Note: The PowerPoint PPT version of this presentation contains animations and movies.

In this talk we want share some of the tech behind the lighting and anti-aliasing used in the Decima engine.
And for those who don’t know what that is:
The Decima Engine is the game engine created by Guerrilla Games for the Killzone series.
But it has been extended and optimized since, to support bigger and more beautiful worlds, including that of Horizon Zero Dawn.
And recently, we also started sharing the Decima engine with Kojima Productions, and they’re now using it develop Death Stranding.
Today, Kohei and myself will talk about a few different and separate ways in which we recently extended the engine. I'll start with our spherical area light implementation. Kohei will then talk about height fog system that's being developed for Death Stranding. And then I'll talk about the anti-aliasing and checkerboard rendering in Horizon.
So, first up is spherical area lighting.
The Decima engine was probably one of the first game engines to support lighting using AREA lights. But that initial implementation didn’t support GGX, and that was becoming a limiting factor for us. So we recently started upgrading our area light system, and we wanted to add GGX to our SPHERICAL area lights first. However, integrating a micro-facet BRDF like GGX into an area light is difficult and requires some sort of approximation technique. In fact, a few different techniques have emerged over the years to do just that. Some of these techniques prefer quality over performance and some prefer the opposite. And many of these technique are particularly suited for just one type of shape. For our new SPHERICAL area lights, we looked around, and ended up picking the light bending trick described by Brian Karis. It’s cheap, and it’s successfully being used by many different game engines. And the results are often pretty decent. But it can cause some unexpected shapes to emerge, especially at grazing angles. You can clearly see that in the image on the left. But we found one way to improve that technique, and get what you see in the image on the right. So, how does this work?
Well, let’s start by going over Brian Karis’ technique first. The idea is that we want to approximate the result on the right with a single point light. And we can do that by moving the point light towards the perimeter of the area light in the direction of the reflection vector, like so...
This works quite well for Phong-based lighting, and was already described in ‘92. But it doesn’t always work that well for microfacet-based models like GGX. That’s because the selected position for the point light isn’t always a good representation of the most dominant influence.

In other words, the general idea here is to bring the peak towards the pixel, but the pixel can still miss the peak this way.
What we want to do instead is to move the point light to a different position on the perimeter of the area light. And we choose that position such that the point light will light the pixel the most. Ignoring the Fresnel term, the Geometry term, and the N dot L term, this happens when the normal N and the half vector H are the closest.

In other words, we want to look for the light vector that maximizes N dot H.
To solve this problem, we need a bit of math. And let’s start by defining the necessary variables.

Suppose we have the normal N, the view vector V and the reflection vector R. And suppose we have an spherical area light.

Now let’s assume that the center of this light is in the unit direction Lc. And let’s define the spherical SHAPE by the sine s, which is equal to the perimeter’s radius over its distance.
Now, let’s also define the unit vector $T$, which is the vector pointing towards $R$ but made orthonormal to $L_c$. That is, $T$ is simply the reflection vector $R$ that has projected on the plane perpendicular to $L_c$, and is then renormalized.

With this unit vector $T$, we can easily find the unit vector $L$ that’s pointing to the perimeter of the light and is closest to the reflection vector $R$. We do that by rotating $L_c$ to $L$, using the sine $S$ we already got, and the cosine we can calculate from $S$.

And this $L$ can be used to calculate the half vector $H$.

This is exactly the half vector used in Karis’ approach.
We can easily extend this idea by introducing another vector called B.

B is a bi-tangent, and together, Lc, T and B create an orthonormal basis. We can use that to define a normalized light direction L that points to any point on the perimeter of the area light...

...Including one that’s rotated around on the perimeter by some angle phi.

And, going back to our original problem, the trick is to now find the phi leads to the largest N dot H.
That way, we’ll find a more dominant specular response for the pixel we want to shade. And, in turn, that leads a better approximation to the area light.
But solving for \( \phi \) directly is rather hard. That’s because there’s a bunch of square roots, sines and cosines involved.

