



March 21-25, 2022  
San Francisco, CA

# An Overview of the Diablo II: Resurrected Renderer

**Kevin Todisco**  
Lead Software Engineer, Graphics  
Blizzard Entertainment

#GDC22



Hello! Thank you for joining me today. It's great to be back in person at GDC and I hope you all have had a fantastic week so far. I'm Kevin Todisco, Lead Graphics Engineer at Blizzard and I have the pleasure of presenting the work of a very talented team that made Diablo II: Resurrected's renderer a reality. Before we get going, I'd like to remind everyone to fill out those surveys at the end of the talk so that I can make my talks better and also get invited back in the future! Ok, administrative stuff out the way...

I'm here today to pull back the curtain on the bespoke renderer that we created for the remaster. I'll talk about how we approached the project technically, the development philosophies we used to govern our technology choices, and give examples of technologies we incorporated and how they fit with these philosophies. I'll also talk about some of the lessons we took away from this experience of building a renderer from the ground up -- an experience which is quite rare these days -- which are still applicable even when using pre-existing engines. To get us going, it's best to take a look at the game we're talking about developing a renderer for, so...

NOW EXPERIENCE THE TERROR



# What year is it?

- The camera angle is fixed.
- We can make reasonable assumptions about what features we need.
- We know what the gameplay patterns are.
- But we have a content team with nothing to work in.

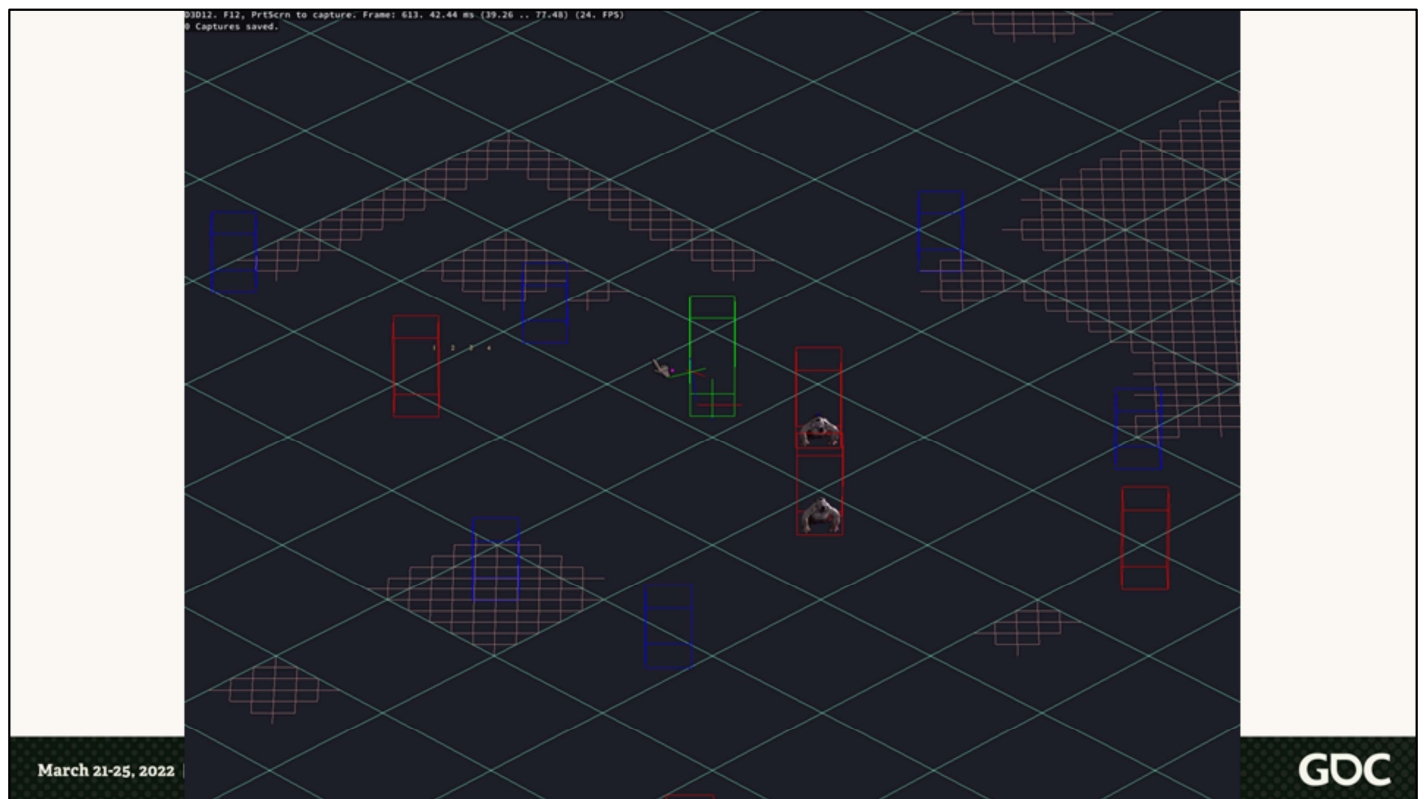
So that clip gives us a pretty good sense of what we set out to make – a 1:1 visual reproduction of the original game, in 3D with high fidelity, realistic visuals.

Early on, the decision was made to use the original Diablo 2 engine for the remaster in order to preserve the game feel and experience. As a team, we didn't believe we could recreate all the nuances of the game in a different engine. This, in turn, precluded us from using an existing engine, whether proprietary or commercial. The plan formed to build a bespoke renderer on top of the 20 year old engine to load and manage the 3D assets. We considered that we had some advantages that would help when it came to building out this new part of the engine. A key one was that the camera angle throughout the game is fixed – it is not user controlled, there are no long vistas, only a limited number of things can be in view at once. Another we know up front, roughly, are what features we need to support. We also know common gameplay patterns and just how intense gameplay can be. The disadvantage, of course, is that you have a content team with a deadline and no engine to start working in yet.

# Philosophies

- Prioritize rapid development

To set ourselves up for success, we had to establish some guiding principles for the development of this part of the engine. The largest challenge at the beginning was making sure that the team could be making forward progress. While most projects would start with some technology to make assets, we had nothing.



I mean literally, nothing.

# Rapid Development

- Live-asset updating
  - Right-click -> Build asset
  - Hot reload
- For programmers, too
  - Forward+ renderer
  - Automatic shader reload

The very first thing we needed to do was get an asset pipeline up and running. Our timeline was pretty tight, so we arrived at our first philosophy: every aspect of development must be rapid. Live-updating of assets in the engine is essential. In the early days in the absence of a suite of tools we had a context menu utility that the team could use where they could right-click on a file and build it, then drag-and-drop it into a prototype viewer application for the engine. As tools matured, this turned into a hot-reloading pipeline. But we also can't ignore ourselves as programmers. We chose to go with a forward-plus renderer, not just because of its rising popularity but also because it enables us to rapidly develop new materials -- you can write a custom shader for the material in the forward pass and you're done. On top of this, of course, we had things like automatic shader reload.

# Philosophies

- Prioritize rapid development
- No baked lighting

Another philosophy was that no lighting in the game would be baked. This... probably seems crazy at first when you consider that our art direction was pushing for realistic rendering, but this is probably the strongest example of a hard choice we made to ensure success. We took this stance for several reasons.



# No Baked Lighting

- Time
- Procedural levels
  - Dynamism is also where the title will shine.
- Mod-ability

First, baking takes time. Art would not be able to work fast if they were relying on extensive light and shadow bakes in order to give final sign off on environments. It also costs time just to implement baking, even if you're using a pre-existing solution -- you still have to integrate it into your pipeline and fit it to your data layout.

Second, Diablo II is a heavily procedural game. Levels are generated using building blocks called presets, which are placed adjacent to each other and *may* contain fixed lighting sources. We can't predict what geometry will be adjacent to a light in an adjoining preset, and also some light sources are dynamically placed. From an aesthetic standpoint, Diablo II is a very effects-heavy game. Spells fly in every direction and many of those spells illuminate the world in the original sprite game, revealing more information to the player as they cut through the crippling darkness of dungeons. We stand the highest likelihood of being a visually impressive game if we can lean into the dynamism of this lighting.

Third, though least impactful, was consideration for the modability of the game. If we expect players to create their own models to put into the game, we don't want those models to take on a different appearance because they weren't run through some complex baking process.

# Philosophies

- Prioritize rapid development
- No baked lighting
- Always consider scalability

Another principle was scalability.

# Scalability

- List of platforms grew over time.
  - Addition of the Switch puts extra emphasis on scalability.
- Every individual technique must be scalable.
- Or turned off completely
  - For example, hair does not require TAA to render correctly.

This originally was in place because we needed to support a wide range of PC hardware, but later it became even more important as we added Nintendo Switch as a SKU. Each technique chosen for the renderer needed to be able to be turned off or be scalable in some way. As an example, our hair and fur isn't dependent on TAA, so TAA can be turned off to save cost. Our hair, while using a complex lighting model, was also developed with precomputation in mind, so it's roughly the same cost to render as any other material.

# Philosophies

- Prioritize rapid development
- No baked lighting
- Always consider scalability
- Off-the-shelf technologies

But perhaps the confluence of all of these things and most important is the technology we chose to implement. We had to be very mindful that while we were presented with this great opportunity to start with a clean slate, we can't try to reinvent the wheel. We have to stick to what works, and that means relying on established techniques and picking them off-the-shelf, so to speak.

# Off-the-shelf Technology

- Pick techniques that are like plug-and-play.
  - Lighting model
  - HDR color grading
  - Order-independent transparency
  - Skin rendering
  - SSAO
  - TAA
- Carefully consider where to innovate
  - Don't reinvent the wheel.
  - Align innovations with team expertise

Some examples I'll talk about today are our lighting model, HDR color grading, order independent transparency, and specular aliasing. And there are more examples like skin rendering, SSAO, TAA. Each of these is unique in *why* it was included or *how* it was included, but each shares a foundation in how it was selected - it was already proven somewhere else, and it was adaptable.

We don't have a lot of room to innovate, and where we do we have to pick our battles carefully. Not only do innovations have to be favorable to time and very deliberate, they also have to align with our art and technology goals, and our team expertise.

That's not to say that we weren't innovative in some spaces. Our HDR color grading built on the work that it was inspired by, our hair rendering was a leap forward, our solutions for skin rendering and specular aliasing needed to be tweaked to fit in the engine, and our order independent transparency was heavily optimized over time. I'm getting a bit ahead of myself...

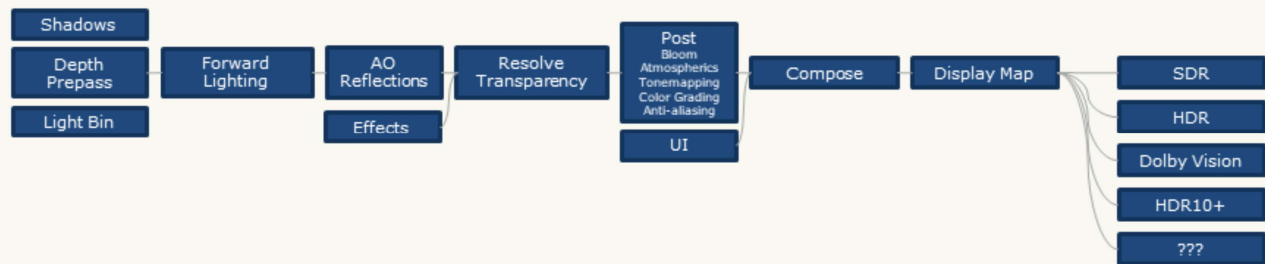


# Agenda

- Intro
- Overview of the renderer
- Deep dive into select techniques
- A frame on the GPU
- A frame on the CPU
- Streaming performance
- LODs
- Takeaways

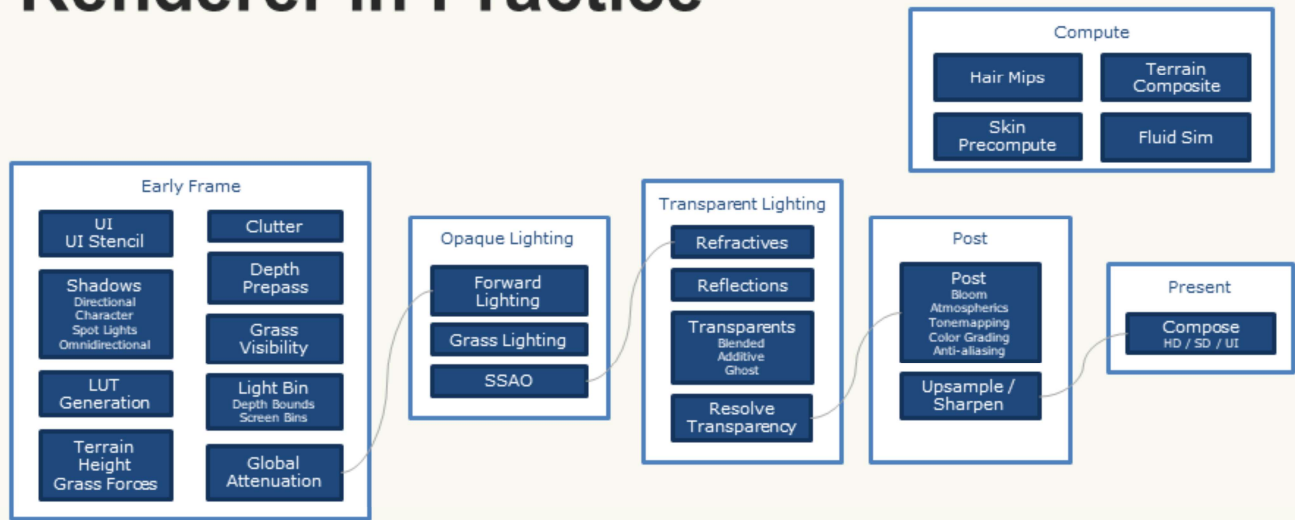
Before we dive in, here's a quick look at everything we'll cover today.

# Renderer Design



Building a renderer from the ground up gives us a unique opportunity to design from the start with a modern pipeline in mind. The advent of high dynamic range output necessitated a shift that challenged conventional methods of color grading and tonemapping, so these were two areas that we could implement with a modern approach from the outset, instead of having to overhaul a piece of established technology. This, here, of course is an idealized layout – the kind you can plan for up front.

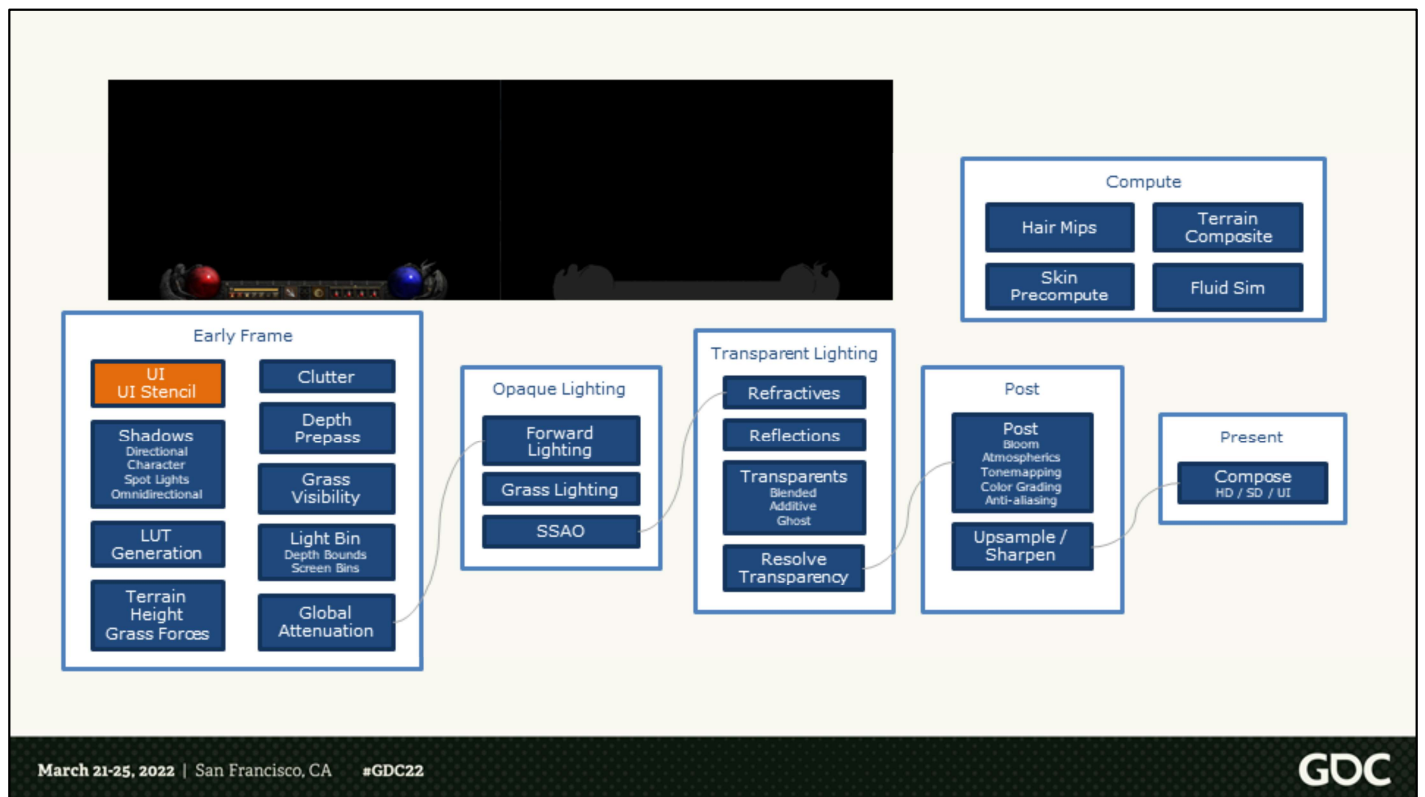
# Renderer in Practice



In practice of course it changes quite a bit.

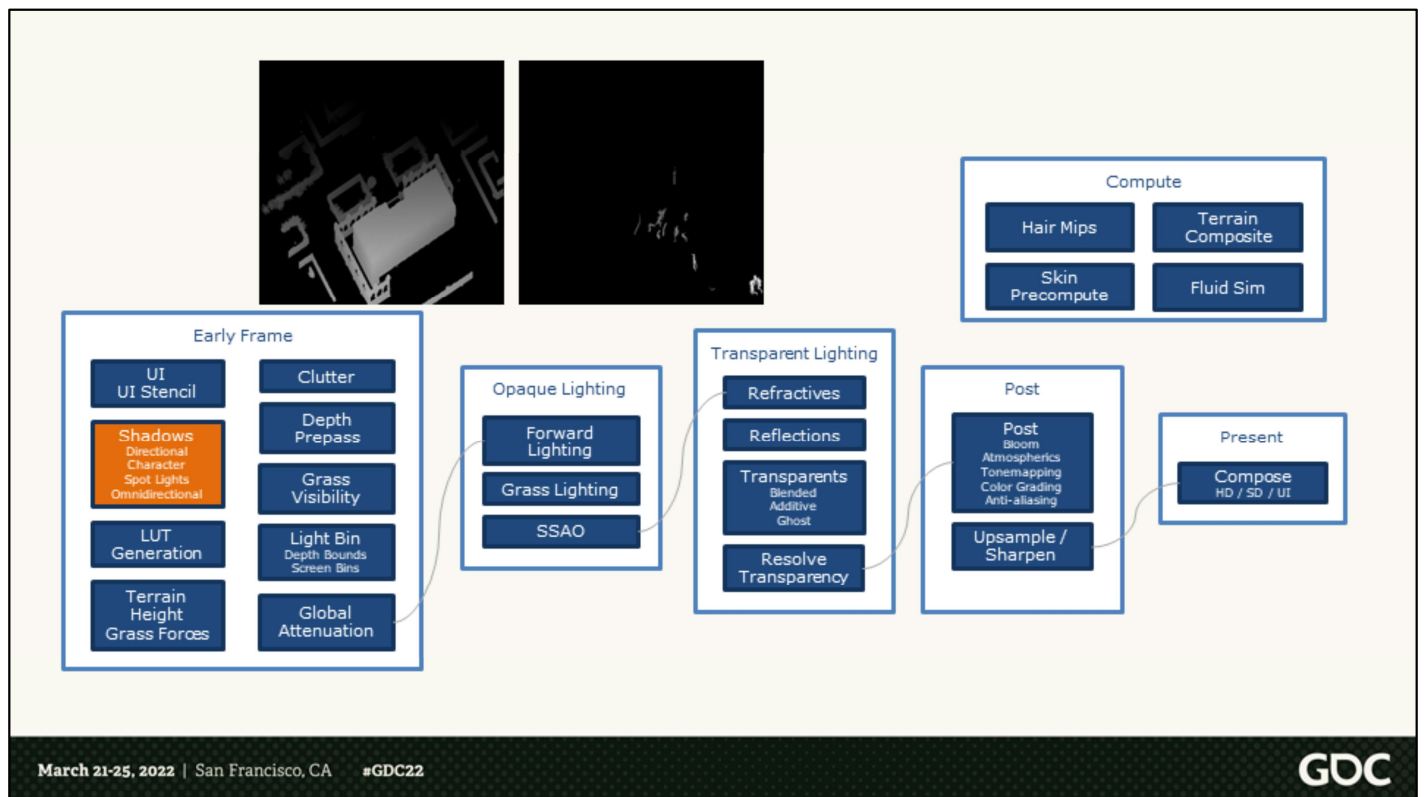


Let's take a look at a full frame first.

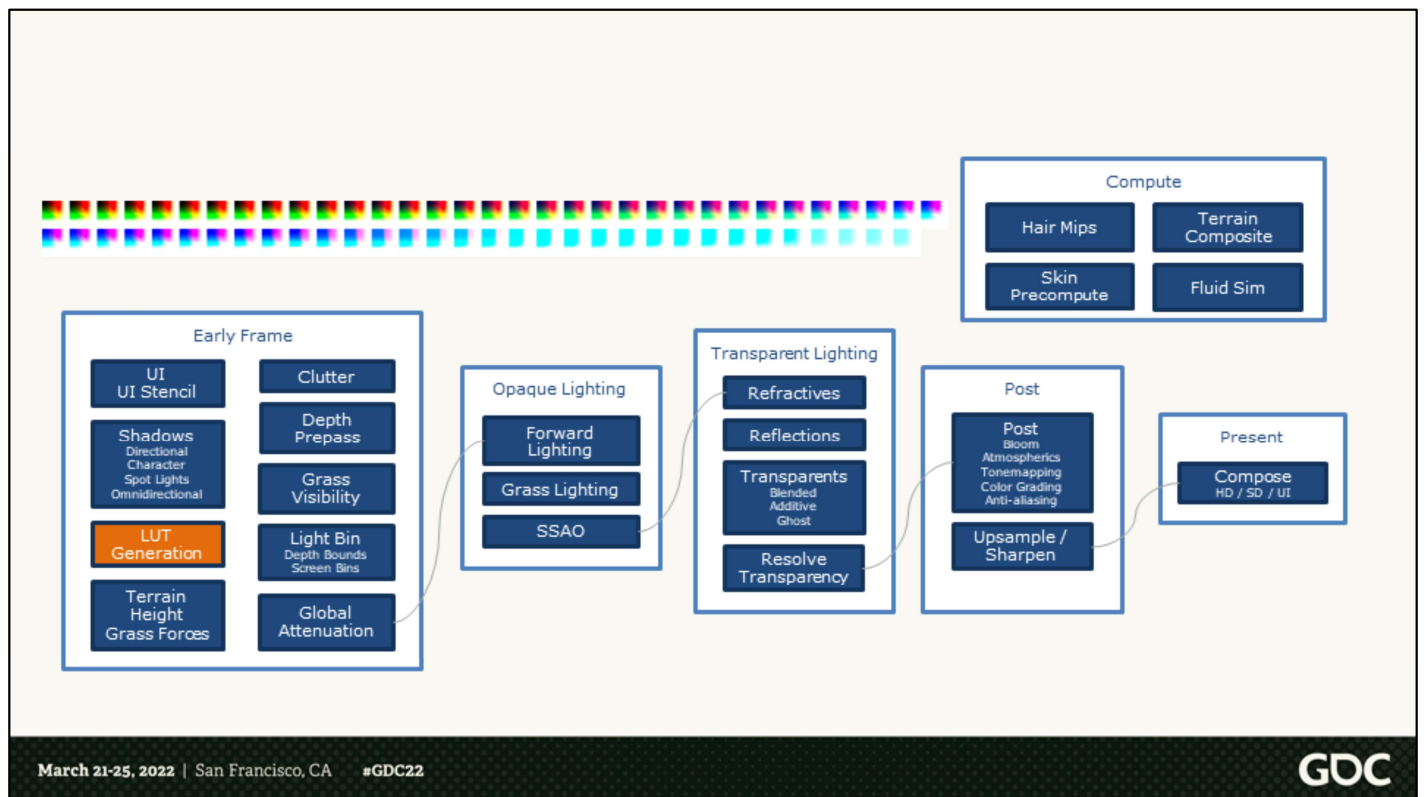


We have UI rendering at the start of the frame in order to generate a stencil to prevent shading those pixels that you won't see behind the UI.

