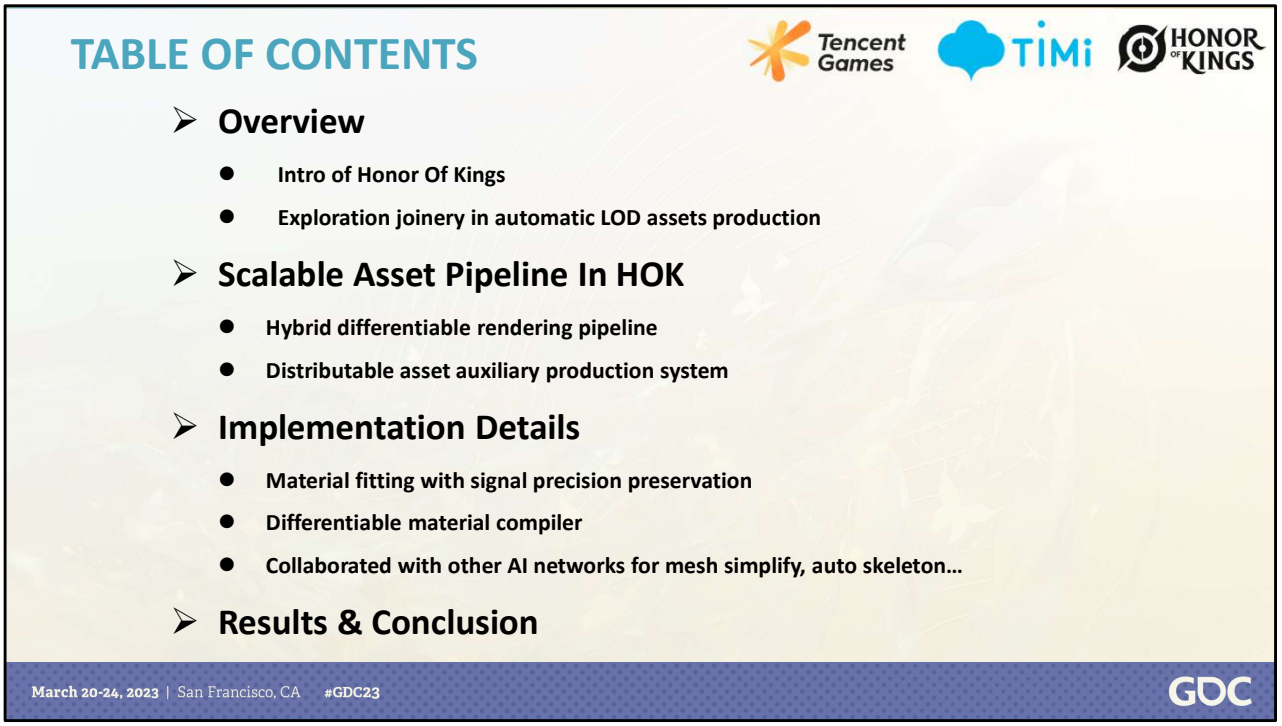
Welcome, everyone! Thank you for joining us. My colleague Frei and I are exciting to be presenting at GDC. Today, we'll be discussing how to use differentiable rendering to build a scalable and intelligent game asset pipeline.

My name is Fei, and I currently working at Timi L1 Studio in Tencent, serves as a principal engine programmer, and today Frei will also be speaking.

We've had success in automatically creating a lot of high-quality LOD assets for low-end devices using this pipeline, and we hope to share some valuable insights with you, especially for who are interested in building their own intelligent game asset pipelines.



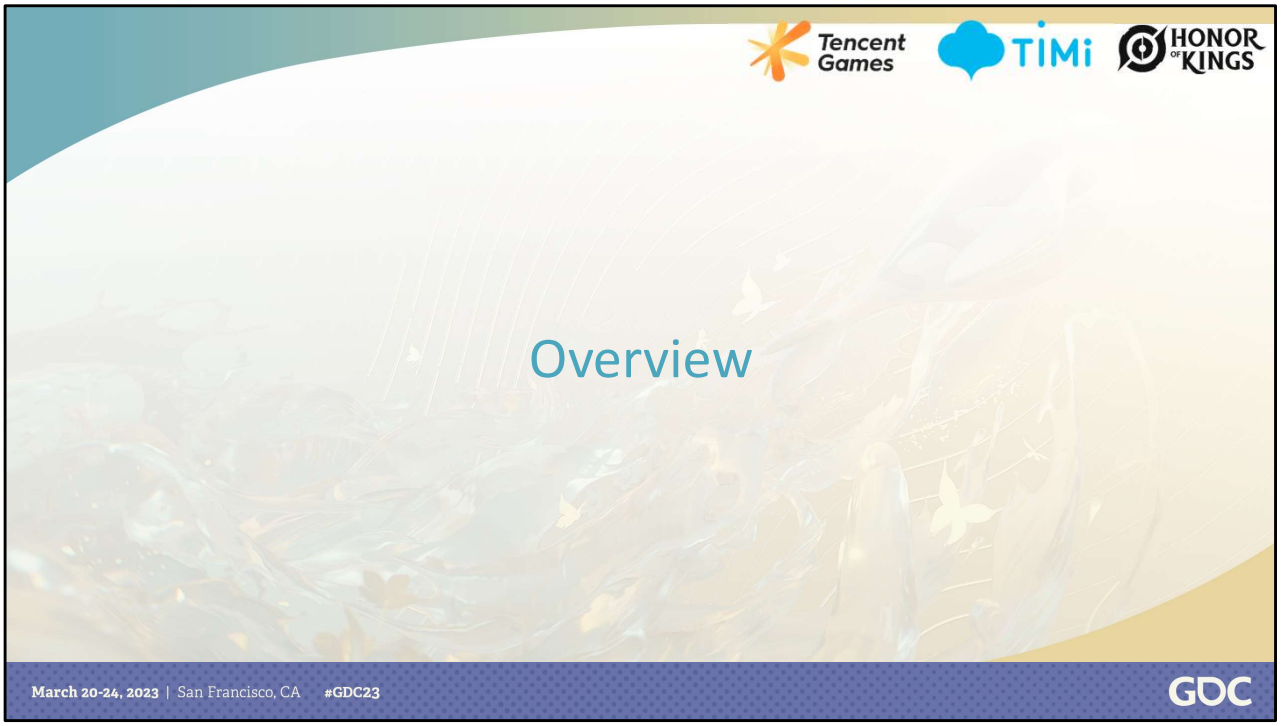
In this presentation, we'll be covering four topics. First, I'll give a quick overview of Honor of Kings, the most played MOBA game in the world.

Then, I'll talk about the production challenges faced by many mobile and multi-terminal games and share our experiences in automating the creation of high-quality LOD assets.

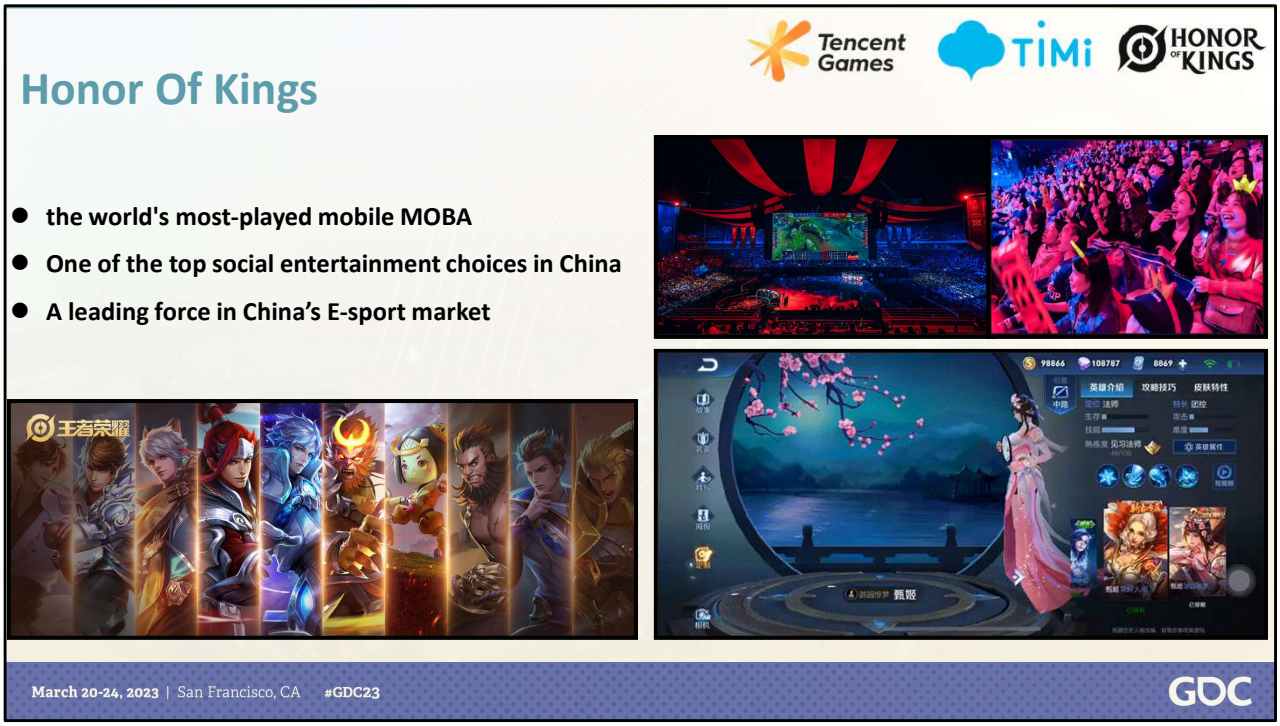
[click]
After that, I will introduce our in-house game asset production system that utilizes differentiable rendering.

[click]
My colleague will then present Implementation details for automatically creating various types of high-quality LOD assets. He will also demonstrate how we use this system in conjunction with other AI techniques to tackle many really difficult tasks.

[click]
Finally, we'll summarize the key points and takeaways from the presentation.



I hope this will be interesting, Let’s move on

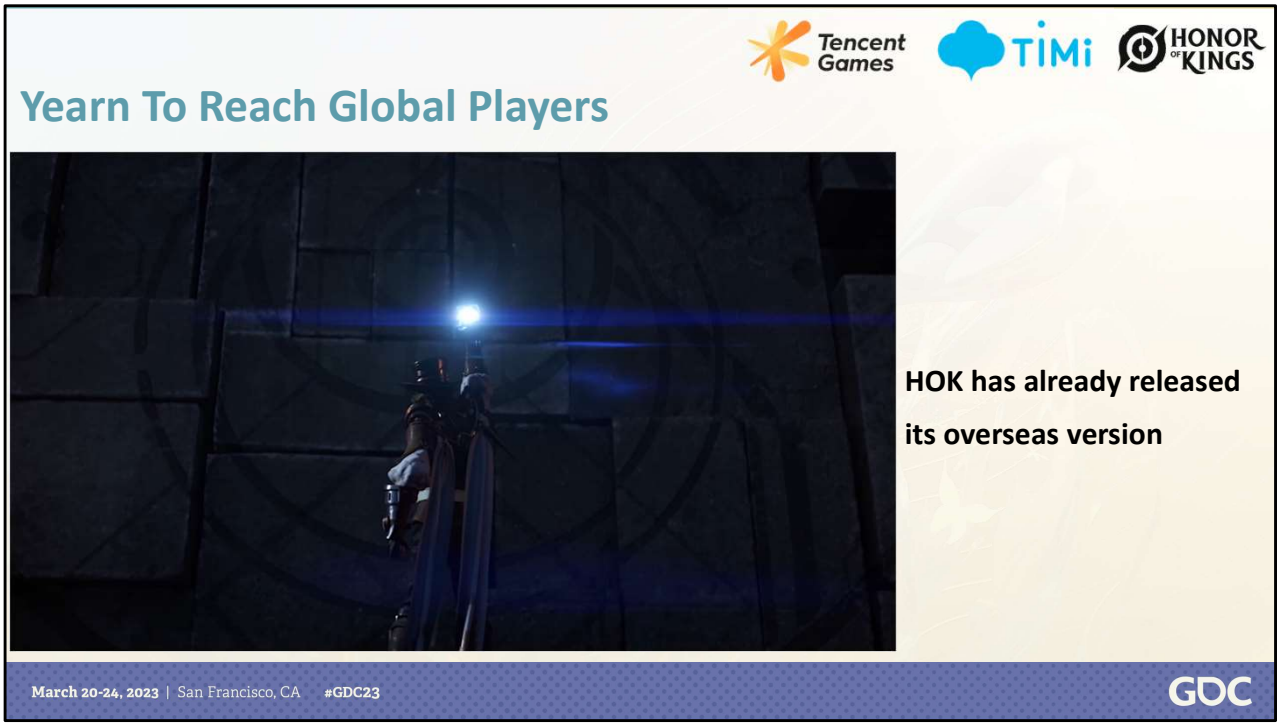


Honor of Kings is the world's most-played mobile MOBA game. Everyone can play their styles, utilize their skills and carry the team to victory!

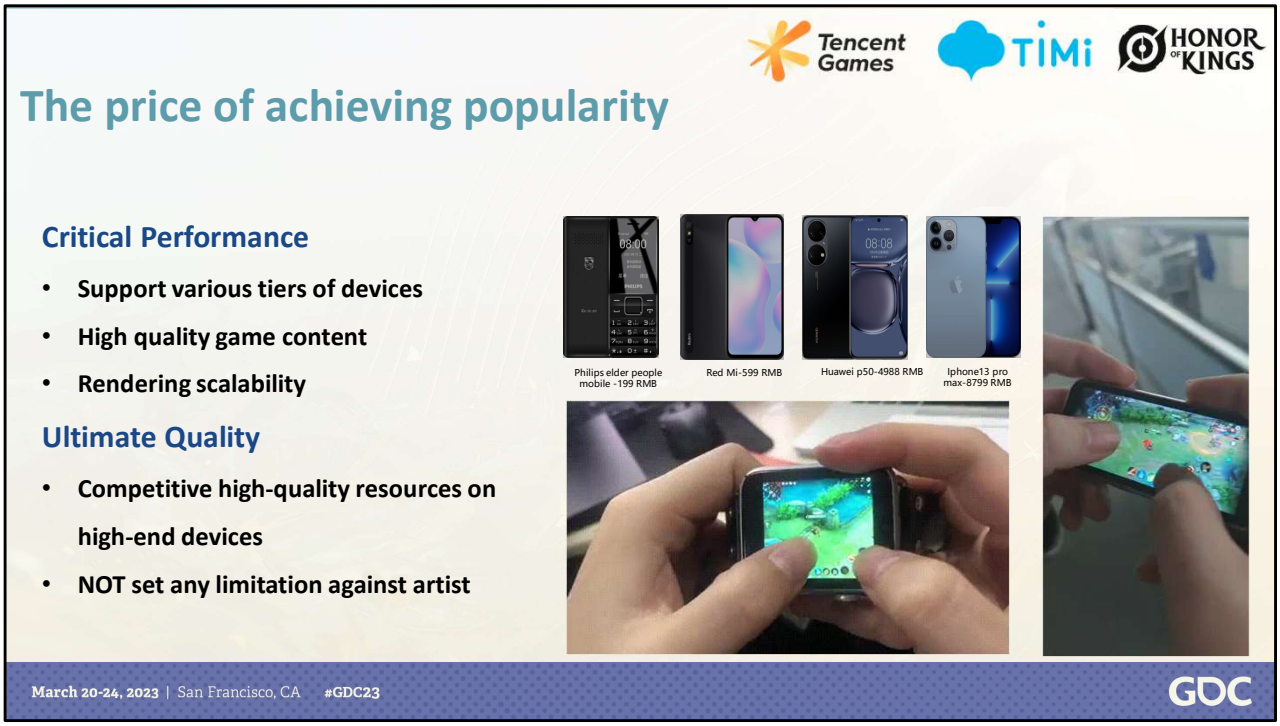
Become immersed in the battlefield as you squad up with your friends, choose from unique heroes with amazing skills, and enjoy the extreme fun of fierce team fights.

In 2015, Honor of Kings was released by TiMi Studio Group in China. After years of dedicated work in character designs, worldview narratives, and

gameplay upgrades, the game has become the top social entertainment choice in China, recorded 100 million average daily active users in 2020.



Currently, we are eager to share our amazing game with international players and friends. HOK has already released its overseas version in Brazil. we are working hard to accelerate its globalization process. To conclude my introduction of HOK, I will show you guys a promotional video for one minute.



To reach a wider audience, it is important that Honor of Kings must be available on a variety of devices, including low-end ones such as older smartphones and [click]smartwatches.

While it might not be practical to play a MOBA game on a smartwatch, it's still important to make sure the game performs well and has high-quality contents on these devices.

Low-end devices often have limited memory, processing power, and graphics capabilities, which can make it hard to maintain good performance and high-quality content.

[click]
At the same time, high-end devices have more powerful hardware and usually want to use more complex and detailed rendering effects with higher-quality resources.

Doesn’t those seem contradictory? Of course, so, in order to give players the best possible experience on all kinds of devices, it's common in game development to use different levels of detail (LOD) game assets. This helps to balance performance and quality across devices with different capabilities.

LODs in Honor of Kings

Fitting PBR To Unlit

Mesh Simplification Addition With Auto Skeleton

Practices in material simplification, mesh simplification and auto skeleton for HOK

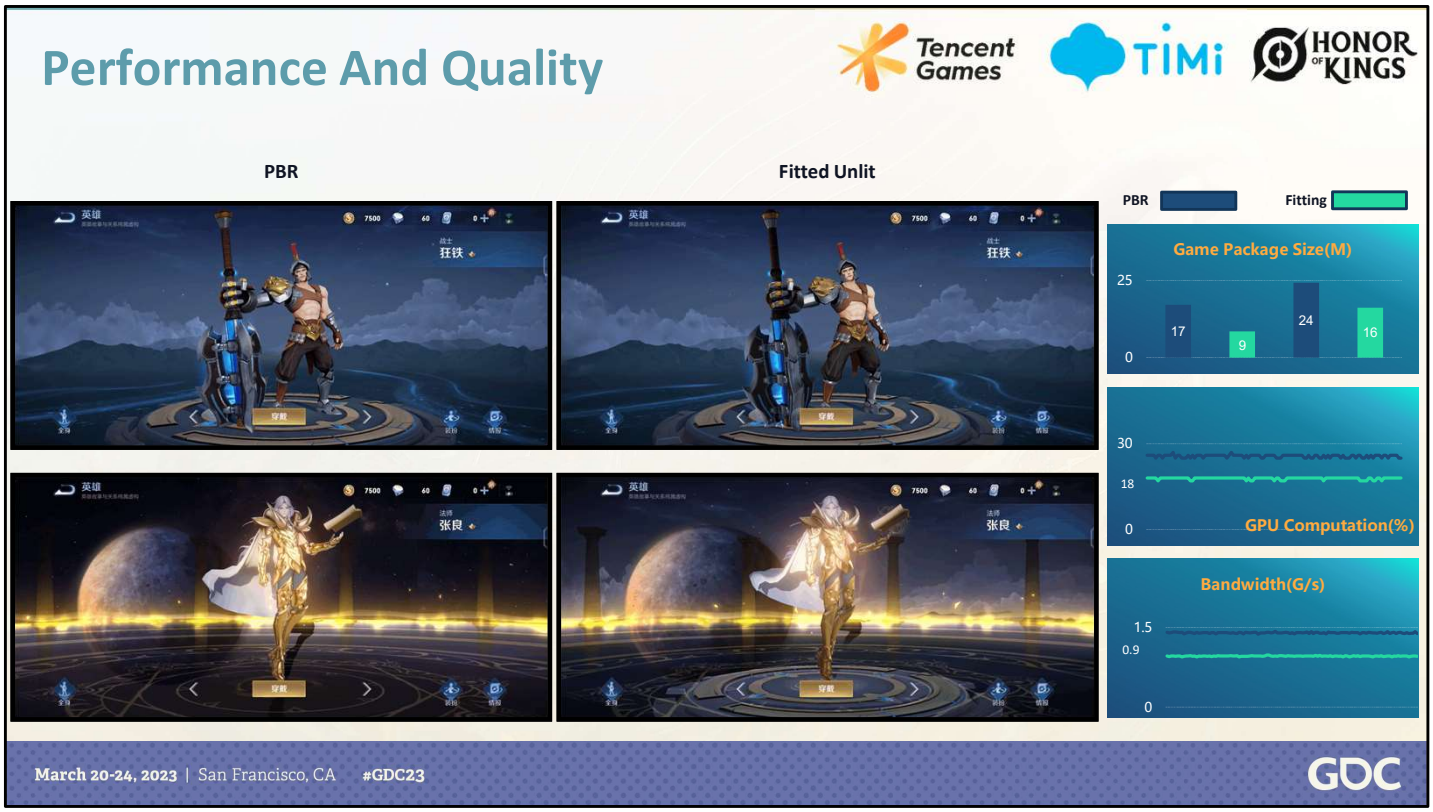
March 20-24, 2023 | San Francisco, CA #GDC23

In Honor of Kings, high-quality LOD game assets are extensively used, but the difference is many of them are generated automatically.

For material LOD, we automatically fitted the complete physically-based rendering (PBR) shading model to a single unlit texture.

[click]
Additionally, we utilized high-quality, auto-generated, simplified LOD meshes for the "hero moment" on some low-end devices.

[click]
To generate high-quality skeletons for those simplified meshes, we also utilized a highly optimized auto skinning AI network with the help of differentiable rendering.



After using these generated LOD assets widely, we observes a significant performance boost on various mobile devices.

For instance, even on our low-end baseline device with only a 1.3 GHz quad-core processor and 1 GB of RAM, our game runs smoothly and without any noticeable loss in graphics quality.

Using these LOD assets can also reduce the bandwidth usage, leading to lower heat production and decreased power consumption.

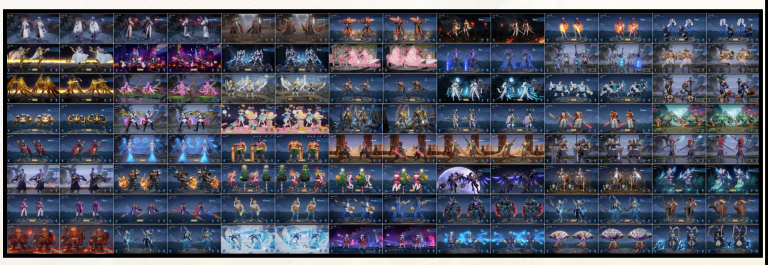
It can also help to reduce the size of the game package, making this technique even more beneficial.

Obstacles to traditional LOD pipeline

Employ LOD assets works fine but still suffers many drawbacks

Productive Issues

- **Artists want to work only with one suit of assets**
- **High costs in LODs' production**

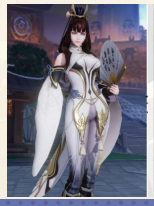


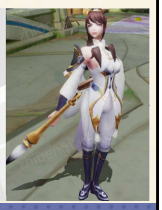
Quality Issues

- **Measuring LOD differences is difficult**
- **Outsourcing teams sometimes have negative impacts**

Automated?

Align In Quality?



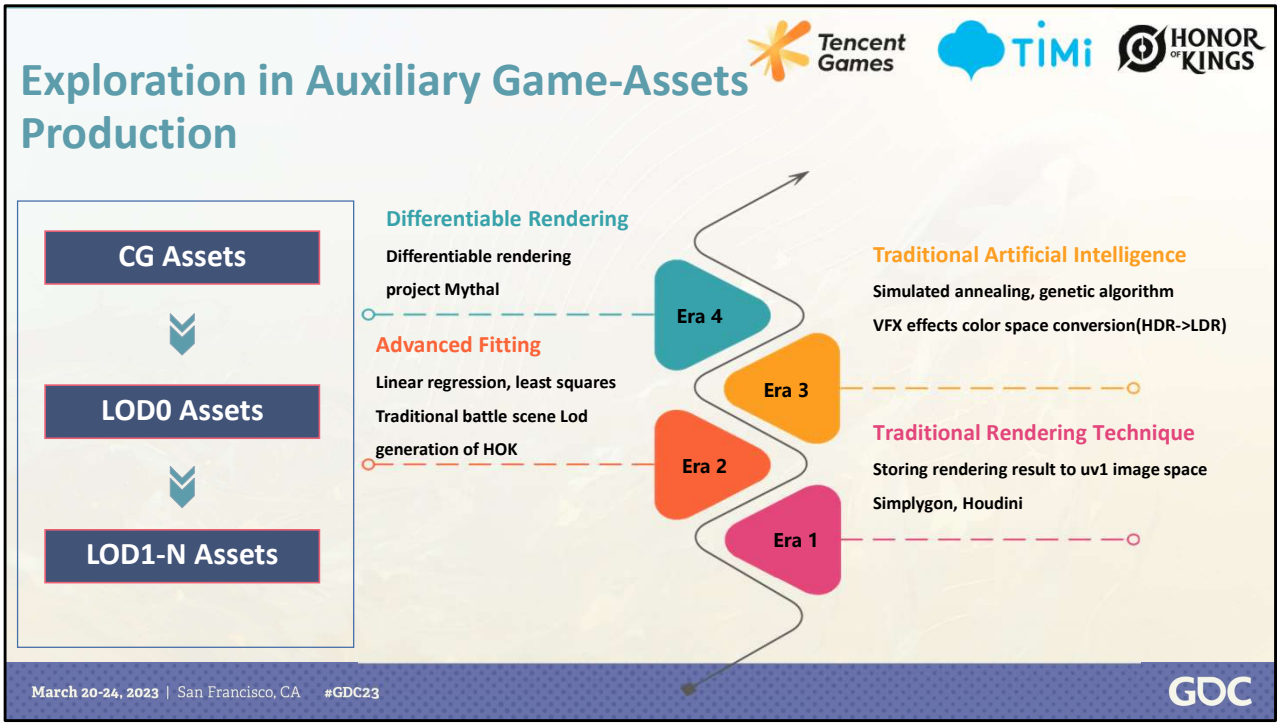


High quality LOD assets are undoubtedly useful, but a traditional LOD pipeline in game development will be facing many challenges due to the complexity.

