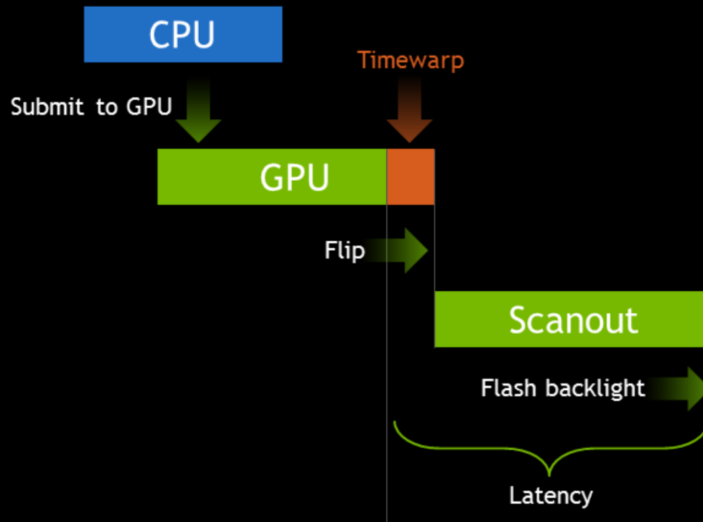
Timewarp can sometimes produce minor artifacts due to the image-space warping, but nonetheless it's startlingly effective at reducing the perceived latency.



Here's what the ordinary graphics pipeline looks like, without timewarp. The CPU does some work to set up a frame to be rendered. Ordinarily, the head pose is sampled early in the CPU frame and baked into the command buffer.

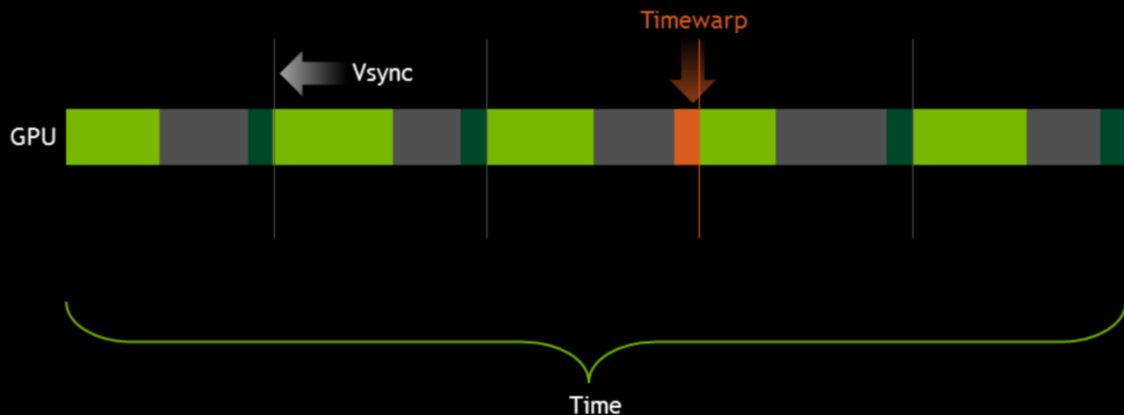
Then the frame is submitted to the GPU, which renders it. At the next vsync, the display is flipped to the new frame just rendered, and the frame is scanned out. Then the backlight on the display is flashed, at which point the user actually sees the image. (This is assuming a low-persistence display with a global flash, which VR headsets are converging to.)

As you can see, there's quite a few milliseconds of latency between when the head pose is sampled on the CPU, and when the user finally sees it.



When timewarp is enabled, we can re-sample the head pose a few milliseconds before vsync, and warp the image that we just finished rendering to the new head pose.

This enables us to cut perceived latency down by a large factor.

