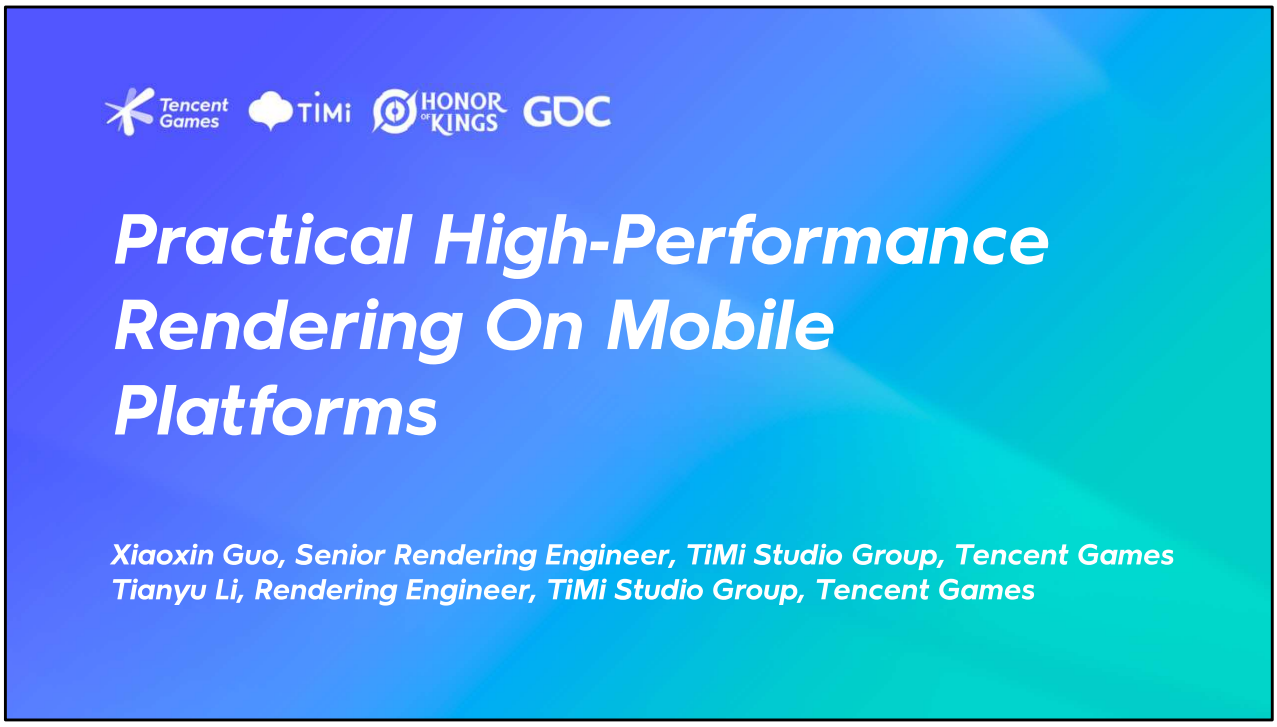


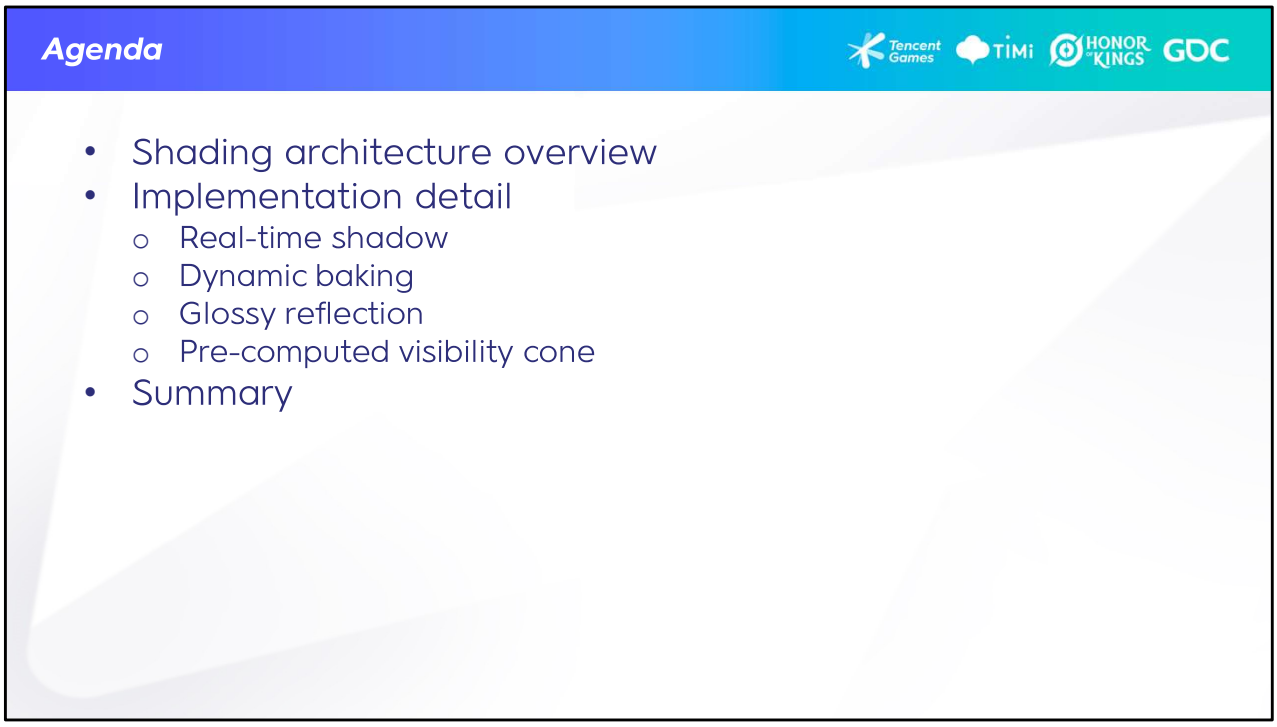
Good morning/afternoon/evening, and thank you for joining us. My name is Xiaoxin Guo, and I'm a senior rendering engineer at Timi L1, a Tencent Game studio.

My colleague Tianyu Li and I have collaborated with our engineering team to bring you this presentation, "Practical High-Performance Rendering On Mobile Platforms". Unfortunately, Tianyu couldn't join us at GDC this year, so I will be presenting on our behalf.

During this presentation, we'll be discussing the challenges we encountered while developing rendering technique on mobile platforms, and how we solved them.

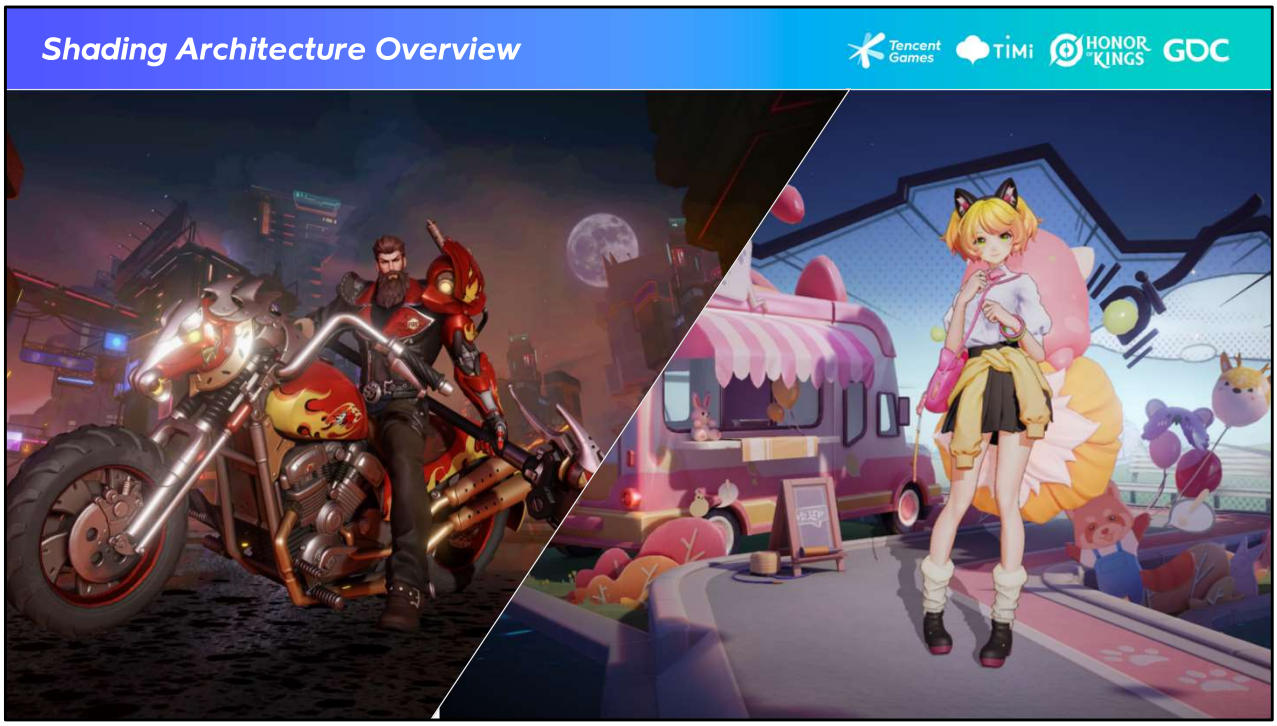
Before we get started, I'd like to play a prompt video of our game. [CLICK]

This video showcase two standard rendering scenarios: the top-down view during gameplay and short cinematics of the character performing. Today's discussion will focus on the latter.



To begin, I will provide a brief overview of our shading architecture.

Afterward, we will discuss various topics such as real-time shadow, dynamic baking, glossy reflection, and pre-computed visibility cone.



We use a forward render path in our renderer to support the different material types needed for our game, which includes both photorealistic and non-photorealistic art styles.

Our lighting system consists of various components, including real-time punctual lights, dynamic and static baked lights, and normalized image-based lights.

In terms of real-time punctual lighting, we offer support for a restricted number of directional, spot, and point lights.

Additionally, we provide support for other types of lights such as mesh lights for both dynamic and static baking. However, the number of dynamic baked lights is limited due to performance constraints.

In this section, we will discuss real-time direct lighting and the challenges associated with it.

Real-time Direct Lighting

- $L_d = \int Fr * L_e * V * \cos(N, L)\, dW$
- Shadow quality is crucial for direct lighting
- What are the challenges on mobile?
  - Unexpected shadow bounding sphere
  - Shadow bias behaves differently on each platform

Direct lighting refers to the integration of the BSDF (bidirectional scattering distribution function), incoming radiance, shadow, and cosine terms over the spherical domain.

As you can see in the screenshot, shadows are an essential visual phenomenon for direct lighting.

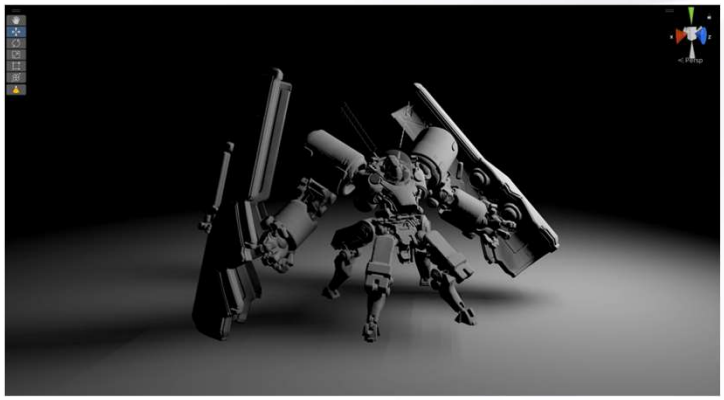
To solve the problem of shadows in real-time rendering, shadow mapping is the most commonly used technique.

However, this technique is not artifact-free, and we face more difficulties using it on mobile platforms due to performance limitations and platform differences.

Shadow Bounding Sphere

Problem statement

- Parallel split shadow map (PSSM in abbrev.) didn't fit our game
- Spot light uses a large opening angle causing low shadow map texel density
- Spot light shadow is not stable during animation



One of the issues we encountered during shadow mapping development is the unexpected shadow bounding sphere.

The shadow bounding sphere is basically a sphere that covers all the objects that cast a shadow

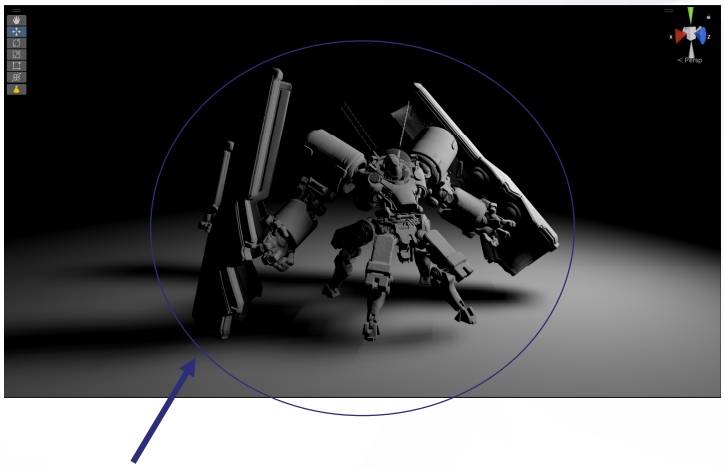
Typically, the bounding sphere for directional light is managed by the PSSM (parallel split shadow map) strategy. However, it doesn't fit our game due to the shadow split transition artifact, and we can't afford to blend between different shadow splits.

