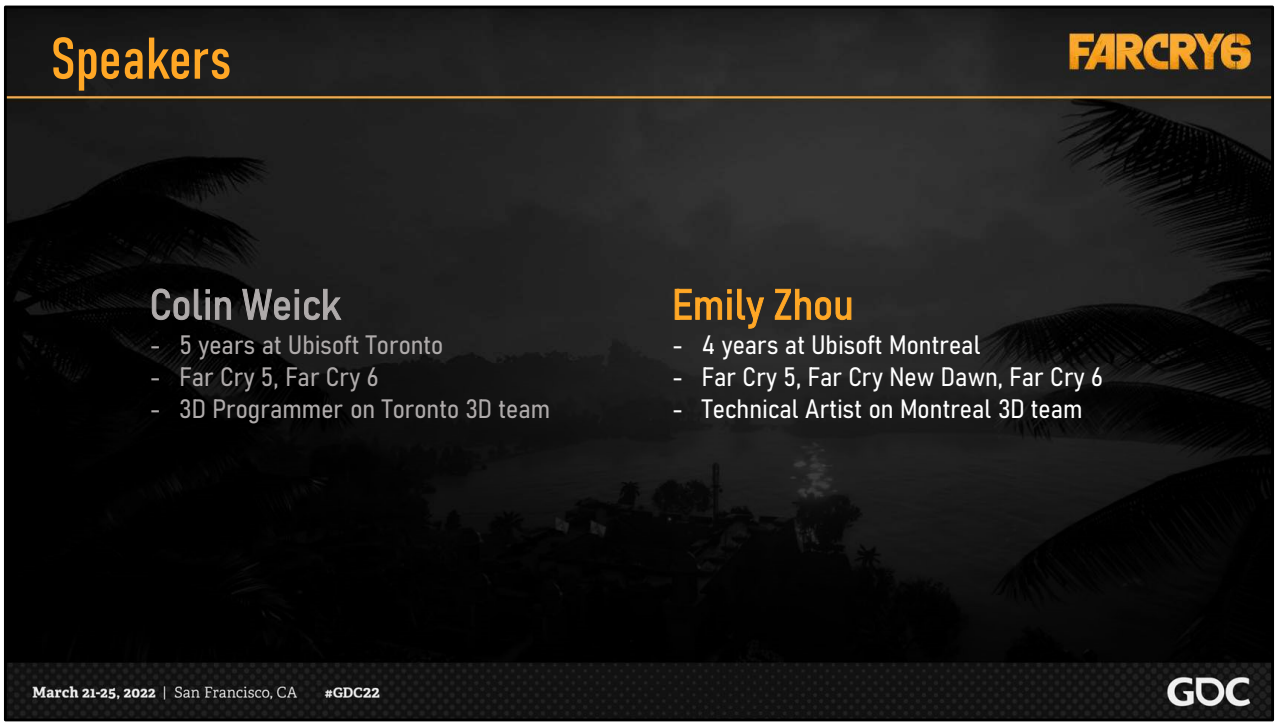
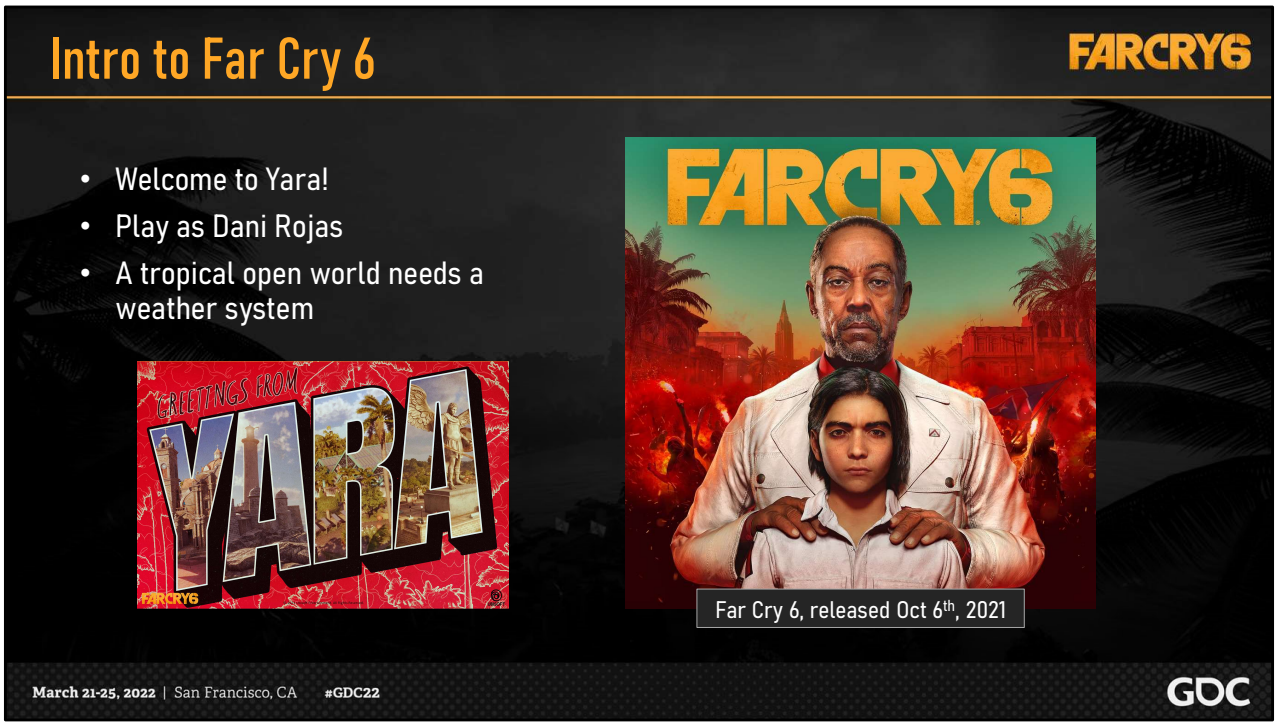
Hello everyone and welcome to our talk! The topic we will be presenting today is Simulating Tropical Weather in Far Cry 6.



Before we begin, let me first introduce myself.

My name is Emily Zhou, and I have been a technical artist at Ubisoft Montreal for the past 4 years.

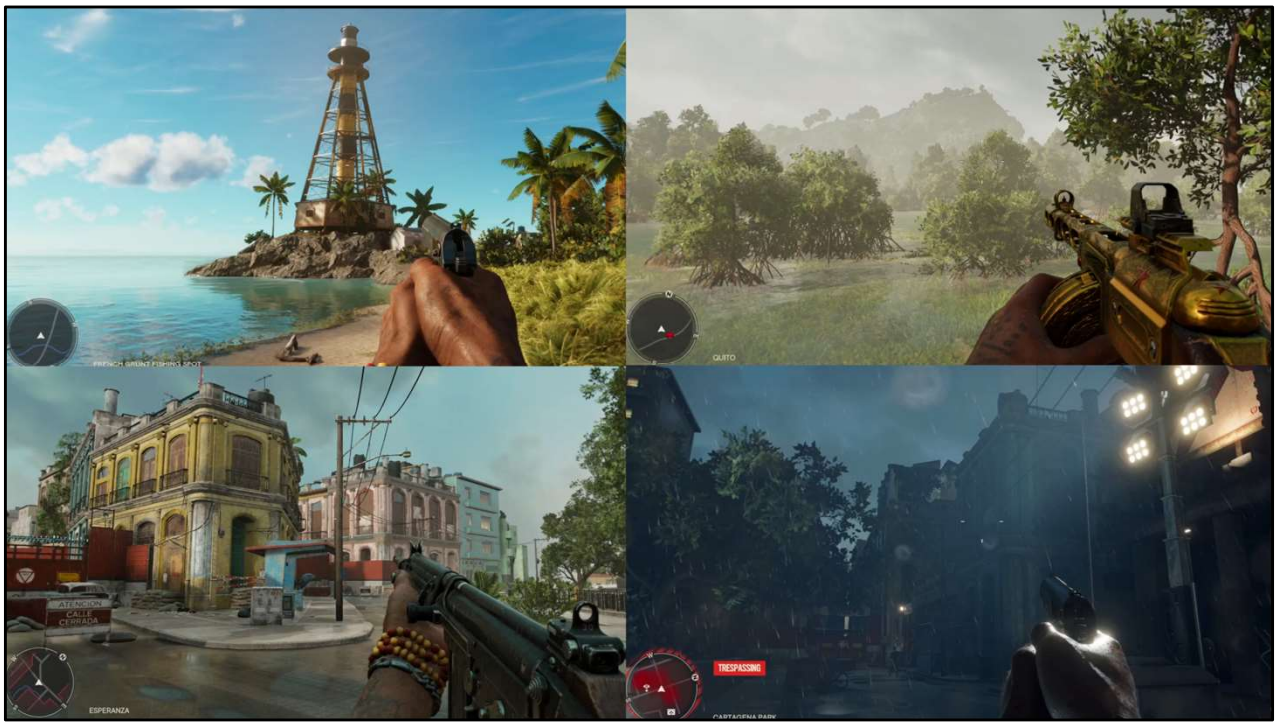
Colin will introduce himself a little later when he covers the second half of this presentation.



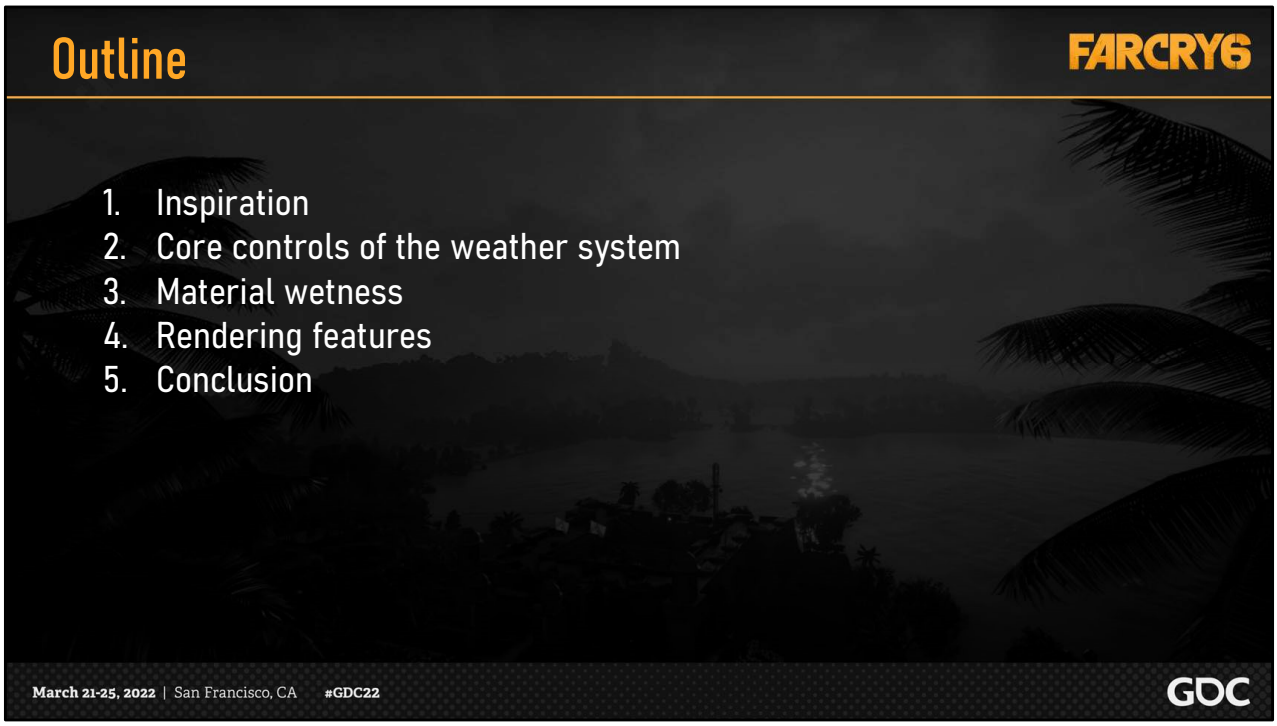
The game that our talk is centered around is Far Cry 6, the latest installment of the Far Cry franchise. If you are not familiar with Far Cry, it is an open-world first-person shooter series, where each title takes place in a new environment. In this case, we introduce the players to Yara, a fictional Caribbean island that was inspired by Cuba and other countries. Here, players will take on the role of Dani Rojas, a local rebel fighting to topple the regime of dictator Anton Castillo and his son Diego.

Far Cry's dynamic open worlds are a shining trait of the franchise – each world has a life of its own, with many systems constantly interacting with one another.
To add a tropical island experience to the mix, we needed something that would be new to Far Cry – a complete, dynamic weather system.

[Artist: Vincent L]



To give you an idea of what dynamic weather entails, here is quick sneak peak at what you might see in Far Cry 6 as a player.



Today, we will be presenting to you our weather system in its entirety. As such, we have condensed many topics into the following categories.

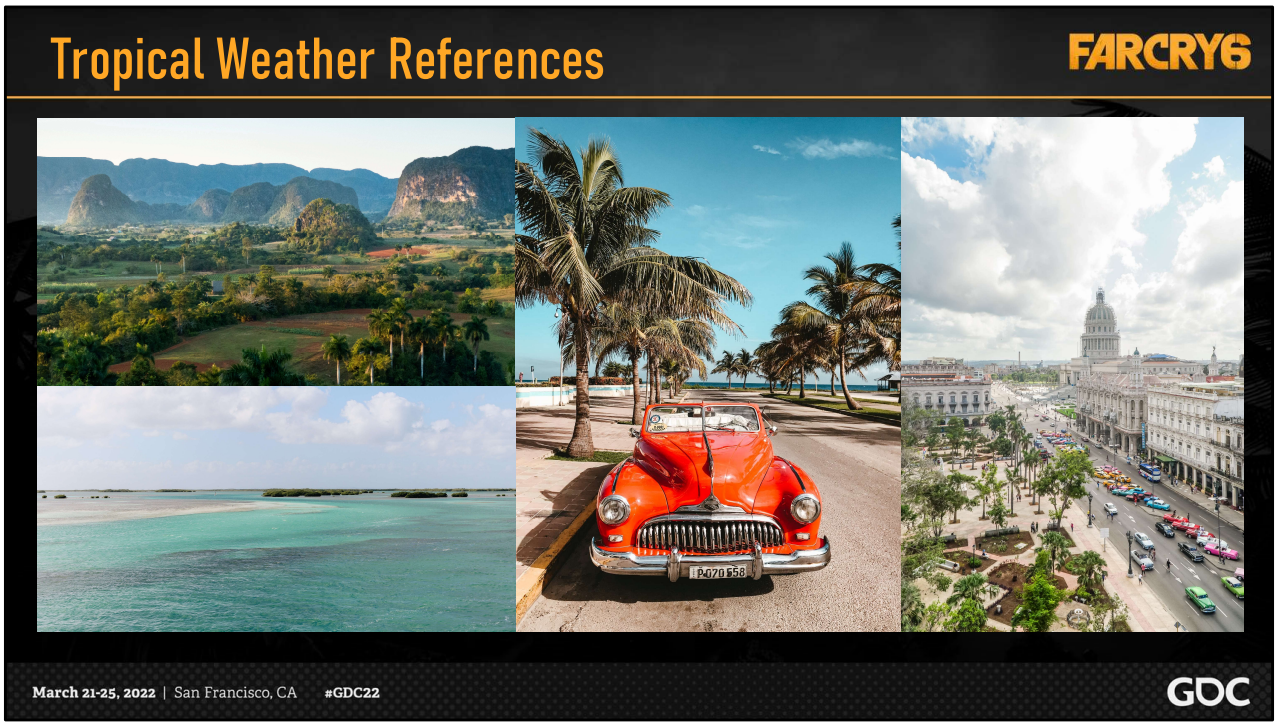
To start, I will go over our inspiration and core controls.

I will then explain how we tackled material wetness, by covering each asset type.

Following that, Colin will step through the technical details for each rendering feature that supports weather.

Lastly, we will conclude with some final thoughts.

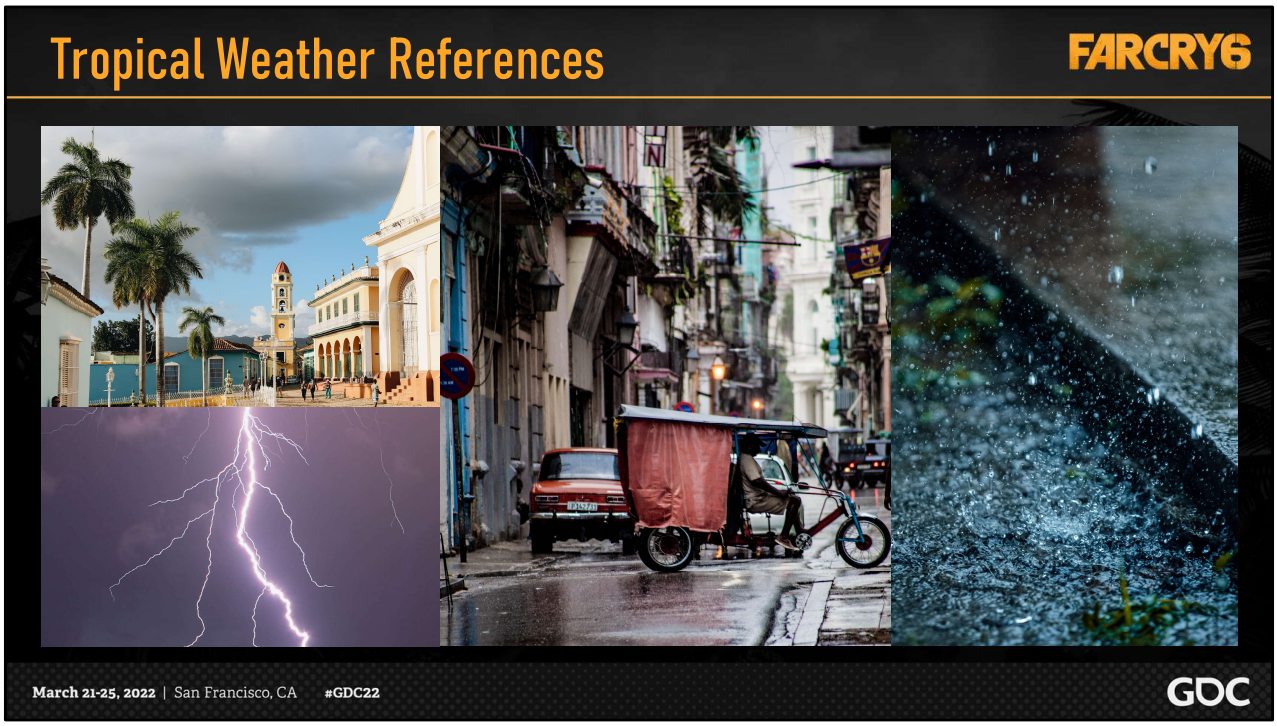
For our inspiration, we looked at the weather for various tropical locations, such as Cuba.



Island locations are typically home to very distinct and varied tropical weather.

At the start of the project, research was conducted on tropical scenery and weather so that we could give our players an authentic experience as they explored the world.

We soon found that we needed to incorporate both the iconic sunny weather shown here…



…as well as the flip side of tropical weather, which includes heavy rain and thunderstorms.

We needed to emulate the way that weather could change back and forth drastically, sometimes within hours.

[Stop]

We then took these elements that we wanted to highlight, and included them in our concept art.

Yara was pitched as an ideal tropical paradise. It should serve as a convincing escape into our game's fantasy.

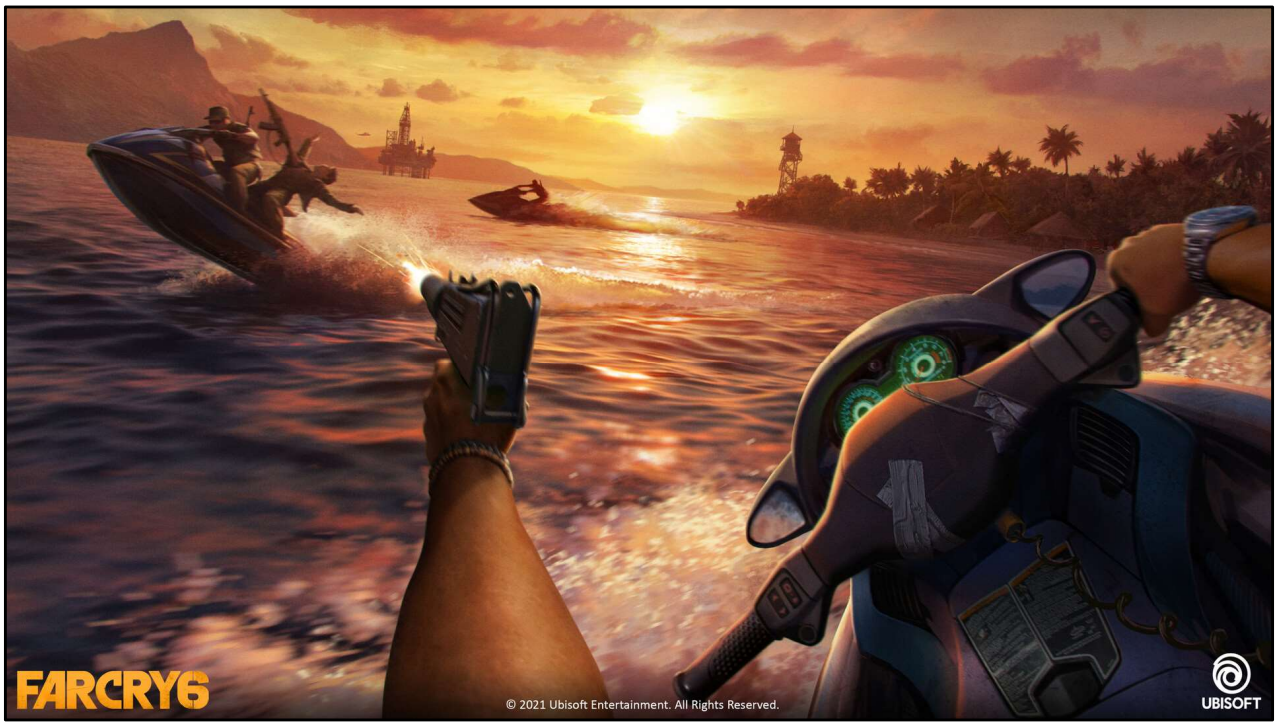
[Concept Art: FC6 Fankit]

However, our art direction wanted to contrast the picture-perfect weather with foreboding thunderstorms.

[Concept Art: Stephanie Lawrence]

When the player gets caught in a storm, the rain and wetness should be felt and convincing.

[Concept Art: Vitalii Smyk]



And of course, Yara is an island so the ocean is a key part of the equation, even for weather.

[Concept Art: Vitalii Smyk]

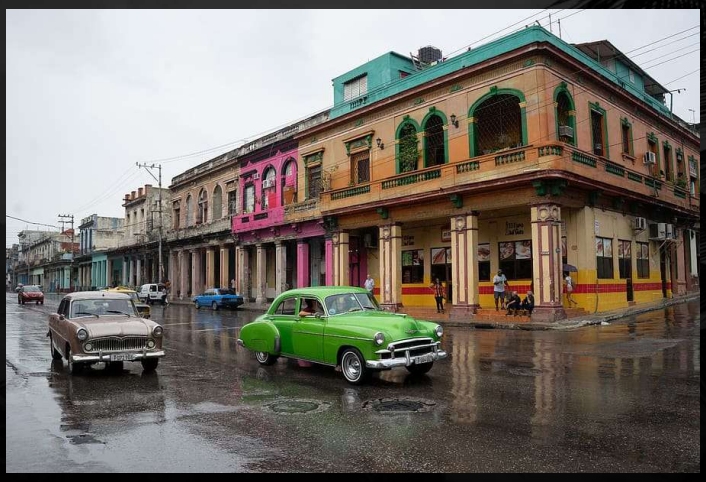
Goals

FARCRY6

- Convincing weather states
- Dynamic weather
- Natural transitions
- Efficient

Remember:

We are building a weather system for an open world game.



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

GDC

With that in mind, let's translate the direction into goals for implementing weather.

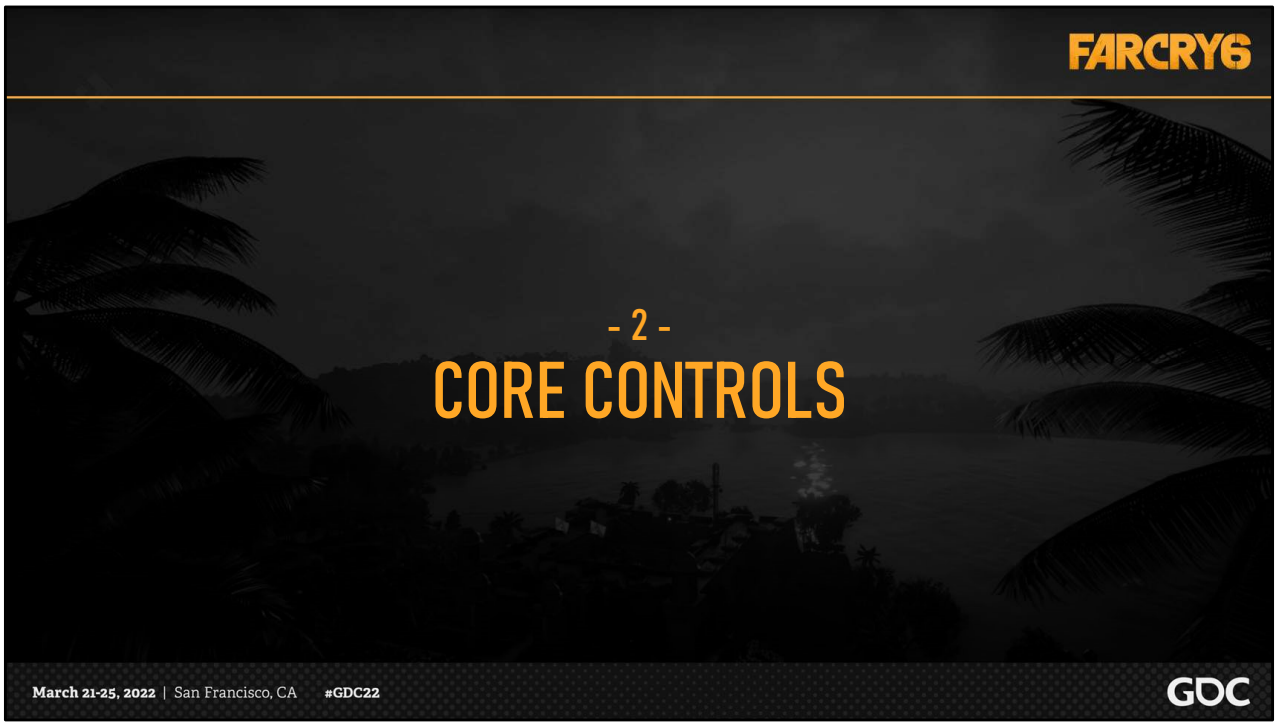
We want to have a collection of different weather states.

We want the weather to be dynamic and varied.

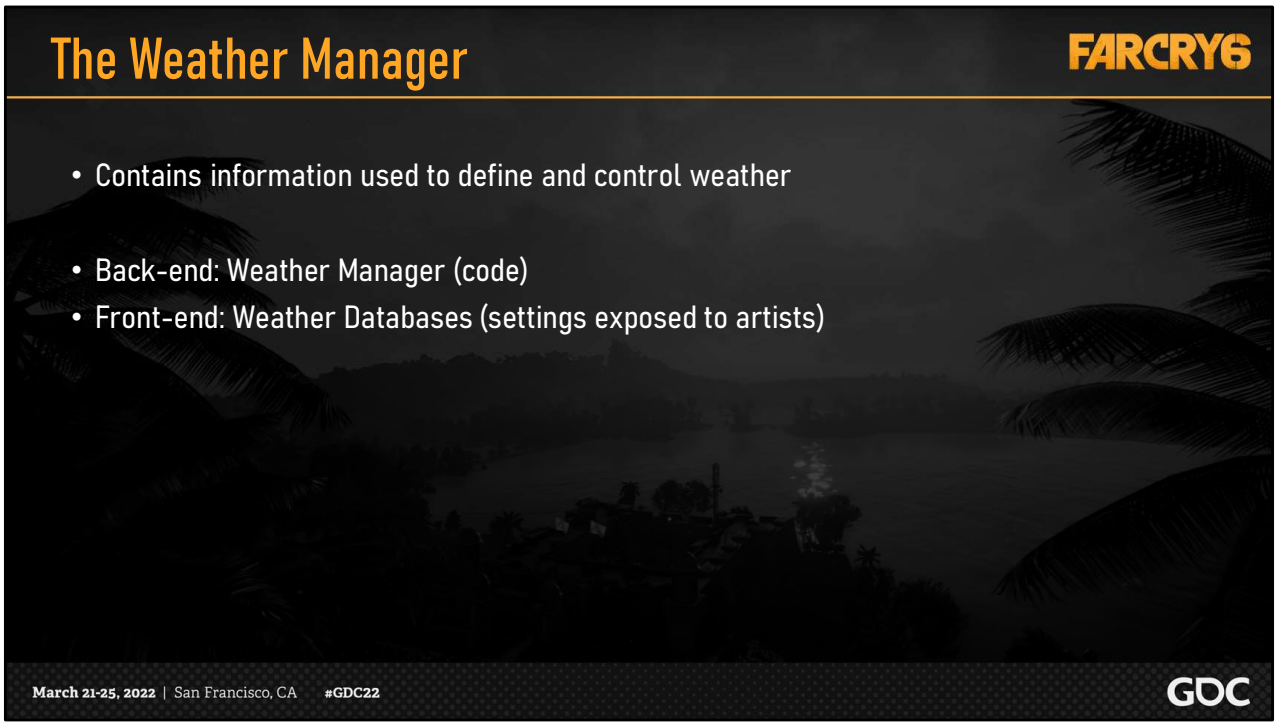
We want a system where the transitions make sense. For example, a storm should be preceded with darkening clouds and followed by remaining water puddles.

And of course, everything needs to be efficient and fit within our budgets.

