Warning: this presentation heavily relies on video recordings.
Visit link below to download full PPTX with videos.
https://advances.realtimerendering.com/s2026/index.html#vrrt

Hello, I'm Michał and I work as a Senior Expert Engineer at Infinity Ward Poland. I'm also the lead ray tracing engineer on Call of Duty franchise.

Today I have for you something that I have been working on and off (mostly off) for good 7 years. After a full prototype in 2021, 5 years later, I finally found time to put everything into *Call of Duty* engine.

Behold...

… Variable Rate Ray Tracing.

If you thought – "oh, so like **variable rate shading,** but for ray tracing?“

It's more than that.

VRRT is an umbrella term I use for couple different functionalities that reallocate rays to fix common screen space denoising issues. This presentation is just as much about denoising as it is about ray tracing.

Consistent image quality and performance are ultimate goals for a dynamic multiplayer action game. If you're familiar with denoising, you might have already guessed where I'm going with this. If not, I'll be elaborating in detail throughout the presentation – with lots of videos!

It's important to establish the context – why do we care about solving specific issues and how did we come up with our solutions? I will establish this context it in two ways.

First, chronologically, we will go over early *Call of Duty* engine ray tracing and denoising experiments and how variable rate shading & ray tracing fits into those.

Second, we'll analyze all denoising issues we came across during our experiments and show why we need to fix them.

Finally, I'll go over **4 main pillars of our VRRT solution.**

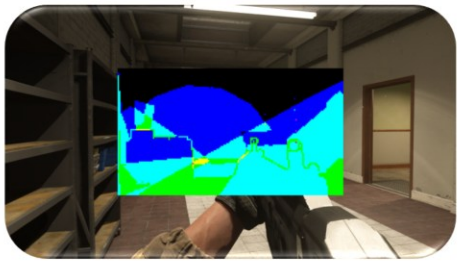
Early RT experiments in *Call of Duty*

SIGGRAPH 2026
LOS ANGELES
19–23 JULY 2026

Early RT experiments in *Call of Duty*

(2019) *Call of Duty: Modern Warfare* (Infinity Ward)

- RT local light shadows (PC)
  - Per-light TLASes
- 1440p @ 60Hz RTX 2070
- Raygen/denoiser @ 720p
- No RT-hit shading



GPU load balancer

- Keeps screen-wide average of 4spp per pixel (720p), adjusts rays per light based on light culling

Raytraced Shadows in Call of Duty: Modern Warfare [Olejnik20]

For our first attempt, we wanted to evaluate viability of ray tracing for the future of the franchise. We choose local light shadows for two reasons: they allowed us to keep BVH simple thanks to per-light TLASes, and they didn't require shading – something that would require larger scale re-work due to lack of bindless support in our F+ engine.

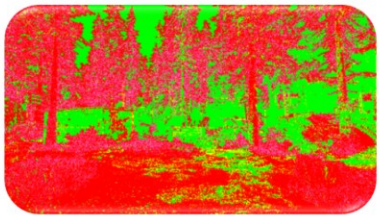
Perhaps our most notable addition to state of the art was the **GPU load balancer**. Before (and even years after our title's release) the industry standard was to do raytraced shadows with 1 denoiser per light, meaning spikes and performance drops when there were more than 1 light on screen. I would compare it to classic deferred lighting, while our approach was raytraced equivalent of tiled light culling. Denoiser could keep track of 4 shadow signals per pixel (so 4 lights per pixel, but those could be different light for each pixel or in different order), and we used light culling output to count average number of light per pixel on whole screen and decide on ray count per light.

You can reference my presentation from 2020 for more details. In the end, the light load balancer did become a **basis for a more generalized load balancer that is one of the pillars of VRRT** (I'm color coding those pillars in light blue on the slides) – we will discuss this soon in more detail.

Early RT experiments in *Call of Duty*

(2020) *Call of Duty: Black Ops Cold War* (Treyarch)

- RT sun/local shadows (PC/PS5/XSX), RTAO (PC)
- 1440p @ 60Hz on consoles
- Raygen/denoiser @ 720p
- No RT-hit shading



VRRT prototype

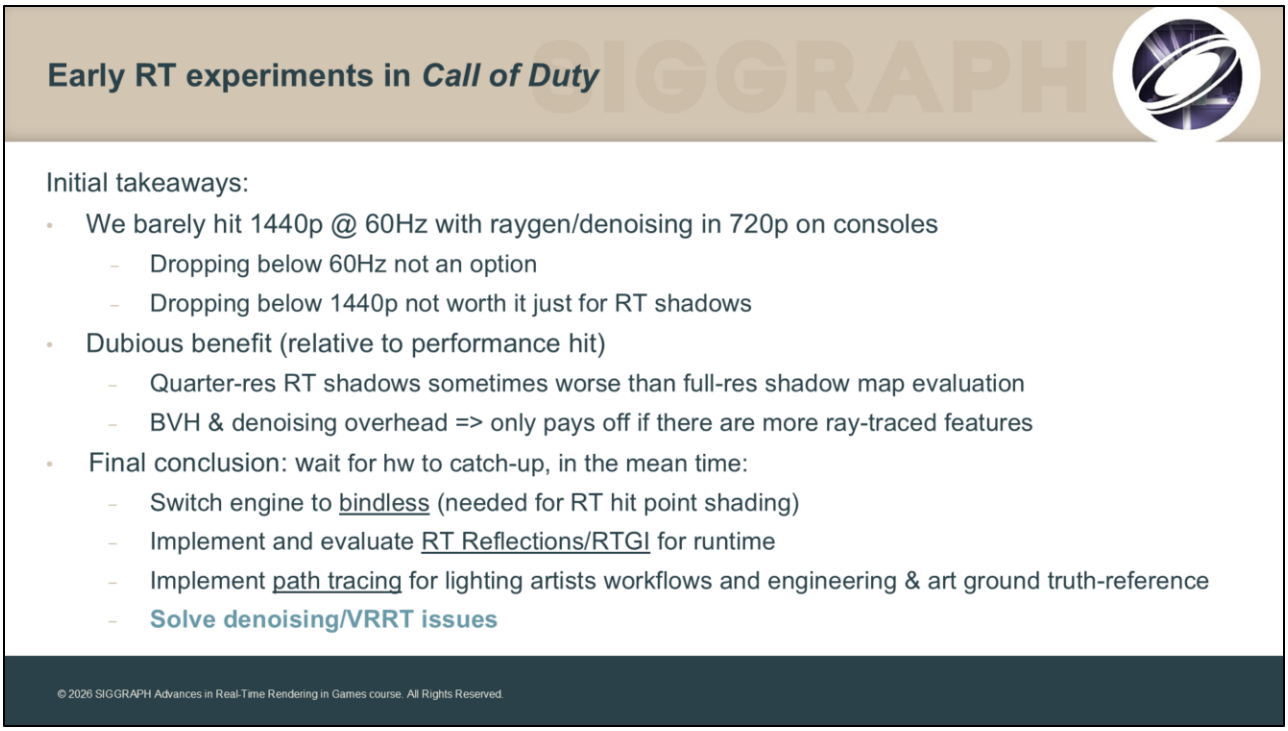
- 720p, with <1spp (1-4 samples per quad)
- Based on *Software-based VRS in Call of Duty: Modern Warfare* [Drobot20]
- Not shipped (history invalidation issues on dynamic shadows)

Shadows of Cold War: A Scalable Approach to Shadowing [Myers21]

At the same time, there was parallel development on *Call of Duty: Black Ops Cold War* lead by Treyarch. We ended up shipping similar features with similar constraints and performance targets. This was also first year of current-gen consoles.

I was asked to help on this project in its last stretch. I quickly realized that raygen performance on consoles was a bit more of a bottleneck than in *Call of Duty: Modern Warfare.* We had an idea to apply principles of variable rate shading to ray tracing - to lower the effective ray counts. As we already had a successful **VRS** implementation in *Call of Duty: Modern Warfare (*was presented by Michał Drobot in 2020 on this course), it became the basis for the VRRT implementation.

Unfortunately, we were already ray tracing and denoising in quarter res (1440p => 720p), and applying VRRT to this to drop resolution/rays even further (720p => 360p) caused issues with variance-based color clamping on dynamic shadows. I'll be elaborating more on this throughout the presentation - the important takeaway is that **we needed more reliable way of detecting motion for VRRT to be successful.**

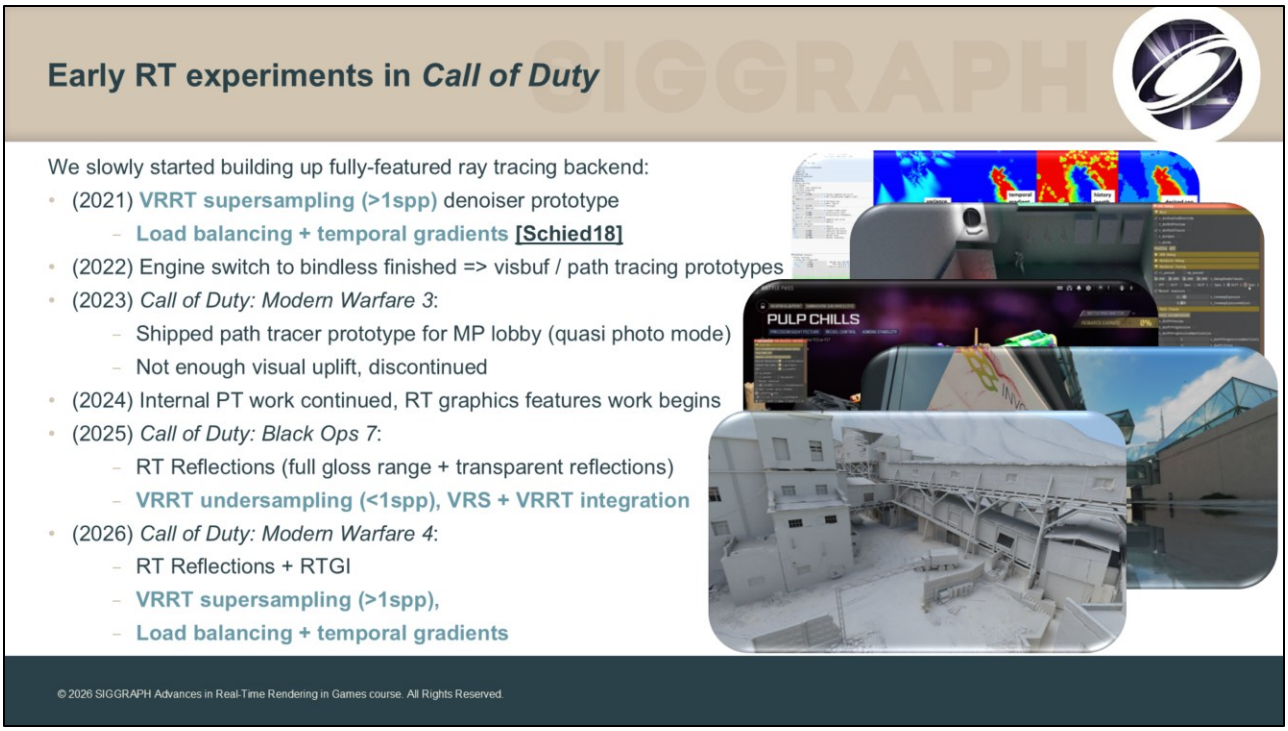


In the end, as primarily a multiplayer console title, we realized that it will be hard to justify ray tracing becoming part of our core engine rendering tech. This was mostly due to our high standards w.r.t. framerate (120hz performance mode / 60hz quality mode) and resolution (needed for clarity during firefights). Ray traced shadows and AO were nice, but they came with major performance penalty – and due to lower resolution of raygen/denoising, they were not even better than shadow maps 100% of the time.

In the end, we decided we can slowly build up our ray tracing backend while waiting for hardware to catch up. Our primary engine development direction was to add bindless materials and geometry so we can do an uber shader approach necessary for performant ray hit shading. This was a bit more difficult for us than majority of other game engines – because we were a primarily F+ pipeline (not deferred) we became reliant on shader permutations and this needed a major rework to find a good balance with RT hit point shading. We were also not satisfied with simplified implementation of bindless shading - we wanted full material feature parity, because our most immediate interest was path tracing for lighting artists and as a reference renderer.

Personally, I wanted to focus on solving screen-space denoising issues that I encountered during those 2 projects – **we had major concerns about impact of denoising artifacts on multi-player gameplay clarity and fairness.**

Sidenote:
It's also worth outlining that **our need for RTGI wasn't that big**, as we already successfully shipped an 8x8km battle royale on PS4 in 1080p/60Hz in 2019 - with fully baked high-quality lighting, leveraging our vast server architecture for baking over network. Over time, we extended this even more.
https://research.activision.com/publications/archives/precomputed-lighting-in-call-of-dutyinfinite-warfare
https://research.activision.com/publications/2020-01/ray-guiding-for-production-lightmap-baking
https://research.activision.com/publications/2020-09/the-design-and-evolution-of-the-uberbake-light-baking-system
https://research.activision.com/publications/2021/09/large-scale-global-illumination-in-call-of-duty
https://research.activision.com/publications/2024/08/Neural_Light_Grid



And so, we started long-term raytracing work.

In 2021, while waiting for other people to finish bindless, I spent some time researching **VRRT supersampling** (i.e. to cast more than 1 ray for pixels with invalidated temporal history), including **generalized load balancer** to keep it in budget. I also added **temporal gradients for dynamic event detection,** replacing variance-based color clamping. We will be discussing each of those parts it in detail later.

With this prototype complete, I finally managed to convince myself that we can fix common screen space denoising issues. This allowed us to fully embrace ray tracing future. Alas, this was all done in my prototyping toy engine, and it took years to fully implement in *Call of Duty* engine.

In 2022, with basic building blocks for bindless finished, we created a prototype for a simplified path tracer (simple material model), as well as visibility buffer shading (which finally ships this year in *Call of Duty: Modern Warfare 4*, meaning we will have a hybrid F+/V+ raster pipeline).

In 2023, Nvidia released DLSS-RR (machine learning denoiser + upsampler combo), and we collaborated on a little path tracing experiment for Call of Duty: Modern Warfare 3. We choose our MP lobby as a quasi path traced photo mode, where we could push path tracing to its limit while showcasing it on our hero assets (guns +

characters) in a limited environment. In the end, it turned out that all the incredible effort that artists and engineers across Activision put into our renderer throughout the years meant that path tracing didn't add that much visual uplift, so we decided to abandon this for future releases.

Improving our path tracer still yielded major benefits for our internal tooling, and in 2024 we finally had time to start on runtime ray tracing features.

In 2025, we finally shipped RT reflections on PC, supporting full range of gloss as well as reflections on transparencies (glass etc.). It also shipped with first basic version of VRRT that had **undersampling with temporal reconstruction** – this is the work I initially failed to make work on Call of Duty: Black Ops Cold War (2020) for shadows.

This year, we want to add RTGI on top of it for *Call of Duty Modern Warfare 4*. The denoiser now also supports full VRRT, with all the pillars from my initial 2021 prototype in place. It also supports **multi signal denoising**, similar to multi-shadow signal denoising in *Call of Duty: Modern Warfare* (2019), but generalized for more workloads, i.e. Reflections + GI for purposes of retail title, but also e.g. 4 signals for our internal path tracer (direct/indirect diffuse/specular).

Sidenote:
It's also worth mentioning that fully fledged RTGI that replaces entire baked lighting is not always a 100% win. In our case, we managed to utilize our baked lighting to amortize direct lighting as well – for example, we have a concept of static lights (bake only, for performance optimizations) and residual lighting – e.g. our lights have finite range in real-time, but the physically correct infinite 1/r^2 falloff is being put in lightmaps. We can't reliably replicate this with RTGI without adding significant cost, and the differences are notable, so expect a hybrid RTGI-baked solution.

