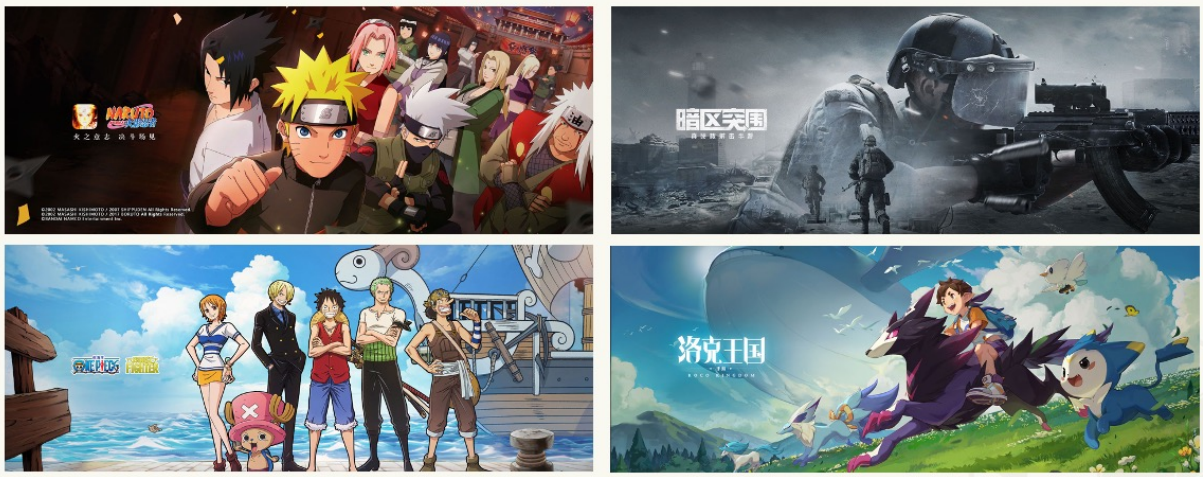
Good evening every one,

This is KM Chan and RC Ying of MoreFun Studios at Tencent Games

We belongs to the game engine team

Today, we would like to share details on the weather system for our upcoming mobile game, Project: Dark

MoreFun Studios



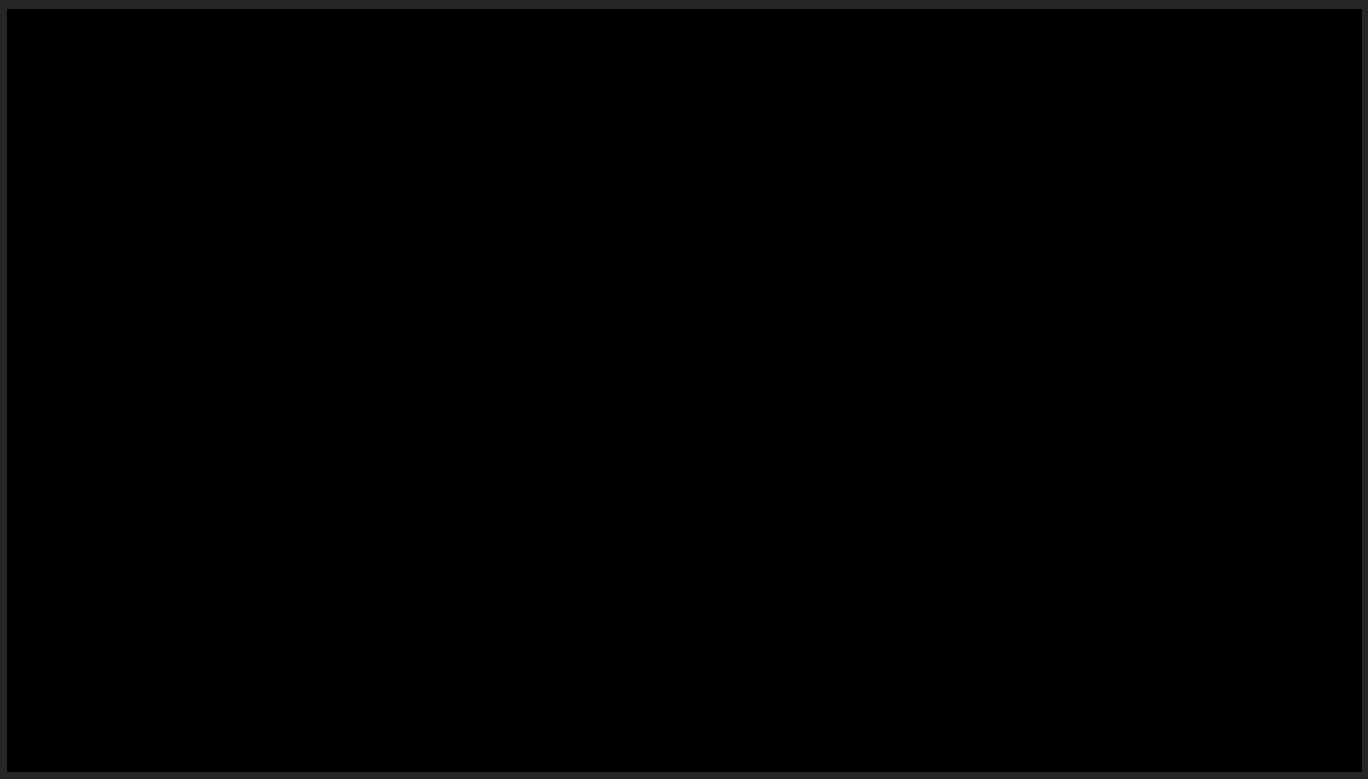
More titles can be found at https://morefun.qq.com/

First of all, let me introduce our studio.

More Fun is one of the Tencent Games' studios and she was found around 2010

We had made a number of titles based on Chinese and Japanese anime IPs

Recently, we will launch a mobile FPS, and let's take a look at the trailer.

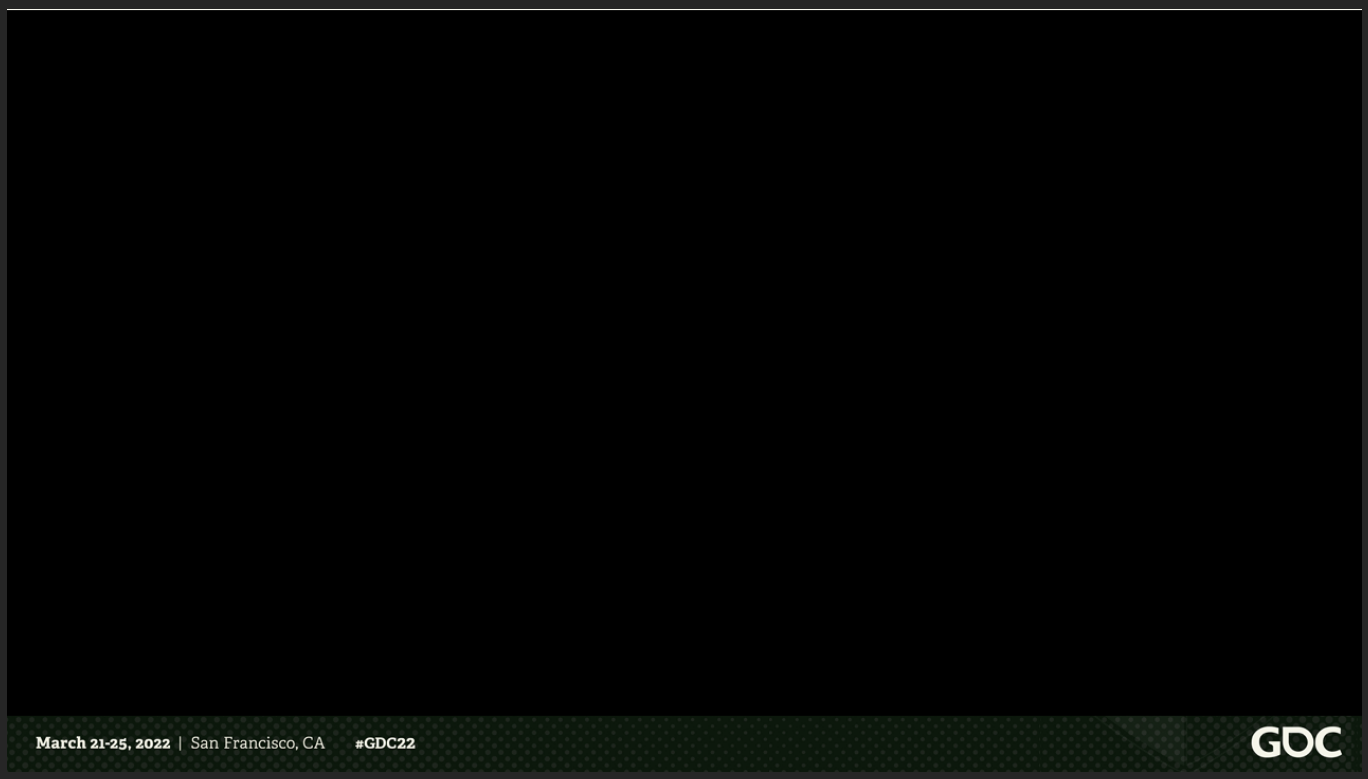


This is our first realistic shooting title, and we have learnt a lot during the development process.

Our game is based on heavily customized UE4 to put in a lot of console grade features.

For example, dynamic global illumination, in-game climate and time of day changes;
All these features are trying to make the PVP system more interesting and attractive.

[Wait until finish]
And here is another short video shows how our weather system looks like
[Click]



Instead of static sky box, our weather system do support real time volumetric cloud and atmospheric scattering.

Also the sky will influence the scene lighting, everything is dynamic.

Agenda

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

- **Portable Dynamic Sky Rendering**
  - Atmosphere
  - Volumetric Cloud
- **Extensions and Weather Effects**
  - Cloud Shadow
  - Rain
  - Lightning
- **Designs of weather system (depends on time)**

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

GDC

So, let’s get started with the technical details

[click]

First, we will focus on the dynamic sky rendering solution in our weather system

This include atmospheric scattering and volumetric clouds and the related optimizations for bring them to mobile.

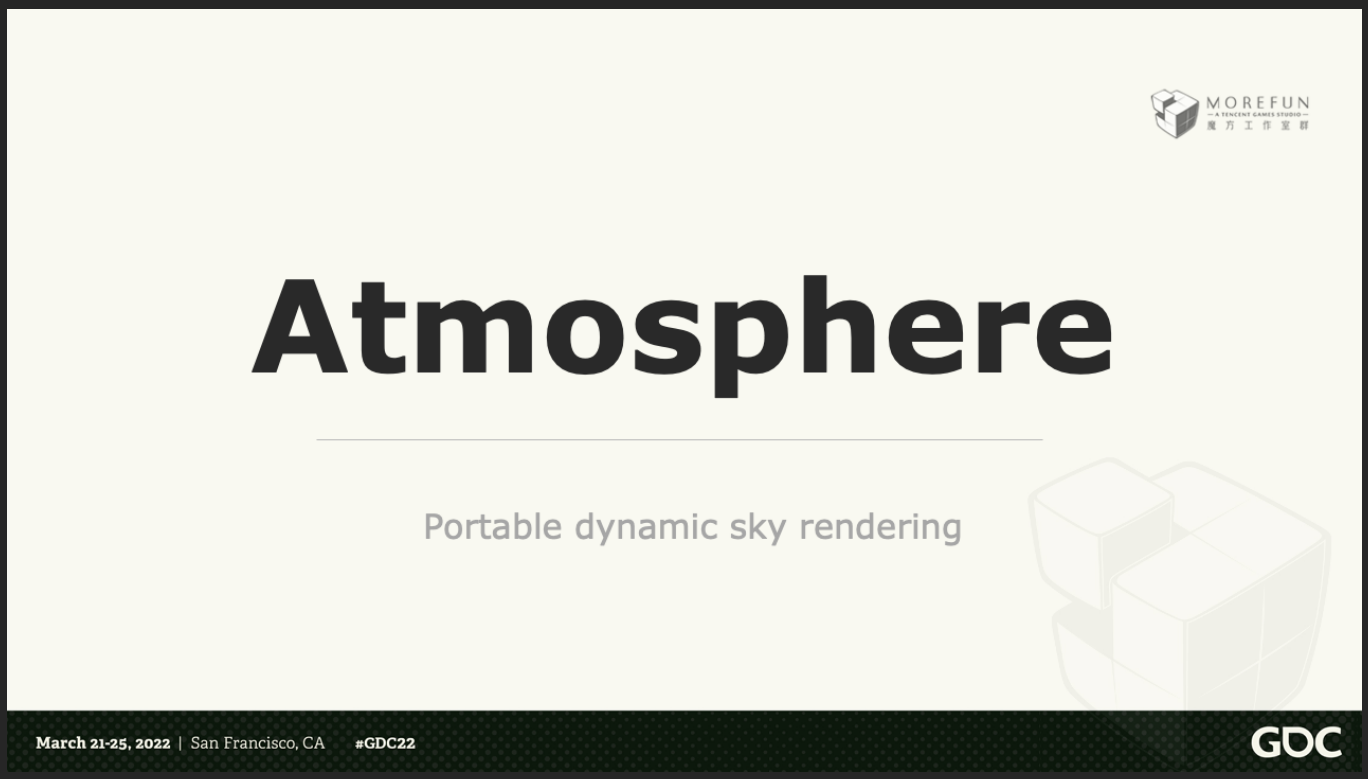
Please notice that our techniques also applies to portable gaming consoles like Nintendo switch and Steam deck.

[click]

After that we will shared some of the weather effects which are related to the sky

[click]

If we still have time, we will cover some bonus slides about the design of our weather system.



First I will take about the atmospheric scattering and the sky background

About the atmosphere

MOREFUN
— A TENCENT GAMES STUDIO —
魔方工作室群

- **Our atmosphere consist of particles or molecules**
- **Sky Color: determined by scattered sun light**

Rayleigh Scattering
Molecules (air) smaller than the photon wavelength
Blue Sky

Mie Scattering
Molecules (dust, pollen) with size similar to the photon wavelength
White Fog and Haze
Pale blue in Lower Sky

Non selective Scattering
Molecules (water) much larger than the photon wavelength
White Cloud in Blue Sky

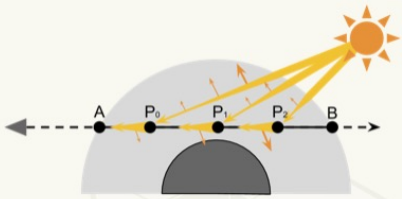


Diagram from
https://www.alanzucconi.com/2017/10/10/atmospheric-scattering-1/

Atmospheric scattering is complex, which involve lots of physics.

So, I will try to explain it in a simple way.