We must emphasize here that this weather system will be for an open world game. Our decisions are often influenced by data and performance budgets, and the need to support all times of day and all locations in the world.



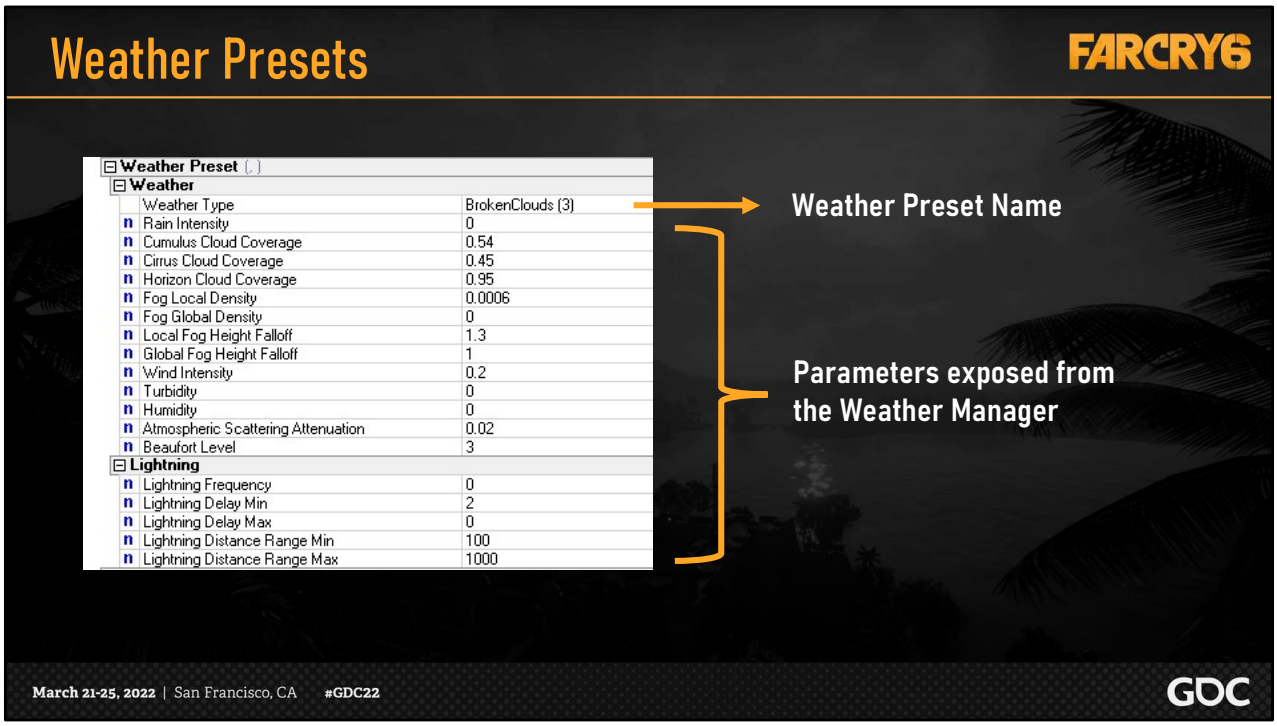
Let’s move on to our implementation, starting with the core weather controls.



The weather manager is the core code for our weather system and is what essentially drives weather behind the scenes.
It contains information used to define and control weather, some of which is exposed as settings in our weather database.

Consider the weather manager as the back-end of the system, and the weather database as the front-end.

Let's go over the setup for our system, starting with the weather presets.



The collection of weather types is referred to on the project as weather presets. These are defined in the weather database using the exposed parameters from the weather manager.

To get an idea of what we can achieve with these parameters, let's step through some of our final presets and compare a few of their values.

For the Few Clouds preset, we used low cloud coverage values, shown in the bottom right.

In comparison, the Broken Clouds preset uses higher cloud coverage to give us big fluffy clouds.

Next we can see that the Mist preset has added light fog in the distance.

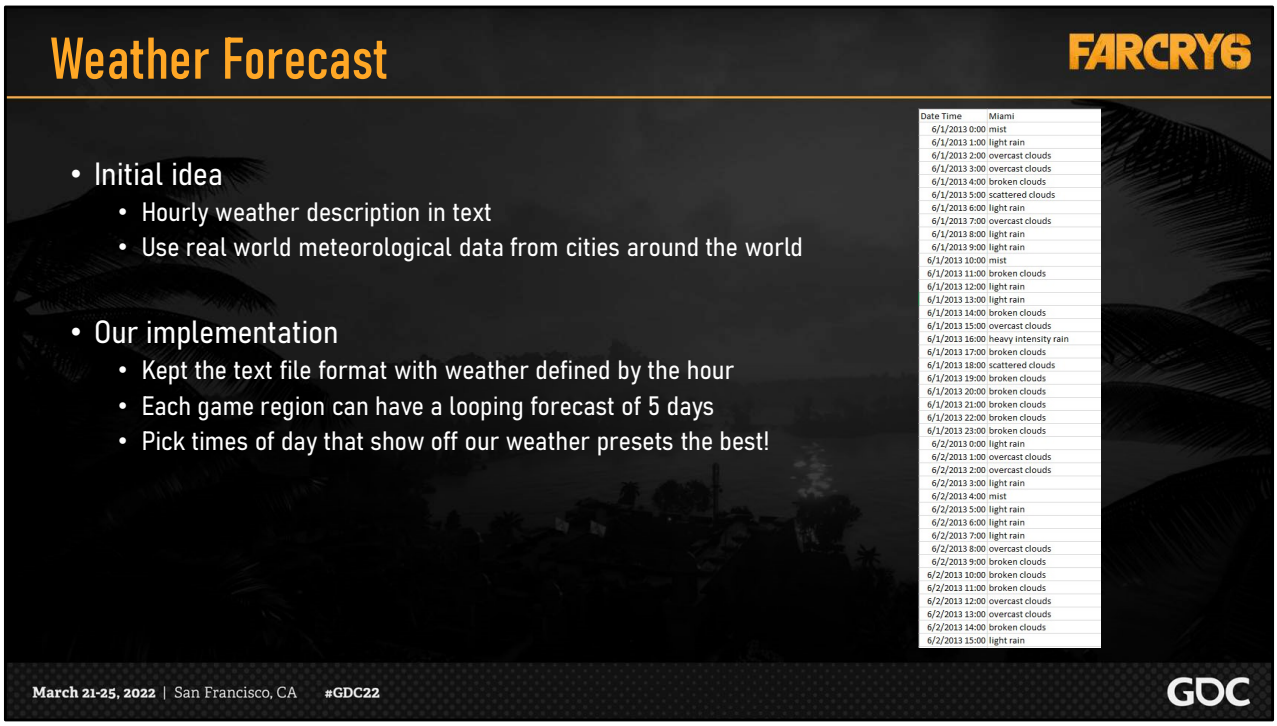
In comparison to that, we have our heavy Fog preset, with fog values cranked up much higher.

The next few presets will show the progression of rain intensity, starting with Light Rain…

Then moderate rain…

Followed by heavy rain, which maximizes the rain intensity and darkens the sky.

Which leaves us with our most intense preset, the Thunderstorm, which adds lightning.

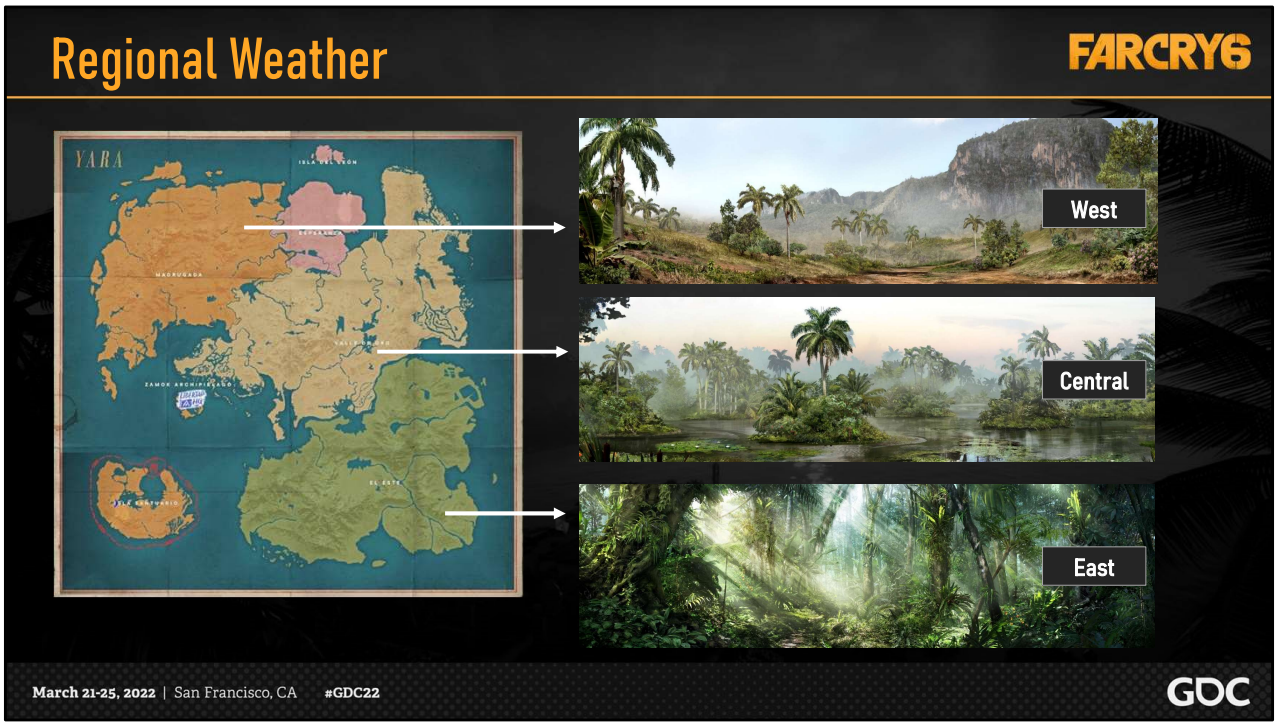


Now that we have weather preset building blocks, we need to create a cycle or weather pattern.

In the very beginning, our initial idea was to collect and use real world meteorological data from cities such as Miami. As shown in the image, we obtained the description of the weather at every hour from a period of time 2013.

Although it would have been nice to use real world data to drive our forecast, we wanted more artistic control.

The method that we ended up going with was a similar text file, but we timed our chosen weather presets ourselves, and designed up to 5 full day cycles per region.
This freedom also allowed us to showcase our weather presets where they looked the best!



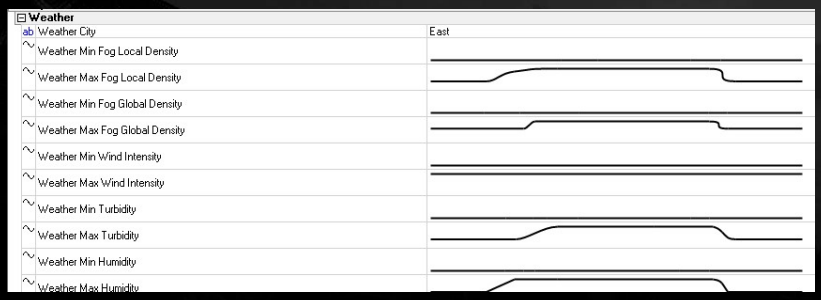
Now we need to take some time to discuss regions. The world of Far Cry 6 has been subdivided into 3 main regions, West, Central and East, each with it's own visual identity.
The west was the dry region, central was the wetlands, and the east was the jungle region.

To enhance these distinctions, we wanted our weather to differ between each region.

Regional Weather

FARCRY6

- Regions also include zones like interiors, caves, etc.
- Each region has limits to certain weather settings
  - E.g. interior limits fog to 0
- Moving around the world blends min/max values



Note that our definition of regions also extends to smaller zones, such as interiors, caves, and anything else that artists might need to define.

For each region, we exposed min and max curves to limit certain weather properties. This was how we altered weather based on where the player is moving in the world.

An example usage case would be how we set the max fog curve to 0 for the indoor zone, to prevent fog from appearing inside.
The limitation to this is that, if your indoor area has windows, you would see the fog disappear outside as soon as you walked in. This was why we assigned our zones carefully.

Our weather manager then took in all the potentially overlapping regions around the player, and interpolated the curves accordingly to output the adjusted weather.

Weather at Runtime

FARCRY6

- Overrides
  - Missions
  - Pre-rendered cutscenes
  - Replicated weather for co-op players

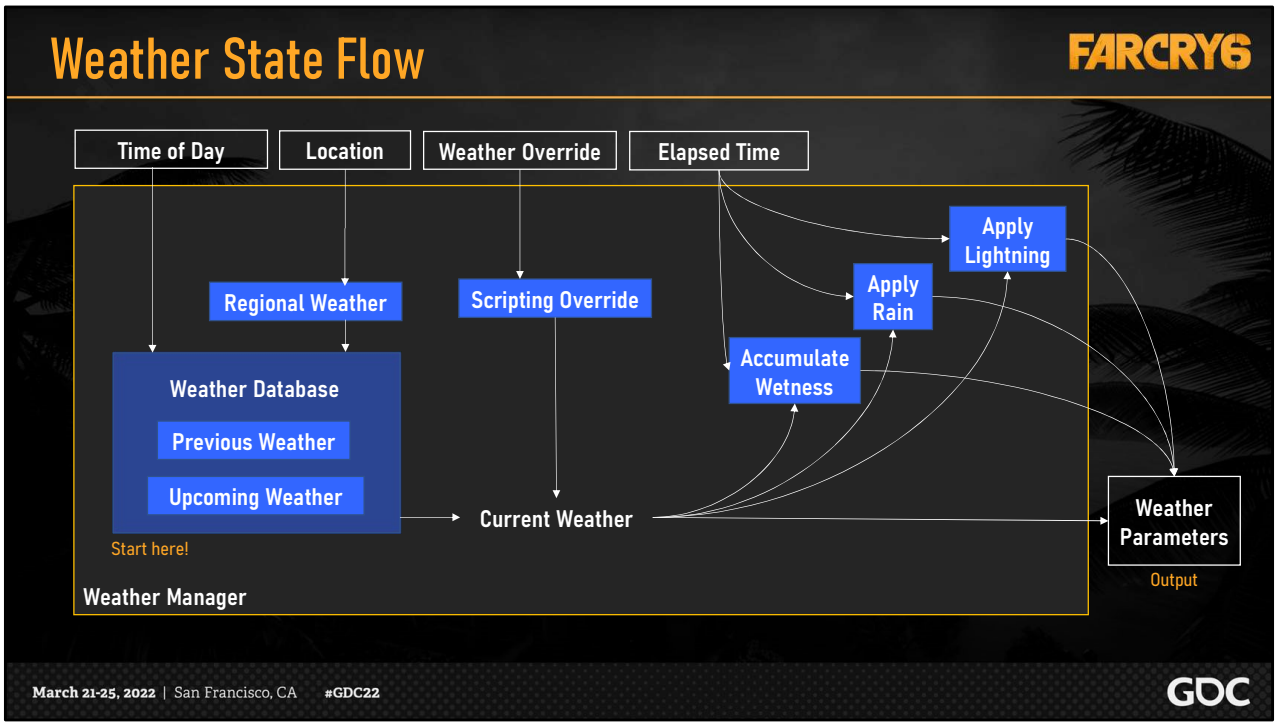
Opening mission of Far Cry 6

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

GDC

The weather manager also needed to allow for overrides, such as for gameplay missions and pre-rendered cutscenes.
For example, the opening mission of the game has the player fleeing through the city streets in the middle of a nighttime thunderstorm.

Additionally, our game is available in multiplayer co-op, so weather needed to be replicated for all players.



This flow chart illustrates the final order of operations.

To summarize, we first consult the forecast to get the weather preset at the current time of day. Then, based on where the player is, we apply regional adjustments. This gives us the current weather state, which can be overridden for gameplay.
Finally, we use this weather state to update the wetness, rain and lightning with time, and output the final parameters. These are variables like WetnessFactor, RainFactor, etc., which we can now access to drive visuals, audio, gameplay and so on.

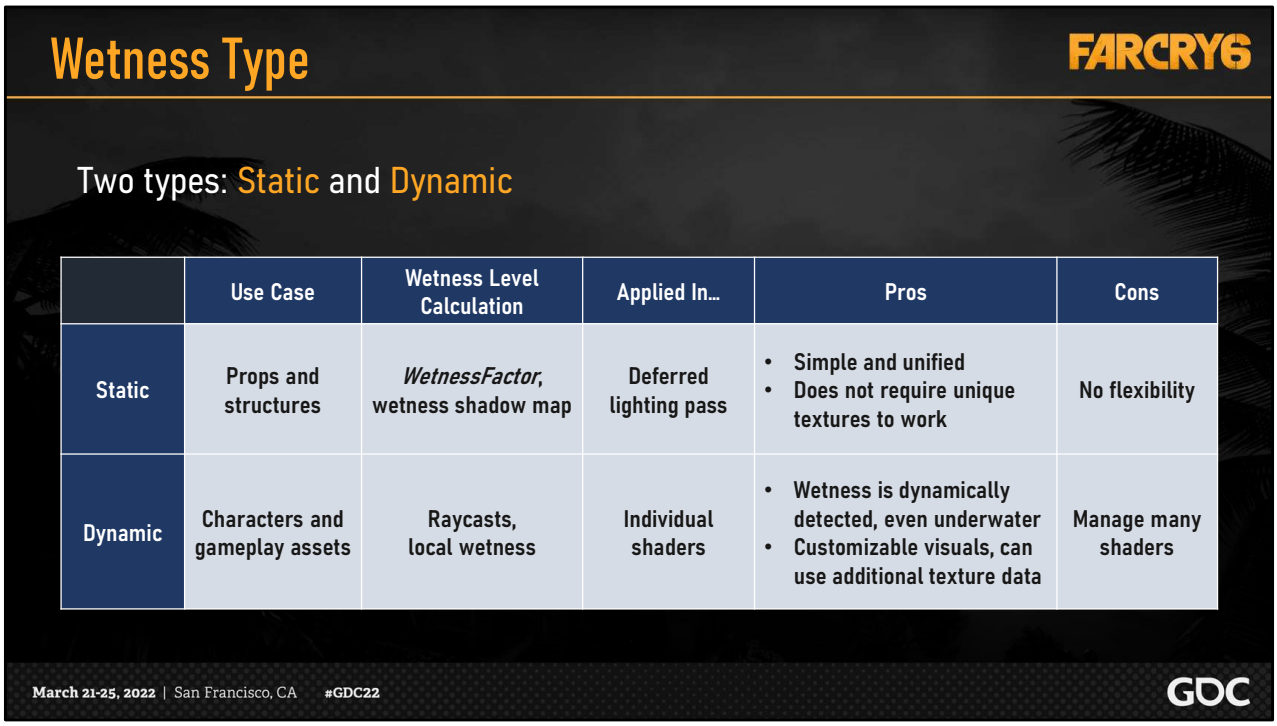
Now that we've established the inner workings of our weather manager, we need our materials to respond. In other words, how did we make our assets wet?

Wetness is a huge component of weather; when it rains, we need to see the world change. This means that every asset needs a wet state, so that we can blend between dry and wet.

The risks associated with creating a wet version of every asset are:
1. There are too many assets, so requiring extra data will add a lot of production time
2. Our art teams all have their own asset pipelines and shaders. We could be risking a lack of cohesion, which would be hard to control given our project's scale.

Our solution was to work in parallel with asset creation. To do this, it needed to be simple and unified as much as possible.
We also decided that it would be primarily tech-artist driven and it should work "out of the box".
This means that we should be able to drag and drop assets in the world, and they should get wet in the rain. Of course, some materials receive more detail and polish, but for the simplest props or legacy assets, wetness should work without revisiting them.

So, let's dive into our wetness implementation.



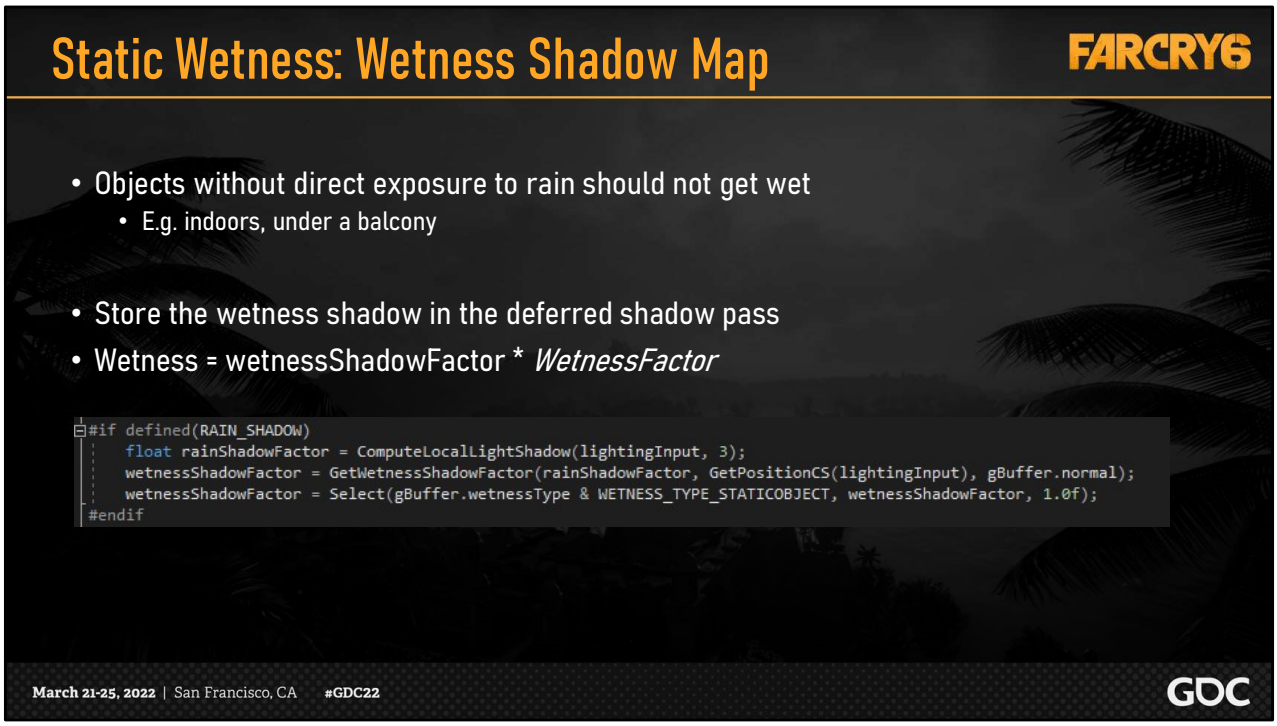
	Use Case	Wetness Level Calculation	Applied In…	Pros	Cons
Static	Props and structures	*WetnessFactor*, wetness shadow map	Deferred lighting pass	• Simple and unified • Does not require unique textures to work	No flexibility
Dynamic	Characters and gameplay assets	Raycasts, local wetness	Individual shaders	• Wetness is dynamically detected, even underwater • Customizable visuals, can use additional texture data	Manage many shaders

First and foremost, we split our solution into two parts by having two wetness types: Static and Dynamic

As their names imply, static wetness is for objects that will never move, and dynamic is for objects that will move.

Static wetness applies mostly to our Object Bank assets which includes props and structures.
The wetness level is determined by using the WetnessFactor parameter provided by the weather manager. We also use a wetness shadow map to mask out wetness where appropriate.
Static wetness will be the simplest visually, and will be applied in the deferred lighting pass, which means it will be applied in one place,
The pros to static wetness is its simplicity and the fact that everything can be tweaked at once. But the con is that there is no flexibility.

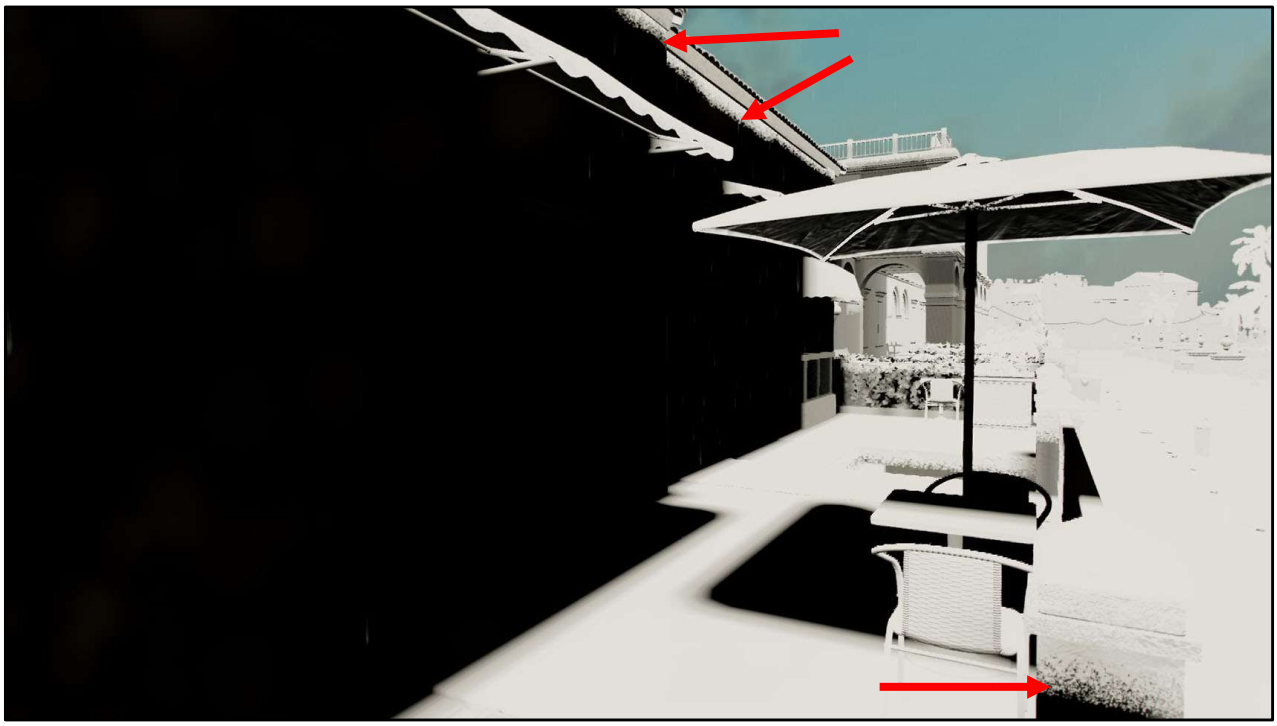
Dynamic wetness is reserved primarily for weapons, vehicles, and characters.
Their wetness level is calculated by raycasts to detect exposure to rain. Their wetness also includes a bonus feature called local wetness which handles submersion in water.
For dynamic wetness, the visual change will be applied in each individual shader; which means that we can tailor the look a bit more.
The con to this is that we have to manage every shader that could be used for dynamic assets.



First, let's look into the static wetness type. The main wetness calculation needed is the wetness shadow map, which occludes objects that are, for example, under a balcony or indoors. This is important because we reuse our props, so their wetness should be accurate no matter where they are placed.

Since our static wetness is applied in the deferred lighting pass, we stored the wetness shadow in our deferred shadow pass.
We then multiply the sampled shadow with the weather manager's WetnessFactor to get the final wetness.

Let’s look at a wetness shadow example. In this scene, there are lighter, dry areas on the concrete floor and on the wooden table.



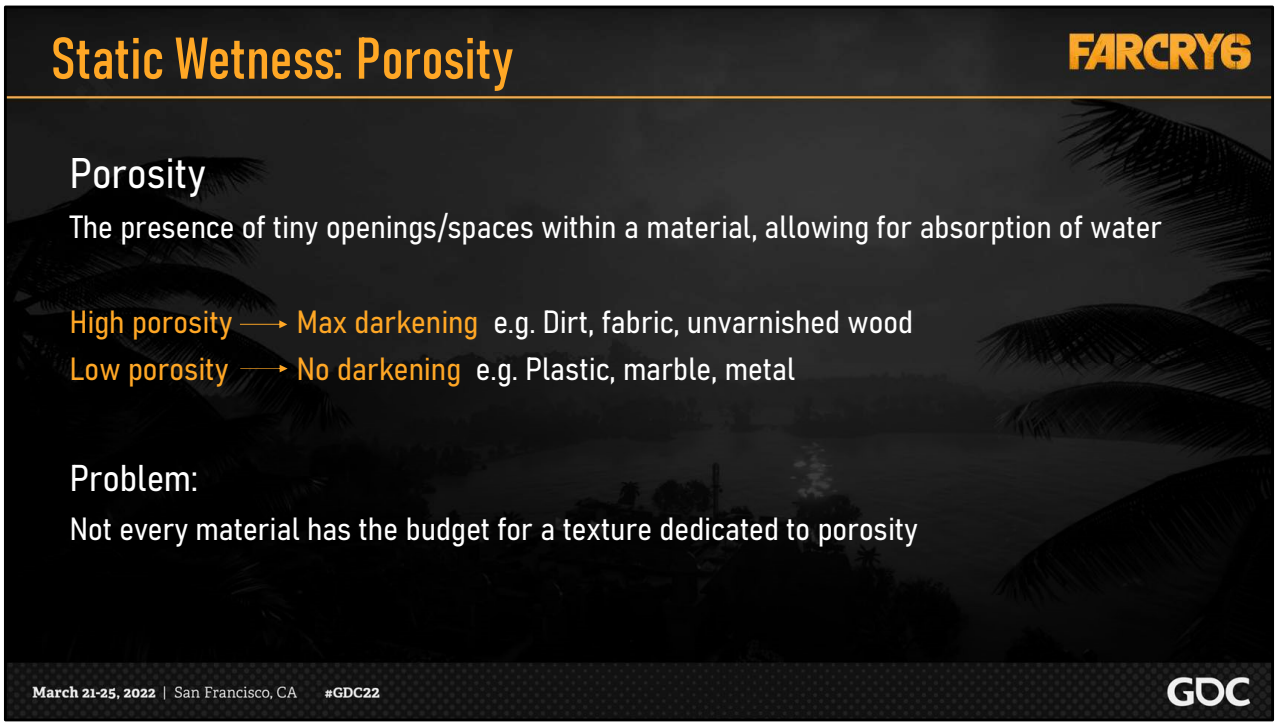
Here is a visualization of the wetness shadow. As you can see, it works quite well, but there are some precision limitations, particularly on vertical surfaces.