Those are the 4 main pillars that I consider to be parts of **VRRT**, and they all aim at improving various aspects of screen space denoising:

- **Supersampling** in places where temporal denoiser didn't converge – to get rid of artifacts caused by lack of history
- **Undersampling** with temporal reconstruction – to lower the ray rate in 'safe' areas (i.e. those with full temporal history) and reallocate them to more important areas
- **Load balancer** – to maintain fixed millisecond budget, and allow supersampling & undersampling to balance each other
- Improved dynamic event detection via **temporal gradients**

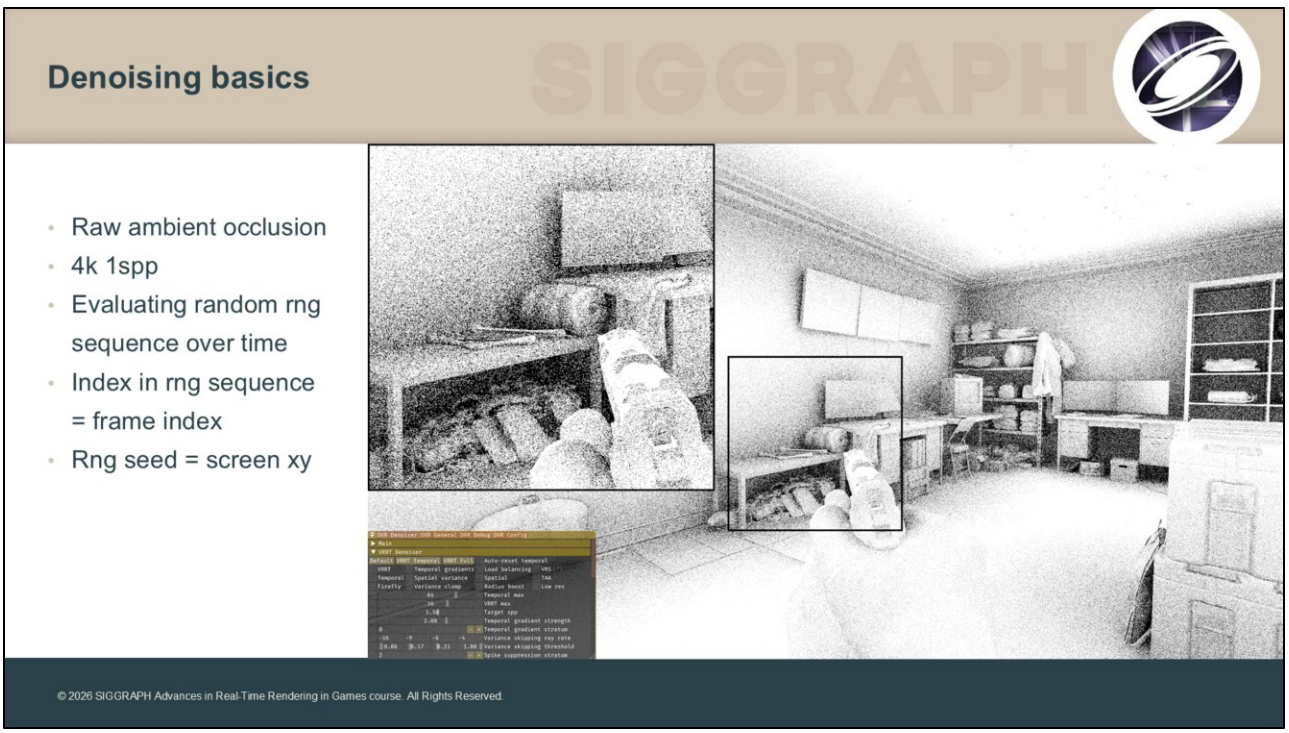
When I presented originally on SIGGRAPH, I asked the audience if there is anybody who knows what **temporal gradients** are – only one person raised their hand. They were developed by Christoph Schied for Q2VKPT/Q2RTX, and the original papers were released before first ray tracing hardware. Perhaps this is the reason they flew under the rader. I consider them an **extremely powerful tool for dynamic event detection.** It's a shame they never became common place in our industry – I think it's a mistake, and I'll spend more time at the end of the presentation explaining why you should try them out if you're serious about denoising.

But first things first - let's now go over denoising basics and common artifacts, and then we'll discuss how **VRRT** fixes those issues.

Denoising basics

SIGGRAPH 2026
LOS ANGELES
19–23 JULY 2026

Denoising basics



[video] Here's something you're probably familiar even if you never worked on RT – raw, noisy, importance sampled signal for ambient occlusion with 1 sample per pixel – this is what we need to make smooth via temporal and spatial filters. Every game engine already does something like this for non-RT features like SSR or SSAO. The difference with ray tracing is that we traces are more expensive, so we're usually limited to ~1spp – it's way more noisy.

Important takeaway is how we evaluate our random numbers for importance sampling. We treat it as progressive renderer (even if camera moves), and we evaluate random sequence over time (rng index = frame index). Each pixel has different seed (determined by pixel xy coordinates), so each pixel evaluates different uncorrelated random sequence. This is effectively white noise, but it makes it easier to apply VRRT later.

Sidenote:
If you're new to denoising, I'd recommend starting with Dimitri Zhdan's and Christoph Schied's work – two different schools of denoising. VRRT itself was based on SVGF/ASVGF papers from Schied. There are also publicily available denoisers for Q2VKPT/Q2RTX and NRD on github that you can study. Please refer to bibliography for links.

Denoising basics

- Temporal accumulation
- 64 frames
- lerp(prev, curr, 1/n)
- Disocclusion detection
  => reset frame counter
- See **[Tardiff]**

[video] Here you can see temporal denoising in action

- We can denoise raw signal via temporal accumulation.
- We only need to track current and previous frame color buffer, and we use exponential smoothing to combine them.
  https://en.wikipedia.org/wiki/Exponential_smoothing
- We also store number of samples accumulated inside ray tracing payload and reproject it.
- We detect disocclusion, and reset the counter when it happens.
- Suffers from temporal lag – more on how to combat it later.

Denoising RT lighting is simpler than temporal anti-aliasing or upsampling, because we don't need to resolve edges

- We can test each texel In reprojection footprint via depth/normal bilateral filter and reject them individually (code provided later in the presentation)
- It's still worth understanding the basics behind TAA – see links in bibliography

Denoising basics

- Spatial blur r=8
- Atrous filter **[Dammertz10]**
- Edge detection
- Unstable
- Biased

[video] And here's spatial denosing with small blur radius

- To avoid temporal lag, we can instead blur the signal with any spatial filter.
- The most popular one is atrous ('with holes') edge-stopping wavelet filter, but you can use anything - only need to perform bilateral edge detection to avoid blurring over geometry edges.
  Of course, this will introduce bias to our signal, as we are combining importance sampled data from different world space positions.
- On top of it, it will be unstable if you use radius that is too small.

Sidenote:
You can try to combat this by using repeatable rng seed in small spatial window (with radius equal to spatial filter size), but this introduces different kind of bias.

Denoising basics

- Spatial blur r=32
- Stable enough for TAA
- Even more biased
  - Overblurs
  - Detail loss

[video] You can increase spatial filter radius, but this increases bias. It can still be unstable, but usually low-frequency instabilities are ok as inputs to temporal anti-aliasing/upsampling.

Side note:

- You can also use dedicated temporal stabilization pass instead (like TAA, but with wider variance clamp) [Zhdan20]
- Temporal before spatial is called accumulation
- Temporal after spatial is called stabilization

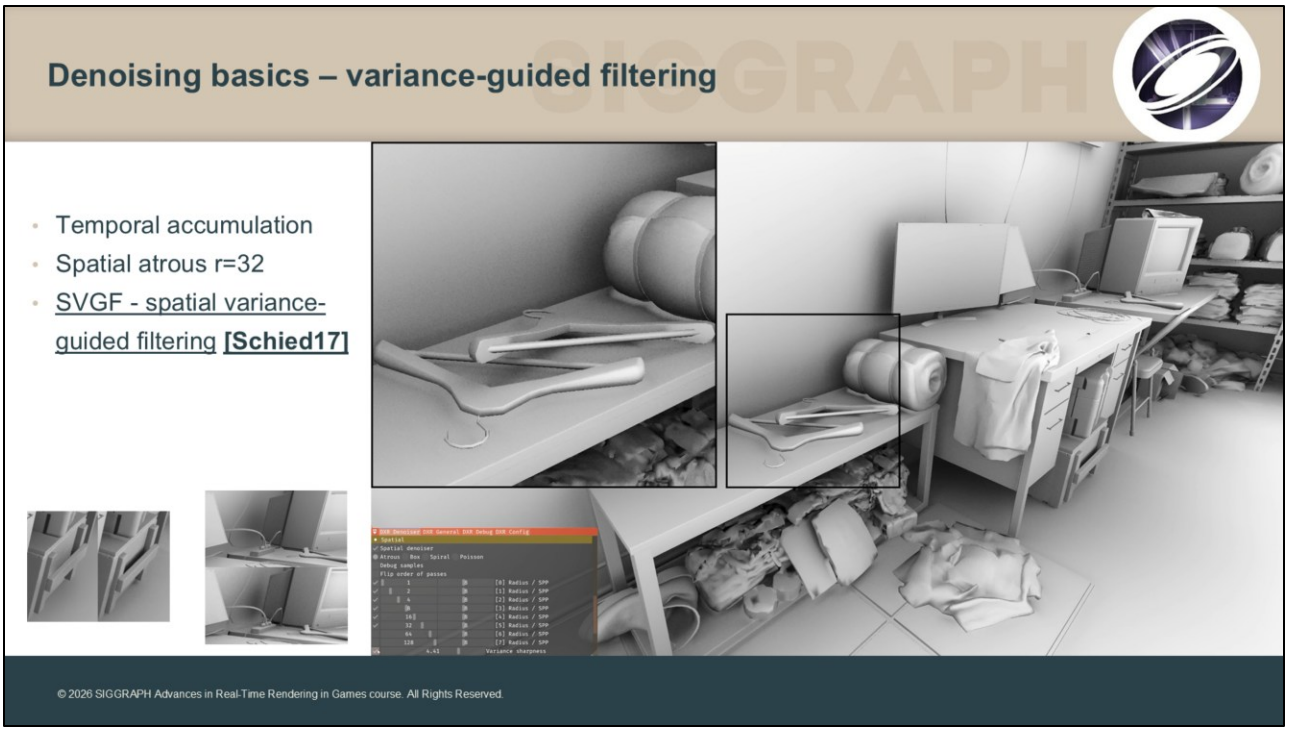
Doing only temporal after spatial means that you're giving up on per-pixel detail from the start – that's the opposite of we want in *Call of Duty*, but still viable for low-frequency signals like GI if you want to compute them cheaply. State of the art of this approach was done last year by *Doom: The Dark Ages GI*, which later combines it with SSAO to reclaim some of the lost detail [Sousa25]

Denoising basics

- Temporal accumulation
- Spatial blur r=8
- Good enough...
- ...in simple cases (AO)
- Still rough for MP game

[video] Of course, you can do both temporal and spatial at the same time. By doing temporal accumulation with smaller spatial blur, you will only see the instability during couple first frame after disocclusion.

This is often acceptable with simpler signals like AO. But even so, in *Call of Duty* most of the action is around your weapon that constantly dissoccludes environment and enemy players, which negatively impacts visual clarity and therefore gameplay.



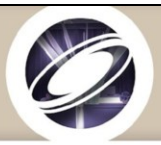
[video] The real holy grail would be to allow for large spatial filter that stabilizes the signal, while simultaneously not overblurring the high-frequency detail. And this was already solved by Christoph Schied in 2017- so before first ray tracing GPU - with *SVGF*, which has now become the industry standard.

Details in a moment, but let's see what it looks like on video – it might be a bit overwhelming, so I advise you to pause video and compare different timestamps.

- I start by stacking multiple atrous blur passes, increasing effective spatial filter radius
- After that, I change the tuneable 'variance sharpness' parameter, and toggle it back and forth multiple times
- You can see how the sharpness parameter affects spatial filter strength and preservers indirect contact shadows even with large blur

Note that we need multiple samples to estimate signal variance, that estimate is done over time by analyzing temporal history – I'll explain soon why it's a problem.

Denoising basics – variance-guided filtering



- Variance is computed from moments
- Just need to track *signal* + *signal*^2
- Classic temporal anti aliasing techniques use variance to reduce ghosting (clamp previous frame color)
- See [**Tardiff**]

```
float ClassicTAA( uint2 dtid )
{
    float curr = currTexture[dtid];
    float prev = ReprojectHistory( prevTexture );

    float m1 = 0;
    float m2 = 0;
    for ( int i = -1; i <= 1; ++i )
    for ( int j = -1; j <= 1; ++j )
    {
        float tap = currTexture[dtid + int2(i, j)];
        m1 += tap;
        m2 += tap * tap;
    }
    m1 /= 9.0;
    m2 /= 9.0;

    float variance = m2 - m1*m1;
    float stdDev = sqrt( variance );
    prev = clamp( prev, m1 - gSigma * stdDev, m1 + gSigma * stdDev );

    return lerp( curr, prev, 0.95 );
}
```

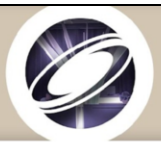

Quick refresher about variance and moments

We need to track signal and signal^2 to calculate variance.

You're probably familiar with some basic TAA variance-based history color clamping. We analyze 3x3 pixel area of current frame color, compute variance, and clamp reprojected previous frame signal to 3x3 mean +- standard deviation (multiplied by some arbitrary scale).

This effectively happens to limit TAA ghosting to 1px neighborhood – which happens to be a bit like anti-aliasing. This approach is not perfect, but the core idea is very elegant.

Denoising basics – variance-guided filtering



- TAA uses 3x3 curr frame neighbourhood
- In denoising, we track moments over time and use variance to guide the spatial blur

```
void Raygen( uint2 dtid : SV_DispatchThreadID )
{
    float result = TraceRay(...);
    float m1 = result;
    float m2 = result * result;
    outTexture[dtid] = float2( m1, m2 );
}

void TemporalDenoiser( uint2 dtid, uint2 reprojectedPos )
{
    float currM1 = currTexture[dtid].x;
    float currM2 = currTexture[dtid].y

    float prevM1 = prevTexture[reprojectedPos].x;
    float prevM2 = prevTexture[reprojectedPos].y;

    float m1 = lerp( currM1, prevM1, 0.95 );
    float m2 = lerp( currM2, prevM2, 0.95 );

    outTexture[dtid] = float2( m1, m2 );
}
```


With *SVGF*, we instead use temporal variance estimate, meaning we track m1/m2 over time instead of in spatial neighborhood. Fortunately, we don't need to track X frames of history, we just interpolate an extra moment in the same way we did with original signal, so it's memory efficient.

Denoising basics – variance-guided filtering

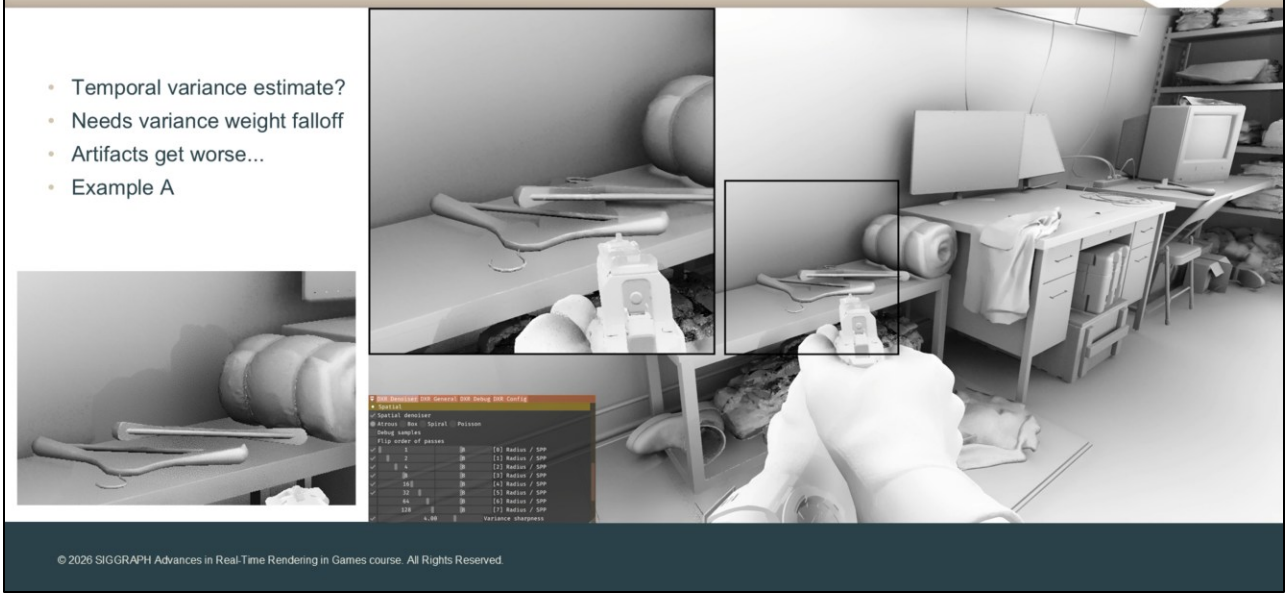
```
float Weight_Variance( float variance, float centerTap, float tap )
{
    float absDelta = abs( centerTap - tap );
    // Smaller variance => the closer signals must be to blend
    // Bigger variance => blend aggressively
    return exp( -gVarianceWeightSharpness * absDelta / variance );
}
float SpatialDenoiser( uint2 dtid )
{
    float currM1 = temporalTexture[dtid].x;
    float currM2 = temporalTexture[dtid].y;
    float variance = currM2 - currM1*currM1;

    float sum = currM1;
    float weights = 1.0;
    for ( int tap = 0; tap < nTaps; ++ tap )
    {
        int2 tapOffset = ...;
        float tapM1 = temporalTexture[dtid + tapOffset].x;
        float w = Weight_Variance( variance, currM1, tapM1 );

        sum += tapM1 * w;
        weights += w;
    }

    return sum / weights;
}
```

If you have an existing spatial bilateral blur in your engine, variance weighting would sit just next to classic depth/normal weighting. It just needs per-pixel variance, which we can calculate from temporal texture.