For spotlights, the shadow bounds is closely related to the light radius and opening angle. Due to our artistic choice, we may use a very large opening angle for spotlights, and it can be animated. [CLICK]
This video shows the resulting artifacts.

Shadow Bounding Sphere

Our observation

- We can use more tight bounding volume in many cases
- PSSM is a methodology to manage shadow bounding spheres, we can generalize the idea



Use tight bounding sphere

In this next screenshot, the light illuminates the robot and the ground, covering a large area.

However, the robot is the only shadow caster, so we should use a tight shadow bounding sphere for it.

This type of situation occurs frequently in our production assets.

We noticed that PSSM is a methodology for managing shadow bounding spheres, and we can generalize the idea and apply it to spotlights to improve their shadow quality.

Shadow Bounding Sphere

Implementation

- Art places bounding spheres in the scene
- Loop over visible lights:
  - Cull invisible bounding spheres for current light
  - Sort bounding spheres by texel density : bounidng_sphere_radius/shadow_map_size
  - Render shadow split for sorted bounding spheres
- For shadow map filtering in pixel shader, loop over sorted shadow split list, use the first valid one as it has the best quality over others.

The implementation of this technique is simple.

The artist places the bounding sphere in the scene.

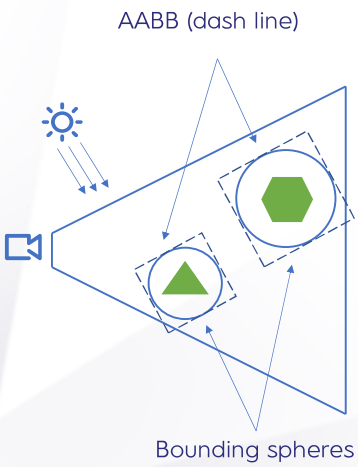
During the shadow map rasterization pass, we loop over all visible lights.
For each light, the first step is to cull invisible bounding spheres for the current light,
then sort them by texel density,
and finally, render the shadow split for the sorted bounding spheres.

To render the shadow split with the bounding sphere, we need to handle perspective and orthogonal projection differently, which we will explain in the next slides.

To test light visibility at the pixel stage, we loop over the sorted shadow split list for the current light and use the first valid shadow split as it has the best quality over others.

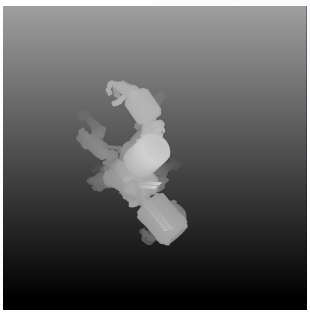
Shadow Bounding Sphere

Render shadow split: directional light

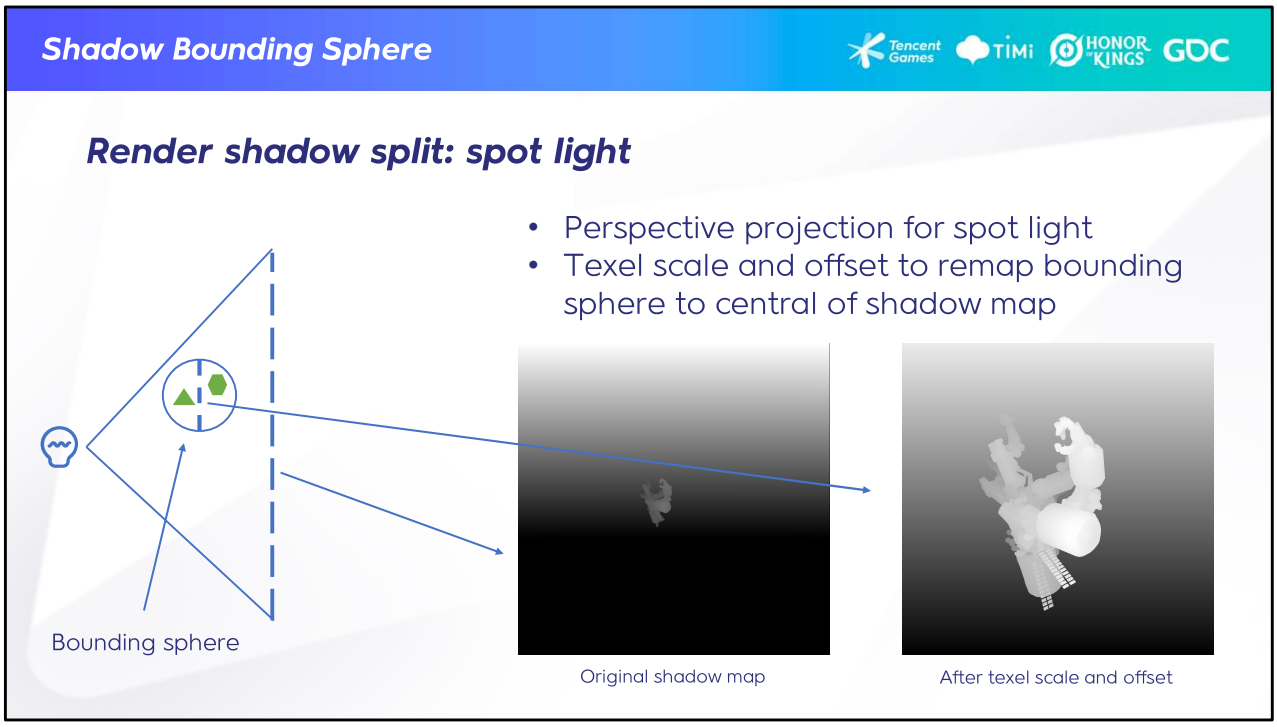


- Enclose the bounding sphere by AABB in light space
- Orthogonal projection


Shadow map without bounding sphere

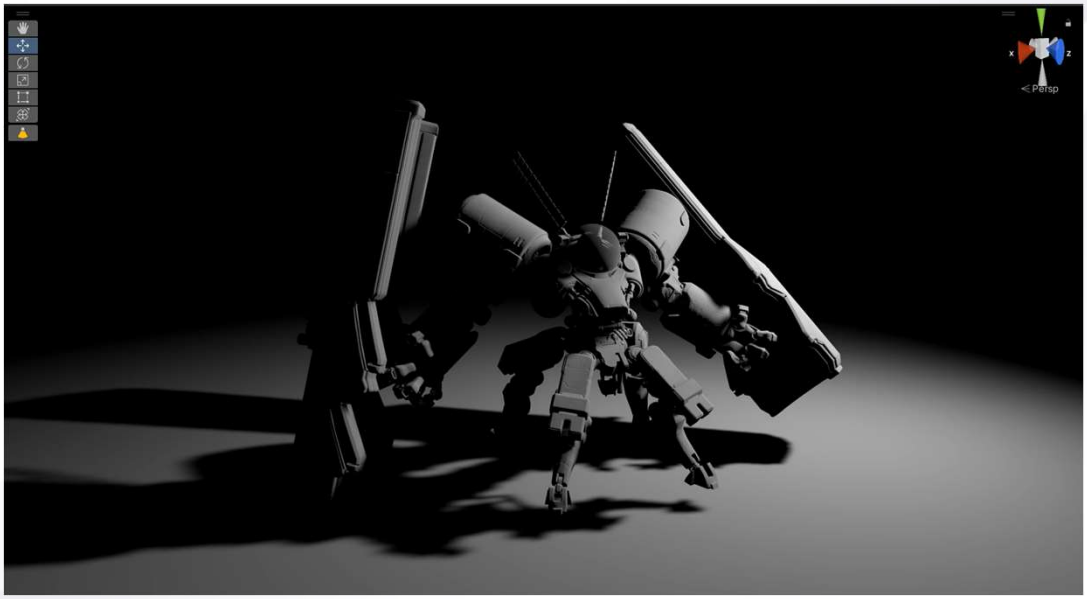

Shadow map with bounding sphere

As our method is a generalization of PSSM, the shadow split render for directional light is the same compared to PSSM. Therefore, I won't dive into the details in this talk.



For spotlights, the shadow split render uses the standard view and projection matrix, [CLICK] and we add texel scale and offset to the shadow matrix [CLICK] to make the bounding sphere located in the center of the shadow map.

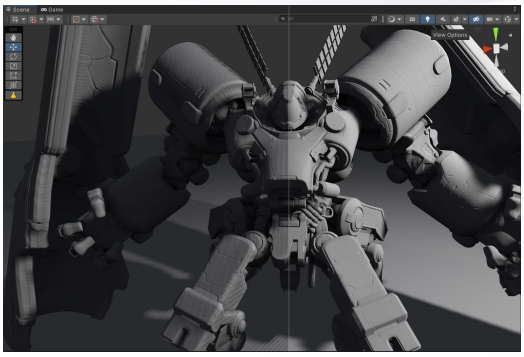
Shadow Bounding Sphere



With fine-tuned shadow bounding spheres, we've seen a dramatic improvement in shadow quality, and they are now artifact-free during animation.

Shadow Bias

- Shadow acne artifact
- Usually solved by applying
  - Depth slope offset during shadow map rasterization
  - Normal offset during shadow map filtering
- Depth slop offset is problematic for cross-platform development
  - Graphics API difference
  - Depth buffer format difference
  - Art tweak parameters on PC, may look bad on Phone



Another common artifact with the shadow map technique is shadow acne, which is caused by oversampling or undersampling, and is difficult to fix completely within the rasterization scope.

To reduce this artifact, we need to apply a depth slope offset during shadow map rasterization, as well as a small normal offset during shadow map filtering.

However, determining the optimal depth bias and normal offset for a given scene requires light-by-light tweaking, as there is no one-size-fits-all parameter.

Additionally, depth slope offset can pose problems for cross-platform development, as it behaves differently for different graphics APIs and depth buffer formats. Art tweaked for PC may not look good on a phone.

Shadow Bias

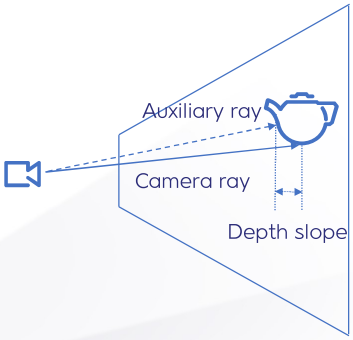
Requirements

- High performance and robust
- Support both directional, spot, and point lights (perspective and orthogonal projection)
- Independent from graphics API
- Independent from other parameters. For example shadow map resolution, shadow bounding spheres, …

Our goal is to develop a depth slope offset technique that meets the following requirements:
high performance and robust,
support for directional, spot, and point lights,
and independence from graphics APIs and other parameters such as shadow map resolution, shadow texture format, bounding spheres, and so on

Shadow Bias

- Depth slope offset revisit:
  - Maximum of the horizontal and vertical slope of the depth value at the pixel
  - depth_slope = max(ddx(pixel_depth), ddy(pixel_depth))
- Can we compute depth_slope in vertex shader?
  - Has analytic solution for triangle use ray differential
  - Approximate face normal by vertex normal

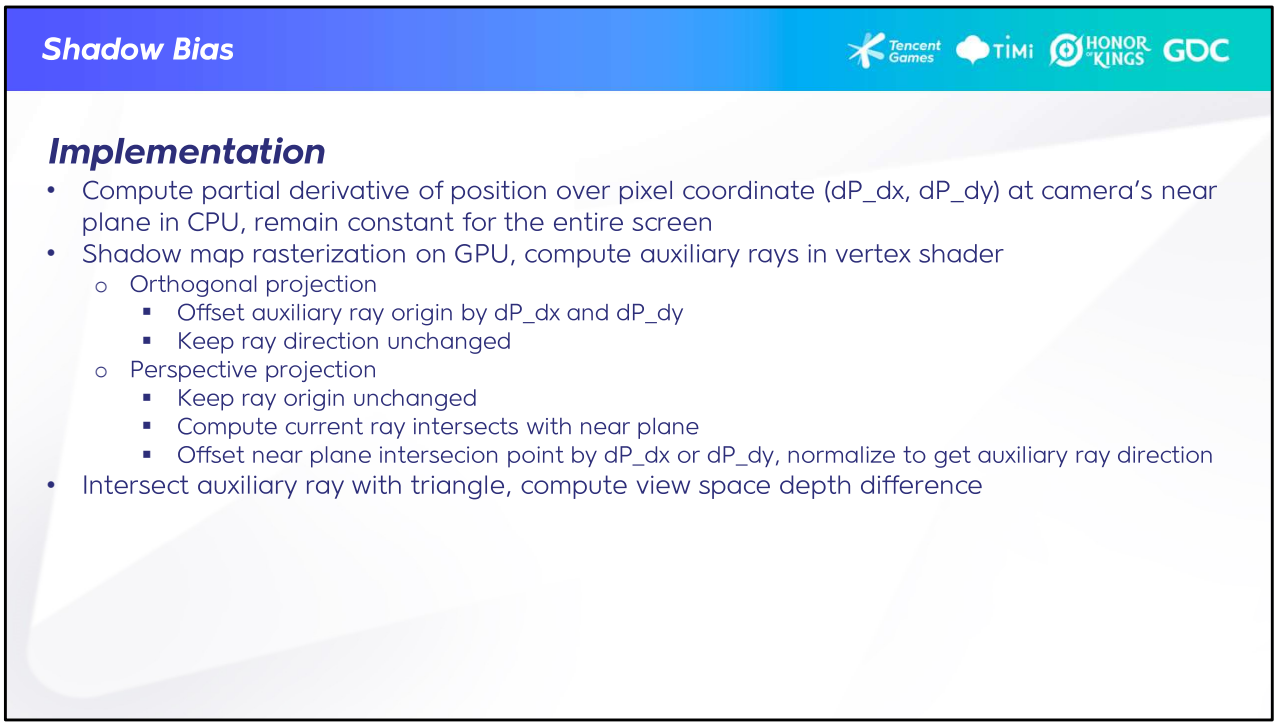


To understand depth slope offset, we must first define it as the maximum of the horizontal and vertical slopes of the depth value at the pixel.