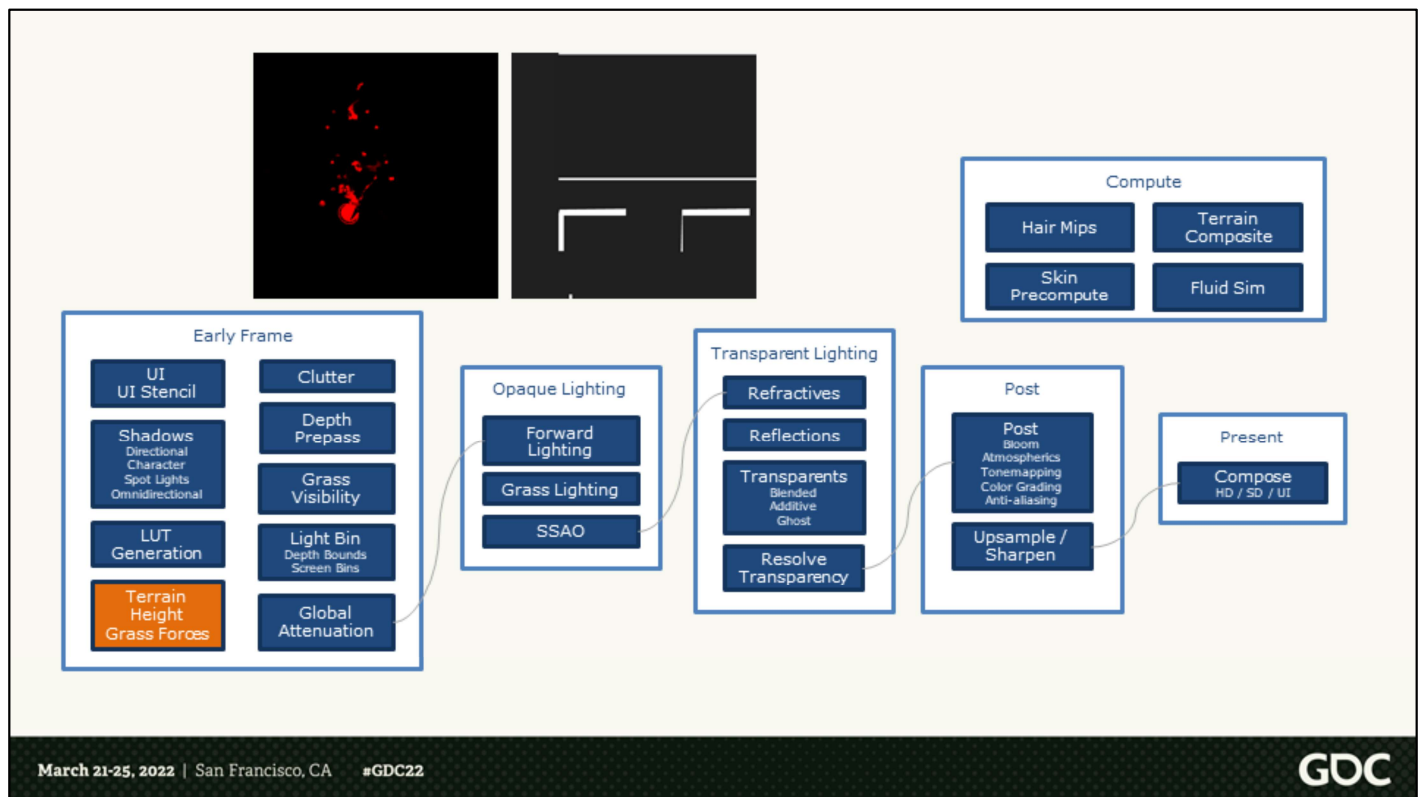




Next are shadows. One main directional shadow caster, another map for character shadows specifically, spot light shadow maps are cached with the results from static geometry, and the omnidirectional player light shadows. Something notable here is that we did not need to spend time implementing cascade shadow maps, because of the advantage of the fixed camera, and limited view range.



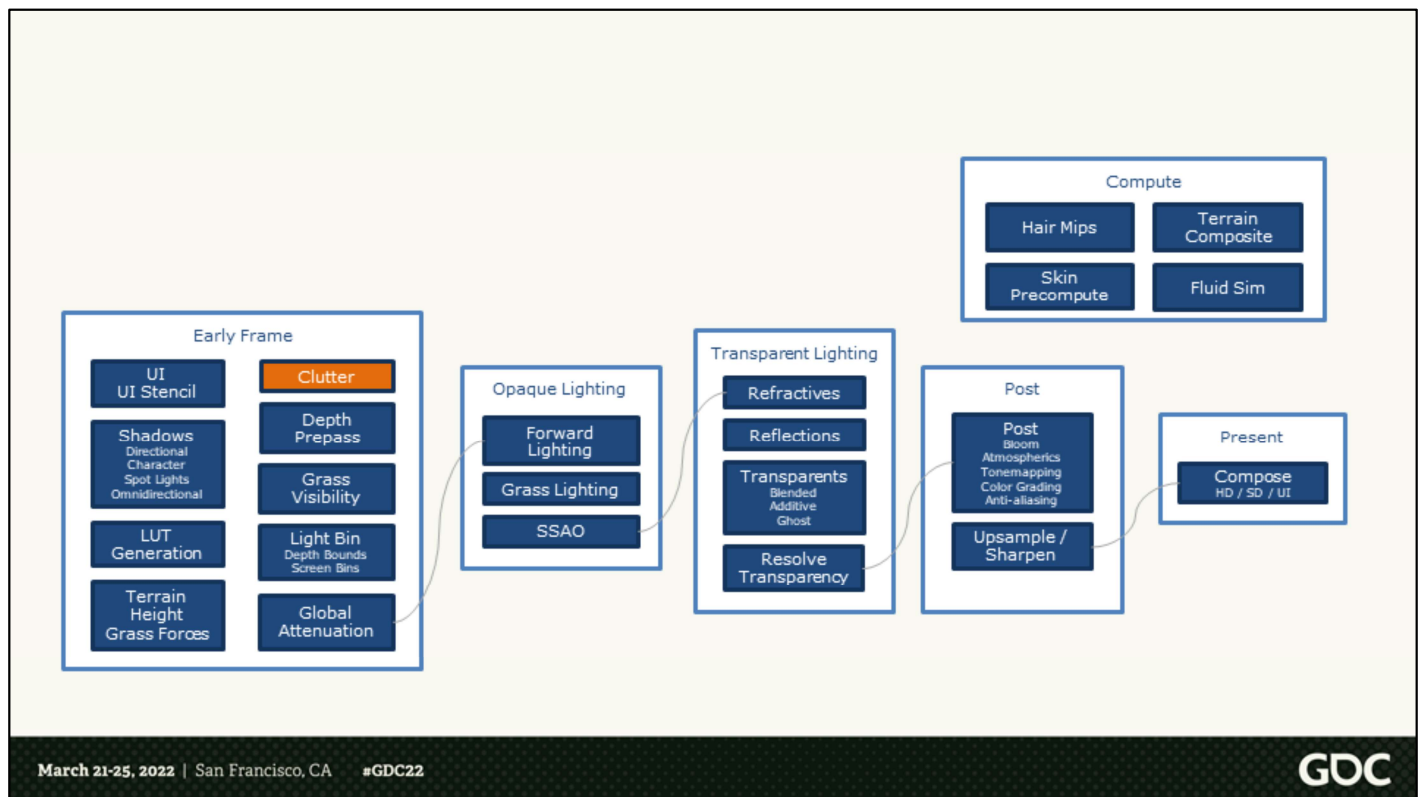
Next we have a LUT generation pass which combines the active color grading LUTs from the scene and also precalculates tonemapping and display mapping so that we can save the cost of the full tonemapping ALU for every pixel. The result is stored in a  $65^3$  LUT and is used later during tonemapping.



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

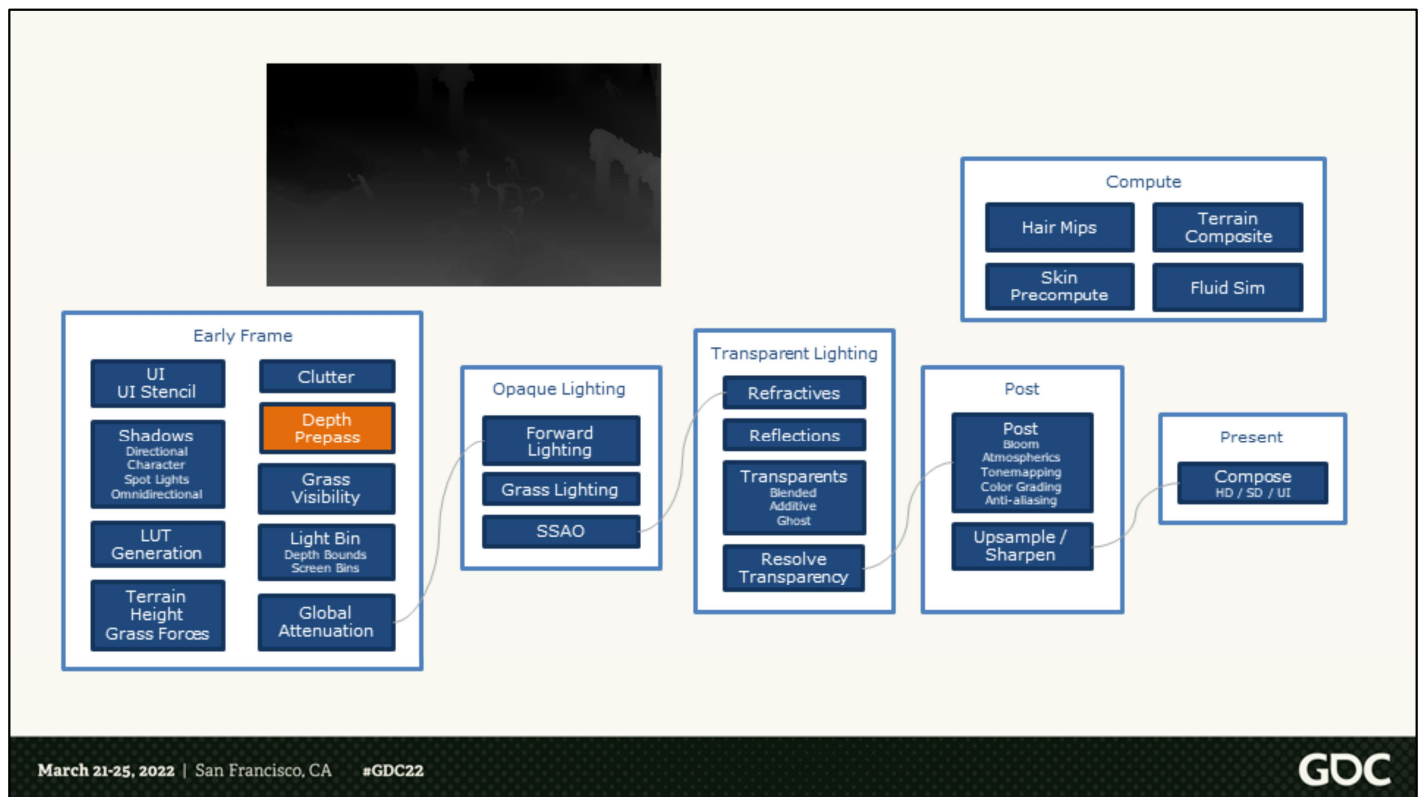
Then we render out the terrain mesh to a height map, which will be used for a few different things including the placement of grass and other ground clutter. We also render things like actors, items, and objects into a top-down view so that they can influence grass that we'll draw later. This is how we can flatten out grass where important gameplay elements like dropped items need to be seen.



March 21-25, 2022 | San Francisco, CA #GDC22

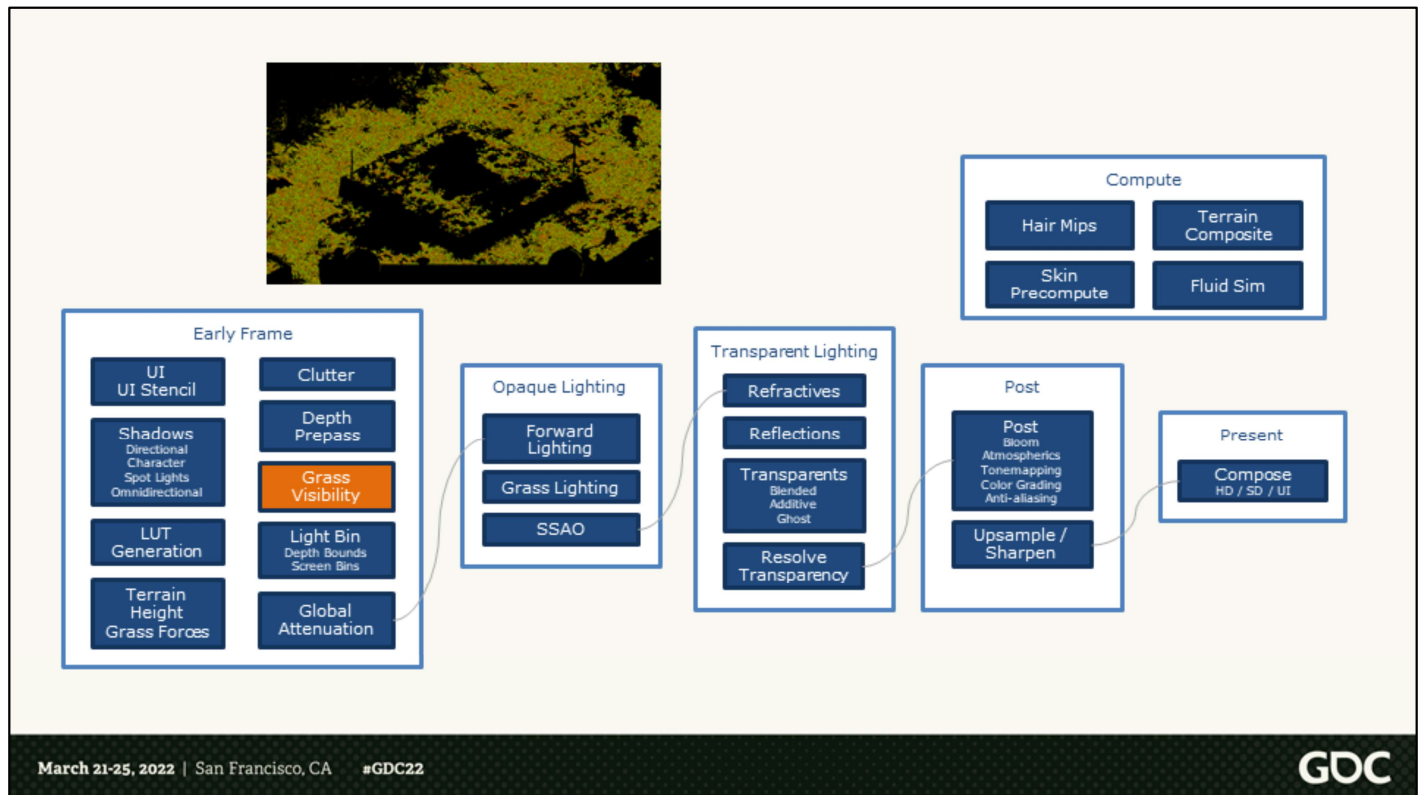
GDC

We then run a compute job using the terrain height and the grass force map to calculate positions and properties for grass blades and clutter models along the ground plane. The system is driven procedurally by artist-controlled thresholds and noise based on world position to determine whether grass or other clutter should appear at particular locations. Each grass blade and piece of clutter are stored to an indirect draw buffer to be drawn during the forward pass.

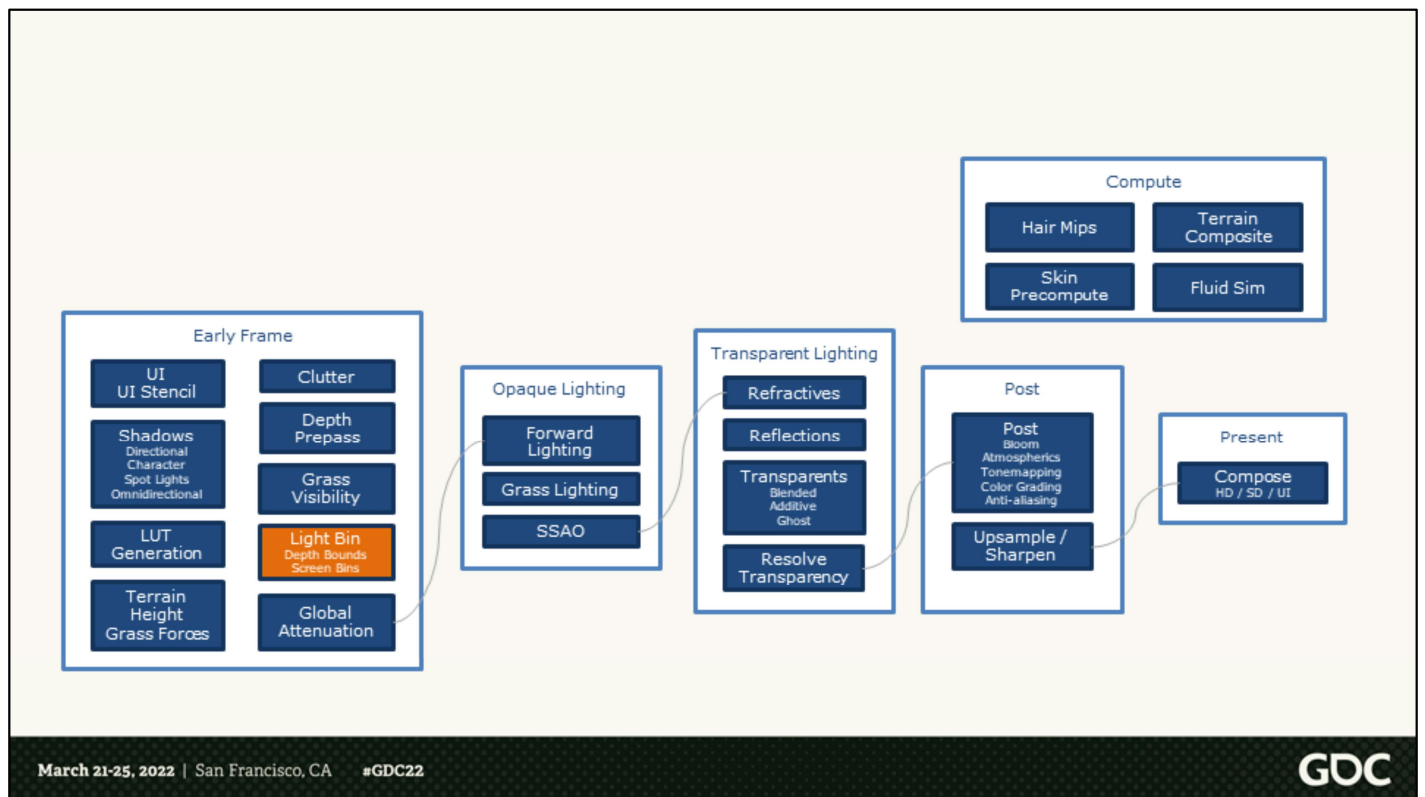


Now we have our depth prepass, which includes the clutter draw call list we just assembled. Something notable about the game is that our depth range is extremely narrow because the camera is relatively far from the scene with a narrow field of view.

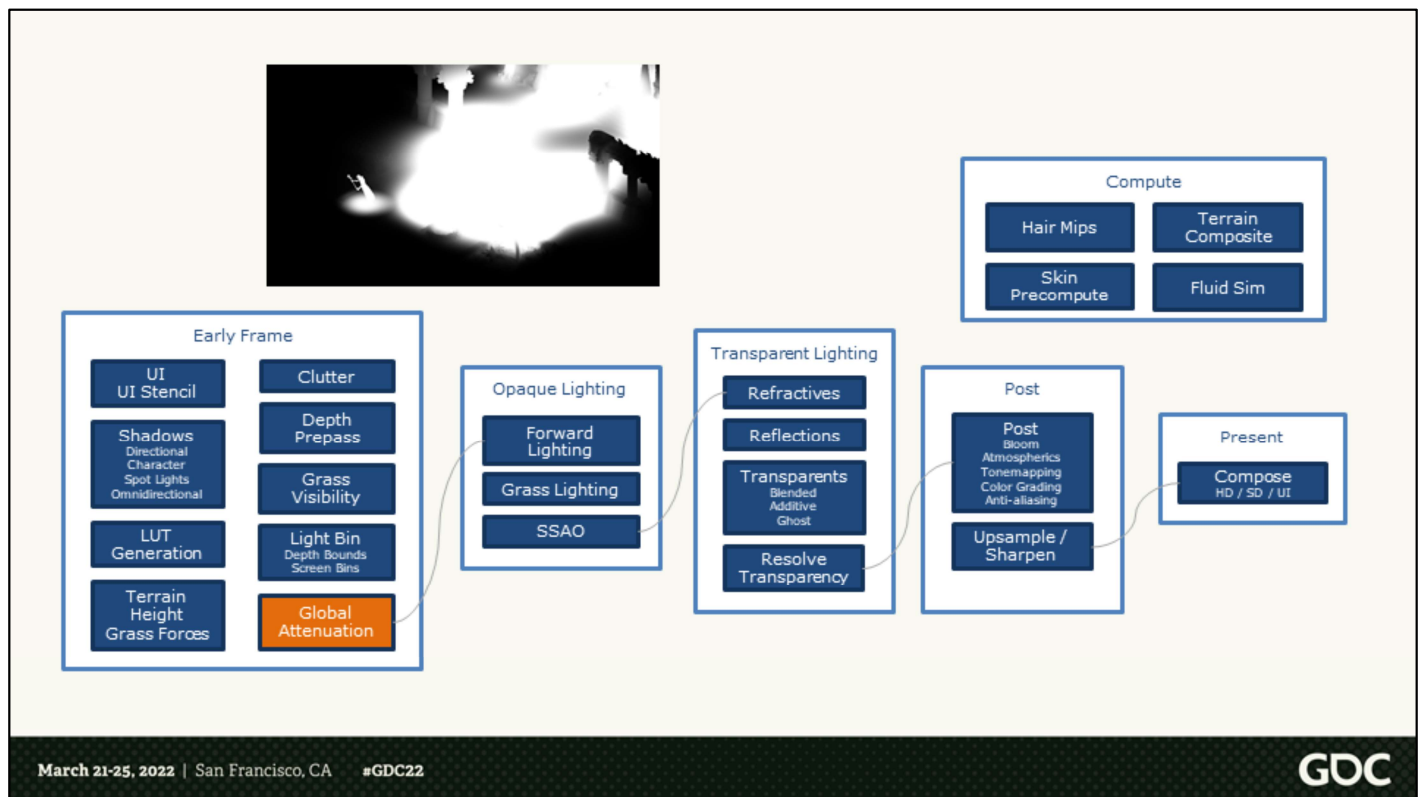




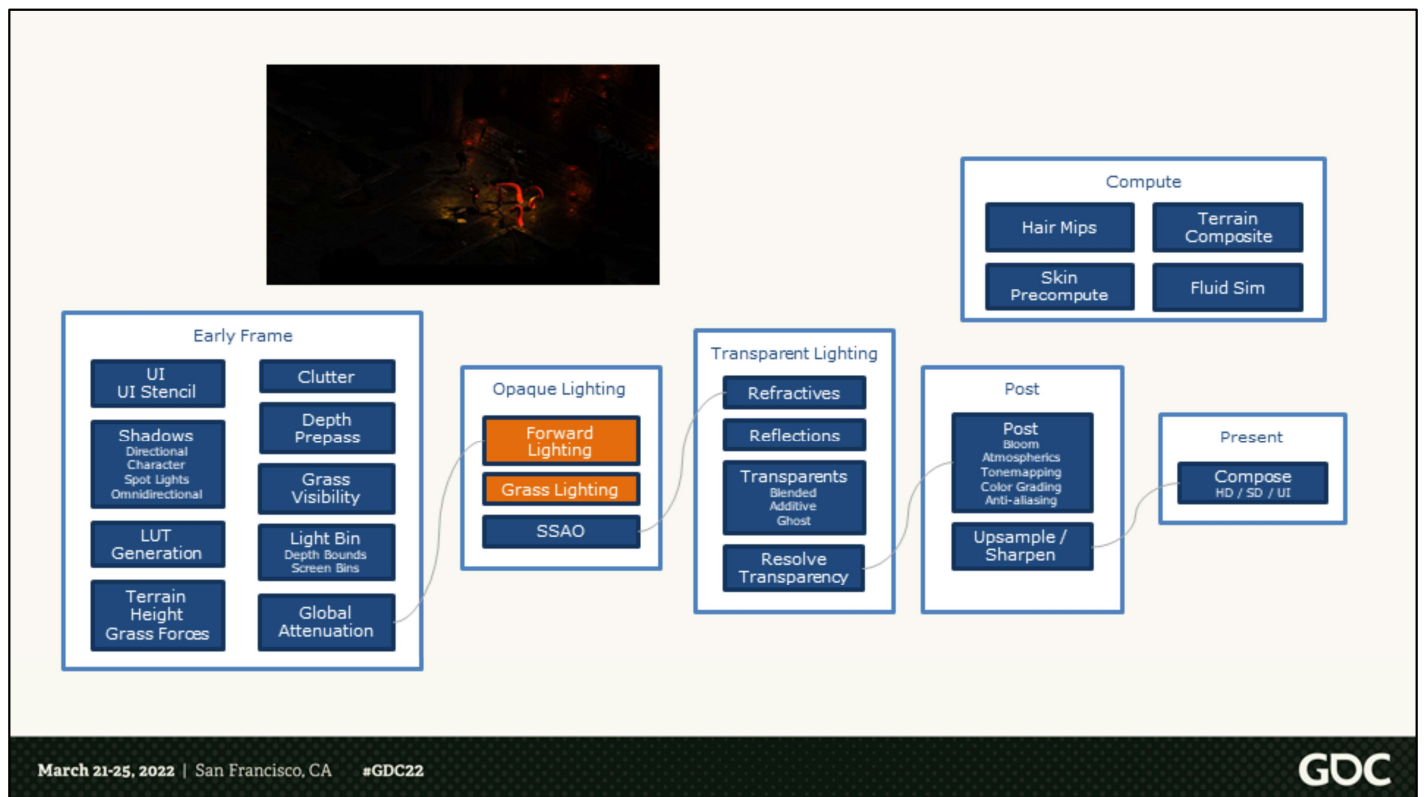
Grass is the one thing that is deferred shaded in the engine. This was done because it gave a significant performance boost on our lower-spec platforms. In this pass we render the grass to a visibility buffer and a data buffer, which allows us to stencil out pixels from the forward lighting pass that will be obscured by grass.



We then bin our lights into screenspace bins. This is done in two compute passes. The first calculates min and max depth bounds for the bin, which allows us to discard lights from a bin if their energy isn't going to reach anything in the bin, in terms of depth. With the bounds established we can bucket lights into screen bins.

