Advanced Lighting (3)

GPU Programming

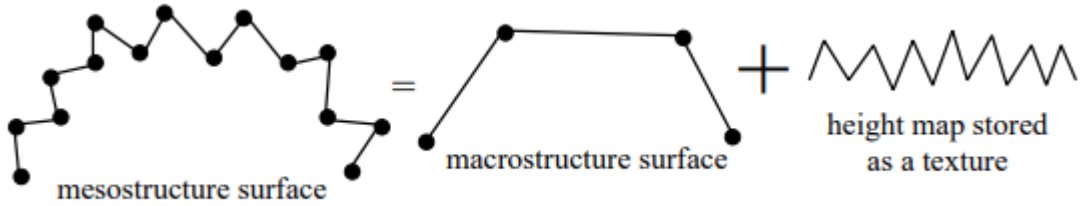
2022학년도
2학기

ⓒ nvidia

Displacement Mapping

Displacement Mapping

- 변위 매핑 (displacement mapping)
  - 매끈한 표면을 높이 맵(height map)에 따라 옮겨서(displace) 디테일을 추가해주는 기법



szirmay2006.pdf (fu-berlin.de)

- Per-vertex displacement mapping
  - Tessellation shader를 사용하여, geometry의 정점들(vertices)을 직접 수정

- Per-pixel displacement mapping (aka inverse displacement mapping)
  - Fragment shader를 사용하여, geometry의 변경 없이 fragment의 텍스처 좌표 수정
  - Parallax mapping, Steep Parallax Mapping, Parallax Occlusion Mapping, Pyramidal (quadtree) Displacement Mapping 등

Displacement Mapping

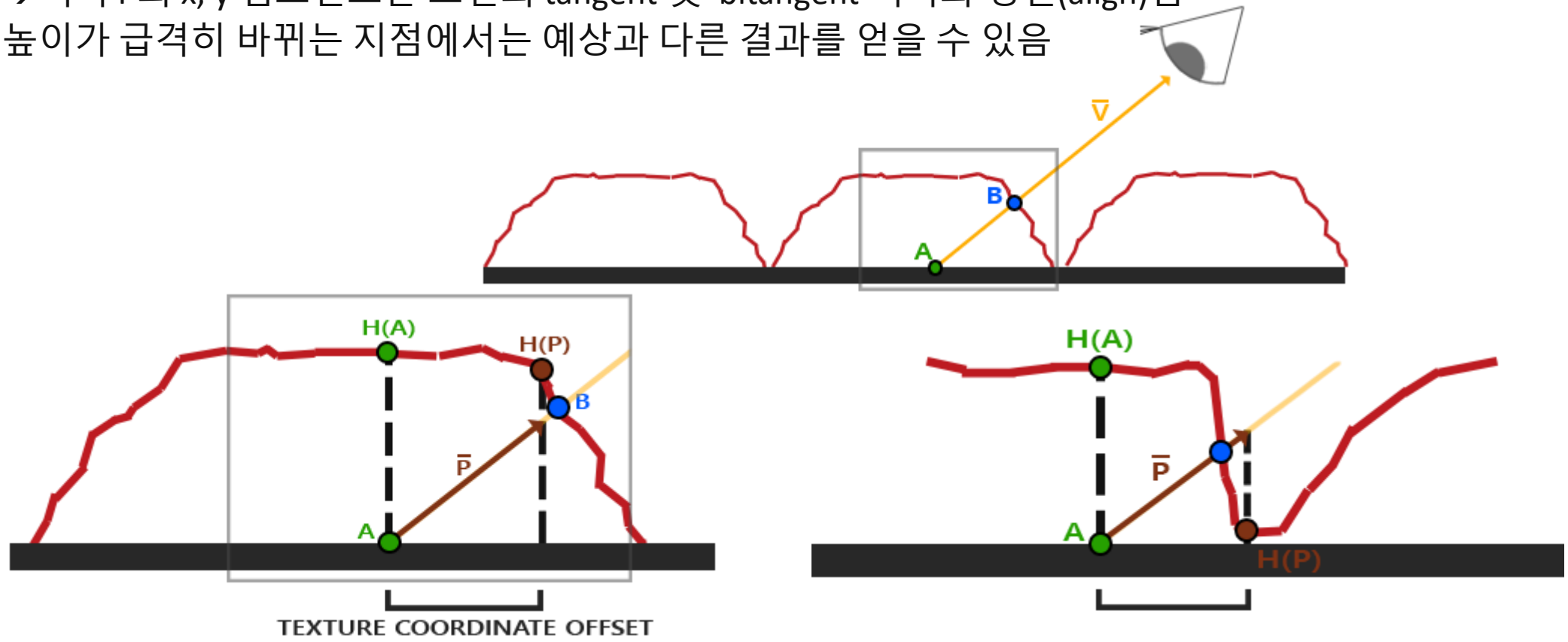
- 수많은 정점들로 이루어진 geometry를 사용하는 대신에 변위 매핑을 수행하는 이유는?
  - Level-of-details (LOD) 가능 – 시점에 가까운 물체는 세밀하게, 먼 물체는 간단하게 렌더링
  - CPU → GPU 데이터 이동 감소
  - Per-pixel displacement mapping은 vertex shading 비용까지 감소 가능

- 노멀 매핑 대신에 변위 매핑을 수행하는 이유는?
  - 단순히 노멀값만 바꾸는 것에 비해
    훨씬 더 효과적으로 깊이감을 전달 가능
  - 특히 뉘어서 보는 시점 또는 그림자가 지는 경우 큰 차이가 남

Trainz Dev Diary - Parallax vs Normal Mapped Objects - YouTube

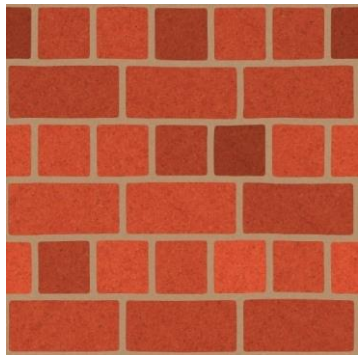
Parallax Mapping

- 텍스처 좌표를 수정함으로써 fragment의 표면을 실제보다 높거나 낮게 보이게 하는 대표적인 방법
  - Fragment-to-view 방향(V)으로, 높이 맵(H)에서 읽어 들인 높이에 비례하여 텍스처 좌표를 이동(offset)
  - 표면이 임의의 방향으로 회전되어 있어도 동작이 가능하도록 V를 tangent space로 변환하여 사용(P)
    →벡터 P의 x, y 컴포넌트는 표면의 tangent 및 bitangent 벡터와 정렬(align)됨
  - 높이가 급격히 바뀌는 지점에서는 예상과 다른 결과를 얻을 수 있음

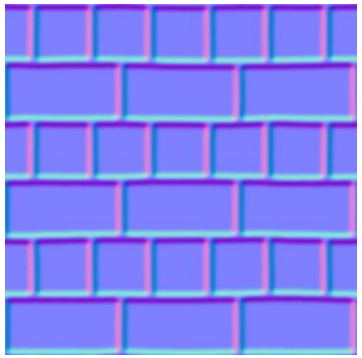


Parallax Mapping

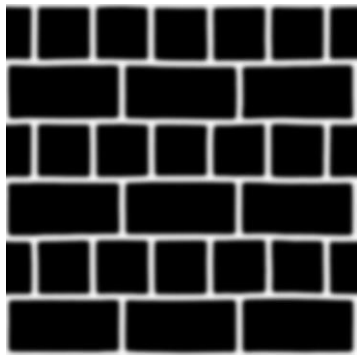
- Parallax mapping 예제에서 쓰이는 세 가지 맵



Diffuse map



Normal map

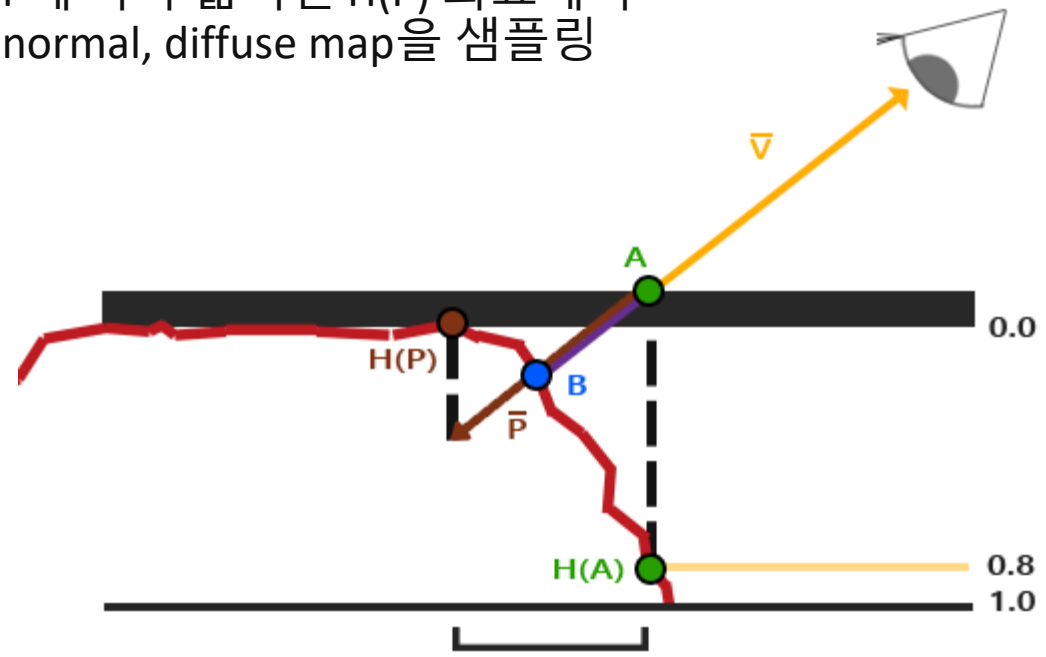


Displacement map

- 위와 같이 displacement map이 height map의 역일 경우
  - Height가 아닌 depth로 고려해 처리

Parallax Mapping

- FS에서의 구현
  - P: V의 반대 방향(viewDir.xy)으로 향하는 2차원 벡터로, viewDir.z에 반비례, H(A)에 비례함
  - P에 따라 옮겨진 H(P) 좌표에서 normal, diffuse map을 샘플링



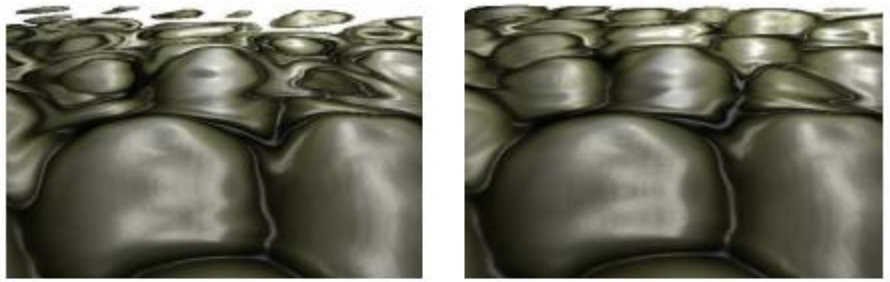

```
vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
    float height =  texture(depthMap, texCoords).r;
    vec2 p = viewDir.xy / viewDir.z * (height * height_scale);
    return texCoords - p;
}
```

FS

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} fs_in;

uniform sampler2D diffuseMap;
uniform sampler2D normalMap;
uniform sampler2D depthMap;

uniform float height_scale;

vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir);

void main()
{
    // offset texture coordinates with Parallax Mapping
    vec3 viewDir   = normalize(fs_in.TangentViewPos - fs_in.TangentFragPos);
    vec2 texCoords = ParallaxMapping(fs_in.TexCoords,  viewDir);