But we can reformulate that problem to an equivalent problem that’s easier to solve. For example, instead of solving for the angle \( \phi \), we can solve for \( x \), where \( x \) is the TANGENT of half \( \phi \).

This derivation is a bit tedious to show here, but it will allow us to get rid of all the sines and cosines using common mathematical half-angle identities. And we can do more. We can try to maximize \( N \cdot H \) squared instead of \( N \cdot H \). This will free us from the remaining square roots.

And what we end up with is a relatively simple rational polynomial called \( f(x) \).

And all we need to do now is find \( x \) that maximizes this \( f(x) \), and thus maximizes \( N \cdot H \) squared.

Sadly, that’s is still a non-trivial problem to solve analytically. But we can still solve this numerically using an iterative approach.
An iterative approach starts with a guess and tries to improve that. So we start with Karis’ approach, which is equivalent to starting with $x$ is 0.

And we then apply a single iteration of Newton’s method to get $x_1$. This brings us already a lot closer to the maximum, and we’ll simply take that as our final estimate.

Plugging $x_1$ into $f$ of $x$ gives us the $N$ dot $H$ squared that we were looking for, and we use that directly as input to the GGX function.
To get a feel for how complex that is to implement, here’s what that looks like in HLSL. This is an implementation of Karis’ approach, all in scalar math.
And this is what it looks like when we add one Newton iteration.
As you can see, this does add quite a few instructions, but compared the whole light shader, this may be a relative small change.
So depending on how this is used, the difference in performance might be noticeable, or it might be virtually free.
So is it worth it?
Well, here you see the sun’s reflection with and without the N.H maximization.
And as you can see, it works quite nicely, as it gets you a more natural looking reflection
at low angles.

I should mention though, than even though these screenshots have been made in
Horizon Zero Dawn,
the technique was only finished after Horizon shipped.
But it has been retro-fitted for internal use, and is ready for use for other titles.
The next topic is about height fog and will be covered by Kohei.
In this section I will introduce our height fog model, which is a unique Decima Engine customization created by Kojima Productions.

First of all, please let me explain why this optimization was necessary. Decima Engine has a flexible atmospheric scattering system linked with clouds and outdoor lighting. However, at Kojima Productions, there was a request for an atmospheric scattering system optimized for photo-realistic rendering. Precomputed atmospheric scattering is one of the best methods for photo-realistic games. In this model, the aerial perspective is strongly dependent on the precomputed Look Up Tables. It works well for the photo-realistic atmosphere, but we often need to express artistic quantities such as dense fog and colored fog in the actual game.

So, it would be nice if we can express the photo-realistic aerial perspective and the artistic height fog with a single height fog model. To achieve this, we introduced an analytic height fog model, and combined with a precomputed atmospheric scattering model. Let’s take a look at how this combination works.
This is an overview of our approach.

* The volume rendering equation

The starting point is the volume rendering equation, and the earth's atmospheric conditions. The earth's atmospheric conditions are described by the Rayleigh scattering and the Mie scattering, and the participating media with exponentially decreasing air density with altitude. Both of the precomputed model and the analytic model share these conditions.

* Precomputed atmospheric scattering model

The precomputed atmospheric scattering model performs numerical integration, and stores the in-scattering and transmittance information into Look-Up-Tables.

* Height fog model

On the other hand, the height fog model calculates the fog directly when we render the scene. This calculation is an analytic solution of the volume rendering equation with assuming the single scattering. This kind of analytic solution is known for the uniform fog model[HP02], but we couldn’t find a satisfactory solution for the height fog model.
* Height fog model
So, we derived it independently.
In order to derive it, we assumed single scattering and single scale-height.
Here, the scale height is a parameter to describe how decreases the fog density with altitude.

* Combination
In our approach, we supplied the skylight and the sunlight information from the precomputed atmospheric scattering model to the height fog model.