You can imagine a hard shadow edge. Shadow signal on pixels on one side of it will be all 0s, and all 1s on the other side => variance is 0, blur weight is 0 => you retain hard shadow edge.

If you have a soft shadow, pixels in penumbra will have some combinations of 0s and 1s, so variance will be > 0 and will smooth out pixels inside a penumbra.

Denoising basics – variance-guided filtering



[video] This is what SVGF looks like in action.

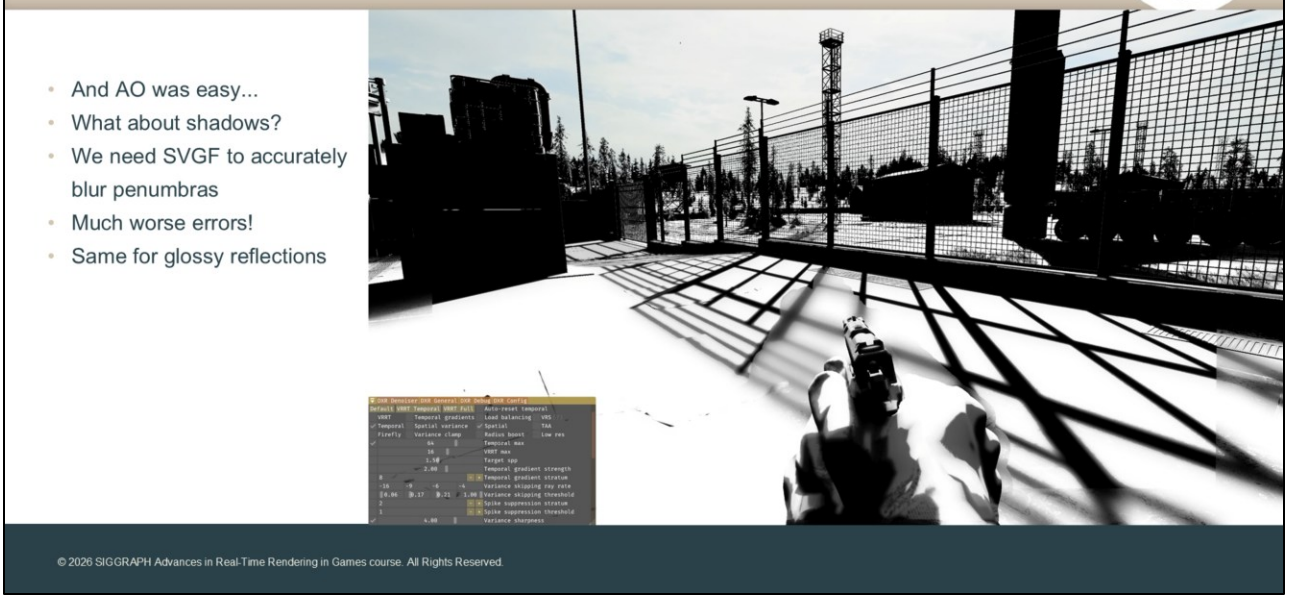
- The obvious problem is what to do when we don't have any temporal history.
- With 1 sample variance is 0, so we need to turn off variance weights to retain any kind of spatial blur.
- What happens in practice is that we will have strong spatial filter until history converges, leading to different denoising strength on neighboring areas depending on accumulated temporal history.
- You can see the artifacts in motion on video.

Denoising basics – variance-guided filtering

- Temporal variance estimate?
- Needs variance weight falloff
- Artifacts get worse...
- Example B

[video] And here is more pronounced example, where you can 'paint' the screen in artifacts by moving your character model around. It's obvious in first person games like Call of Duty.

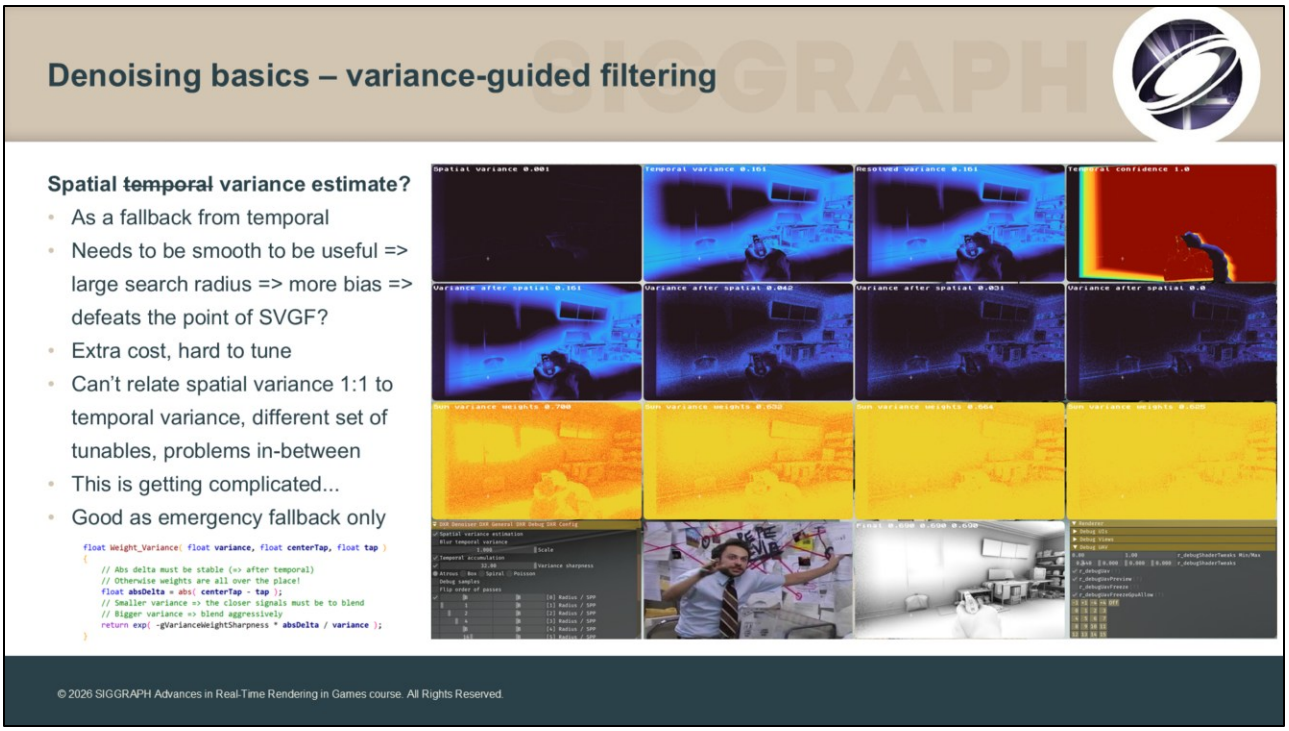
Denoising basics – variance-guided filtering



[video] As I mentioned – AO is probably easiest RT feature to denoise – in fact, we don't really need SVGF just for AO.

But for mixed frequency signals like shadows, we want to have both hard shadow edges and soft penumbras, and those errors become incredibly distracting. We had similar problems with sharp reflections in *Call of Duty: Black Ops 7* (2025).

Denoising basics – variance-guided filtering



The obvious solution is to utilize spatial fallback. I don't want to spend too much time dwelling on this – let's just say it's incredibly hard.

You can see the debug overlay I use to tune this – it tracks temporal/spatial variance estimates, the temporal confidence factor used to blend between those 2 estimates, as well as variance after each spatial atrous iteration and effective blur weights (that are smaller with each iteration due to decreasing variance).

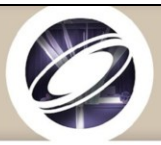
Even with such tool, it's too much for a human to tune. This is where machine learning techniques excel. However, even ML approach will still be **fundamentally limited by the fact that temporal history is empty or short**. What if we could fix that instead? You probably remember that I mentioned using temporal supersampling for such cases. We'll get to this soon.

The takeaway here is that **we can't really escape the consequences of short temporal history**

- Spatial filter is unstable when using small radius with short history
- Increasing radius causes bias
- Fixing bias with *SVGF* still leaves issues with variance estimate with short history

The reason why I'm spending time talking about SVGF is because if you need SVGF on top of classic spatio/temporal denoiser, it adds another *enormous* reason to care about solving denoising artifacts.

Denoising basics – history invalidation



- **Disocclussion** is not the only source of instability
- We also need to cancel temporal history on motion
  - Shadows/reflections from moving objects (characters, foliage...)
    - Need to react quickly
  - AO/GI lagging can be ok, depends on the context, e.g.:
    - Dynamic time of day – ok
    - Occlusion between characters and environment – BAD!

- **Disocclusion** is easy to detect – and already difficult to handle
- **Invalid history** is hard to detect – and equally difficult to handle

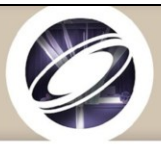
But what's state of the art in invalid history detection?

We talked a lot about disocclusion, but temporal history is also cancelled (or rather: it should be cancelled) when we detect dynamic events, like moving objects, lights, animated textures etc.

This is crucial for shadows and reflections, a bit less so for low-frequency signals like AO/GI. You could completely ignore temporal lag from animated sky, and nobody would notice, but occlusion between characters and environments (especially near shoes) makes lag very noticeable.

Disocclusion was at least easy to detect, invalid history requires extra work. And if we don't do it perfectly, errors will compound through the denoising pipeline. Unfortunately, the most known state of the art technique is not particularly impressive.

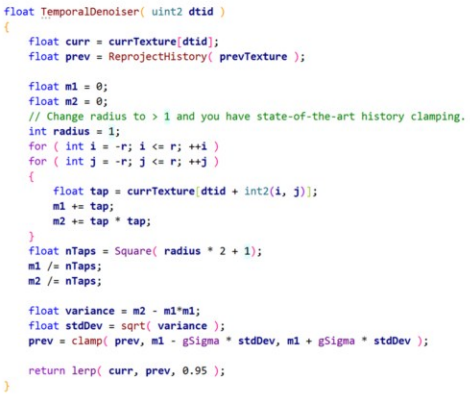
Denoising basics – history invalidation



```
float ClassicTAA( uint2 dtid )
{
    float curr = currTexture[dtid];
    float prev = ReprojectHistory( prevTexture );

    float m1 = 0;
    float m2 = 0;


    for ( int i = -1; i <= 1; ++i )
    for ( int j = -1; j <= 1; ++j )
    {
        float tap = currTexture[dtid + int2(i, j)];
        m1 += tap;
        m2 += tap * tap;
    }

    m1 /= 9.0;
    m2 /= 9.0;

    float variance = m2 - m1*m1;
    float stdDev = sqrt( variance );
    prev = clamp( prev, m1 - gSigma * stdDev, m1 + gSigma * stdDev );

    return lerp( curr, prev, 0.95 );
}
```


Remember TAA history clamping to stop ghosting? How would we apply this concept to temporal denoising?

Denoising basics – history invalidation



```
float TemporalDenoiser( uint2 dtid )
{
    float curr = currTexture[dtid];
    float prev = ReprojectHistory( prevTexture );

    float m1 = 0;
    float m2 = 0;
    // Change radius to > 1 and you have state-of-the-art history clamping.
    int radius = 1;
    for ( int i = -r; i <= r; ++i )
    for ( int j = -r; j <= r; ++j )
    {
        float tap = currTexture[dtid + int2(i, j)];
        m1 += tap;
        m2 += tap * tap;
    }
    float nTaps = Square( radius * 2 + 1);
    m1 /= nTaps;
    m2 /= nTaps;

    float variance = m2 - m1*m1;
    float stdDev = sqrt( variance );
    prev = clamp( prev, m1 - gSigma * stdDev, m1 + gSigma * stdDev );

    return lerp( curr, prev, 0.95 );
}
```


Increase the spatial radius for variance analysis window and this is pretty much state of the art.

Of course, there is some nuance, and various techniques that help to improve variance-based color clamping. But ultimately, we're using a technique not originally created for our problem (temporal accumulation of noisy signal), and we face fundamental limitations.

Sidenote:
Relying just on color and its variance information is very limiting. Consider this: we do have information about per-object (even per-triangle) motion in our game engines – why are we not utilizing it for denoising? We'll revisit this train of though later with **temporal gradients** (one of VRRT pillars).

Denoising basics – history invalidation

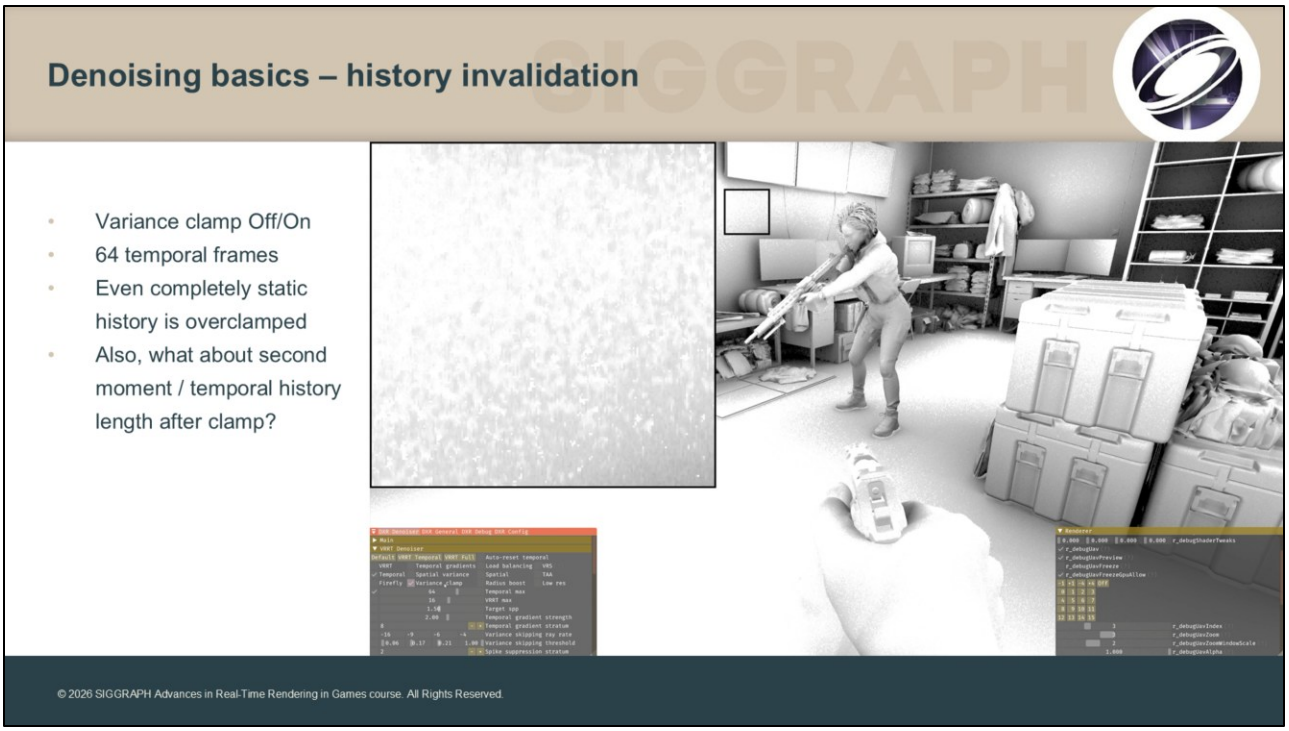
[video] You can see one of my denoising pet peeves – AO lag near character's shoes. This is without any history rejection.

Denoising basics – history invalidation

[video] We can apply variance-based clamp, but we'll never get it 100% right. It's especially quite common in games with crowds.

Only tunables we have are spatial radius and scale of standard deviation for our clamping window – not very human brain friendly.

You can also try toggling between this and previous slide to notice some differences even in areas without motion.



[video] Turns out variance clamp can easily over-clamp even static signal, as evident on video where I toggle the clamp on and off, after all, it only works on color. Here, I used longer temporal history length. To properly clamp, we should increase size of variance analysis window to match the length of history, but that's extra expensive.

Variance-based clamping is not really an invalid history detection scheme, but rectification. With *SVGF*, we also must do something about second moment – we can recompute it based on spatial variance we used for clamp, but it's also a bit of an arbitrary heuristic. There is also no reasonable way to reset/fade-off history counter, as rectification happens every frame – not just once – making it hard to stabilize any history length clamping attempts.

