“본 강의 동영상 및 자료는 대한민국 저작권법을 준수합니다. 본 강의 동영상 및 자료는 상명대학교 재학생들의 수업목적으로 제작・배포되는 것이므로, 수업목적으로 내려받은 강의 동영상 및 자료는 수업목적 이외에 다른 용도로 사용할 수 없으며, 다른 장소 및 타인에게 복제, 전송하여 공유할 수 없습니다. 이를 위반해서 발생하는 모든 법적 책임은 행위 주체인 본인에게 있습니다.”

Advanced Lighting (2)

GPU Programming

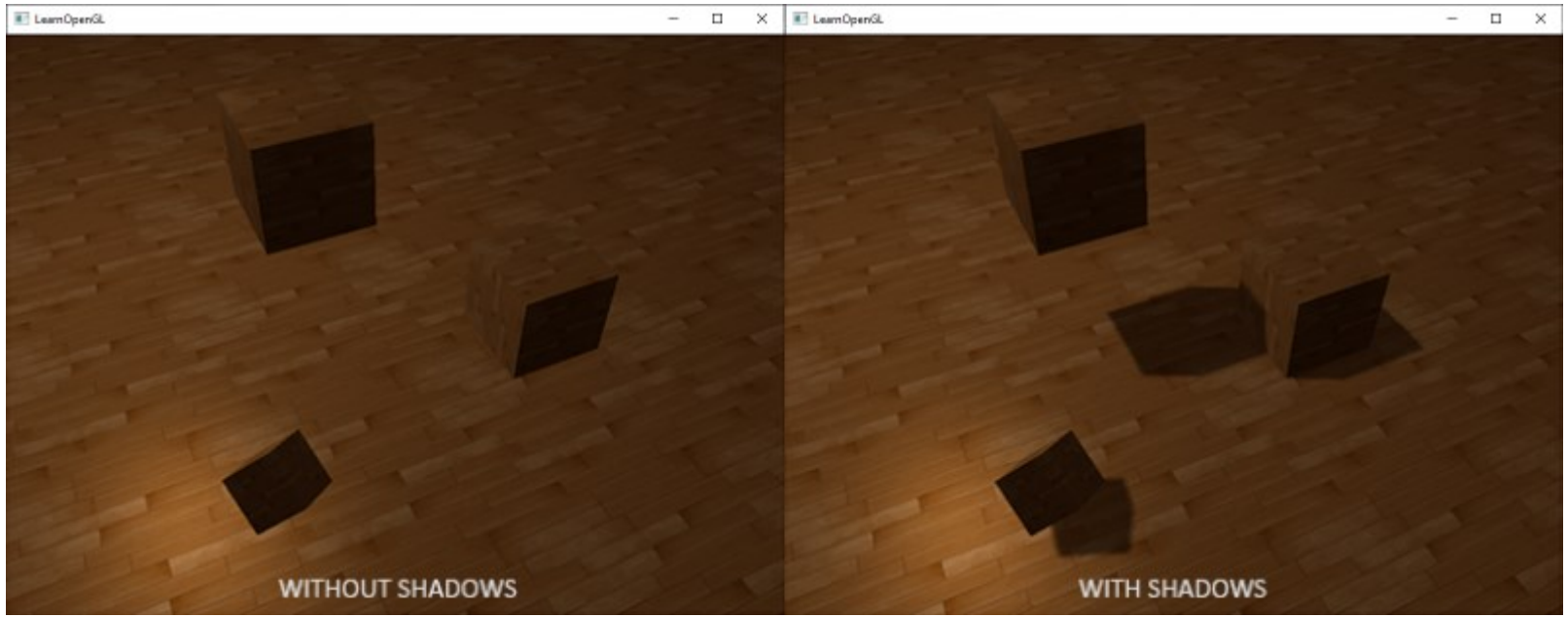
2022학년도
2학기

ⓒ nvidia

Shadow Mapping

Shadow Mapping

- 그림자(shadow)
  - 빛이 어떠한 물체에 도달하기 전에, 중간의 다른 물체에 의한 가려짐(occlusion)으로 인해 생기는 현상
  - 밝은 장면의 사실감을 증대하고, 물체 간의 공간적 관계를 더 쉽게 관찰할 수 있게 하며, 사물에 깊이감을 줌

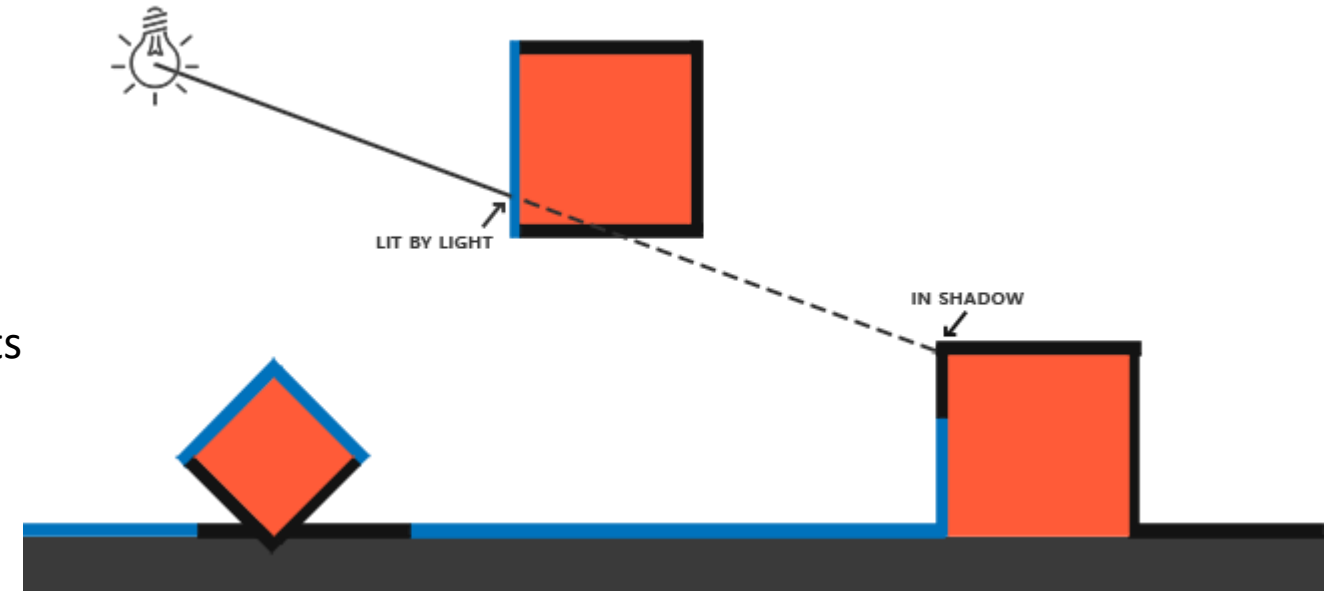


- Rasterization 기반의 실시간 렌더링에서 완벽한 그림자 알고리즘은 존재하지 않음
  - 다양한 shadow approximation (근사) 방법은 화질 및 성능상 문제로 인해 까다로운 구현이 필요
- Real-time ray tracing은 훨씬 고품질의 그림자를 제공하지만 많은 연산량을 요구함

Shadow Mapping

- Shadow mapping
  - 실시간 렌더링에서 가장 많이 사용되는 shadow 기법
  - 적당한 결과를 제공하고 고성능도 필요치 않음
  - 이해하기 쉽고 비교적 쉽게 구현 가능
  - 눈에 띄는 결점을 없애기에는 다소 까다로움

- Shadow가 생기는 예시
  - 파란 선: light source 시점에서 보여지는 fragments
  - 검은 선: 같은 시점에서 보여지지 않는 fragments
  - 떠 있는 box에 의해, 맨 오른쪽 box의 일부 영역은 그림자가 져 있는 것을 알 수 있음

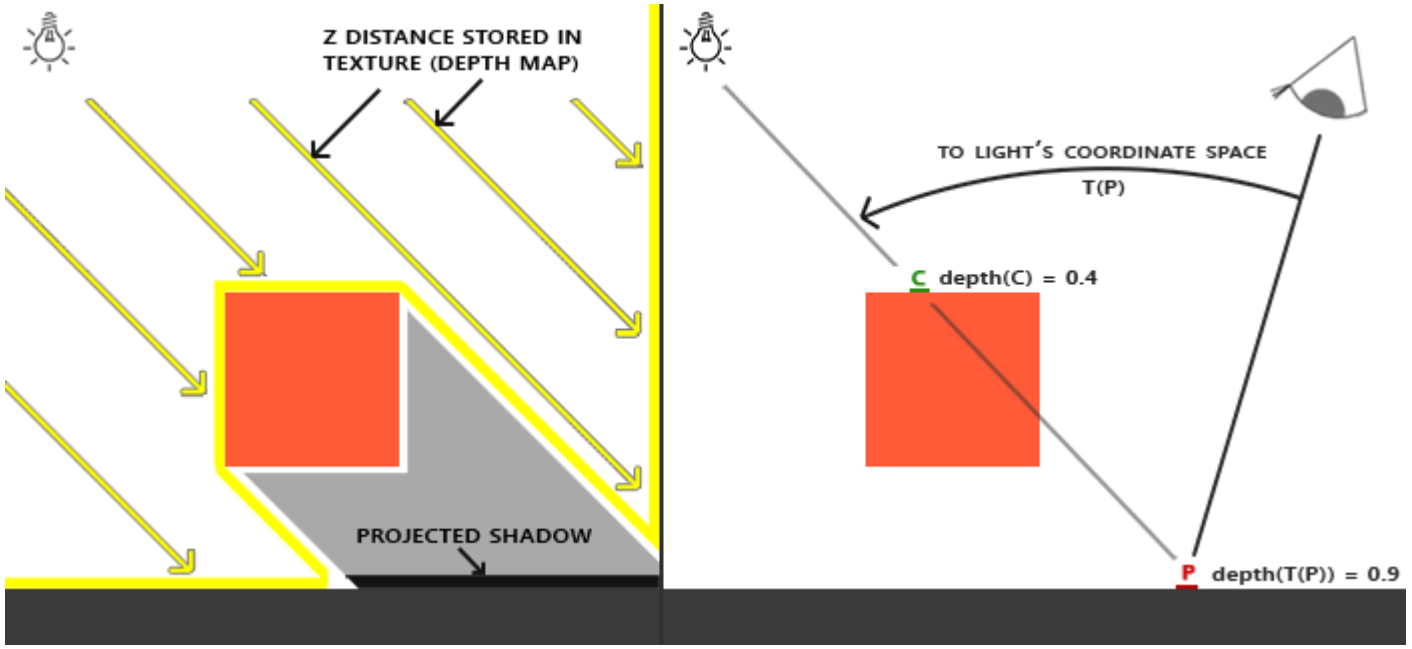


- Shadow를 처리하는 정석 기법 – ray tracing
  - Light source로부터 object의 특정 지점으로 향하는 shadow ray(s)를 생성하여 이 ray의 경로를 추적
  - 이 shadow ray가 occluder를 만난다면? → 해당 지점은 그림자 진 영역으로 판단

Shadow Mapping

- Shadow mapping의 원리
  - 1단계) Light source의 시점에서 view/projection matrix를 이용해 scene을 렌더링하고, 이 결과 깊이 값을 깊이 맵(depth map) 또는 그림자 맵(shadow map)이라 불리우는 텍스처에 저장
  - 2단계) 깊이 맵의 값을 사용해, light source가 만나는 가장 가까운 점(closest hit point)이 현재 fragment인지 확인
    → 아닐 경우 이 fragment는 그림자 내에 있는 것으로 판단

- Directional light source에서의 예시
  - Light source는 무한히 멀리 모델링되어 있어 특정 위치를 갖지 않으나, shadow mapping을 위해서는 light 방향 위에 있는 적당한 위치를 지정, 이를 통해 depth map 생성
  - 포인트 P가 그림자 진 영역인지 판단하기 위해, transform T를 이용하여 light의 좌표 공간으로 이를 변경
  - Depth(C) < Depth(T(P))이므로, P는 그림자 진 영역임



The Depth Map

- 깊이 맵 생성
  - 깊이 맵을 그릴 FBO 생성

```
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

  - 프레임버퍼의 깊이 버퍼로 사용할 2D 텍스처 생성 (아래 예에서는 1024x1024 해상도)

```
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;

unsigned int depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
             SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

  - 프레임버퍼의 깊이 버퍼에 위에서 만든 2D 텍스처를 attach
    (컬러 버퍼가 없으므로, 명시적으로 draw 및 read buffer를 NONE으로 설정)

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

The Depth Map

- 깊이 맵 생성 (cont.)
  - 이제 shadow mapping의 2-pass 렌더링을 수행
  - 그림자의 해상도와 화면의 해상도는 다르므로, 각 패스별로 glViewport()를 따로 호출해줘야 함

```
// 1. first render to depth map
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    ConfigureShaderAndMatrices();
    RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 2. then render scene as normal with shadow mapping (using depth map)
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene();
```

The Depth Map

- Light space transform
  - 첫번째 패스의 ConfigureShaderAndMatrices() 함수에서 처리하는 부분
  - Directional light에서는 모든 광선(ray)이 평행하므로, 직교 투영 행렬 사용.
    투영 행렬은 가시 범위를 결정하므로, 깊이 맵에 포함되어야 하는 객체가 frustum 내에 존재하도록 설정

```
float near_plane = 1.0f, far_plane = 7.5f;
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
```

  - Light source의 위치에서 scene의 중심을 바라보도록 view matrix 설정

```
glm::mat4 lightView = glm::lookAt(glm::vec3(-2.0f, 4.0f, -1.0f),
                                  glm::vec3( 0.0f, 0.0f,  0.0f),
                                  glm::vec3( 0.0f, 1.0f,  0.0f));
```

  - 위 두 행렬을 결합하면 world space의 벡터를 light space로 변환시켜 주는 행렬(T)가 됨

```
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
```