Our atmosphere consist of different particles or molecules. The sky color is determined by sun light being scattered by these particles

[click] For molecules smaller than the photon wavelength, Rayleigh scattering occurs which favour blue frequency lights, and that's why we have blue sky

[click] For molecules with similar size, **Mie** scattering occurs and all frequency of lights are scattered equally, this makes haze or fog appears in white color or pale blue at the bottom of the sky

[click] For molecules that are much larger, **Non selective Scattering** occurs. For example the water in liquid or ice form which make up clouds. They scatter all frequencies equally and so clouds appears white in color while the sky is blue.

[click]

In order render the atmosphere, we have to ray march among the view direction (say from point B to A), calculate the amount of scattering reaching each sampling point P_i

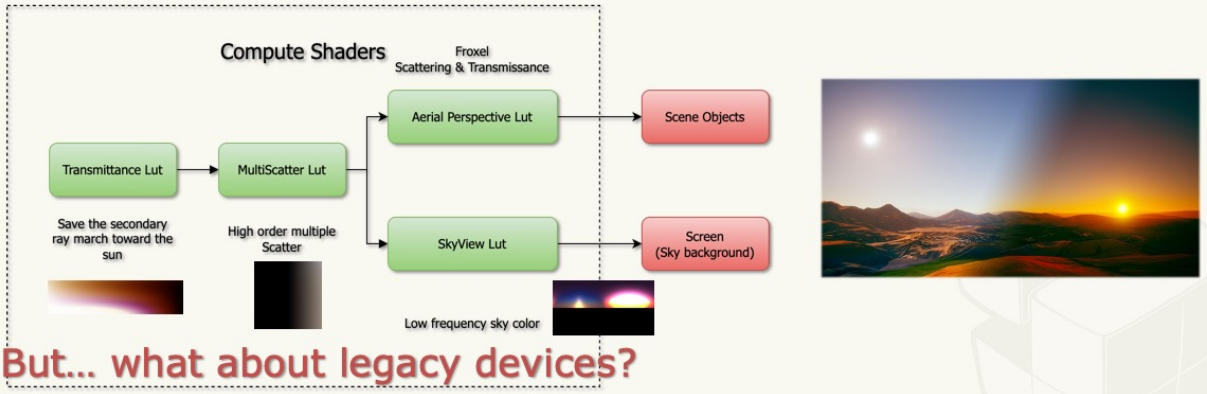
And then integrate them together.

However it is too heavy for games and real time applications, so we usually use 3D / 4D look up tables to speed up the process.

UE's Sky Atmosphere

[Hil 20] Physically Based and Scalable Atmospheres in Unreal Engine

- Physically based and artist friendly
- Good Performance on high-end devices (LUT based)



But… what about legacy devices?

Let's take a look at Unreal Engine's solution

In UE4, she use a series of look up tables to greatly reduce the amount of calculations

Simply put, she will generate 4 essential LUTs using compute shaders in each frame, they correspond to

[Click] Transmittance LUT which contains the shadowing information

[Click] MultiScatter LUT for fast high order multiple scatter lookup

[Click] The sky view LUT which precompute the color for the distance sky.

And there is froxel of pre-computed the scattering & transmittance values mainly use by aerial perspective effect

[Click] The scene is rendered by looking up these LUTs and here is an official screenshot.

It is physically based and artist friendly, plus good performance on high-end mobile devices

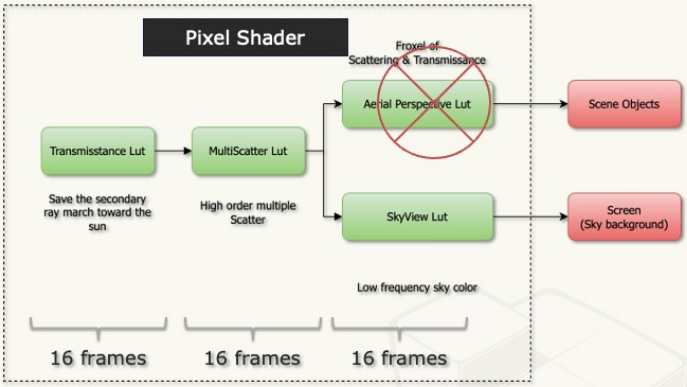
For example, it takes ~0.2ms on iPhone 11.

[Click]But what about legacy devices?

Where they have very limited bandwidth and we just can’t afford so much LUT updates per frame?

Optimizing for Legacy

- **Remove Aerial Perspective LUT**
- **Use PS for better compatibility**
- **Separate LUT calculations (across 48 frames)**
- **SkyViewLut -> hemi-octahedron**
  - Below the horizon is always black in our game
  - Faster ALU ops when look up
- **Evaluation can be done in CPU! (0.5ms)**



So we did some optimizations to UE's implementation:

[Click]

First we discard the aerial perspective data, since we only concern about the distance sky background and we use height fog for our scene instead of aerial perspective.

[Click]

All remaining LUTs are 2D, so we can now use pixel shader to update them.

This is very important since many mobile devices still have poor computer shader support.

[Click]

Then we evaluate the each LUT across 16 frames and 48 frames in total, so we only evaluate a few pixels in each frame and this is acceptable on low end devices.

This is ok for our game since time of day is changing gradually

[Click]

To further reduce the amount of calculations, we use hemi-octahedron parameterization for the SkyViewLUT and discard everything below the horizon.

This not only save 50% of ray marching and also save the expensive square root instruction while looking up the sky color.

[Click]

Surprisingly, we found that these updates is fast enough to be done in CPU!

So we did it for legacy devices.

Here are some comparisons between the Original and Optimized version, at 3 different time of day.

[click]

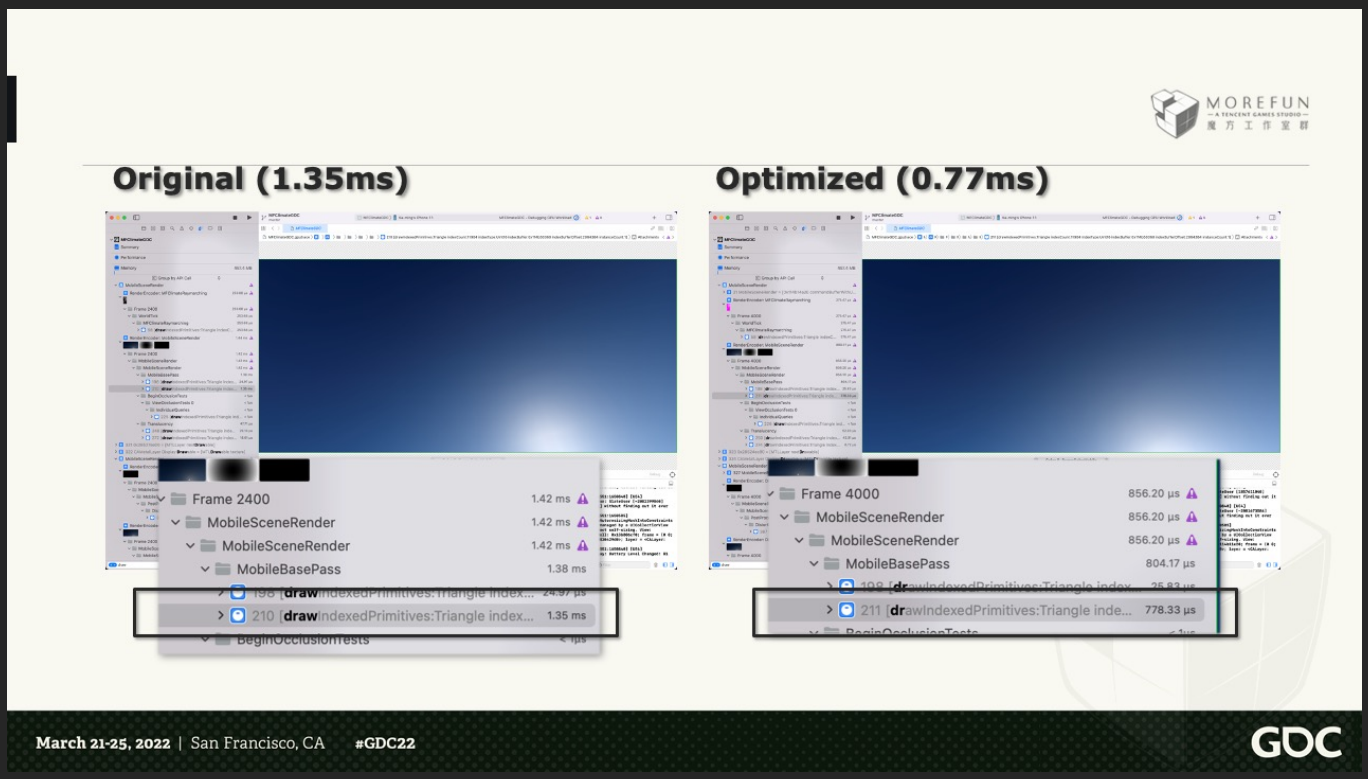
[click]

[click]

There are some differences around the sun disk, because we use a relatively low resolution for the LUT

but our artist think this is acceptable

and the performance is significantly improved as shown in the next slide



We can see hemi octahedron projection saved nearly 40% of GPU time while rendering the sky

Sky intensity changes

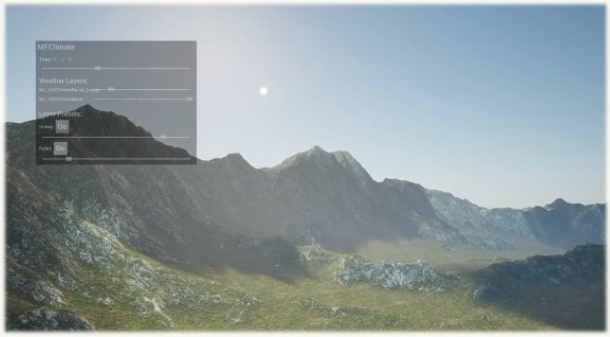
MOREFUN
— A TENCENT GAMES STUDIO —
魔方工作室群

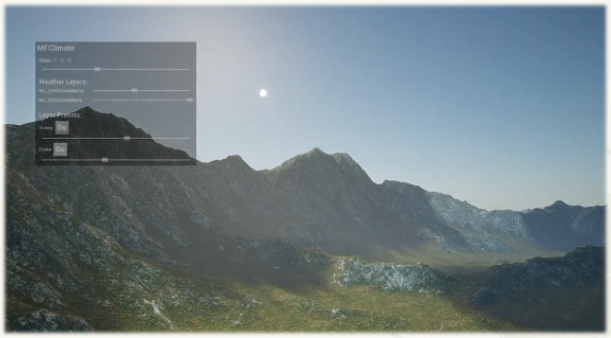
Artists may need to dim the sky for dramatic effects

- **DO NOT adjust the sun intensity!**
- Physically incorrect
- Cause flickering

Use constant sun intensity for atmosphere

- **Sky**: multiply sky illuminance factor after LUT lookup
- **Scene**: scale the scene light intensity from atmosphere





And here is one small suggestion about the sky intensity

Some times artist may need to dim the sky for more dramatic effects

And most likely they will come up with adjusting the sun intensity directly

Please don’t do this

[click]Not only this is physically incorrect, but also this is not friendly to our optimization, which will causes flickering

[click] Let take a look at this video, the sky color is changing like stop motion.

What we suggest is try to keep the sun intensity constant

[click] For the sky we can simply multiply the sky illuminance factor after the LUT lookup

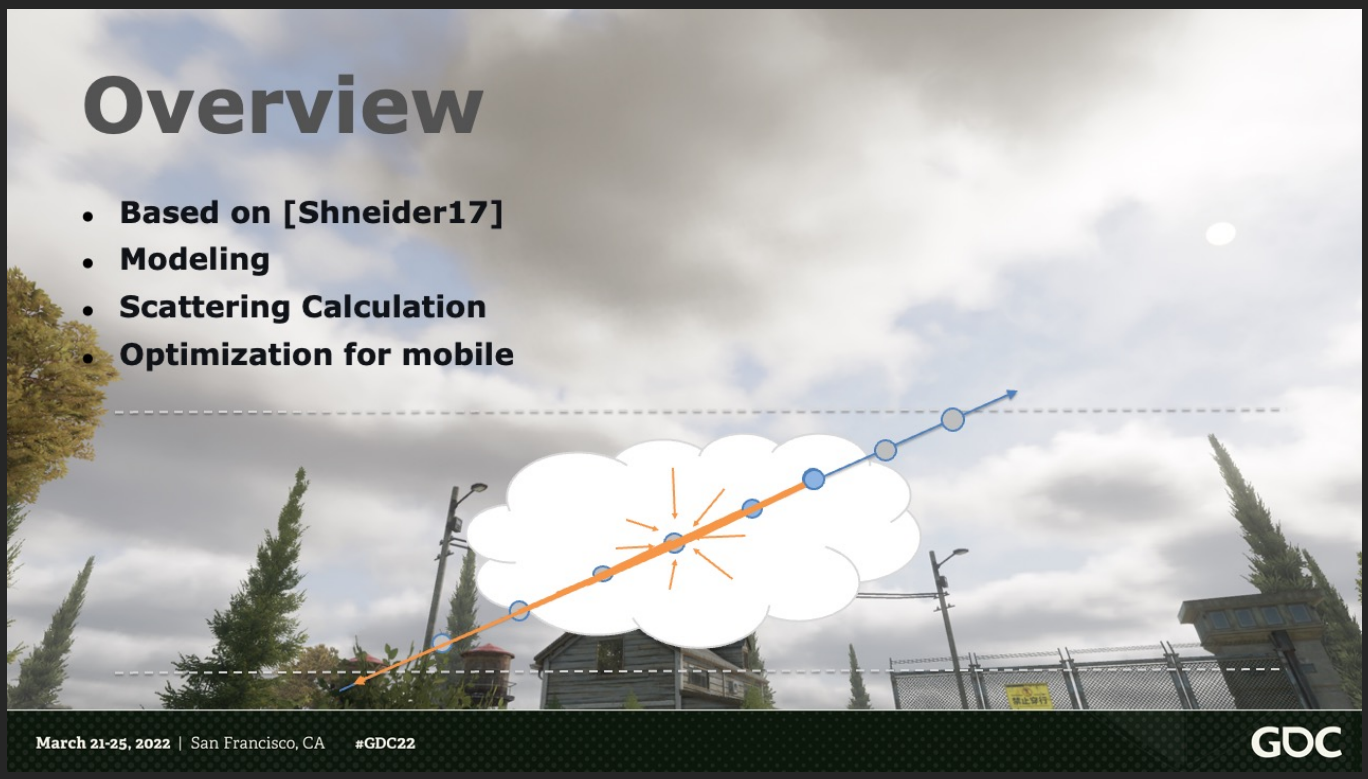
[click] For the scene, our system will first calculate a scene light intensity according to the sun angle

then we use a separate multiplier for artist to adjust it.

[click] So the flickering is gone and we got a more reasonable scattering results

That’s all about the atmosphere

Now, let RC continue with the volumetric cloud rendering, modelling and shading



So here we talk about volumetric cloud.

[click]

First, our work is heavily based on the course from Horizon team.

