Lecture 07

Rendering on Game Engine

Render Pipeline, Post-process and Everything

WANG XI

GAMES 104

2022

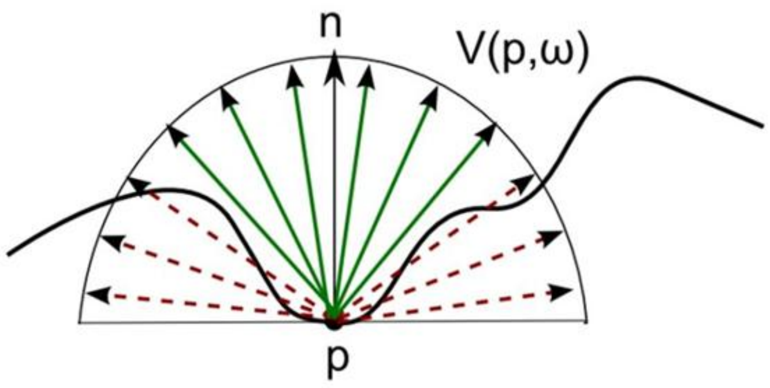
Ambient Occlusion


AO Off


AO On

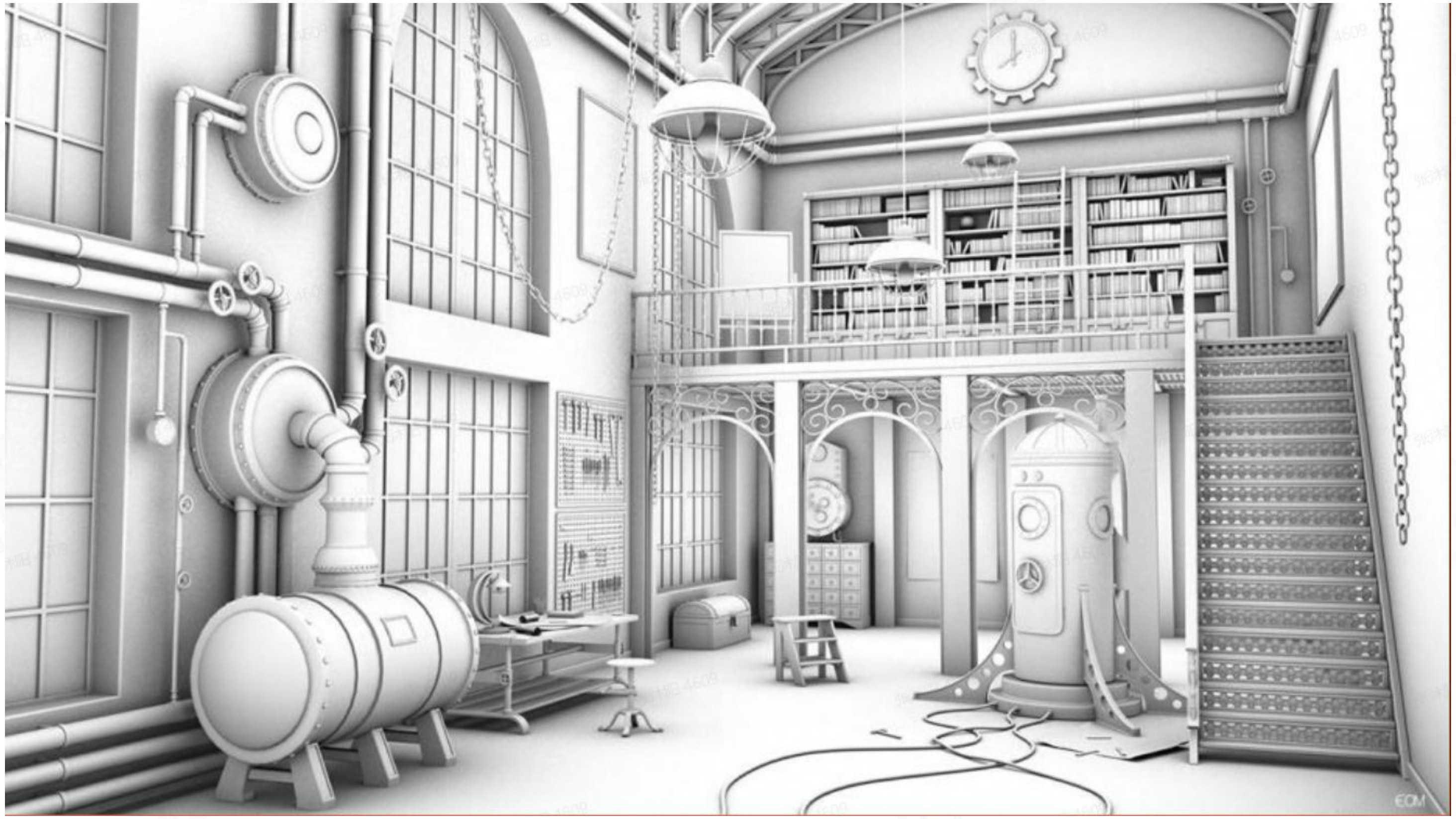
Ambient Occlusion

- Approximation of attenuation of ambient light due to occlusion



$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} \boxed{V(\mathbf{p},\omega)}\,\boxed{\mathbf{n}\cdot\omega}\, d\omega,$$

Visibility Along W **Normal Weight**



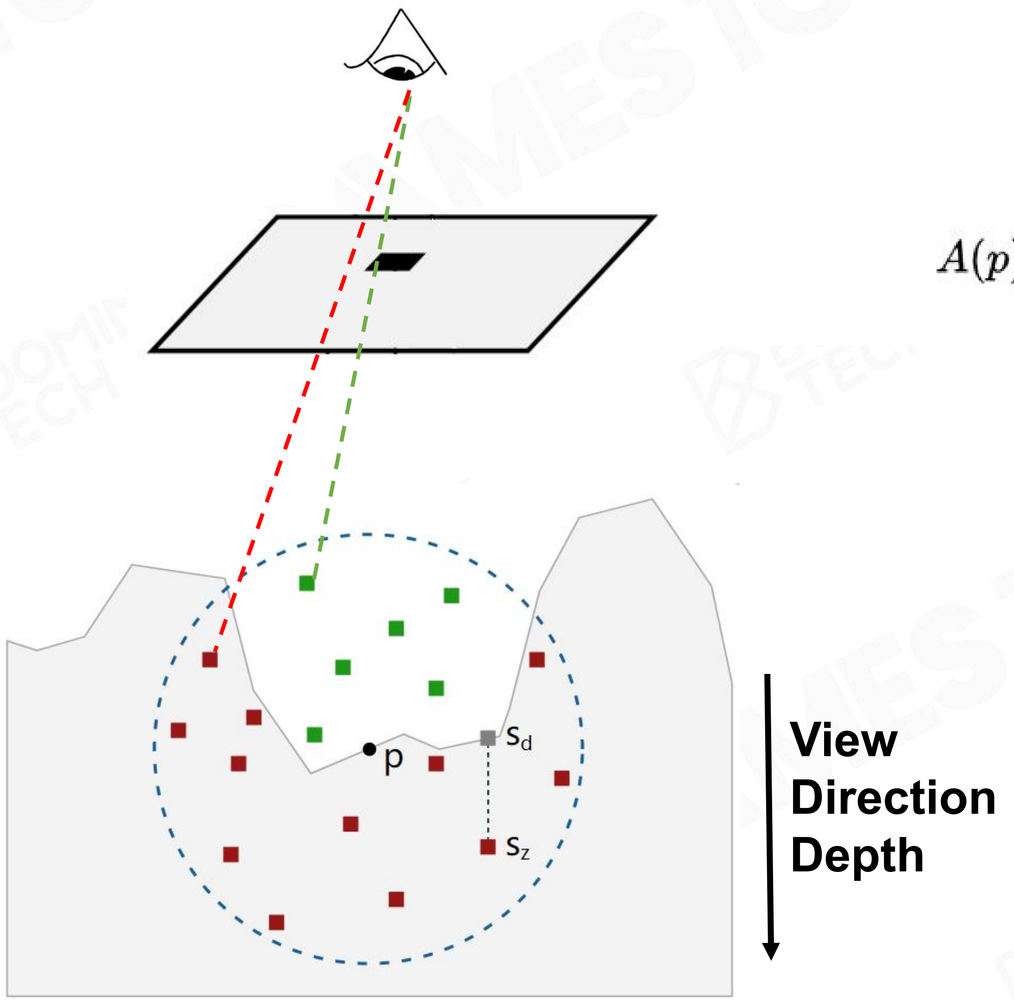
Precomputed AO

Using ray tracing to compute the AO offline and store the result into texture, which is widely used in object modeling process

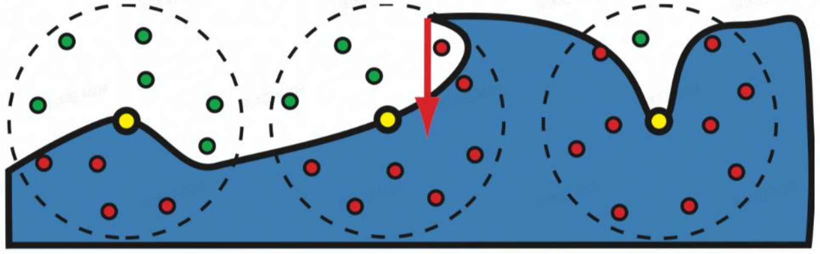
- Extra storage cost
- Only apply to static object

Original model | With ambient occlusion | Extracted ambient occlusion map

Screen Space Ambient Occlusion (SSAO)



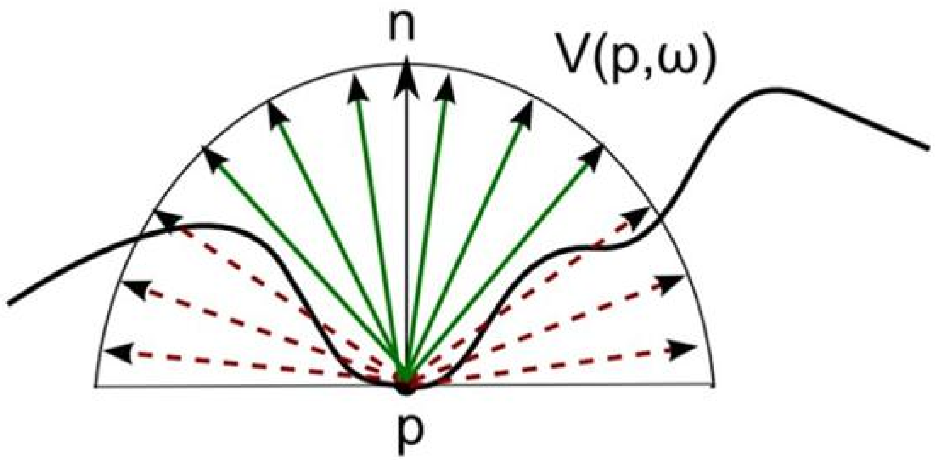
$$A(p) = 1 - \frac{Occlusion}{N}$$



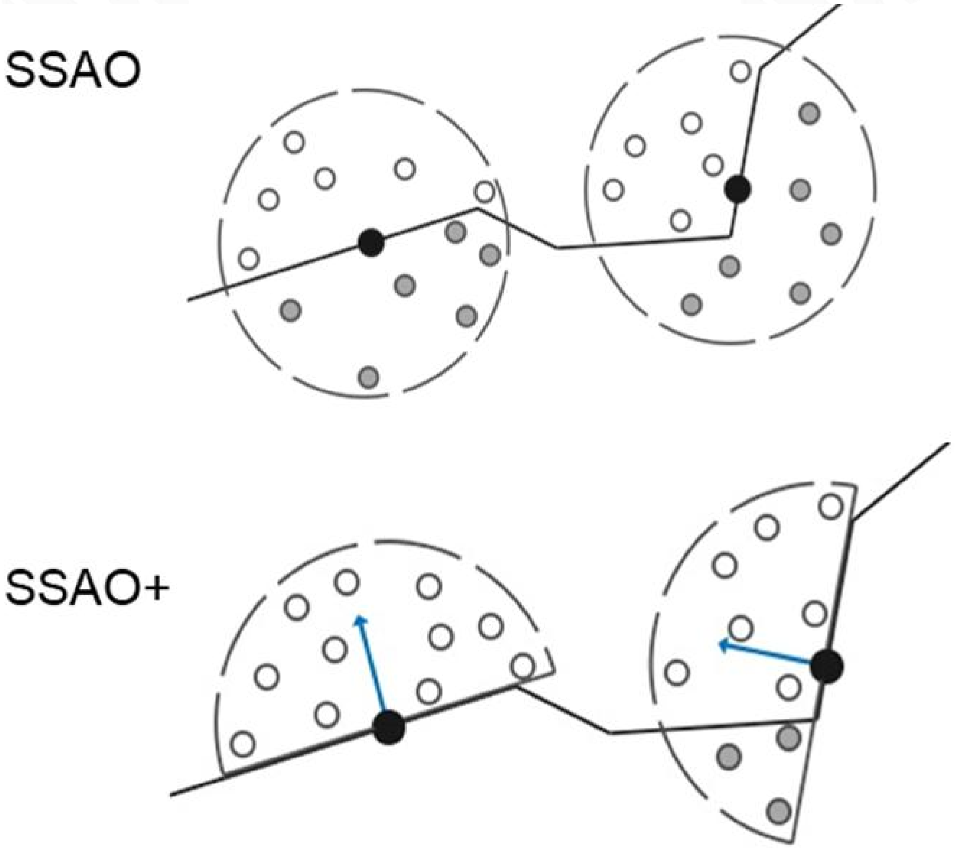
- Generate N random samples in a sphere around each pixel p in view space
- Test sample occlusions by comparing depth against depth buffer
- Average visibility of sample points to approximate AO

SSAO+

- Recall the AO equation is acutally done on the normal-oriented hemisphere



$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\, d\omega,$$

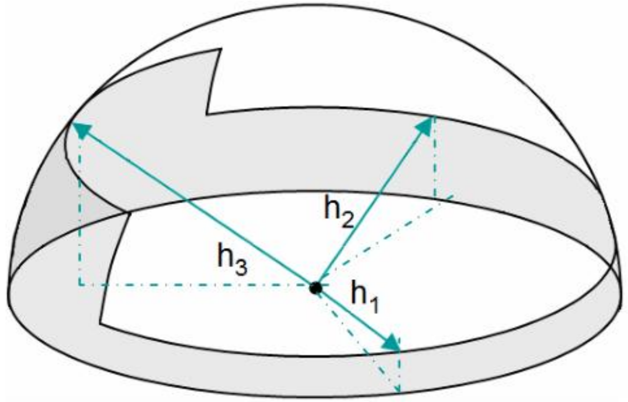



update clip data -1
SSAO+ Off


update clip data -1
SSAO+ ON

HBAO - Horizon-based Ambient Occlusion

- Use the depth buffer as a heightfield on 2D surface
- Rays that below the horizon angle are occluded

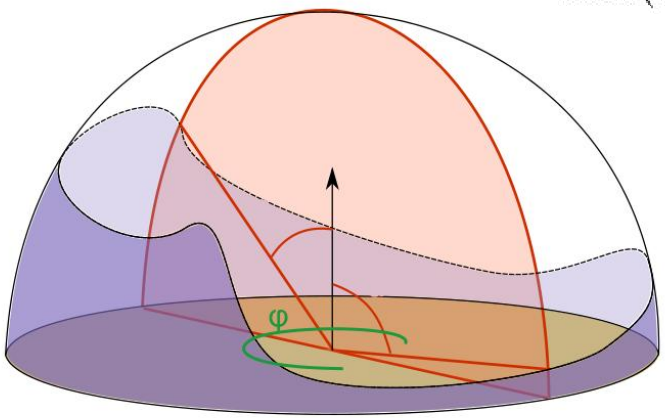


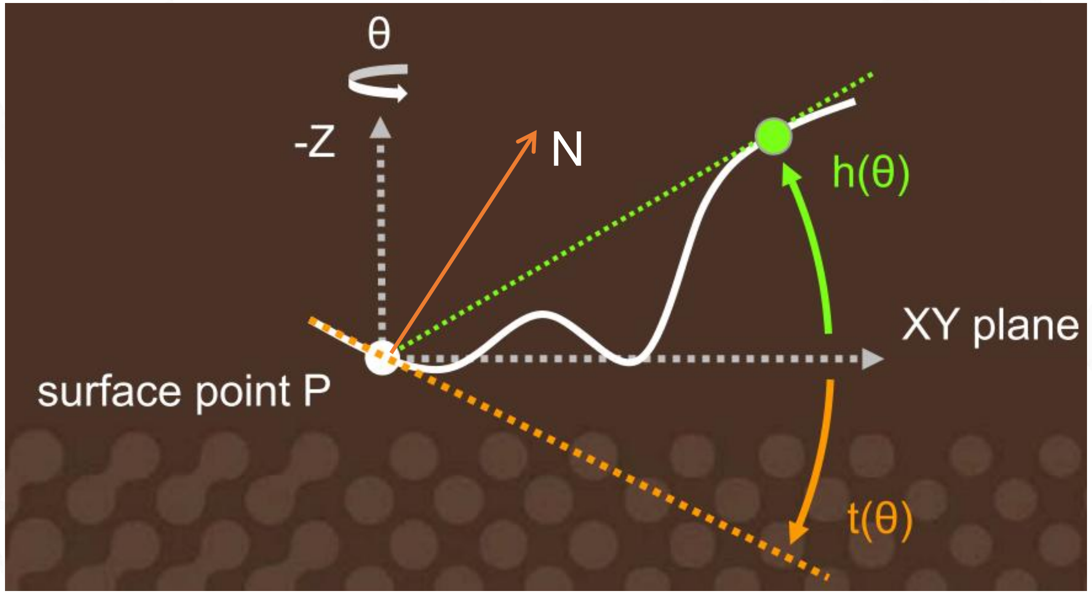
Occluded Area

$$A = 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} \int_{\alpha=t(\theta)}^{h(\theta)} \boxed{W(\vec{\omega})} cos(\alpha) d\alpha d\theta$$

attenuation function

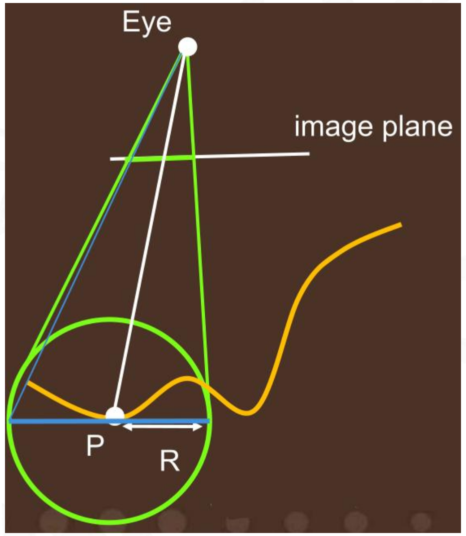
$max(0, 1 - r(\theta)/R)$



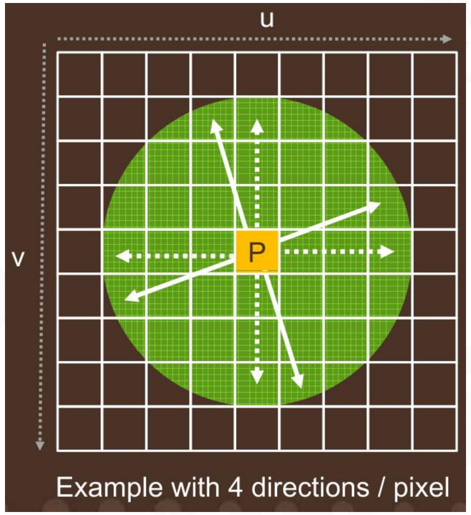


HBAO Implementation

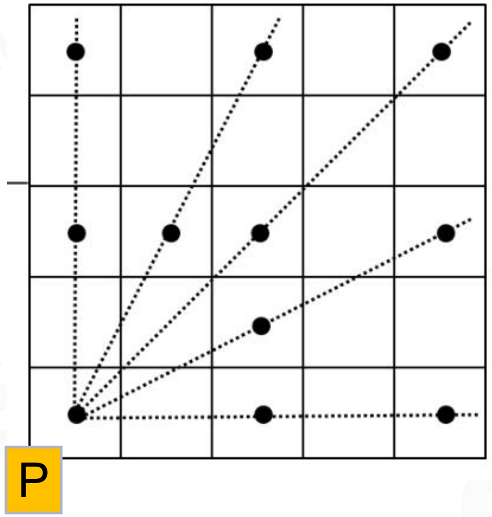
- Use the depth buffer as a heightfield on 2D surface
- Trace rays directly in 2D and approximate AO from horizon angle



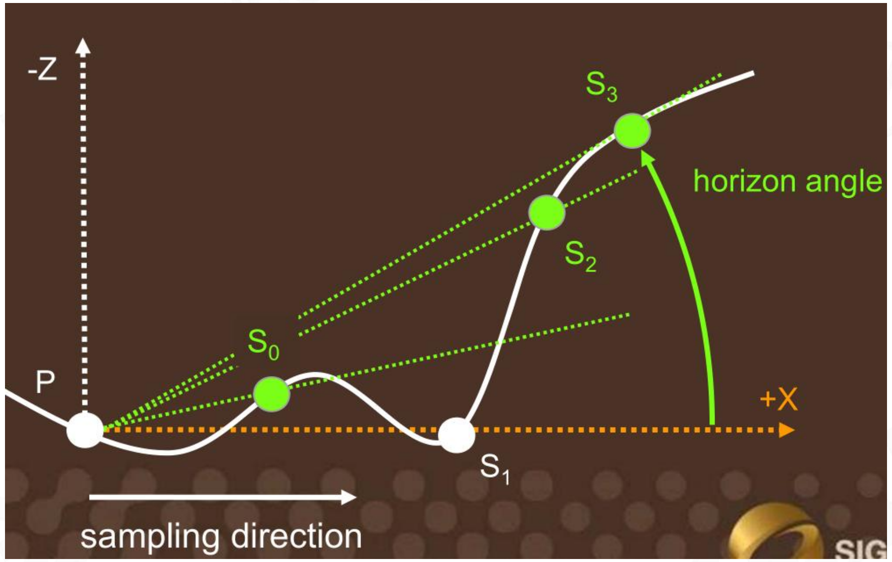
Depth Image



Ray Marching



Randomly jitter the step size and randomly rotate the directions per pixel



Find the max horizon angle

GTAO - Ground Truth-based Ambient Occlusion

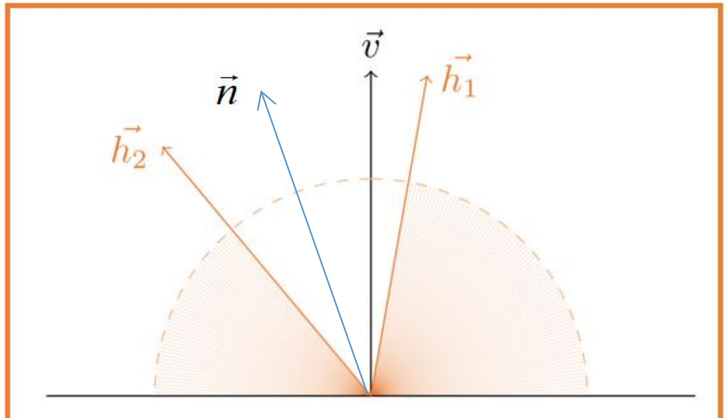
GTAO introduces the missing cosine factor, removes the attenuation function, and add a fast approximation of multi bounce

cosine factor

$$\hat{A}(x) = \frac{1}{\pi}\int_0^{\pi}\int_{\theta_1(\phi)}^{\theta_2(\phi)} \boxed{\cos(\theta-\gamma)^+}\,|\sin(\theta)|\,\mathrm{d}\theta\,\mathrm{d}\phi$$

$$\gamma = angle(\vec{n},\vec{v})$$

Analytic solution per slice



Side View (Slice)

GTAO
Cosine + Single Bounce

Monte Carlo Ground Truth

GTAO
Cosine + Colored Multi Bounce

GTAO - Ground Truth-based Ambient Occlusion

Add multiple bounces by fitting a cubic polynomial per albedo

$$V_d' = f(V_d) = ((aV_d + b)V_d + c)V_d$$

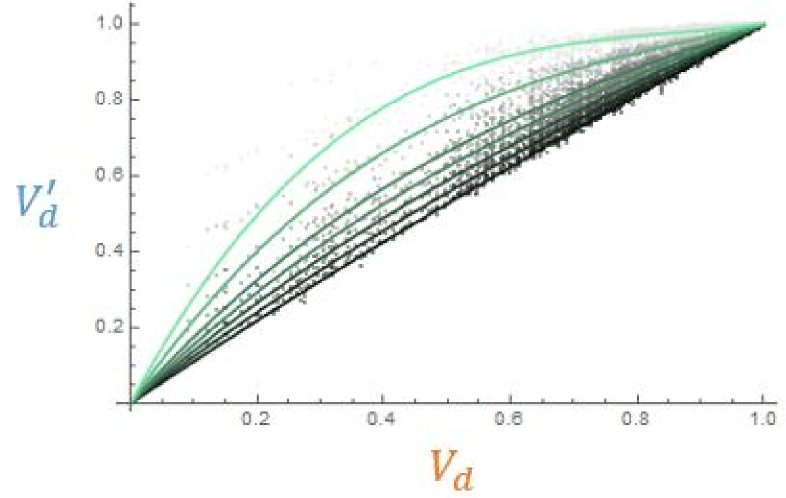
$$a(\rho) = 2.0404\rho - 0.3324$$
$$b(\rho) = -4.7951\rho + 0.6417$$
$$c(\rho) = 2.7552\rho + 0.6903$$

```
float3 GTAOMultiBounce( float visibility, float3 albedo )
{
        float3 a =  2.0404 * albedo - 0.3324;
        float3 b = -4.7951 * albedo + 0.6417;
        float3 c =  2.7552 * albedo + 0.6903;

        float x = visibility;
        return max( x, ( ( x * a + b ) * x + c ) * x );
}
```