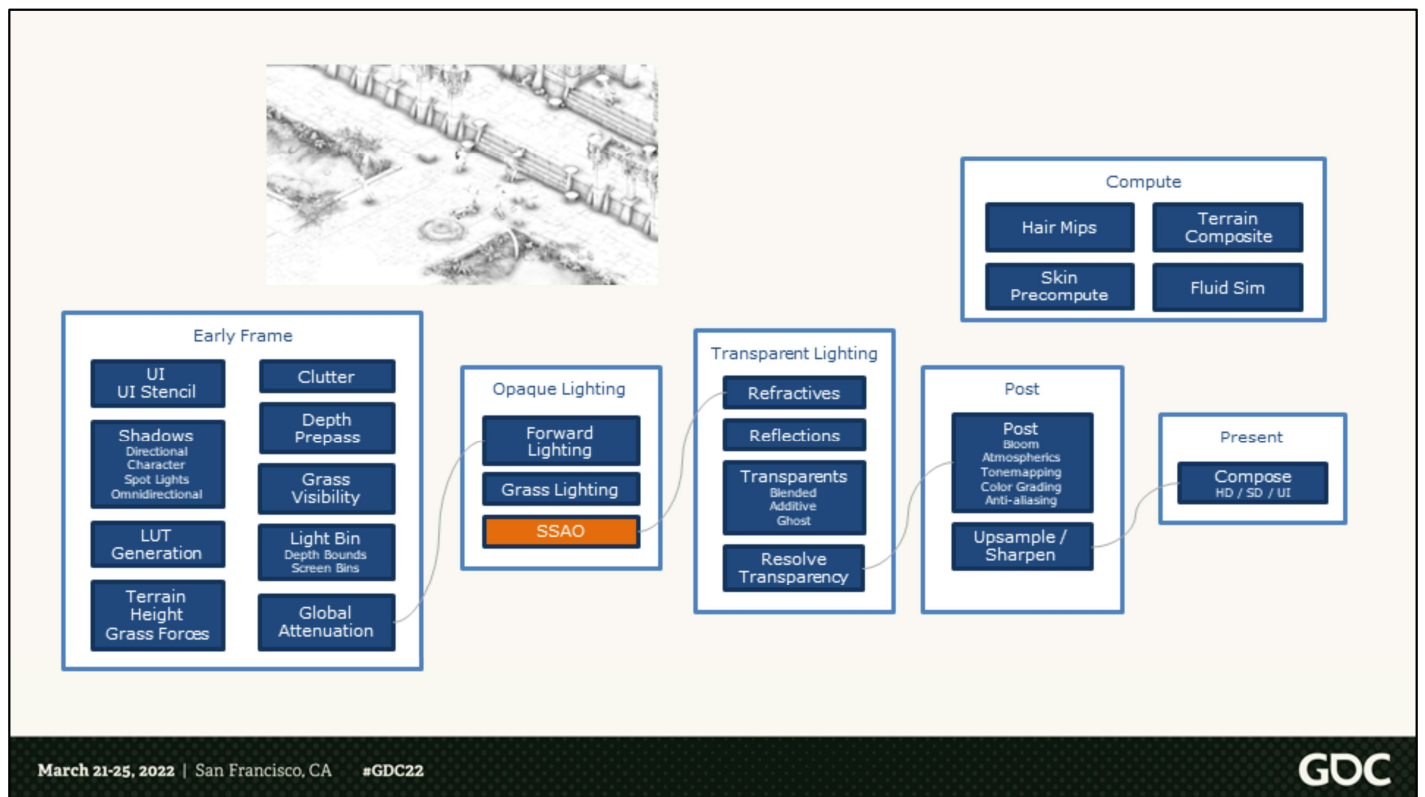


We then use the binned lights and our depth buffer to render a buffer we call global attenuation. This is a bespoke technique specific to Diablo II that allows us to remove ambient light influence from areas that are not otherwise being lit by a punctual light source. This was a critical element to mimicking the visuals of the original game and maintaining the dark atmosphere that Diablo II is known for.

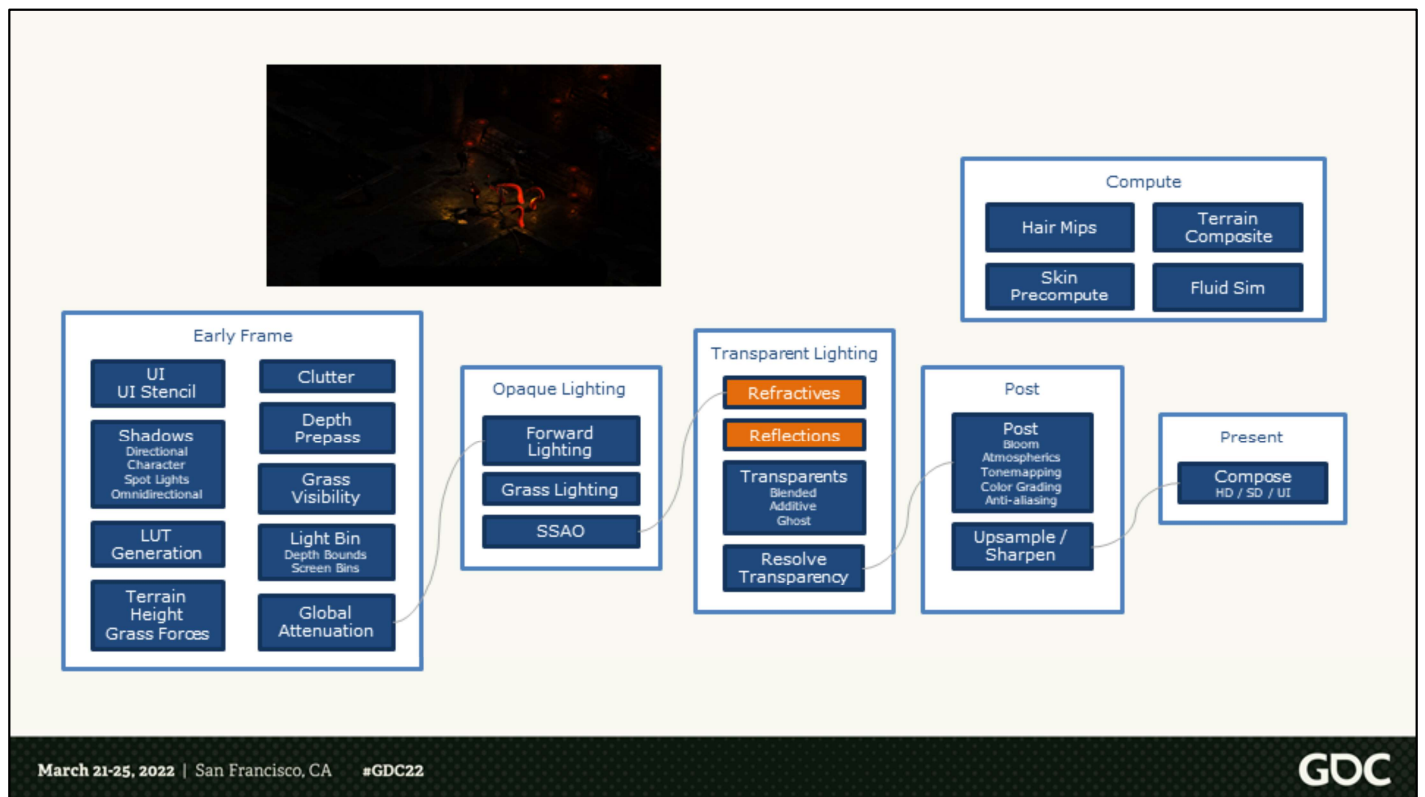


Finally, we get to forward lighting. This will take up a large chunk of the frame as it churns through all the elements of the scene.

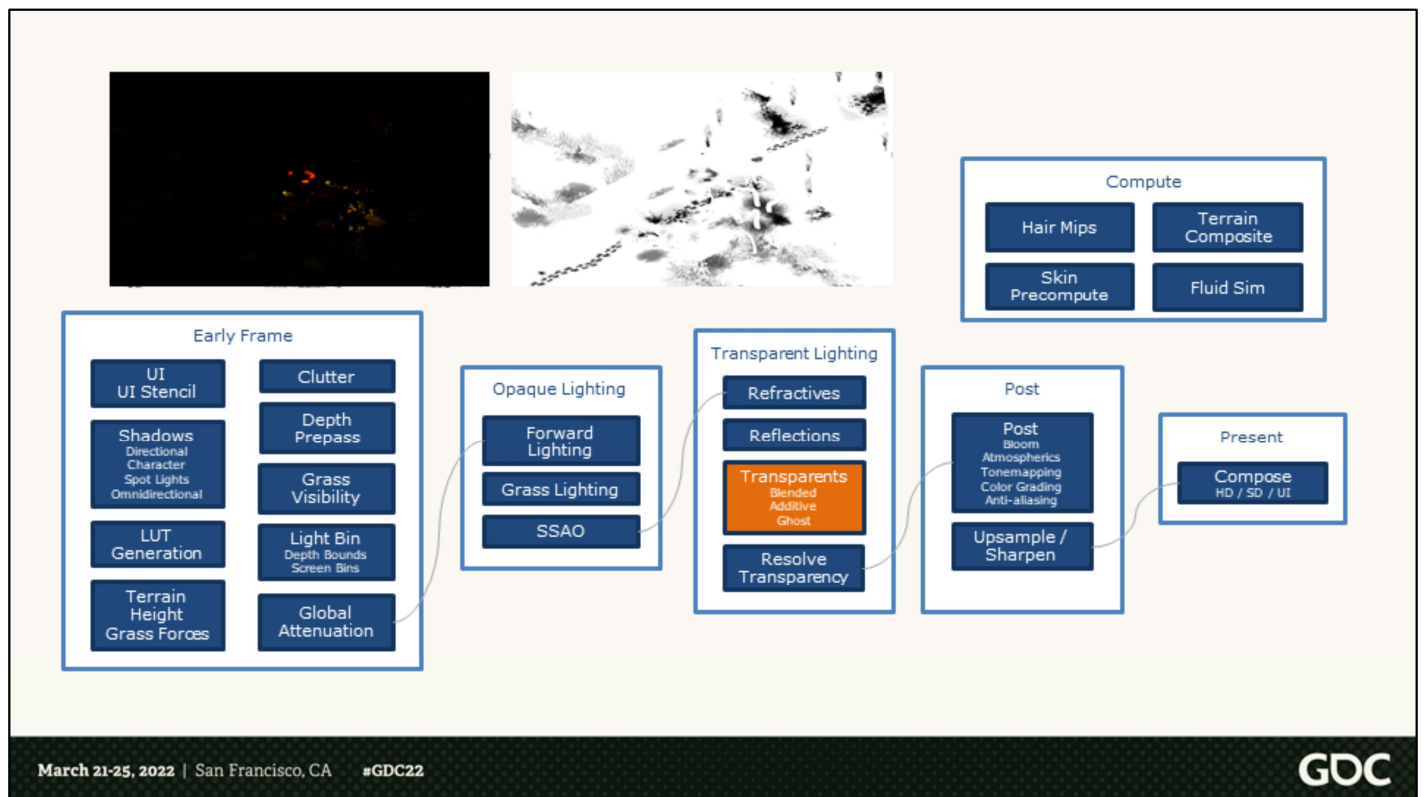
We then perform a deferred shading pass on grass, only on the pixels stenciled from the visibility pass.



With all the opaque geometry rendered, we can do an SSAO pass.

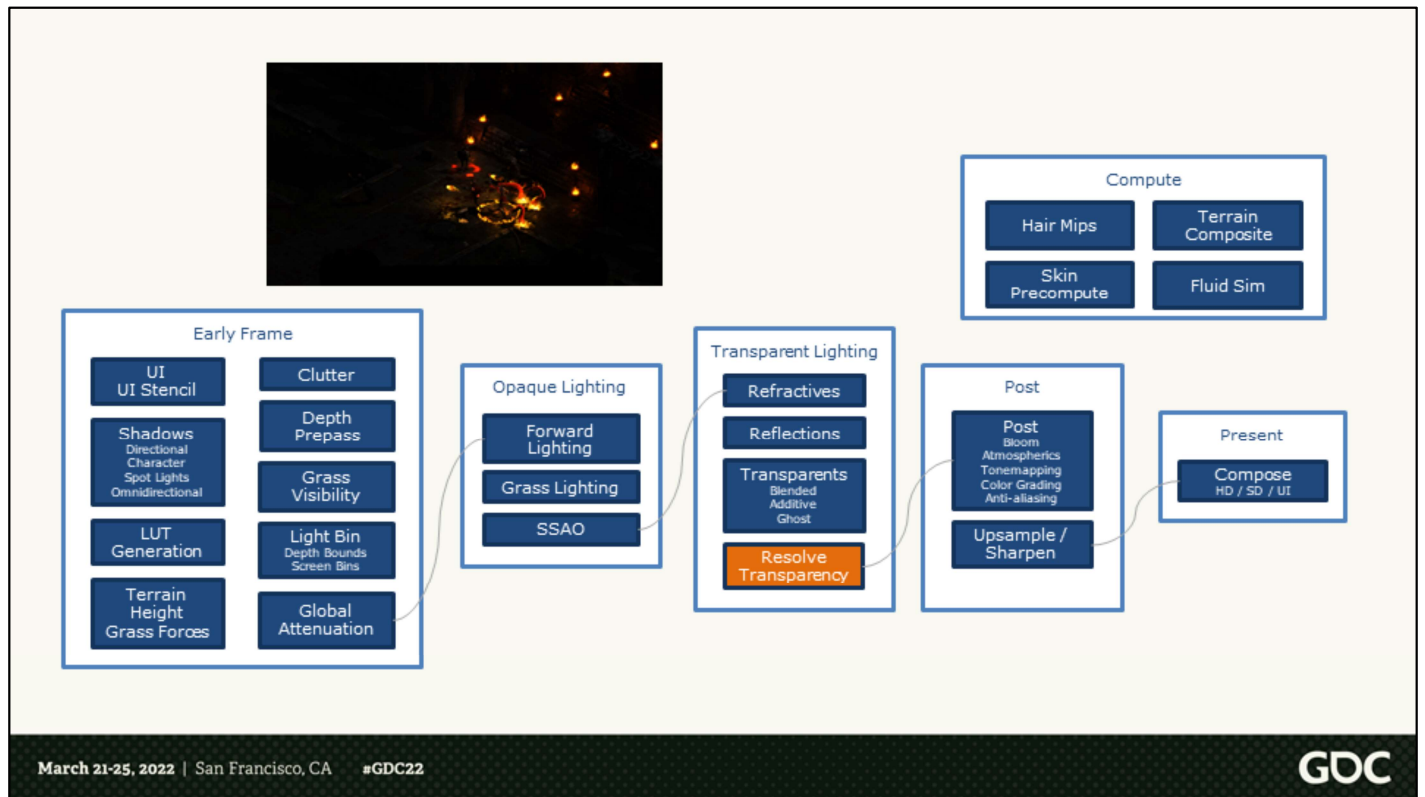


We then calculate mipmaps of the opaque scene, and use that to render out refractive materials like glass and water. We'll also use the mip chain in our screenspace reflections pass.



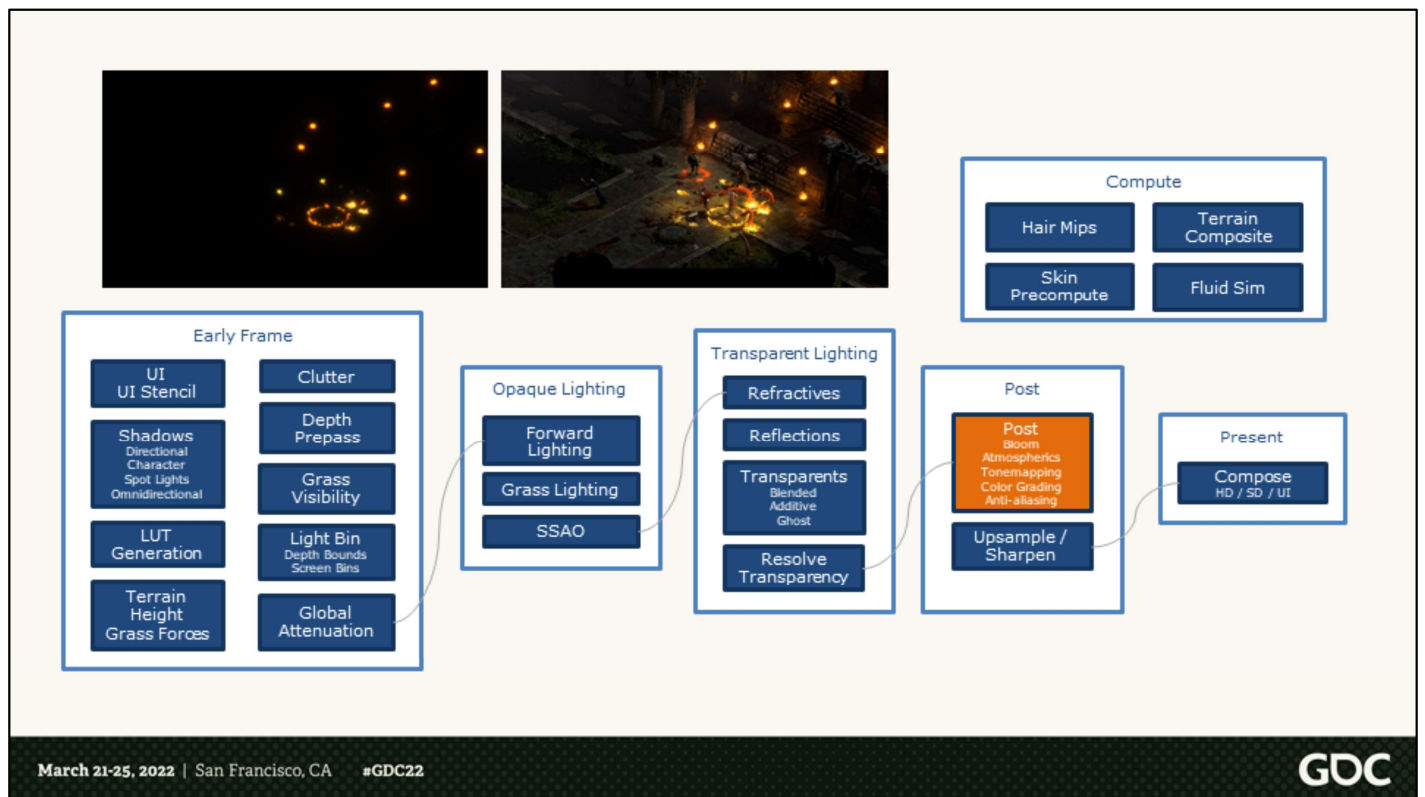
We then render transparent elements to the screen in three batches - blended, additive, and what we called ghost. Ghost are those elements that are typically opaque, but need to be rendered transparent. This includes walls, characters, and items. For example, some skills in the game make your character transparent, and some monsters are ghostly. The ghost technique renders the subject first to the depth buffer, then a second time with an equality depth test so that only the frontmost surface of the subject is shown.

Transparency is rendered in tiers, with some effects rendering at half or even quarter resolution, depending on both individual effect settings as well as the quality level for the platform.



We upscale the transparent mips and then resolve the blended and additive effects buffers to the main framebuffer, and prepare for post.

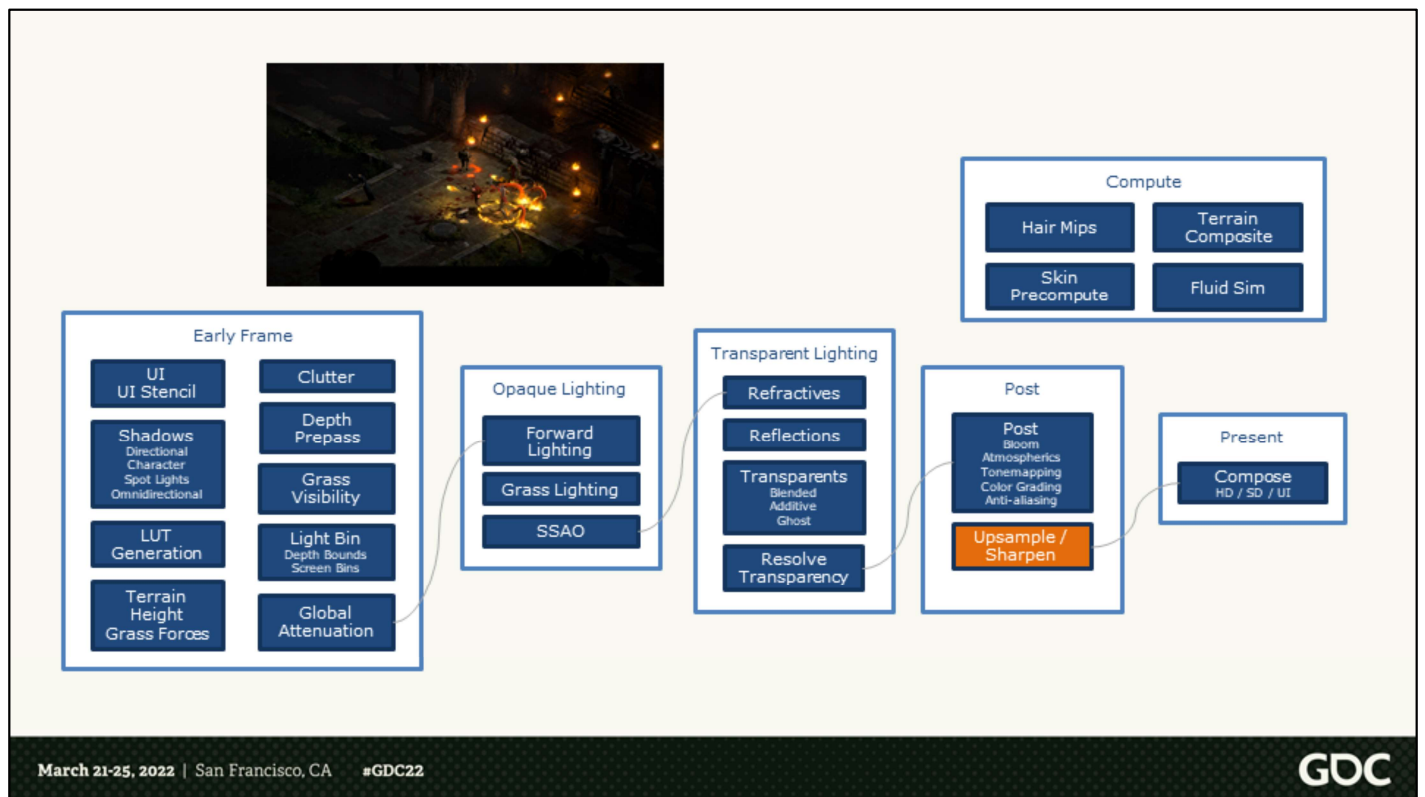




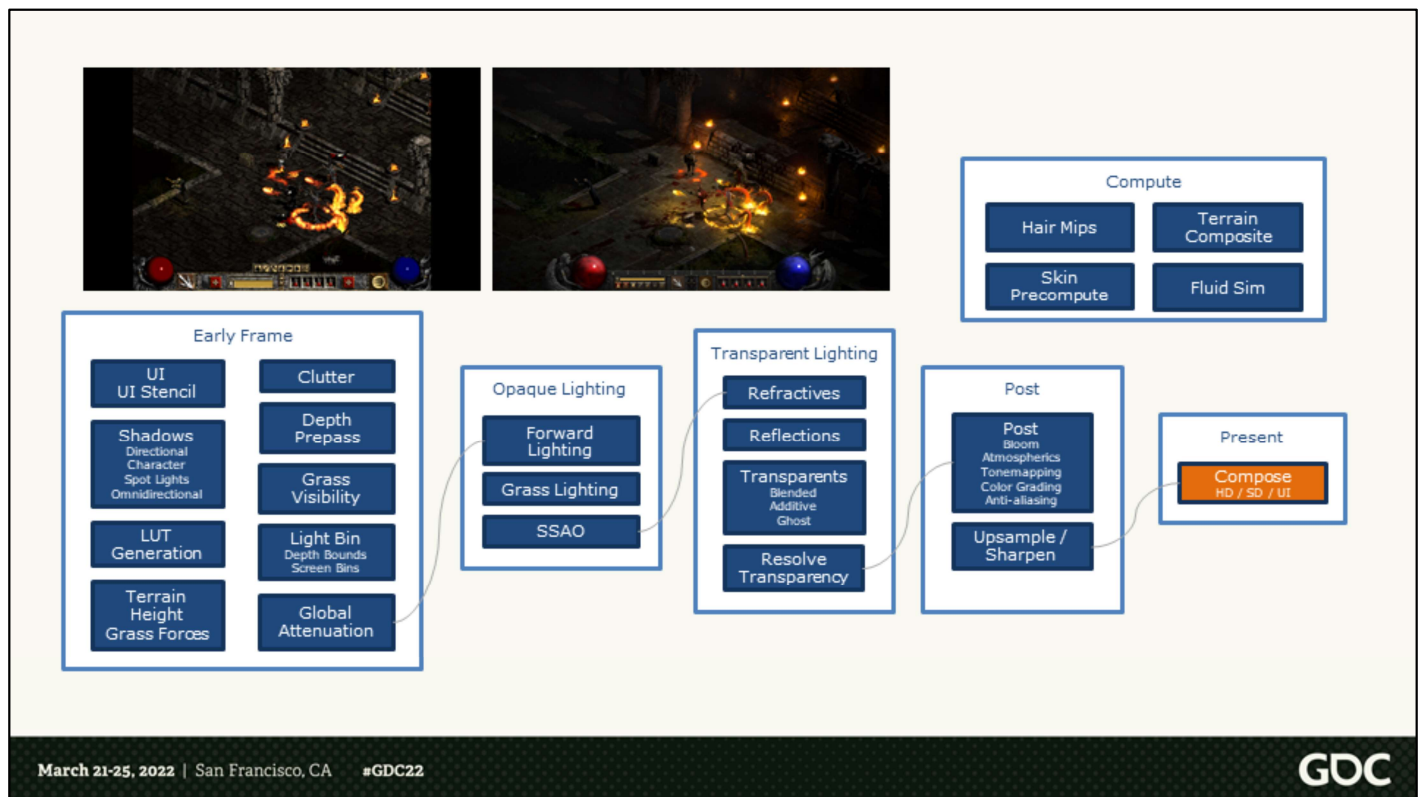
March 21-25, 2022 | San Francisco, CA #GDC22

GDC

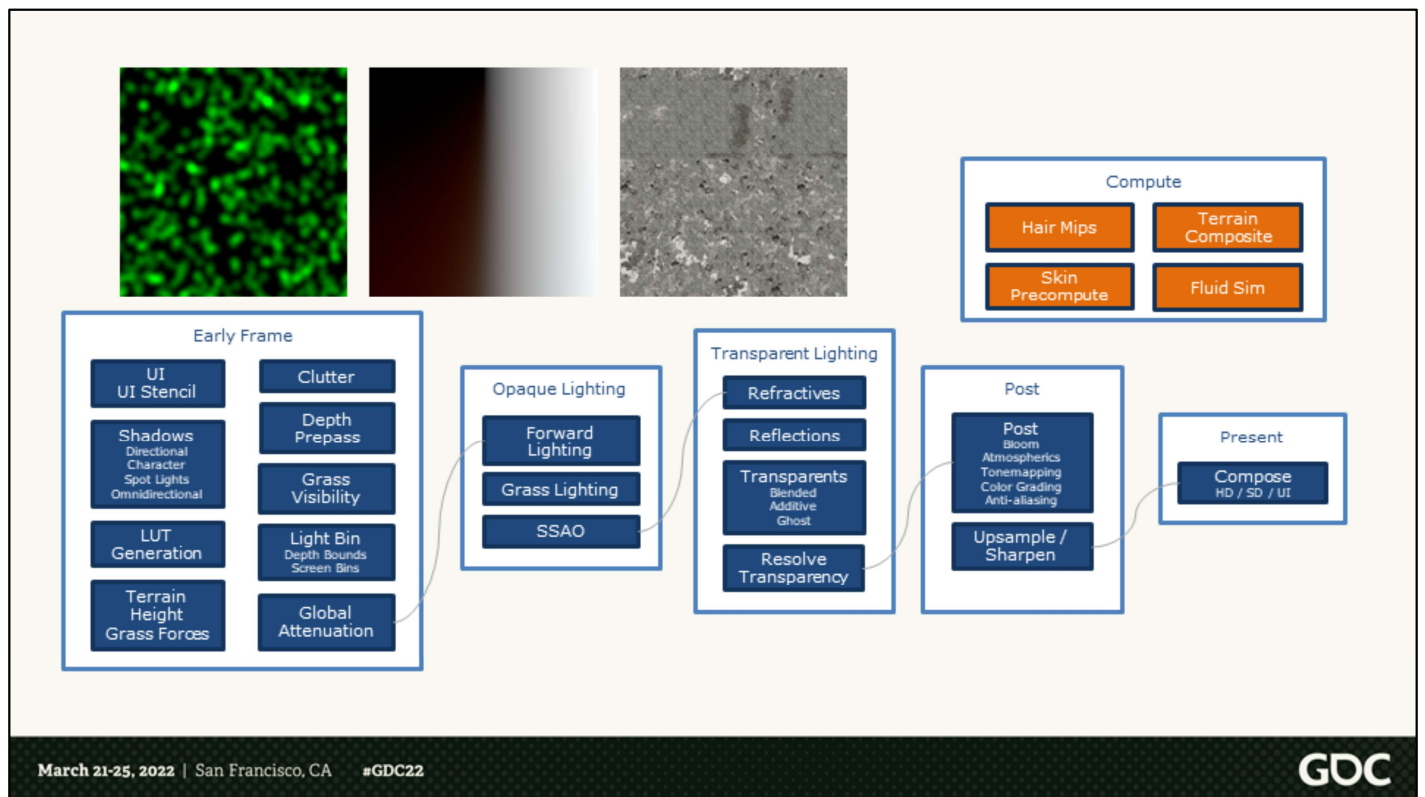
Our bloom is based on the Call of Duty implementation, with the addition of blue noise in order to dither out faint bloom over the often dark backgrounds of Diablo II in order to prevent banding artifacts. Atmospherics only consists of depth fog and height fog, both driven by our time-of-day system, and finally tonemapping applies the generated LUT from earlier in the frame, and we smooth out the edges with temporal morphological anti-aliasing.



