*Please note that the original powerpoint contained movie files and gif animations which became converted into still images in the pdf.

Image sharpness was also lost during the powerpoint to pdf conversion.

Kosuke Tanaka

Programmer @ Tango Gameworks since 2011.

- VFX Programmer for “The Evil Within”.
- Graphics Programmer for “The Evil Within 2”.
- Lead Graphics Programmer for “Hi-Fi RUSH”.

We’d like to start off the talk with brief speaker introductions.

Hi, I’m Kosuke. I’ve been working as a graphics programmer at Tango Gameworks since 2011.

For “Hi-Fi RUSH”, I worked as lead graphics programmer working on our core toon rendering.

I went from bloody vfx for zombie headshots and moody survival horror lighting to toon rendering.

Takashi Komada

Programmer @ TangoGameworks since 2016.

- Physics and graphics programmer for “The Evil within 2”.
- Physics and animation programmer for “Ghostwire: Tokyo”.
- Graphics, physics and animation programmer for "Hi-Fi RUSH“.

Hi, I’m Takashi Komada.
I am a graphics and physics programmer.
I’ve been working as a programmer at Tango Gameworks since 2016.

On "Hi-Fi RUSH“, I was graphics, physics and animation programmer.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination

Toon Face Shadow

Final Words

Here’s the agenda for today’s talk.

I’ll cover the core toon rendering topics in the blue frame and Komada-san will present our toon face shadow implementation in the green frame.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Volumetric Fog
Toon Face Shadow
Final Words

Okay, I’m going to start off with a brief introduction of our game.

Pop-Up Character Introductions

Who is this mystery man, and does he have a food name?

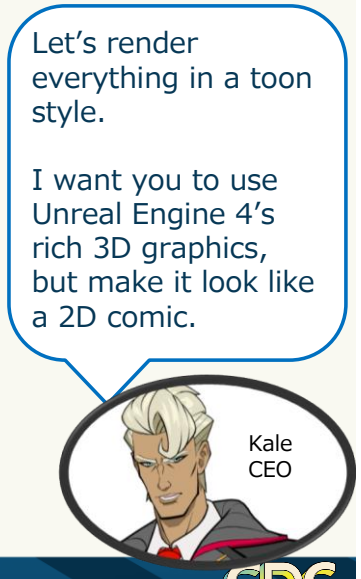
I'm sure not all of you have played our game, so I'd like to introduce our presentation's pop-up characters first.

The character on the left, Kale, is one of the enemy characters in the game. Since he's an evil guy, he'll mostly be making difficult demands for the graphics team.

The character in the middle, Chai, is the main protagonist of the game. Since he's our friendly hero, he'll mostly be making enthusiastic comments.

And finally, the robot on the right is John. Despite not having a catchy food name, John made a strong creative push for our toon visuals, had a conveniently available texture, and is Hi-Fi RUSH's creative director, so I've included his comments.

Hi-Fi RUSH Features (Toon Rendering)

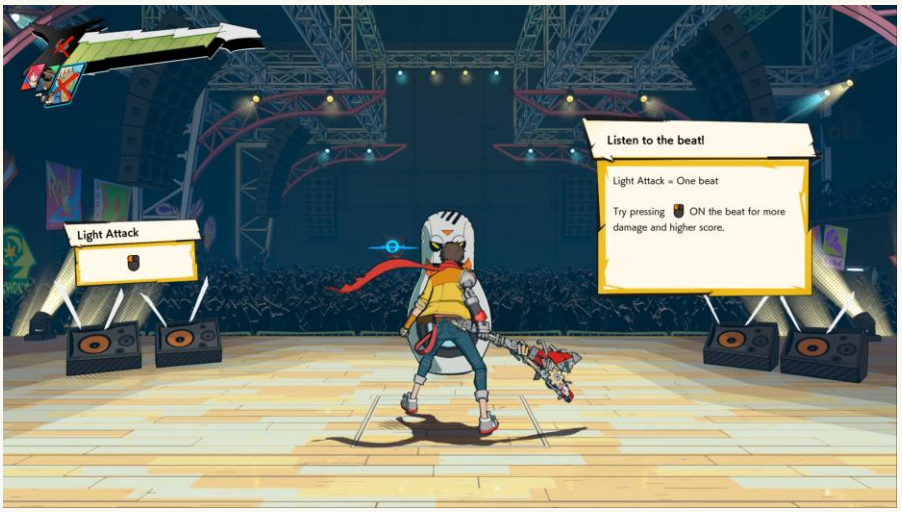


It's the main topic of our talk today, but Hi-Fi RUSH is a game utilizing toon rendering.

A lot games use toon rendering for characters, but not the environment. In our game everything is rendered in a toon style.

A big challenge for the graphics team was how to utilize Unreal Engine 4's modern graphics pipeline and make it work for a 2D cartoon look.

Hi-Fi RUSH Features (60FPS)



60FPS is a must. We're a rhythm action game.

BUT I want nice graphics as well.

Keywords are sharp, clean, colorful.

MARCH 18-22, 2024 #GDC2024

Another defining characteristic of our game is that we're a rhythm action game that requires a rock solid 60 FPS in order to minimize input latency and provide the best gameplay feel.

Programmers and artists were told from the very start of development, FPS is top priority.

But at the same time, we were going to support the new Gen9 hardware, and the team wanted to pursue great graphics that would do justice to the hardware.

With goals of great graphics, the keywords that our art director defined for the team were sharp, clean, and colorful.

Rock Solid 60 FPS & High Resolution

Xbox Series X 4K 60FPS
Xbox Series S 1440p 60FPS

Needs to run great on lower-end PCs.

Let's strive for native resolution but support super resolution plugins for lower end specs.

Everyone, I want our renderer and our assets need to be well optimized!

I'll go over the technical make up of our attempt at great graphics in more detail but notice that two of our key words were sharp and clean.

Image quality was very important to us, and in addition to high frame rate, one of the key technical features of our game is its high resolution.

We aimed for native resolution on our console targets for sharp, clean, "artifact-free" image quality. We're aware of super resolution tech, and we support super resolution plugins for PC, and they're great. Because our resolution is high to begin with, they allowed us to push the limits of our PC low end specs even further.

Balancing Visual Quality & Performance

Digital Foundry Review
- Adventurous Toon Rendering
- Solid 60FPS
- Xbox Series S 1440p
- Xbox Series X 4K

We did hit our visual quality & performance goals!

https://www.youtube.com/watch?v=8qppoWhanwk

Because we prioritized performance and image quality, a nagging fear during development was that people would be disappointed by our technology. In the end, it was a great win for us that the game's visual arts was well received, and we got great technical reviews. Balancing performance, resolution and rendering features is hard. As graphics programmers we're excited by new rendering tech and want great graphics, but our game emphasized performance and image quality, and the rendering features we chose for the game were carefully considered so we could hit our goals.

Agenda

Game Introduction
Deferred Toon Rendering
- ・Overview
- ・Post Process Lighting

Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

If a graphics programmer were to introduce our game that would be it.

I'll now discuss our toon rendering starting off with an explanation of the core tech; our deferred toon renderer.

Deferred Rendering

Hi-Fi RUSH uses Unreal Engine 4, which out of the box is a photorealistic deferred renderer and not a toon renderer.

The lighting is forced, but this is what Hi-Fi RUSH looks like when toon rendering features are disabled from the game.

The main point of the image is that, if we use the default UE4 lighting, there is gradation everywhere, and the image ends up looking too much like it’s rendered using a 3D engine.

Deferred Toon Rendering

This is the same scene rendered with toon rendering features re-enabled. This is what we were going for as a toon look. Shadow gradation is replaced with sharper shadows and our comic shader has been applied to the various 3D render passes adding a 2D touch to the image. This is what sharp, clean and colorful looks like.

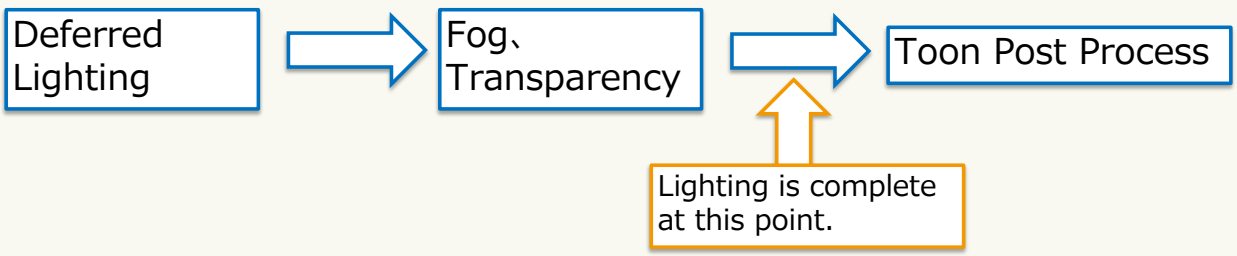
Deferred Toon Renderer 3D Features

- Toon Light
- Dynamic Shadow Map
- Static Shadow Map
- Capsule Shadow
- AO Volume Shadow
- Diffuse Global Illumination
- SSAO
- SSR
- Post Process Outline
- Post Process Toon Shading
- Volumetric Fog
- Camera Motion Blur
- Toon Motion Blur
- Tone Map
- Bloom
- Lens Flare
- Temporal AA

Compared to past games with strictly cel-shaded environments, we incorporate a lot of 3D lighting/rendering features into the game.

We added rendering passes like static shadow maps and decal toon lights, but we also stylized and extended the excellent base UE4 graphics features such as SSAO and SSR and made them work for cel-shading.

Toon Rendering (No Engine Modification)



It’s possible to use a standard UE4 post process material to implement a toon post process.

It’s possible to write a toon renderer without engine customization in UE4.

In the simplest case of using UE4 post process materials, the toon post process is applied after the scene lighting is finished and most of the rendering is complete.

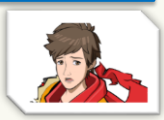
Toon Rendering (Engine Modification)

We're a small team, so we must be careful with engine modifications, but I think we can do better.

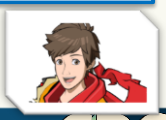
It's the simplest approach and we wanted to do better.

Toon Rendering (Lighting Layer Toon Stylization)

Apply a single toon stylization to the final rendering result.



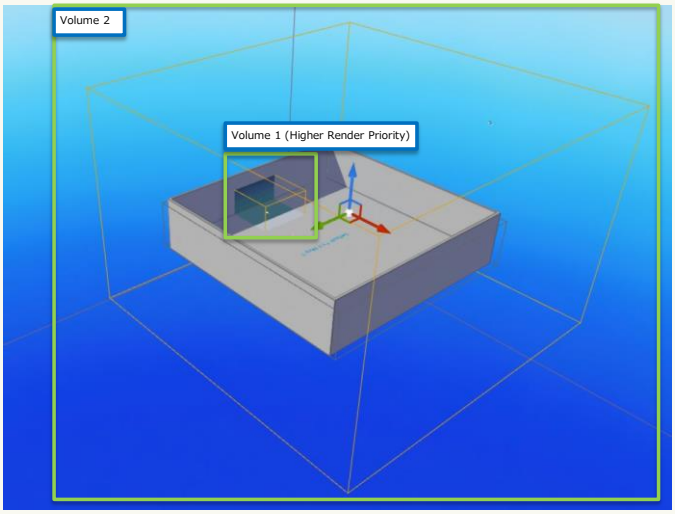
Apply individual toon stylizations to each layer of lighting.



GDC

Since we planned on supporting many lighting features, we wanted to be able to apply toon stylization to each lighting layer with their own stylization parameters.

Toon Rendering (Per-Volume Rendering)

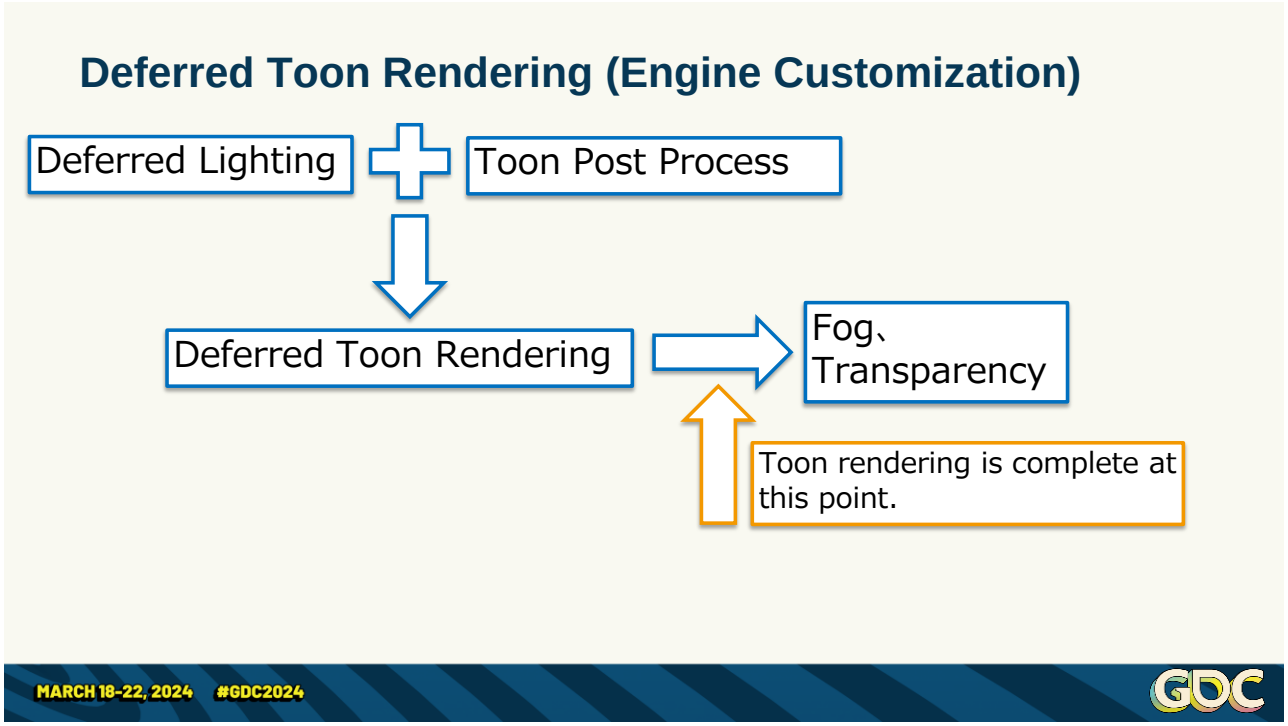


Toon post processing is applied per camera.

Toon post processing is applied per volume.

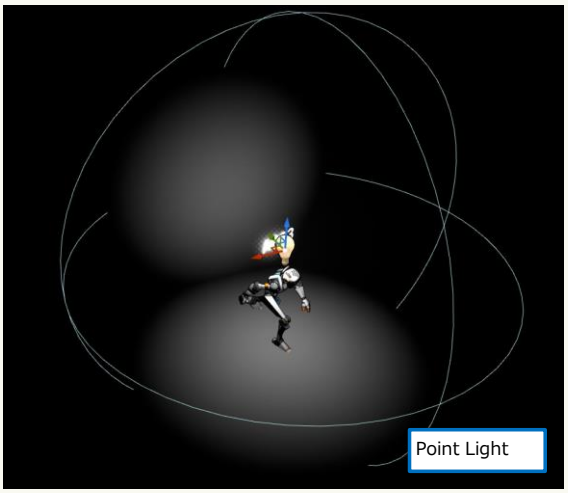
Another thing we wanted to improve on is that UE4 post process materials are applied per camera.

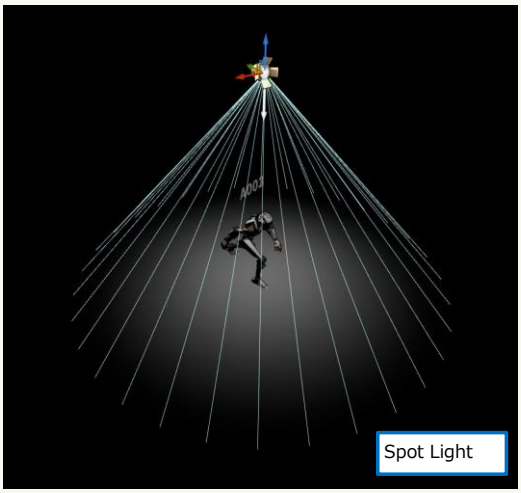
We felt this was too restrictive and wanted to apply different toon colors and stylizations to different areas within the same camera.



To achieve our goal, we customized unreal engine 4 and combined deferred lighting and toon post process into a single deferred toon rendering pass.

Regular Deferred Rendering

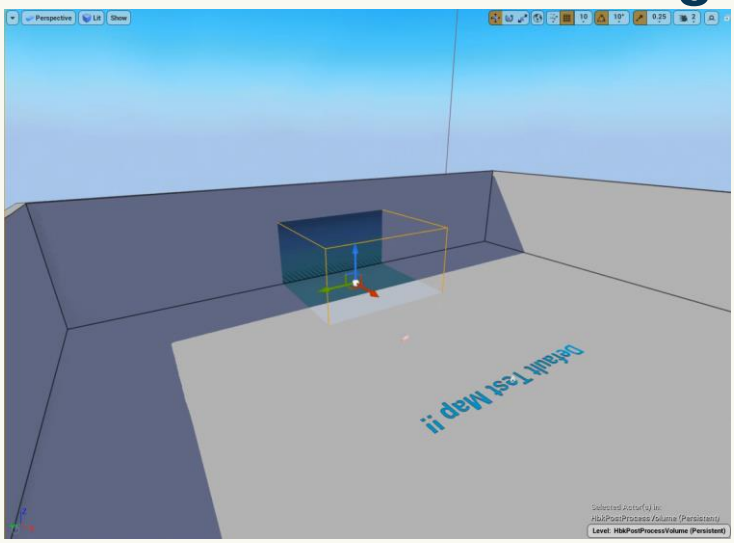




Decouple the lighting pass from the geometry pass.

Here's a quick review of regular deferred rendering. Material information are rendered into GBuffer render targets and the lighting pass is decoupled from the geometry pass. Point lights are rendered with sphere geometry and spot lights are rendered with cone geometry in a pass independent from the geometry they are lighting.

Deferred Toon Rendering



Similar to deferred lights except the geometry is a box and the shader is a toon post process.

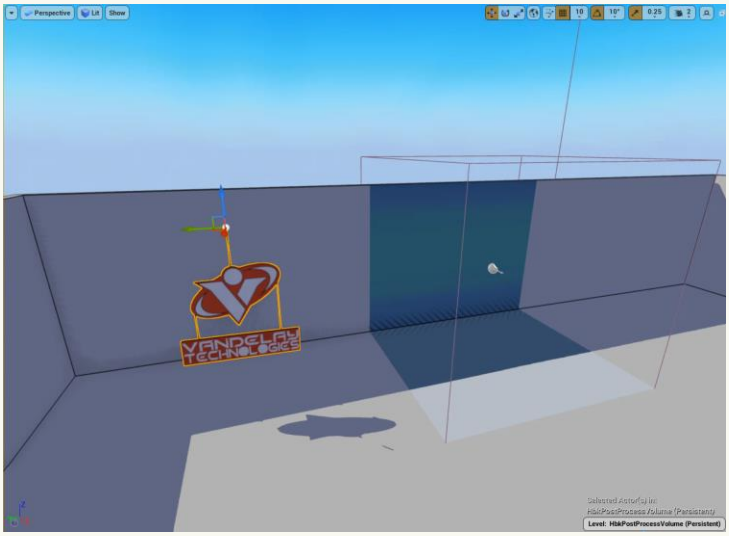
Decouple the toon rendering pass from the geometry pass.

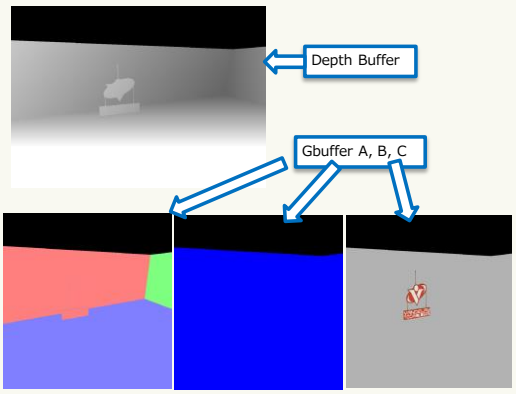
In a similar fashion, we want to decouple toon rendering into a pass separate from the mesh's actual geometry rendering.

Our toon post process volumes work like deferred lights except the shader is a post process.

Like deferred lights, our artists can freely move around the toon post process volume to locally change the toon rendering.

Deferred Toon Rendering (Box Volume Rendering)





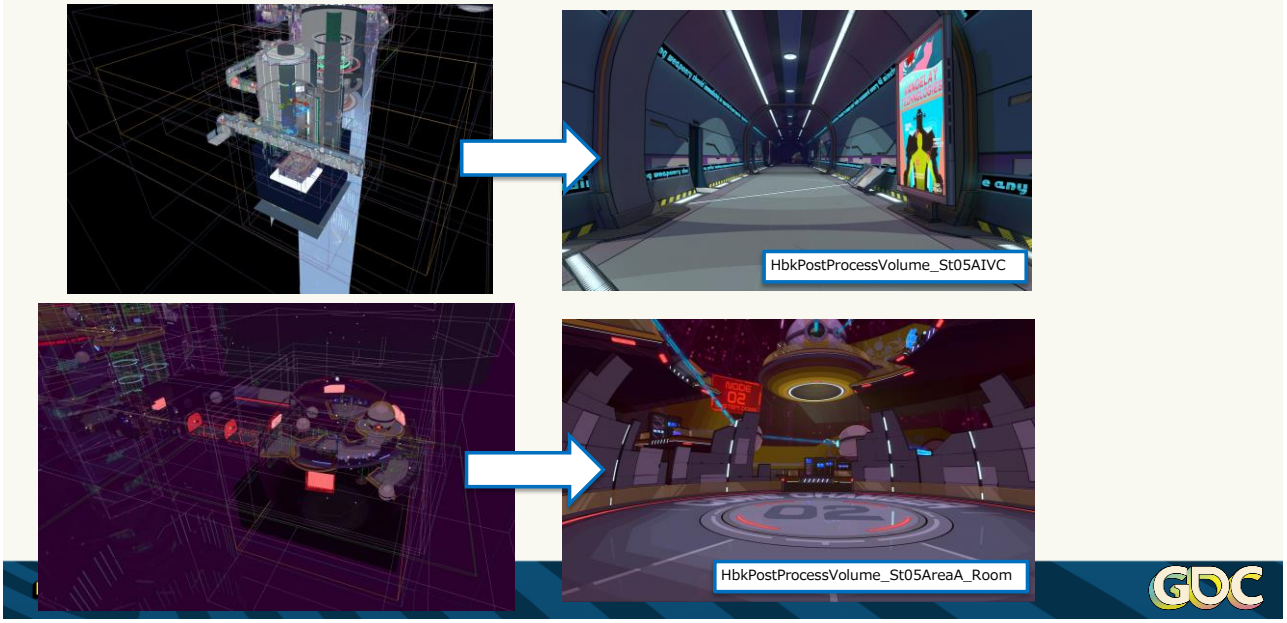
We use standard deferred rendering calculations.

We use the GBuffer and world depth information to apply toon post processing on the scene geometry enclosed by the box.

In the slide video, when I move the billboard mesh inside the smaller toon post process volume, notice that its toon rendering changes.

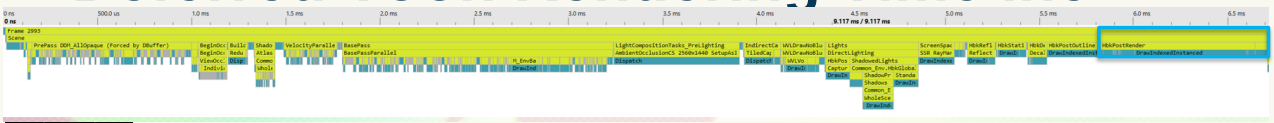
At its core, our toon post process volumes are using standard deferred rendering calculations. We use the scene depth to recreate the world position and use GBuffer information to apply the volume's toon post processing on not the volume box geometry, but the underlying scene geometry.

Lighting With Toon Post Process Volumes



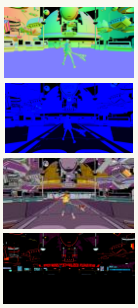
Our environment artists place many toon process box volumes throughout our levels. By placing these volumes, our artists adjust toon shading locally per volume within each area of the level.

Deferred Toon Rendering Timeline



Depth Buffer

Shadow Depth



GBufferA,B,C

SSAO, Capsule Shadows

AOVolume

Global Illumination

Dynamic Screen Space Shadows

Static Shadow Map

SSR

Decal Light

Post Process Outlines

Shadow color volume and ambient cubemap volume passes explained later are part of the toon process pass shader, so no render targets here.

*Custom render passes are in the yellow frame.