Key takeaway here is that industry standard **variance-based color clamping is simply not good enough** – and a major roadblock when it comes to achieving reliable and high quality temporal denoising.

Denoising basics - summary

Consistency is extremely important for a competitive MP FPS.

- **Disocclusion** is easy to detect
- **Invalid history** is hard to detect
  - this is where **temporal gradients** shine
- By **supersampling** both, we ensure consistent quality

- More reliable **detection** allows to **undersample...**
  - ... and still be able to recover on dynamic event
- With a **load balancer**, we can ensure consistent performance

We went over the basics – artifacts with classic spatio-temporal denoising pipeline, how SVGF can make them worse, and how unreliable variance-based clamping is.

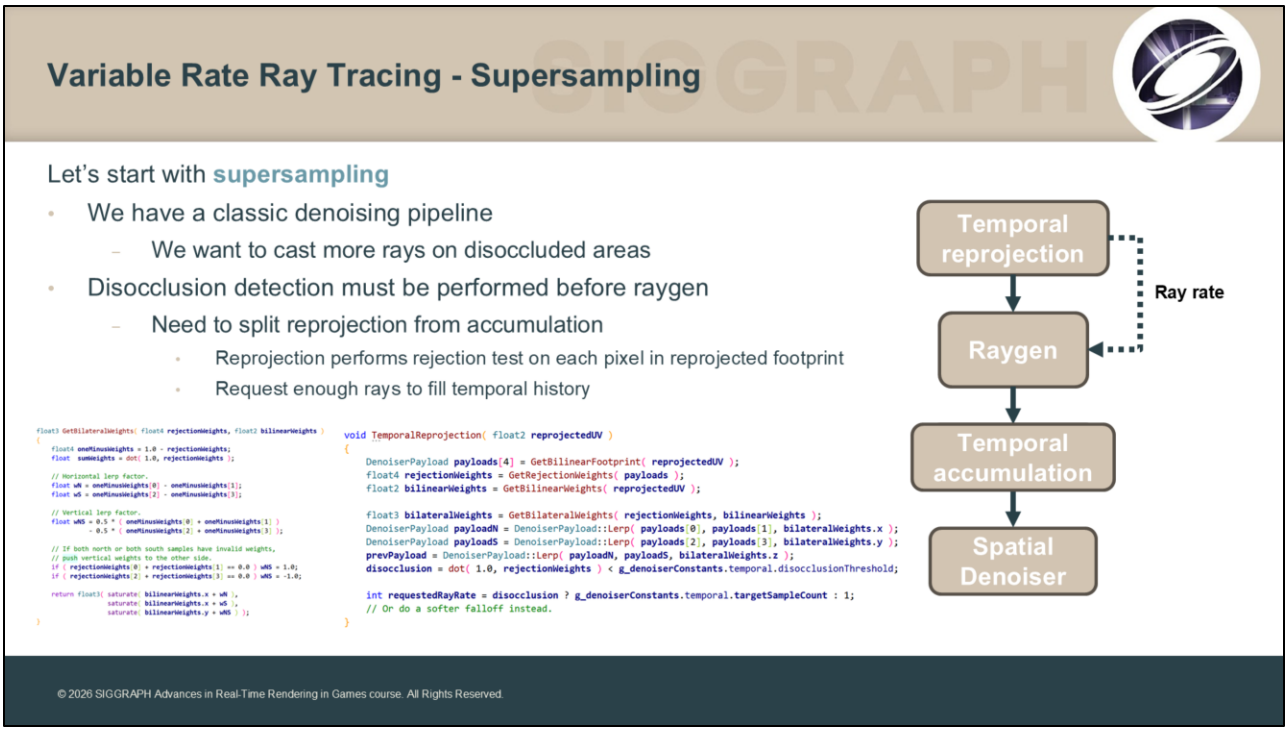
We can now more clearly see how the **VRRT** components are going to work together.

- Disocclusion is easy to detect - and understand, hence why we started with it.
- But invalid history detection is much more complex problem – hence my obsession with **temporal gradients** – so we can get rid of variance-based color clamping.
- Having reliable invalid history detection means we know where to supersample…
- ... and crucially allows us undersample other areas while still beeing able to recover on motion (with clamping, we would have less samples in variance analysis window, making it ineffective).

Therefore, having high-quality invalid history detection is crucial to **VRRT** if we want to truly enforce **consistent quality**. And for **consistent performance**, we need a load balancer.

Let's go over all *VRRT* pillars, starting with supersampling.

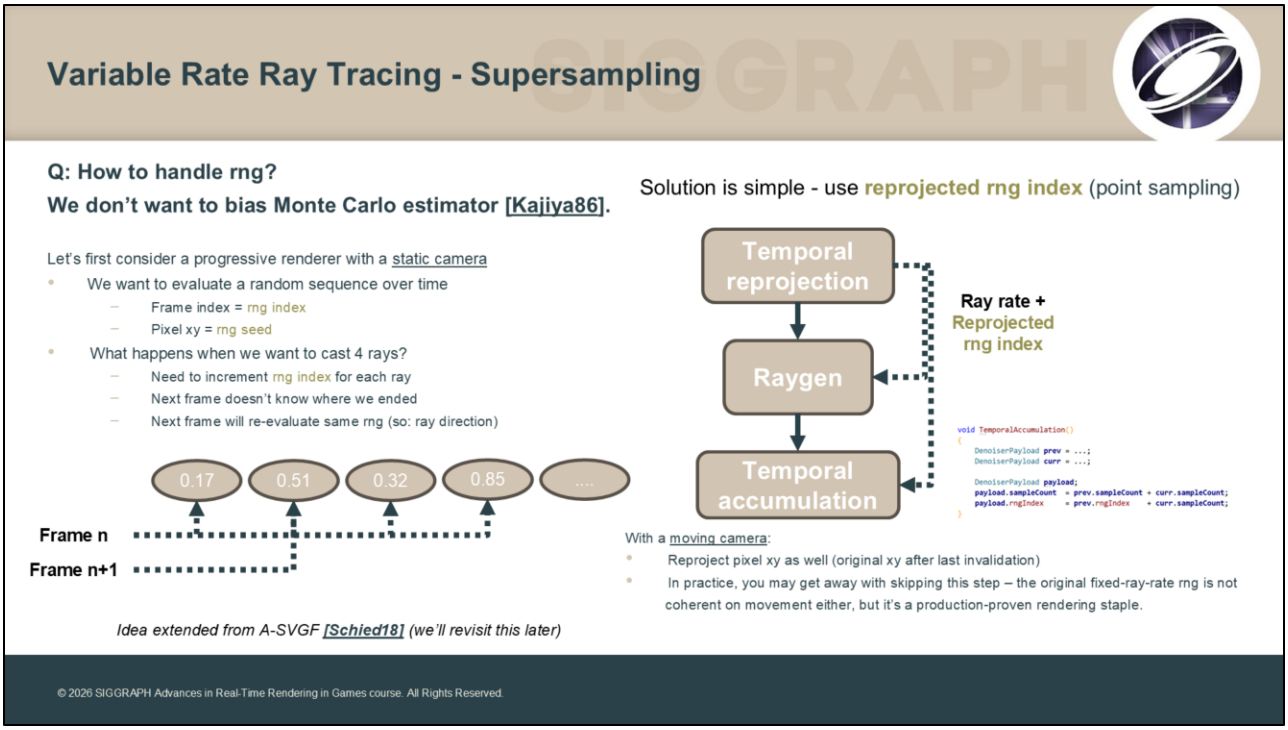
Variable Rate Ray Tracing



Unlike with VRS, we are not limited to fixed raster pixel grid, so **VRRT** is a fully 'software' solution.

Let's consider how to **supersample** a disoccluded area. We need to detect disocclusion before raygen, therefore temporal denoiser must be split into temporal reprojection and accumulation passes. This way, we can decide how many rays we want to cast during reprojection , and pass this information to raygen. In simplest implementation possible, we have a binary decision if we consider pixel disoccluded or not, then request enough rays to reach full temporal history.

Sidenote:
To fully leverage **VRRT**, I suggest using per-texel rejection weights in reprojection footprint (e.g. bilinear) to improve accumulated signal quality (you can keep longer history if its higher quality). Do not interpolate depth/normals, treat each texel separately with point sampling and cancel the contribution of disoccluded texels individually. This is also necessary if you do custom encoding (e.g. RGB9E5) and pack multiple signals manually into UINT texture – you need to interpolate those manually anyway.



There is one quirk we need handle here related to rng used for importance sampling (idea extended from *A-SVGF* / temporal gradients paper from Christoph Schied).

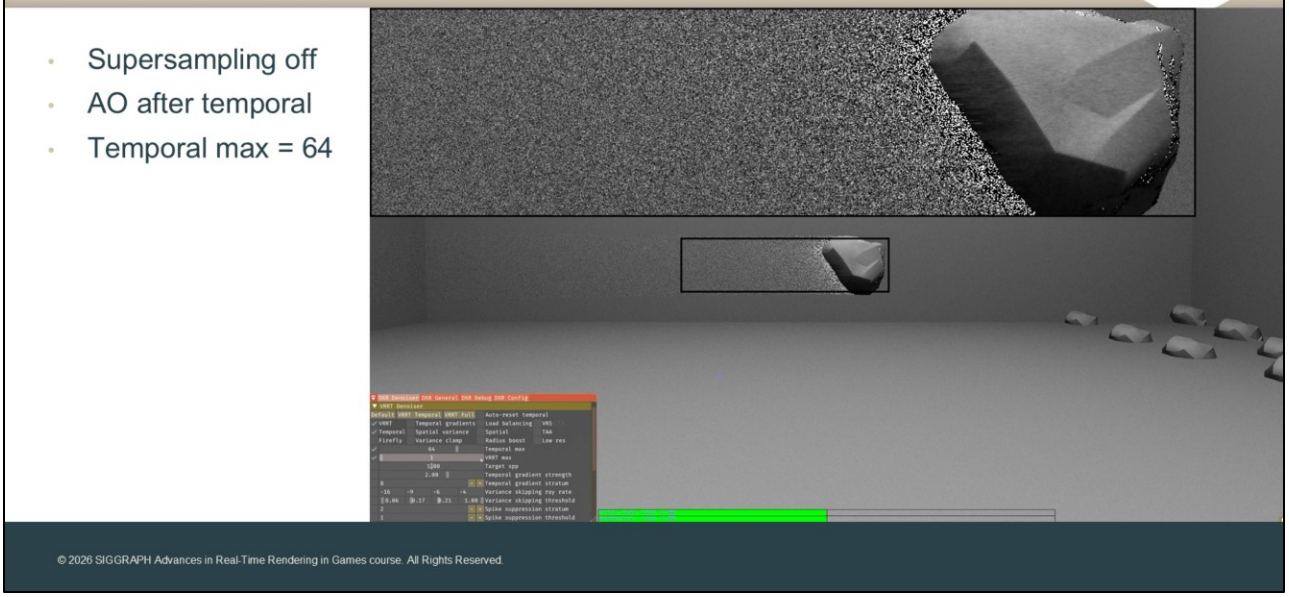
In a classic rng scheme that I suggested earlier where we use 'progressive rendering' with temporal pattern spread over time, we use frame index as rng index into that temporal pattern (different pattern for each pixel). This is no longer valid if we have variable number of rays per pixel each frame, as we will start re-using same rng output (so: casting the same ray twice), as you can see on animation on slides.

Solution is simple – we reproject rng index together with the rest of the payload (m1/m2/sampleCount). This can start as frame index, but it will be increment by number of rays for each pixel (see code above). We need to pass rng index from reprojection pass to raygen and accumulation pass (on top of requested ray rate). It does not make any sense to do bilinear interpolation on an 'index', so we use point sampling – which in practice work very well (phew!).

With moving camera, we technically should reproject starting pixel xy coordiantes as well, however this can be skipped in practice. The original 'progressive' scheme before **VRRT** was also not technically accurate with a moving camera but worked well and is widely used across industry.

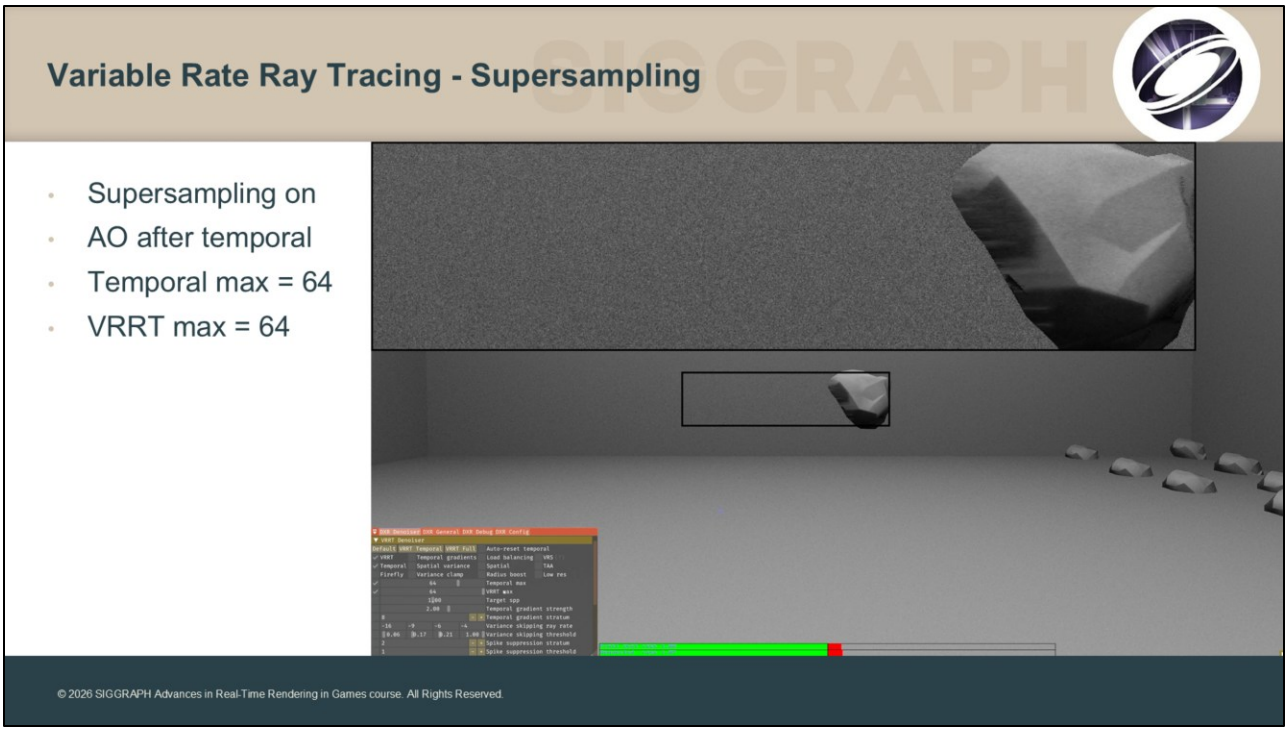
Sidenote: Ideally, we would extend this to support blue noise [Heitz19][Wolfe24] – it's on my TODO.

Variable Rate Ray Tracing - Supersampling



- Supersampling off
- AO after temporal
- Temporal max = 64

[video] First, let's look at temporal output of AO without **supersampling** (64 samples, I bumped AO radius high enough that it essentially became sky occlusion – to make errors more obvious).

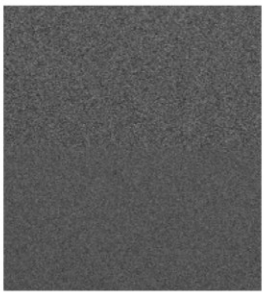


[video] Now, with temporal **supersampling**, you can see that disocclusion area is barely affected. The minor difference comes from the fact that exponential moving average technically accumulates 'infinite' number of samples (with weights approaching zero), even if we clamp the lerp factor.