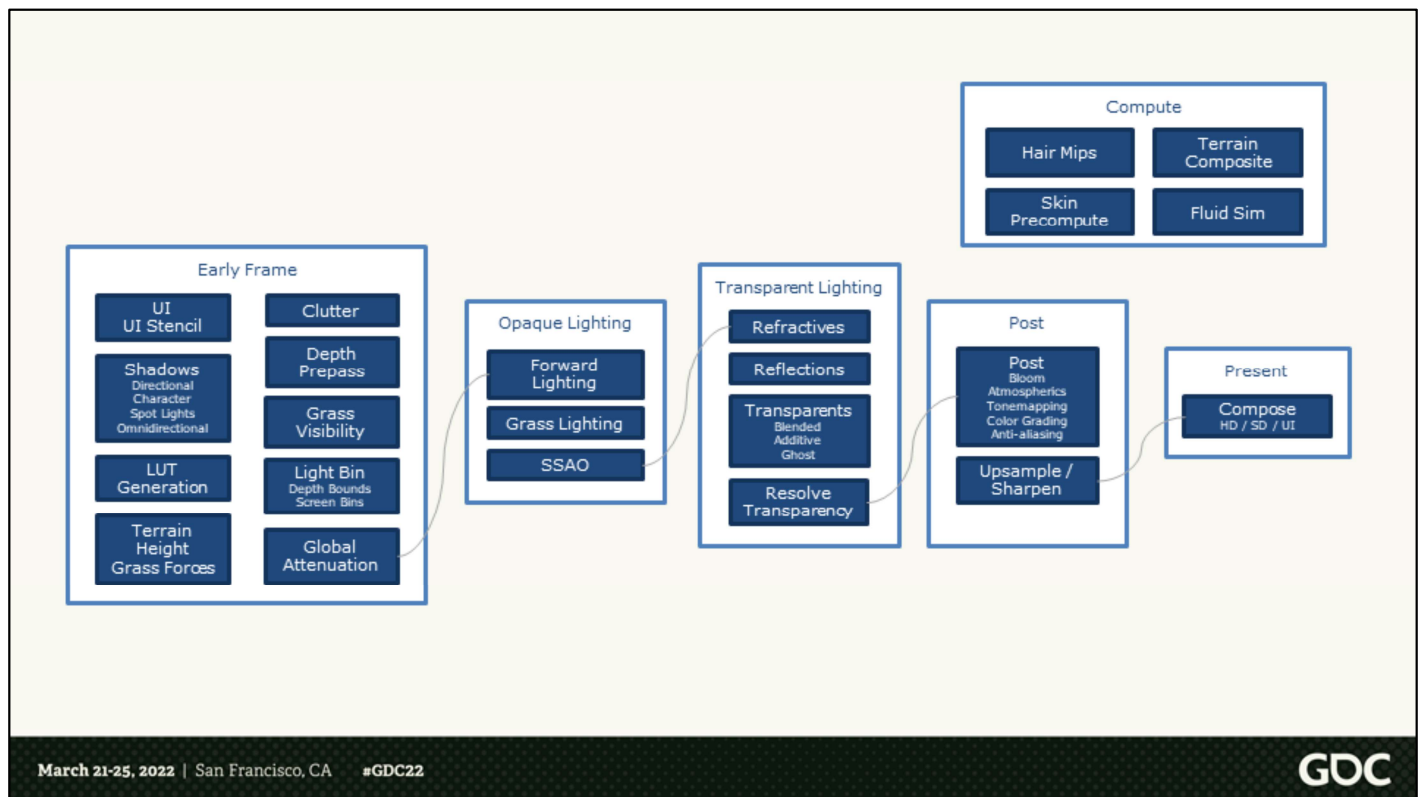
Depending on the platform, perform upsampling and sharpening.



And finally, a bit different than most renderers, we not only compose UI and the 3D scene, but also the 2D scene as well. You could look at the render engine in the game being two completely separate ones -- the original sprite renderer, left largely untouched from what it was; and the new 3D renderer. The legacy toggle feature is simply choosing a fork in the road.



There are also some compute jobs that are either run intermittently, on startup, or asynchronously with rendering. On startup, we precompute a light diffusion texture for skin rendering. As hair textures are loaded, we calculate mips chains for the hair -- the reason for this is that the mip chain calculation for the hair is more involved than traditional mip calculation. And asynchronously we run a job to resolve the terrain layer stack into two large terrain textures used for terrain rendering during the forward pass.



Let's take a closer look at lighting.

# Lighting

- A lot of prior research on PBR
- Based on papers on Unreal and Frostbite
- Important to establish expectations with Art
- Off-the-shelf
  - GGX / Schlick / Smith
  - Multiscattering energy conservation
- High geometric detail promotes aliasing

Our lighting model was inspired by previous writing on physically based rendering on both Frostbite and Unreal. One thing we found helpful as we were standing up the renderer was to establish a contract with our artists that the visual behavior we were looking for would be similar to Unreal Engine, which would allow them to prototype in a space where they'd have some confidence they would understand what the final assets would look like.

We use a GGX specular term with Schlick's approximation and the Smith/GGX correlated visibility term. We also added a multiscattering energy conservation term derived from Fdez-Aguera 2019.

One notable thing that we also included in our forward pass is a term to reduce specular aliasing. Our art team pushed visual fidelity very high, and many of our art assets are extremely dense in polys. As a result of the far viewpoint of the game camera, the sampling rate even at high resolutions is insufficient to represent the high geometric normal variance on much of the world geometry. The result is a high amount of specular aliasing.

# Specular Antialiasing

- Idea: modulate roughness by geometric variance.
  - Kaplanyan, 2016 and 2019
  - High variance = high sub-pixel roughness
- Paper uses derivatives of half-vector
  - We have too many lights

Our solution to this was based on a technique proposed by Kaplanyan et. al. from 2016, filtering the NDF using variance of the half vector across the shading quad. While intended to be adaptable to a forward renderer, making it an ideal candidate for us, using the half vector was not feasible for performance reasons. Because we were aiming to have so many dynamic lights in the scene, we'd be dealing with dozens or even hundreds of half-vectors to calculate quad derivatives for. We really only had room to calculate the derivatives for one. The spirit of the technique is to leverage the derivatives to understand the surface complexity of the geometry being shaded, and that's also possible to determine using the surface normals, so we do this instead.

```
float3 nDu = ddx(geometricNormal);  
float3 nDy = ddy(geometricNormal);  
float variance = 0.5f * (dot(nDu, nDu) + dot(nDy, nDy));  
linearRoughness += variance;
```

Here's a code snippet. We take in the geometric normal and get the derivatives across the quad, then use the same variance calculation from the paper and add that to our linear roughness value used for shading.



# Scalable Lighting

- Scalability is a key philosophy.
- ... but there's no baked lighting.
- How to make the many lights problem scalable?

Now, I mentioned at the beginning how a key philosophy to development was to make everything scalable. I also mentioned that we didn't have any baked lighting, in order to embrace the dynamism of spells. So, an appropriate question at this point is "how do you make the many lights situation scalable?"



Here is somewhere we embraced the advantage that the game camera is fixed – always the same angle and distance from the scene. This allows us to maintain a 3D lightmap of the area surrounding the player and render lights into this 3D radiance map during a preprocessing step in the frame.

# Scalable Lighting

- Two 32x8x32 FP16 textures.
  - One for accumulated light direction relative to the local grid point.
  - The second for the accumulated radiance of nearby lights.
  - Radiance is encoded with a power function to improve interpolation between cells.
- Resolution of the maps can be varied.
  - Light maps are only used on platforms below recommended specification.
- Radiance and direction are sampled during forward lighting.
  - Evaluated like a directional light.

The lights are encoded with a compute shader into two 32x8x32 fp16 textures, for precision. One texture accumulates the directionality of the light relative to the grid position, while the second encodes the light's radiance. The resolution of the maps can vary to tradeoff between quality and performance. On our recommended specification or above, we just calculate complete lighting for every individual light and do not use the light maps unless under very heavy load.

We then use these light maps during forward rendering by calculating the position within the map of the point being shaded and using the radiance value as if it were a directional light.



Here's a scene where there are several instances of an effect, charged bolt, that each have a light. In this case, every light is being individually calculated.



When we use the radiance map much of the diffuse lighting is preserved but the drawback is that the impact to specular highlights are mixed. The highlight from the torch is very close, but the highlights from our effects have almost disappeared in some places.



# Scalable Lighting

- Calculate direction reflected around floor normal.
  - Weight light intensity by how close reflected vector is to view direction.
- Approximates specular highlights off of flat surfaces.

To attempt to reclaim some of the complexity lost in the form of specular highlights, we also included a weighting to the light intensity of light directions that are close to the view direction.



And going back to original radiance map image, this is where we were.

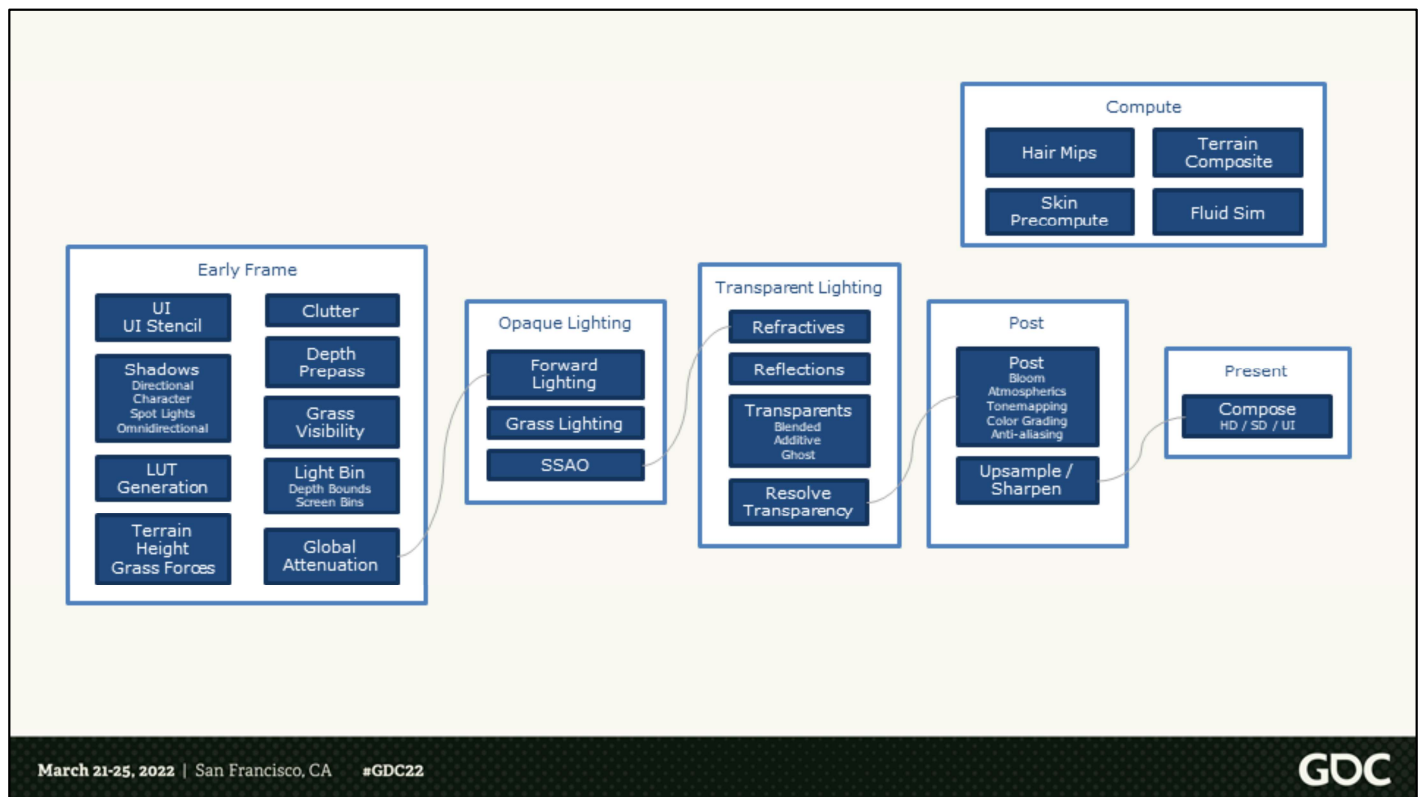


And this is what we get with specular weighting.





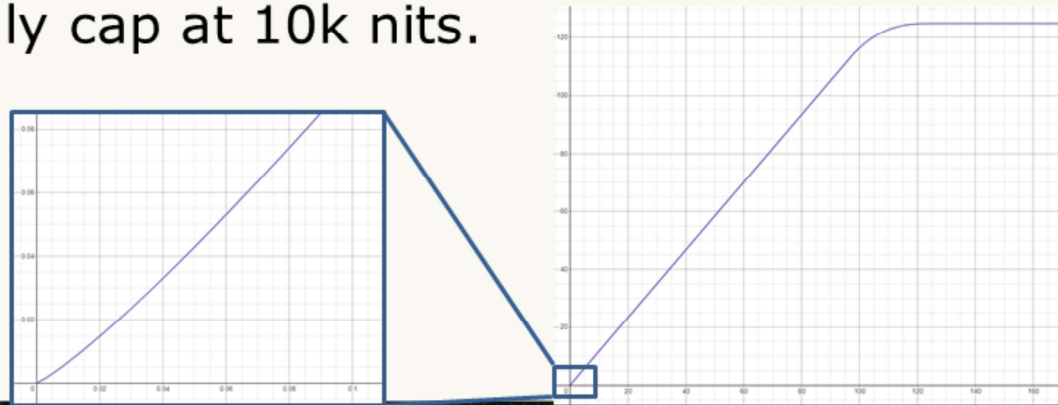
Now if you directly compare this to the original, it's closer at giving us nice specular but isn't close to the ground truth. This is of course, a contrived example. When there are many effects flying around the screen and we're under heavy load, the difference is less noticeable and was considered an acceptable tradeoff.



Let's take a look at tonemapping and color grading next.

# Tonemapping

- Minimal “look” with the tonemapping curve.
  - Leave that to color grading.
  - Only cap at 10k nits.



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Our approach to tonemapping and color grading is based on the work done for Call of Duty. This was an area where we had an opportunity to try new ideas because it aligned with our expertise. The work from Call of Duty reinforced the idea of two separate tonemapping operations – tonemapping as look development, and display mapping for the target display. In D2R, we gave some, but very limited artistic control over the tonemapping curve, to the point where it was nearly linear. The toe deviated only slightly from linearity. The curve's primary purpose was to round the peak brightness in the scene off to 10,000 nits, the peak brightness of the HDR standard. The idea was to move most of the responsibility for look development to color grading.

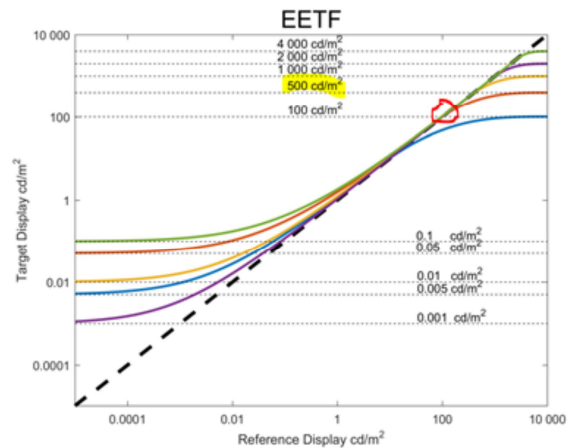
# Display Mapping

- LDR -> sRGB
- HDR -> BT.2390

22

Rep. ITU-R BT.2390-3

FIGURE 20  
Example EETFs of various target displays

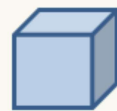


The second tonemapping curve is contextual to the display -- typical sRGB for LDR displays, and a variable tonemapper for HDR displays that can be adjusted depending on the peak brightness of the display. For this curve we use the BT.2390 specification just like Call of Duty.

# Color Grading

- Previous work
  - *HDR in Call of Duty*
  - *High Dynamic color grading and display in Frostbite* – Alex Fry
    - <https://gdcvault.com/play/1024253/High-Dynamic-Range-Color-Grading>
- Key improvement:
  - Remove the need to embed the grading LUT into the image data

For color grading we built on previous work from both Call of Duty and Frostbite by leveraging Davinci Resolve but removing the need to embed the grading LUT in the sample image.

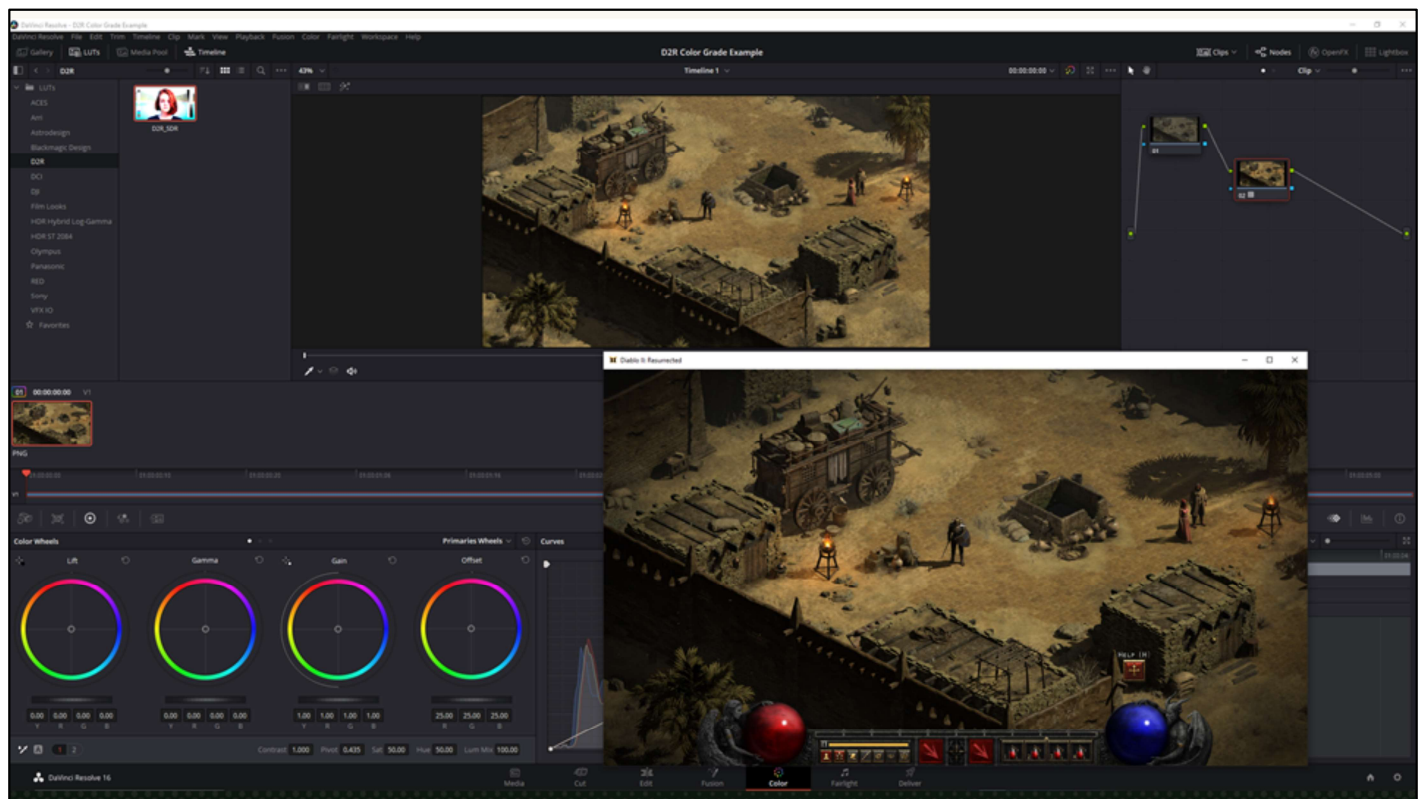


.cube LUT

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

A linear HDR screenshot of the game is exported in EXR format. The screenshot is taken without any tonemapping or post-processing applied. We also export a LUT that encodes our LDR tonemapping pipeline.

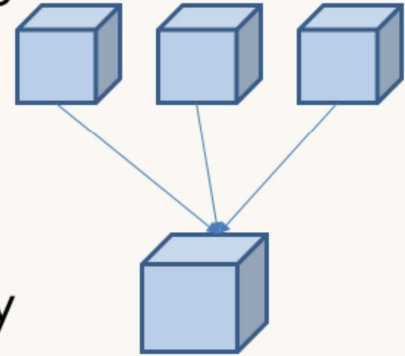


Over in Resolve, the project can be set up by importing the EXR screenshot and configuring the project to apply our LDR tonemapping LUT. This ensures that what the artist is seeing in Resolve is an exact match to what they can expect to see in the game. When an artist is done grading a scene, they can export a 65x65x65 LUT also in the cube file format as the color grading data for that scene.



# Color Grading

- Cube LUTs are built into 3D DDS textures
- Color grading sources:
  - Time of day
  - Visual data volume
- Combined into universal CLUT
  - Plus tonemapping and display mapping
- Applied prior to anti-aliasing



At build time, these cube files are consumed by our build process and converted into 3D dds textures, which are then compressed and ready to be used at runtime.