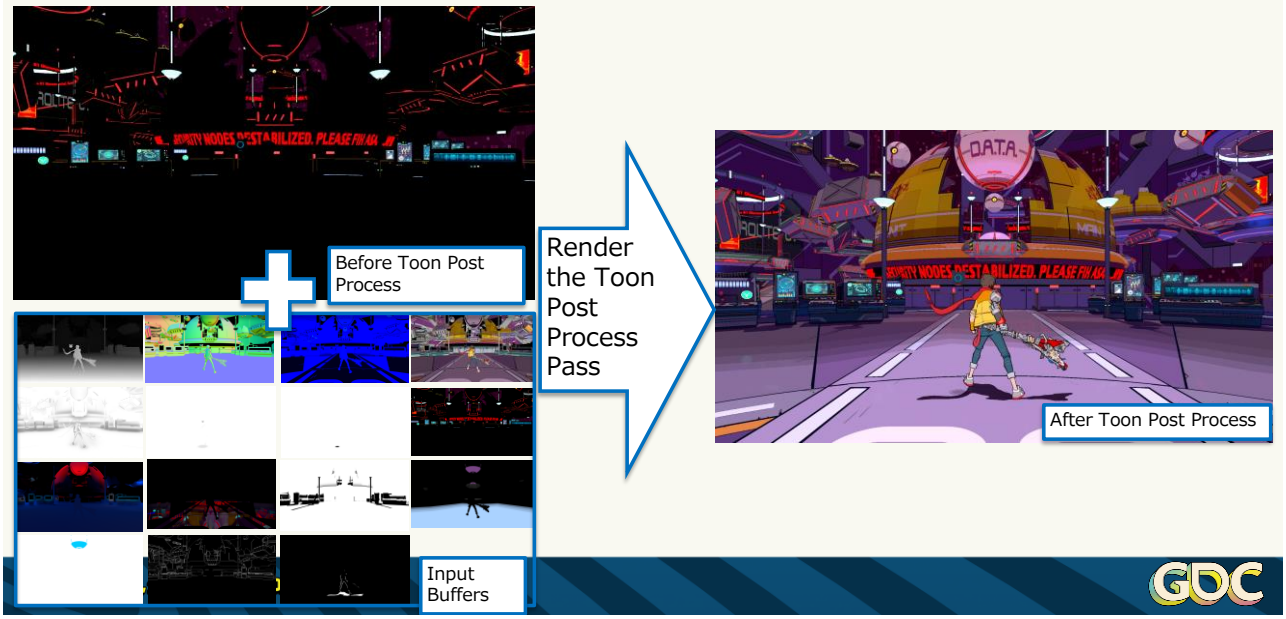
The above is a Pix for Windows capture of the timeline for our deferred toon rendering pipeline.

I’m showing you the render passes that are executed in an actual scene in preparation for the toon post process pass, the render pass of the box volumes in the previous slides.

Deferred Toon Rendering Input Buffer

Quite a few render targets end up getting generated for the toon post process pass. My earlier test map was very simple looking, but there are quite a bit more lighting layers in an actual scene. Toon shaded environment tend to look simplified, but there are a lot of layers to our environmental lighting.

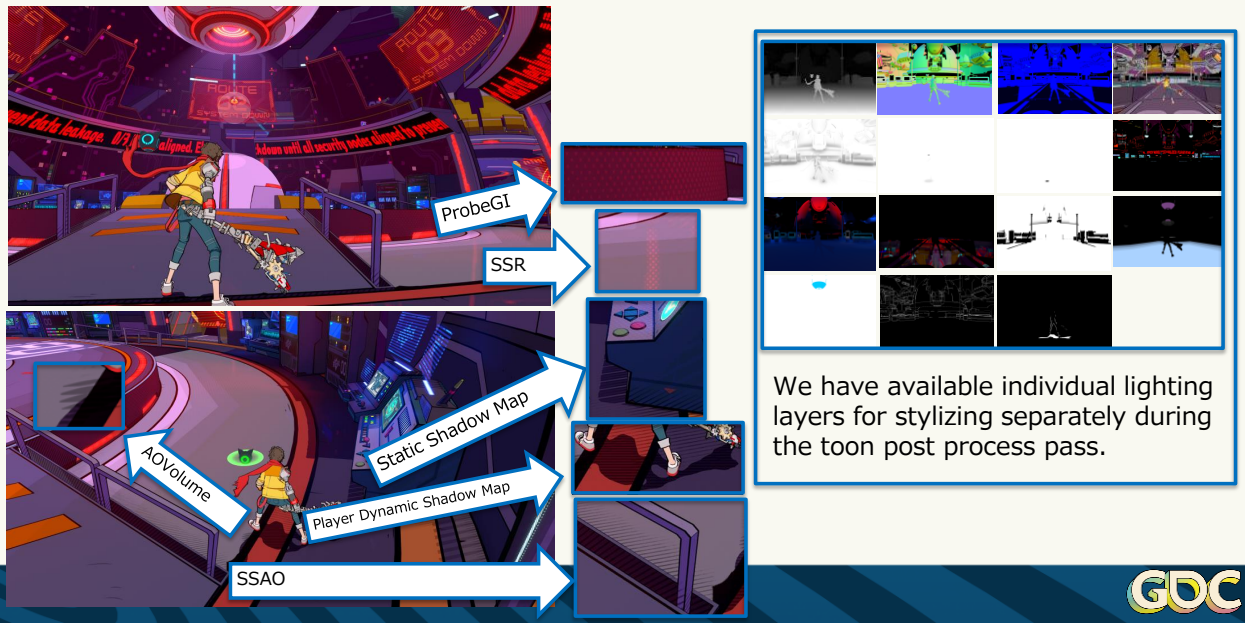
The Toon Post Process Pass



Because our deferred toon rendering is a combination of toon lighting and toon post process, notice that the scene color is completely dark except for emissives before the toon postprocess rendering is applied.

A regular deferred renderer has scene depth and GBuffers. For deferred toon rendering, we have scene depth、GBuffers plus the various individual lighting pass results all waiting to be applied.

What We Do With Our Numerous Input Render Targets (Toon Stylization)



The toon post process has available to it all these render target. What do we do with them?

Because we have all lighting results as input textures, as was one of our initial goals, we can apply comic stylizations to each lighting layer with individual parameters.

What We Do With Our Numerous Input Render Targets (Shadow Overlap)

We have a lot of shadow types.

- parallel light shadows,
- shadow-only light shadows
- static shadow maps
- capsule volume shadows
- ao volume shadows

Player shadow drawn inside cascaded shadow maps.

Because we apply all shadow types to the scene inside the toon post process pass, we can also easily control the visual look when different shadow types overlap. For example, when the player shadow is drawn inside our cascaded shadow maps, we can detect this overlap and adjust the player shadow darkness, so it doesn't appear too dark or light. Stylized shadows inside shadows is a very cool toon thing and we wanted to make it look good.

Making Our Toon World Look Good With A 3D Engine

Deferred Toon Renderer 3D Features

- Toon Light
- Dynamic Shadow Map
- Static Shadow Map
- Capsule Shadow
- AO Volume Shadow
- Diffuse Global Illumination
- SSAO
- SSR
- Post Process Outline
- Post Process Toon Shading
- Volumetric Fog
- Camera Motion Blur
- Toon Motion Blur
- Tone Map
- Bloom
- Lens Flare
- Temporal AA

GDC

We're using UE4. Let's use its graphical prowess for toon.

MARCH 18-22, 2024 #GDC2024

GDC

UE4 has a great-looking deferred rendering implementation, and it was a natural progression for us to extend it to a deferred toon renderer. We tried to utilize UE4's strengths but at the same time didn't want to lose our own visual personality by using the engine too directly.

Locally toon shading different areas by placing toon post process volumes like deferred lights seemed like good useability for our artists.

Deferring all lighting and toon post processes into a single pass made sense because we wanted our artists to be able to control how different lighting layers combine and interact for toon.

Agenda

Game Introduction
Deferred Toon Rendering
- ・Overview
- ・Post Process Lighting

Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

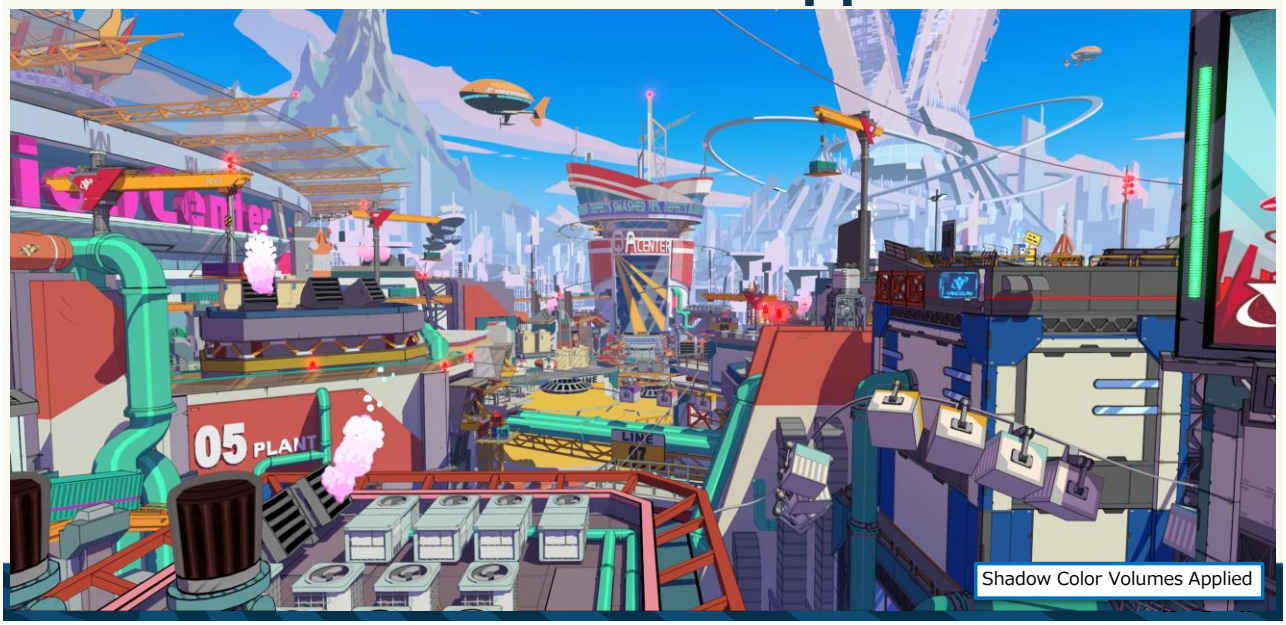
That was an overview of our deferred toon rendering. Next, I'd like to highlight unique aspects of our toon lighting during deferred toon rendering.

I'll cover shadow color 3D textures and ambient cubemaps. The major requirement for our toon lighting was artists need to be able to add color and adjust brightness, but it can't look too 3D. To make the scene not look artificial for toon, artists needed to be able to paint the colors themselves.

Environmental Shadow Color

Environment artists need a way to give color to each location. Our art director’s key words were sharp, clean and colorful, so it was important that artists be able to control the environment’s shadows with the colors they wanted. In the slide image, the shadow color is a uniform black color.

Shadow Color Volume Applied



This is what the scene looks like with shadow color volumes applied.

The black shadows are now properly colored with artist authored shadow colors.

I'll explain in more detail in the following slides, but shadow color volumes work by mapping a 3D texture to a 3D world position.

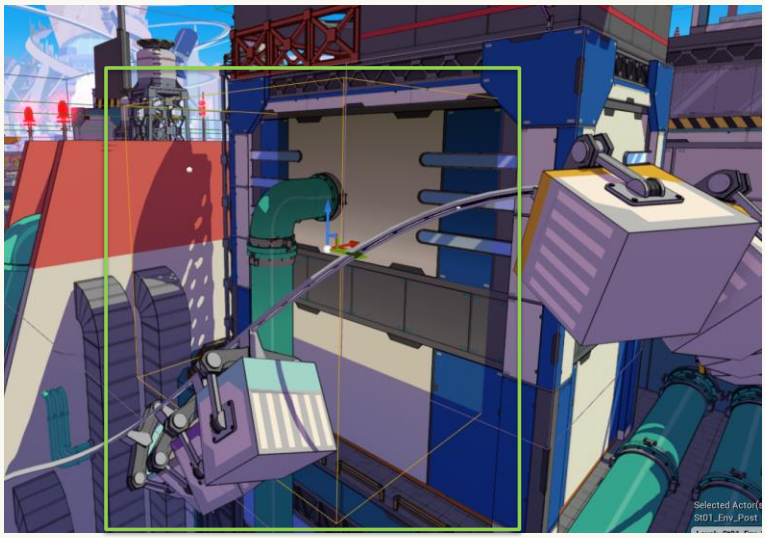
Shadow Color Volume Rendering

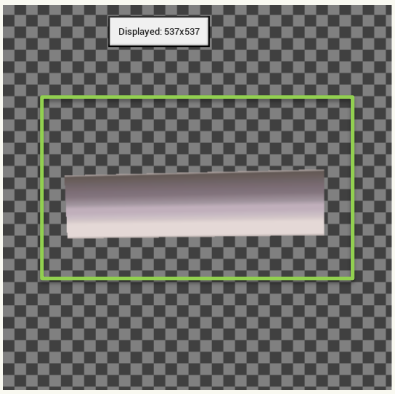
Here in the slide video, I'm adding a new local shadow color volume to the scene with a whiter/greyer shadow color 3d texture.

The rendering of the shadow color volumes is like our toon post process volume rendering.

The shadows are not applied to the volume box geometry but are projected onto the geometry enclosed by the box.

3D Texture Mapped to World Positions

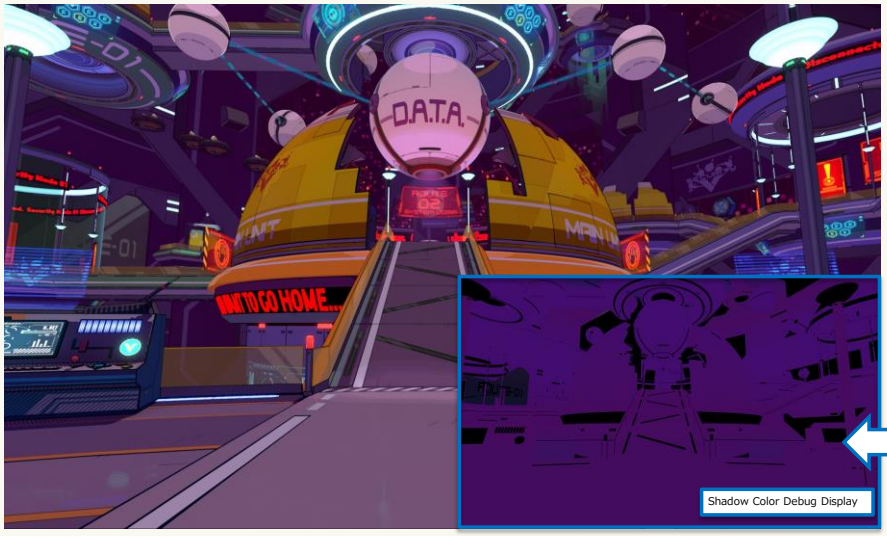


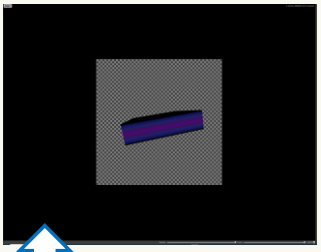


Surface world position converted to normalized local coordinates in the [0,0]-[1,1] range and used as texture coordinates for the 3D texture.

If we look carefully, the shadow color volume colors gradually become darker with increasing height. This positional color change happens because the enclosed surface world position is converted into the shadow color volume's normalized local 3D coordinates and this coordinate is used to the sample the 3d texture. The gradually changing colors inside the 3d textures are hand adjusted by artists to give the desired shadow color nuance our art director wants for the scene.

Shadow Color Volume Applied





The scene's shadow color 3D texture.

Interior scenes are usually initially unlit, making it possible to adjust the overall color make up using the shadow color volumes.

Here's an interior scene and its corresponding shadow color 3d texture.

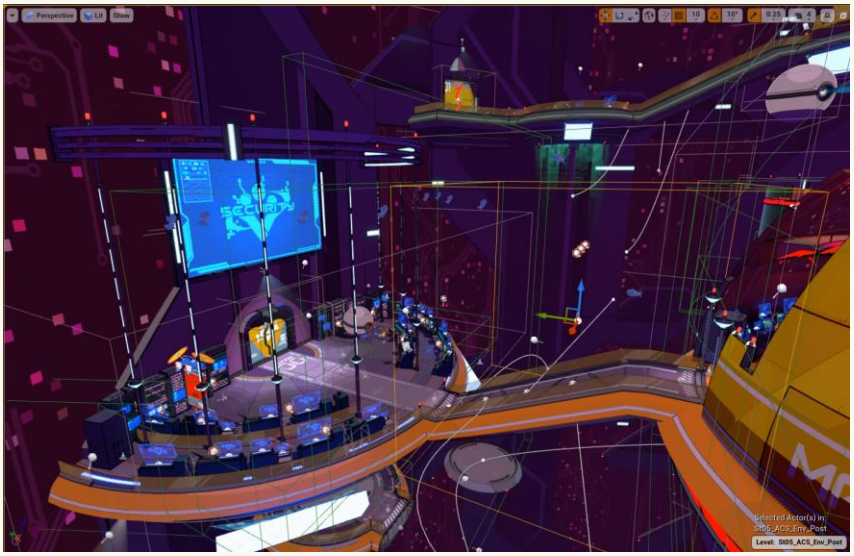
For interior scenes, our environment is usually initially unlit and in shadow and we light by adding lighting layers to this state.

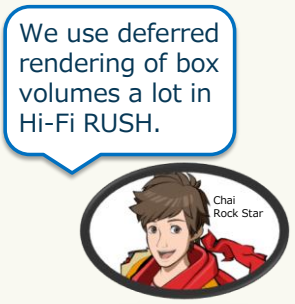
Because the scene is initially in shadows, artists can use shadow color volumes to adjust the overall base color make up of the scene.

Ambient Cubemap Volume

In addition to 3d textures for shadow colors, we also use cubemaps for postprocess ambient brightness adjustments.

Ambient Cubemap Volume

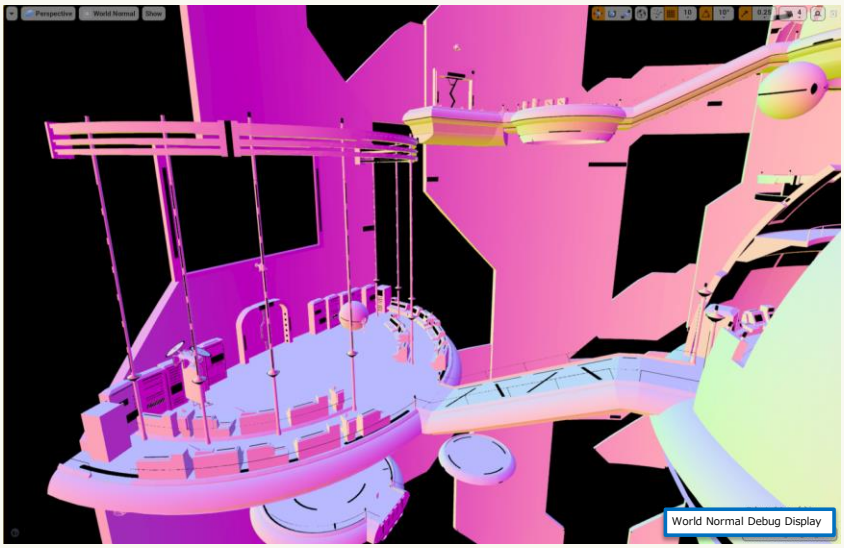




Like with our shadow color volumes, we apply ambient cubemaps to the underlying geometry using a box volume.

As mentioned earlier, our interior scenes are initially unlit, and to this state we add lighting layers, one of which is ambient cubemaps.

Ambient Cubemap Volume



The GBuffer world normal is used to sample the cubemap texture.

Cubemaps need a sampling direction. Because we are a deferred toon renderer, we can use the world normal stored in the GBuffer for this purpose.

Ambient Cubemap Volume On/Off Difference

In the slide image, the ambient cubemap volume is being toggled on and off.

Notice that in the scene, the underside of the surface is lit brighter than the top and the sides, representing a difference in directional ambient brightness. This directional brightness comes from the cubemap texture, which our environment artists create by hand.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

In the next section, I'd like to talk about our comic shaders.

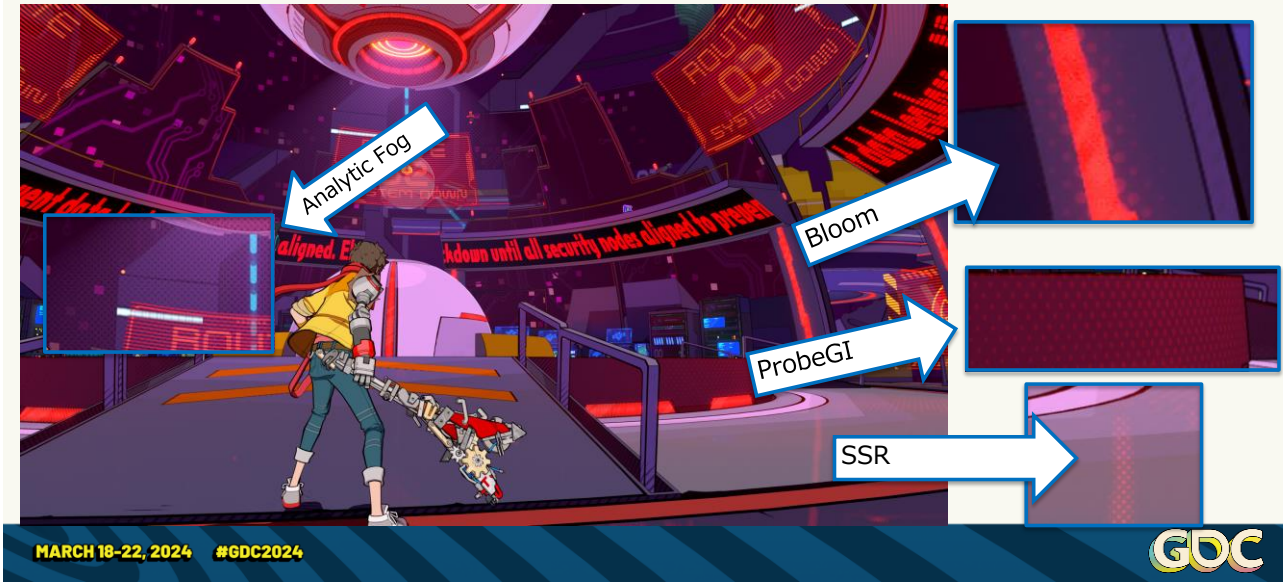
Comic Shader

Lit
- Halftone Dot

Shadow
- Hatching Lines

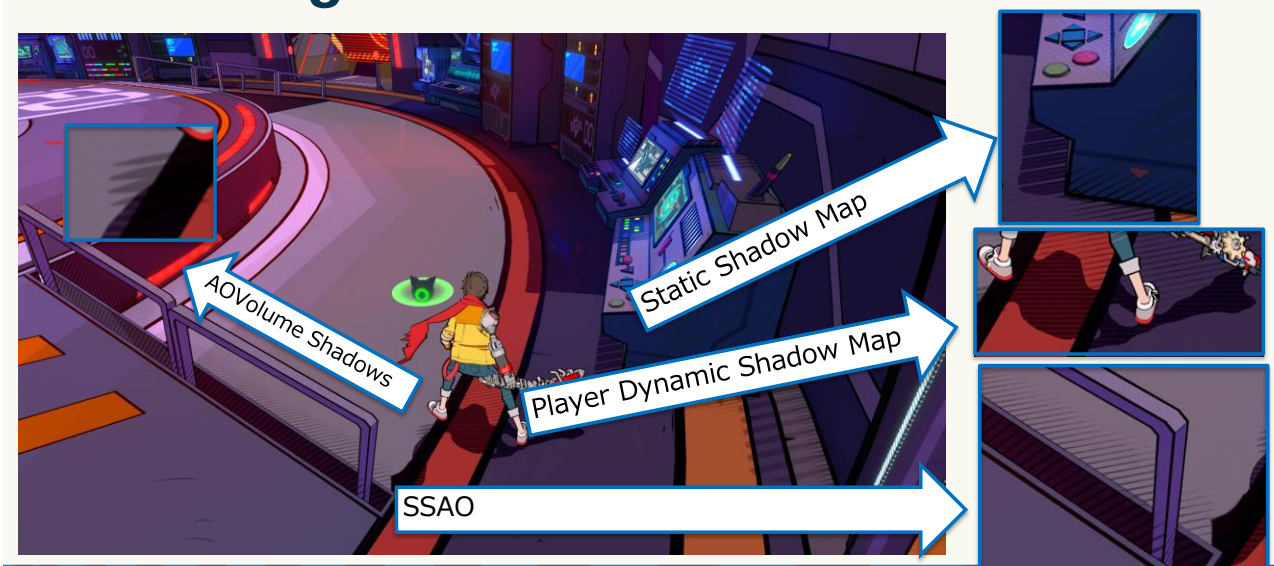
It’s a recognizable trait of our toon look, but we implemented a comic shader for rendering halftone dots and hatching lines.

Halftone Dots



Halftone dots are the round dots that we apply to the lit portions of our lighting.

Hatching Lines



Hatching lines are the rotated lines we apply to the dark shadowed areas of lighting.

Why Apply A Comic Shader

Our goal is to render a 3D world as 2D toon.

Our comic shader removes gradation, simplifies colors and makes the rendering look more 2D.

The goal of our toon renderer is to render a 3D world as 2D toon.

Our comic shader is a stylization, but at the same time it's an important element in our deferred toon rendering for making the world look less 3D and more 2D toon.

Comic Shader Off

Here’s what a scene from our game looks like with the comic shader turned off.

SSAO, GI, Bloom make the world look richer, but since we’re just applying 3D engine functionality, the final image looks 3D.

Also, the performance-oriented, low-sample count nature of some of our rendering passes become especially noticeable on our clean toon textures.

Comic Shader On

Here’s what the same scene looks like with the comic shader enabled.