The Depth Map

- Render to depth map

VS

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}
```

FS

```
#version 330 core

void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}
```

CPU

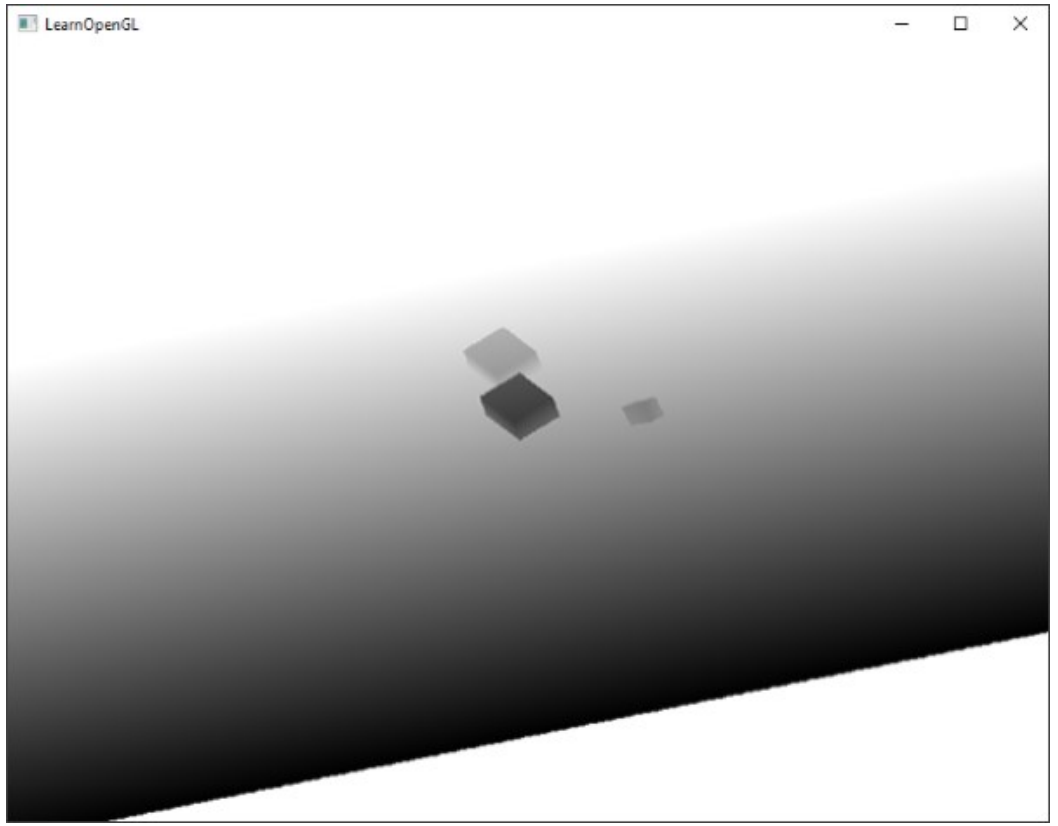
```
simpleDepthShader.use();
glUniformMatrix4fv(lightSpaceMatrixLocation, 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    RenderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

The Depth Map

- Render to depth map (cont.)
  - 디버깅을 위해 별도 패스로 2D render quad를 이용해 이미 만들어진 depth map을 화면에 그려 준 결과

FS

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D depthMap;

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    FragColor = vec4(vec3(depthValue), 1.0);
}
```



Rendering Shadows

- FS에서 shadow check를 하기 전에, VS에서는 이에 필요한 light-space position을 추가로 넘겨줘야 함
  - 이러한 FragPosLightSpace 멤버를 포함한 전체 출력 변수는 interface block을 통해 FS에 전달

VS

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

void main()
{
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;
    vs_out.TexCoords = aTexCoords;
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);
    gl_Position = projection * view * vec4(vs_out.FragPos, 1.0);
}
```

Rendering Shadows

- FS에서는 Blinn-Phong 모델을 이용하여 lighting을 수행하면서 그림자도 함께 반영
  - 그림자 진 곳에 있으면 shadow값이 1.0, 아니면 0.0
  - 빛의 산란(light scattering)으로 인해 그림자가 아주 완전하게 어둡진 않기 때문에, ambient component는 이러한 shadow값에 영향을 받지 않음

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;

uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

float ShadowCalculation(vec4 fragPosLightSpace)
{
    [...]
}
```

FS

```
void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(1.0);
    // ambient
    vec3 ambient = 0.15 * lightColor;
    // diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // calculate shadow
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

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

Rendering Shadows

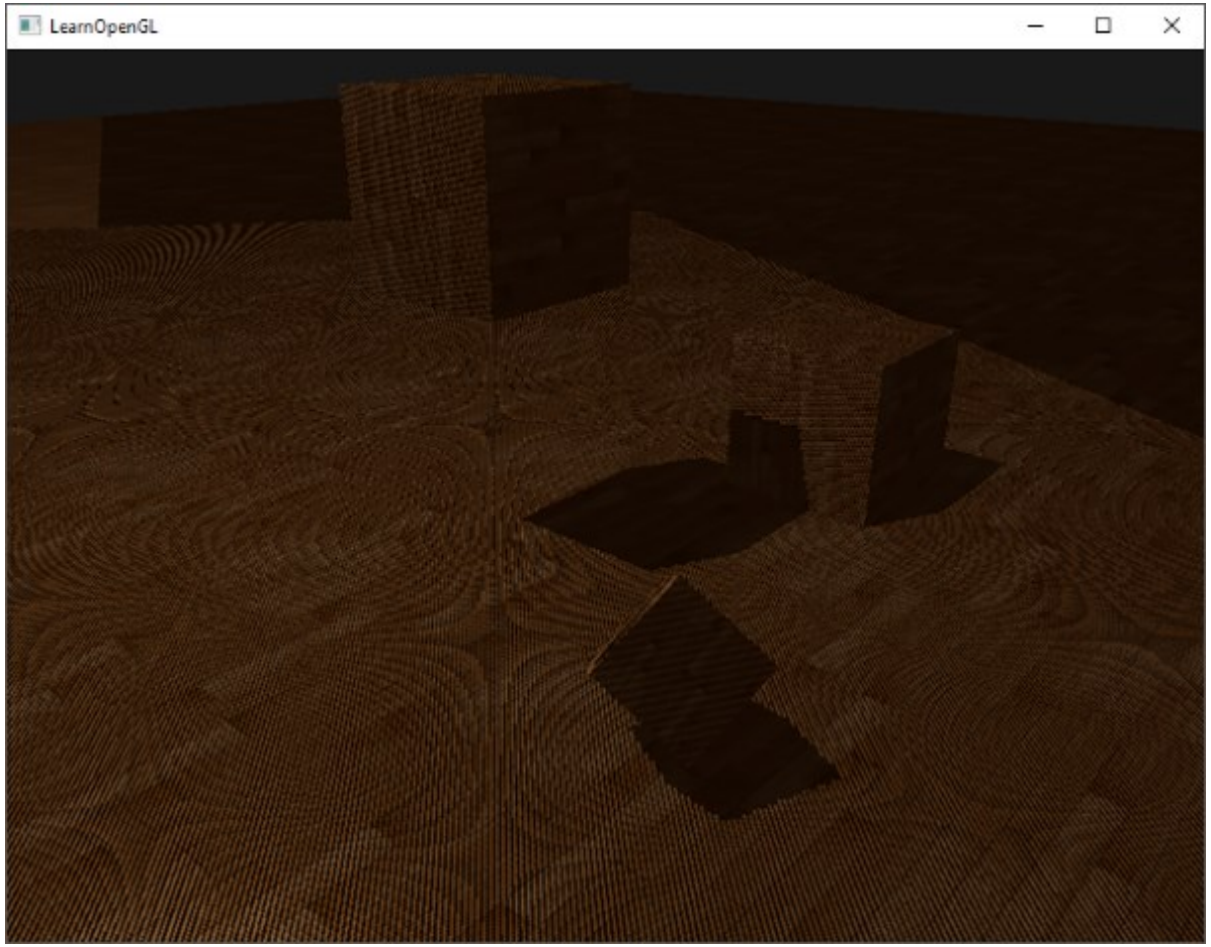
- FS의 ShadowCalculation() 함수는 그림자가 져 있는지 아닌지 판단
  - Perspective division을 먼저 수행하여, [-w,w] 범위의 클립 공간 좌표를 [-1,1]로 수동 변환 (이는 원근 투영일 때에만 의미가 있으나, 직교 투영일 때 사용해도 무방)
  - 이 좌표를 [0, 1] 범위로 변경
  - 이 좌표값의 xy 성분은 shadow map에서 이 좌표에 해당하는 깊이값을 샘플링할 때 사용
  - 이 좌표값의 z성분은 현재 fragment의 깊이값이 됨
  - 이 두 깊이값을 비교하여, 현재 fragment가 그림자 안에 있는지 판단

FS

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // check whether current frag pos is in shadow
    float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;

    return shadow;
}
```

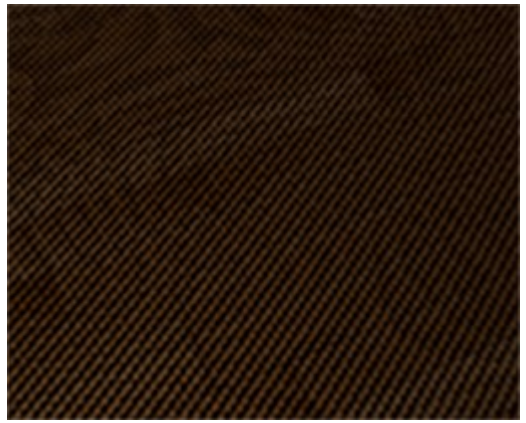
Rendering Shadows

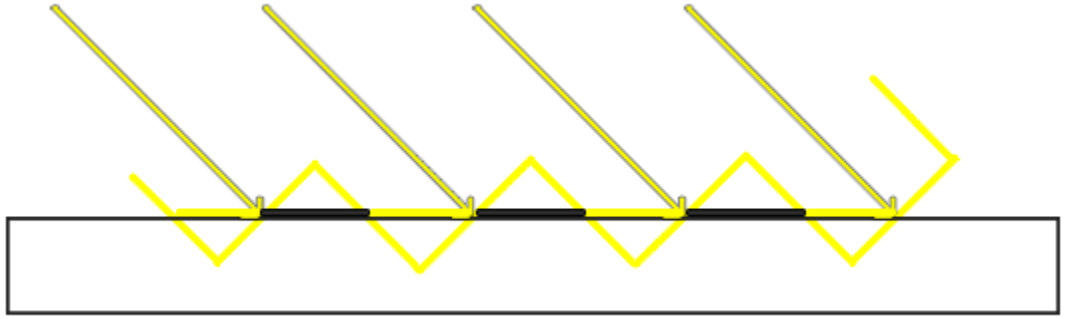
- 실행 결과
  - 그림자가 추가된 걸 확인 가능
  - 하지만 여러 artifact들이 눈에 보임



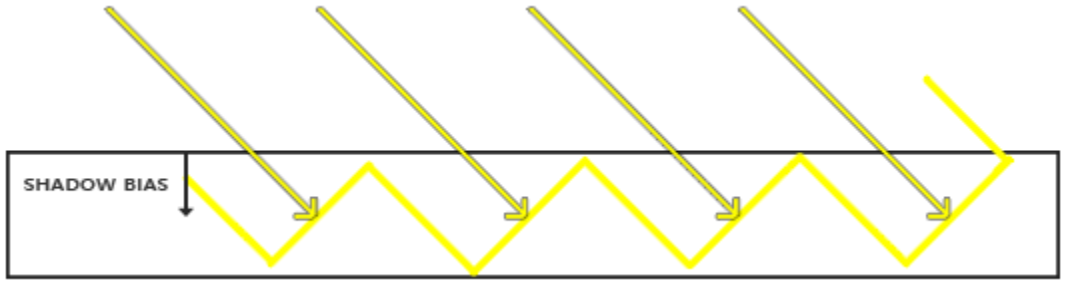
Improving Shadow Maps – Shadow Acne

- Shadow acne
  - 그림자지지 않아야 하는 영역에서 보이는 Moiré-like pattern
  - Light source가 멀리 떨어져 있을 경우 여러 개 fragment들이 shadow map의 같은 지점을 샘플링 가능
    → 이 때 광원과 표면 간의 각도에 따라 그림자 간의 불일치가 발생할 수 있음 (잘못된 self-shadowing)





- Shadow bias
  - Shadow acne를 완화하는 간단한 방법
  - 표면 또는 shadow map의 depth에 작은 bias를 줌



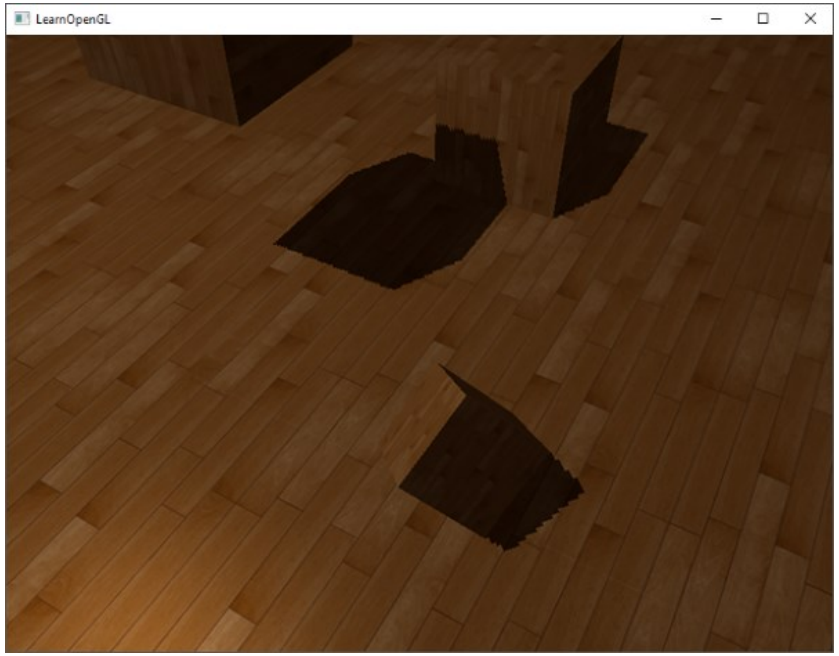
Improving Shadow Maps – Shadow Acne

- Shadow bias (cont.)
  - 빛에 대한 표면의 각을 기준으로 0.005에서 0.5에 이르는 bias를 줄 수 있음
  - 수직이면 bias가 작고 뉘어 있으면 bias가 커지게 되므로 단일 bias값보다 효과적

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```
FS

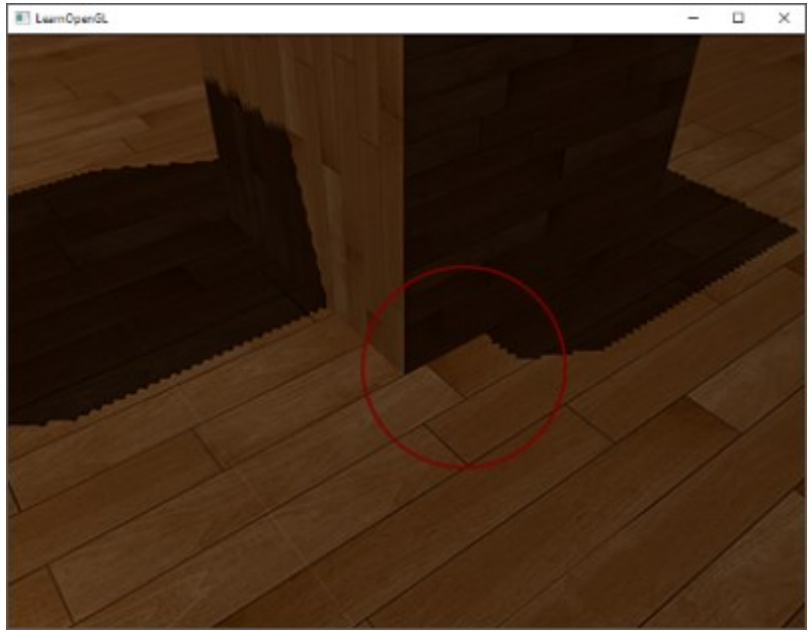
  - Scene별 최적 bias값은 acne가 없어질 때까지 조금씩 늘려가면서 확인

- Shadow bias 적용 결과



Improving Shadow Maps – Peter Panning

- Shadow bias의 문제점 – Peter panning
  - 너무 큰 bias 값은 물체의 실제 깊이값에 offset을 주는 것과 동일하므로, 아래 화면과 같이 그림자의 위치가 밀려서 물체와 그림자 간 단절 가능
  - 이를 Peter Panning이라고 함



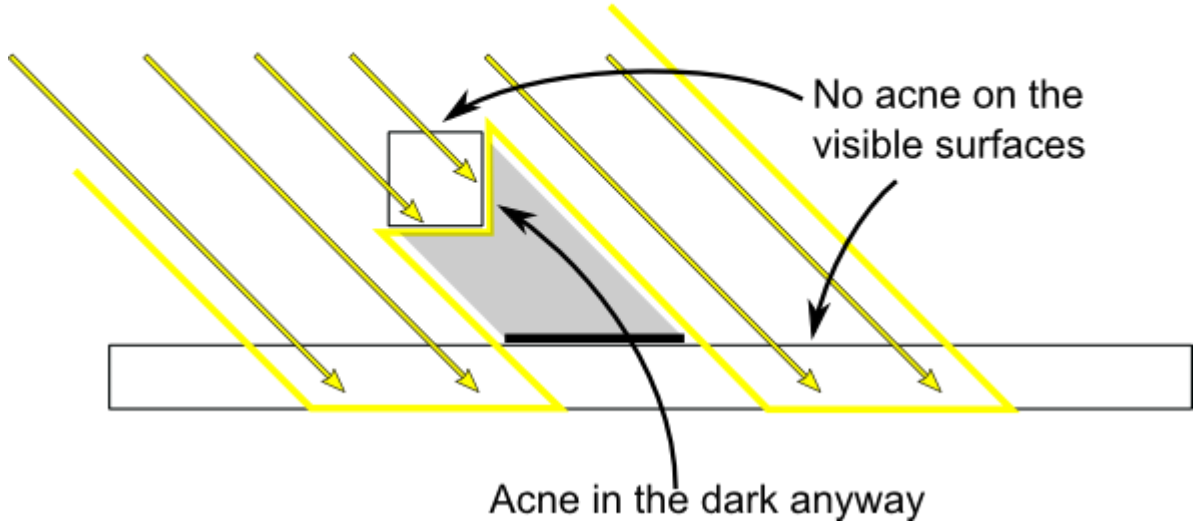
과다한 bias를 준 예

PETER PAN Clip - "Peter's Shadow" (1953) - YouTube

Improving Shadow Maps – Peter Panning

- Peter panning 없이 shadow acne를 완화하는 또다른 방법 – front-facing culling
  - Depth map을 그릴 때, front face를 culling하도록 함
  - Acne 현상이 visible surface가 아닌 back face에 생기므로, 화면상에 보이지 않음
  - 단, 안이 뻥 뚫려 있지 않은 solid object에만 적용 가능
  - 또한, 단면으로 이루어진 object에는 적용 불가능