We can implement this in a pixel shader using the max, ddx, and ddy functions. However, for high-performance implementation, we cannot afford to calculate it in the pixel shader. Instead, we can compute it in the vertex shader using an analytic solution for triangles that uses ray differentials.

Ray differentials have many applications in computer graphics, one of them is to perform texture mipmap filtering. The key idea behind ray differential is to use it to compute the partial derivate of some values, which is the depth slope in our use case.

Using ray differential requires to access the face normal in our case, but we do not have access to it in the vertex shader, instead, we can approximate it using the vertex normal. Fortunately, we do not observe any artifacts resulting from this approximation.

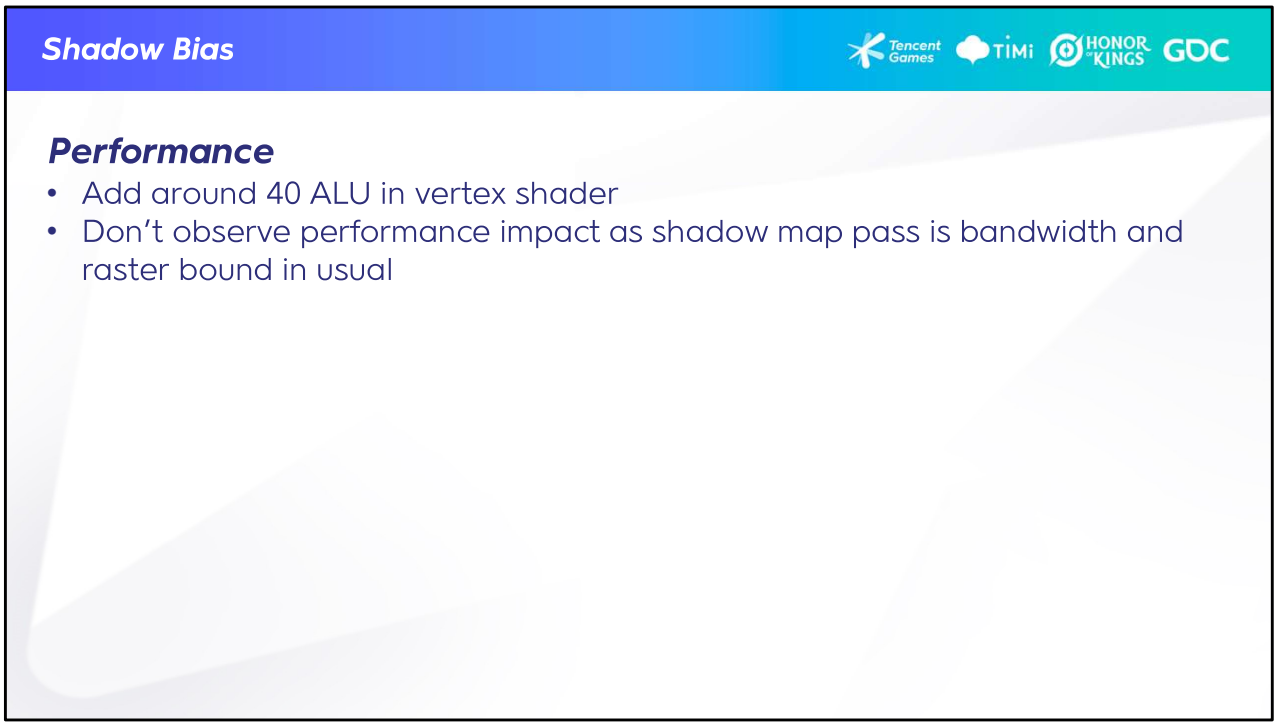


To compute depth slope offset in vertex shader, by ray differential, we use two auxiliary rays that intersect with the triangle.

On the CPU side, we calculate the partial derivatives of position over pixel coordinates at the camera's near plane, denoted as dP_dx and dP_dy, which remain constant for the entire screen.

During shadow map rasterization on the GPU, we compute auxiliary rays in different ways depending on the projection type.
For orthogonal projection, we offset the ray origin by dP_dx and dP_dy while keeping the ray direction unchanged.
For perspective projection, we keep the ray origin unchanged and compute the current ray's intersection with the near plane, we then offset the near plane intersection point by dP_dx and dP_dy, connect them with ray origin, normalize the result to get the ray direction.

Once we have the auxiliary rays, we intersect them with the current triangle, which gives us three intersections. We then calculate the depth difference of the intersections in view space and use it as the depth slope to offset the vertex.



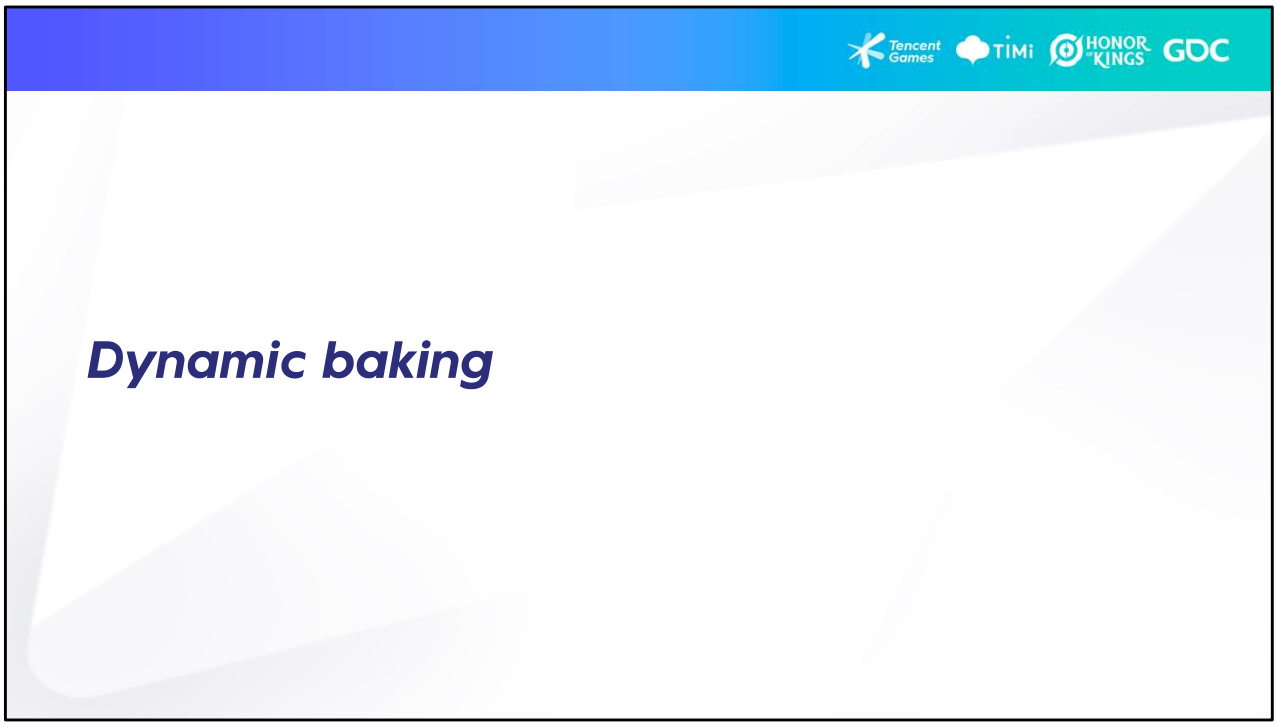
Using our software-based depth slope offset adds approximately 40 arithmetic logic units (ALUs) to the vertex shader.

Despite this increase in complexity, we didn't observe a significant performance impact since the shadow map pass is typically bandwidth and raster bound.

Direct Lighting + Static Baking

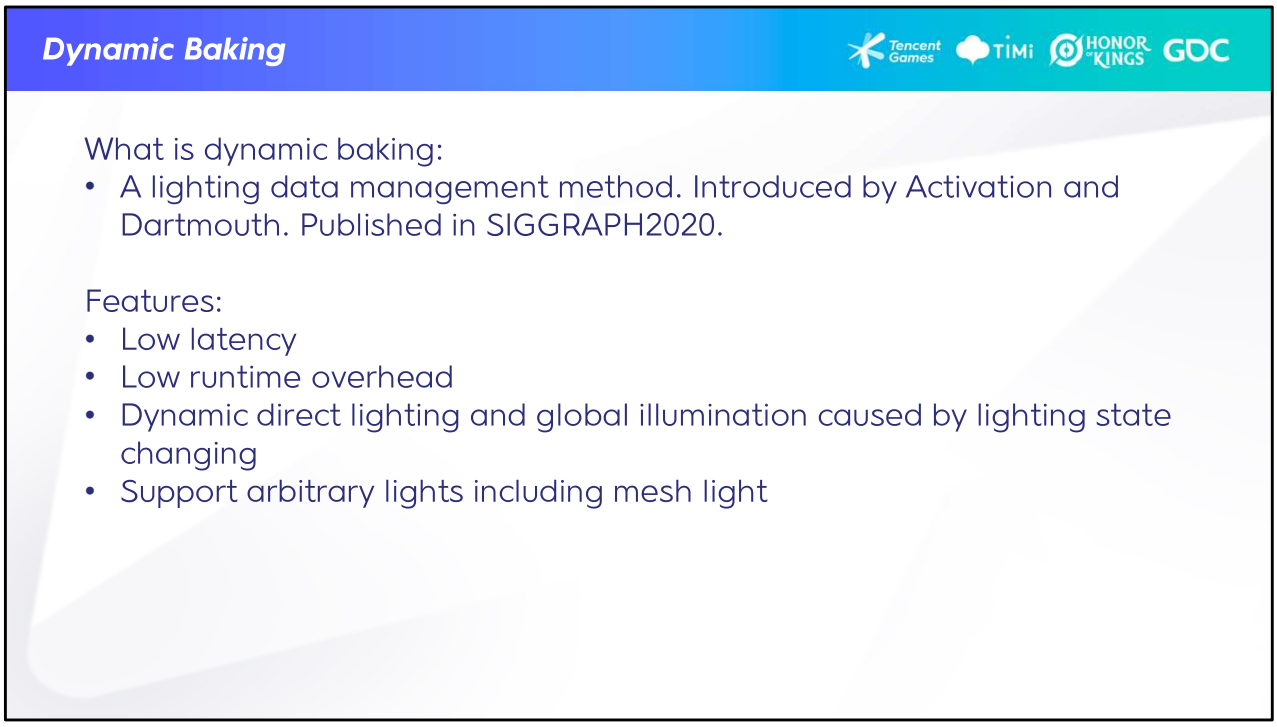
Is boring if we have limited lights and everything is static baked. Can we do better?

With direct lights and static baked lights, we have something that start to looks good. However it's still boring if we have limited lights and everything is static baked? Can we do better?



Yes, we can! with dynamic baking.

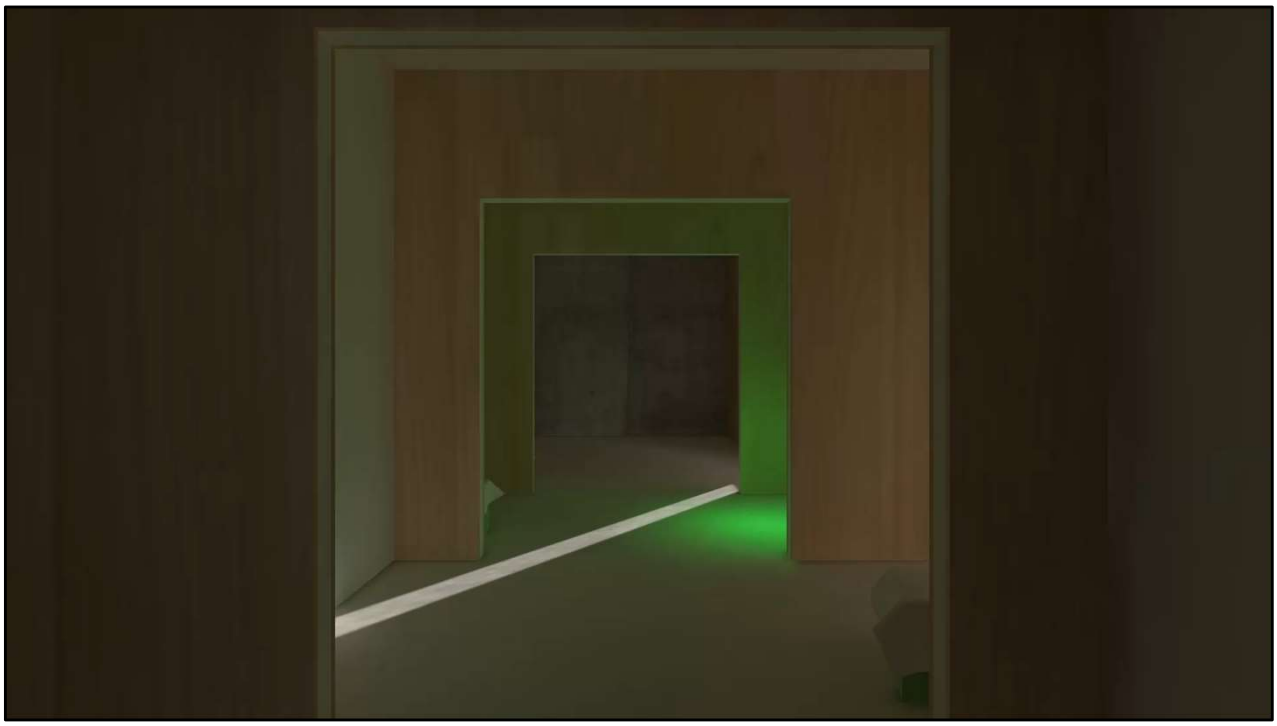
Check out this video to see the impressive results that can be achieved with dynamic baking.