    // then sample textures with new texture coords
    vec3 diffuse = texture(diffuseMap, texCoords);
    vec3 normal  = texture(normalMap, texCoords);
    normal = normalize(normal * 2.0 - 1.0);
    // proceed with lighting code
    [...]
}
```

Parallax Mapping (시차 매핑)

- Parallax mapping with offset limiting
  - viewDir.z으로 나누는 부분을 삭제하여, offset의 범위를 heightmap의 범위와 일치시킴
  - 시점 벡터(V)와 표면이 평행에 가까운 경우 offset이 무한대로 늘어날 수 있는 오류를 방지
  - 대신, V와 표면이 수직에 가까운 경우 시차 효과가 떨어짐
  - Parallax mapping은 근사(approximation) 기법이므로, offset은 상황에 맞게 설정하면 됨

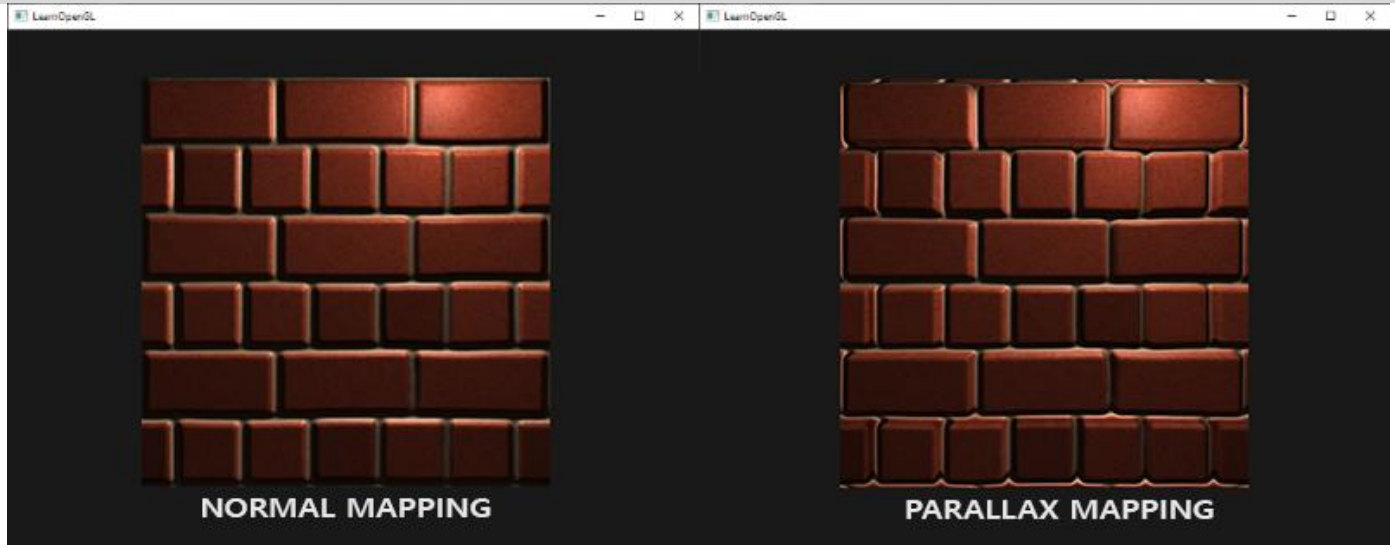


FPS = 675 FPS = 680

Figure 12: *Comparison of parallax mapping (left) and parallax mapping with offset limiting (right) setting BIAS* $= -0.14$ *and SCALE* $= 0.16$.

Parallax Mapping

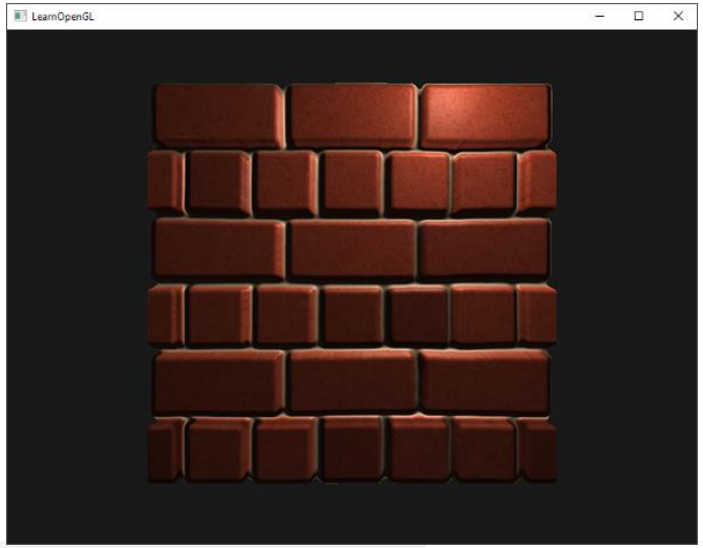
- Parallax mapping 실행 결과



- 기본 범위를 벗어난 fragment를 버리면,
  평면의 edge는 실루엣을 올록볼록하게 바꿀 수 있음

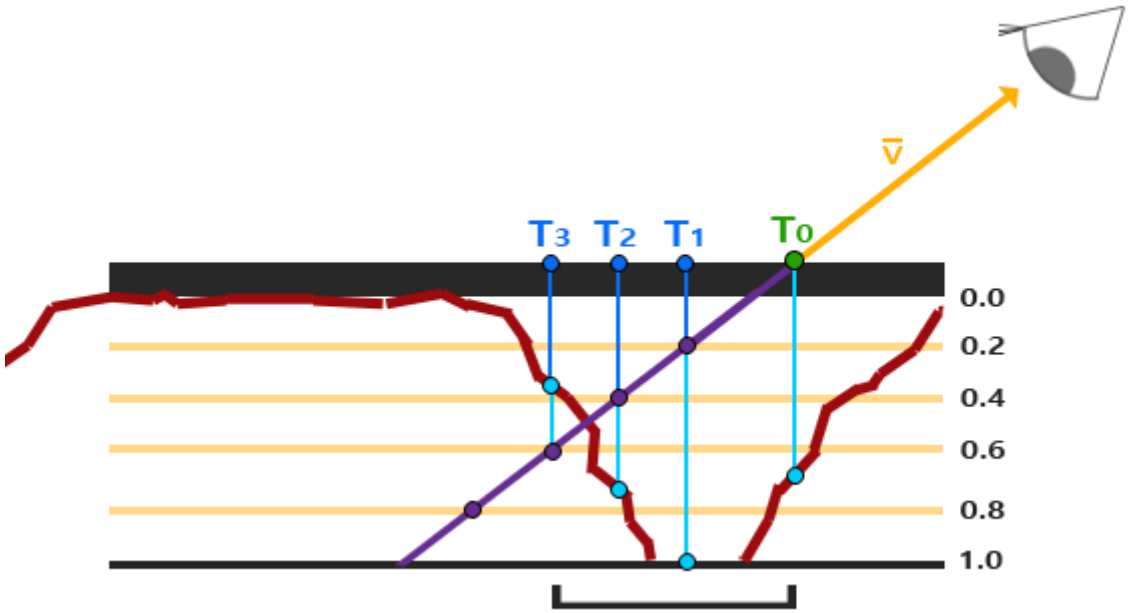
FS

```
texCoords = ParallaxMapping(fs_in.TexCoords,  viewDir);
if(texCoords.x > 1.0 || texCoords.y > 1.0 || texCoords.x < 0.0 || texCoords.y < 0.0)
    discard;
```



Steep Parallax Mapping

- Parallax mapping에서 샘플 개수를 1개에서 다수개로 확장
  - 가파른(steep) 높이 변화에도 대응 가능
- Ray와 height/depth field간 교차 검사 수행
  - 교차 지점에서 탐색을 멈추고 offset 설정




```
vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
    // number of depth layers
    const float numLayers = 10;
    // calculate the size of each layer
    float layerDepth = 1.0 / numLayers;
    // depth of current layer
    float currentLayerDepth = 0.0;
    // the amount to shift the texture coordinates per layer (from vector P)
    vec2 P = viewDir.xy * height_scale;
    vec2 deltaTexCoords = P / numLayers;

    [...]
}
```

FS

```
// get initial values
vec2  currentTexCoords     = texCoords;
float currentDepthMapValue = texture(depthMap, currentTexCoords).r;

while(currentLayerDepth < currentDepthMapValue)
{
    // shift texture coordinates along direction of P
    currentTexCoords -= deltaTexCoords;
    // get depthmap value at current texture coordinates
    currentDepthMapValue = texture(depthMap, currentTexCoords).r;
    // get depth of next layer
    currentLayerDepth += layerDepth;
}

return currentTexCoords;
```

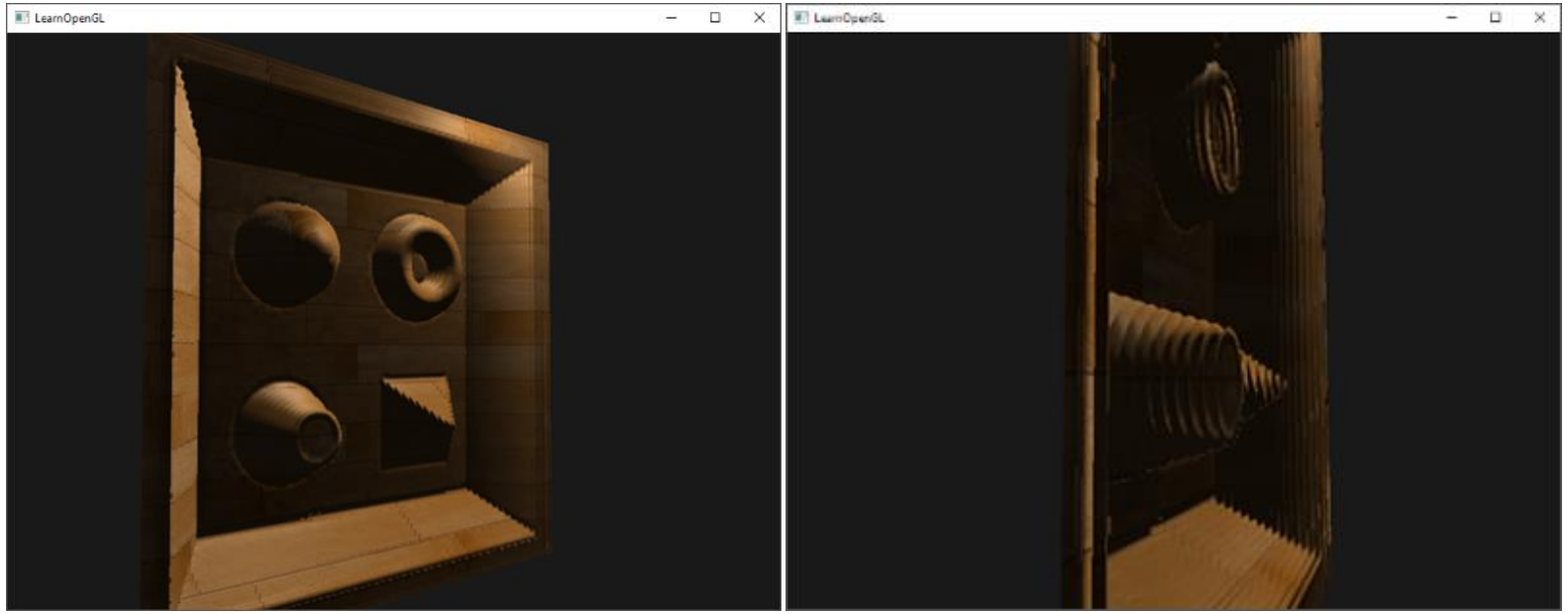
Steep Parallax Mapping

- 시점 벡터(V)와 표면의 각도에 따라 layer의 개수를 다르게 설정 가능
  - 평행에 가까워지면 많은 횟수로 탐색하여 품질을 높임
  - 수직에 가까워지면 적은 횟수로 탐색하여 성능을 높임

FS

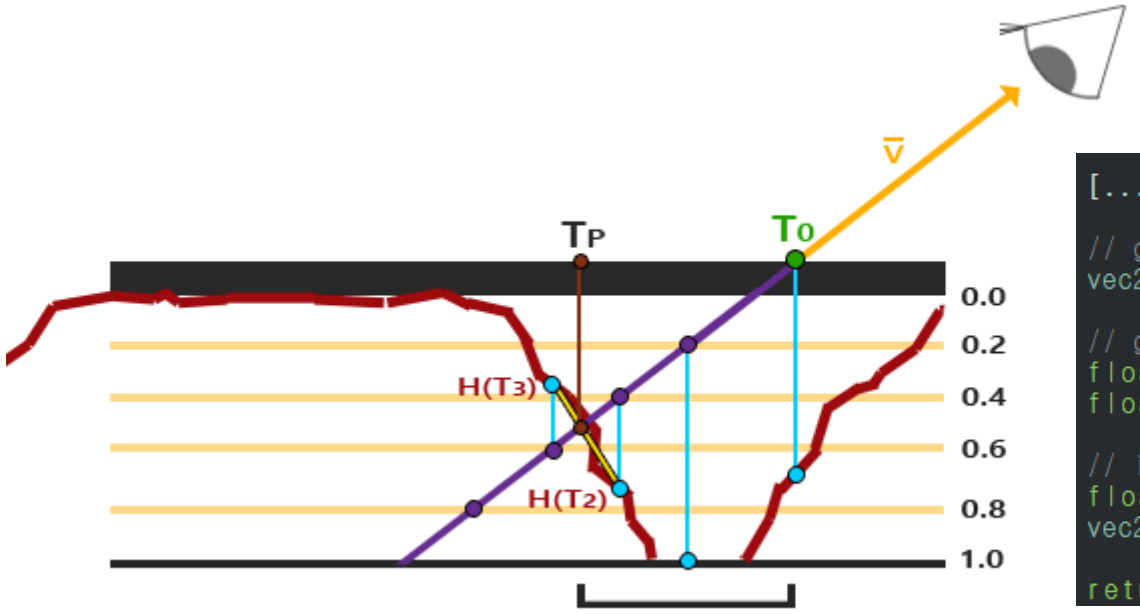
```
const float minLayers = 8.0;
const float maxLayers = 32.0;
float numLayers = mix(maxLayers, minLayers, max(dot(vec3(0.0, 0.0, 1.0), viewDir), 0.0));
```

- 결과
  - 각도에 따라 stepping artifacts가 보임
  - Linear search 결과 ray와 height field와의 교차를 놓치는(miss) 경우가 발생할 수 있기 때문



Parallax Occlusion Mapping

- 교차 전후의 depth layer 사이에서 선형 보간 (linear interpolation) 수행
- 보간만으로도 정확도를 높이는 데 큰 도움이 됨

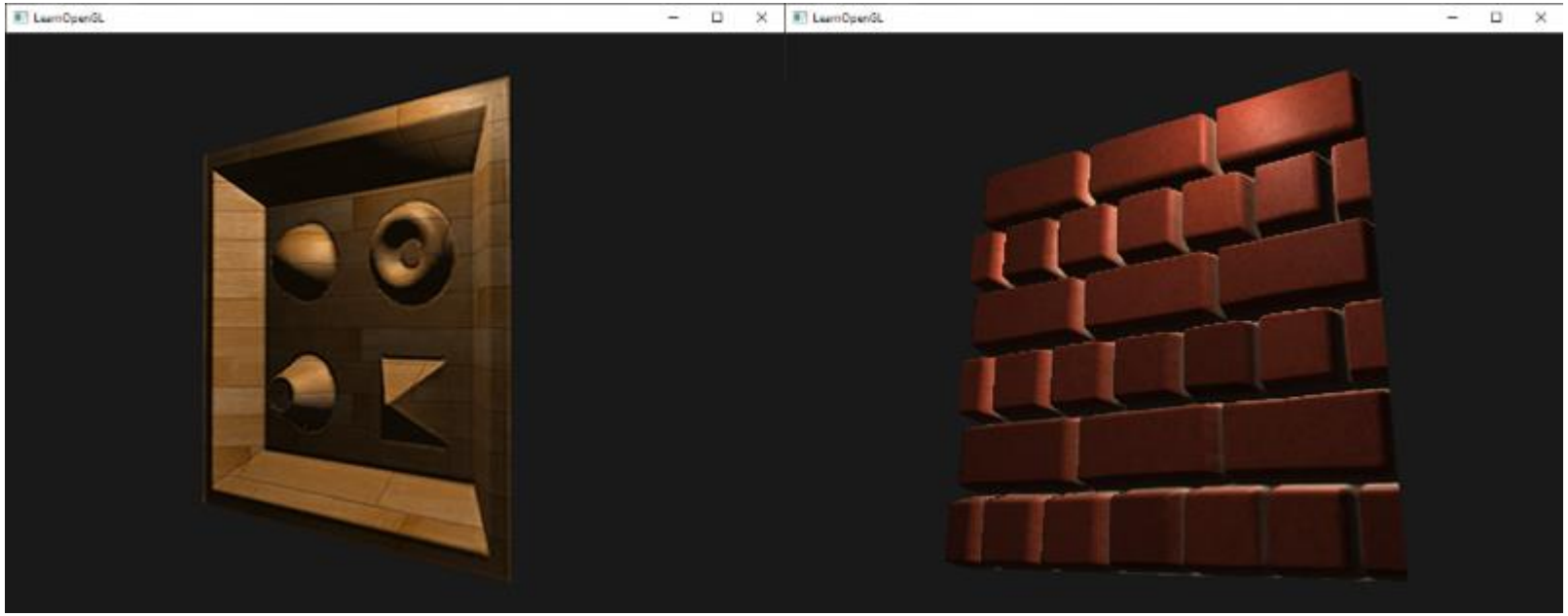


FS