So if you're interested in a more detailed explanation of volumetric cloud, please check that out. It's a really great course.

For now, I'll just do a quick review.

[click]

With this method, to get cloud in a specific view direction, you would launch the view ray

[click]

into the cloud layer, And do raymarching inside it,

[click]

For example these points are the raymarching steps.

In each step, we calculate how much light current sample point would receive

[click].

We'll later talk about how to calculate this.

After knowing how much light current sample receives, we need to calculate how much of it , will further scatter towards camera

[click]

And we do the same for all samples

[click]

[click],

and just sum result from all the samples, then we get the final looking of the cloud.

So here remains two problem,

First is how do we model the cloud

[click]

. In each step, we need to know how much cloud is at current position.

So we need to define it with something.

Second is how we calculate the light scattering.

[Click]

The fact is that for such white scattering medium, rays could bounce inside it lots of times, makes it much harder to calculate than a normal surface object. So we would do many approximations. Details will be discussed later.

And third, also very important, such raymarching methods are expected to be expensive.

[click]

So we'll talk about how we get it running on mobile phones.

Btw you may notice that, cloud in the intro video, looks pretty different than this.

That's because the video was recorded pretty early, and we've actually improved the shading many times.

Modeling

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

1. 3D Noise

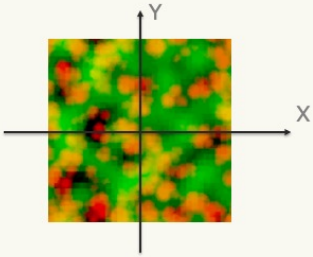
- **Worley** noise
- Tiled across sky
- Basic cloud density

- 64x64x32 R8 for Base
- 32x32x32 R8 for Detail

2. Weather Map

- Covers 40KM
- **R: Coverage** added with noise
- **G: Type** for indexing cloud profile
- Evaluated at runtime

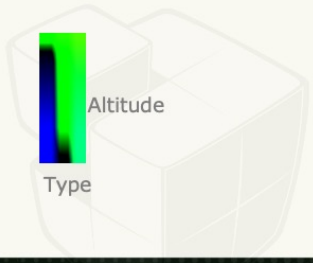
- 128x128 R8G8



3. Cloud Profile

- Shape cloud based on altitude, for each type

- 6x16 RGBA16F



So here's the modeling part.

[click]

First we have 3D noise texture,

It's made of worley noise, which has bubble like shape.

It tiled in the sky, to define basic cloud density.

We have two noise textures, base and detail. The final result is computed by subtracting detail from base noise.

The detail noise is tiled more times, so we can add more details without super high texture resolution.

But they are not enough to create cloudscape for the whole sky.

[click]

So we add a weather map.

It's a 2D texture, viewing the scene from top to bottom.

It covers 40 kilometers in our case, and tiles beyond edge.

Its pixel contains coverage value, which is directly added to the noise.

So higher value makes the cloud more dense.

And it also has a cloud type value, which is used for next part.

[click]

This is a texture called the cloud profile.

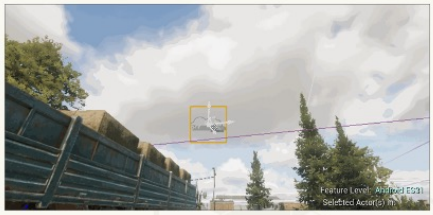
In real life different cloud types have different shape over altitude. And weather map is only in XY plane.

We use this cloud profile texture to describe how each cloud type looks like, based on altitude.

So each cloud can have different shape at different height level.

Weather Map

- Made of **Cloud Mask**s
  - UE4 actors placed in level
  - Use **material** to specify drawing content
    - Respond to **Global Coverage/Type**
  - One big base mask covering whole sky
  - Many little mask add/remove cloud



And more details about the weather map.

Although the weather map is just a 2D texture

we won't paint it manually, but rather use a little system to generate it dynamically.

[click]

In our system, the weather map is made of what we call **cloud masks.**

[click]

It's a standard unreal actor can be placed in level,

so users can use the built-in transform tool to indicate where the cloud is, as you can see in the right image.

[click]

Also each cloud mask has an assigned material, to specify the drawing content.

For example in the image on right, you see I'm dragging an cloud mask actor.

This mask, is assigned a material that, output white color, with a sphere mask.

And white means higher coverage, so the cloud becomes more dense.

we could also output black color, so the cloud of that position will get erased.

[click]

And also, to control the clouds conveniently for different weather, we have two global values, named global coverage and global cloud type.

Which is just material parameters passed to the cloud mask material.

So the material will respond to these two global parameters, and change drawing content correspondingly.

[click]

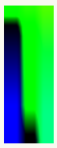
So, to use this system, we would have one big cloud mask covering the whole sky, rendering a basic weather map. It could use some Worley noise to create some basic clouds.

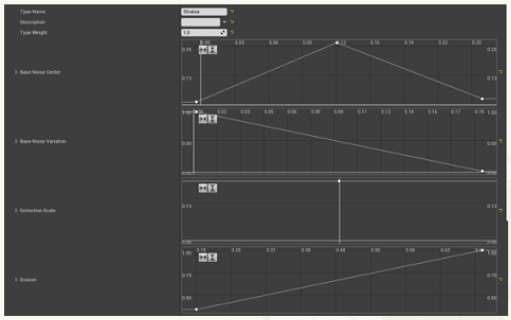
[click]

Next we would use more little cloud masks, like you see in the image, to add or remove some cloud, to make sure final result looks good.

Cloud Profile

- **LUT containing cloud shape info**
  - **R,G**: Adjust base noise range
  - **B**: Density scale
  - **A**: Flip detail noise
- **Packed into a RGBA16F LUT for using in shader.**
  - **X** for cloud type, **Y** for altitude.
  - 6x16
- **Use curve for authoring**





And the next one is cloud profile.

As I mentioned, it's used for controlling cloud shapes over altitude, for each cloud type.

[click]

The first two channels, we store base noise range, used for remapping the base noise

[click]

B channel stores density scale, so we can have different density based on altitude

[click]

And A channel stores detail noise flip.

This basically allows you flip your detail noise, so on 0.0, you get some sharp detail, and on 1.0 you get some bubble like details.

We chose these 4 parameters, because we think these affects the shape most. And of course you can have different parameters.

[click]

And these parameters got packed into a tiny LUT for shader to use,

For the texture, Its X is cloud type, and Y is normalized altitude.

In our case its size is 6 by 16.

[click]

Last, we use built-in curve tool in Unreal, so user can create the LUT easily in the editor.

This is how the tool looks like, you see we have 4 curves for each channel. X is normalized altitude, and Y is its value.

Scattering

- **Final scattering contains..**
  - Single scattering
  - Sky Ambient, ground ambient
  - Multiple scattering
  - *Atmosphere Scattering (Not discussed)*
    - *Faked in the game*
  - *Emissive (Not discussed)*

That’s all about cloud modeling.

Let’s talk about shading.

[click]

Basically we have following components for light up the cloud,

[click]

The most basic one, single scattering,

[click]

Then ambient,

[click]

And multiple scattering.

[click]

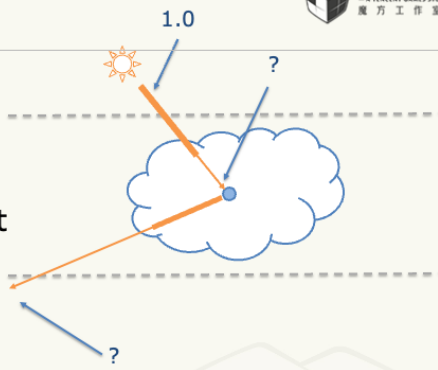
Also there’re atmosphere scattering and emissive, but it won’t be covered here.

That’s because our game doesn’t have full atmospheric scattering calculated for performance reasons.

Also emissive is straight forward, so it is not covered.

Beer-Lamebrt

- Integrate **Extinction Coefficient** along segment
  - Known as **Optical Depth**
- Transmittance = exp(-OpticalDepth)



Before we go into details, we need to know a physical law named Beer-lambert.

So for example, in this graph, if we shoot a light beam from sun into cloud, assume the intensity of beam is 1.0,

[click]

What's the intensity after it reaches the sample point?

[click]

To solve this, we need to know the **integration** of **extinction coefficient**, along segment.

Extinction coefficient is a value that linearly scaled with density of cloud.

So denser the cloud is, fewer light will reach our sample point.

[click]

This value is what we called optical depth.

[click]

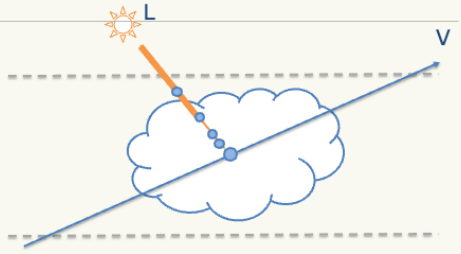
With optical depth, we can calculate transmittance with a simple exp operator, and this is Beer-

Lamber law.

We would use this in 2 place,
firstly is when we calculate cloud self-shadow, as you see here.

[click]
secondly is when we calculate how much light is reflected towards camera,
as the light need to pass through cloud from the sample point,
it will got attenuated along the path.

Single Scattering



- Energy = Sun * **Shadow** * **Phase**
- **Shadow**
  - 4 steps for optical depth
  - Use x^2 distance pattern.
  - Low LOD, no detail
- Mix two Henyey-Greenstein(**HG**) lobes as **Phase**
  - **HG**(VoL, g) = (1.0 – g * g) / pow(1.0+ g*g - 2.0 * g * VoL, 1.5) / (4.0 * PI)
  - -1.0<g<1.0, isotropic when g==0.0, sharper otherwise
  - **Phase** = lerp(**HG**(VoL, 0.8), **HG**(VoL, -0.1), 0.5)

For single scattering, when is light is directly coming from the sun,

[click].

We calculate the energy by multiplying sun light, and shadow, and phase function

[click]

For shadow, we use 4 samples, to calculate optical depth,

And we distribute samples in a square distance pattern

[click].

Like this in the right image.

So close occlusion is evaluated more accurately.

Also when sampling, detail noise is not considered. And we use higher LOD at every sample.

And for phase function, we use a classic method,

[click]

By mixing two HG functions as final phase function.

And HG is a function that

[click]

, basically you input the **dot** product of **view direction** and **sun direction**, VoL here, and a parameter **g,** which controls the overall shape of the function.

[click]

G here is smaller than 1 and bigger than negative 1. When it's 0, the function is isotropic, meaning it returns same value for all the direction.

And otherwise it becomes sharper, which means light will likely keep going forward when scattered. [click]

And for final phase, we just mix two HG functions with different **g** values.

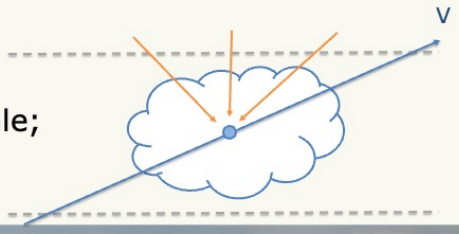
The first one would have a g value close to 1, which brings a sharp forward scattering,

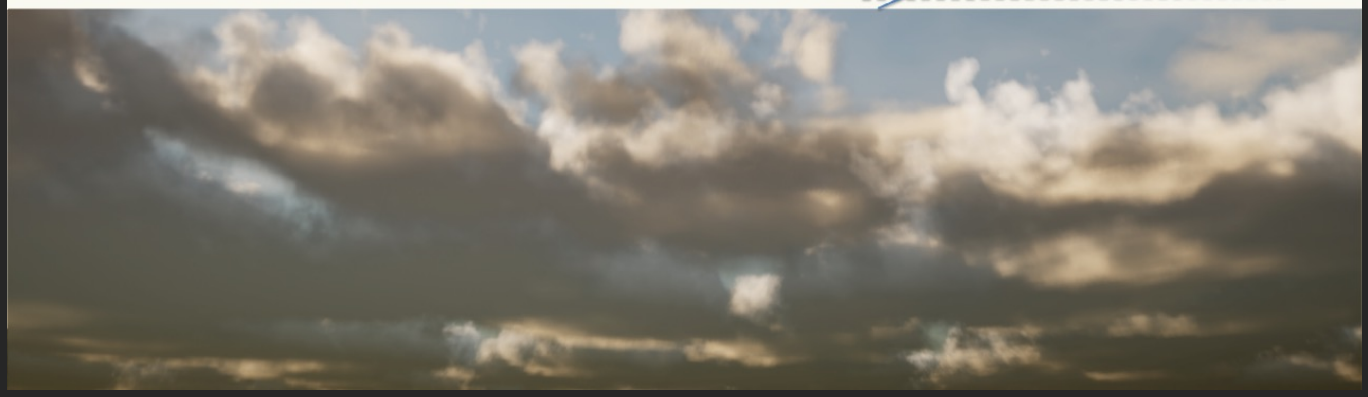
This is for simulating the silver lining effect you would see when you see backlit cloud.

And second has a negative value close to 0, makes cloud in the opposite direction not too dark.

Sky Ambient

- *Not physical*
- Energy = SkyColor * (1.0f – NormAltitude) * Scale;
  - Trick from UE4 Volumetric Cloud





For the ambient part,

First this is nothing physical.

But just apply color based on altitude.

[click]

For the sky ambient, it’s calculated like this,

Basically higher cloud gets more sky color.

Btw this is actually a trick from UE4 volume cloud. We find it work pretty well.

[click]

Here we can see the result without sky ambient,

You can see that, when it’s mostly shadowed, it’s hard to tell the shape.

Everything is just same color.

[click]

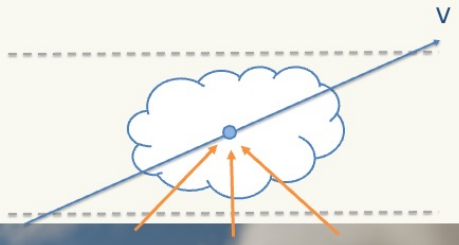
And with it on.

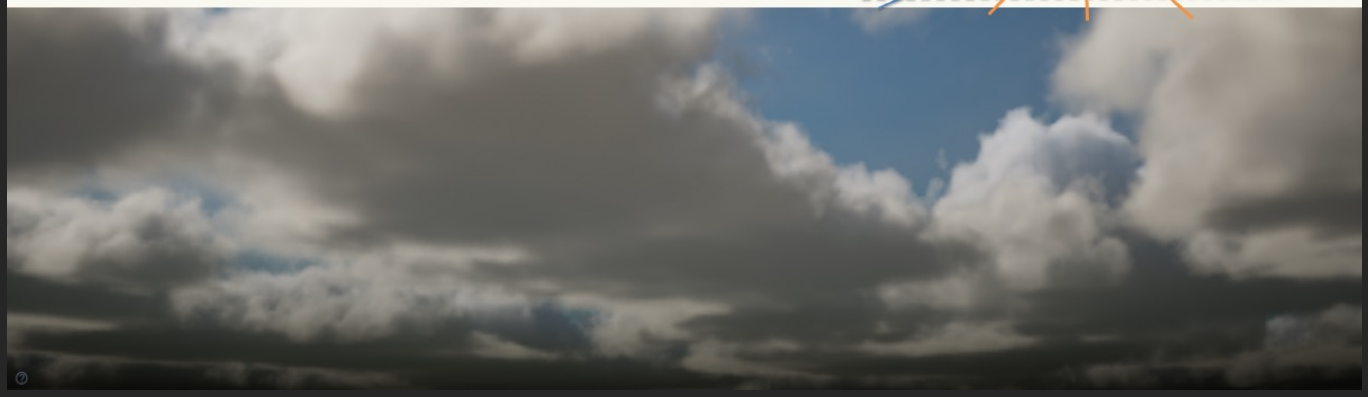
It adds nice blue tint in the shadowed part,

The overall shape is much more clear.