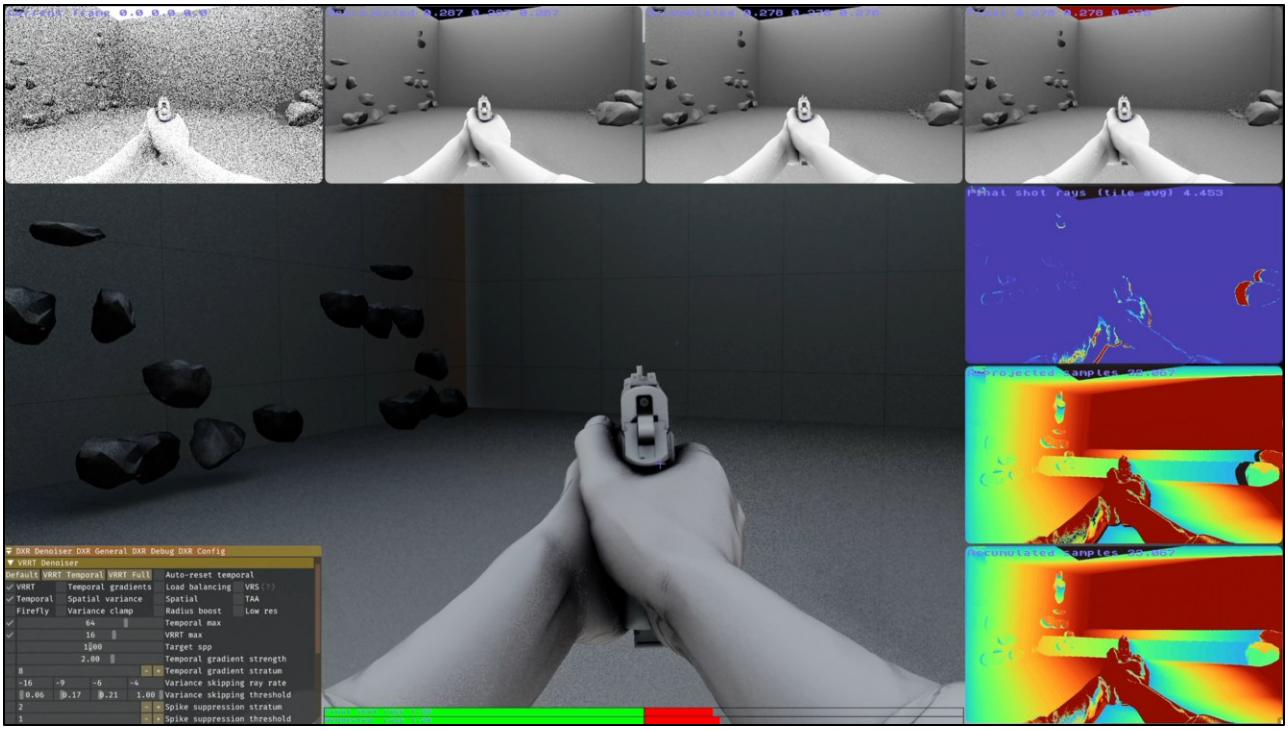
You can also see the progress bars on the bottom of the screen. Those show how many rays were requested on entire screen (1.0 = average of 1 per pixel), and how many were traced (without load balancer those are equal). When it gets red, it means that part is over budget (which happens here due to lack of load balancer).

Variable Rate Ray Tracing - Supersampling

- Supersampling on
- AO after temporal
- Temporal max = 64
- VRRT max = 64



[video] Here is a comparison of rng index reprojection (but not pixel xy for rng seed). Differences are not that huge, but notable. In general, you do not want this kind of errors compounding across denoising pipeline, and rng index is just a couple bits in the payload, so I recommend always doing this.



[video] And here we go between **VRRT** Off / max 16 ray rate request / max 64 ray rate request per pixel (temporal length = 64 always, vastly exaggerated AO).

Pay attention to overlays on the right. We can see which pixels request more rays when **VRRT** is on ("Final shot rays"). We can also see heatmap of history length (after reprojection and after accumulation). With full **VRRT**, the "accumulated samples" stays full.

It's worth noting that even if we do partial **supersampling** (max 16 rays, 64 temporal length), it still will be enough for good spatial filter and *SVGF*, even if temporal history is not full.

Variable Rate Ray Tracing - Undersampling

Basic **undersampling**

- Quarter rate - pick 1 random active ray per quad
  - Fixed cost, can be matched with DLSS-RR/FSR-RR
- VRS - use variable rate shading mask [Drobot20]
  - Load balancing managed by VRS
  - With full rate ray tracing – trace all active pixels in a quad
  - With quarter rate – pick 1 random active ray from active pixels

```
if ( resolvedRays < 1.0 )
{
    float rng = BlueNoiseSpatioTemporal( uint3( dtid, gFrameIndex ) ).x; // <0, 1>
    resolvedRays = resolvedRays > rng ? 1 : 0;
}
```

Advanced **undersampling**:

- Each ray picks probability to be traced <0, 1>
- Use noise for thresholding
- Variance-based ray skipping heuristic:
  - If temporal history is full, pick probability based on variance
  - We use 4 buckets so it's easier to tune, could use a curve
  - Can start skipping before full history if variance is really low (risky)
- Can use global ray rate < 1

```
if ( prevPayload.sampleCount >= g_denoiserConstants.temporal.targetSampleCount )
{
    float variance = prevPayload.Variance();
    for ( int i = 0; i < 4; ++i )
    {
        float threshold = g_denoiserConstants.vrrt.varianceSkippingThreshold[i];
        if ( variance < threshold )
        {
            requestedRayRate = g_denoiserConstants.vrrt.varianceSkippingRayRate[i];
            break;
        }
    }
}
```

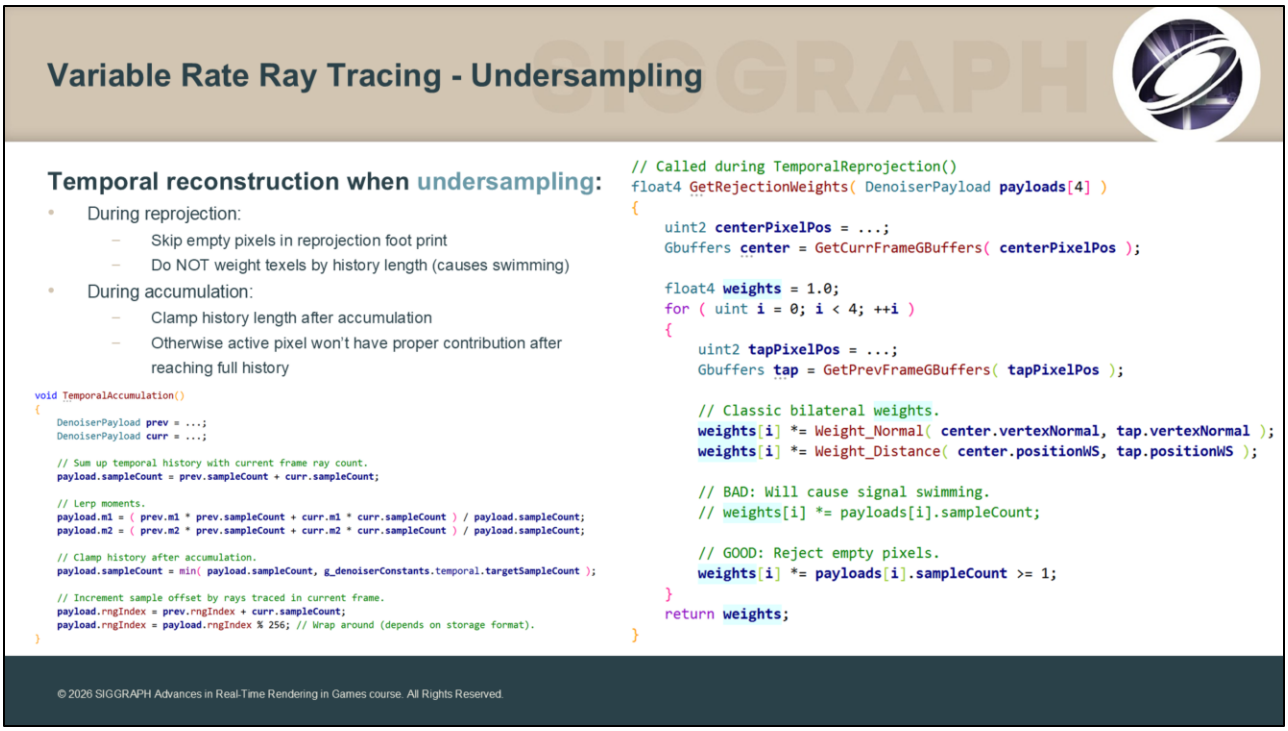

On the opposite end, we have **undersampling** (<1spp) with temporal reconstruction (upsampling).

In Call of Duty: Black Ops 7, we had following undersampling modes:

- Quarter rate mode – randomize one ray per quad of pixels, with each pixel having equal probability of casting ray
- VRS-based mode – cast rays for all valid pixels in VRS mask (skip ray for pixels that do not contribute to the image due to VRS)
- Quarter rate with VRS mode – instead of equal probabilities per quad, we use equal probabilities for all active pixels in VRS mask

On Call of Duty: Modern Warfare 4, we intend to ship more generalized approach by allowing each pixel to request 'fractional ray rate', meaning probability of it being traced – each pixel decides individuall, without being aware of neighbours (like with previous „quad" example).

We use a blue noise texture for thresholding and decide if we want to cast given ray or not. We employ a variance-based ray skipping heuristic, where we wait until temporal history buffer is full, then we drop ray rate request based on variance. This allows us to skip rays in 'safe areas'. We use 4 buckets to tune this, but this could be generalized to a curve. What's nice about this approach is that our last year's quarter rate mode can be trivially achieved with probabilities of 0.25. Of course, if VRS mask is inactive for this pixel, we just use probability of 0%.



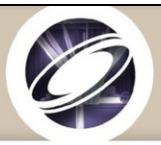
Before we go onto how load balancer combines **supersampling** with **undersampling**, let's make sure our temporal reconstruction is correct.

Fortunately, temporal reconstruction in denoising is much simpler then TAAU, because we don't have to resolve the edges.

Remember how I suggested to use per-texel rejection weights in reprojection footprint? This is partially because we also need to perform 'empty history' rejection test here as well. Just be sure not to weight the texels by actual history length – this is known to cause signal swimming and decimates image quality.

After you reach temporal history, you could have following pixels in a quad with history of: 64, 64, 64, 65 (when max temporal history length is 64). Make sure not to clamp that 65 too early, or the 'refreshed' sample will not have proper contribution to temporal filter, and it will cause signal to dissipate over time instead of converging. See provided code for how to perform correct accumulation.

Variable Rate Ray Tracing - Undersampling



Numerical precision issues when undersampling

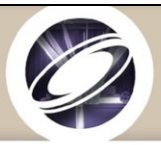
- For development (and your sanity) - keep full precision denoiser implementation
  - Moments, motion vectors, depth buffer (for world position reconstruction)
  - Helps with debugging – i.e. is it a precision issue or an actual bug?
- In final version – detect quantized moments getting stuck (fp16/rgb9e5/r8unorm)
  - If fp32 value after accumulation is different from history value, but they are the same after quantization – push it by 1 bit manually
  - Happens more often with longer history length

```
uint Fp16QuantizationCorrection( float accumValue, float prevValue )
{
    uint accumFp16 = f32tof16( accumValue );
    uint prevFp16  = f32tof16( prevValue );

    bool stuck = ( accumValue != prevValue ) && ( accumFp16 == prevFp16 );
    if ( stuck )
        accumFp16 = accumValue > prevValue ? min( accumFp16 + 1, 0x7BFF ) : max( accumFp16, 1u ) - 1;
    return accumFp16;
}
```


I would recommend always keeping a full precision implementation (for signals, depth - for world position reconstruction, motion vectors etc.) to be able to separate numerical precision issues from implementation bugs.

Then, when using quantized values (e.g. fp16) for 'final' product, make sure manually detect potential stuck signals. It's possible that value before and after temporal accumulation is different, but it is the same when quantized as lower precision – in that case, push it manually by 1 bit into direction the uncompressed values changed. As an example, if you use R8_UNORM encoding for shadow signal, and there is a completely unshadowed (1.0) pixel that accidently failed to reject its history (0.0) after occluder moved away , it will start interpolating slowly to 1.0 but with 64 frames of temporal history epsilon will get so small it will get stuck around 0.8 and never reach 1.0.

Variable Rate Ray Tracing - Undersampling



Spatial reconstruction when undersampling

- If temporal reconstruction failed (e.g. on first frame when global ray rate < 1)
- Leverage spatial blur
  - Turn off variance weights for empty pixels
  - Keep normal/distance weights
  - Optional: track weighted/unweighted result and pick the latter if sumWeights < 1 (flood fill)

If you use < 1.0 global ray rate, you need to handle first few frames on camera cut, when some pixels are fully black. This is easiest achieved with leveraging spatial blur to fill in the gaps by removing variance weights. You can also remove depth/normal weights if you couldn't find any relevant pixels as it's usually better than black pixels. In our implementation, spatial output is NOT beeing fed to temporal (like in 'Recurrent blurs' approach linked earlier), so it won't get stuck in history.

[video] Temporal reconstruction with static camera

- Full rate vs quarter rate (raw)
- Full rate temporal convergence
- Quarter rate temporal convergence

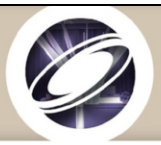
Ideally, this will only happen on camera cut, because with partial screen invalidation we will always supersample quickly.

[video] Temporal reconstruction with dynamic camera

- Full rate in motion
- Quarter rate in motion – signal gets a bit more dissipated in motion due to lower ray rate, but eventually converges back to full sharpness after stopping

Note: blurriness without temporal is due to disabled spatial variance estimate (variance weights are off until temporal history converges, causing over blurring)

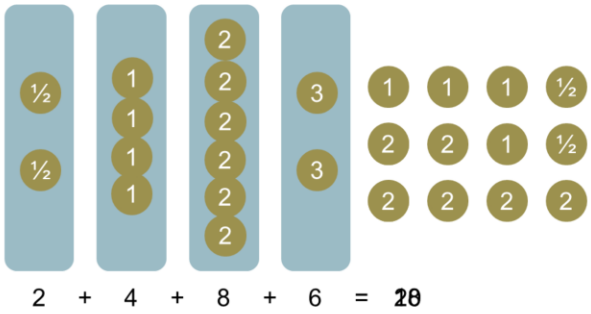
Variable Rate Ray Tracing – Load balancing



Load balancer

- Water filling algorithm
- Specify target average per-pixel ray rate, e.g. 1.5
- Setup
  - Each pixel requests ray rate
  - Quantize ray rate requests into buckets
  - Sum up number of pixels in each bucket
  - Sum up all ray rates on screen
  - Best done at the end of temporal reprojection shader
- If over budget:
  - Find highest spp bucket with non 0 pixel count
  - Drop those pixels to lower bucket
  - Repeat until in budget
    - This is your „bucket threshold"
    - Clamp all pixels to "bucket threshold"
- If under budget – do reverse (start from lowest buckets)

```
#define VRRT_MIN_RAY_RATE   64 // Read as 1/x.
#define VRRT_MAX_RAY_RATE   16 // Read as x.

float VrrtBucketToRayRate( int bucketIndex )
{
    bucketIndex -= VRRT_MIN_RAY_RATE;
    return bucketIndex >= 0 ? float(bucketIndex) + 1.0 : 1.0 / float(-bucketIndex);
}
```



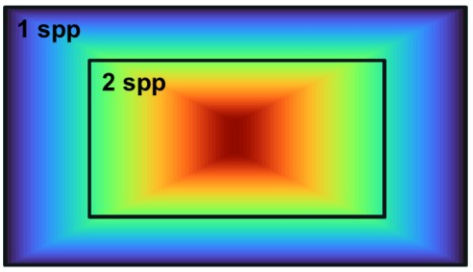
[animated slide] Our current **load balancing** algorithm errs on the side of simplicity.

We want to achieve a fixed budget per frame (same number of rays each frame) - this is expressed as an 'average rays per pixel' slider, e.g. 1.5. Each pixel decides its own requested ray rate in isolation (so via approaches described in supersampling and undersampling sections), and those need to be quantized into buckets so we can properly count (globally) number of pixels in each bucket, as well as total screen sum of all ray rates. In fact, you can specify ray rate requests in already quantized index instead and skip quantization step.

Counters for each bucket can be accumulated at the end of temporal reprojection shader, which already has all important data determining ray rates fetched. You can do it at virtually no extra cost by leveraging InterlockedAdd on groupshared memory, then reswizzling values within compute group so each lane performs InterlockedAdd on separate bucket index (i.e. you shouldn't use InterlockedAdd on global memory using the same address on multiple threads within compute group)

To apply load balancing, e.g. when total screen sum of rays is over budget, we just find the highest bucket with any active pixel, and we drop those pixels 1 bucket lower, until we are in budget. For under budget case, we do opposite. **This means load balancer doesn't work on per-pixel ray requests, only determines thresholding on bucket level.**

Variable Rate Ray Tracing – Load balancing



Load balancer - refinement

- Imagine fully converged image:
  - All pixels request 1 ray => we bump them to 2...
  - … when target was 1.5
- Refine with foveated prioritization via rings
  - Find „ring threshold" inside your „bucket threshold"
  - Need counters for each bucket in each ring
- Now inner half of screen casts 2 rays, outer casts 1 ray
  - Could be instead refined with fractional >1 buckets
  - E.g. 1.5 ray rate bucket, then use stochastic thresholding like for <1 ray rate buckets

```
#define COMPUTE_GROUP_SIZE     16
#define VRRT_RING_DIM          ( uint2( COMPUTE_GROUP_SIZE * 2, COMPUTE_GROUP_SIZE ) )

uint VrrtGetRingCount()
{
    uint2 nTiles = DIVIDE_AND_ROUND_UP( gRenderTargetSize, VRRT_RING_DIM );
    return min( nTiles.x, nTiles.y ) / 2;
}

uint VrrtGetRingId( uint2 dtid )
{
    uint2 tileId = dtid / VRRT_RING_DIM;
    uint2 nTiles = DIVIDE_AND_ROUND_UP( gRenderTargetSize, VRRT_RING_DIM );
    uint2 ringId = select( tileId.xy < nTiles / 2, tileId.xy, nTiles - tileId.xy - 1 );
    return min(ringId.x, ringId.y);
}
```


This approach has an obvious fail case due to its coarseness, e.g. we can bump **all** pixels in one bucket, but not a fraction of those pixels, so we can overestimate. To fix this, we refine the search with 'rings' (pictured, not very round, but can be computed trivially and synced to compute group size for easier math). So, on top of 'bucket threshold' we need to find 'ring threshold' inside this bucket, so we can do a partial bucket bump. This requires us to add extra counters for each ring (each ring x each bucket), but this also is cheap to do at the end of reprojection shader.