```
[...] // steep parallax mapping code here

// get texture coordinates before collision (reverse operations)
vec2 prevTexCoords = currentTexCoords + deltaTexCoords;

// get depth after and before collision for linear interpolation
float afterDepth  = currentDepthMapValue - currentLayerDepth;
float beforeDepth = texture(depthMap, prevTexCoords).r - currentLayerDepth + layerDepth;

// interpolation of texture coordinates
float weight = afterDepth / (afterDepth - beforeDepth);
vec2 finalTexCoords = prevTexCoords * weight + currentTexCoords * (1.0 - weight);

return finalTexCoords;
```

Parallax Occlusion Mapping

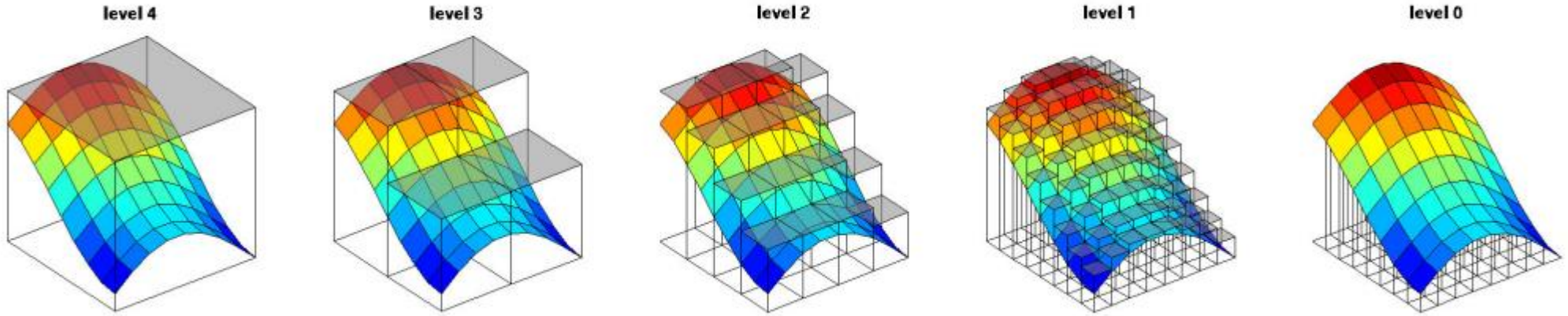
- 실행 결과
  - Steep parallax mapping에 비해 artifact가 많이 감소
  - 단, linear search 특성상 artifact-free 수준까지 도달하기 위해서는 높은 탐색 횟수가 필요할 수 있음
  - Linear search와 binary search를 함께 사용하기도 함
    Dynamic parallax occlusion mapping with approximate soft shadows (amd.com)



- 관련 기법 비교 데모 - Exploring bump mapping with WebGL (apoorvaj.io)

Pyramidal (Quadtree) Displacement Mapping

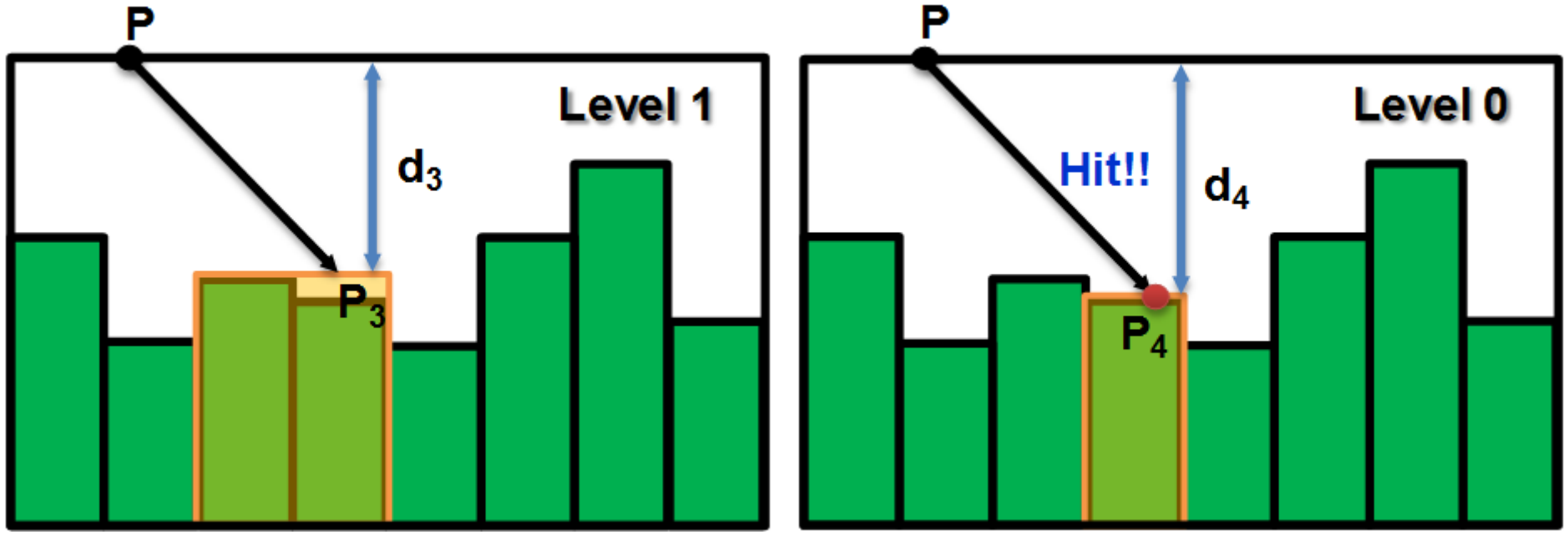
- Artifact-free rendering을 보장하면서도, 완전한 binary search를 통해 성능 하락도 최소화하는 방법
- 전처리를 통해, height (또는 depth) map을 mipmap (image pyramid)형태로 구성
  - 단, 네 pixel 중 가장 큰 값을 취하여 더 높은 level의 mipmap을 구성함으로써, conservative search 보장



  - Maximum Mipmaps for Fast, Accurate, and Scalable Dynamic Height Field Rendering (i3D 2016, inria.fr)

Pyramidal (Quadtree) Displacement Mapping

- 이후 실시간 렌더링시 quadtree 형태의 mipmap을 탐색하며 정확한 교차지점을 찾음



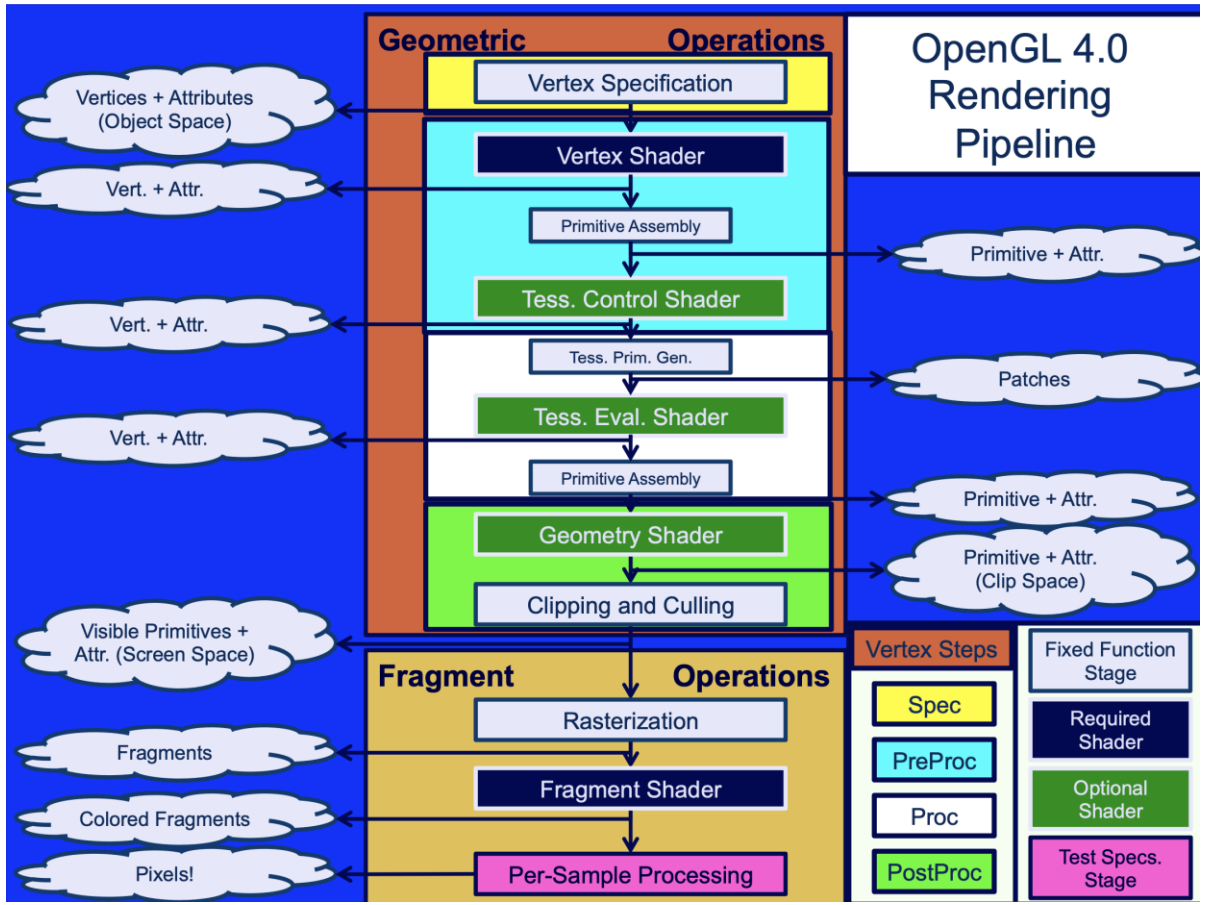
- 이 quadtree를 어떻게 탐색하느냐에 따라 높은 성능 향상을 얻을 수 있음
  - 자세한 내용은 아래 논문 참조
    Effective traversal algorithms and hardware architecture for pyramidal inverse displacement mapping (nahjaeho.github.io)

Displacement Mapping with Tessellation

- Inverse displacement mapping은 본질적으로 텍스처 좌표만 수정한다는 한계를 지님
  - 실루엣 처리 등에서 한계를 가질 수 밖에 없음

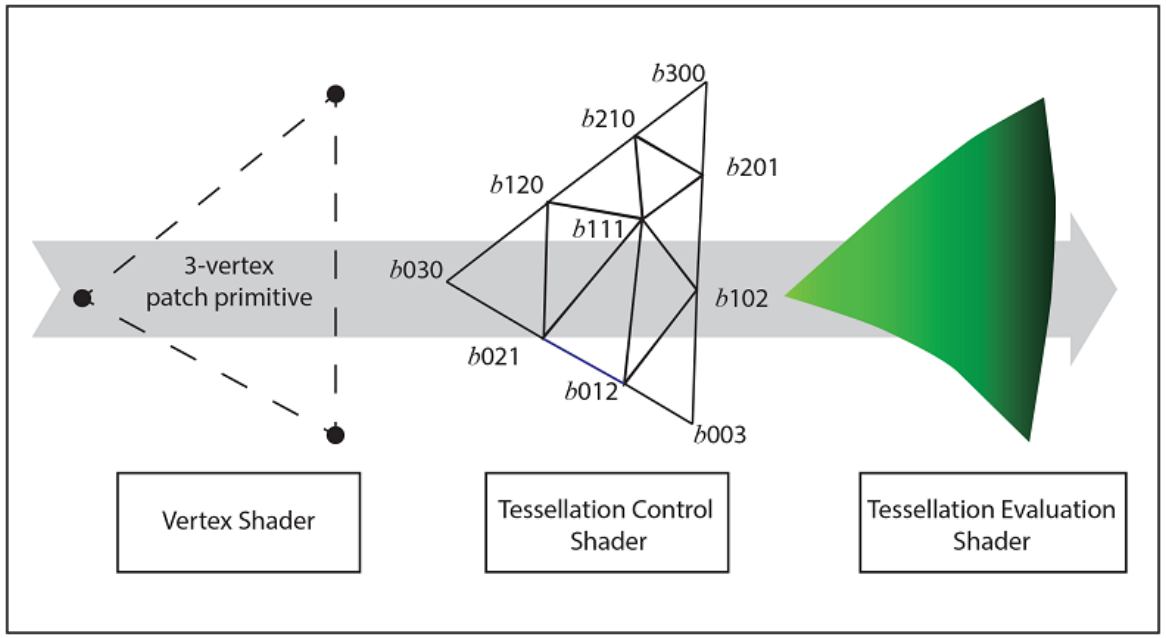
- Tessellation
  - Primitive를 잘게 나눠 새로운 primitive를 만드는 방법

- OpenGL 4.0+, OpenGL ES 3.2+에서는 tessellation shader 지원
  - Vertex의 위치를 직접 변경하는 displacement mapping 수행 가능



Displacement Mapping with Tessellation

- Tessellation control shader
  - Patch primitive를 어느 정도로 잘게 나눌 것인지 결정
- Tessellator
  - 실제로 patch를 잘게 쪼개는 fixed pipeline 단계
- Tessellation evaluator shader
  - 쪼개진 patch에서의 vertex 위치를 계산
  - Height map을 이 단계에서 읽어 들이면 이에 따라 vertex의 위치를 변경 가능



PD-13 - GPU Programming for GIS Applications | GIS&T Body of Knowledge (ucgis.org)

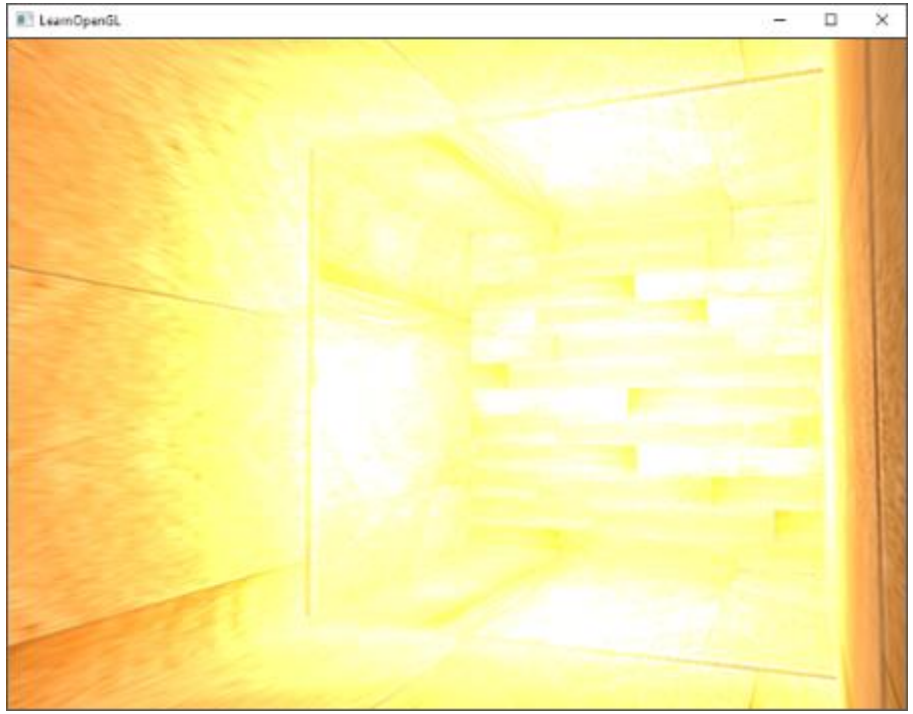
- 데모: Unigine HAVEN tessellation on-off wire frame on-off – YouTube
- 예제 코드: LearnOpenGL - Tessellation

ⓒ nvidia

HDR

HDR (High Dynamic Range)

- 기본적으로 밝기와 색상값은 프레임버퍼에 저장시 [0.0, 1.0]의 범위를 가짐
  - 만약 여러 밝은 광원을 사용해 특별히 밝은 곳을 본다면? → 이 부분은 1.0으로 clamping됨



- 위와 같은 문제를 해결하는 방법 – HDR
  - 색상 값이 일시적으로 1.0을 초과하도록 허용하고, 최종 단계에서 이를 [0.0, 1.0] 사이로 매핑

HDR (High Dynamic Range)

- 사진에서의 HDR
  - 노출이 적으면 밝은 부분의 디테일이, 노출이 많으면 어두운 부분의 디테일이 잘 나타남
  - 스마트폰 카메라에서는 노출이 다른 여러 장의 이미지를 합성하여 밝은 곳과 어두운 곳의 디테일을 모두 살림

- Real-time rendering에서의 HDR
  - [0.0, 1.0] 사이의 LDR (low dynamic range)에 비해 훨씬 더 넓은 범위의 색상 값을 사용
  - 어두운 부분과 밝은 부분을 광범위하게 수집
  - 모든 HDR값을 다시 LDR로 변환 (tone mapping)
    HDR display을 출력으로 사용할 시에는, 이 tone mapping 방법도 달라짐

Floating-Point Framebuffers

- 프레임버퍼의 컬러 내부 형식을, HDR을 표현 가능한 16/32비트 floating-point로 지정 가능
  - GL_RGB16F, GL_RGBA16F, GL_RGB32F, GL_RGBA32F

CPU