Yes, it is complex of creating and managing multiple versions of each asset. This process can be particularly time-consuming and resource-intensive, especially for large and complex games.

It can also be hard to keep track of and apply changes to all the different versions of assets.

[click]
Additionally, using artist judgement will also lead to quality inconsistencies. It will be frustrating for both players and developers. These challenges can be especially tough to deal with, as they require a lot of effort and resources to resolve.



Our team has been working to develop a system that can automatically create high-quality LOD assets.

This envisioned intelligent asset production pipeline would allow artists free to create the highest-quality assets without being limited by performance. It is similar to how they working with CG assets.

When it comes to performance, by providing the system with our performance budget, it will automatically produce the best LOD assets that meet standards.

[click]
We have experimented with various algorithms and strategies, both common and lesser-known, to make this vision a reality. I will share a breif retrospective on these efforts during following pages

Brute Force Baking

Auto Generated Battle Scene LOD

- Lack of error quantification
- Lost crucial metal material expressiveness in PBR
- Discontinuity in uv1 image space lead SEAMS

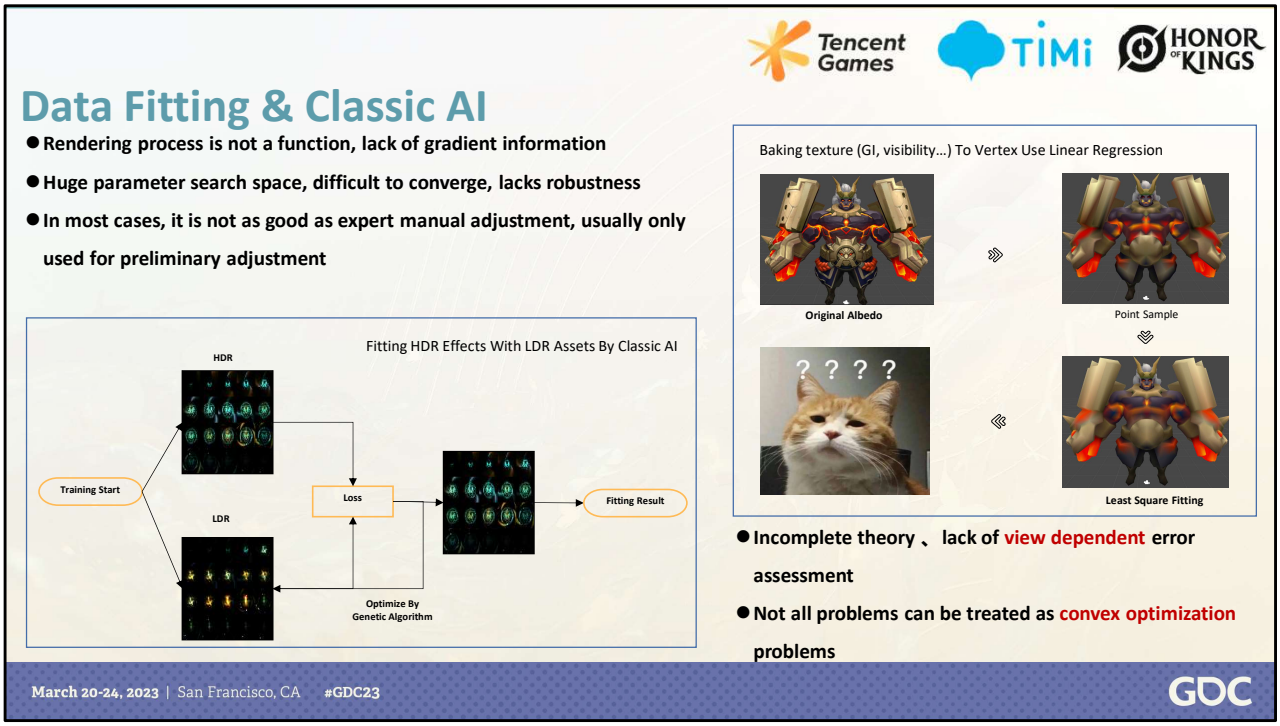
- Lack of robustness, strongly related to specific scenes
- Requiring post-modification by art experts
- Unfriendly iteration

Auto Generated LOD in "Hero Show"

In our initial experiment, we created textures by baking PBR materials into texture space with a fixed viewpoint, and used them directly as LOD assets. While this method is simple and can be useful in some fixed view scenes, it obviously has several drawbacks.

[click]
These include a lack of error quantification, decreased expressiveness, and discontinuities in the UV1 image space that are difficult to fix.

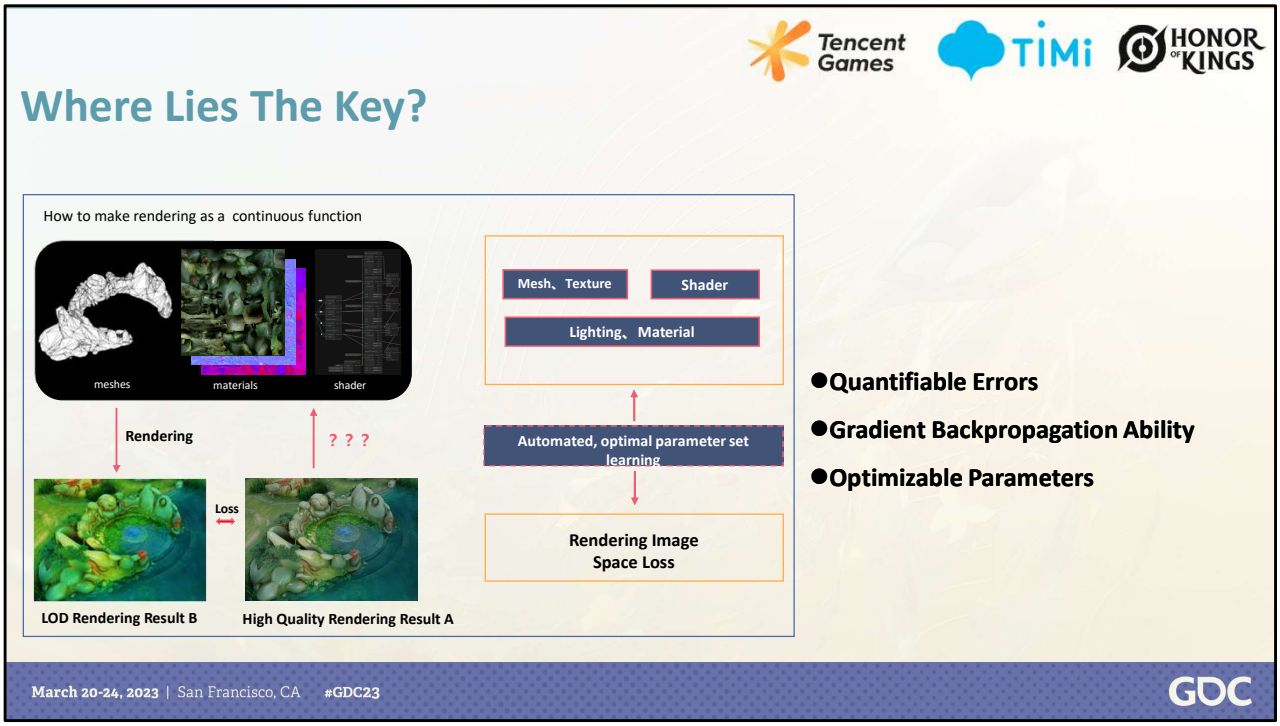
[click]
Additionally, this method not suitable for a wide range of scenes and requires extensive post-modification by experts, leading to an inefficient iteration process.



Linear regression models like least squares can improve the error quantification in baking process, but they also have some sever limitations.

For example, they don't have a complete theory and can't account for view-dependent errors. They were not work for all problems neither, as not all problems can be reduced to convex optimizations.

[click]
Classic AI approaches like genetic algorithms haven't been very effective for rendering tasks. Because they usually rely on searching through a large space of parameters during rendering, which are very slow and impractical.



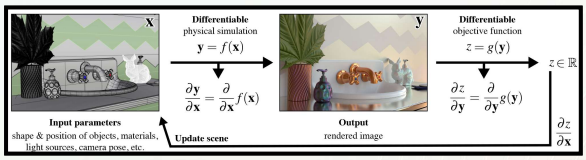
Let's take a moment to pause here and briefly summarize what has been said so far.

We are looking for a method. It can guide our optimization efforts through quantifiable errors in different views.

This method should be gradient-based, allowing for quick and stable optimization of non-convex problems such as rendering. It should also be applicable to the original parameter sets in order to optimize them.

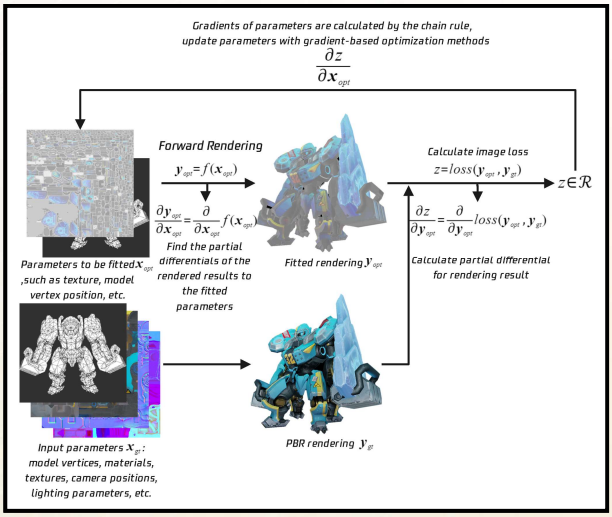
In a word, Our ultimate goal is to find a method that is effective and efficient in optimizing during the rendering process.

Differentiable Rendering



"Physics-Based Differentiable Rendering: From Theory To Implementation"

- Treat rendering process as a function
- Focus on the function's continuous
  - Shading(Rasterize or Monte Carlo Method)
  - Visibility
  - Geometry
- Back propagate gradient though different space
- Optimizer(Adam,,,)



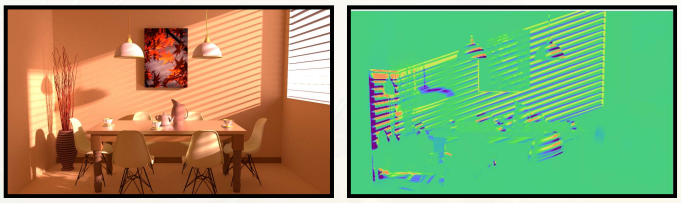
Now, its time to mention differentiable rendering. It is such a technique that optimizes rendering processes using gradient-based methods.

It differentiates the rendering process based on parameters such as object color, reflectivity, and texture, as well as ensures continuity in shading, visibility, and geometry.

This enables the optimization of these parameters to improve the efficiency and effectiveness of the fitting tasks during rendering.

Greatly Inspired By Pioneers

Mitsuba 2/3



"Path-Space Differentiable Rendering"

NV Diffrast(Modeling)

There already existed a lot of genius researches in DR. Some important factors we need to take into account:

- Accurate
- Global Rendering Effects
- Performance
- Noise

Nowadays, differentiable rendering is a popular technique for optimizing rendering tasks in various applications. Our work has been inspired by many pioneering works in this field.

But In practical terms, there are several important factors to be carefully considered, particularly in game asset productions.

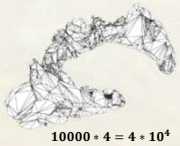
[click]
First, we need an accurate rendering process that supports global effects and maintains high performance. Specifically, methods lack of accurate are not suitable for our needs, such as PyTorch 3D.

Additionally, we must carefully incorporate with biases to quickly achieve noise-free rendering results. Noise-free is an essential prerequisite for gradient propagation passes. These factors are important for optimizing the fitting tasks based on rendering.

Obstacles In Practice

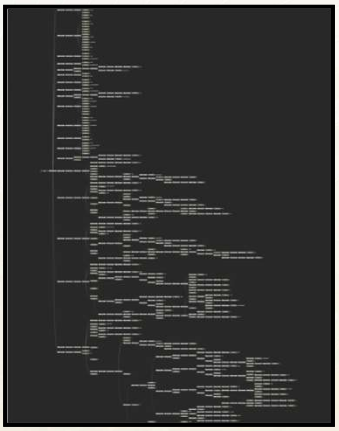
Huge Number Of Parameters

$3 * 1024 * 1024 * 3 = 9.4 * 10^6$



$10000 * 4 = 4 * 10^4$

Huge Computation Graph



Other Issues

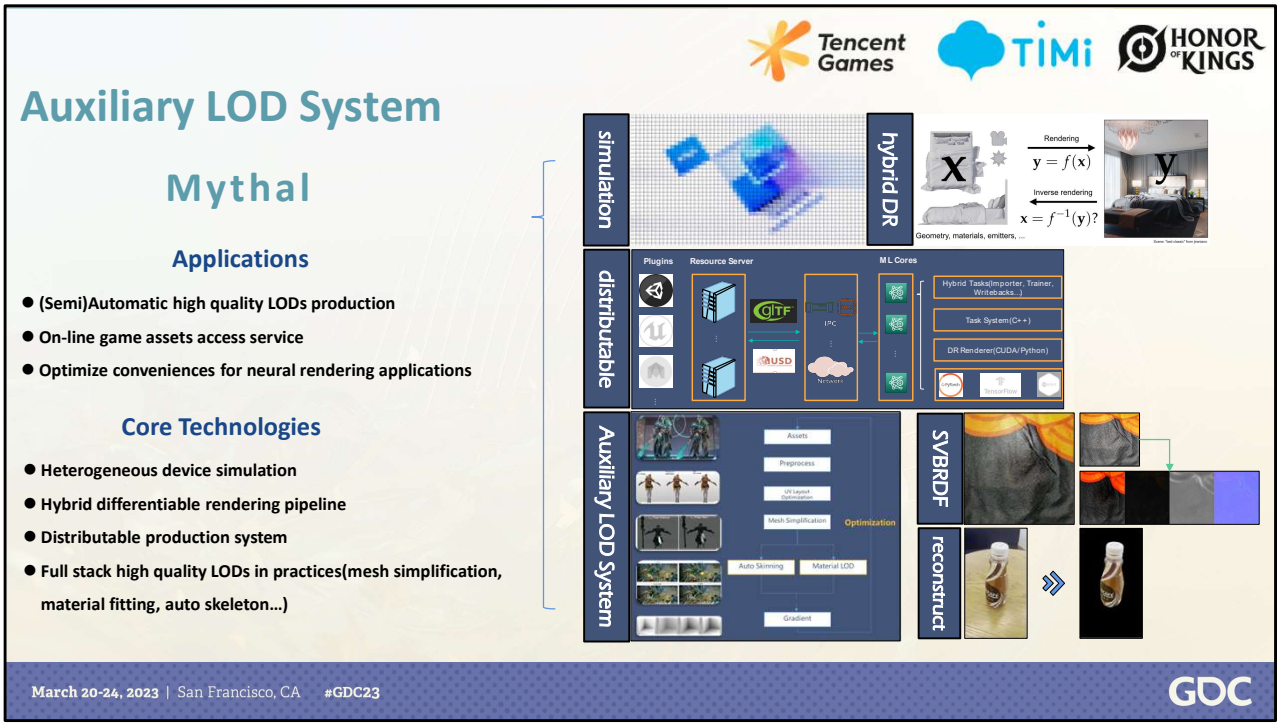
- How to address large scenes?
- How to communicate directly with commercial engines?
- How to deal with extremely customized materials?
- How to fitting complex rendering effects, such as multi-pass materials, purely metallic materials…?
- How to use DR in combination with other AI technologies?

There are other issues that should also be considered when using differentiable rendering for game asset production.

One of the most important is performance. Generating LOD for high-quality assets can be very resource-intensive, as it requires a lot of memory space to store fitting parameters and gradient compute graphs.

[click]
We also need to figure out how to fit large scenes, how to communicate with commercial engines such as unity and unreal natively, how to handle highly customized materials, and how to integrate with other technologies. These are all challenges we need to address.

Unfortunately, there were no existing open solutions to all these issues. Therefore, we have developed our own auxiliary production system, and have been trying to address all of them.



Last year, our team implemented Mythal. It is a project that can significantly improve our production capacity for various high-quality LOD assets, including heroes, cosmetics and game scenes. Mythal has a production capacity that is 30 times greater than traditional methods.

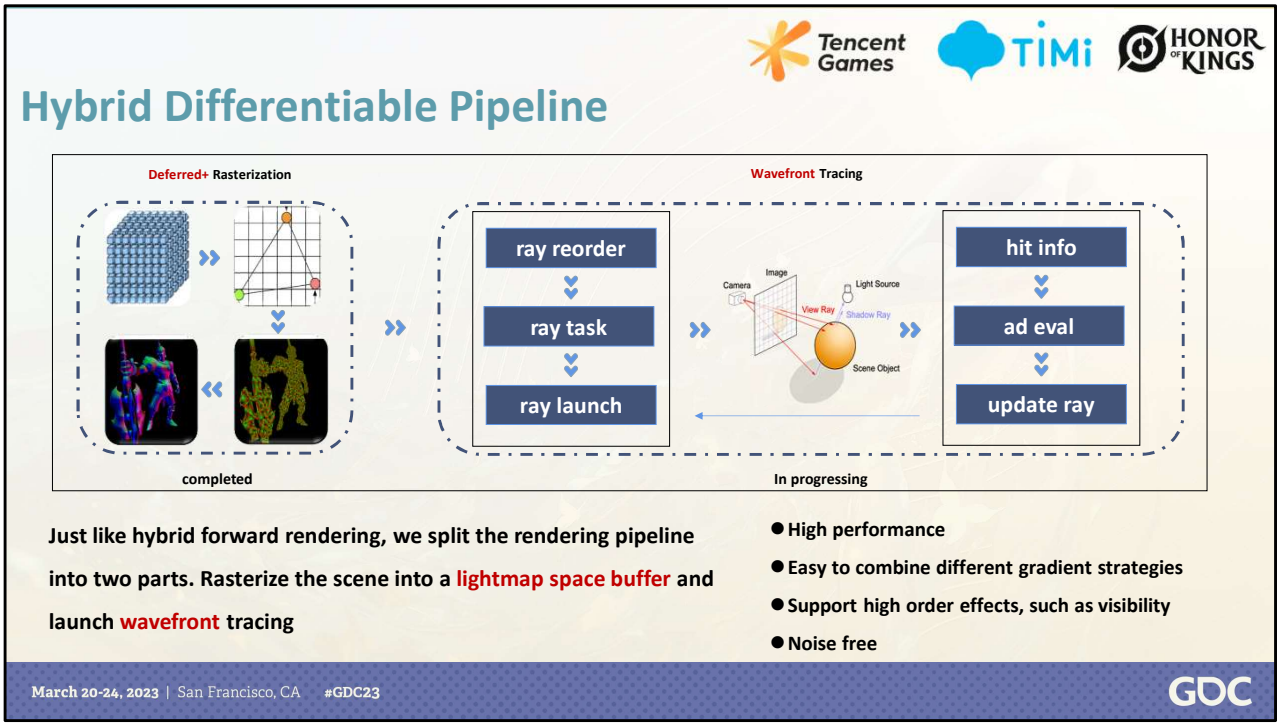
It can produce assets with high graphics quality similar to the original PBR shading method, but with only a 10% performance requirement in extremely fittings.

As an auxiliary production system, It offers many microservices, including online access to our game assets. We also had used it to improve the convergence performance of many neural rendering applications of our team. And parts of these will be discussed further in Frei section.

[click]
Let’s looking back, Mythal consists of several core components, including a heterogeneous simple device simulation, a hybrid differentiable rendering pipeline, and auxiliary systems like scene and asset management and network facilities.

It also includes a hand of algorithms that support advanced fitting and simplification

tasks for high-quality LOD generation.



Our rendering pipeline includes a range of differentiable rendering techniques. One of these is the raster method, introduced by NVidia's nvdiffrast. it is commonly used in the game industry due to its high performance and the ability to produce noise-free results.

We've implemented this method in the first stage of our pipeline and made some important optimizations, that focuses on improve the efficiency in reverse rendering. I will provide more details about this on next two pages.

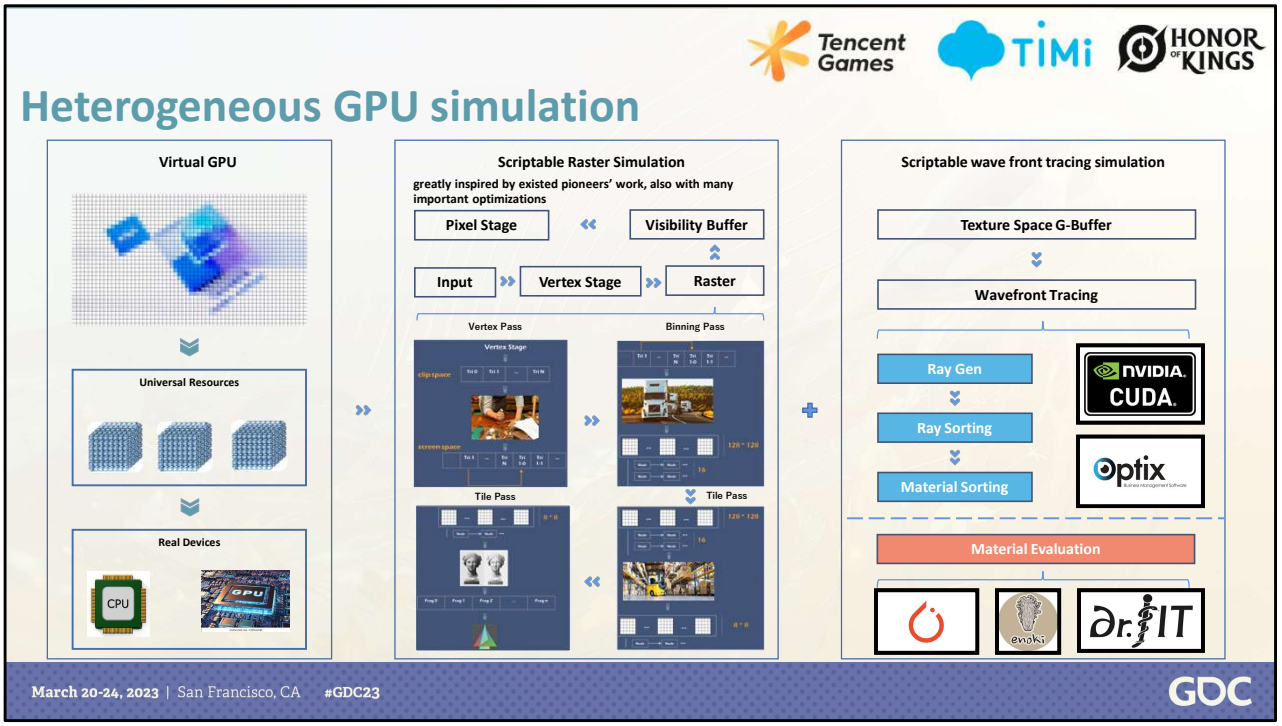
In addition to the raster method, we've also included wavefront ray tracing to handle more advanced, high-order rendering effects, particularly related to visibility. Our pipeline is flexible and can work with other automatic gradient frameworks such as enoki and PyTorch.

The system is designed to be highly deferred and modular, with the goal of dividing the rendering and reverse rendering process into as many independent parts as possible.

The deferred concept, both used in raster and raytracing pipeline, is crucial here. It

can significantly simplify the overall process and facilitate integration with other auto-gradient frameworks.

Later on, I'll demonstrate how to build an easily extensible hybrid differentiable rendering pipeline while keeping this key point in mind.



We implemented both the raster and ray tracing process based on C++ and CUDA, so that we can add any customized gradient methods at specific hardware stages. This decision helps us working without being limited by any real device capabilities.

For instance, we have added manual gradient methods on the hardware blending and the hardware ROP process in our simulated raster.

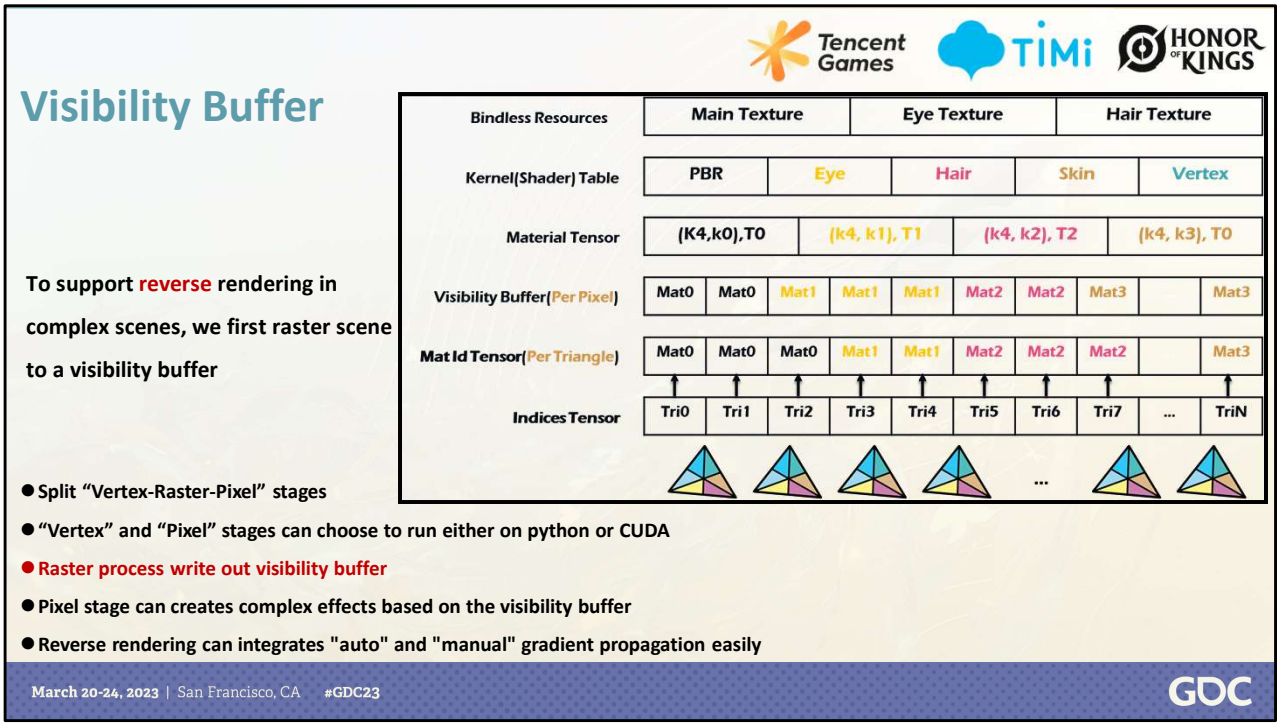
If our raster switches to the blend state, the visibility buffer will also record references for extra historical information per-texel. By using these historical information, we can manually calculate the correct gradients for blending operations.