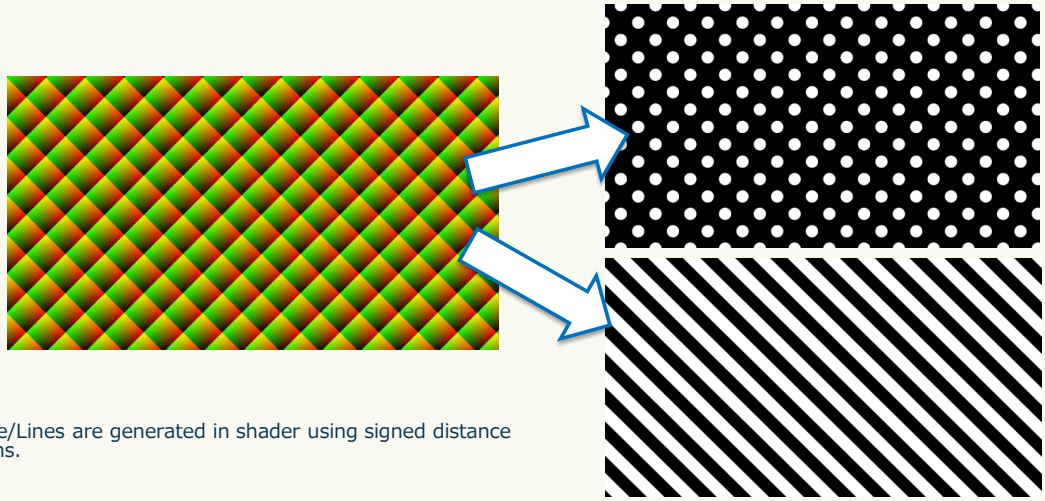
The scene is stylized, but at the same time, gradation is also removed, making everything look less 3D.

The dirty look of GI and SSAO have been made sharper with lines and dots, and the image looks much cleaner.

Comic Shader On/Off

Here’s the comic shader on/off as an animation.

Signed Distance Function

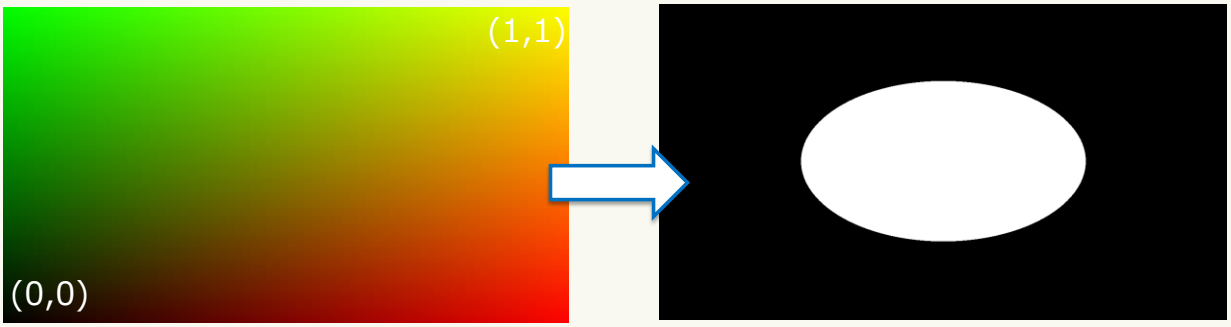


Halftone/Lines are generated in shader using signed distance functions.

2D signed distance functions are often used in procedural shaders.

For our comic shaders, programmers used SDFs to create the comic shader halftone and hatching lines procedurally inside our shaders.

Implementing Halftone (First Step)



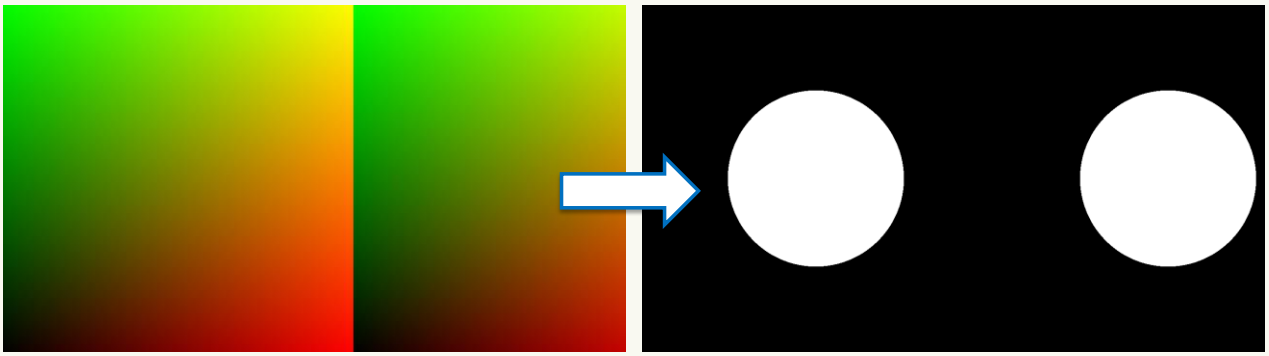

```
float3 CalcHalftone(float2 UV, float3 HalftoneColor, float3 BGColor, float Radius)
{
    float2 UV2 = 2.0*UV.xy - 1.0;
    float Dist = length(UV2);
    if(Dist < Radius) { return HalftoneColor; } // Radius is 0.5.Halftone Color is white.
    return BGColor; // BGColor is black.
```


We just do basic stuff in terms of SDF. For clarity, I'll go over an example halftone shader.

My example shader is a postprocess which uses screen space UVs.

The first step is to calculate the distance from the center of the screen. If this distance is below the halftone radius, we draw a white color, if the distance is above the halftone radius, we draw a black color.

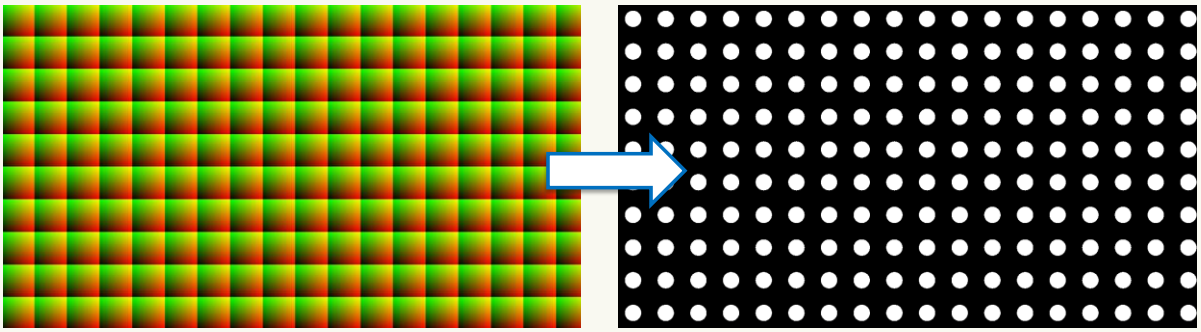
Implementing Halftone (Aspect Ratio)



UV.x = UV.x * (Res.x/Res.y); //Take into consideration aspect ratio

We take into consideration the aspect ratio to go from an ellipse to a circle.

Implementing Halftone (Repeating Grid)

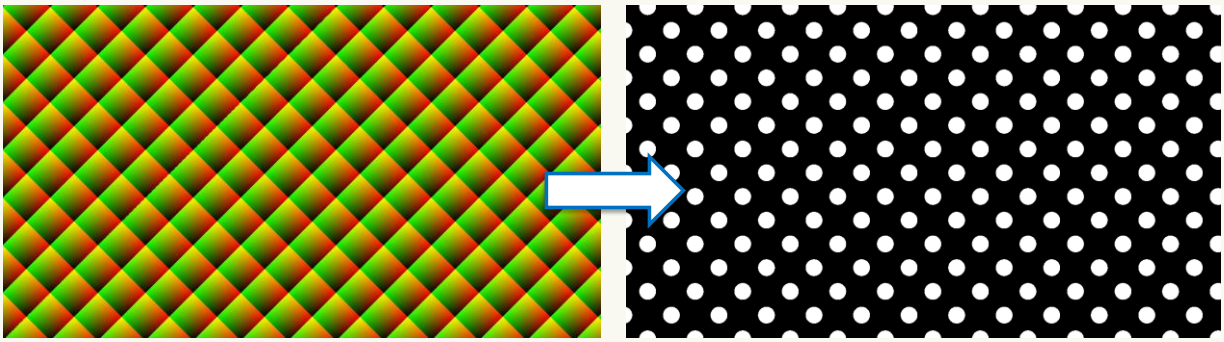


```
float3 CalcHalftone(float2 UV, float3 HalftoneColor, float3 BGColor, float Radius, float Freq)
{
    float2 UV2 = 2.0*frac(UV.xy*Freq)-1.0;
    float Dist = length(UV2);
    if(Dist < Radius) { return HalftoneColor; } // Radius is 0.5.Halftone Color is white.
    return BGColor; // BGColor is black.
}
```


We don’t want large circular dots, but want multiple smaller dots in a grid across the screen.

Using the frac instruction, we scale and wrap the screenspace UVs.

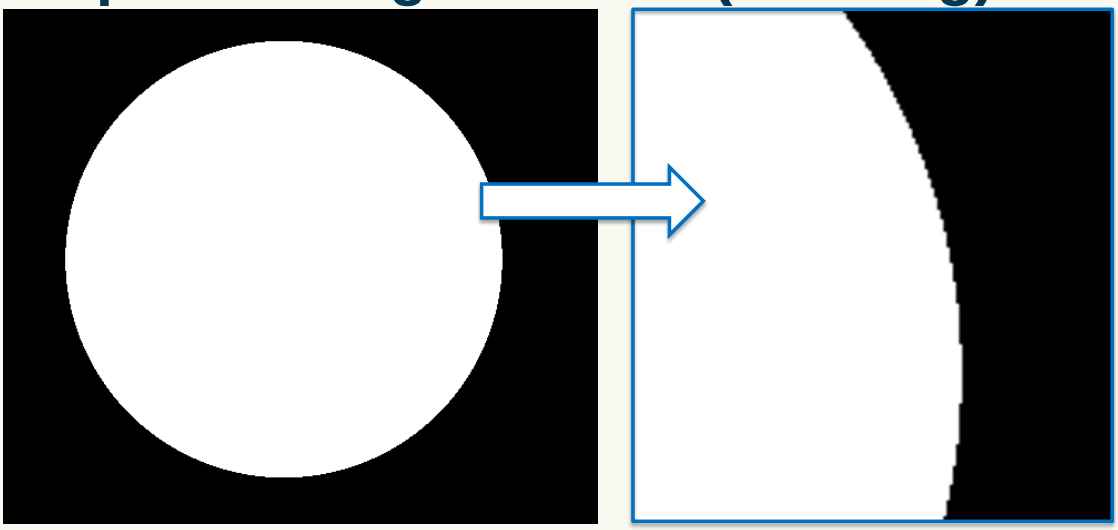
Implement Halftone (45 Degree Rotation)



Art requested that the halftones be rotated 45 degrees.

Art wanted the halftones rotated 45 degrees. For this we just rotate the grid UV.

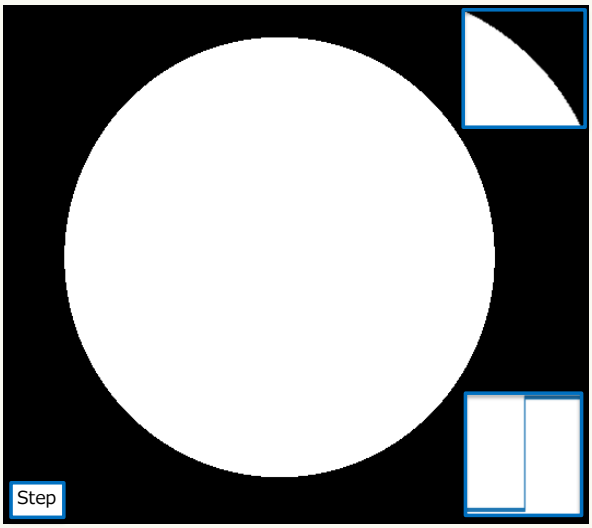
Implementing Halftone (Aliasing)

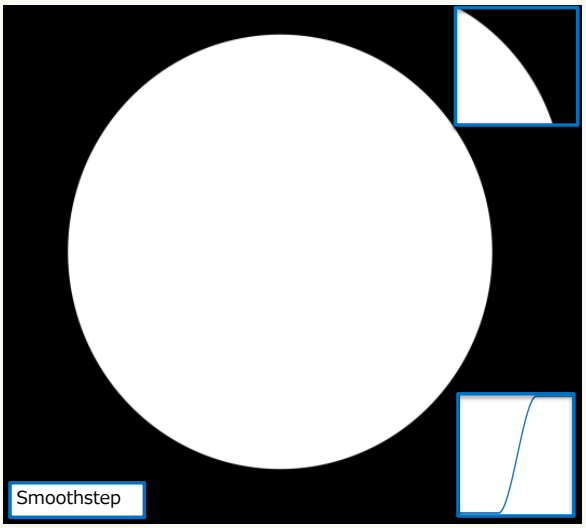


Aliasing can occur due to the simple 0/1 thresholding.

Because we are using a simple 0/1 step function for the distance function threshold, aliasing can occur.

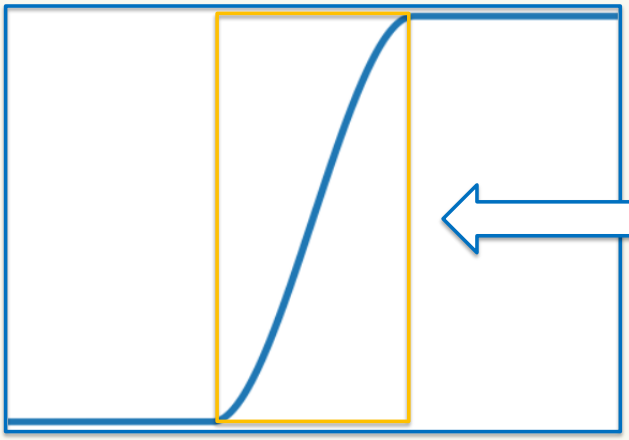
Implementing Halftone (Anti-Aliasing)





By using smoothstep instead of step and alpha blending the boundary region of the halftone dots, we can implement anti-aliasing.

Implementing Halftone (Smoothstep Width)



Where do we want to apply the antialiasing?

We want to apply anti-aliasing on threshold border pixel.

smoothstep(Radius-Width, Radius+Width, Dist)

The yellow frame area is 2*width.

How to calculate the width?

Use ddx, ddy instructions to calculate the per-pixel distance gradient.

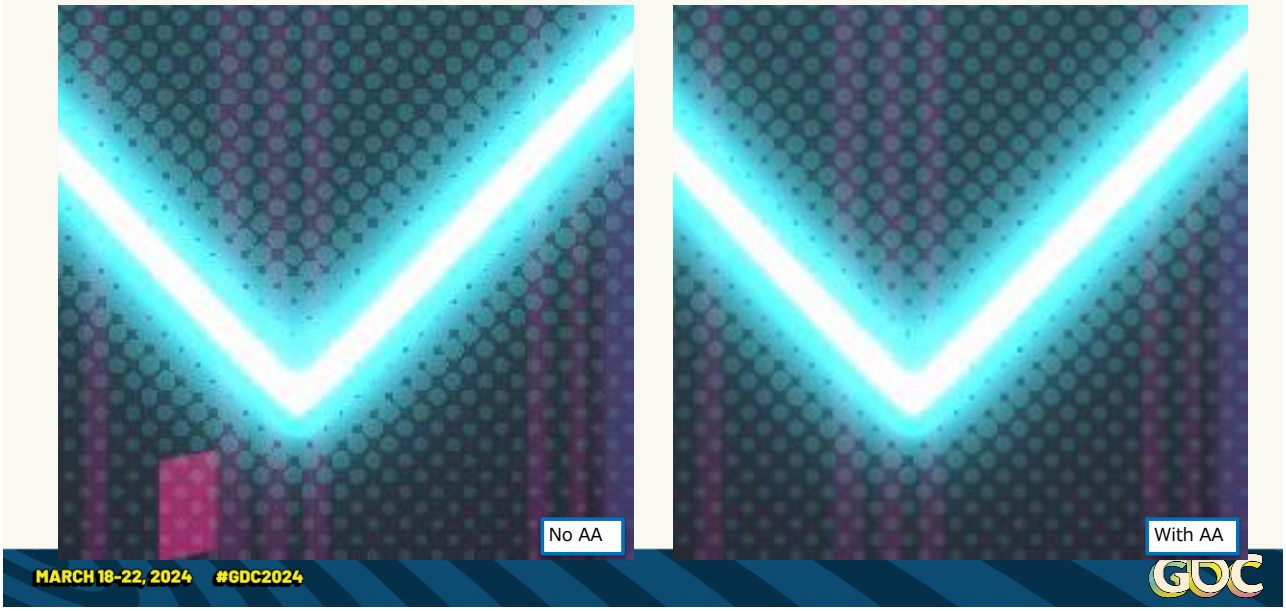
Stefan Gustavson. 2012. “Procedural Textures in GLSL”. Linköping University Electronic Press (OpenGL Insights, Chapter 7).

Reference

The area inside the yellow frame is the width of the smoothstep function where AA is applied.

Since we want to apply anti-aliasing only on the threshold border pixel, we use the ddx, ddy instruction to calculate the per-pixel rate of change of the distance function and use this value as the basis for our width.

Bloom Halftone AA On/Off In-Game (Zoomed)

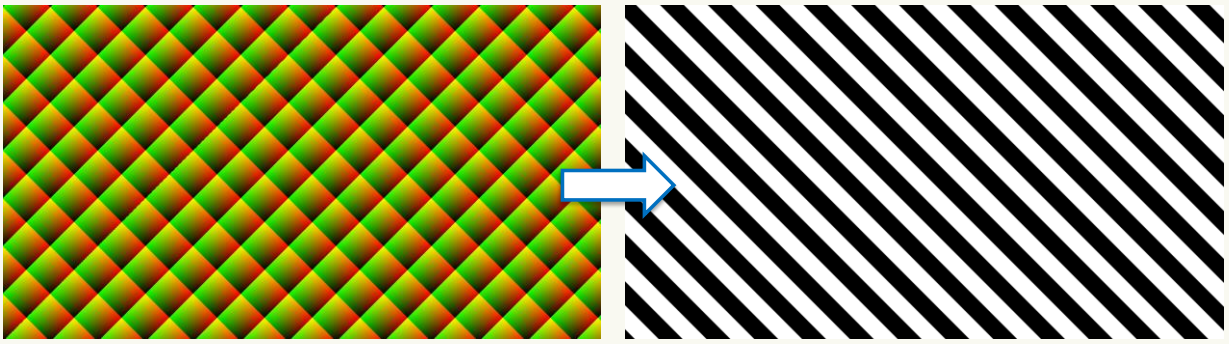


UE4's TemporalAA blurs the halftones, so the AA is only obviously noticeable on halftones applied after TAA such as our bloom halftones.

The difference also becomes not very noticeable with camera movement, so the benefits of AA is somewhat limited for us, but we kept the calculations for the moments that the user might notice the difference.

*Image sharpness was lost during the powerpoint to pdf conversion, so it may be hard to see a difference in the pdf file here.

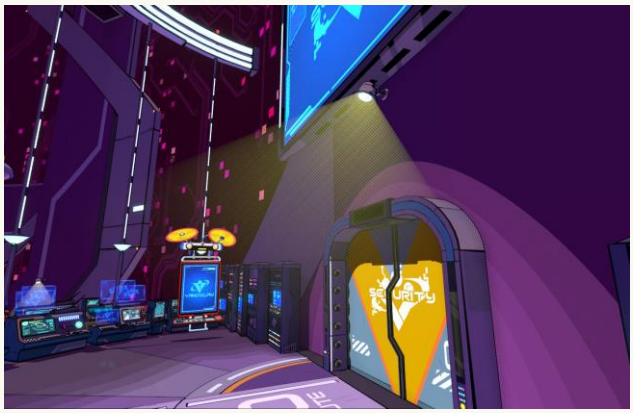
Implementing Hatching Lines



```
float3 CalcLines(float2 UV, float3 LinesColor, float3 BGColor, float Radius, float Freq)
{
    float2 UV2 = 2.0*frac(UV.xy*Freq)-1.0;
    UV2.y = 0.0;
    float Dist = length(UV2);
    return lerp(LinesColor, BGColor, StepWithAA(Radius, Dist));
}
```


So that was an explanation of how halftones are rendered using screen space UVs. Hatching lines are the same shader code, just with UV.y fixed to 0.

SDF UV Grid Generation (Screen Space)



Screen Space UVs

Screen Space Halftone Using Screen Space UVs

For effects such as bloom and volumetric fog, we use screen UV coordinates in actual scenes.

We don’t do anything complicated, and the example shader is close to how we render actual in-game screen space halftones.

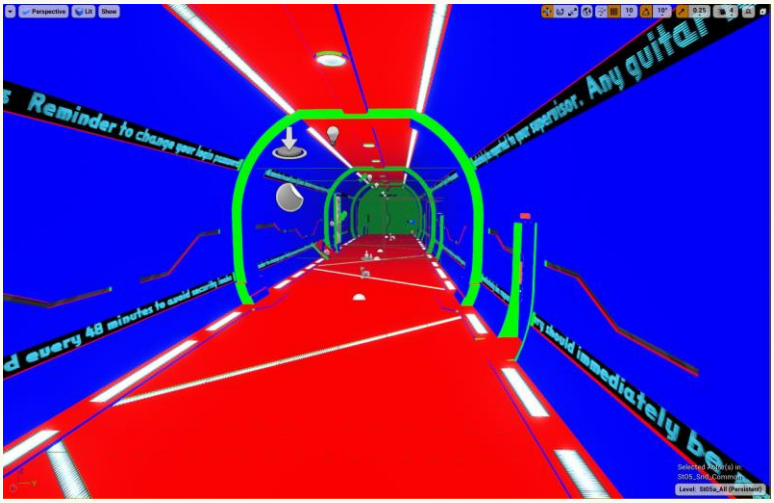
SDF UV Grid Generation (World Space)

・GI halftones on the wall are using world space UVs.

・Bloom halftones on the floor lights are using screen space UVs.

Screen space UVs work well for stuff without depth, such as bloom and volumetric fog, but on actual polygonal surfaces, we want our halftones and lines to look as though they are printed on the surface and not attached to the front of the camera. For these cases, we use world space UVs.

SDF UV Grid Generation (World Space)



Use the Gbuffer world normal to map the plane on which to generate the world space UVs.

Only 3 planes, XY, YZ, XZ.

Z-axis is dominant in the RED plane.
Use XY Plane. UV.xy is WorldPos.xy.

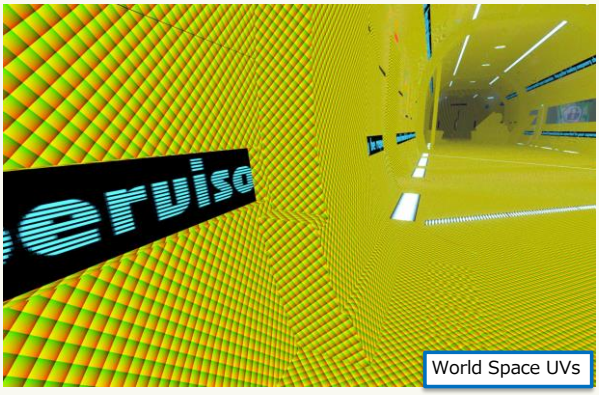
X-axis is dominant in the GREEN plane.
Use YZ Plane. UV.xy is WorldPos.yz.

Y-axis is dominant in the BLUE plane.
Use XZ Plane. UV.xy is WorldPos.xz.

To calculate world space UVs, we use the GBuffer world space normal to calculate the dominant axis of the surface and just project onto this axis's plane.

For example, if the z-value is the largest direction of the world normal, we want to project onto the xy plane, and so we simply use a scaled WorldPosition.xy as our UV coordinates.

SDF UV Grid Generation (World Space)



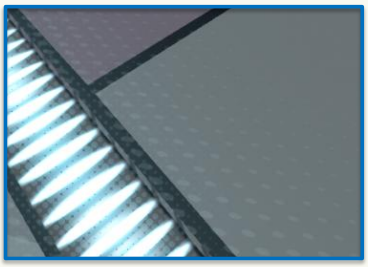
For intersections of planes, we don’t use triplanar mapping.

For the intersections of planes, we don’t use triplanar mapping. We experimented with it, but blending artifacts were noticeable and we simply project onto a single plane. I didn’t get complaints about the halftones and lines cutting off at the plane intersections, and the final implementation was kept simple.

In-Game Comic Shader (Gradation)

・We adjust halftone, hatching size based on luminance.

・It's a 2D toon expression of gradation.



One final important feature of our comic shader.

*Start animation.

Because we procedurally generate our halftones and lines, we can easily adjust their look based on the scene lighting conditions.

We use scene luminance to adjust halftone and hatching line sizes.

For the image in the slide, the GI halftones become larger the closer to the emissive light source and smaller the farther away.

It's a simple technique that allows us to express the lighting's distance attenuation while avoiding using gradation, which makes our scenes look more computer generated.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

In the next section, I'd like to talk about our toon lights.