So, what is dynamic baking exactly?

In short, it's a method for managing lighting data that offers
low latency,
low runtime overhead,
and supports dynamic direct lighting and global illumination caused by changing lighting states.

It also supports arbitrary lights, including mesh lights.



Here's a simple example that illustrates how dynamic baking can be used. The following slides will go into more detail about the implementation.

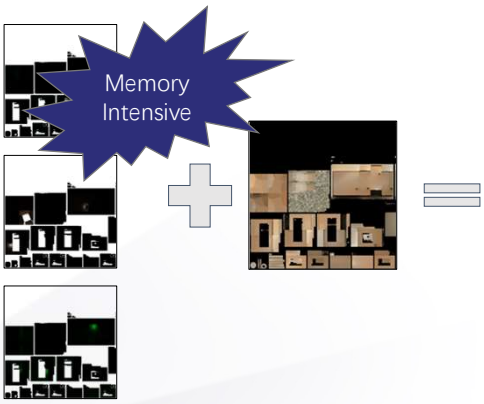
How To

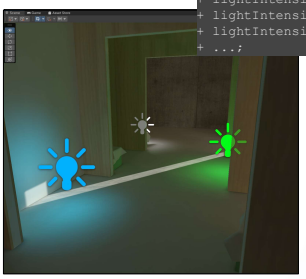
Naïve way: bake two lightmap, sample both of them

- Lightmap count is proportional to dynamic lights

```
vec3 lightmapSampleA = texture2d(lightmap0, uv);
vec3 lightmapSampleB = texture2d(lightmap1, uv);
vec3 lightmapSampleC = texture2d(lightmap2, uv);
vec3 lightmapSampleD = texture2d(lightmap3, uv);
...

vec3 result = lightIntensityA * lightmapSampleA
+ lightIntensityB * lightmapSampleB
+ lightIntensityC * lightmapSampleC
+ lightIntensityD * lightmapSampleD
+ ...;
```

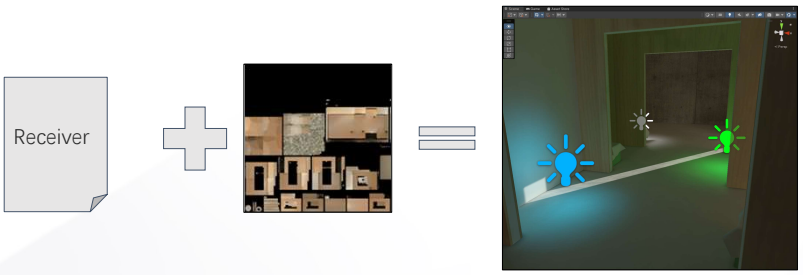




The simplest implementation is to bake separate lightmap for each light, [CLICK] sample both of them, and sum them up at shading time.
[CLICK] However this approach results in a lightmap count that is proportional to the number of dynamic lights.
[CLICK] Which is unacceptable for intensive memory usage.

How To

Need to design an efficient data structure. Which is of
- Low memory usage
- Low runtime overhead



To address this issue, we need to design an efficient data structure which is of [CLICK]:
- Low memory usage
- Low runtime overhead [CLICK]

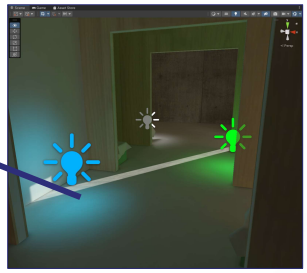
This is where the "receiver" comes into play.

#1: Sparse Texel Storage Per Light

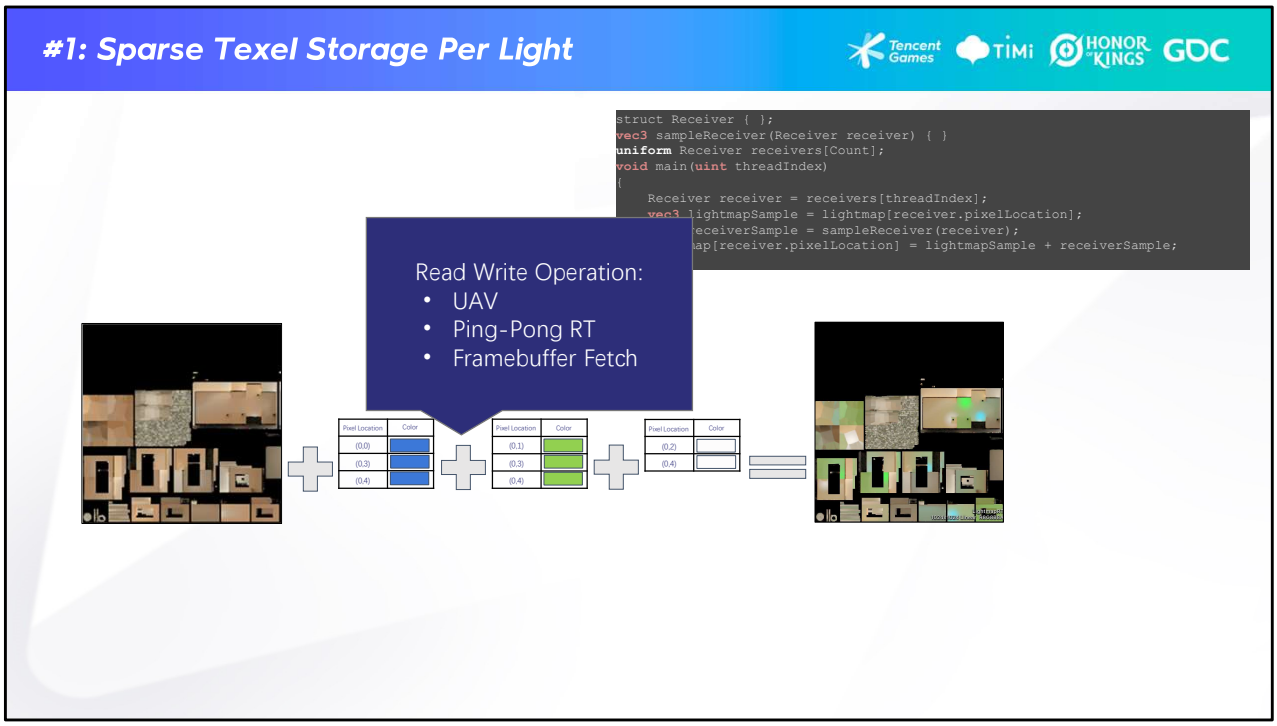
Store a sparse texel buffer for each dynamic light

- Only store useful data

Pixel Location	Color
(0,0)	
(0,1)	
(0,2)	
(1,0)	
(1,1)	



Our initial idea was to store a sparse texel buffer for each dynamic light,, [CLICK] only storing useful data.



Then possible implementation could be a compute kernel.

The problem is, we have to do ReadWrite operation, which is, Reading lightmap texel of current receiver, calculating new lightmap value, then writing it back. [CLICK]

We have three possible options to address this on mobile paltform, 1st is UAV, second is Ping-Pong RT, the third is Framebuffer-fetch.

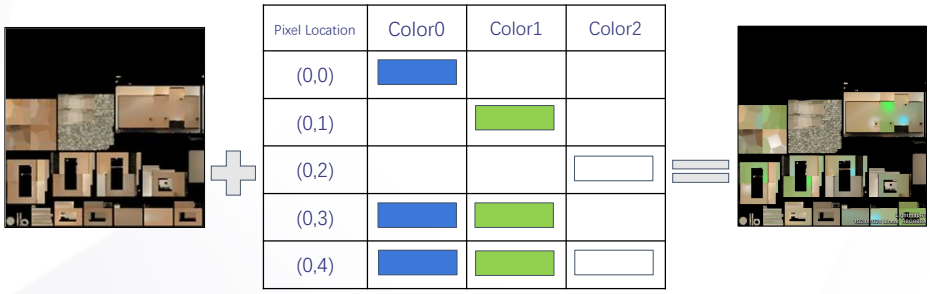
Many mobile device models don’t support UAV and framebuffer fetch very well. Ping-Pong RT is portable enough but need extra memory allocation.

Even all of above issues can be ignored. RW-Operation itself is hazardous for parallel computing, which makes this implementation not practical.

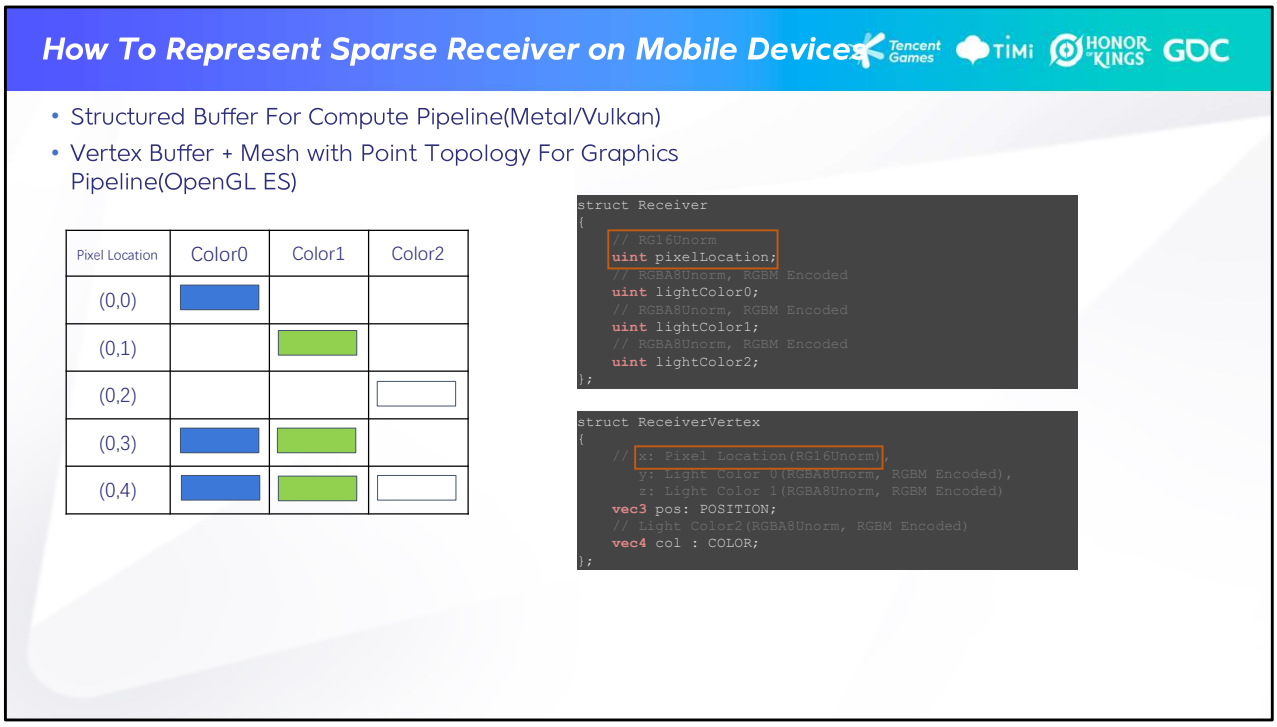
So how can we get rid of RW-Operation?

#2: Sparse Storage Per Texel

Store multiple light info for each texel
- Remove RW operation requirement



Instead of store lightmap info per light, we can actually store multiple lightmap info per texel. [CLICK]
With this data structure, the final lighting result is the accumulation of each weighted lightmap samples, so that we avoid RW operation.



Implementation choice of our data structure is trivial for compute pipeline, just use structured buffer for receivers and UAV for lightmaps.
During compute stage, simply unpack receiver information and write into target pixel location .

But for mobile devices, we have to cover devices without robust compute pipeline support, which is not a negligible portion of device models. [CLICK]

So for those devices, the graphics pipeline is used instead of compute pipeline, so we put our sparse data into vertex buffer with point topology. [CLICK]

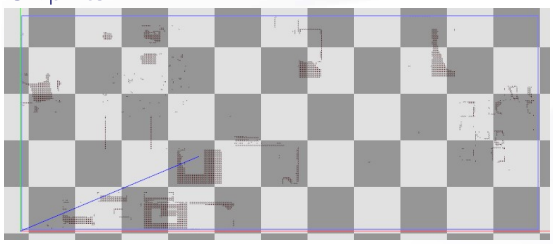
During vertex shader stage, we just unpack the pixel location, and then move the vertex to the pixel location in texture space. [CLICK]

This image is captured in RenderDoc. It is a visualization of vertex stage, we can see many points which correspond to receivers.