This allows us to do 'foveated' prioritization, when inner part of the screen will have 1 more ray than outer part in given bucket – we usually want better quality near the center, especially in an FPS.

Of course, we could expand our approach to do fractional buckets that are > 1 (and spread those rays uniformly across the screen), and leverage thresholding via noise like with undersampling, but this increases bucket counts significantly, so it is not trivial.

Variable Rate Ray Tracing – Load balancing

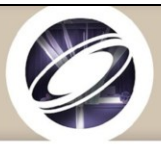
Load balancer pass

- Naive 1 thread dispatch that finds:
  - Bucket threshold
  - Ring threshold
- Can be optimized to 1 group with LDS pre-load
  - < 0.01ms even with 80 buckets and 60 rings (4k)
  - Ideally just stick it somewhere on async compute

```
void LoadBalancer( uint dtid : SV_DispatchThreadID )
{
    // Only the first lane computes and outputs anything, the rest is kept around for LDS preloading.
    if ( dtid < VRRT_BUCKETS_COUNT )
        gsVrrtBuckets[dtid] = VrrtLoadRayBucketCount( bufInput0, dtid );
    GroupMemoryBarrierWithGroupSync();

    float screenSumRays             = ...;
    float screenSumPixels           = gRenderTargetSize.x * gRenderTargetSize.y;
    float screenBudget              = g_denoiserConstants.vrrt.targetSpp * screenSumPixels;
    float raysUnderBudget           = screenBudget - screenSumRays;
    float raysOverBudget            = -raysUnderBudget;
    uint ringCount                  = VrrtGetRingCount();

    int bucketThreshold = 0;
    int ringThreshold = 0;
    int bucketRingThreshold = 0;
    if ( raysOverBudget > 0 )
    {
        // Drop buckets until we reach budget, starting with highest ray-rate buckets.
        float droppedRays = 0;
        float affectedPixels = 0;
        for ( bucketThreshold = VRRT_BUCKETS_COUNT - 1; bucketThreshold >= 1; --bucketThreshold )
        {
            float pixels = gsVrrtBuckets[bucketThreshold];
            float currRate = VrrtBucketToRayRate( bucketThreshold );
            float nextRate = VrrtBucketToRayRate( bucketThreshold - 1 );
            float drop = ( affectedPixels + pixels ) * ( currRate - nextRate );
            if ( droppedRays + drop >= raysOverBudget )
                break;
            droppedRays += drop;
            affectedPixels += pixels;
        }

        // We know that the current bucket (==`bucketThreshold`) goes under budget, refine it with rings (starting from the edges).
        for ( ringThreshold = 0; ringThreshold < ringCount - 1; ++ringThreshold )
        {
            // Count affected pixels in buckets above bucketThreshold in this ring (they have already been bumped).
            if ( bucketThreshold <= dtid && dtid < VRRT_BUCKETS_COUNT )
                gsVrrtBuckets[dtid] = VrrtLoadRingBucketCount( bufInput0, ringThreshold, dtid );
            GroupMemoryBarrierWithGroupSync();
            float affectedRingPixels = 0;
            for ( bucketRingThreshold = VRRT_BUCKETS_COUNT - 1; bucketRingThreshold > bucketThreshold; --bucketRingThreshold )
                affectedRingPixels += gsVrrtBuckets[bucketRingThreshold];

            // Now drop over the last bucket in this ring.
            float pixels = gsVrrtBuckets[bucketRingThreshold];
            float deltaRate = VrrtBucketToRayRate( bucketRingThreshold ) - VrrtBucketToRayRate( bucketRingThreshold - 1);
            float drop = ( affectedRingPixels + pixels ) * deltaRate;
            if ( droppedRays + drop >= raysOverBudget )
                break;
            droppedRays += drop;
            affectedRingPixels += pixels;
        }
    }

    if ( raysUnderBudget )
        // Reverse of above.

    bufOutput0.Store2( 0, uint2( bucketThreshold, ringThreshold ) );
}
```


The load balancer itself is a trivial program that runs on single shader lane (or single group, if you want to prefetch buckets using LDS). The inner nested loop can have a lot of iterations, but since it's a single thread we can get away with it. This is something I would like to revisit (e.g. when doing fractional > 1 buckets). Code provided for your convenience.

Variable Rate Ray Tracing – Load balancing



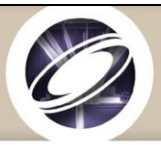
VRRT Apply / Produce ray args

- Cheap full screen dispatch
- Reads ray rate requests texture:
  - Applies thresholding
  - Does stochastic test on fractional ray rates
- Produces ray args:
  - Indirect dispatch args
  - Thread descriptors with ray count and rng offset
  - [Drobot20] [Hammer25]

```
uint VrrtApply( uint2 dtid, int bucket, int bucketThreshold, int ringThreshold, bool overBudget )
{
    int isInnerPixel = VrrtGetRingId( dtid ) > ringThreshold ? 1 : 0;

    if ( overBudget && bucket >= bucketThreshold )
        bucket = bucketThreshold - 1 + isInnerPixel;
    if ( !overBudget && bucket <= bucketThreshold )
        bucket = bucketThreshold + isInnerPixel;

    // Transform bucket index to actual ray rate.
    float resolvedRays = VrrtBucketToRayRate( bucket );
    if ( resolvedRays < 1.0 )
    {
        float rng = BlueNoiseSpatioTemporal( uint3( dtid, gFrameIndex ) ).x; // <0, 1>
        resolvedRays = resolvedRays > rng ? 1 : 0;
    }

    return (uint)resolvedRays;
}
```


Most of the cost of ray balancing comes from pass that needs to 'trim' ray requests to buckets/ring thresholds we found. Fortunately, it only needs to read one small texture, and the code for thresholding is very simple, so cost is miniscule. This pass also must compute indirect dispatch args for raygen shader and a buffer (list) with thread descriptors (ray count, rng offset, pixel xy coordinates). This can also be made almost free – utilize WavePrefixSum + InterlockedAdd on groupshared memory for your buffer counter and for computing output indices for each thread.

Variable Rate Ray Tracing – Load balancing



Tips

- Either sort threads so higher ray counts go first (to avoid long waves stalling entire pass)
  - or splat pixels with multiple rays to multiple threads
    - helps with vgprs due to lack of loop
    - will require extra work to add results for each pixel – might not be worth it
- Do not use actual IndirectRaygen on NVIDIA – up to 2x slower
  - use direct raygen and early out trailing threads (should be within margin of error with load balancer)
- When using split raygen and hit point shading
  - Instead of full screen texture/buffer storing shading inputs, make it an atlas/buffer
  - Use the previously set global target ray rate (e.g. 1.5) to determine atlas/buffer size

Here are some implementation tips you can use. We currently use split TraceRay/Shading, and we employ bucketing based on hit geometry/material features, so each ray (even with supersampling) has its own thread, and we have a small pass that adds them up together before temporal accumulation.

[video]

- **VRRT** off
- Disocclusion artifacts

[video]

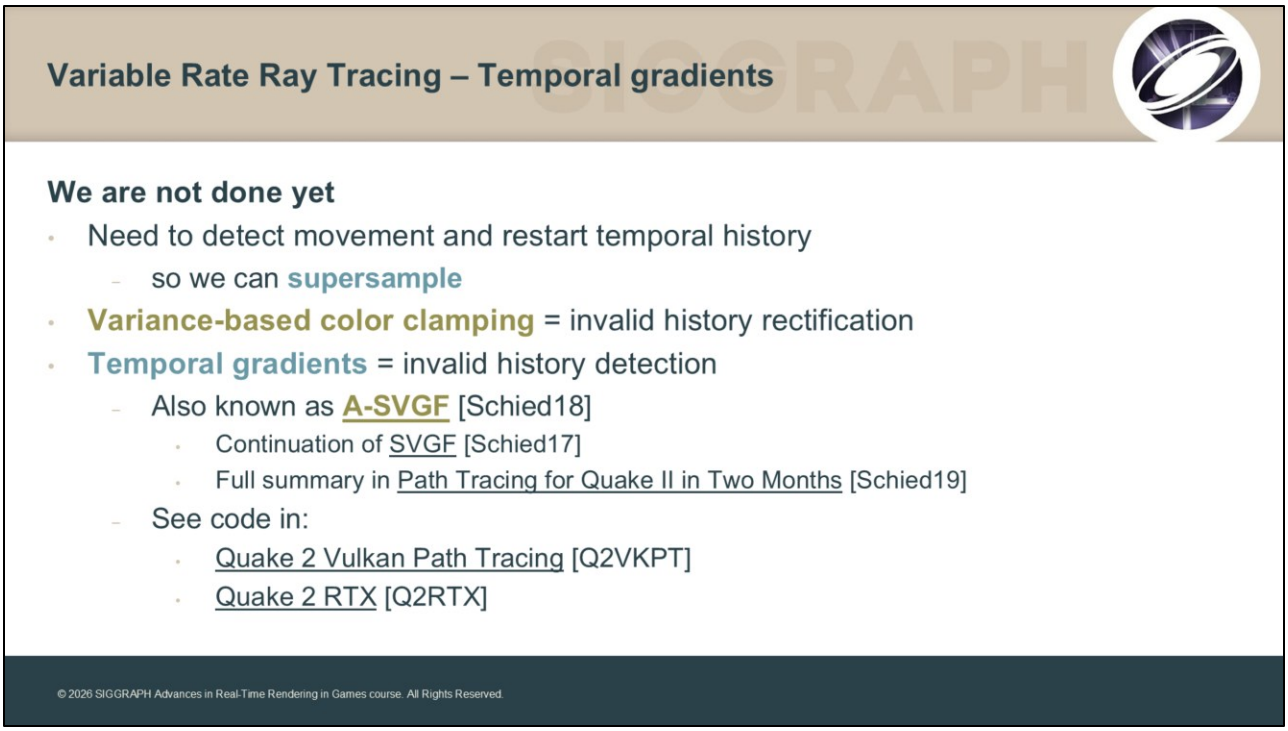
- **Supersampling** on
- Disocclusion artifacts fixed
- But overbudget
- See „final shot rays” (bottom bar)

[video]

- **Supersampling** on
- **Load balancing** on
- In budget
- But not enough rays to fully fix disocclusion
- Still better than no supersampling
- Probably enough samples for *SVGF*
- But we can do better

[video]

- **Supersampling** on
- **Under sampling** on
- **Load balancing** on
- Disocclusion artifacts fixed
- We moved rays from stable areas to unstable
- While maintaining ray budget

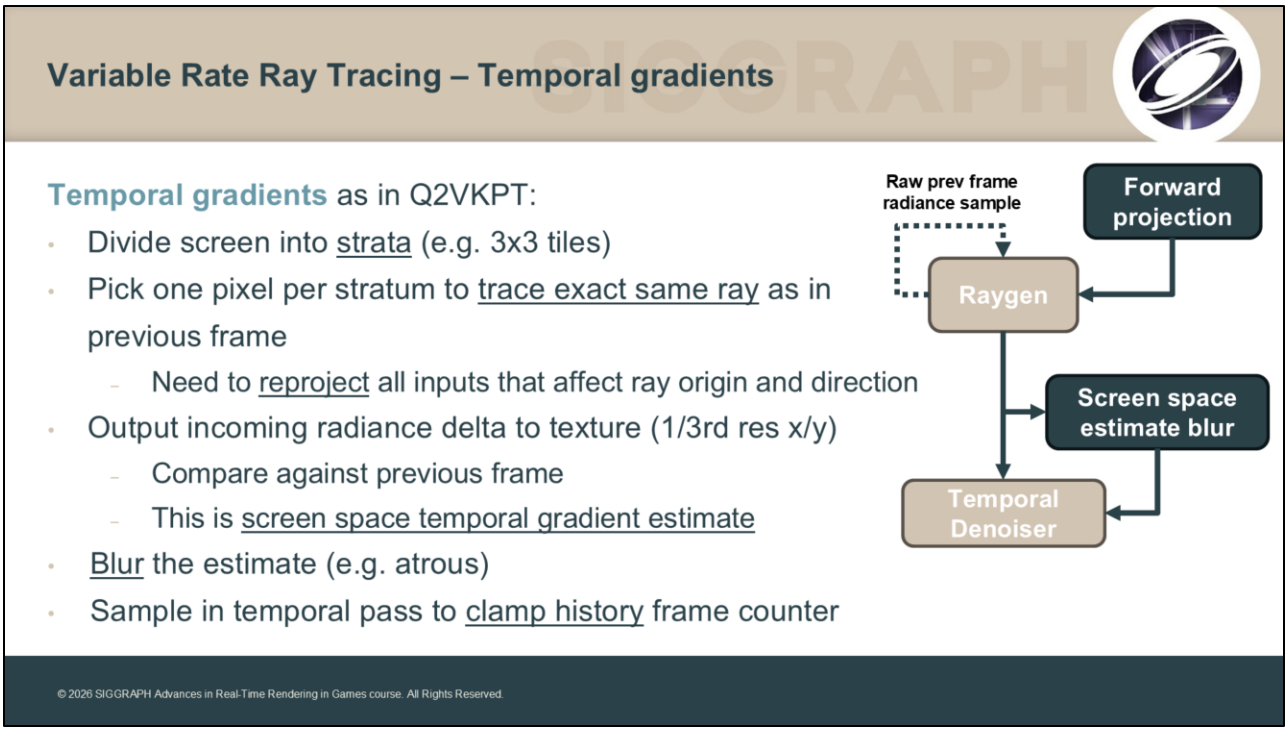


Now that we have reliable method of moving rays where they matter most, we 'just' need to handle dynamic event detection and history rejection, so we can apply the same principles as for disocclusion. As explained earlier, variance-based color clamping is not a 'detection' algorithm, but more of a history rectification algorithm. We want something better.

Thankfully, Christoph Schied, author of *SVGF*, did a follow-up paper that introduced concept of **temporal gradients**, which is an excellent variance-based clamping alternative.

I strongly recommend getting familiar with available materials on topic of temporal gradients. The main paper is more abstract, but there is both a practical presentation on the core concept linked on the slide, as well as two different open-source implementations available. It's a bit more engine-side work than variance-based clamping, but it's a very powerful tool. I think we're really sleeping on this as an industry.