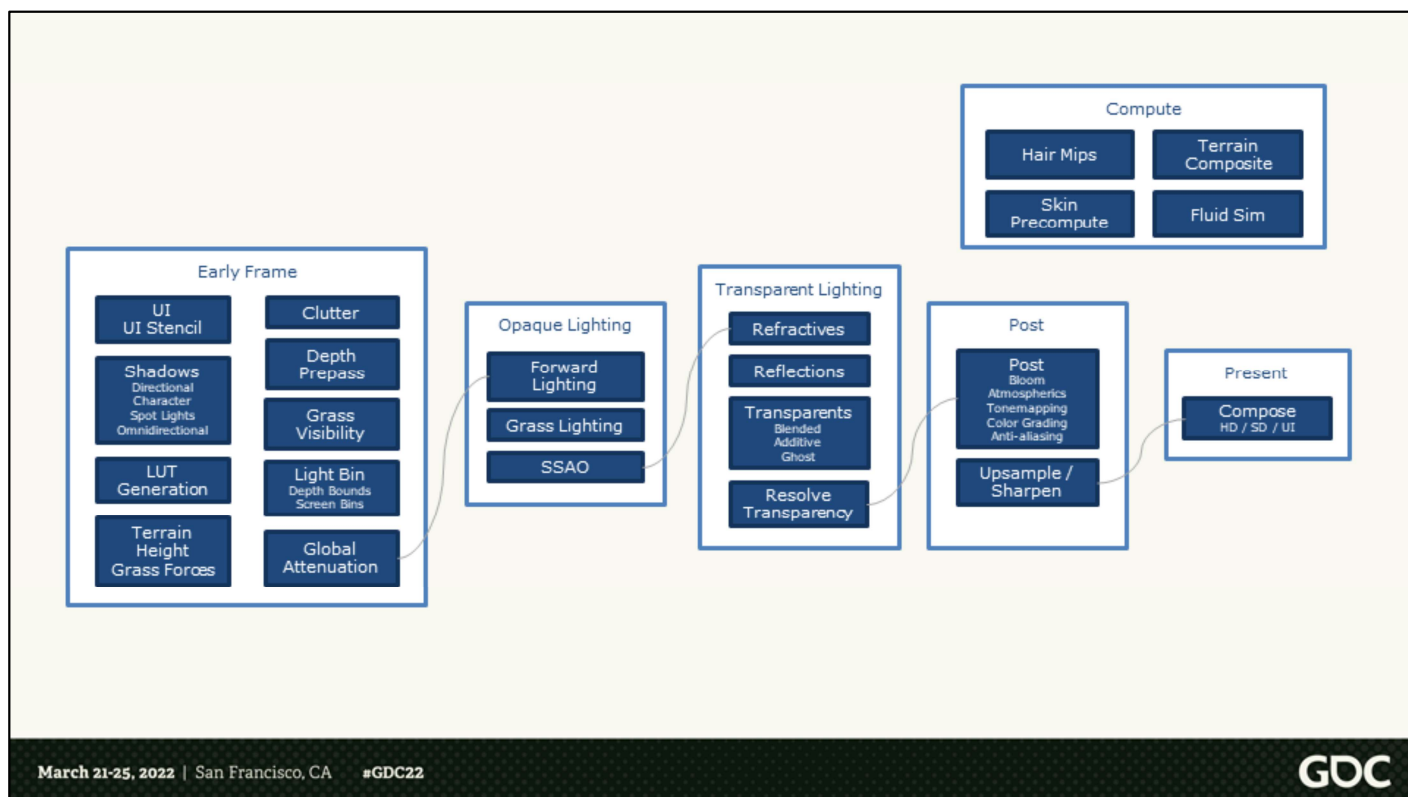
There are two systems through which an artist can leverage color grading in the game – time of day and visual data volumes. At any point in time during gameplay, grading LUTs from two time-of-day keyframes are combined along with any active visual data volume. These three LUTs are combined in a compute process into our universal CLUT, which also includes both tonemapping and display mapping, and is applied just prior to anti-aliasing at the end of the frame.



# Precision

- Most samples are concentrated in mid-tones.
  - Undersampling can lead to some serious artifacts.
  - Good explanation in *Advanced Techniques and Optimization of HDR Color Pipelines* – Timothy Lottes, GDC 2016
- Solution is to use a shaper function
  - We used the PQ curve
- Opportunity
  - PQ is not the cheapest
  - A cheaper shaper function can be considered a micro-optimization

In the 3D LUT, precision can become an issue. The majority of our samples out of the LUT are going to be at the low end [maybe a histogram view of the sampling pattern out of the LUT?]. This gives us insufficient precision if we store the values linearly, so a shaper function is required for better distribution. We chose PQ as our shaper function, but I'd like to note that this is far from optimal as the PQ encode/decode scheme involves several transcendental functions.



Let's talk about transparency next.

# Order-Independent Transparency

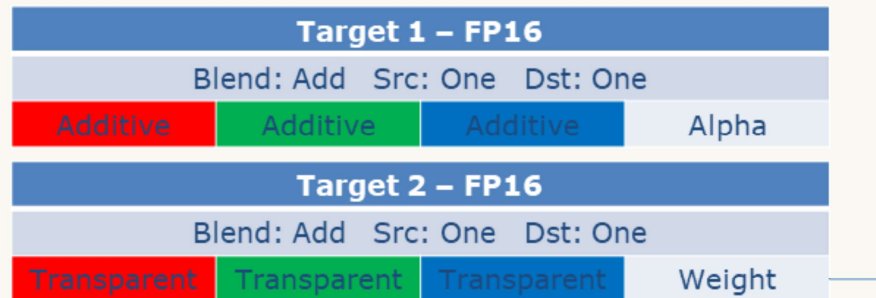
- Why?
  - Diablo II is a very heavy on effects and spells.
  - The game camera yields a very narrow depth range for content.

The first thing to talk about here is why we chose order-independent transparency – OIT – over traditional ordered transparency methods. The primary reason was the density of effects we expected to have in the game, and the limited depth range we expected to be working with. The combination of these two things made us anticipate many situations where the sorting of effects would become unusually complex.

# Order-Independent Transparency

- Four blend modes
- Two output targets

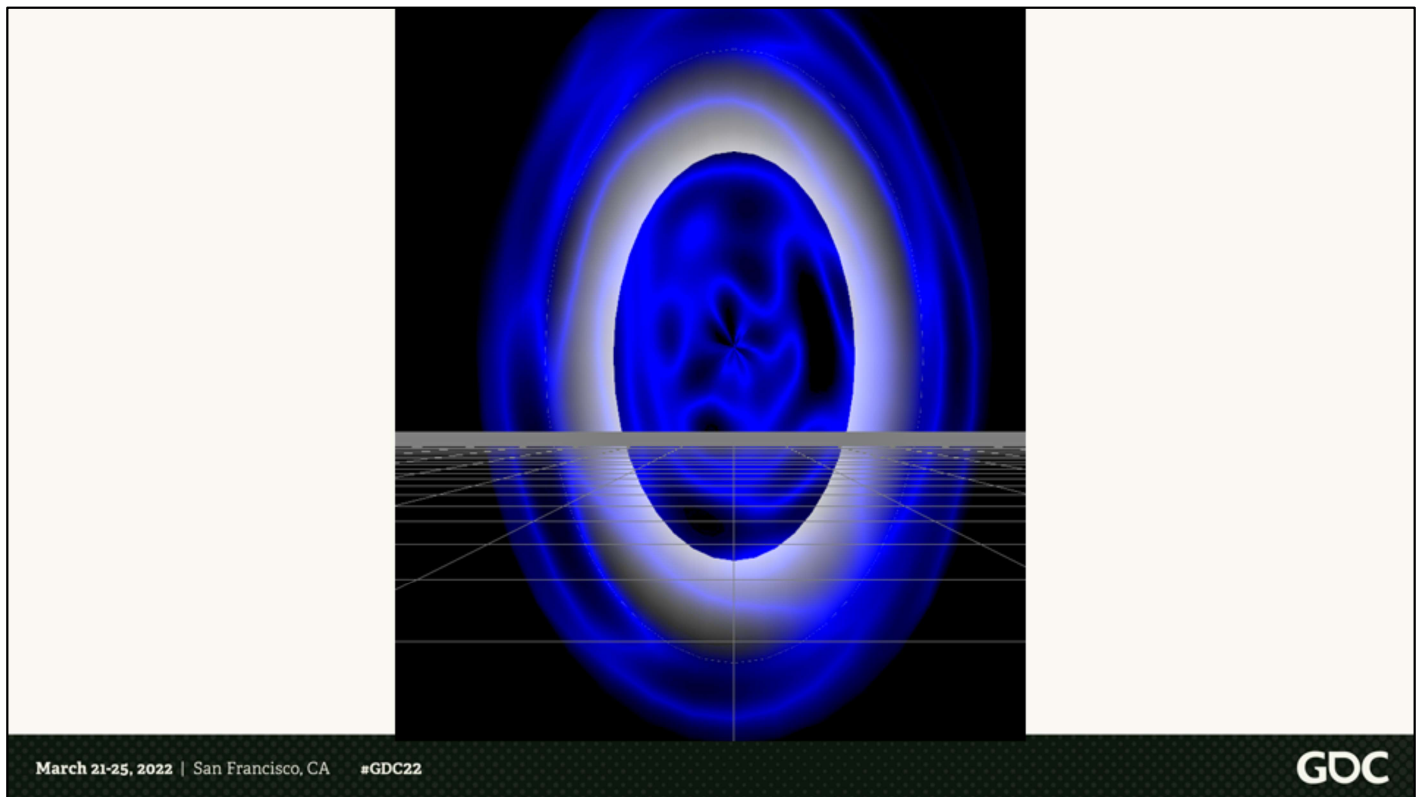
```
float z2 = z * z;  
float z3 = z2 * z;  
return max(0, 1e6 / (1e5 + z3));
```



Our initial implementation was based on a weighted blended order-independent transparency paper from JCGT by Morgan McGuire. It supported four blend modes – additive, additive with alpha, premultiplied, and alpha blend. All transparent objects would be rendered into two targets – a color buffer containing the additive color and the effect's alpha value, and a transparency buffer containing blended effect colors weighted by alpha and depth, with a weight encoded into the alpha channel. Our weighting formula went through several iterations over time but we settled on this calculation here. This first version has 128 bits per pixel of bandwidth during transparent rendering, which we'll see became the main focus of optimization.

```
float4 color = g_color.Sample(g_sampler, In.uv);  
float4 transparency = g_transparency.Sample(g_sampler, In.uv);  
if (transparency.a > 0)  
{  
    transparency /= transparency.a;  
    color.rgb = lerp(transparency.rgb, color.rgb, color.a);  
}  
  
return color;
```

To resolve the two buffers, a fullscreen pass reads the transparency value of a pixel and weights the transparent color, then the alpha value of the pixel lerps between this color and the additive or scene color.



Before we get to optimization, one of the things we learned throughout development is switching an art team from traditional sort-based transparency to order-independent transparency is not an automatic sell. There were a lot of growing pains. We got many questions during development about how to make effects sort differently or how to manipulate the weight of different effects to force them to sort in a particular way. It required a lot of collaboration on the part of the team to help our effects artists get the results they wanted.

Much of the issues we had were around additive effects. The first problem fielded was about how additive effects were being dominated by alpha blended cards. A frequent question we would get was “how do I make this card sort in front of this other card” to which the response was usually a less-than-helpful “well... there is no sorting.”

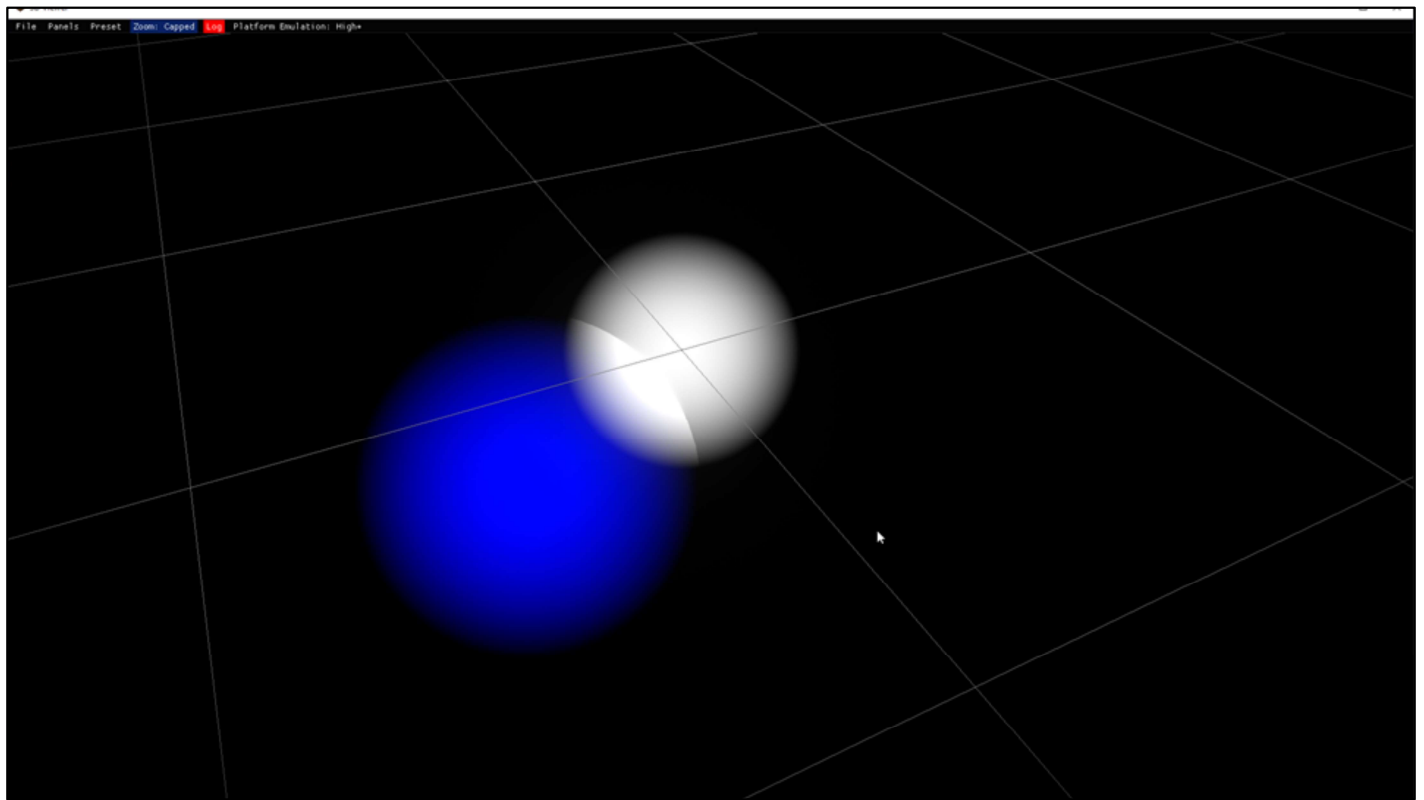
Additive /  
Additive Alpha

Target 1 – FP16			
Blend: Add Src: One Dst: One			
Additive	Additive	Additive	Alpha
Target 2 – FP16			
Blend: Add Src: One Dst: One			
Additive * Weight	Additive * Weight	Additive * Weight	Alpha * Weight

Premultiplied /  
Alpha Blend

Target 2 – FP16			
Blend: Add Src: One Dst: One			
Transparent	Transparent	Transparent	Weight

Our first solution to this was to put some additive color into the transparent buffer, so that it had influence over both color contributors in the resolve step. This worked for a time, but it wasn't long before our effects artists found some new problems.



The line of questioning essentially became “additive effects are not really behaving like we expect additive effects to behave.” In this case, we can see the additive white sphere blending unexpectedly with the alpha blended blue sphere where there is overlap. Our team’s expectation was that the white sphere would just plainly draw over whatever was behind it.

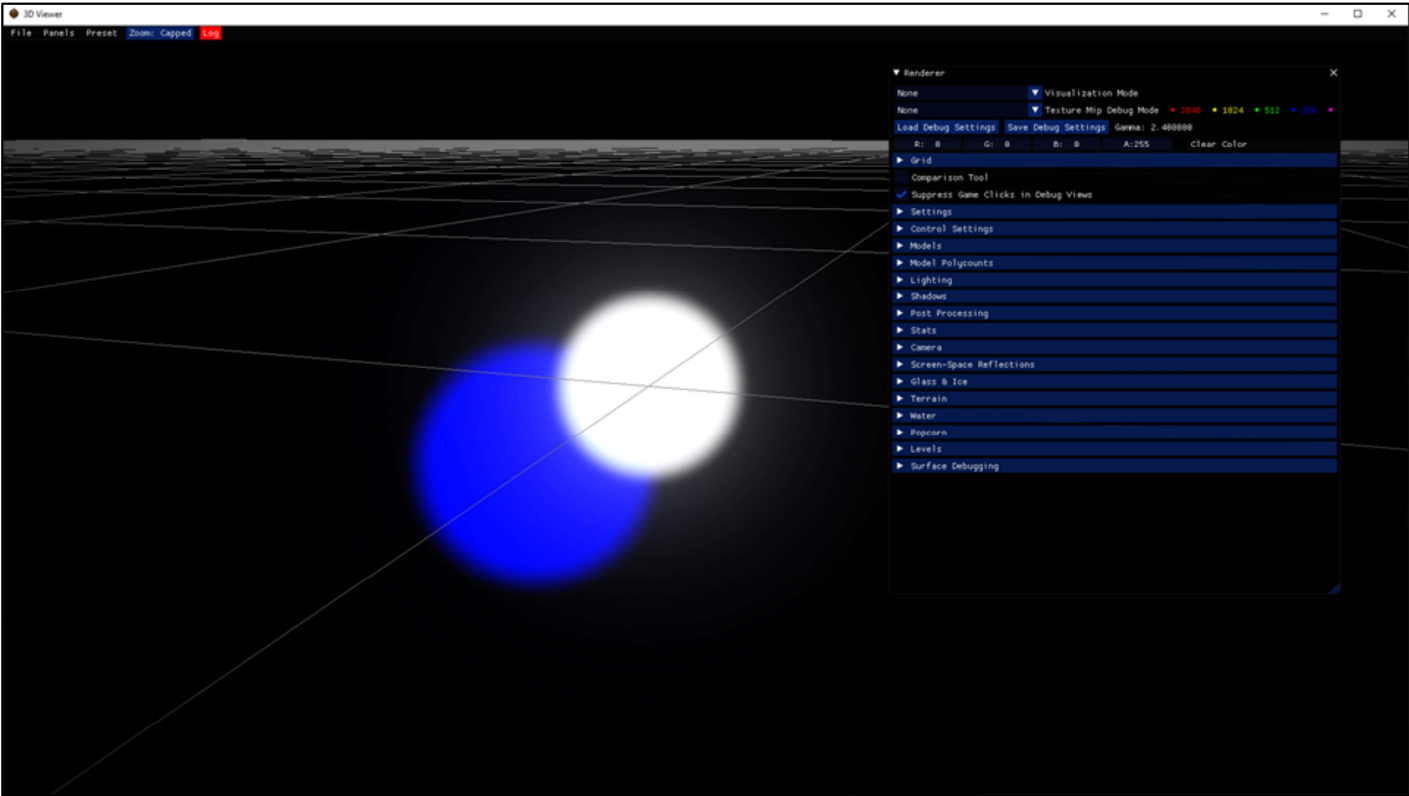


```
float4 color = g_color.Sample(g_sampler, In.uv);  
float4 transparency = g_transparency.Sample(g_sampler, In.uv);  
if (transparency.a > 0)  
{  
    transparency /= transparency.a;  
    color.rgb = lerp(transparency.rgb, color.rgb, color.a);  
}  
  
return color;
```

To solve this, we separated out additive effects into their own render target and modified the resolve so that additive effects always drew over blended effects. This aligned better with our artists' expectations of how additive effects would work.

```
float4 color = g_color.Sample(g_sampler, In.uv);  
float4 transparency = g_transparency.Sample(g_sampler, In.uv);  
if (transparency.a > 0)  
{  
    transparency /= transparency.a;  
    color.rgb = lerp(transparency.rgb, color.rgb, additive.a);  
}  
  
color.rgb += additive.rgb;  
return color;
```

To solve this, we separated out additive effects into their own render target and modified the resolve so that additive effects always drew over blended effects. This aligned better with our artists' expectations of how additive effects would work.



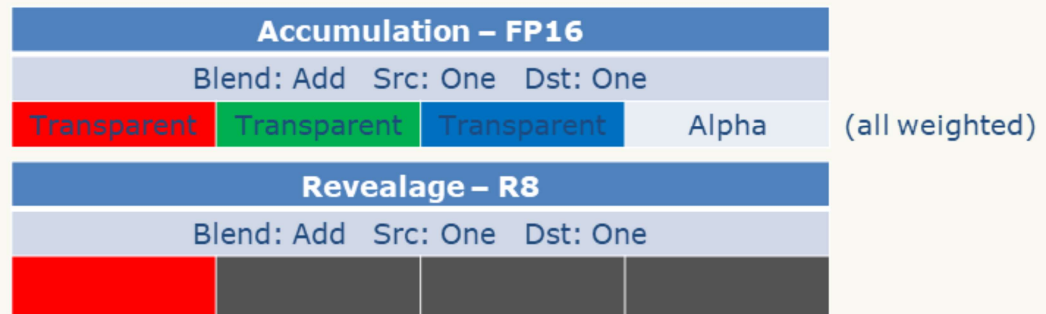
# Optimizing OIT

- Observation: separation of additive effects means transparency can be simplified.
- Idea:
  - Put additive effects into a different render pass.
  - Reduce information required for blended effects to color, weight, and alpha value.

Now, return to optimization, 128 bits-per-pixel is a high output bandwidth requirement, and this harmed the performance of our transparents rendering on lower-end platforms.

Our change to apply additive color over the resolved transparency color offered us a new optimization opportunity.

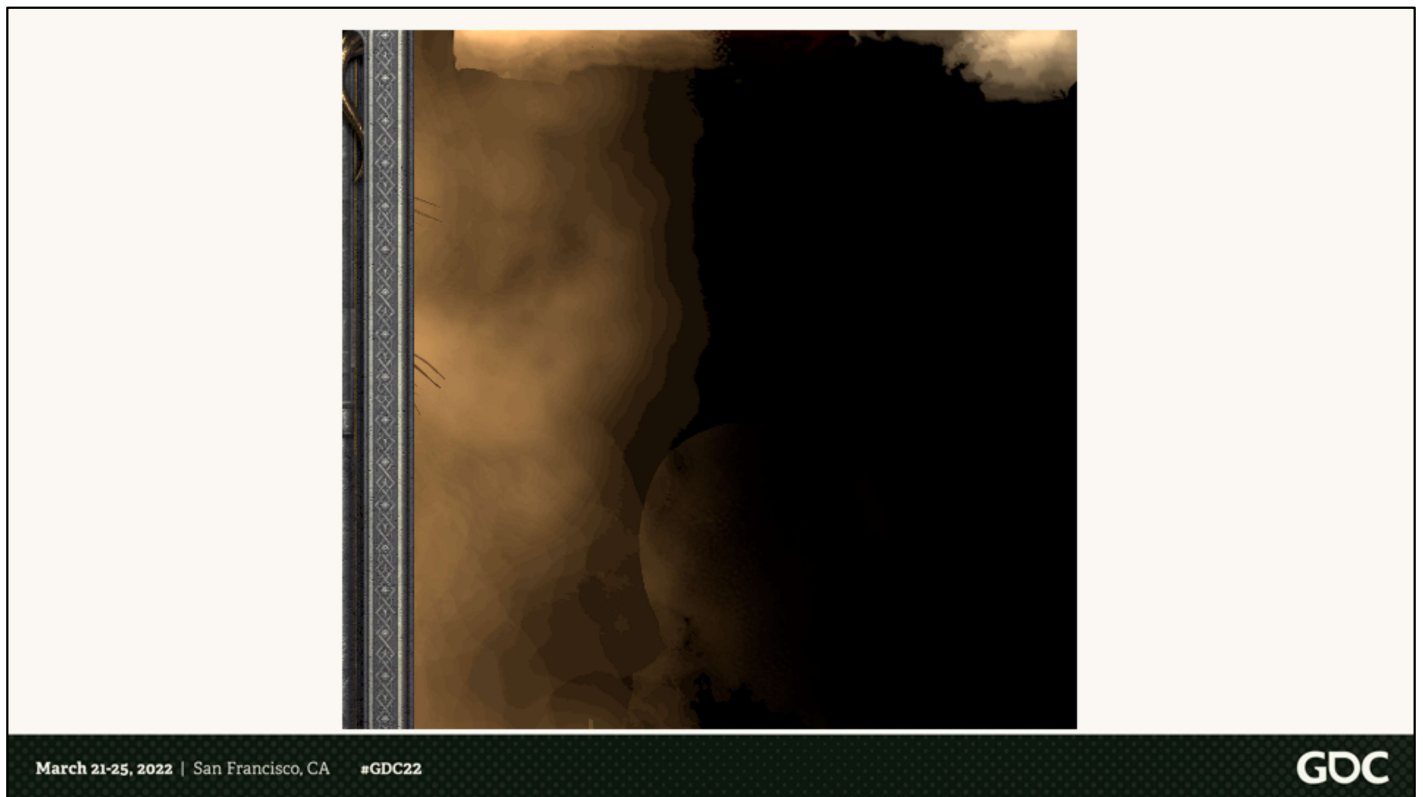
The idea is that we can move additive effect application to an entirely different render pass. The cost of this would be a new render target – we'd be trading memory for performance here, but this was worth it for us. We can then use that change to reduce the amount of information required to resolve alpha blended effects to just transparent color, weight, and alpha.



So this is our new render target layout. Still FP16 for the weighted transparent color and alpha, and the second target is an 8-bit target that we called the revealage buffer which effectively describes the relative visibility of an effect.

```
float4 color = g_color.Sample(g_sampler, In.uv);  
float4 accumulation = g_accumulation.Sample(g_sampler, In.uv);  
if (accumulation.a > 0)  
{  
    accumulation.rgb /= accumulation.a;  
    color.rgb = lerp(accumulation.rgb, color.rgb, revealage.a);  
}  
  
return color;
```

In our resolve step we take the accumulation value and treat it just like our transparent value from before, and use the revealage value to blend between it and the scene color. Additive effects are then rendered over top in an entirely different render pass.



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

This brought the bandwidth down to 72bpp, however we had to bump revealage up to r16 in order to have sufficient bit depth for HDR

Accumulation – FP16			
Blend: Add Src: One Dst: One			
Transparent	Transparent	Transparent	Alpha
Revealage – R16			
Blend: Add Src: One Dst: One			

Leaving us at 80bpp.



# OIT Retrospective

- Reduction of bit depth was a win.
  - Saved 15-20% GPU time spent on transparents in our effects stress test.
- It can work
  - But be prepared to be agile and undergo several revisions before you settle on something your effects team is comfortable with.

The reduction in output bit depth was a huge win for us, saving up to 20% GPU time spent on transparent rendering in our stress test for effects – this was measured on a GTX 1080, so those ballooned even further on consoles.

The takeaway for us after implementing OIT is that it can work great, but recognize that you'll have to go through several revisions and collaborate closely with effects artists to get the right algorithm for the game you're making.

# Hair Rendering

- *Hair and Fur Rendering in 'Diablo II: Resurrected'*
  - Ace Stapp
  - Friday, March 25, 1:30-2:30 PM
  - Room 202, South Hall
- Go check it out!

Now, hair rendering. I'm actually not going to go into any detail here, because there's an entire talk about it by my colleague Ace this Friday at 1:30 in 202. Go check it out if you're interested.