Compared to commonly used depth peeling, our approach can improve the efficiency of gradient backpropagation for transparent materials.

Additionally, In order to increase flexibilities and enable collaboration between different devices. We've also developed a virtual device mechanism to let users work only with front-end devices, and without needing to know the backend details.

This design can also bring some easier experiences for developers. They can freely

switch between CPU and GPU backends during the whole development. Obviously, It is very convenance for them to debug and deploy.



In practice, another challenge that exists in original differentiable raster is how to fit complex scenes or models with multiple materials.

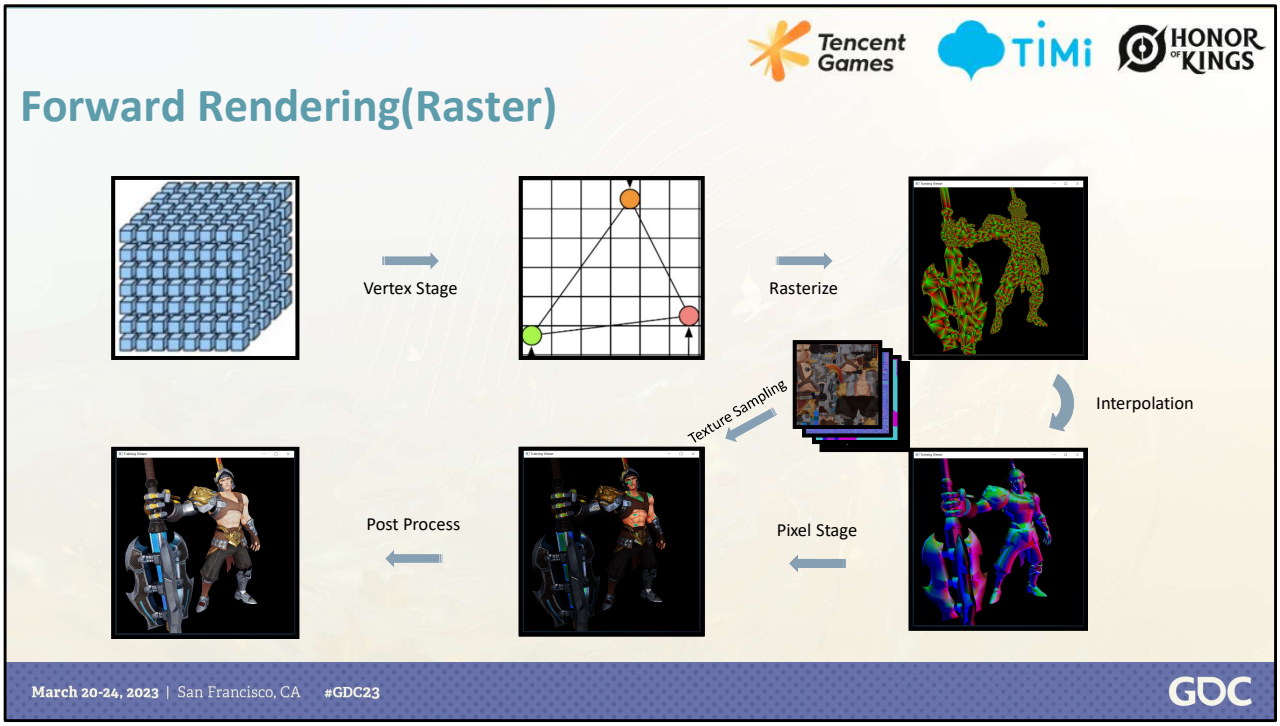
The difficulty lies in determining the correct backpropagation paths and parameter sets.

One common solution for that is to only render parts of the same materials in different epochs, but this can easily lead to suboptimal performance.

We have implemented a more robust method inspired by UE5's nanite. We use a visibility buffer that includes clip space barycentric coordinates and their gradients, as well as extra information like material identifiers.

These extra information will allow us to handle complex scenes and models by accurately work with multiple materials in both the deferred pixel stage and the reverse shading passes.

For this, a further detail will be discussed in the reverse rendering section.

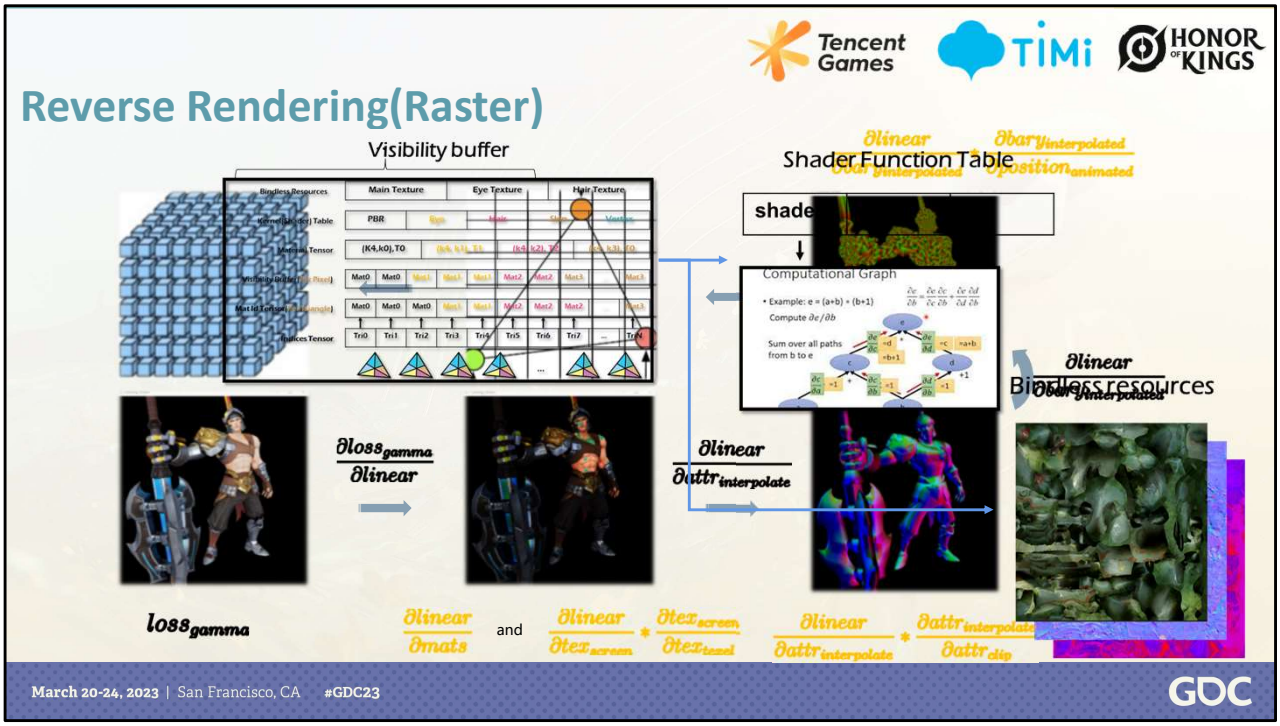


Let’s first look at these images. They illustrate the different stages of the forward raster rendering process in our pipeline.

The process starts with the vertex stage, [click]followed by simulated rasterization and attribute interpolation. [click]The pixel stage comes next, [click]followed by final post-processing.

The deferred style here, allows us to implement the vertex, pixel and any other stages in either Python or a CUDA kernel independently.

Of course, this can make the pipeline more flexible, adaptable and easy to extend when facing different situations.



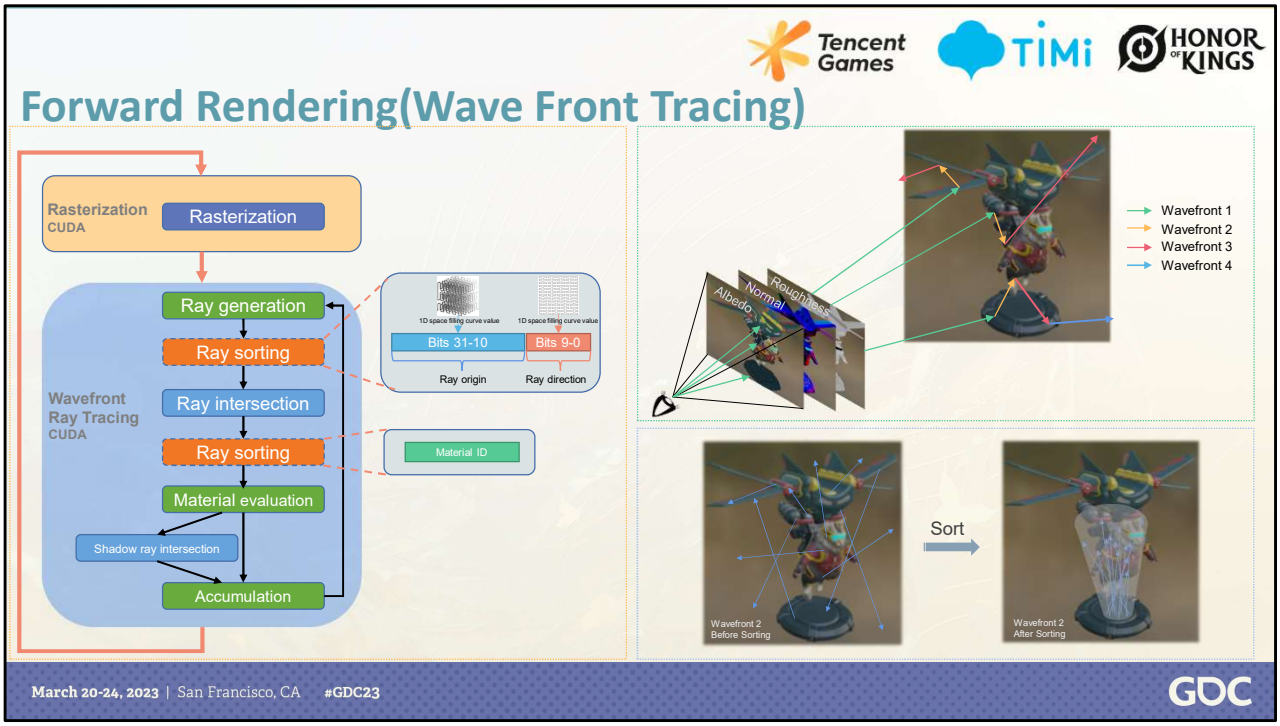
The reverse rendering process in raster pipeline focuses on how to propagate image space loss backwards through it.

Recalling the visibility buffer, that stores the clip space gradients, and the material identifiers.

It now can help us to determine which gradients should be used. We will then use these information to select the appropriate compute graphs from our simulated shader function table and then use them with other bindless resources to calculate and backpropagate gradients correctly.

[click]
The whole process utilizes both automatic and manual differential techniques. Because the forward rendering process is differed, it now allows for splitting and combining of various gradients of textures, materials, and geometries, etc.

Therefore, the system's different stages are all possible to independently calculate and backpropagate these gradients.



Now, it's time to move on to our deferred ray tracing pipeline, also known as wavefront ray tracing. It is used after the raster stage.

Currently, we only use it for some specific purpose such as to fit local visibility data in HOK. Its capabilities are still in development

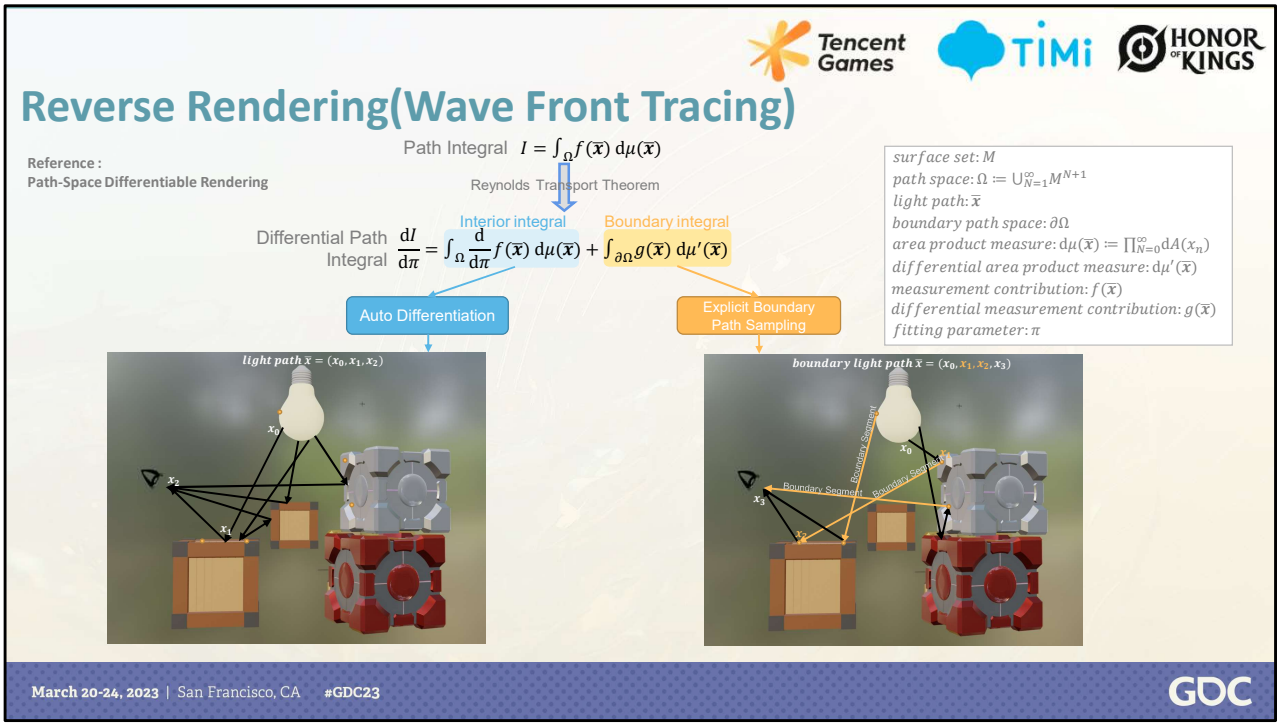
Back to current page, we can find a ray generation stage initializes a extension ray buffer at the beginning.

The next stage is ray sorting. We use space filling curves to reorder all rays for cache consistency.

Control flow divergence during complex material evaluations should also be eliminated. We incorporated material sorting as a key part of our coherent material evaluation.

After the material evaluation, the integrator will decide whether to continue forward path tracing, terminate any ray path, or generate shadow rays by selecting light sources.

Wavefront ray tracing also has some favorable deferred and modular properties. It means that we can easily separate the whole material evaluation stage apart, solving material's shading and reverse rendering process with the help of other auto-differentiation frameworks.



The backward computation of wavefront ray tracing uses a path-integral formulation. Here I will only give a brief introduction

To comprehend these complicated equations, let's just mark 'M' as the combination of all object surfaces, 'Ω' as the path space, and '$\bar{x}$' as a specific light path.

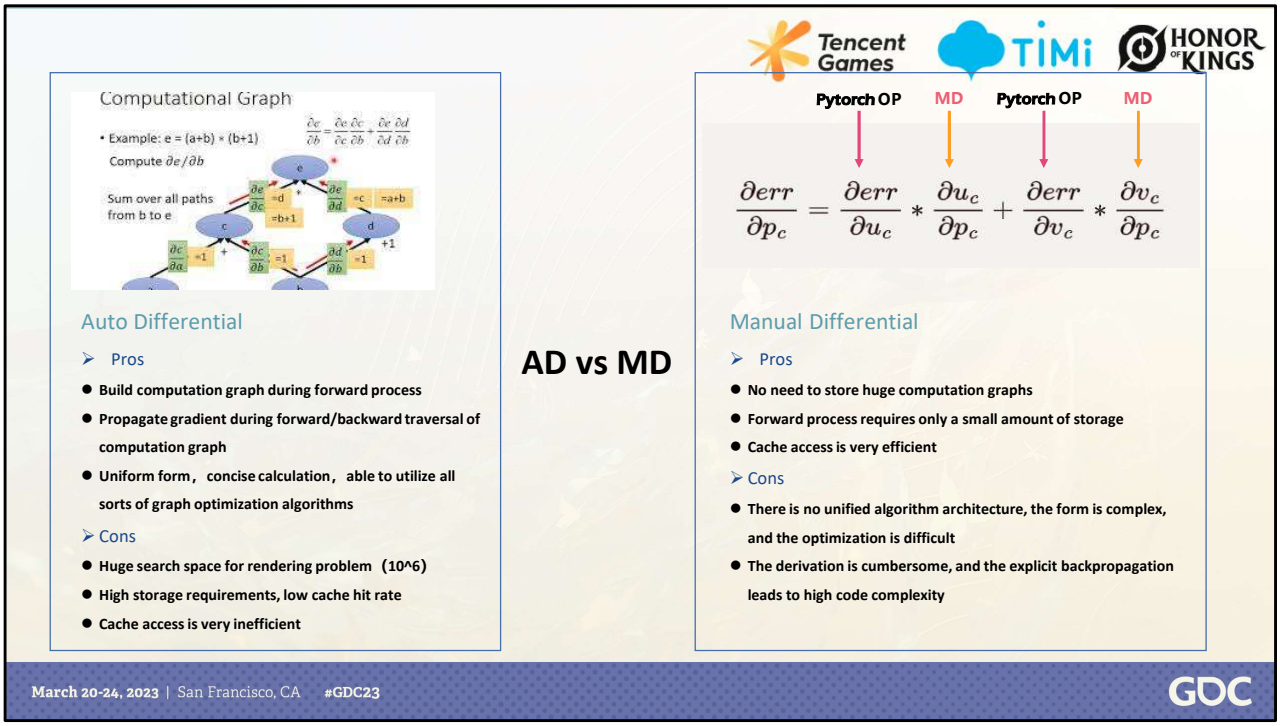
The most important idea is to separate the entire derivation into two terms based on the Reynolds transport theorem: the derivation of the interior integral and the derivation of the boundary integral.

The derivation of interior can be solved using conventional path tracing and auto-differentiation. The derivation of boundary is much complex, it requires a explicit boundary path sampling to estimate the scattering contributions on object geometries.

The derivation theorem in path-space differentiable rendering is quite complex. I think it may be difficult to fully explained today.

[click]

For those who are interested, I provide a further reference here for additional information.

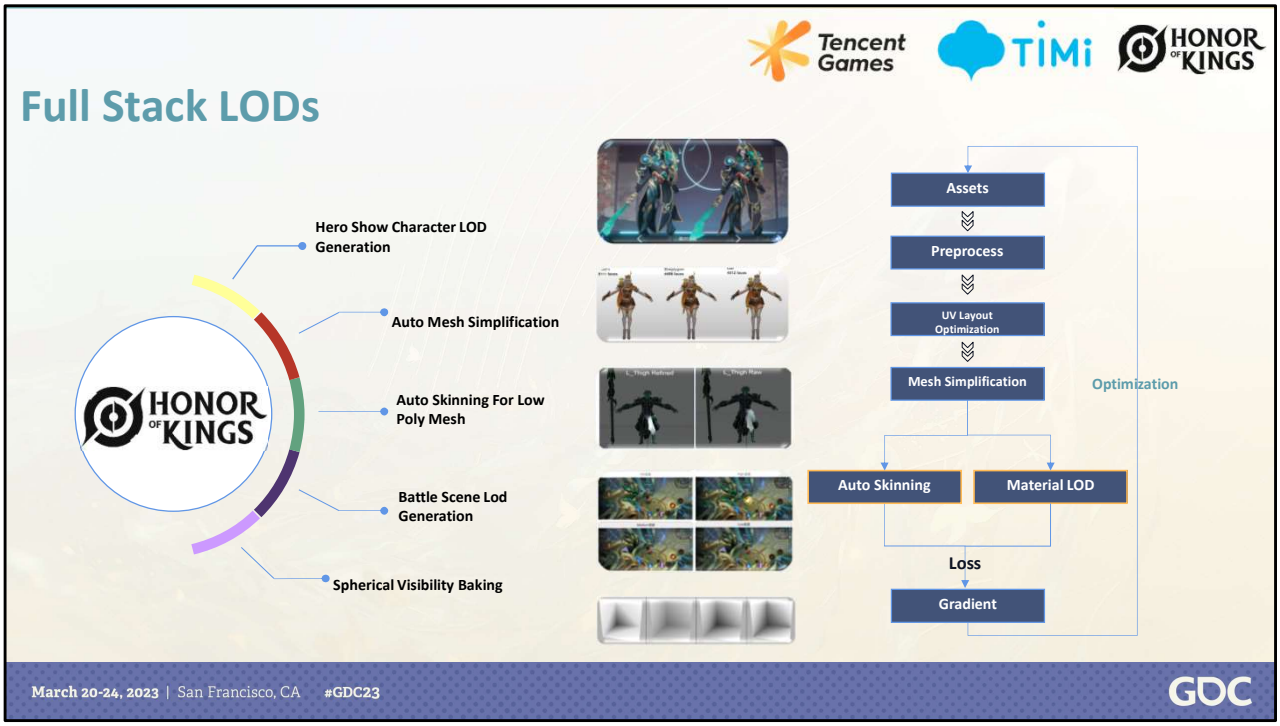


Delving into the backpropagate process, our pipeline allows for free choice between automatic and manual gradient propagation.

Automatic differential technique uses a compute graph for complex, diverse tasks like material shading. It's general and simple, especially when working with our in-house differentiable material language compiler. But it requires a lot of memory storage and usually be slowed down by cache misses.

[click]
On the other hand, manual gradient method uses the chain rule to calculate and propagate gradients. It performances quit faster, so is typically used in many fixed hardware stages like rasterization, attribute interpolation and boundary path space sampling. But it is less general and not as elegant as automatic method.

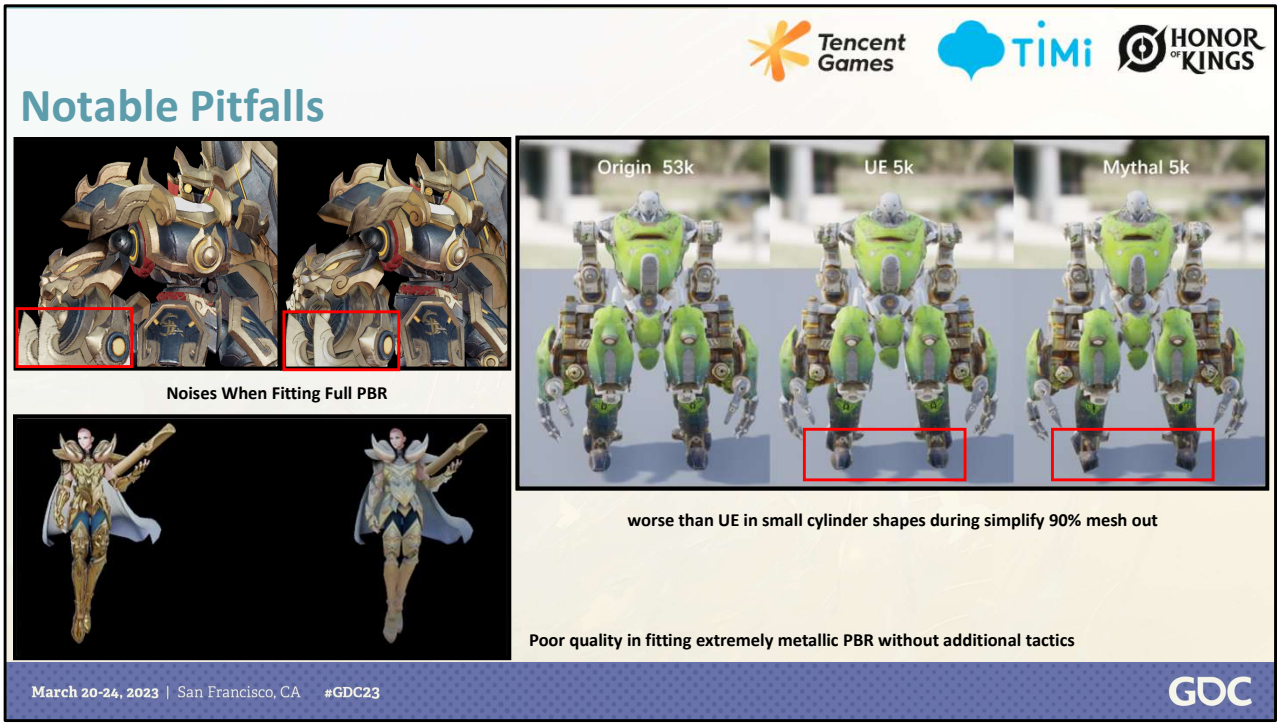
Both approaches have their pros and cons, so it's best to use a combination of them.