Ground Ambient

- *Not physical*
- Energy = GroundColor * NormAltitude * Scale;
  - Ground treated as pure color lambert surface





And we also have a ground ambient, for light coming from ground.

This is calculated the same way, while inverting the height.

So lower cloud gets more ground color.

And the color is calculated by treating ground as pure color lambert surface and calculate the reflection with main light source.

[click]

Here we can see without ground ambient,

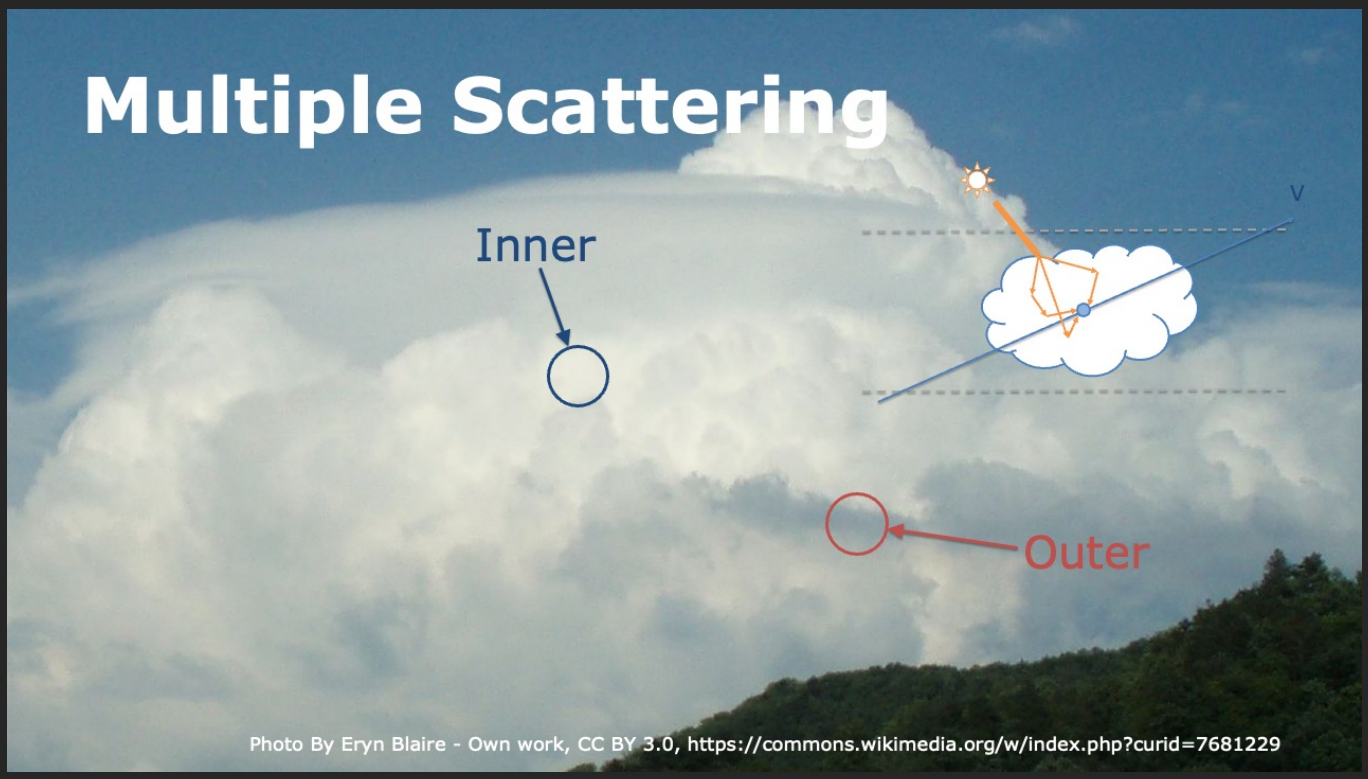
You can see the bottom of cloud is kind of too dark.

[click]

And with ground ambient on.

Now the bottom is brighter,

and it’s closer to what we would see in daily life.



Next is multiple scattering.

Where part of light is bounced more than once.

And it is essential for correct visual of cloud.

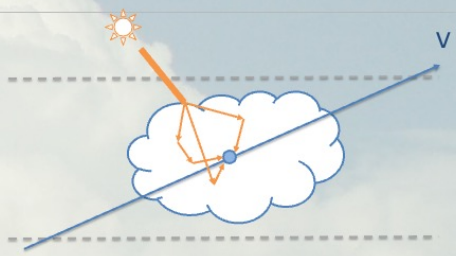
Because cloud is so white, when a light beam hits cloud particle, most of it would bounce away and keep going inside cloud, rather than absorbed.

And this becomes more intense when it's deeper in cloud.

This brings some very counter-intuitive effect to cloud.

You should see in this photo that, the inner cloud, which believed to be shadowed more, is actually brighter than the outer one.

Multiple Scattering



- Expectations
  - **Somewhat physical**
  - **Brighten shadowed cloud**
  - **Dark edges on surface cloud**
  - **Not too many parameters to adjust**

Overall brightness ↓ Dark edge ↓

$$Energy = MSApproximation(x) * InscatteringProbability(x) * Scale$$

So for simulating this effect, we've got some expectations.

[click]

First it should be somewhat physical.

We can have some parameters for artist, but we need something solid for artist to start with.

[click]

Second, it should be able to brighten cloud that is not too deep.

[click]

Third we should see dark edge on cloud surface.

[click]

And last, we don't want to add too many parameters.

[click]

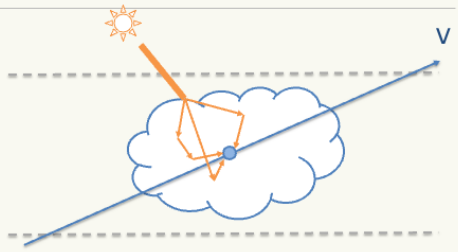
In our final solution we have three parts for the multiple scattering,

Multiple Scattering Approximation, which is used for calculating a more correct overall brightness

Inscattering Probability, for creating dark edge on cloud

And artist scale for more control.

Multiple Scattering



- Based on [Wrenninge13] [Hillaire16]
- EnergyMS[**i**] = **Sun** * **Shadow** * **Phase**, but
  - **Sun** energy is attenuated by pow(**a**, i)
  - **Shadow** is calculated with density attenuated by pow(**b**, i)
  - **Phase** is more isotropic, g=g*pow(**c**, i)
- **a,b,c** are hyper parameters ranged (0, 1)
  - Considered energy-conservative when a<b

Energy=MSApproximation(x)*InscatteringProbability(x)*Scale

So for the first part,

[click]

we use a method proposed by Wrenninge in offline rendering, which is then adapted by Hillaire to realtime volumetric cloud rendering.

And here's a quick review of it.

[click]

As we earlier discussed single scattering, here is how we calculate it.

And now, to calculate multiple scattering of octave **i**, we do some changes

[click]

First, the sun energy is attenuated by pow(a, i),

[click]

Second, shadow is calculated with cloud density attenuated by pow(b, i)

[click]

Third, phase function is more isotropic, by making the g value towards 0.

[click]

Here abc are hyper parameters ranged 0 to 1.

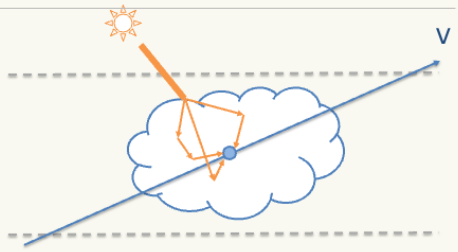
[click]

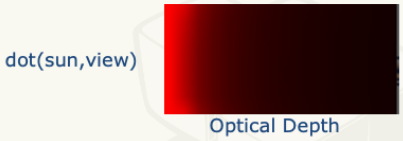
And is considered energy-conservative when a<b.

With this, we can calculate multiple scattering during each raymarching step, with very little overhead.

Multiple Scattering

- Fix **a,b,c** for pre-compute a LUT
  - **Precompute result in unit sun illuminance**
  - **X for optical depth, Y for VoL**
  - **16x8 R16F**





Energy=MSApproximation(x)*InscatteringProbability(x)*Scale

In Frostbite and Unreal implementation, at most 3 octaves are used.

We did a little change that, we fixed the hyper parameters in the method, so we could precompute a LUT ahead, with any octave number we want.

The LUT is calculated under unit illuminance,

And is indexed by optical depth and sun dot view.

So at runtime, after calculating single scattering, we get optical depth, we can then index into the LUT to get multiple scattering.

And here's a quick comparison

This is without multiple scattering.
You should see with only single scattering,
the light can’t reach deep inside cloud,
and the cloud looks more like smoke.

And this is with multiple scattering enabled.

It’s apparent, the overall brightness is now more accurate, more like the cloud in real life.

But we still don’t have the dark edge effect.

Multiple Scattering

- Use LOD sample to estimate surrounding density[Shneider17]
  - Higher density, more scattering towards sample point
- **InscatteringProbability**(x) = Albedo*(1.0-exp(-ExtinctionCoefficient_Lod));
  - Normalized output
  - Consistent for different raymarching parameters

Energy=MSApproximation(x)*InscatteringProbability(x)*Scale

So for adding dark edge effect,

we use a method similar to the one in Horizon.

[click] This method is basically, to use sample with LOD,

to estimate how much cloud is surrounding current sample point.

And more cloud is around current sample, we should get more multiple scattering.

[click] So we have this magic code. Basically it takes the lod sample, outputs a 0~1 value about scattering intensity.

This is probably not physical correct,

[click] but it gives a normalized output,

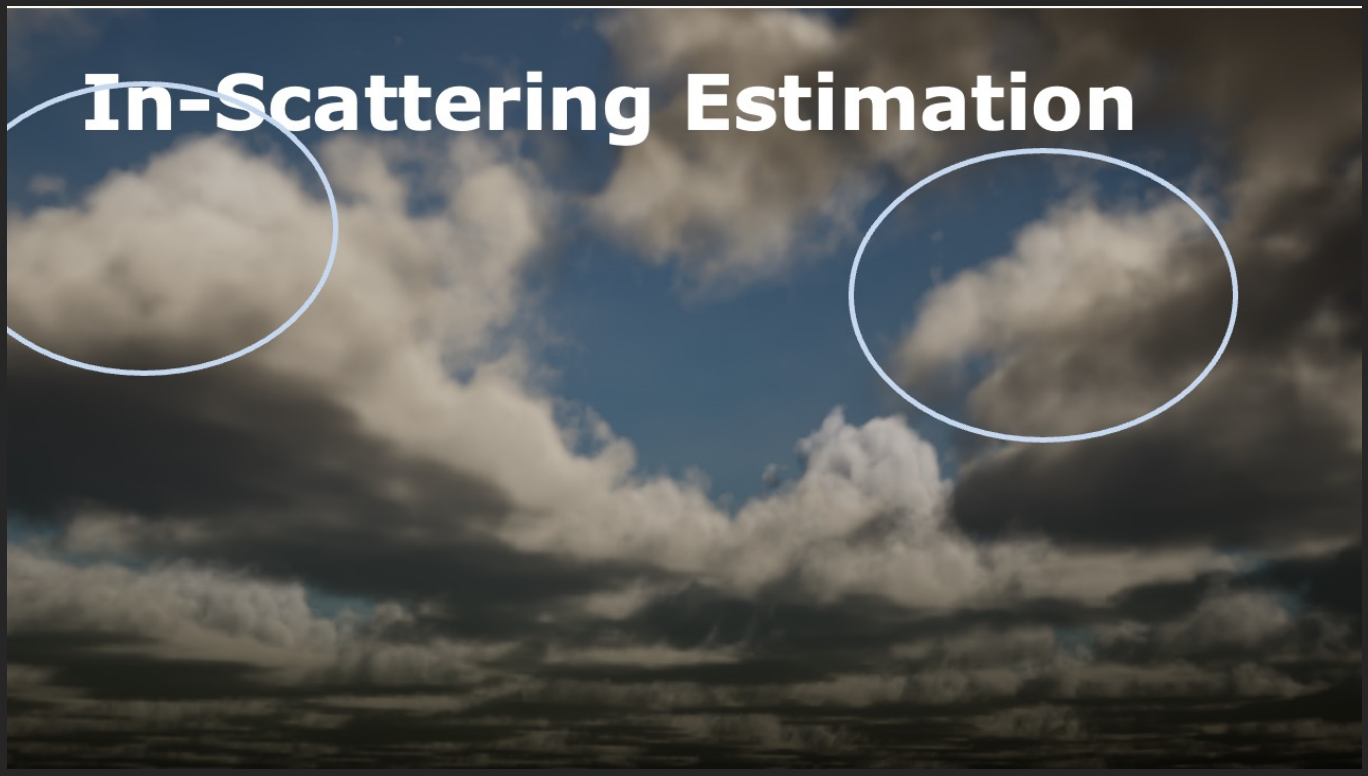
[click] And it gives same result under different raymarching parameters. So we're good with it.

Here’s a quick comparison,

This is without the in-scattering estimation.

Notice the part in circle.

It’s hard to tell the shape in these parts.



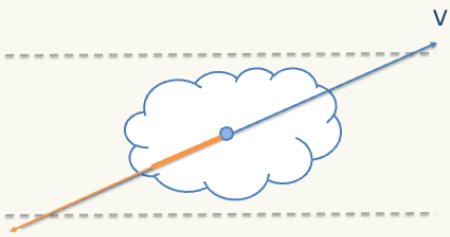
And we add the in-scattering estimation.

Now you should notice all the edge of cloud is darker, which is expected

And the circle part also has more details now.

Integrate Scattering

```
half3 FinalScattering = 0.0f;
half TransmittanceCam = 1.0f;
for (int iSample = 0; iSample < N; iSample++)
{
    half ExtinctionCoefficient  = [Modeling];
    half3 FinalEnergy = [Scattering];
    FinalScattering += TransmittanceCam * FinalEnergy *
ExtinctionCoefficient * Albedo;
    TransmittanceCam *= exp(-ExtinctionCoefficient * ds);
}
```



- **Inconsistent when swapping scattering/transmittance integration**
- **[Hillaire15] for more robust integration**

So that's everything we need, to calculate scattering at each sample point.