```
glBindTexture(GL_TEXTURE_2D, colorBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
```

  - 기존 8비트 컬러 버퍼 대비 2~4배의 메모리가 필요
    (32비트 대신 16비트로도 충분한 정밀도를 얻을 수 있다면 이를 사용)

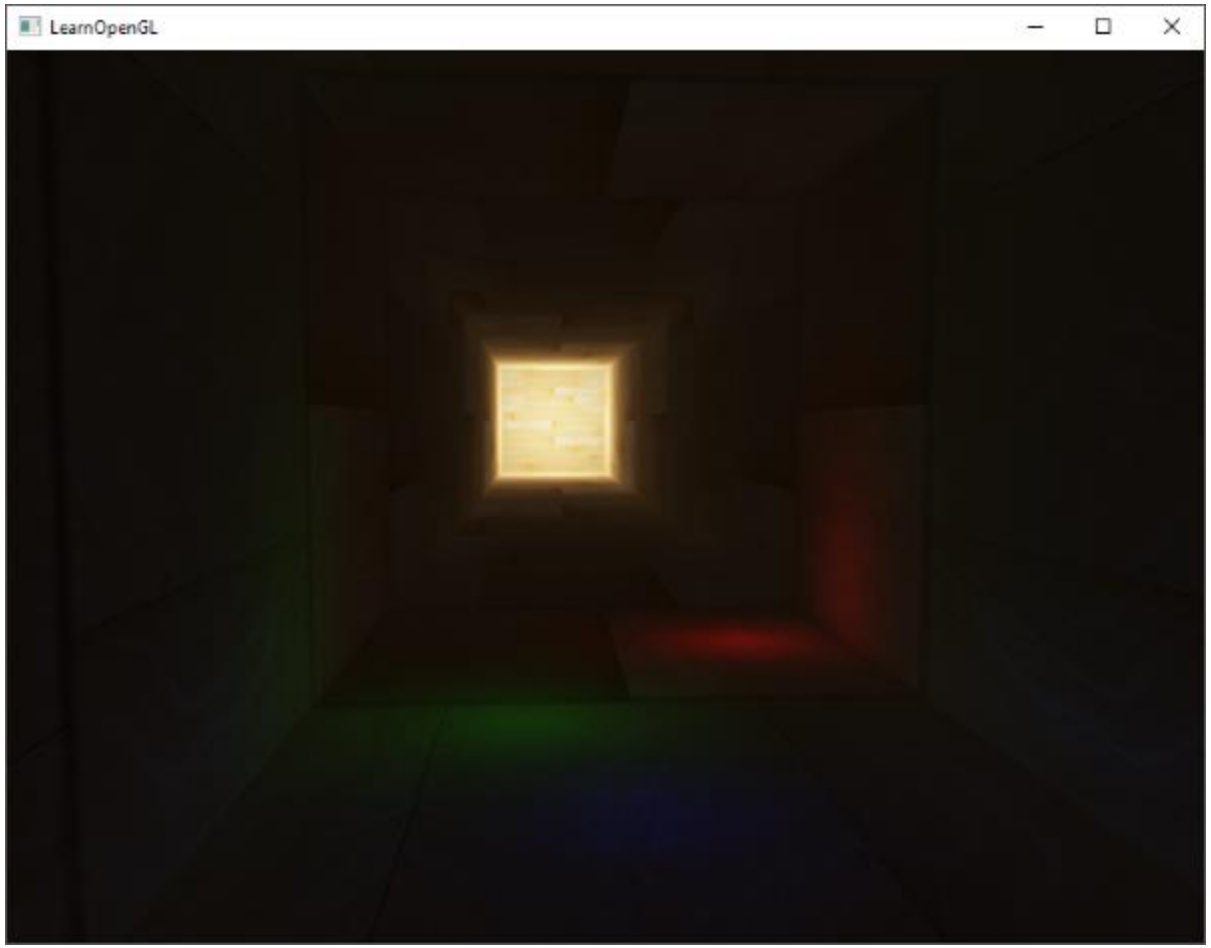
- 이제 1.0을 넘는 밝기도 프레임버퍼에 저장 가능

Tone Mapping

- Reinhard tone mapping

FS

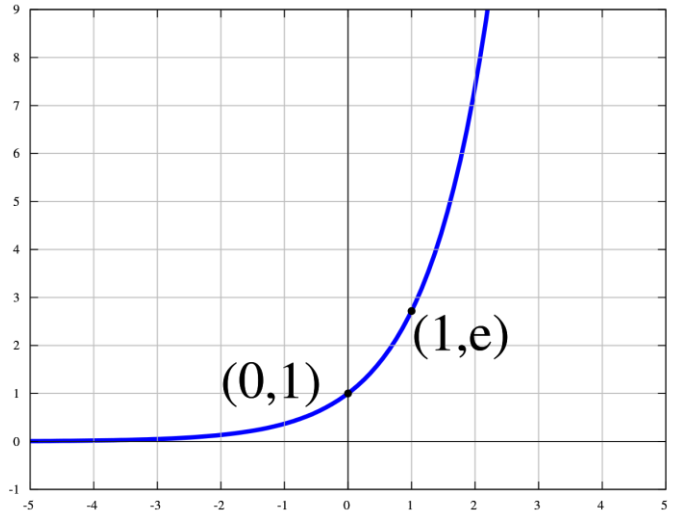
```
void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;