After completing the core components of Mythal, we have successfully used it to build a one-stop game asset production pipeline for HOK and other games in our studio.

Differentiable rendering is a perfect exceptional algorithmic framework. It can greatly assist with many tasks, such as mesh simplification, material optimization, auto skeleton generation, and animation compression.

As I mentioned before, it can also be used to optimize some precomputed data used in conventional rendering, such as local visibility data used in HOK.



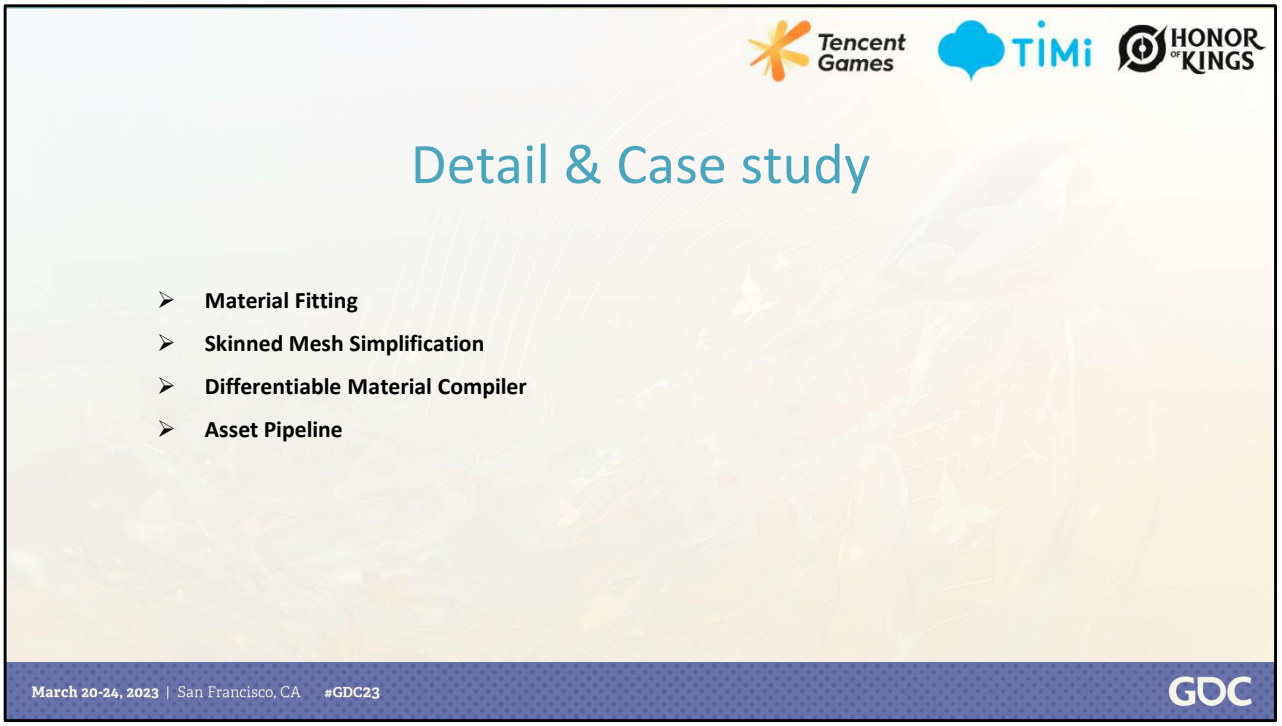
To be honest, differentiable rendering is a powerful tool, but there are also notable pitfalls to consider when used in practical.

The essential reason is that it is still a local minimal optimization, which can easily resulted in noisy fittings. For instance, it is difficult to converging when fitting a full PBR material to a single, size limited, texture if only with naïve using.

[click]
It is also difficult to fit extremely metallic materials if without additional overfitting strategies.

[click]
Differentiable rendering is effective in Mythal for mesh simplification in many cases but may not perform as well as traditional methods in some corner cases.

It is important to consider best practices when using it in HOK and other and applications. My colleague Frei will now discuss these best practices with more implementation details.

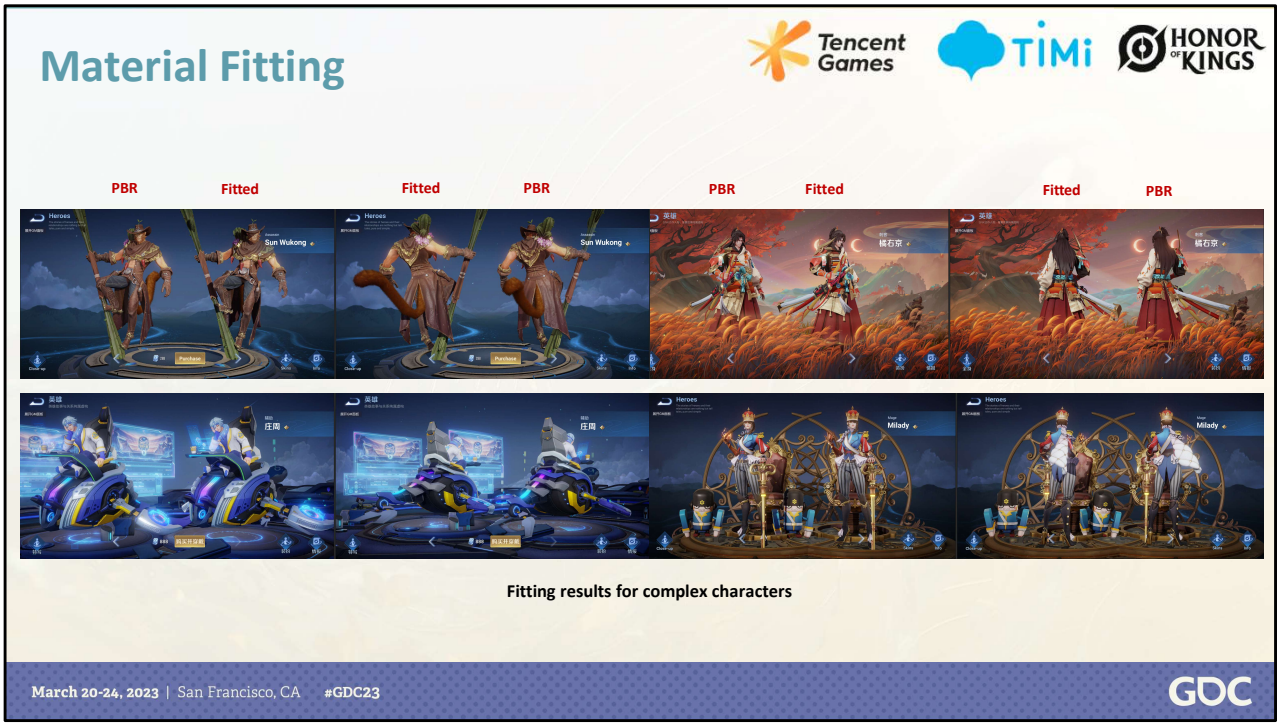


Hello everyone, I'm excited to be here at GDC with Fei to share with you our differentiable rendering system and its applications in Honor of Kings.

As Fei mentioned, our team has developed a high-performance differentiable renderer and a scalable asset pipeline to leverage the power of differentiable rendering. During his presentation, he gave us an overview of the system and showcased some impressive use cases to demonstrate its capabilities.

Moving on from that, I'd like to delve deeper into some of our core system designs(like material compiler, asset pipeline) and several detailed applications. These are crucial to the performance and functionality of our system, and I believe these will be interesting to you guys.

Let's get started!



As a mobile game developer, one of the key challenges we've encountered during development is to provide a consistent user experience across a wide variety of mobile devices.

So, we have adopted physically based rendering with some variations and tweaks which are more optimal on mobile platforms (Our colleague will share more on the core concept talk).

Though PBR brings a consistent & appealing visuals and a streamlined production process, it also brings some challenges to the performance, especially on devices with limited computing power.

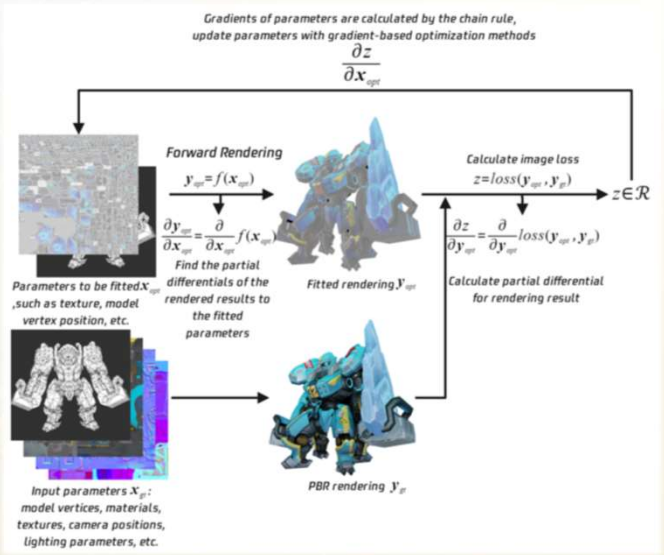
Hence, we’re doing the material fitting to reduce the shading cost. Here are some examples of our fitting results.

And then, let’s dive into the detail.

Material Fitting

- Generate different camera positions, orientations
- Calculate image space loss for simplified shading and Ground-Truth shading
- Update corresponding parameters using gradient descent

$$x^*_{opt} = \arg\min_{x_{opt}}(loss(\quad , \quad))$$



As Fei mentioned, classic material fitting have several drawbacks, make it hard to use in practice.

With differentiability provided by our piepline, we could perform gradient-based optimizations and even extend it into deep neural networks.
For the sake of simplicity, let's focus on the optimization problem today.

As we know, the goal of an optimization problem is to minimize the objective function iteratively with the help of the gradient.

Our objective function uses a screen space loss to compare the reference rendering result and the desired fitting result, both of which are rendered by our differentiable renderer.

By iteratively updating the fitting parameters with the guidance of the gradient, the loss will eventually converge to a desired value. This usually means that the reference and the fitting result will look indistinguishable.

The diagram on the right illustrates the general fitting process, which I think is

straightforward to understand.

So, let's move on for a deeper look.
~~~~~~~~

Our objective is to optimize the LOD mesh shape and material parameters to closely resemble the visual appearance of the reference high-standard PBR-rendered model.

We begin the optimization process by initializing a parameter set that includes the model vertices, textures, and other relevant attributes for the LOD asset.

To optimize the LOD asset, we utilize a differentiable rendering pipeline during each epoch of the optimization process. This pipeline consists of a sequence of mesh operations, a rasterizer, and a deferred shading stage. Once the LOD asset is rendered, we calculate an image-space loss between the resulting image and a reference image that is generated by the high-standard PBR-rendered model under the same lighting and viewing conditions. By computing this loss, we can iteratively adjust the parameter set to reduce the difference between the target and reference image, thereby improving the overall visual fidelity of the LOD asset.

Specifically, our differentiable renderer enables us to fully differentiate the rendering pipeline, allowing us to calculate the gradient of the loss with respect to the parameters of the LOD asset, such as vertex positions and texture contents. This enables us to optimize these parameters and improve the visual similarity between the LOD asset and the reference PBR rendering image.
The diagram on the right illustrates the general pipeline for fitting the LOD asset parameters. The function "f" represents our differentiable renderer. We can use $f(\boldsymbol{x}_{gt})$ to obtain the reference PBR rendering image, and $f(\boldsymbol{x}_{opt})$ to obtain the LOD asset rendering image.

By utilizing image space loss, such as L1/L2 loss, we can calculate the dissimilarity z between the reference PBR rendering image $f(\boldsymbol{x}_{gt})$ and the LOD asset rendering image $f(\boldsymbol{x}_{opt})$. We can then compute the gradient $\frac{\partial \mathrm{z}}{\boldsymbol{y}_{opt}}$. Because f is differentiable, which in turn enables us to calculate $\frac{\partial \boldsymbol{y}_{opt}}{\boldsymbol{x}_{opt}}$. By further applying the chain rule, we can compute the gradient with respect to the input parameter set $\boldsymbol{x}_{opt}$, resulting in $\frac{\partial \mathrm{z}}{\boldsymbol{x}_{opt}}$.
We iterate over a large number of image pairs, varying the camera positions and orientations either randomly or with careful generation, and using an optimizer such as Adam, we gradually adjust the parameters of the LOD asset to match the appearance of the reference model. This process continues until the loss is minimized
~~~~~~~~

and the visual fidelity of the LOD asset is maximized.

In our applications, we typically use hundreds to thousands of epochs to fit the parameters.

The fitting result, which involves an LOD asset with optimized vertex positions and texture contents, can be rendered directly in our game.

In the following section, we will use several examples to demonstrate the effectiveness of our LOD asset fitting pipeline.

~~~~~~
~~~~~~

Material Fitting

- Mythal fits the rendering results of characters from various perspectives onto a single texture, while ensuring rendering quality, to meet the high-performance requirements of the most demanding environments.
- The standard PBR rendering can be fit to a single texture within approximately two minutes using a 2080Ti GPU (1024 * 1024).

Original PBR | Single Albedo Map Fitting | Error heat map

<CLICK>This is the fitting process for one of the characters, with the original PBR rendering on the left, the fitting result in the middle, and a heat map display of the screen space difference on the right.

You see, even with only a single texture map budget, a good rendering result can still be achieved through the fitting. There is no seam, most of the specular parts are preserved.

Material Fitting

Tencent Games TiMi HONOR OF KINGS

Fitting PBR to Pure Vertex Color

- Where geometric details are rich, there is less quality loss
- Where the geometry details is insufficient, we can further optimize by the second-order gradient technique

<CLICK>This is another extreme example that fitting PBR onto vertex colors. Of course, it is not perfect, as the final quality depends on the number of vertices and the topology of the mesh.

This method is still acceptable for rendering characters when the camera is far.

Material Fitting

Tencent Games TiMi HONOR OF KINGS

Interactive Material Baking

- Interactive fitting pipeline provides artists with finetuning tools.
- Visual interaction makes artists feel “everything under control”.

March 20-24, 2023 | San Francisco, CA #GDC23 GDC

To give artists more control over the fitting process while maintaining good training efficiency, we provide an interactive training mode. As the video shows,
<CLICK>
It starts with detailed scene manipulation and parameter tuning.

<CLICK>During the training process, <CLICK>the loss and other metrics will give us an intuition on how it goes.

<CLICK>Artists are able to finetune the fitting result according to their profession and needs.

<CLICK>We support global or local fitting in either world or texture space. <CLICK>

You can see from this example that artist is assigning more weight to a certain part of a model to increase its detail.

Material Fitting

Tencent Games TiMi HONOR OF KINGS

Support Different UV Layout

- **Colors stored with uv0 coordinates**
- **Colors stored with uv1 coordinate**
- **UV0 store brightness + UV1 store color**

Quantization optimization

Automatic fitting and compression of HDR data to LDR data

UV0 UV1

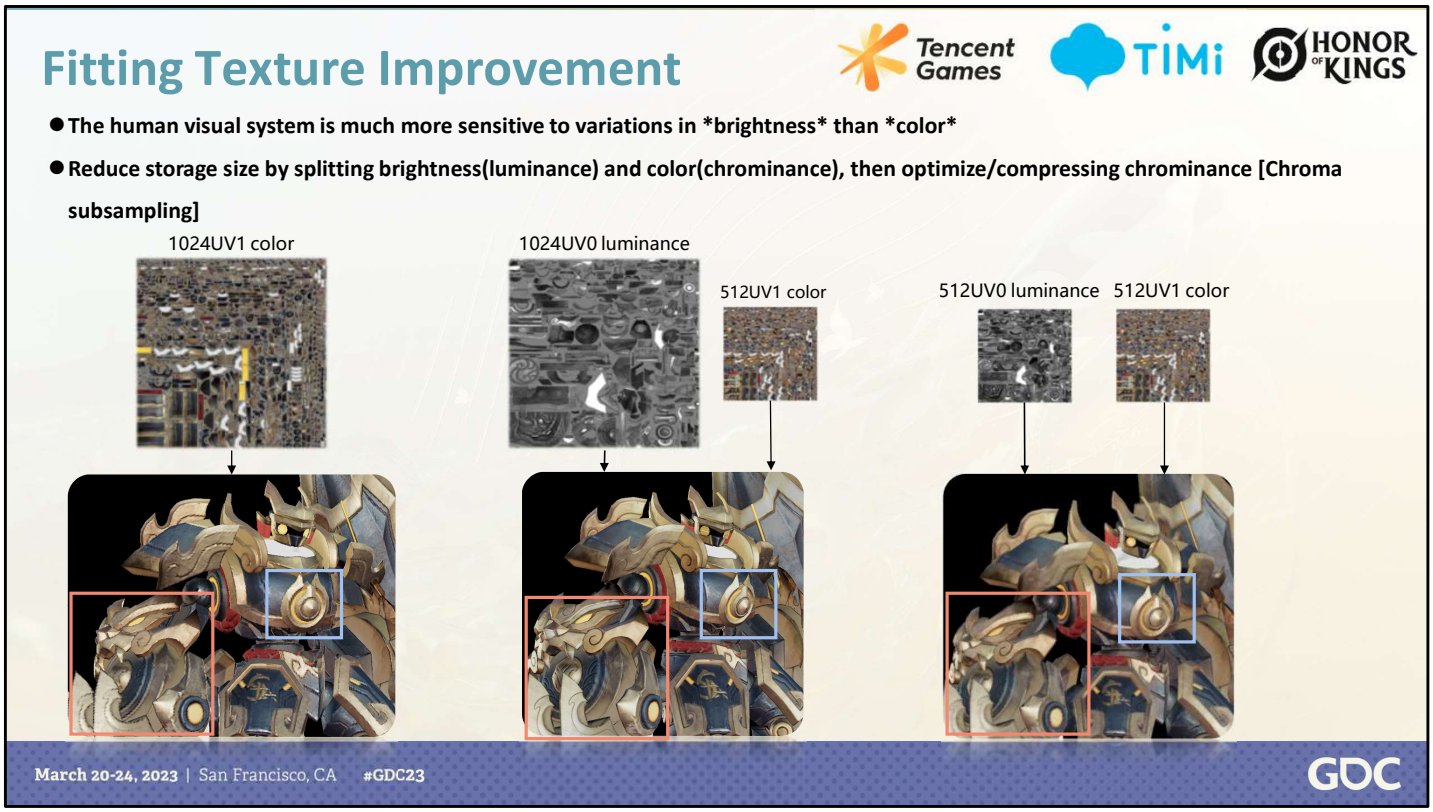
March 20-24, 2023 | San Francisco, CA #GDC23 GDC

As we run the fitting process with full precision to ensure a good convergence rate and result quality.

But, saving the results with full precision isn't always feasible, as their size can be quite substantial. Therefore, we've come up with several solutions to tackle this issue.

Such as splitting the result texture map by visual contribution, applying differentiable quantization, and automatic dynamic range compression, etc.

Let’s have a quick look.



As a background, I'll introduce some terminology. UV mapping is the process of creating a 2D representation of a 3D model's surface, is done by unwrapping the model's geometry into a flat 2D texture, which is called a UV map or UV texture.
A UV island is a connected group of polygons in a UV map.

In our game, the first UV map of a model (referred to as uv0) may contain overlapping or mirrored UV islands in order to increase the average island resolution thus higher texel density and better visual quality.

And the second UV map (referred to as uv1) is generally used for storing our fitting result, there is no overlapping or mirrored UV islands in this map because we don't want to mess up the specular part of the fitting result.

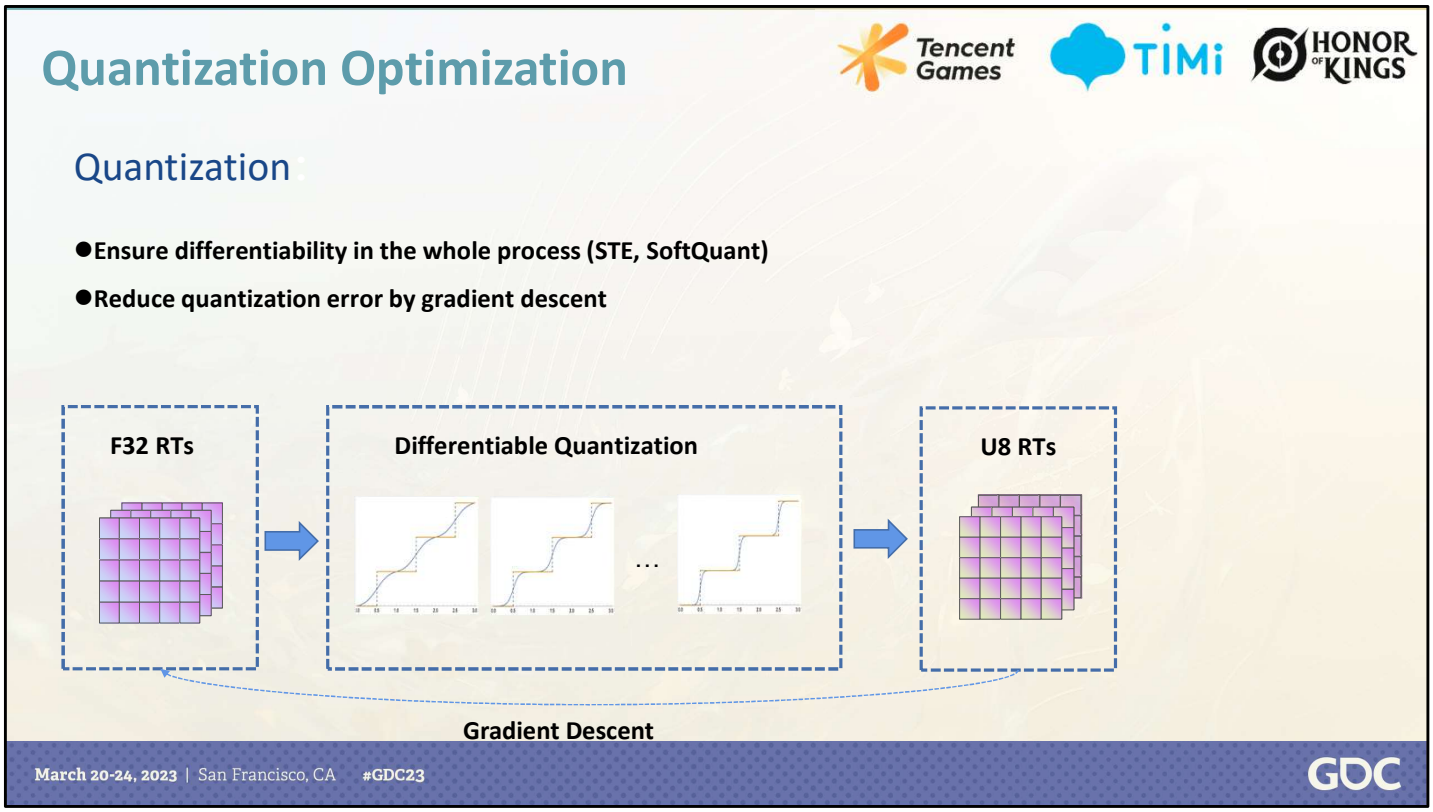
Back to the topic,

To increase the fitting quality of the original result on the left, the most straightforward way is to increase the resolution of the texture. But we can't always afford to do that due to the package size constraint.

Based on several works related to chroma subsampling for image and video compression, which utilize the fact that human eyes are more sensitive to brightness than color, we could split the fitting result into two parts, one for brightness(or luminance) and one for color(or chrominance).

Then we could store the luminance part in a higher resolution texture, and store the color part in a lower one. Since the uv0 texture is optimized to have more uv space per island as I mentioned, it's more suitable to store the luminance part. And then the color part can be stored in uv1 to save space.

We have made several experiments on this idea. Eg. The fitting result on the middle with optimized uv0 and smaller uv1 texture is better than the original.
And we could also use a smaller uv0 and uv1 texture as a combination to approximate the original one. The fitting result on the right is a bit worse, but it's still acceptable in some cases.