The next part is about how to integrate these samples together. Let's look at the code.

[click]

In the raymarching shader, we would start with initial value,

final scattering which means final color we see, is zero,

And transmittance to camera, meaning the transmittance between camera and sample point.

Since we haven't started raymarching, it's 1.0.

As we raymarching through cloud, scattering value will be greater, and transmittance will be smaller.

Finally is a for loop for stepping into cloud.

[click]

In the loop body, we would first calculate extinction coefficient, which corresponds to our modeling part.

And final energy, which is sum of all our scattering parts.

[click]

And we would add final scattering like this.

The scattering current camera actually receives, is by multiply these parts together.

[click]

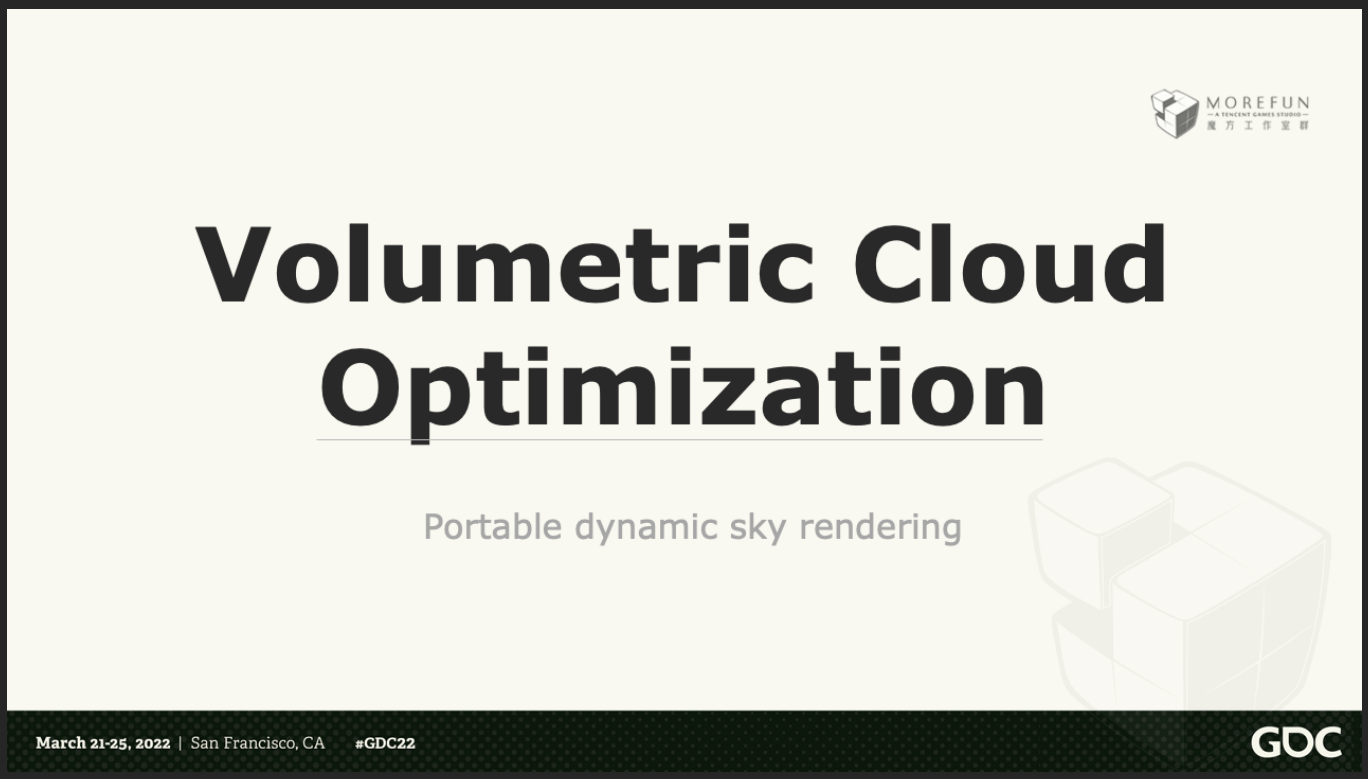
And then we update transmittance to camera, by taking transmittance of current segment in.

And that’s it.

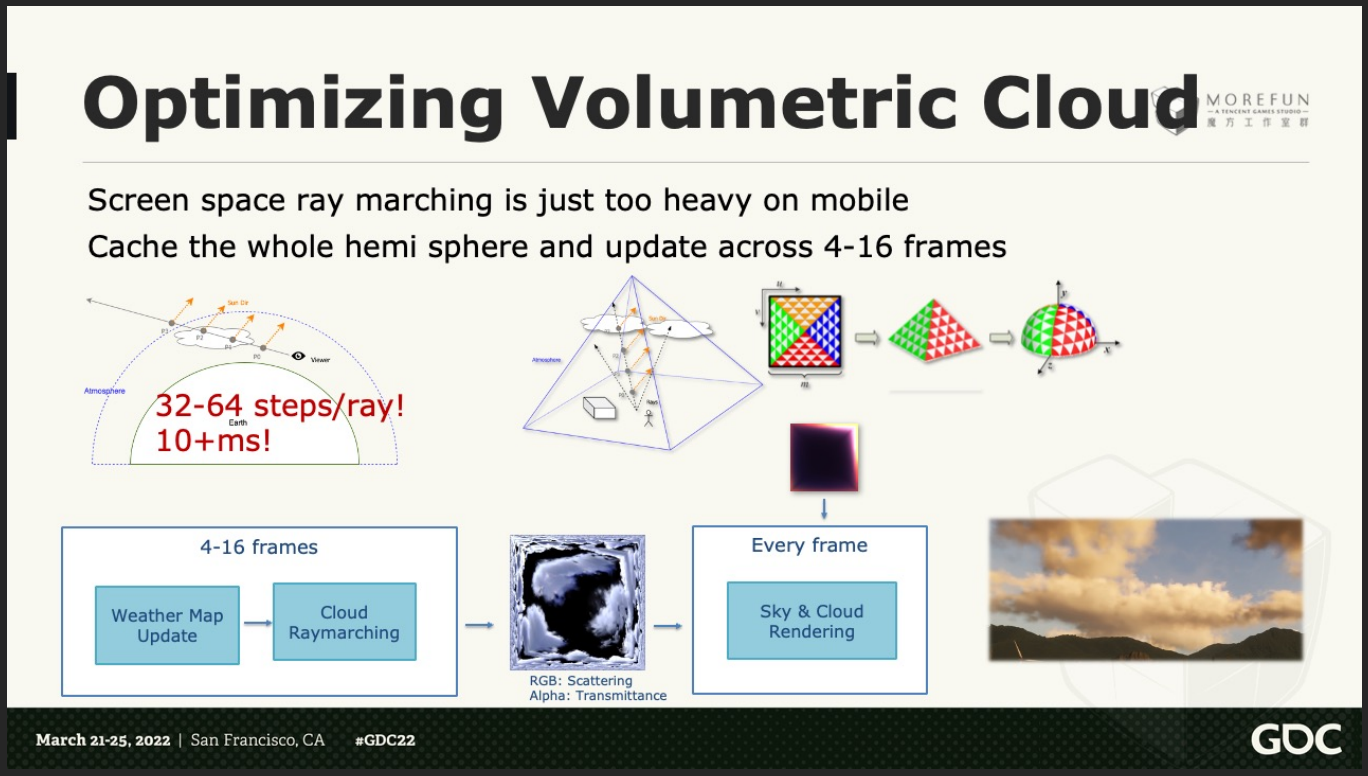
[click]

But this integration has a problem that, the result could be inconsistent if we swap scattering and transmittance calculation.

We at last used a method proposed by hillaire in his great course, which give consistent result. Be sure to check it out.



Km will continue with the optimization techniques in our solution.



Thank You RC, as he explained, rendering realistic cloudscape involves large amount of calculations in each step of ray marching.

[Click]

For example we need at 32 - 64 steps for each ray, and every step involves so many texture reads and ALU instructions

It is too just heavy to do it at 1080p in screen space for mobile

[Click]

Again, our friend hemi-octahedron comes to rescue

We project the sky with hemi-octahedron mapping, and cache all the ray marching results in a 512 x 512 2d texture.

[Click]

Since the weather condition or the sun direction is changing gradually in our game, so we can split the cache update across multiple frames

[Click]

In every frame, we use the cache results to combine cloud scattering with the sky.

Why Hemi Octahedron Caching

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

- **View Location & FOV independent**
  - Same cache for dynamic reflection captures
- **Reprojection and multi-frame update friendly**
  - Checkerboarding
  - Slicing
  - Temporal Up-sampling
  - HDR compression

So, what is the advantages?

[Click] First, it is View Location and FOV independent, so we can reuse the cache for any dynamic reflection captures

[Click]

For example, our game use planar reflection for water surfaces, and we don’t need to do an extra ray marching for the reflected view.

[Click]

Another advantage is that, cloud planar movement is relatively slow in octahedron space, reprojection works pretty well with it

We can effectively restrict the amount calculations to avoid overheating the mobile GPU.

And we applied the following strategies.

[Click]

Checkerboard Update

Slicing

Temporal Up-sampling

HDR compression

And I will explain them one by one

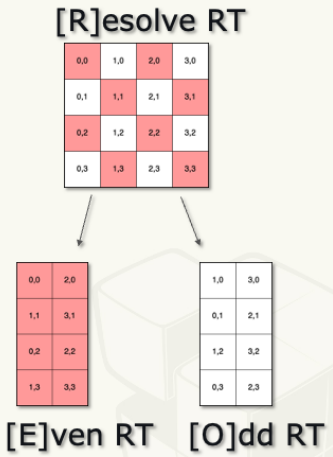
Checkerboard Update

MOREFUN
— A TENCENT GAMES STUDIO —
魔方工作室群

- **Pack the even / odd pixels into 2 textures**
- **Ray march either on [E] / [O] for each cycle (reduce the cost by 50%)**
  - No need to use stencil masking
  - Texture Cache friendly when resolve

```
float2 CheckerboardToResolved(float2 SvPosition, float EvenOdd) {
  float2 Coord = floor(SvPosition.xy);
  float Shift = fmod(Coord.y + EvenOdd, 2);
  Coord.x = 2 * Coord.x + Shift;
  return Coord + float2(0.5, 0.5);
}

// convert packed SvPosition to Ray March Direction
const float3 RayMarchDir = OctDecode(CheckerboardToResolved(SvPosition, CheckerboardEvenOddId) * CloudRTInvSize);
```



Inspired by the previous checkerboard rendering techniques, they help use to save 50% of rays needed to be calculated per frame

And we using a special kind of pixels arrangement.

We have 3 render targets:

[Click] Firstly we have one full sized rt called R, which contains the resolved results.

[Click]

And we split the pixels in R into two half-sized rt called E and O, which store the even and odd pixels for each row in R

We perform ray marching in E or O in round robin manner

[Click] By doing so, we can update only half of the pixels without any stencil masking, so we can precisely control how much will the GPU write back to main memory.

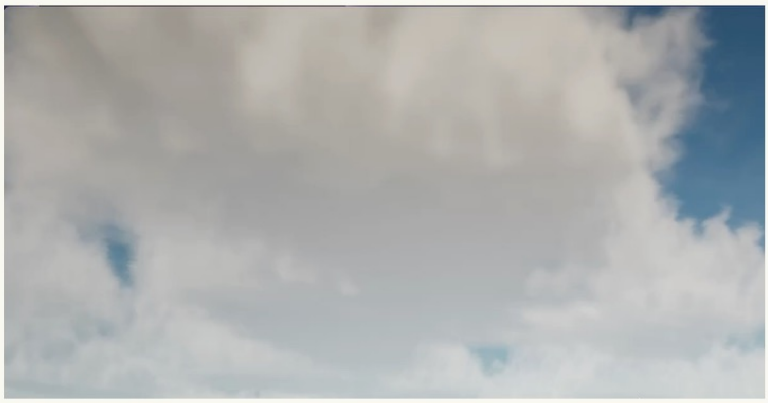
[Click] And we use this piece of code to convert the SvPosition into ray marching direction.

Checkerboard Resolve

Use copy in resolve pass

Discard the fragments does not match current 'even / odd ID'

- Save 50% of texture fetches



```
float2 ResolvedToCheckerboard(float2 SvPosition, float EvenOdd) {
  float2 Coord = floor(SvPosition.xy);
  Coord.x = floor(SvPosition.x * 0.5);
  return Coord + float2(0.5, 0.5);
}

void MainPS(
    in float2 UV : TEXCOORD0,
    in float4 SvPosition : SV_POSITION,
    out float4 OutColor : SV_Target0
    ) {
  float2 Checkerboard = ResolvedToCheckerboard(SvPosition, EvenOddId);
  float2 Reconstructed = CheckerboardToResolved(Checkerboard, EvenOddId);

  // discard pixels which does match current event / odd id
  // save 50% of texture fetches
  if (abs(Reconstructed.x - SvPosition.x) > 0)
    discard;
  OutColor = CurrTexture.Load(float3(Checkerboard, 0));
}
```


After we finish updating either E or O, we resolve their results into R immediately, so they are available for display

[click] We simply copy the results and it is actually good enough.

[click] Since the resolve shader apply on every pixels on R

We discard any pixels that do not belong to the current input, so we can save 50% of unnecessary texture fetches.

[Click] Here is the code, the idea is converting the SvPosition to local position in E or O,

and then convert it back to resolved position.

If there is any different between the original and the reconstructed position, it mean this location doesn't belong to the input.

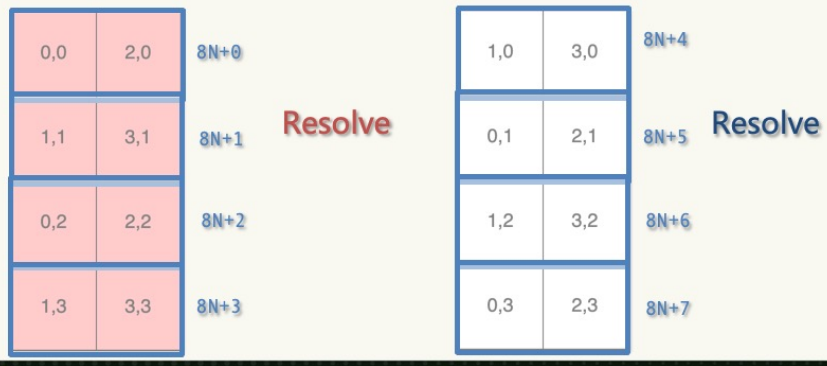
[Click] And this close up video shows the checkerboard update and resolve in action

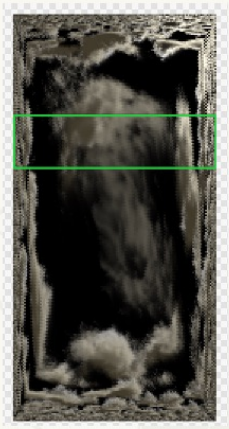
[wait 5 seconds]

It is hard to notice any artifacts.