Cubic Polynomial Coefficients

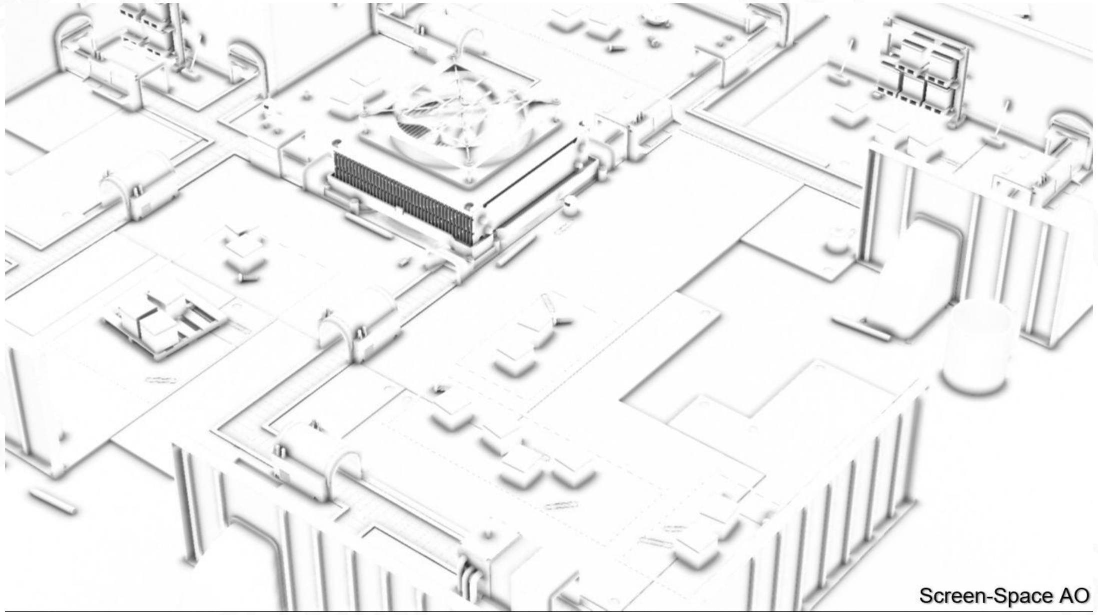
ρ = 0.1	a → 0.0363517	b → -0.162324	c → 1.12599
	a → 0.0999267	b → -0.376556	c → 1.27692
	a → 0.183839	b → -0.632143	c → 1.44889
ρ = 0.4	a → 0.289824	b → -0.933065	c → 1.64413
	a → 0.437788	b → -1.3147	c → 1.87812
	a → 0.805044	b → -2.22354	c → 2.4206
ρ = 0.9	a → 1.35375	b → -3.48326	c → 3.13291

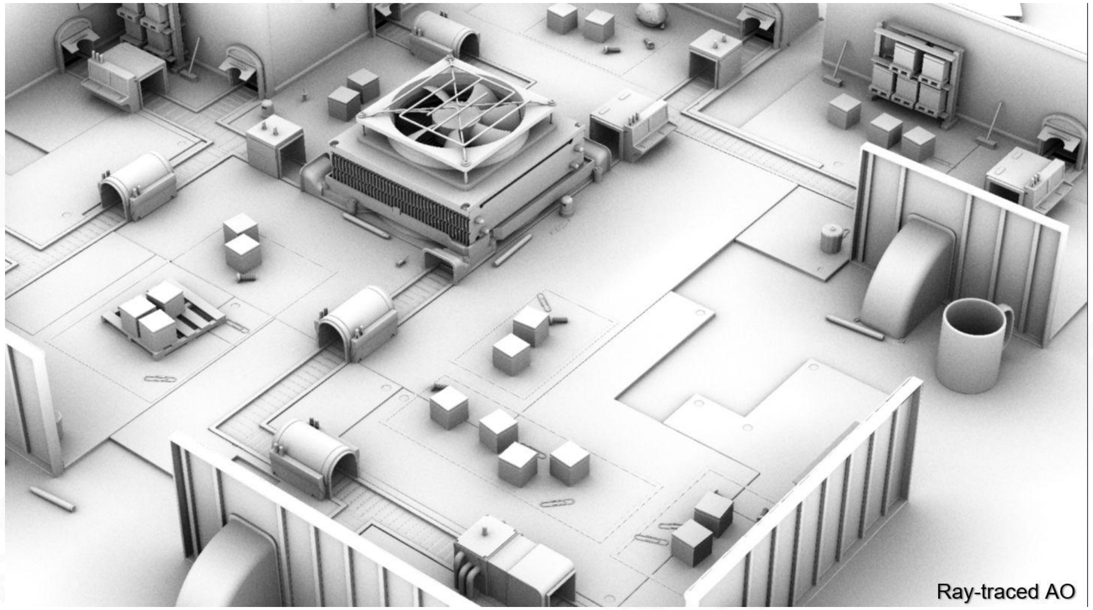
Single Bounce (V_d) Multi Bounce (V_d')



Ray-Tracing Ambient Occlusion

- **Casting rays from each screen pixel using RTT hardware**
  - 1 spp(sample per-pixel) works well for far-field occlusion
  - With 2-4 spp, can recover detailed occlusion in contact region





Fog Everything

Depth Fog

Linear fog:

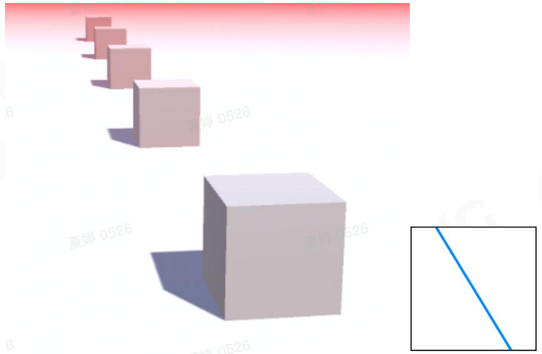
factor = (end-z)/(end-start)

Exp fog:

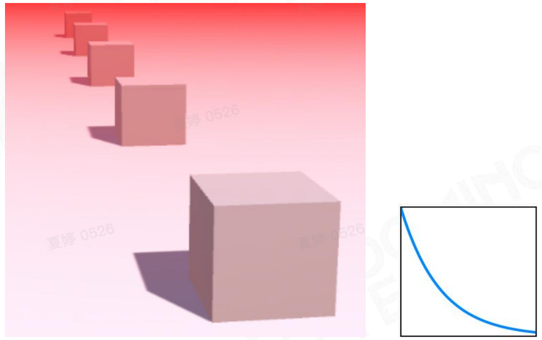
factor = exp(- density*z)

Exp Squared fog:

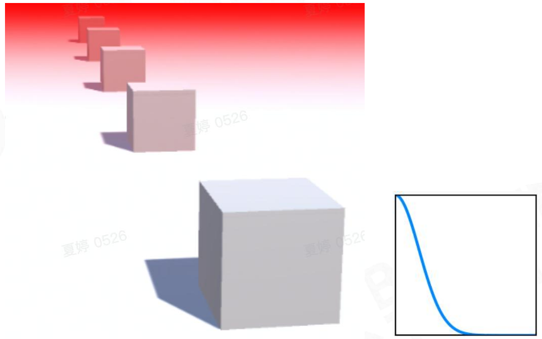
factor = exp(- (density*z)^2)



Linear



Exp



Exp Squared

Height Fog

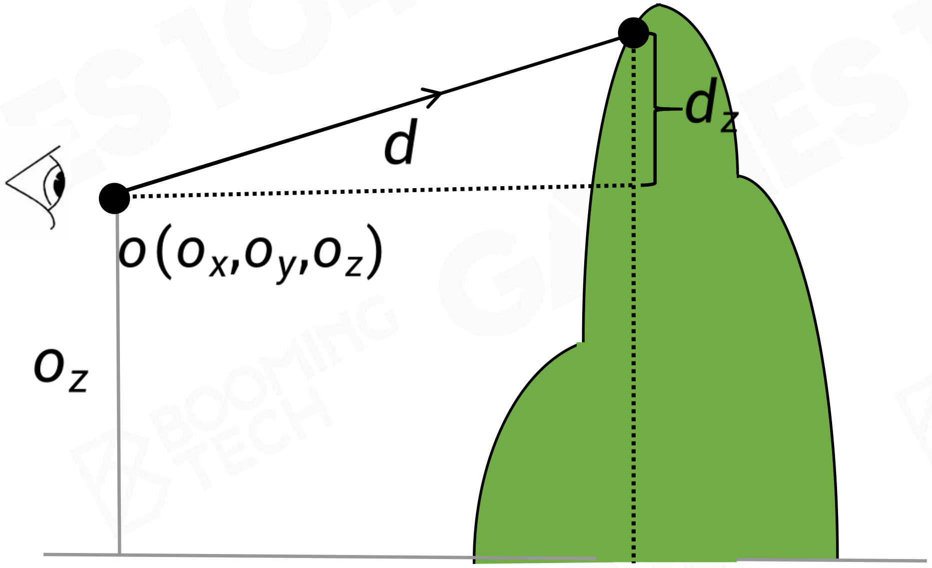
- Height Fog integration along view diretion

$$D(h) = D_{max} \cdot e^{-\sigma \cdot max(h - H_s, 0)}$$

$$FogDensityIntegration$$

$$= D_{max} \cdot d \int_0^1 e^{-\sigma \cdot max((v_z + t * d_z - H_s, 0)} dt$$

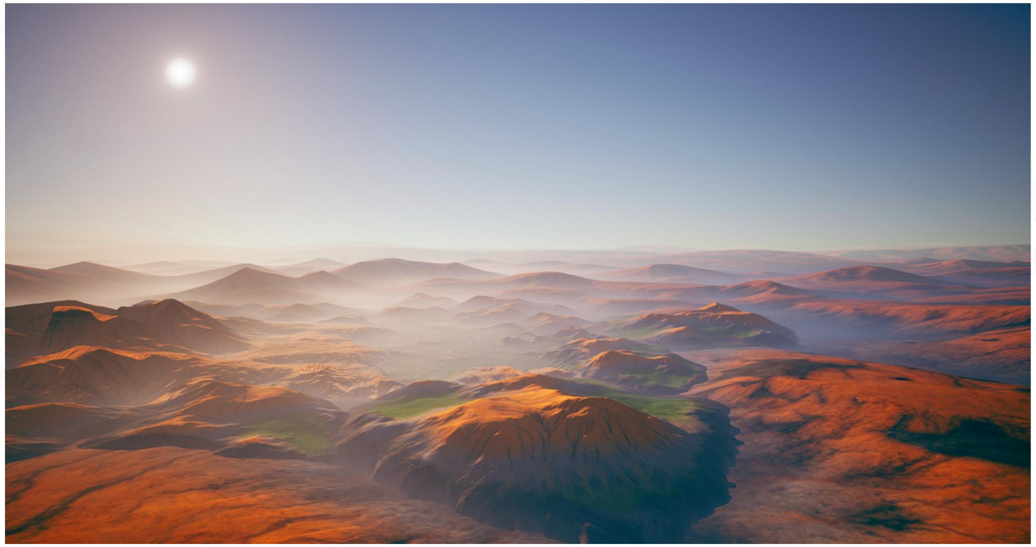
$$= D_{max} \cdot d \cdot e^{-\sigma \cdot max(v_z - Hs, 0)} \frac{1 - e^{-\sigma \cdot d_z}}{\sigma \cdot d_z}$$



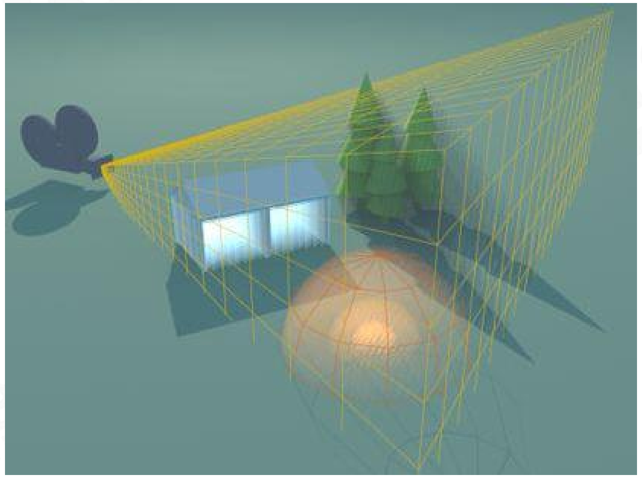
- Fog color after transmission

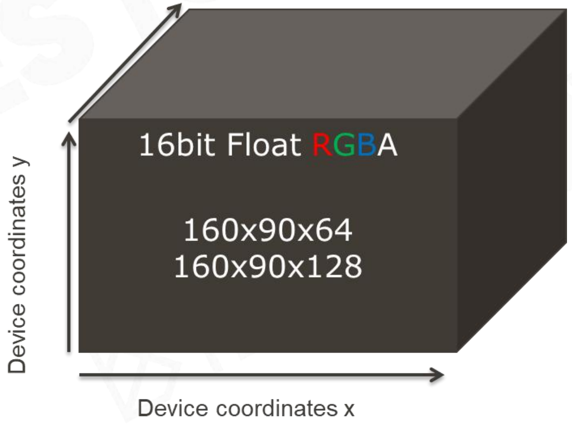
$$FogInscatter = 1 - exp^{-FogDensityIntegration}$$

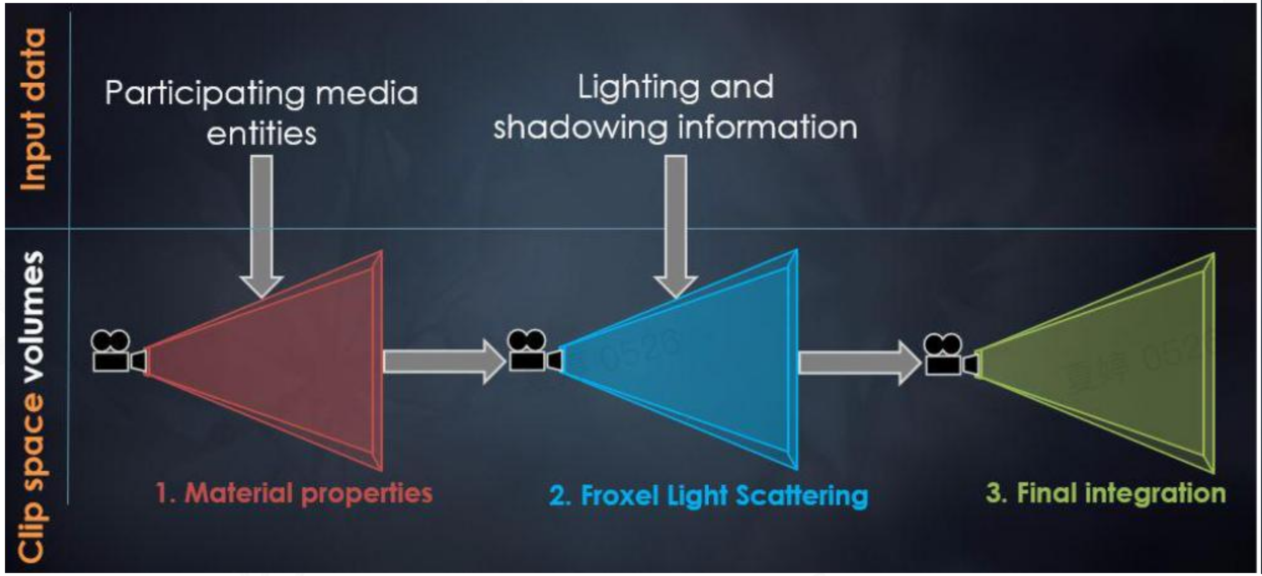
$$FinalColor = FogColor \cdot FogInscatter$$



Voxel-based Volumetric Fog



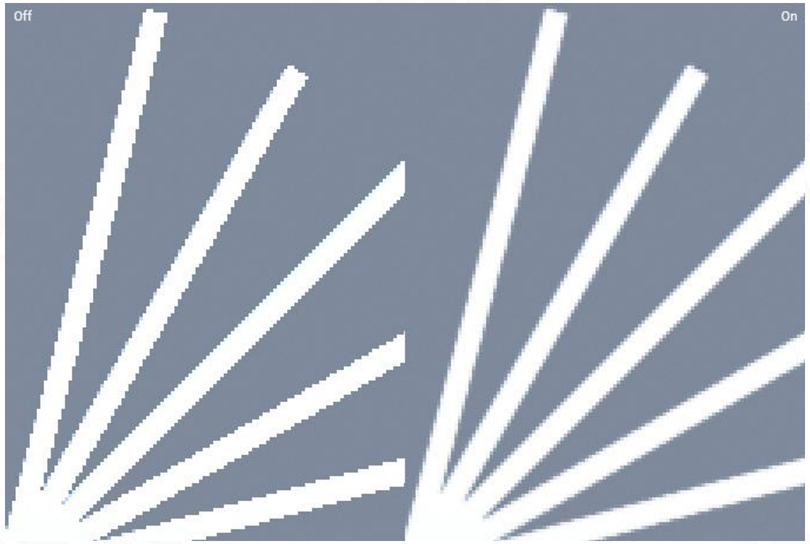




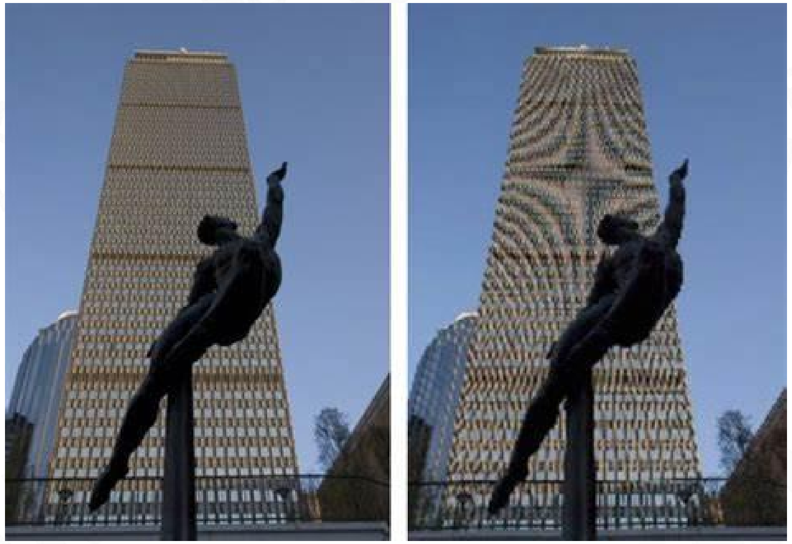

Anti-aliasing

Reason of Aliasing

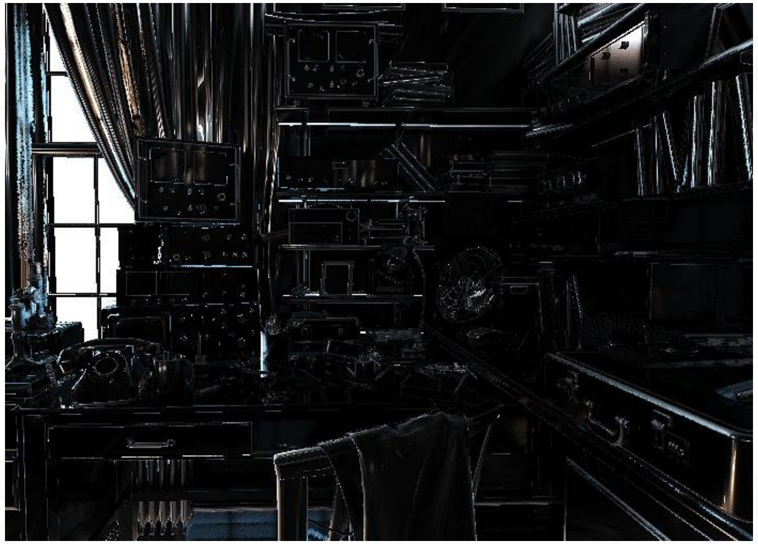
- Aliasing is a series of rendering artifact which is caused by high-frequency signal vs. insufficient sampling of limited rendering resolutions



Edge Sampling



Texture Sampling

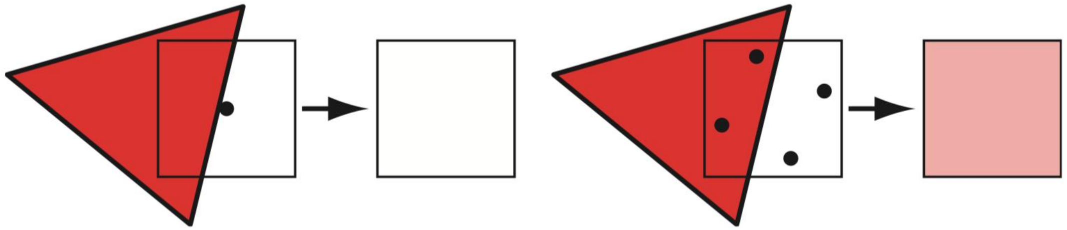


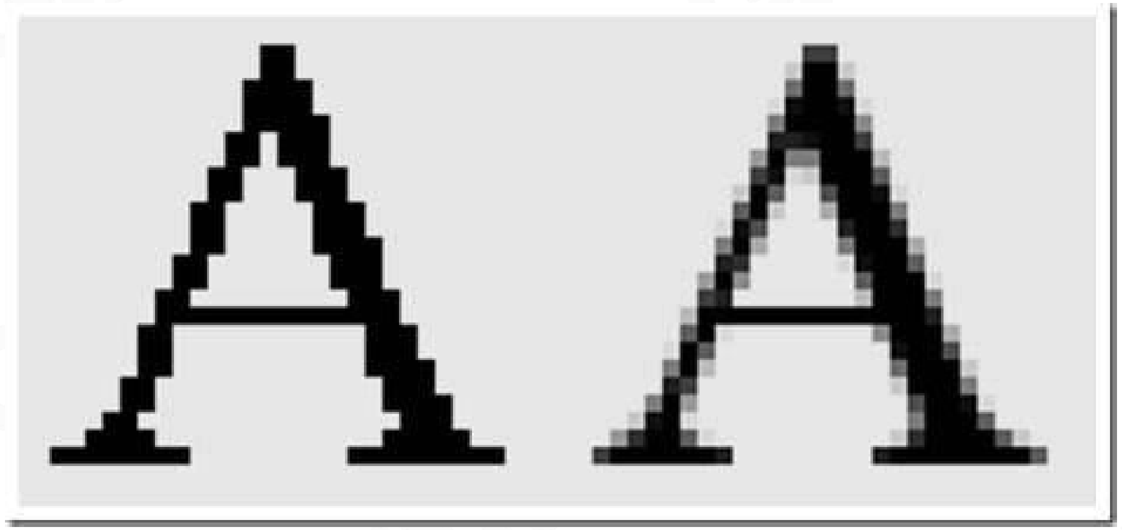
Specular Sampling

Anti-aliasing

- The general strategy of screen-based antialiasing schemes is using a sampling pattern to **get more samples** and then **weight and sum samples** to produce a pixel color

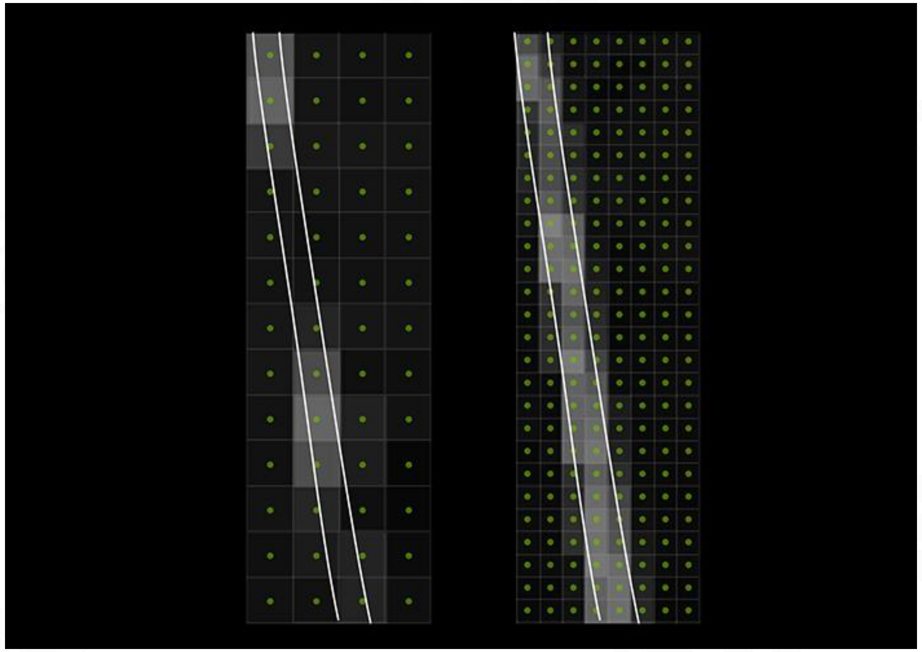
$$\mathbf{p}(x, y) = \sum_{i=1}^{n} w_i \mathbf{c}(i, x, y)$$