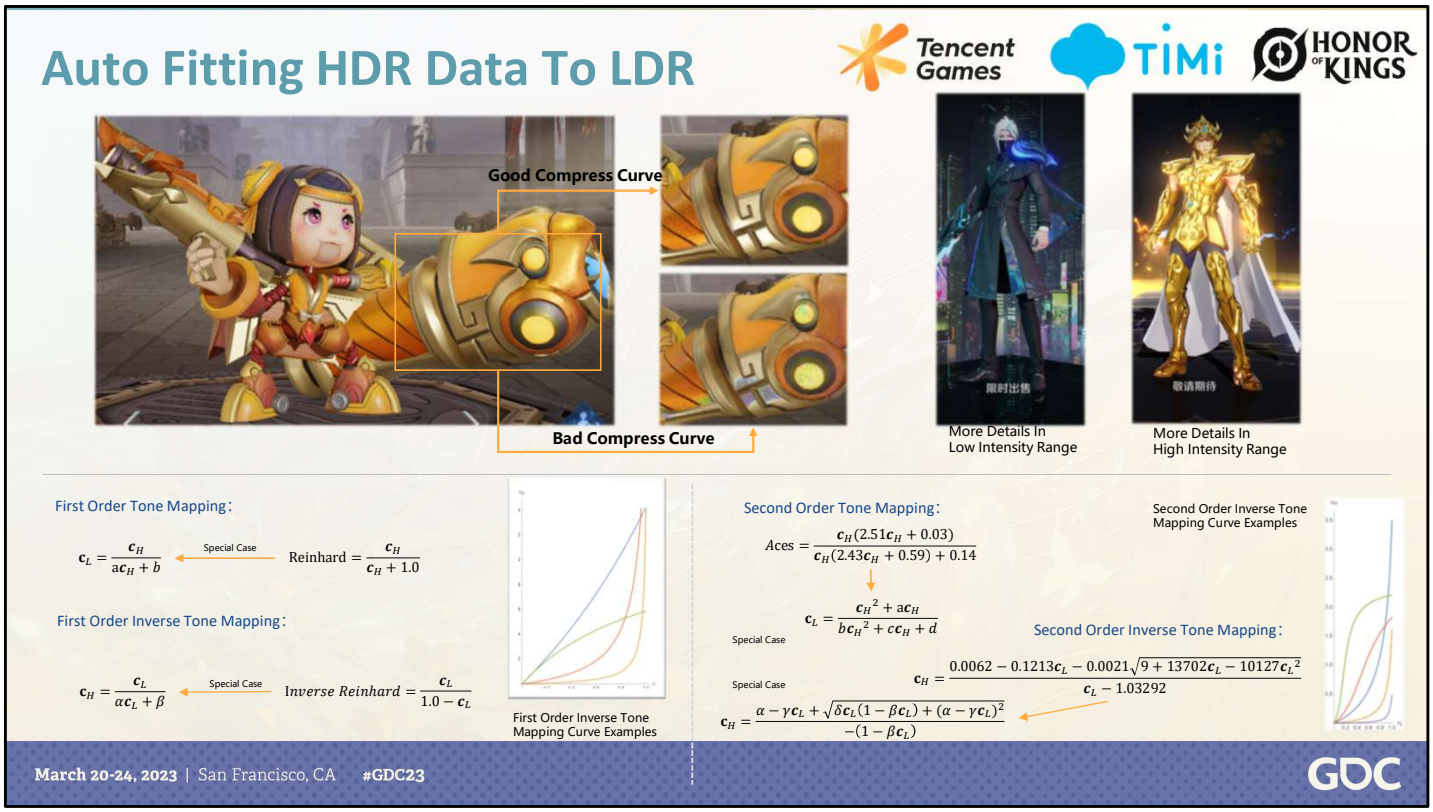
In practice, the rendering result is typically stored as an 8-bit integer, whereas the shading calculation is performed using floating point. As a result, it becomes necessary to quantize the floating point data to an 8-bit integer.
This involves the process of quantization, which converts continuous values into discrete ones by clipping and rounding, and often results in a loss of precision.

To alleviate this issue, we want to take the quantization error into account during the loss calculation of our optimization process.
But quantization is not differentiable, all the gradients will vanish at some point.

So, How could we solve this problem? Turns out there are several solutions to this problem,
such as using the straight-through estimator (STE) or a soft quantization function etc.

These all work by replacing the quantization operation with a differentiable approximation.

Based on our experiments, we found that the soft quantization function is more stable than the STE method.



Let’s talk about tone-mapping.
Since we have adopted physically based rendering, which implies that HDR rendering/lighting is used, it also means that ideally our fitting result should be stored in HDR format as well.
But due to the limitation of the storage size, we store the resulting texture in LDR format through tone mapping. And in rendering, we apply the inverse tone mapping to convert it back to HDR format.

As a simple background, you can treat tone mapping as an invertible non-linear function which maps HDR values to LDR.
And by adjusting the parameters of the tone mapping function, we can control the details of dark and bright areas in the image.

It's not optimal to use fixed tone mapping parameters in practice, because the scene may have different dynamic ranges.

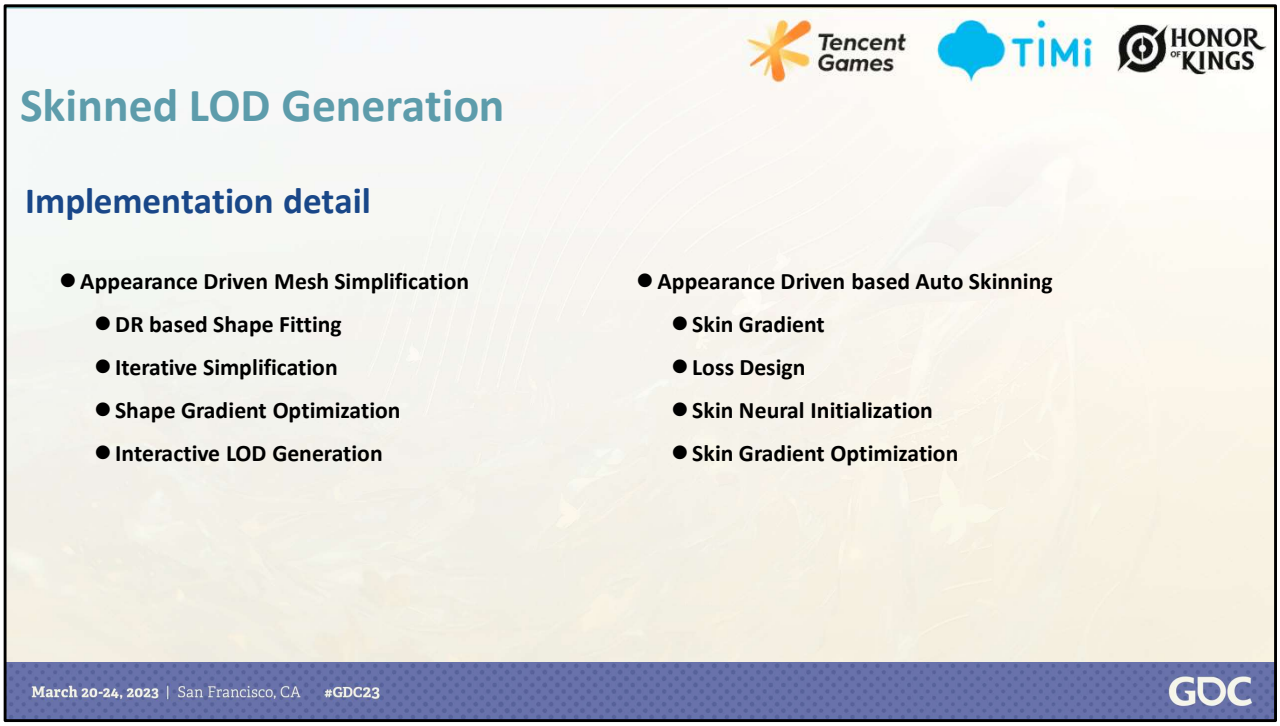
For instance, let's take a look at the character model on the left, which features a

shiny surface. Mapping too much detail to the dark areas will make the bright areas look dull. So we need to adjust the tone mapping parameters to give more details to the bright areas.

Since our rendering pipeline is differentiable, we can optimize these parameters automatically with respect to appearance factor.

We've tried several tone mapping functions, and found that the modified Reinhard function works for most of the cases. For complex scenes we also support more advanced/high-order tone mapping functions, with more computation cost as a trade-off.

<EOP> So, the material fitting part is over, hope you guys get some ideas.



After getting the material fitting done, we hope to further generate the Skinned Mesh LOD assets with the help of differentiable rendering.

Yeah, differentiable rendering is really a fundamental building block of our asset pipeline. It opens up a lot of possibilities for us.

The following section will present our work on Appearance Driven Mesh Simplification and Auto Skinning.

Mesh Simplification

Related solutions

- Vertex Cluster / Decimation Based
- QEM Edge Collapse Based
- Remesh Based

Cons

- Lacking consideration for the visual importance of mesh vertexes and faces
- Common remesh algorithm can not retain original UV
- Conventional mesh simplification methods lack consideration for skinned meshes.

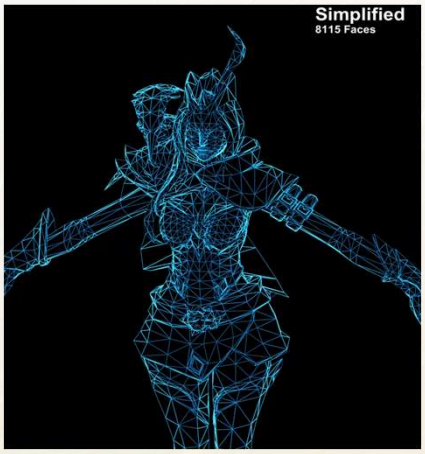
When it comes to mesh simplification, there are many classic methods available, such as vertex decimation, edge collapsing, and remesh-based techniques. However, these methods typically only consider geometry-based error metrics and often fail to consider appearance factors and skin weights.

To address this issue in mesh simplification, we have come up with a hybrid approach to deal with it.

Let's see how it works.

Appearance Driven Mesh Simplification

- **Iterative Simplification：** **Repeatedly choose areas with minimal visual impact to compress**



Iterative Simplification

The first step is a **coarse mesh decimation** operation that using classic algorithms as a speed-up.
The error metrics of this step are based on the image space difference produced by our renderer.

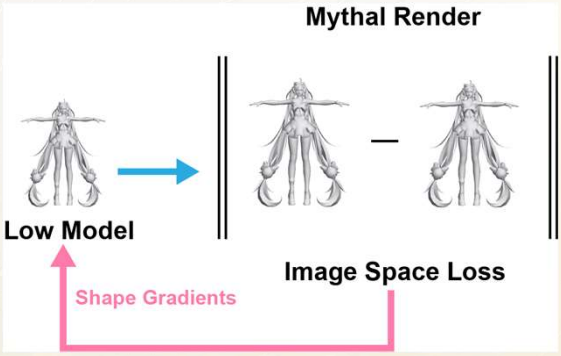
Furthermore, we will apply additional weights to the vertices around joints to ensure the skin quality of the simplified models.

And then,

Appearance Driven Mesh Simplification

- **Iterative Simplification:** Repeatedly choose areas with minimal visual impact to compress
- **Gradient Optimization:** Optimize the compressed meshes with shape and skin gradients, reducing visual discrepancy

Iterative Simplification



Appearance Driven Shape Optimization

After **The Coarse Simplification**, we will apply the **shape gradient optimization** on the decimated mesh to reduce visual discrepancy.

Let's see how **shape gradient descent** works.

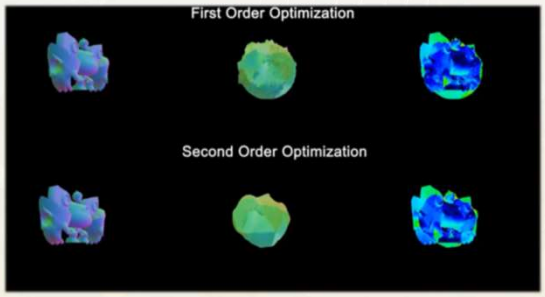
DR based Shape Fitting

Shape Gradient Descent

Efficient shape fitting can be achieved by running shape gradient decent on the following Optimization Goal:

$$\underset{x\in\mathbb{R}^{nX3}}{\operatorname{argmin}}\,\Phi(R(x))+\frac{\lambda}{2}tr(x^{T}Lx)$$

(L is Discrete Laplace regularization, R is differential render, Φ is loss function)



First-Order / Second-Order Shape Fitting

First-Order Optimization

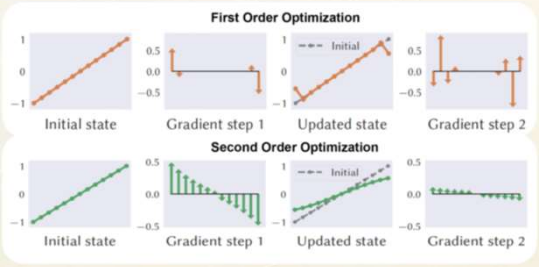
$$x \leftarrow x-\eta\left(\frac{\partial\phi}{\partial x}+\lambda Lx\right)$$

Second-Order Optimization [Nicolet2021Large]

$$x \leftarrow x-\eta\left(\frac{\partial^{2}\phi}{\partial x^{2}}+\lambda L\right)^{-1}\left(\frac{\partial\phi}{\partial x}+\lambda Lx\right)$$

approximate the **Hessian with** Identity matrix:

$$x \leftarrow x-\eta(I+\lambda L)^{-p}\frac{\partial\phi}{\partial x}$$



[Nicolet2021Large]

By leveraging **differentiable rendering**, we can ensure the differentiability of the entire pipeline,
which enables us to perform appearance-driven mesh optimization.

To achieve this,
we designed a **loss** function that takes into account both appearance and geometry factors.
Moreover, to ensure the smoothness of the geometry, we introduced a **Laplacian operator** into the process.

<CLICK>The conventional first-order optimization may converge slowly and is more likely to be trapped in local minima. Because complex geometry optimization is usually a non-convex optimization problem. And the Laplacian operator as a hyperparameter is not easy to tune.

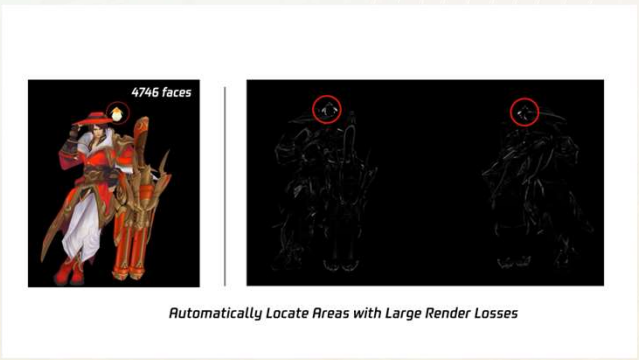
Therefore, following the recent research results, a **second-order** optimization is applied by using an approximated Hessian Matrix to

increase the convergence speed.

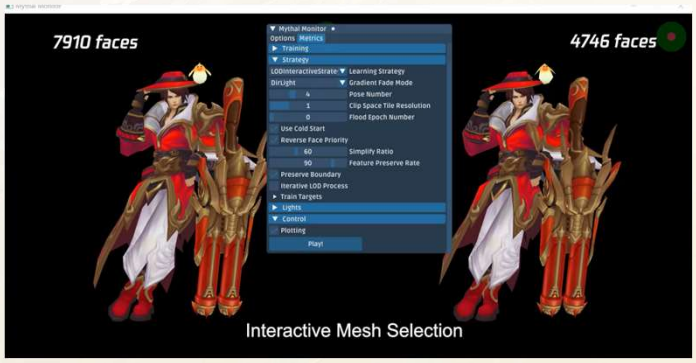
From the sphere fitting example in the video <CLICK>, we can see that second-order optimization can achieve much faster convergence

Interactive Mesh LOD Generation

- Our System can automatically Locate areas with large render losses and optimize them with gradient descent and backtracking.
- Besides, artists can Interactively paint on meshes to precisely select areas for Finetuning .



Automatically Locate Areas with Large Render Losses



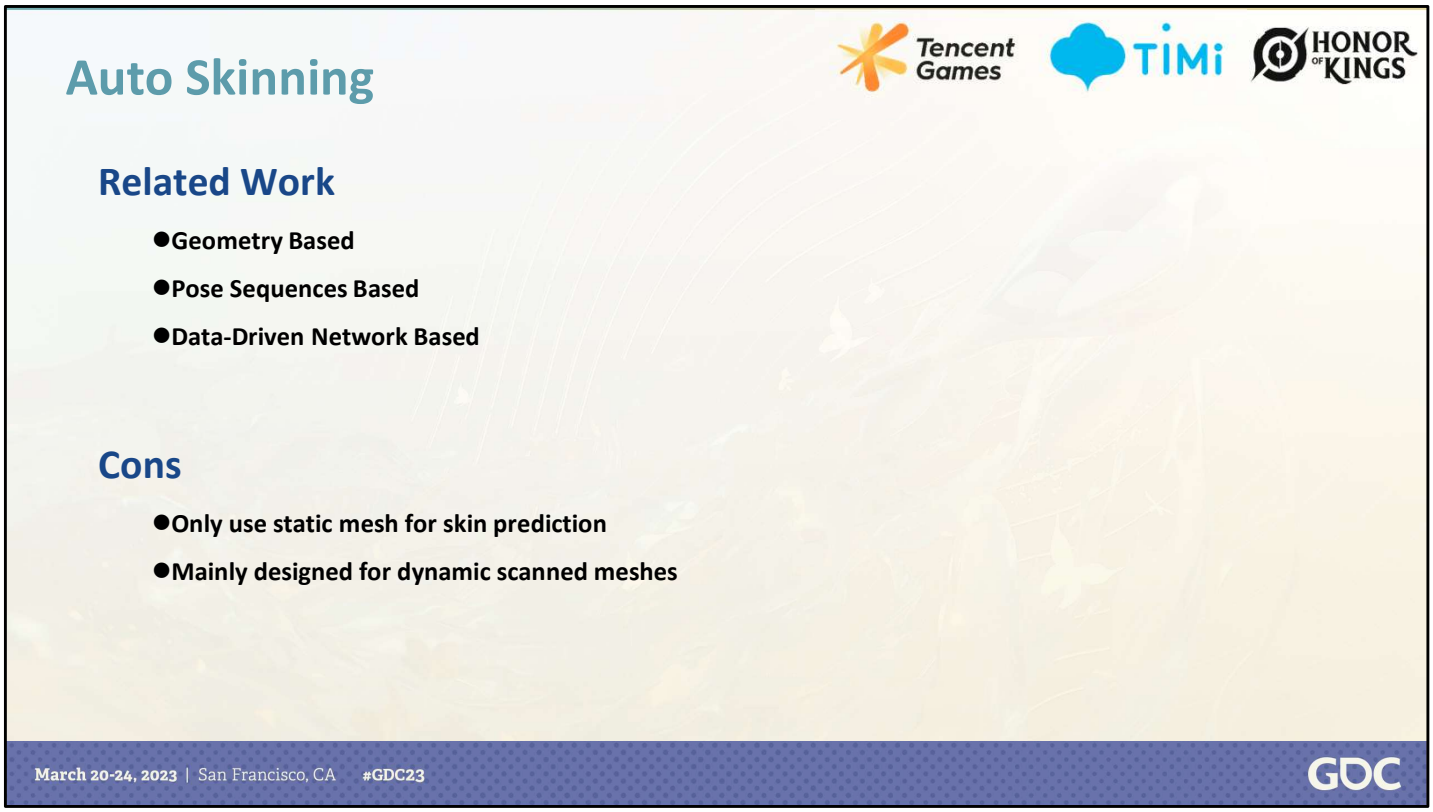
March 20-24, 2023 | San Francisco, CA #GDC23

Sometimes, there are corner cases we have to deal with. For example, some complex meshes may produce large gradients, which make it hard to converge.
Therefore, we have implemented a backtracking step to redo the undesired modification. <CLICK>

Besides, we also give artists a way to interactively paint on meshes to precisely select areas for fine-tuning. <CLICK>

Since the simplification process generally changes the topology of the

mesh, so the skinning weights of the mesh need to be fixed as well.

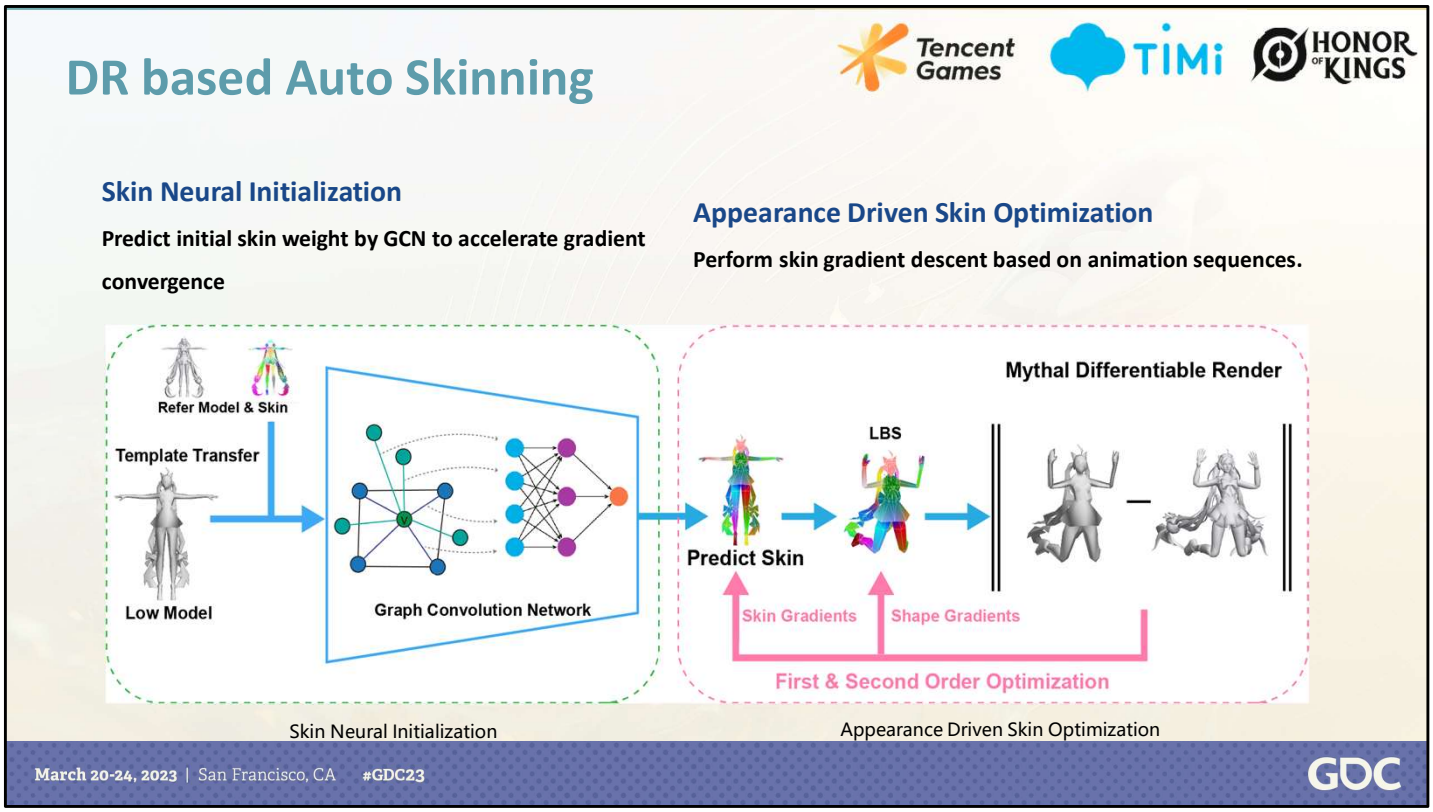


To fix the skin weight from last step,

Most of the existing auto skinning methods, like geometry and neural network-based only use static mesh for skin prediction, lacking consideration for the existing animation sequences. While pose sequences based methods are mainly designed for dynamic scanned meshes,

Besides, without making use of the existing skin, the generated skin quality can not be guaranteed.

Therefore, we designed a differentiable rendering based auto-skinning system to address these issues.



Our system is composed of

a skin neural initialization module, which predicts initial skin weights by the neural network to accelerate the gradient convergence,

and an appearance driven skin optimization module, which performs further skin optimization based on existing animation sequences.

Skin Neural Initialization

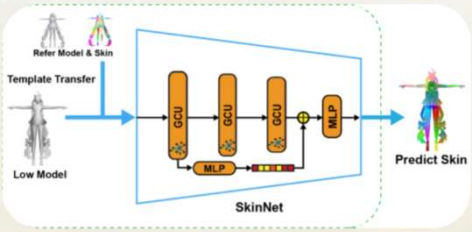
SkinNet Structure (ref:[xu2020rignet])

- Multi Layer GCU : Ideally suitable conv layer for graph based data (Mesh, Skeleton)
- Global / Local Encoding
- Output skin probability distribution

Training

- Data: open source dataset to pretrain, game assets to finetune
- Loss: Cross-Entrophy $L_s(w_s) = \frac{1}{V}\sum_v \sum_r \hat{s}_{v,r} \, log \, s_{v,r}$

  (V is vertex set, r is bone set, $\hat{s}_{v,r}$ is gt skin, $s_{v,r}$ is predict skin)



Optimization

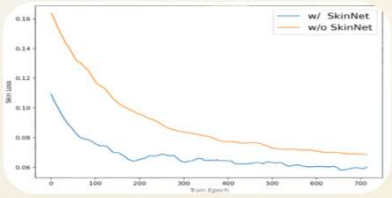
- Optimizing for skin mixture :

  Add vertex visibility matrix,

  Add Humanoid Skeleton Prior

Benefit

- Accelerate the gradient convergence
- Avoid falling into local minima



Skin convergence rate w/ or w/o SkinNet

The **skin neural initialization** module is based on SkinNet, a network designed for initial skin weights generation for low-poly meshes. Our work is strongly inspired by the paper RigNet.

The SkinNet comprises three layers of Graph Convolution Units (GCU) to capture both global and local feature of input mesh.
With volume geodesic neighboring and attention driven clustering, we got bone probability distribution and finally the skin weights.

To improve generalization, we use open-source dataset to pretrain and fine-tune with HOK's own models.

We use cross-entropy loss and Adam optimizer with Edge Dropout during training to speed up convergence and
alleviate the over-fitting issue.

From the graph on the bottom right, we can see that the convergence rate increases

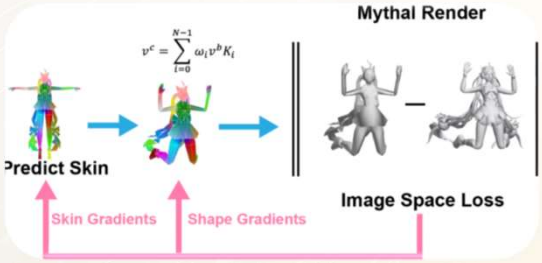
a lot after the **skin neural initialization** module is applied.