    // reinhard tone mapping
    vec3 mapped = hdrColor / (hdrColor + vec3(1.0));
    // gamma correction
    mapped = pow(mapped, vec3(1.0 / gamma));

    FragColor = vec4(mapped, 1.0);
}
```



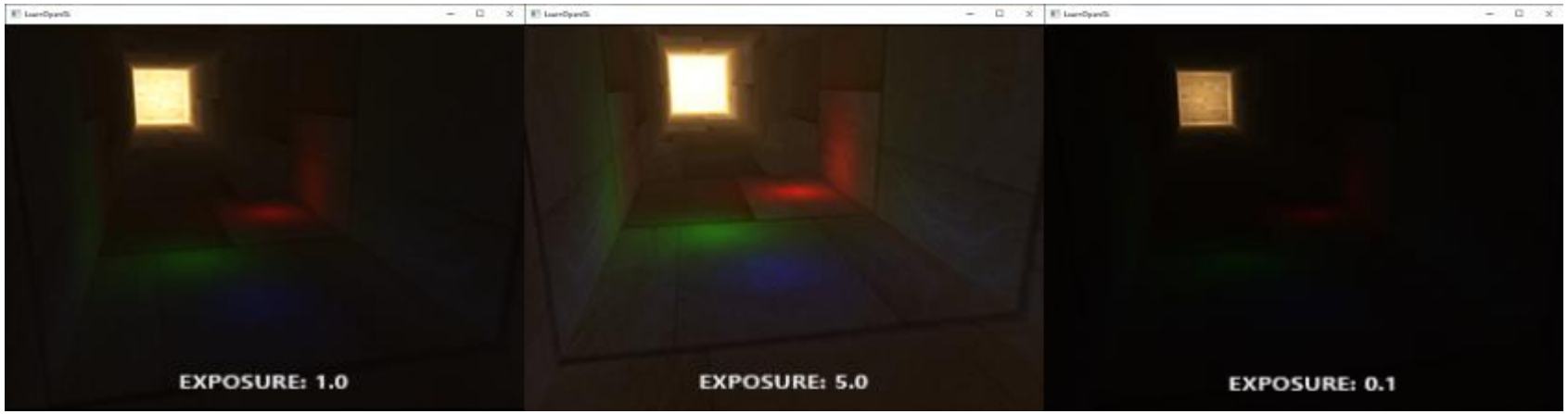
Tone Mapping

- Exposure tone mapping

```
uniform float exposure;                                    FS

void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;

    // exposure tone mapping
    vec3 mapped = vec3(1.0) - exp(-hdrColor * exposure);
    // gamma correction
    mapped = pow(mapped, vec3(1.0 / gamma));