Key Light (Lit/Shade)

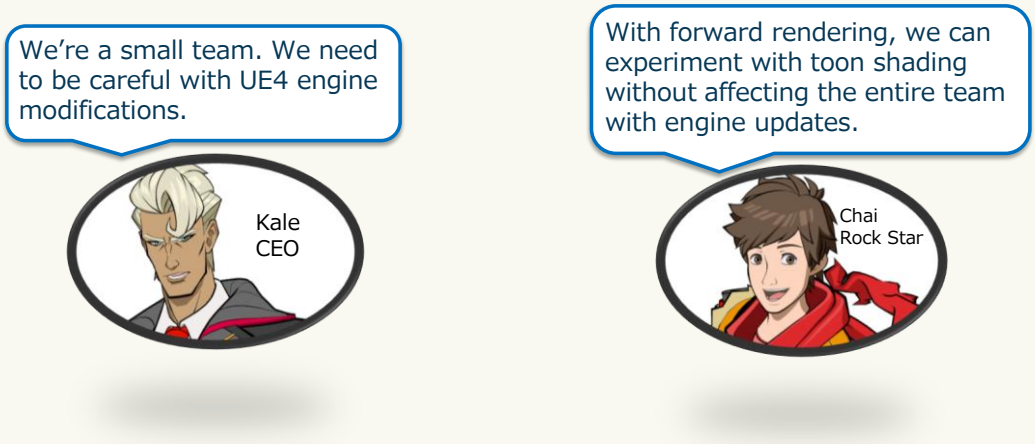
・Hi-Fi Rush uses a simple Lit/Shade 2-tone toon shading.

・A global key light is used to determine the direction of the lit/shade toon shading using forward rendering.

Hi-Fi RUSH's toon shading uses a simple 2-tone lit/shade toon shading model.

The toon shading's lit/shade calculation is not performed in a deferred pass but is calculated by a global key light that is forward rendered in UE4's base pass.

Why Forward Render Key Lights?



We chose to forward render our key lights because we could do so without UE4 engine modifications.

In later stages of development, we began utilizing UE4 render commands for data passing and render thread delegates for render pass callbacks that would allow us to implement original render passes game-side. We're now more experienced about how to approach UE4 engine modifications, but that realization came after our core tech was implemented.

The Case for Deferred Rendering Key Lights

A single keylight struggles when different lighting conditions occur within a single camera frame.

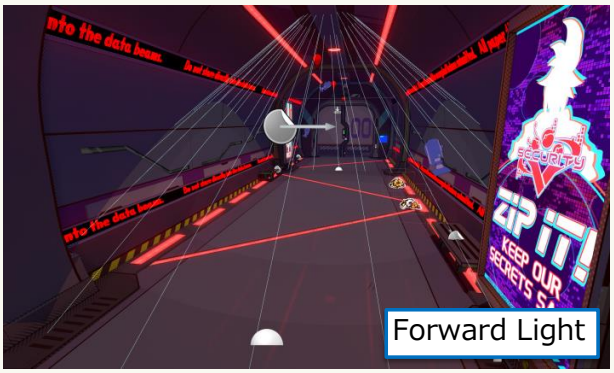
Two separate key lights are used here. One for indoor and another for outdoor.

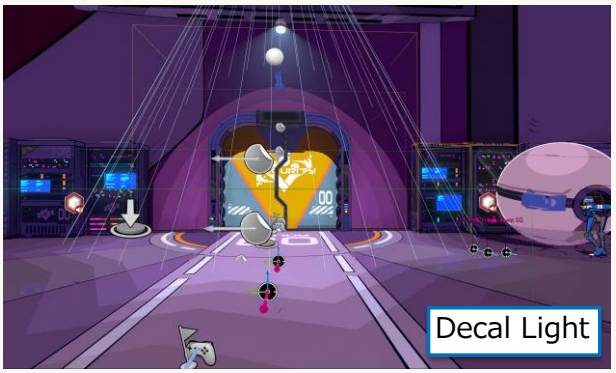
Supporting multiple key lights add complexity to our base pass.

Deferred key lights would have been more efficient for multiple key lights.

We eventually needed to support 2 key lights and it may have been more performant to associate the key light with a toon post process volume and render it in a later deferred pass. Our lighting's mix of deferred and forward is probably something we can improve on.

Placeable Toon Lights

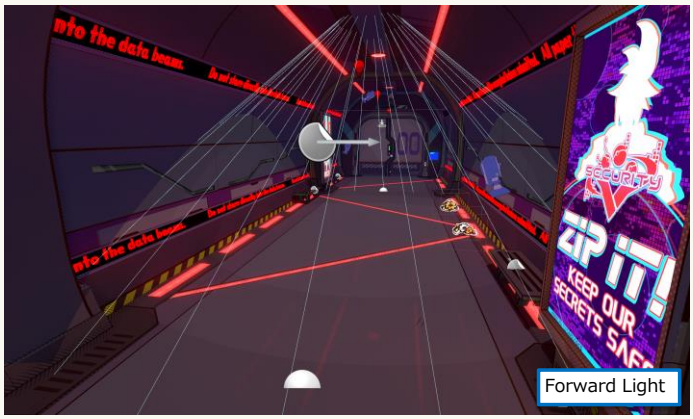


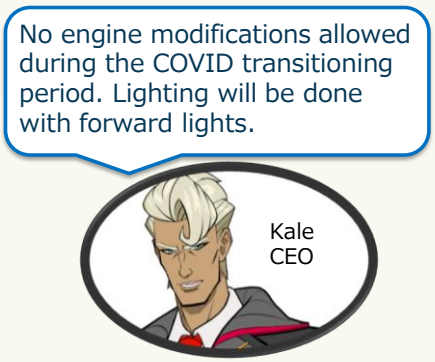


Light actors such as forward lights and decal lights can be placed to perform lighting.

Besides key lights, artists can use familiar light actors such as point lights and spotlights to perform local environmental lighting.

Toon Light (Forward Light)





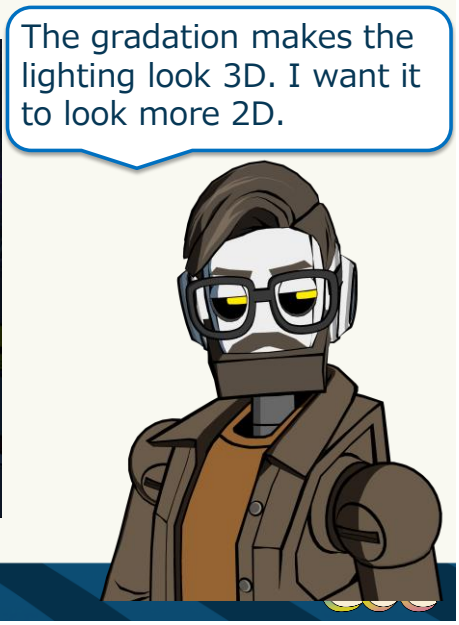
・Forward lights can be implemented without engine modification.
・Decal lights, implemented with engine modification eventually became a faster more powerful superset of forward lights.

For placeable lights, again we supported forward lights for the same reason as key lights. A lot of smaller teams probably struggle with this, but how much UE4 engine modifications is safe is something we struggled with especially in the earlier phases of development.

We initially experimented with forward lights, but decal lights, which run in an original deferred pass, were easier to optimize and eventually more useful for our artists.

My toon light explanation will focus on decal lights since they’re more interesting.

3D Light (With Gradation)



Usually with placeable light actors in a 3D engine, light distance attenuation is expressed with color gradation.

For Hi-Fi RUSH, this was a big No, because using gradation led to the scene looking 3D.

Toon Light (Cutouts)

In addition to halftones and hatching lines, we use cutouts to 2d stylize our toon light distance attenuation.

The color becomes darker the father away from the light source in cutout steps.

Toon Light (Decal Light)

Arbitrary cutout textures can be used with decal lights.

Our art director prepared 7 cutout texture patterns to be used by environment artists.

MARCH 18-22, 2024 #GDC2024

The scene in the slide looks unlit.

* Play animation displaying the decal light.

Add a decal light with cutout light attenuation and the scene is lit by a 3D light without making the scene look 3D.

Decal lights support arbitrary cutout textures. Our art director prepared 7 cutout texture patterns to give variation to the decal lights used throughout our maps.

Decal Light Rendered Simply As Lights

Spot lights are rendered with cone geometry just like a regular deferred lights.

Decal lights are Hi-Fi's RUSH deferred lights, and so when a decal light is placed in the world, they are deferred rendered with a sphere or cone geometry just like regular deferred lights.

Decal Lights Rendered Using Decal Volumes

How to use a decal light decal volume

1）Place a decal light decal volume on the ground.

2）Assign a decal light to the decal light decal volume.

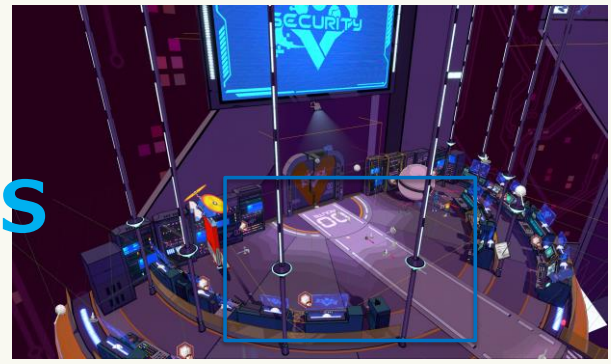
3) Regular decal light geometry rendering will be skipped, and the decal light will be rendered as a decal.

Decal light's namesake and what makes them special is that they can be assigned to decal light decal volumes to render as a decal.

Environment artists can place lights as light sources in a traditional way, but they can limit the projection volume of the light separately using decal volumes.

Decal Light Decal Volume Optimization

VS



Rendering the spot light cone.

Rendering a thin box decal volume.

One obvious benefit of projecting to a decal volume is a smaller area to render and better performance.

Adding A Decal Light For The Wall

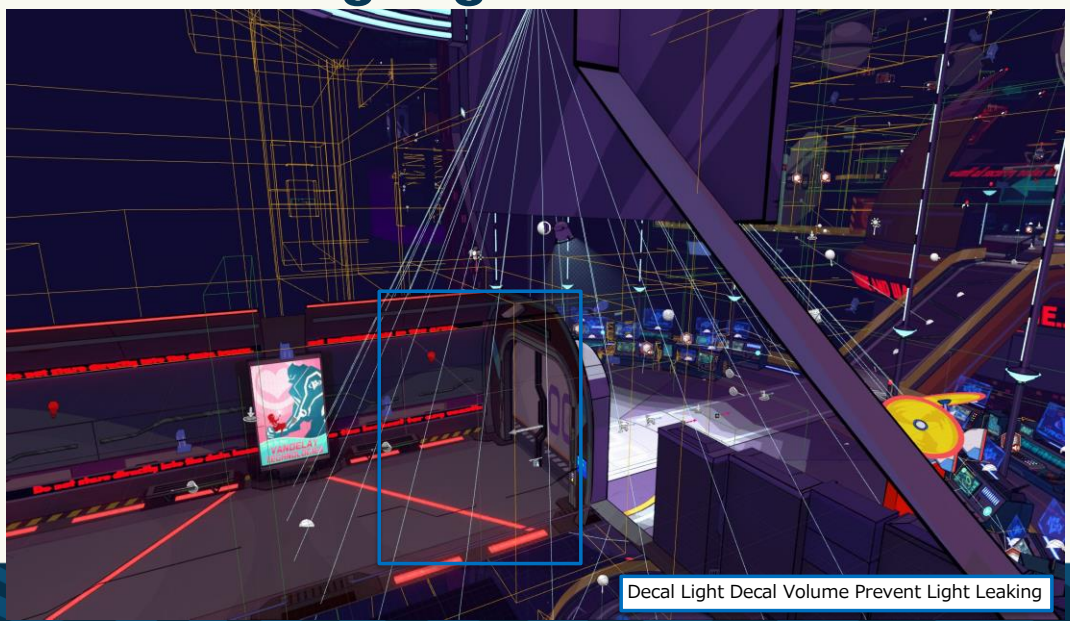
In addition to performance, decal lights provide finer artistic control over which parts of the environment the light affects.

For example, we can add a separate decal light with its own projection texture specifically to light the wall.

- Play animation.

The scene displays a star cutout projection for the ground, but a teardrop cutout projection for the wall.

Preventing Light Leaks

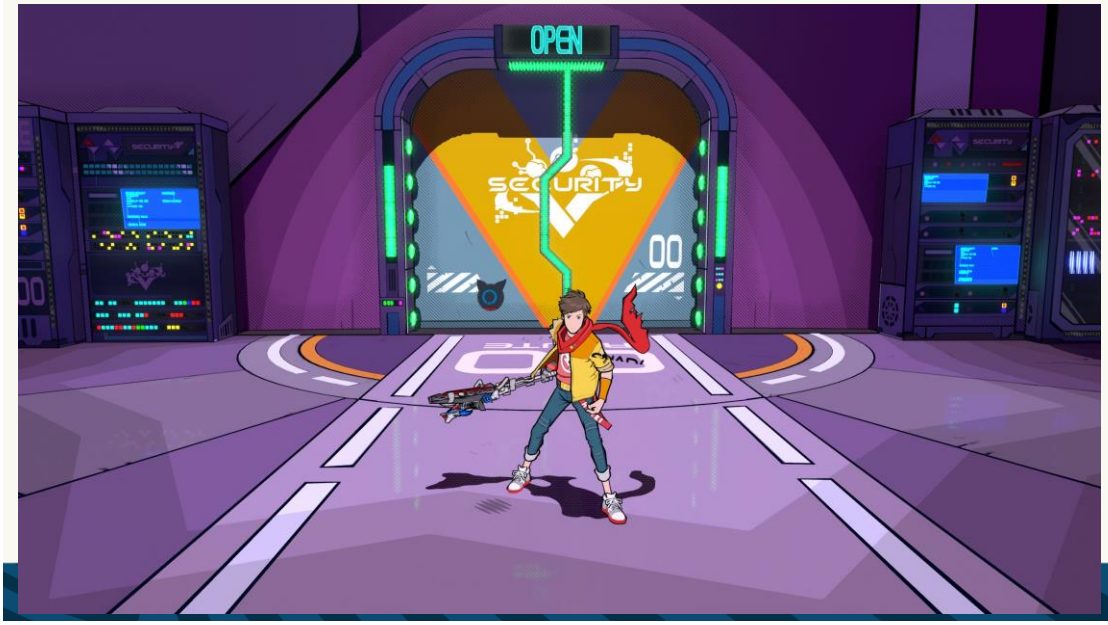


In the Evil Within 2, our PBR lights could use clip volumes to prevent spotlights from light leaking to the next room.

Decal light decal volumes also function as clip volumes and can prevent light leaking.

* Play animation.

What About Characters?



Some of you might be wondering, doesn't decal volumes mean characters aren't properly lit.

Let's try lighting our game's hero Chai with a decal light.

- Play decal light animation.

In Hi-Fi RUSH, character lighting is a separate system from environmental lighting. Forward lights and decal lights don't affect characters.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

Next, I'd like to explain what we do for shadows in Hi-Fi RUSH.

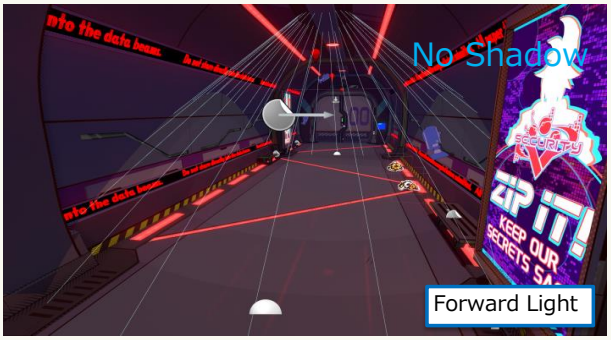
Shadows

Having proper shadows for the environment and characters is extremely important for quality even in toon rendering.

Shadows can also be very performance intensive if applied without a strategy. In UE4, shadows look good but can be costly.

To balance quality and performance, we were careful in how we chose to render our shadows.

Toon Lights Cannot Cast Shadows



Toon lights cannot cast dynamic shadows.

We limit shadow casting lights for performance.

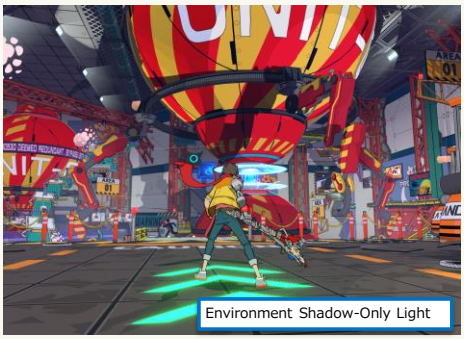
Toon lights are unable to cast shadows.

Cascaded Shadow Map (Dynamic Shadows)

For outdoor environmental shadows, we use UE4's standard cascaded shadow maps.

We have our own pre-baked static shadow map system and use it for parts of directional light shadows for performance, but for the most part, we needed the extra quality that dynamic cascaded shadow maps provide us with.

Shadow-Only Lights (Dynamic Shadows)



- Shadow-only lights are modified UE4 PBR lights customized to only cast shadows.
- Toon lights and shadow-only light are separate light actors.
- Shadow-only lights usage is limited to areas where the quality/cost tradeoff made sense.

For dynamic shadows other than cascaded shadow maps, artists are required to use shadow-only lights.

Shadow-only lights are customized UE4 PBR lights specializing in only casting dynamic shadows.

Because we were going for a 2D toon look, having different lights for lighting and shadowing was not a problem.

Player Shadow-Only Light

An exclusive shadow-only light was assigned to the player character.

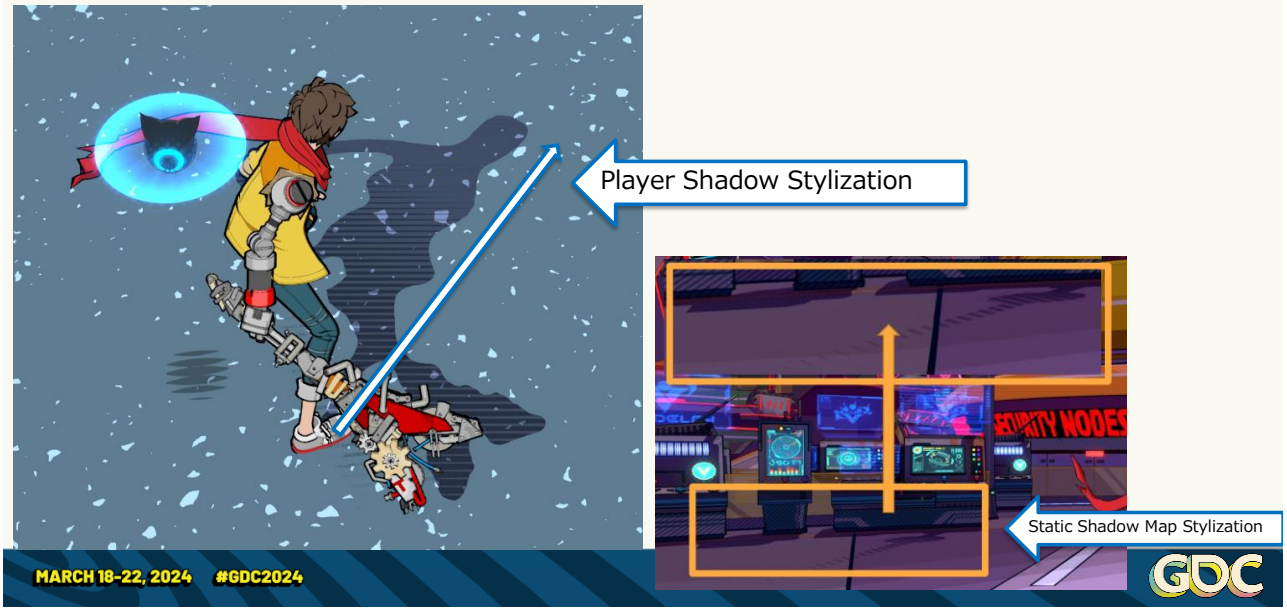
In outdoor areas, the player shadow is not drawn by the cascaded shadow map, but by this separate, player-exclusive shadow-only light.

Character Shadow-Only Light

Giant bosses are allowed their own shadow-only light because their shadows can look very dramatic.

The character shadow-only light in the slide doesn’t cast shadows from the player or the environment and only casts shadows from the giant boss.

Shadow/Light Transition Stylizations



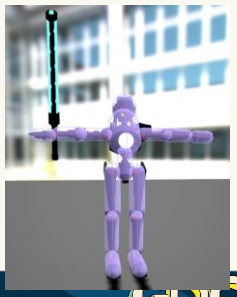
Player shadows and some of our static shadow maps are drawn with stylizations for the shadow-to-light transition.

The hatching lines become thinner the further away the shadow becomes from the shadow casting object representing a 2d toon style shadow to light transition.

Capsule Shadows

Non-player characters use capsule shadows.

We use standard UE4 capsule indirect shadows.



Non-player characters use the more cost-effective capsule shadows for their shadowing.

Capsule shadows are a standard UE4 functionality. Artists prepare capsule shapes approximating the character mesh and these shapes are used to calculate per-pixel visibility towards a global light direction to approximate soft shadows.

It wouldn't work for a photorealistic game, but we can get away with the blobby look of capsule shadows because we are toon.

Capsule Shadows Cast Shadows Anywhere

Because capsule shadows don’t require shadow casting local lights, we get cost-effective character shadows outdoors, indoors, regardless of lighting conditions.

AO Volume Decal Shadows

・Flying robots and our partner cat are too far away from the ground for capsule shadows.

・CPU raycasts are performed in order to find a world position where a decal shadow is rendered.

Capsule shadows become too large and dispersed for small characters that are floating in the air far away from the ground.

For these characters, we implemented a simple AO volume decal system. CPU raycasts are used to find the proper decal placement world position.

Environment Shadow-Only Light

Finally, we also allow a single shadow-only spotlight for the environment team.

For the scene in the slide, a shadow-only light casts exclusively from the moving giant robot arms.

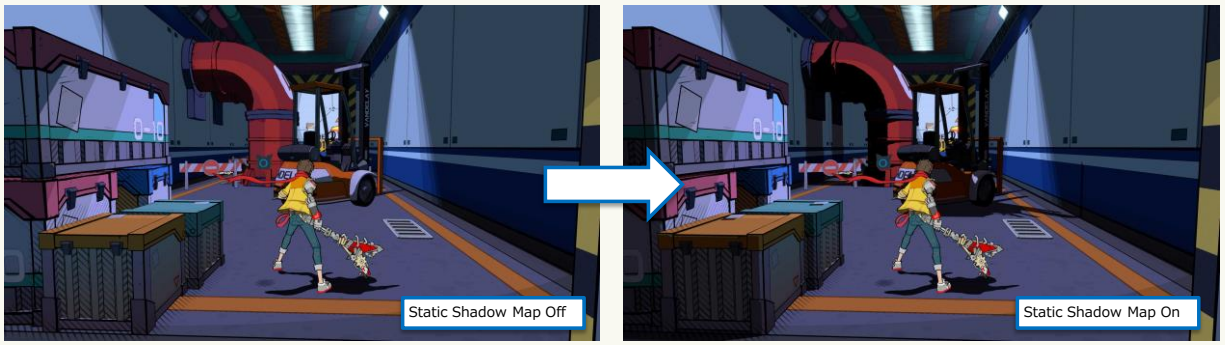
The only thing shadow-only lights can do is cast shadows, but just like for the character boss shadows, a well-placed shadow spotlight can be very dramatic in a toon look.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

I’ve mentioned we have a static shadow map system. I’d like to explain this system in more depth in the next section.

Static Shadow Map



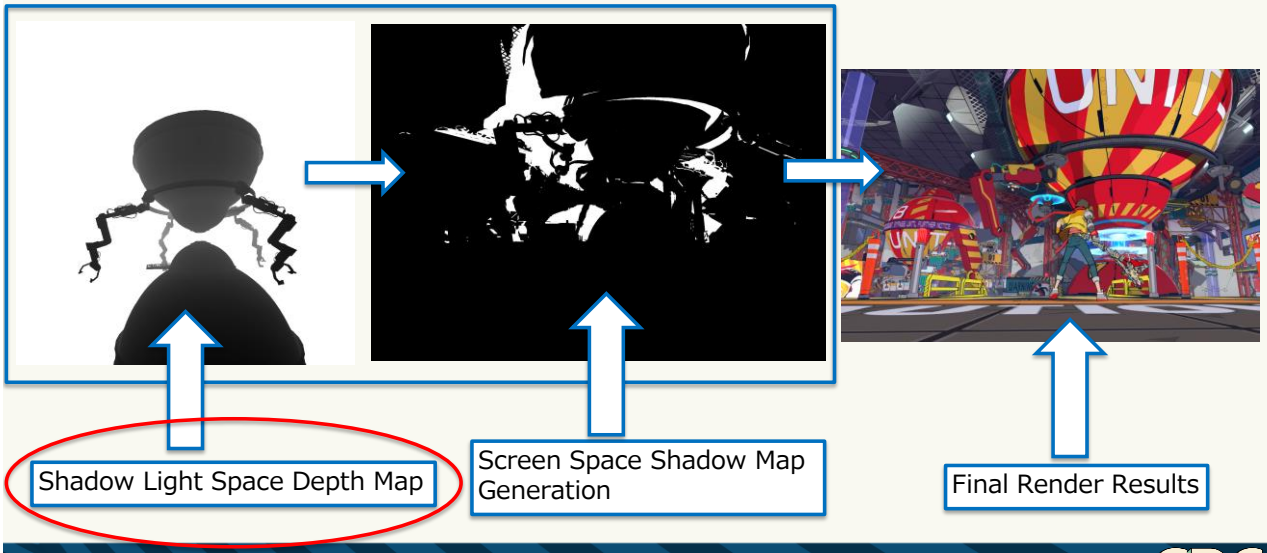
Static shadow maps are an important element of Hi-Fi RUSH's shadow system.