Appearance Driven Skin Optimization

Skin Gradient

Skin Gradients can derive from Linear Blend Skinning (LBS) equation

$$v^c = \sum_{i=0}^{N-1} \omega_i v^b K_i$$



Skin Optimization with Animation

- Optimize LOD skin weight from the animation appearance of the reference model
- The learned weights are more optimized for the animation compared with conventional conventional static mesh based methods

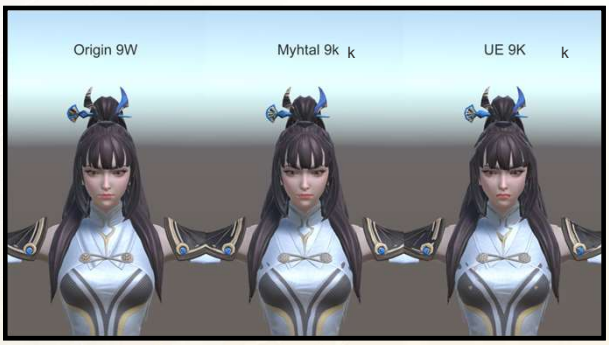
Using the initially predicted skinning weights by neural network as a start, we aim to further optimize them through a gradient-based method with the help of differentiable rendering.

- First, we obtain skin derivatives using the Linear Blend Skinning equation, which can be integrated with our differentiable rendering pipeline to effectively propagate gradients.
- Then, we iteratively refine the skinning weights with respect to the appearance loss.
- Additionally, we leverage the original mesh animation sequences to perform further optimization of the model. (learn from more samples)

Here is a short video to show the optimization process. <CLICK> The convergence is quite fast with the help of initial predicted skinning weights.

Let's go ahead

Skinned LOD Results



Compare with Other Methods

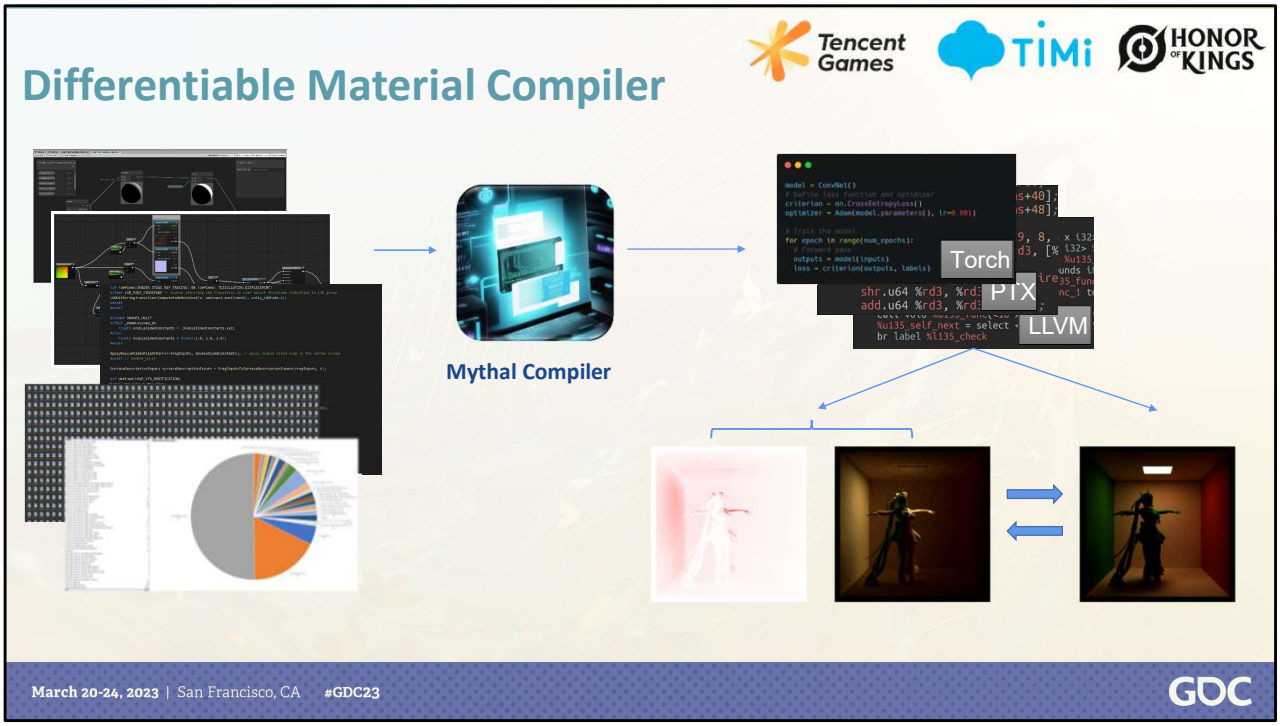
- Shapes are more desirably preserved around important region
- Necessary vertexes are preserved around joints
- High quality animation skinning is preserved

The comparison results indicate that when the same number of faces are used, shapes are better preserved in crucial areas such as faces and hands.

In addition, essential vertices are retained around joints to improve the overall skin quality.

<EOP>

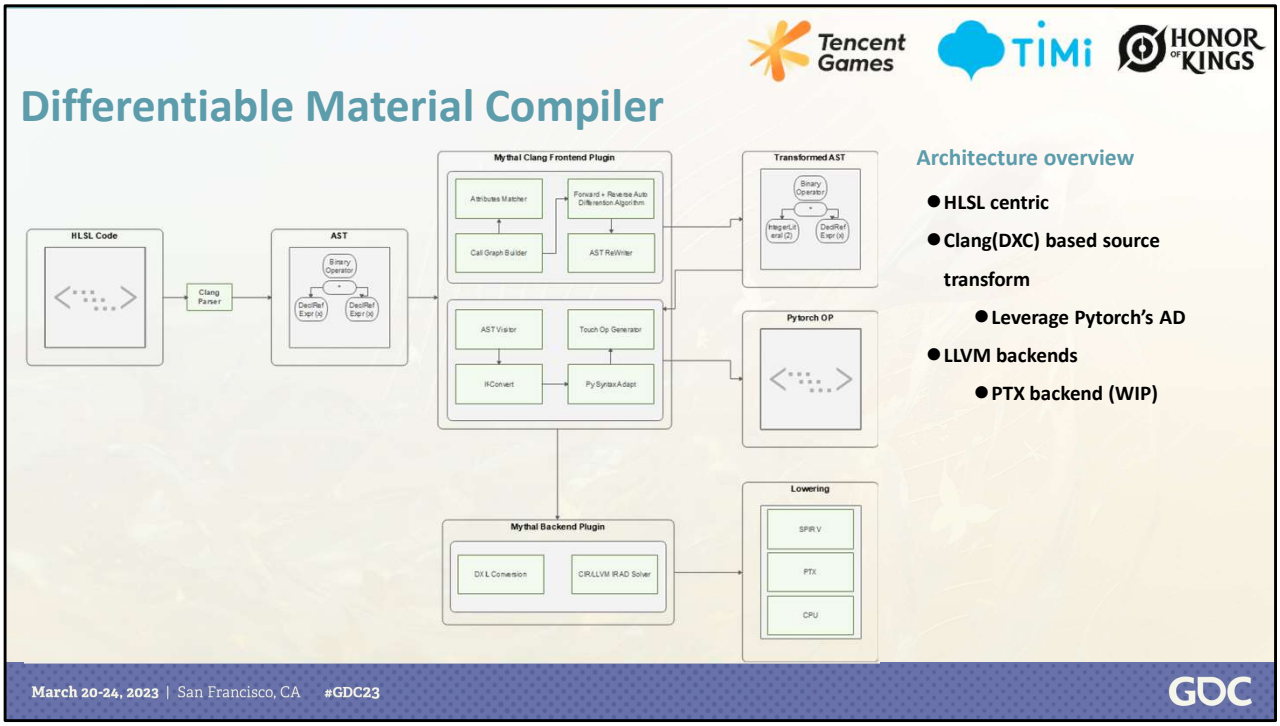
Let's turn our attention to the material compiler, another core component of our system.



Material complexity is a key issue in our projects, especially when it comes to differentiable rendering.
As we learned from Fei, the material shading part of differentiable rendering framework needs to be differentiable as well.

This has presented a challenge for us, as we need to come up with a plan to differentiate these shaders in order to make our rendering process differentiable.

So we have created a compiler on top of Microsoft's open-source project DXC.



This is the big picture of our compiler framework.

We have implemented a Clang-based source-to-source transpiler which transforms HLSL code to python
and we also have a work-in-progress LLVM backend that compile HLSL code to PTX.

In order to achieve differentiability, the transpiled Python code will make use of PyTorch's automatic differentiation (AD) framework to do the trick.

Let's dig into it.

Differentiable Material Compiler

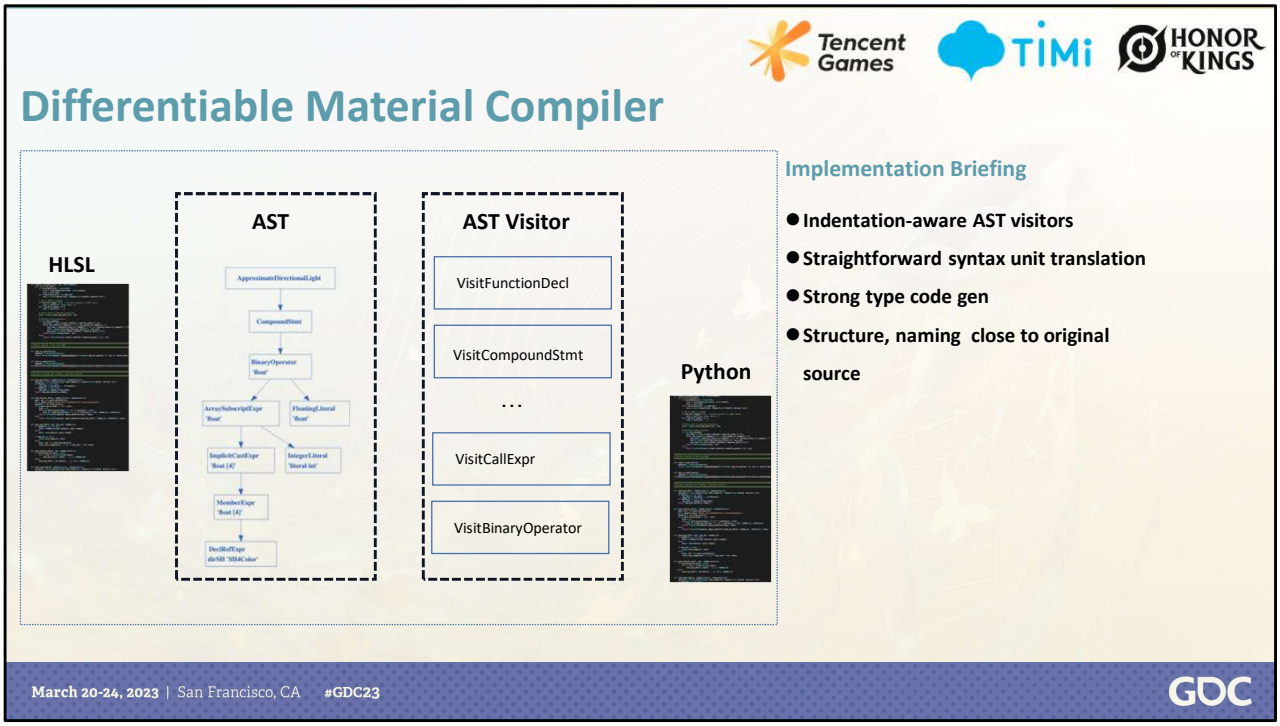
Implementation Briefing

- Indentation-aware AST visitors
- Straightforward syntax unit translation
- Strong type code gen
- Structure, naming close to original source

March 20-24, 2023 | San Francisco, CA #GDC23

Yes, you guessed right, my compiler work is truly busy. --- Because there are a lot of visitors I have to treat.

Joking aside, let's see how it works ☺

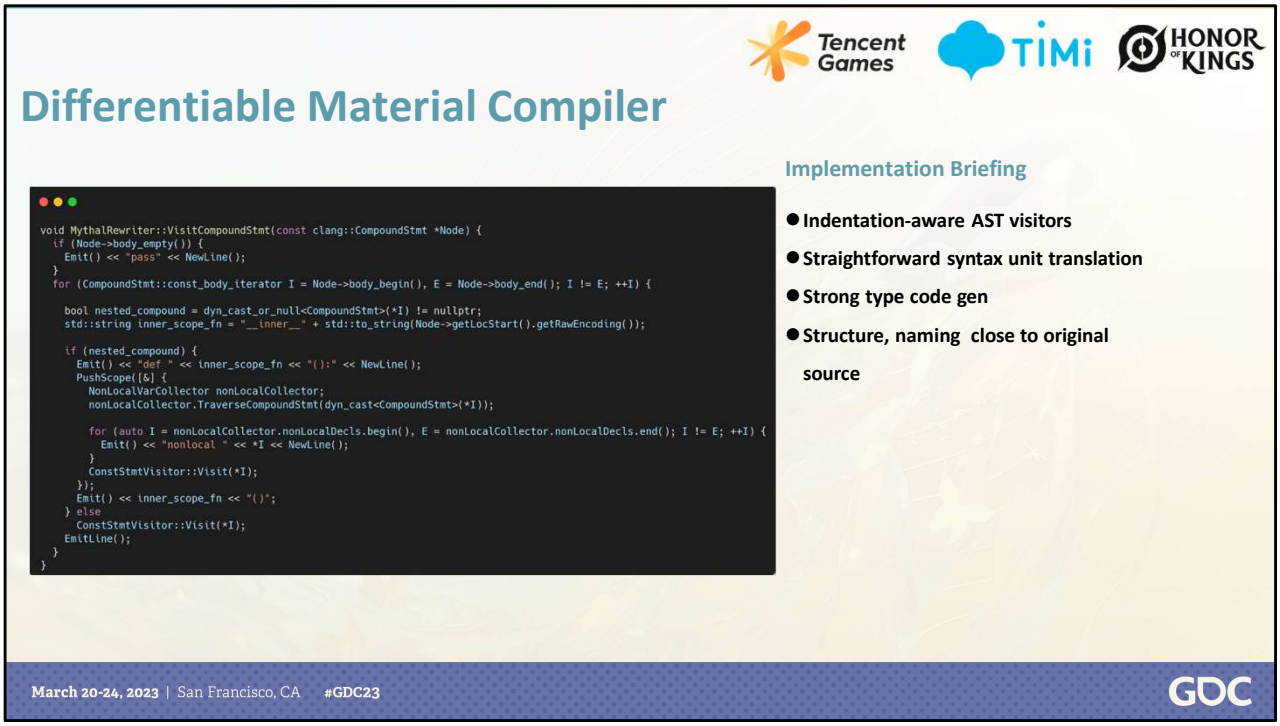


The implementation is quite straightforward, because the AST of HLSL and python shares a lot of similarities.

Of course, there are some differences we need to take care of,
like python usually can't pass function arguments by reference like HLSL's `out` keyword do,
and python don't support function overloading etc.

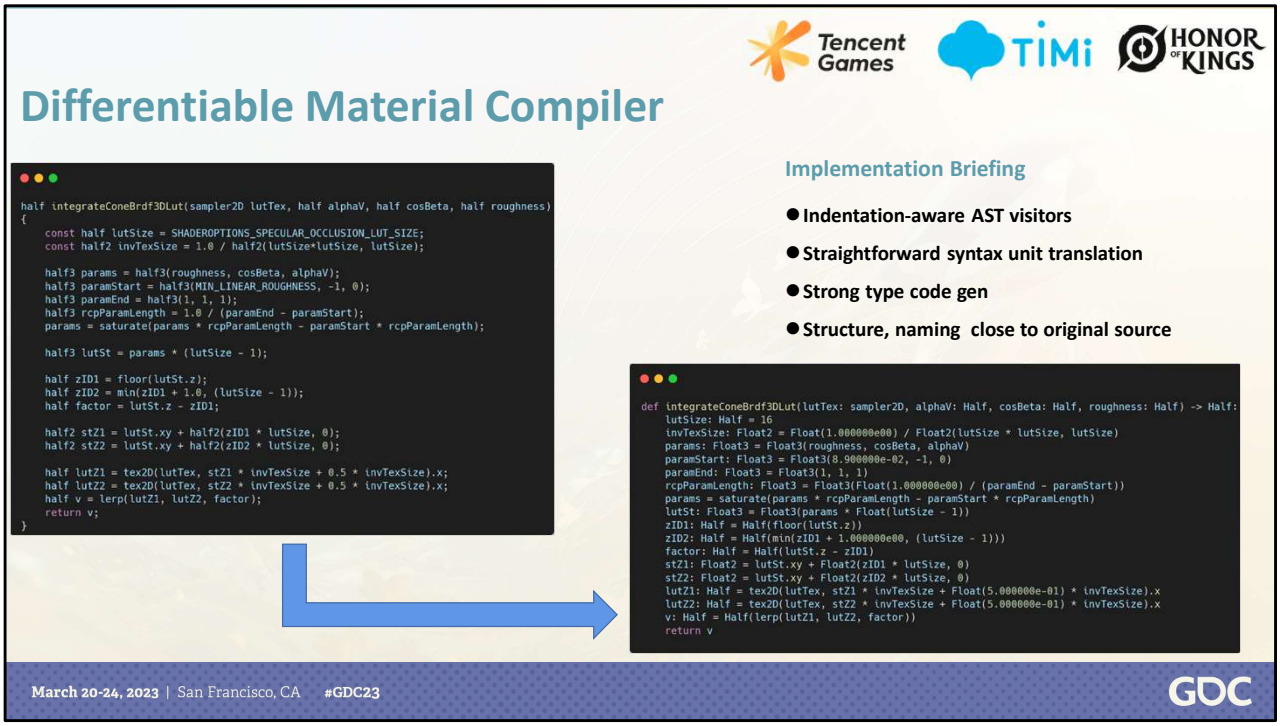
The former can be solved by python's multiple return value and structured binding.
And the latter can be solved by implementing a simple name mangler.
(If you guys are interested in the detailed implementation, I have left some bonus slides at the end of the presentation, we can talk about them later.

So, in brief



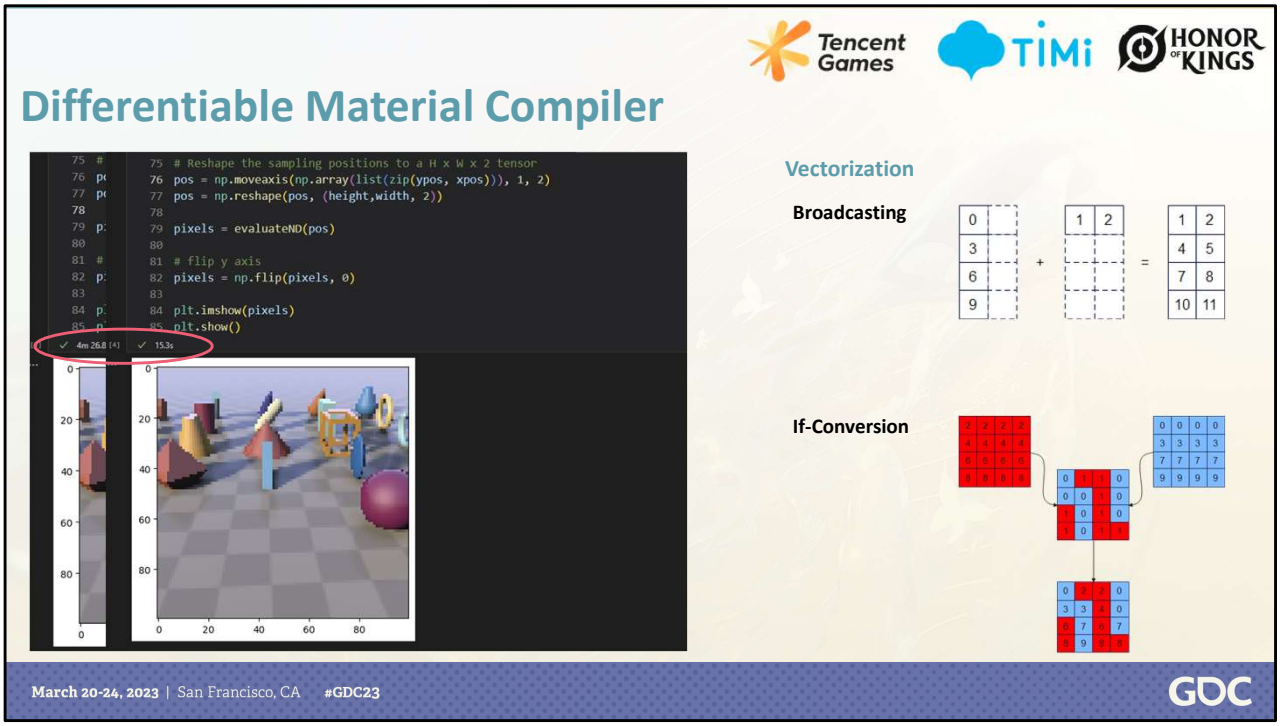
The overall design of our solution includes several key features:

Indentation-aware AST visitors, this allows us to accurately process the structure of the code and maintain proper indentations in the resulting Python code. TLDR: scope to indentation
<CLICK>



Strong typed code generation, with type information, the generated code is more efficient and easier to read, it also helps for later static analysis and jit code generation

And Our solution aims to preserve the structure and naming conventions of the original HLSL code as much as possible, making it easier to understand and debug.

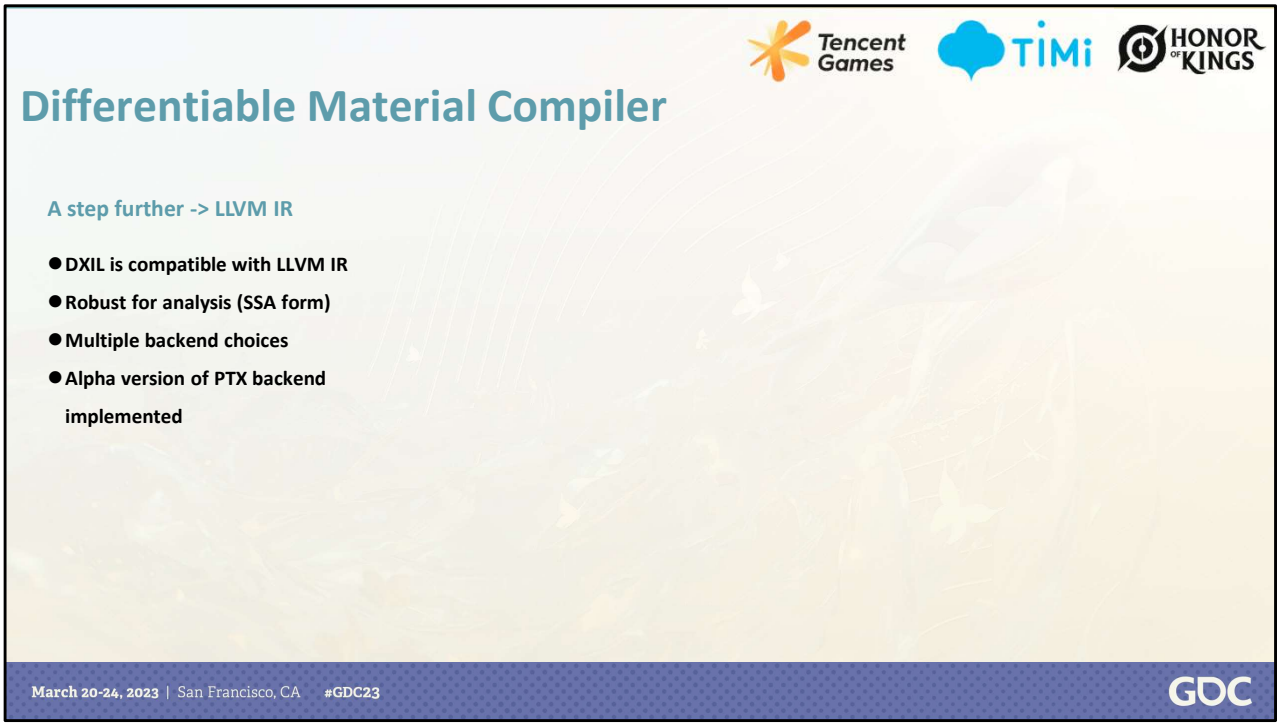


From this figure, you see we can use Python code to run Shader Toy. I used iq's SDF playground as a test case because it is complex enough to demonstrate our transpiler's functionality.

We know that shader code is executed in SIMT mode on the GPU, highly parallelized. However, the generated Python code is executed in scalar mode on the CPU and is therefore naturally very slow.

<CLICK>
So I have implemented automatic vectorization by if-conversion, which translates control flows into masked operations. (sound familiar? SIMD folks

We can see that the performance improvement is quite impressive here.