    FragColor = vec4(mapped, 1.0);
}
```



[지수 함수 - 위키백과, 우리 모두의 백과사전 (wikipedia.org)]



ⓒ nvidia

Bloom

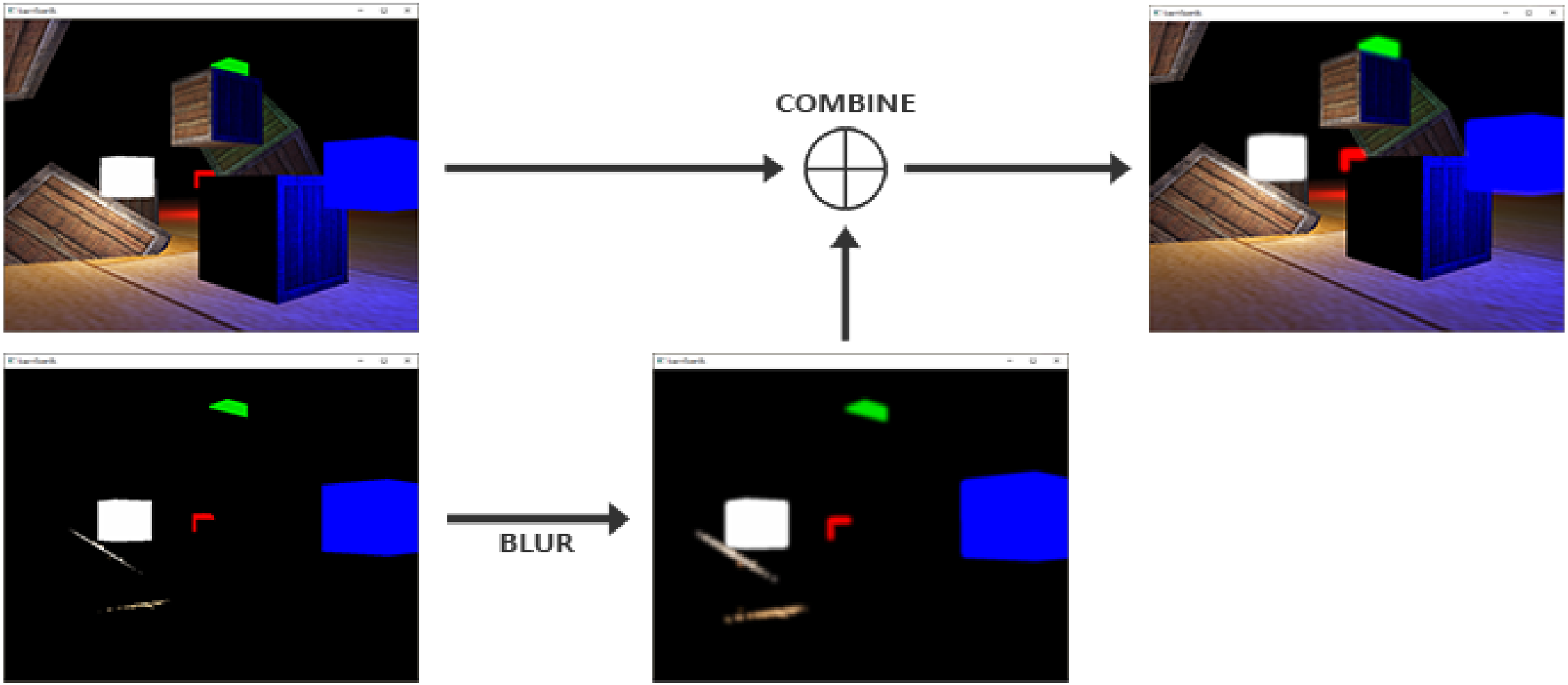
Bloom

- Light bleeding (또는 glow effect)
  - 밝은 광원 주변으로 빛이 흘러나오는 현상
  - Bloom이라고 불리우는 post-processing으로 구현 가능
  - 해당 물체가 정말 밝게 빛나고 있다는 시각적 신호를 줄 수 있음

- HDR 렌더링과 조합시 유용함
  - 눈에 띄게 밝은 영역을 더 쉽게 강조할 수 있기 때문
  - 단, bloom 효과는 8bit 프레임버퍼에서도 구현 가능

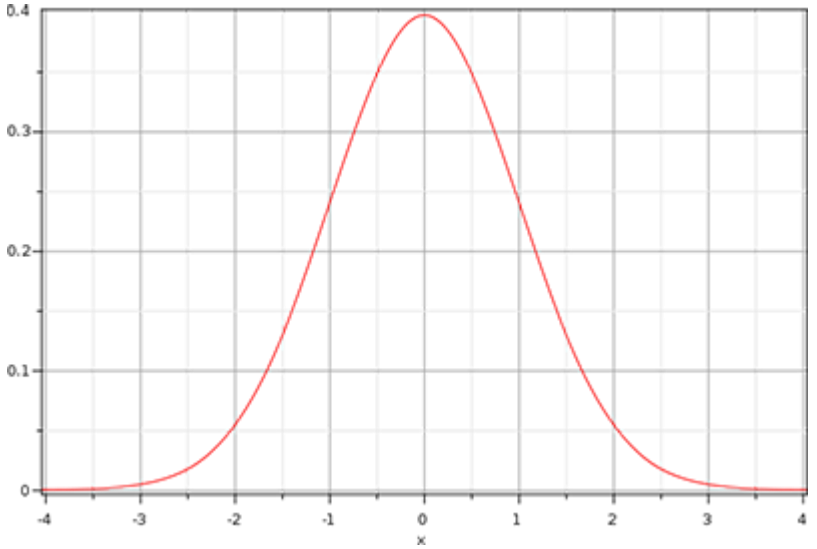
Bloom

- HDR 컬러 버퍼를 사용할 때의 bloom 효과 적용 예시



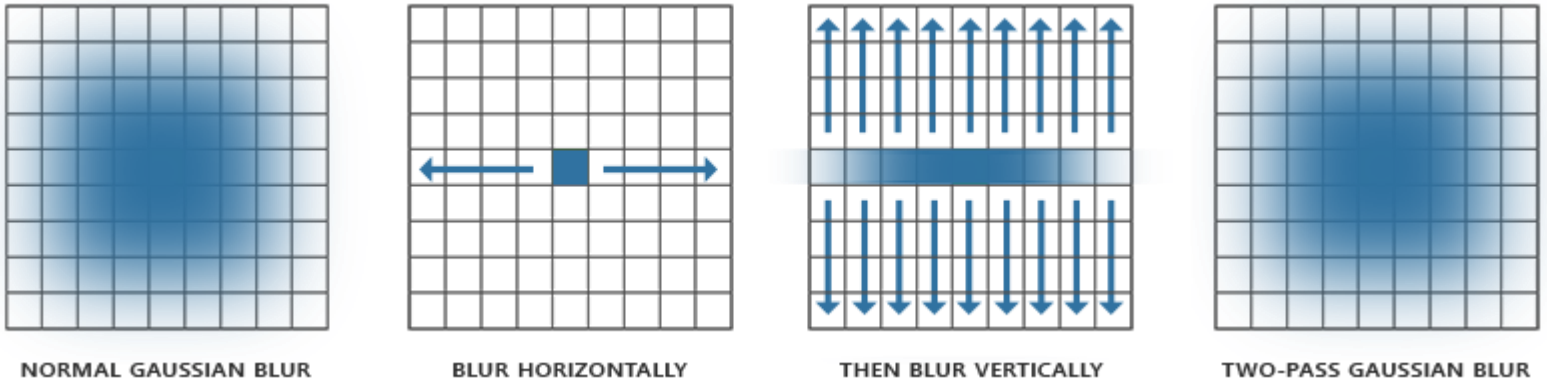
Gaussian Blur

- 중심으로부터의 거리에 따라 값이 달라지는 종 모양의 Gaussian curve를 사용하면, bloom에 필요한 blur 효과를 얻을 수 있음



- 2차원 screen space에서 가우시안 블러를 사용할 경우, blur 커널의 크기에 비례하여 texture sampling 횟수가 늘어남
  - 32x32 커널 → fragment당 1024회 sampling

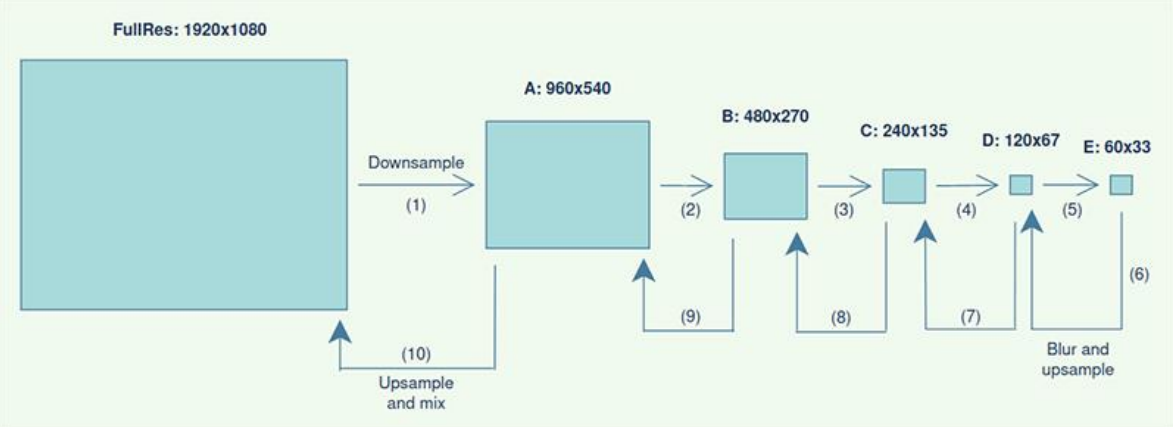
- Two-pass Gaussian Blur를 사용하면 연산량을 줄일 수 있음
  - w*h가 아닌 w+h가 됨
  - 다만 첫번째 pass의 결과를 framebuffer에 저장해야 함



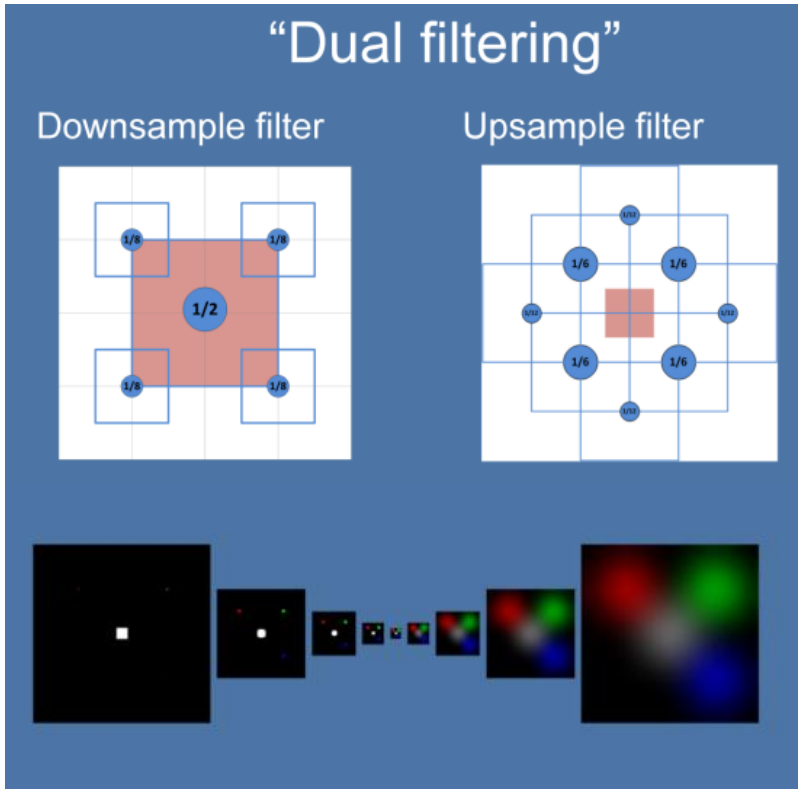
Multi-Resolution Approach

- 2-pass Gaussian Blur도 여전히 저사양 (모바일 등) 기기에서는 연산량이 부담이 됨

- 다단계 downsampling & upsampling을 이용하는 방법



LearnOpenGL - Phys. Based Bloom



Bandwidth-Efficient Rendering (ARM)

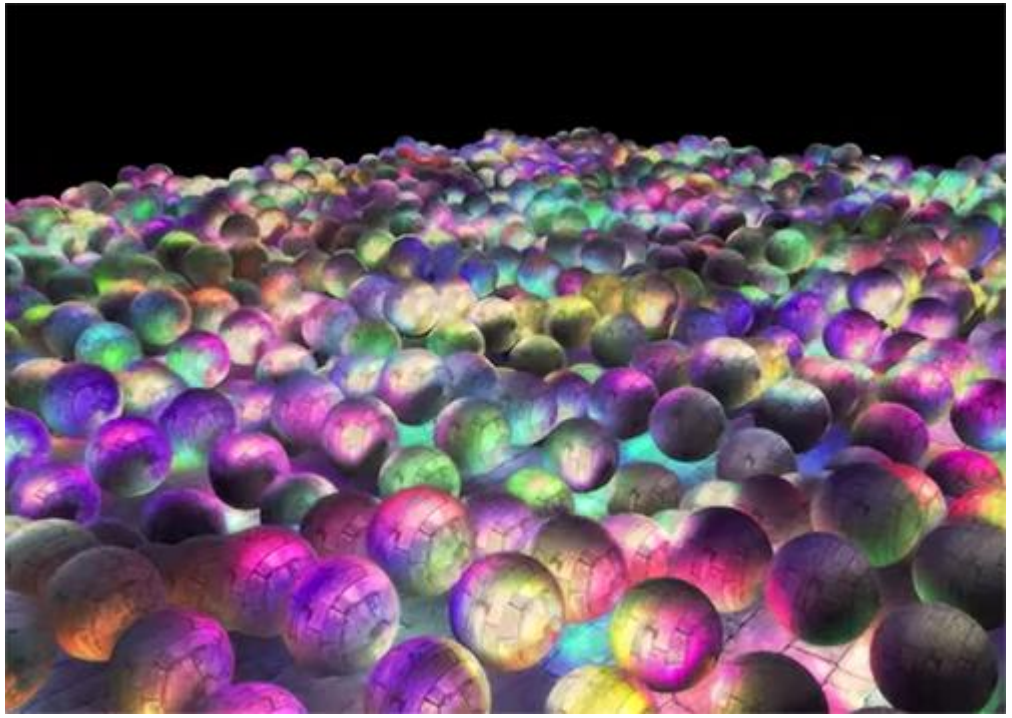
ⓒ nvidia

Deferred Shading

Forward vs Deferred Shading

- Forward shading (또는 forward rendering)
  - 본 수업에서 지금까지 lighting을 하기 위해 수행해 온 방법
  - Scene 내의 물체의 개별 fragment 단위로 lighting을 계산
  - Depth complexity가 높은 scene(=overdraw가 많음)에서 fragment shading 낭비 가능

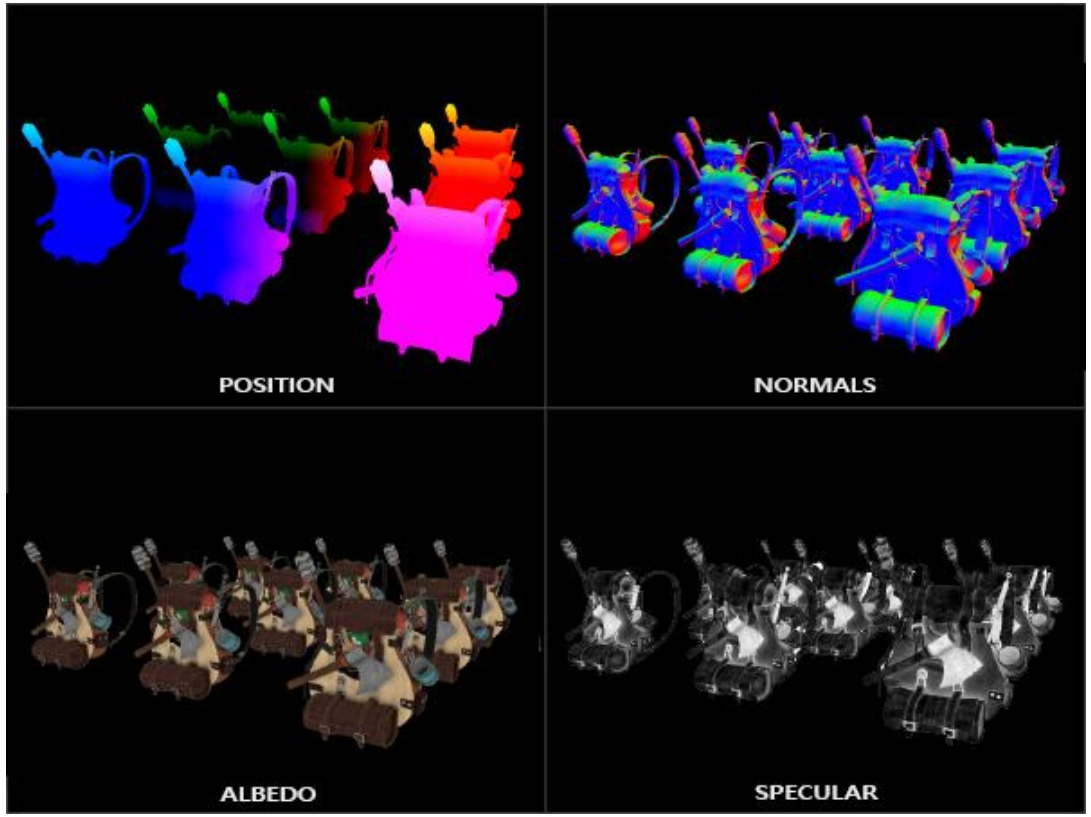
- Deferred shading (또는 deferred rendering)
  - 무거운 렌더링 요소 계산(예: lighting)을 나중 단계로 연기(defer)
  - Depth complexity가 높을 때에도, 수백~수천개의 조명을 효과적으로 렌더링 가능



1847개 point light를 사용한 scene

Deferred Shading

- 1단계 - geometry pass
  - Scene을 한번 렌더링하여, 지오메트리와 관련된 정보를 텍스쳐 형태로 메모리에 저장 (G-buffer)
  - Multiple render targets(MRTs)를 사용하면, single pass로 여러가지 정보를 한꺼번에 프레임버퍼에 저장 가능
  - 위치, 컬러, 노멀, specular intensity 등 저장.
  - 모든 lighting 관련 변수들은 같은 좌표계 상에 있어야 함



Deferred Shading

- 1단계 - geometry pass (cont.)

CPU

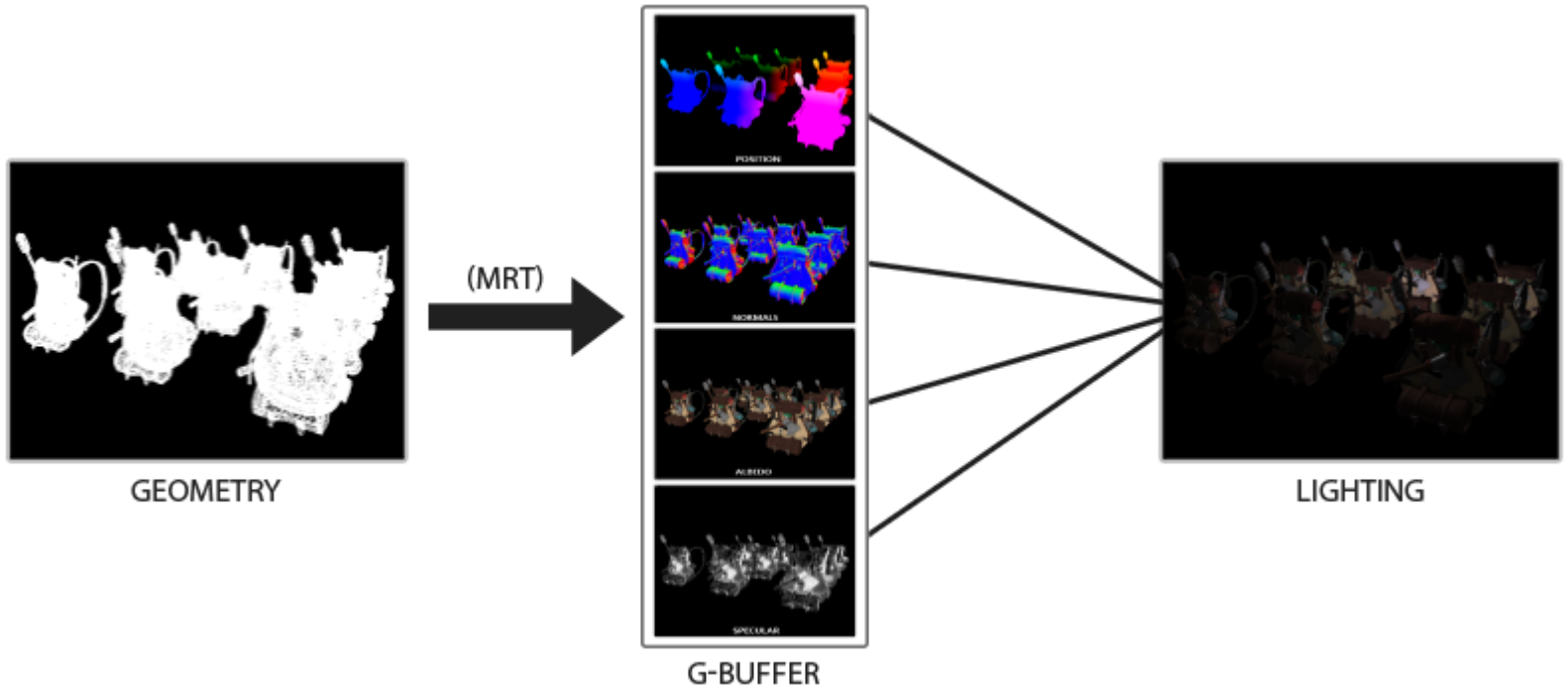
```
unsigned int gBuffer;
glGenFramebuffers(1, &gBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
unsigned int gPosition, gNormal, gColorSpec;

// - position color buffer
glGenTextures(1, &gPosition);
glBindTexture(GL_TEXTURE_2D, gPosition);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, gPosition, 0);

// - normal color buffer
glGenTextures(1, &gNormal);
glBindTexture(GL_TEXTURE_2D, gNormal);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, gNormal, 0);

// - color + specular color buffer
glGenTextures(1, &gAlbedoSpec);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, gAlbedoSpec, 0);

// - tell OpenGL which color attachments we'll use (of this framebuffer) for rendering
unsigned int attachments[3] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2 };
glDrawBuffers(3, attachments);

// then also add render buffer object as depth buffer and check for completeness.
[...]
```

FS