Static shadow maps are offline baked shadow maps.

They are important for us both in terms of performance and artist's ease-of-use.

UE4 has a static shadow map system which is integrated into Lightmass baking, but we created our own for better integration with our deferred toon rendering pipeline.

What Gets Offline Baked?



Shadow mapping is composed of two steps.

The first step is light space depth map generation from the perspective of the shadow casting light.

The second step is comparing the pixel's light space depth against the depth map to generate the actual shadow map.

What we prebake in static shadow maps is the first depth map capture step.

Static Shadow Map On/Off

In the slide image, I’m toggling on and off the static shadow map display.

The environment team had strict limitations on dynamic shadow map usage and interior environmental shadows will typically disappear completely without static shadow maps.

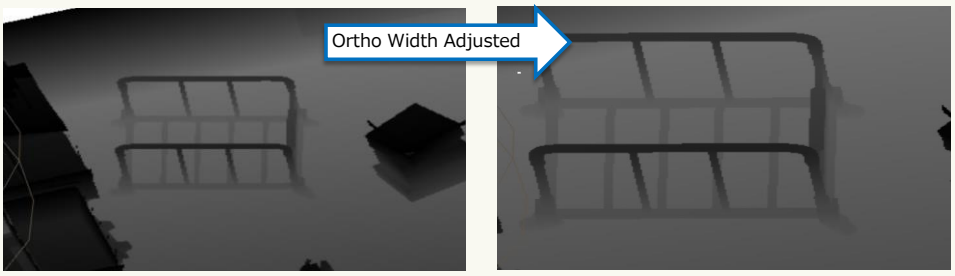
Static Shadow Map Actor

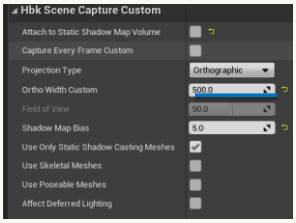
By placing static shadow map actors in the environment, environment artists can use static shadow maps.

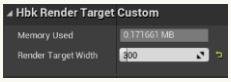
The static shadow map actor is composed of two components. A scene capture camera for capturing the depth map and a deferred box volume for rendering the shadow map.

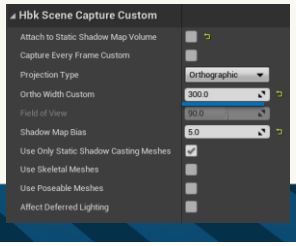
When the static shadow map actor is selected, the captured depth map is displayed in the lower right corner of the screen in the editor.

Maximizing Depth Map Coverage





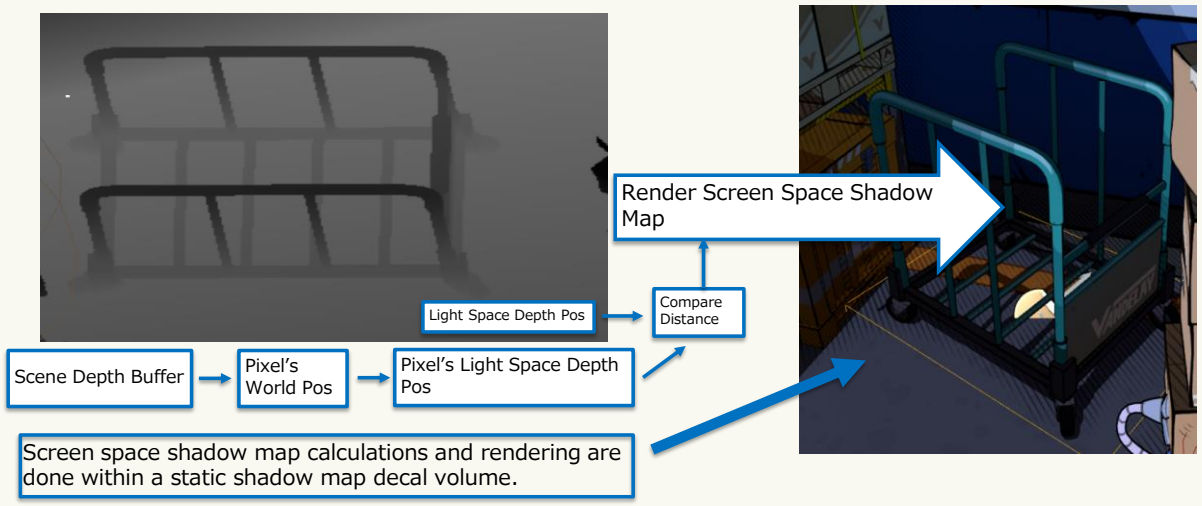




Artist manully adjust camera direction, position, projection parameters such as ortho width to maximize the shadow caster's depth map draw area.

Artists use the depth map preview in the lower right portion of the screen to manually maximize depth map coverage of shadow casters to improve shadow map quality.

Calculating The Screen Space Shadow Map



Standard comparisons between the pixel's light space depth position and the depth map's light space depth position are performed to render a screen space shadow map inside the static shadow map decal volume.

Screen Space Shadow Map AA

- 4x4PCF is used for shadow map antialiasing.
- The Gather4 instruction is used for optimization.
- Our code is based on UE4's ShadowFilteringCommon.ush.

Without AA, our screen space shadow maps look very jaggy. We use 4x4 PCF for antialiasing our shadow maps.

Screen Space Shadow Map AA

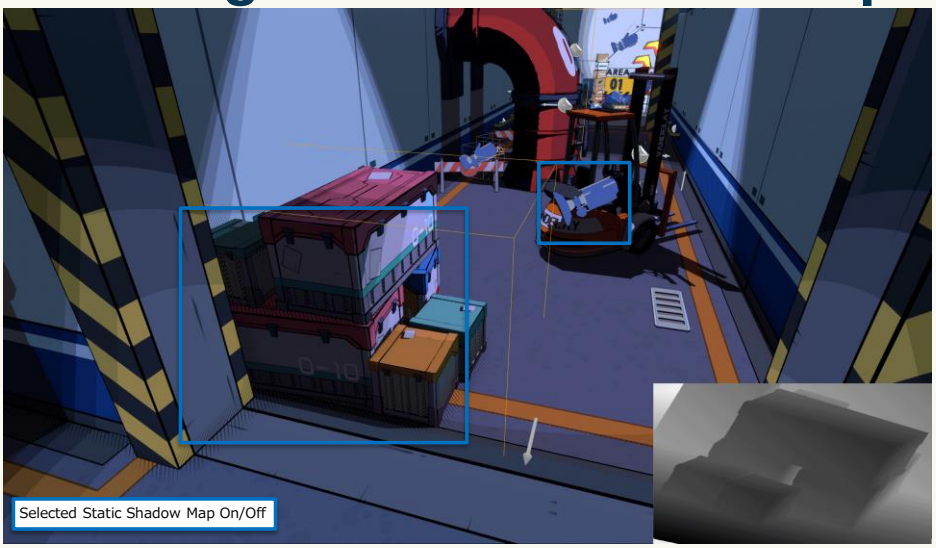
Here’s a GIF animation of 4x4 PCF being toggled on and off.

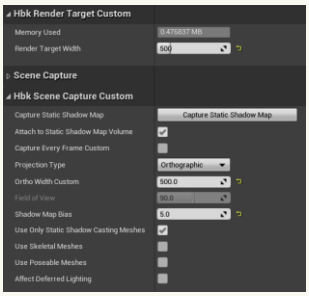
As you can see, the algorithm does a pretty good job of alleviating jaggies for us.

Static Shadow Map Camera Placement

Here’s a screen shot of static shadow map camera placement in an actual scene. Many static shadow map cameras with small coverage are placed throughout our levels.

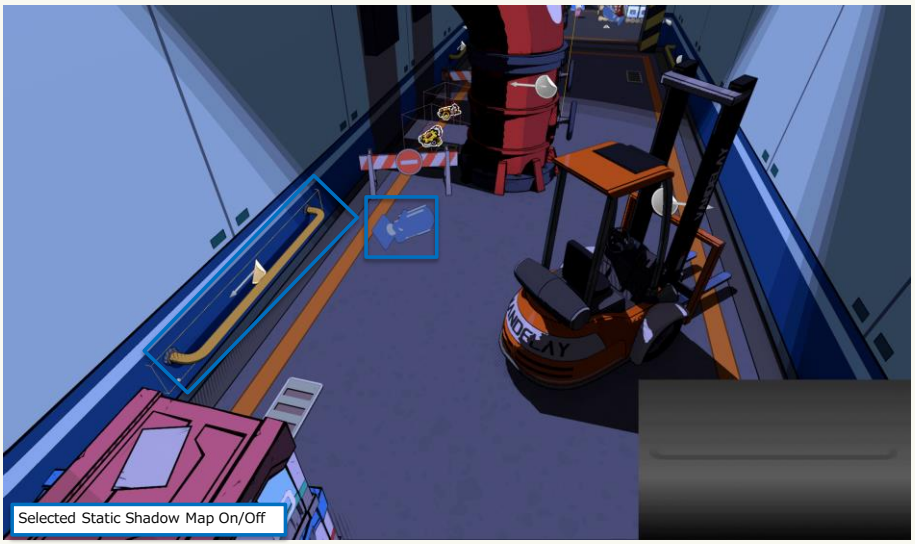
A Single Static Shadow Map Camera

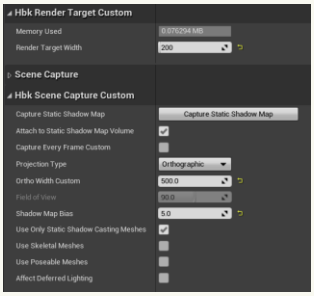




Each static shadow map camera renders its own depth map and its associated render volume owns that static shadow map’s depth map texture data.

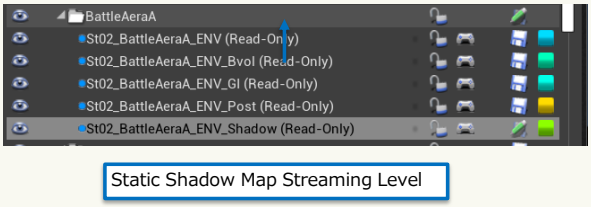
A Different Static Shadow Map Camera

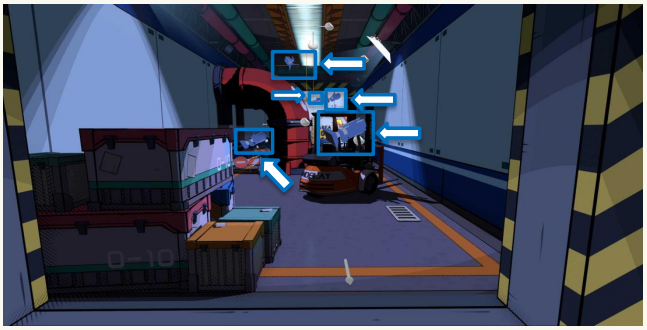




Even the small handrail in the slide image has its own static shadow map.

Static Shadow Map Streaming





- Focus on efficiently distributing shadow map texture streaming across multiple frames.
- Multiple static smaller shadow maps instead of a single large static shadow map.
- Static shadow map streaming levels with a per-frame memory limit.

Our artists use many static shadow map cameras not just for artistic reasons, but for texture streaming efficiency as well.

One of the first concerns we had when we considered using static shadow maps was memory usage and streaming hitches.

We managed static shadow map texture streaming workloads by monitoring texture size and focusing on being able to efficiently distribute streaming across multiple frames. For this reason, we can’t have individual static shadow map textures being too large.

Our static shadow maps are placed in their own streaming level and within the streaming level, we distribute texture streaming across multiple frames by placing a per-frame streaming memory limit.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

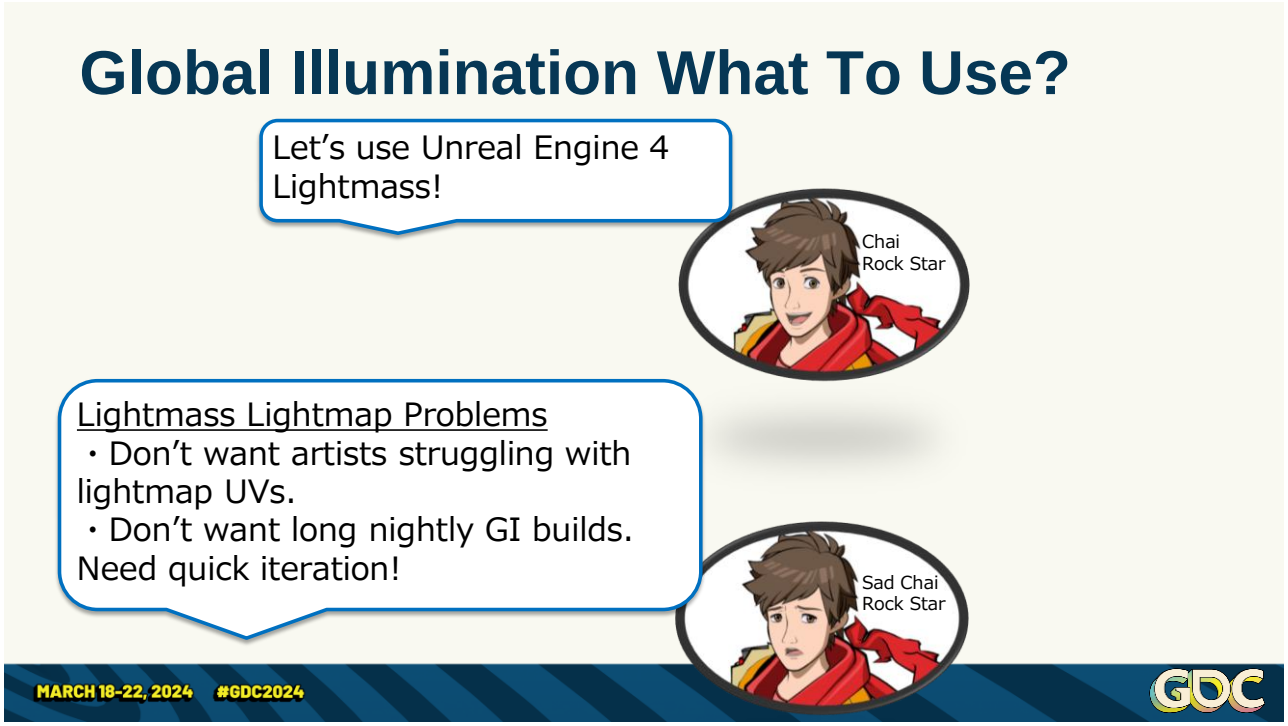
The final topic I'd like to talk about is how we handle global illumination.

Toon with Global Illumination

GI is noticeable around the center data sphere.

Global illumination is being toggled on and off in the slide image.

We felt supporting standard 3D lighting features in our toon rendering was important for quality and this included GI.



We decided early on that we wanted to try to use UE4 Lightmass since we were satisfied with its quality and baked lighting is great in terms of performance.

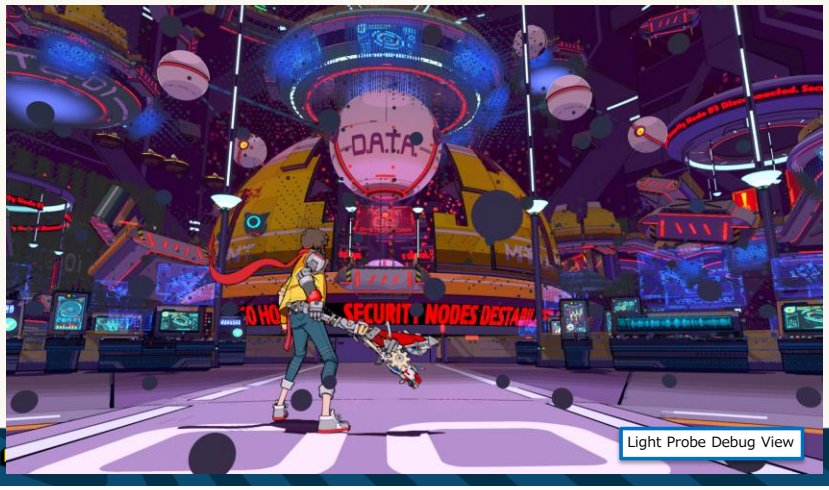
At the same time, we also thought, a workflow using Lightmass lightmaps was too costly for our artists due to needing to deal with lightmap UVs and possibly long GI bake times.

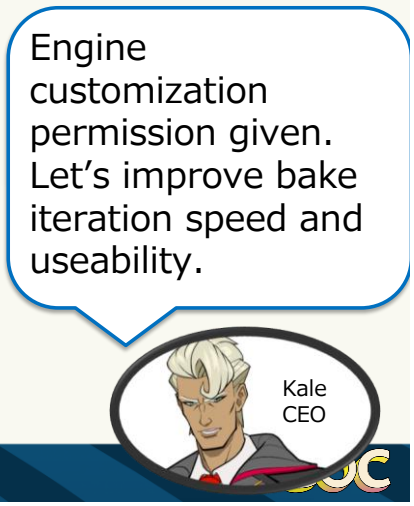
Toon assets require artist handcrafting for quality and all artists including our environment team were super busy throughout the project.

Let’s Customize UE4 Volumetric Lightmap

What to do ?

・Customize Lightmass volumetric lightmaps and use light probes.





We had noticed that Lightmass volumetric lightmaps could be used for the environment. Lightmass volumetric lightmap light probes bake faster and don’t require lightmap UVs.

We decided that we would customize the volumetric lightmap functionality with a focus on further improving bake iteration speed and useability.

Local Volume GI Lighting

A big factor that allowed further customization is the fact that our game has a toon art direction, and we can be artistically selective in which parts of the map we apply global illumination.

In the slide image, only a small area around the window has global illumination, and that can be okay for our look.

GI Baking Limited To Necessary Areas

Here’s the same level with the GI light probe debug display enabled.

The environment artist has decided to bake probes around strong emissives, but in areas without a strong light source, no probes are baked.

World Volume Lighting Volume

What's the key point in our volumetric lightmap customization.

Per-Level Baking changed to Per-Actor Baking

Baking & rendering done per World Volume Lighting Volume.

UE4 has a functionality called Lightmass importance volume that limit light probe baking to the area within the volume.

We customized this functionality to implement our own global illumination lighting volume actor.

In vanilla UE4, light probe data is stored per-level. We customized UE4 so that, in our game, light probe data can be stored per actor.

Optimizing Volumetric Lightmap Data

AmbientVector + SHCoefficient[0-6] = 8 x 3D Textures

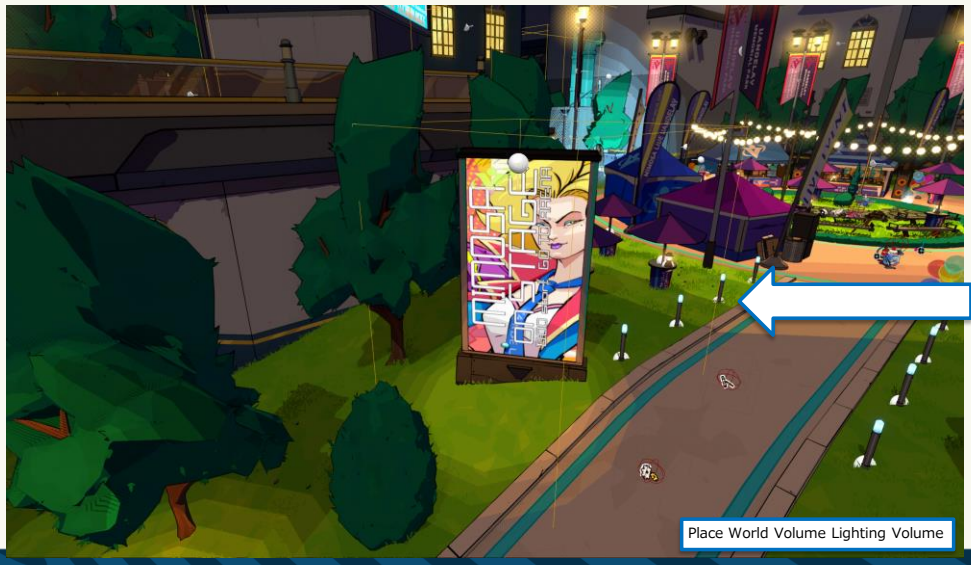
Need only directionless ambient color.

AmbientVector 3D Texture = 1 x 3D Texture

*An indirection 3D Texture x 1 is also used.

I'll should mention that we did optimize the UE4 volumetric lightmap data to save memory. Because we only need directionless ambient global illumination, we cut down the light probe spherical harmonics 3d textures from 8 to 1. It's a big memory saving.

World Volume Lighting Baking Workflow

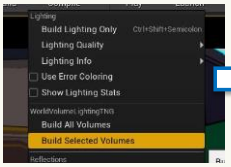


1) Enclose the area you want to bake GI with a World Volume Lighting Volume actor.

Since I emphasized bake iteration speed and useability as keypoints, I'll give a quick overview of what our GI bake workflow looks like.

We start by enclosing the area we want to add global illumination to with a GI lighting volume.

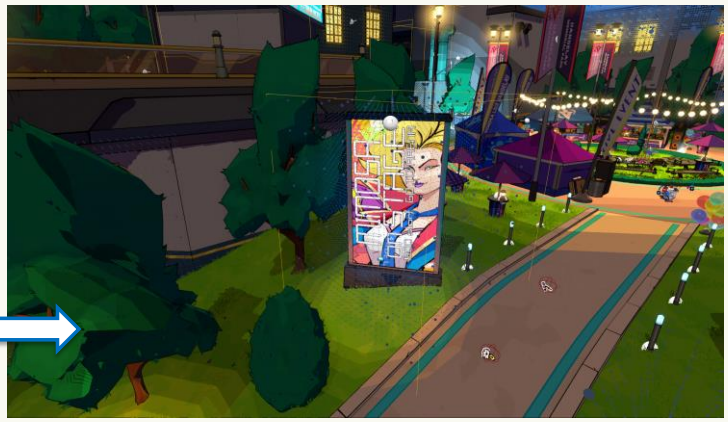
World Volume Lighting Baking Workflow



2) Start a bake from the menu.

3) Wait 1, 2 minutes.

4）The area enclosed by the World Volume Lighting Volume will have its global illumantion baked.



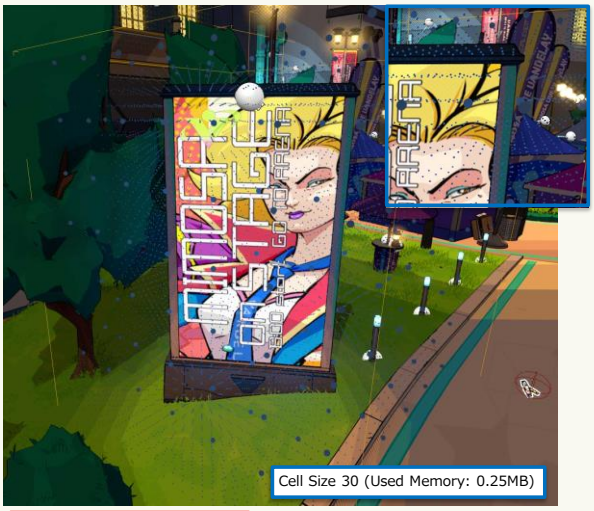
After the volume is placed, artists select from the menu “Build Selected Volumes.”

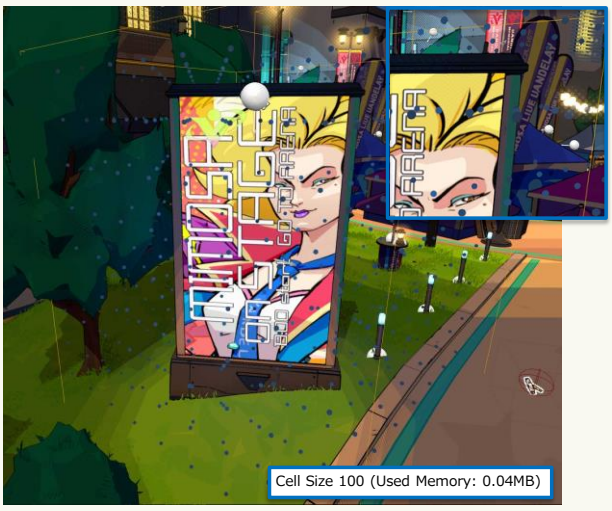
A customized Lightmass GI baking code is executed, and the GI bake is limited to the area enclosed by the lighting volume.

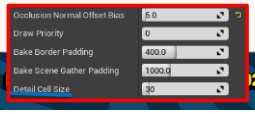
Bake times are dependent on Lightmass settings and PC specs, but as a general idea, for a standard dev PC at our company, a GI bake should finish in a minute or two.

Since the bake is isolated to the selected actor, artists can iterate baking on a specific area quickly.