[Click for animation]

If you look closely at the highlighted areas, you will see some speckled details. This is because the cutoff was originally a harsh line, which we softened using dithering.

Now let's move on to how we applied the visual of static wetness.

We first referred to photo references to analyze what inputs we would need and what changes they would drive.
The two visual changes that we identified as necessary were darkening the albedo and increasing the smoothness. Some materials like cardboard would darken but wouldn't get too shiny. Materials like tiles would not darken but would get much shinier. Some materials fall in between. The key to all this was how absorbent the material was.

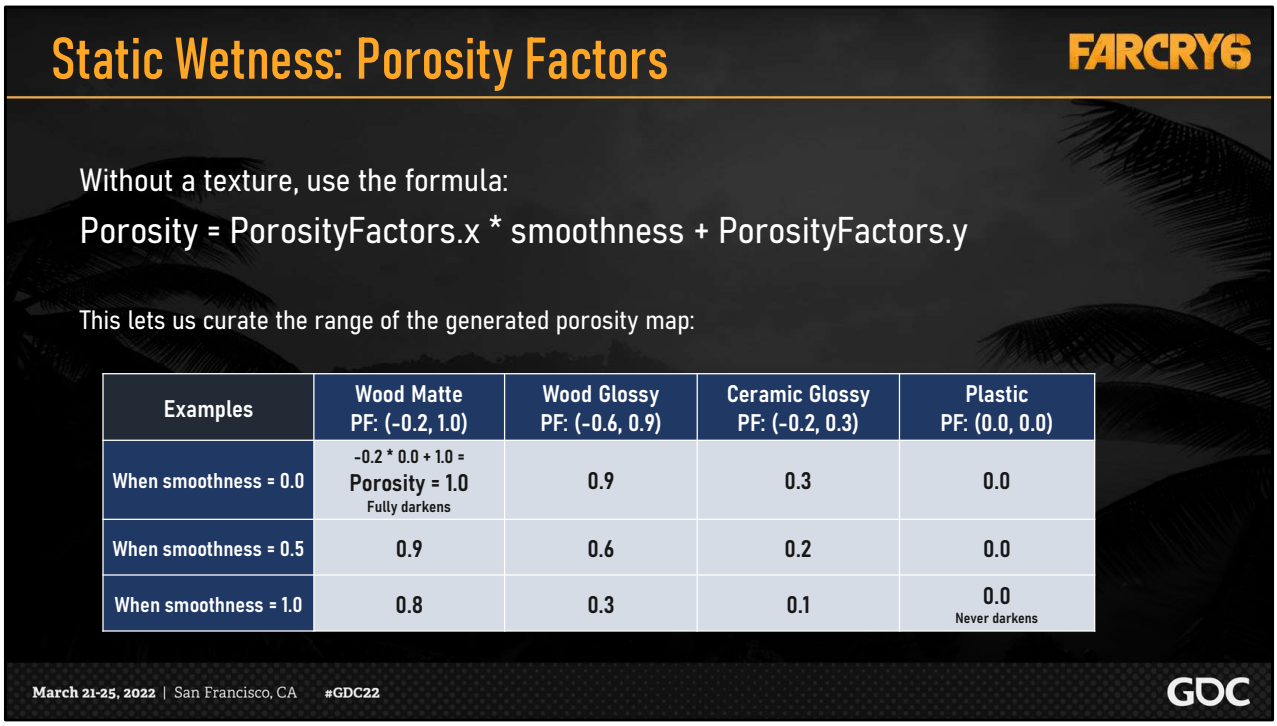


As a result, the input we chose was Porosity.

Porosity is most directly related to albedo change.
High porosity materials like dirt, fabric, and raw wood will become saturated and appear very dark.
Low porosity materials like plastic, marble, and metal will have water pooling on their surfaces rather than getting absorbed, which means that their colors should be unchanged.

Now that we have our input, we could just use a porosity map and move on.
Except for one problem - not every material can afford an extra texture map; that would exceed our budgets for not much reward.

So, we need a way to derive porosity whenever a porosity texture is not available.



This is where we introduce porosity factors. Since porosity and smoothness are properties that are conceptually connected, our solution was the following basic formula, where the two PorosityFactor values are floats that we could specify per material type.

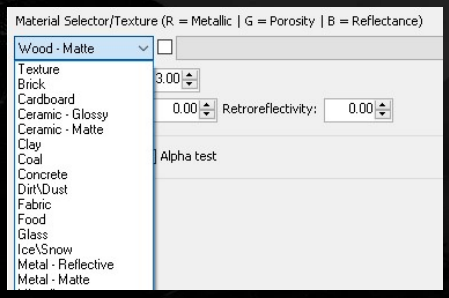
This formula essentially generated a porosity map with all the details of the smoothness map, but with its values lying in the range that we curated.

Static Wetness: Material Setup

FARCRY6

Make use of the existing Material Selector with hardcoded preset values:

Artist material setup



Material presets defined in the shader code

```
<parameter name="BrickProperties"            type="float4" defaultvalue="0,0.4,0.16,-0.2"
<parameter name="CardboardProperties"        type="float4" defaultvalue="0,1.0,0.21,-0.2"
<parameter name="CeramicGlossyProperties"    type="float4" defaultvalue="0,0.3,0.39,-0.2"
<parameter name="CeramicMatteProperties"     type="float4" defaultvalue="0,0.9,0.16,-0.6"
<parameter name="ClayProperties"             type="float4" defaultvalue="0,0.9,0.26,-0.6"
<parameter name="CoalProperties"             type="float4" defaultvalue="0,0.9,0.20,-0.6"
<parameter name="ConcreteProperties"         type="float4" defaultvalue="0,0.8,0.24,-0.6"
<parameter name="DirtProperties"             type="float4" defaultvalue="0,0.6,0.19,-0.3"
<parameter name="FabricProperties"           type="float4" defaultvalue="0,1.0,0.31,-0.2"
<parameter name="FoodProperties"             type="float4" defaultvalue="0,0.9,0.22,-0.6"
<parameter name="GlassProperties"            type="float4" defaultvalue="0,0.0,0.46, 0.0"
<parameter name="IceProperties"              type="float4" defaultvalue="0,0.0,0.05, 0.0"
<parameter name="MetalReflectiveProperties" type="float4" defaultvalue="1,0.0,0.84, 0.0"
<parameter name="MetalMatteProperties"       type="float4" defaultvalue="1,0.0,0.16, 0.0"
```

Properties = float4(Metallicity, PorosityFactors.y, Reflectance, PorosityFactors.x)

Luckily for us, our object shaders already had a dropdown of material presets with hardcoded PBR values. All we had to do was add in the porosity factors.

This kept the artist workflows unchanged and prevented any setup bugs.

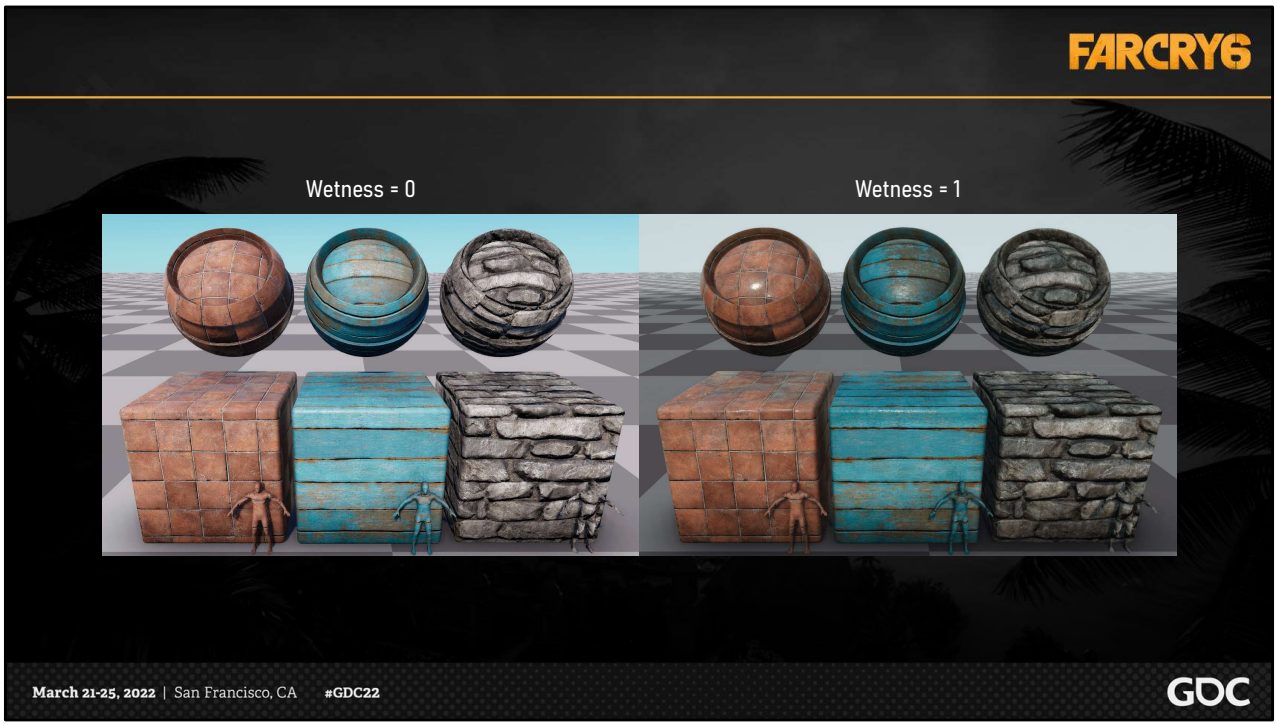
Static Wetness: ApplyWetness()

FARCRY6

- Darken the albedo using porosity
- Boost the smoothness
- Not perfect - we judge the overall scene, not per asset

```
void ApplyWetness( inout float3 albedo, inout float smoothness, inout float metallic, inout float specularReflectance, float wetness, float porosity)
{
    // From vertex shader, underwater adds +1.0 to wetness value. Decompose that into mask and actual wetness.
    float underwaterMask = saturate(2.f - wetness);

    // Darken the albedo based on porosity, except if metallic
    albedo             *= lerp(1.0f, 0.25f, wetness * porosity * (1.0f - metallic));

    // Generate a wet version of the smoothness map using porosity and original smoothness
    float baseSmoothness = lerp(0.9f, 0.6f, porosity);
    float smoothnessAdjustment = lerp(-0.6f, 0.4f, sqrt(smoothness)); // Use an exaggerated version of the smoothness map to add or subtract wetness
    float finalSmoothness = saturate(baseSmoothness + smoothnessAdjustment);
    // The new smoothness should not be lower than before or higher than max of 0.9
    finalSmoothness = clamp(finalSmoothness, smoothness, 0.9f);

    // Apply new smoothness and reflectance, except if underwater
    smoothness          = lerp(smoothness, finalSmoothness, wetness * underwaterMask);
    specularReflectance = lerp(specularReflectance, WaterReflectance, wetness * underwaterMask);
}
```


For an approximation with very few inputs, it would be impossible to get every material response correct.
It was more important to judge the bigger picture rather than individual assets. This helped us avoid an endless loop of tweaking values.



Here is a dry vs wet comparison for some of our materials.

Now let’s look at a full scene of assets transitioning from dry…

… to wet.

Dynamic Wetness: Wetness Component

FARCRY6

- Characters, weapons, vehicles
- Perform a raycast every few frames to check if exposed to rain
  - Vehicles need multiple raycasts
- Wetness increase/decrease happens gradually
- Additional feature: local wetness
  - Handles wetness from submersion in water

Now, if we recall, the benefits of dynamic wetness was that we could customize the way we apply wetness. To gather our ideas, we once again revisited real world reference, such as staring at parked cars outside in the rain.

Dynamic Wetness: Characters

FARCRY6

- Clothing
  - A clothing-specific drop-down menu this time
  - Also included a max smoothness for each material type
- Hair
  - Only one set of porosity factors

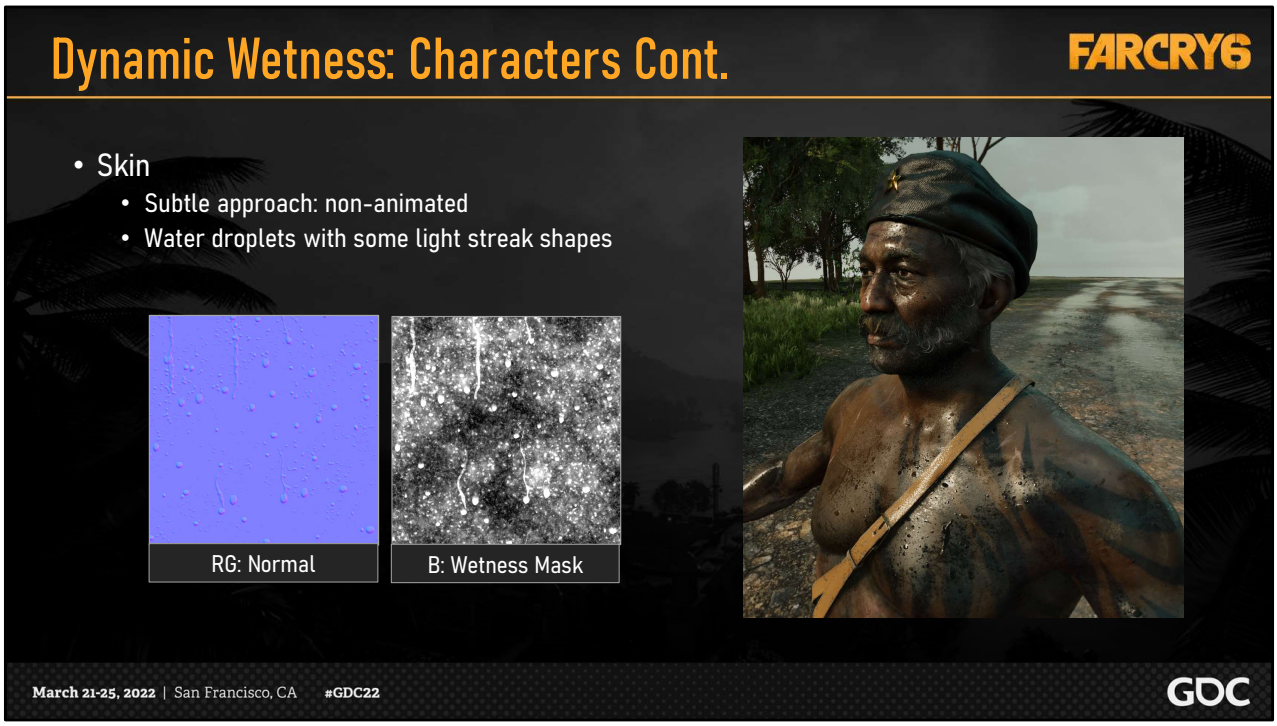



March 21-25, 2022 | San Francisco, CA #GDC22

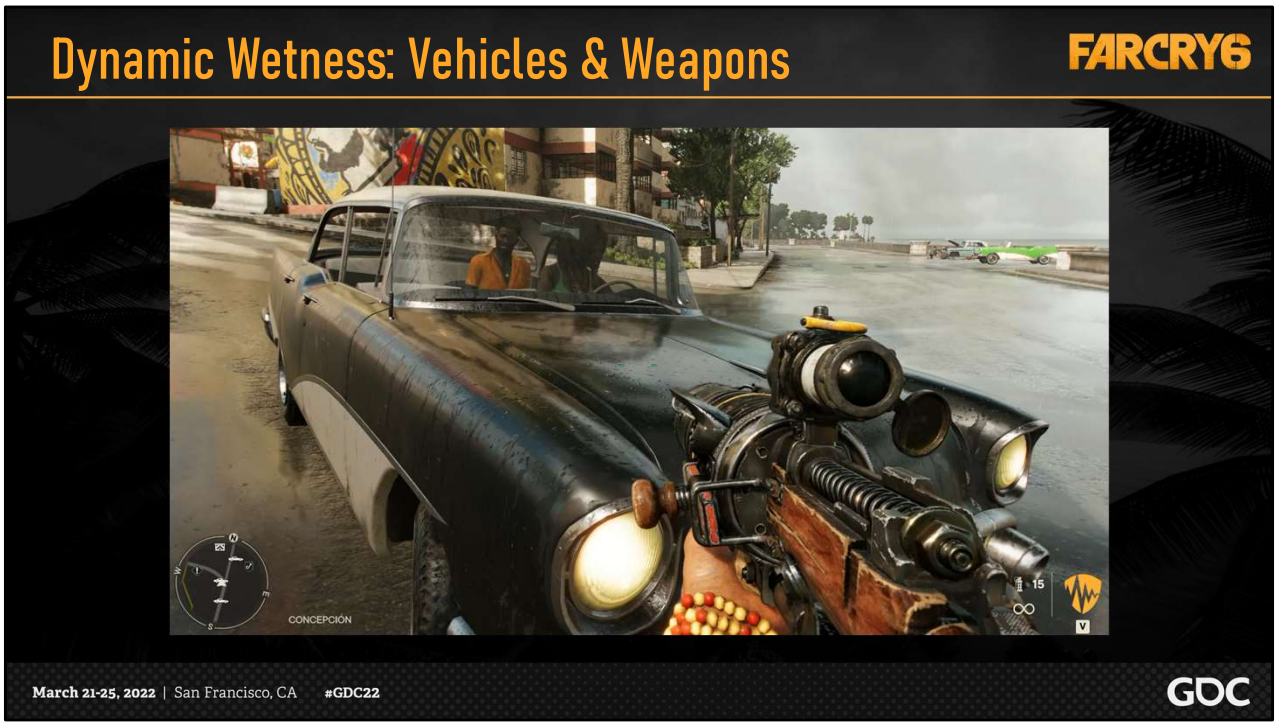
GDC

For character clothing, we utilized the same approach as for our props. We used an existing dropdown menu but also added an upper limit to the final smoothness, to give us better results for wet fabric.

Character hair was the simplest change we made; we only used one set of porosity factors. At the early stages of brainstorming, we wanted a way for hair to clump together when wet, but it was too ambitious for our scope of work.

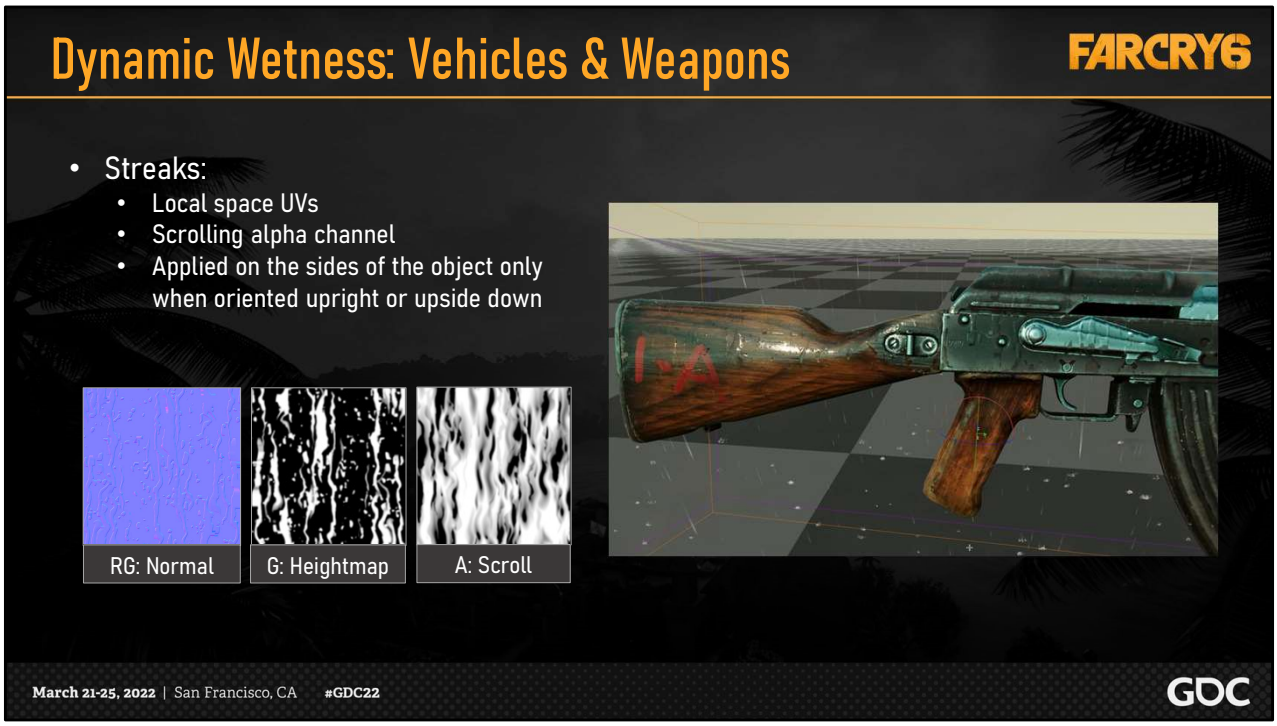


For skin, we wanted rain to be visible, but not in a way where scrolling could appear unnatural or distracting. We went with a subtle approach by using a single texture, which contained a droplets normal map and a wetness mask. With these effects applied, skin material wetness was quickly finalized.



On our project, weapons and vehicles are closely related. They are both gameplay elements and are seen up close from the first-person perspective. For this reason, we gave these assets animated rain effects, which were applied using animated textures updated per frame.

These effects were divided into two parts, streaks and droplets, which were applied on vertical and horizontal surfaces respectively.



For the streaks, our input data was only one texture, which packed in a streak normal map, a heightmap, and a scroll mask.
The scroll mask drove the vertical movement of the streaks, which we applied with local space UVs.
We made sure to only apply the streaks when the asset is oriented upright or upside down; in the latter case we reversed the scroll direction.

For the droplets, we supplied another texture, containing a normal map, a heightmap and an ID map. To drive the animation, we sampled the droplets two times to vary the pattern of droplets fading in and out with varying timing cycles, made possible with the ID map.

One thing we were still missing was the tiny static droplets that tend to build up over time on hard surfaces, so we managed to squeeze them into the heightmap texture later on.

We also made sure to use manual mips on the droplets texture to reduce sparkling artifacts.

Dynamic Wetness: Vehicle Interiors

FARCRY6

Problem #1:

Vehicle interiors are getting wet!

Caused by shared materials between interior and exterior

Solution:

Vertex paint interior mask

- Disable when convertible cars remove their tops

Interior Mask Debug

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

GDC

As we all know, adding a feature often results in unexpected issues and edge cases popping up. One issue we faced was that wetness effects were showing up inside vehicles.
This was because our vehicles sometimes share materials between the exterior and interior to save on drawcalls.

The fix for this was simply to use the red vertex paint channel as an interior mask.

Once this was painted by artists, we were able to remove the interior rain effects, except of course in edge cases such as when our convertible cars put down their tops.

Another problem came up when moving windshield wipers were implemented by our gameplay team, while rain streaks were added by our 3D team, resulting in no interaction between them!

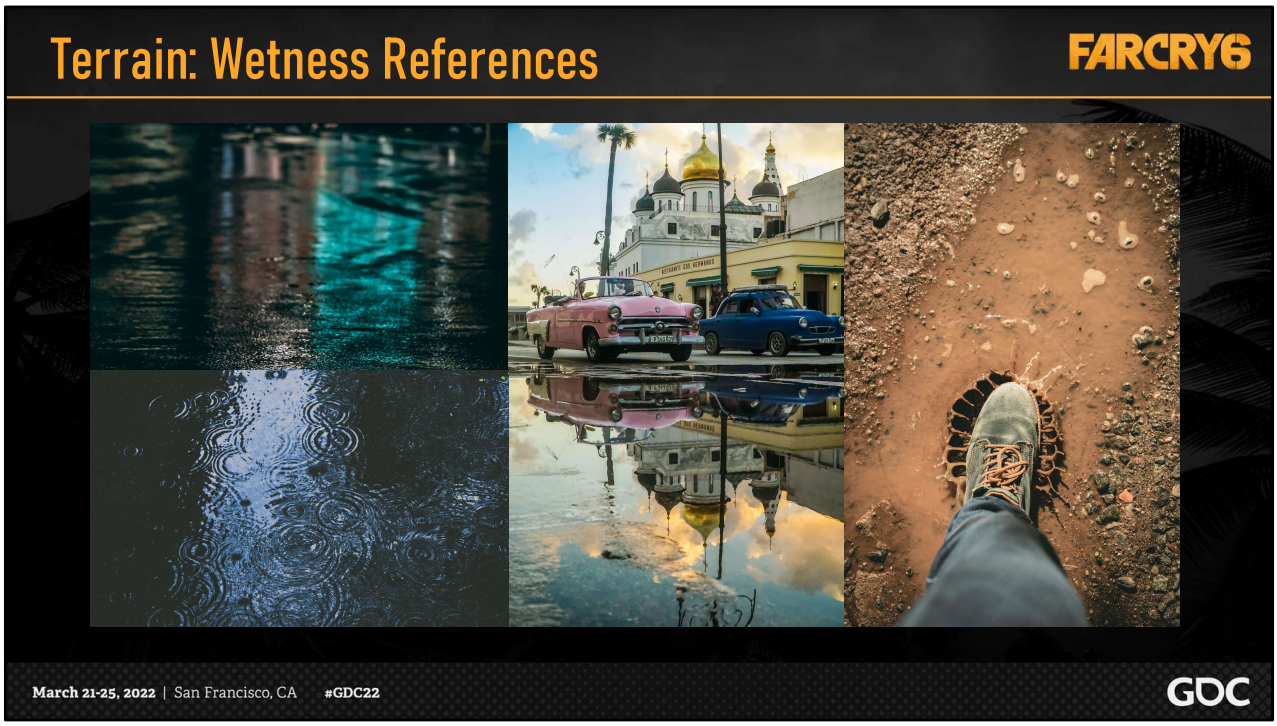
What we did was pack in a windshield gradient map and set up the shader to mask the rain according to the current gradient value. We then passed this value over to gameplay to hook everything up.

This was a small feature that we never initially planned, but inconsistencies like this can take players out of an experience very quickly, so we were happy to add it.

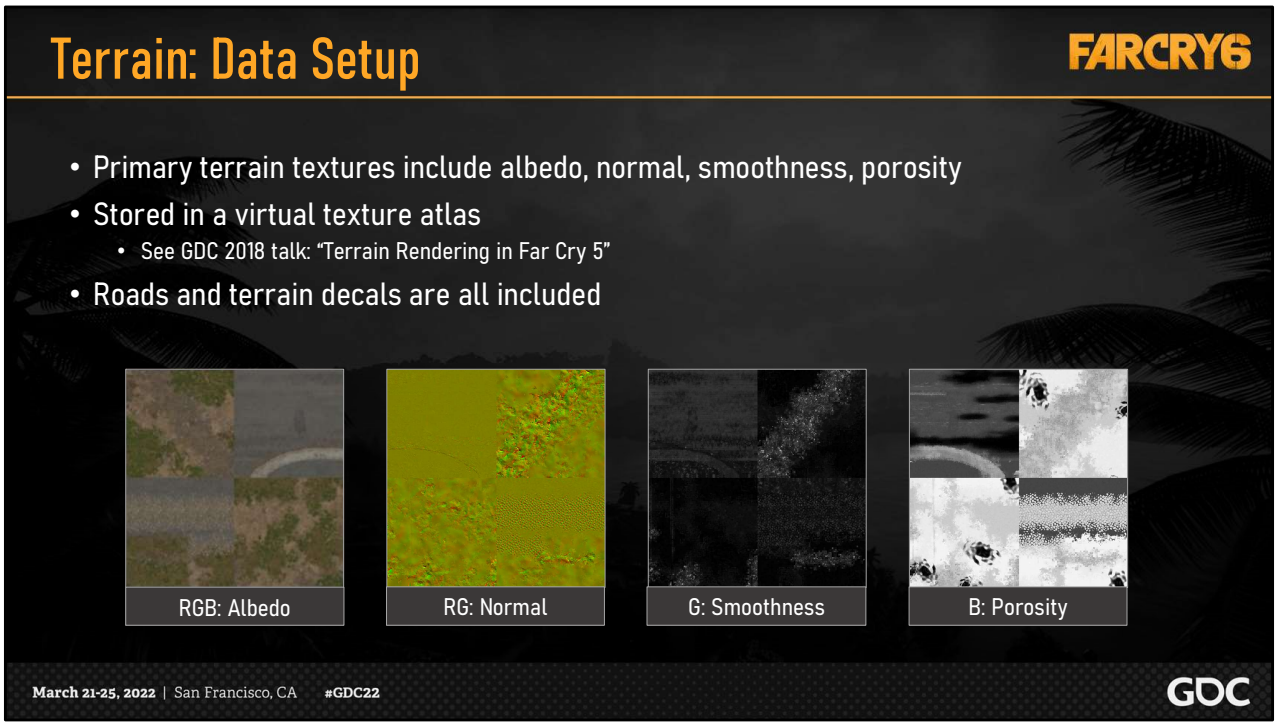
For vegetation, we originally used the static wetness approach but this led to two issues:
- Firstly, trunks and plants near the jungle floor were getting shadowed from the rain, which is strange to see in nature.
- Secondly, we were getting bugs explaining that the trees were appearing metallic. The was because the flat leaf cards were giving off a uniform reflection of the grey sky. This visual was so strong that it effectively reduced the realism of the trees by exposing where the cards were.

To fix this, we converted vegetation over to the dynamic wetness type, just so that we could tailor the wetness in the shaders independently. Of course, this meant that there would no longer be any shadowing, but 99% of the time, our trees are not hand-placed indoors, so we went through with the swap.

We used a water droplets texture on the leaves, which finally matched reality a lot more and fixed the reflection bugs.



Terrain was the final area to conquer. Not only is it always in the player's view, but as we can see from photo reference, it also requires far more detail.



Let's first summarize very quickly what we need for our terrain. The primary textures used include albedo, normal, smoothness and, the new texture added for wetness, porosity.