```
glCullFace(GL_FRONT);                                          CPU
RenderSceneToDepthMap();
glCullFace(GL_BACK); // don't forget to reset original culling face
```



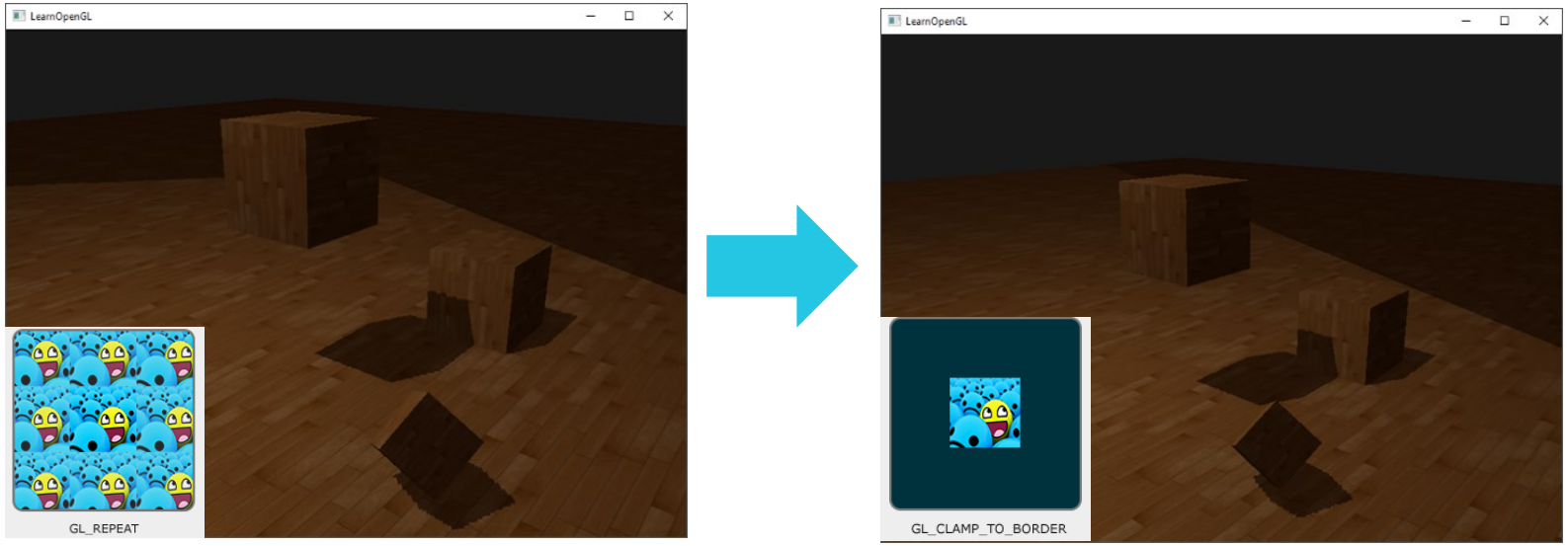
Tutorial 16 : Shadow mapping (opengl-tutorial.org)

Improving Shadow Maps – Over Sampling

- 빛의 visible frustum 밖에 있는 영역이 그림자 안에 있는 것으로 판단되는 현상
  - [0, 1] 좌표 사이의 밖을 sampling하게 되므로, texture wrapping 방법에 따라 잘못된 결과 도출
- 적절한 texture wrapping 설정은 이 현상을 완화
  - GL_CLAMP_TO_BORDER로 wrapping 모드를 설정하고, border color를 1.0으로 지정

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

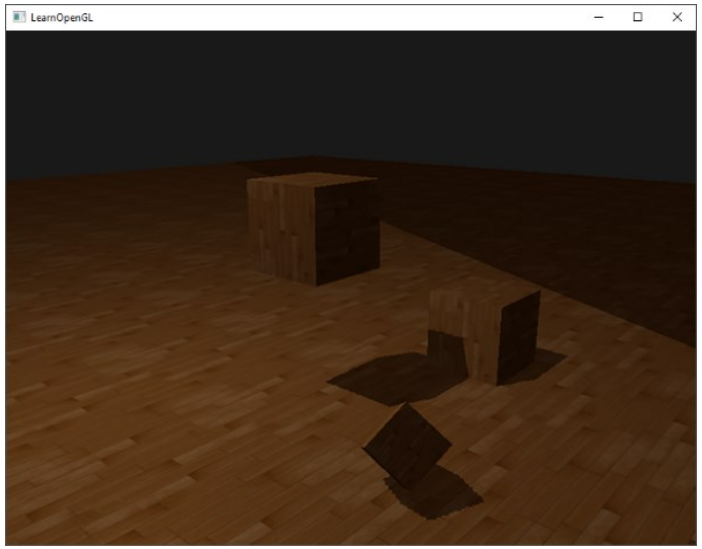
CPU

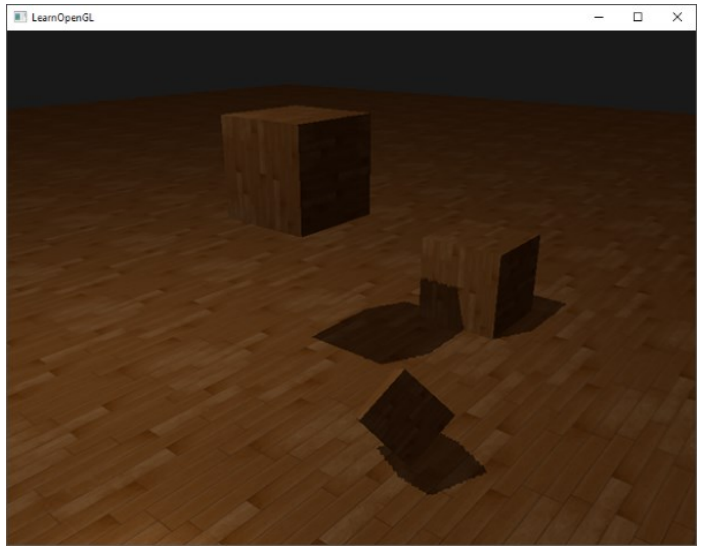


Improving Shadow Maps – Over Sampling

- Light source의 frustum 끝부분(far plane 밖)이 여전히 그림자 져 있음
  - 이 영역에서는 light space로 투영된 fragment의 좌표의 깊이값이 border color값인 1.0보다 크기 때문
  - 간단히 FS에서 이 영역에 대해 shadow 여부를 0으로 설정하도록 함으로써 해결
- 이제 shadow는 light frustum 내부에만 존재 가능

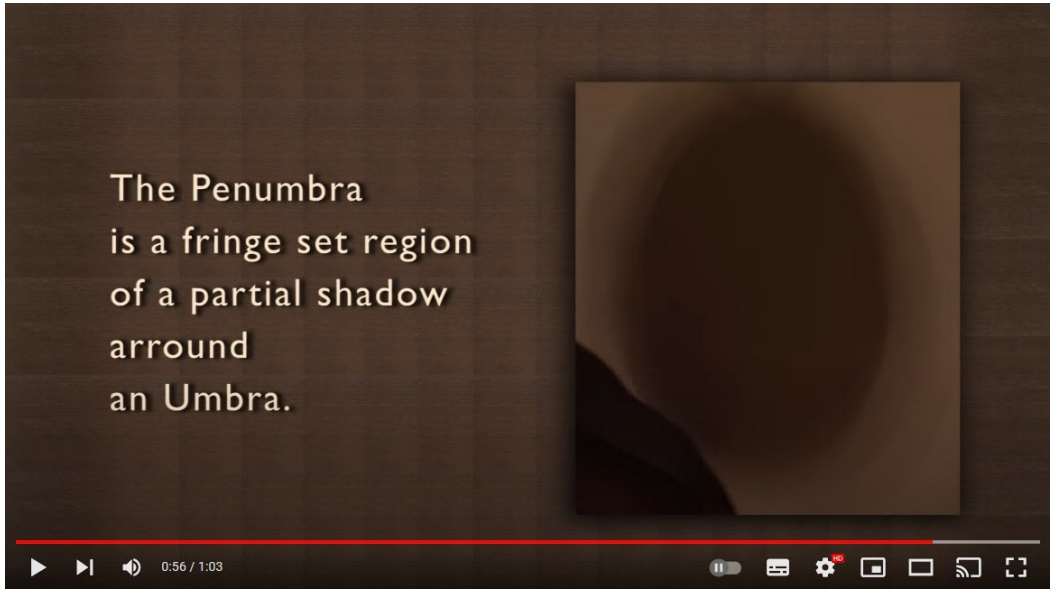
```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    [...]
    if(projCoords.z > 1.0)
        shadow = 0.0;