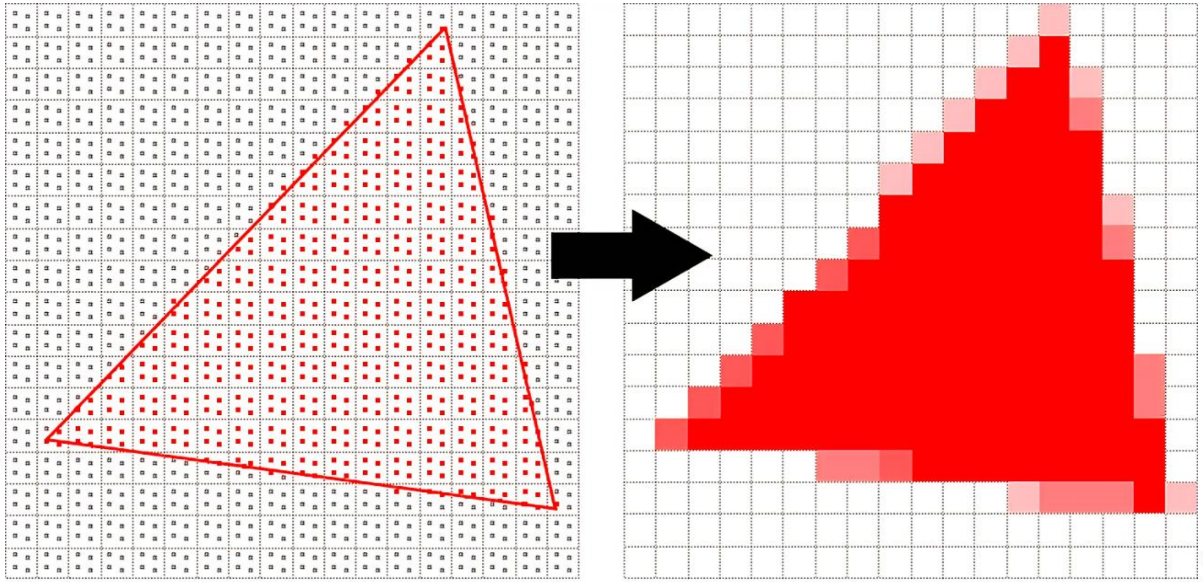


Super-sample AA (SSAA) and Multi-sample AA (MSAA)

- Super sampling is the most straightforward solution to solve **AA**



SSAA, 4x rendering resolution
4x z-buffer and framebuffer
4x rasterization and pixel shading

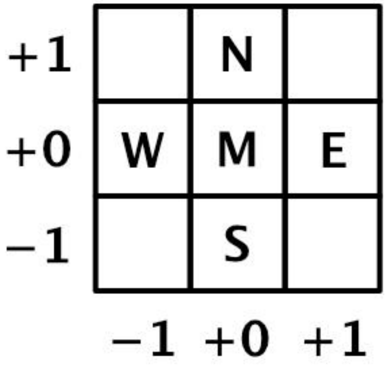


MSAA, only multi-sampling necessary pixels
4x z-buffer and framebuffer
4x rasterization and 1+ x pixel shading

FXAA (Fast Approximate Anti-aliasing)

Anti-aliasing based on 1x rendered image

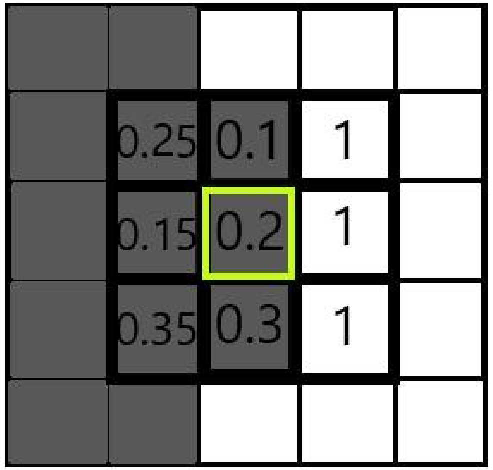
- Find edge pixels by luminance
- Compute offset for every edge pixel
- Re-sample edge pixel by its offset to blend with a neighbor

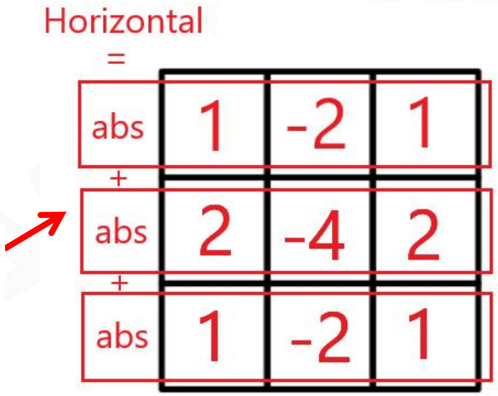


M: Luminance of middle pixel
(L = 0.299 * R + 0.587 * G + 0.114 * B)

```
#define _MinThreshold 0.05

float MaxLuma = max(N, E, W, S, M);
float MinLuma = min(N, E, W, S, M);
float Contrast =  MaxLuma - MinLuma;
if(Contrast >= _MinThreshold)
...
```

Compute Offset Direction

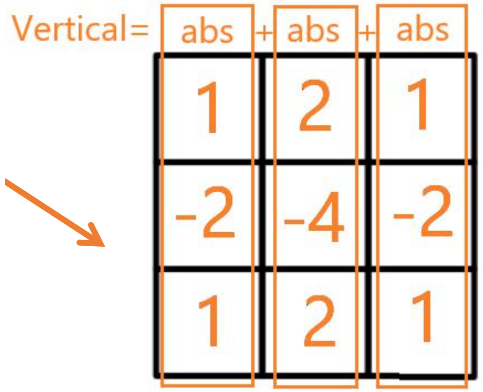




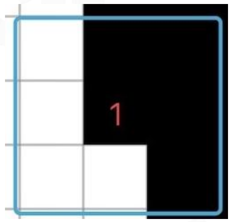
Horizontal = abs(1*0.25+1*1-2*0.1)
+abs(2*0.15+2*1-4*0.2)
+abs(1*0.35+1*1-2*0.3) = **3.3**

3.3 > 0.3

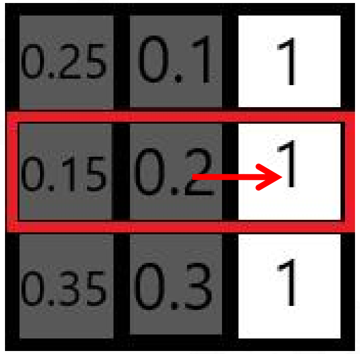
Direction is horizontal



Vertical = abs(1*0.25+1*0.35-2*0.15)
+abs(2*0.1+2*0.3-4*0.2)
+abs(1*1+1*1-2*1) = **0.3**

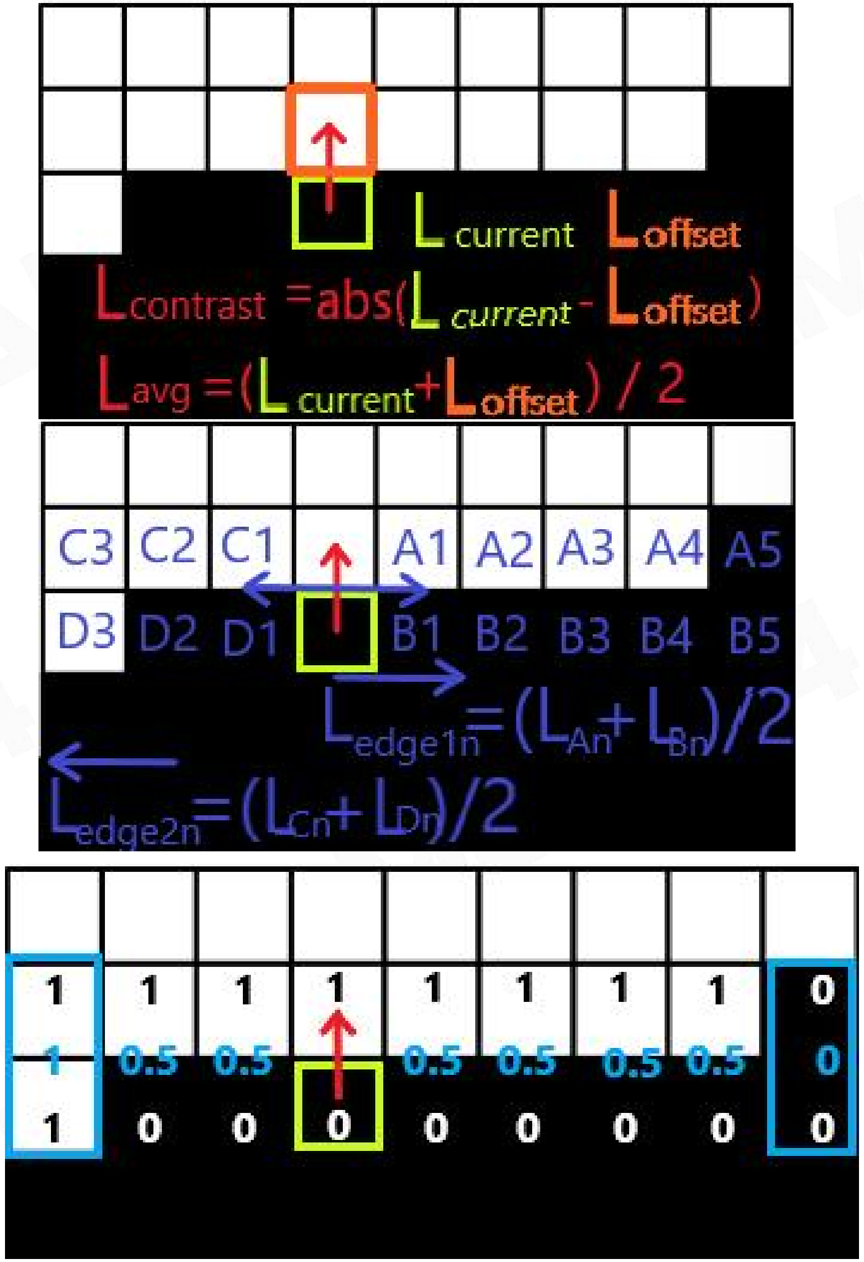


abs(1 - 0.2) >
abs(0.15 - 0.2)



Edge Searching Algorithm

- Find aliasing edge that the pixel is in
  - Record constrast luminance and average luminance of current pixel and offset pixel
    L_{avg} $L_{contrast}$
  - Search along the **2** **perpendicular** direction and calculate the average luminance
    L_{edge1n} L_{edge2n}
  - Until abs(L_{edge1n} - $L_{current}$) > 0.25 $L_{contrast}$
    abs(L_{edge2n} - $L_{current}$) > 0.25 $L_{contrast}$

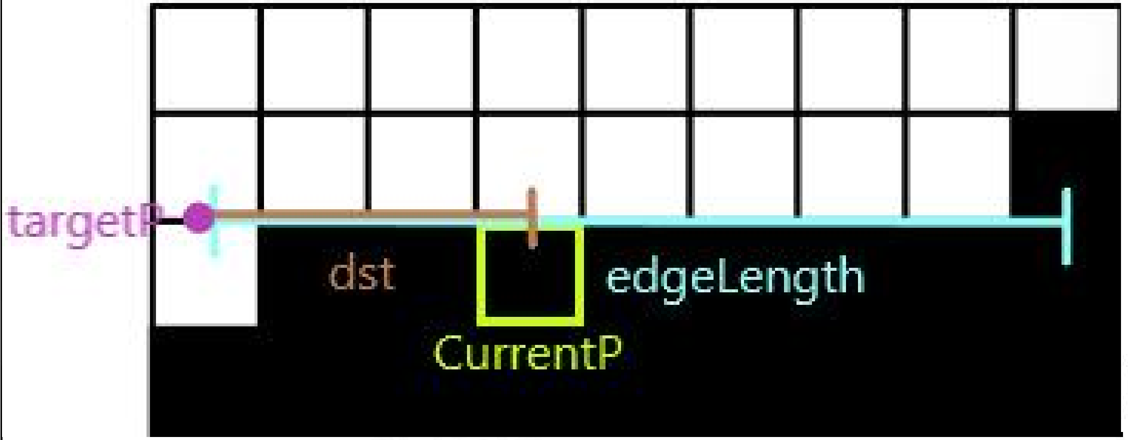


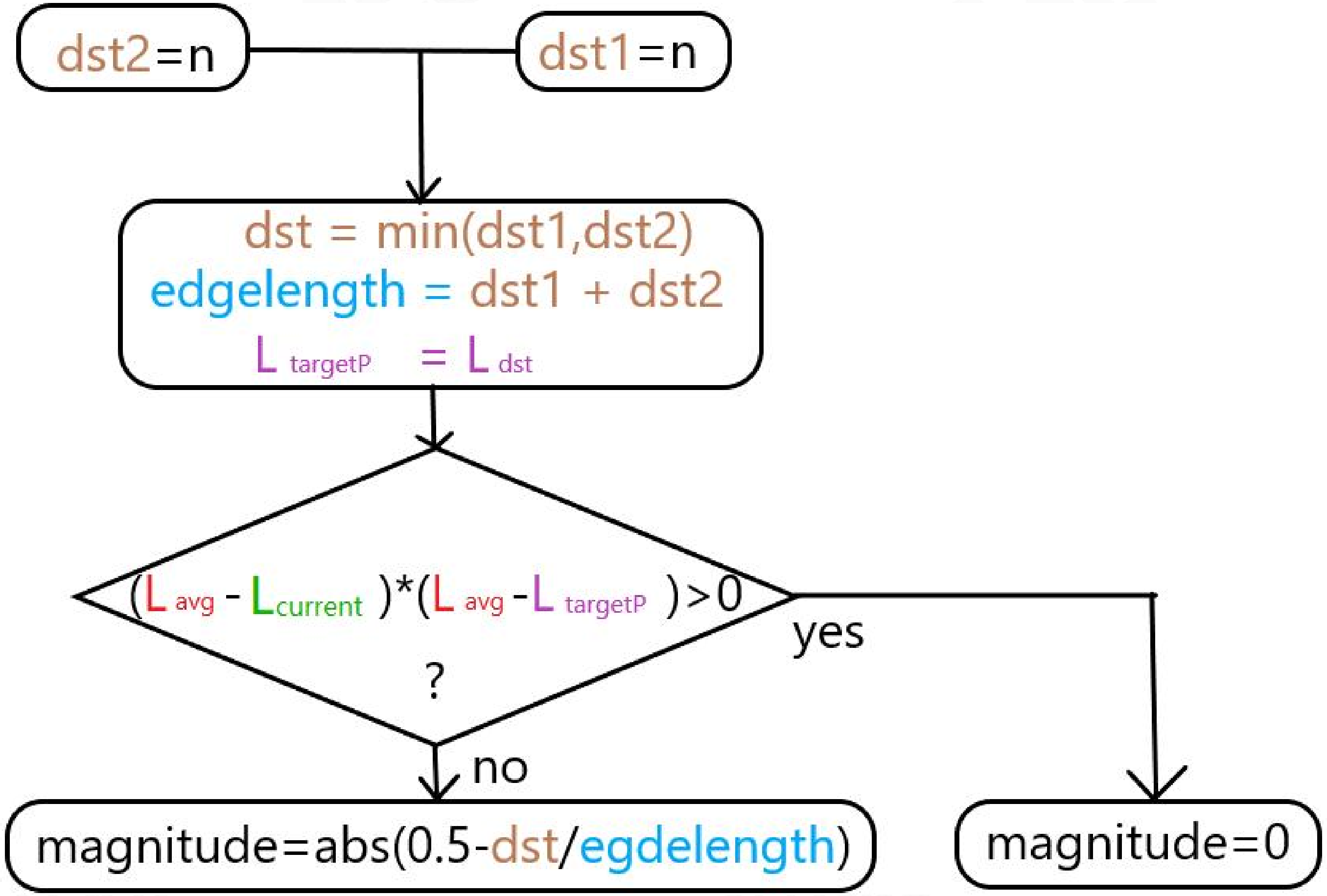
Calculate Blend Coefficient

- Compute blender coefficient

targetP is the nearer edge end of CurrentP

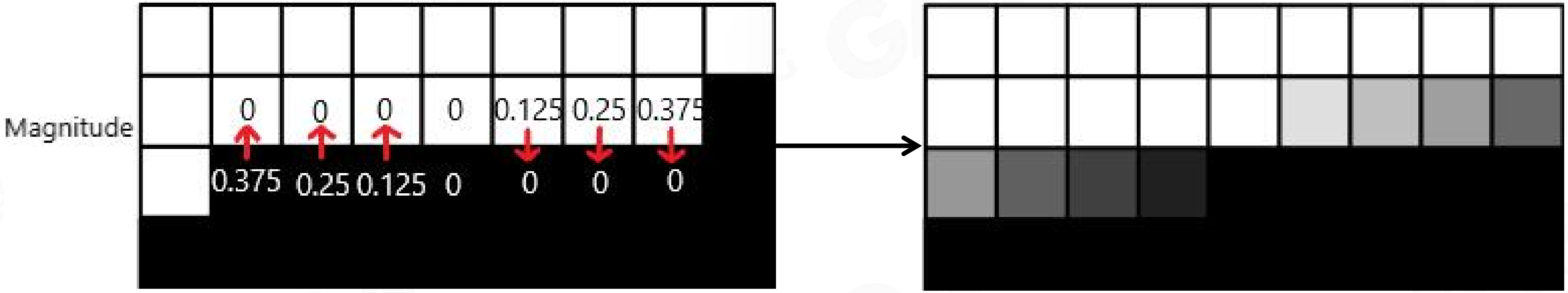
if((L_{avg} - $L_{current}$) * (L_{avg} - $L_{targetP}$) > 0)
    magnitude = 0;
else
    magnitude = abs(0.5 - dst / edgeLength);





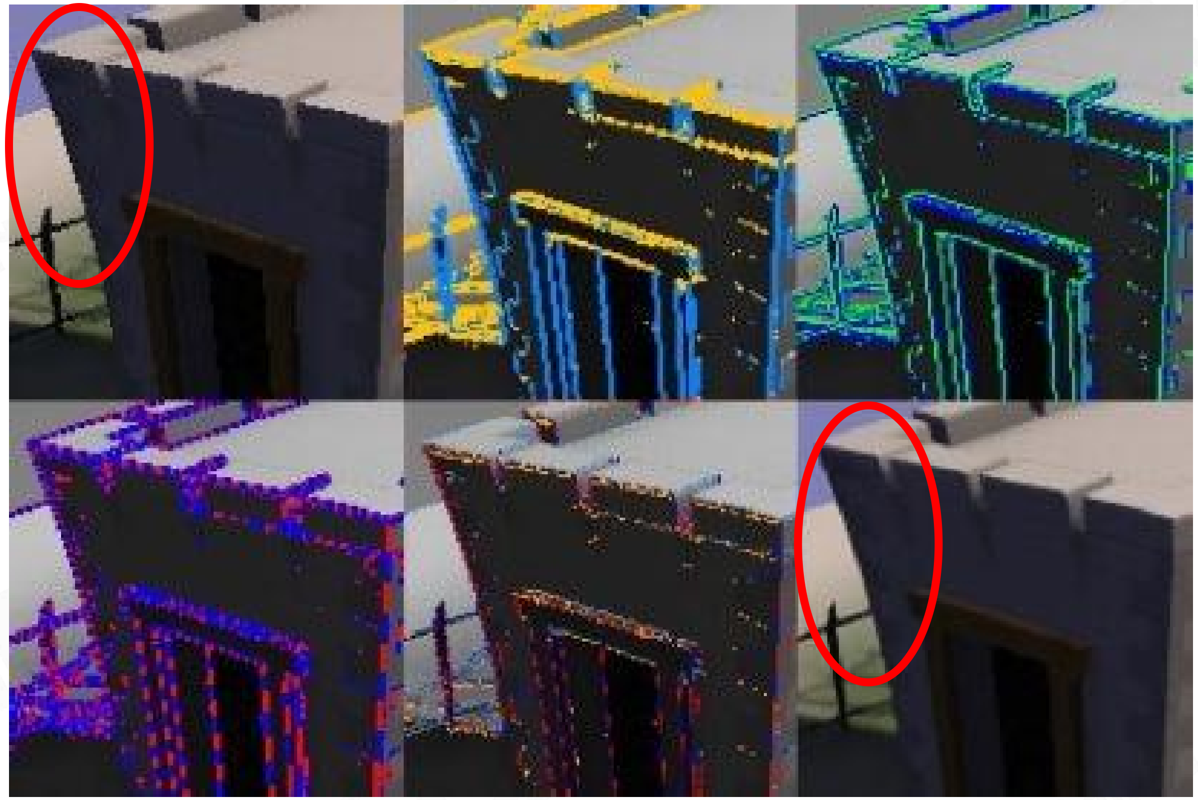
Blend Nearby Pixels

- Compute blender coefficient



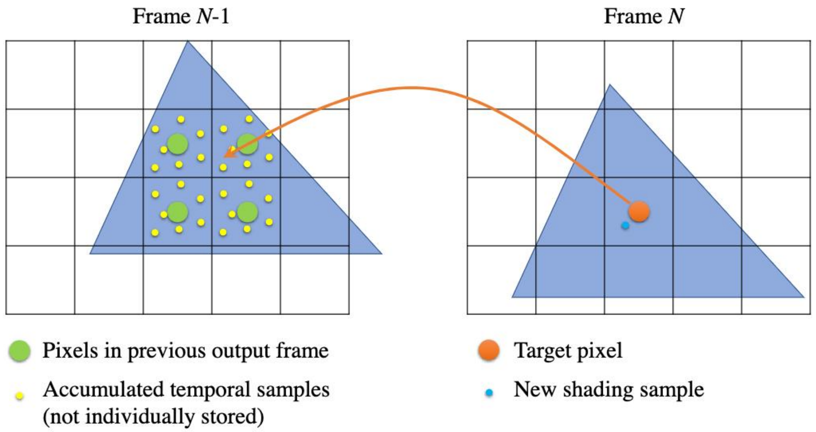
PixelNewColor = Texture(CurrentP_UV + offset_direction * offset_magnitude)

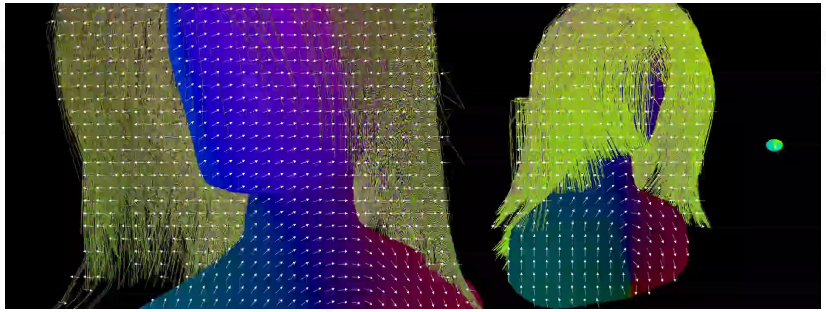
FXAA Result

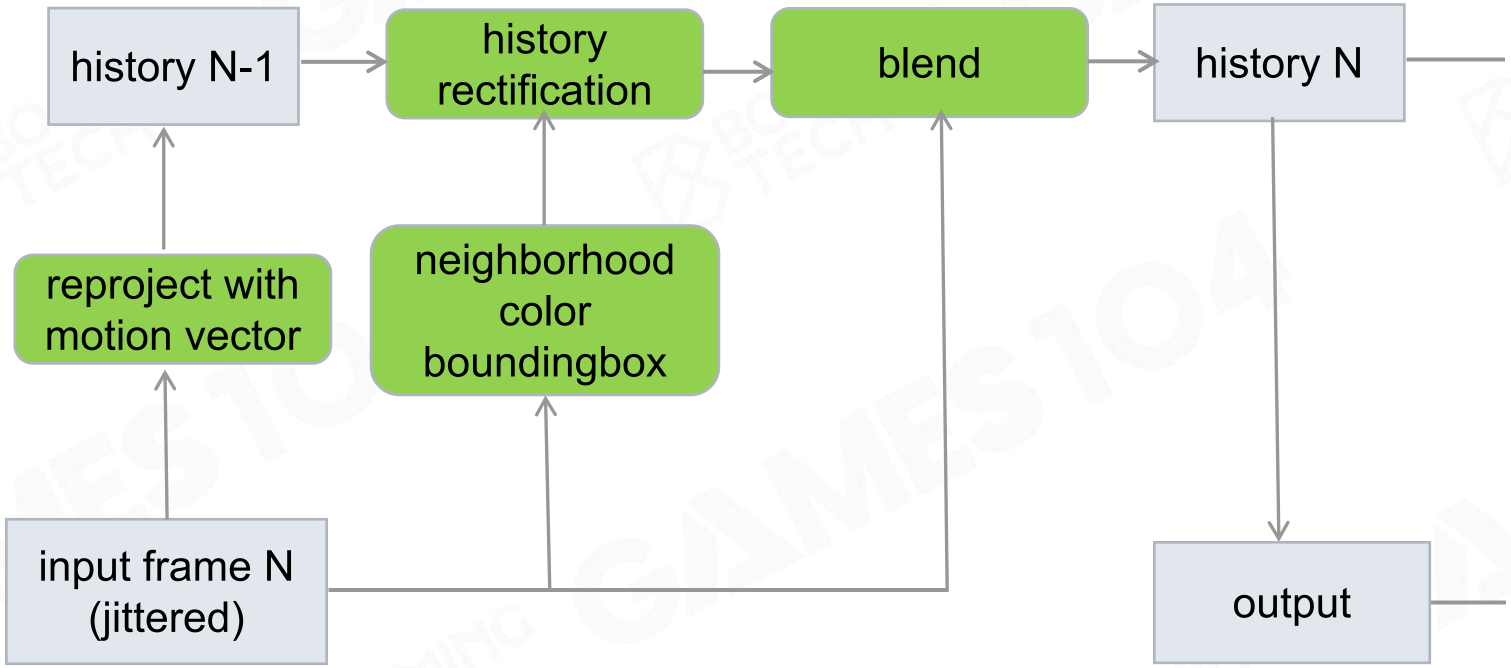


TAA (Temporal Anti-aliasing)