World Volume Lighting Volume Parameters







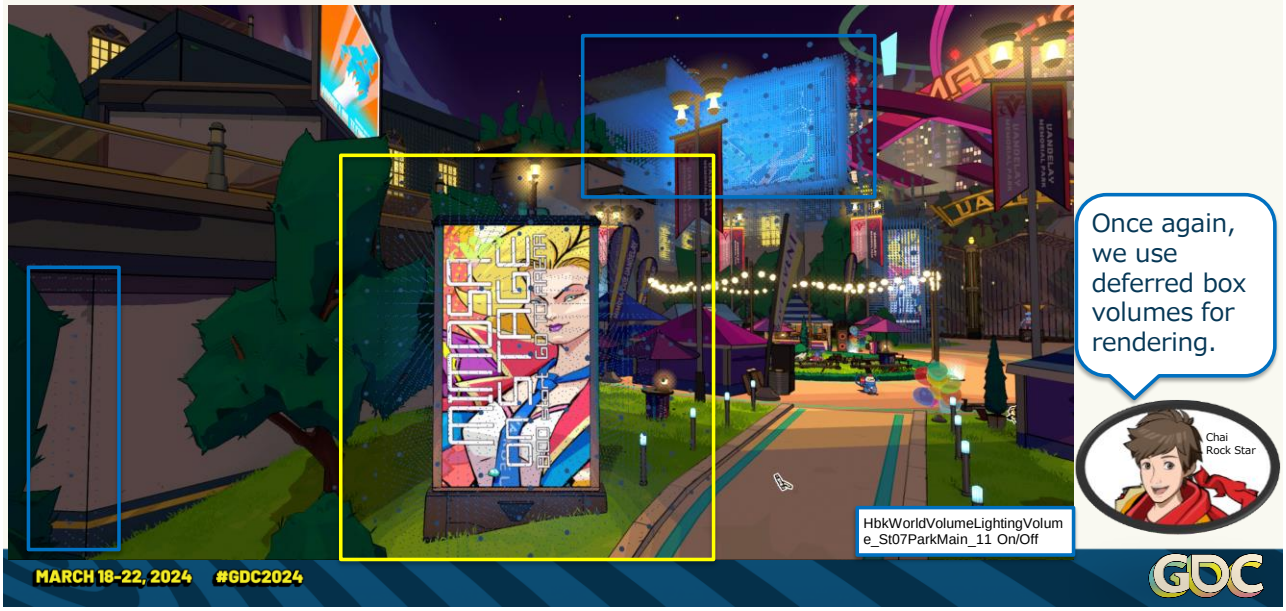
Light probe cell size is adjusted to balance memory usage and bake quality.

The GI lighting volume contains parameters for both baking and rendering.

An often-used parameter for baking is the detail cell size parameter, which controls the distribution granularity of the generated light probes.

Our environment artists use this parameter to balance bake quality and memory usage.

Rendering A World Volume Lighting Volume



Finally, the GI lighting volume not only defines the GI bake extent, but also serves as a rendering volume.

Here, I'm toggling on/off the GI box rendering for a specific volume. Note how all the other volumes in the scene are unaffected by the toggling.

Deferred rendering a box volume is a local lighting technique that is used throughout our game and makes a final presentation appearance in this slide.

My References

Stefan Gustavson. 2012. “Procedural Textures in GLSL”. Linköping University Electronic Press (Same as OpenGL Insights, Chapter 7).

Kevin Myers. 2016. “Sparse Shadow Tree”. SIGGRAPH 2016.

Li Bo. 2019. “A Scalable Real-Time Many-Shadowed-Light Rendering System”. SIGGRAPH 2019.

Zhenzhong Yi. 2020. “From Mobile to Console: Genshin Impact’s rendering technology on Console”. Unite Seoul 2020.

Nikolay Stefanov. 2016. “Global Illumination in Tom Clancy’s The Division”. GDC 2016.

Noriaki Shinoyama. 2018. “Lightmass Deep Dive 2018 Vol.1”.

Noriaki Shinoyama. 2018. “Lightmass Deep Dive 2018 Vol.2”.

Carsten Wenzel. 2006. “Real-time Atmospheric Effects in Games”. Siggraph 2006.

Miles Macklin. 2010. “Inscattering Demo”. http://blog.mmacklin.com/2010/05/29/in-scattering-demo/

Yossarian King. 2000. “2D Lens Flare”. Game Programming Gems 1.

*Includes bonus slide references.

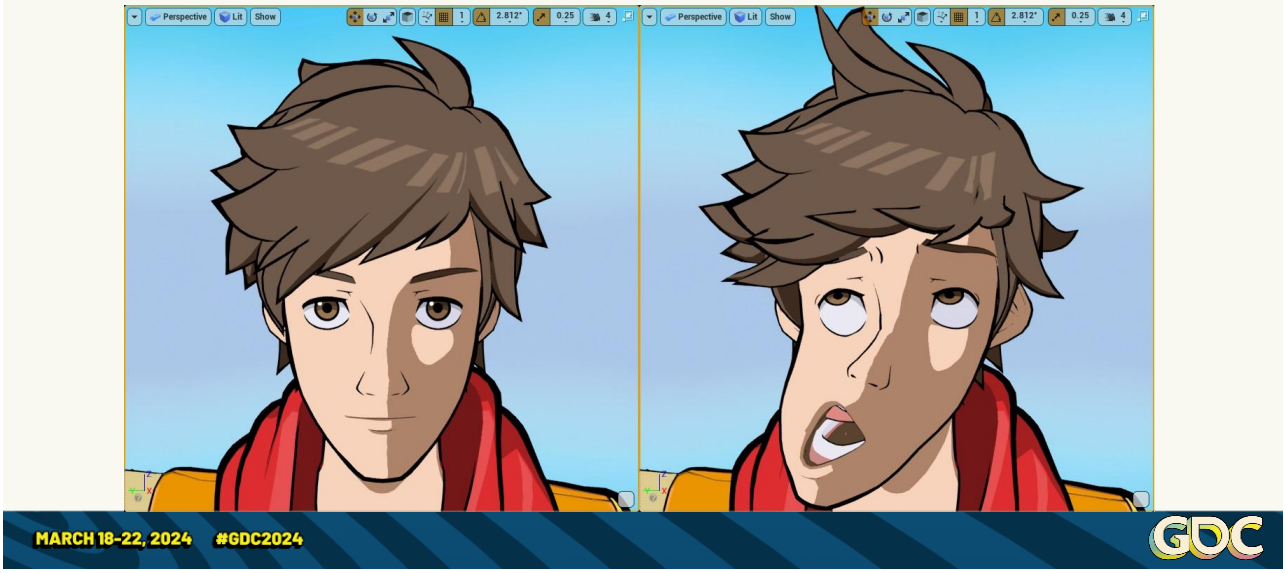
Here are my references.

Now, I’m going to hand it off to Komada-san for his section on the tech behind our toon face shadows.

Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

Toon Face Shadow



I’m going to explain the face shadow of Hi-Fi Rush in this section.

Hi-Fi RUSH Character Self Shadow

Cel-shading style.

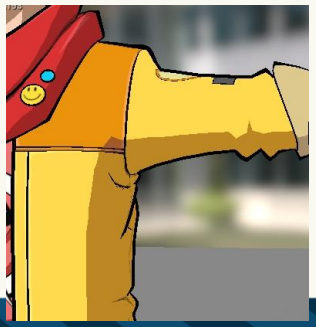
→The shapes of the shadow areas should be “always clean”!

The self-shadows of the characters in Hi-Fi RUSH are in a cel-shaded style.
The shadowed and non-shadowed areas are clearly separated, and the shadowed areas are given a specified shadow color.
With shadows like this, the quality is all about the choice of shadow color and the shape of the shadow area.
Hi-Fi RUSH’s art concept was sharp, clean, and colorful, so the shadows also needed to always have clean shapes.

Hi-Fi RUSH Character Self Shadow

The shadow shapes are determined by NdotL
“except for the face”.

```
// Pixel Shader
NdotL = Dot(normal, lightVector)
If (NdotL > threshold) {
        return baseColor ;
} else {
        return shadowColor ;
}
```



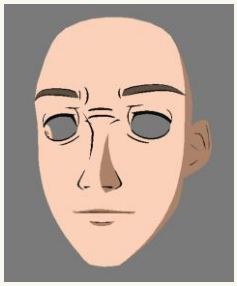
Whether a pixel is in a shadow area is determined by whether the dot product of the pixel's normal and the light vector is less than a threshold.

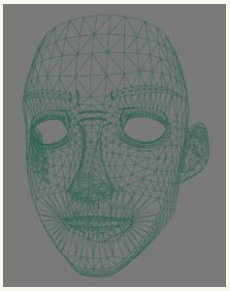
This is a typical method for cell shading.

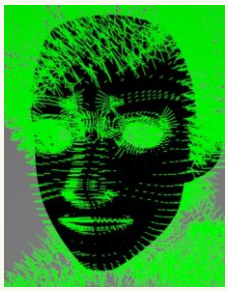
The vertex normal is used for the normal.
However, this method is not used on the face.

Problems With Vertex Normals

Problem 1: It’s difficult to generate a smooth curve.







The face requires higher quality than other parts.
However, there were problems in achieving the desired quality with vertex normal shadows.

The first problem is that using vertex normals makes it difficult to create a smooth curve.

As shown in the image on the right, depending on the direction of the light, the shadow shape can become quite messy.
It is because there is a limit to how much geometry can be divided, and the shadow shape is affected by how it is divided.

Problems With Vertex Normals

Problem 2: Facial motion breaks the shadow shape.

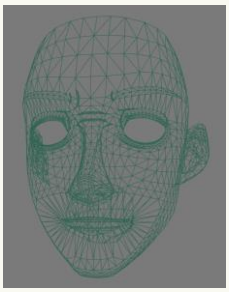
The second problem is that the shape of shadows created by vertex normals is easily broken by facial motion.

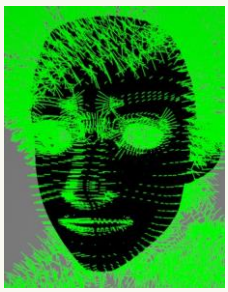
When the bone orientation changes due to facial motion, the orientation of the vertex normals also changes, which unintentionally breaks the shadow shape.
In this scene where the facial expression is extreme, the shadow shape becomes considerably broken depending on the direction of the light.

Problems With Vertex Normals

Problem 3: It is difficult for artists to solve problems 1 and 2 by adjusting the model.

Problem 1: It’s difficult to generate a smooth curve.
Problem 2: Facial motion breaks the shadow shape.





The third problem is that it is difficult for artists to make adjustments.
Artists can adjust the geometry and vertex normals, but it is very difficult for them to avoid problems 1 and 2 completely through manual adjustments.

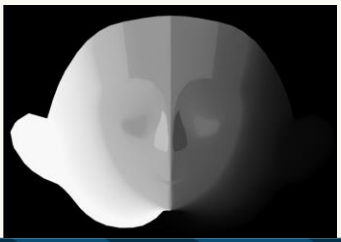
It is difficult to intuitively understand the resulting shape of the vertex normal and geometry adjustment in all the various lighting directions.
Even if you use a normal map instead of vertex normals, problem 1 may be solved, but problems 2 and 3 will not be solved.

Shading With A Threshold Map

- Create a height map texture called a "threshold map".
- Convert the horizontal orientation of the light to a threshold value.
- Determine the shadow shapes by height.

```
lightDirHorizontal = Normalize(lightDir - Dot(lightDir , headBoneUp))
lightAngleHorizontal = ArcCos(Dot(lightDirHorizontal , headBoneForward))
threshold = lightAngleHorizontal / PI
```

```
// pixel shader
height = TexSample(thresholdMap , uv)
If (height > threshold) {
        return baseColor ;
} else {
        return shadowColor ;
}
```



To solve these vertex normal problems, in Hi-Fi RUSH, we used a heightmap-like texture.

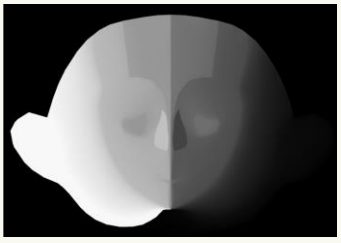
Internally, we call this texture a threshold map.

I will explain how to author it in later slides, but will first explain the calculations involved.

First, calculate the angle between the horizontal component of the light direction and the forward direction of the head.
The horizontal component here refers to the horizontal component in the coordinate system of the head bones.
The threshold value is the angle divided by pi and normalized to a range of 0 to 1 .
Then, it is determined whether the area is a shadow area or not based on whether the sample value of the threshold map is greater than the threshold value.

Shading With A Threshold Map

- The shadow only moves horizontally.
- When the light hits from the left side, flip the texture horizontally.



Reference:

Unity Seoul 2018

" From mobile to high-end PC: Achieving high quality anime style rendering on Unity "

Jack He (miHoyo)

MARCH 18-22, 2024 #GDC2024

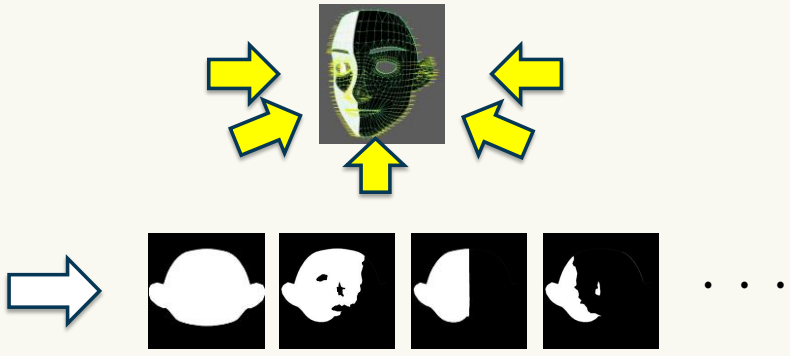
Since only the horizontal component is considered, the shadow can only be moved in the horizontal direction.
However, this was enough for Hi-Fi RUSH.

In this video, the light is shining on the right side of the face, but when it is on the left side, the texture is reversed horizontally.

How to use this threshold map is a reproduction of the method introduced in the session by miHoYo at Unity Seoul 2018.

How To Author A Threshold Map

1. Do a 180 degrees light bake of the vertex normal shadows into a textures on the DCC tool.

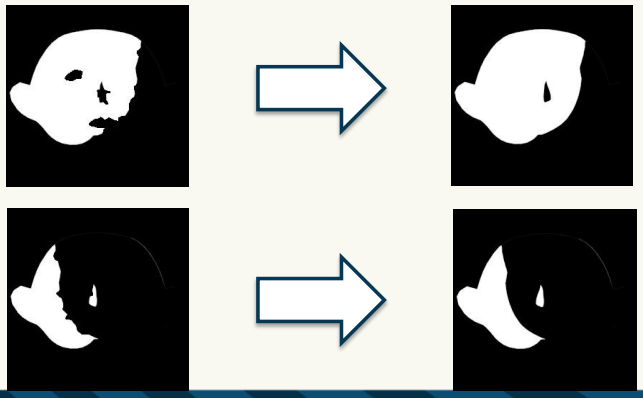


I will now explain how to author a threshold map.

First, use a DCC tool to bake the shadows of the face into textures by moving a directional light at fixed angles 180 degrees around the character, as shown in the image.
We moved the light every 5 degrees, so we baked 36 textures.
At this point, shadows are shaded using the vertex normals, so the shape may not be smooth or shadows may appear in unnecessary areas.

How To Author A Threshold Map

2. Artists can manually retouch the baked textures.



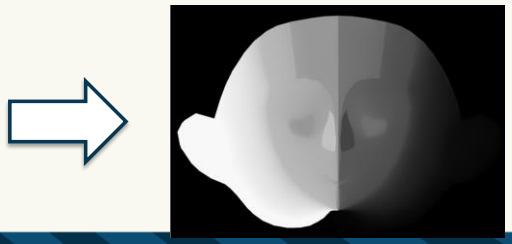
These textures are then retouched to the artist's satisfaction.
This process creates a clean shape, so aritsts don’t have to create perfect vertex normals.

As you can see, retouching textures is much more intuitive to adjust than vertex normals.

Artists can also preview the final look of shadows in our DCC tool.

How To Author A Threshold Map

3. Merge into one sheet using distance field-like interpolation using in-house tools.

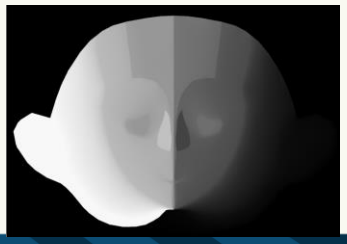


Finally, using a dedicated in-house tool, all textures are interpolated like a distance field interpolation and merged into a single texture.
The threshold map is now complete.

Why Threshold Maps Solve The Problems

Problem 1: It's difficult to generate a smooth curve.

→Smooth curves are always guaranteed with artist retouching and distance field interpolation.



MARCH 18-22, 2024 #GDC2024

I'll now review how threshold maps solved our original vertex normal problems.

Regarding problem 1, the shape does not become a smooth curve with the vertex normal, but with a threshold map, it is always smooth as shown in the video.

This is thanks to the artist adjusting the shadow shape through retouching.

Also, since the textures are connected smoothly using distance field-like interpolation, the shape changes smoothly when the direction of the light changes.

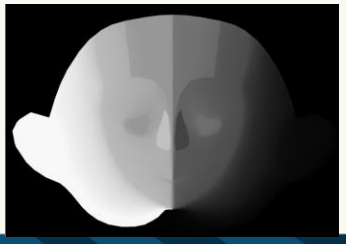
As an aside, threshold maps have similar characteristics to distance field textures, and are less prone to artifacts even when the resolution is lowered.

However, in order to maintain a sharp shape even when zoomed in, we used 2K textures.

Why Threshold Maps Solve The Problems

Problem 2: Facial motion breaks the shadow shape.

→The shadow area is mapped using UVs and unaffected by bone movements.



Problem 2, where the shape is broken due to facial motion, does not occur with the threshold map.

Because in the case of threshold map, the shape is mapped with UV so is not affected by bones.

Why Threshold Maps Solve The Problems

When using vertex normals.

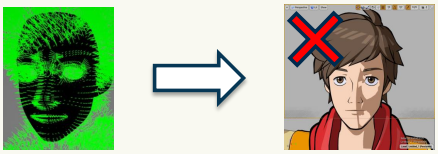
When using threshold maps.

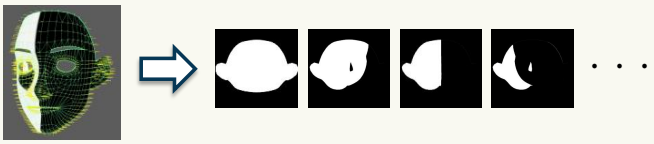
As an example, let's change the direction of the light and compare.
When shading with a threshold map, shadow shapes are clean in all light directions.

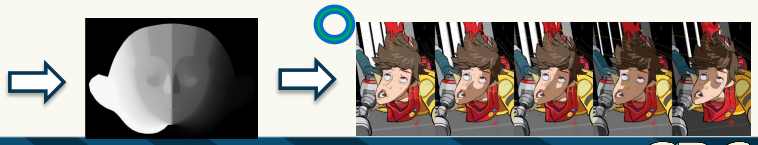
Why Threshold Maps Solve The Problems

Problem 3: It is difficult for artists to solve problems 1 and 2 by adjusting the model.

→Retouching each texture is much more intuitive.







Problem 3 is that shading using vertex normals is difficult for artists to adjust. Again, in the case of a threshold map, artists can intuitively modify the shape by retouching the baked textures.

When shading using vertex normals, the vertex normals were directly used for shading.

When shading using a threshold map, in between light baking and texture merging, there is a step where artists can make intuitive quality adjustments.

This concludes the explanation of face shadows in our game.

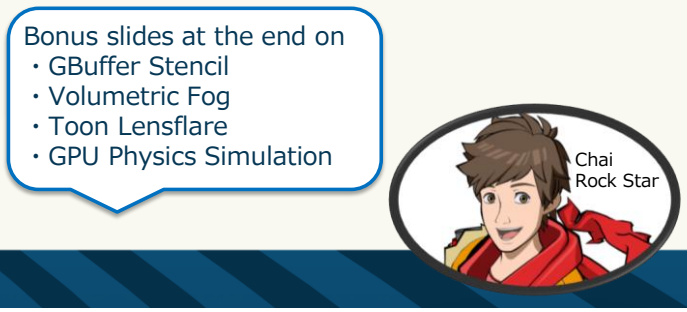
Agenda

Game Introduction
Deferred Toon Rendering
Comic Shader
Toon Light
Shadows
Static Shadow Map
Global Illumination
Toon Face Shadow
Final Words

I’d like to end our talk with some final words.

Final Words

- Unique toon art style with unique rendering challenges.
- Deferred rendering adapted for stylized toon rendering.
- Provide the team with the best tech for our goals.



Our game wanted to do a stylized cel-shaded look for both characters and the environment and this presented unique rendering challenges.

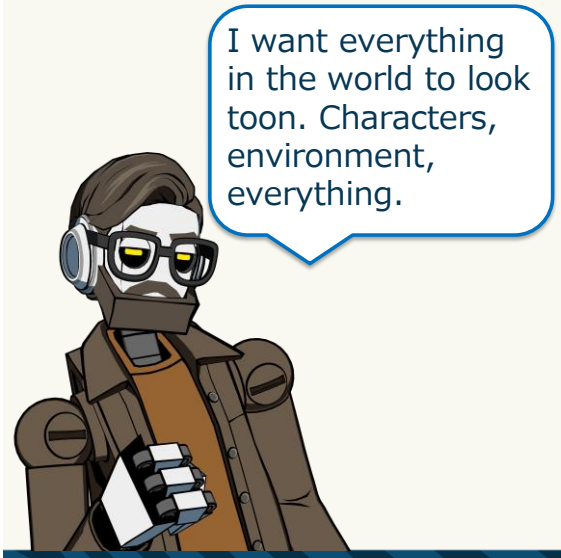
Unique problems call for unique solutions and we developed an original deferred post process volume approach for our attempt at toon rendering.

We're using UE4 and we adapted many 3D graphics features for toon rendering. We use well established graphics techniques and build on the great works of others.

We talked about tech today, but its toon rendering and as you can imagine there was a lot crafting and optimizations by our artists not covered today that ultimately determined the quality of the toon graphics. We had goals of a rock solid 60fps, high resolution and great looking toon and the tech we talked about today helped the Hi-Fi RUSH team achieve these goals.

We had more tech we wanted to share info about but didn't want to tire people with too much information so there are unpresented bonus slides. They'll be available for download with our slides.

Special Thanks



Our Art Director & Artists
System Programming Team
Environment Programming Team
Our QA & Bethesda QA
Intel, AMD, Nvidia Tech Support
Microsoft ATG
Epic Games Japan

Everyone on the Hi-Fi RUSH team

GDC Special Thanks
Liz from Bethesda
Our GDC Advisor Chris

We get to present this stuff, but our toon rendering was made possible by the hard work of everyone on the Hi-Fi RUSH team.

Our final slide is a special thanks page to give a shout out to all the people who made today's talk possible.

*Chris, our advisor, gave great feedback to help us revise, came to our talk and was just overall awesome. Thanks Chris!

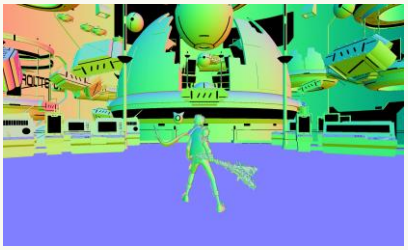
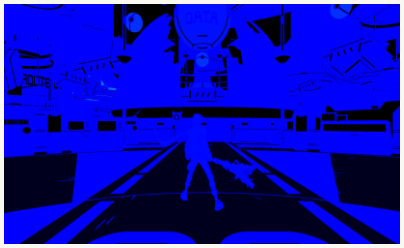
Stuff We Couldn’t Fit In Our Talk

So apparently, we made too many slides...

Agenda (For Stuff We Couldn’t Fit In The Talk)

GBuffer Stencil
Volumetric Fog
Toon Lensflare
GPU Physics Simulation

Deferred Toon Renderer GBuffer


- We keep a standard UE4 GBuffer setup.
- RGBA8 x 3 GBufferA, B, C.

For performance and wanting to minimize engine modifications, we didn't change the standard UE4 GBuffer setup of 3 RGBA8 render targets.

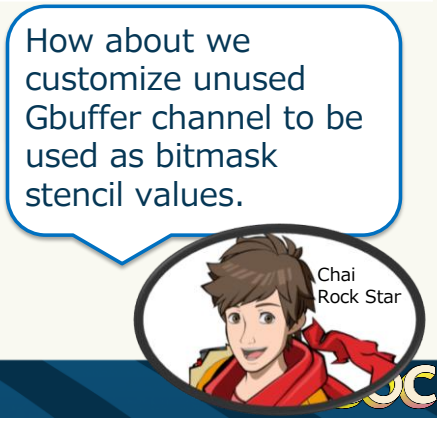
Our GBuffer Layout

	GBufferA R8G8B8A8
R	World Normal X
G	World Normal Y
B	World Normal Z
A	PerObjectGBufferData

	GBufferC R8G8B8A8
R	Base Color R
G	Base Color G
B	Base Color B
A	GBufferAO (Customize)

	GBufferB R8G8B8A8
R	Metallic (Fix at 0)
G	Specular (Customize)
B	Roughness
A	ShadingModelID+SelectiveOutputMask

・We customized GBufferAO, Specular channels for toon rendering.



Though reluctant to add additional rendertargets, we needed a way to add our own extra information.

Because we're using not using PBR, a couple of the default UE4 PBR GBuffer channels were unnecessary for us.

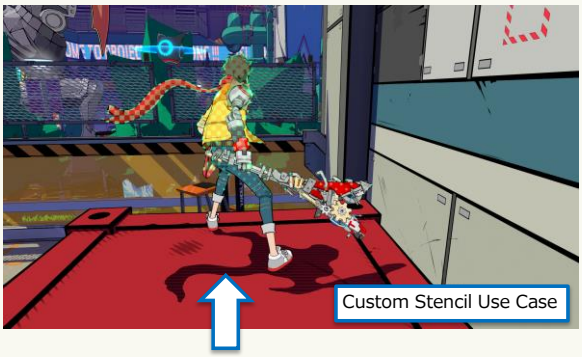
We ended up modifying the engine source code to use the specular and GBufferAO channels for our own purposes.