That concludes the basic overview.
In the next slide, we will focus on this combination.
The combination we are assuming is a hierarchy approximation of the volume rendering equation.

First, let us assume there is sunlight.

This extraterrestrial sunlight is incident into the earth, and interacts with the air. Our precomputed model generates its multiple scattering information, and stores it as the LUT $L_{\text{inscatter}}$, $L_{\text{sun}}$ and $L_{\text{sky}}$.

The height fog model receives the background radiation, the sunlight and the skylight from the precomputed LUT.

It seems a trivial task to achieve this type of LUT if we develop the precomputation tool appropriately. However, the problem is whether such a height fog solution can be obtained or not, especially for non-uniform fog.

In the following slides, I will explain the precomputed model and the height fog model in detail.
Our precomputed atmospheric scattering model is nothing special, but the LUT is different from the typical model. Our implementation is mainly based on Bruneton’s [BN08] and Elek’s model [EK10] with some optimizations [Hil16]. For details of the precomputation algorithm, please refer to the references shown on this slide.

A typical precomputed atmospheric Model has an In-scatter and Transmittance LUT. But we replace the transmittance LUT with Sunlight and Skylight LUTs, because we want to give the Sunlight and Skylight information directly to the Analytic Height Fog model.

By the way, we precomputed the two weather types into the LUT. Here, “two weather types” means two different scattering settings. This is because we found that we can express the various weather conditions with no runtime precomputation if there are sunny and dense fog weather conditions as a base.
This is an example of our LUT.
These sizes are optimized for games on the ground or near the ground.

Now we have the scattering information as 3 LUTs.
Let’s see how these LUTs are used in the height fog model on the next slide.
Before getting into the height fog model, one comment about the sky.
Background radiation, namely the sky is made with In-scatter LUT and the phase function.
This is the same as the Elek’s model[EK10].

OK, let’s take a look at the analytic height fog model.
The model has a form of blending background and sunlight + ambient with $\alpha$.
$L_{background}$ means the background radiation, sky.
$L_{sun}$ describes single scattering component of sunlight.
And $L_{amb}$ is constant ambient light component from the sky.
$\alpha$ is a blending factor with the background. The decreasing fog density with altitude is expressed here.
At first glance, this alpha blending structure looks like a model that has been adjusted for the game.
However we can derive it from the volume rendering equation directly.
If you are interested in the derivation of this model, please contact me after this session.
This is a sample code of our height fog model.
This is the result.
I will explain these in order from the left end.
The left end is the final result.
Next is the background radiation.
Next is the background radiation multiplied by the alpha. We can see how the transmittance alpha is causing the color of the sunset as well.
The next is the sunlight component.
The right end is the ambient component.

When we sum up the three on the right, we get the result on the left.
This is a sample scene from the Decima Engine.
We can see an aerial perspective from sunny to cloudy weather with the inclusion of an artistic atmosphere.

So, with this result, we achieve both a realistic aerial perspective and an artistic height fog with a single height fog model.
In the next section I’ll be talking about our anti-aliasing solution.
There's many different ways to do anti-aliasing, including super-sampling, multi-sampling, post processing, and temporal AA.
For Horizon, we tried to only use FXAA...
... and SMAA for some time, but it was clear that any technique purely based on post-processing wouldn’t cut it. These technique are pretty good at removing jaggies from clean and clear input, but we don’t have a lot of that in Horizon. Instead, our AA solution would need to cope with heavily curved, undersampled, ambiguous, alpha-tested and alpha-blended input.

But what’s more is that post processing techniques can also be rather EXPENSIVE and scene dependent. SMAA, for example, can takes up to 2 milliseconds when looking down at the grass in Horizon. And that’s 3 times slower than just looking up at a clear sky.
Obviously, instead of morphological post-processing techniques, we could have considered multi-sampling techniques or analytical techniques, for example. But we simply couldn’t afford the overhead in complexity and performance that that would have given us, as we were looking for something that would cost less than 1 millisecond in total.