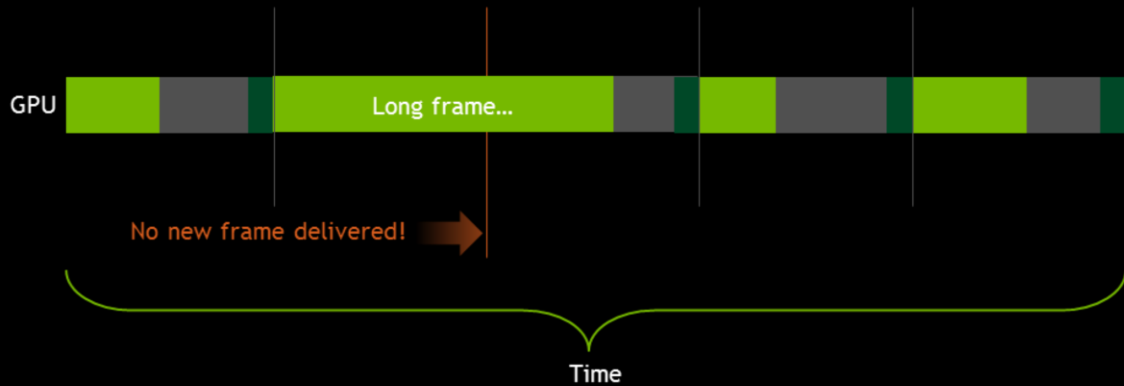


If you use timewarp and your game is rendering at a steady framerate, this is what the timing looks like over several frames.

The green bars represent the main game rendering, which takes variable amounts of time as you move around and different objects show up on the screen. After the game rendering is done for each frame, we wait until just before vsync before kicking off timewarp, represented by the little cyan bars. This works great as long as the game is running consistently within the vsync time limit.

NVIDIA.

HITCHING

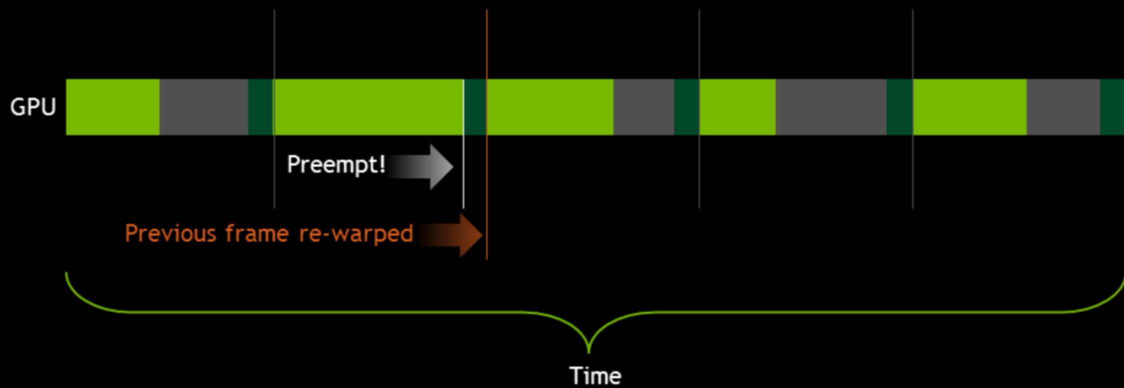


gameworks.nvidia.com

However, games never run 100% consistently at vsync. PC operating systems are fundamentally not capable of guaranteeing that. Every once in awhile, Windows will decide to start indexing files in the background or something, and your game will get held up, producing a hitch.

Hitches are always annoying, but in VR they’re really bad. Having the previous frame stuck on the headset produces instant dizziness.

ASYNC TIMEWARP



gameworks.nvidia.com

That’s where asynchronous timewarp comes in. The idea here is to make timewarp not have to wait for the app to finish rendering. Timewarp should behave like a separate process running on the GPU, which wakes up and does its thing right before vsync, every vsync, regardless of whether the app has finished rendering or not.

If we can do that, then we can just re-warp the previous frame whenever the main rendering process falls behind. So we don’t have to live with a stuck image on the headset; we’ll continue to have low-latency head tracking even if the app hitches or drops frames.

NVIDIA.

HIGH-PRIORITY CONTEXT

- NVIDIA supports high-priority graphics context
  - Preempts other GPU work
- Main rendering → normal context
- Timewarp rendering → high-pri context

gameworks.nvidia.com

The question for NVIDIA is how can we enable VR platform vendors to implement async timewarp on current hardware?

We're doing this by exposing support in our driver for a feature called a high-priority graphics context. It's a separate graphics context that takes over the entire GPU whenever work is submitted to it – pre-empting any other rendering that may already be taking place.

With this feature exposed, an app can submit its work to an ordinary rendering context, while the VR platform service can run a second, high-priority context where it can submit timewarp work.