These properties are stored in a virtual texture atlas that we use for the entire world. To see more about this system, you can refer to our past talk on our terrain system.

Note that roads and terrain decals are all baked into the virtual texture atlas, so it truly is one complete system that we need to apply wetness to.

Terrain: Wetness Transition

FARCRY6

- ApplyWetness() again
- We wanted a better transition than just a linear fade
- Splatter transition is a height field simulation [Eggers10]
- Resulting animated mask is used as the wetness factor
- Temporarily increase porosity for darker droplets

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

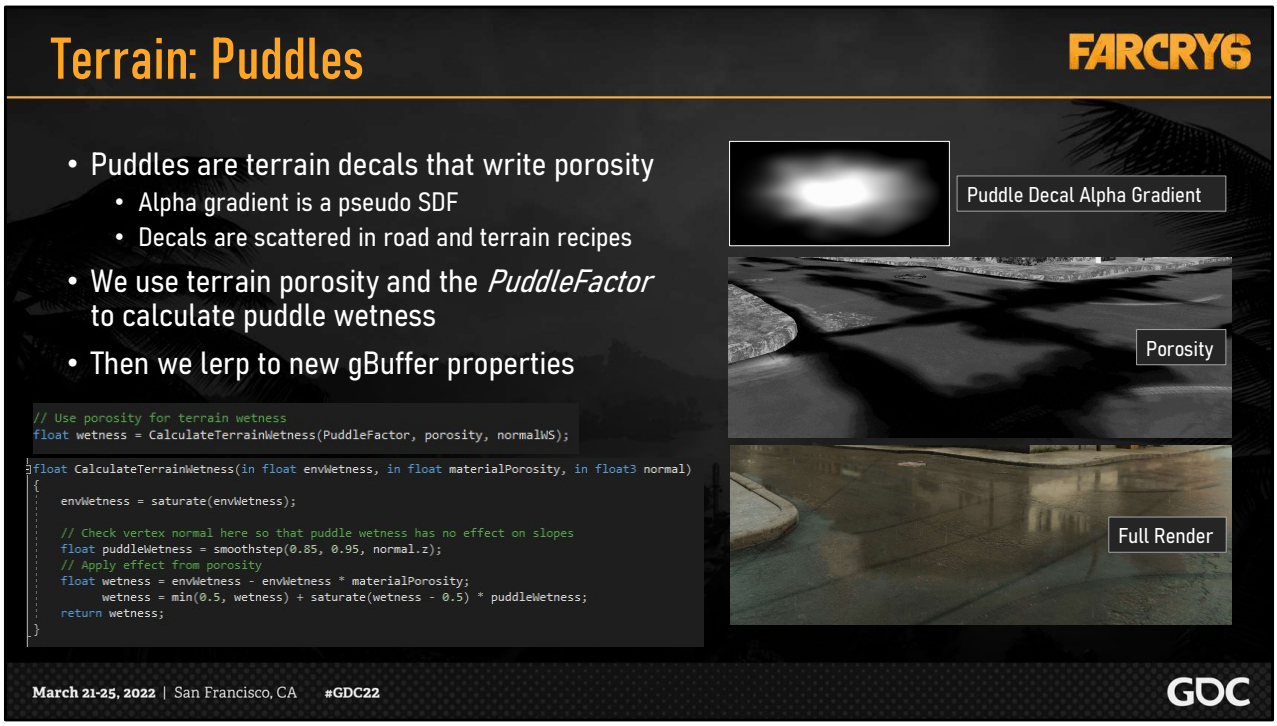
GDC

To make terrain wet, we actually use the same ApplyWetness function shown for our static objects.

The key difference for terrain is actually the transition from dry to wet. We didn't want a linear fade, as we thought this was a bit too artificial for such a large surface area.
The new transition we created was our raindrop splatter effect; as shown in the video.

To do this, a height field simulation was done on the GPU, which took in some parameters such as drying and spreading speed for the splatters.

The resulting animated mask was used as our custom wetness factor.
We also chose to temporarily increase the terrain porosity to accentuate the effect with darker splatters.



Now that the terrain has it's wetness applied, we must turn our attention to one of the largest visual features needed to present wetness: puddles.

Puddles were also a heavily desired feature for art direction because they would provide reflections, which ultimately makes the environment and lighting more appealing.

The setup for puddles was very simple. A puddle itself was a terrain decal, but all it contained was an alpha gradient and a checkbox to only write to terrain porosity.
The gradient served as a pseudo signed distance field (SDF), allowing puddles to build up from the center.
We only had a few puddle decals, which were scattered by our existing terrain and road recipe systems, similar to how cracks and potholes would be scattered.

We calculated the puddle wetness based on the terrain porosity and the PuddleFactor provided by the weather manager.

Once we had the puddle wetness, we once again altered the terrain's gBuffer values accordingly.
We slightly blended the albedo to a muddy puddle color and blended the smoothness to 1.0. For the normals however, we initially used a flat normal, but that brings us to our next implementation…

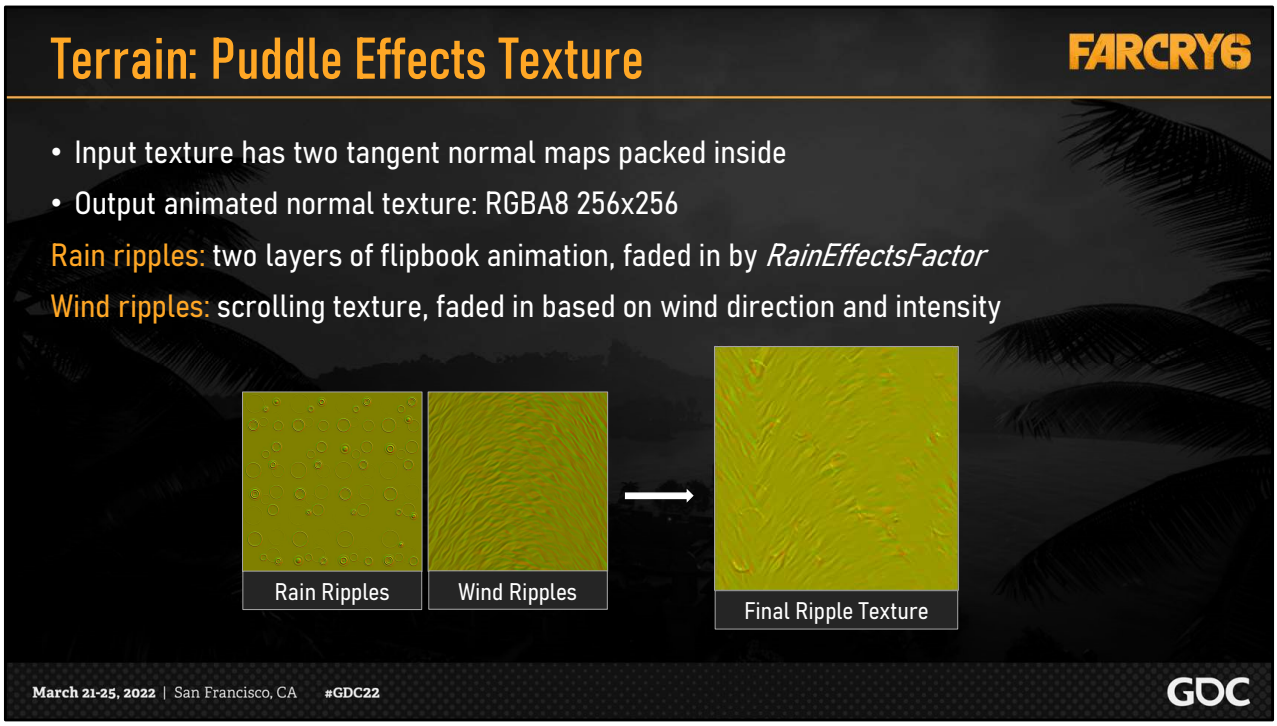
Terrain: Puddle Effects

FARCRY6

Two types of ripples

- Rain: circular ripples
- Wind: waves

- Adds realism and movement
- We generated a tiling animated texture that is used across all puddles



Our input data was a single texture with two normal maps packed inside, one for the rain ripples, and one for the wind ripples.

The output was a single animated normal texture.

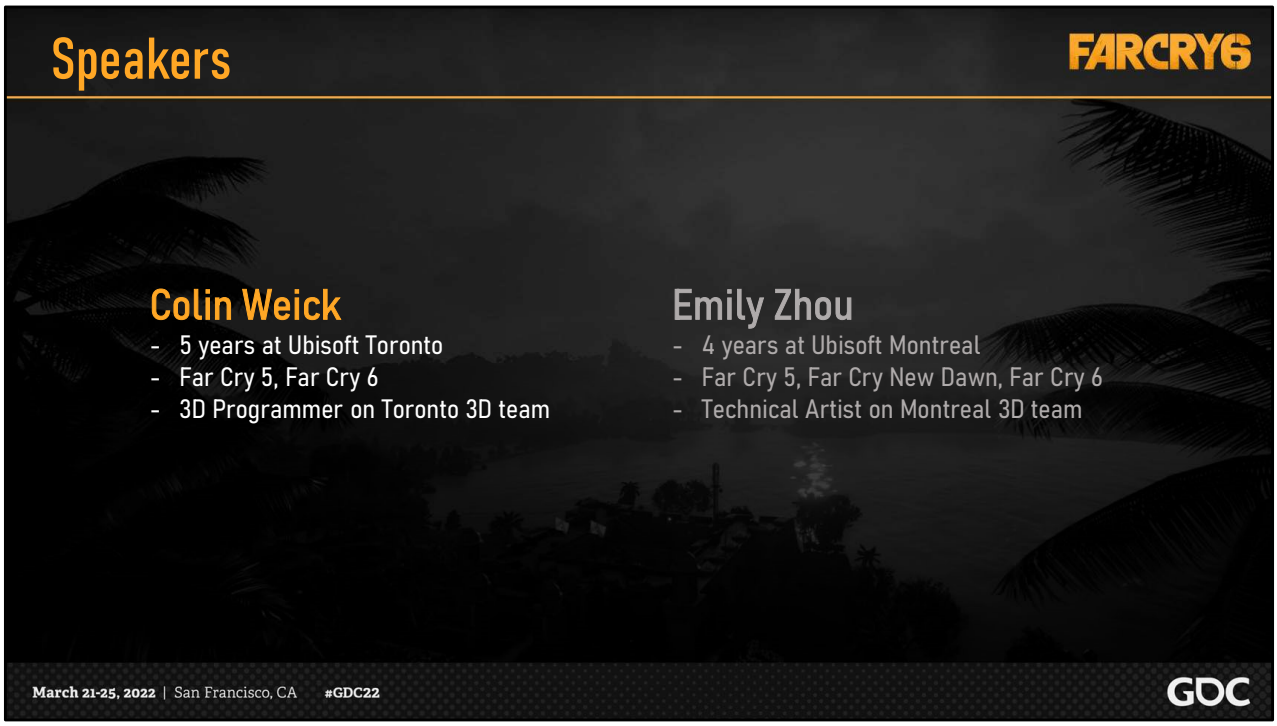
For the rain ripples, we sample the normal texture twice to have two overlapping layers of flipbook animations. These are faded in based on the weather manager's RainEffectsFactor, which will ensure that these ripples only appear when it is actively raining.

For the wind ripples, we scroll the texture based on the current wind direction and intensity. As a result, this effect can be present whether it is raining or not, such as when puddles are still on the ground just after a storm.

Finally, we blend these effects together to get the result that we use for our puddle normals.

[Stop]

This concludes the collection of shader changes we needed to support material wetness. Now, Colin will take you through the technical features that allowed us to render the weather.



Hi, my name is Colin Weick

I am a 3D programmer at Ubisoft Toronto, where I've been for the past 5 years.

And I worked on Far Cry 5 and Far Cry 6.

Now that Emily has shown us the intricacies of the weather manager and the implementation of wetness, let's take a look at the rendering features we needed to realize dynamic weather.

Technical Implementation

FARCRY6

- Far Cry 6 shipped on 9 platforms
  - Xbox One, Xbox One X, Xbox Series S, Xbox Series X, PS4, PS5, PC, Stadia and Luna
- 60 FPS on Next Gen, 30 FPS on Last Gen
- 10km^2 open-world area
- Full day night cycle
- Indoor and outdoor environments
- A new urban city
- Now with dynamic weather!

Image from Base PS4

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

GDC

But first let's take a look at the technical constraints we had.

Our game shipped on 9 platforms, spanning multiple console generations as well as PC.

We were aiming for 4k 60fps on next gen and 30 fps on the previous gen.

This was also our largest open world for the IP to date.

As Emily mentioned, we have a full day night cycle and an open-world which features indoor and outdoor environments

And now we need to do all of this with ***dynamic weather*** .

[Next]

Let’s look at a sample scene to see how these rendering features come together to complete the depiction of our weather states.

First, we have the atmospheric scattering.

[Next]

Then we have the clouds on the horizon and overhead.

[Next]

Then we have volumetric fog obscuring regions and producing light shafts escaping from clouds

[Next]

Next we have cubemaps and reflections. Notice the power lines reflected on the road.

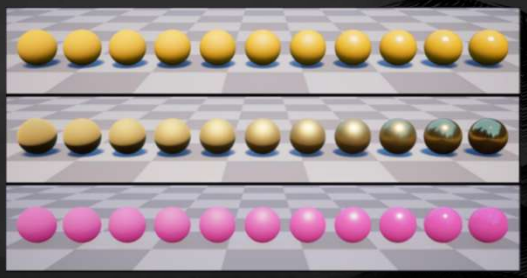
[Next]

And finally we have the rain and lightning which complete the scene of a intense thunderstorm.

Lighting

FARCRY6

- Physically based, energy conserving
- Multi-scattering diffuse and specular BRDFs
  - BRDFs are stored in 3D texture lookup tables (LUTs)
  - Area Lights with Linearly Transformed Cosine LUTs
  - Fallback on low-end to GGX specular and Lambertian diffuse BRDFs
- Translucency – vegetation, canopies, curtains
  - Wraps diffuse lighting for subsurface scattering
  - Adds a second diffuse lobe
- Reference: [McAuley19]



Surface Type	Diffuse BRDF	Specular BRDF
Skin	Pre-Integrated Subsurface Scattering(Lambert)	GGX + Multiscattering Lobe
Hair	Multiscattering Diffuse	Modified Marschner + ???
Car Paint	Multiscattering Diffuse	Two GGX + Multiscattering Lobes(top and bottom layer)
Cloth	Multiscattering Diffuse	Ashikhmin Cloth
Translucent	Two wrapped Lambert Lobes(front and back)	GGX + Multiscattering Lobe
Default	Multiscattering Diffuse	GGX + Multiscattering Lobe

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

GDC

Before we look at individual techniques, let's start with a primer on our lighting model.

We use physically based formulas and aim to be energy conserving as a goal.

New for Far Cry 6, we have a higher quality multiscattering diffuse BRDF, a GGX specular BRDF with a multiscattering lobe, and support for area lights.

The chart on the right shows how we handle specific materials, but one surface type of note was translucency, which is used for vegetation.

For translucency, we wrap diffuse lighting for subsurface scattering and added a second diffuse lobe to simulate light going through the surface.

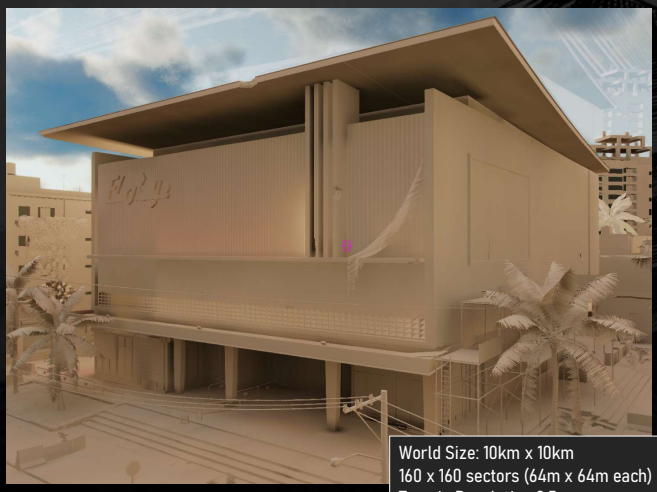
You can refer to Steve McAuley's i3D talk from 2019 to learn more about these improvements.

[Next]

Global Illumination

FARCRY6

- Global Illumination light probes
  - Placed by artists
  - Baked daily
  - Urban environments tested the limits
- GI Data: 13 packed frames
  - 11 time of day increments (sun and moon)
  - 1 key frame for local lights (night)
  - 1 key frame for sky occlusion
- Problem: doesn't include clouds or weather
- Solution: when cloud coverage is high, fade out indirect lighting to simulate cloud shadows



For global illumination we use a light probe system where probes are placed by artists throughout the world and baked daily to incorporate changes in the world.

GI data is stored in voxels. We pack 13 frames of data, 11 of which are time of day increments to give coverage to the sky, sun and moon.

There is one keyframe for local lights, which is mostly used at night.

And one key frame for sky occlusion.

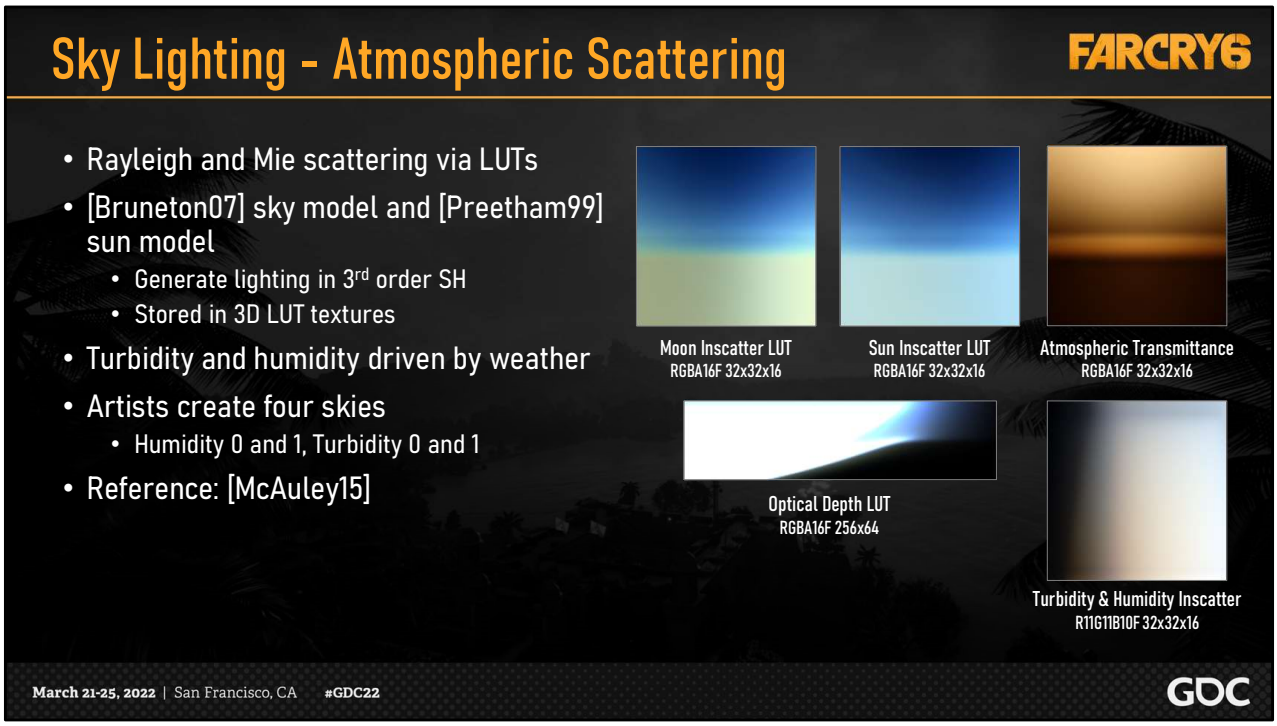
But there's a problem, these GI probes don't include clouds or any impact from the weather.

The solution we had was to fade out indirect lighting when cloud coverage is high.

This could create a problem if the local lights were in all the keyframes, but since they are isolated to one, we can avoid fading artificial lights during high cloud coverage.

Additionally, urban environments tested the limits of this system. We leaned into the sparse nature of the data and the variable probe size to increase precision near and within indoor environments.

[Next]



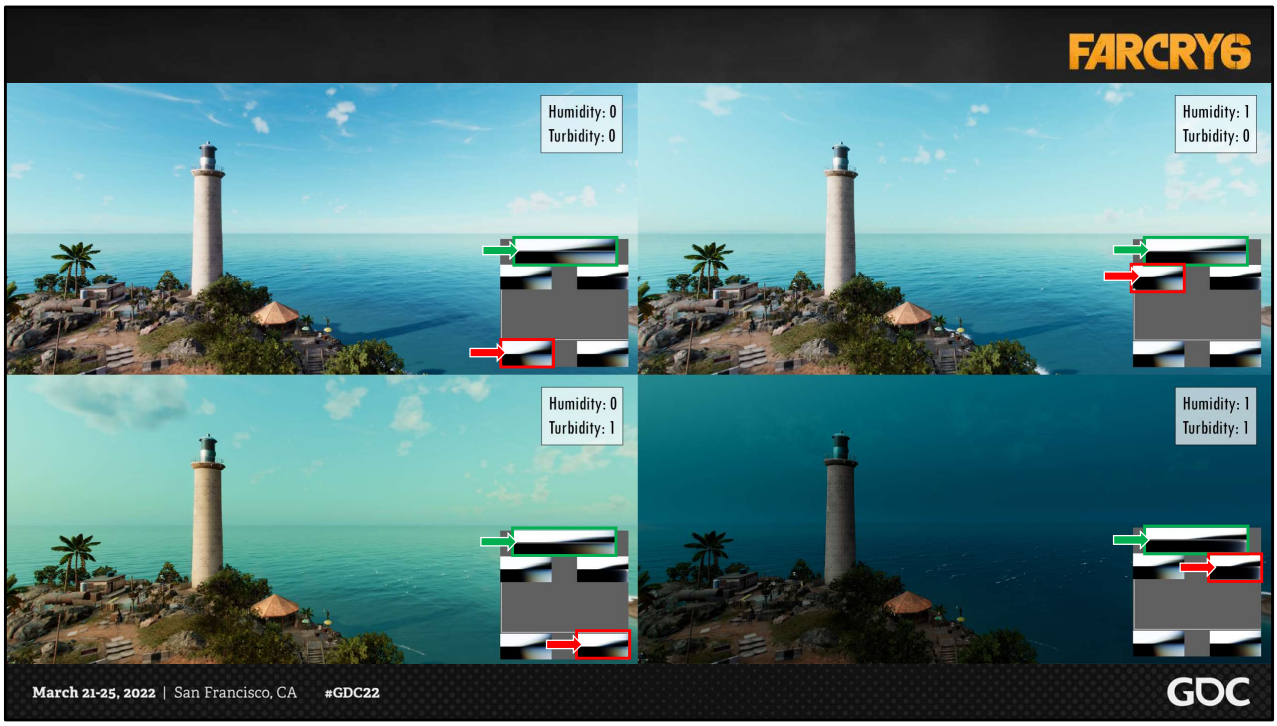
We use Brune-ton sky model as well as the Pree-tham sun model, with improvements made such as added ozone

This setup involves LUTs calculated offline, previously this was generated at run-time and that was very slow

Artists create four skies: Humidity 0 and 1, Turbidity 0 and 1 and we blend between these at run-time to get variation

You can check out our GDC talk from 2015 about Lighting in Far Cry 4 to see an earlier implementation if these concepts

[Next]



Here we show the effect of the humidity and turbidity parameters on the optical depth LUT and the blended in-scattering LUT. We blend four pairs of data together to get our result.

The green arrow points to the blended result for each frame and the red arrow points to LUTs with the greatest weight for the frames humidity and turbidity parameters.

[Next]

This is what gave us our stunning sunrises and sunsets, which we see casting beautifully on the edges of the clouds.

That leads us into our next subject, volumetric clouds

Volumetric clouds

FARCRY6

- Why real-time volumetric clouds?
  - A skybox is not sufficient
    - Poor results in motion
    - Difficult to blend weather states
  - Not just backdrop, but grounded in the world
- Weather
  - Cumulus coverage
  - Cirrus coverage
  - Horizon coverage
- Built upon prior work
  - Data setup [Schneider2015]
  - Ray marching [Schneider2015][Hillaire2016]
  - Checkerboard render[Bauer2019]

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

GDC

So why do we want real-time volumetric clouds?

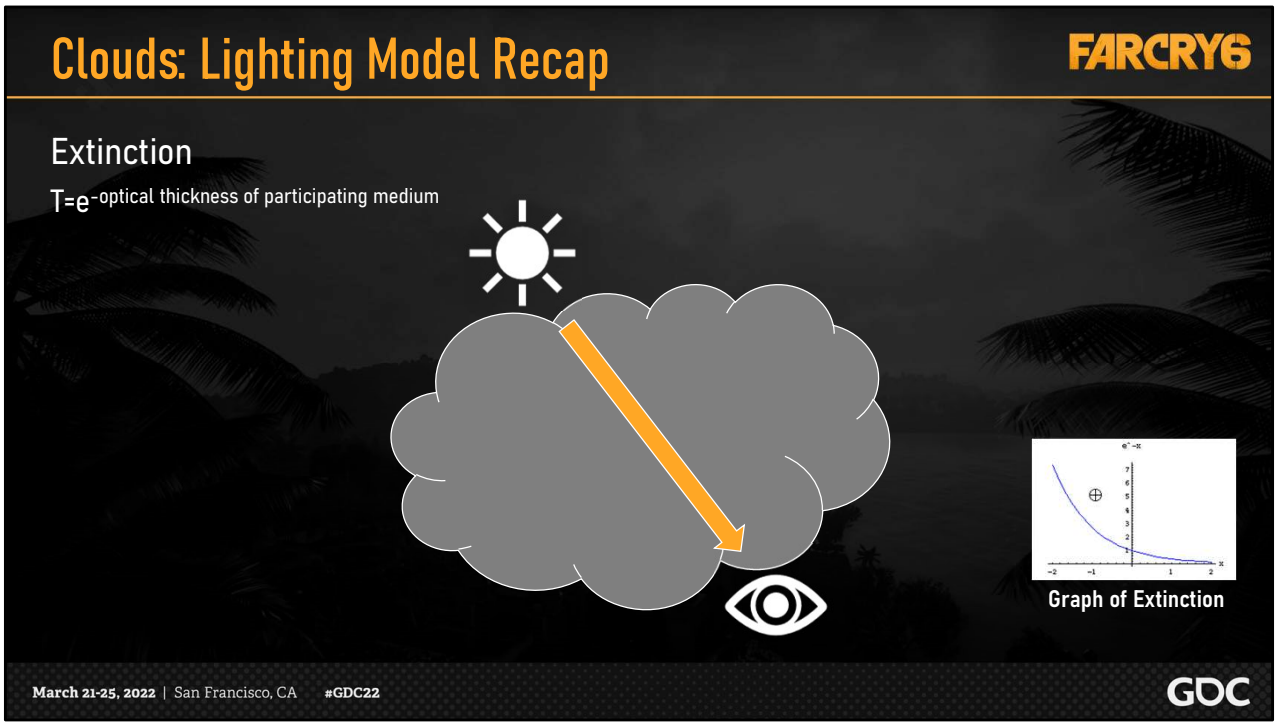
Simply put, a skybox is not sufficient. It has poor results in motion and is difficult to blend between different weather states.

A skybox is also more of a backdrop, whereas we wanted something grounded that interacts with the world.

Additionally, we needed our clouds to respond to weather with varying cloud coverage levels.

Our eventual solution was build upon prior work, as listed here.

[Next]



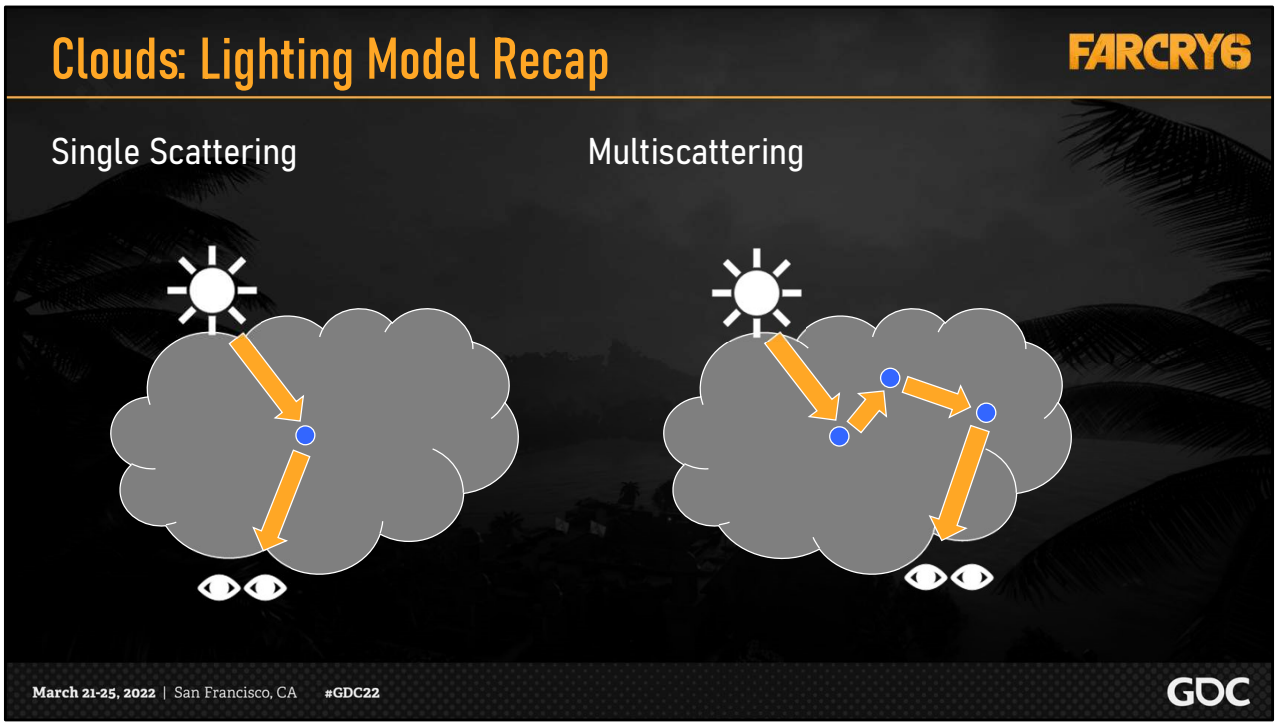
The most important part of cloud lighting is extinction. As light travels through a volume, it loses energy due to the interaction with water particles that clouds are composed of.