    return shadow;
}
```
FS





PCF (Percentage-Closer Filtering)

- Blocky shadows의 발생
  - 깊이 맵은 고정된 해상도에서 그려지므로,
    여러 fragment들이 깊이 맵에서 하나의 깊이 값을 함께 sample하는 경우가 발생
  - 이 경우 jagged blocky edges를 만들어 냄
- Blocky shadows의 완화 방법
  - 깊이 맵의 해상도를 올림
    → 하지만 그만큼 성능이 저하되고, 무한정 올릴 수는 없음
  - Light frustum을 scene에 가능한한 딱 맞게 위치
    → 이 역시 항상 가능한 해결법은 아님
  - Hard shadows 대신 soft shadows 생성
    → 다만 무조건적인 soft shadows 생성은
    물리적으로 정확한 방법은 아님
    (soft shadows는 area light를 비출 때
    penumbra 영역에서만 생겨야 함)



Light rays, opaque objects, umbra and penumbra regions | Light | Physics - YouTube

PCF (Percentage-Closer Filtering)

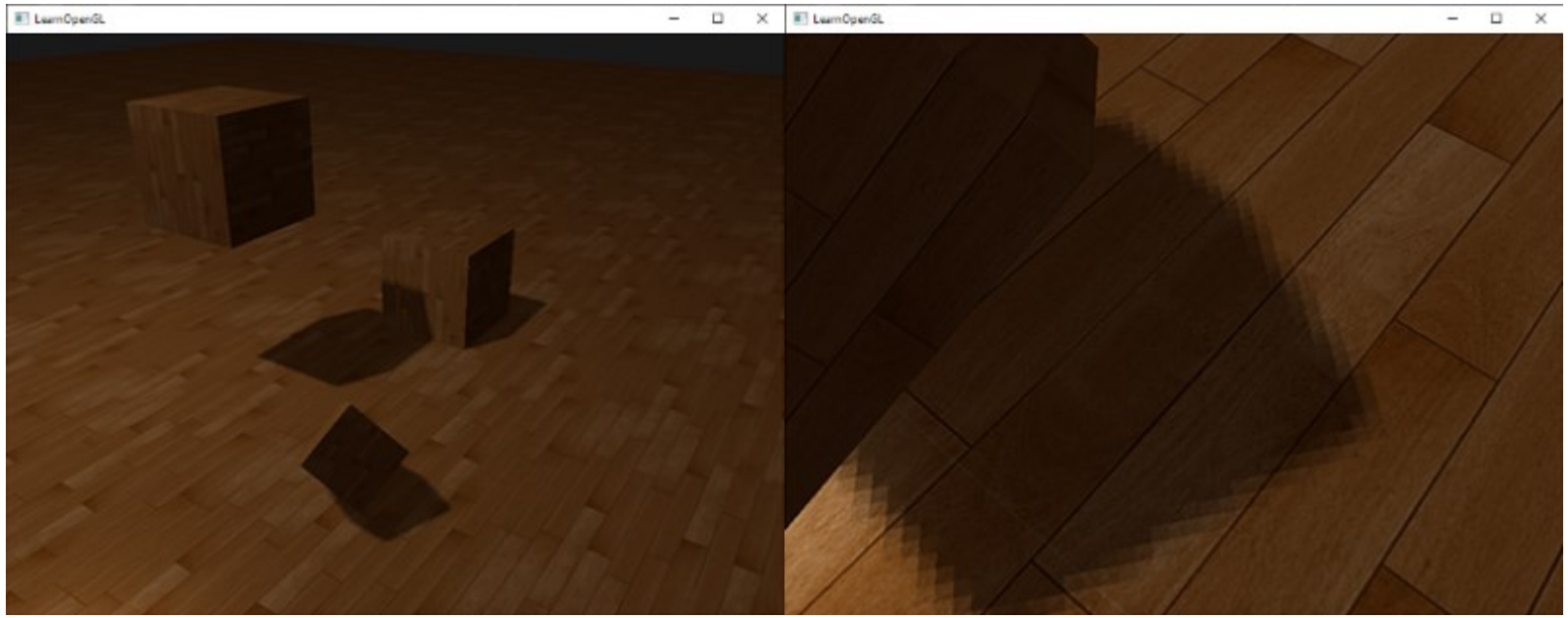
- PCF
  - 약간씩 다른 텍스처 좌표를 이용하여 여러 번 depth map을 샘플링하는 방법
  - 각각의 샘플이 그림자 안에 있는지 없는지 판별
  - 이러한 결과를 결합하여 부드러운 그림자를 만듦
  - 아래 코드에서 textureSize(sampler, lod)함수는 텍스처의 크기(가로&세로)를 반환
    그러므로, 아래 코드는 텍셀의 크기±1 만큼 좌표를 이동시켜 총 9개의 샘플을 취하게 됨

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```

FS

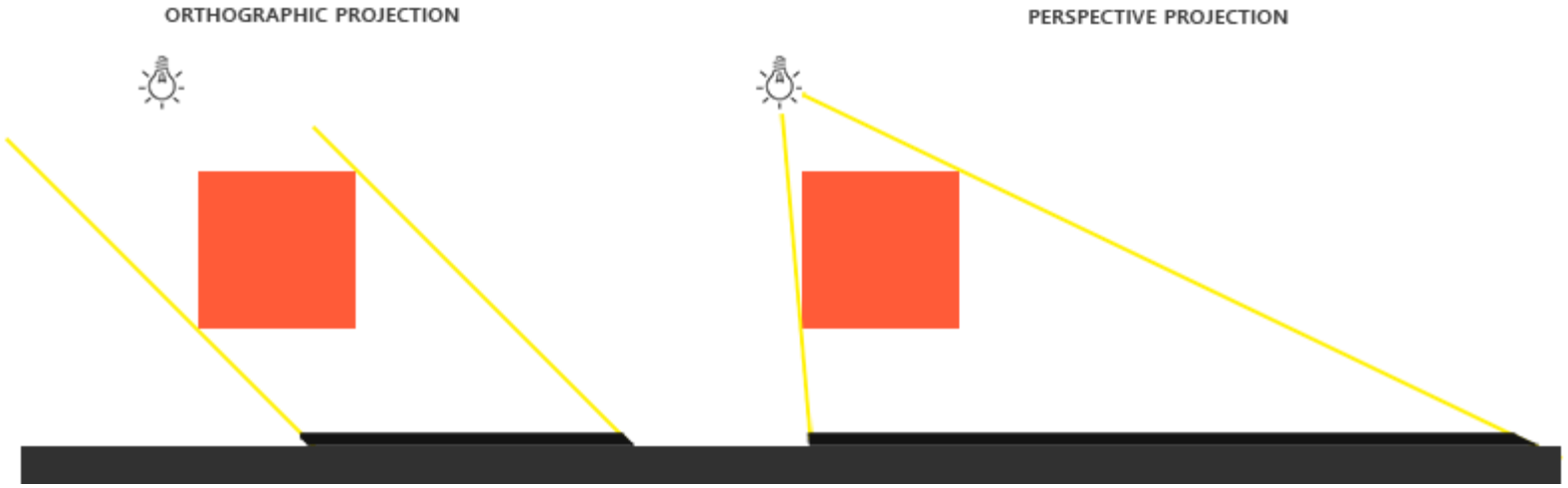
PCF (Percentage-Closer Filtering)

- PCF 렌더링 결과



Orthographic vs Perspective

- Directional light에는 직교 투영이 적합하지만, point light에는 원근 투영이 적합
  - Perspective 투영 사용시, 디버깅 셰이더는 비선형 깊이 값을 선형으로 변환하여 화면에 출력해야 함 (7주차 강의자료 참조)



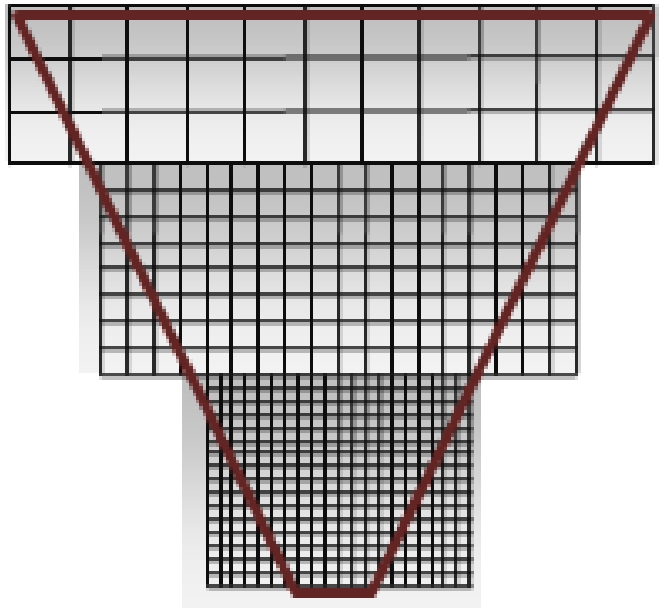

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D depthMap;
uniform float near_plane;
uniform float far_plane;
```

FS

```
float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane - near_plane));
}

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    FragColor = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // perspective
    // FragColor = vec4(vec3(depthValue), 1.0); // orthographic
}
```

Cascaded Shadow Mapping

- Blocky shadows는 근거리에서 눈에 잘 띄는 경향이 있음
  - 여러 fragment들이 하나의 depth map texel을 공유할 가능성이 높아지기 때문
- Cascaded shadow mapping
  - 이러한 점에 착안하여 frustum을 여러 개로 나누고, 각 frustum별로 별도의 shadow map을 만드는 방식
  - 즉, 원거리보다 근거리에서 더 촘촘한 shadow map을 사용하게 됨
- 각 shadow map은 3D 형태의 texture array로 저장되고,
  이후 각 fragment별로 적절한 map을 가져다 sampling함
- 메모리 사용량 및 shadow 연산량이 그만큼 늘어나지만,
  단순히 shadow map의 해상도를 늘리는 것보다는 대체로 효과적
- 코드는 guest article에서 확인 LearnOpenGL – CSM
- 동작 화면: Cascade Shadow Maps in DirectX 9 – YouTube



Cascaded Shadow Maps - Win32 apps | Microsoft Learn

ⓒ nvidia

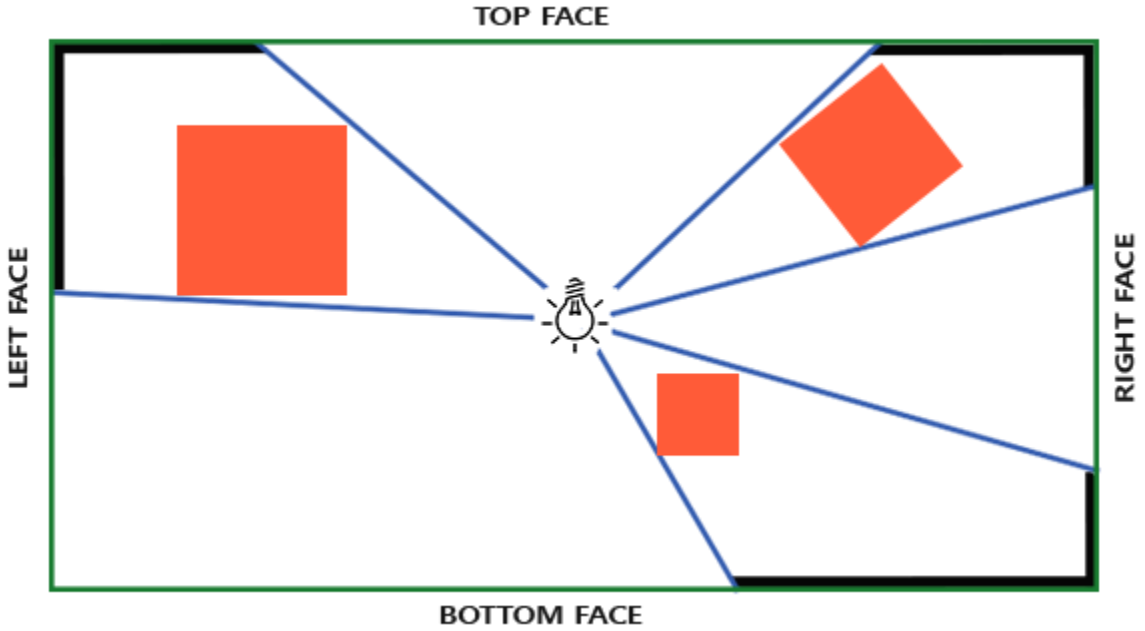
Point Shadows

Point Shadows

- 앞서 소개한 기법은 directional (또는 spot) light에 의한 그림자 처리에 적합
  - Directional shadow mapping이라고도 불림

- Point light에 의해 전방향으로 만들어지는 그림자는?
  - 이를 처리하는 방법은 point (light) shadows 또는 omnidirectional shadow maps라 불림
  - 전체적인 과정은 directional shadow mapping과 유사하나, 깊이값을 cube map에 저장

- Point shadows 처리 과정
  - 전체 scene을 큐브맵의 각 면에 렌더링
  - 이 깊이 큐브맵은 방향 벡터로 큐브맵을 샘플링하는 lighting fragment shader로 전달
  - 이 shader에서는 광원 시점에서 해당 fragment로의 가장 가까운 깊이 값을 얻어내 그림자를 판별



Generating the Depth Cubemap

- 깊이 큐브맵을 생성하는 직관적인 방법 – 큐브맵의 각 면을 6번 렌더링
  - 서로 다른 view matrix로 scene을 6번 렌더링
  - 매번 다른 큐브맵의 면을 FBO에 attach
  - Directional shadow mapping 대비 6배의 render call이 필요
  - 정적인 맵을 읽어들이는 지난번 cube mapping 코드와 달리,
    이번에는 큐브맵을 매 프레임마다 동적으로 생성하므로 추가 최적화가 요구됨

```
for(unsigned int i = 0; i < 6; i++)
{
    GLenum face = GL_TEXTURE_CUBE_MAP_POSITIVE_X + i;
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, face, depthCubemap, 0);
    BindViewMatrix(lightViewMatrices[i]);
    RenderScene();
}
```

CPU

- Geometry shader를 이용하면 단일 패스로 렌더링 가능
  - 단, GPU 구조에 따라 오히려 느려질수도 있으니 반드시 프로파일링을 통해 성능 향상 확인

Generating the Depth Cubemap

- GS를 이용해 단일 패스로 큐브맵을 생성하는 방법
  - 큐브맵 생성

```
unsigned int depthCubemap;
glGenTextures(1, &depthCubemap);
```