PREEMPTION

- Current GPUs: draw-level preemption
- Can only switch at draw call boundaries!
- Long draw can delay context switch

gameworks.nvidia.com

Since we’re relying on preemption, let’s talk a bit about its limitations. All our GPUs for the last several years do context switches at draw call boundaries. So when the GPU wants to switch contexts, it has to wait for the current draw call to finish first. So, even with timewarp being on a high-priority context, it’s possible for it to get stuck behind a long-running draw call on a normal context. For instance, if your game submits a single draw call that happens to take 5 ms, then async timewarp might get stuck behind it, potentially causing it to miss vsync and cause a visible hitch.

NVIDIA.

DEVELOPER GUIDANCE

- Still try to render at native framerate! (90 Hz)
  - Better experience
  - Async timewarp is a safety net
- Long draws could cause hitches
  - Split up draws that take >1 ms or so
  - E.g. heavy postprocessing: split in screen-space tiles

gameworks.nvidia.com

So what should developers take away from this? First of all, I want to caution you against relying on async timewarp to “fix” low framerates. It’s not a magic wand for performance issues; it’s a safety net in case of unpredictable hitches or performance drops. Rendering at the native framerate is still going to give people a better experience for a variety of reasons, so you should try to ensure you hit 90 Hz as much as possible.

And, like I just mentioned, watch out for long draw calls. If you have individual draw calls taking longer than 1 ms or so, there’s a danger that they could interfere with async timewarp. To avoid that, you can split the draws up into smaller ones. For example, a postprocessing pass where you’re running a heavy pixel shader over the whole screen might take several milliseconds to complete, but you can split up the screen into a few tiles and run the postprocessing on each tile in a separate draw call. That way, you provide the opportunity for async timewarp to come in and preempt in between those draws if it needs to.

DIRECT MODE

- Prevent desktop from extending onto VR headset
- Hide display from OS, but let VR apps render to it
- Better user experience

gameworks.nvidia.com

Direct mode is another feature targeted at VR headset vendors. It enables our driver to recognize when a display that's plugged in is a VR headset as opposed to a screen, and hide it from the OS so that the OS doesn't try to extend the desktop onto the headset (which no one ever wants). Then the VR platform service can take over the display and render to it directly. This just provides a better user experience in general for working with VR.

FRONT BUFFER RENDERING

- Normally not accessible in D3D11
- Direct Mode enables access to front buffer
- Enables low-level latency optimizations
  - Render during vblank
  - Beam-racing

gameworks.nvidia.com

Direct mode also enables front buffer rendering, which is normally not possible in D3D11. VR headset vendors can make use of this for certain low-level latency optimizations.

GAMEWORKS VR

Faster performance, lower latency, and better compatibility

MULTIRES SHADING

Increase performance via an innovative new way to render for VR

VR SLI

Scale performance with multiple GPUs

CONTEXT PRIORITY

Minimize head tracking latency with asynchronous time warp

DIRECT MODE

Plug and play compatibility from GPU to HMD

FRONT BUFFER RENDERING

Reduce latency by rendering directly to the front buffer

gameworks.nvidia.com

And that's where GameWorks VR comes in. GameWorks VR refers to a package of hardware and software technologies NVIDIA has built to attack those two problems I just mentioned — to reduce latency, and accelerate stereo rendering performance -- ultimately to improve the VR experience. It has a variety of components, which we'll go through in this talk.

API/PLATFORM/HW SUPPORT

- Currently D3D11 only
  - OpenGL and other APIs: later
- Windows 7+
- Multi-res shading: GTX 900 series+ only!
- Everything else: GTX 500 series+
- NDA developer SDK available now

gameworks.nvidia.com

Before we leave I just want to mention which APIs, platforms, and hardware will support these technologies I've been talking about.

The APIs we've been talking about are currently implemented for D3D11 only. We will be bringing support for these features to OpenGL, D3D12, and Vulkan in the future.

GameWorks VR will work on Windows 7 and up, and as far as hardware is concerned, most features will work on all GeForce GTX 500-series cards and up. That's the Fermi, Kepler, and Maxwell architectures, if you follow our internal codenames. The only exception there is that multi-res shading requires a GTX 900 series card, since it depends on the fast geometry shader that only the Maxwell architecture has.

We have an NDA developer SDK with GameWorks VR available now, so if you're interested in looking at this, get in touch with us and we'll set you up.