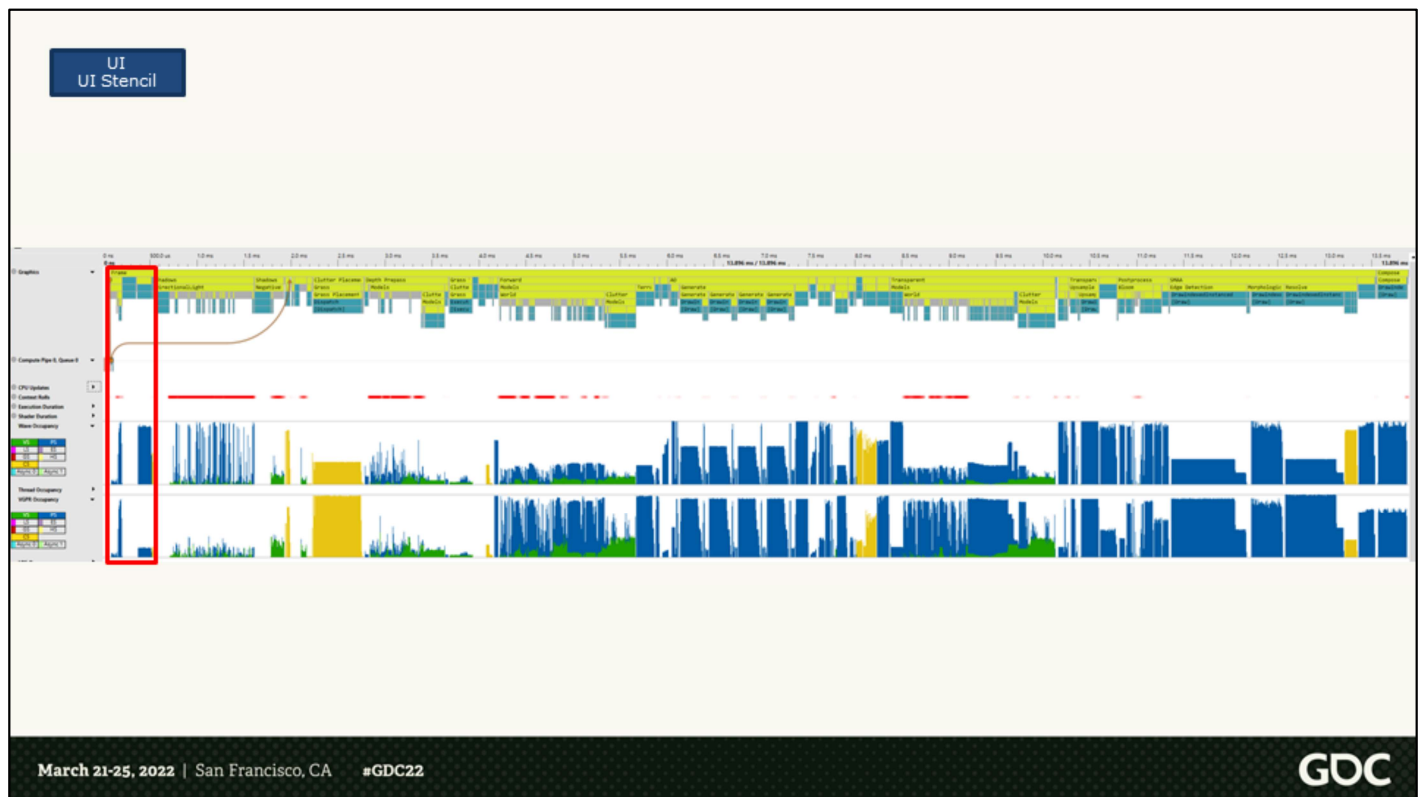
# Agenda

- Intro
- Overview of the renderer
- Deep dive into select techniques
- A frame on the GPU
- A frame on the CPU
- Streaming performance
- LODs
- Takeaways

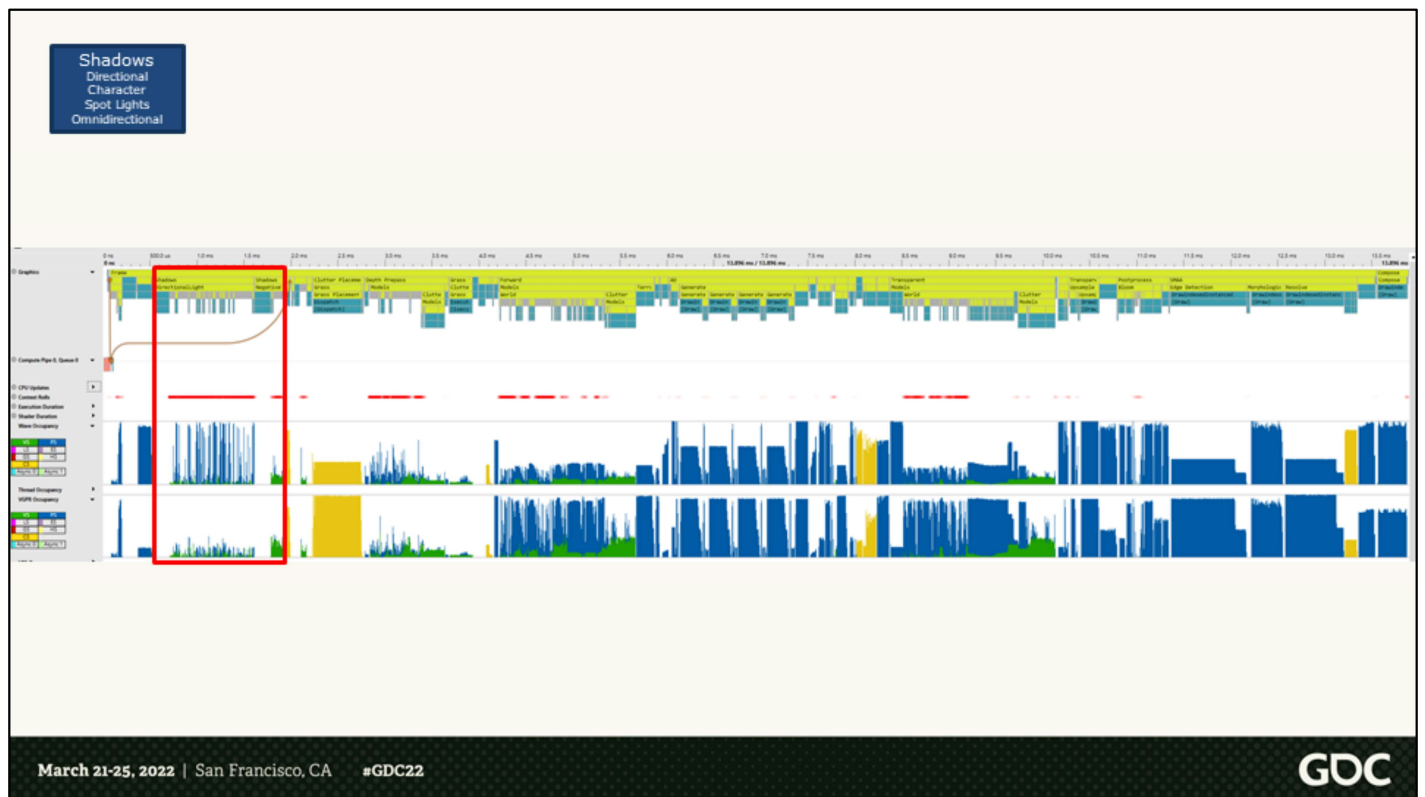
Quick stock of where we are.



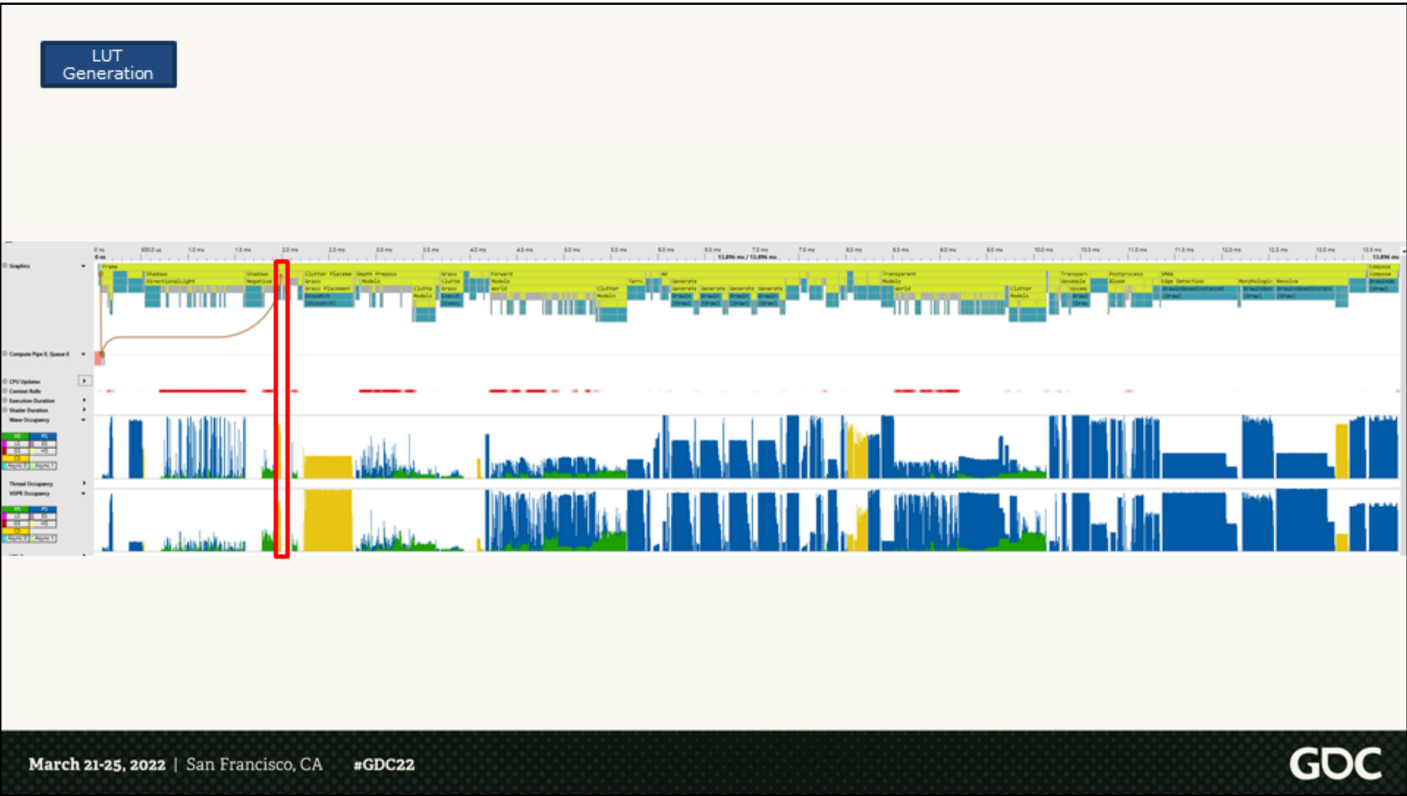
Ok, so what does all this look like on the hardware? Here's a capture of a frame on an Xbox Series X.



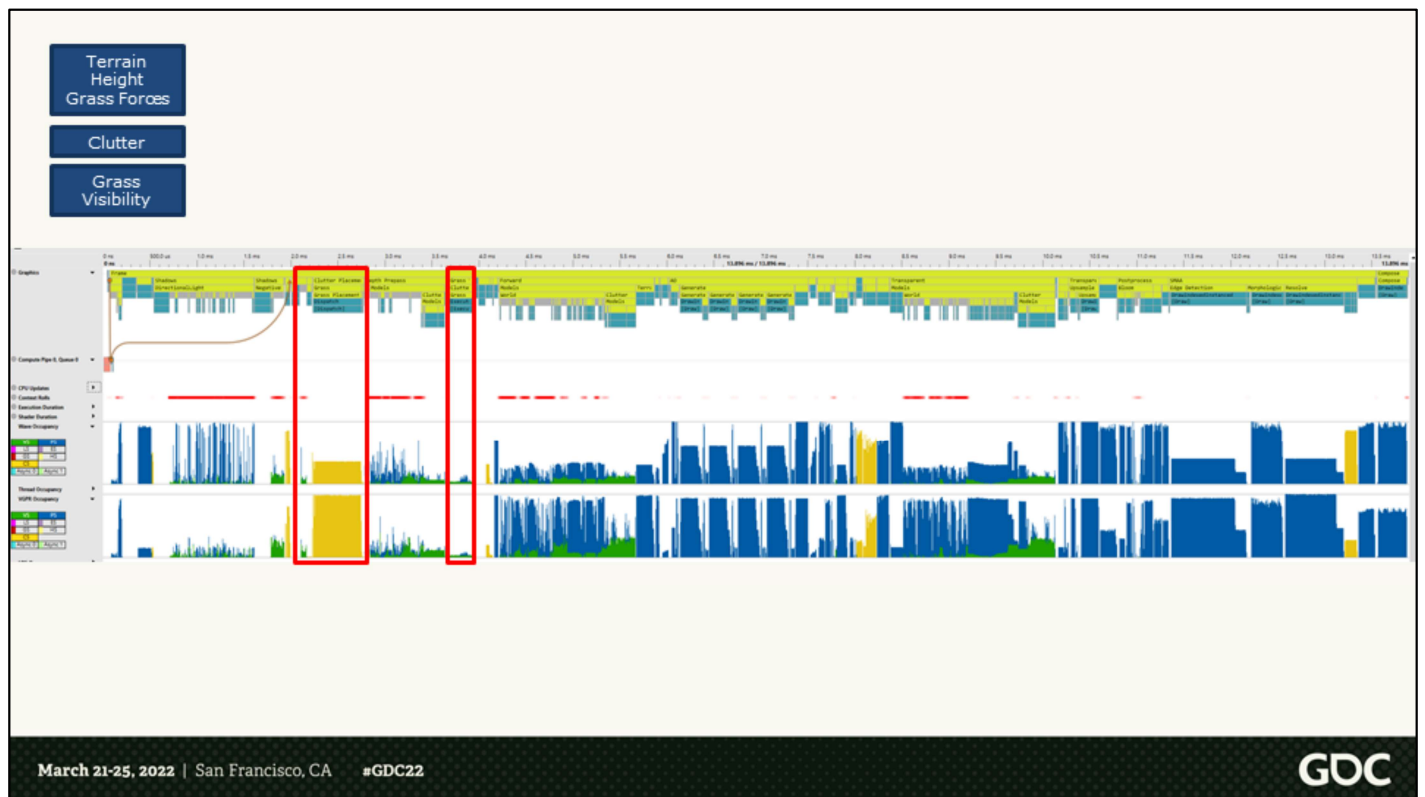
We start with UI rendering.



Followed by shadows.

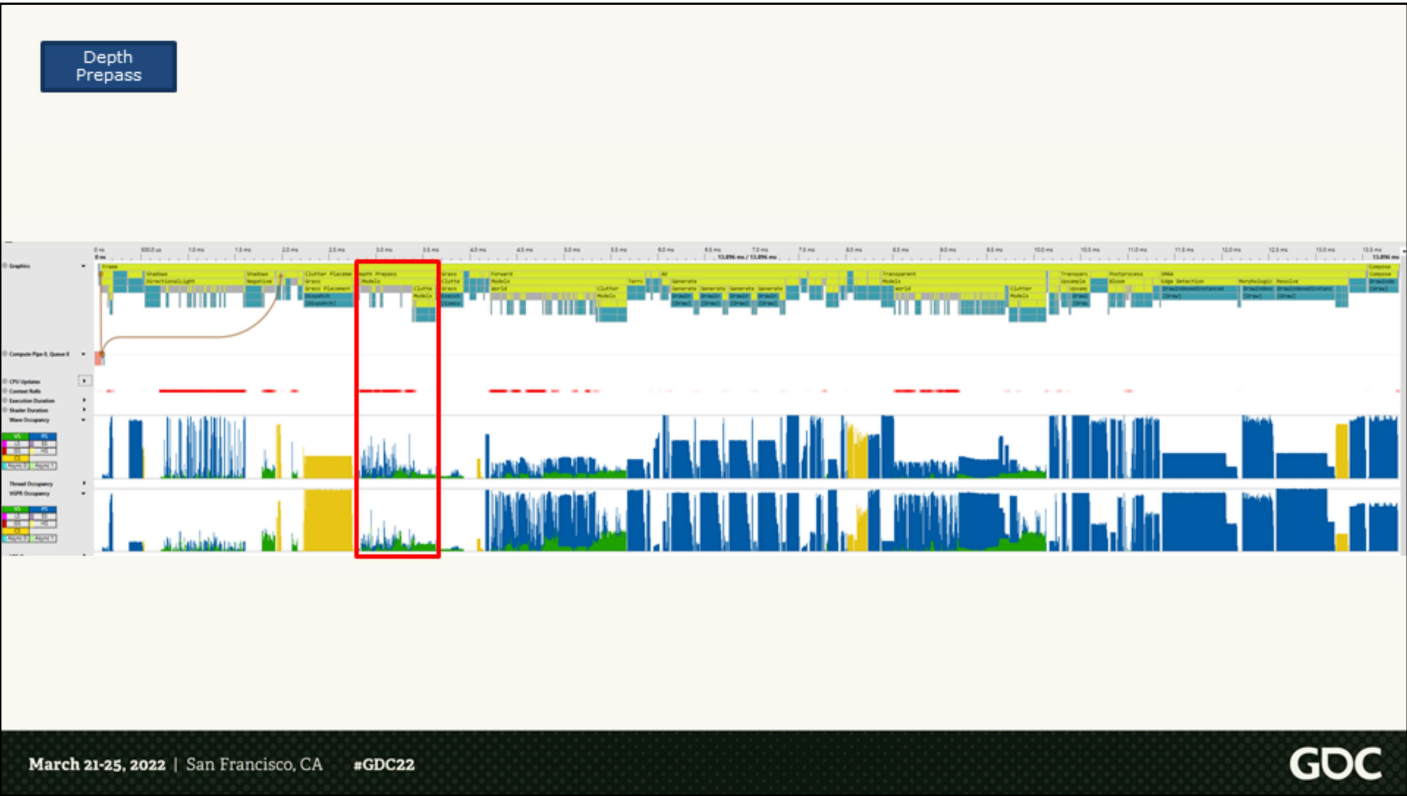


LUT generation



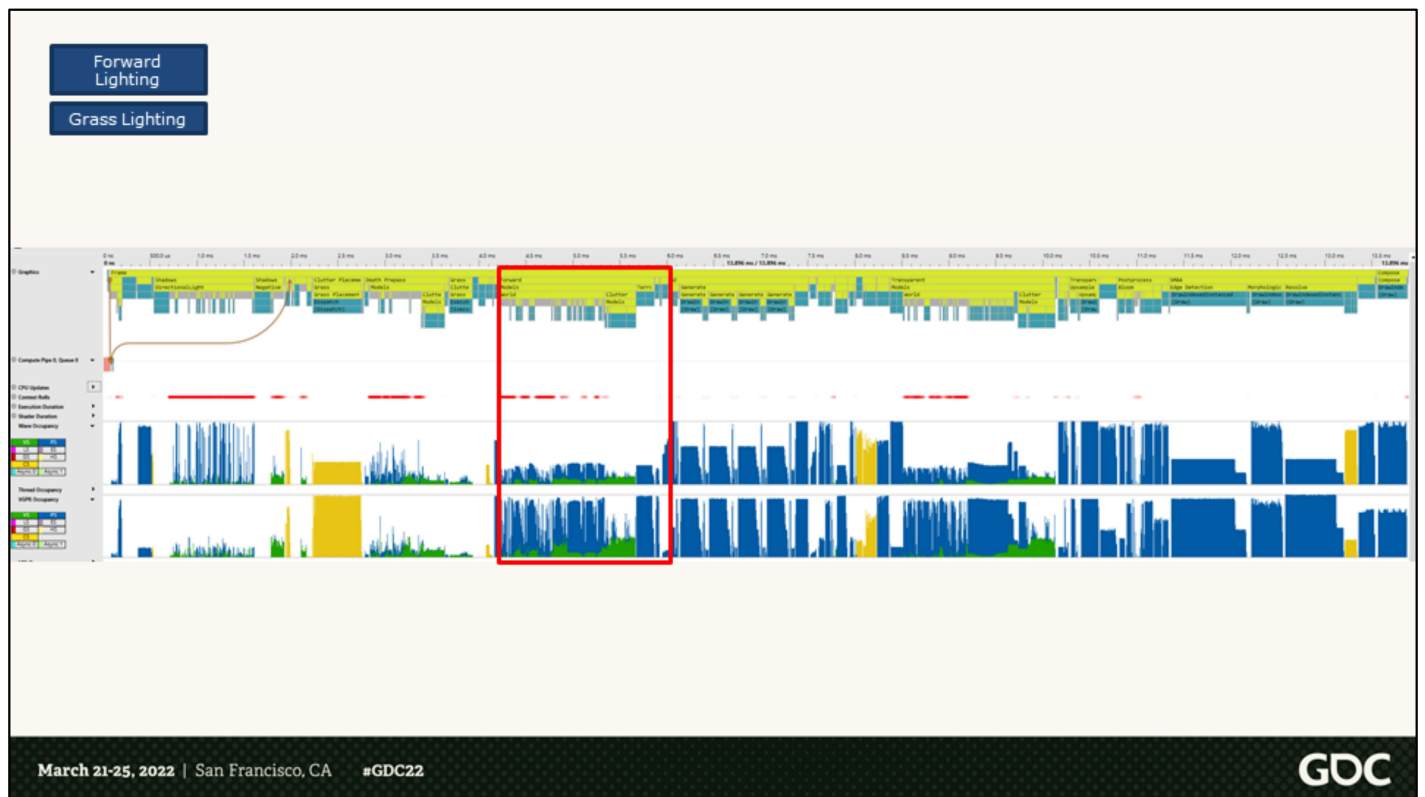
Terrain height map and grass and clutter generation.





Depth prepass

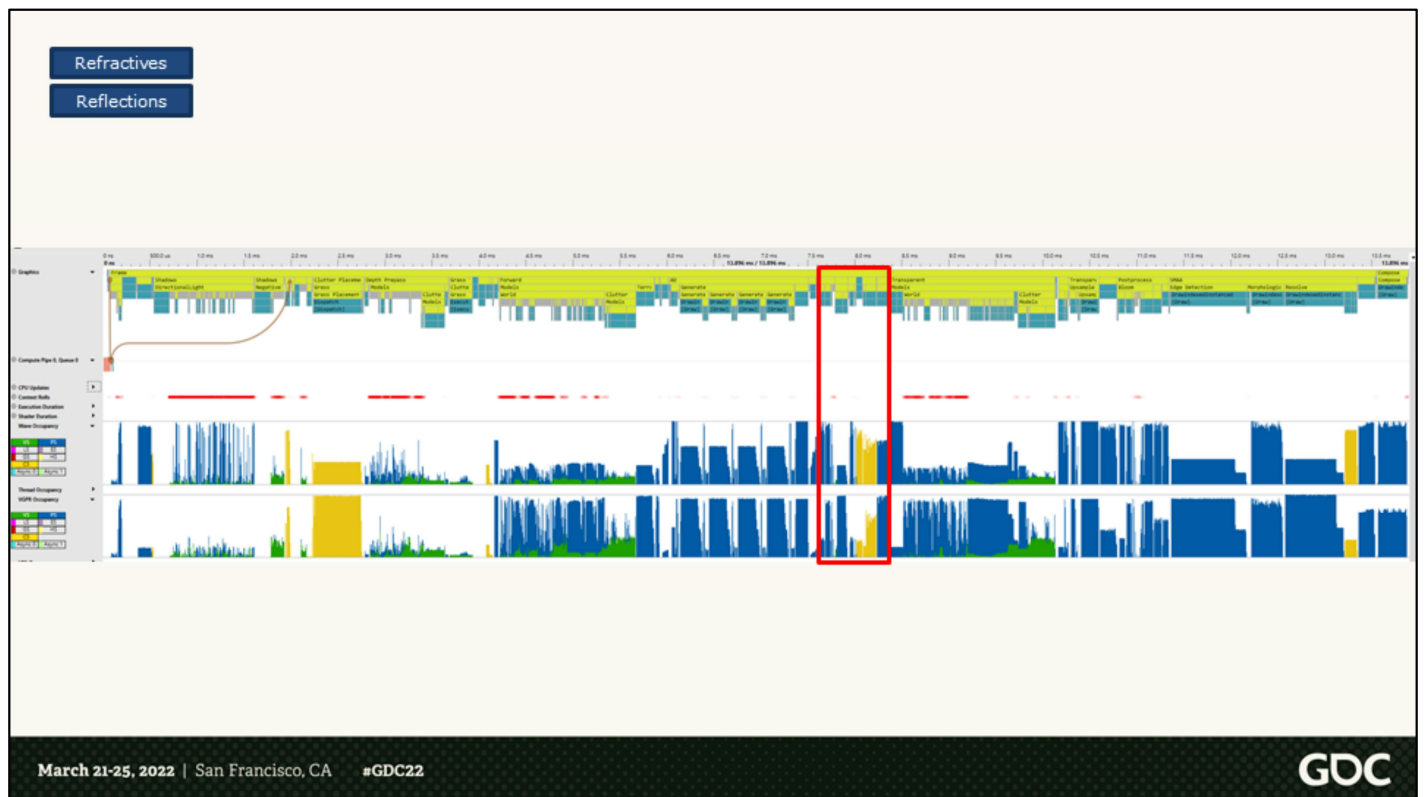




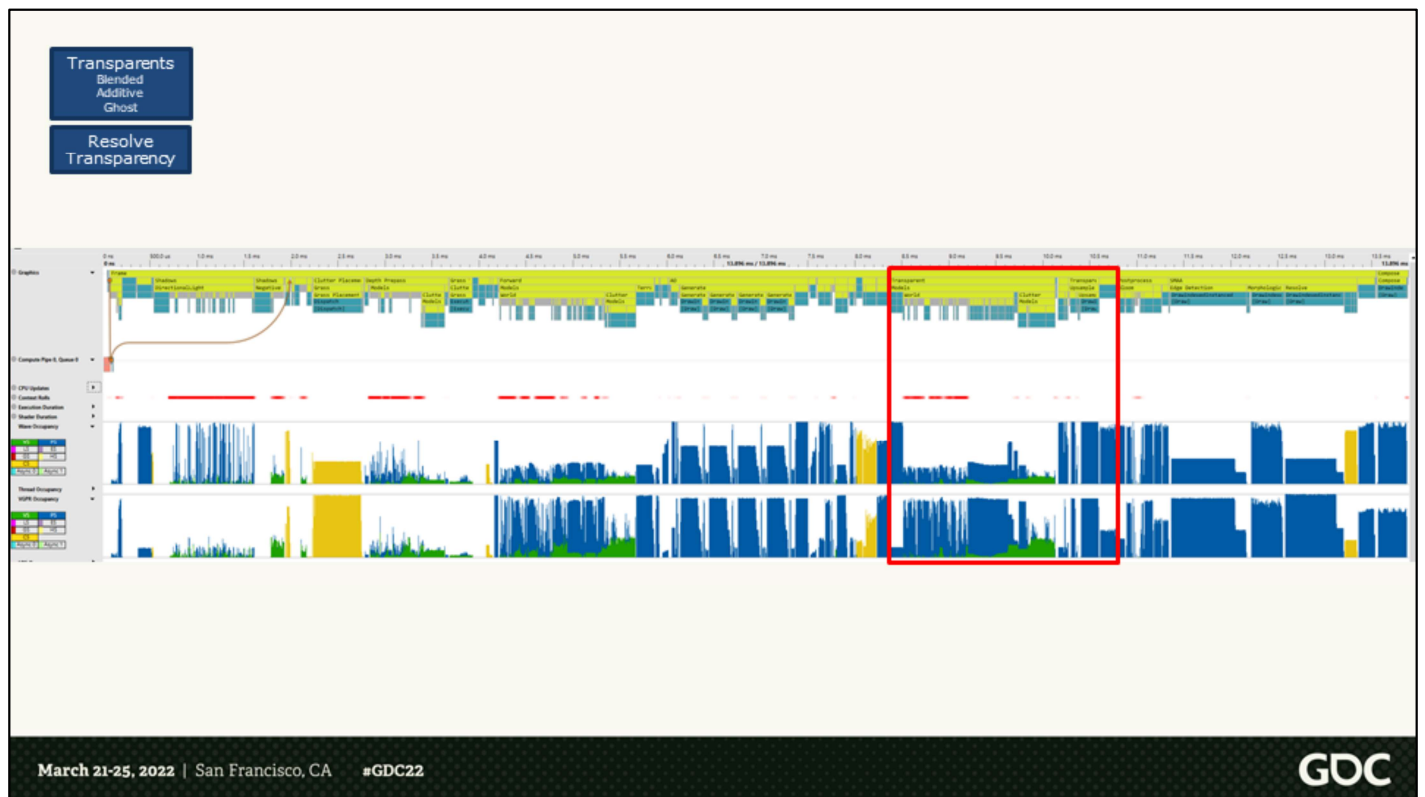
Then the main event, forward lighting and deferred grass lighting.