This extinction is determined by the Beer-lambert law, which we will henceforth refer to as transmittance.

Extinction consists of both absorption, which is actually quite low for clouds, and scattering.

The light that eventually hits the eye after being scattering is measured in terms of radiance.

[Next]



Single-scattering refers to when light enters a cloud encounters one scattering event before traveling in the direction towards the observer.

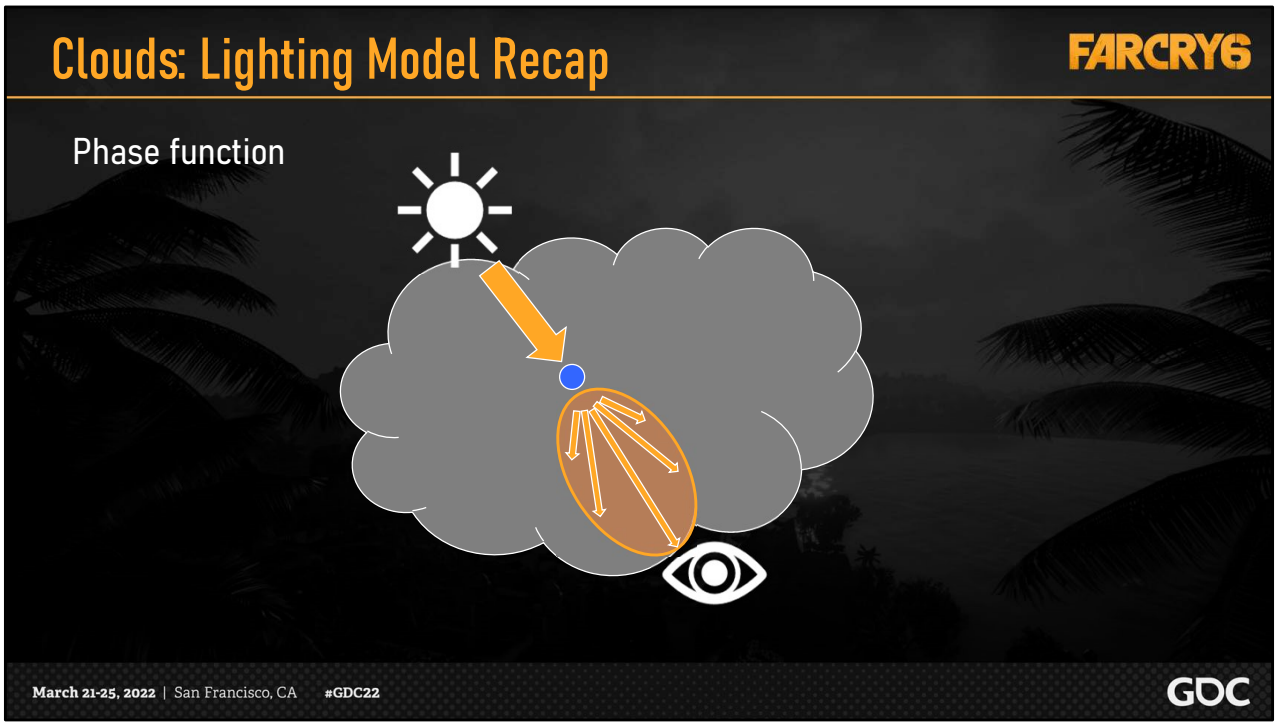
Multi-scattering refers to when light encounters a near limitless number of scattering events within the cloud before traveling in the direction towards the observer.

[Next]

Here we show the effects of the single scattering on the clouds

Next we isolate the effects of multiscattering on the clouds.

And finally we have the effect of both. Notice the greater sense of depth in the clouds.



All of these scattering events can be modeled by a phase function which projects how much light and in what direction it will travel after it is scattered.

In this diagram we have the oval shape representing the phase function approximation of light scattering after the scattering event at the blue dot.

[Next]

Clouds: Phase Function

FARCRY6

Phase Function Equations:

$$f(g,\theta) = \frac{1}{4\pi} \cdot \frac{1-g^2}{|1+g^2-2gcos\theta|^{\frac{3}{2}}}$$

$$d(g,\theta) = |1+g^2-2gcos\theta|$$

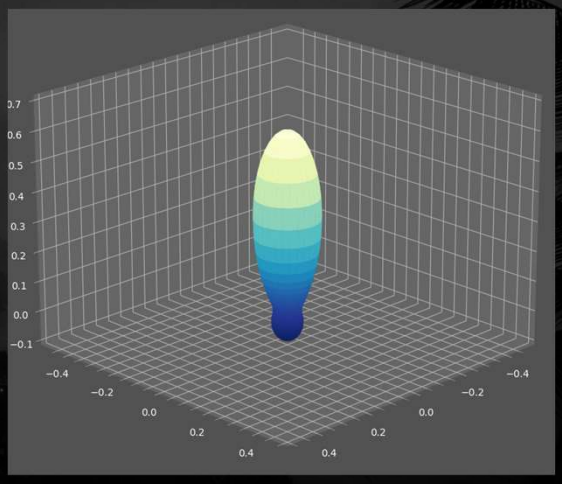
$$b(x,y,\alpha) = x(1-\alpha)+y\alpha$$

$$r = b(f(g_0,\theta), d(g_1,\theta), \alpha)$$

Constants:

$$g_0 = 0.65$$
$$g_1 = -0.30$$
$$\alpha = 0.45$$

```
float DoubleLobedHenyeyGreensteinFast(float cosPhi, float gFactor0, float gFactor1, float alpha)
{
    cosPhi = max(min(cosPhi, 0.99), -0.99);

    float gFactor02 = gFactor0 * gFactor0;
    float d0 = abs(1 + gFactor02 - 2 * gFactor0 * cosPhi);

    float gFactor12 = gFactor1 * gFactor1;
    float d1 = abs(1 + gFactor12 - 2 * gFactor1 * cosPhi);

    return OneOver4Pi * fastRcpNR0(d0 * d1) * lerp((d1 - d1 * gFactor02) * fastRcpSqrtNR0(d0), (d0 - d0 * gFactor12) * fastRcpSqrtNR0(d1), alpha);
}

float GetPhaseFunction(float cosPhi, float g0Factor, float g1Factor, float alpha)
{
    // Using a double-lobed phase function helps stop lighting being too dark opposite the sun, as we lack back scattering
    // This is more physically correct than the approach proposed by RDR2 in their SIGGRAPH 2019 talk, which just clamps the amount of backscattering:
    // http://advances.realtimerendering.com/s2019/slides_public_release.pptx
    return pi * DoubleLobedHenyeyGreensteinFast(cosPhi, g0Factor, g1Factor, alpha);
}
```



Double Lobed Henyey-Greenstein Phase Function

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

GDC

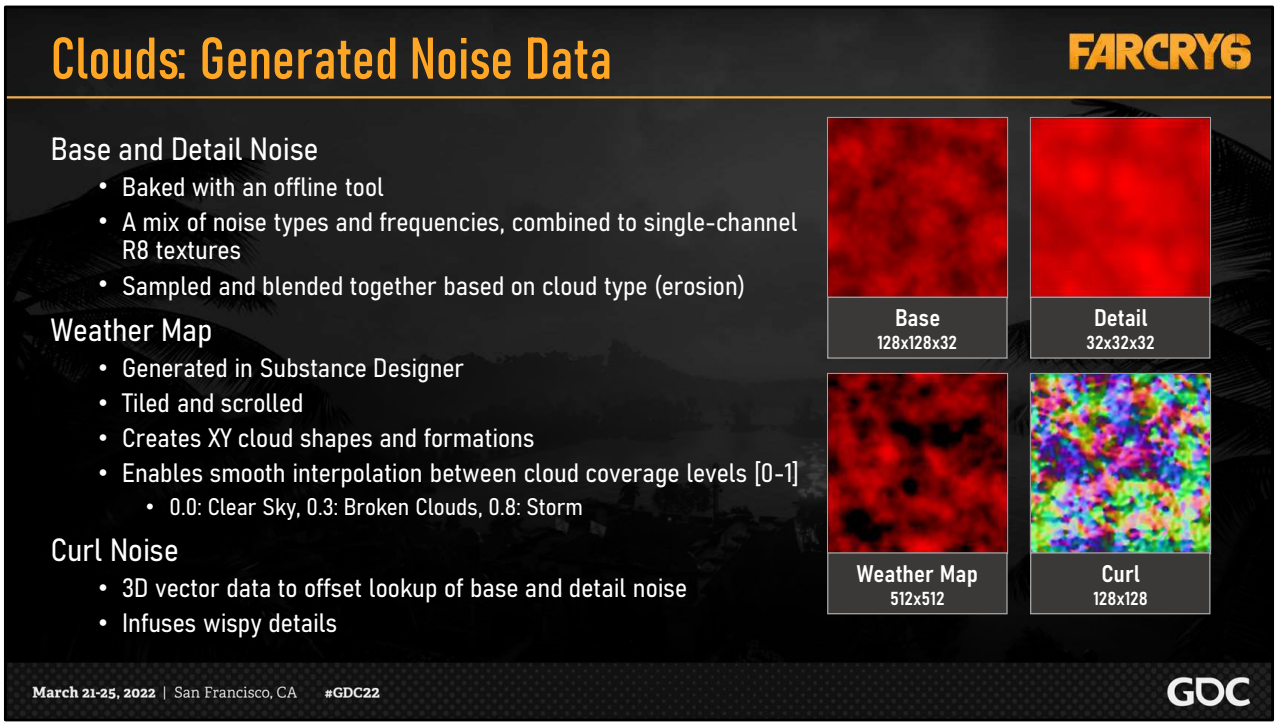
On the right is a plot of the Double-lobed Henyey-Greenstein Phase Function we used in our implementation.

The real phase function for clouds would be much more complex and far too expensive to calculate in real-time.

You can also reference this and other lighting code in the bonus slides.

[Next]

Graph of Fog Phase Function https://www.desmos.com/calculator/dcgvobzm4n



Let's talk about how we authored cloud data, which represents the density of particles in the clouds to be used for our scattering techniques.

We created base and detail textures, containing mixes of noise types and frequencies, which were collapsed into a single channel.

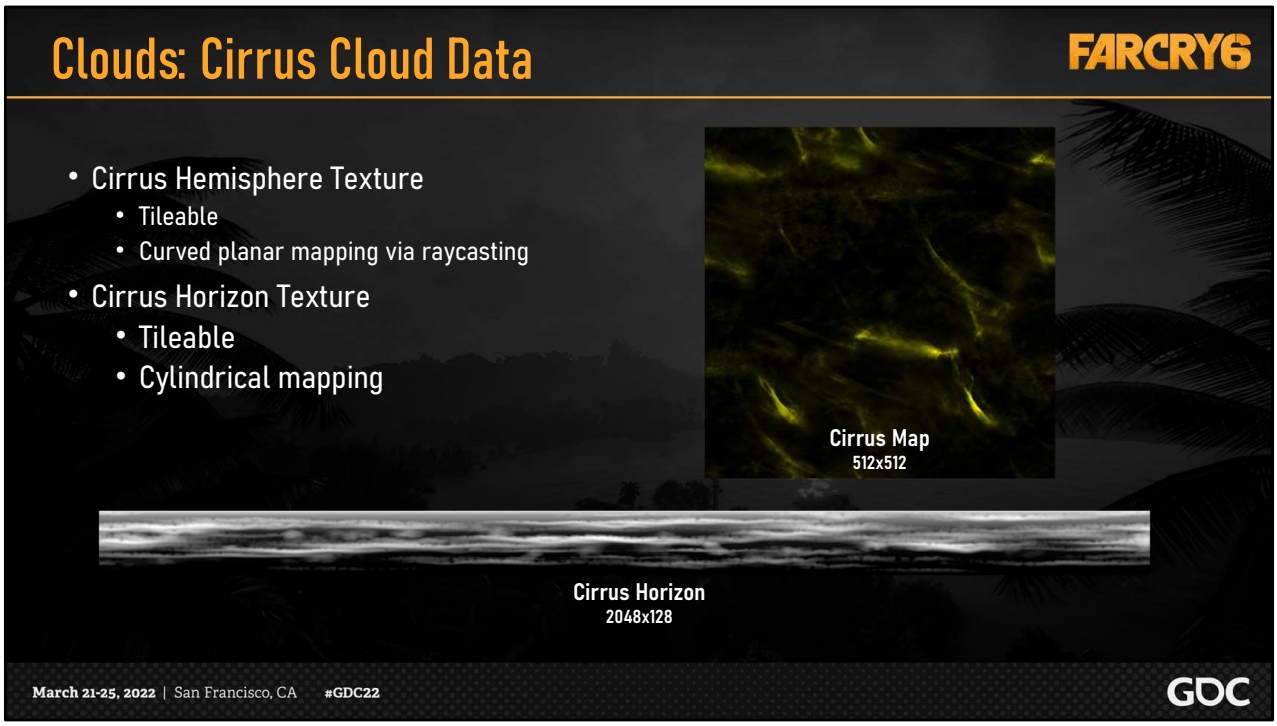
These textures are then volumetrically sampled and blended together to create clouds shapes.

Next we generated a weather map to be tiled in the world and scrolled in the wind direction from our weather manager.

This builds our XY cloud formations and enables us to smoothly interpolate between levels of clouds coverage.

We also have a curl noise, which contains 3d vector data which we use as an origin-offset when shooting rays into the base and detail noise.

[Next]

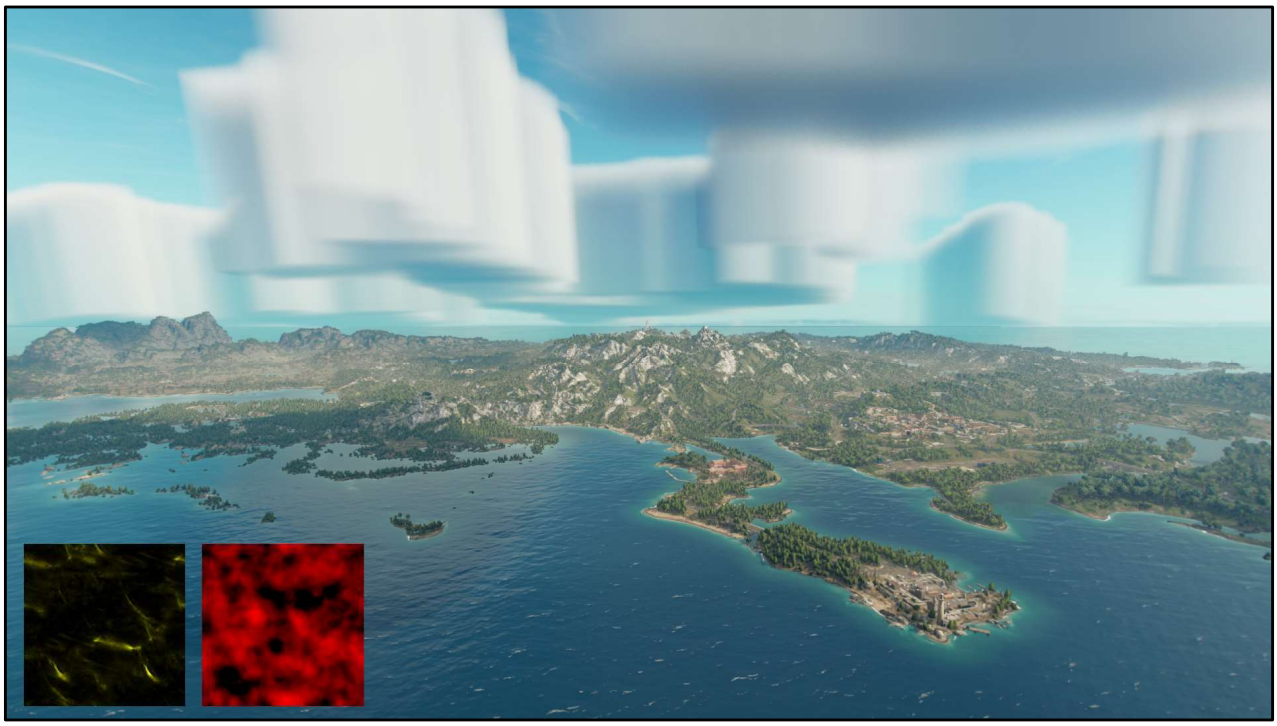


Finally we have the cirrus cloud texture which we map hemispherically to the sky and cirrus horizon texture which we map in a cylinder around the camera.

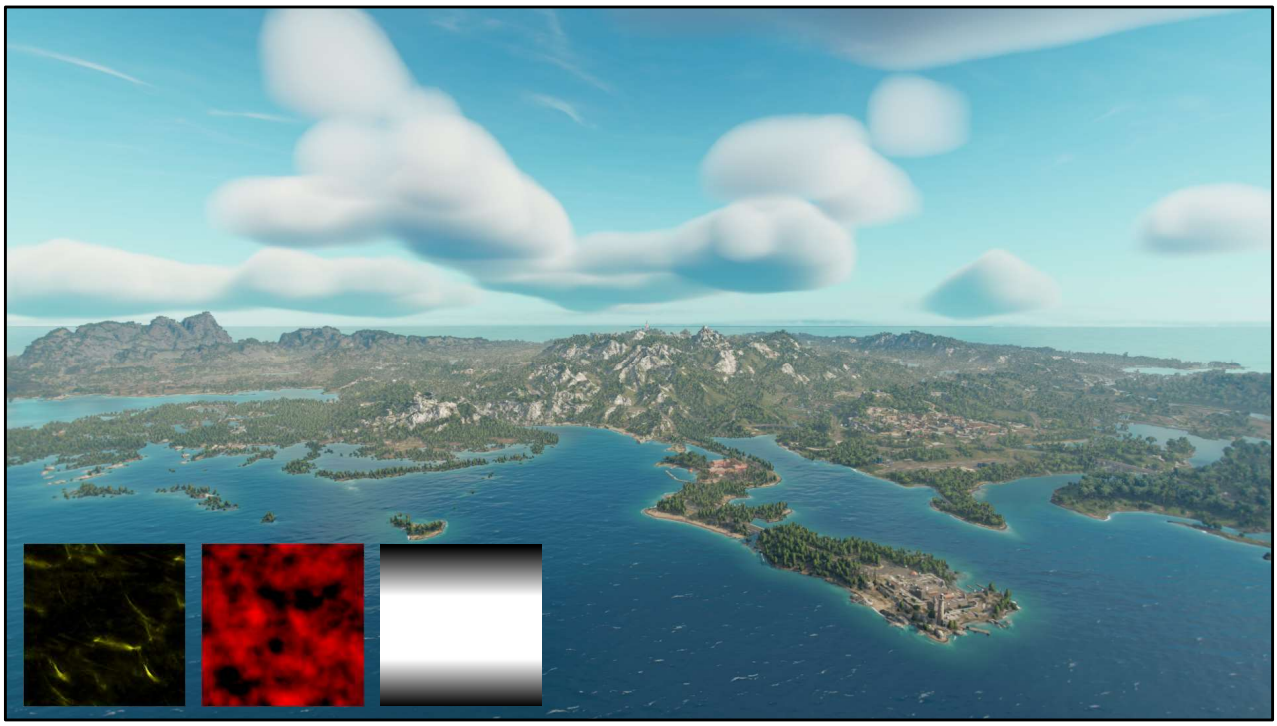
[Next]

Let’s put these all together. Here we start with a clear sky and just our atmospheric scattering.

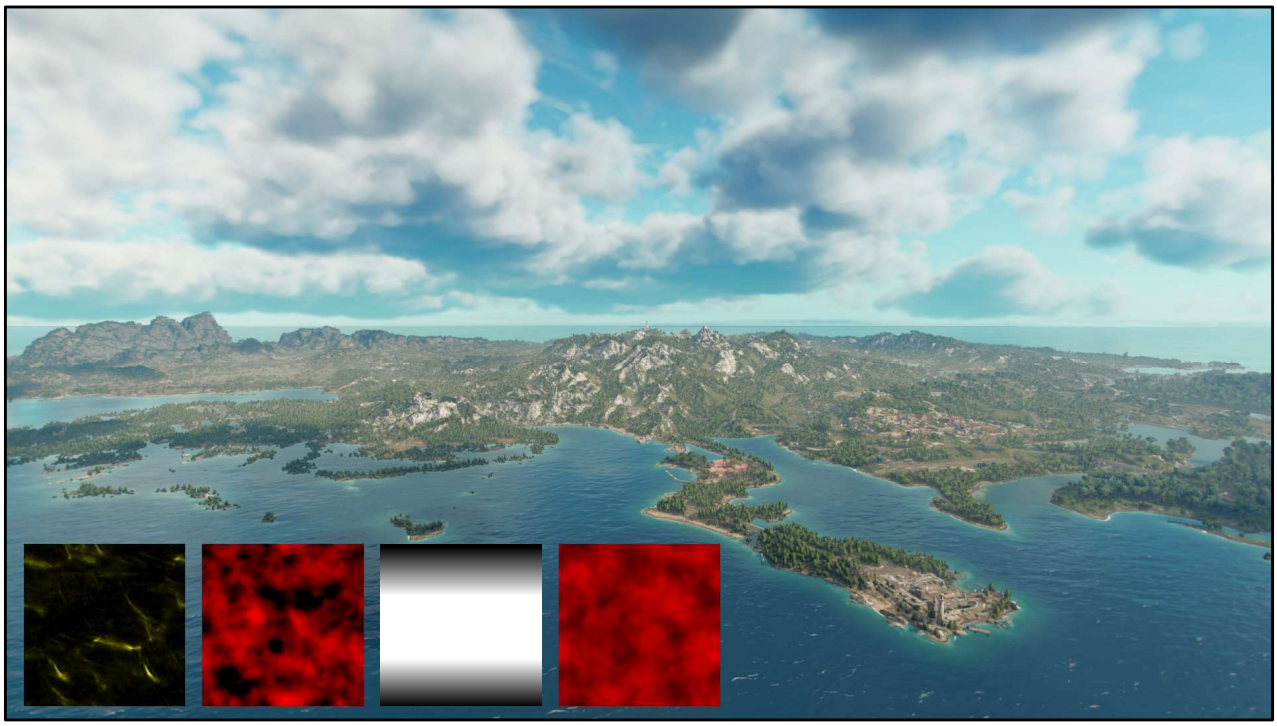
Then we add the cirrus hemisphere and horizon clouds.



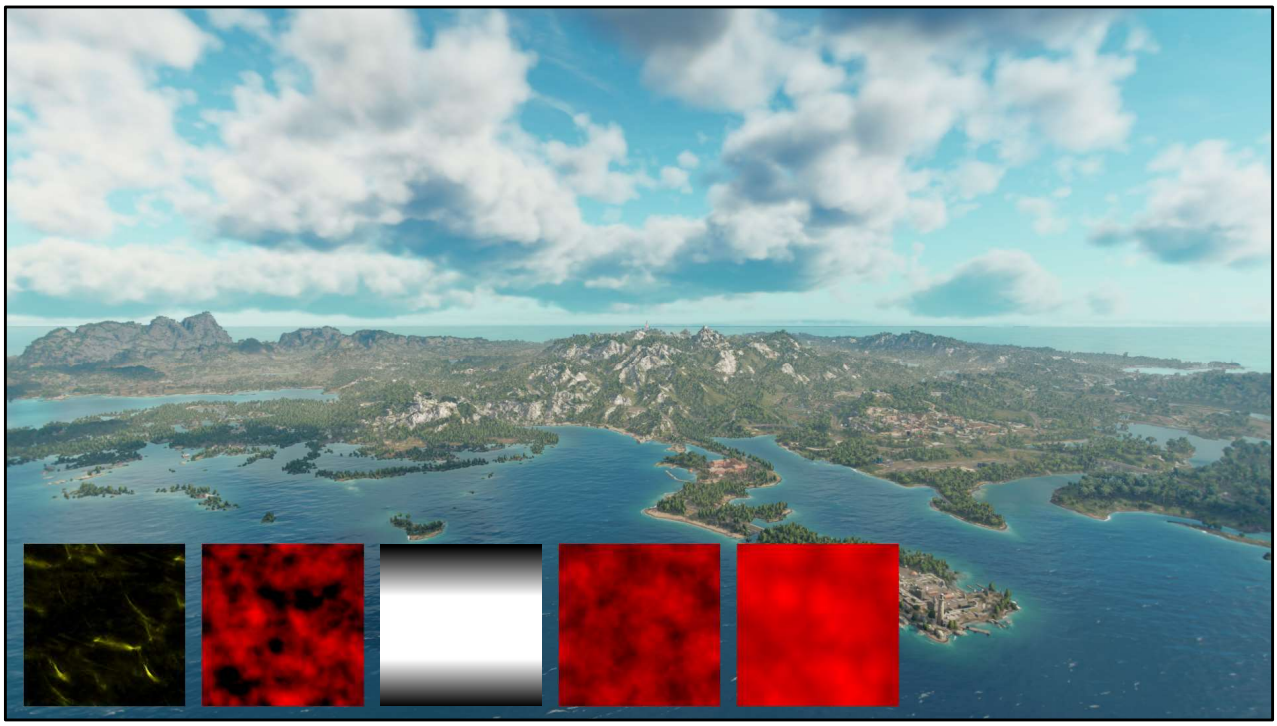
Then we add the weather map and you start to see the xy shape of the cloud formations



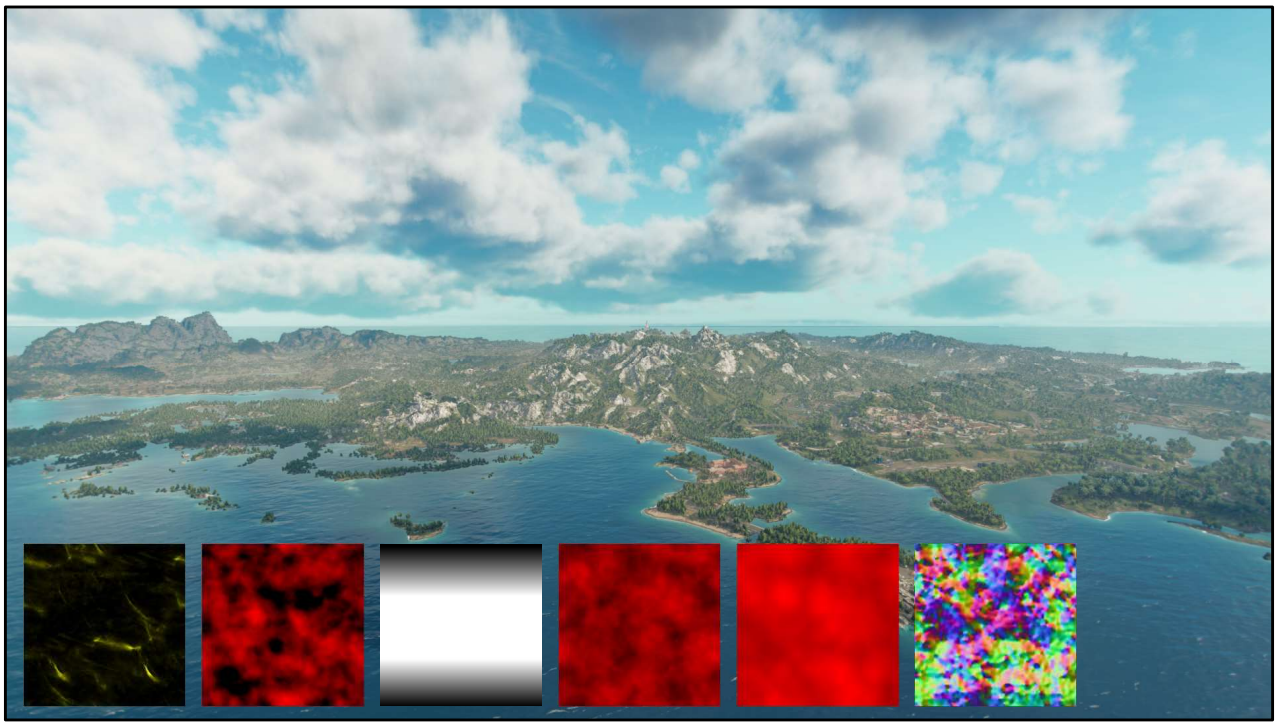
Then we combine in the gradient texture to define the shape of the cloud. This cumulus gradient ended up being the only gradient we needed to ship, although we did experiment with others.



Then we use the base detail noise volume texture to get the distinctive cumulus cloud shapes.