So, like many others, we turned to temporal anti-aliasing, which is effectively doing anti-aliasing over multiple frames.

And in the end, we settled on a combination of FXAA and our own variation on TAA, and use each of their strengths as effectively as possible. And using that, we get this..

And after the necessary optimizations, our FXAA plus TAA PLUS UI compositing system is now taking up at most 1 millisecond per frame on the PS4 in 1080p.
Let’s look at a more detailed comparison.

And focus on the highlighted area.
This is that zoomed-in area without any AA technique. Here, you can clearly see the jaggies on the steps, the undersampling between the stones, and the rough look of the hair.
And this is with FXAA. And you can see, it does help a bit, but it only hides the jaggies in clear straight lines, like the horizontal lines between the stair steps.
This is with SMAA. This again, is slightly better than the previous one,

But it still doesn't fix these areas.
And this is with FXAA plus TAA. The TAA uses different sample positions for odd and even frames, and resolves to THIS after only two frames. Notice how the undersampled areas are now closed and a lot smoother.

You probably also notice the slight halo around the dark diagonal line. That’s because we also sharpen the output of the TAA a bit.
Not only does our TAA help with sampling more details. But it also helps with temporal STABILITY.

Here’s a short clip of a typical environment in Horizon.

And let’s zoom in a bit.

This is without any AA. Notice how noisy and flickery both the short and tail grass looks.

This is SMAA. It clearly helps, but it’s still quite noisy.

This is TAA. Notice how much calmer everthing looks now.
OK. So TAA helped us a lot. But getting that to work RELIABLY was somewhat of a challenge.

That’s because all TAA implementations depend on some form of history reuse. And this is typically done by accumulated many frames over times, and that CAN work great. But doing that didn’t work that well for Horizon.

The reason for that is that our AA is applied on often very thin geometry, Plus we apply AA only AFTER we already overlayed a lot of visual FX and all post effects. And that includes effects like Depth of Field and vignetting.

And lastly, we base our TAA reprojections on a HALF-RES motion buffer, which is probably not ideal. And some types of objects don’t even output motion vectors at all.
So, for those reasons, we prefer not to use any previous frame for too long. And so, instead of somehow accumulating the output from previous frames, we chose to only reuse the input to the AA system from the previous frame, but not the previous output.

That is, we use the raw render of the previous frame and the current frame, but nothing more. That way, we prevent any long ghosting trails caused failed rejects, and we reach a stable and final result in only two frames, making our technique very responsive.
So let’s dive into some the details of our solution.

For one, we never render at the pixel centers, but only on the pixel’s edges. In fact, we render HORIZONTALLY BETWEEN the pixel centers for odd frames, and VERTICALLY between pixel centers for even frames. We do this by jittering the camera in this precise pattern, and we don’t rely on any form of multi-sample hardware like EQAA. And we don’t have to process or output pixel shaders any differently.

After we’ve rendered the jittered frame, we apply all post effects, and run FXAA on these samples at the pixel’s edges. After the FXAA, the samples are then resampled from the pixel’s edges to the pixel’s centers.

You could do this by simply averaging the two samples closest to each pixel center. But instead, we use a 4-tap resampling kernel with negative lobes to counter any texture blurring. This is a bit like Catmull-Rom resampling, but cheaper, and the amount of sharpening is determined based on local contrast.
The final resolve tries to merge the previous sharpened frame with the current sharpened frame.
If history is accepted, this gives us effectively 4 samples of data per pixel.
And when history is rejected, we still get to use 2 samples per pixel that were already treated with FXAA.

As an implementation detail, we actually only run two passes to do all this. First, we run the FXAA pass.
And in the second pass, we sharpen the current frame, and OUTPUT that to one of the two history buffers.
In the meantime, from within the same shader, we also READ from the OTHER history buffer, which contains the sharpened version of the PREVIOUS frame, and we use that to either reject or reproject.