How To Represent Sparse Receiver on Mobile Devices

- Structured Buffer For Compute Pipeline(Metal/Vulkan)
- Vertex Buffer + Mesh with Point Topology For Graphics Pipeline(OpenGL ES)

Pixel Location	Color0	Color1	Color2
(0,0)			
(0,1)			
(0,2)			
(0,3)			
(0,4)			

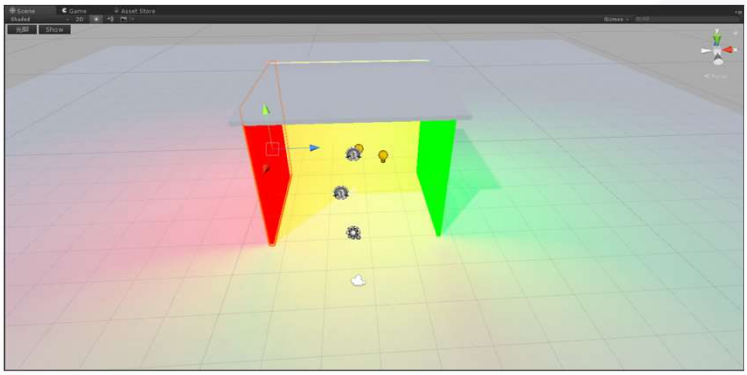


```
struct ReceiverVertex
{
    // x: Pixel Location(RG16Unorm),
       y: Light Color 0(RGBA8Unorm, RGBM Encoded),
       z: Light Color 1(RGBA8Unorm, RGBM Encoded)
    vec3 pos: POSITION;
    // Light Color2(RGBA8Unorm, RGBM Encoded)
    vec4 col : COLOR;
};
```

How To Represent Sparse Receiver on Mobile Devices

The objects can only be influenced by three lights at maximum

- This limitation can be addressed by drawing multiple times with additive blend mode.
- We provide tools for debug visualization.



We limit the maximum number of lights that can affect an object to 3, which gives artists enough lighting flexibility without introducing significant performance degradation.
This limitation can be addressed by drawing multiple times with additive blend mode, but we don't find a use case in our production asset.
We also provide tools for debug visualization to help the artist identify if the maximum number of lights was exceeded.

Distribute Lightmap Updating Over Several Frames

Use different update patterns to distribute computation over several frames:

- Checkerboard: each frame updates ½ texels
- 2x2 Block: each frame updates ¼ texels

Maintain a global light changing rate and choose update patterns based on it.

In order to make the implementation performant enough on all tiers of hardware, we use different update patterns to distribute computation over several frames: they are checkerboard and 2 by 2 blocks.

For the checkerboard, only half of the texels are updated per frame, 2x2 block updates quarter texels per frame.

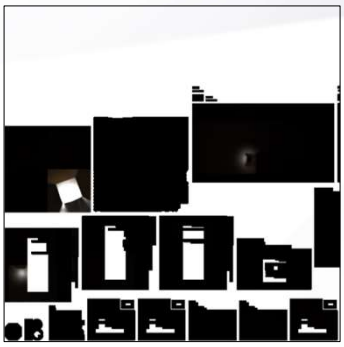
This video shows the 2x2 block update pattern, we use a fixed pattern in this video to emphasis the artifacts. [CLICK]

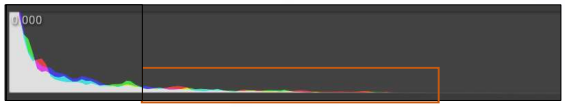
To hide the update pattern, we maintain a global light changing rate and choose update patterns based on it. As you can see, the update pattern is unnoticeable.

Clamp Dark Or Unimportant Texels

There are many dark texels in lightmaps:

- The distribution of irradiance is often a long tail distribution.
- We do not want to store any trivial texels.
- Sort and clamp texels based on luminance.





You may have noticed that some texels in our lightmaps appear dark, [CLICK]
This is because the distribution of irradiance often follows a long-tail pattern. [CLICK]
And storing trivial texels with minor contributions to the final image is unnecessary. [CLICK]
To address this, we sort and clamp the texels based on their luminance.

After Clamping

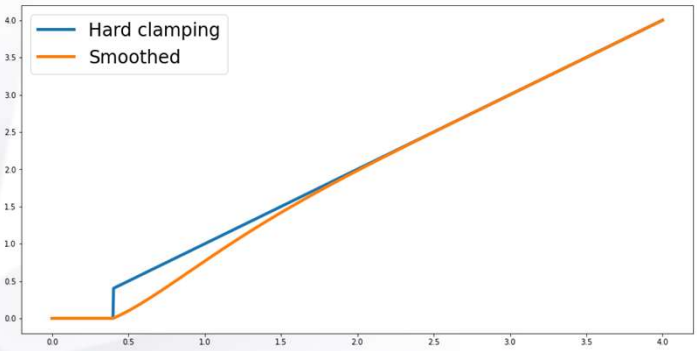
However, clamping the texels introduces discontinuity in our lighting data.

Smooth The Discontinuity

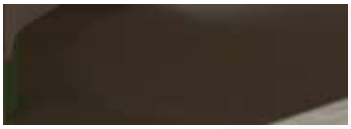
For irradiance $I \in [a, b]$,

$$I_{smooth} = (1 - ((b - I)/(b - a))^4) \times I.$$

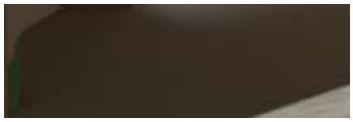
Where a is luminance threshold, b is conservative threshold.



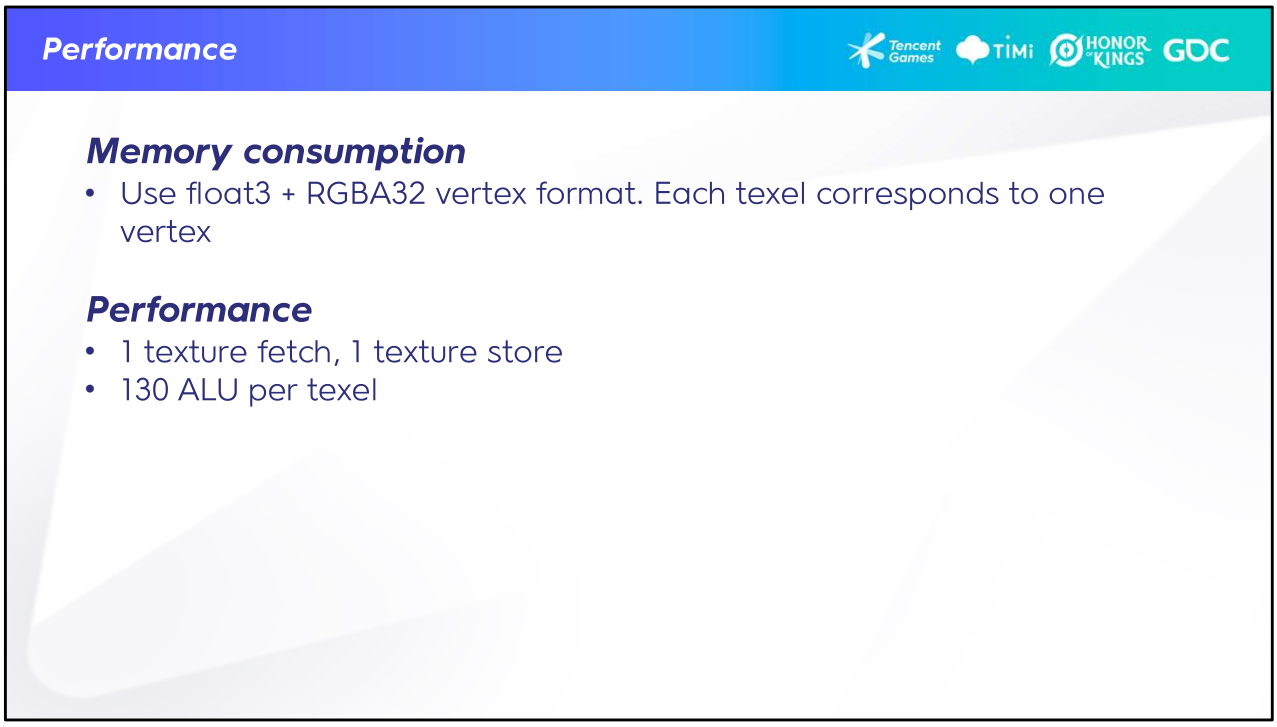
Before Smoothing



After Smoothing

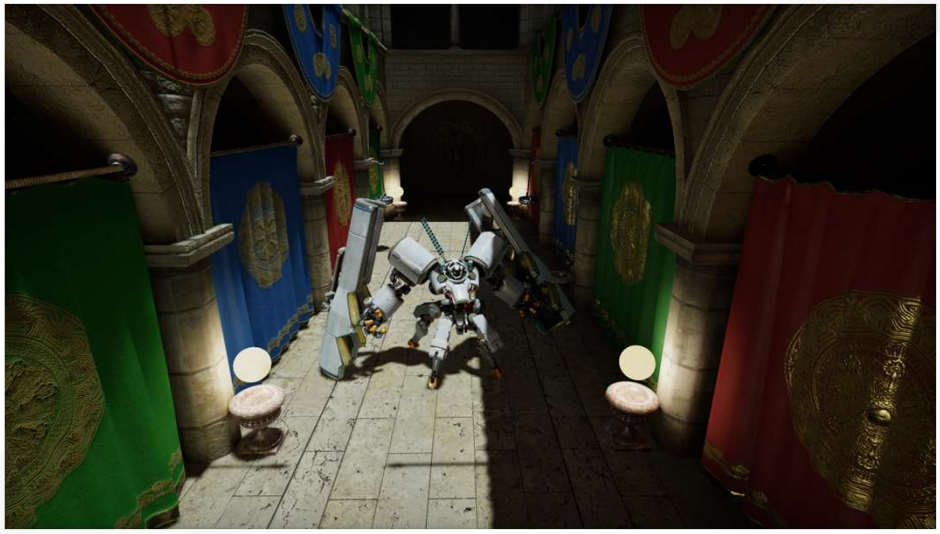


To smooth out the discontinuities, we set the luminance threshold as the lower boundary of the interval we want to smooth.
And we want to keep bright pixels away from any possible blurring.
So we use the conservative threshold as the upper boundary.
Inside the interval, we use an attenuation function to smooth all the values. [CLICK]
Here is an artifact we observed with hard clamping, [CLICK] and here are the improved results after smoothing.



For memory efficiency, we use a float3 + RGBA32 vertex format where each texel corresponds to one vertex.
In terms of performance, we require one texture fetch and one texture store, as well as 130 ALU per texel for lightmap update. The number of ALU is depend on the lightmap encode and decode options and number of dynamic lights.

Direct Lighting + Static Baking + Dynamic Baking

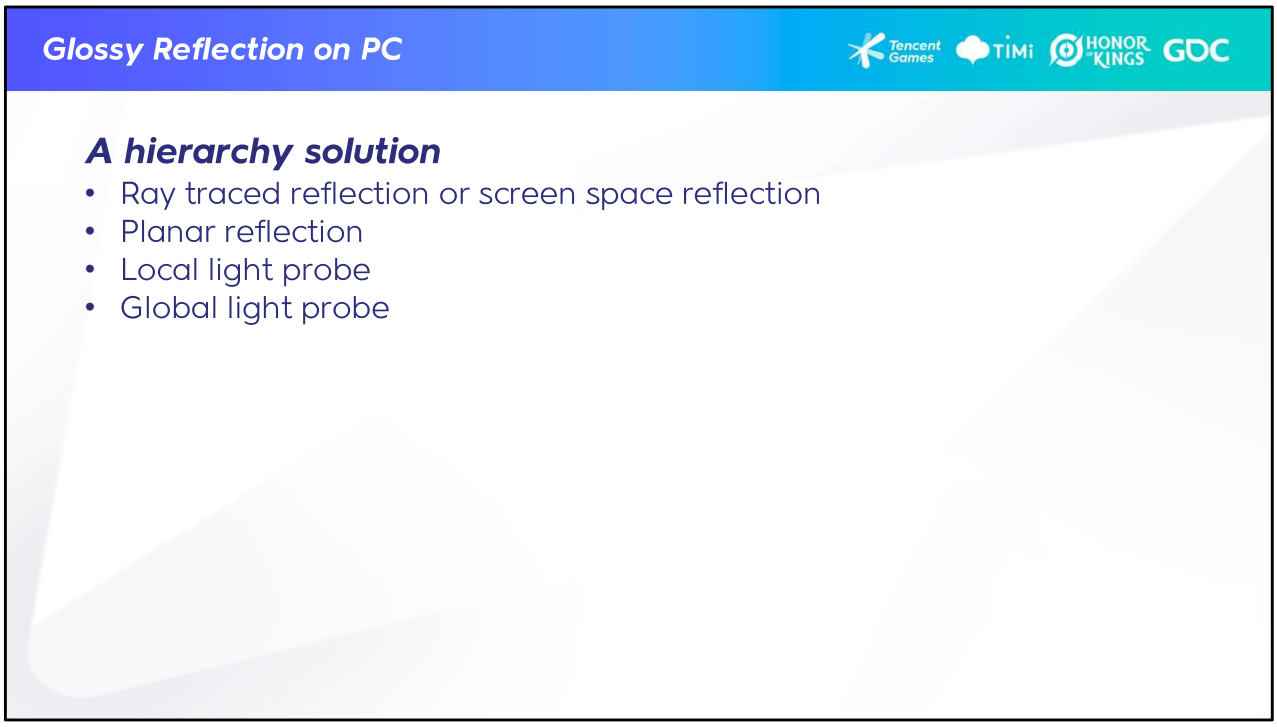


Every pixel is shinny! Add glossy reflection.