To get the most information out of 2 8-bit channels, we converted the 2 channels to bitmask stencil values.

GBuffer Stencil

```
// GBUFFERAO GBUFFER STENCIL BITS (Partial Listing)

#define HBK_AO_STENCIL_CHARACTER_MASK          4
#define HBK_AO_STENCIL_SKIP_NORMAL_OUTLINE     16
#define HBK_AO_STENCIL_SKIP_DEPTH_OUTLINE      32

// SPECULAR GBUFFER STENCIL BITS (Partial Listing)

#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_MAIN                1
#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_2                   2
#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_3                   4
#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_4                   8
#define HBK_SPECULAR_STENCIL_ENV_SKIP_AMBIENT_CUBEMAP        16
#define HBK_SPECULAR_STENCIL_ENV_SKIP_STATIC_SHADOWMAP       32
#define HBK_SPECULAR_STENCIL_ENV_SKIP_SSAO                   64
```

Partial listings of our GBuffer stencil flags.



UE4 custom stencils are used for some game-side post processes.

GBuffer stencil requires no additional render passes and is practically free.

I think packing stencil flags into a GBuffer channel is something a lot games do. We internally called our modification the GBuffer stencil.

This is to distinguish it from UE4's custom stencil functionality, which we also used for special game-side post processes.

It doesn't work for emissive or transparencies, but a great benefit of GBuffer stencil over standard custom stencil is that we're already rendering to the GBuffer and we don't incur additional render pass costs.

GBuffer Stencil (Character Stencil)

Pixels With Character Stencil Mask Are Drawn In White

```
// GBUFFERAO GBUFFER STENCIL BITS (Partial Listing)

#define HBK_AO_STENCIL_CHARACTER_MASK          4
#define HBK_AO_STENCIL_SKIP_NORMAL_OUTLINE     16
#define HBK_AO_STENCIL_SKIP_DEPTH_OUTLINE      32
```

・Characters have their own GBuffer stencil bitmask.

・Characters and environment have different toon lighting system.

・Important performance win for us to be able to distinguish character pixels without using custom stencils.

The character bitmask is an important bit in our GBuffer stencil.

*Play animation.

In Hi-Fi RUSH, characters and environment are both toon shaded, but have their own toon lighting systems.

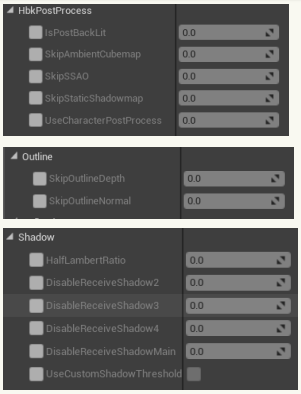
A lot of times, we want to skip the environmental lighting from affecting characters.

During battle, the number of characters drawn on screen and the screen space they occupy can become large. Being able to distinguish character pixels without an additional render pass was an important performance win for us.

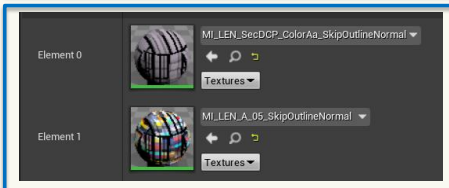
GBuffer Stencil (Material Parameters)

```
// GBUFFERAO GBUFFER STENCIL BITS (Partial List)

#define HBK_AO_STENCIL_CHARACTER_MASK          4
#define HBK_AO_STENCIL_SKIP_NORMAL_OUTLINE     16
#define HBK_AO_STENCIL_SKIP_DEPTH_OUTLINE      32

// SPECULAR GBUFFER STENCIL BITS (Partial List)

#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_MAIN                1
#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_2                   2
#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_3                   4
#define HBK_SPECULAR_STENCIL_SKIP_SHADOW_4                   8
#define HBK_SPECULAR_STENCIL_ENV_SKIP_AMBIENT_CUBEMAP        16
#define HBK_SPECULAR_STENCIL_ENV_SKIP_STATIC_SHADOWMAP       32
#define HBK_SPECULAR_STENCIL_ENV_SKIP_SSAO                   64
```

＊Character Mask is automatically set depending on character materials.



GBuffer Stencil can be set through material instance parameters.



Artists prepare unique material instances for each GBuffer stencil combination.

＊Some Gbuffer stencil values can be set per-actor as well.

A lot of our GBuffer stencil values are parameters that artists use to skip the application of certain lighting passes such as shadows, ssao, toon outlines, etc to control the look of the toon shading.

We allow artists to set the GBuffer stencil values per-material and for some parameters per-mesh actor as well.

GBuffer Stencil In Action

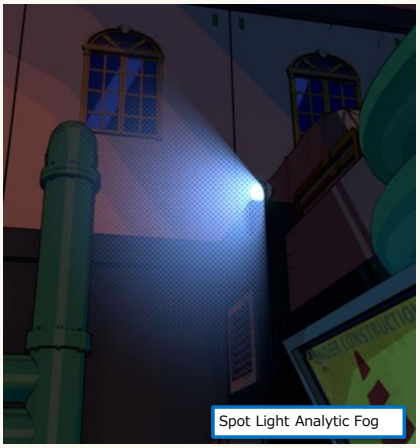
Here's an image showing our GBuffer stencil in action giving artists more control over how lighting layers are applied to the scene.

SSAO and static shadow maps are toggled on/off on the front door using GBuffer stencil flags. Notice that the ground uses a separate material from the door and is unaffected.

Agenda (For Stuff We Couldn’t Fit In The Talk)

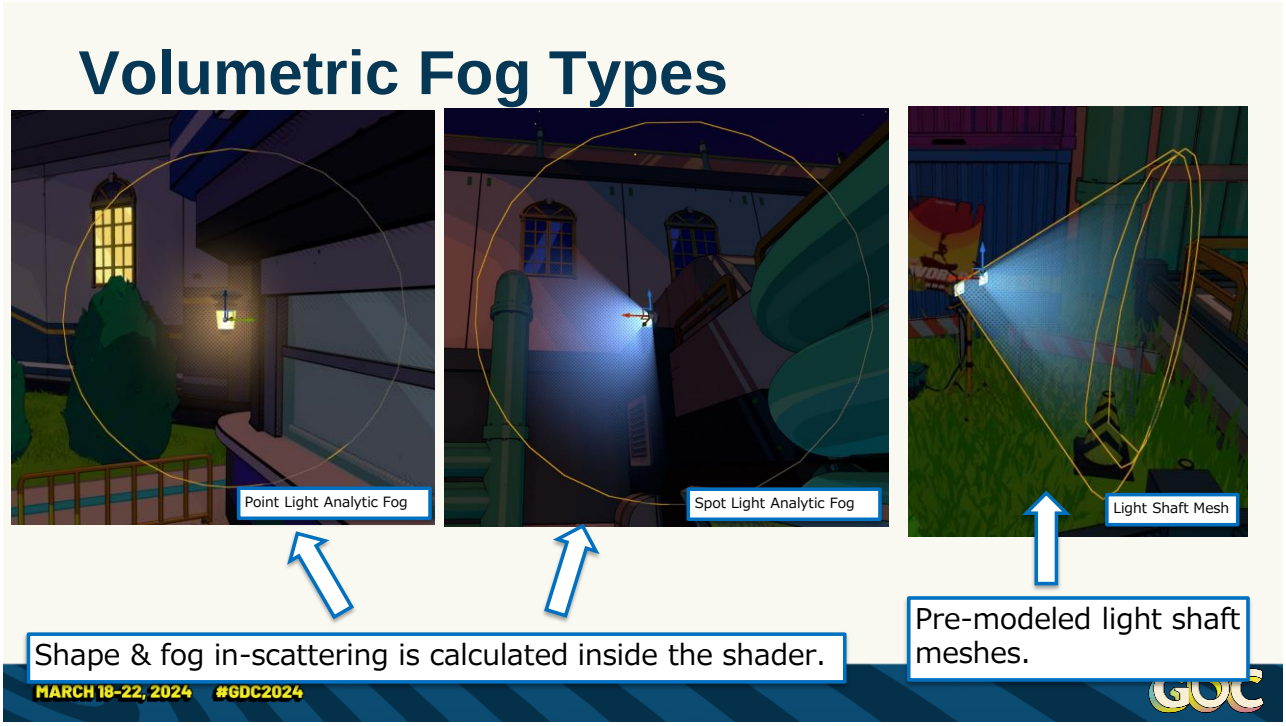
GBuffer Stencil
Volumetric Fog
Toon Lensflare
GPU Physics Simulation

What Is Analytic Fog?



Analytic fog is Hi-Fi RUSH's volumetric fog technique.

We called our volumetric fog technique analytic fog. The technique name might sound unfamiliar, but the algorithm we use is nothing new



There are two types of volumetric fog in Hi-Fi RUSH.

In one type we calculate the light shape and scattering inside the shader for a richer look. This is our analytic fog.

The other type we display a pre-modeled light shaft mesh. Both types are part of our single volumetric fog actor, the analytic fog actor, so that artists can place a single volumetric fog actor in the scene.

The first type looks better, but the second type is more performant, and artists have more freedom with the light shaft shape. Each are used accordingly and work well for our toon look.

Analytic Fog Actor Level Placement

Environment artists place analytic fog actors locally to light the scene with volumetric fog. Easy artist control in our toon environment was a big reason we choose our volumetric fog solution.

Analytic Fog Implementation Details

・Possible to implement without engine modifications.

・Combination of standard meshes + transparent material.

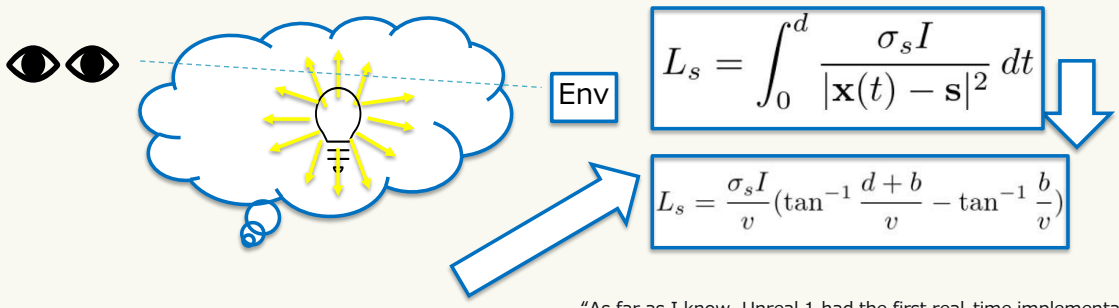
・Shader requires only depth buffer for world position reconstruction which UE4 transparent materials can access.

Our analytic fog shader requires only the scene depth texture meaning analytic fog can be implemented inside standard UE4 transparent materials.

UE4 has a froxel-based volumetric fog implementation with local control through vfx materials. Besides needing engine modifications, we noticed lowering the froxel resolutions could result in block artifacts which goes against our goals for a sharp clean look.

Fog Scatter Calculations

Inscattering Demo Blog Article
Inspired by Miles Macklin's blog article.



An equation that can be implemented in a UE4 custom node.

✗ Use raymarching to solve scattering.
○ Analytically solve the scattering integral.

"As far as I know, Unreal 1 had the first real-time implementation of volumetric dog. I used a formula table in an 1800's math book(!) to calculate a line integral on a 1/r^2 light falloff function using an arctan. ~200 cycles on Pentium, but fast enough on a 16x16 grid." Tim Sweeny's tweet

Apparently Unreal Engine 1's volumetric fog used similar calculations.

The rich look of our analytic fog shader comes from the fog in-scatter calculation.

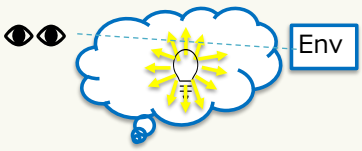
We calculate fog scattering by analytically solving the line integral for how much a point light source scatters through a fog medium before it reaches the camera.

Our method can be ALU intensive; the more expensive spot light analytic fog actor drawn at full-screen can cost around 1ms at 1440p on XSS, but it is performant enough to be used where effective.

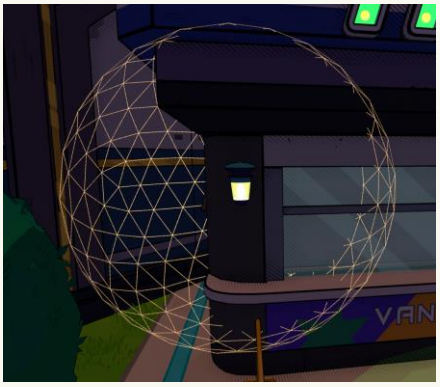
We decided against raymarched approaches, because we thought our solution would provide a cleaner look without any sampling artifacts.

Point Light Analytic Fog

Render a sphere shaped mesh.



Place a point light in the middle of sphere.



Draw mesh back face, so the camera can enter the sphere volume.

＊DepthWrite is on in the above screenshot to make it more obvious that the mesh backface is drawn. DepthWrite is off during the actual rendering.

For point light analytic fogs, we use a sphere mesh geometry.

A point light source is placed in the center of the mesh, and in-scattering without raymarching is calculated inside a UE4 transparent material custom node.

In order to handle fog meshes entering the camera, we cull the mesh's front face and draw the back face.

Point Light Analytic Fog Depth Calc

Depth write is disabled, so the fog pierces through backgrounds.

Scene Depth < Fog Volume Mesh Depth

If this happens.

Use the scene's world position in the fog calculations.

Because depth write is disabled, we do a depth check inside the shader, so that objects behind walls don't bleed through.

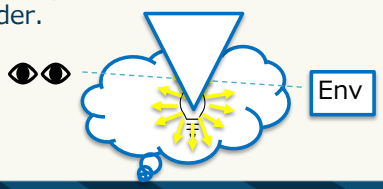
When the scene's depth is drawn in front of the fog mesh, we calculate the scene's world coordinates from depth and use that position instead of the fog mesh's in our fog scattering calculations.

Artists are careful with overdraw but point light analytic fogs have good performance and our artists use them widely across maps.

Spot Light Analytic Fog

Spot light analytic fog also render using a sphere mesh.

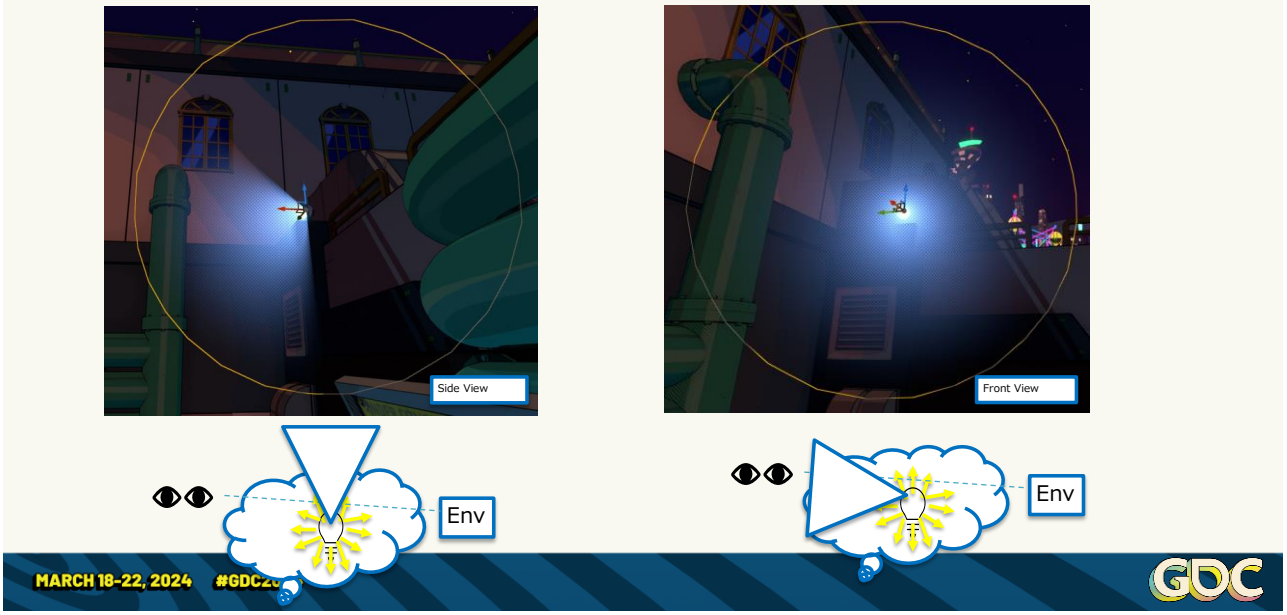
We place a point light source inside the center of the sphere.
The cone shape is generated inside our shader.
Cone-ray intersection is calculated inside the shader.



Our spot light analytic fog has a similar setup to our point light type.

We use a sphere mesh as the fog mesh and the spot light's cone shape is generated by calculating a cone-ray intersection inside the shader.

Spot Light Analytic Fog Aperture



We use a sphere mesh to be able to more dramatically render spot light analytic fogs with a wide aperture. Procedurally generating meshes such as a sphere-tipped cone may have been more efficient and our spot light mesh is something that could have been improved on.

Viewing Spot Light Analytic Fog From Different Angles

This is what our spot light analytic fog looks like with camera movement.

The scattering shape correctly widens to reflect the true width of the spotlight cone and looks pretty cool with camera movement.

Like mentioned before, spot light analytic fog can be costly when looked at head on. 1 ms is a big cost for a 60FPS game, so they are selectively used in spots where they are effective.

Agenda (For Stuff We Couldn’t Fit In The Talk)

GBuffer Stencil
Volumetric Fog
Toon Lensflare
GPU Physics Simulation

Implementing Game-Side Render Passes In UE4

We want to add original render passes without UE4 engine modifications.

What to do?

Use render commands to sync rendering data between threads.

+

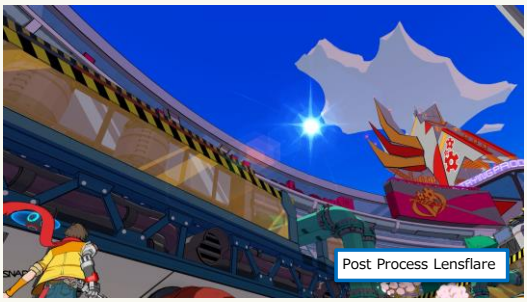
Use engine-side render thread delegates to call game-side render passes.

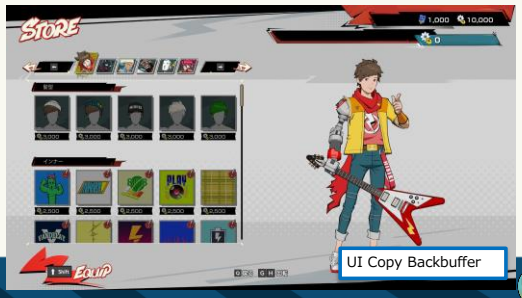
For our post process lensflare, we added render passes to our game-side code without engine modification.

We used standard UE4 render commands to sync rendering data between the game thread and the render thread.

We utilized render thread delegate callbacks to execute from the engine-side render thread, render passes implemented in our game module.

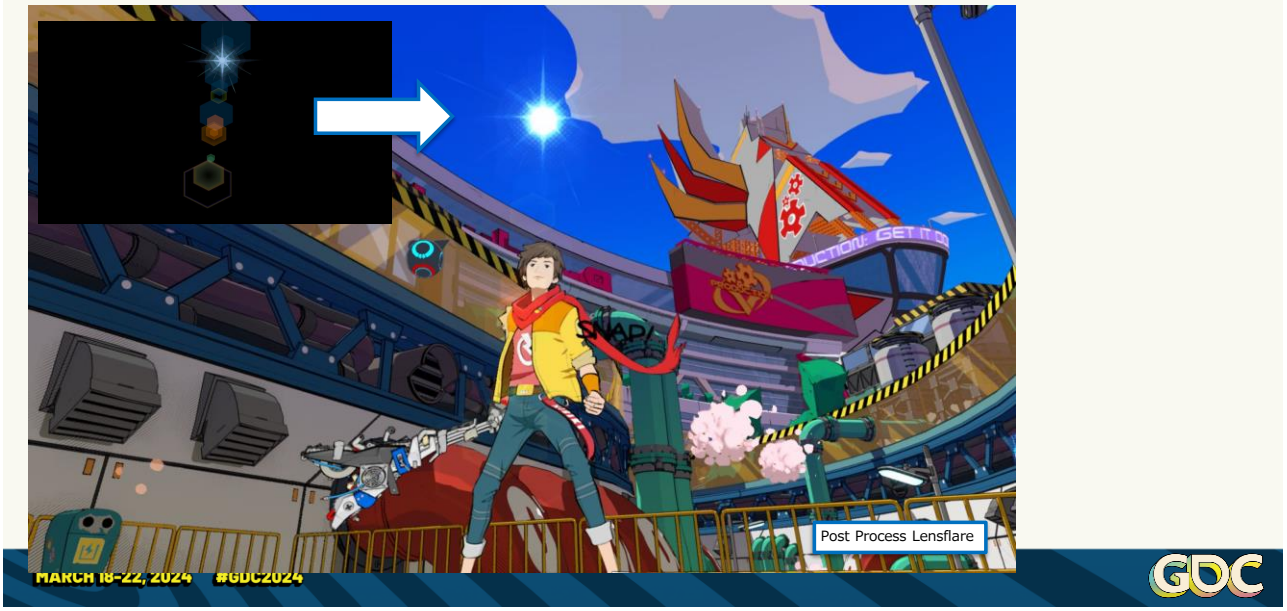
Render Thread Delegate Render Passes





We implemented various render passes game-side using render thread delegates.

Post Process Lensflare



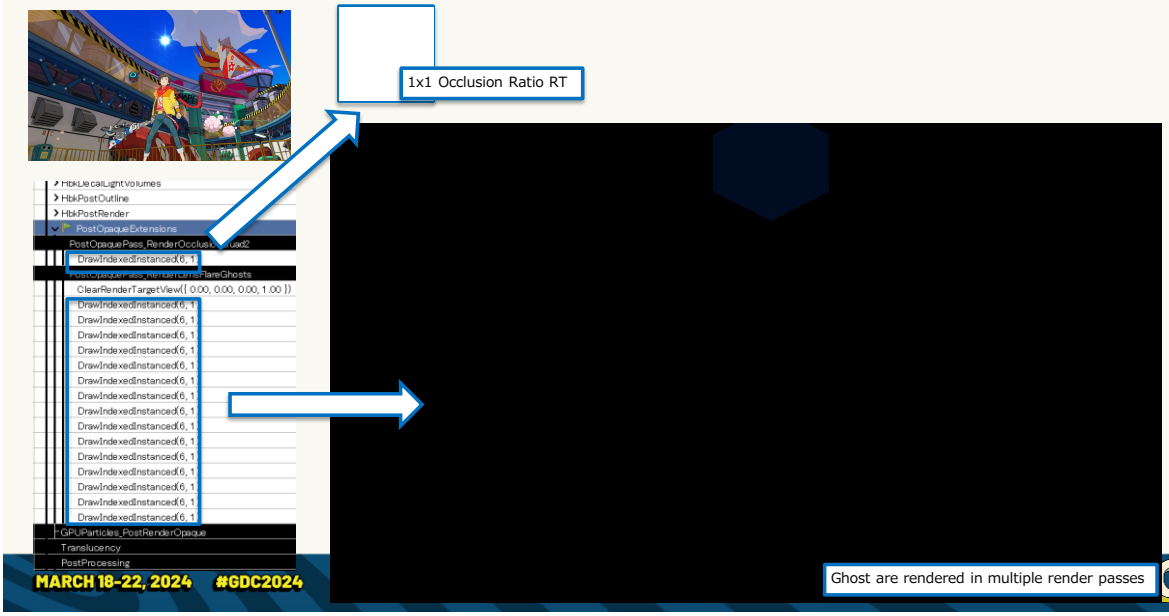
UE4 has a post process lensflare implementation. However, for Hi-Fi RUSH, art requested a lensflare implementation where the lensflare ghosts could be drawn with artist prepared textures.

Taking A Look At Our Post Process Lensflare

The goal of our post process lensflare is to show camera lens ghosting that can occur from strong light sources.

The ghost sizes, intensity, and screen space position changes depending on where the light source is relative to the camera.

Post Process Lensflare Render Passes



The slide shows a gpu capture of post process lensflare render passes.

A post process pass calculating the lensflare occlusion ratio run first then each individual ghosts are rendered as individual quads.

Post Process Lensflare Final Results

The slide shows the final rendering result. All lensflare ghosts have been rendered into a single render target.

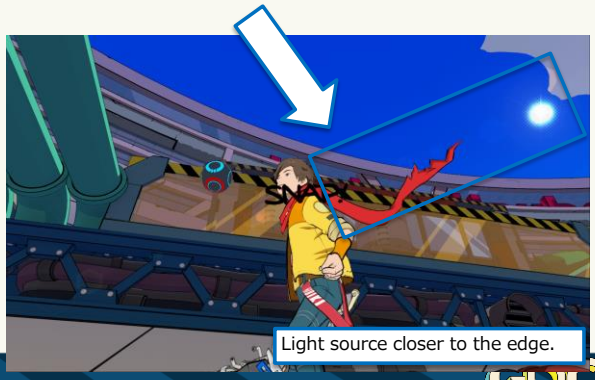
Ghost Quad Rendering

Ghosts are drawn along a line between the screen center and the light source.

Our reference:
Yossarian King. 2000. “2D Lens Flare”. Game Programming Gems 1.

Ghosts are drawn smaller and dimmer, the further away the light source is from the screen center.

Light source in screen center.



Light source closer to the edge.