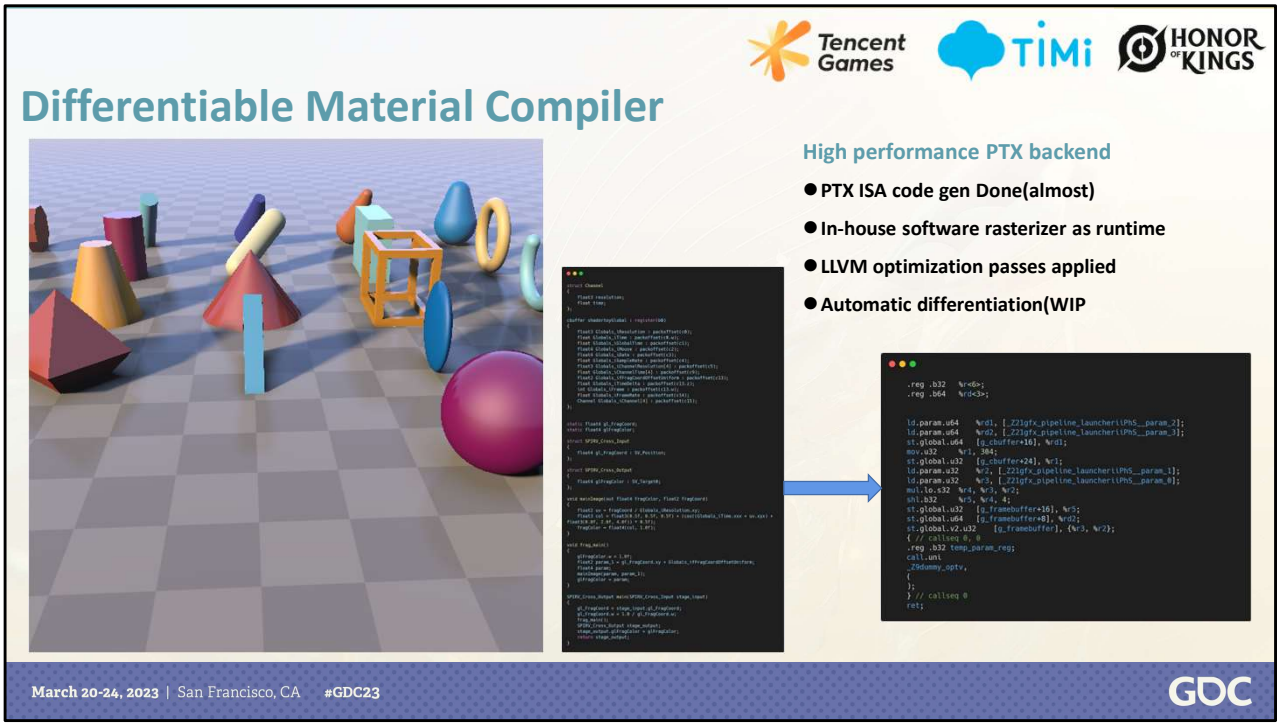


Though it's a good start, the transpiled python code is a bit verbose and it's hard to apply optimization algorithms on AST

We aim to make our solution more robust and performant.

Since we're on top of LLVM, and HLSL's ir format DXIL is compatible with LLVM IR, so existing SSA-based optimization algorithms can be applied.

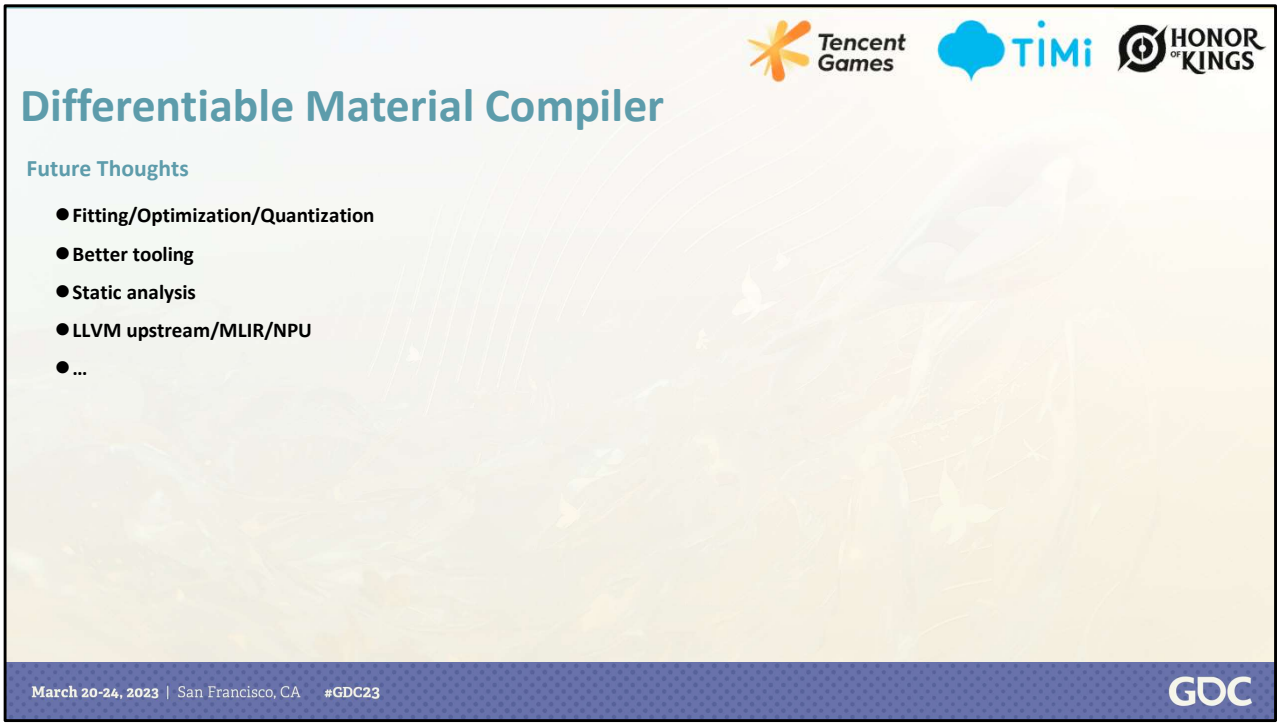
Therefore, we have an initial version of PTX backend implemented.



This is the result of our PTX backend, and it is currently working well.
Just kidding.

Those who have written CUDA programs may recognize this bug: it's caused by control flow divergence without proper synchronization.
<CLICK>

Currently, I have solved this issue by inserting sync points based on control flow graph analysis.
Though there is still room for improvement, like employing data flow analysis to make finer-grained sync point insertion, (or leverage memory coalescing)



Looking ahead, there are several potential directions we could take:

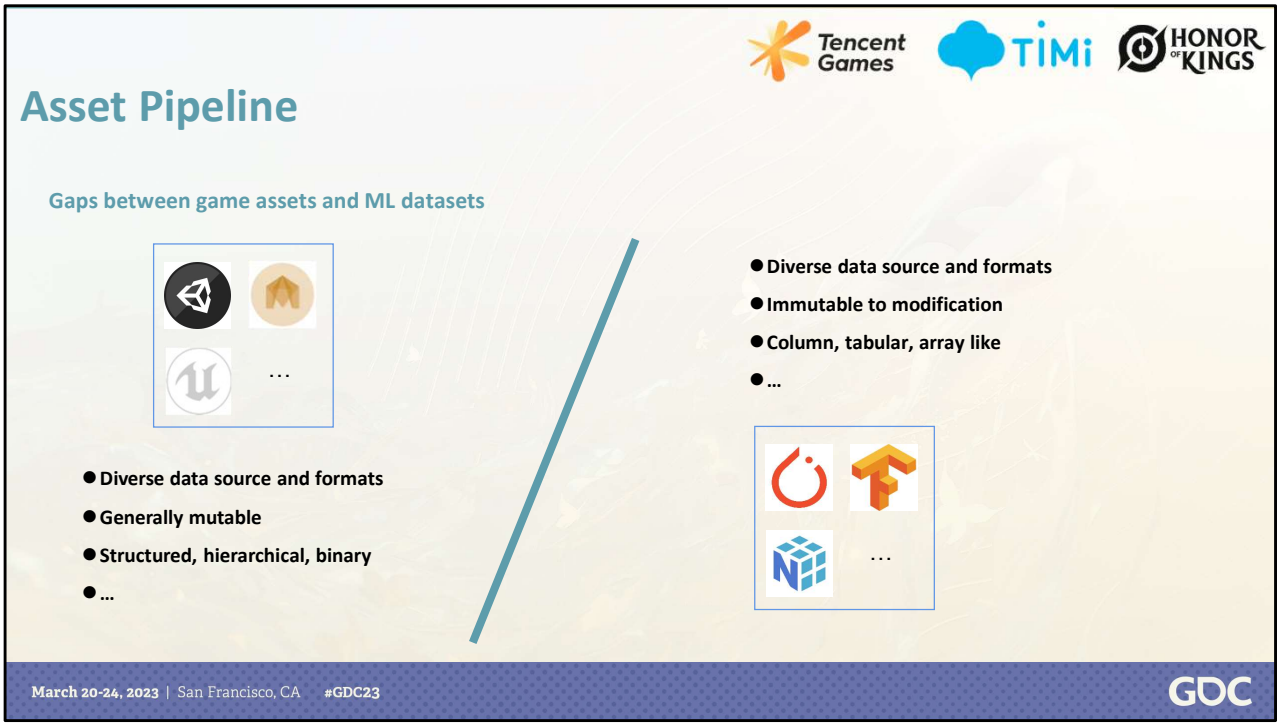
Since performance is a huge factor on mobile platform, so fitting, quantization and mixed precision calculation is certainly an important direction.

We could also bridge python's ecosystem to unlock more potential for us, like simpler visualization and better tooling.

We see Microsoft have already paved the way of bring DXC to LLVM mainline. This is a great step forward for the community.

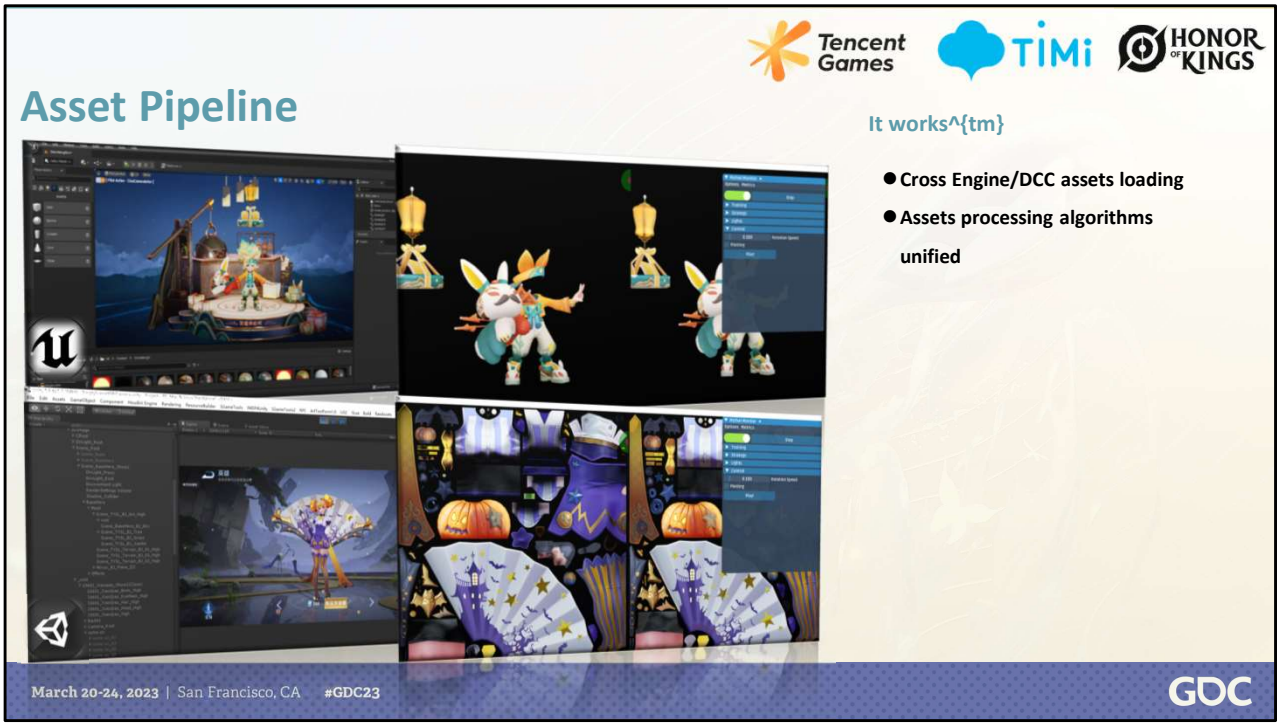
It also opens the door for us to research deep learning compilers for further optimization
And we could even deploy our models to the mobile on-chip NPU.

<EOP>

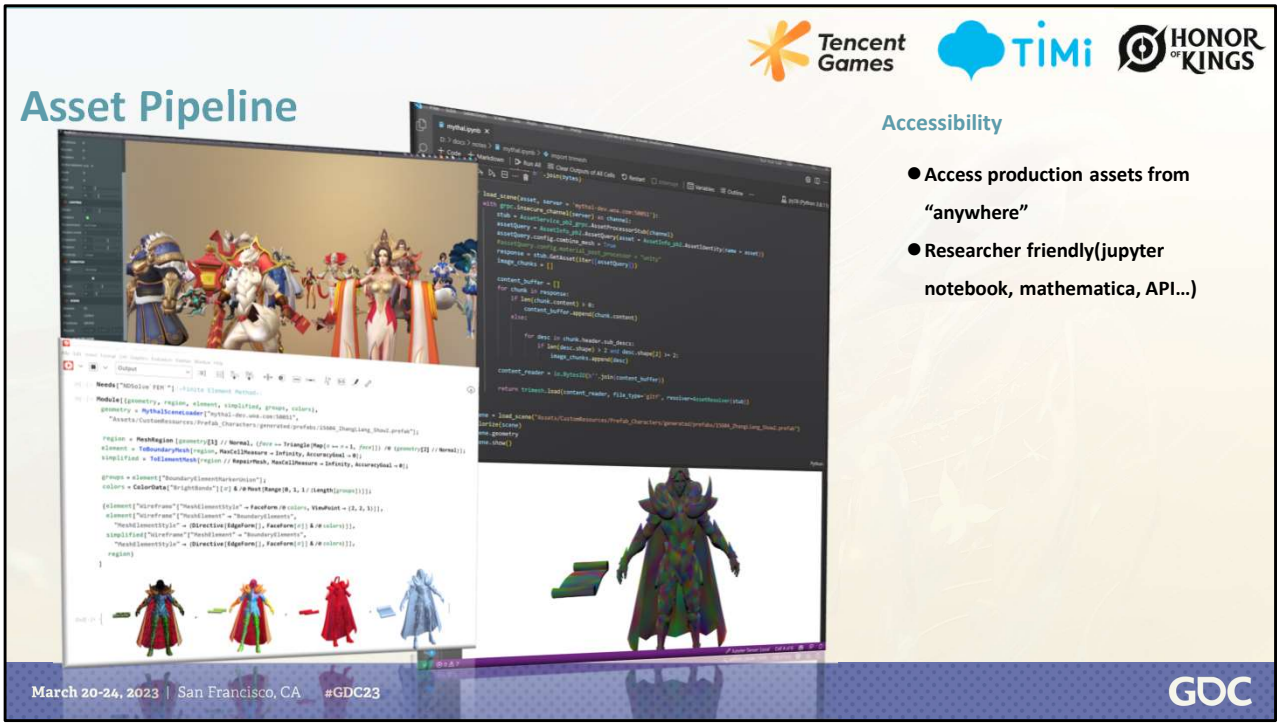


In the field of game development, the use of machine learning has been growing rapidly in recent years. However, one challenge that still needs to be addressed is the gap between game assets and machine learning dataset.
CLICK
Game assets, like 3D models and textures, are typically designed for real-time rendering and interactive use. They come in a variety of formats, produced by different game engines and DCCs.
CLICK
Similarly, machine learning datasets are generally structured in a different way. They're typically stored in tabular or array-like structures.

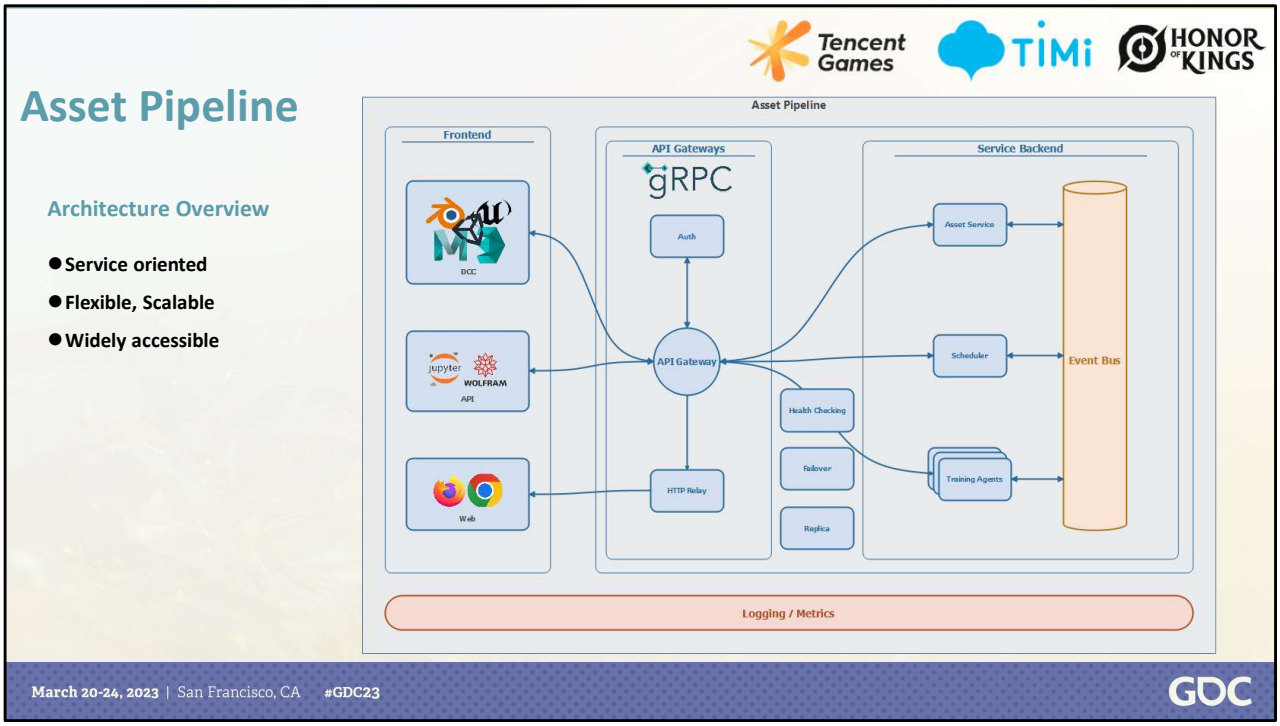
Filling the gap between game assets and machine learning datasets can be challenging, but it is necessary to achieve the full potential of machine learning in game development.



As demonstrated by these screenshots, you see that our system is able to perform machine learning tasks across various game engines.



Here are also some pictures showing that our game asset service offers seamless access to a diverse collection of game assets through multiple clients, including our machine learning agent, web browser, Jupyter notebook, and even Mathematica, etc. This convenience and versatility make it an ideal choice for research and experimentation, enabling quick and efficient prototyping.



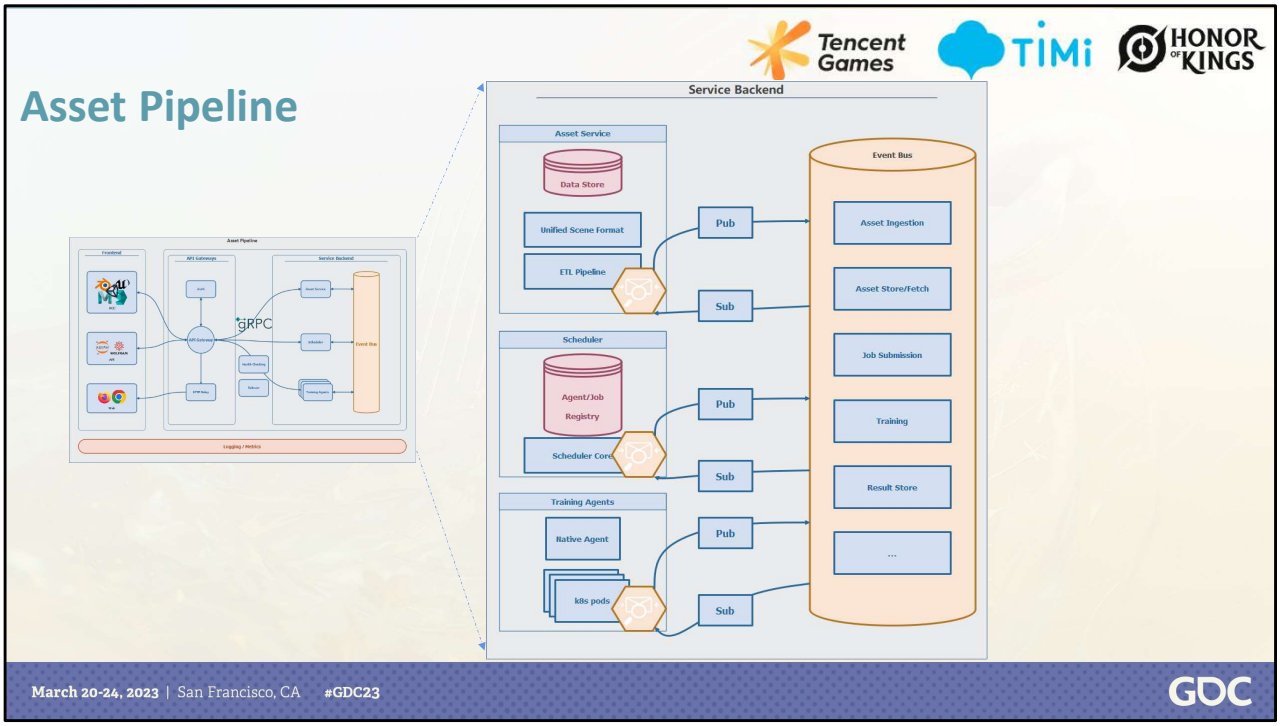
Let’s talk about the architecture.
You may have heard of microservice and how it’s becoming increasingly popular in building a service-oriented platform. I'm going to give you a high-level overview of it.

Our architecture has three major parts: frontends, a centralized gateway, and backend services. I'll go over each one in brief.

The frontend of our asset pipeline including DCC plugins, tools, and web browsers, etc.

This is made possible by a centralized gateway that simplifies API routing and reduces client-service coupling.
It employs gRPC as its standard communication protocol, offering improved streaming and accessibility, and we also have an HTTP relay for seamless integration with the web stack.

Lastly, the backend services run seamlessly behind the scenes and communicate with each other via the Event bus.
Let’s zoom into the **service backend** for more detail.

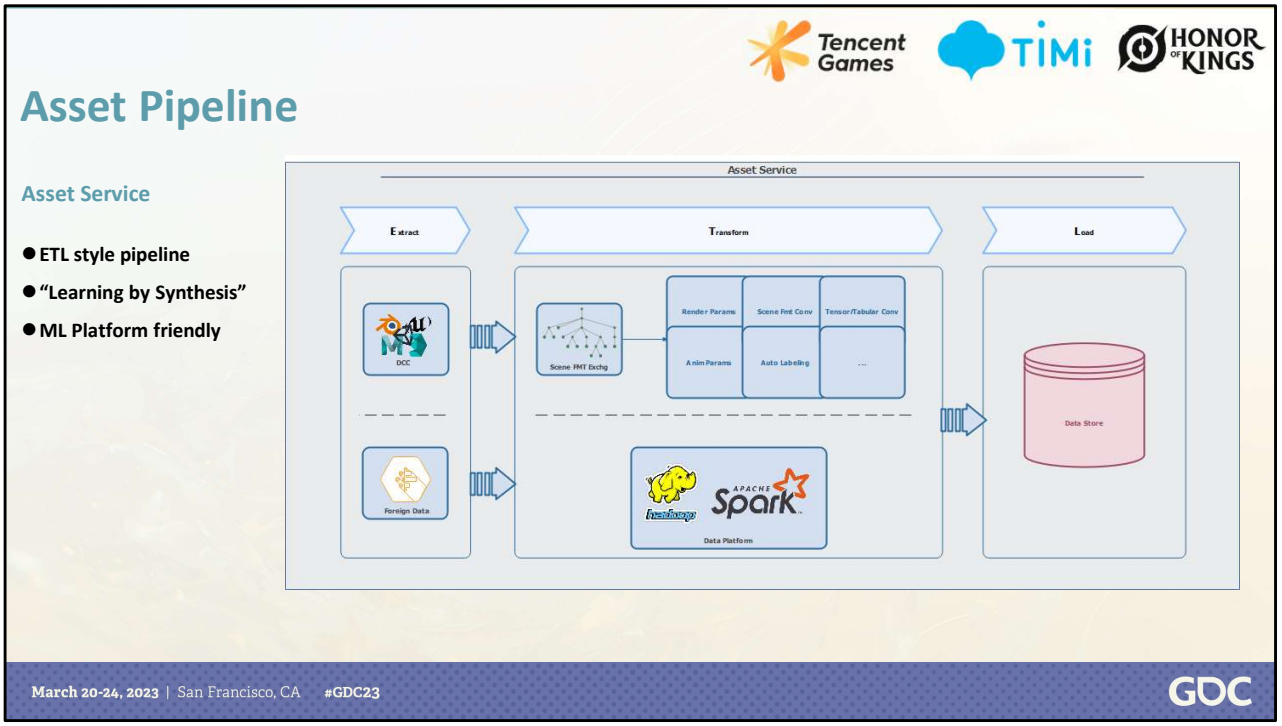


The service backend has three major services:
- The Asset service for asset processing
- The Scheduler service for training job scheduling
- The Training agent service for agent management

These services communicate through an event bus that uses a pub-sub pattern. This allows for asynchronous, decoupled communication, where each service can subscribe to relevant events and publish events to the bus.

For instance, the **Scheduler service** may publish a training job event which the **Training agent service** can receive through subscribing to it.

This way each service can evolve independently while the event bus ensures message reliability and durability, avoiding loss or duplication even during failures.



The role of the asset service in our platform is crucial.

As previously stated, data is of paramount importance in machine learning. Hence, utilizing game assets effectively is critical.

We have adopted the ETL framework as conventional machine learning platforms do to ensure data consistency, cleanliness, and efficiency for training.