Slicing

- **Use viewport and scissor rectangle to restrict GPU write**
- **Divide [E] / [O] updates into 4 - 16 frames**
  - (87% - 96% cost reduction)





We can further restrict the amount of calculations by use slicing.

[click] The idea is simple, we use scissor rectangle to constraint which portion of the render target will be updated.

As you can see the green rectangle in the screen shot

[click] And each render target will be split into 4 - 16 slices and we only update one slice per frame.

I will demonstrate how it works:

[Click]

For example, each target has 4 slices and we have 4 rows for each target,

So we will take 8 frames to update both E and O

[Click] Let’s start with E and we only ray march one row per frame.

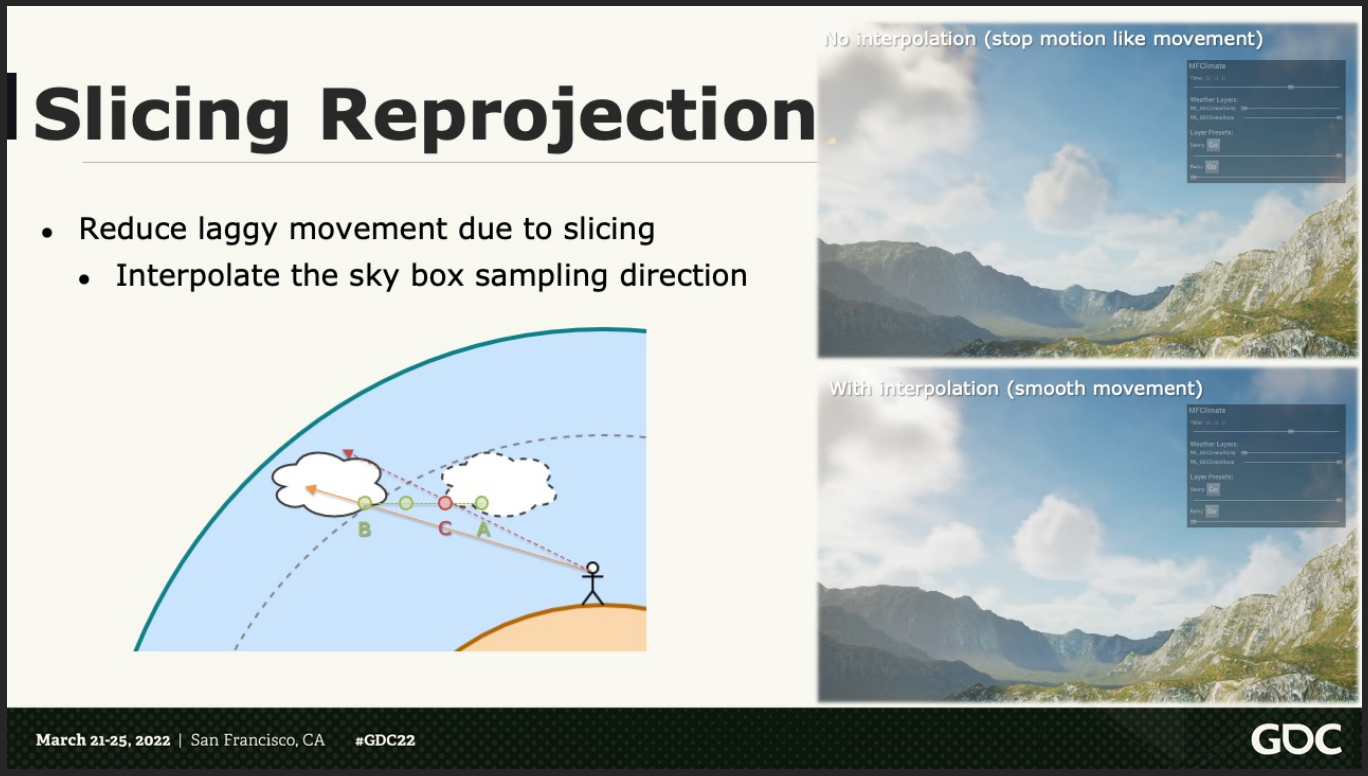
[click] [click] [click] At this movement, the update of E is completed, so we resolve it to R, so its content will be displayed on screen.

[Click]

Then, we continue to ray march on O. [click] [click] [click] [click]

Once O is fully updated, we resolve it to R [click]

Then we start another cycle with E



However the performance gain with slicing doesn’t come for free!

[Click] As slicing will cause stop motion like cloud movement as shown in this video

To solved this problem, we can interpolate the cache sampling direction for the sky box.

[Click] For example the cloud is moving from A to B with 4 frames

[Click] Assume we are look at point B in the current frame and we are now 1 frame ahead the last update cycle.

[Click] Since the amount of cloud movement is known, so we can trace backward and find point C

[Click] The direction from viewer to C will be the reprojected direction.

[Click] Let see the effect in the bottom video

Temporal up-sampling

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

Can we do better with 32-64 steps / ray?

Temporal method comes to rescue

- **similar to TAA**
- **Per-frame global jitter using 1D Halton Sequence (b=3)**
- **16 steps / frame is good enough (50% reduction)**

```
// GlobalJitter is an uniform variable
float4 CurrState =RayMarch(Start, End, GlobalJitter);
float4 PrevState = ResolvedRT.Sample(PointSampler, OctEncode(End - Start));

// we use 0.05 for TemporalUpsampleAlpha
Result = lerp(PrevState, CurrState, TemporalUpsampleAlpha);
```

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

GDC

As we stated earlier, 32 to 64 steps is needed for each ray in order to render beautiful cloudscape,

So we would like to optimized a little bit to save more GPU power for other effects

[click]And we employ a temporal technique which is similar to TAA.

[click]In each frame, we apply a global offset to every ray and this offset is updated after each ray marching cycle.

Then we blend the incoming result with the history stored in cache.

Let take a look at this example:

In the first cycle, we evaluate the blue dots

[Click]

Then in the second cycle, we evaluate the orange dots

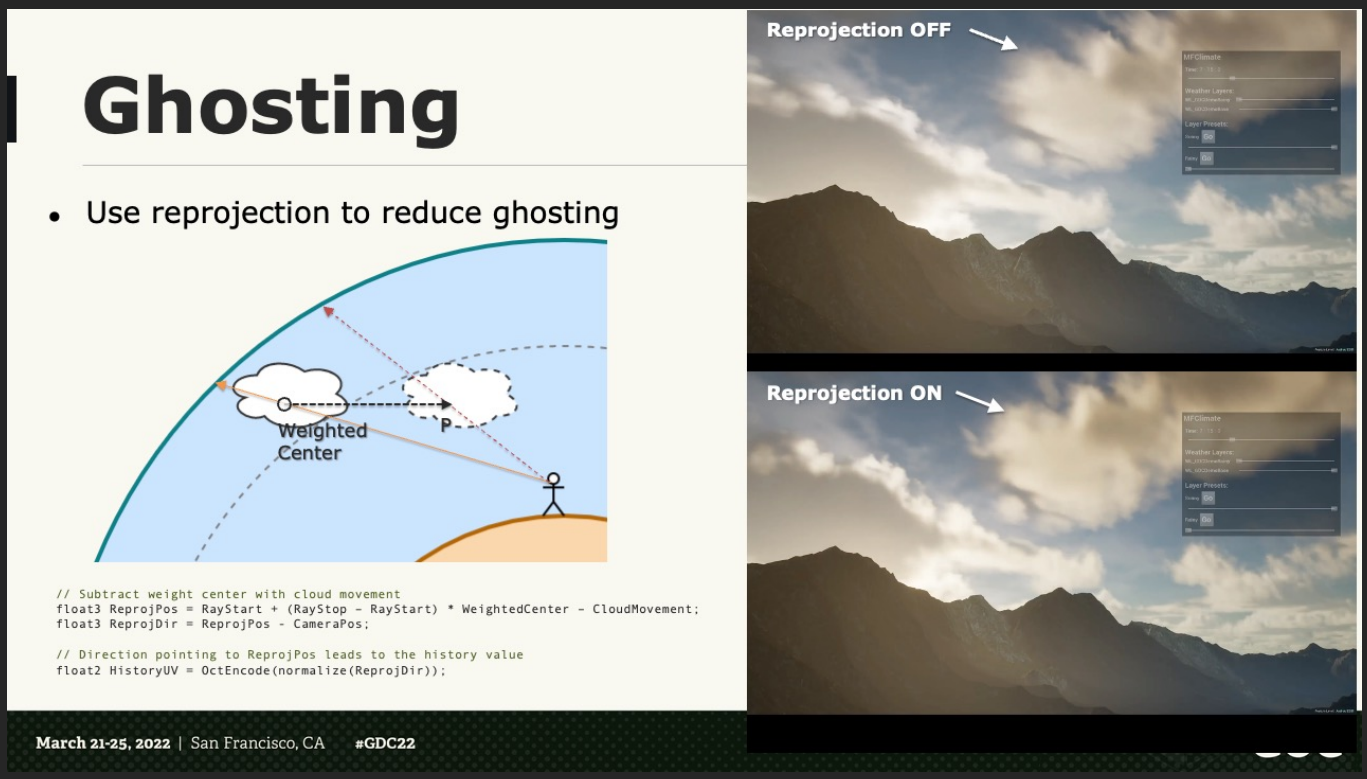
[Click]

In the third cycle, we evaluate the red dots, and so on

[Click]

By doing so, we are virtually evaluating many samples for each ray over time, and the results usually converge within several frames.

[click] In our game, 16 steps per frame can achieve a good result,



However, ghosting will appear when cloud is moving too fast (say 100m / seconds);

[Click] Let's take a look at the problem

Again we can apply reprojection on the direction used for looking up the history value.

[Click] During the ray marching process, we calculate a weighted-center for each ray according to the cloud transmittance.

[Click] Then we subtract the weighted center with the cloud movement and we get point P

[Click] Subtracting P with the viewer location will roughly equals to the pervious ray marching direction.

[Click] And here is the result for applying the reprojection, we can see ghosting is greatly reduced.

HDR Compression

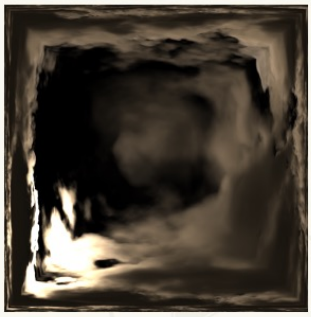
Minimize memory storage and bandwidth

RGBA16f

- RGB for scattering, A for transmittance
- 2MB (512 x 512)

Challenges for RGBA8 (portability) support?

- HDR Values
- Phase function peak at sun direction
- Numerical stability in temporal up-sampling



Phase function peak

Besides discussing on how to reduce the calculation amounts, we like to share on how to minimize the required memory storage and bandwidth.

Since both of them are important in mobile.

[Click] Normally we use half float to store the raymarching results, and it takes 2MB for 512 x 512

[Click]This is acceptable for mainstream devices

[Click] However, half float is not so friendly on older devices, so we have to consider using the 8-bit RGBA format

And here are the challenges:

[Click]First we use physically based lighting, so all input and output are in high dynamic range

[Click]Secondly, there is a very strong phase function peak at the sun direction which make the situation worst

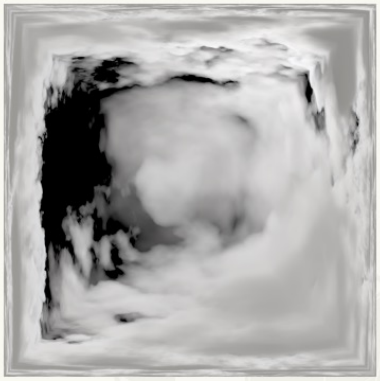
[Click]In meanwhile, we also needed to consider the numerical stability for temporal up-sampling

So how can we achieve this goal?

HDR Compression

"Normalization" trick

- $OutColor = float4(pow(\frac{S/Phase}{PreExposure}, \frac{1}{2.2}), T)$
- **Phase Term** = $lerp(P, P_{iso}, 0.75)$
- **PreExposure**
  - Calculated with sun illuminance, ambient color, and scaled by magic number etc..
- **Gamma**
  - pow(scattering, 1.0f/2.2f)



And we comes up with a "normalization" trick that applies to the scattering outputs.

Let's take a look at this expression.

[Click]

Firstly, we divide the scattering by a phase term, which could lower the peak value.
[Click]

Please notice that we are not directly dividing by the raw phase function, but blending with the isotropic version,
This avoid over compressing the shadowed pixels around the peak

[Click]

Secondly, we divide the scattering by a "PreExposure" value.

This one is pretty simple, just adding up light illuminance, ambient color, and scale by some magic number to ensure multiple scattering is in range.

[Click]

Finally, we do a gamma 2.2 encode.

And it Is how the output looks after normalization

Scalability & Performance

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

- Performance measured with:
  - iPhone 11 (Xcode 13.2.1)
  - GTX 1070 (Nsight Graphics 2019.6.1)

Resolve GPU Cost
- iPhone 11 ~100us
- GTX 1070 ~0.01ms

	Non Optimized	High	Medium	Low	Legacy
Temporal Up-sampling	OFF	ON	ON	ON	ON
Checkerboard	OFF	ON	ON	ON	ON
Step Count	64	16	16	16	16
Slicing	OFF	4	8	8	8
HDR Compression	OFF	OFF	OFF	OFF	ON
RT Resolution	512	512	512	256	256
iPhone11 GPU Cost	10.47ms	640us	335us	315us	220us
GTX1070 GPU Cost	1.26ms	0.09ms	0.08ms	0.06ms	0.06ms

And here are some performance measured on iPhone 11 with different scalability profile

In addition we also measured the performance on a GTX 1070 graphics card with the same test scene.

For the non-optimized version, it takes 10ms for the ray marching on mobile

For highest quality on mobile, we just divide the cache into 4 slices and it takes 0.6ms on mobile and only 0.09 on desktop.

We double the slice count for the middle quality, and the cost scale down on mobile

For the lowest quality that will be used on legacy devices, we turn on HDR compression and reduce the cache size to 256 x 256,

It takes 220us and we can see the effect of HDR compression when compare to the non-compressed one.

For the resolve pass, it take 100us on mobile and only 0.01ms on desktop, which is ignorable.

That’s concluded the optimization part.