  - 각 큐브맵의 면에 2D 깊이 텍스처를 할당

```
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
for (unsigned int i = 0; i < 6; ++i)
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_DEPTH_COMPONENT,
                     SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
```

CPU

  - 텍스처 파라미터 설정

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Generating the Depth Cubemap

- GS를 이용해 단일 패스로 큐브맵을 생성하는 방법 (cont.)
  - GS를 이용해 단일 패스로 큐브맵을 렌더링 할 것이기 때문에, glFramebufferTexture()를 이용해, 직접 cubemap을 프레임버퍼의 depth attachment로 지정

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthCubemap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

CPU

  - 그 다음 코드는 p.7의 기존 directional shadow mapping과 거의 동일 (차이점은 큐브맵을 binding)

```
// 1. first render to depth cubemap
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    ConfigureShaderAndMatrices();
    RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 2. then render scene as normal with shadow mapping (using depth cubemap)
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
RenderScene();
```

Light Space Transform

- 큐브맵의 각 면에 대해 각각 light space transformation matrix를 설정
  - Projection matrix는 각 면이 같은 값을 공유
    단, FOV를 90도로 설정하여 각 면을 충분히 채워야 큐브맵의 edge에서 각 면이 서로 align됨

```
float aspect = (float)SHADOW_WIDTH/(float)SHADOW_HEIGHT;
float near = 1.0f;
float far = 25.0f;
glm::mat4 shadowProj = glm::perspective(glm::radians(90.0f), aspect, near, far);
```

  - View matrix는 각 면마다 따로 설정하여, 큐브맵의 각기 다른 면을 향할 수 있도록 함 CPU

```
std::vector<glm::mat4> shadowTransforms;
shadowTransforms.push_back(shadowProj *
                 glm::lookAt(lightPos, lightPos + glm::vec3( 1.0, 0.0, 0.0), glm::vec3(0.0,-1.0, 0.0));
shadowTransforms.push_back(shadowProj *
                 glm::lookAt(lightPos, lightPos + glm::vec3(-1.0, 0.0, 0.0), glm::vec3(0.0,-1.0, 0.0));
shadowTransforms.push_back(shadowProj *
                 glm::lookAt(lightPos, lightPos + glm::vec3( 0.0, 1.0, 0.0), glm::vec3(0.0, 0.0, 1.0));
shadowTransforms.push_back(shadowProj *
                 glm::lookAt(lightPos, lightPos + glm::vec3( 0.0,-1.0, 0.0), glm::vec3(0.0, 0.0,-1.0));
shadowTransforms.push_back(shadowProj *
                 glm::lookAt(lightPos, lightPos + glm::vec3( 0.0, 0.0, 1.0), glm::vec3(0.0,-1.0, 0.0));
shadowTransforms.push_back(shadowProj *
                 glm::lookAt(lightPos, lightPos + glm::vec3( 0.0, 0.0,-1.0), glm::vec3(0.0,-1.0, 0.0));
```

Depth Shaders

- Geomerty shader
  - VS에서 받은 삼각형들을 6배로 만들어 각 면을 생성 (gl_Layer에 큐브맵에 그릴 면을 지정)
  - 이 때 각 vertex를 light space로 변환하여 방출(emit)하게 됨
  - 변환 전 gl_Position도 FS로 출력 (lightDistance 계산에 쓰임)

VS

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(aPos, 1.0);
}
```

GS

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices=18) out;

uniform mat4 shadowMatrices[6];

out vec4 FragPos; // FragPos from GS (output per emitvertex)

void main()
{
    for(int face = 0; face < 6; ++face)
    {
        gl_Layer = face; // built-in variable that specifies to which face we render.
        for(int i = 0; i < 3; ++i) // for each triangle vertex
        {
            FragPos = gl_in[i].gl_Position;
            gl_Position = shadowMatrices[face] * FragPos;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Depth Shaders

- Fragment shader
  - p. 9의 empty shader와 달리, lightDistance를 [0, 1] 사이로 매핑한 선형 깊이 값을 계산하여 출력
  - 이후 ShadowCalculation() 함수 구현이 좀 더 직관적이 됨

FS

```
#version 330 core
in vec4 FragPos;

uniform vec3 lightPos;
uniform float far_plane;

void main()
{
    // get distance between fragment and light source
    float lightDistance = length(FragPos.xyz - lightPos);

    // map to [0;1] range by dividing by far_plane
    lightDistance = lightDistance / far_plane;

    // write this as modified depth
    gl_FragDepth = lightDistance;
}
```

Omnidirectional Shadow Maps

- 실제 shadow를 그리는 단계는 기존과 유사
  - 단, 2D 텍스처 대신 큐브맵 텍스처를 바인딩하고, light projection의 far plane을 shader에 전달
  - RenderScene()은 중앙의 광원 주위에 흩어져 있는 큰 큐브 룸에서 몇 개의 큐브를 렌더링

CPU

```
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shader.use();
// ... send uniforms to shader (including light's far_plane value)
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
// ... bind other textures
RenderScene();
```

Omnidirectional Shadow Maps

- VS/FS도 기존과 유사하나,
  VS_OUT에서 FragPosLightSpace 삭제
  - FS가 light space 상의 fragment 위치를 사용하지 않고 방향 벡터로 깊이값을 sample하기 때문에,
    VS에서도 위치 벡터를 light space로 변경할 필요 없음
- 기타 FS 변경사항
  - depthMap type이 sampler2D→samplerCube로 변경
  - uniform 변수로
    far_plane 추가

VS

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec2 TexCoords;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;
    vs_out.TexCoords = aTexCoords;
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

FS

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
} fs_in;

uniform sampler2D diffuseTexture;
uniform samplerCube depthMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

uniform float far_plane;

float ShadowCalculation(vec3 fragPos)
{
    [...]
}

void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.3);
    // ambient
    vec3 ambient = 0.3 * color;
    // diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // calculate shadow
    float shadow = ShadowCalculation(fs_in.FragPos);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

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

Omnidirectional Shadow Maps

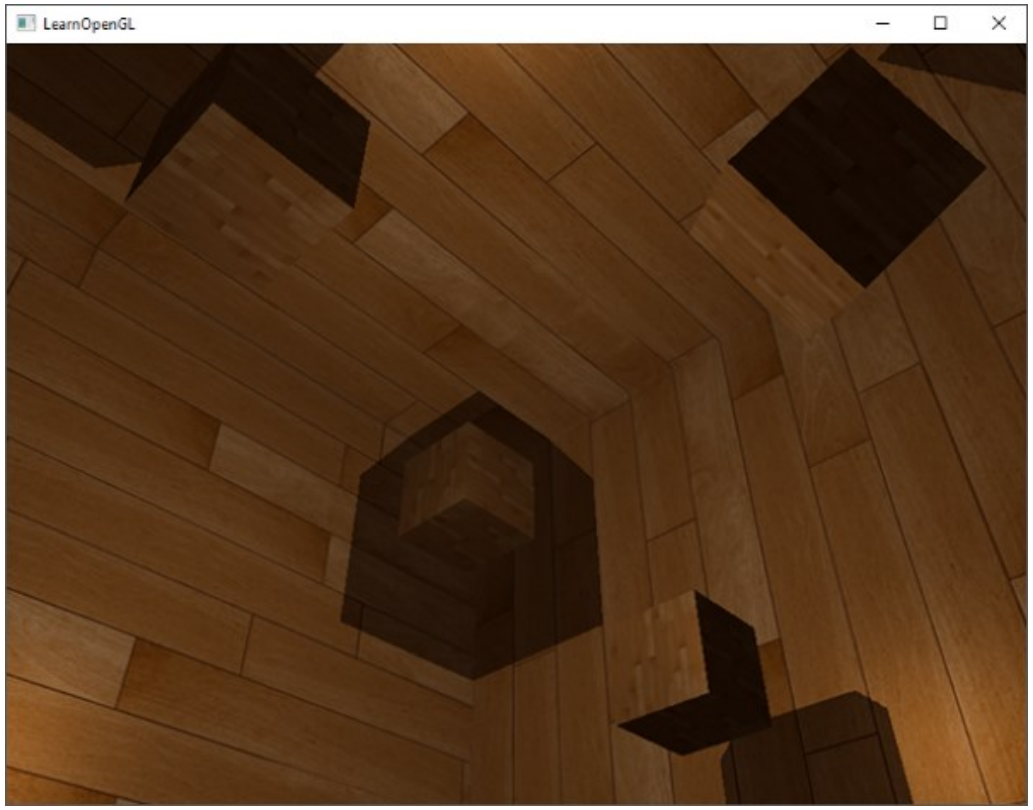
- 또한 FS의 ShadowCalculation() 함수도 변경
  - Fragment와 light 위치 간에 계산된 선형 깊이(currentDepth)와, [0,1]로 저장된 closestDepth값에 far_plane을 곱해 원상복구시킨 closetDepth를 비교
  - 이 때, shadow acne를 방지하기 위해 bias값 사용

FS

```
float ShadowCalculation(vec3 fragPos)
{
    // get vector between fragment position and light position
    vec3 fragToLight = fragPos - lightPos;
    // use the light to fragment vector to sample from the depth map
    float closestDepth = texture(depthMap, fragToLight).r;
    // it is currently in linear range between [0,1]. Re-transform back to original value
    closestDepth *= far_plane;
    // now get current linear depth as the length between the fragment and light position
    float currentDepth = length(fragToLight);
    // now test for shadows
    float bias = 0.05;
    float shadow = currentDepth -  bias > closestDepth ? 1.0 : 0.0;

    return shadow;
}
```

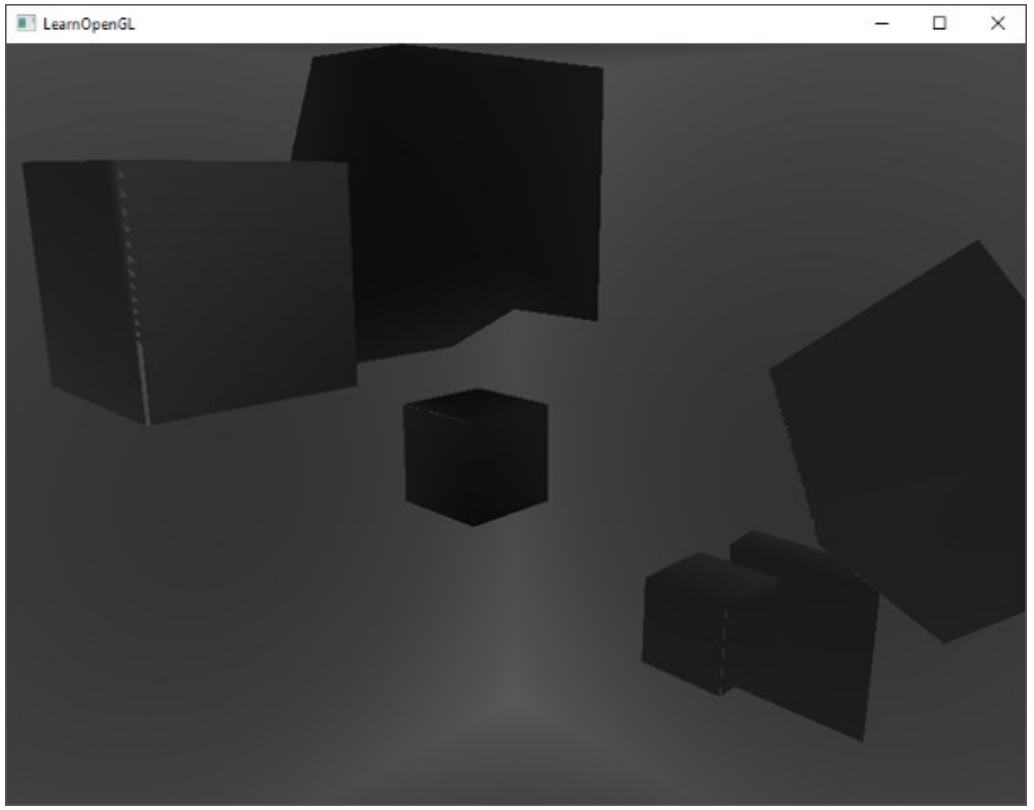
Omnidirectional Shadow Maps

- Light의 시점에서 렌더링한 결과



- 아래 코드를 이용하여 cubemap을 visualization한 결과

```
FragColor = vec4(vec3(closestDepth / far_plane), 1.0);
```
FS

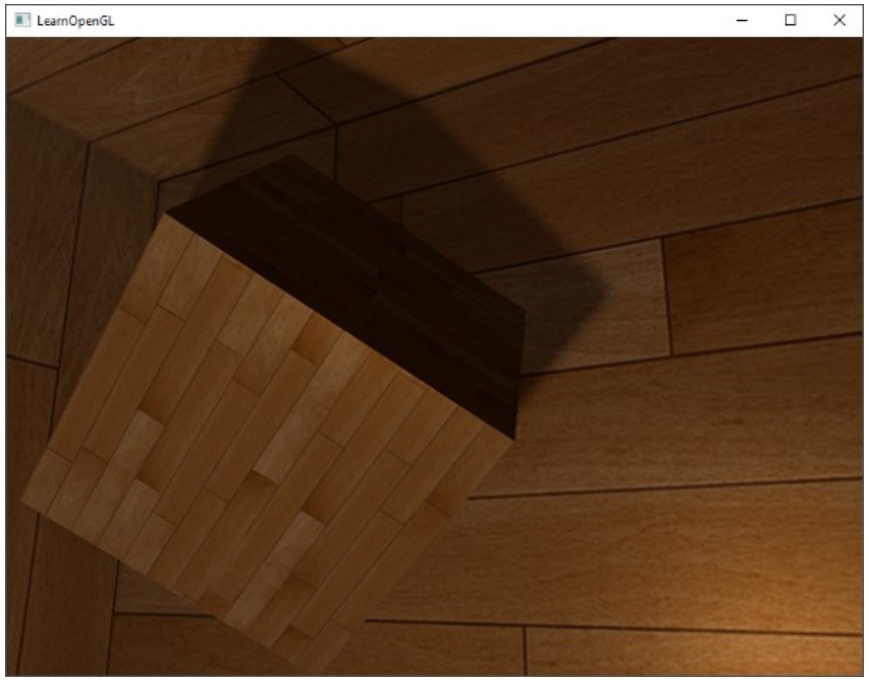


PCF

- Omnidirectional Shadow maps에서도 jagged edge를 없애기 위해 PCF 사용 가능
  - 단, 2D가 아닌 3D offset을 사용하게 됨
  - 아래 예는 총 64 (4^3)개의 샘플을 사용

FS