```
#version 330 core
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;

uniform sampler2D texture_diffuse1;
uniform sampler2D texture_specular1;

void main()
{
    // store the fragment position vector in the first gbuffer texture
    gPosition = FragPos;
    // also store the per-fragment normals into the gbuffer
    gNormal = normalize(Normal);
    // and the diffuse per-fragment color
    gAlbedoSpec.rgb = texture(texture_diffuse1, TexCoords).rgb;
    // store specular intensity in gAlbedoSpec's alpha component
    gAlbedoSpec.a = texture(texture_specular1, TexCoords).r;
}
```

Deferred Shading

- 2단계 - lighting pass
  - Post-processing 단계와 유사하게, 화면을 꽉 채우는 render quad를 렌더링
  - G-buffer로부터 lighting에 필요한 정보를 얻어 lighting을 수행



Deferred Shading

- 2단계 - lighting pass (cont.)

CPU

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, gPosition);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, gNormal);
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
// also send light relevant uniforms
shaderLightingPass.use();
SendAllLightUniformsToShader(shaderLightingPass);
shaderLightingPass.setVec3("viewPos", camera.Position);
RenderQuad();
```

FS

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;

struct Light {
    vec3 Position;
    vec3 Color;
};
const int NR_LIGHTS = 32;
uniform Light lights[NR_LIGHTS];
uniform vec3 viewPos;

void main()
{
    // retrieve data from G-buffer
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
    vec3 Albedo = texture(gAlbedoSpec, TexCoords).rgb;
    float Specular = texture(gAlbedoSpec, TexCoords).a;

    // then calculate lighting as usual
    vec3 lighting = Albedo * 0.1; // hard-coded ambient component
    vec3 viewDir = normalize(viewPos - FragPos);
    for(int i = 0; i < NR_LIGHTS; ++i)
    {
        // diffuse
        vec3 lightDir = normalize(lights[i].Position - FragPos);
        vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Albedo * lights[i].Color;
        lighting += diffuse;
    }

    FragColor = vec4(lighting, 1.0);
}
```

Deferred Shading

- Deferred shading의 장점
  - 실제 화면에 보여지는 fragment에 대해서만 lighting이 수행

- Deferred shading의 단점
  - G-Buffer를 저장하기 위한 메모리 공간 필요
  - Blending이 어려움 (가장 앞에 있는 fragment만 저장되어 있기 때문)
  - MSAA 처리가 어려움 (render quad를 사용하기 때문)
  - Lighting pass에서 동일한 lighting 알고리즘 사용
    → 더 복잡한 lighting을 위해서는 material 관련 data가 더 필요

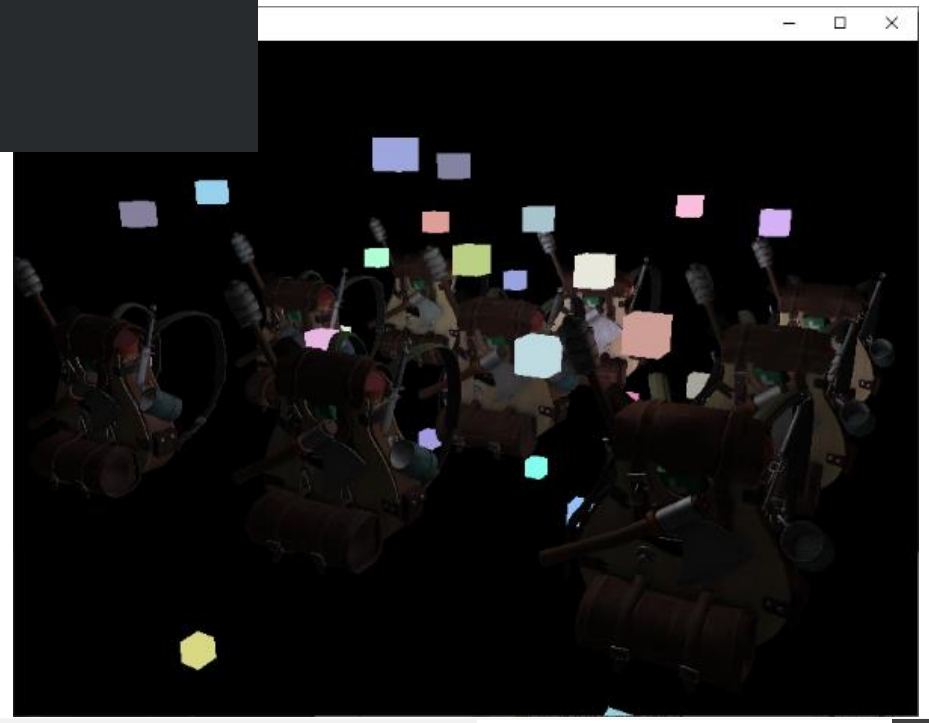
- Deferred shading과 forward shading의 결합은 위 단점 중 몇가지를 완화 가능

Combining Deferred Rendering with Forward Rendering

- Deferred rendering 이후 forward rendering의 결과를 결합하고자 할 때
  - G-buffer에 저장된 depth값이 forward rendering시에도 적용될 필요가 있음
  - 이는 glBlitFramebuffer()를 통해 가능 (multisampled FBO를 사용하면 MSAA도 가능)

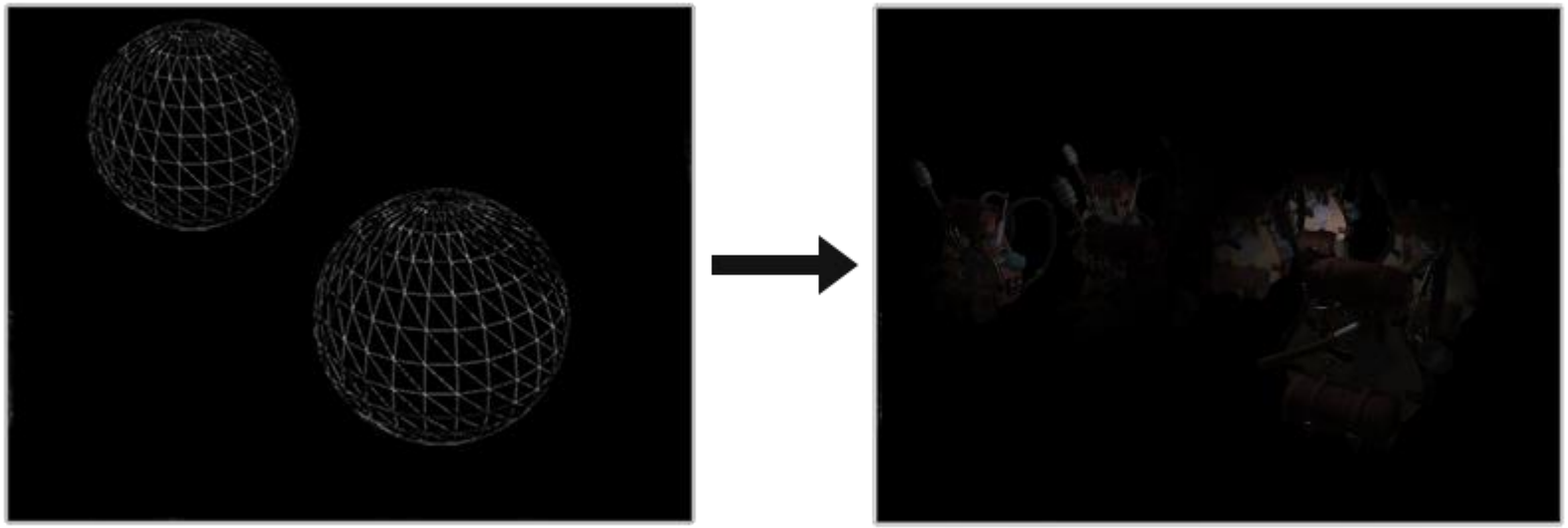
CPU

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, gBuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0); // write to default framebuffer
glBlitFramebuffer(
  0, 0, SCR_WIDTH, SCR_HEIGHT, 0, 0, SCR_WIDTH, SCR_HEIGHT, GL_DEPTH_BUFFER_BIT, GL_NEAREST
);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// now render light cubes as before
[...]
```



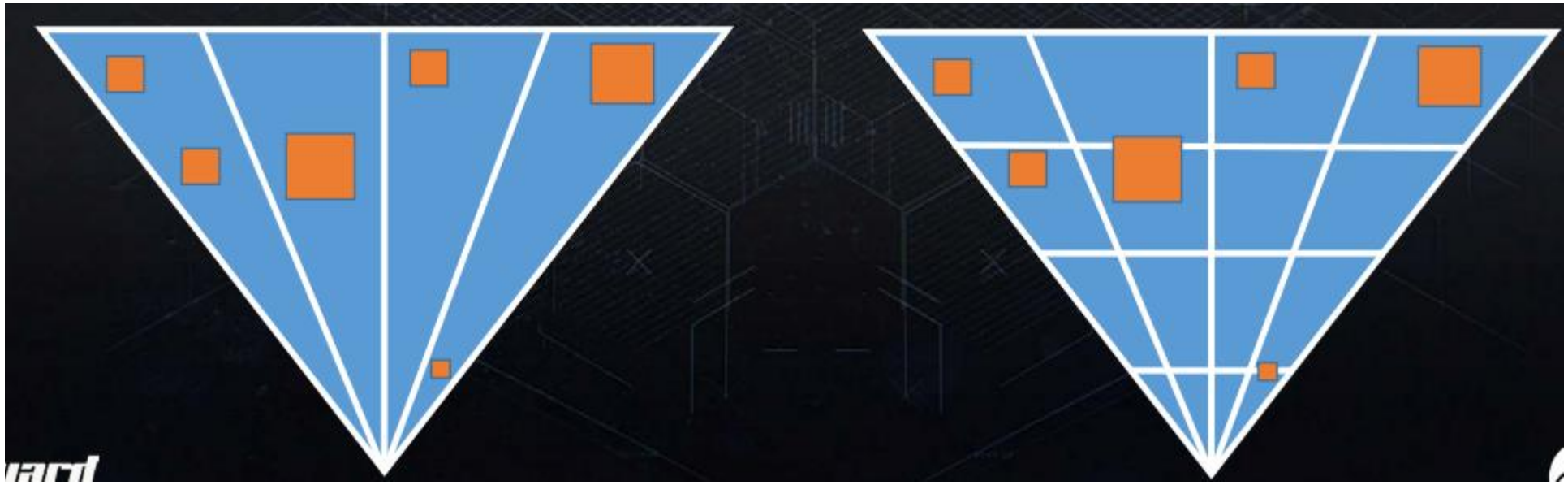
A Larger Number of Lights

- Light volume의 사용
  - Back-face culling을 활성화한 후, light volume을 렌더링하면서 deferred shading 적용
  - 실제 광원이 영향을 끼치는 부분에서만 해당 lighting 계산이 수행
  - 연산량 감소: nr_object * nr_lights → nr_objects + nr_lights



A Larger Number of Lights

- Tiled shading
  - 화면을 2D 타일로 나누고,
    이 타일에 영향을 미치는 광원만 처리
- Clustered shading
  - View frustum을 잘게 나누고,
    각 cell에 영향을 미치는 광원만 처리



Rendering of COD:IW (realtimerendering.com)

- 데모: Clustered Shading - YouTube

ⓒ nvidia

Ambient Occlusion

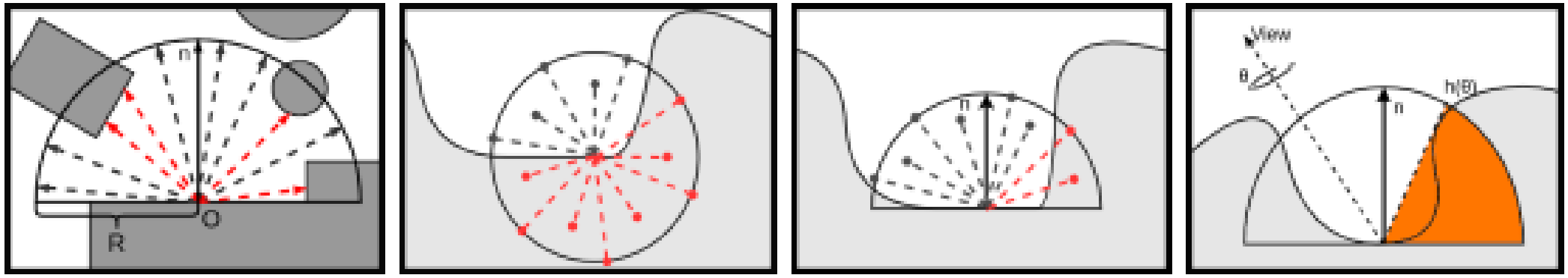
Ambient Occlusion

- Ambient occlusion (AO)
  - Ambient lighting을 좀 더 현실에 맞게 근사하는 방법
  - 빛은 여러 방향으로 산란되기 때문에, 간접적으로 빛을 받는 영역도 서로 다른 강도(intensity)를 가져야 함
  - AO는 표면이 공간에 노출되어 있는 정도로 간접광(indirect light)이 들어오는 정도를 다르게 계산