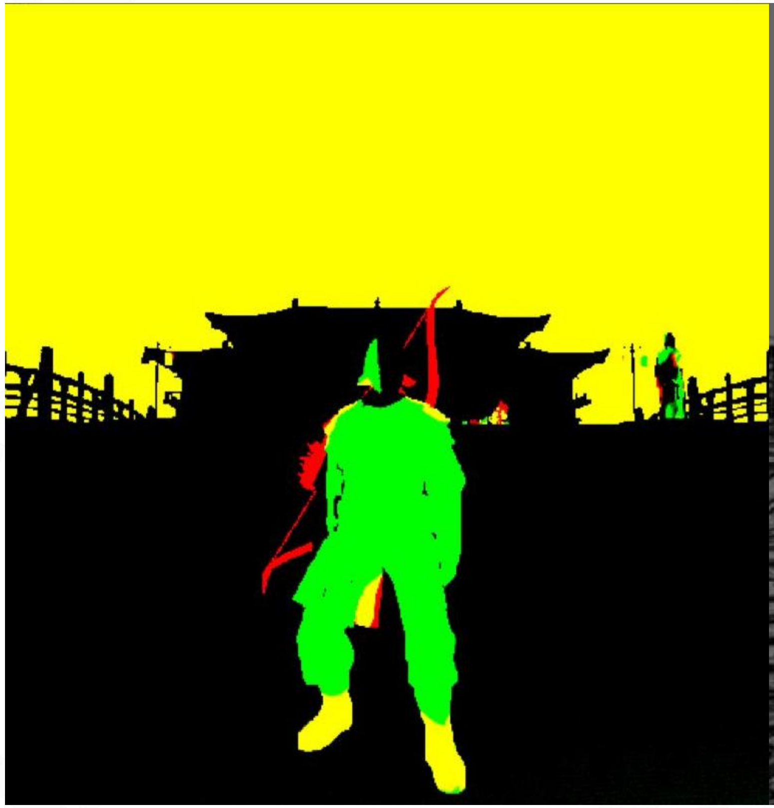
Utilize spatial-**temporal** filtering methods to improve AA stability **in motion**







TAA (Temporal Anti-aliasing)



Motion Vector

Blend Ratio

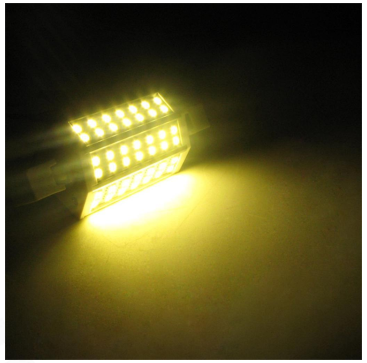
Blend Result


Le Petit Coin
MENU
TAA On/Off

But, the real magic in Post-process...

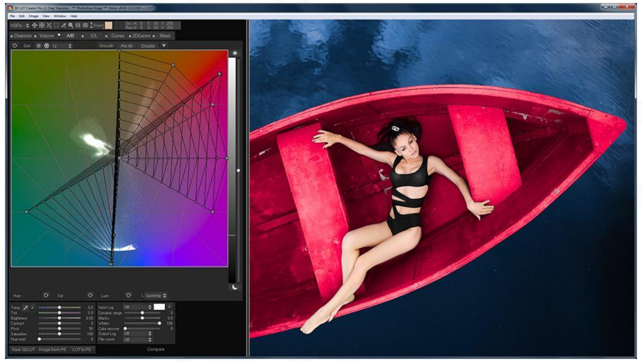
Post-process

Post-process in 3D Graphics refers to any algorithm that will be applied to the final image. It can be done for stylistic reasons (color correction, contrast, etc.) or for realistic reasons (tone mapping, depth of field, etc.)



Bloom

Tone Mapping

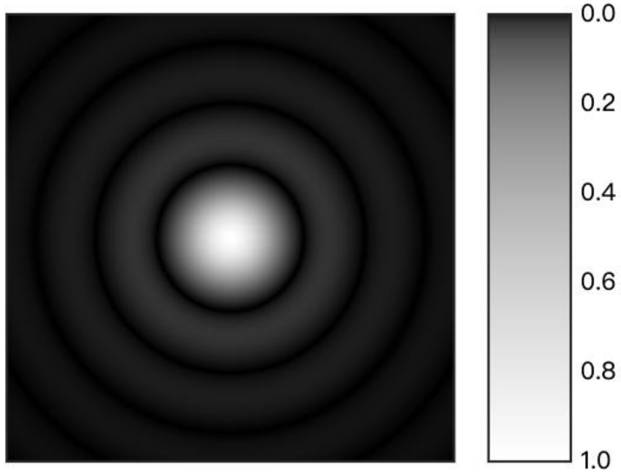


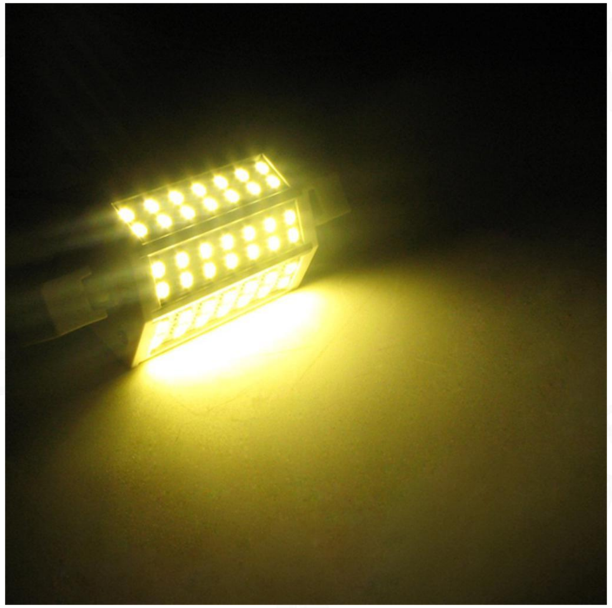
Color Grading

Bloom Effect

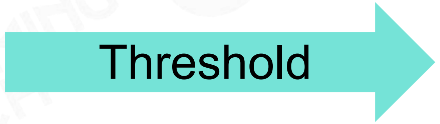
What is Bloom

- The physical basis of bloom is that, in the real world, lenses can never focus perfectly
- Even a perfect lens will convolve the incoming image with an [Airy disk]





Detect Bright Area by Threshold



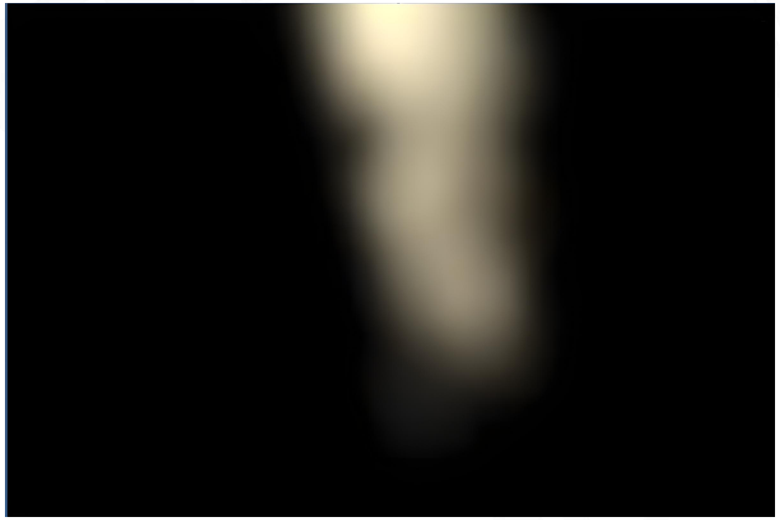
Find Luminance (Y) apply the standard coefficients for sRGB:

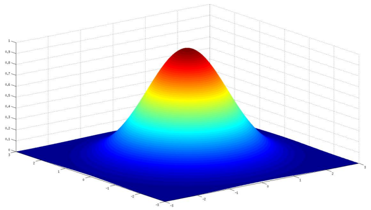
$$Y = R_{lin} * 0.2126 + G_{lin} * 0.7152 + B_{lin} * 0.0722$$

```
float threshold;

float4 computeHighlightArea()
{
    [...] // first do normal lighting calculations and output results
    float4 scene_color = float4(lighting, 1.0f);
    // check whether fragment output is higher than threshold, if so output as highlight color
    float luminance = dot(scene_color.rgb, vec3(0.2126f, 0.7152f, 0.0722f));

    float4 highlight_color = float4(0.0f, 0.0f, 0.0f, 1.0f);
    if(luminance > threshold)
        highlight_color = float4(scene_color.rgb, 1.0f);
    return highlight_color;
}
```

Gaussian Blur

Blur





Gaussian distribution

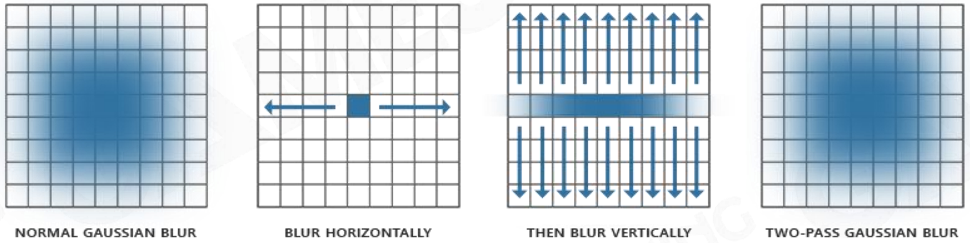
$$\frac{1}{256} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} = \frac{1}{256} \cdot \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

A classic gaussian kernel

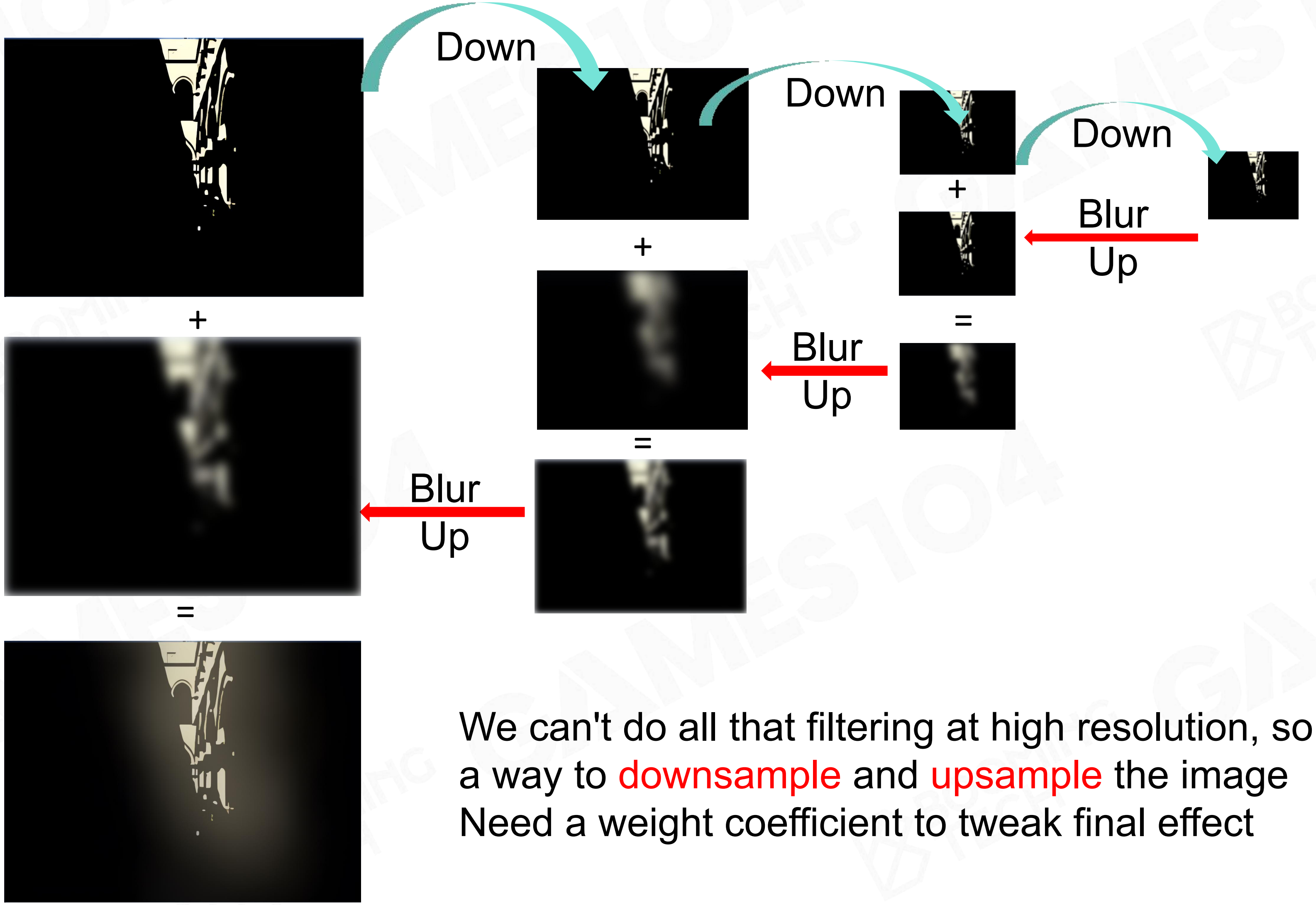
5*5(25) samples per pixel

Linearly separable

5+5(10) samples per pixel



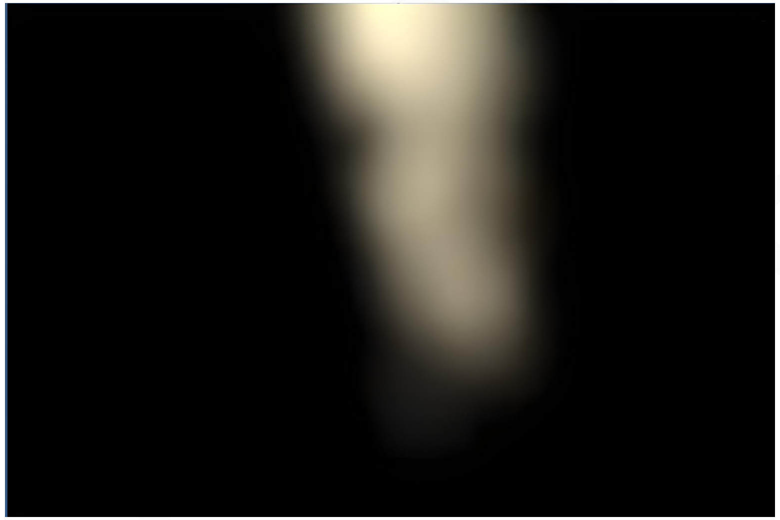
Pyramid Guassian Blur



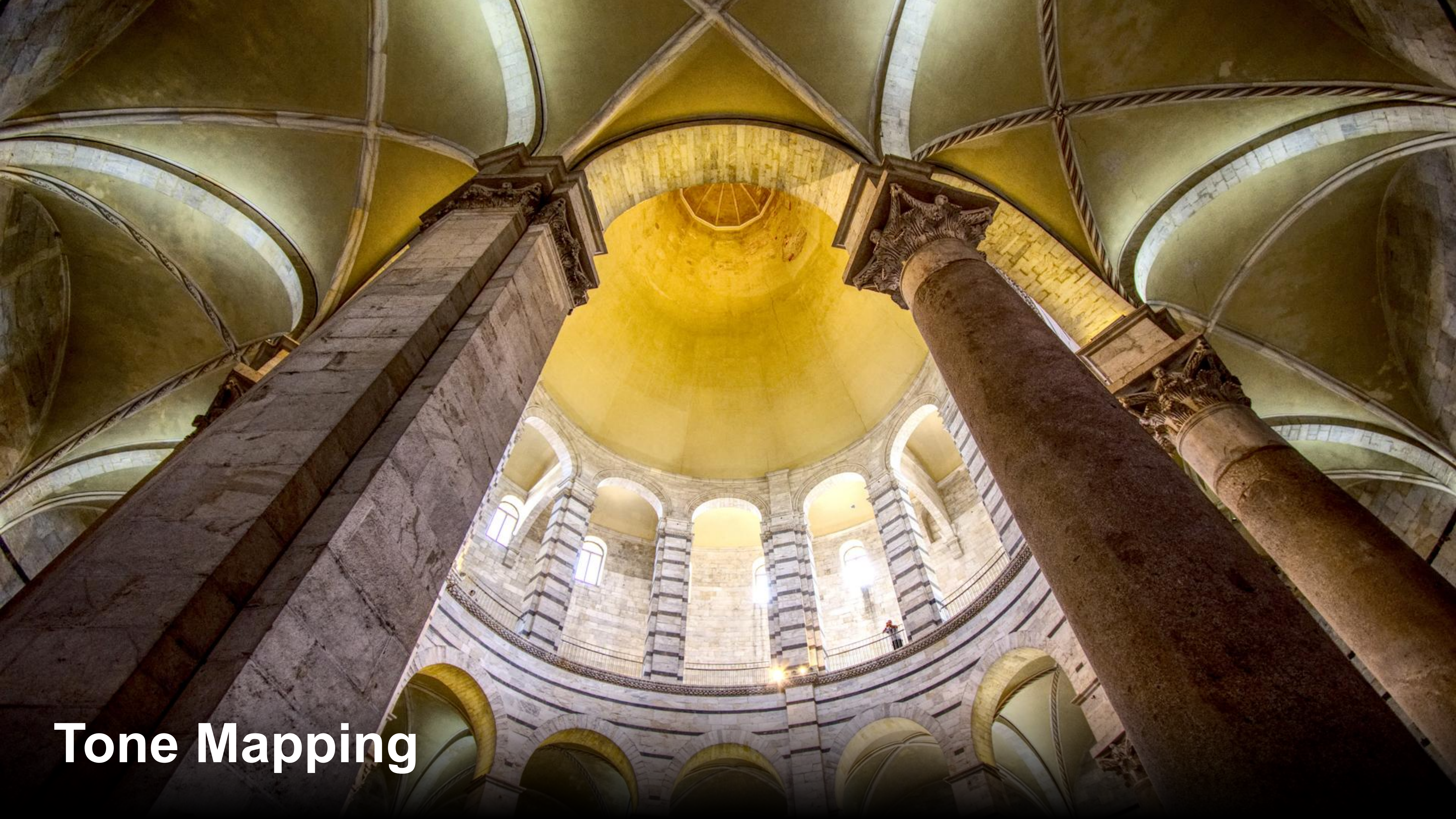
We can't do all that filtering at high resolution, so we need a way to downsample and upsample the image
Need a weight coefficient to tweak final effect

Bloom Composite

+

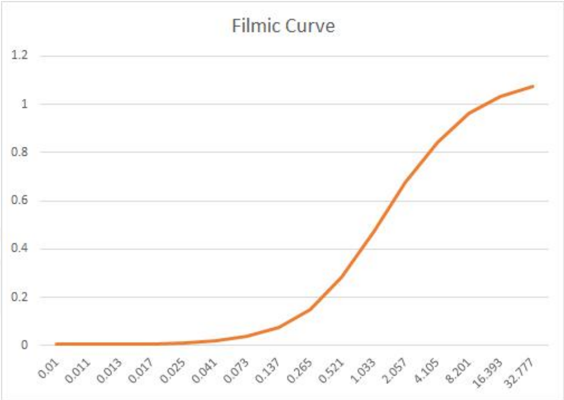



Bloom
On
Bloom
Off


Tone Mapping

Tone Mapping

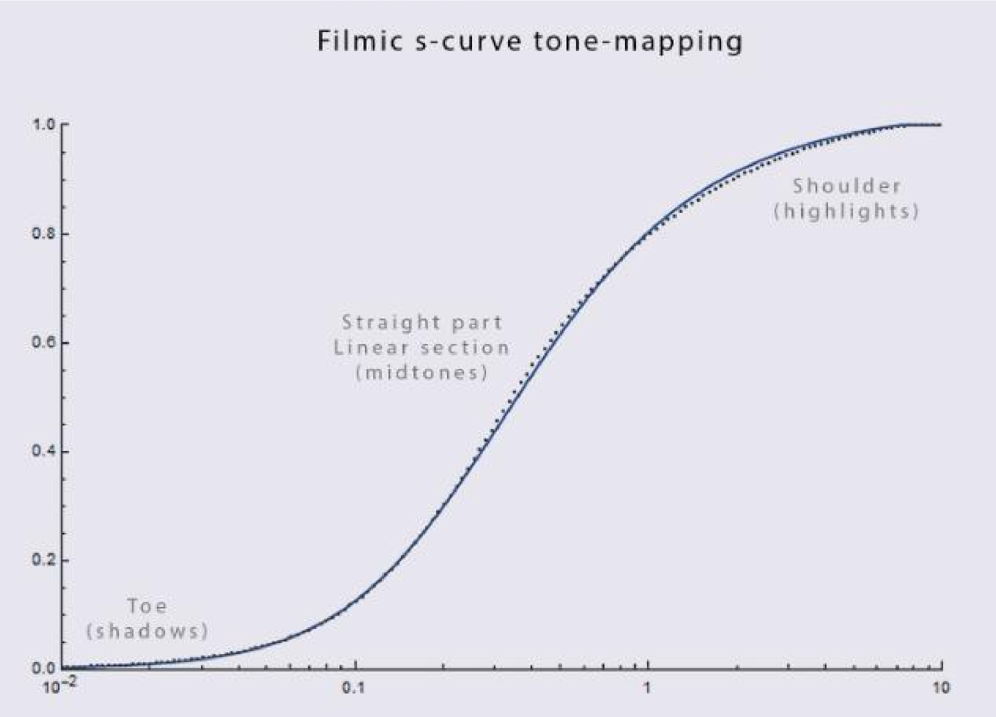
- No way to directly display HDR image in a SDR device
- The purpose of the **Tone Mapping** function is to map the wide range of high dynamic range (HDR) colors into standard dynamic range (SDR) that a display can output



Tone Mapping Curve

Get a filmic look without making renders dirty
Give images proper contrast and nicely roll off any pixels over 1

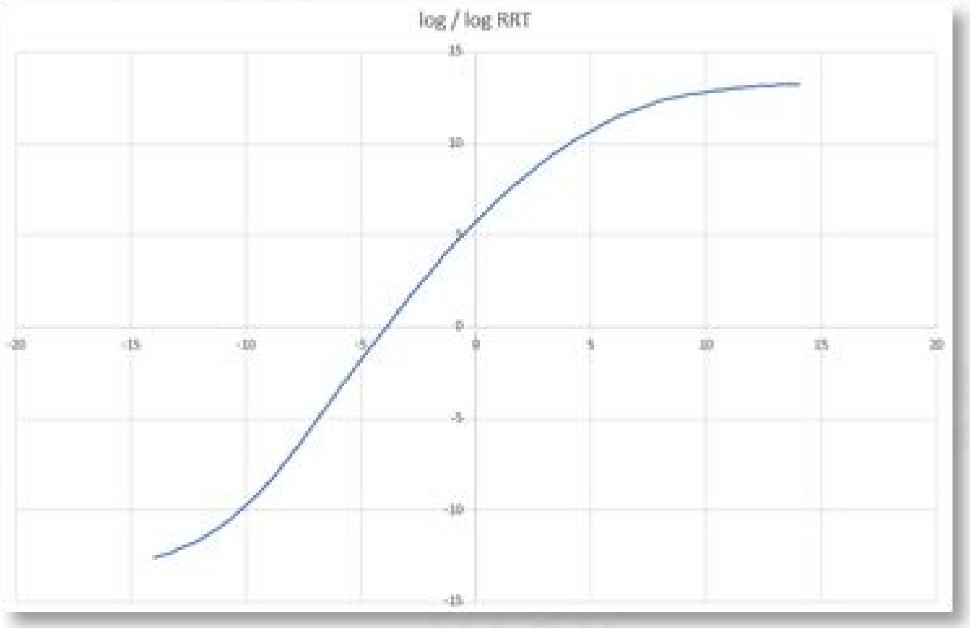
```
float3 F(float3 x)
{
        const float A = 0.22f;
        const float B = 0.30f;
        const float C = 0.10f;
        const float D = 0.20f;
        const float E = 0.01f;
        const float F = 0.30f;

        return ((x * (A * x + C * B) + D * E) / (x * (A * x + B) + D * F)) - E / F;
}

float3 Uncharted2ToneMapping(float3 color, float adapted_lum)
{
        const float WHITE = 11.2f;
        return F(1.6f * adapted_lum * color) / F(WHITE);
}
```