With direct lights and baked lights, we are about to complete pixel shading, and the last missing part is glossy reflection. [CLICK]

Glossy reflection is an important element of lighting that adds to the overall realism of the scene.

Glossy Reflection

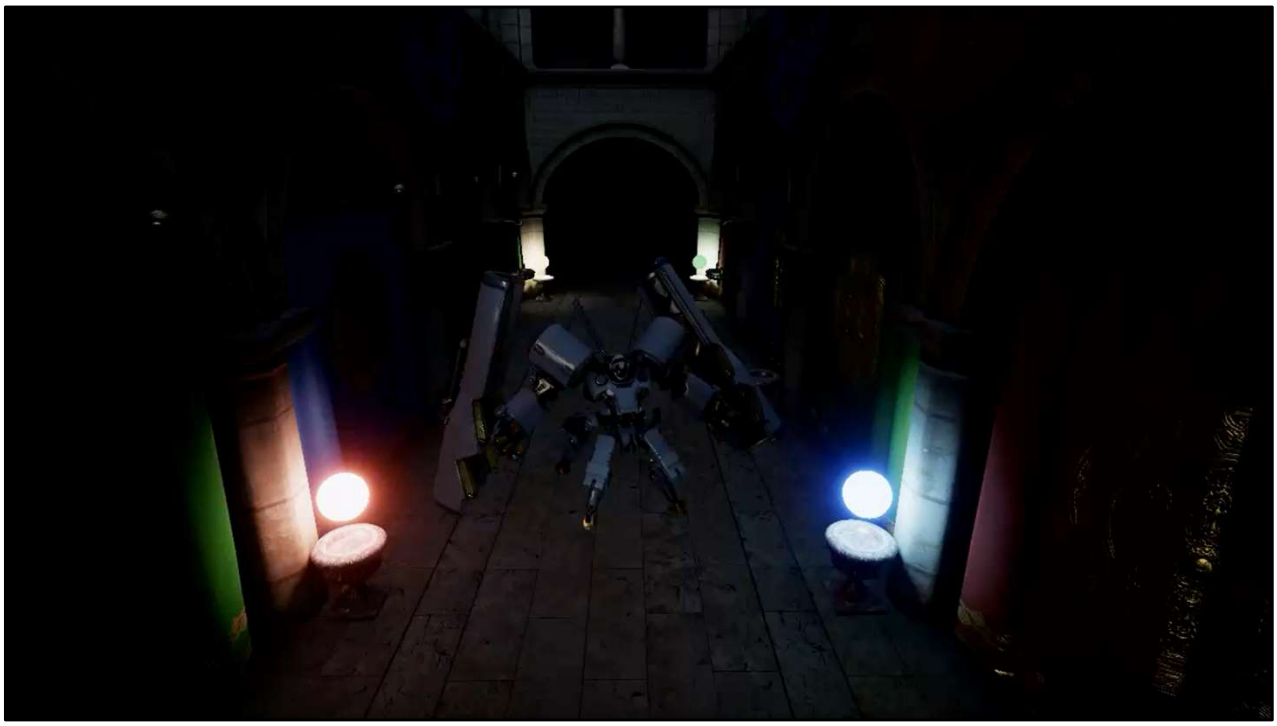


Reflection in PC is implemented through a hierarchy solution, with a first pass of ray traced or screen space reflection. If no intersection is found, it falls back to local light probes or planar reflections, and finally to global reflection probes.

Glossy Reflection on Mobile

- Single reflection probe for both interior and exterior
- Normalized IBL, enhanced by dynamic baking
- Alleviate light leaking by considering specular occlusion, using pre-computed visibility cone

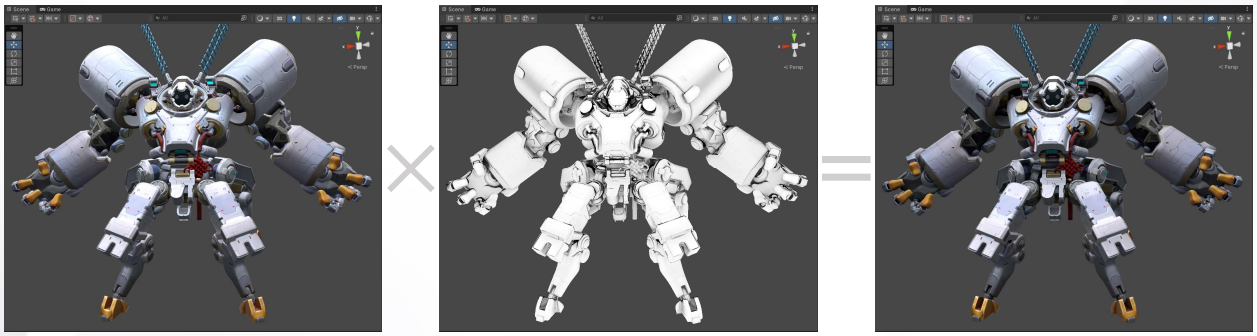
On mobile platforms, due to limited resources, we can only afford to use a single light probe for both interior and exterior lighting.
To make this work, we normalize the image-based lighting (IBL) by average irradiance and rescale it with a correction factor that takes into account the diffuse global illumination at pixel shading.
To further improve visual quality, we also consider lighting variation from dynamic baking.



We have a video to showcase normalized IBL with dynamic baking, which will help to illustrate the process.

Glossy Reflection

Alleviate light leaking by compute specular occlusion from pre-computed visibility cone

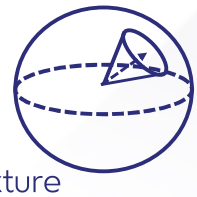


The Split Sum Approximation for image-based lighting can introduce light leaking, as shown in the left image.
To fix this issue, we use specular occlusion, as proposed in the paper "Practical Real-Time Strategies for Accurate Indirect Occlusion."
This involves splitting the lighting into two terms: un-occluded lighting and specular occlusion, which is evaluated from pre-computed visibility cones.
Combining these terms results in much better image quality.

Pre-computed Visibility Cone

What is Pre-computed Visibility Cone

- Idea from “Precomputed Lighting in CoD IW”, Activation
- Geometry visibility information: binary function on spherical domain over mesh surface
- Spherical binary function approximated by a cone
  - Axis
  - Aperture
  - Scale (0~1)
- Data stored either as vertex attribute over mesh or a texture



The idea of pre-computed visibility cone comes from the work of Activation, as published in "Precomputed Lighting in COD".
This technique uses a cone to approximate a visibility function on a spherical domain over a mesh surface.
The cone data includes the cone axis, aperture, and scale, which can be encoded into four floats.
This data can be stored as a vertex attribute over a mesh or as a texture.

How To Compute It

- At each sample on the mesh surface, trace ray for each direction, evaluate distance attenuated visibility function, project it onto SH
- Approximate SH function by a cone with the least square fit, solved analytically. See "Precomputed Lighting in CoD IW" P24

The computation for pre-computed visibility cone is straightforward:
sample points are drawn on the mesh surface,
and for each sample point, a ray is traced in every direction [CLICK].
The distance-attenuated function is then evaluated and projected onto spherical harmonics.
Using the SH coefficients obtained, we can approximate it by a cone with least square fit and solve it analytically [CLICK].

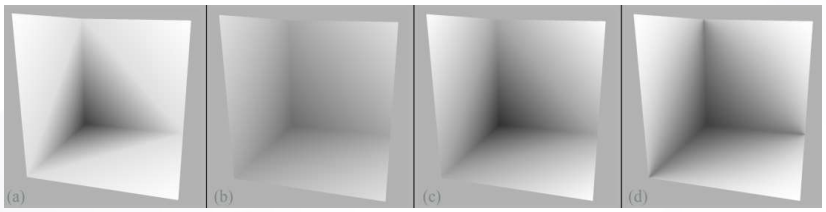
I recommend reading the original presentation for more details.

Visibility cone data can be stored as a vertex attribute over mesh data or as a texture.
For many cases, storing the visibility cone as a vertex attribute over mesh data is preferable due to
its compact storage and
flexible compression format options, such as unsigned normalized 8 bits integer, 16 or 32 bits float, etc.

Additionally, it avoids the need to deal with UV warping.

Visibility cone vertex baking

- Draw coarse samples at mesh surface, compute visibility info
- Least square fits the per-vertex data
- Gradient-based regularization
- Reference implementation from Nvidia: https://github.com/nvpro-samples/optix_prime_baking



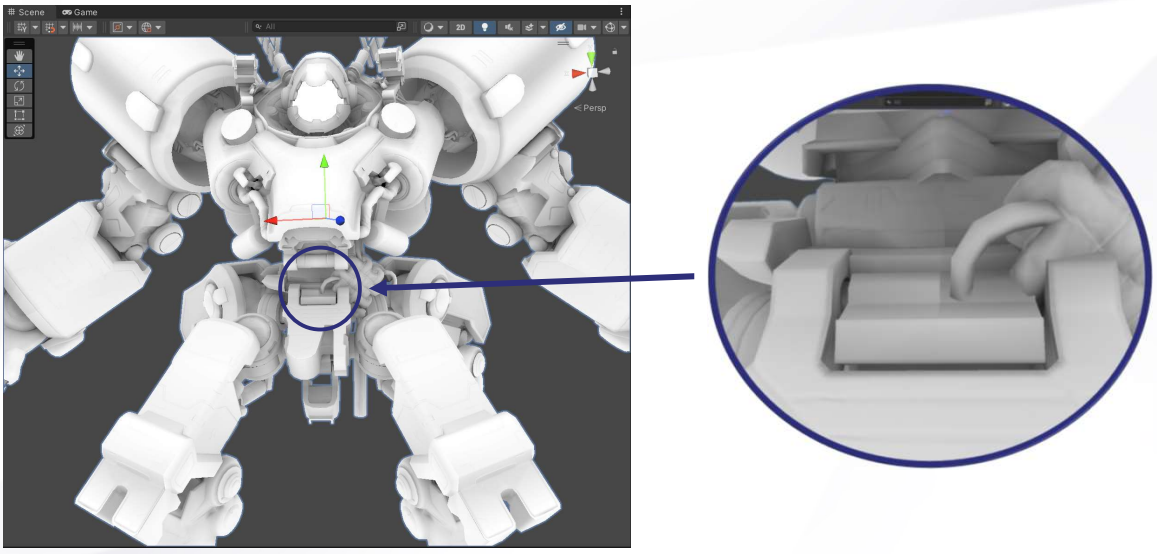
Least Square Vertex Baking

Computing visibility on vertices directly can cause artifacts due to triangle interpolation. Least square vertex baking is a technique used to solve this problem.

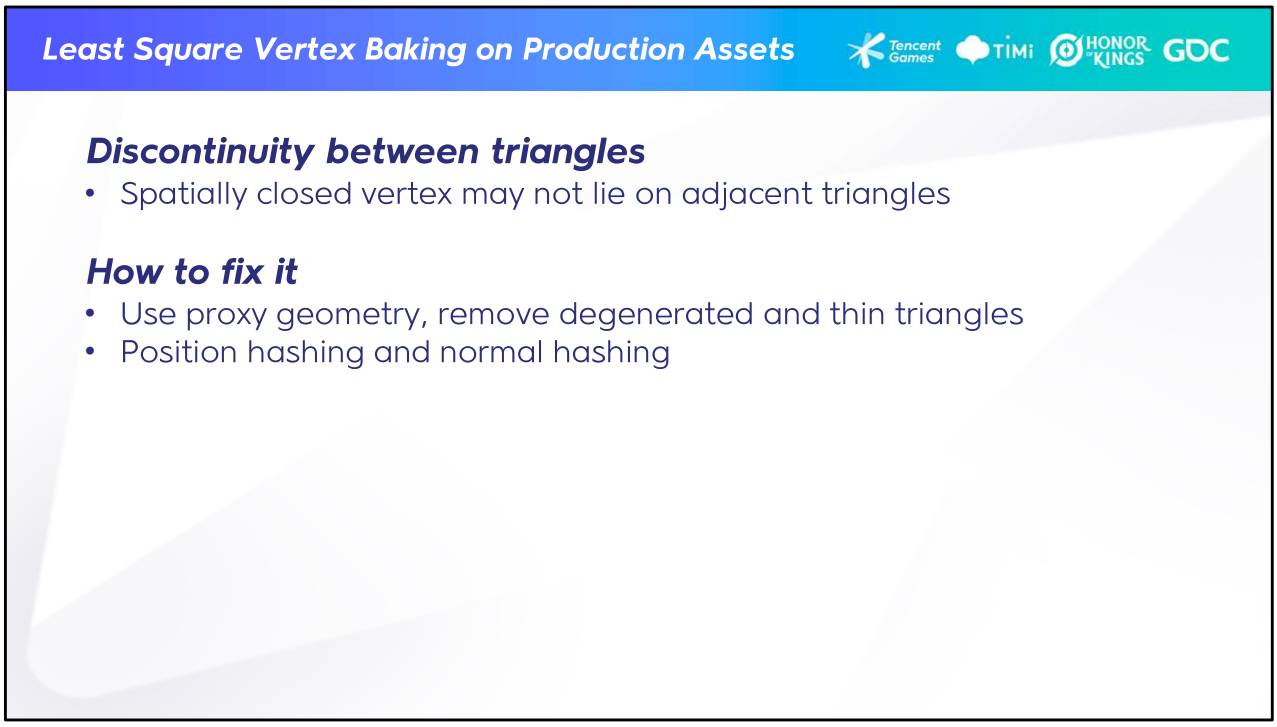
The implementation is straightforward:
coarse samples are drawn uniformly at the mesh surface,
visibility information is computed at each sample point,
then the per-vertex data is obtained by least square fit and gradient-based regularization which means to solve a huge and sparse linear system.