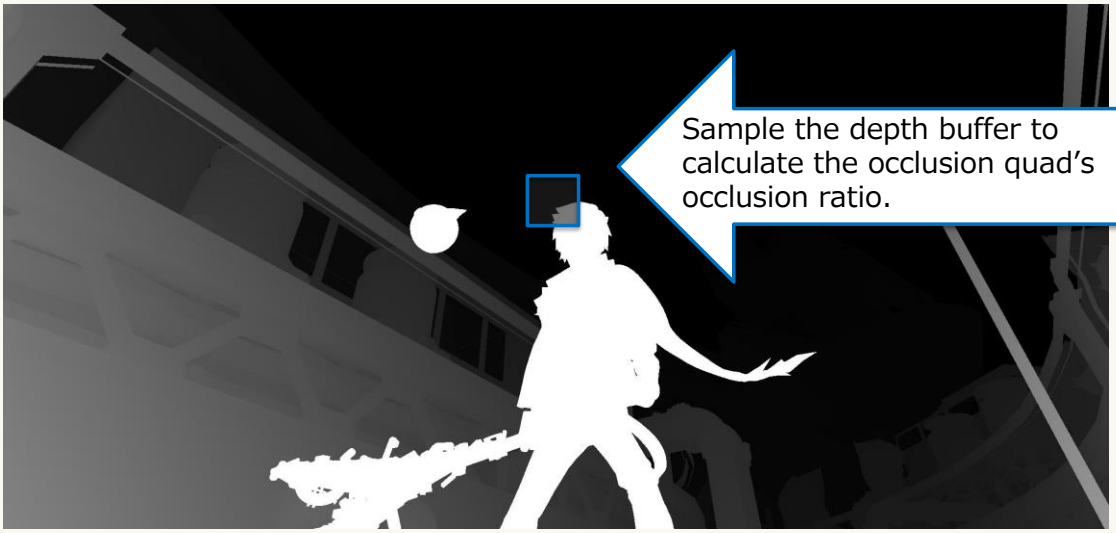
MARCH 18-22, 2024 #GDC2024

The rendering algorithm for our lensflare ghost is a classic technique. We used Game Programming Gems 1 as a reference for our implementation.

Lensflare Occlusion

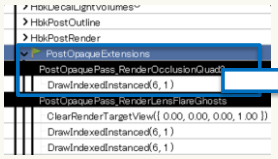
When the lensflare light source is occluded, we want the lensflare to fade depending on how occluded the light source is.

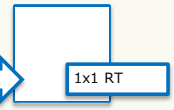
Calculating Lensflare Occlusion



It's possible to use gpu hardware occlusion querys to calculate the light source's occlusion ratio. However, for our game, we wrote a shader to calculate an occlusion quad's occlusion ratio using the depth buffer. Using a shader has lower UE4 implementation costs and it also avoid hardware GPU query's 1 frame lag due to CPU/GPU query result handoff.

Calculating Lensflare Occlusion Ratio





The occlusion quad's depth buffer area is sampled 32 times to calculate the occlusion ratio.

The more occluded the occlusion quad, the more the ghost's intensity is lowered.

The occlusion quad's depth buffer is sampled with a poisson distribution in 32 locations. The depth buffer values are compared to the occlusion quad's center depth value and the occlusion quad's occlusion ratio is calculated. The occlusion ratio result is written out to a 1x1 render target.

The subsequent ghost quad render passes sample the 1x1 occlusion ratio rendertarget and adjust each ghost's intensity accordingly.

This is not an original technique and though I can't find my original reference, it's a technique that I've seen discussed in various places.

Merging the Lensflare Results

The lensflare rendertarget is merged with the scene color inside a UE4 post process material rendered after tonemapping to complete the lensflare rendering.

Agenda (For Stuff We Couldn’t Fit in The Talk)

GBuffer Stencil
Volumetric Fog
Toon Lensflare
GPU Physics Simulation VFX

GPU Physics Simulation VFX

From here we will move on to the GPU physics simulation section, thank you.

Later Stage Battle Spoiler Alert

If you don't like spoilers, cover your eyes and ears!

In this section, we will talk about the technology used in a battle in one of the final stages.
Spoiler warning!
If you want to avoid spoilers, I think it's best to close your eyes and cover your ears.

Let’s watch a video first.

In the boss battle at the end of the game, there is a scene where you fight in a room full of coins.

We needed a coin effect that would react to the movements of the characters and gimmicks, but it would be difficult to create effects manually for every movement, so we considered implementing it in as a physics simulation.

Coin Effect Requirements

- Scattering in response to the movement of characters or gimmicks
- Want simulation scale and volume
- Can't let performance become a problem
- Natural but dramatic movements like that of a hand-made effect

The coin effect required:

First, coins needed to scatter and move in response to the movements of characters and gimmicks.

The scene called for a large number of coins.

Couldn’t let performance issues become a problem.

At the same time, wanted natural movements that could match the quality of hand-animated effects.

GPU Rigid Body Simulation

Our coin effect ended up as a simple rigid body physics simulation implemented by Niagara and running on the GPU.
Niagara is the runtime VFX module of Unreal Engine.

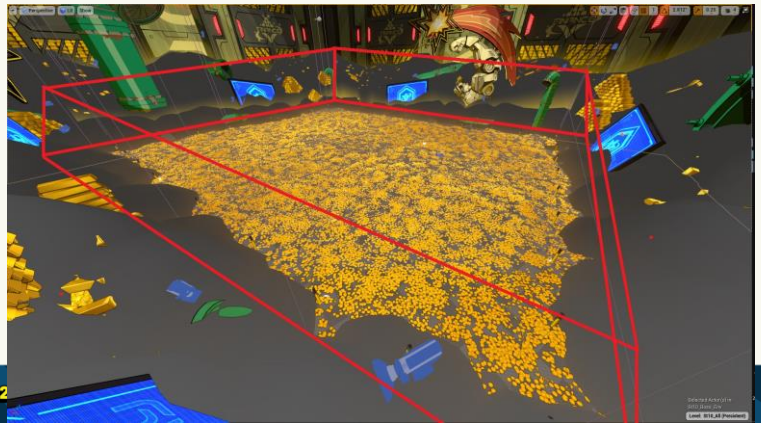
In the following slides, we will explain our implementation.

This is a still image taken in photo mode.

It's hard to see because of the glitter, but I think you can see that there are a lot of coins flying around.

Physics Simulation Scene Setup

- Set a box to cover the area where you will fight the boss.
- Set 64,000 coins in the box.
- Collisions with characters and gimmicks to create scattering effects.



This section explains the scene setup for the physics simulation.

First, I’m setting up a box to cover the area where you'll fight the boss.

64,000 coins are spawned and simulated inside the box.

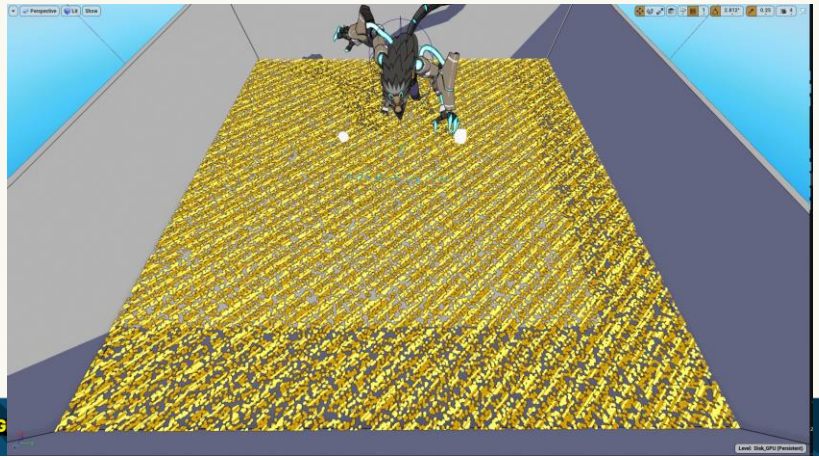
The coins collide with characters and gimmicks inside the box and scatter creating the coin simulation effect.

(The coins you see in this image are static mesh foliage coins covering the surface and are not the simulation coins.)

Physics Simulation Scene Setup

Initial placement of coins is random.

If the placement is in a grid pattern, the scattering trajectory of the coins will be too symmetrical and unnatural.



The initial placement of coins is not in a grid pattern, but rather in a slightly random placement as shown in the image.

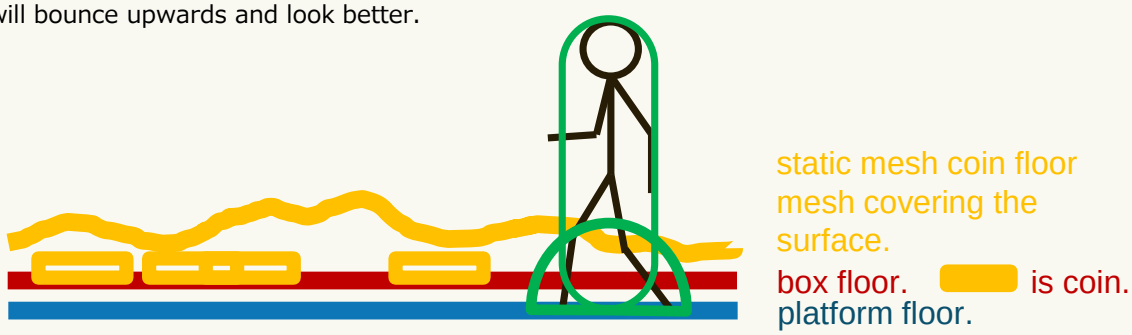
This is done to add noise to the trajectory because if it were in a grid pattern, a strong symmetry would appear in the trajectory of the coins scattering and it would feel unnatural.

Physics Simulation Scene Setup

- The simulation box floor is sandwiched between the static mesh coin floor and the platform floor.
- Coins being simulated are not visible to the user when sleeping on the floor.

→ By not showing the coins that fell on the box floor, you can prevent coins from falling into each other and hide the lack of quantity.

→ By raising the collision box floor, scattering from collision with the player collider sphere will bounce upwards and look better.



The simulation box is placed between the scaffold floor and the coin floor mesh covering the surface.

The only time the simulated coins are visible to the user is when they pop out of the static mesh coin floor.

The reason for this setting is to avoid ruining the appearance.
Even though there are 64,000 coins, this number is not enough quantity to cover the entire surface.

Also, since collisions between coins are not calculated, coins may penetrate through each other.

These problems are hidden by the static mesh coin floor.

The reason why the box floor is higher than the platform floor is because this creates a better effect as the coins scatter upwards.

The Setup For Physics Simulation

- Since it uses rigid body physics, it also simulates the rotation of a coin.

  Use a moment of inertia that matches the shape of the coin.

- Coins collide with the following:

✓ Box

✓ Dedicated collider added to characters and gimmicks.

Spheres and capsules are supported as collider shapes.

The objects in the scene are setup as following:

Since coins are treated as rigid bodies, rotational forces are also simulated.

I am using the moment of inertia that matches the shape of the coin.

Coins collide with boxes and dedicated colliders added to characters and gimmicks.

The collider shape supported are spheres and capsules.

For example, the player character has a capsule and sphere collider set up like in the picture.

PBD References

- Used Position Based Dynamics for physics simulation. (hereinafter referred to as PBD)

  Used due to robustness of behavior.

- Referenced paper.

  Matthias Müller, Miles Macklin et al.

  Detailed Rigid Body Simulation with Extended Position Based Dynamics.

  ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2020

For physics simulation we used Position Based Dynamics.

I used PBD because of its robust behavior.

In my implementation, I mainly referenced the paper in the slide.

This is a relatively new paper which is about extending PBD to rigid body simulation.

In this session, we will omit a detailed explanation of PBD.

Physics Simulation Implementation

We do the following every frame:

- Colliders update
- Sleep states update
- Integrations
- Collision constraints solving

This is an overview of our physics simulation implementation.

We do the following every frame:

Colliders update
Sleep states update
Integrations
Collision constraints solving

I will explain each step in the following slides.

Colliders Update

- the box is immovable

- Colliders of characters and gimmicks can move and enter/leave.

✓ Colliders inside the box with specific collision channels are detected every frame.

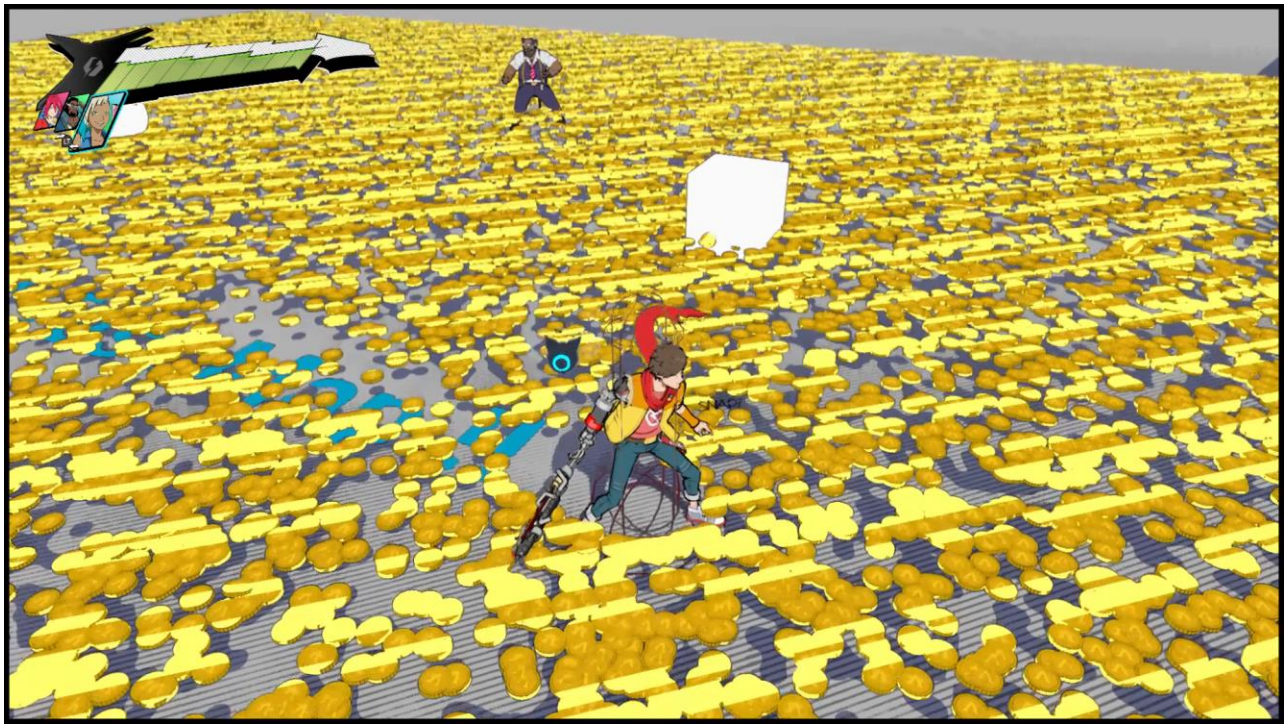
✓ Pass information about the shape, position, orientation, and size of the colliders to the simulation.

First, our colliders update.

The box that covers the entire scene does not move.

On the other hand, the collider attached to characters and gimmicks can move around and enter and leave the simulation.

Therefore, every frame, the CPU side implementation detects colliders with a specific collision channel and passes the information about the shape, position, orientation and size of colliders to the simulation.



Next is the sleep state update.

Please watch this video first.

Out of the 64,000 coins, many coins do not need to be moved, so I implemented a sleep state.

If speed drops below a certain level, coins will go to sleep.

Coin speed is naturally attenuated by the dynamic friction of the floor.

Once coins go to sleep, I return them to their initial placement position.

The reason for this is because if you leave the coins scattered, coin density will become uneven within the box, creating areas where the coin effect will not be produced.

Sleeping coins will wake up when a collider such as a character or gimmick approaches with a certain distance.

Sleep States Update

- 64,000 coins, many coins are not moving.

  Cull rendering and stop simulations of sleeping coins.

  Reduce load. Rendering is culled by Particles.MeshIndex of Niagara.
- Goes to sleep when speed drops below a certain level

  The speed is attenuated by the dynamic friction of the floor.
- Return to initial placement position after sleep

  If you don’t return, there will be unevenness in areas with many coins and areas with few coins.
- When character and gimmick colliders get within a certain distance of sleeping coins, they wake up.

Cull the sleeping coins from the simulation.
In the video, sleeping coins were being rendered, but in reality, they are actually culled from rendering.
You can cull meshes from the Niagara Mesh Renderer by altering the value of Particles.MeshIndex.

Integrations

Normal PBD integration.

Since it is a rigid body, it also handles direction and angular velocity.

Next are the integrations.

Basically the same as normal PBD integration, but since it is a rigid body, it also handles orientation and angular velocity.

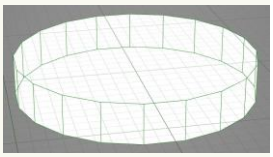
(In our implementation, there is no external force that would result in torque.)

Collision Constraint Solving

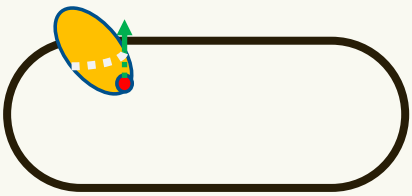
Collision judgment

Normal rigid body simulation: Handled with cylindrical mesh shape

→ Calculating collision for this shape can be too costly for our performance targets.



Our solution: Use disk shapes without thickness. The collision impact point is the deepest point in the disk.



Finally, collision constraint solving.

I think that in normal rigid body simulation implementations, coins are often treated as cylindrical meshes, as shown in the image on the right.

However, this would require calculations for multiple vertices and edges, which would increase the load.

Therefore, we included two approximations in our implementation.

The first is that coins are treated as disk shapes without considering the thickness.

Second, only the deepest point in the disk is treated as a impact point.

As shown in the figure, when the coin is penetrating into the capsule, the impact point, the penetration depth, and the normal of the collider at the closest point are calculated from the red point that is the deepest penetration point in the disk.

Collision Constraint Solving

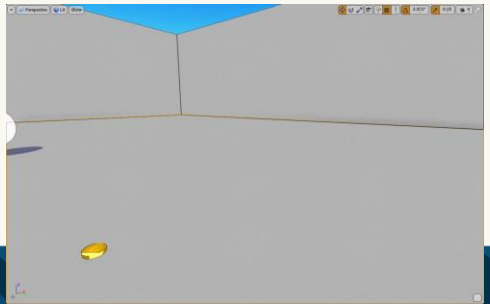
Apply the following to collided coins:

Extrusion of penetration

Repulsion, dynamic friction (box only)

Repulsion and dynamic friction were not used in the characters and gimmicks because there was no problem with the expression even if they were not included.

* The box and collider do not move even if they collide. Treated as infinite mass.



If the coin collides with the box, apply an extrusion of penetration, dynamic friction, and repulsion.

If the coin collides with a character or gimmick collider, only the extrusion will be applied.

For characters and gimmicks, simulation results lacking repulsion and dynamic friction were acceptable, so are omitted as a performance optimization.

Boxes and colliders do not move when they collide. Only coins are move.

As a result of the above implementation, the behavior is like what is shown in the slide video.

By including two approximations, the physical accuracy of the behavior has decreased, but since there are a large amount of coins, it’s not noticeable.

This concludes the explanation of the implementation of the physics simulation.

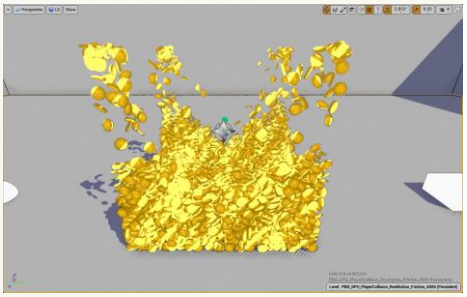
Static Friction And Coin Deposition

- Dynamic friction was sufficient, so I did not include it.
- Essential for pile expression

Pile cannot be maintained without static friction to collision constraints between coins.

- Initially, I tested coin piling, but gave up due to our strict performance limitations.

If we calculate collisions between coins, even with spatial division structure optimizations, the calculations were too costly for a large scale coin simulation.



Static friction was also removed because dynamic friction alone was sufficient to express friction.

Collision and static friction between coins are essential for convincing coin piling.

In the early stages of our implementation, I actually tested collisions between coins.

The video shows what happened at that time.

However, even using spatial division structures, the performance cost was quite large, and we had to conclude that coin piling wasn't doable for our simulation scale with our strict performance limitations.

Other Optimizations

- MeshRenderer frustum culling

- The size of the coin is unrealistically large.

✓ We want to reduce the number of simulations.
✓ We also want to reduce the rendering load on the coin floor.
✓ It's difficult to see in real size.

→ Taking the above 3 points into consideration, a larger but not unnatural looking size was settled on.

I'm using frustum culling for the MeshRenderer.

Also, the fact that our coins are unrealistically large is an optimization.

If the coins were of real size, the number of simulations would increase, making it difficult.

In addition to simulations, there are also issues such as the rendering load of the coin floor, and the fact that realistic sizes were actually hard to see.

The size was decided so that it would not look too unnatural.

Performance

- Measurement condition

Xbox Series S　Test build 1440p　64000 pieces

Measured at the timing when many coins are scattered.

- Measurement result

✓ Simulation

Simulation by Niagara : 0.40ms

✓ Rendering (increased processing time due to rendering of coins. coins do not have shadows.)

PrePass : 1.10ms → 1.27ms

Velocity : 0.40ms → 0.64ms

BasePass : 2.08ms → 2.27ms

Total processing time for coin effect is 1.0ms.

Performance measurement results.

The measurement conditions are XboxSeries S, test build, resolution 1440p, using 64000 coins.

During the boss battle, I selected a scene where a lot of coins were scattered and measured it.

Measurement results were the following:

The simulation by Niagara takes 0.40 milli-seconds.

The rendering load is mainly from PrePass, Velocity and BasePass.

This is a comparison of the processing time when there is no coin rendering and when there is.

In total, the coin effect processing time is 1.0 milli-seconds.

I think I have achieved my goal of not letting performance hold me back.

Lies To Look Good

- We want to rotate more.
  → Scale parameter of moment of inertia
- Symmetry is still visible in the scattering trajectory.
  Collision shapes are box, sphere, capsule, and disc, have symmetry.
  So the scattering trajectory has symmetry.
  → Set the maximum speed value and vary the value for each coin.
  (Only in height direction, there is no limit to the speed at which the coin falls.)
- There are action instances where the effect quality is insufficient with simulation alone.
  → Supplement with additional artist VFX.

If you just do a straightforward physics simulation, the movement will be too symmetrical and feel unnatural.

In the first place, although the purpose is to reduce the processing load of collision itself, since we only have primitive shapes such as spheres and capsules, it can make movements unnaturally symmetrical.

Here I will introduce what I did to get as close to the quality of hand-made effects as possible.

・Moment of inertia scale parameter

I created a parameter to scale the moment of inertia, increasing the speed of the coin's rotation and emphasizing its movement.

・Maximum speed

By setting a maximum speed value and varying the value slightly for each coin, I tried to break the symmetry of movement.

However, it felt strange if the coin fell slowly in the height direction, so I decided not to limit the falling speed.

・Supplement with hand-made effects

VFX artists thought that the quality was insufficient with simulation alone in some action scenes, and hand-made effects were added these scenes.

Recommendations For Implementing GPU Physics Simulation

- Implement CPU side first.

  Shader debugging for physics simulation tends to be more difficult than rendering.

  Do sufficient test and implementation on CPU side where it is easy to do stepping execution.

  Port them to GPU when you removed all bugs.

- Perform the simulation in local coordinates, not world coordinates.

  float have precision of 7 or 8 digits as a decimal number.

  If you move 100km away from the origin, there will be an error of several millimeters due to the accumulation of calculations.

  On a coin scale, even an error of a few millimeters causes unstable behavior.

  Do simulation in the local coordinate system, and only rendering in the world coordinate system.

Based on my experience, I recommend the following two points for those who want to implement GPU physics simulation in the future.

The first is to implement things that run on the CPU side first.
We recommend implementing it on the CPU side first and porting it to the shader once it has been tested and bug-free.

This is because debugging shaders is difficult.

When using a shader for drawing, I think it is common practice to debug it by writing the progress to a buffer and displaying it on the screen.

However, in physical simulations, debugging is often difficult unless you can step-execute and check the physical quantities.

The amount of implementation will be doubled, but I think the time to develop will actually decrease in the long run.

Second, to avoid numerical error problems, it is better to run simulations in local coordinates rather than world coordinates.
When calculating using float, the precision is 7 or 8 decimal digits, and errors may occur if calculations are repeated in world coordinates at a location

about 100 km away from the origin.

On a scale the size of a coin, even an error of a few millimeters can cause the behavior to become unstable.

It is safe to proceed with calculations in the local coordinate system as much as possible, and only make the final output passed to the rendering in world coordinates.

GPU Physics Simulation PC Settings

- The default low is 64,000 coins. Medium is 128,000 coins. High is 256,000 coins.
- I haven't seen any reviews that understand the effects of this option.
- If it is high, there will be too many coins scattered, making it difficult to see the player character, so you can also adjust the difficulty of boss battles by this option.

There is a setting to increase the number of coins in the coin simulation.

The default low setting is 64,000, but the medium setting is twice as many at 128,000, and the high setting is 256,000.

If you play on high, there will be too many coins and it will be difficult to see the player character, so you can not only adjust the flashiness of the effects but also the difficulty of the boss battle.

Like this.

I've seen many reviews and impressions of Hi-Fi RUSH, but I've never seen any mention of this setting, so it still remain a mysterious one for Hi-Fi Rush users.

This concludes the GPU physics simulation section.