If a pixel from the history is reprojected, it’s blended 50/50 with the current SHARPENED frame, and we output that to the final backbuffer.
But if the history sample is REJECTED, we simply output the CURRENT frame’s pixel WITHOUT sharpening, to hide any potential aliasing as good as we can.
You may have noticed that, if you’d successfully merge two frames and ignore the sharpening, we effectively get the sampling pattern that you see on the left.

And in a way, this is a bit like the PATENTED FLIPQUAD technique, which also uses samples on the pixel’s edge. However:
- Our pattern is slightly sharper as the samples are closer to the center.
- And unlike rendering the FLIPQUAD pattern over two frames, we don’t need any additional logic or features to get correct mipmap sampling, but this was already hinted at by Michal Drobot in 2015.
- Another nice thing about our pattern is that don’t require any multi-sample features to render at the right sample positions. We only need to jitter the camera per frame.
- But the downside of all is that it’s less good at removing jaggies from very regular straight patterns like lines and grids, but that’s largely fixed by the FXAA pass we also apply.

OK. For other aspects of our TAA solution like temporal reprojection and neighborhood-based rejection criteria, we used some more traditional approaches, so we won’t cover those here.
Instead, we’ll jump to the next and last topic, which is about how we render for 2160p on the PS4 Pro using checkerboard rendering.
Let’s start with a comparison.

On the left you see the same image I’ve shown you before, which is a zoomed-in 1080p screenshot using our FXAA plus TAA solution.

In the middle, you see what you’d ideally get on 2160p. This wouldn’t run at 30fps on a PS4 Pro, especially when using 16 times supersampling. But it does show what we’d ideally achieve if performance wasn’t a problem.
On the right, you now see what you COULD get at 30 fps on a PS4 Pro: A native 1512p image using the same AA technique as in 1080p. Notice how it's clearly sharper than 1080p, but it still misses much of the detail that you DO see in the reference.

Ok, so how does this compare to our checkerboard rendering?
Well, our checkerboard implementation (that you now see on the left) also runs at 30 fps, just like the native 1512p mode, but it’s clearly more detailed. In fact, it shows all the details that are also visible in the reference.

But when you compare the sharpness of our checkerboard result to the 16x supersampled reference, you do see a difference. This is an inherent property of how we do checkerboard rendering: It values more detail over more sharpness. And we prefer it that way, if we’re forced to choose between the two. Sharpness is great when you’re looking at imagery from close by. But actual detail tends to be visible, or at least be hinted at, over a wider range of viewing distances, And it also contributes more to the final image quality when having to downscale to a 1080p TV, for example.
Our checkerboard solution is also quite stable TEMPORALLY. Here’s a quick comparison.

Let’s start with the exact same scene as shown before, in 1080p with FXAA+TAA.

And this rendered with our 2160p checkerboard rendering. And notice how similar it is in visual stability.

And again, 1080p.

And 2160p.
Alright, so how does this work?

Well, with checkerboard rendering you typically SHADE only 50% of the pixels.

The other 50% is shaded next frame.

So, ideally, you get a native-res result after two frames.

However, dynamic scenes complicate this process quite a bit.

That’s why you’d typically ALSO render NATIVE-res hints EACH FRAME.
Hints like depth, or PrimID.
And to complicate this even further: Alpha-tested and alpha-blended objects may still need to be sampled and processed at native res, each frame.
All that native-res overhead prevented us from running Horizon at 30 fps.

Obviously, we could have done what many other titles do, and that is to simply render at checkerboarded 1800p instead. And maybe that would have given us a similar or even better sharpness, but the images would’ve been less detailed and would have gotten a more filtered look. So instead, we chose to look for a way that still gave us the detail level of 2160p and would still run at 30 fps, but at the cost of sacrificing some of the potential sharpness.