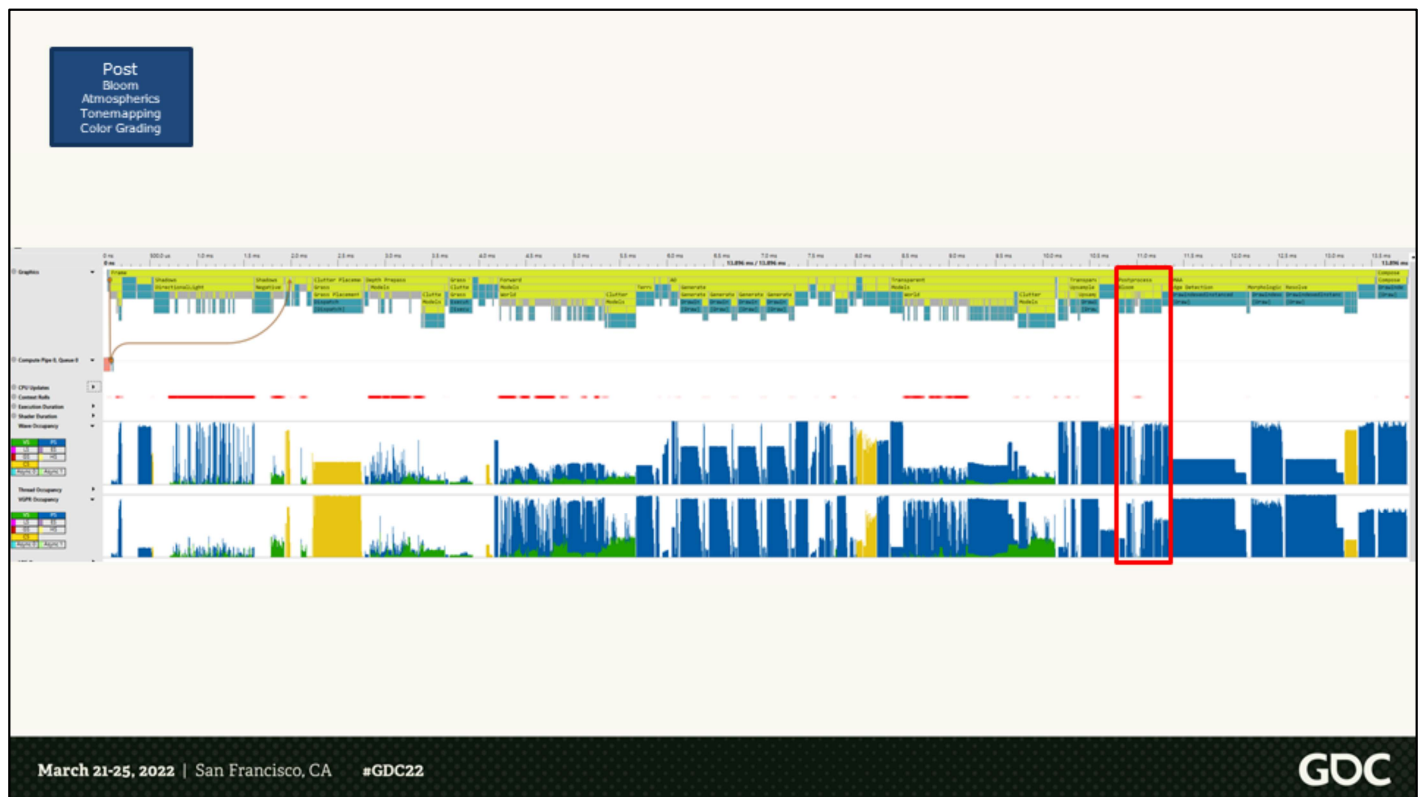
SSAO



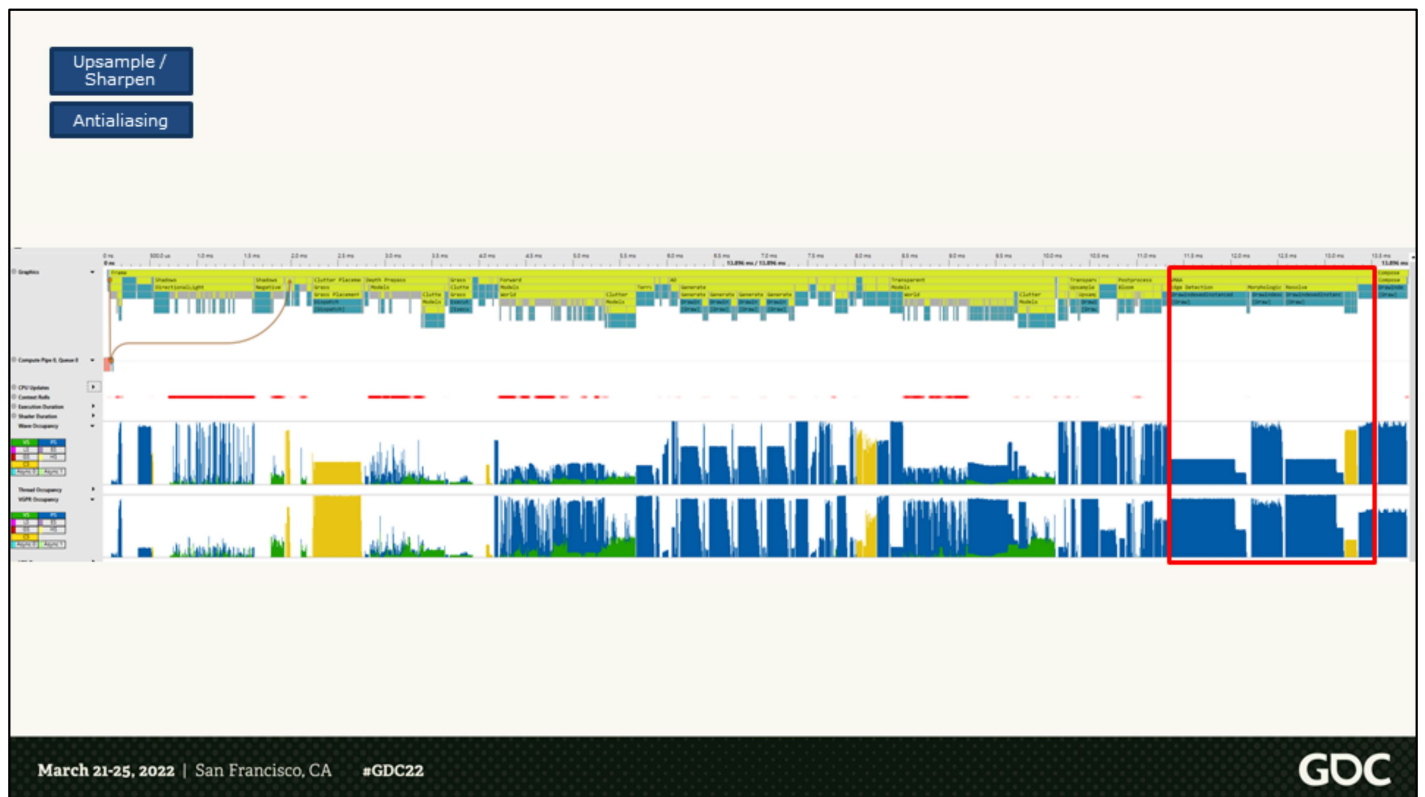
Refractive materials and screenspace reflections



Transparent rendering and the OIT resolve step



Post processing like bloom, fog, tonemapping, and color grading

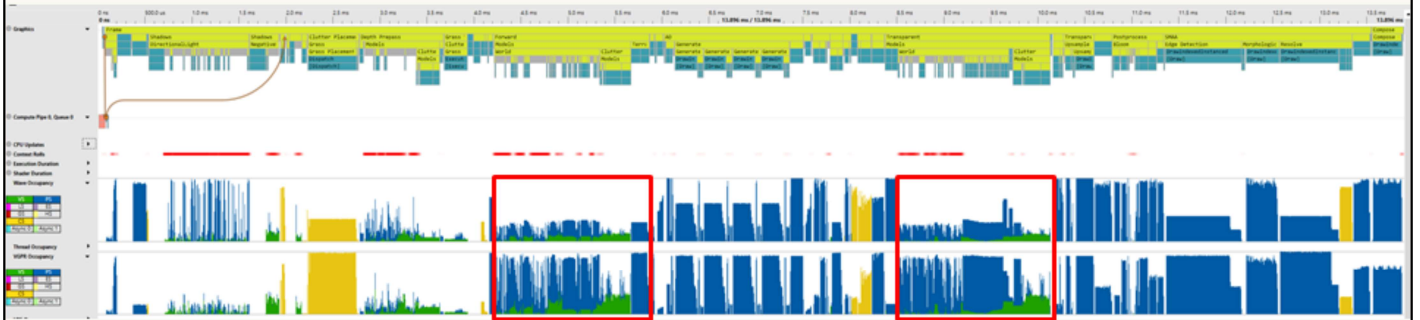


Morphological antialiasing and upsampling and sharpening





# Occupancy Woes



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

One of the major tradeoffs to a forward renderer which we fought throughout the project was wave occupancy. With the entire lighting model built into one shader, VGPR usage is very high. We can see here that in the forward pass achieving high occupancy is difficult because of the high VGPR usage of the pixel shaders. Later in the frame, our transparency shaders suffer from a similar problem both because some of them are lit, and because of the high complexity of our effects shaders.

# Occupancy Woes

- Constant battle to keep VGPR usage down.
  - Some permutations were once at 120+ VGPRs (measured in Razor GPU for PS4)
- Any change affects VGPR usage
  - Therefore, any lighting change needed to include an audit.
  - Any new request needed to be balanced with a cut.
- Just changing the order of lights could change usage.
- Advantage: All shaders are handwritten.

We had a constant battle throughout the project to try to keep VGPR usage down while also responding to changes requested by lighting that would affect the feature set and intricacies of the lighting routine. Any change to the order of application of lights needed to be audited for its impact on VGPR usage, and after a certain point any addition needed to be balanced with a cut. We found in some cases that simply changing the order in which we calculated contributions from different lights or shadow terms could have an impact on VGPR usage. [measured in Razor GPU for PS4]

But we did have one advantage: all our shaders were entirely hand-authored. We made a conscious decision not to implement a shader graph system. This meant we had a little more freedom when it came to hand optimization. Let's look at an example that highlights all these points.

# Global Attenuation

```
globalAttenuation = 0

For each point light:
    distanceAtt, intensity = EvaluateLight()
    globalAttenuation += distanceAtt * intensity

For each spot light:
    distanceAtt, intensity = EvaluateLight()
    globalAttenuation += distanceAtt * intensity

For each directional light:
    distanceAtt, intensity = EvaluateLight()
    globalAttenuation += distanceAtt * intensity

Ambient = EvaluateIBL() * globalAttenuation
```

The global attenuation system that we included to suppress ambient light and achieve unnaturally pitch-dark areas of play, was initially calculated inline in the pixel shader. Pseudo-code would have looked like this – for each light of each light type, evaluate the contribution of the light and accumulate contributions into a variable. At the end of the shader, calculate ambient light contributions (IBL), and multiply the contribution by the accumulated amount of direct light (clamped to 1).

# Global Attenuation

```
globalAttenuation = 0

For each point light:
    distanceAtt, intensity = EvaluateLight()
    globalAttenuation += distanceAtt * intensity

For each spot light:
    distanceAtt, intensity = EvaluateLight()
    globalAttenuation += distanceAtt * intensity

For each directional light:
    distanceAtt, intensity = EvaluateLight()
    globalAttenuation += distanceAtt * intensity

Ambient = EvaluateIBL() * globalAttenuation
```

The problem with this is it means that a single variable is stored for almost the full duration of the shader, which means it will hold onto a VGPR the whole time. In theory, freeing this would only free up a single VGPR, but in practice it can actually lead to much more opportunity for the optimizer and open up other opportunities for savings.

# Global Attenuation

```
For each point light:  
    distanceAtt, intensity = EvaluateLight()
```

```
For each spot light:  
    distanceAtt, intensity = EvaluateLight()
```

```
For each directional light:  
    distanceAtt, intensity = EvaluateLight()
```

```
Ambient = EvaluateIBL() * textureLookup()
```

We separated out global attenuation into a separate pass, and rearranged terms so that each light loop was self-contained and could recycle VGPRs. This brought our VGPR usage down by up to 20 registers in some of the most complex shaders.

# VGPR Lessons

- Minimize variables with long lifetimes.
- Avoid storing repeated information.
- Favor independent blocks of work.
  - That don't share information.
- Experiment with rearranging terms.
  - Even simply reordering can affect VGPR usage.

Some of the lessons learned from these exercises – minimize or avoid variables that have long lifetimes. Explicitly avoid storing the same information in more than one variable. Favor contained blocks of work that don't share information. Experiment with rearranging terms because simply reordering can have an impact on VGPR usage. I don't think that this is because shader compilers aren't smart; on the contrary, it's because they are so complex.

[illegible]

A large reason for this is the lack of maturity of the engine. I talked about in the beginning how we need to make tradeoffs of time and effort in order to reach the finish line, and this is a great example of that. While these opportunities for optimization exist, being in budget was top priority and once there, our priorities would become quality or other art concerns. In a more mature engine, these optimizations are tackled over the course of several iterations of the engine, potentially over multiple titles.

Speaking of asynchronous compute, not shown in this frame is the largest job for which we used asynchronous compute – terrain generation. Terrain is not computed every frame, but rather cached in a large virtual texture and generated at intervals as you traverse through the world. In order to avoid spikes in framerate,



we attempt to hide the cost of computing new terrain tiles in asynchronous compute.



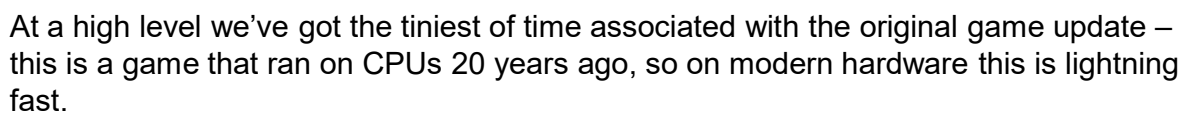
Here's a frame where the entire visible terrain is being regenerated, for demonstration purposes. We can see how even when regenerating such a large area of terrain, the cost is absorbed well by filling gaps at the start of the frame.

# CPU Goals

- Take advantage of multi-threaded command list generation.
- Producer/consumer paradigm with original engine.

Shifting gears over to the CPU. We had two goals with the CPU design of our renderer. The first was to take advantage of multi-threaded command list generation made possible by modern APIs. And the second was a producer/consumer paradigm with the original engine, which would offer up the data on what we needed to render and that data would be consumed by the rendering engine to put the 3D assets in their respective places in the world.

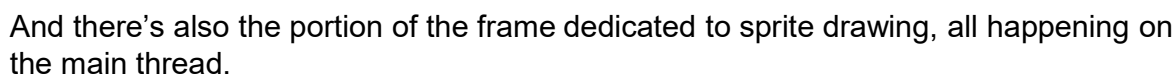


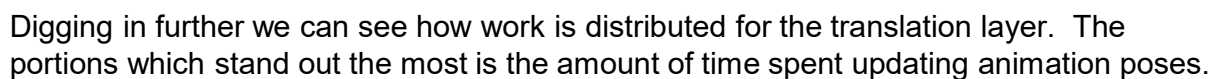


At a high level we've got the tiniest of time associated with the original game update – this is a game that ran on CPUs 20 years ago, so on modern hardware this is lightning fast.



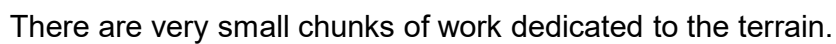
Right after that we find a translation layer acting on any new information available from the game simulation. I'll elaborate on that a bit more in a second.

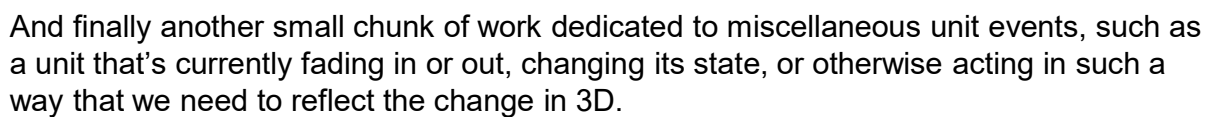




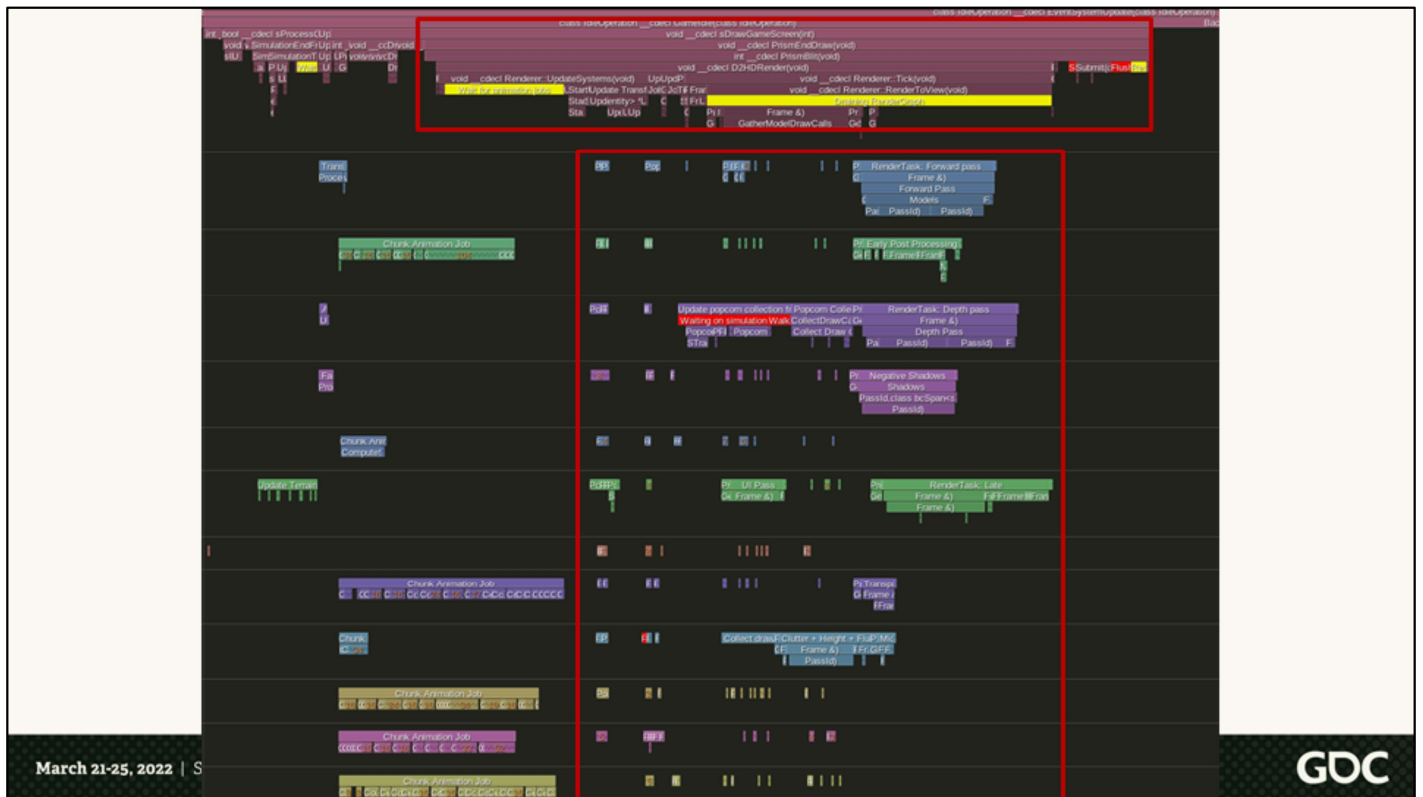
Digging in further we can see how work is distributed for the translation layer. The portions which stand out the most is the amount of time spent updating animation poses.



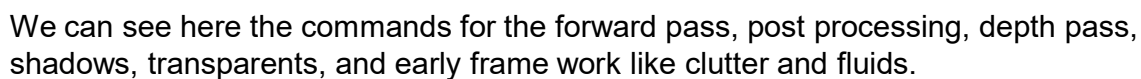


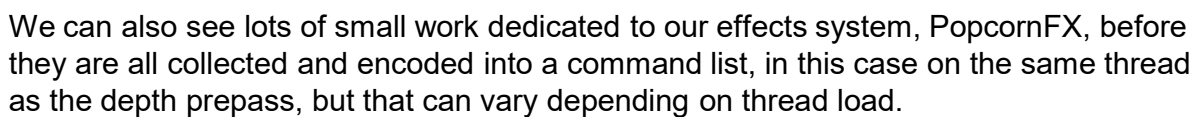


And finally another small chunk of work dedicated to miscellaneous unit events, such as a unit that's currently fading in or out, changing its state, or otherwise acting in such a way that we need to reflect the change in 3D.

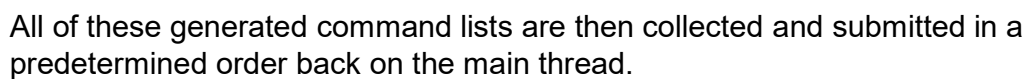


Pertaining to rendering, we can see how work is generated across multiple threads, coordinated by the latter portion of the main thread – we didn't have a separate main thread/render thread model.





We can also see lots of small work dedicated to our effects system, PopcornFX, before they are all collected and encoded into a command list, in this case on the same thread as the depth prepass, but that can vary depending on thread load.



GDC

# Architecture

- Initially, single threaded.
  - It's a 20 year old game, how much time could the original logic really take?
  - We didn't plan for the simulation time translating from 2D space to 3D space.
- But... what actually happens in Diablo II?

Perhaps something very notable about the upbringing of the renderer is that we did not start by constructing it with multithreaded command list generation. We did know that we had the option to leverage this capability of DirectX 12, but with the goal of standing something up fast, opted to keep render work initially single-threaded and even run it on the main thread rather than a render thread. One of the justifications, at the time, felt sound - the main thread of Diablo II, a 20 year old game, is extraordinarily lightweight and was unlikely to interfere with the time spent submitting render work.

What we didn't count on was the time taken by our translation layer which lived between the original game engine and our new renderer. We created this shim as a means to manage a correspondence between all things in the 2D game -- known as units -- to all resources used for them in 3D. So for example you might have a monster, that's a unit, and so this monster when spawned by the game would pass through the translation system and we'd load and create a model and animation set to represent this monster in 3D. Similarly for objects, items, missiles; everything.

Now, with the amount of stuff that can be happening at any given time in Diablo II...

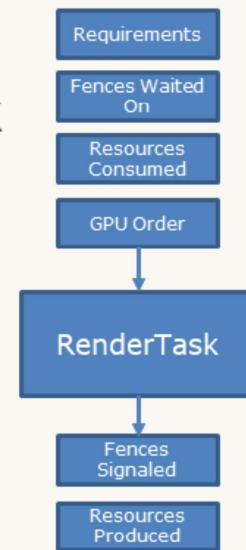


... managing this translation layer becomes pretty taxing on the CPU, leaving us less room to perform a single-threaded pass for all render commands. So we broke the work up into a job-based system



# Architecture

- Core unit of work is RenderTask
  - Producer/consumer
- RenderTasks are added to a RenderGraph.
- RenderGraph is compiled to create jobs.