```
float shadow  = 0.0;
float bias    = 0.05;
float samples = 4.0;
float offset  = 0.1;
for(float x = -offset; x < offset; x += offset / (samples * 0.5))
{
    for(float y = -offset; y < offset; y += offset / (samples * 0.5))
    {
        for(float z = -offset; z < offset; z += offset / (samples * 0.5))
        {
            float closestDepth = texture(depthMap, fragToLight + vec3(x, y, z)).r;
            closestDepth *= far_plane;   // undo mapping [0;1]
            if(currentDepth - bias > closestDepth)
                shadow += 1.0;
        }
    }
}
shadow /= (samples * samples * samples);
```



PCF

- 전방향에 대한 샘플을 취하는 것은 낭비
  - 샘플의 대부분은 원래 방향 벡터에 가깝기 때문
  - 샘플 방향 벡터의 수직 방향으로만 샘플링을 하면 샘플 감소 가능
  - 아래 예에서는 20개 방향을 취하고, diskRadius로 scale 조절

```
vec3 sampleOffsetDirections[20] = vec3[]
(
   vec3( 1,  1,  1), vec3( 1, -1,  1), vec3(-1, -1,  1), vec3(-1,  1,  1),
   vec3( 1,  1, -1), vec3( 1, -1, -1), vec3(-1, -1, -1), vec3(-1,  1, -1),
   vec3( 1,  1,  0), vec3( 1, -1,  0), vec3(-1, -1,  0), vec3(-1,  1,  0),
   vec3( 1,  0,  1), vec3(-1,  0,  1), vec3( 1,  0, -1), vec3(-1,  0, -1),
   vec3( 0,  1,  1), vec3( 0, -1,  1), vec3( 0, -1, -1), vec3( 0,  1, -1)
);
```

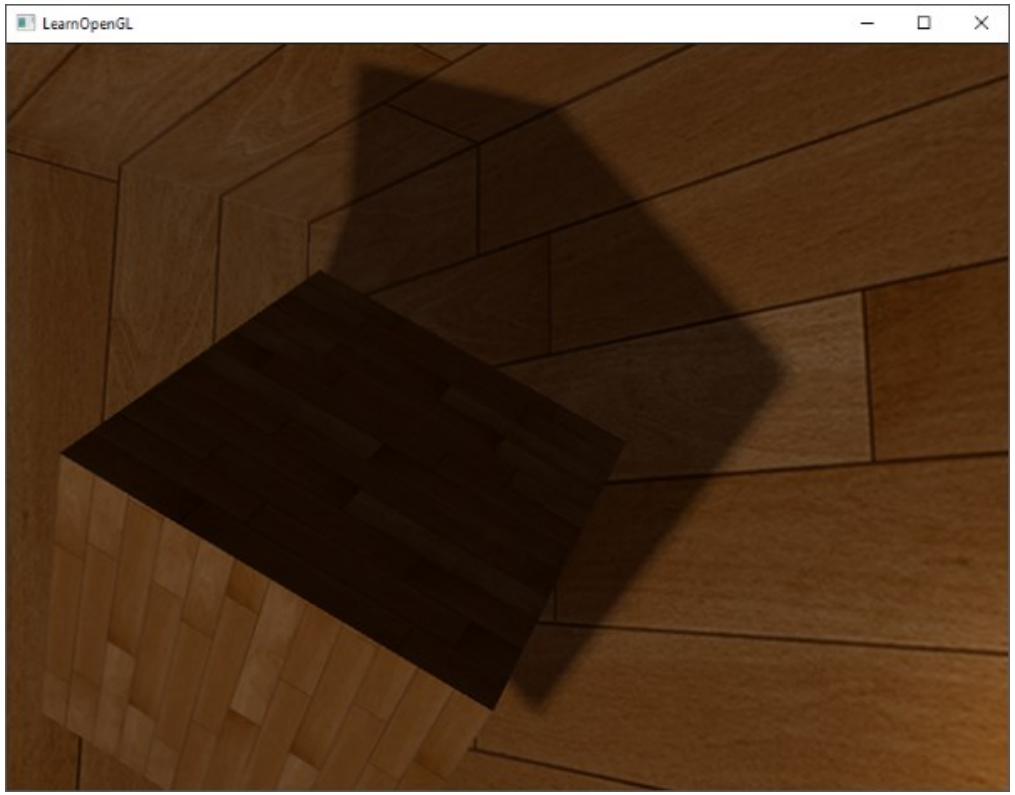
FS

```
float shadow = 0.0;
float bias   = 0.15;
int samples  = 20;
float viewDistance = length(viewPos - fragPos);
float diskRadius = 0.05;
for(int i = 0; i < samples; ++i)
{
    float closestDepth = texture(depthMap, fragToLight + sampleOffsetDirections[i] * diskRadius).r;
    closestDepth *= far_plane;   // undo mapping [0;1]
    if(currentDepth - bias > closestDepth)
        shadow += 1.0;
}
shadow /= float(samples);
```

PCF

- diskRadius값을 viewer로부터 fragment로의 거리에 따라 변경시
  - 시점에서 가까운 그림자는 날카롭게(sharper), 먼 그림자는 부드럽게(softer) 만들 수 있음

```
float diskRadius = (1.0 + (viewDistance / far_plane)) / 25.0; FS
```



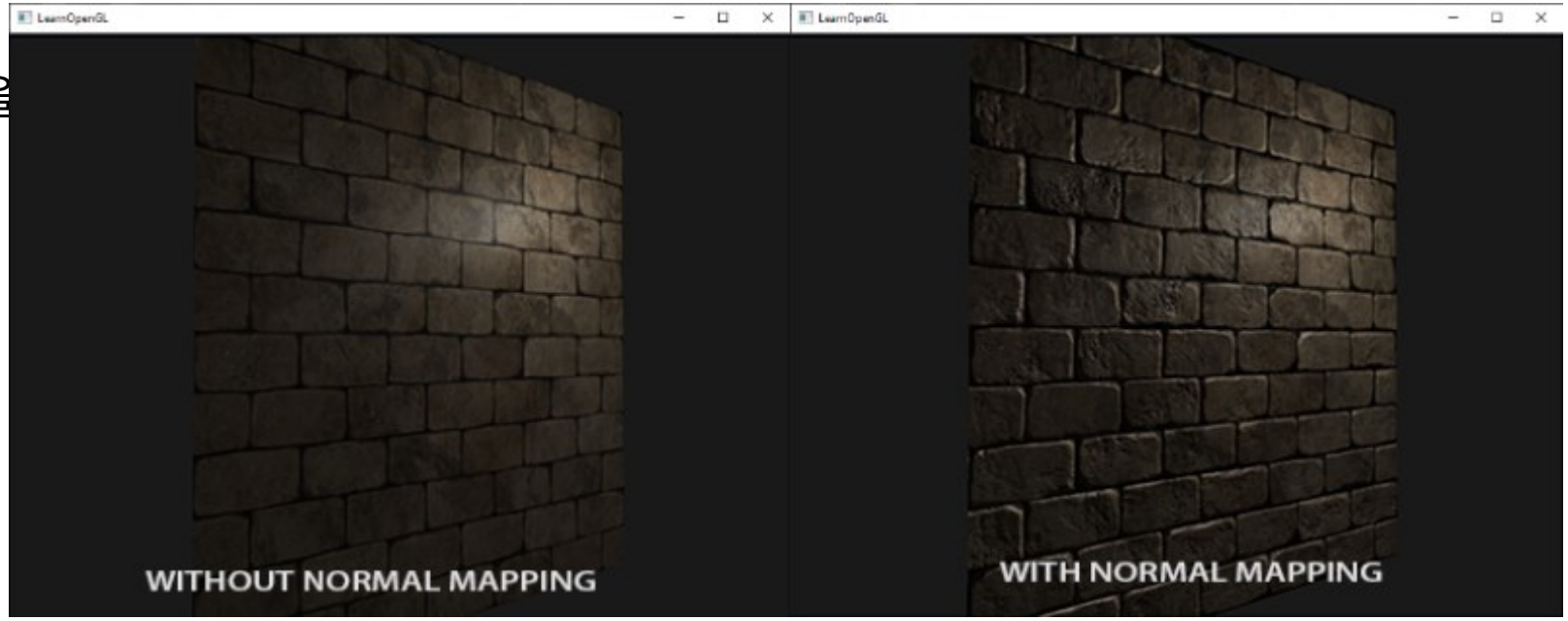
ⓒ nvidia

Normal Mapping

Normal Mapping

- Texture mapping의 한계
  - Texture map은 평평한 삼각형에 입혀지며, 현실세계의 질감을 표현하는 데에는 도움이 됨
  - 하지만, 벽돌과 같이 울퉁불퉁하고 깊이가 서로 다른 물체의 세부사항은 어떻게 표현?
  - 오목하게 들어간 곳은 빛을 덜, 볼록하게 나온 곳은 빛을 더 받아야 하는데,
    단순한 텍스처 매핑만으로는 표현이 힘듦 (specular map은 하나의 hack일 뿐 근본적 해결책은 아님)

- Normal mapping
  - 텍스처의 표면에 detail과 깊이감을
    부여하는 가장 간단한 방법
  - 삼각형 내부의 normal값을
    interpolation(보간)하는 대신
    normal map의 정보로
    per-fragment normal값을 부여
  - 평면의 normal vector가 바뀌면
    lighting시 마치 굴곡이
    들어간 것 같은 효과 표현 가능

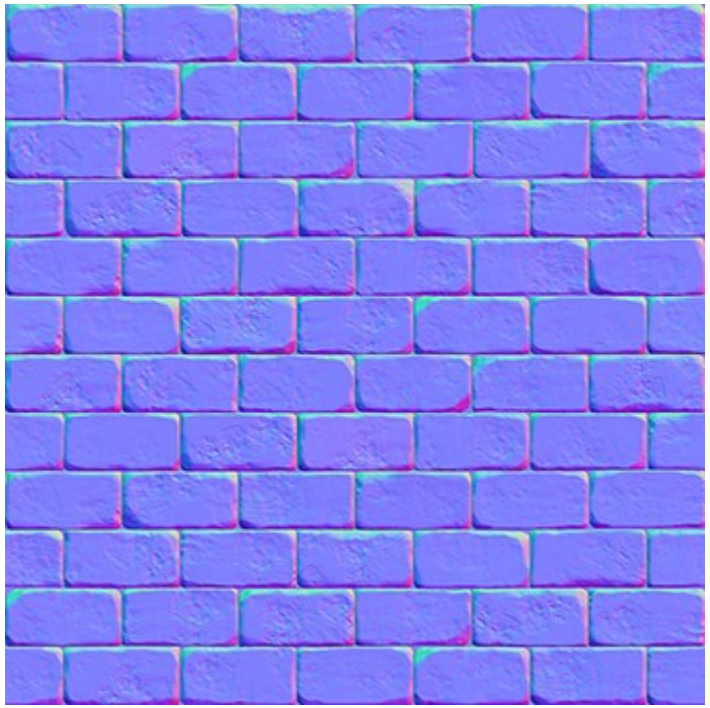


Normal Mapping

- Normal map 생성
  - 기존에 사용했던 색상 또는 lighting data 대신에, 2D texture에 per-fragment normal 저장해야 함
  - 노멀의 XYZ값을 텍스처의 RGB 컴포넌트에 저장 (XY만 저장하고 Z는 셰이더에서 계산하기도 함)
  - 단, 노멀 벡터의 각 성분은 [-1, 1]의 범위에 있으므로, 이를 텍스처의 [0,1] 범위로 변환해야 함