ACES

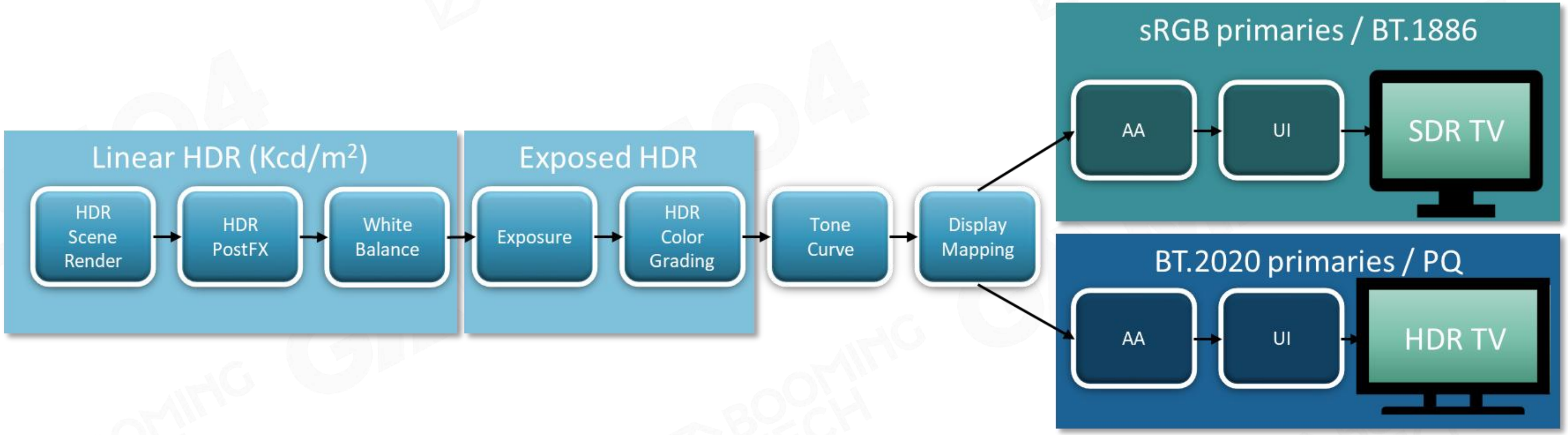
- **A**cademy **C**olor **E**ncoding **S**ystem
  - Primarily for Film & Animation
  - Interesting paradigms and transformations

- The useful bits
  - Applying Color Grading in HDR is good
  - The idea of a fixed pipeline up to the final OTD transforms stage is good
    - Separates artistic intent from the mechanics of supporting different devices

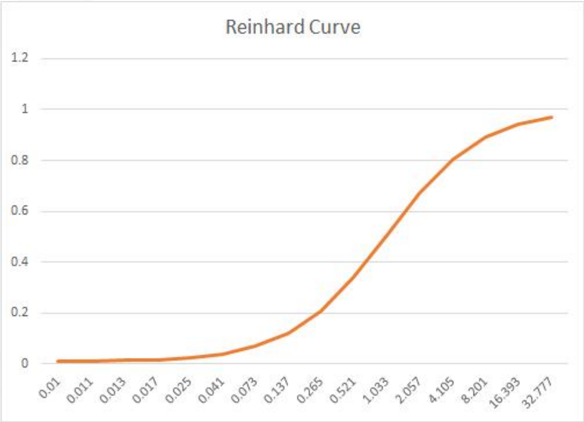


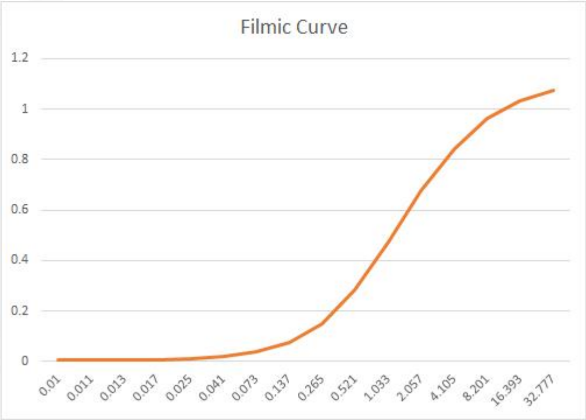
HDR and SDR Pipeline

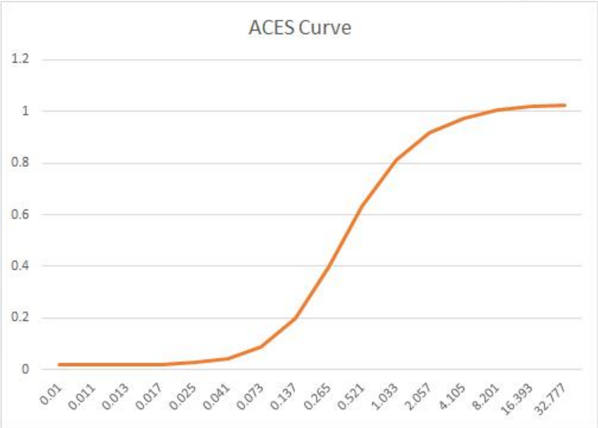
- Visual consistency between HDR / SDR
- Similar SDR results to previous SDR color pipeline
- High quality
- High performance
- Minimal disruption to art teams
  - Simple transition from current color pipeline
  - Minimal additional overhead for mastering HDR *and* SDR

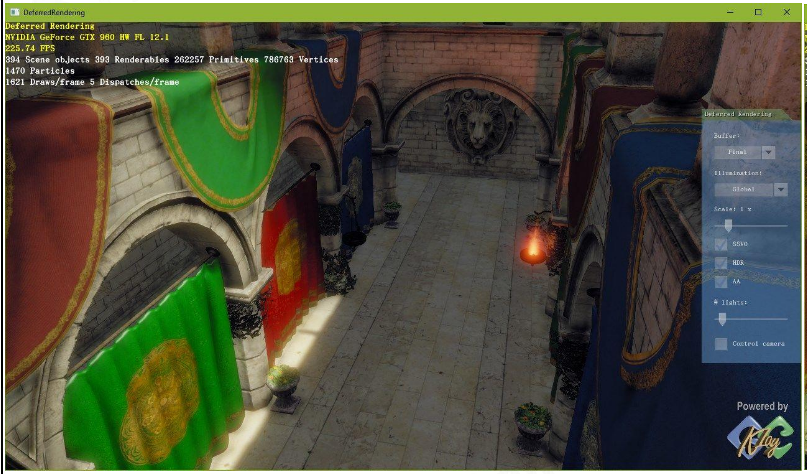


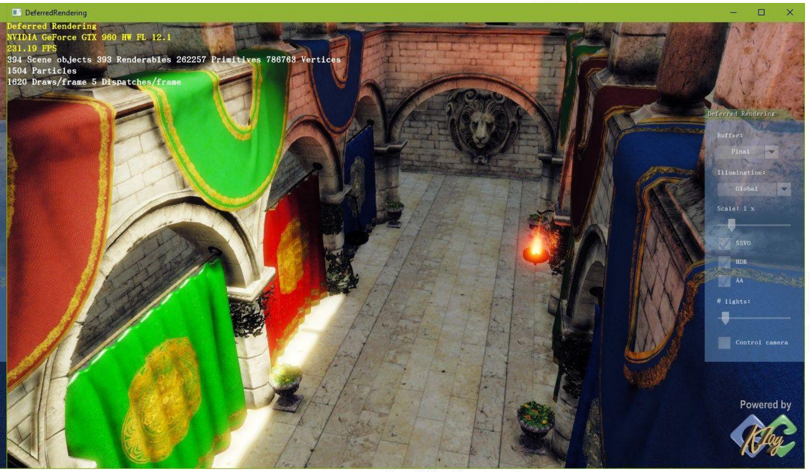
Tone Mapping Curve Comparison









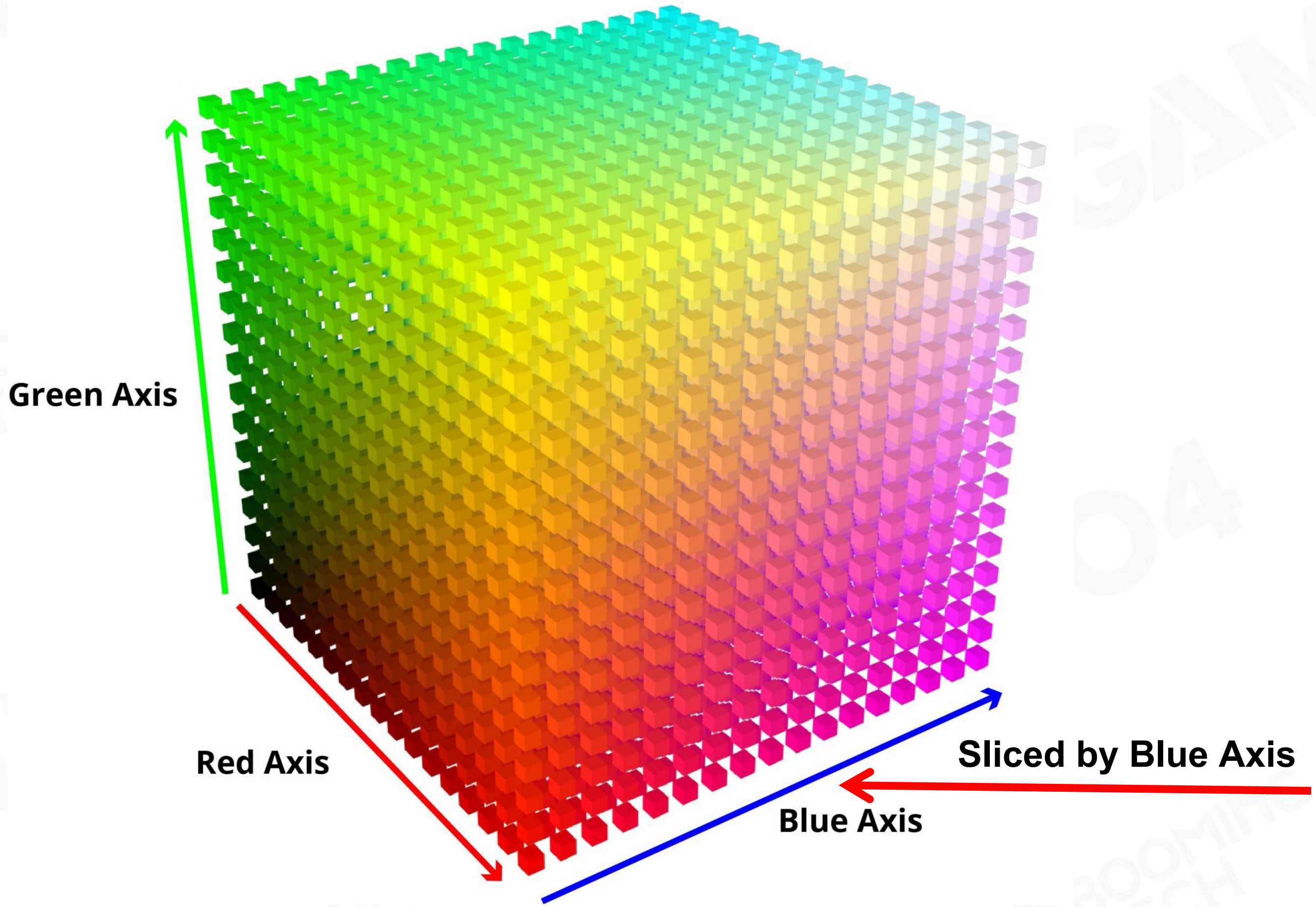



Color Grading

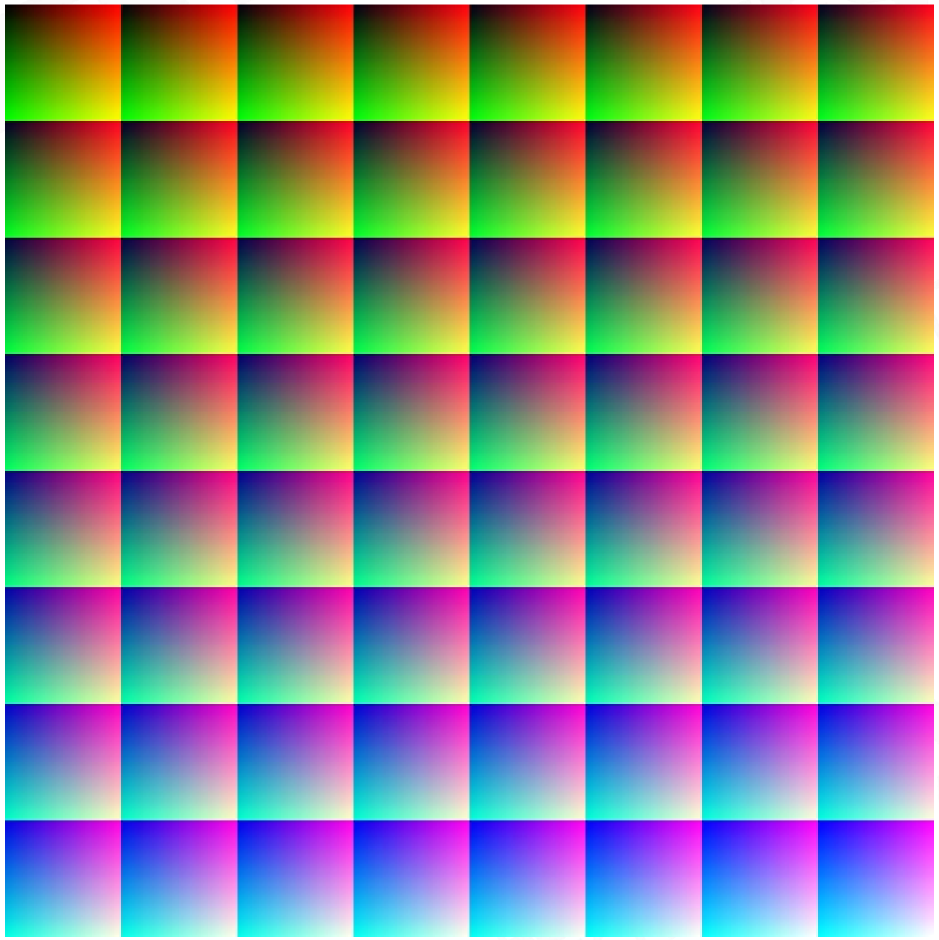
Lookup Table (LUT)

- LUT is used to remap the input color values of source pixels to new output values based on data contained within the LUT
- A LUT can be considered as a kind of color preset that can be applied to image or footage