Our architecture is not too dissimilar from others out there. The core unit of work is a RenderTask. It operates on a producer/consumer model – render tasks take in different requirements, like what view space constants they need; they can wait on fences, specify what resources they consume, and even have an order they're submitted to the GPU. Tasks will output any fences they signal, as well as any resources they produce that may be consumed by other render tasks.

All render tasks are added to a complete render graph which represents the work generation for the entire frame, and then that graph is compiled at which time it can be checked for dependency issues or conflicts, and generate the actual jobs that will run to generate command lists.

# Lessons

- A job system is something you can take for granted.
  - We had to integrate one first!
- Retrofitting a renderer with a multithreaded job system is hard.
  - Like very precise surgery.
  - There are going to be synchronization issues.
- Start with multithreaded design first.
  - It's worth the upfront cost.

We learned a few things retrofitting a single threaded renderer with a job architecture. The first was that it's easy to take a job system for granted. When we started this, we didn't have one so we had to invest in integrating one.

Second, adding multithreading to any application is complicated. Something is going to go wrong, and it's going to be a huge disruption. This was very difficult for us because there wasn't very much room for error – the entire team needed to keep working which put a lot of pressure on getting something very complex right the first time.

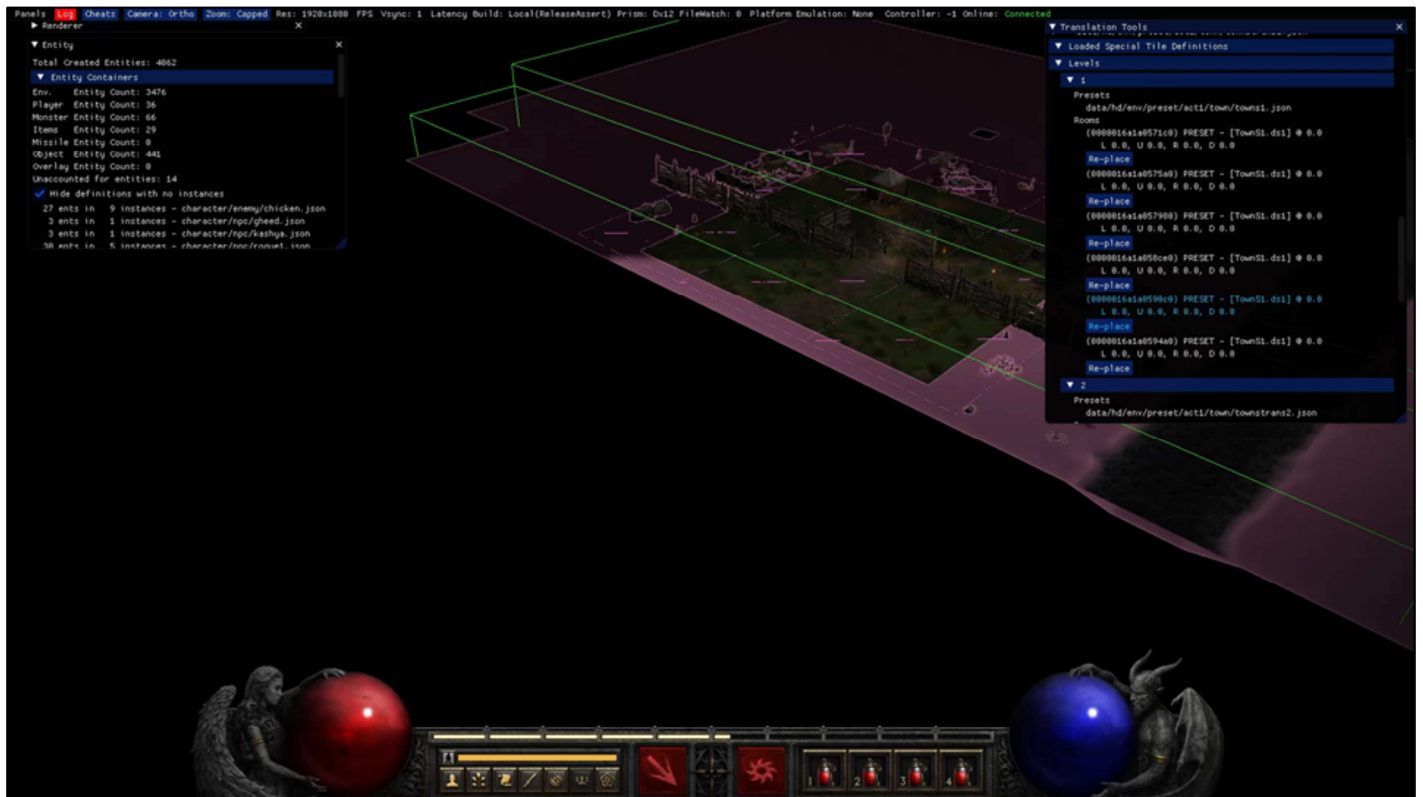
Ultimately, it's very worth it to start with the multithreaded design first to avoid this disruption. Even if your upfront cost seems high and it takes longer to get a point where your team can start working at full speed, it saves the even greater cost of disruption late in the project.

# Streaming

- Diablo is heavily procedural
- Everything must be loaded asynchronously
- Retrofitted asynchronous loads

Some quick notes about streaming in the game. Diablo is a heavily procedural game with large outdoor spaces. Even the original did not load all sprites for an act at once and streamed them into a cache as needed.

For us, everything needed to be loaded asynchronously in order to guarantee smooth gameplay and traversal across Sanctuary. However, we retrofitted these asynchronous loads after initially developing with blocking loads, since, up front in the absence of any existing streaming system, that was easier. However, just like retrofitting multithreaded command list generation, this was very difficult and very risky.



As you move through the world assets are streamed and instantiated around you not too far outside the view of the camera. There are several tiers of streaming range – the core of the tech is based off of the room architecture of the original game. The purple debug regions indicate the rooms adjacent to the room the player is currently in. The green boxes are the bounding boxes of entire presets, which are larger scale than rooms. You can think of a room as a logic unit and a preset as a variable chunk of the environment. There is a room activation radius around the player that will load the assets needed for that room if it falls within the activation radius – this radius is larger than the rooms immediately adjacent to the player.

Entities and models are instantiated on a different range. Here we can see this in action. In the upper left are some statistics about how many entities there are, and to the right is a list of the active rooms and presets. Generally the goal of the activation radii was to ensure a consistent stream of assets over sudden spikes of file IO. However, when approaching a town there still can be a lot of streaming requests to absorb, as the towns are the largest presets in the game.

# Streaming Textures

- Bottleneck becomes file IO
- Texture manifest
  - All properties required to create footprint
- Thumbnails as stand-in

Now, over the course of the project we worked to get texture loading entirely blocked by file IO, particularly by relying on direct copies to video memory on consoles. In order to prevent file IO from blocking the game, we created a texture manifest of all textures which contained the properties required to create a footprint for the texture – things like size, format, mip count. As a load request would come in for a texture, we could look up the texture in the manifest and create the object with empty data, and simultaneously kick off the asynchronous load of the actual texture. Later when the load finished, we would copy the new texture contents to video memory.

In a crude implementation of mip streaming, we would also create thumbnails for each texture that were small enough to be directly embedded in the manifest, once again to entirely avoid file IO being a bottleneck to texture loading.

# LODs

- Art pushed *very* high fidelity
- New renderer – don't know perf metrics
- Have to start somewhere
  - Previous titles, similar platforms
  - 2.4M for characters
  - 600k-800k for environments

One more thing I think is worth talking about is our experience with LoDs. This was a big challenge for us for a few reasons. First, our art team was pushing a very high fidelity level for this game. Yet, at the same time, we've got a brand new engine that's being developed while assets are being made. The reality is, we don't know what our performance characteristics are going to be. So the best thing we can do is guess in order to give some guidelines for the team to work with - not giving them any guidelines would be a huge mistake - but the guess comes with a big asterisk that it's subject to change later in the project. The challenge here is that if you are conservative with your guess, your art team may not be too comfortable with the answer, and you don't yet have the evidence to back it up.

But there are some things that we can use to make an educated guess. We have prior games that went through the same exercise and while the budgets may not be exactly the same, they can give us a reasonable starting point. As an example, for our highest SKU we set an initial budget of 2.4M polygons per scene for characters, and 6-800k for environments. These were based off of a previous title we released on the Xbox One and PS4, and were extrapolated out to newer PC hardware. I didn't say that backwards, this is very lopsided from what you'd expect in a budget, but the reasoning was that there could potentially be hundreds of characters on screen at one time, and the environments always have a limited area in view and are relatively simplistic.

# LODs

- Categorize by density
  - Bosses – one at a time
  - Standard – a few at once
  - Mob – many visible simultaneously
- Categorize by load
  - Lite – 10 instances
  - Medium – 30 instances
  - Worst case – 100 instances

Further into development, as the engine matured, we were able to do something more scientific. For characters, we classified them into categories: bosses, which you only ever seen one of at any given time; standard, which may appear in small numbers; and mob characters, which can have many instances on screen at the same time. We also categorized by load: combat scenarios like light, medium, and worst case, where 10, 30, or 100 or a particular monster would appear on screen at once.

We took a few monsters to represent each category and measured the impact that different polygon counts had on both a GTX 1080 and a PS4.



# LODs – PS4

Combat	LoD	Triangles	Shadow (ms)	Forward (ms)	Gained (ms)
Lite	0	26376	0.949	1.872	--
	1	5995	0.771	1.197	0.853
	2	2997	0.736	1.002	0.230
	3	1544	0.721	0.869	0.148
Medium	0	26376	1.494	4.945	--
	1	5995	0.923	2.921	2.295
	2	2997	0.842	2.394	0.608
	3	1544	0.804	2.026	0.406
Worst Case	0	26376	3.382	15.729	--
	1	5995	1.463	9.134	8.514
	2	2997	1.220	7.312	2.066
	3	1544	1.100	5.941	1.490

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Here's an example of that data. On the PS4 we can see that we can gain two and a half milliseconds between shadows and forward lighting in the medium load case by reducing this monster by 75%, and a whopping 8.5 ms in the worst case. But we're also missing some other potential data points between lods 0 and 1.



# LODs – PS4

Combat	LoD	Triangles	Shadow (ms)	Forward (ms)	Gained (ms)
Lite	0	26376	0.949	1.872	--
	1	24996	0.932	1.850	0.035
	2	19996	0.896	1.697	0.190
	3	14995	0.845	1.531	0.217
Medium	0	26376	1.494	4.945	--
	1	24996	1.446	4.863	0.116
	2	19996	1.303	4.419	0.588
	3	14995	1.154	3.979	0.589
Worst Case	0	26376	3.382	15.729	--
	1	24996	3.211	15.513	0.438
	2	19996	2.748	13.952	2.024
	3	14995	2.251	12.444	2.005

Looking at those we can see that savings aren't necessarily linear with reduced polygons, at least in the engine we were building. There was a point where we'd see diminishing returns and the effort required to reduce geometry further didn't make sense given the performance gain we'd realize. So this told us that we could raise the LoDs on some platforms, and we picked values that represented the best tradeoff between decimation effort and returned performance. And this was an important reduction of the optimization burden on the art team because you probably raised an eyebrow when I said decimate by 75%.

# LOD Tools

- 90+% decimation
- Started with proprietary tool
- Added Simplygon as another option
  - Integration with Maya
  - Hand-fixup an option

In fact, because of our range of SKUs there were some cases where we would have to decimate over 90% of the original LOD, and this is a huge problem because very rarely is this just going to "work" well. In practice, we ended up implementing two choices for our artists to decimate: we started with a proprietary decimator because - following a theme here - we could leverage internal expertise. But later we also added a Simplygon pipeline and created an iteration loop in maya that would allow an artist to decimate and then see the results immediately in Maya in order to fix up any unwanted artifacts by hand. They could export these adjusted LODs as overrides to the automated ones.

# LODs – Lessons

- Make every effort to gather performance metrics early
- Focus on the big picture
  - Does it run within time allotted?
  - Where is the detail going?

One thing we can take away from this experience is the importance of establishing your performance metrics early. But the real lesson coming out of our journey with LoDs is that it's important to avoid getting lost in numbers and instead focus on the bigger picture. We ended up focused too often on polygon counts, when that wasn't ultimately what mattered. If it runs in the time it needs to, you've accomplished what you need for that scene, even if on paper the geometric density of the scene sounds too high. And on the other hand, it's important to ask where the detail is going. And to illustrate that point...

# LODs in Reality



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Here are four LODs of the barbarian viewed from the game camera. How easy is it to tell the difference between them? Not very. There are some artifacts that stand out if you look very close, but ultimately with how small things appear in the game, we had opportunity to get away with a lot of reduction.

# What could we have done better?

- Scalability didn't come in where we wanted.
  - Lacking some meaningful tradeoffs in options.
  - Great, but also not-so-great.
- Lead with a multithreaded work generation architecture.
  - Cost of disruption is high in an environment where disruption is highly impactful.
- Design for asynchronous operations.
  - Again, cost of disruption is high.

So, in the end, we shipped a game, we resurrected Diablo II with modern visuals, it was a job well done, but what could we have done better?

The first thing that comes to mind is that our scalability, while impressive across a range of hardware like top-of-the-line PC down to the Nintendo Switch, did not actually represent significant tradeoffs between performance and quality. I talked in the beginning about how scalability was one of the key philosophies throughout development – every technique included needed to be scalable or be turned off. We almost did a little too good here, and it was a double-edged sword – on the one hand even our lower quality visuals look decent next to our highest quality visuals, which is good news for those on less powerful hardware; but on the other hand, without a significant enough quality increase with certain options, one would be inclined to ask what the benefit of sacrificing framerate is for a minimal visual quality gain.

The lesson here is to invest time and effort into ensuring that your quality options are meaningful to your players. This can be a delicate conversation with the art team. But each option must feel like an important tradeoff between performance and visuals.

And of course the second improvement would be to lead with a multithreaded work architecture, and asynchronous streaming too. It's one thing to make a significant multithreading overhaul as a key technical effort during a project when you already have an engine that people can work in. It's another thing entirely to do it while you're still building that engine and the technology that your art teams are relying on. Risk is significantly higher, as is the cost of disruption.

# Lessons

- Don't underestimate the breadth and value of supporting technology.
- Fit technology to context.
  - Forward+ for rapid development.
  - No baking – the game content is highly procedural.
  - Transparency – the game is very heavy on visual effects.
- Innovate in alignment with your goals and expertise.
- Look out for advantages or shortcuts.

And finally, the lessons we learned, which I hope are useful for all of you today.

First, if you're setting out to make a new renderer, do not, do not underestimate just how much value you get from existing supporting technology. It's easy to take this for granted and you only miss it when it's not there. When we first went down the road of multithreading we were hit with the realization that we didn't even have a job system, so we started several steps behind. Even if we can pull one off the shelf, that's more work that needs to be done to add that foundational technology. And as another example, we had to roll our own shader permutation building and loading scheme.

Second, fit the technology you're making to the context. We made our technology choices to favor rapid development and work in tandem with a highly procedural game, and service the salient points of gameplay.

Third, if you are choosing to innovate, make sure your innovations align with your project goals and your teams' expertise. Aligning these will help make your game better and save you time and effort.

And finally, always be on the lookout for advantages you have or shortcuts you could take. I can't overstate just how much work the fixed camera in Diablo II saved us overall.

# Special Thanks

- The D2R graphics team
  - Chad Layton
  - Ace Stapp
  - Joel Peters
  - Jon Lee
  - Gustavo Samour
  - Nish Sundharesan
  - Anushka Nair
- Iron Galaxy
  - Jeff Campen
  - Martin Holtkamp
  - Mike O'Connor
  - Alex Delesky
- The entire D2R team
- Julien Merceron
- Michael Bukowski




# There's more!

- *Resurrecting a Classic: Bringing 'Diablo II' Into the 3<sup>rd</sup> Dimension*
  - Kevin Todisco
  - Up next!
  - Right here!
- Stick around!

Also, if you're interested in the complete technical story of how we turned a 2-dimensional game into a 3-dimensional game – which, why wouldn't you be? – stick around to see me talk about that in this same room!

# Questions?

 @kevintodisco  
ktodisco@blizzard.com

We're hiring!  
<https://careers.blizzard.com/global/en>  
<https://careers.blizzard.com/global/en/albany>

# Stats

- Lines of code written for 3D layer: 135,224
  - 102k for D2Render, 20k for Translation, 5k for D2Entity, 7k for D2Glue
- Lines of shader code written: 31,132
- Total # shader files: 229
  - 141 HLSL and 88 HLSLI
- Total # HD textures: 29,990
- Total size of all source textures: 322.7 GB
- Total #Granny files: 29,708
- Total # VFX files: 1,851

# Stats – Rogue Encampment

- A look at render statistics for the Act 1 town
- Avg 3.35M prims at the start location, 2.6k draw calls
- Total asset counts and cache sizes are shown to the right

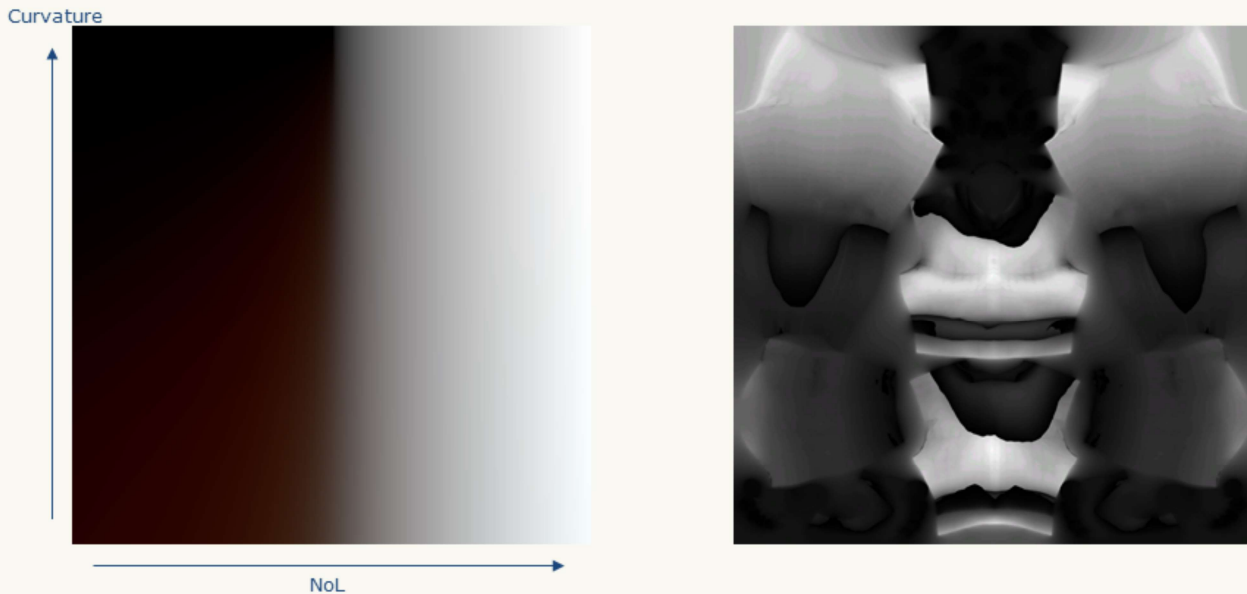
▼ Stats			
Memory Usage: 6458 / 12288 KB (52.6%)			
Draw Calls:			2618
Primitives:			3353325
Particles:			789
Popcorn Heap			
In Use			0 KB
Allocated			16448 KB
Mesh Heap			
In Use			988 MB
Allocated			1824 MB
Texture Heap			
Estimated			0 MB ( )
In Use			1823 MB
Allocated			1536 MB
Texture CPU Memory (kb): 28328 / 24576			
Job System (kb): 12 / 5120			
► Timers			
▼ Asset Caches			
	Refs	Cached	Size
Models:	4363	1497	2758
Skeletons:	341	196	512
Animations:	6169	139	512
Animation Timelines:	231	119	512
Variant Files:	64	43	128
State Machines:	247	154	384
VFX:	634	481	1824
▼ Instances			
	Current	High Water	
Model Instances:	1958	1958	
Skeleton Instances:	58	58	
Average mesh instances:	1.312821	1.312821	
▼ Material Stats			
	#Mat	#MatPass	#MaxPass
All Materials	5178	47238	11
▼ Descriptor Tables			
	Descrs	Tables	
IGlobal	24789	2294	
Biome	11	1	
Biome Grass	11	1	
Compute	288	42	
Fullscreen	331	32	
Material	23896	2173	
Popcorn Uber	188	45	
▼ Pipelines			
Render:			
Cached: 1629 / 4896 <a href="#">Clean</a>			
Compute:			
Cached: 17 / 32 <a href="#">Clean</a>			

# Stats – Lut Gholein

- A look at render statistics for the tavern in the Act 2 town
- Avg 3.85M prims, 3.7k draw calls
- Total asset counts and cache sizes are shown to the right

▼ Stats			
Memory Usage: 8746 / 12288 KB (71.2%)			
Draw Calls:			3750
Primitives:			3852369
Particles:			831
Popcorn Heap			
In Use			0 KB
Allocated			16440 KB
Mesh Heap			
In Use			1198 MB
Allocated			1280 MB
Texture Heap			
Estimated			0 MB ( )
In Use			1230 MB
Allocated			1536 MB
Texture CPU Memory (kb):	28519	/	24576
Job System (kb):	15	/	5120
► Timers			
▼ Asset Caches			
	Refs	Cached	Size
Models:	4938	1830	2750
Skeletons:	613	270	512
Animations:	6740	190	512
Animation Timelines:	281	157	512
Variant Files:	181	68	128
State Machines:	274	207	384
VFX:	722	446	1824
▼ Instances			
	Current	High Water	
Model Instances:	2637	2984	
Skeleton Instances:	264	270	
Average mesh instances:	1.198331	1.315189	
▼ Material Stats			
	#Mat	#MatPass	#MaxPass
All Materials	6885	56750	11
▼ Descriptor Tables			
	Descrs	Tables	
IGlobal	27312	2544	
Biome	22	2	
Biome Grass	22	2	
Compute	280	42	
Fullscreen	331	32	
Material	26393	2408	
Popcorn Uber	264	66	
▼ Pipelines			
Render:			
Cached:	1774	/	4896 <a href="#">Clean</a>
Compute:			
Cached:	17	/	32 <a href="#">Clean</a>

## *Pre-integrated Skin Shading, Penner SIGGRAPH 2011*

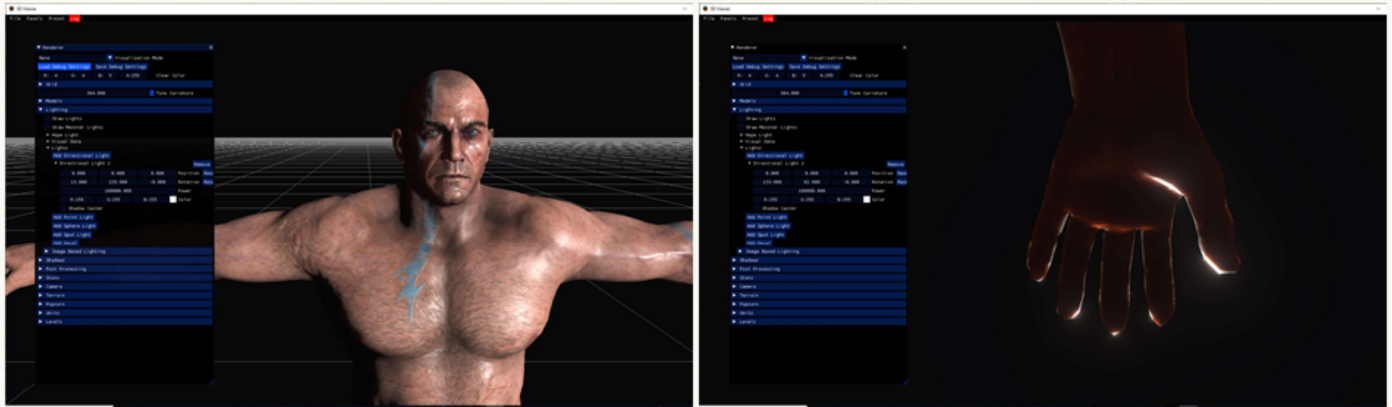


March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Our skin rendering is based on Penner's pre-integrated skin rendering from SIGGRAPH 2011. A diffusion profile for human skin is calculated and parameterized on NoL and curvature. While Penner's original technique calculates curvature using normal variance across a change of world space position, we found that it was convenient for our character artists to export a thickness texture alongside the typical albedo, normal, and roughness maps; and we could use thickness as a substitute for curvature and still get compelling visual results for skin.

## Pre-integrated Skin Shading, Penner SIGGRAPH 2011



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Here we can see some results.

# What about grass?

- Grass was challenging.
- We started with geometric grass. Great in 4K, but crazy noisy at any lower resolution.
  - Blades are tiny from the camera viewpoint
- Switched to grass cards with texture atlases for blade variety.



# Citations

- *Moving Frostbite to Physically Based Rendering*, Legarde and Rousiers, EA
- *Real Shading in Unreal Engine 4*, Karis, Epic Games
- *A Multiple-Scattering Microfacet Model for Real-Time Image-based Lighting*, Fdez-Aguera, JCGT 2019
- *Filtering Distributions of Normals for Shading Antialiasing*, Kaplanyan et. al. HPG 2016
- *Improved Geometric Specular Antialiasing*, Tokuyoshi and Kaplanyan 2019
- *Advanced Techniques and Optimization of HDR Color Pipelines*, Timothy Lottes, GDC 2016
  - <https://www.qdcvault.com/play/1023512/Advanced-Graphics-Techniques-Tutorial-Day>
- *High Dynamic Range Color Grading and Display in Frostbite*, Alex Fry, GDC 2017
  - <https://www.qdcvault.com/play/1024253/High-Dynamic-Range-Color-Grading>
- *HDR in Call of Duty* – Paul Malin
  - <https://research.activision.com/publications/archives/hdr-in-call-of-duty>
- *Weighted Blended Order-Independent Transparency*
  - <https://icgt.org/published/0002/02/09/>
  - <http://casual-effects.blogspot.com/2015/03/implemented-weighted-blended-order.html>
- *Pre-Integrated Skin Shading*, Penner, SIGGRAPH 2011
  - <https://www.slideshare.net/leeqoonz/penner-preintegrated-skin-rendering-siggraph-2011-advances-in-realtime-rendering-course>