Now, let’s move to some dynamic weather effects in project: dark

Cloud Shadows

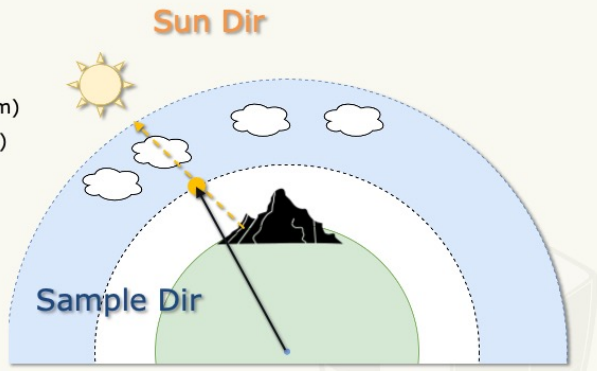
Reusing the ray marching cache

- Ray-intersect (from shading position) with sky bottom
- Normalize intersection point as cloud sampling direction

Notes

- Compute cloud shadow in base pass
  (Reconstruction from depth method suffers from precision problem)
- Move Ray-intersection to vertex shader (for better performance)
- Need interpolating the sample-direction (mentioned in Slicing)

```
float3 CloudLayerPos = RayIntersect(WorldPos, SunDir, SkyBottomSphere);
Float3 SampleDir = normalize(CloudLayerPos + ReprojectionOffset);
return CloudTex.Sample(OctEncode(SampleDir)).a;
```



The first one is cloud shadows

Since we already computed the cloud transmittance of the whole sky, so we can project this on the ground for cloud shadows effect

The idea is simple, first we trace a ray from the shading location towards the sun

[Click]

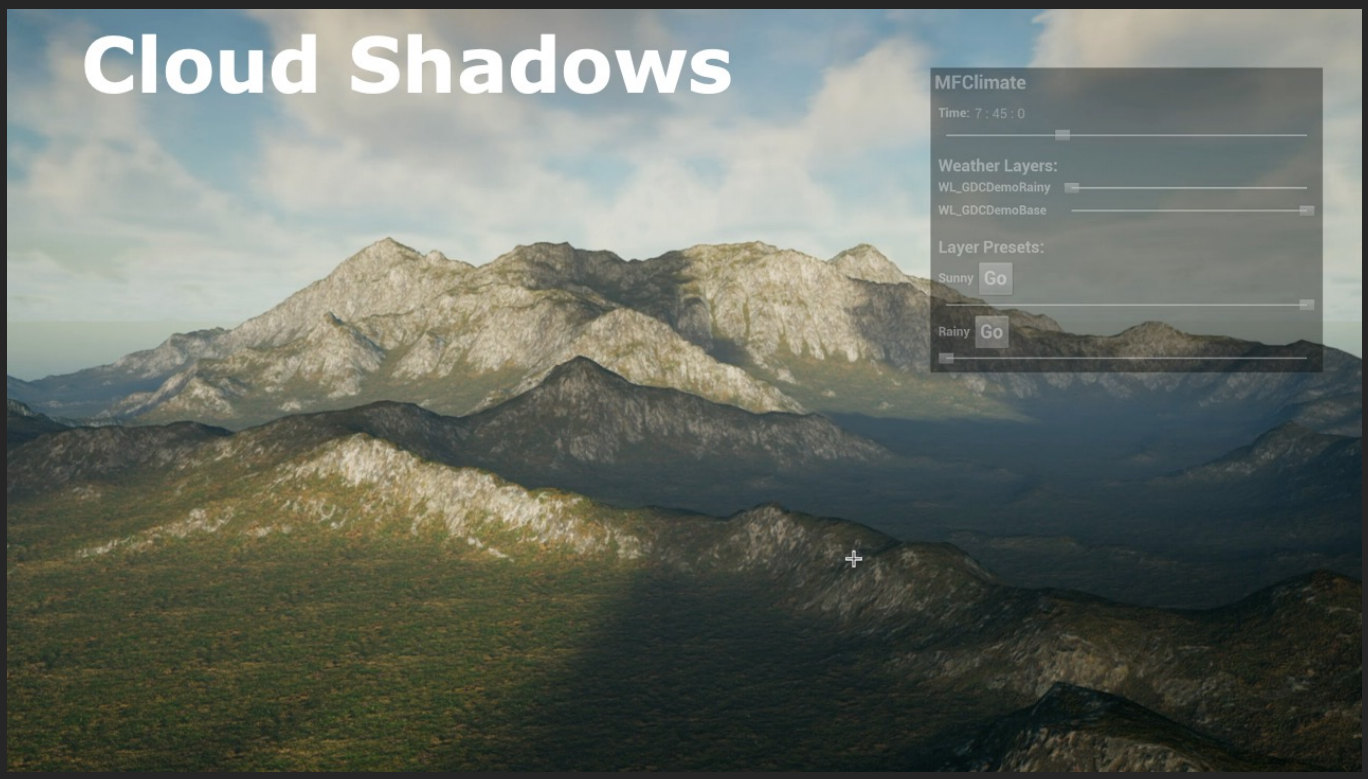
Then we find the intersection for this ray with the bottom cloud layer

[Click]

After that we subtract this point with the earth's center and get our sampling direction

[Click]

Let's see how it works in action



The cloud shadows can interact with the sun direction and weather changes.

Please notice that this should be applied in object's base pass,

as reconstructing the world position from scene depth will suffers from precision and stability problems

For better performance, we can also compute the

ray-intersection in vertex shader.

Next part is rain.

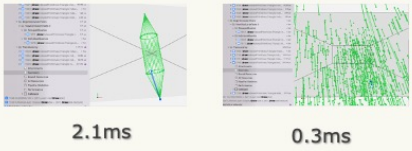
Here’s a short clip we recorded in editor.

It contains rain particles, and material wetness effect

[wait finish and click]

Rain

- **Particle rain**
  - Per-vertex occlusion test
- **Wet surface by adjusting material parameters [Waylon16]**
- **Cheap occlusion map by using CSM Scrolling[Acton12]**
- **Major problems for now**
  - Occlusion sampling is heavy (4 taps for PCF)
    - Considering ESM
  - Some materials don't have specular
  - …



[click]

So the rain itself is rendered using particles.

We've measured that on GPU, it's far more performant than using a spindle.

You can see the test result on the right.

[click]

And the surface wetness is created by adjusting the material parameters, using the method from Waylon.

But we currently removed puddle and ripple effect, because it is costly for mobile.

[click]

Also we support fast occlusion map update, by making use of CSM scrolling.

So the occlusion map is rendered at runtime,

And if it's too far away from camera center, it will scroll itself and re-use any part if possible.

[click]

However there still exists some major problems.

Firstly, occlusion sampling is pretty heavy.

If we want smooth edge, we will need 4 taps PCF. We're considering using ESM which can be pre-filtered.

And also, some material doesn't have specular in the game. For example the tree you just saw in the video. Also if graphics settings is low, all the material are set to be fully rough, and don't have specular.

And there're many other problems also, for example, if it's raining, then the sun light would be little, and shadow is gone. So the whole scene would look too simple, without AO, the players just think it too ugly. So we also have to do something about that.

And what about reflection? Since everything become smooth, without correct reflection it just feels wrong. And we don't have SSR for now.

Anyway, still much work to do.

Lightning

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

- **Lightning leader**
  - Barely visible
  - Create "lightning channel"
- **Return stroke**
  - First wave electrons flow through channel
- **Re-stroke**
  - Flickering light

Slomo video from [TSMG19]

Next is lightning.

We use real life references to make the lightning effect as real as possible.

And I'd like to share some key points to make realistic lightning in real time rendering.

The animation on right is taken from a YouTube channel called **the slomo guys**. It clearly shows the lightning process.

So, lightning in real life, has three stages.

[click]

First is lightning leader,

which is the spider-web like path, growing from cloud to ground.

In real life this is barely visible for human eyes because it's so fast.

But it's cool, so we decided to implement it.

And this stage creates what we call "lightning channel", when it reaches the ground.

Once the channel is created, huge current flows through the channel

[click]

and this is what we call a return stroke.

The stroke would heat up air to a very high temperature causing flash and thunder.

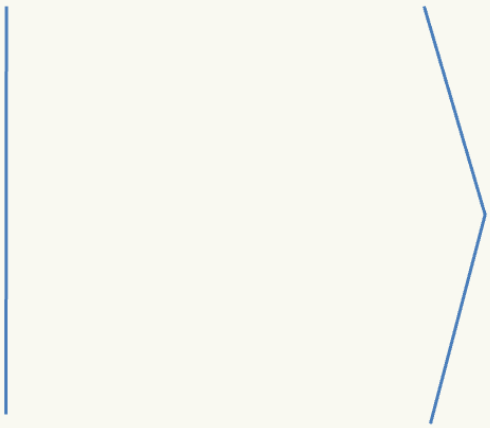
[click]

After return stroke, there could be some re-strokes.

It’s some strokes that happen after the return strike, following same channel.

This is usually smaller than return stroke, and happens average 3 to 4 times, creates the flickering light you would see.

Lightning



To create lightning, we need to model it.

Here we use a fractal algorithm, to create lightning channel.

The algorithm is straight forward, and we start with a straight line from start to end

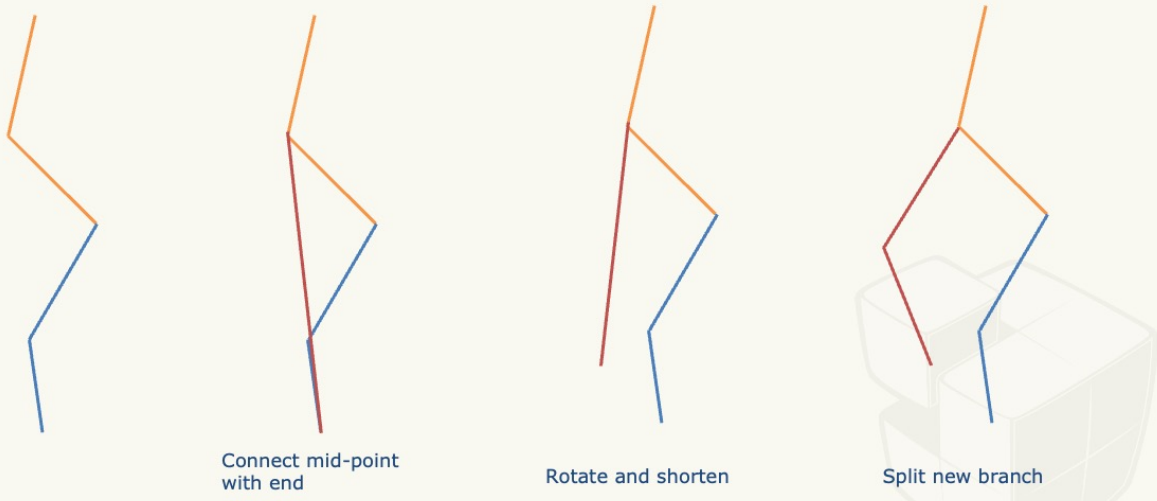
[click]

Then you pick its middle point as a new node, offset it in the plane perpendicular to the original line

[click]

Then you just do this process recursively on child segments, until the segment is short enough. But that's only one branch. We need a spider web like structure to simulate the lightning leader process.

Lightning



To create branches, we do a random branching test, while splitting.

For example, here is the splitting result from the last page.

Assume we just split the orange segments.

[click]

If the random test passed, we create a new node at the end,

And its parent is current splitting node.

So the new segment is the red one.

[click]

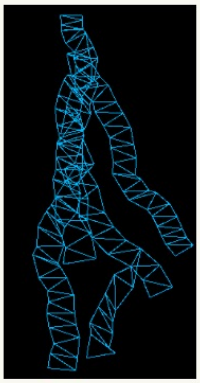
Then we just shorten it by some percent, and rotate it away from the original direction by some degrees.

[click]

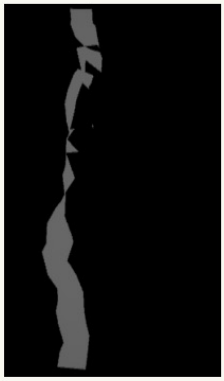
And we do recursive splitting on the created branch as well.

Lightning

- Final mesh
  - <0.2ms to create mesh on Snapdragon 865

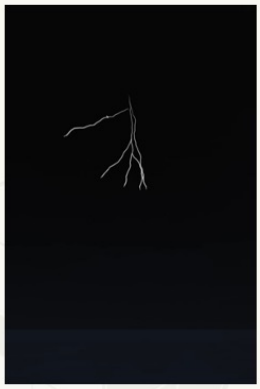


Wireframe



R: Is Main Branch

G: Normalized Distance



Then we convert the result to quad list mesh.

And here is the result.

[click]

Also we store some info in vertex color.

Its R channel, tagging if a vertex is main branch. So we could toggle to show the main branch only

And G channel stores a normalized distance from lightning start. We use this for the cloud-to-ground growing animation

[click]

And here's how it looks like when animated. I've slow down the speed of lightning leader so we can see it clearly.

So here's how it looks in the game

And I'll also share about how the scene and cloud is light up.

[click]

for lighting up the scene,

We simply boost up main light

And calculate the intensity based on squared falloff, using distance between camera and lightning.

So shadow is not correct. But it happens really fast, so it is hard to notice.

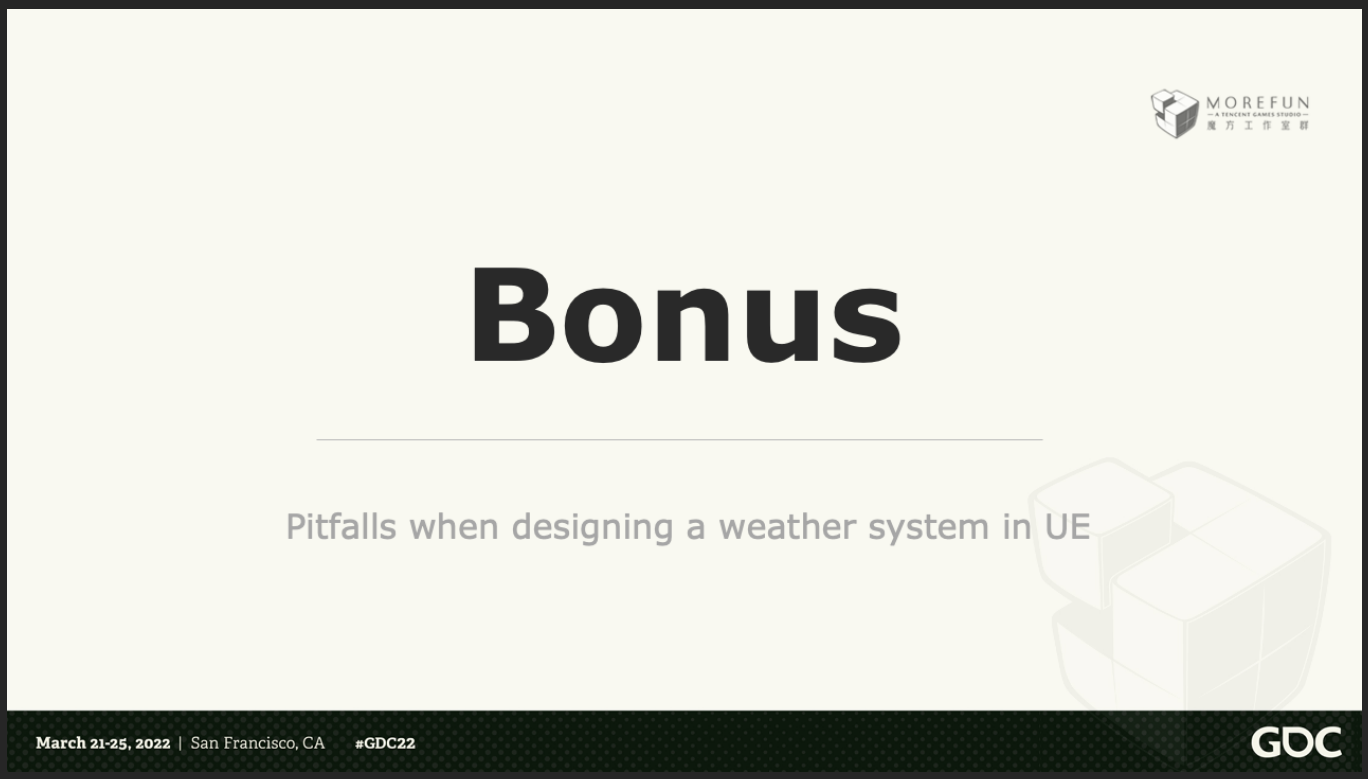
[click]

For lighting the cloud,

We add the lighting while render the skybox.

intensity is based on exponential falloff.