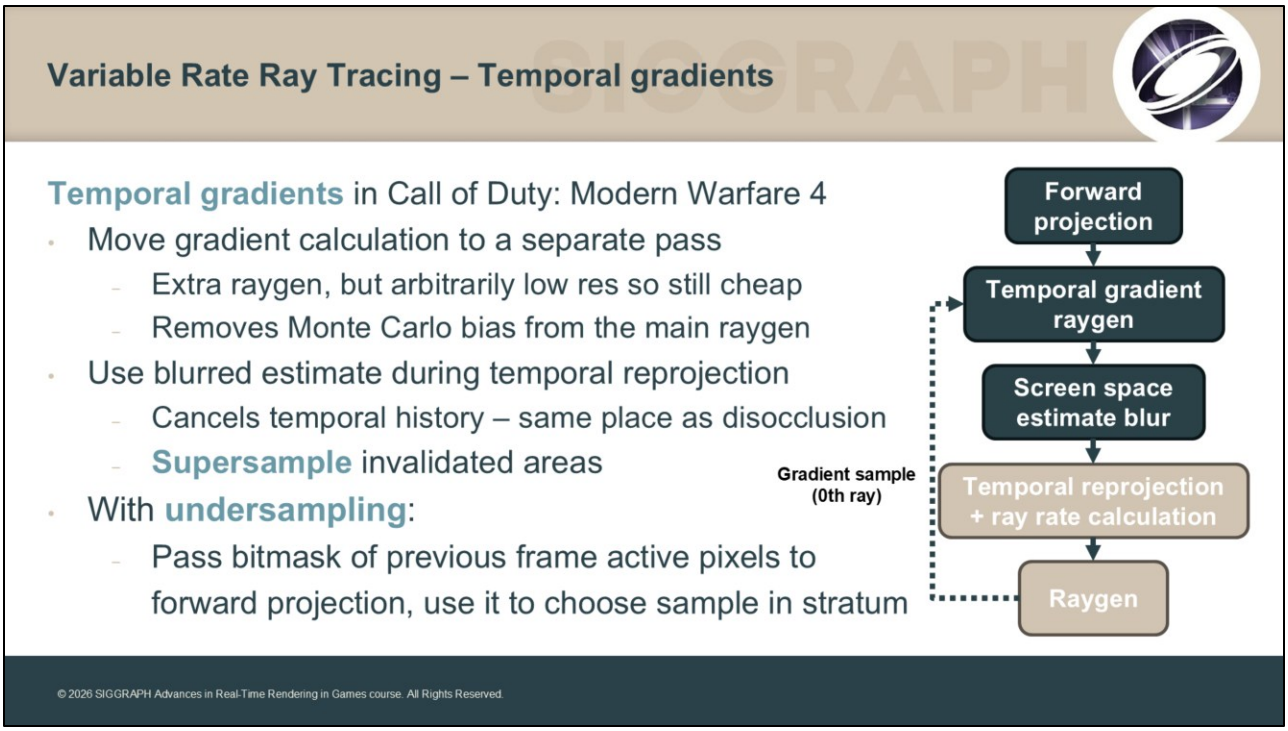
Variable Rate Ray Tracing – Temporal gradients



The main concept behind temporal gradients, as implemented in *Q2VKPT* is as follows: divide your raygen into small tiles (3x3 in *Q2VKPT*), pick random pixel per tile and cast exact same ray as in previous frame – then compare incoming radiance, and output it to a small texture. This is your temporal gradient, aka rate of change in radiance. It's a coarse screeen space estimate, so it needs to be blurred to be useful, but it allows you to detect which parts of your screen (more or less) have dynamic lighting events going on. You can use to clamp your temporal history buffer length (this is an arbitrary tunable).

In order to cast the same ray, we need to reproject all data from previous frame that affect ray origin and direction. Refer to code on later slides for more details.

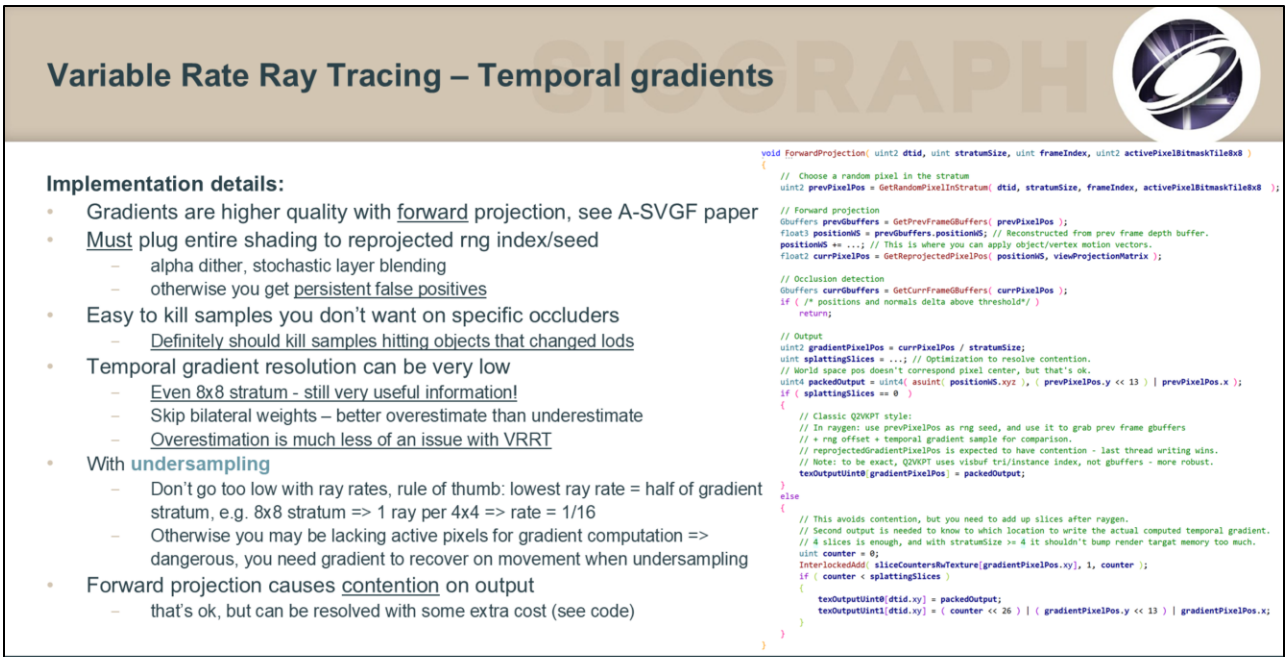
Variable Rate Ray Tracing – Temporal gradients



In *Call of Duty,* we implemented temporal gradients as a prepass – a separate, smaller raygen, that computes temporal gradients before main raygen. This allows us to have temporal gradient information available before we make decision where to cast main rays, so it can be fed into temporal reprojection pass – next to disocclusion data – to determine the rate rates.

There is an extra cost to such pass due to extra rays, but temporal gradients are powerful even in lower res (e.g. 8x8 downsample), and the downsides of lower res are automatically mitigated by **VRRT** supersampling. As long as in average case temporal gradient estimate covers only a small portion of the screen, we should be able to supersample those areas while fitting into ray budget.

Variable Rate Ray Tracing – Temporal gradients

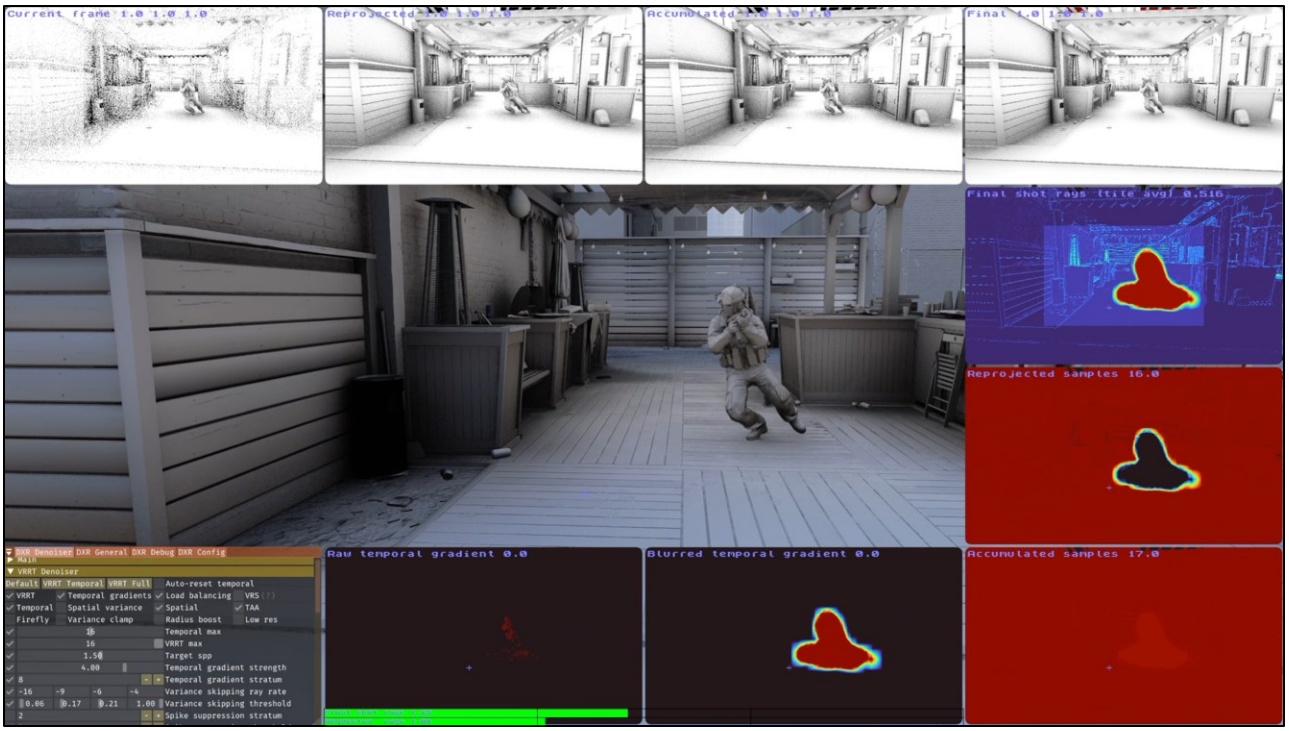


Implementation details:

- Gradients are higher quality with forward projection, see A-SVGF paper
- Must plug entire shading to reprojected rng index/seed
  - alpha dither, stochastic layer blending
  - otherwise you get persistent false positives
- Easy to kill samples you don't want on specific occluders
  - Definitely should kill samples hitting objects that changed lods
- Temporal gradient resolution can be very low
  - Even 8x8 stratum - still very useful information!
  - Skip bilateral weights – better overestimate than underestimate
  - Overestimation is much less of an issue with VRRT
- With **undersampling**
  - Don't go too low with ray rates, rule of thumb: lowest ray rate = half of gradient stratum, e.g. 8x8 stratum => 1 ray per 4x4 => rate = 1/16
  - Otherwise you may be lacking active pixels for gradient computation => dangerous, you need gradient to recover on movement when undersampling
- Forward projection causes contention on output
  - that's ok, but can be resolved with some extra cost (see code)

```
void ForwardProjection( uint2 dtid, uint stratumSize, uint frameIndex, uint2 activePixelBitmaskTile8x8 )
{
    //  Choose a random pixel in the stratum
    uint2 prevPixelPos = GetRandomPixelInStratum( dtid, stratumSize, frameIndex, activePixelBitmaskTile8x8  );

    // Forward projection
    Gbuffers prevGbuffers = GetPrevFrameGBuffers( prevPixelPos );
    float3 positionWS = prevGbuffers.positionWS; // Reconstructed from prev frame depth buffer.
    positionWS += ...; // This is where you can apply object/vertex motion vectors.
    float2 currPixelPos = GetReprojectedPixelPos( positionWS, viewProjectionMatrix );

    // Occlusion detection
    Gbuffers currGbuffers = GetCurrFrameGBuffers( currPixelPos );
    if ( /* positions and normals delta above threshold*/ )
        return;

    // Output
    uint2 gradientPixelPos = currPixelPos / stratumSize;
    uint splattingSlices = ...; // Optimization to resolve contention.
    // World space pos doesn't correspond pixel center, but that's ok.
    uint4 packedOutput = uint4( asuint( positionWS.xyz ), ( prevPixelPos.y << 13 ) | prevPixelPos.x );
    if ( splattingSlices == 0  )
    {
        // Classic Q2VKPT style:
        // In raygen: use prevPixelPos as rng seed, and use it to grab prev frame gbuffers
        // + rng offset + temporal gradient sample for comparison.
        // reprojectedGradientPixelPos is expected to have contention - last thread writing wins.
        // Note: to be exact, Q2VKPT uses visbuf tri/instance index, not gbuffers - more robust.
        texOutputUint0[gradientPixelPos] = packedOutput;
    }
    else
    {
        // This avoids contention, but you need to add up slices after raygen.
        // Second output is needed to know to which location to write the actual computed temporal gradient.
        // 4 slices is enough, and with stratumSize >= 4 it shouldn't bump render targat memory too much.
        uint counter = 0;
        InterlockedAdd( sliceCountersRwTexture[gradientPixelPos.xy], 1, counter );
        if ( counter < splattingSlices )
        {
            texOutputUint0[dtid.xy] = packedOutput;
            texOutputUint1[dtid.xy] = ( counter << 26 ) | ( gradientPixelPos.y << 13 ) | gradientPixelPos.x;
        }
    }
}
```


The devil is in the details, so I wrote down some practical tips for temporal gradients implementation and code sample – should be a bit easier to read for initial recon than **Q2VKPT**/RTX github repos where you need to jump between different files.
The most important thing is that when casting rays for temporal gradients, entire shader must be 'plugged' into the same rng that was used last frame – otherwise you will get persistent false positives which will needlessly be **supersampled** by **VRRT**.

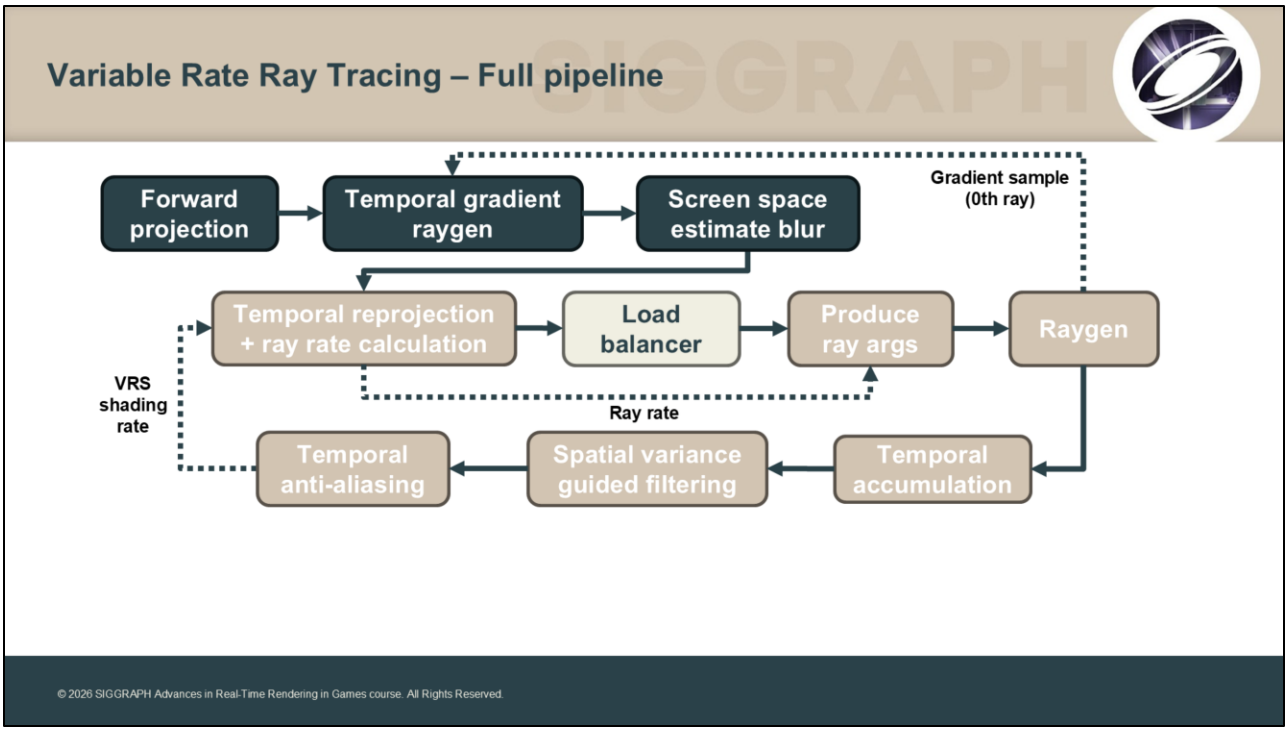


[video] This is what temporal gradients look like in action

- **Temporal gradients** on (8x8 stratum)
- Temporal + spatial + TAA
- **Supersampling** + **undersampling** + **load balancing**
- Observe temporal gradient overlays at the bottom
- And shot rays / temporal overlays on the right
- No variance clamping, yet no ghosting
- Stable AO everywhere
- Even on movement
- Not enough rays when NPC gets close
- But still over 1spp

[video] Here I go over all debug overlays in a more dynamic scenario

- (Debug overlays off) Indirect lighting with RTAO
- AO after temporal – static areas stable, dynamic (around cars) invalidated by temporal gradients
- AO after spatial – stable even on dynamic areas (overbudget, but still > 1spp giving stable blur)
- Raw temporal gradient – coarse
- Blurred temporal gradient – properly catches vehicle movement
- Reprojected samples – heatmap of temporal history length after invalidation by temporal gradients
- Accumulated samples – same heatmap, but after raygen, including supersampled areas
- AO after spatial – again, this time for enemy NPCs – see that there is no temporal lag and moving AO is stable
- (Debug overlays off) - indirect lighting, you can now observer all overlays at the same time – worth going over couple times to see different interactions!