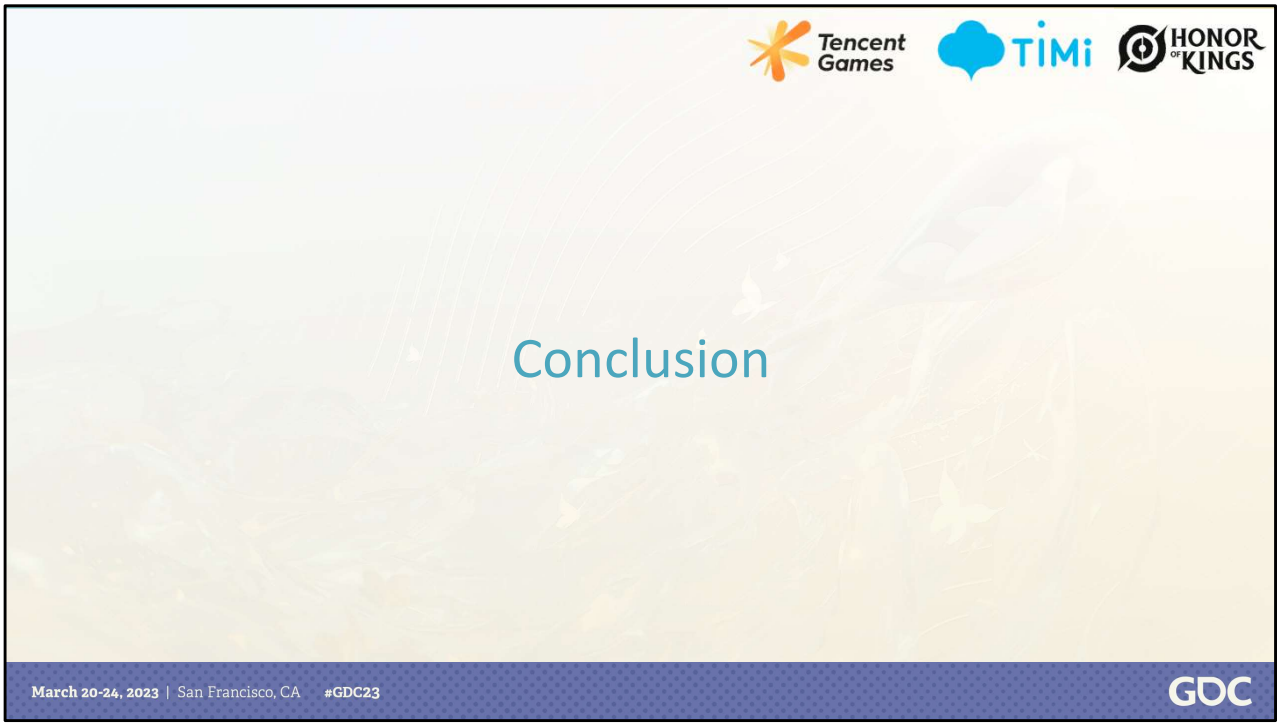
By leveraging the flexibility of game assets, we use a "learning by synthesis" approach to enhance the data set. Based on asset type, usage, or other metadata available in the game engine or DCCs, we can synthesize new scene parameters, character pose, camera views, and formats(like GLTF, USD), and we can even do automatic labeling based on this approach.

Btw. Our system is also compatible with traditional machine learning data pipelines, like Spark and Hadoop, etc., Making it even more flexible

So, my part is over.
I understand that the concepts presented may have been dense and numerous, but I

hope you have found the information helpful. Thank you for your attention and understanding, and please feel free to reach out with any questions or clarifications.

Now it’s Fei’s turn to give us a conclusion and takeaways



Welcome to the conclusion section. It is my pleasure to conclude today's key points. I sincerely hope that today's presentation was not boring and it provided you all with some valuable insights.

Productivity

Mythal(v1.0.0.0) has been deployed to produce high quality LOD assets for HoK to meet high demanding overseas environments

- We are able to achieve a 30-fold increase in production efficiency compared to traditional methods
- In terms of picture quality, high-quality LOD assets are almost authentic to non-professionals on mobile phones
- Visual error analysis-driven optimization algorithms have the potential to continue evolving by incorporating the expertise of art professionals

March 20-24, 2023 | San Francisco, CA #GDC23

Before I summarize, let's take a look at some of HOK's actual achievements.

We have used Mythal to create a lot of LOD assets that meet the demanding requirements for a range of overseas environments.

Our new technique has increased production efficiency by 30 times compared to traditional methods, while delivering high-quality LOD assets without much loss in graphics quality.

It ensures that HOK will meet the highest visual standards for various players and devices worldwide.

In addition, we can improve the system's visual error analysis by continuously learning from human experience.

Results

- With 2080 graphics card, 1.8 million vertices, 600,000 triangles, 1024 resolution, HTBR rasterization process perform less than 1ms
- Rendering result is utterly the same with hardware rendering
- Using wavefront ray tracing, an iteration typically takes 1.4 seconds with 3 bounces and 64 spp

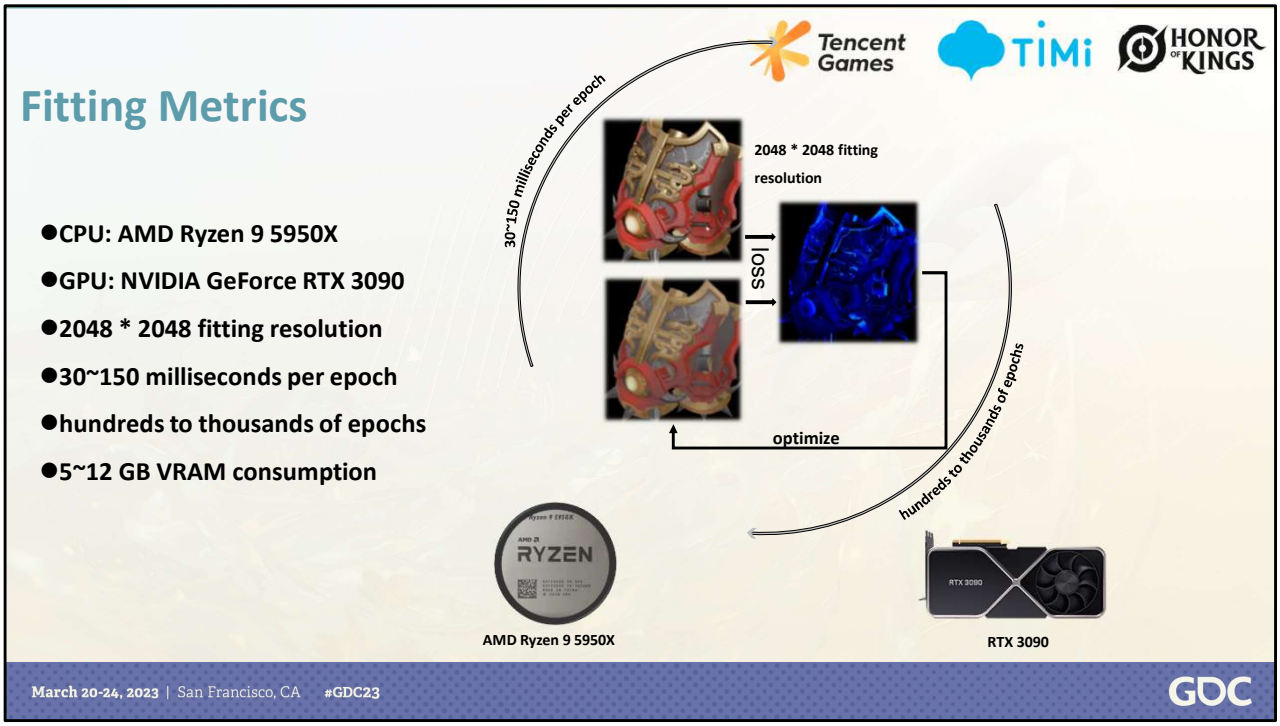
Our simulated rendering pipeline has been able to achieve the same rendering results compared to the references using hardware graphics APIs.

Although our method may not be as fast as its references in forward rendering, it still performs quite well.

It's important to emphasize that the tradeoff of forward rendering performance here is valuable. It gives the ability to freely extend gradient backpropagation algorithms. For instance, we easily added a more efficient custom gradient method on blending operations.

This makes the system more potentially in reverse rendering than other methods.

However, if we need to further increase the rendering speed, as we learned before, our system is distributable and heterogeneous, we can simply add more GPUs or other devices to the system.

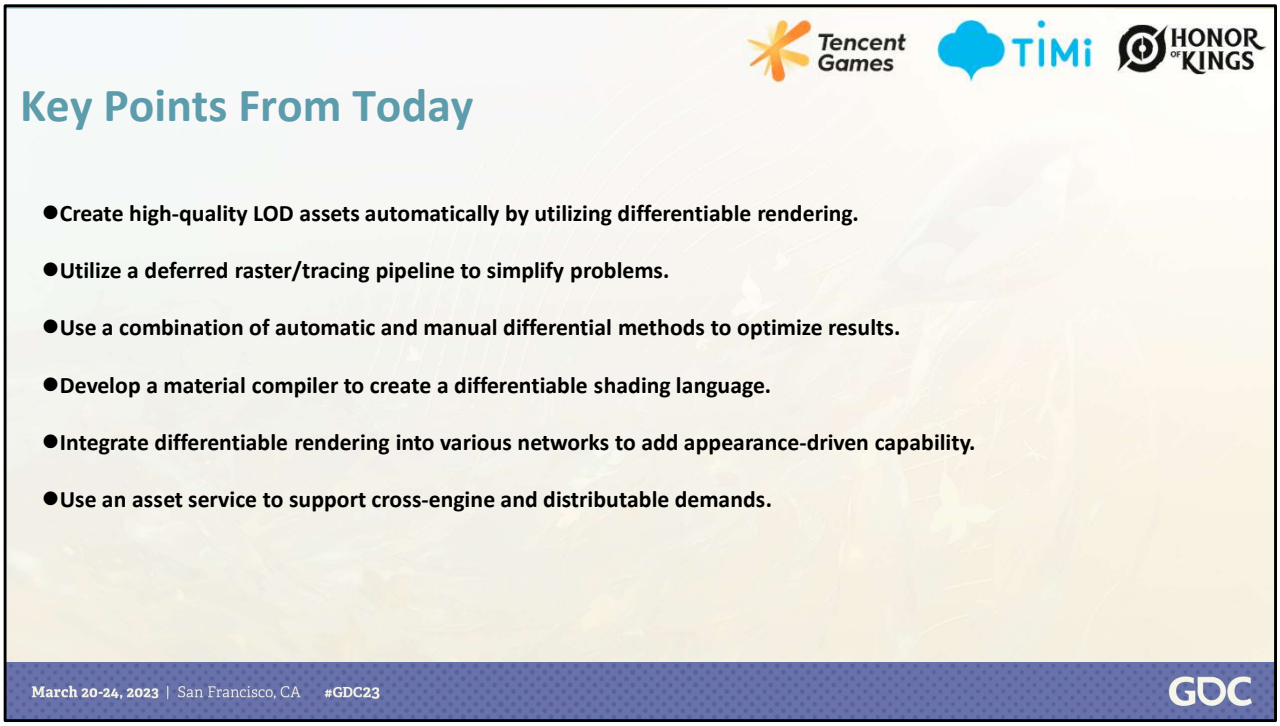


We use a powerful AMD Ryzen 9 series processor and an RTX Compatible graphics card to run the auxiliary asset production system efficiency.

For single fitting tasks, with 2k fitting resolution, our system requires around 5 to 12 GB of video memory.

During each cycle of our system's operation, called an "epoch," it often takes anywhere from 30 to 150 milliseconds to complete.

After runs for hundreds of epochs, the system usually produce the best possible LODs from the origin game assets.



Okay, It’s time to wrap up today's presentation. I'd like to give you a quick summary.

First, with modern technology, we can now free our artists to create without worrying much about performance. We can also use gradient-based methods such as differentiable rendering to automatically generate the best LOD assets for games.

[click]When building a differentiable rendering pipeline, we can simplify almost everything by using the deferred concept, both on raster and raytracing rendering pipelines.

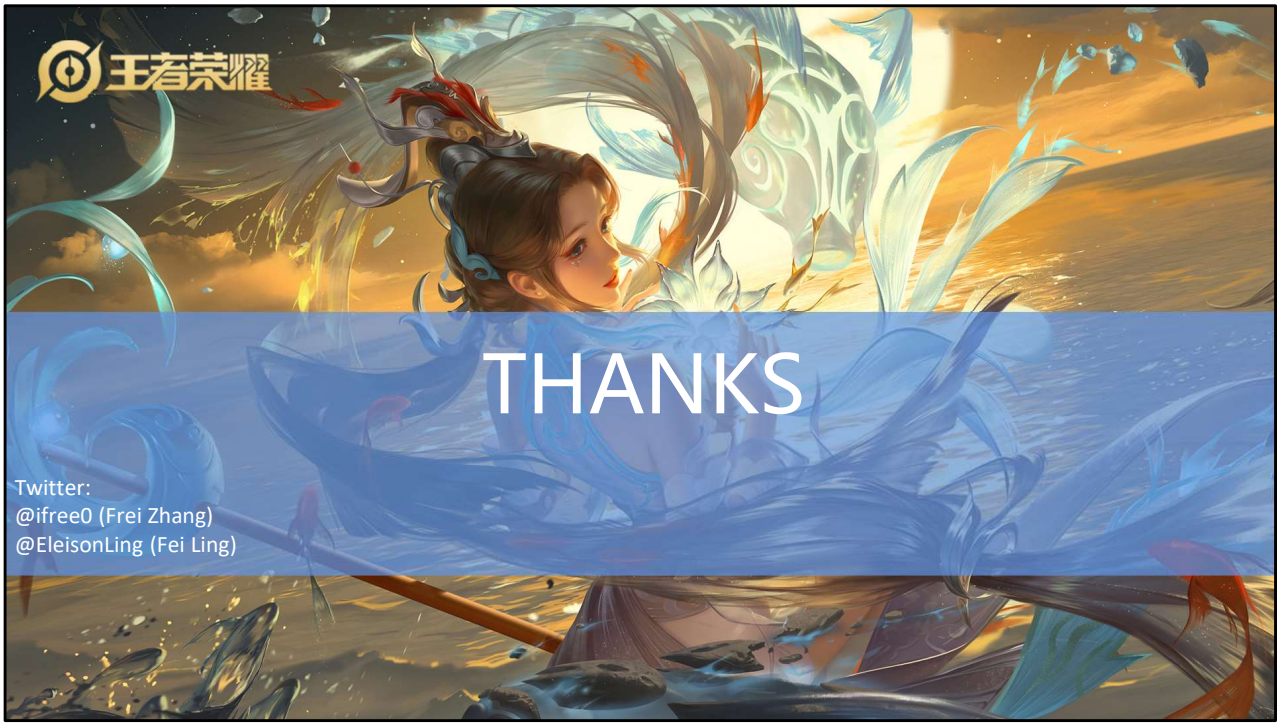
[click]For the best results, we can use a combination of automatic and manual differential techniques, depending on the situations.

[click]An in-house differentiable material language compiler can be a big help in handling highly customized materials. Following LLVM and DXC will give you to a great start.

[click]It is a great idea to combine differentiable rendering with other AI techniques. Differential rendering can greatly enhance the AI network’s performance due to add

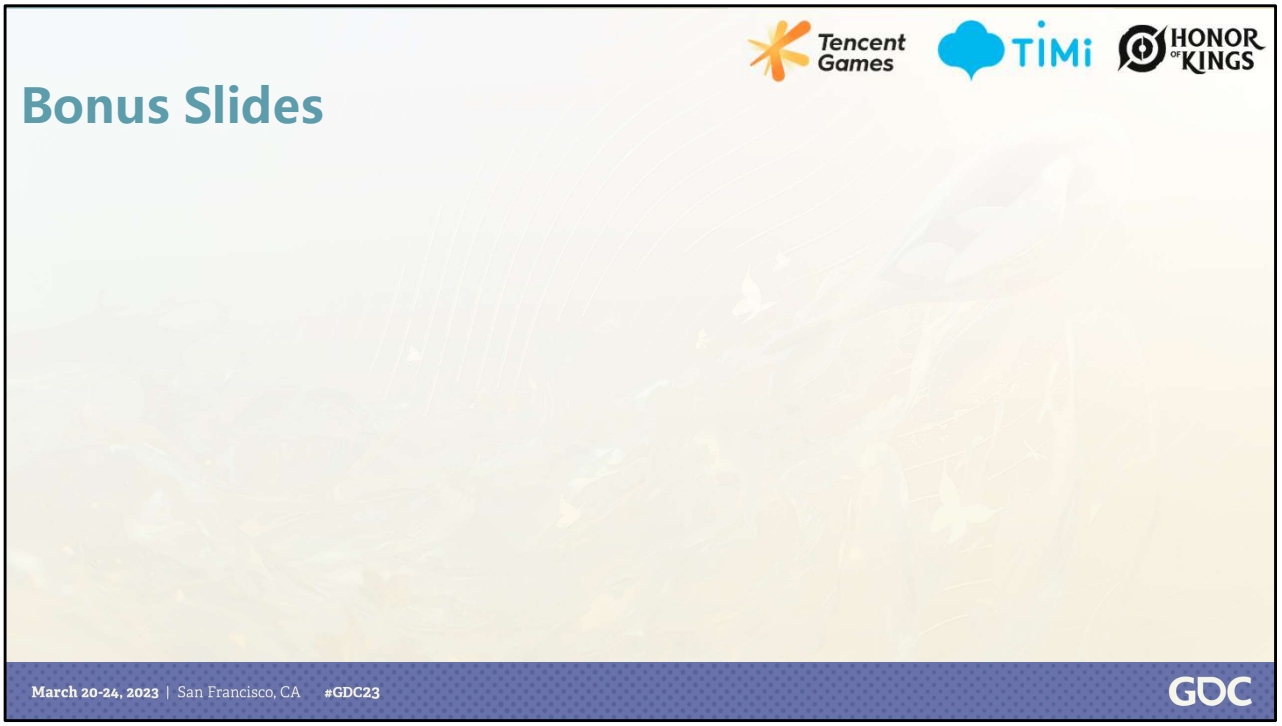
some important appearance-driven features to them.

[click]Finally, It will be a valuable addition work by incorporating some industrial designs such as game asset services. It can be helpful to tackle many complex fitting tasks.



That’s all for today, Thank you for taking the time to join us.

I think we don't have much time left for onsite QA today, but feel free to ping us on Twitter if you have any questions.



Here comes the corner cases as I previously mentioned.

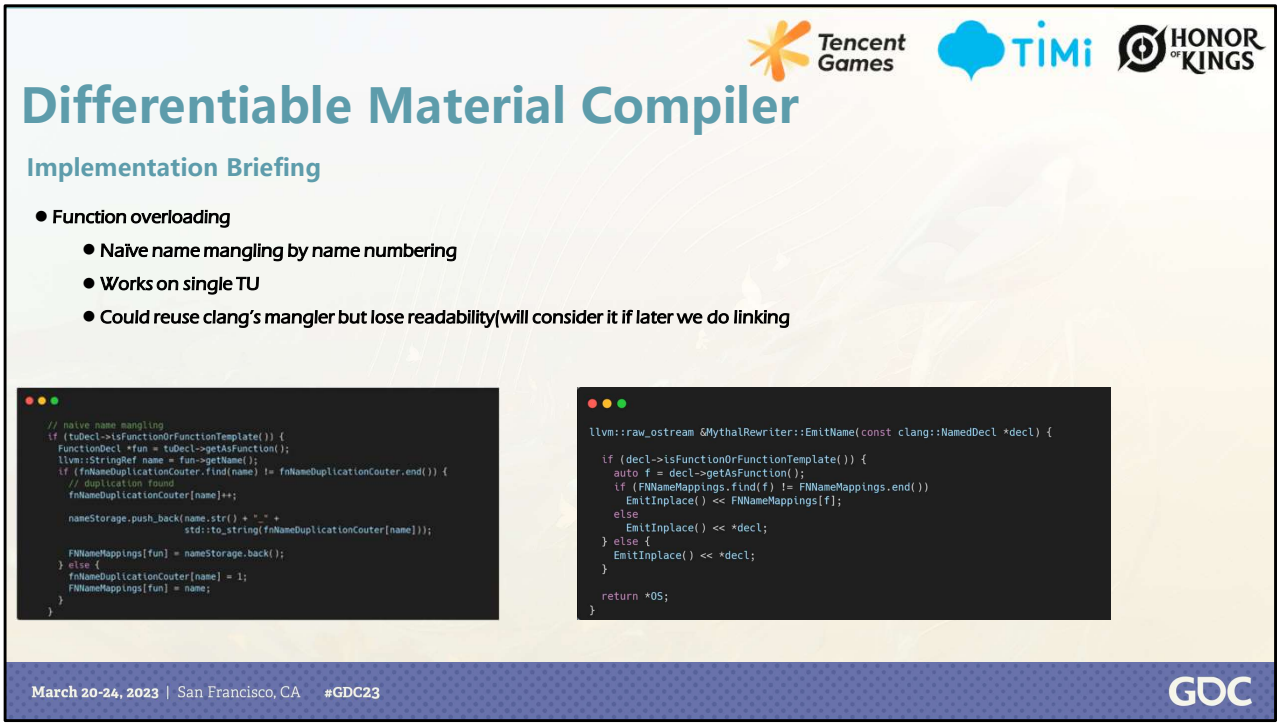
One of the challenges we faced in our solution was handling function overloading, which is not supported in Python.

To address this issue, we made use of Clang's built-in name mangler. While this effectively solved the problem of function overloading, it resulted in excessively verbose function names that were difficult to read.

As an alternative, we implemented our own simple name mangling approach, which gives an index-based suffix to function name to distinguish it from others. This helps to maintain the readability of the code while still allowing us to handle function overloading.

The code is simple enough to be self-explanatory.

1. Group functions by name, each group member has a unique index
2. The code gen part will check the group when output function name



Here comes the corner cases as I previously mentioned.

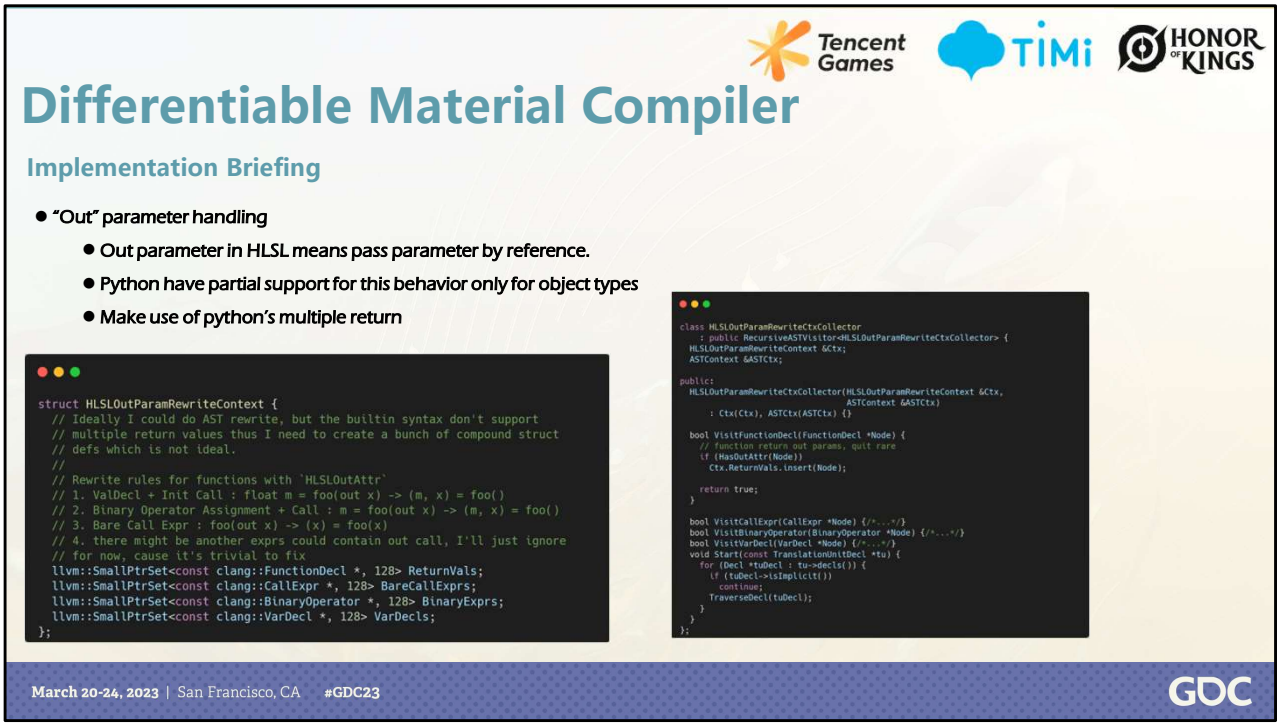
One of the challenges we faced in our solution was handling function overloading, which is not supported in Python.

To address this issue, we made use of Clang's built-in name mangler. While this effectively solved the problem of function overloading, it resulted in excessively verbose function names that were difficult to read.

As an alternative, we implemented our own simple name mangling approach, which gives an index-based suffix to function name to distinguish it from others. This helps to maintain the readability of the code while still allowing us to handle function overloading.

The code is simple enough to be self-explanatory.

1. Group functions by name, each group member has a unique index
2. The code gen part will check the group when output function name



Another challenge we encountered was handling "out" parameters in HLSL, which means pass parameters by reference.

While Python only support this behavior for object types to some extent(object mod but no assignment), we certainly needed a more robust solution.

To address this issue, we made use of Python's ability to return multiple values from a function.

This allowed us to effectively emulate the behavior of "out" parameters in HLSL and ensure that our code was correctly handling these types of parameters.

There are several conversion rules you could see in the comments.