Cloud position is calculated by tracing against the cloud layer, thus result is not hundred percent accurate, but still, good enough for mobile



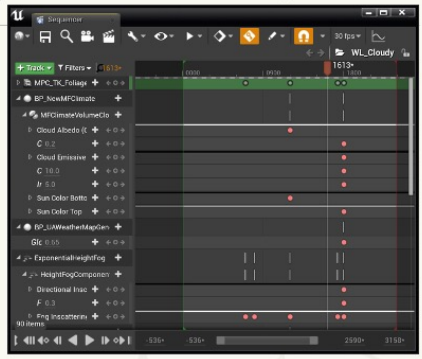
[Check time]

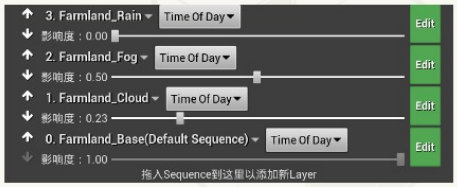
So the next part,

I’d like to share some extra things about user experiences and software engineering, in our system.

Weather Authoring

- **Add weather properties to Level Sequence**
  - Evaluated by TOD/Sun Angle
- **Layering sequences**
  - Values in higher layer overrides bottom layers
  - Layer has "opacity"
- **Require parsing Level Sequence manually**





First is how we define a weather in our system.

So normally we would think, we could just have a parameter struct, contains all the weather parameters,

For example, cloud coverage, fog density, etc..

Then different weather, is just a different group of values.

But in our case, we want our user could have most flexibility.

So weather parameters are not hard-coded.

[click]

Instead, user can directly add any property into control, by using the level sequence.

And the sequence will be evaluated based on time of day, or sun angle, depending on user setting.

So these weather parameters could changed with time.

[click]

Second, we allow multiple sequences to work at same time, by layering them in an order.

Values in higher sequence will override those in lower sequence.

And each layer has an **opacity** value. So by controlling the opacity value, you can fade the sequence in and out.

In such a way, user could group weather elements in different sequence,

For example, one layer for cloud, one layer rain, then user can just combine final weather using different opacities.

[click]

It should be noted, Level sequence by default doesn't support blending operation.

We did some coding to parse the level sequence and evaluate the values manually.

Weather Authoring

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

- "**Weather**" is a preset of layer opacity values
- Blend opacity values to change weather

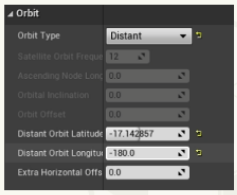
	Sunny	**Cloudy**	**Rainy**
Layer: Opacity	Base: 1.0 Cloud: 0.0 Fog: 0.2 Rain: 0.0 ..	Base: 1.0 Cloud: 0.7 Fog: 0.4 Rain: 0.0 ..	Base: 1.0 Cloud: 1.0 Fog: 0.8 Rain: 0.5 ..

So finally, a weather in our system, is just a preset of layer opacity values.
[click]

For example, let's say a sunny weather,
[click]
It would contain opacity of some layers, for example, base layer is 1, cloud is 0, fog is 0.2 etc.
And we can make another preset called cloudy,
[click]
Its cloud layer opacity is higher, fog layer is higher
[click]
And another one called rainy, which finally gives the rain layer some value so it's raining.
These presets are what gameplay module sees.
[click]
So when gameplay asks system to change weather, basically it just blends these opacity values.

Celestial System

- Position celestials(sun/star/any stuff in sky) correctly
  - **Time**
    - Time of day/Time of year/Time zone
  - **Geography**
    - Latitude/Longitude
  - **Celestial properties**
- Output direction in Unreal coordinate system



The next is celestial system.

This is what we used for positioning the sun or moon or anything on sky,

This system allows you define time and geography properties for the scene,

Including time of day, time of year, time zone, latitude and longitude etc.

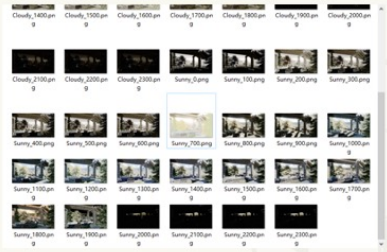
And properties of celestial,

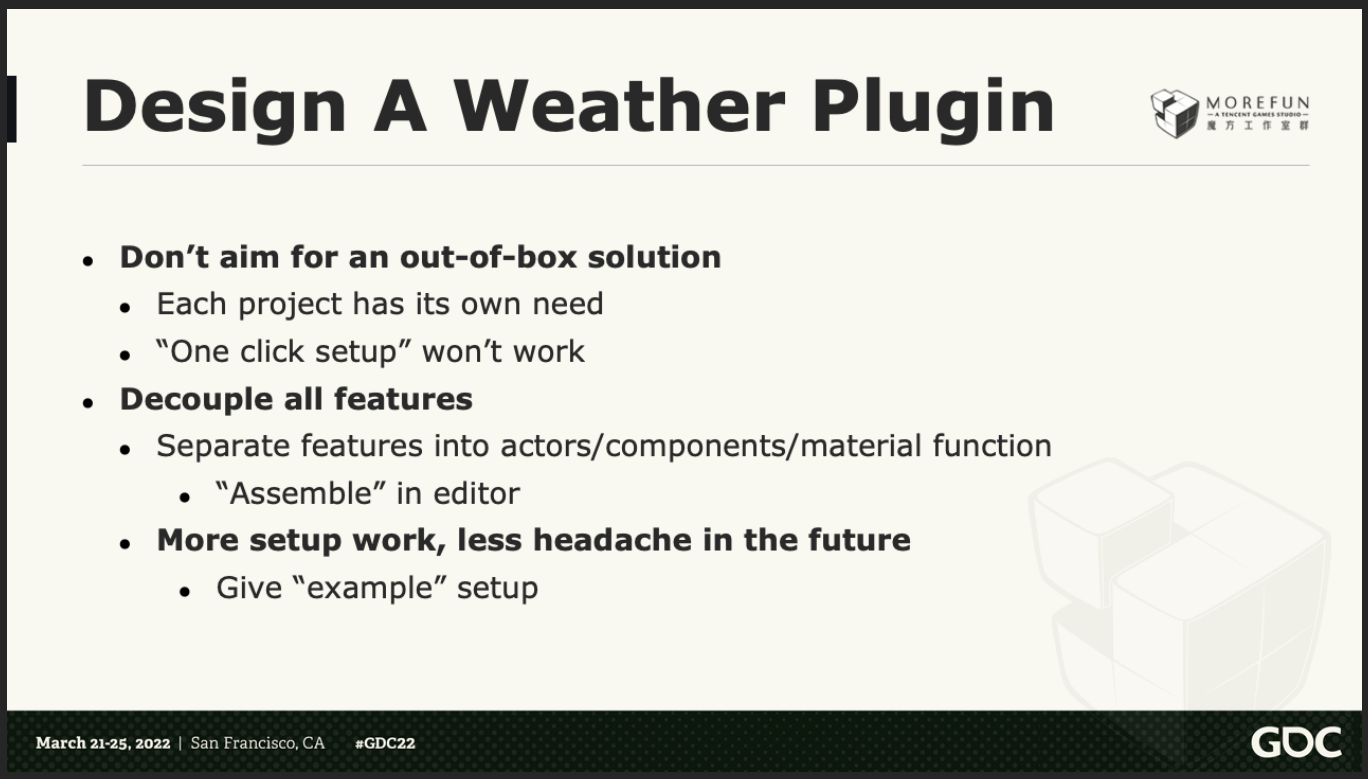
And the system could give a direction in the unreal coordinate system, pointing to the celestial position.

Image Reference Capture

MOREFUN
A TENCENT GAMES STUDIO
魔方工作室群

- Place virtual cameras in level
- Capture images under different TOD/Weather
- Support Cubemap/HDR





And last, here’s some advice for creating a similar weather plugin.

[click]

First, don’t try to create an out-of-box solution.

Each project would have their very specific needs.

For example, not every project needs volume cloud and physically based atmosphere.

And some project may have some special requirements about material, maybe all the material needs plug-in some material function etc.

If you’re doing an out-of-box solution, then such requirements make you have to maintain branch for each project.

And in Unreal, material and blueprint are hard to manage versions since they’re binary. Maintaining multiple branches will definitely be problematic.

[click]

We ended up with a solution, that try to decouple all the features,

All the features are separated into actor, component, or material functions

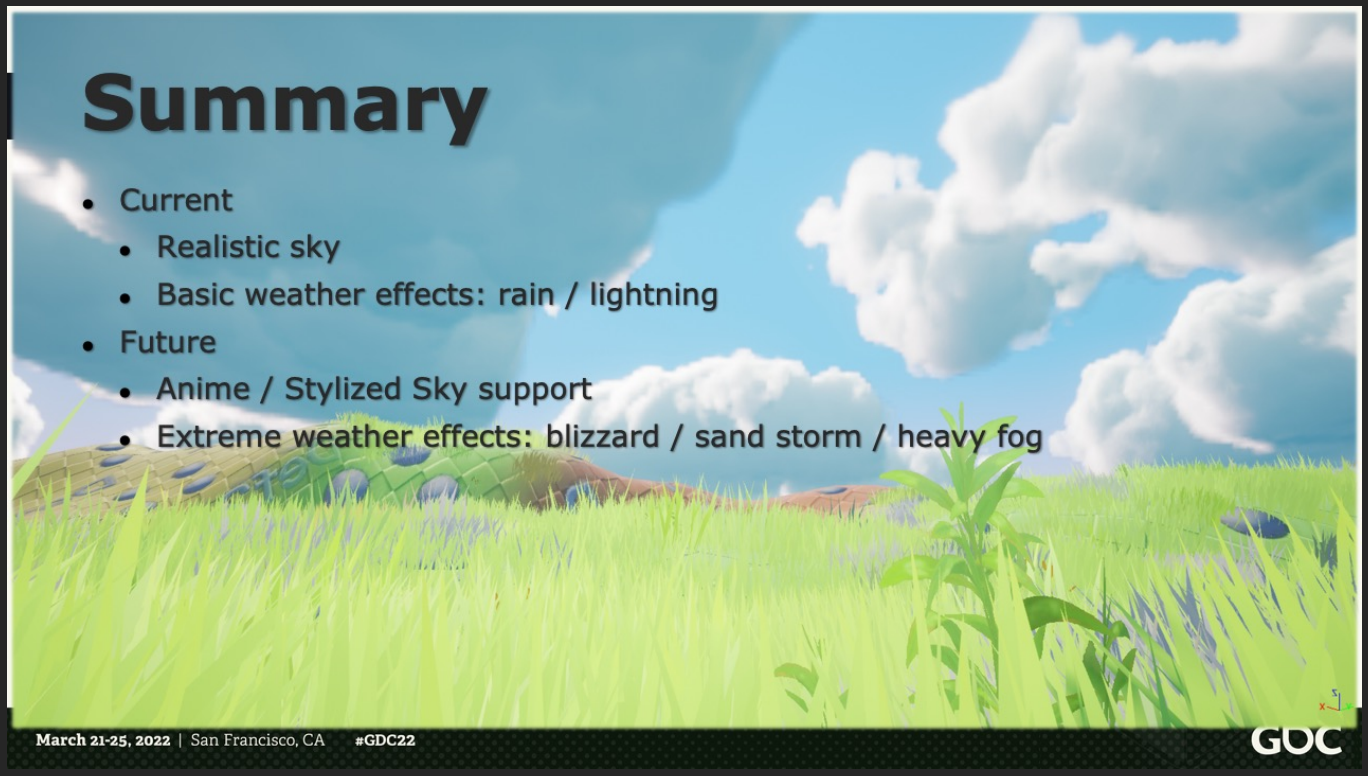
And user could just assemble these stuff in the editor, using blueprint editor or material editor.

This brings more work about setup, but much less headache in the future.

And we also have what we call “example” setup, which contains as many features as possible. so user could follow the example to start.

And that’s everything about our system design.

I’ll leave the summary part to KM.



Today we had shared how our weather system render realistic sky and some basic weather effects.

But it is not the end!

We are actually working on anime style support

and researching on how to bring more extreme weather conditions like blizzard, sand storm to mobile

Last but not the least, we would like to say thank you to MoreFun for providing us a lot of supports.

And especially our technical director, Milo, for giving us many good suggestions while preparing this presentation.

References

- [Shneider17] Nubis: Authoring Real-Time Volumetric Cloudscapes with the Decima Engine
- [Wrenninge13] Oz: The Great and Volumetric
- [Hillaire15] Physically Based and Unified Volumetric Rendering in Frostbite
- [Waylon16] The Technical Art of Uncharted 4
- [TSMG19] Lightning Strike at 103,000 FPS
- [Hillaire16] Physically Based Sky, Atmosphere and Cloud Rendering in Frostbite
- [Acton12] CSM Scrolling: An acceleration technique for the rendering of cascaded shadow maps
- [Zucconi] Volumetric Atmospheric Scattering
  https://www.alanzucconi.com/2017/10/10/atmospheric-scattering-1/
- [Krukar18] Light scattering in the Earth's atmosphere
  https://www.mkrgeo-blog.com/light-scattering-in-the-earths-atmosphere-part-3-clouds-haze-and-surface/

Here are the references, and we had try our best to list all of them.

We are also working on various areas of game technologies such as: GPU driven rendering, Real time global illumination, Fluid Simulation, Physical Animation

So we need a lot of *talented* person to join us.

If you are interested, please visit our website / send e-mails to km / simply scan this QR code .

Thank you very much and please remember to fill in the feedbacks for our session.

it is time for Q & A.