```
vec3 rgb_normal = normal * 0.5 + 0.5; // transforms from [-1,1] to [0,1]
```

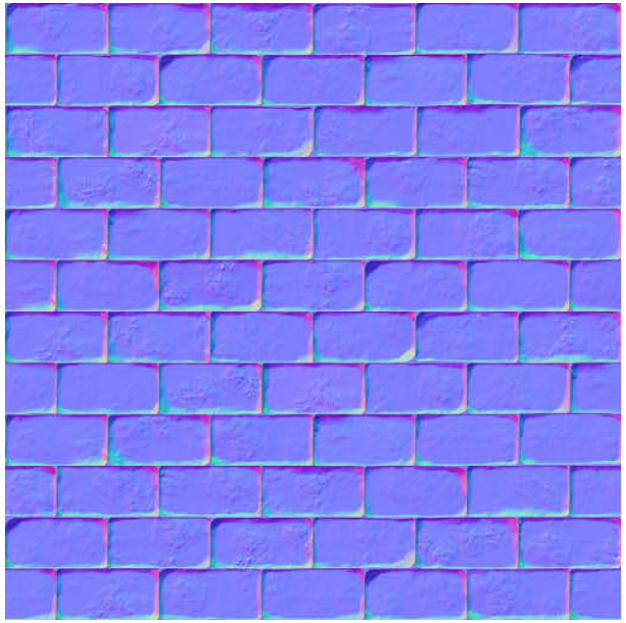
- Normal map이 퍼렇게 보이는 이유?
  - 노멀 벡터의 방향은 대체로 양의 Z 방향을 향하고 있음
    (다른 쪽을 향하는 vector들이 깊이감을 주는 것임)
  - 각 벽돌의 위쪽은 녹색이 강조된 것을 보이고 있는데,
    이는 노멀의 Y축 성분의 값이 1에 가까움을 가리킴을 의미함

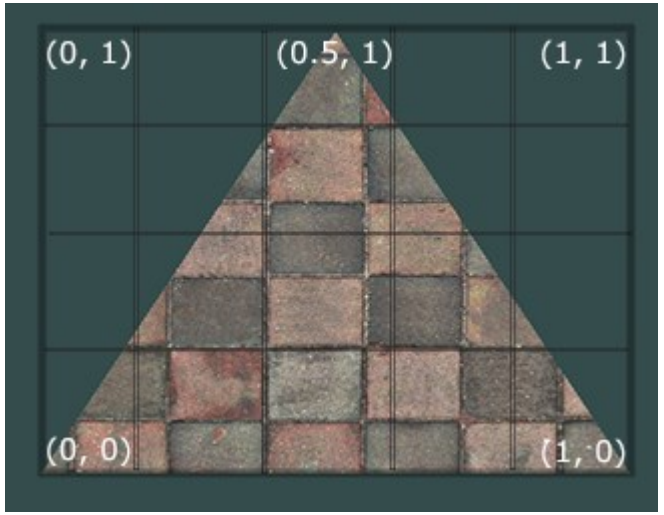


벽돌 노멀 맵의 예

Normal Mapping

- 예제 scene 렌더링에 실제로 사용할 벽돌 color map과 normal 맵
  - Normal map이 앞의 예와 달리 벽돌 아래쪽이 하늘색을 띄는 것을 확인 가능
  - 그 이유는 OpenGL의 st좌표계는 Y축의 방향이 텍스처를 만들 때와 반대이기 때문 (3주차 강의 참조)





Normal Mapping

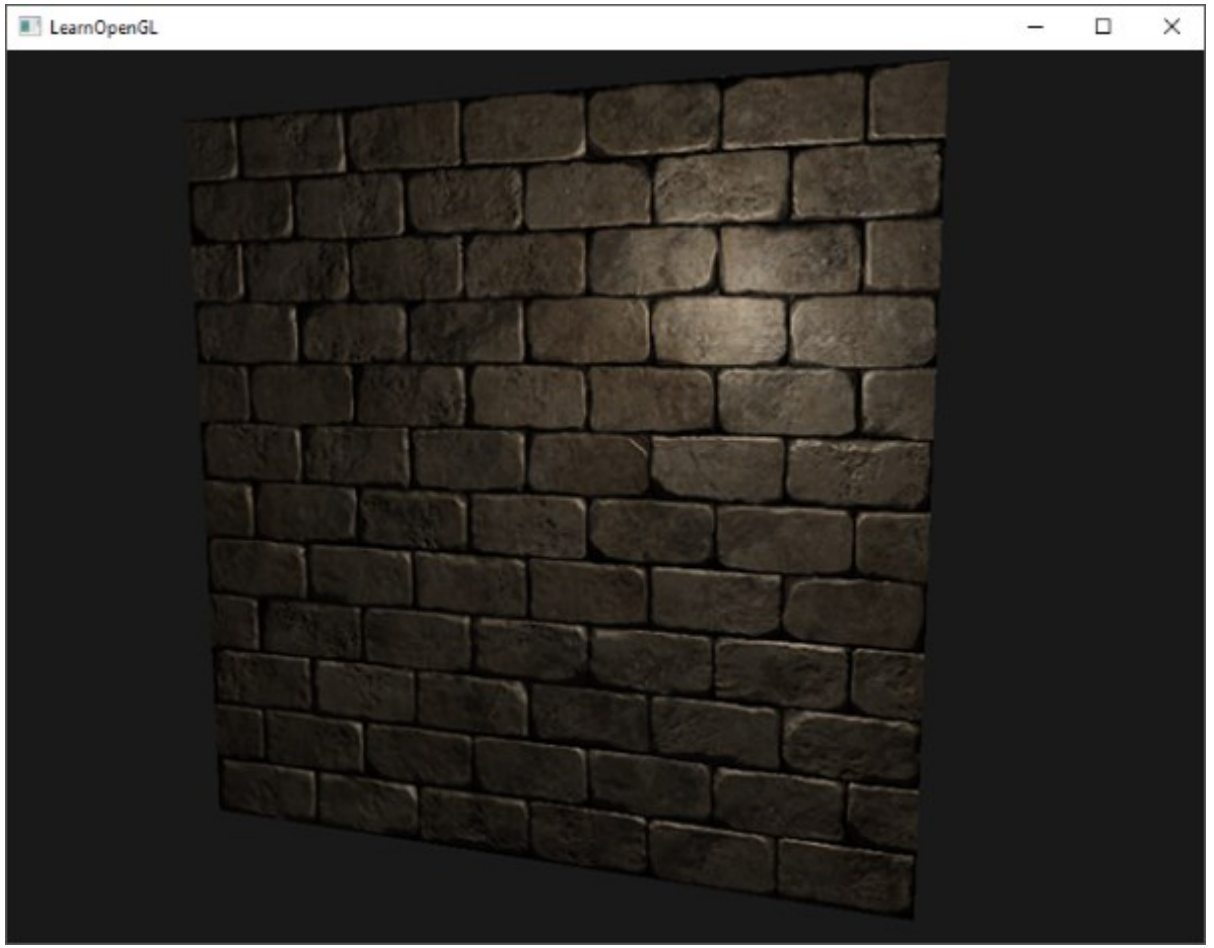
- Fragment shader에서의 normal mapping

FS

```
uniform sampler2D normalMap;

void main()
{
    // obtain normal from normal map in range [0,1]
    normal = texture(normalMap, fs_in.TexCoords).rgb;
    // transform normal vector to range [-1,1]
    normal = normalize(normal * 2.0 - 1.0);

    [...]
    // proceed with lighting as normal
}
```

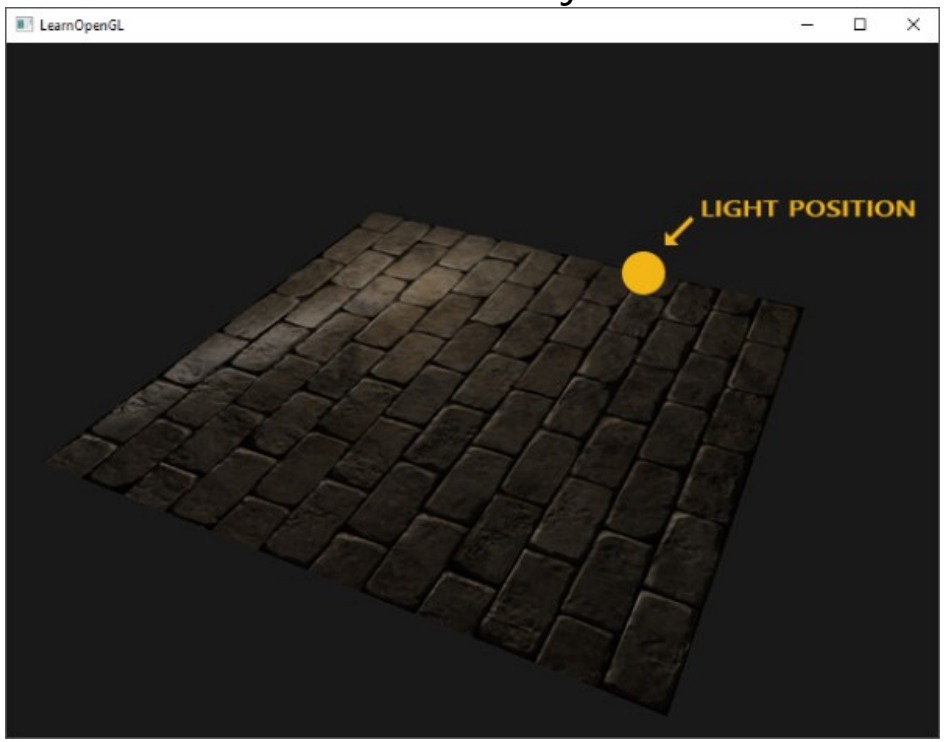
- 결과가 그럴 듯 한데, 이걸로 끝?

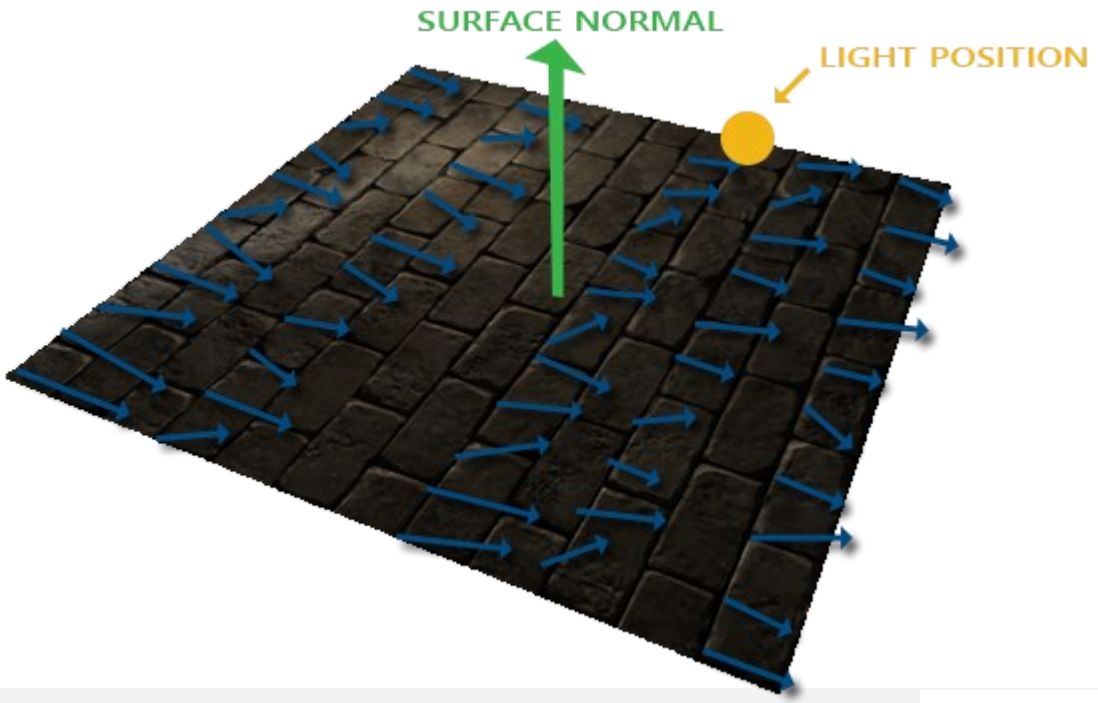
- Blinn-Phong shader에서 normal map을 사용한 화면



Normal Mapping

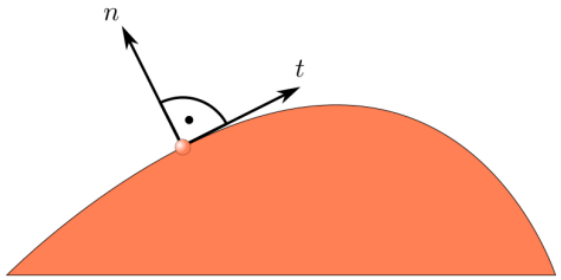
- Normal mapping의 문제점
  - 노멀맵이 입혀진 표면이 향하는 방향에 따라 lighting 계산이 제대로 안될 수 있음
  - 이는 표면의 실제 normal vector(녹색)와 normal map의 vector들(파란색)이 완전히 불일치할 수 있기 때문
- 표면의 각 방향을 가리키는 normal을 따로 따로 저장하면 어느 정도 해결 가능하지만…
  - Cube와 같이 간단한 object에서만 가능

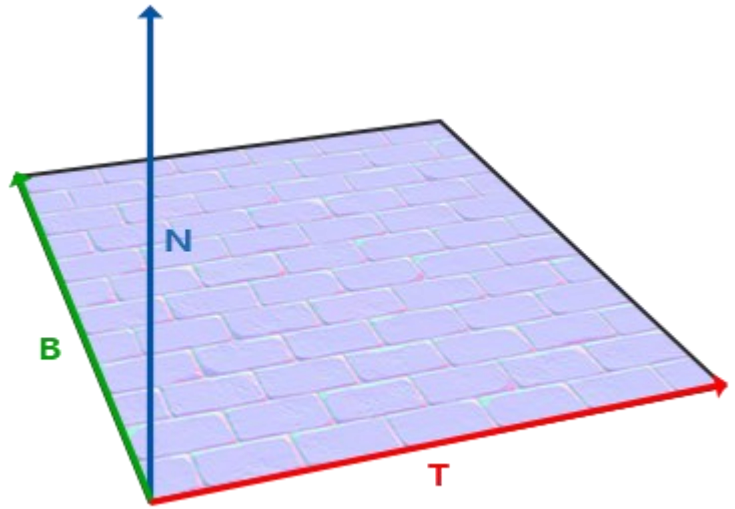




Tangent Space

- Normal map의 normal vector
  - Normal이 항상 양의 Z 방향을 대략적으로 가리키는 tangent space 안에 있다고 표현 가능
- Tangent space이란?
  - Normal vector(N), tangent vector(T), 이 둘을 cross product한 bitangent vector(B)를 축으로 하는 공간
- 이러한 지역적인(local) tangent space의 normal vector들을 world/view 좌표들로 바꿔주는 기저 변환 행렬(change-of-basis matrix)을 사용한다면?
  - Surface normal에 맞게 normal map의 vector를 변환할 수 있음!





Tangent Space

- TBN (tangent-bitangent-normal) matrix

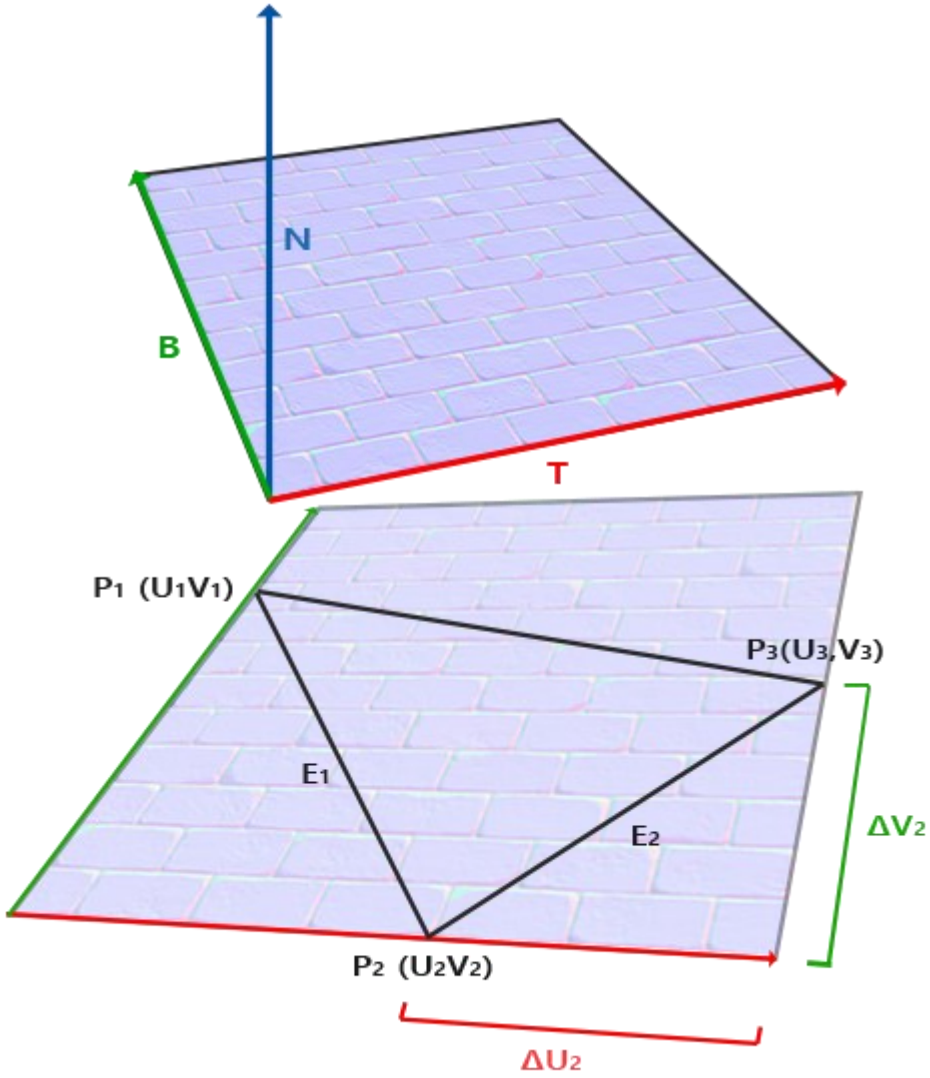
$$E_1 = \Delta U_1 T + \Delta V_1 B$$
$$E_2 = \Delta U_2 T + \Delta V_2 B$$

$$(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1 (T_x, T_y, T_z) + \Delta V_1 (B_x, B_y, B_z)$$
$$(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2 (T_x, T_y, T_z) + \Delta V_2 (B_x, B_y, B_z)$$

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$



Tangent Space

- Tangent들과 bitangent들의 수동 계산