LUT 3D or 2D

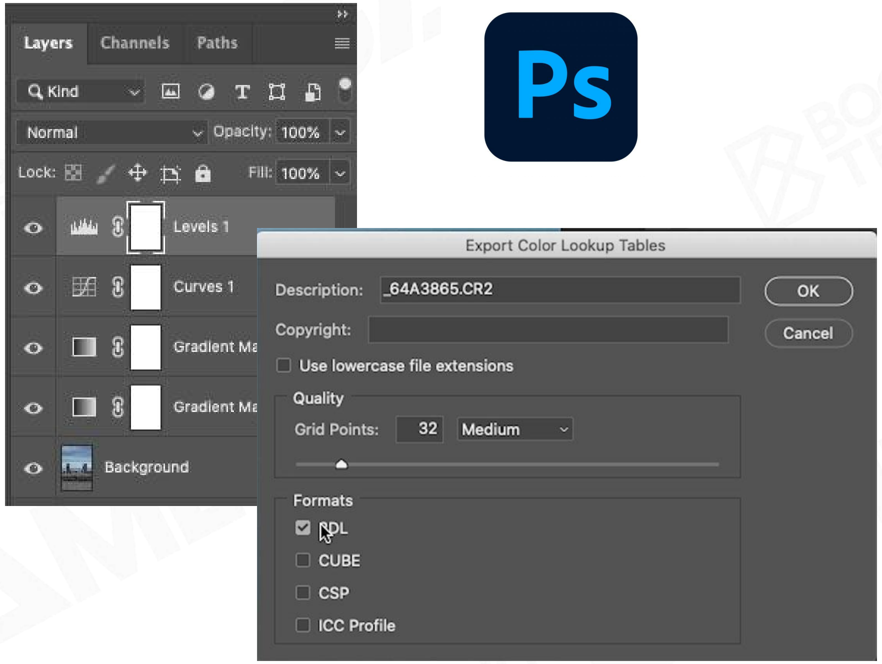


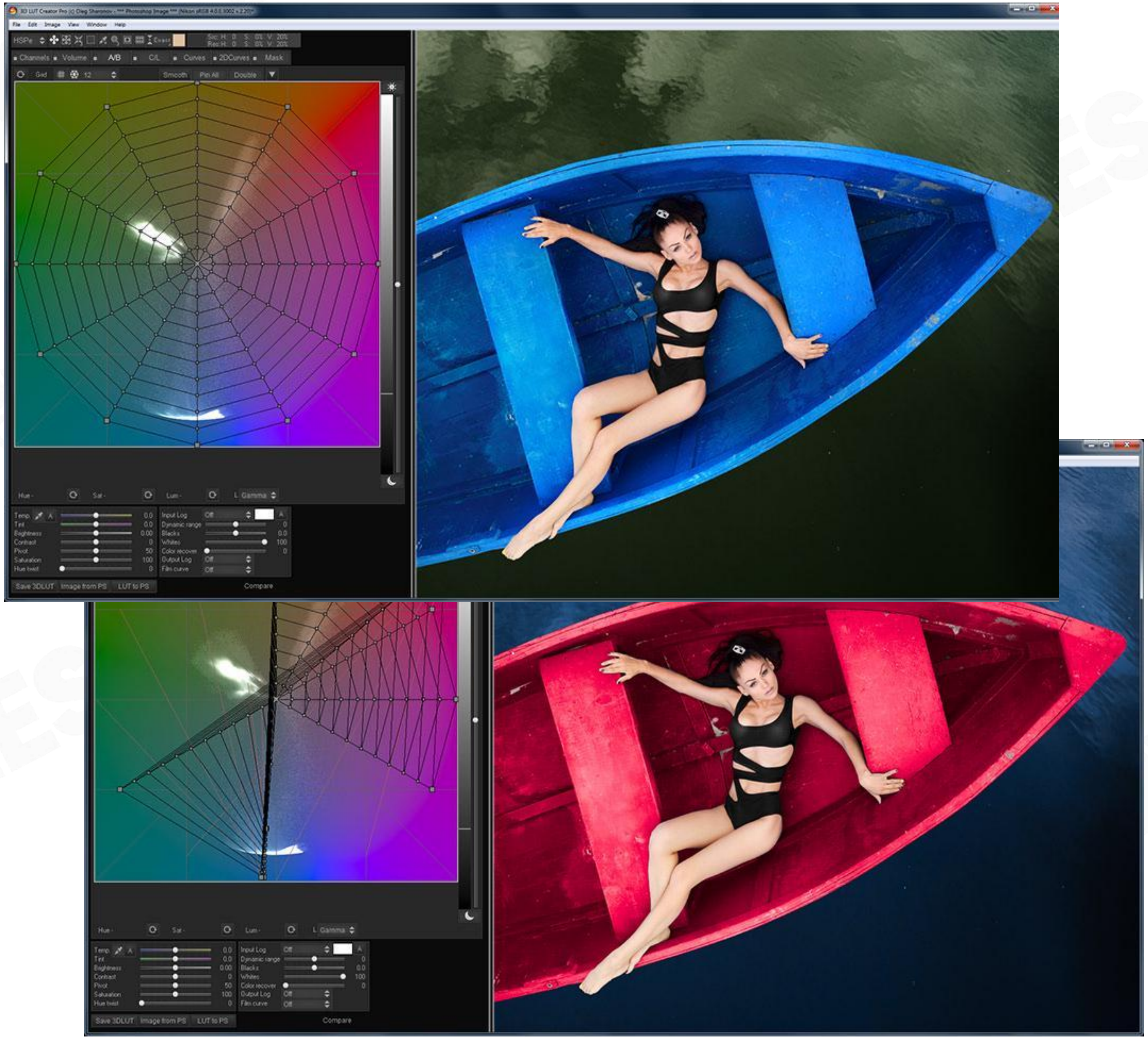
3D



2D Slices

Artist Friendly Tools

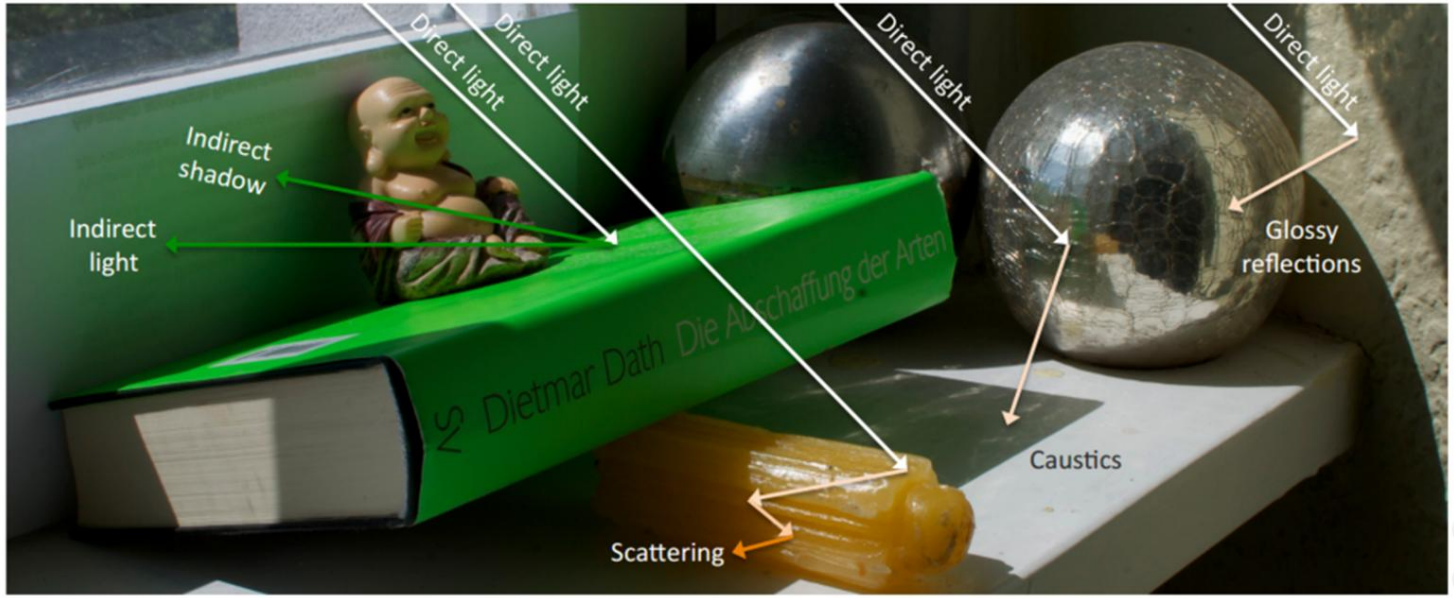


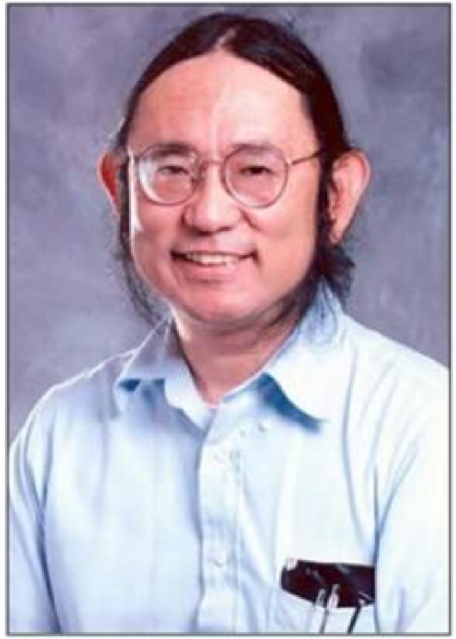


Color grading is the most cost-effective feature of game rendering

Rendering Pipeline

One Equation for Everything

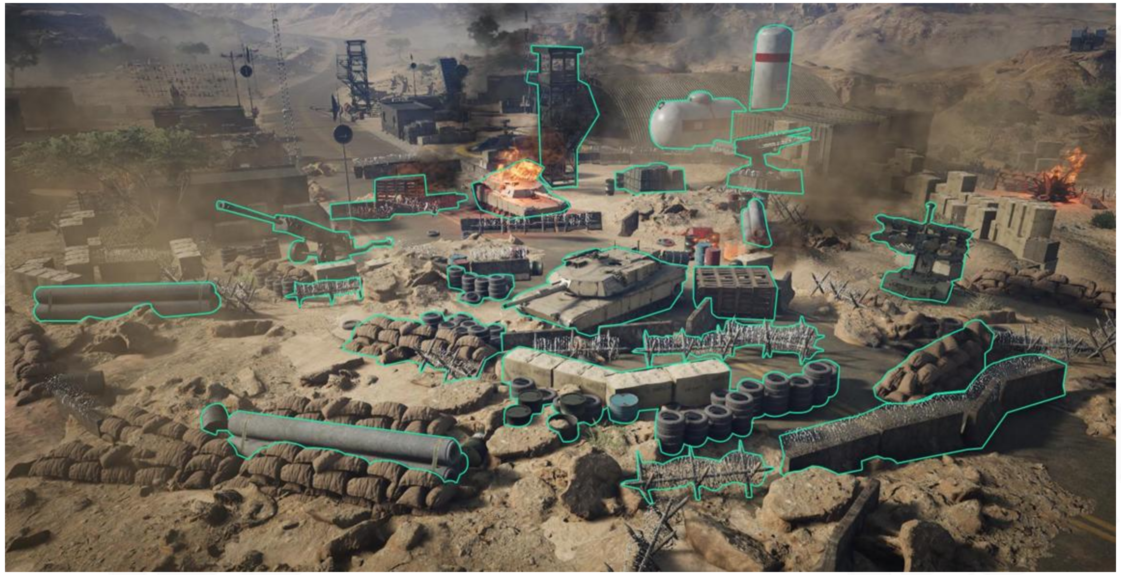




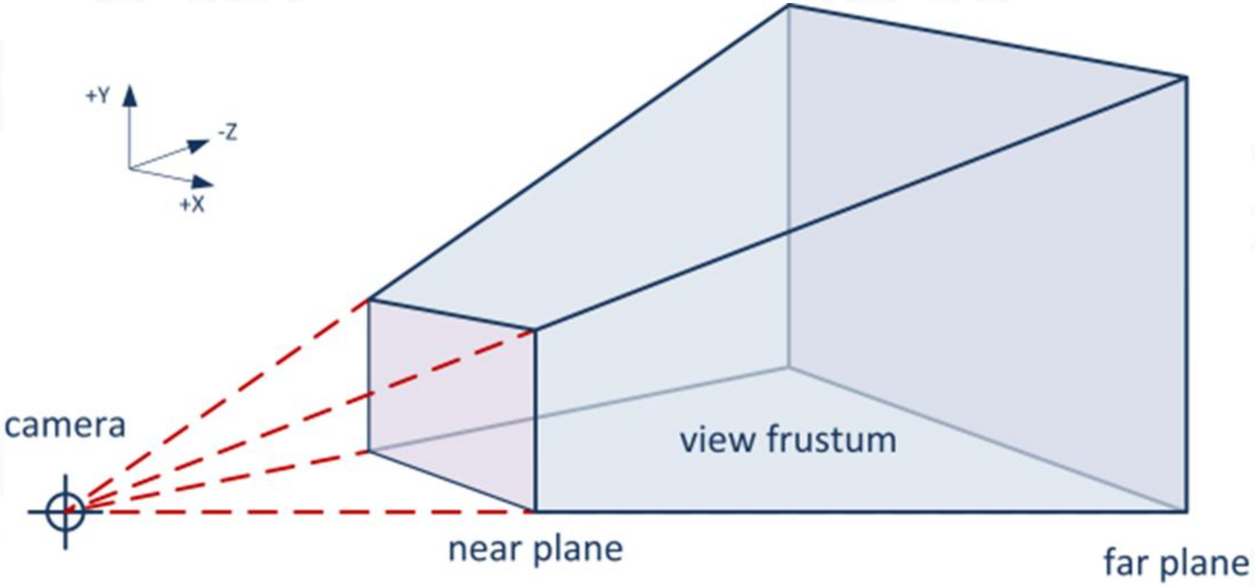
$$L_o(\mathbf{x},\omega_o) = L_e(\mathbf{x},\omega_o) + \int_{H^2} f_r(\mathbf{x},\omega_o,\omega_i)L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i$$

outgoing emitted reflected

What We Learned about Rendering (1/4)

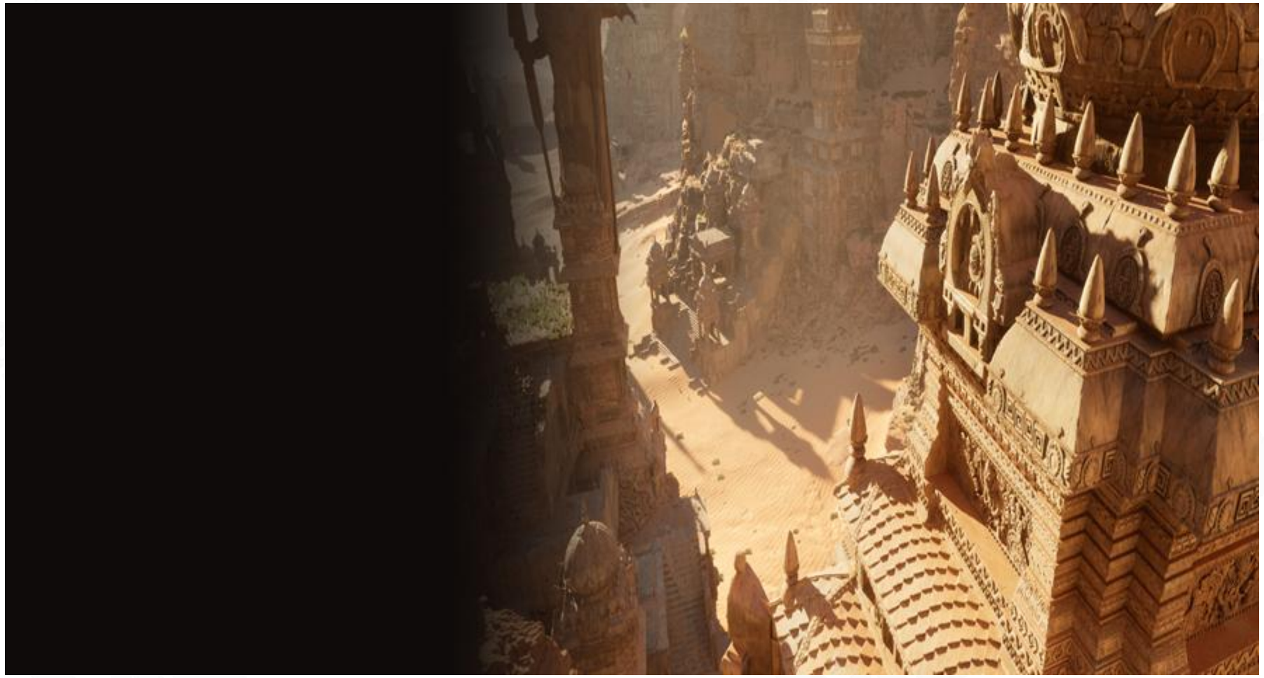


Rendering objects with meshes, texture and shaders



Culling

What We Learned about Rendering (2/4)



Lighting, Shadow and Global Illumination

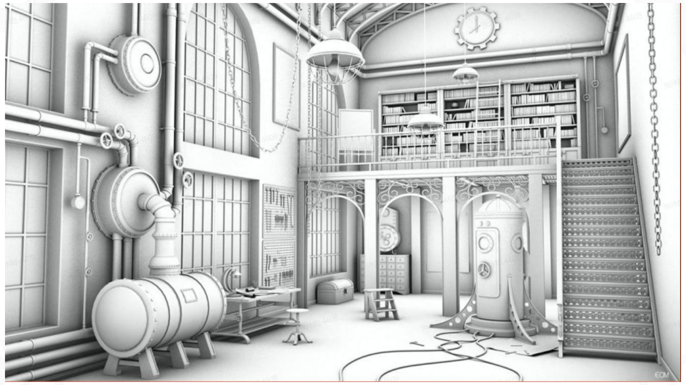
PBR Materials

What We Learned about Rendering (3/4)

Terrain

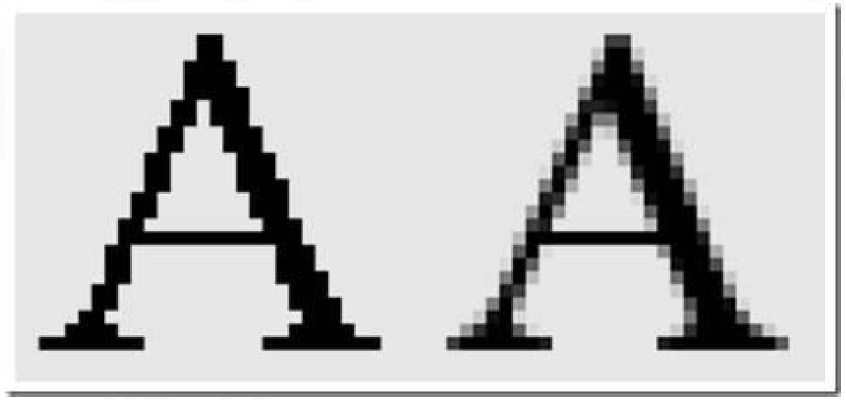
Sky and Cloud

What We Learned about Rendering (4/4)

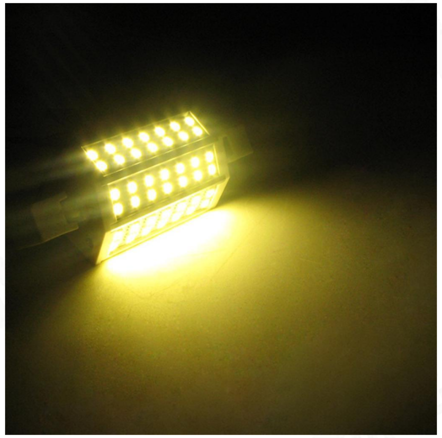


Ambient Occlusion

Fog

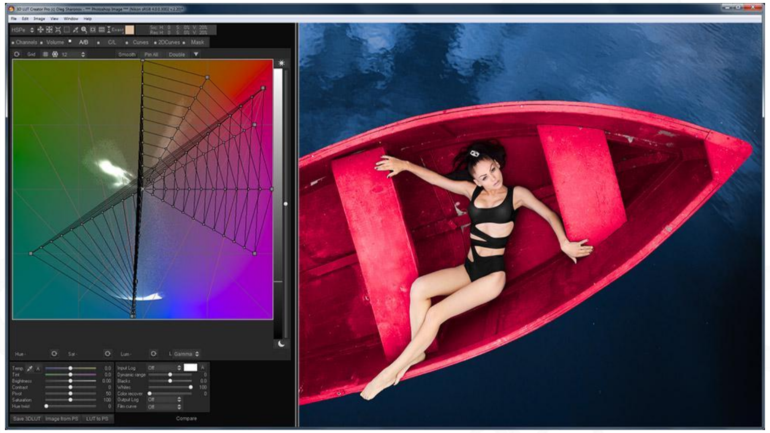


Anti-aliasing



Bloom

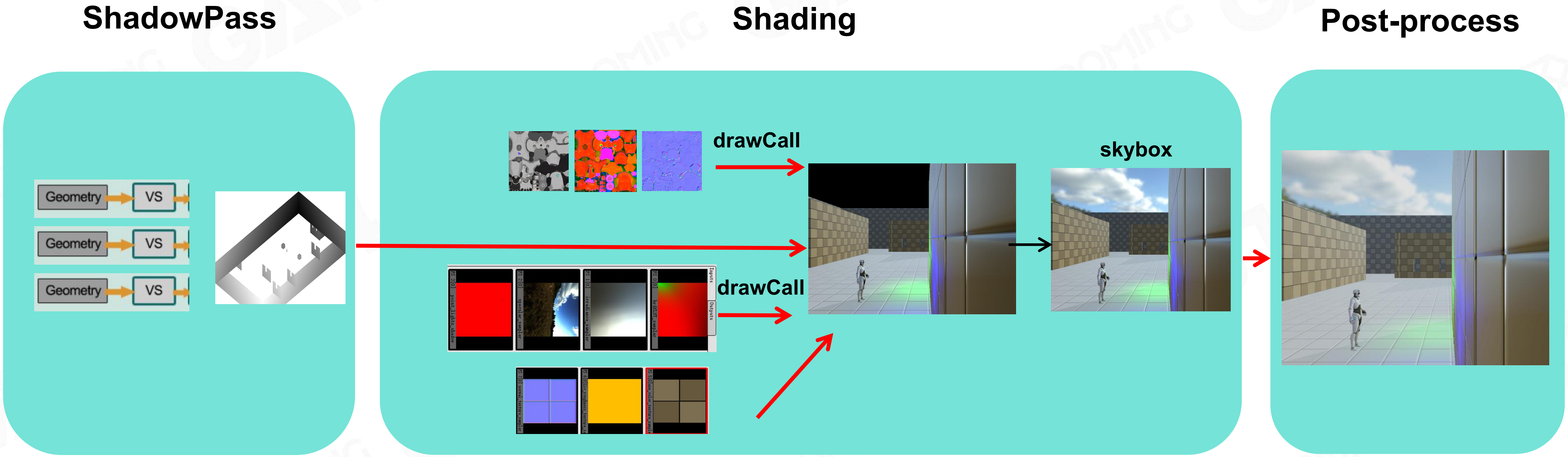
Tone Mapping



Color Grading

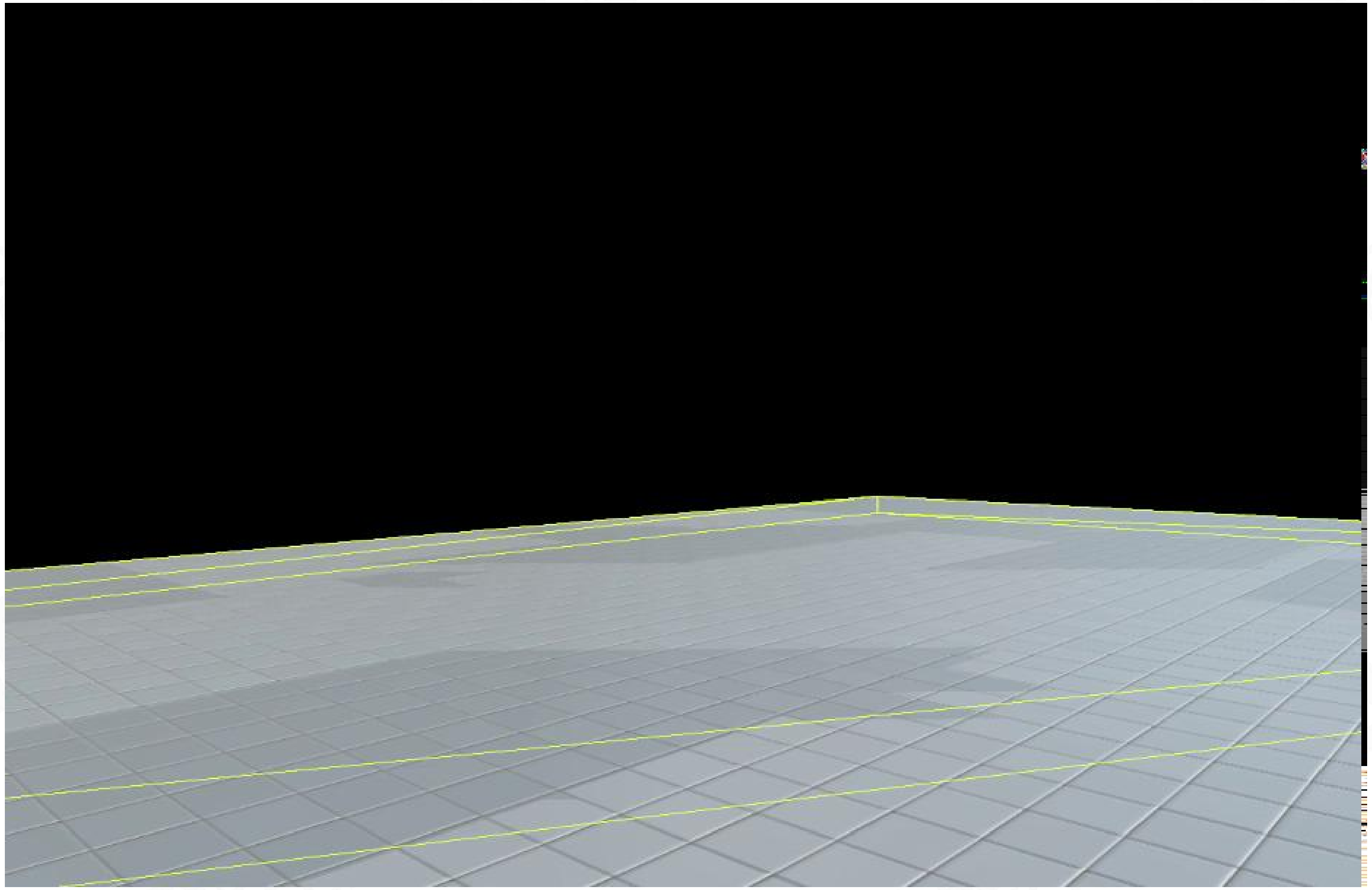
Rendering Pipeline

- **Rendering pipeline** is the management order of all rendering operation execution and resource allocation

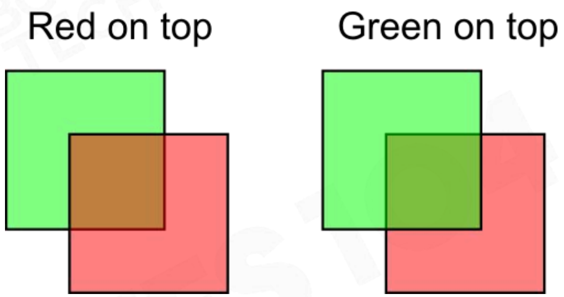


Forward Rendering

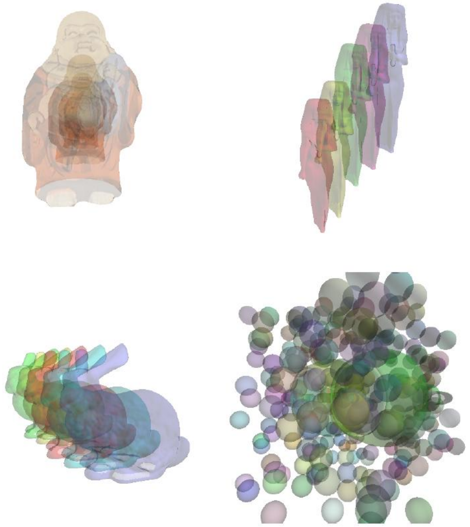
```
for n meshes
    for m lights
        color += shading(mesh, light)
```



Sort and Render Transparent after Opaque Objects



Transparent Order



Render from far to near

Forward Rendering

Just Cause 1 2006

Heavy Rain 2010

Rendering with Many Lights

Deferred Rendering

```
for each object
    write G-Buffer
```

Pass 1

```
for each pixel
    gbuffer = readGBuffer(G-Buffer)
    for each light
        computeShading(gbuffer, light)
```

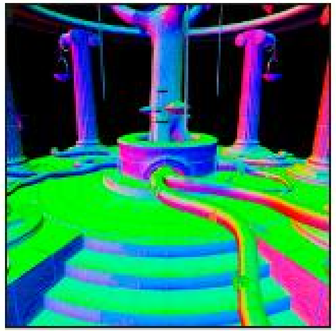
Pass 2

Pass 1: Rendering G-Buffer

Albedo

Specular

Normals



Depth

Final render

Pass 2: Deferred Shading

Deferred Rendering

Pros

- Lighting is only computed for visible fragments
- The data from the G-Buffer can be used for post-processing

Cons

- High memory and bandwidth cost
- Not supporting transparent object
- Not friendly to MSAA

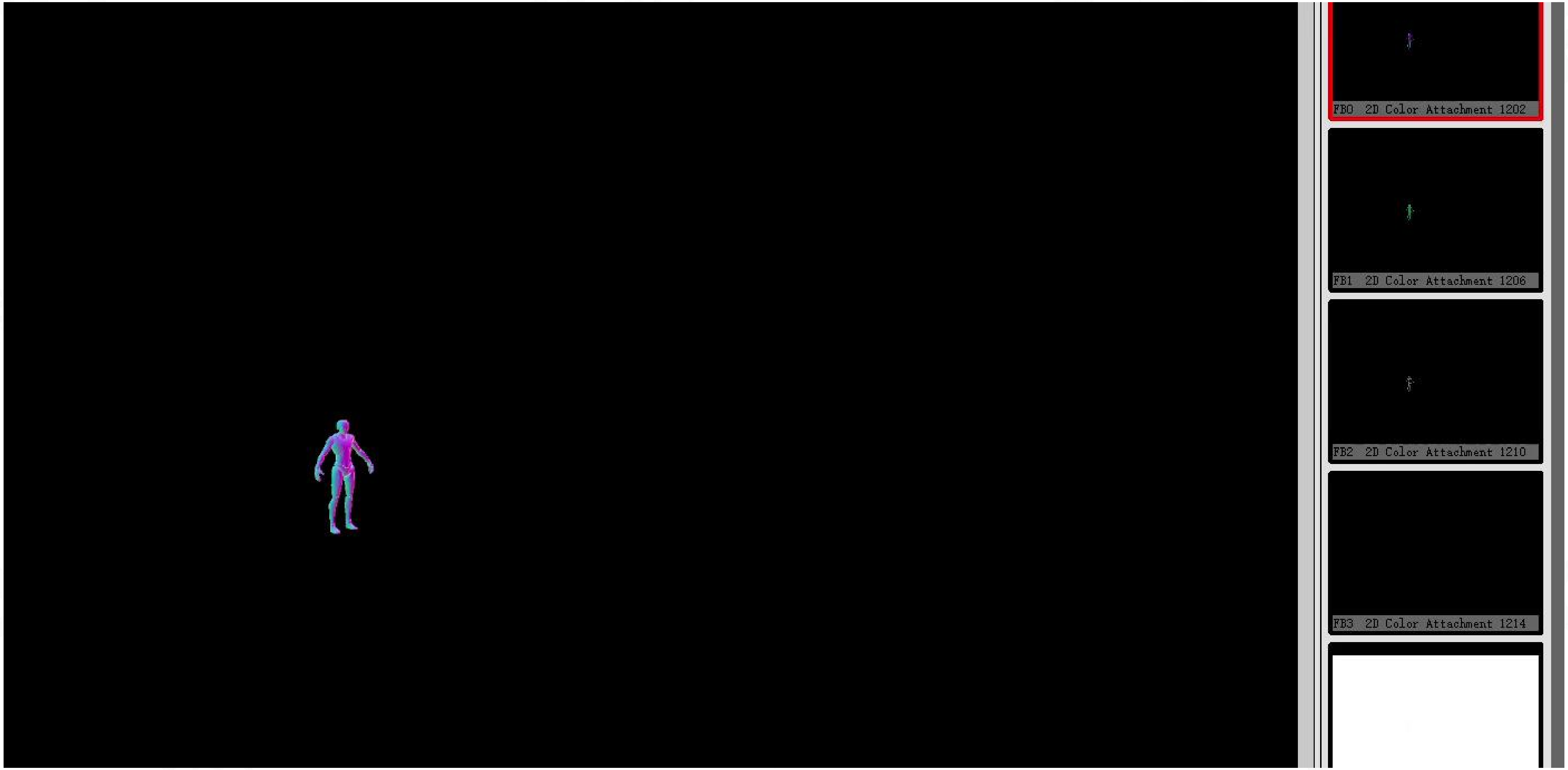
	R	G	B	A
GB0	Normal (10:10)		Smoothness	MaterialId (2)
GB1	BaseColor			MatData(5)/Normal(3)
GB2	---------	MetalMask	Reflectance	AO
GB3	Radiosity/Emissive			

Table 2: GBuffer layout for Disney deferred base material.