A reference implementation from Nvidia is available. If you are not familiar with the idea, I highly recommend to read the original paper.

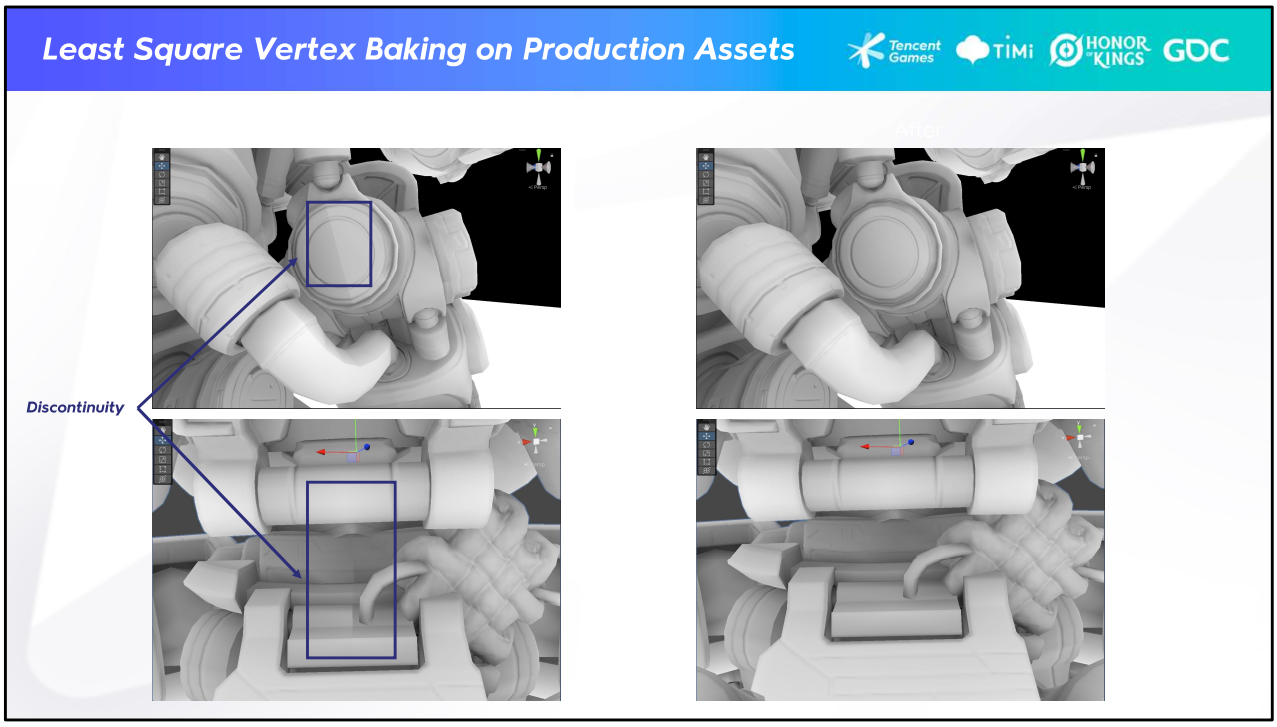
Least Square Vertex Baking on Production Assets



Although the implementation is simple, when we apply it to our production assets, we encountered several issues that were not mentioned in the original paper. The screenshot shows an obvious discontinuity artifact.



The artifact is caused by spatially closed vertices that do not lie on adjacent triangles, leading to a lack of correlation within the linear system.
To fix this issue, we used a proxy mesh to build the linear system and reproject the baked result from the proxy mesh to the original mesh.
To build proxy mesh, we use position hashing and normal hashing to removed degenerated and thin triangles.

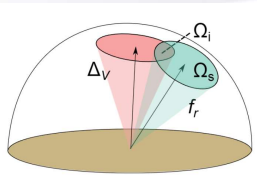


After fixing the discontinuity artifacts using the proxy mesh approach, we were able to eliminate the artifacts.

Specular Occlusion From Visibility Cone

First attempt

- Approximate BRDF by a cone
- Evaluate the intersection of BSDF cone and visibility cor



Pros

- Analytic solution

Cons

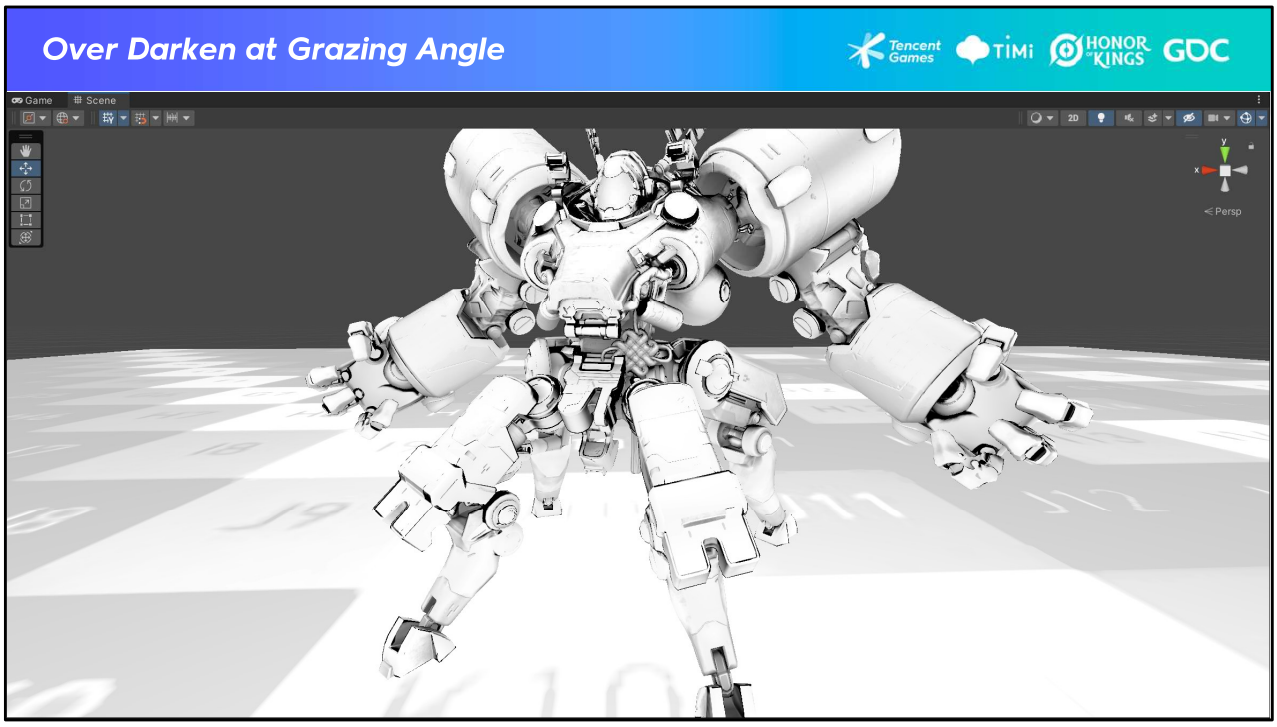
- High cost, around 60 ALU
- Over darken at grazing angle, cone intersection didn't account for horizon clipping

Once visibility cone data is ready, we can compute specular and ambient occlusion from it.

For specular occlusion, our initial attempt was to approximate the BRDF by a cone and evaluate the intersection of the BRDF cone and visibility cone.

However, this approach had poor performance and resulted in over-darkening artifacts at grazing angles.

It required around 60 ALU and did not account for horizon clipping.

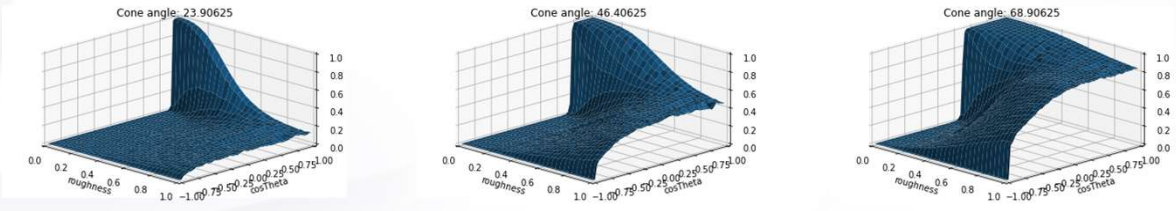


The over-darkening artifact can be seen in this image, especial on the ground plane. We also observed that the artifact depended on the roughness of the material, which made it more obvious.

Specular Occlusion From Visibility Cone

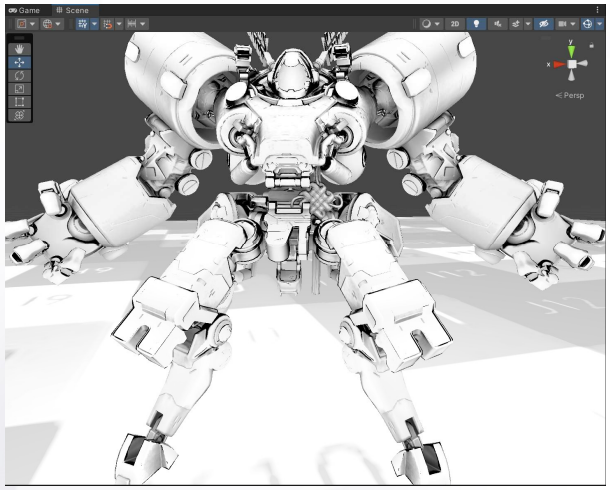
Current solution

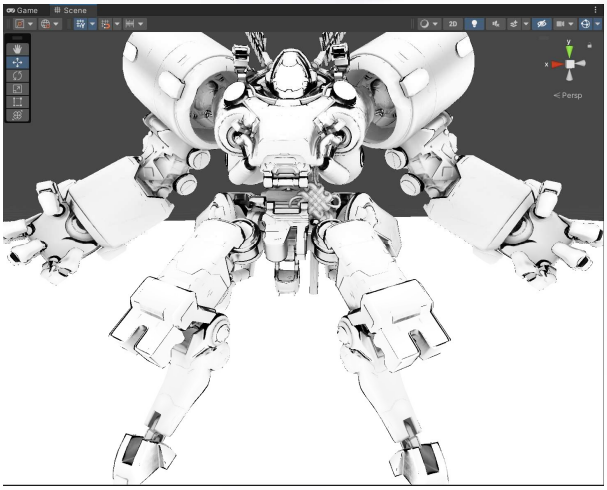
- Use exact BRDF and cone integration, account for BRDF horizon clipping
- Bake specular occlusion into 3D LUT, 16x16x16 works well for us
- Use cone aperture, material roughness, dot(reflection_vector, cone_direction) as parameters
- Future work: numeric fit instead of LUT



Our final solution for specular occlusion was to use a precomputed lookup table (LUT).
Since the LUT was precomputed offline, we were able to use the exact BRDF for integration, which accounted for BRDF horizon clipping.
The LUT was 3D and a size of 16 for each dimension worked well for us.
We indexed the LUT using cone aperture, roughness, and cosine term between the reflection vector and cone direction.
The following plots show how our data looks like, and we believe we can find a numeric fit to optimize it, leave it for future work.

Over Darken at Grazing Angle





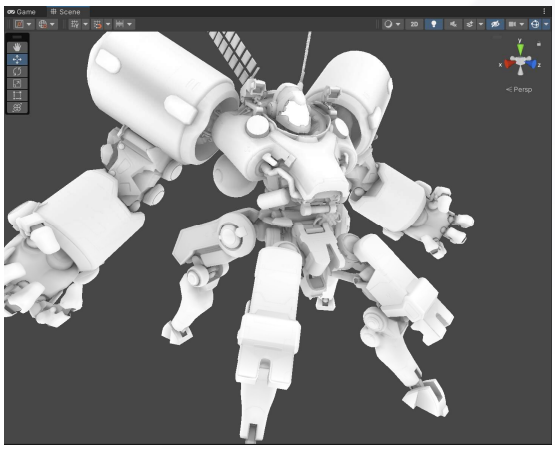
The final result using the precomputed lookup table for specular occlusion is much better compare to previous approach.

Ambient Occlusion From Visibility Cone

- Analytic equation:

$$\mathrm{dot}(\vec{D},\vec{N}) * \left(1 - \frac{1}{1+\tan(a * \pi * 0.5)^2}\right)$$

$\vec{D}$: cone direction
$\vec{N}$: normal vector
a: cone aperture



We can use visibility cone data to compute ambient occlusion.
It's pretty much the same as computing specular occlusion, but this time we use diffuse BRDF and we have an analytic solution.
However, the computation involves a trigonometric function that can be optimized.