- 주름, 구멍, 서로 맞닿아 있는 표면 등을 어둡게 처리하여, 깊이감을 부여 가능

Ambient Occlusion

- Ray-traced AO (RTAO)
  - Shading point에서 반구로 ray들을 쏜 다음, 차폐되는(occluded) ray의 비율 계산
- Screen-space ambient occlusion (SSAO) by Crytek (2007)
  - 2D screen-space 상에서, 인접한 픽셀의 깊이 버퍼의 값을 이용하여 AO를 근사하는 방법
- Horizon-based ambient occlusion (HBAO) by NVIDIA (2008)
  - Horizon mapping을 이용하여, horizon angle의 각을 기준으로 차폐 정도 계산

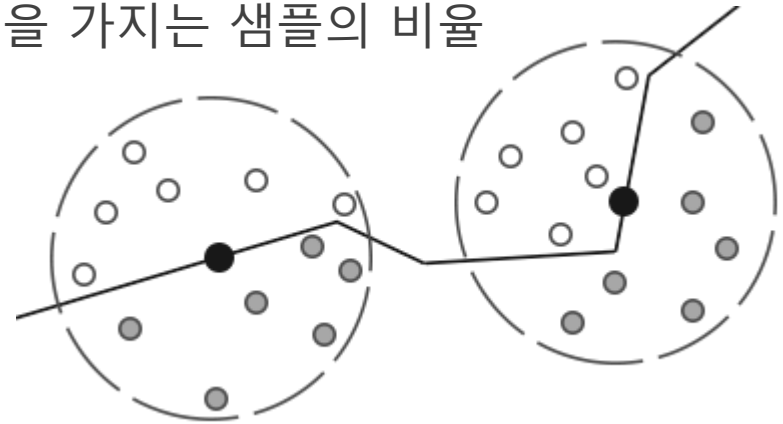


(a) Object space AO (b) Crytek SSAO (c) SSAO+ (d) HBAO

Pyramid HBAO --- a Scalable Horizon-based Ambient Occlusion Method (ceur-ws.org)

SSAO

- Occlusion factor 계산
  - 구 형태의 sample kernel 안에서 현재 fragment보다 높은 깊이값을 가지는 샘플의 비율

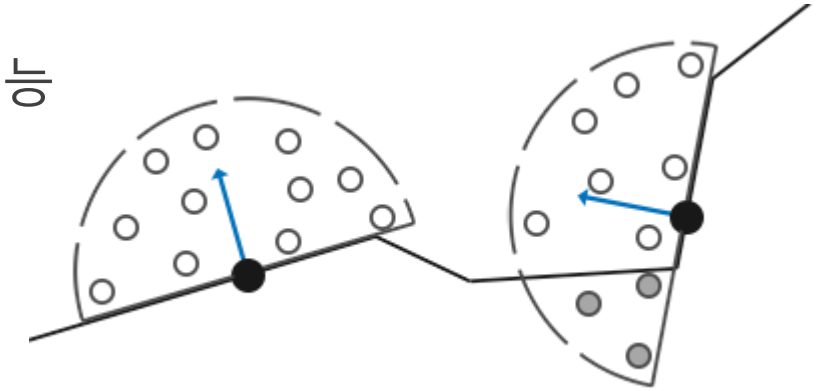


- 낮은 sample수로 인한 banding 문제 보완
  - 각 fragment별로 random하게 kernel 회전 후 blur

SSAO

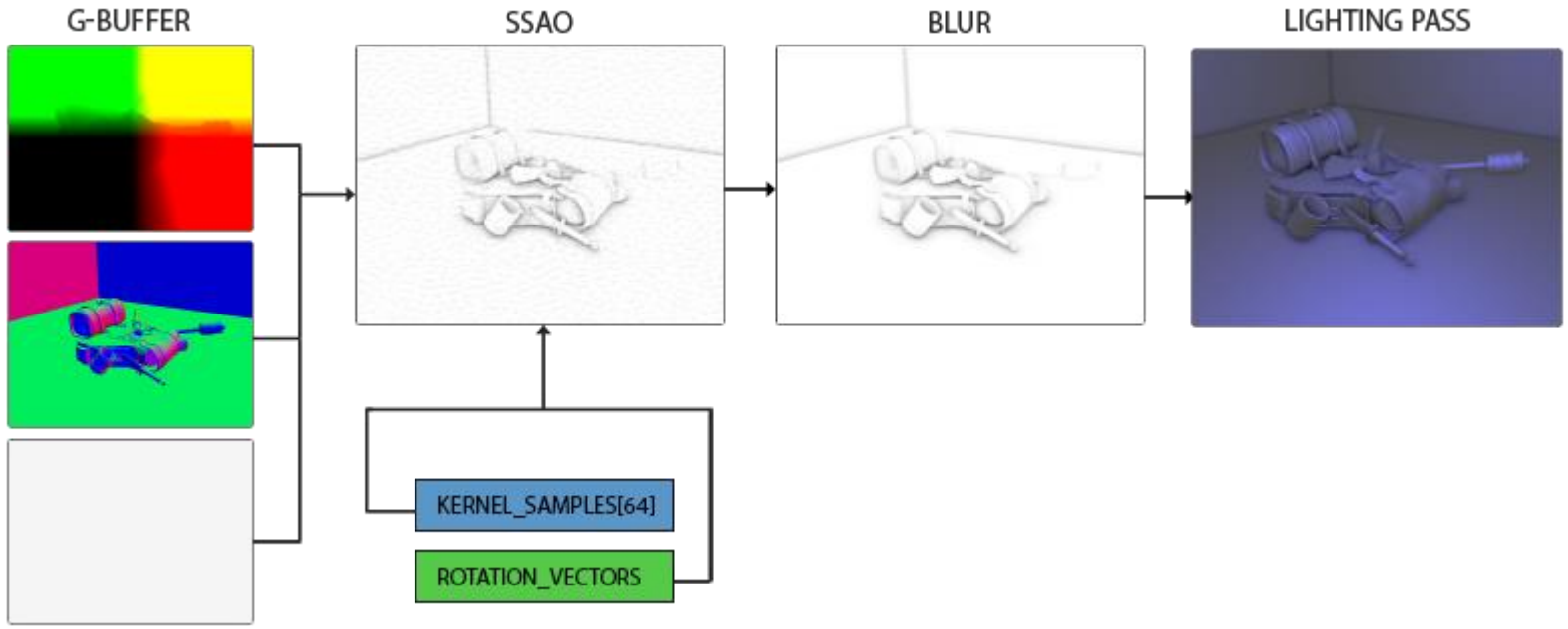
- SSAO의 초기 버전은 전체적으로 이미지가 회색톤으로 나오는 문제 발생
  - 빛을 완전히 받는 바닥마저도...

- 구 대신 반구 형태의 커널을 취하면 위 문제를 해결 가능
  - 노멀 기준으로 반구를 선택
  - Fragment 아래 있는 지오메트리는 AO 계산시 무시
  - Blizzard에서 스타크래프트2에 사용 (SSAO+)



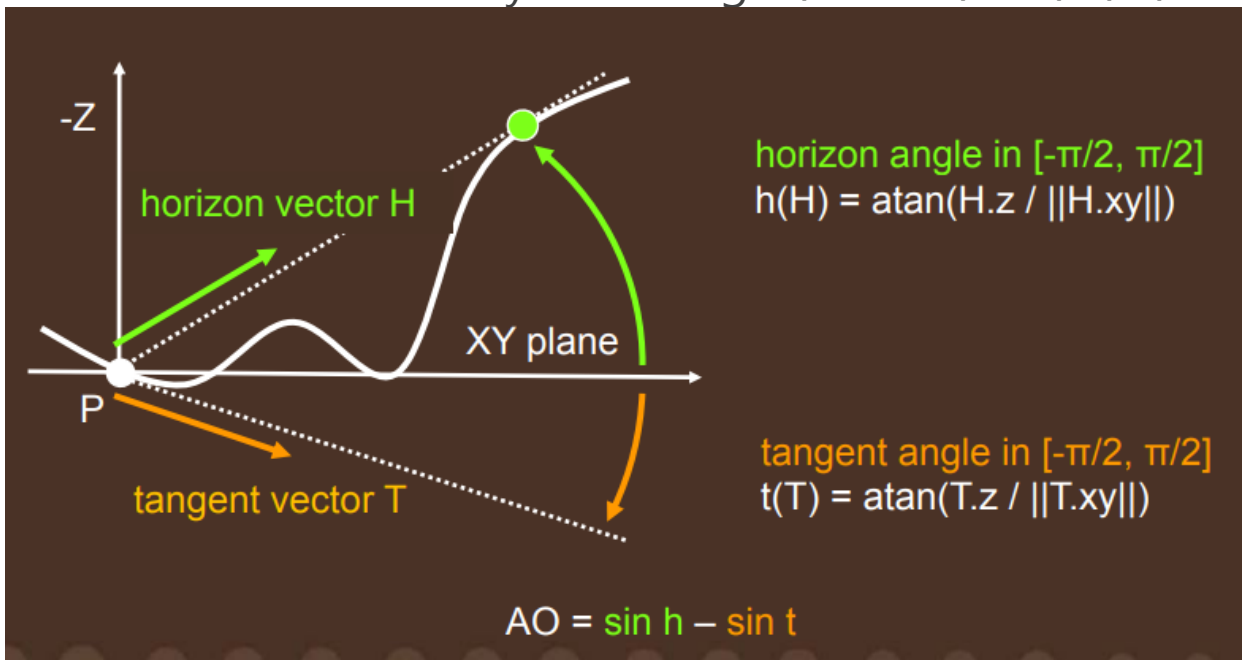
Sample Buffers for SSAO

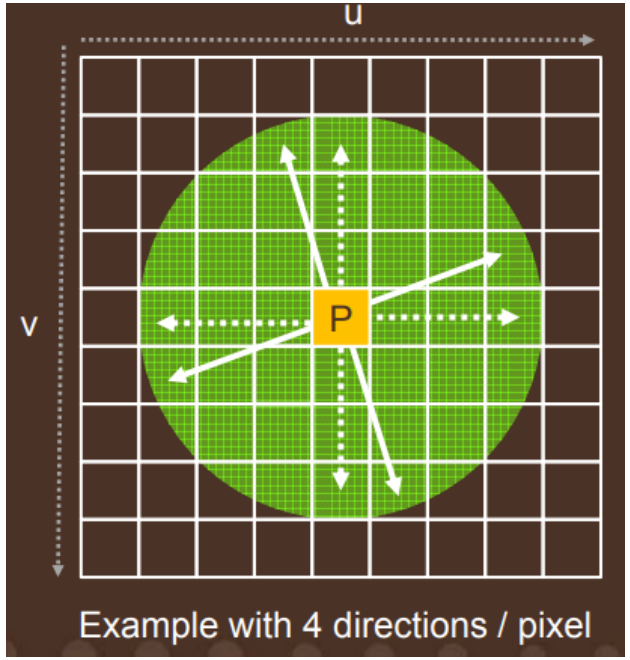
- SSAO 계산을 위해서는 아래 데이터 필요
  - A per-fragment position vector, normal vector & albedo color
  - A sample kernel
  - A per-fragment random rotation vector used to rotate the sample kernel
- Deferred shading 사용시, G-buffer에 저장된 데이터를 그대로 사용 가능



HBAO

- SSAO는 2D 깊이 버퍼만 사용하기 때문에 정확한 AO 계산에 어려움이 있음
- HBAO는 깊이 버퍼로 height field를 재구성한 후,
  horizon angle과 tangent angle을 구해 이 각도에 따라 차폐 정도를 계산
  - Horizon vector는 ray marching 기법을 사용하여 구함





Microsoft PowerPoint - HBAO_SIG08.pptx [Read-Only] (nvidia.com)

- HBAO+는 이후 나온 타 논문들의 기법을 적용하여 화질 및 성능을 개선
  - GitHub - NVIDIAGameWorks/HBAOPlus: HBAO+

Ray-traced Ambient Occlusion

- Denoising 알고리즘의 발전으로
  적은 수의 샘플(1~2 samples per pixel)로도
  RTAO를 노이즈 없이 구현 가능해짐
- 화질 측면에서 SSAO와는 확연한 차이

Ray Tracing in Games with NVIDIA RTX

ⓒ nvidia

마무리

마무리

- Advanced lighting의 마지막 시간으로, 아래와 같은 내용을 살펴보았습니다.
  - Displacement mapping
  - HDR
  - Bloom
  - Deferred shading
  - Ambient occlusion

- 더 이상의 실습은 없기 때문에, 아주 세세하게 코드 레벨로 개념을 살펴보기보다는 관련 내용을 얕고 넓게 소개하고자 하였습니다.
  - 각각의 기법에 관심이 있을 경우 링크를 눌러 논문 또는 발표자료 참조

- 이어서 교재에는 없는 GPU구조 & Ray-tracing의 내용으로 마지막 강의가 진행됩니다.