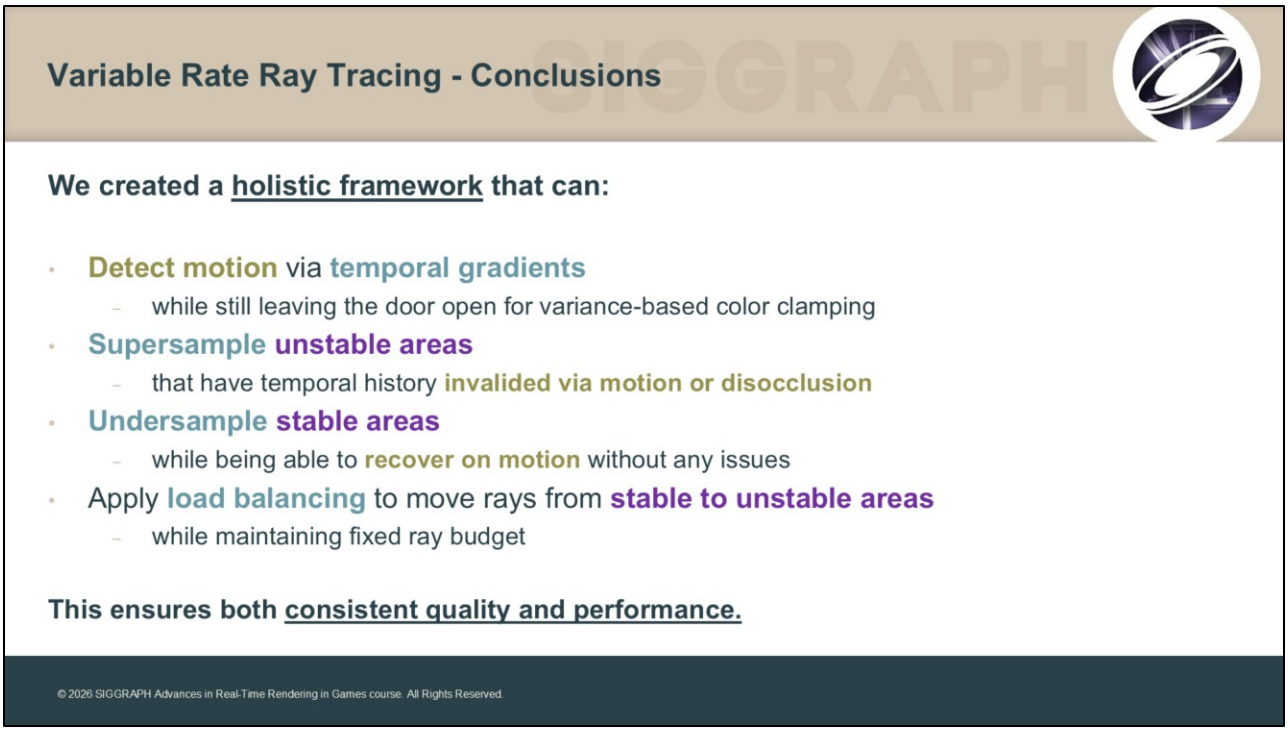
[animated slide] The full pipeline is as follows:

- (Middle row) We start with raygen
  - Temporal reprojection happens before – uses multiple sources to determine ray rates
  - Load balancer is a small dispatch – computes ray rate thresholds
  - Next pass clamps or bumps ray rates after load balancing – and produces indirect args for raygen dispatch with thread descriptors
- (Top row) We can optionally also compute temporal gradients (in arbitrary lower resolution)
  - We compare output of temporal gradient pass against 0th ray from previous frame to compute final screen space estimate and we blur it
- (Bottom row) Then everything goes as in classic denoising pipeline
  - Temporal accumulation is good place to do firefly clamping (and variance-based clamp, if you still want to use it)
  - TAA determines active pixels for variable rate shading – this information is fed to next frame ray rate calculation

Temporal reprojection pass is where most of the magic happens, as it's perfect place to feed all sources of data that affect ray rate calculation and make the final decision (disocclusion, temporal gradients, vrs mask)

Conclusions & future research

SIGGRAPH 2026
LOS ANGELES
19–23 JULY 2026



Now that we have all building blocks of **VRRT**, let's sum up how they fit together (notice color coding on the slide).

Having a reliable way of detecting motion really brings this framework together, as now we don't have any major sources of uncertainty when it comes to temporal history validity.

We can **supersample** unstable areas easily and **undersample** those that are stable while (crucialy!) being able to recover without any frame delay when something starts moving – this is because temporal gradients are computed by prepass, which wouldn't be possible to do with variance clamping because signal would be too sparse with **undersampling**.

Load balancer ties this together – allows us to cast fixed number of rays without exceeding the budget and automatically bumps average ray rate (so improves image quality) if nothing is happening on the screen.

This is how we achieve consistent performance overhead and consistent quality – we significantly raised the bar of the worst-case scenario for denoising.

Variable Rate Ray Tracing - Conclusions

Minimal overhead that pays for itself:

- Small fixed overhead:
  - Load balancer
  - Temporal split (reproject & accumulate)
- Less or no need for:
  - Spatial variance estimation
  - Variance-based clamping
- Tunable:
  - Temporal gradient resolution
- Smarter rays > more rays
  - Overhead can be recovered by lowering ray budget 5-10%
  - Can use fractional global ray rate – it's now a slider = scalability
  - Scales better with expensive raygen (complex BVH or hit shading)

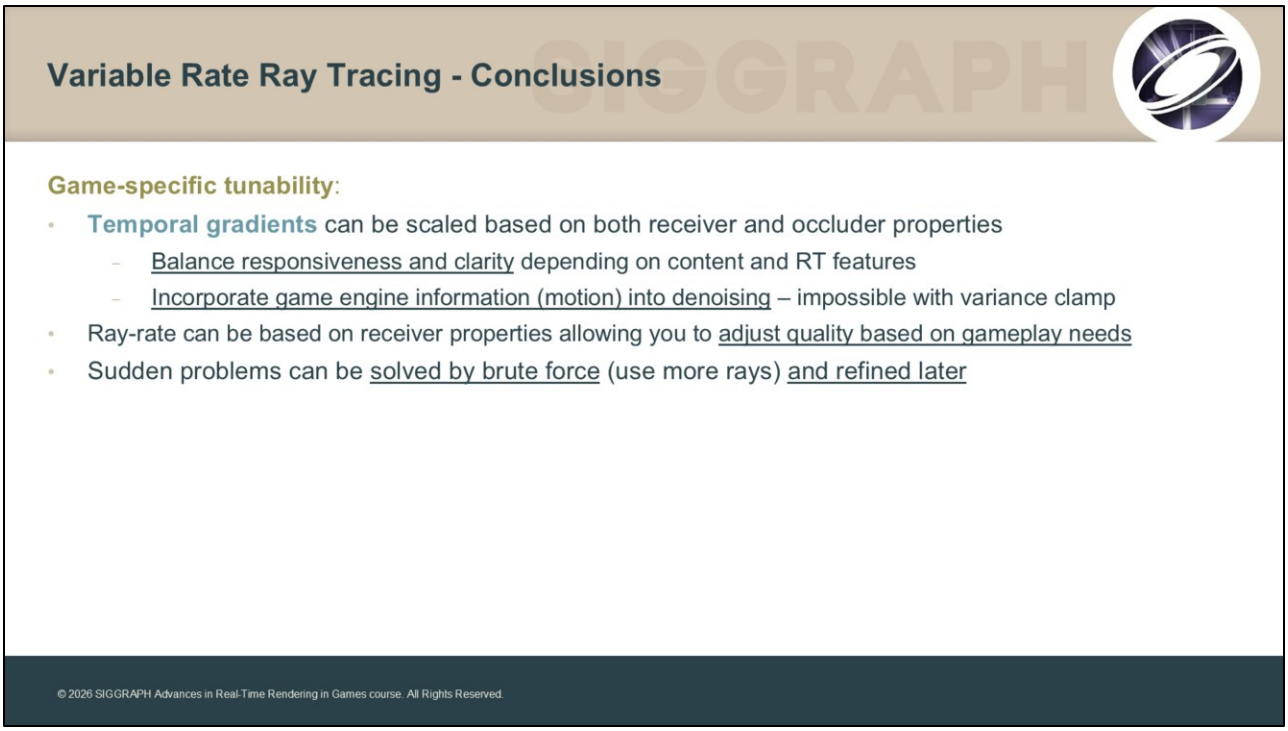
VRAM cost: +144MB for denoiser; +0MB for VRRT, +12MB for 8x8 temporal gradient

1080p @ RTX 5090	VRRT Off [ms]	VRRT On [ms]
Reproject	-	0.13
Accumulate	0.15-0.4	0.09-0.15
Spatial variance	0.11	-
Spatial blur	4x 0.13	4x 0.13
AO Raygen	0.28	0.31
Gradient raygen 8x8	-	0.05
Load balancer	-	0.03
Sum	**0.95-1.20**	**1.13-1.19**

Variance clamp is off with VRRT
Raygen overhead with VRRT due to indirect dispatch
Accumulate spread on variance/firefly-clamp radius

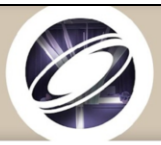
VRRT improves upon classic screen space denoising while maintaining minimal overhead – there is some fixed cost, however this easily pays for itself, and you can easily recover this cost by lowering ray rate a little bit (since it's now effectively a float slider). Compared to the overhead of non-VRRT denoising the extra cost of VRRT is negligible.

In my experience it's always worth it, because without **VRRT** we waste a huge chunk of our ray budget doing nothing.



What I personally like the most about how temporal gradients integrate into **VRRT** is the more human friendly tunables (compared to variance-based clamping). We can now leverage in-game information about properties of receivers and occluders (materials, geometry, motion), to tune our heuristics between lighting responsiveness and clarity. For average gameplay scenario you may want more general heuristics, but you still have the possibility to tune **VRRT** for specific scenario e.g. when polishing a cutscene. Previously, any jira that I got on denoising quality could be only met with a shrugging "I can try improve it but it will break something else". Now I have actual new tool in my toolbox.

Variable Rate Ray Tracing - Future research

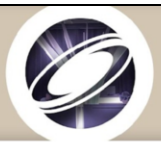


And this just a warm up! So many ideas to try out – some possible because of VRRT, some finally feasible!

- How far can we push undersampling (and how much can we stress temporal gradients)?
- VRRT in non-native res - can we rely on temporal reconstruction?
- How to use blue noise with rng index reprojection without losing its properties [Heitz19]?
- Ray rate requests based on post-fx like depth of field or motion blur
- VRRT with spherical harmonics/gaussians/Lumen-like probes
- Hook the shading/ray-rate slider to the GPU frame timer – instead of dynamic resolution
- Load balancing spatial denoiser performance (number of samples/iterations and/or radius) based on variance/VRS rates
- Stochastic direct lighting leveraging VRRT to scale localy based on per-pixel light counts
- Hybrid bake / RT systems
  - Partial bakes - marking difficult areas for RT
  - Prebaked ray guiding determining ray rates
- Load balancing
  - Across raygens (GI/Reflections/Shadows)
  - Depending on scene material composition (gloss cutoff)
  - Between opaque and transparent reflections
- Applying lessons learnt to VRS
  - Load balancer and proper temporal reconstruction/upsampling for VRS
  - VRS with 8/16 samples + super low ray rates = full visual parity on Switch 2 with RT? 120Hz RT next-gen mode?

It took me many years to finally introduce full **VRRT** into *Call of Duty* engine, but this is just a start for us, not the end. There is a ton of potential research that we want to do over next few years. I believe this might be our main battleground when it comes to squeezing every bit of quality and performance from our future, RT-oriented, version of the engine. Those are just a couple of ideas that I wrote down from top of my head! I'm not going to go over this now, because it would take another couple hours.

Variable Rate Ray Tracing - Conclusions



You can start implementation by extending your denoiser step by step with:

- **Supersampling**
  - tile-based requests first – works good enough!
  - split reprojection from accumulation and use disocclusion for ray rate requests
- **Load balancing**
  - easier to do without fractional ray rates
- **Temporal gradients**
  - See Q2VKPT/Q2RTX code for reference
- **Undersampling**
  - First only for full history - to gauge how much it can boost supersampling
  - Then temporal reconstruction

RTAO/Shadows are easiest (no shading). Temporal gradient and undersampling can be swapped around.

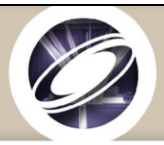
I hope I convinced you of how useful **VRRT** is, and that I've fully realized my initial pitch in the beginning of this talk. It's an extension of classic denoising pipeline, so you can start step by step by extending your denoiser step by step. I wrote down my tips in what order to do it best.

Thank you!

- Enormus gratitude to ray tracing team folks at Activision for all the hard work over the years:
  - **Félix Bridault**
  - **Patrick Ahrens**
  - **Matthew Dibble**
  - **Kamil Grodecki**
- Big thanks to **Michał Drobot**, for supporting this research
- To **Christoph Schied**, for pioneering the research on which VRRT was built
- To colleagues from **Nvidia, AMD,** and **Intel** – for discussions and ideas
- To everybody involved in **Advances in Real-Time Rendering** course – endless source of inspiration and motivation
- ... and to my two assistants

You can always find me here: molejnik@infinityward.com, olej3d @ twitter

Bibliography



Denoising

- [Dammertz10] Edge-Avoiding À-Trous Wavelet Transform for Fast Global Illumination Filtering
- [Mattausch10] High-Quality Screen-Space Ambient Occlusion using Temporal Coherence
- [Bavoil12] Stable SSAO in Battlefield 3 with Selective Temporal Filtering
- [Stachowiak15] Stochastic Screen-Space Reflections
- [Schied17] Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination
- [Schied18] Gradient Estimation for Real-Time Adaptive Temporal Filtering
- [Schied19] Real-Time Path Tracing and Denoising in 'Quake 2'
- [Zhdan19] Exploring Raytraced Future in Metro Exodus
- [Zhdan20] Fast denoising with self-stabilizing recurrent blurs
- [Stachowiak22] Kajiya Global Illumination Overview
- [Kostas22] Raytraced global illumination denoising
- [Sousa25] Fast as Hell: idTech8 Global Illumination
- [Pikulski25] Optimizing spatiotemporal variance-guided filtering for modern GPU architectures

Denoisers with public source code

- NVIDIA Real-Time Denoiser
- Kajiya (MIT)
- Quake 2 Vulkan Path Tracing (GPL 2.0)
- Quake 2 RTX (GPL 2.0)

Sampling

- [Kajiya86] The Rendering Equation
- [Heitz19] A Low-Discrepancy Sampler that Distributes Monte Carlo Errors as a Blue Noise in Screen Space
- [Wolfe24] Filter-Adapted Spatio-Temporal Sampling for Real-Time Rendering

Temporal anti-aliasing & variance-based clamping

- [Lottes11] TSSAA (Temporal Super-Sampling AA) - no longer available online
- [Karis14] High-Quality Temporal Supersampling
- [Drobot14] Hybrid Reconstruction Anti-Aliasing
- [Xu16] Temporal Antialiasing in Uncharted 4
- [Salvi16] An excursion in temporal supersampling
- [Pedersen16] Temporal Reprojection Anti-Aliasing in INSIDE
- [Jimenez16] Filmic SMAA: Sharp Morphological and Temporal Antialiasing
- [Jimenez17] Dynamic Temporal Antialiasing in Call of Duty: Infinite Warfare
- [Tardiff] Temporal Antialiasing Starter Pack

Raytracing:

- [Olejnik20] Raytraced Shadows in Call of Duty: Modern Warfare
- [Myers21] Shadows of Cold War: A Scalable Approach to Shadowing

Variable Rate Shading:

- [Drobot20] Software-Based Variable Rate Shading in Call of Duty: Modern Warfare
- [Hable24] Variable Rate Shading with Visibility Buffer Rendering
- [Hammer25] Variable-Rate Compute Shaders in DOOM: The Dark Ages


ACTIVISION
NOW HIRING
BUILD THE FUTURE OF
CALL OF DUTY & WARZONE
WORLD-CLASS TEAMS
ICONIC FRANCHISE
GLOBAL IMPACT
CAREERS.ACTIVISION.COM | APPLY TODAY!
OPEN ROLES – CALL OF DUTY & WARZONE
SOFTWARE ENGINEER, GAMEPLAY – CALL OF DUTY
Multiple Locations
SENIOR GAME DESIGNER – CALL OF DUTY
Multiple Locations
SENIOR TECHNICAL DESIGNER – CALL OF DUTY
Multiple Locations
AUDIO DESIGNER – CALL OF DUTY
Multiple Locations
SENIOR PRODUCER – CALL OF DUTY
Multiple Locations
LIVE OPERATIONS DESIGNER – WARZONE
Multiple Locations
TOOLS PROGRAMMER – WARZONE
Multiple Locations
ANTICHEAT ENGINEER – CALL OF DUTY & WARZONE
Multiple Locations