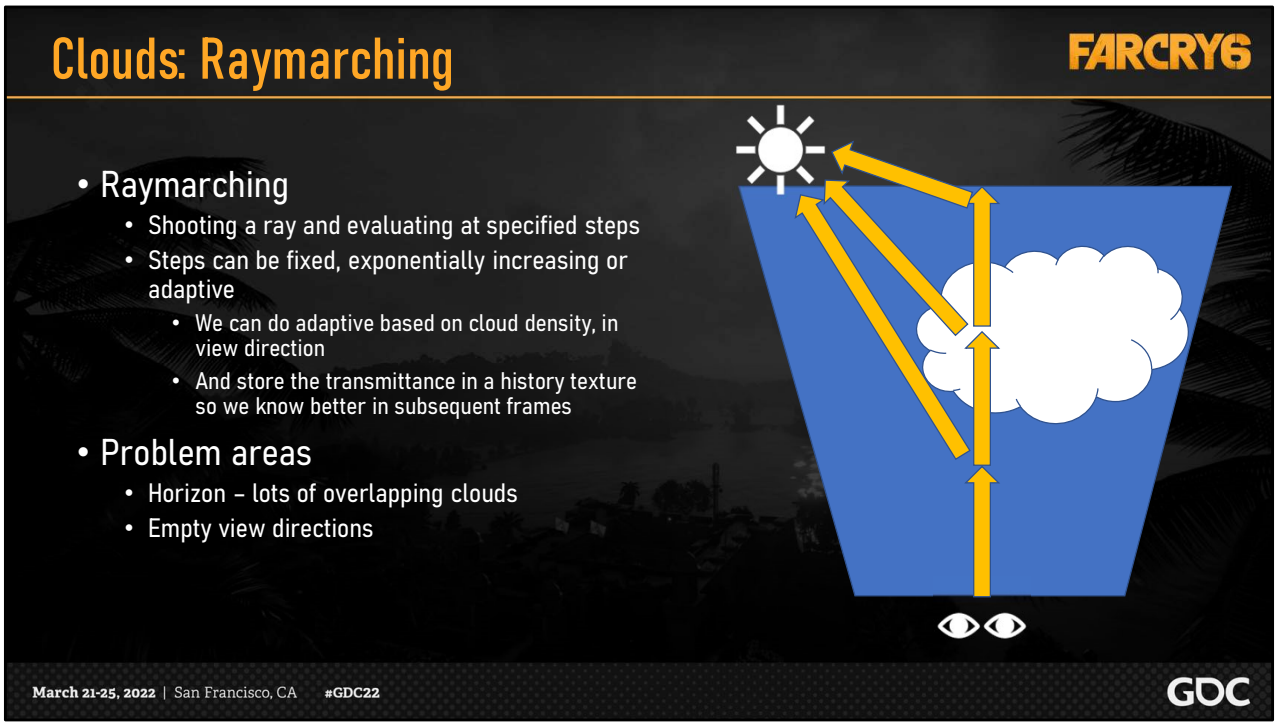


Then we use the detail noise volume texture to erode more detail.



Then we add in the curl noise texture to get wispy details.

[Next]



Raymarching is the process we used to draw clouds efficiently, it involves shooting a ray from the observer and evaluating at specified steps.

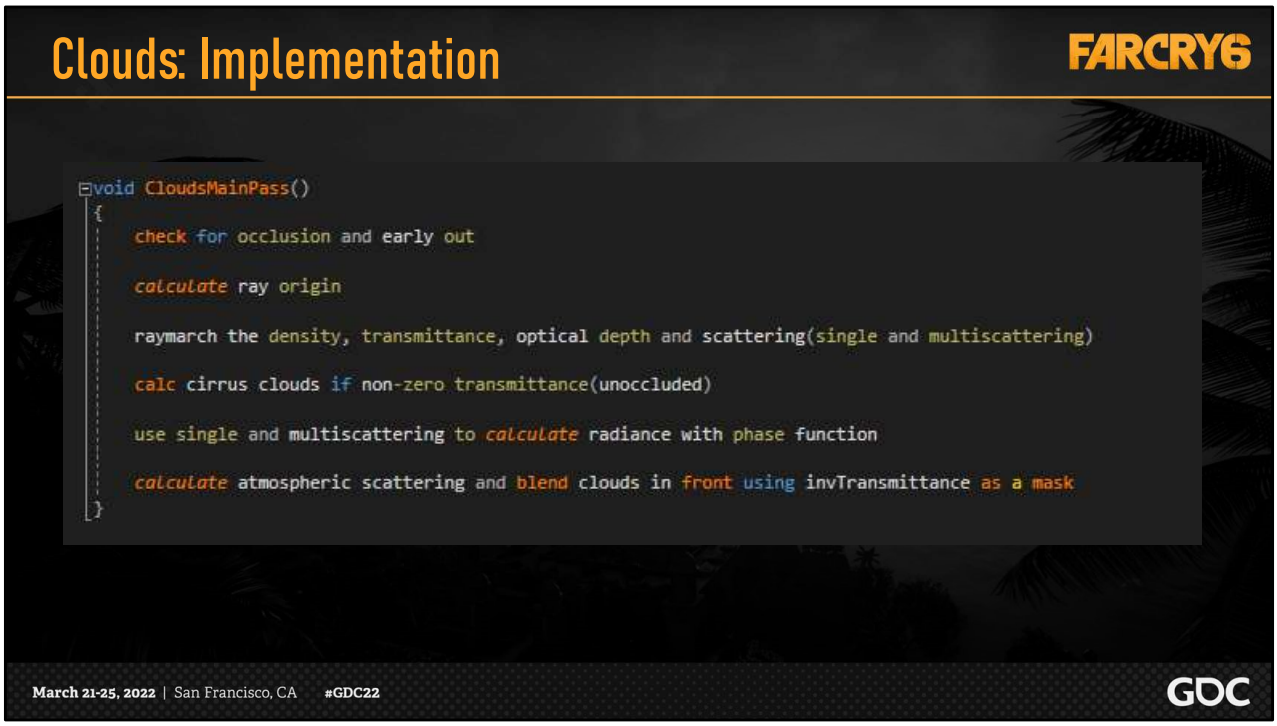
At each step we integrate the density from that point back to the sun(or moon).

We then convert the optical depth that is returned to transmittance which can then be used to approximate single and multiscattering.

We then integrate this transmittance along the segment and accumulate the single and multiscattering for the ray-march total.

We also apply the extinction to the transmittance so that affects later ray steps.

[Next]



The general process for each pixel is therefore as follows:

Calculate the origin from the pixel and ray-march the density, transmittance, optical depth and scattering(single and multiscattering)

Calculate the cirrus clouds if there is a non-zero transmittance

Use the single and multiscattering values to calculate the radiance with the phase function

And then calculate atmospheric scattering and blend the clouds in front using inverse transmittance as a mask.

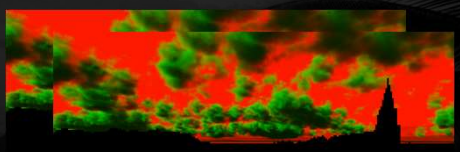
[Next]

Clouds: Runtime Textures

FARCRY6

- Screen space textures for current frame
  - Half resolution, 1920x1080 → 960x540
  - Radiance texture
  - Transmittance texture
    - Red channel: transmittance
    - Green channel: sun scattering used for lightning
- History Textures
  - Radiance & Transmittance
  - Temporally reprojected
    - Camera delta

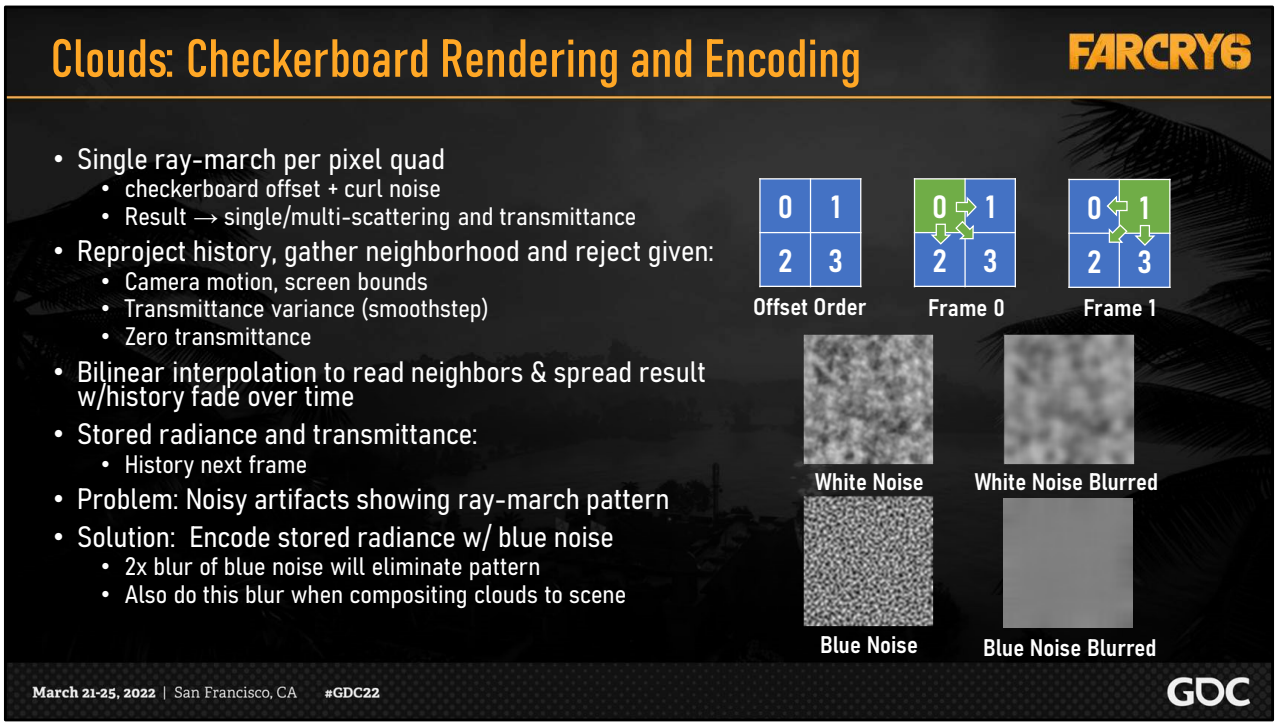
Radiance Textures
960x540 R11G11B10F



Transmittance Textures
960x540 R8G8

It would be far too expensive to ray march at the full screen resolution so we do so at half of the screen resolution.

We have a radiance and a transmittance texture, as well as history radiance and transmittance textures, which we temporally reproject from the last frame.



We fill these runtime textures using a checkerboard rendering process.

We do a single ray-march for each pixel quad, using a checkboard offset combined with curl noise offset to get the ray origin.

The ray-march result is single scattering, multi-scattering and transmittance.

We reproject the history into the current frame, gather the neighborhood and reject based the heuristics listed.

We then use bilinear interpolation of the neighbors, and spread the result of the ray-march over this area, fading the history over time.

The result is stored as radiance and transmittance history that will be used for the next frame.

Particularly of note is the process we use to store the radiance. We encode this with blue noise to help mask the noisy artifacts inherent to raymarching at such a lower resolution.

Notice on the right, when we do a 2x blur of the blue noise it converges to grey, while the white noise does not.

[Next]


Problem
Missing shadows on the ground!

Clouds: Projected Cloud Shadows

FARCRY6

- Tried: ray-marched shadows from gbuffer/depth
- Simple orthographic projection
- Covers up to (configurable) 5,000m in front of the camera
- Used for volumetric fog to get light shafts from clouds


```
float GetCloudShadows(Texture2D<float> projectedCloudShadowTexture,
                      float3 worldPosition,
                      float toEyeDistance,
                      float4x4 worldToUVMatrix,
                      float cloudShadowsNearScale,
                      float cloudShadowsNearOffset,
                      float cloudShadowsFarScale,
                      float cloudShadowsFarOffset,
                      float cloudShadowsFadeScale,
                      float cloudShadowsFadeOffset)
{
    float2 uv = mul(float4(worldPosition, 1.0f), worldToUVMatrix).xy;
    float cloudShadow = projectedCloudShadowTexture.SampleLevel(Border, uv, 0).r;

    float nearClouds = saturate(cloudShadow * cloudShadowsNearScale + cloudShadowsNearOffset);
    float farClouds = saturate(cloudShadow * cloudShadowsFarScale + cloudShadowsFarOffset);
    float distToPlayer = saturate(toEyeDistance * cloudShadowsFadeScale + cloudShadowsFadeOffset);
          distToPlayer = distToPlayer * distToPlayer * (3.0f - 2.0f * distToPlayer);

    return lerp(nearClouds, farClouds, distToPlayer);
}
```

```
void ComputeProjectedCloudShadow(uint3 dispatchThreadId)
{
    float3 worldPos = mul(float4(dispatchThreadId.xy,0,1), CloudShadowThreadID2WorldMatrix).xyz;
           worldPos.z -= WorldOffset;

    // Offset far away to make sure we're below the clouds; the Raymarching code will clamp to the bottom layer anyway
    float3 shadowTraceStartPos = worldPos - SunDirection.xyz * 4000;
    float projectedShadowFactor = RayMarchCloudShadow(shadowTraceStartPos, SunDirection.xyz, CloudShadowSampleCount);
    ProjectedCloudShadowTexture[dispatchThreadId.xy] = projectedShadowFactor;
}
```


We tried ray-marched shadows from the gbuffer but that was too costly.

The solution we landed upon was projected cloud shadows.

It was a simple orthographic projection – nothing fancy.

The resulting shadow texture will be used later on for volumetric fog to get light shafts from clouds.

[Next]

And here we have the clouds on the ground and on the water.

Next we have the volumetric fog which is used for environmental height fog as well as local fog from weather presets.

[Next]

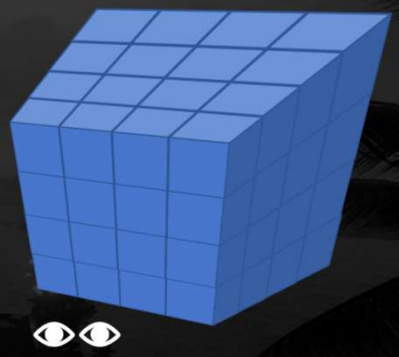
Volumetric Fog

FARCRY6

- Fog settings from weather presets

Fog Local Density	0.007
Fog Global Density	0.0003
Local Fog Height Falloff	0.15
Global Fog Height Falloff	0.2

- Frustum Aligned Volume [Wronski14][Hillaire15]
  - Resolution: 120x68x128
  - R11G11B10F Irradiance volume
  - R16F Attenuation volume
- Local and global fog based on weather
- Local fog for biome/region differences
- Light shafts are created by shadows evaluated within the cells
  - Problems!



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

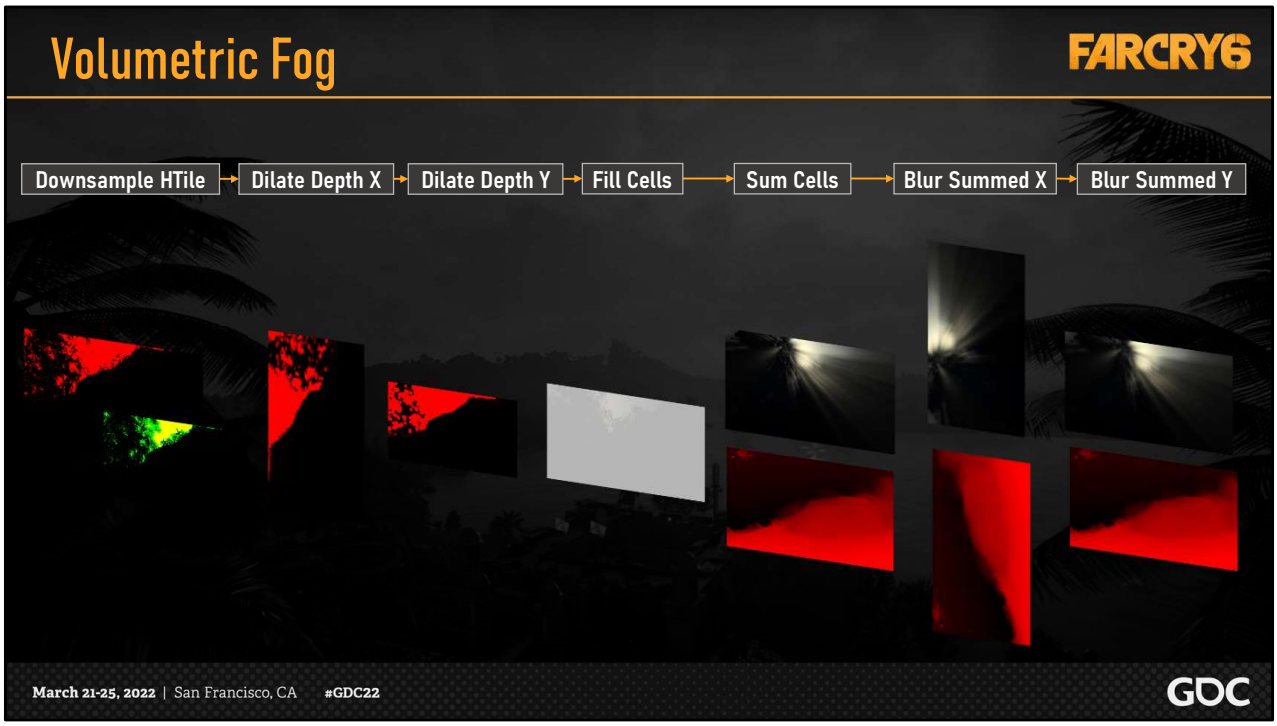
GDC

Our implementation uses a frustum aligned voxel grid presented in prior work

With the volumetric fog, light shafts are created by shadows evaluated within the cells of the volume.

Herein lies some problems!

[Next]



Here's a brief overview of the volumetric fog process.

We prepare our scene depth information in the first 3 steps shown, and cull local fog volumes to screen tiles.

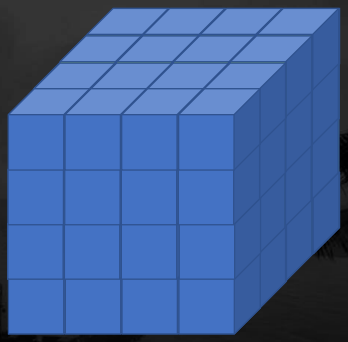
We will cover the remaining steps in the following slides.

[Next]

Volumetric Fog: Fill Cells

FARCRY6

- Temporal filtering
  - Temporal offset and fadeout
  - Only on last gen & PC low-med
  - Requires more memory for history
- Fog particles
- Lighting
  - Point and spotlights
- Phase Function: same as clouds
  - https://www.desmos.com/calculator/dcgvobzm4n



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

GDC

We fill the cells with irradiance. We do this using the local and global densities of fog and fog particles as well as lighting from the sun, atmosphere, indirect lighting and point and spot lights.

Note that we use a temporal filtering on last gen instead of the upcoming bilateral blur.

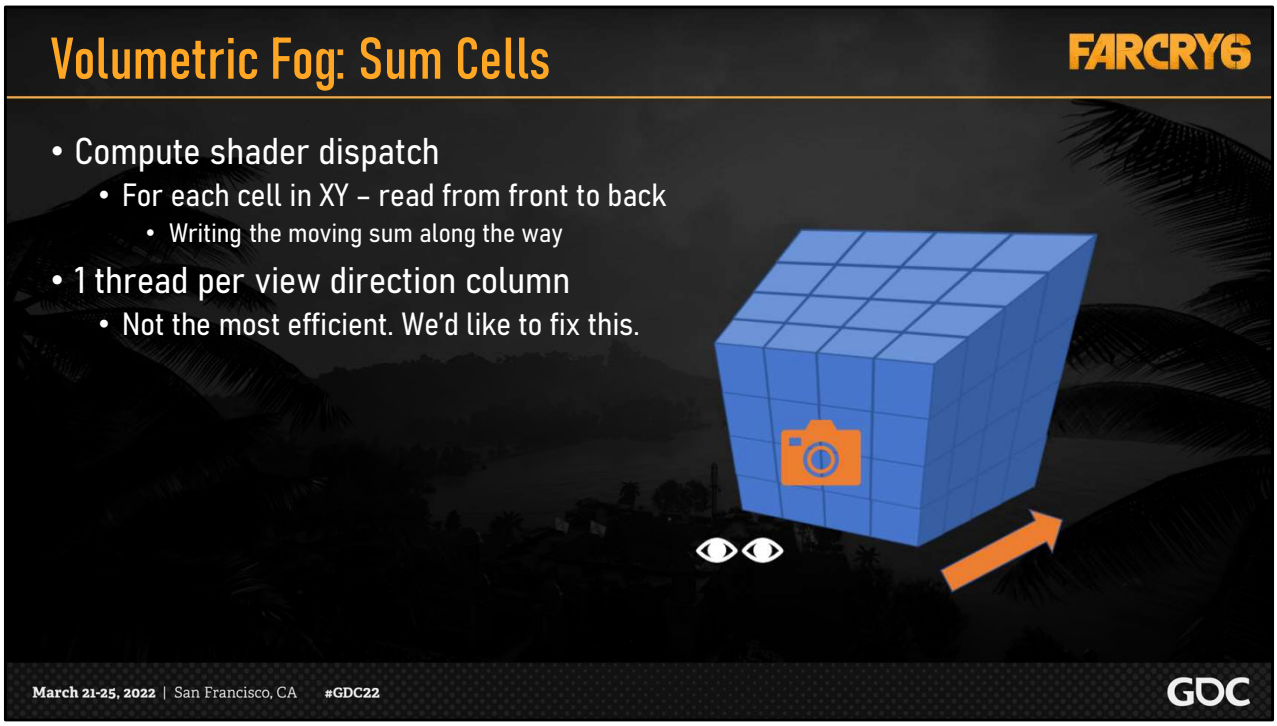
For this we require a history buffer that is reprojected into the current frame and faded out over time.

This requires more memory and unfortunately has some ghosting artifacts, but it is a cheaper solution.

We used the same phase function as with the volumetric clouds.

[Next]

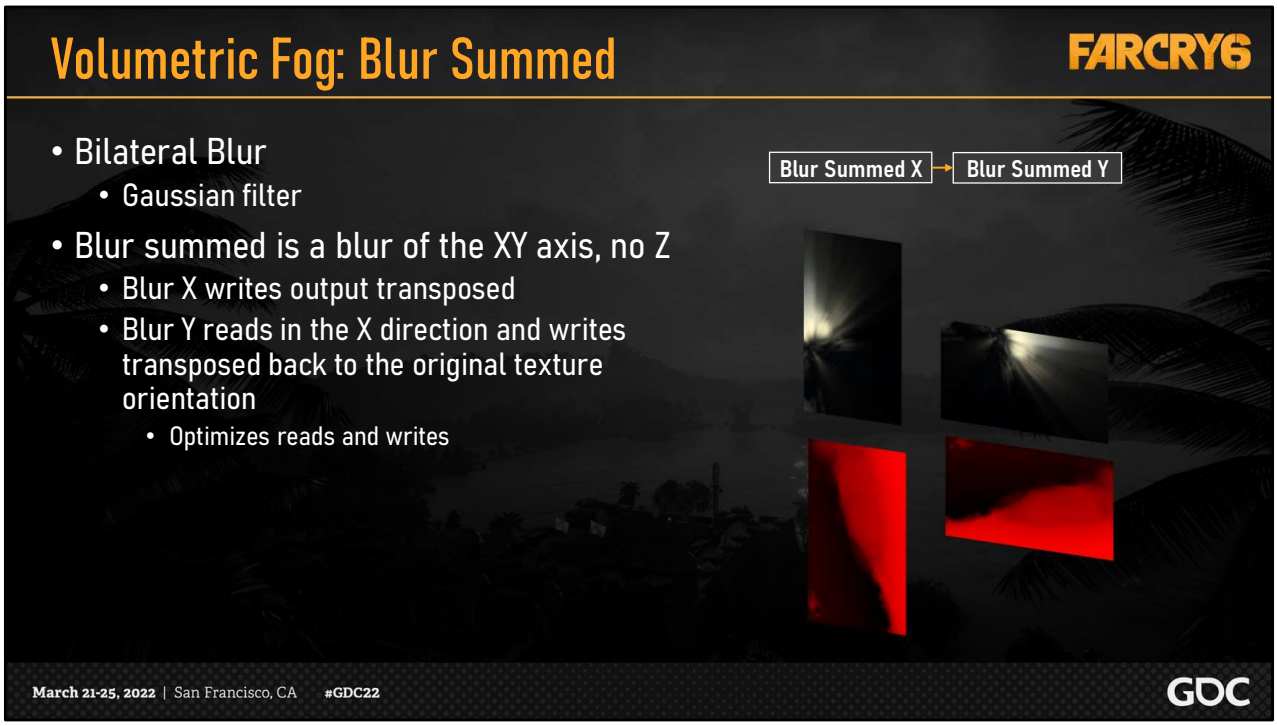
Graph of Fog Phase Function https://www.desmos.com/calculator/dcgvobzm4n



After we fill the cells with irradiance, we calculate a continuous sum towards the back of the volume

Now we can sample any voxel for full irradiance accumulation, from the eye to destination.

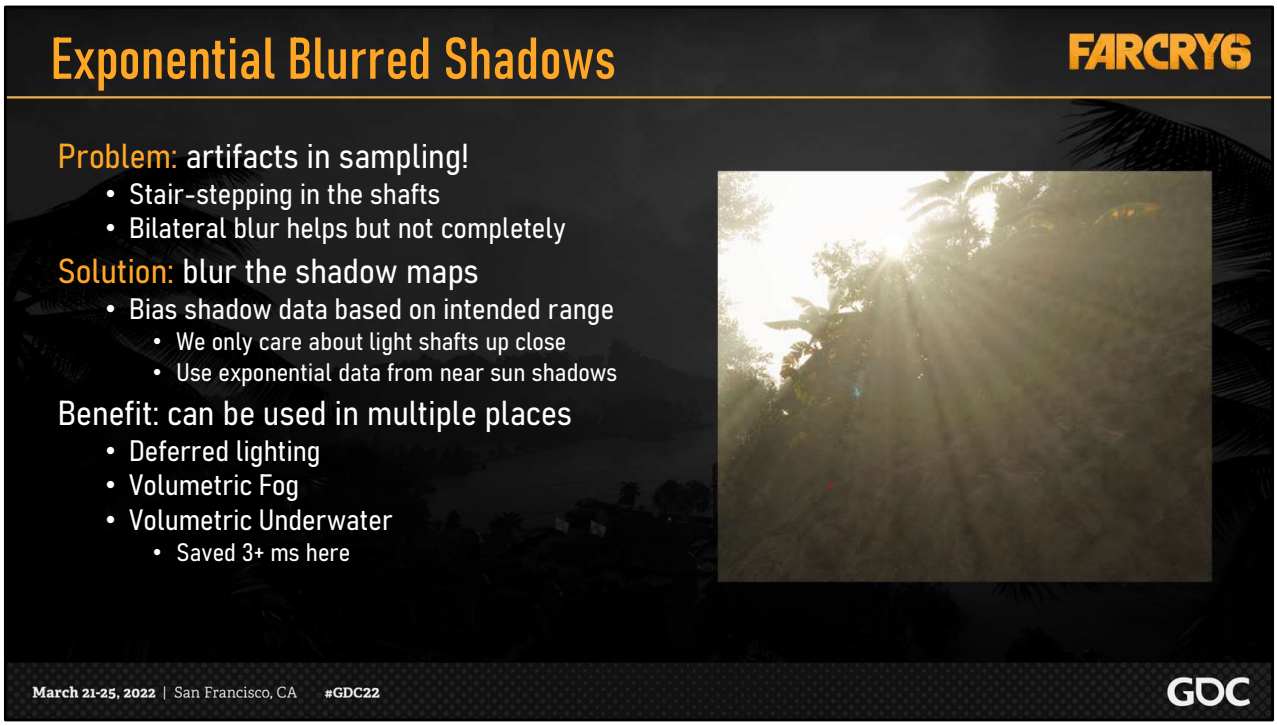
[Next]



And finally, on next gen, we do a two stage process of a bilateral blur in the x and y axis.

Blurring in the X axis outputs a transposed result, and blurring in the Y axis transposes back to the original orientation. This optimizes the reads and writes of the textures.

[Next]



But there's a problem!

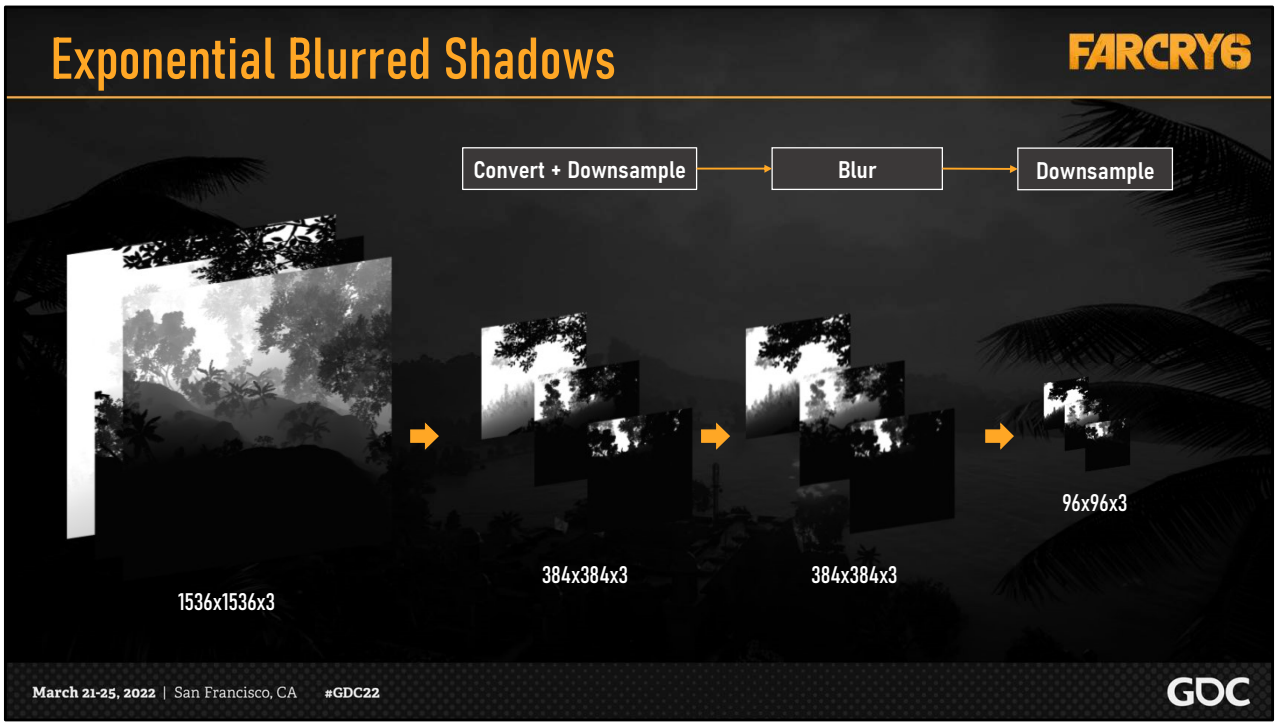
On the right you can see stair-stepping in the light shafts, which is an artifact of sampling!

We want this to be smooth. The bilateral filtering HELPS but doesn't solve this completely.