We eventually found a way to do that, and we ended up removing all native-res hints, and only used native-res buffers for the UI and for the final backbuffer. Everything else ran at checkerboard resolution, including alpha-tested objects, alpha-blended objects and post effects. Well, except for some FXs that always ran at a lower resolution anyway.

Doing all that took care of the performance problems. But we still had to find a way to resolve the checkerboard data without the typical native-res hints. And we found some inspiration in the way we do TAA in 1080p.
Normally, checkerboard rendering is used to render 50% of the pixel CENTERS.
And the other 50% procent is then guessed based on the native-res hints, the history buffer and the current checkerboard buffer.
What we do instead is that we render 50% of the pixel CORNERS.

This gives us 2 samples for each pixel to work with, instead of only 1 sample per 2 pixels. And so, we can simply average the two closest CORNER samples to get the value at the pixel CENTER.

What’s more is that this requires no native-res hints, and not even any morphological tricks. And because every pixel is now treated with the same approach, we implicitly prevent the dither artefacts that are often associated with checkerboard rendering.

And after two frames, ...
... it gives us 4 samples per pixel, resulting in the same AA STABILITY that our TAA gave us at 1080p.
One difference though, is that our TAA works with samples on the EDGES of pixels, while this works on the CORNERS.

Doing the checkerboard resolve like this is fast and clean, and after TWO frames, it contains all 2160p details you’d expect from a native 2160p render. But even with only a single frame, the results are pretty decent.
Here’s an example.

On the left, you see a near-vertical feature at native-res.

But with checkerboard rendering you actually only render what you see in the middle.

After resolving the pixel centers by averaging the two closest pixel corners, you get what you see on the right.
Notice that it’s pretty close to the one on left.

The only difference is that it’s slightly less sharp. But, again, that’s inherent to our technique.
In general, this trick already works pretty well. And it’s a clear step up compared to our 1080p solution.

But it can use some extra work on diagonal lines. That’s because the diagonal direction is the sparsest direction in checkerboard rendering.

In this example you see here, you clearly see that the ‘corner resolve’ can still lead to a somewhat jagged look of the diagonal line.
To solve this, we run FXAA before we apply the corner resolve. And looking at the bottom right, you can clearly see that this helps quite a bit.

Notice that we apply FXAA on the checkerboard pixels, not on the resolved native-res pixels. That’s because it’s twice as fast that way, but also because it works a lot better. Especially because we also trick FXAA to focus on the diagonal lines instead of the more typical horizontal and vertical lines.

We do this by effectively rotating its input. And here’s how.
To apply FXAA, and to reproject the checkerboard render from the previous frame, we don’t want to use the checkerboard data directly. That’s because pixels aren’t square in checkerboard rendering. They’re elongated and behave sort of zigzaggy. But there is a trick we can use to interpret and interpolate this more meaningfully.

When we only look at the checkerboard sample positions and the Voronoi regions they create, ...
... we can interpret checkerboard samples as pixels, rotated by 45 degrees.
But when we rotate the framebuffer by 45 degrees as well, we end up with square pixels again.

So if we could store a rotated framebuffer instead, we could simply use that to sample using bilinear filtering, apply FXAA, and do a plain reprojection again.

The downside, of course, is that this doesn’t look like regular buffer or texture anymore. But we can fix that.
We can transform this rotated buffer into what we call a ‘tangram’. We call it a tangram because it’s sort of like that so-called puzzle game.

We can cut the rotated buffer into parts and shuffle them around like so. The nice thing about that is that it’s completely lossless, and allows the 2160p checkerboard data to be packed into a compact 2160x2160 texture again. And it also still supports bilinear sampling. And because of the exact way we placed these parts, we can use the built-in texture-wrap hardware to do the unwrapping for us, without any additional logic or shader instructions during sampling.

The only thing required during sampling is rotating the native-res UV by 45 degrees, and offsetting this by an offset that’s constant per frame.
You may have noticed there’s actually some padding in this tangram as well. That’s a side effect of this exact shuffling and is required to make the wrap around work correctly. But as a result, a tangram takes up 12% more memory than a regular checkerboard buffer.