Ambient Occlusion From Visibility Cone

- Analytic equation

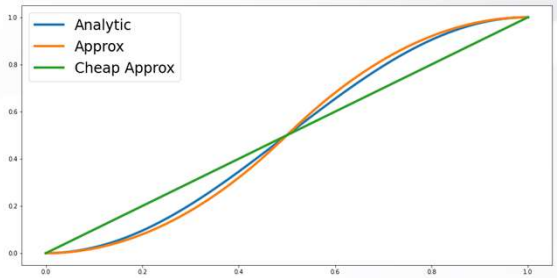
$$ao = \left(1 - \frac{1}{1 + \tan(a * \pi * 0.5)^2}\right)$$

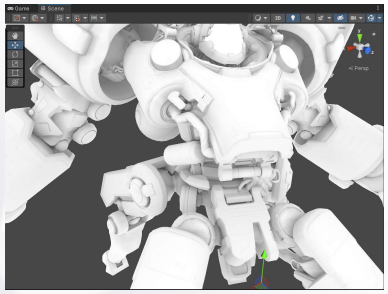
- Approximation

$$ao = \begin{cases} a^2 & ,where\ 0 \le a < 0.5 \\ 1 - (1-a)^2 & ,where\ 0.5 \le a < 1 \end{cases}$$

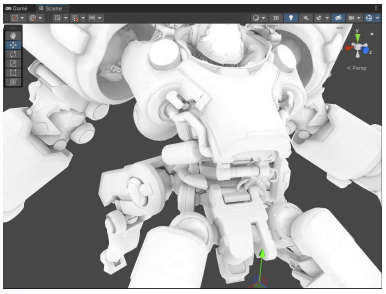
- Cheapest approximation

$$ao = a$$

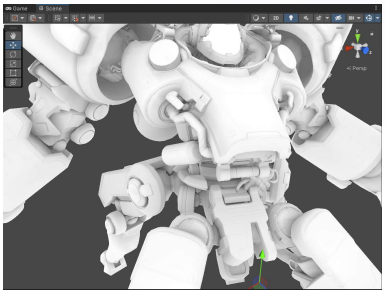




Analytic



Approximation

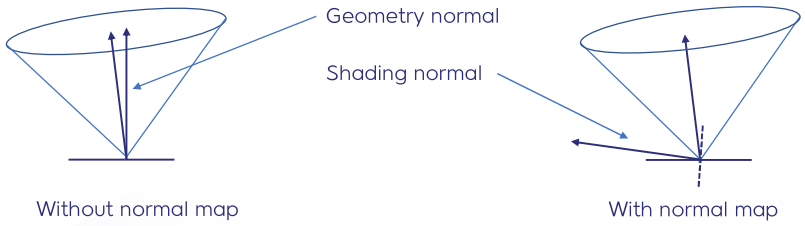


Cheapest approximation

To optimize the computation, we first ignore the cosine term, so we have a 1D function of cone aperture.
We plot it and find that we can use two approximations.
The result is indistinguishable with our approximation and are computationally cheap, and even with the cheapest approximation, the difference is negligible.

Fix Ambient Occlusion with Normal Map

- Problematic to cooperating with normal mapping. Normal map invalidate pre-computed data cause it change the surface geometry. Trick to fix it.



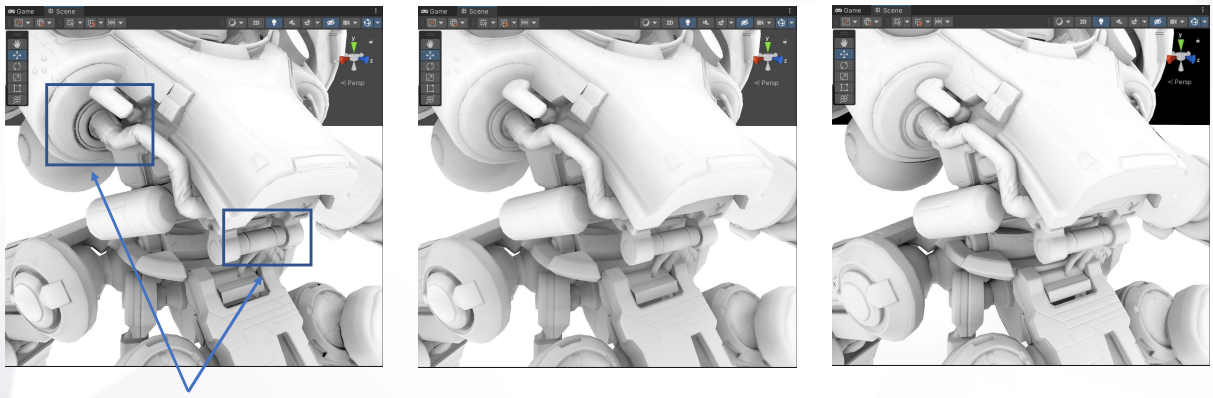

```
half cosTheta = saturate(dot(vCone.direction, normal));
cosTheta = lerp(cosTheta, cosTheta * 0.5 + 0.5, vCone.aperture);
```

We calculate the clamped cosine value between the shading normal and cone direction.
However, the visibility cone was previously computed with the geometry normal. [CLICK]
Simply applying a normal map can invalidate the pre-computed data, as it changes the surface geometry.
Ideally, changing the surface geometry would require recomputing the visibility data, but this is not feasible in real-time.

To address this issue, we use a heuristic-based correction approach. [CLICK]

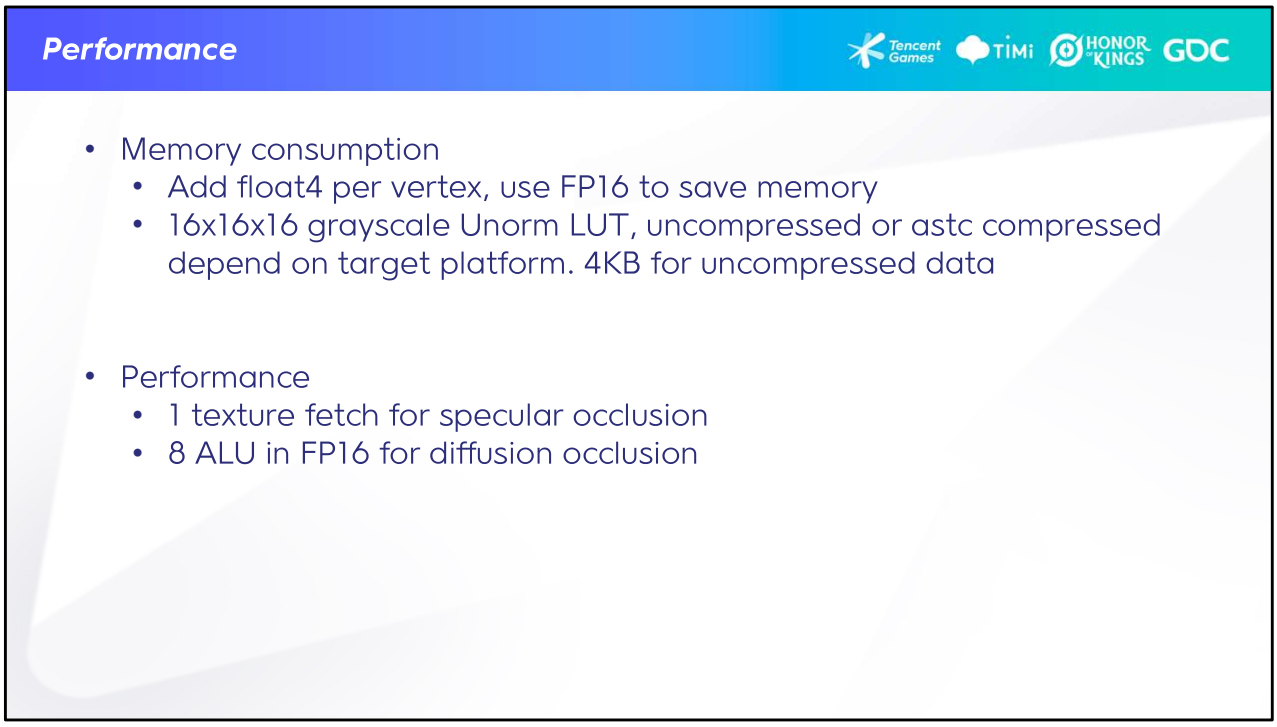
In the worst case scenario, the cone direction aligns with the geometry normal, covering the entire hemisphere, resulting in an ambient occlusion (ao) term value of 1.
However, if the normal map alters the shading normal to make it perpendicular to the cone direction, the ao becomes 0 caused by the cosine term.
If we were to re-compute the cone data and ao term, we would get a value of 0.5.
To account for this, we add a compensation term that is linearly interpolated by cone aperture.
As the cone aperture becomes smaller, this phenomenon becomes less significant.

Fix Ambient Occlusion with Normal Map

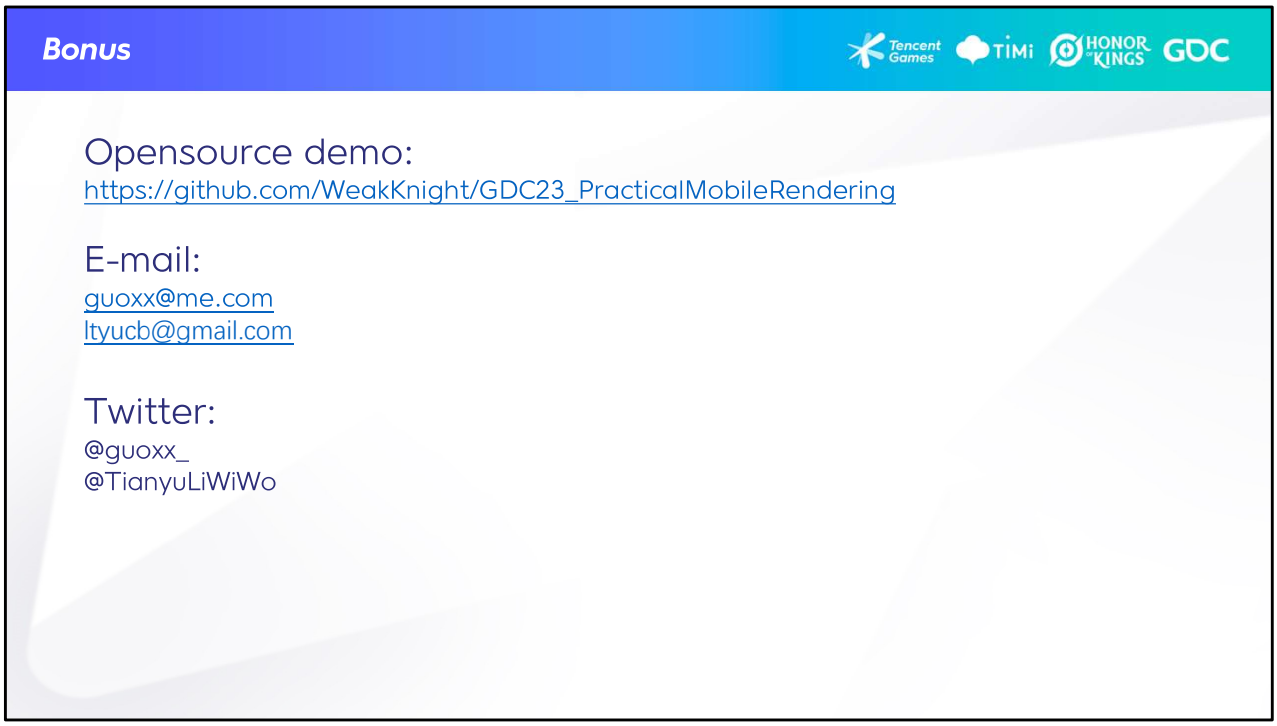


Over darken due to normal map | Heuristic based correction | Ray traced ambient occlusion

The issue of over-darkening artifacts has been fixed with heuristic based correction, and result is closer to ray traced ambient occlusion.



In terms of performance metrics, we considered memory consumption and performance.
For memory consumption, we added a float4 per vertex and used 16 bits float to save memory.
For the 3D grayscale LUT, we could choose between uncompressed or astc compressed data, depending on the target platform. The uncompressed data takes up 4KB.
For performance, we use 1 texture fetch for specular occlusion and 8 ALU in FP16 for ambient occlusion.



Thank you for attending this presentation, and I appreciate your time.
For those who are interested in our technique, we created an open-source demo based on Unity that you can explore. I have included the link here, so you can access the code and play around with it. I hope you enjoy the demo as well as the presentation.
Alright, that's pretty much of it. If you any questions, please feel free to send us email or ping us on Twitter.

References

- Practical Real-Time Strategies for Accurate Indirect Occlusion, Jorge Jimenez, Xian-Chun Wu, Angelo Pesce, Adrian Jarabo.
  https://www.activision.com/cdn/research/s2016_pbs_activision_occlusion.pptx
- The Indirect Lighting Pipeline of 'God of War', Josh Hobson.
  https://www.gdcvault.com/play/1026323/The-Indirect-Lighting-Pipeline-of
- Real Shading in Unreal Engine 4, Brian Karis.
  https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf
- Least Squares Vertex Baking, Ladislav Kavan, Adam W. Bargteil, Peter-Pike Sloan.
  https://users.cs.utah.edu/~ladislav/kavan11least/kavan11least.html
- Precomputed lighting in Call of Duty: Infinite Warfare, Michal Iwanicki, Peter-Pike Sloan.
  https://research.activision.com/publications/archives/precomputed-lighting-in-call-of-dutyinfinite-warfare
- The design and evolution of the UberBake light baking system, Dario Seyb, Peter-Pike Sloan, Ari Silvennoinen, Michał Iwanicki, Wojciech Jarosz.
  https://cs.dartmouth.edu/wjarosz/publications/seyb20uberbake.html
- Sphere ambient occlusion, Inigo Quilez.
  https://iquilezles.org/articles/sphereao/


Thank You
Tencent Games
GDC