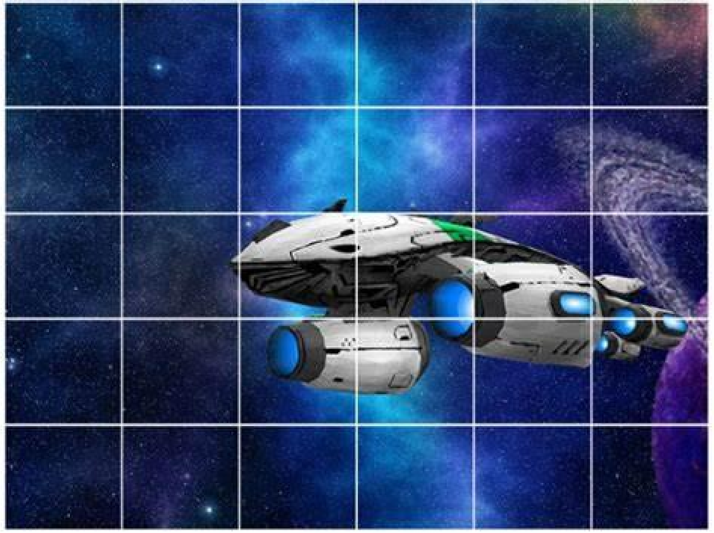
G-Buffer Size:1920*1080, 32bit*1920*1080*4 = 63MB

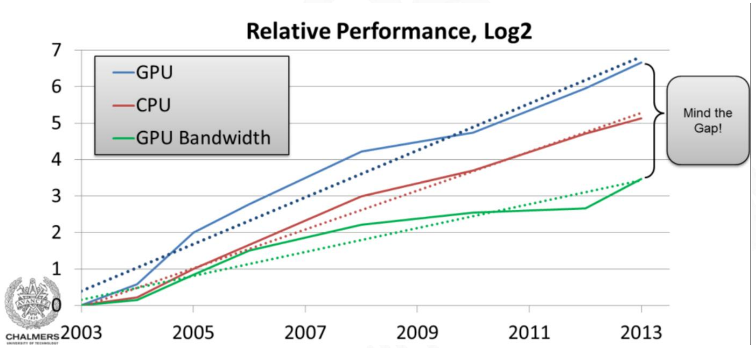
Scene with Many Lights

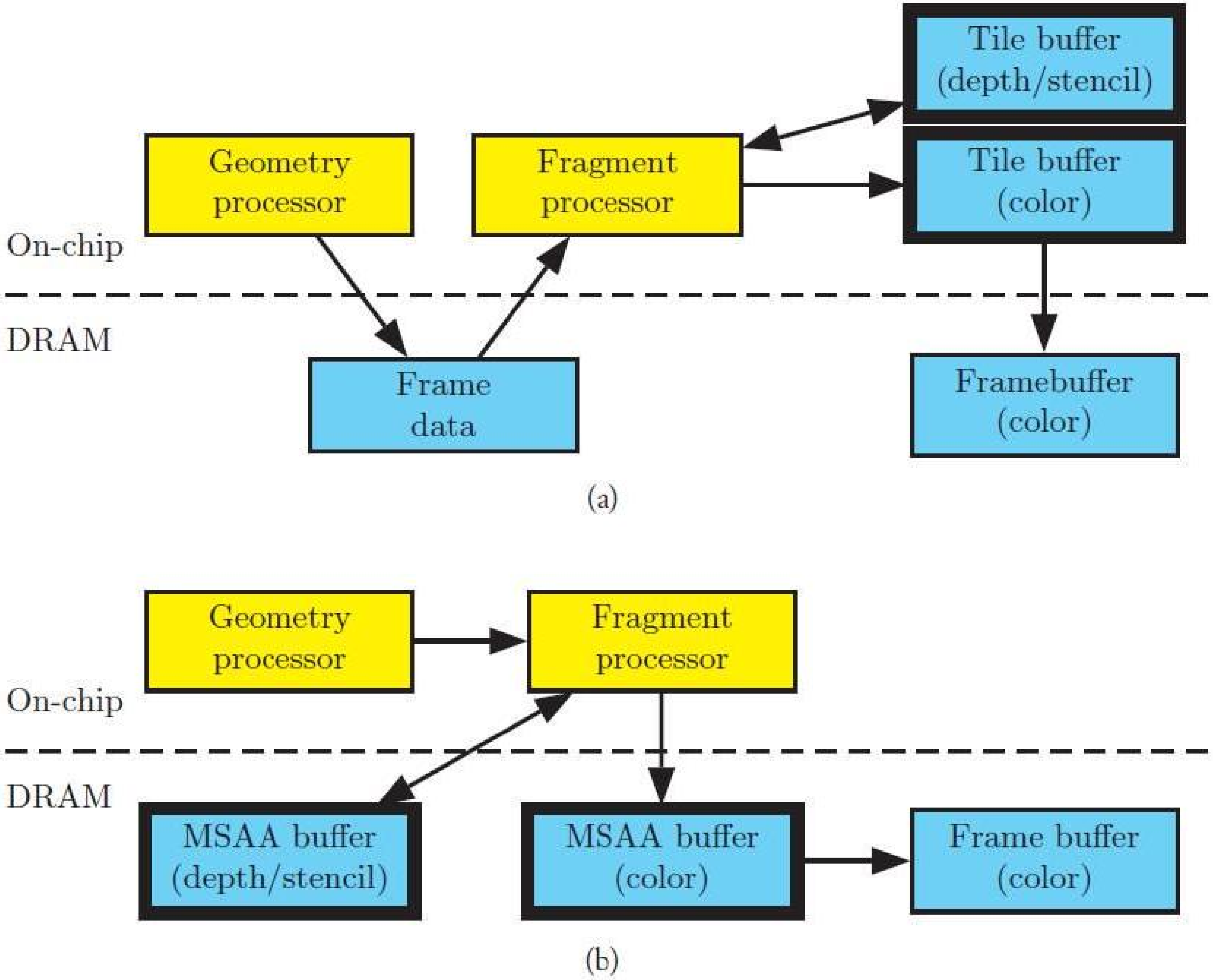
Pilot Engine Deferred Rendering



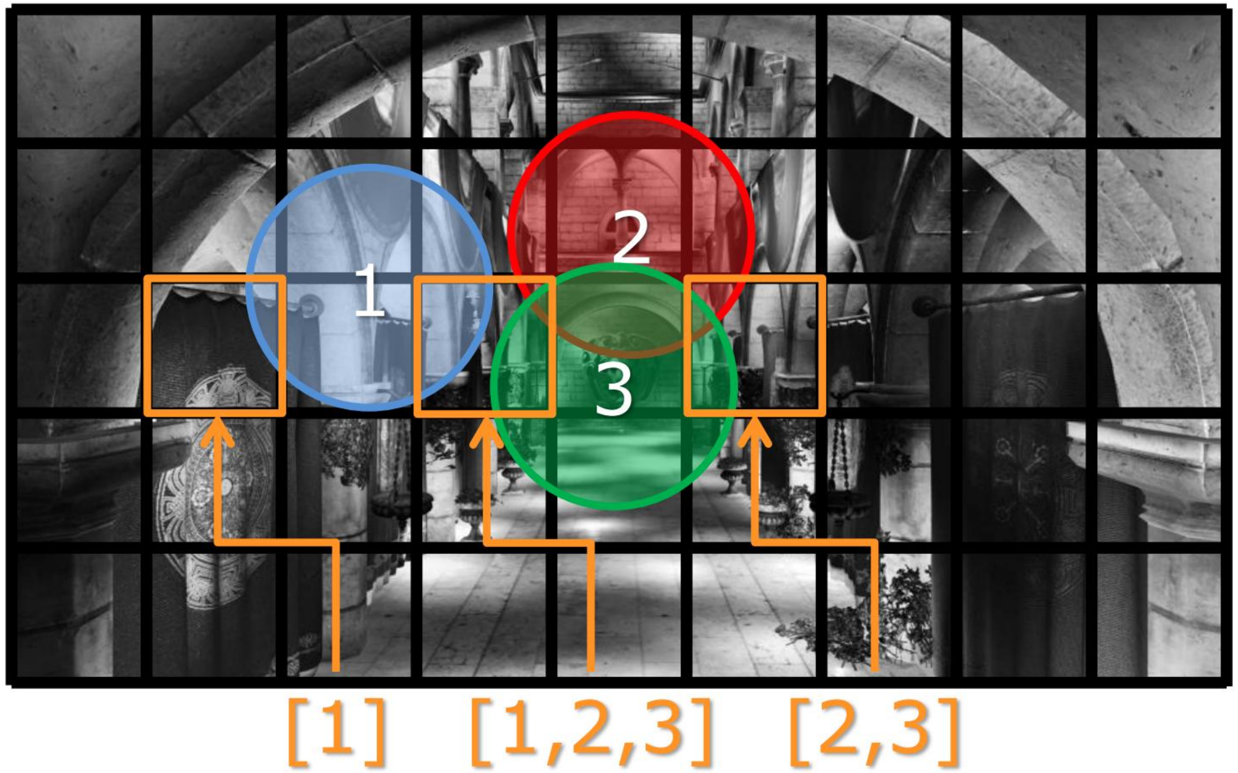
Tile-based Rendering







Light Culling by Tiles



Light List in a Screen Tile

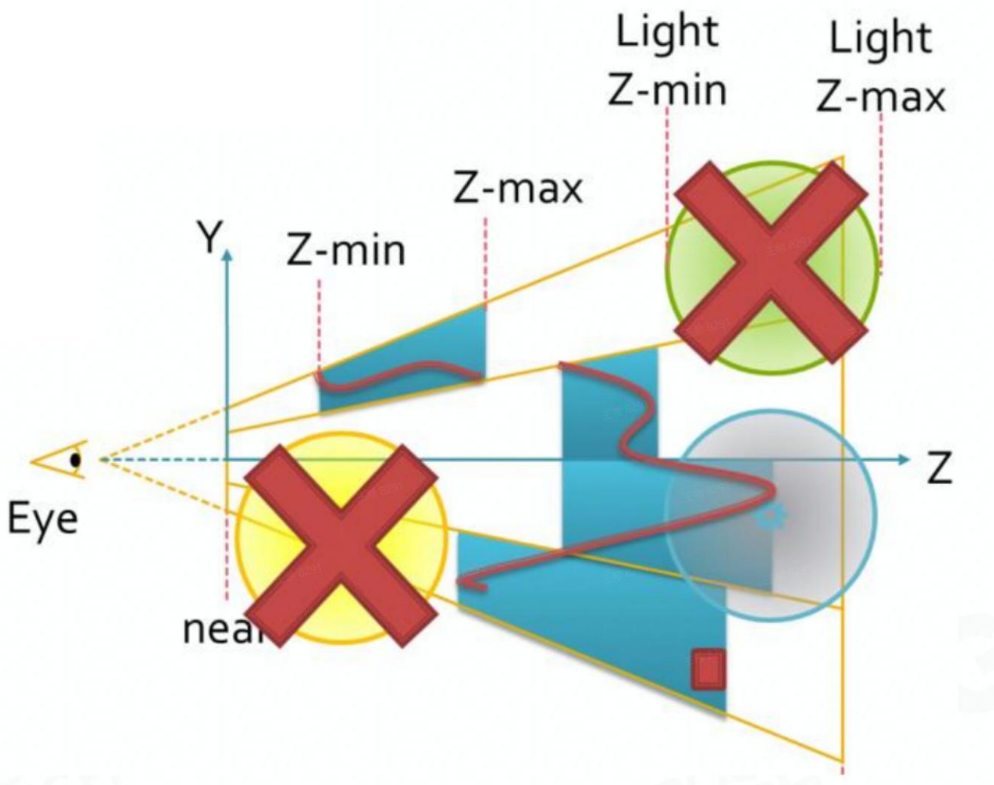
```
for each light
  for each covered pixel
    read G-Buffer
    compute shading
    read + write frame buffer
```

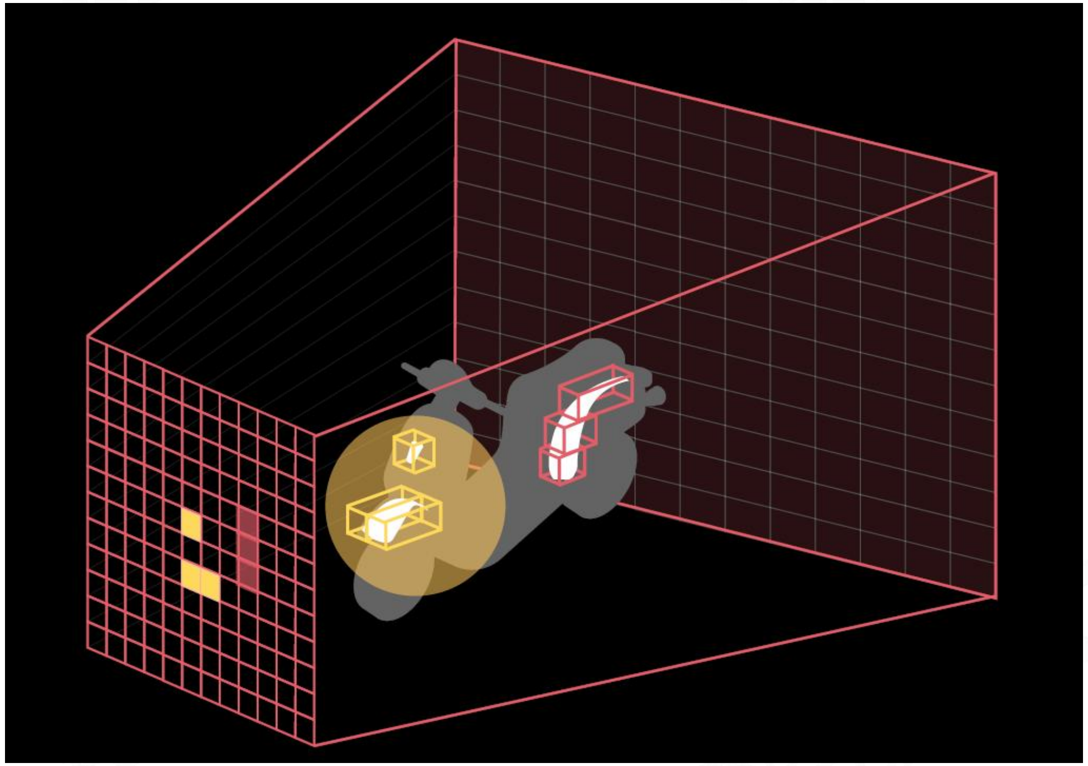
Re-order loops
Load/store -> Outer loop

```
for each pixel
 read G-Buffer
 for each affecting light
   compute shading
 write frame buffer
```

Depth Range Optimization

- Get Min/Max depth per tile from Pre-z pass
- Test depth bounds for each light





Tile-based Deferred Rendering

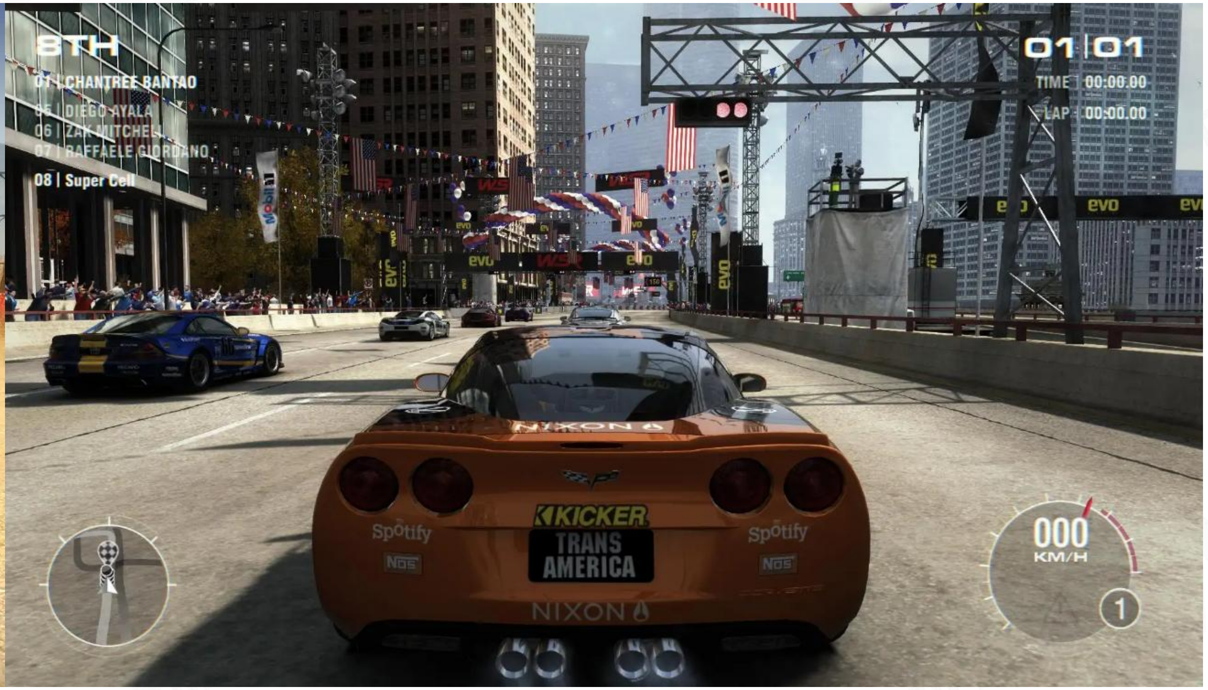
Battlefield 4

Ryse

Forward+ (Tile-based Forward) Rendering

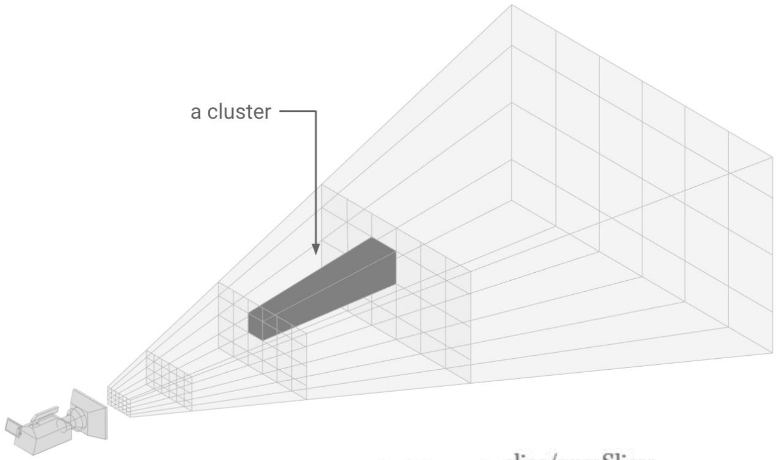
- Depth prepass (prevent overdraw / provide tile depth bounds)
- Tiled light culling (output: light list per tile)
- Shading per object (PS: Iterate through light list calculated in light culling)

DIRT



GRID

Cluster-based Rendering



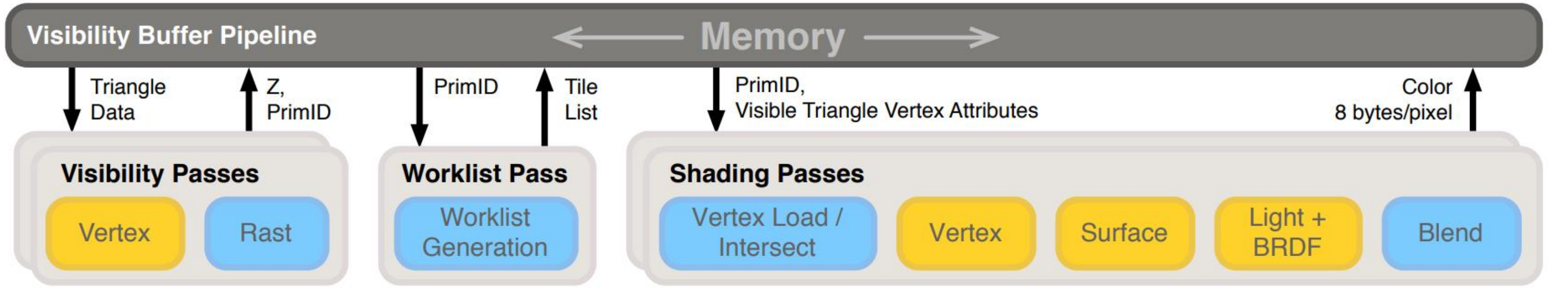
$$Z = \mathrm{Near}_z \left(\frac{\mathrm{Far}_z}{\mathrm{Near}_z} \right)^{\mathrm{slice}/\mathrm{numSlices}}$$

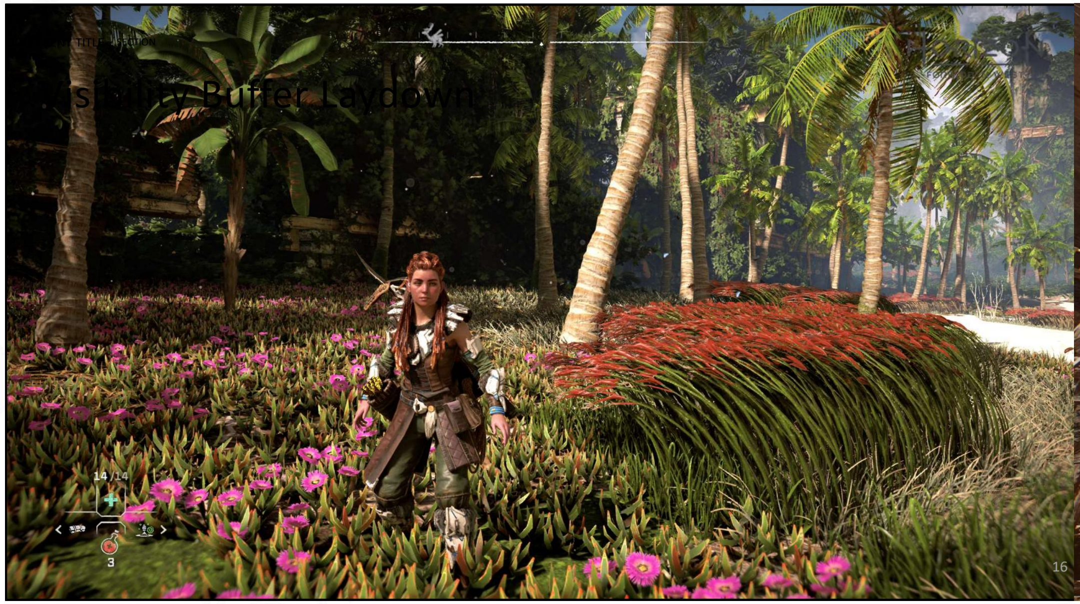
$$\mathrm{slice} = \left\lfloor \log(Z) \frac{\mathrm{numSlices}}{\log(\mathrm{Far}_z/\mathrm{Near}_z)} - \frac{\mathrm{numSlices} \times \log(\mathrm{Near}_z)}{\log(\mathrm{Far}_z/\mathrm{Near}_z)} \right\rfloor$$

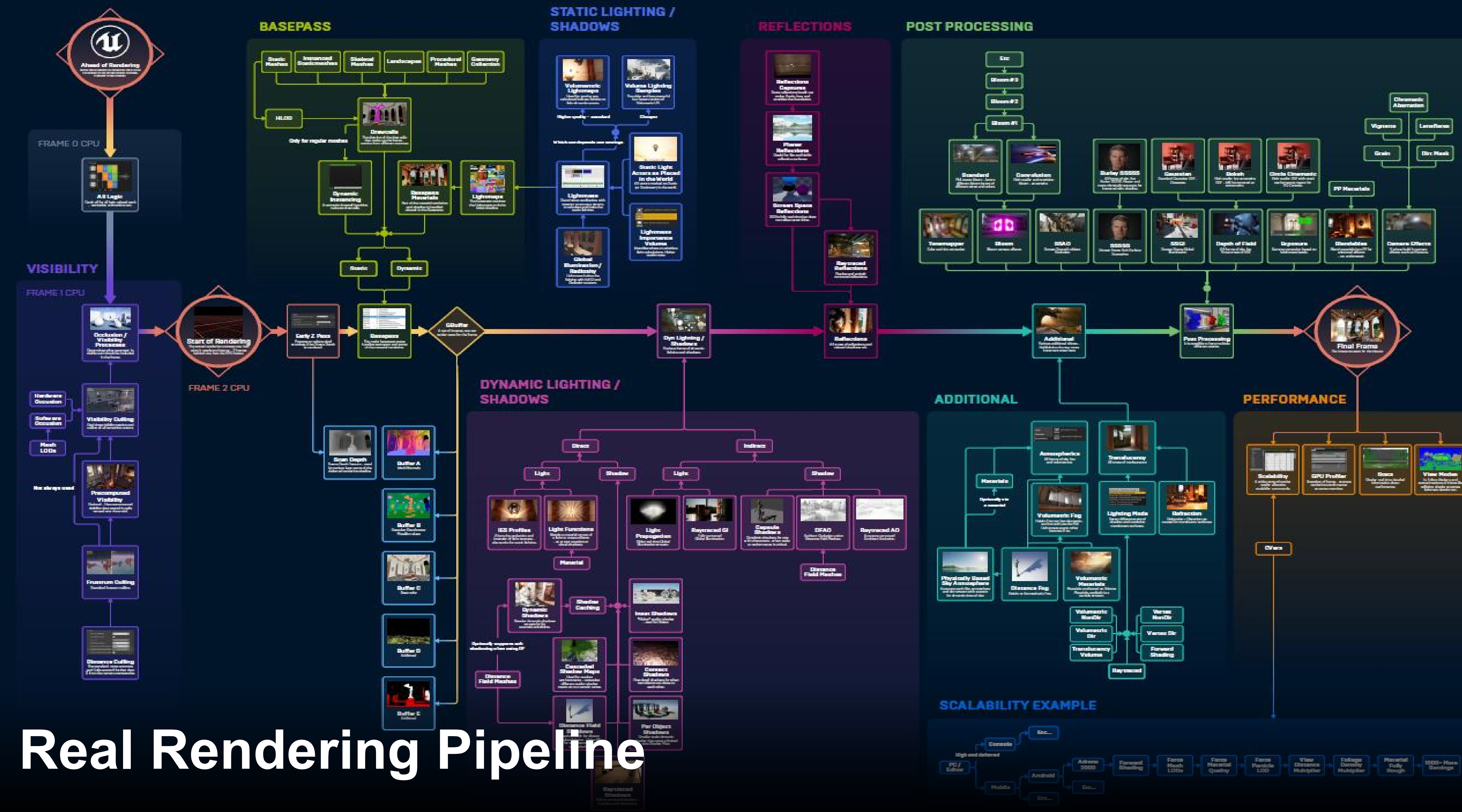
Doom 2016

Visibility Buffer

G-Buffer	V-Buffer
Depth	*Depth*
Normal	PrimitiveID
Albedo	Barycentrics
Roughness	



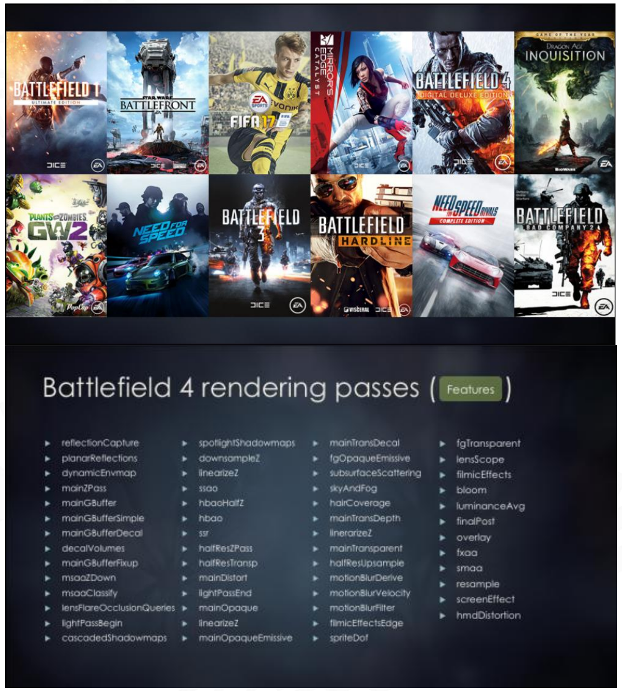


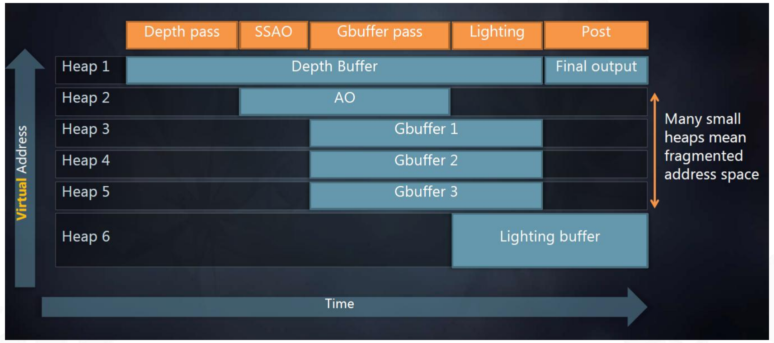


Real Rendering Pipeline

Challenges

- Complex parallel work needs to synchronize with complex resource dependency
- Large amount of transient resource whose lifetime is shorter than one frame
- Complex resource state management
- Exploit newly exposed GPU features without extensive user low level knowledge