The next approach we tried was to blur the actual shadow maps.

One benefit of this approach is that the result can be used in multiple places.

[Next]



Here we see the process for exponential blurred shadows

First, we convert and downsample our three near shadow slices or cascades of our shadow map for the sun.

Then, we blur the slices and downsample again.

As you can see, the result is significantly smaller than the original which is more optimal for quality and performance when volumetrically sampling.

Let’s look at a sample scene with no fog from our weather manager.

Here we see the fog on its own with less than ideal artifacting.

This is the result of blurring the irradiance volume in screen space. It’s better but there are still artifacts that suggest the shape of the voxels.

And finally, we enable the exponential blurred shadows.

Upsample and Apply Atmospherics

FARCRY6

- Up-sample and apply fog and clouds simultaneously[Bauer19]
- Easy to add projected cloud shadows into the fog irradiance calculation
  - Gives nice light shafts from clouds
- Composite with blue noise, at the final screen resolution
  - Temporal AA eliminates artifacting

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

GDC

To this point we have not mentioned how we apply the clouds

We apply atmospherics, clouds and fog at the same time to save on bandwidth writing to areas of the screen where both are visible.

You can also see here that the clouds shadows give nice light shafts in the fog.

One final detail in our composite pass is that we apply the final value with more blue noise at the final screen resolution.

This, when combined with the temporal AA post process, further eliminates raymarching artifacts visible in the clouds.

[Next]

Reflections

FARCRY6

- Necessary to completing the visual concept of wetness
- *WetnessFactor*, *PuddleFactor*
- Does most of the work reflecting objects in the scene
- Provides the crucial reflections on water surfaces
- See our other GDC talk: "Performant Reflective Beauty - Hybrid reflections in Far Cry 6"

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

GDC

Next lets talk about reflections.

Reflections are crucial to completing the visual concept of wetness.

As Emily mentioned before, we use values from the weather system to determine what surfaces are wet.

Then we perform a process where rays are path traced using gbuffer positions and normals to pixels in screen-space to be reflected.

This does most of the work in reflecting objects in the scene, such as the trees, sky, and clouds on the surface of the road as well as the mountains shown on the water surface.

For more information about our SSLR as well as our hybrid raytracing solution, see my colleagues Stephanie and Ihor's talk: Performant Reflective Beauty.

[Next]

Cubemaps

FARCRY6

- Needed fallback for when SSLR is not viable
  - Raytracing not available on most platforms
- Needs to include time of day and now weather too
- Needs to be relit with atmospheric scattering, clouds and fog

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

GDC

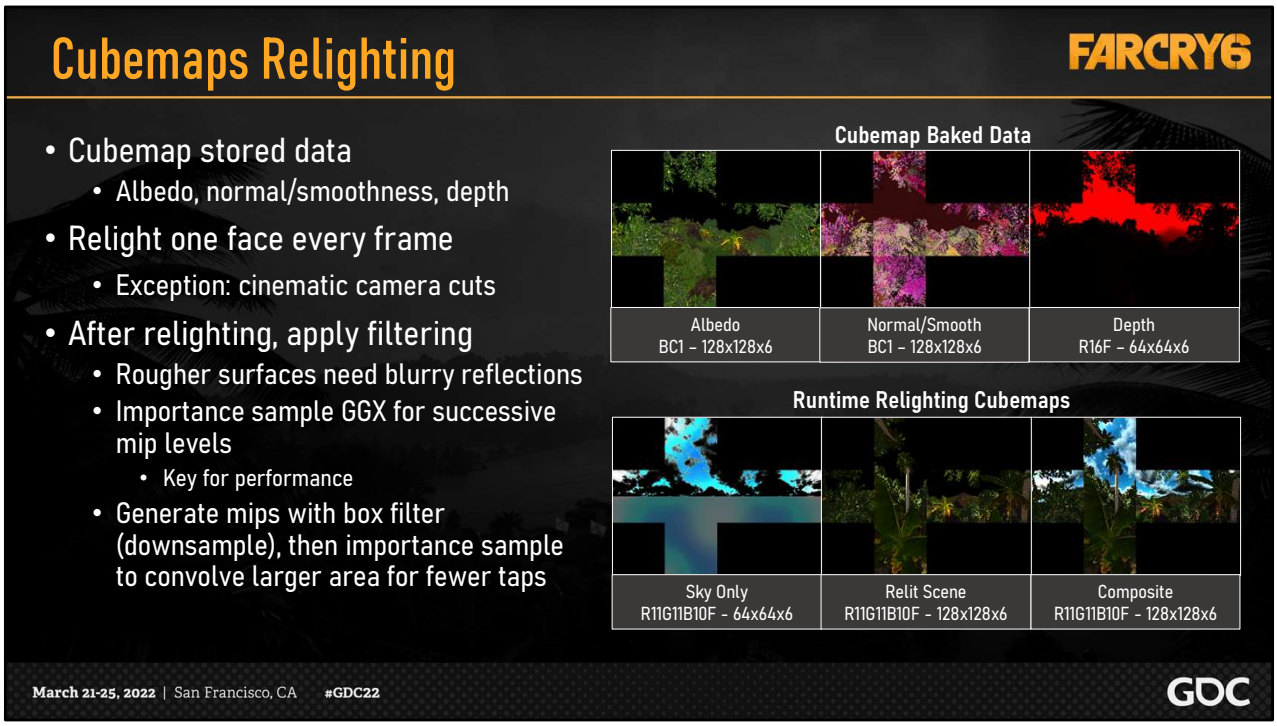
But what happens when we don't have data for screen space reflections, such as when we view water at the angle shown in the image on the right?

We need a fallback when there is no data to drive screen-space reflections.

Our fallback comes in the form of cubemaps, which need to show time of day changes, weather, and clouds, as shown by the sunset reflections on this car.

Our Cubemaps are generated at build time in positions determined by artists. They are then relit with our atmospheric scattering, clouds and fog.

[Next]



Cubemap data is baked and stored as albedo, normal and smoothness, as well as a lower resolution depth.

To achieve acceptable performance, we relight just one face of the cubemap each frame, except in special cases such as cinematic camera cuts.

The process starts with generating a sky only cubemap containing only atmospheric scattering, clouds and fog. The sky-only cubemap used for fallback reflection on our ocean.

We then relight the scene face using the baked cubemap data.

Finally, we composite the sky cubemap into the relit scene cubemap, giving the result shown bottom right.

We progressively update a single face each frame, attempting to process cubemaps when streaming in new areas.

After the relighting process, cubemaps need to be filtered for blurrier reflections on rougher surfaces.

[Next]

Overcast Lighting and Weather

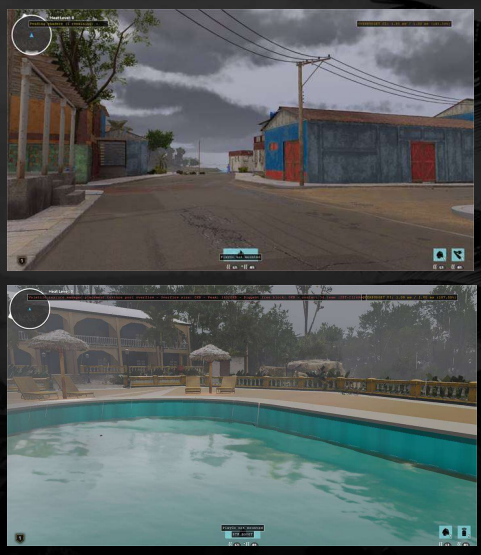
FARCRY6

Problem: Overcast Lighting and Weather

- Not dark enough
- Can appear flat
- Want contrast in clouds, blue sky breaking through

Reasons for this:

- Authored skies too turbid and grey
  - Blue comes from here, needs to be balanced
- Fog washing out the image
  - Fade based on CumulusCoverage
- Sky lighting too bright
  - Clouds never reach horizon, horizon present in cubemap
- Clouds are uniform thickness
  - Limit CumulusCoverage to 0.8 during storm
- Atmospheric scattering has no shadowing



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

GDC

One large problem we had was overcast lighting and weather.

Sometimes it just wasn't dark enough. It could also appear flat.

We want the clouds to have contrast and sometimes have blue sky breaking through

There were several reasons why this was happening. One such example was our authored skies being too turbid and grey. If you want blue at all in your natural lighting, it comes from here.

But be careful, otherwise atmospheric scattering on clouds too blue

We also had issues with the fog washing out the image. To fix this we fade fog based on cloud coverage.

We had Sky lighting that was too bright because clouds never reach horizon and the cubemap has a band of blue at the horizon that contributes to the brightness.

Clouds of uniform thickness could also wash out the image, which is why we limit CumulusCoverage to 0.8 even during a storm.

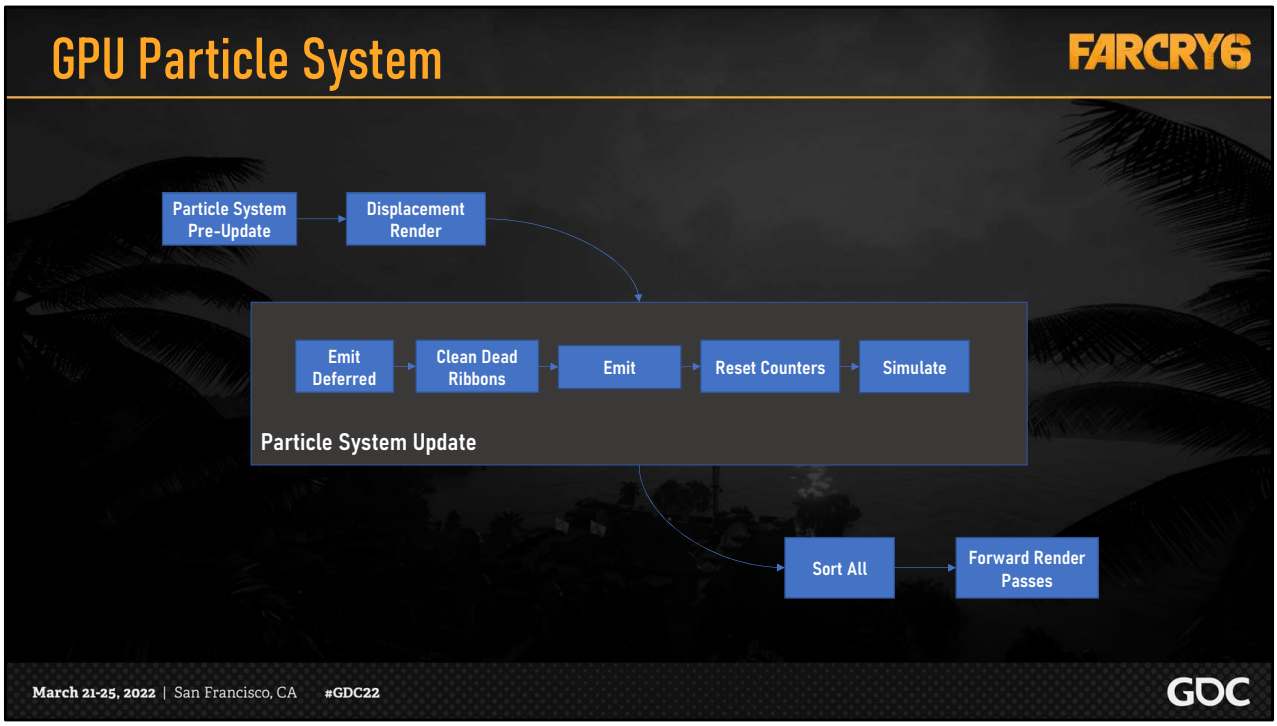
And finally, our Atmospheric scattering has no shadowing which inherently causes problems as well.

[Next]

Rain (GPU Particles)

FARCRY6

- Needed rain particles but CPU system was limited
- Created a GPU particle system for rain
- The system was generalized and leveraged for other GPU particle effects
- Weather FX:
  - Rain streaks
  - Rain splashes
  - Lightning



Here's a brief overview of our particle system you can refer to later.

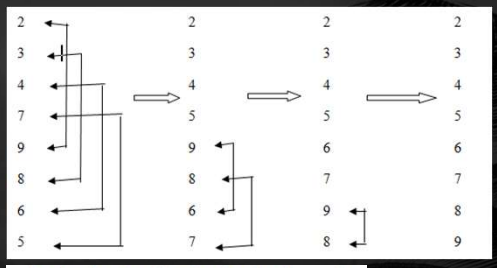
In summary, we use compute shaders to emit, simulate, sort and render all the particles all on the GPU with minimal interaction with the CPU.
[Next]

GPU Particle Sorting

FARCRY6

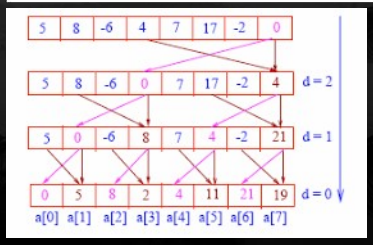
Sorting

- Universal sorting with other GPU particles using Bitonic Sort
  - A parallel algorithm with higher average-case cost, but lowest worst-case cost
- 37 dispatches for full data set, 65k part sort
  - With less population, many dispatch 0 arguments
  - Can use predicated rendering to eliminate this overhead



Filtering

- 6 different passes where particles render
  - Before water, after water, displacement, distortion, small buffer (half and quarter res particles), opacity
- Prefix sum filtering
  - Highly parallelized algorithm – perfect for GPU
  - Useful for eliminating empty space and sum reduction



Rain Particles

FARCRY6

Streaks

- Experimented with refraction, blur and reflections
- Ultimately used transparent textures
  - Closer to the look of rain photography
- Recycled particles in a cylinder around the player
  - Wrapped vertically as well for aerial gameplay
- Used a 3D noise texture as turbulence
  - Shared with
    - Volumetric underwater distortion
    - Volumetric fog particle distortion
- Rain direction is driven by weather system
  - Wind direction and magnitude

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

GDC

Let's go through some of our particle effects. The first particle effect we will look at is the rain streaks.

We experimented with refraction, blur and reflections, but we ultimately used transparent textures to achieve the look of rain in photography, which was our art direction.

We used two frames of variation and a normal texture to aid in the lighting.

We recycle the rain streaks in a cylinder around the player to ensure the number of rain particles remains consistent, regardless of how fast the camera moves.

To add continuous variation in the particles throughout their lifetimes, we used a 3D noise texture mapped to world space as turbulence.

And finally, the emission rate and direction of the rain is determined by rain and wind values provided by the weather manager.

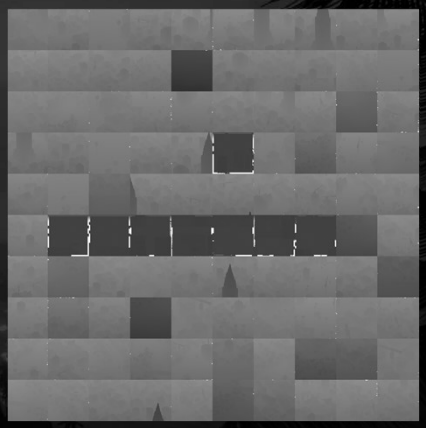
[Next]

Rain Shadow Map

FARCRY6

- Lots of indoor/outdoor environments
- We need to occlude rain wherever it was not plausible
- Rendered directional shadow with view direction of rain falling (driven by wind direction)
- We use this rain shadow map for streaks and also splashes

Rain Shadow Atlas Lookup



Distant Shadow Atlas

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

GDC

Since we had many indoor and outdoor environments, we needed to occlude rain where it was not plausible.

To do this, we created a rain shadow map that we can reference to determine if a rain streak or splash was occluded.

This was a directional shadow set up to cover the playable area near the camera, parallel to the rain direction

We then stored this in our regular sun shadow atlas, shown on the right. The colored image left of it is the look up table that is used to convert UV space into the atlas texture.

This rain shadow map only contains static scene elements and is updated a few sections each frame.

[Next]

Rain Particle Data

FARCRY6

Splashes

- Driven by the event system
  - Emit on collision with depth, terrain or water
    - Ripple, same as [Grujic2018], but only on GPU
- Later spawned more particles around the camera to increase volume
  - Snapped to the depth buffer
  - Not systemic but faked well

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

The next particle effect we had was the splashes that appear when rain hits a surface.

Initially this was just driven by a GPU particle event system, which emits new particles when a parent particle hits a surface on the depth buffer, terrain or water.

This means we get splash particles on any surface which is opaque, including vehicles, characters and weapons.

However, we didn't necessarily want these splashes to be on the sides of buildings, so we incorporated a slope factor.

Another problem was that we weren't getting enough splashes because not every particle that we checked for collision would generate a splash.

So, we spawned more particles in a volume around the camera and snapped them to the depth buffer.

[Next]

Rain Particle Lighting

FARCRY6

- Wanted pixel lighting for rain at night
  - Too expensive
- Used Spherical Harmonic probes
  - Sun, point and spot lights
  - Shadows
  - Coefficients calculated per-vertex
- This gives us better than vertex lighting
  - With directionality
  - Combined with normals we get more accurate specular highlights

Sphere Billboard Particle

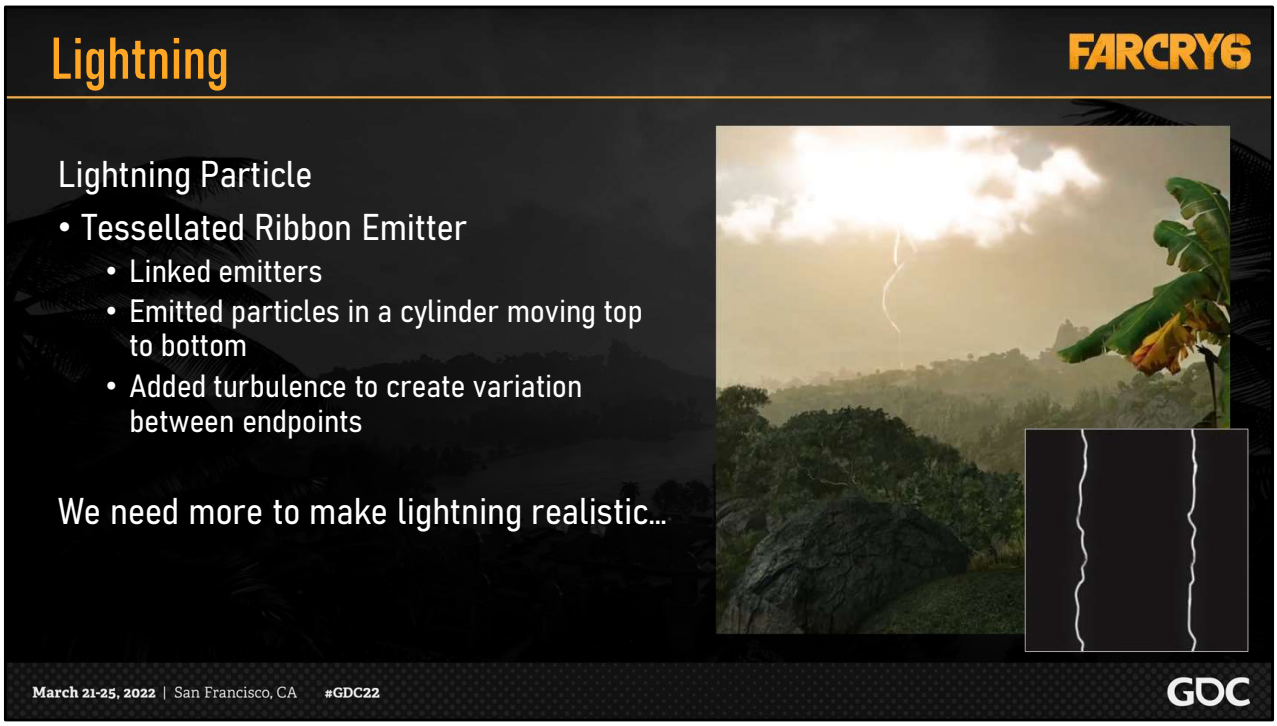
For both the rain streaks and splashes we wanted pixel lighting, especially at night. However this was too expensive and vertex lighting wasn't good enough.

What we did was create a system for generating spherical harmonic light probes for each vertex of the particle. Each of these probes calculates 3rd order SH coefficients and incorporates sun, point and spot lights.

This gives us better than vertex lighting but much cheaper than pixel lighting. In addition, we can sample the normal maps of particles for more directional lighting and better specular highlights.

Here we see an example of this with a particle billboard of a sphere on the right with only a diffuse and normal texture.

[Next]



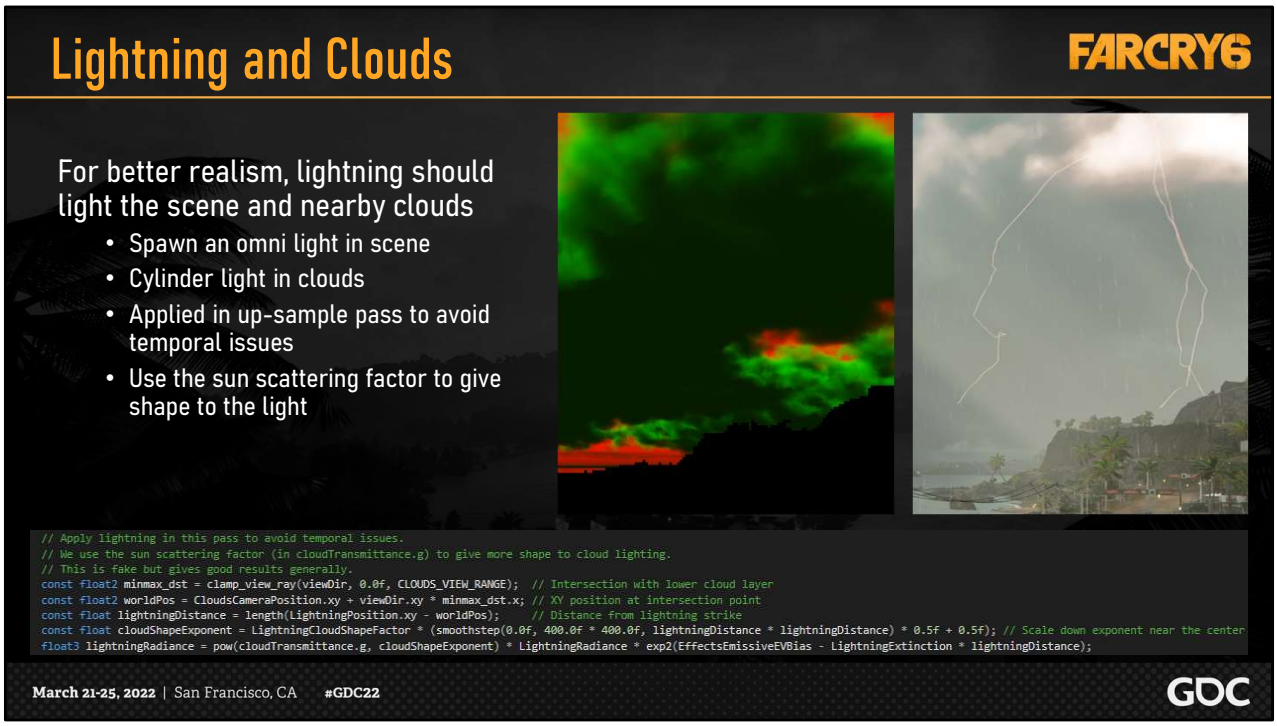
The next particle effect we needed was lightning bolts for our thunderstorms.

For this effect we created a ribbon (or trail) system. We started by creating trails automatically left by individual particles, but then extended this by generating particles and tessellating the geometry between each particle. We called these linked emitters.

Using the linked system, we emitted particles in a cylinder moving from top to bottom, increasing the turbulence in the center of the volume and fading it out at each end.

But this alone was not very realistic – we needed to make the lightning affect the world around it.

[Next]



First, we generated an omni light and placed it in the scene so that the lightning would light the environment.

Then, we had to address the clouds around the lightning bolt.

We applied the lightning to the clouds in the upsample pass so that the lightning doesn't end up in the history buffer and cause ghosting.

In order to determine where in the clouds to add light, we used the sun scattering factor that we saved in the green channel of the cloud raymarching pass.

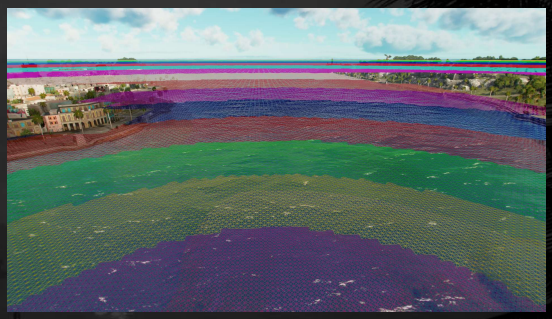
Our implementation functions essentially like a column light.

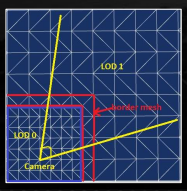
[Next]

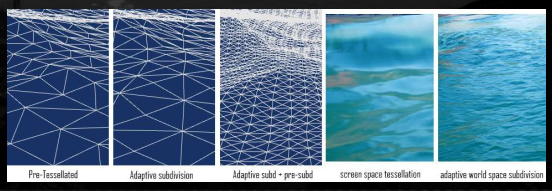
Ocean

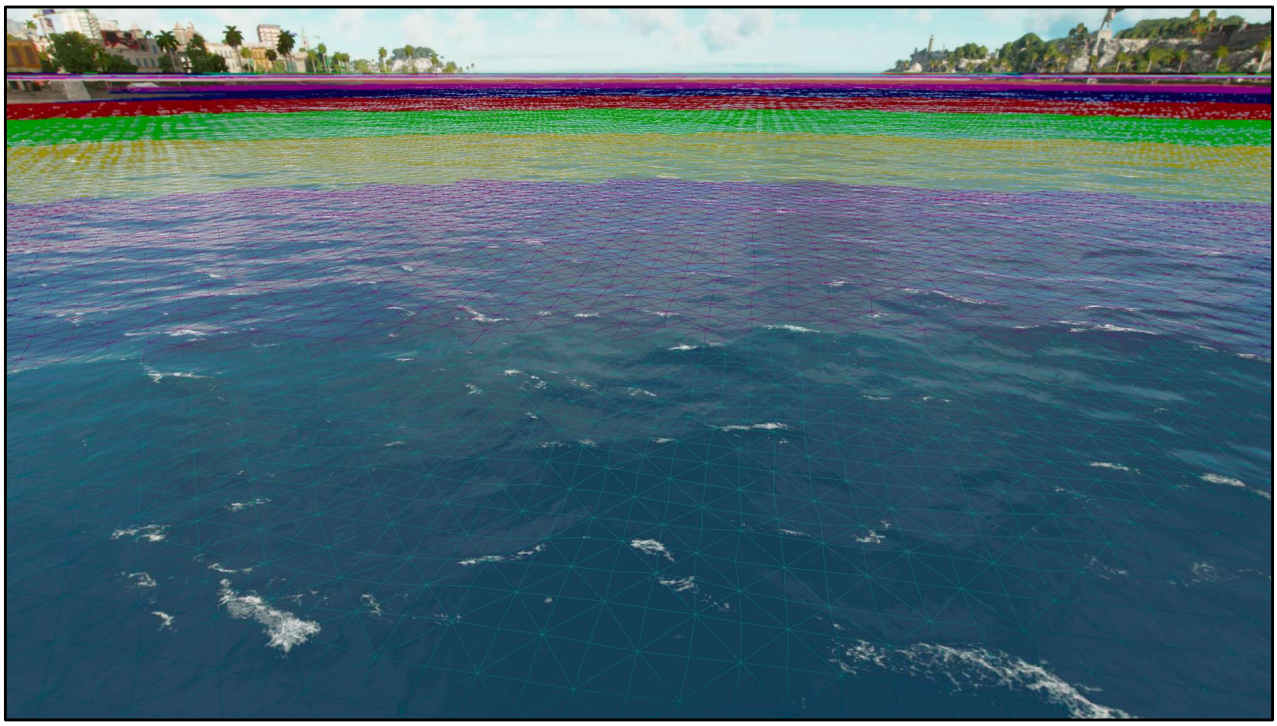
FARCRY6

- Ocean mostly supports the tropical theme
- But ocean is affected by weather in two ways
  - Beaufort level
  - Wind direction
- Screen space tessellation was limited
  - Issues with shoreline waves
  - Poor tiling pattern in the distance
  - We kept this for freshwater (rivers, lakes, streams)
  - For more information see [Grujic18]
- New tessellation (SUBD)









The tessellation we chose for our ocean was based on the iSubD tessellation scheme.

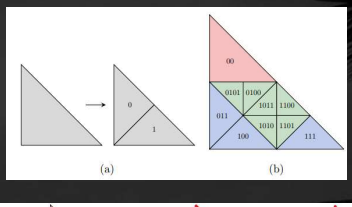
The basic algorithm subdivides the ocean mesh from a single two triangle quad down to the subdivided triangles you see near the camera.

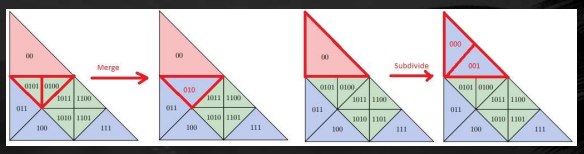
It is a progressive refinement algorithm that will subdivide into smaller triangles each frame as they get closer to the camera and merge triangles into larger ones as they get further away.

Ocean: iSUBD Tessellation

FARCRY6

- Subdivide a coarse mesh [Khoury 2018]
- Used keys to perform subdivision and merge operations
- Unsolved problem was how to handle all the key generation
  - Used parallel prefix sum again
- New tessellation must be blended with screen space tessellation
  - Use displacement for freshwater on ocean and fade at intersections





There is a performance limit on the number of operations that can be done every frame, so we filled a buffer with keys to perform these operations, but this led to a list with lots of empty space.

This was an area of the Subd algorithm that we found to be unsolved, and we were able to use the same prefix sum algorithm from the GPU particle filtering on a different type of data set.

In our world, we have freshwater that meets the ocean, and therefore we needed to blend the previous screen space tessellation with the subd tessellation.

To do this, we used the displacement and normals of the freshwater as an input at the intersections of these do tessellation types and blended in that range. We then used simple stencil testing to avoid redundant pixel shading of the water meshes.