And yet it doesn’t take up any more bandwidth, because there’s simply no reason to sample the padding.
There's also no need to write to the padding areas. In fact, a tangram can be rendered using only 3 precisely positioned quads. That way, every checkerboard pixel is rendered exactly to one position in the tangram.

Depending on the pipeline, it may even be possible to apply this process from within an already existing full-screen pass, making this tangram transform virtually free.
So to finish this topic, let me summarize our checkerboarded rendering approach:

- We first render in checkerboard resolution without any native-res hints. We also apply all post effects at that point, do the tonemapping, encode the output in either sRGB or PQ, and store the result in a 10-bit format.

- We then transform that into a tangram, and we store that in YCoCg color space.

- The YCoCg tangram is then sampled by the FXAA pass, which outputs a new tangram. The reason we use YCoCg is because FXAA does most of its work on luminance data, and this allows us to sample 4 luminance values per texture gather.

- The output of the FXAA pass is stored in a ping-ponged buffer, giving us the current and previous frame to resolve from.
As a final pass, we take the current and previous tangram, convert back to RGB, and reject or reproject based on standard criteria. If history is to be rejected, we only sample the current tangram, giving us effectively 2 samples per pixel. And if history is accepted, we blend the previous and current value 50/50, and we end up with effectively 4 samples per pixel. And conceptually, this last part is quite similar to our TAA solution in 1080p, but without the sharpening.

And that’s how we resolve in checkerboard mode in less than 2 milliseconds.
And that was also the end of the material we wanted to present today.

Obviously, some details had to be were left out for brevity. But for those who are interested, be sure to download this presentation and check out the references and detailed code snippets at the end of this slide deck.
Q&A

Special thanks

Natalya Tatarchuk
Maarten van der Gaag
Jeroen Krebbers
Hugh Malan
Jaap van Muijden
Kevin Örtgren
Roderick van der Steen
Nathan Vos
Andreas Varga
Tobias Berghoff
Michiel van der Leeuw
Hermen Hulst

Kojima Productions graphics team
References


Appendix: Spherical Area Lighting

This sample code is provided as an example on how to use GetNoHSquared(). It also includes some further adjustments to better match our reference:

With the presented light bending techniques, the intensity falls off outside the light’s area reflected on the surface, but is constant inside it, which isn’t realistic for rough materials. That’s why we also reduce the radius based on alpha.

As the light bending techniques mess aren’t energy conserving, we need to compensate for this somehow. We’re currently experimenting with alternative formulas for normalization to better match our reference. This, however, is still in flux.

Lastly, we only use GetNoHSquared() to modify the input to the GGX microfacet distribution. The Geometry, Fresnel, normalization and final N-dot-L are left unadjusted. This leads to zero intensity when the CENTER of the area light is on the horizon, which strictly speaking isn’t correct, but it does prevent shadow acne at the terminator, which is more important to us.

NOTE: The code has been adjusted for this presentation and hasn’t been tested in this form, and (thus) comes without any guarantees.
This is some sample code to sample a tangram based on a native-res UV and, in this case, do a corner-to-center-resolve using a single bilinear sample.

NOTE: The code has been adjusted for this presentation and hasn’t been tested in this form, and (thus) comes without any guarantees.
This is example code to populate a vertex buffer with exactly enough pos&uv vertices to render a list of 3 quad. This can be used to render a tangram of size \( \text{native\_height} \times \text{native\_height} \) using a simple point-sampling blit function. Note that the UV coordinates simply point to the four corners of the raw checkerboarded input buffer.

When the blit function is replaced with a full-screen effect shader that can run on the checkerboard data directly, you can even execute that effect AND do the checkerboard-to-tangram transform in the same pass.

NOTE: The code has been adjusted for this presentation and hasn’t been tested in this form, and (thus) comes without any guarantees.