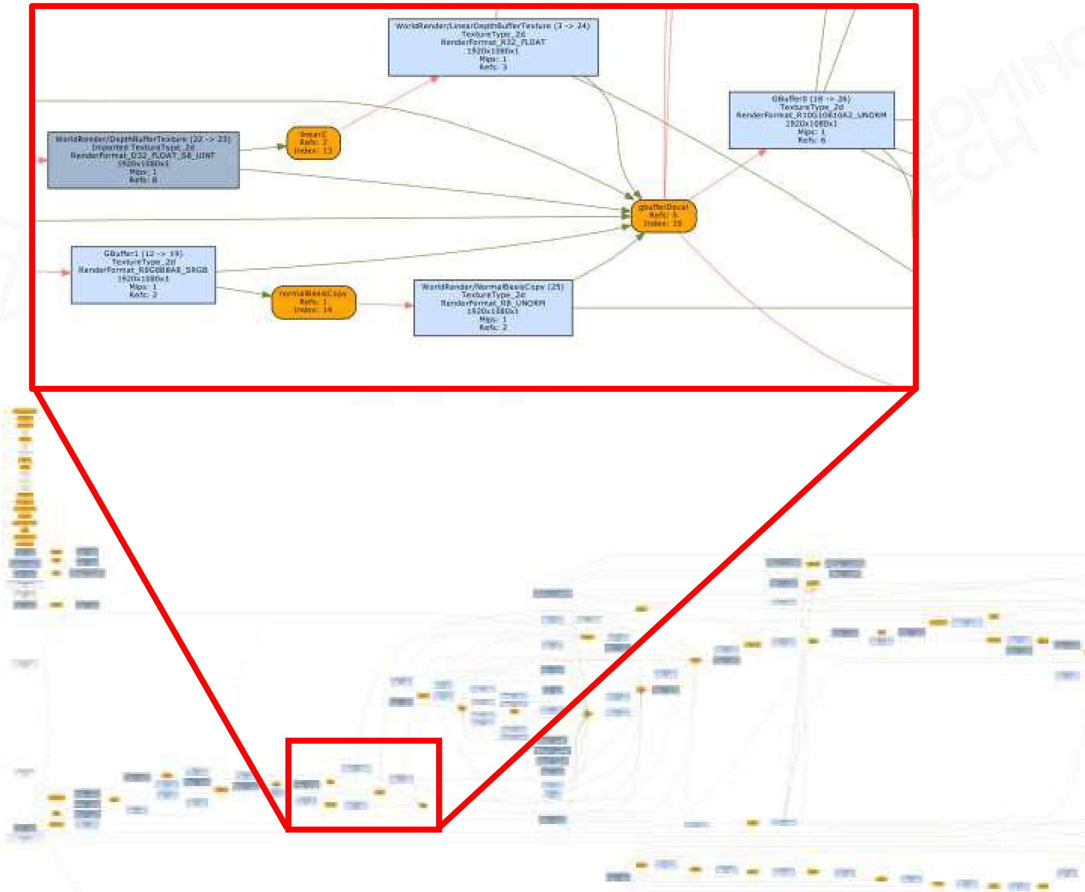


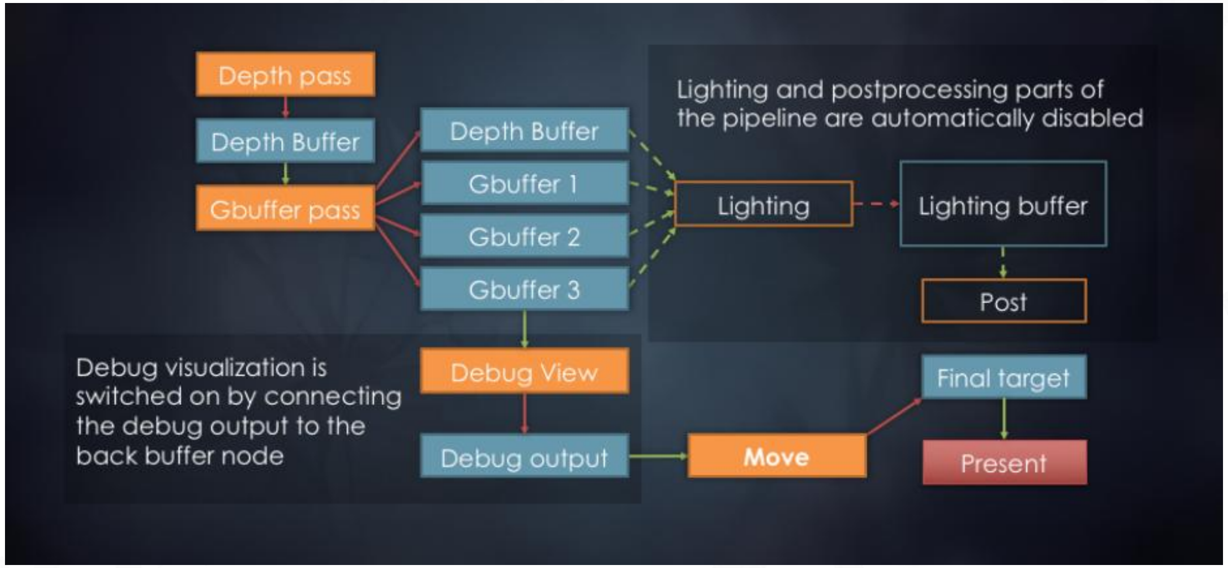


Vulkan DirectX 12 Microsoft® DirectX® 12

Frame Graph

A Directed Acyclic Graph (DAG) of pass and resource dependency in a frame, not a real visual graph



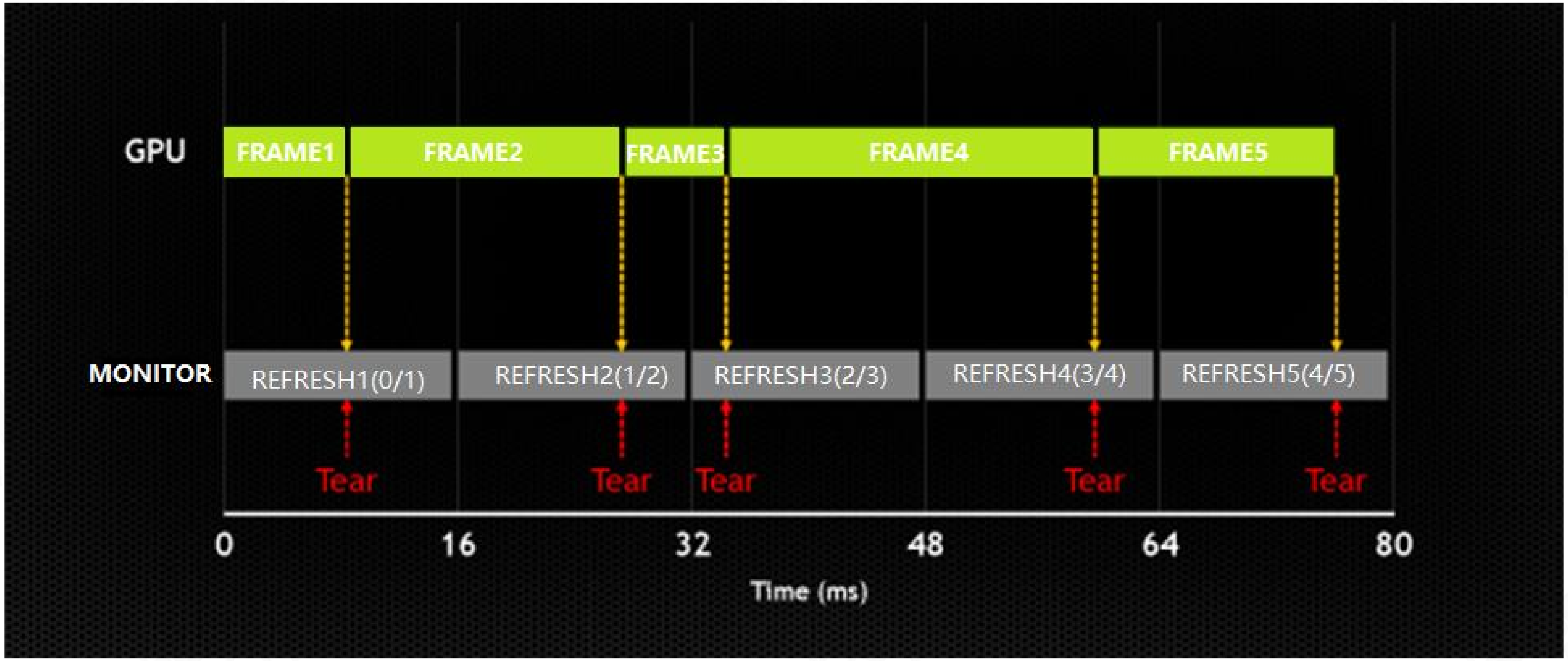


Render to Monitor

Screen Tearing

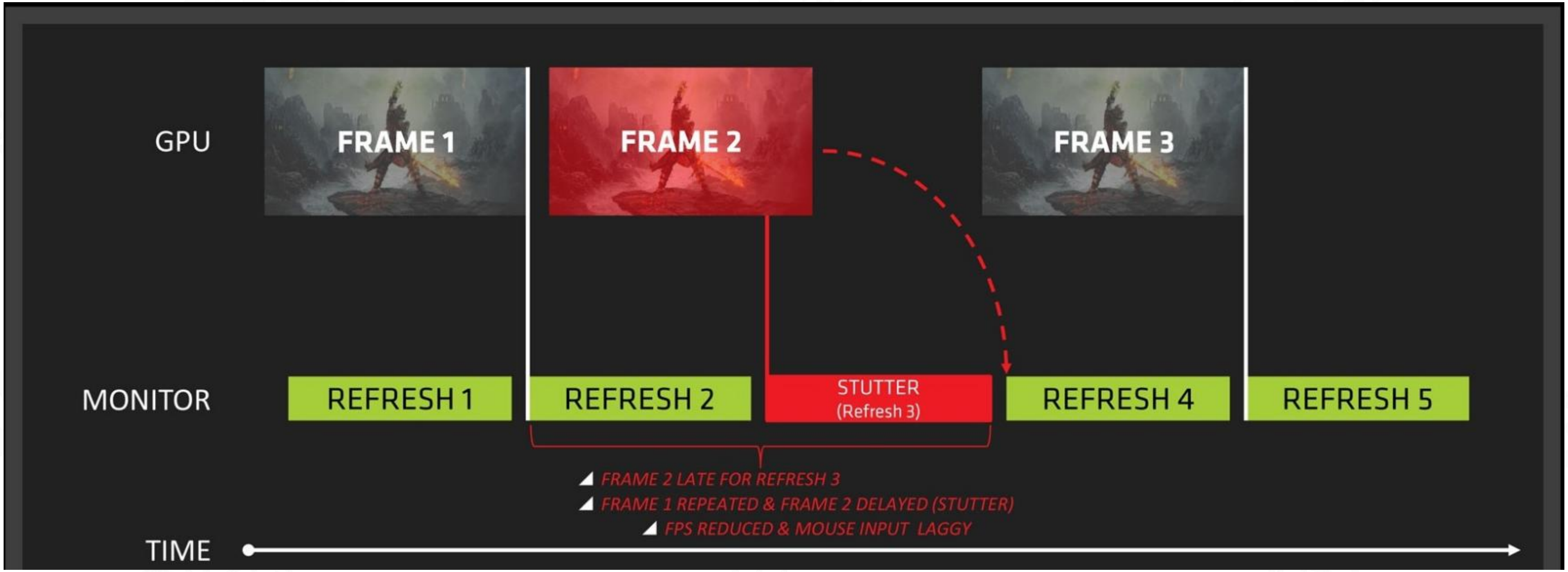
Screen Tearing

In most games your GPU frame rate will be highly volatile
When new GPU frame updates in the middle of last screen frame, screen tearing **occurrs**

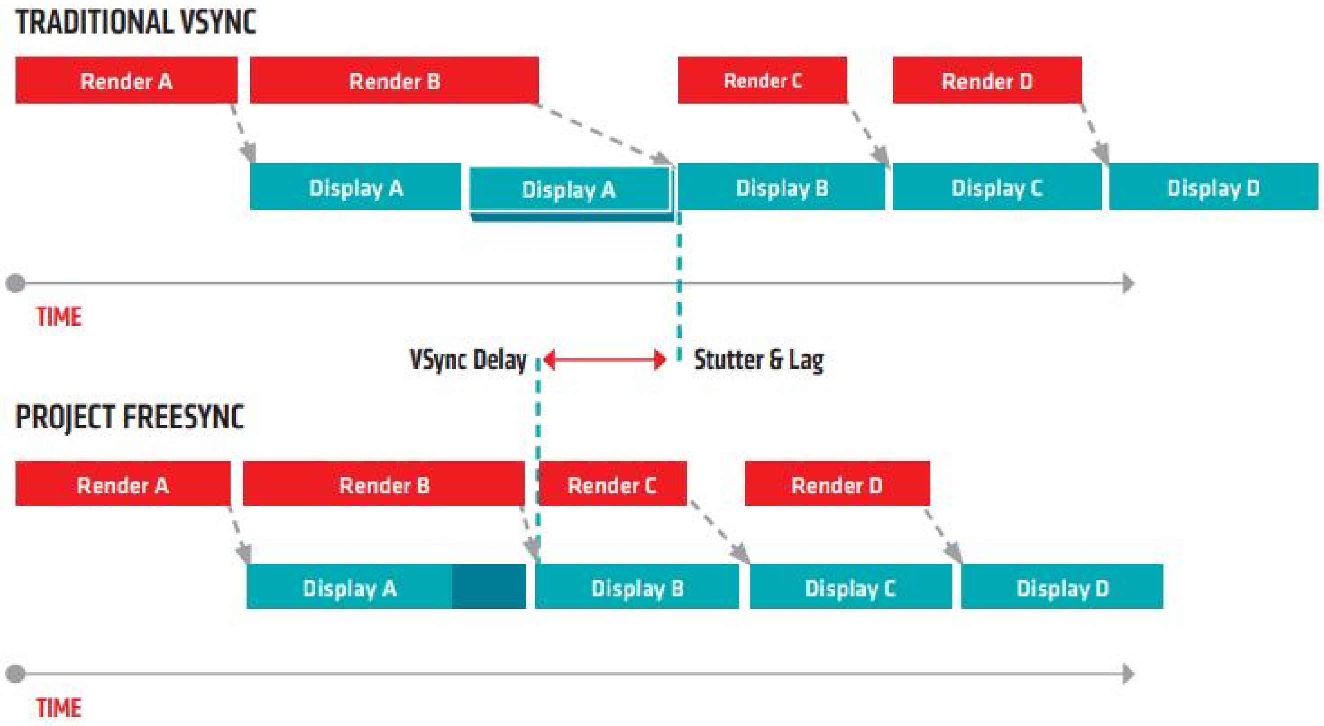


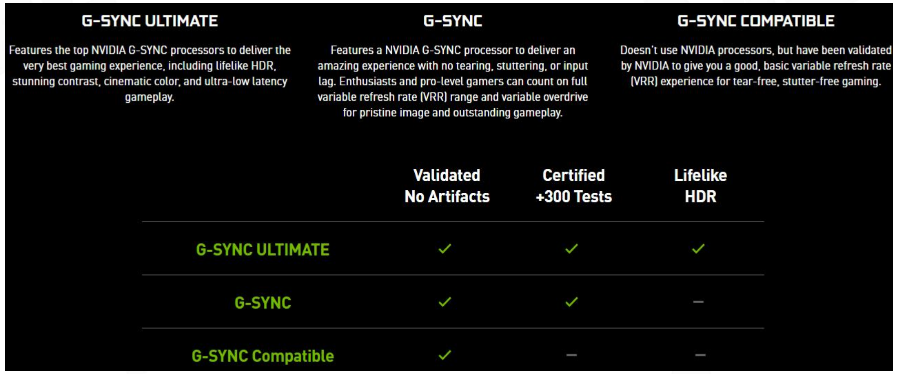
V-Sync Technology

Synchronizing buffer swaps with the Vertical refresh is called V-sync
V-Sync can be used to prevent tearing but framerates are reduced, the mouse is lagging & stuttering ruins gameplay



Variable Refresh Rate





G-SYNC ULTIMATE	G-SYNC	G-SYNC COMPATIBLE
Features the top NVIDIA G-SYNC processors to deliver the very best gaming experience, including lifelike HDR, stunning contrast, cinematic color, and ultra-low latency gameplay.	Features a NVIDIA G-SYNC processor to deliver an amazing experience with no tearing, stuttering, or input lag. Enthusiasts and pro-level gamers can count on full variable refresh rate (VRR) range and variable overdrive for pristine image and outstanding gameplay.	Doesn't use NVIDIA processors, but have been validated by NVIDIA to give you a good, basic variable refresh rate (VRR) experience for tear-free, stutter-free gaming.

	Validated No Artifacts	Certified +300 Tests	Lifelike HDR
G-SYNC ULTIMATE	✓	✓	✓
G-SYNC	✓	✓	–
G-SYNC Compatible	✓	–	–

AMD FreeSync

Every AMD FreeSync™ monitor goes through a rigorous certification process to ensure a tear free, low latency experience. Pair your Radeon™ graphics card with an AMD FreeSync monitor over HDMI® or DisplayPort™ for effortlessly smooth gameplay.

- Tear free experience
- Low latency

AMD FreeSync Premium

AMD FreeSync™ Premium[1] technology adds to the baseline FreeSync technology tier and equips serious gamers with a fluid, tear-free gameplay experience at peak performance:

- At least 120hz refresh rate at minimum FHD resolution
- Support for low framerate compensation (LFC)
- Low latency

AMD FreeSync Premium Pro

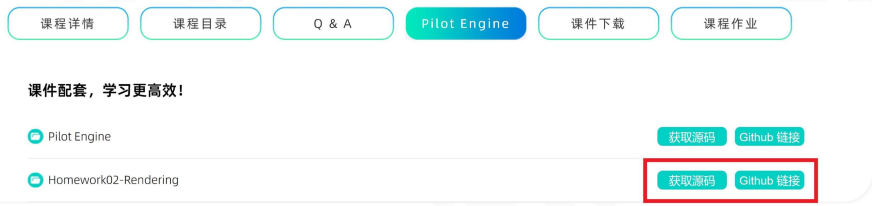
AMD FreeSync™ Premium Pro[1] technology raises the bar to the next level for gaming displays, enabling an exceptional user experience when playing HDR games, movies and other content:

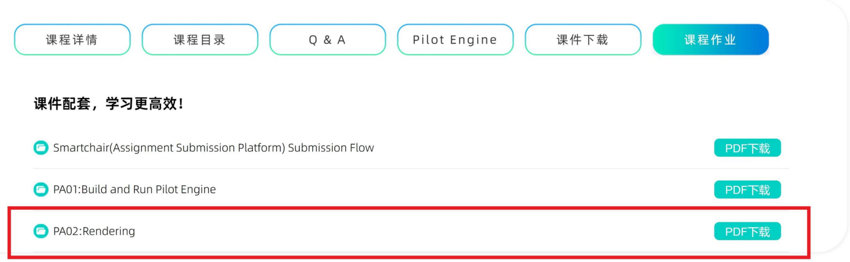
- At least 120hz refresh rate at minimum FHD resolution
- Support for low framerate compensation (LFC)
- Low latency in SDR and HDR
- Support for HDR with meticulous color and luminance certification

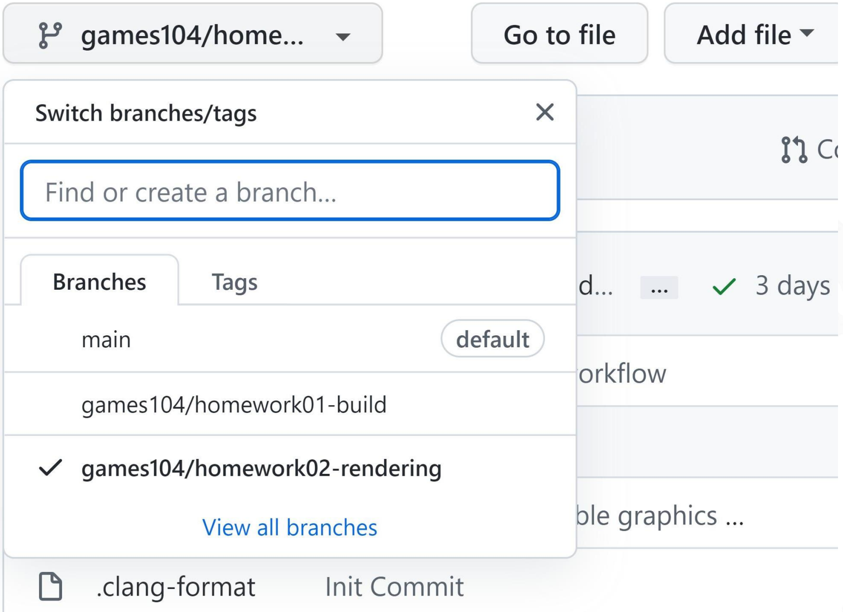
Homework 2

- You are supposed to...
  - Implement ColorGrading shader code
  - Generate own style ColorGrading result
  - Add a new post-process pass that you want (advanced)
  - Write a report document that contains screenshots of your results

- Download at
  - Course-site: https://games104.boomingtech.com/sc/course-list
  - Github: https://github.com/BoomingTech/Pilot/tree/games104/homework02-rendering







Pilot Engine V0.0.3 Releasing – April 26

New Features
- Deferred shading pipeline
- Configurable global rendering resource
- Motor system with accelerations
- Character-following camera blending

Bugfixes
- Fixed image layout transition in “pick” pass
- Fixed overlapped button and cursor twinkling

Optimizations
- Optimized display of rotation as Euler angles
- Optimized AMD and NVIDIA graphic device race when initializing Vulkan
- Optimized editor camera controlling

Contributors

Wlain, AirGuanZ, and 6 other contributors

PILOT
Game engine

Pilot Engine Learning

- The first version of the engine architecture document will be uploaded to Github Wiki and official website on April 30
- Videos of Pilot Engine source code walkthrough will be released in the near future

Labor Day Holiday Arrangement

- Lecture 08 on May 2 will be postponed to May 9
- All subsequent classes will be postponed

Q&A

Lecture 07 Contributor

- 一将
- 光哥
- 炯哥
- 玉林
- 小老弟
- 建辉
- 爵爷
- Jason
- 坤哥
- BOOK
- MANDY
- 婷姐
- 沛楠
- Leon
- 虎哥
- Shine
- 晨晨
- Judy
- QIUU
- C佬
- 阿乐
- 阿熊
- CC
- 大喷

Enjoy ;)
Coding

Course Wechat

Please follow us for further information

Please note that all videos and images and other media are cited from the Internet for demonstration only.