[Next]

The Beaufort scale is an empirical system that relates wind speed to its effect on bodies of water.

Here we will show how the Beaufort levels coming from the weather manager can impact the look of the water.

This is Beaufort level 0.

The Beaufort levels themselves are tuned with data such as wind speed, amplitude, scale and choppiness.

Here we have Beaufort level 1…

[Next]

These values are used to drive the wave simulations we will show later.

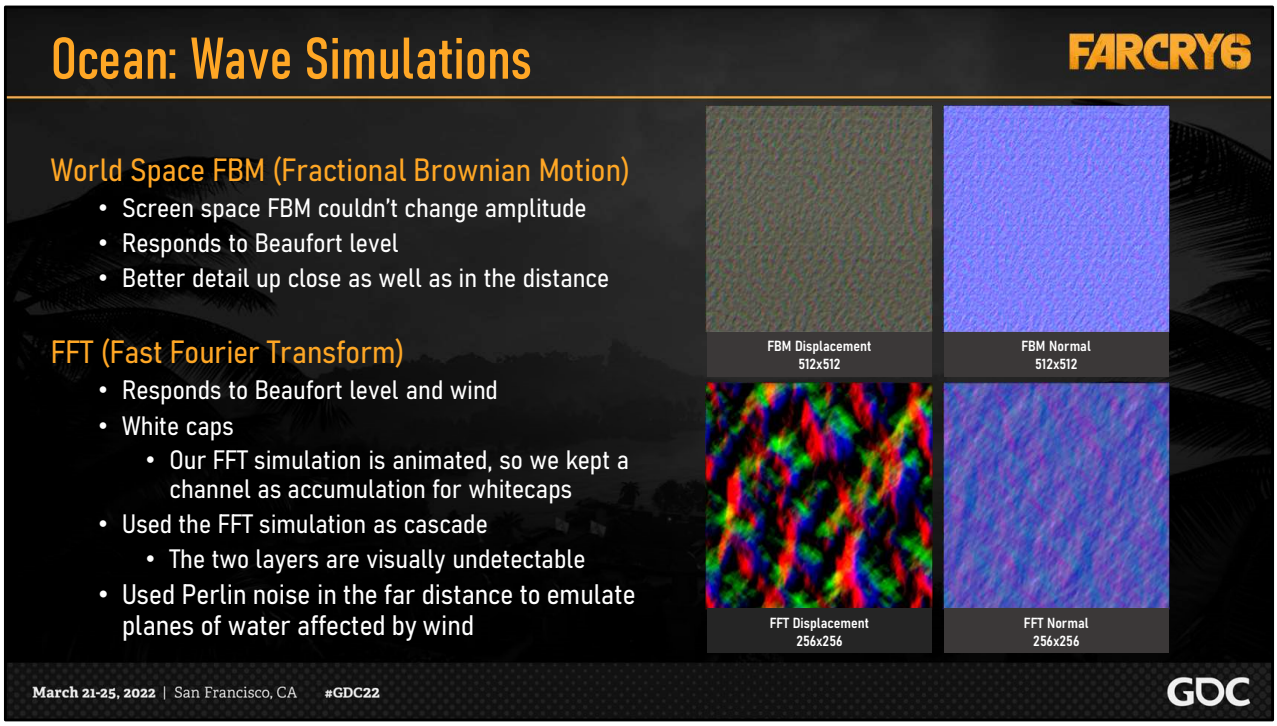
Here we have Beaufort level 2

The Beaufort levels also include settings for the shoreline waves, such as amplitude, frequency, speed, steepness and number of waves.

Here is Beaufort level 3

The Beaufort levels include settings for foam, which you can see accumulating alongside the increased wave size.

And this is Beaufort level 4.



Here we have the ocean wave buffers that we generate every frame in order to create motion in the waves.

These buffer can be mapped back and read on the CPU for physics calculations and ocean level, but we disabled this due to performance limits.

The world space fbm was created to get sharp waves up-close near the camera.

Using a world space buffer gave us the ability to generate a normal directly, instead of from screen space like we did previously.

This gave us better details up close as well as in the distance with mip mapping.

Next, we used the FFT simulation for waves in the distance.

We adjusted our original implementation to respond to the Beaufort level and wind parameters.

We also added an accumulation channel to the texture to create persistent white caps on the waves.

To prevent tiling in the distance, we use the FFT buffer as a cascade at a different scale, as well as a Perlin noise wave texture.

[Next]

Here we see only the FFT buffer we started with.

Notice the tiling in the distance.

Then we have the FFT cascade eliminating some of the tiling.

And finally we have the scrolling Perlin noise based texture removing the rest of the tiling. This does a good job simulating the effect of wind on large patches of water in the distance.

[Next]

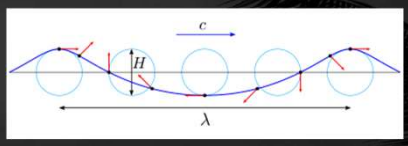
Shoreline Wave

FARCRY6

- Procedural - too many shores to hand-place particles or meshes
- Gerstner wave formula:

$$\xi = \alpha - \sum_{m=1}^{M} \frac{k_{x,m}}{k_m} \frac{a_m}{\tanh(k_m h)} \sin(\theta_m),$$
$$\eta = \beta - \sum_{m=1}^{M} \frac{k_{z,m}}{k_m} \frac{a_m}{\tanh(k_m h)} \sin(\theta_m),$$
$$\zeta = \sum_{m=1}^{M} a_m \cos(\theta_m) \quad \text{and}$$
$$\theta_m = k_{x,m}\,\alpha + k_{z,m}\,\beta - \omega_m\,t - \phi_m,$$

- Controlled with parameters:
  - Amplitude, steepness, parallelity, speed, length, foam
- Generated signed distance field
  - Shore direction and distance
- Added noise to disrupt parallelism
- Supported 5 waves but only used 1



Related to the new ocean tech was the inclusion of shoreline waves, since we needed them for our tropical beaches.

We had far too many shorelines to cover so we needed a procedural method, which led us to choose the Gerstner wave formula.

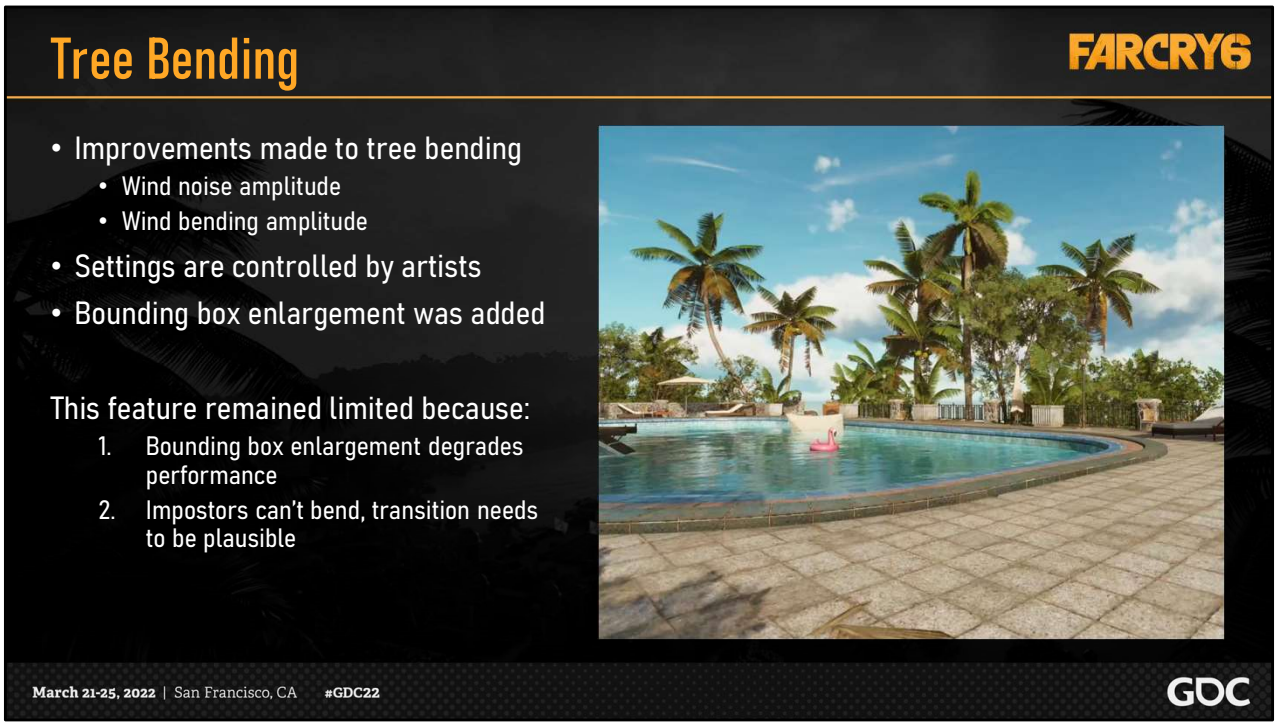
In order to place and move these waves properly we needed to generate a signed distance field from the ocean to the shore to derive direction.

We used parameters such as amplitude, speed, and foam to control the visuals.

We also added a noise parameter to the waves that helps break up the parallelism of the wave, otherwise we would have perfect circular waves approaching our islands.

Our system supported multiple waves, but we only used one in the final game.

[Next]



The last rendering feature we'd like to talk about is tree bending, which works in conjunction with the wind direction and magnitude that comes from the weather manager.

We improved the tree bending settings by adding noise and bending amplitudes. Combined with the wind values, this gives us exaggerated movement in the trees that reflects our stormy weather conditions.

These settings are controlled by artists when setting up the trunk skeletons.

The movement could sometimes exceed the tree's bounding box however, which was problematic because leaves could suddenly get culled and disappear. To fix this, we also included settings to enlarge the bounding box.

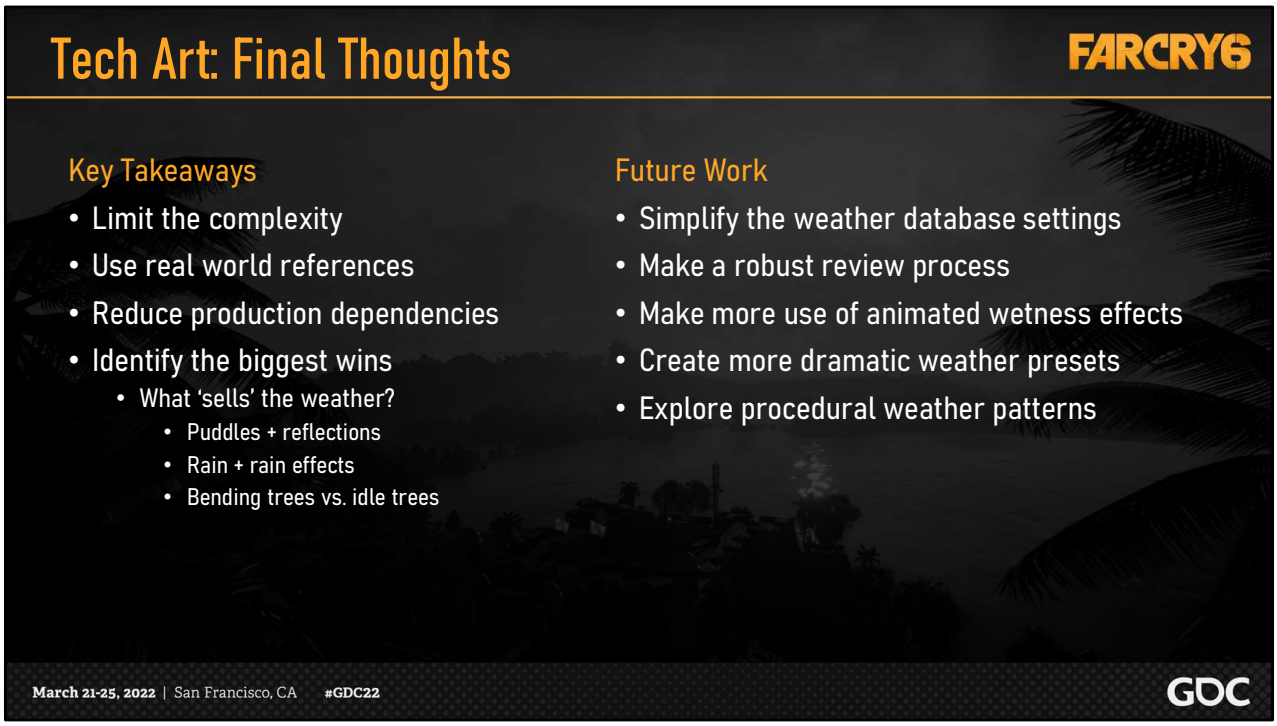
Ultimately, this feature was still rather limited because:

One - the bounding box enlargement affected performance, with more trees staying visible each frame.

and Two – Our distant tree impostors cannot bend, so the movement up-close still needs to allow for an acceptable transition.

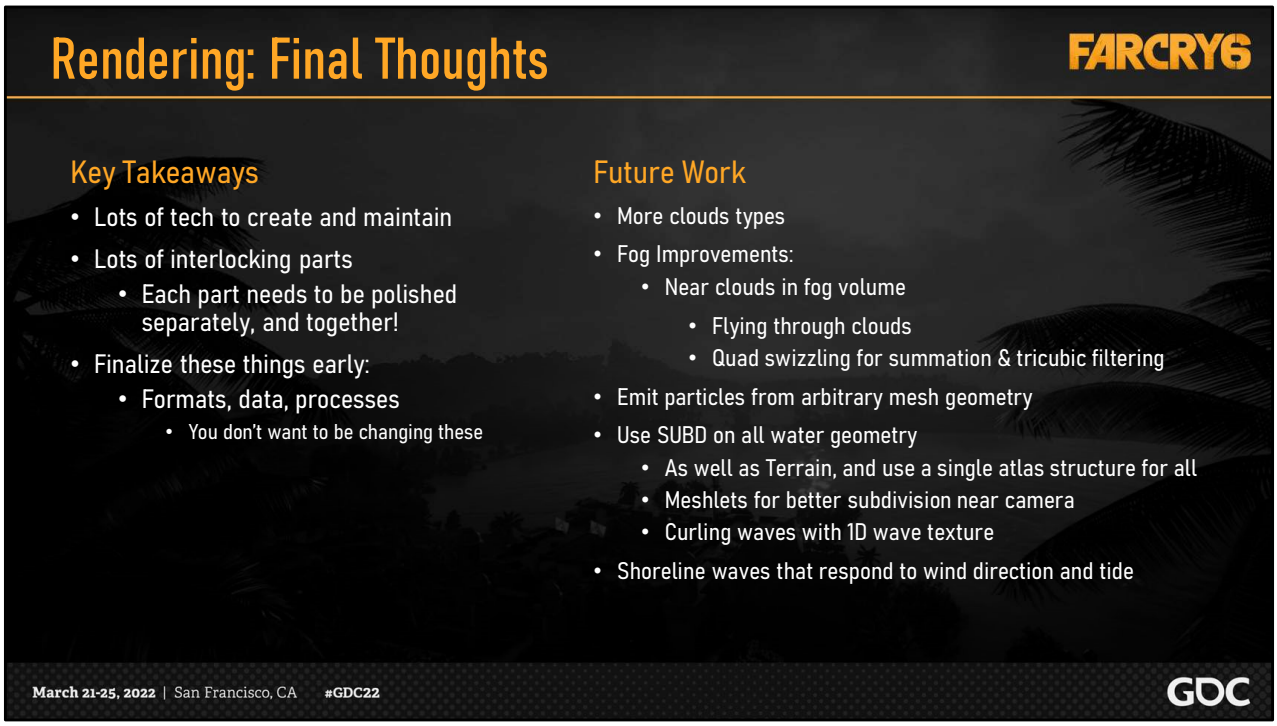
[Next]

To conclude, we would like to share some of our final thoughts from both the tech art and programming perspectives.



For me, things that really drove our success were

For future work, we could simplify the database further, set up an easier way to debug and review weather without overrides confusing us, make more use of wetness effects, push our presets to be more extreme, and investigate procedural weather patterns as opposed to our hardcoded forecasts.

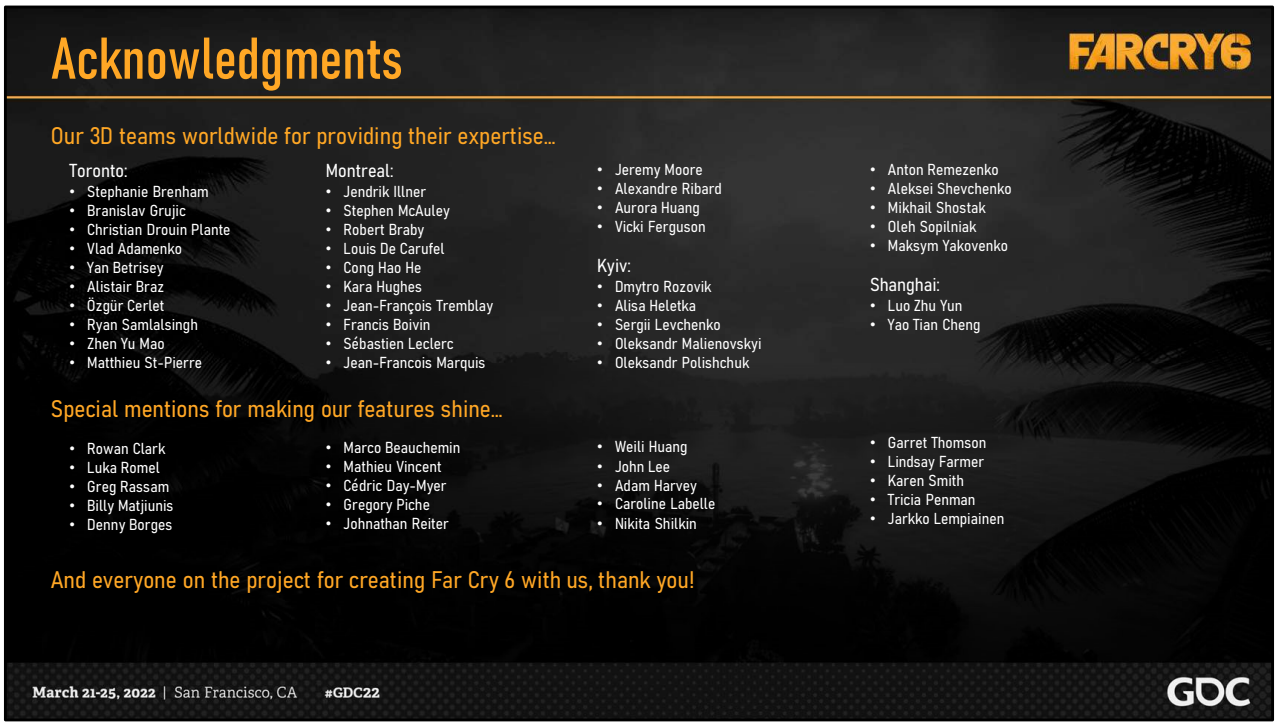


For rendering takeaways, as you saw there was a lot of tech to maintain.

There were many interlocking parts, which needed to be polished both separately and together.

And finalize these things early! That means your data formats, data and processes. You don't want these changing too far into production.

For future work there are several things listed but I think the most interesting to me would be to use the same tessellation for both water and terrain and use the same atlas and virtual texturing.



We want to quickly thank the 3D teams who worked on this feature and everyone who helped us along the way. As you can see, it was a collective effort!

References

FARCRY6

[Grujic2018] Water Rendering in Far Cry 5, Branislav Grujic & Cristian Cutocheras, GDC 2018

[Moore2018] Terrain Rendering in Far Cry 5, Jeremy Moore, GDC 2018

[Preetham1999] A Practical Analytic Model for Daylight, A. J. Preetham et. al., SIGGRAPH 1999

[Wronski14] Wronski, B., Volumetric Fog: Unified Compute Shader-based Solution to Atmospheric Scattering, SIGGRAPH 2014, Advances in Real-Time Rendering

[Hillaire15] Hillaire, S., Towards Unified and Physically-Based Volumetric Lighting in Frostbite, SIGGRAPH 2015, Advances in Real-Time Rendering

[Bruneton08] Bruneton, E., Neyret, F., Precomputed Atmospheric Scattering, Eurographics Symposium on Rendering 2008

[Schneider15] Schneider, A., The Real-time Volumetric Cloudscapes of Horizon: Zero Dawn, SIGGRAPH 2015, Advances in Real-Time Rendering

[Bauer19] Bauer, F., Creating the Atmospheric World of Red Dead Redemption 2: A Complete and Integrated Solution, SIGGRAPH 2019, Advances in Real-Time Rendering

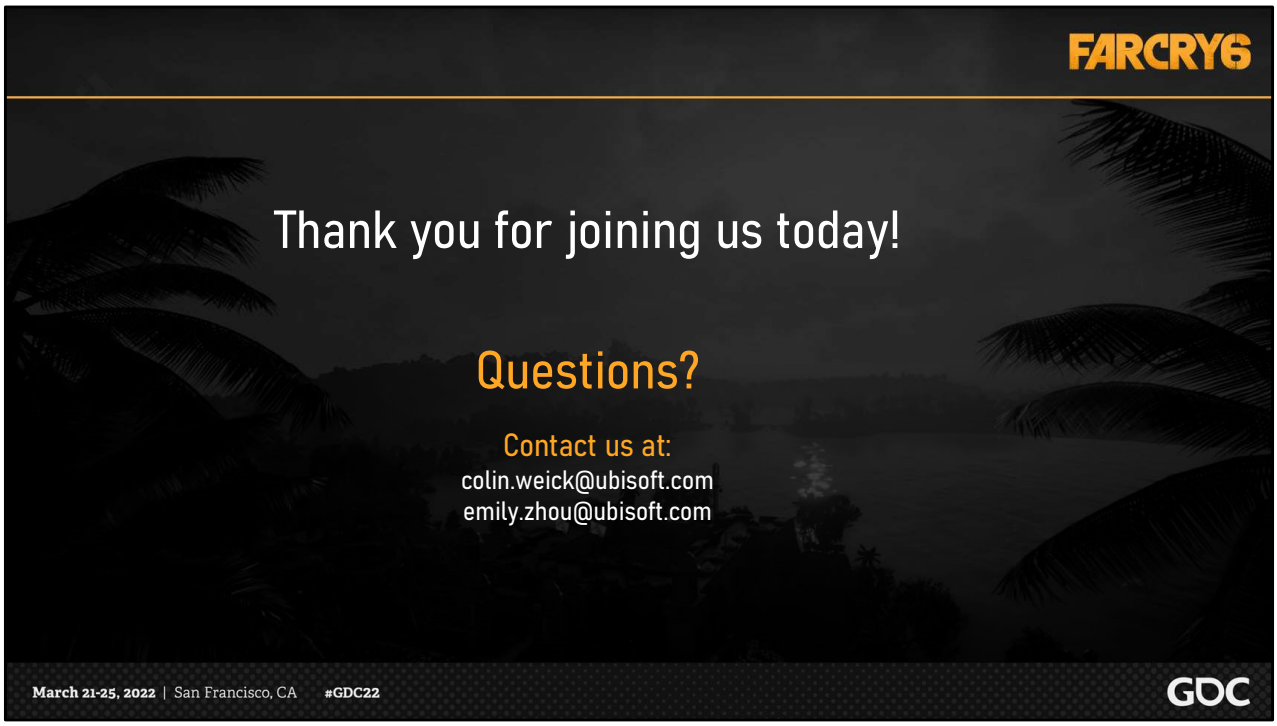
[McAuley19] McAuley, S., Advances in Rendering, Graphics Research and Video Game Production, i3D 2019, ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games

[Eggers10] Eggers, J. et. al., Drop Dynamics After Impact on a Solid Wall: Theory and Simulations, Physics of Fluids 22, 062101 (2010)

Here are the references for this talk…

…And with that, we will close out with a video of our final results.

This wonderful video was put together for us by Aalaap Majgavkar.


FARCRY6
Thank you for joining us today!
Questions?
Contact us at:
colin.weick@ubisoft.com
emily.zhou@ubisoft.com
March 21-25, 2022 | San Francisco, CA #GDC22
GDC

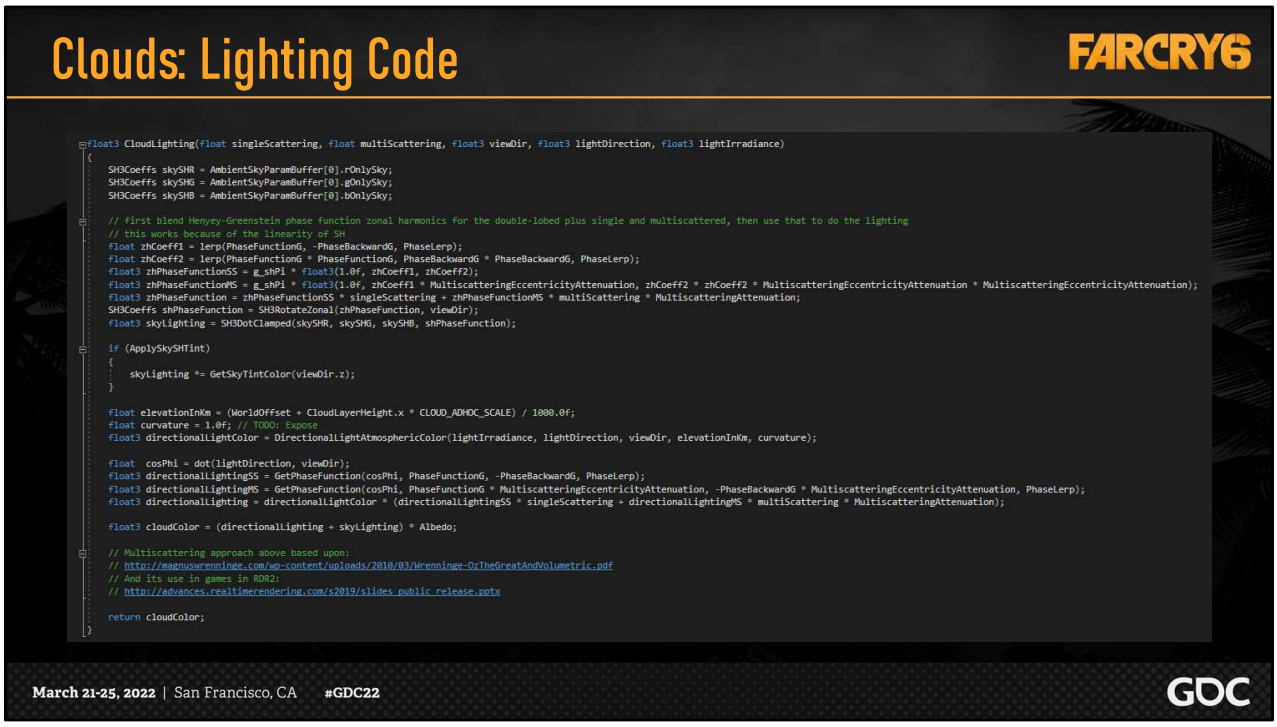
Thank you!

Clouds: Raymarching Pseudocode

FARCRY6

```
float4 ray_march_single(float3 ro, float3 rd, float t0, float t1, uint numSteps, float3 lightDirection)
{
    calculate dt based on far-near divided by primary samples // number of primary samples affects amount of visible noise, memory bandwidth, and detail

    calculate initial t with jitter // jitter the initial ray time along one step length to amortize raymarch artifacts.

    evaluate the density, transmittance, optical depth and scattering at fixed intervals(dt)
    for (numSteps = 0; numSteps < CloudsPrimarySteps; ++numSteps)
    {
        sample cloud density at (ro + rd * t) by applying BaseNoise, DetailNoise, Gradient, WeatherMap and CurlNoise

        integrate density from(ro + rd * t) back to sun to find optical depth // number of sun samples has a significant effect on banding effect on bottom of clouds

        convert optical depth to transmittance: exp2(-optDepth)

        scale transmittance of sun by scattering coefficient of view ray: * density
            sunSingle = exp2(-optDepth) * density
            sunMult = exp2(-optDepth * MultiscatteringExtinctionAttenuation) * density // this is a hack for multiscattering

        integrate transmittance along segment [Hillaire2016]
            sunSingle = (sunSingle - sunSingle * exp2(-dt * density)) * rcpDensity
            sunMult = (sunMult - sunMult * exp2(-dt * density)) * rcpDensity

        apply transmittance of previous view ray segments

        accumulate scattering along segment for single and multiscattering

        apply extinction to view ray transmittance for this step

        min of t to store first scattering event

        t += dt
    }

    return
        r: single scattering to sun
        g: multiple scattering to sun
        b: transmittance
        a: distance to first scattering event
}
```


This is what the raymarching looks like in pseudocode



When calculating the irradiance based on the single and multiscattering from the ray-marching

Clouds: Phase Function Code

FARCRY6

```
float DoubleLobedHenyeyGreensteinFast(float cosPhi, float gFactor0, float gFactor1, float alpha)
{
    cosPhi = max(min(cosPhi, 0.99), -0.99);

    float gFactor02 = gFactor0 * gFactor0;
    float d0 = abs(1 + gFactor02 - 2 * gFactor0 * cosPhi);

    float gFactor12 = gFactor1 * gFactor1;
    float d1 = abs(1 + gFactor12 - 2 * gFactor1 * cosPhi);

    return OneOver4Pi * fastRcpNR0(d0 * d1) * lerp((d1 - d1 * gFactor02) * fastRcpSqrtNR0(d0), (d0 - d0 * gFactor12) * fastRcpSqrtNR0(d1), alpha);
}

float GetPhaseFunction(float cosPhi, float g0Factor, float g1Factor, float alpha)
{
    // Using a double-lobed phase function helps stop lighting being too dark opposite the sun, as we lack back scattering
    // This is more physically correct than the approach proposed by RDR2 in their SIGGRAPH 2019 talk, which just clamps the amount of backscattering:
    // http://advances.realtimerendering.com/s2019/slides_public_release.pptx
    return pi * DoubleLobedHenyeyGreensteinFast(cosPhi, g0Factor, g1Factor, alpha);
}
```


And this is the optimized phase function we used in the previous lighting function.

We used this in the fog as well