```
// positions
glm::vec3 pos1(-1.0,  1.0, 0.0);
glm::vec3 pos2(-1.0, -1.0, 0.0);
glm::vec3 pos3( 1.0, -1.0, 0.0);
glm::vec3 pos4( 1.0,  1.0, 0.0);
// texture coordinates
glm::vec2 uv1(0.0, 1.0);
glm::vec2 uv2(0.0, 0.0);
glm::vec2 uv3(1.0, 0.0);
glm::vec2 uv4(1.0, 1.0);
// normal vector
glm::vec3 nm(0.0, 0.0, 1.0);
```

CPU

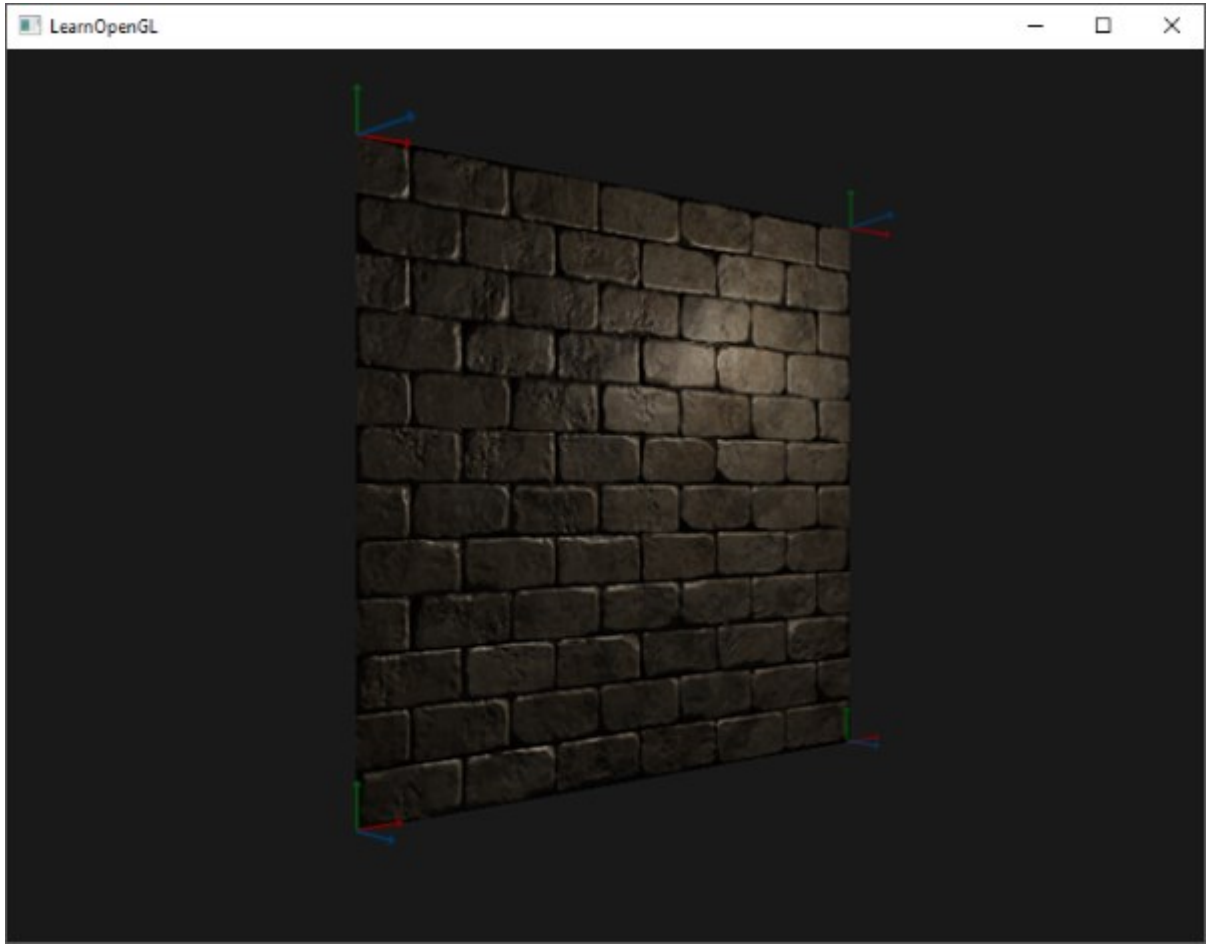
```
glm::vec3 edge1 = pos2 - pos1;
glm::vec3 edge2 = pos3 - pos1;
glm::vec2 deltaUV1 = uv2 - uv1;
glm::vec2 deltaUV2 = uv3 - uv1;
```

```
float f = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x * deltaUV1.y);

tangent1.x = f * (deltaUV2.y * edge1.x - deltaUV1.y * edge2.x);
tangent1.y = f * (deltaUV2.y * edge1.y - deltaUV1.y * edge2.y);
tangent1.z = f * (deltaUV2.y * edge1.z - deltaUV1.y * edge2.z);

bitangent1.x = f * (-deltaUV2.x * edge1.x + deltaUV1.x * edge2.x);
bitangent1.y = f * (-deltaUV2.x * edge1.y + deltaUV1.x * edge2.y);
bitangent1.z = f * (-deltaUV2.x * edge1.z + deltaUV1.x * edge2.z);

[...] // similar procedure for calculating tangent/bitangent for plane's second triangle
```

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

Tangent Space

- Tangent들과 bitangent들의 수동 계산 (cont.)
  - Normal이 (0,0,1)이라면,
    tangent와 bitangent vector는
    (1,0,0)과 (0,1,0)이 됨
  - 이제 적절한 normal mapping을 수행할 준비가 됨



Tangent-space Normal Mapping

- TBN matrix를 VS에서 생성
  - 단, bitangent 변수는 vec3 B = cross(N, T);로 직접 계산도 가능

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in vec3 aTangent;
layout (location = 4) in vec3 aBitangent;
```

VS

```
void main()
{
    [...]
    vec3 T = normalize(vec3(model * vec4(aTangent,   0.0)));
    vec3 B = normalize(vec3(model * vec4(aBitangent, 0.0)));
    vec3 N = normalize(vec3(model * vec4(aNormal,    0.0)));
    mat3 TBN = mat3(T, B, N);
}
```

  - 이 TBN matrix는 어떠한 벡터를 tangent→world space로 변환하거나,
    또는 역행렬을 이용해 반대로 어떠한 벡터를 world→tangent space로 변환하는 데 사용 가능

Tangent-space Normal Mapping

- 첫번째 TBN matrix 적용 방법
  - Normal vector에 TBN matrix를 곱해, 이를 tangent space에서 world space로 변환

VS

```
out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    mat3 TBN;
} vs_out;

void main()
{
    [...]
    vs_out.TBN = mat3(T, B, N);
}
```

FS

```
in VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    mat3 TBN;
} fs_in;
```

```
normal = texture(normalMap, fs_in.TexCoords).rgb;
normal = normal * 2.0 - 1.0;
normal = normalize(fs_in.TBN * normal);
```

Tangent-space Normal Mapping

- 두번째 TBN matrix 적용 방법
  - Lighting 관련 vector에 TBN matrix의 역행렬을 곱해, 이를 world space에서 tangent space로 변환
  - 직교행렬의 역행렬은 이 행렬의 전치행렬과 같으므로,
    inverse()보다 저렴한 transpose() 함수 사용 가능

```
vs_out.TBN = transpose(mat3(T, B, N));
```
VS

FS
```
void main()
{
    vec3 normal = texture(normalMap, fs_in.TexCoords).rgb;
    normal = normalize(normal * 2.0 - 1.0);

    vec3 lightDir = fs_in.TBN * normalize(lightPos - fs_in.FragPos);
    vec3 viewDir  = fs_in.TBN * normalize(viewPos - fs_in.FragPos);
    [...]
}
```

Tangent-space Normal Mapping

- 두번째 TBN matrix 적용 방법 (cont.)
  - lightPos와 viewPos가 fragment마다 갱신될 필요는 없으므로, FS에서의 계산을 VS로 보내면 계산 비용 절감 가능

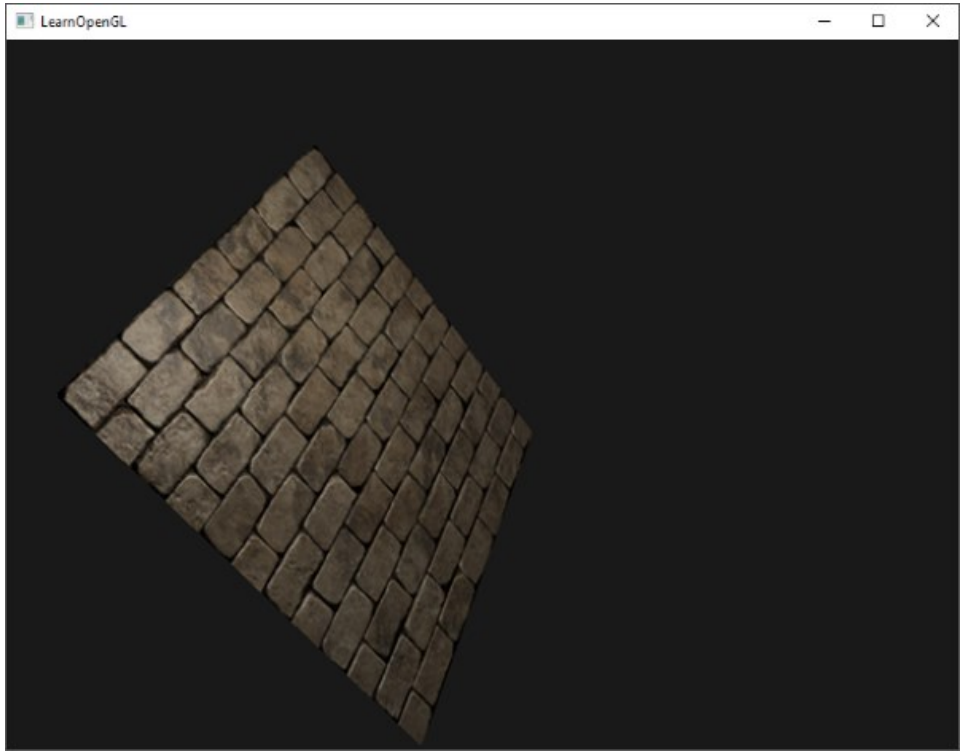
VS

```
out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} vs_out;

uniform vec3 lightPos;
uniform vec3 viewPos;

[...]

void main()
{
    [...]
    mat3 TBN = transpose(mat3(T, B, N));
    vs_out.TangentLightPos = TBN * lightPos;
    vs_out.TangentViewPos  = TBN * viewPos;
    vs_out.TangentFragPos  = TBN * vec3(model * vec4(aPos, 1.0));
}
```

Tangent-space Normal Mapping

- 평면을 회전시키는 코드를 사용하면 normal mapping이 이제 제대로 구현되었음을 확인 가능

```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, (float)glfwGetTime() * -10.0f, glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
shader.setMat4("model", model);
RenderQuad();
```



CPU

Complex Objects

- ASSIMP는 정점들(vertices)를 읽을 때 부드러운 tangent 와 bitangent vector를 계산하는 옵션 제공

```
const aiScene *scene = importer.ReadFile(
    path, aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_CalcTangentSpace
);
```

- 이를 통해 이미 계산된 tangent를 쉽게 검색 가능

```
vector.x = mesh->mTangents[i].x;
vector.y = mesh->mTangents[i].y;
vector.z = mesh->mTangents[i].z;
vertex.Tangent = vector;
```

CPU

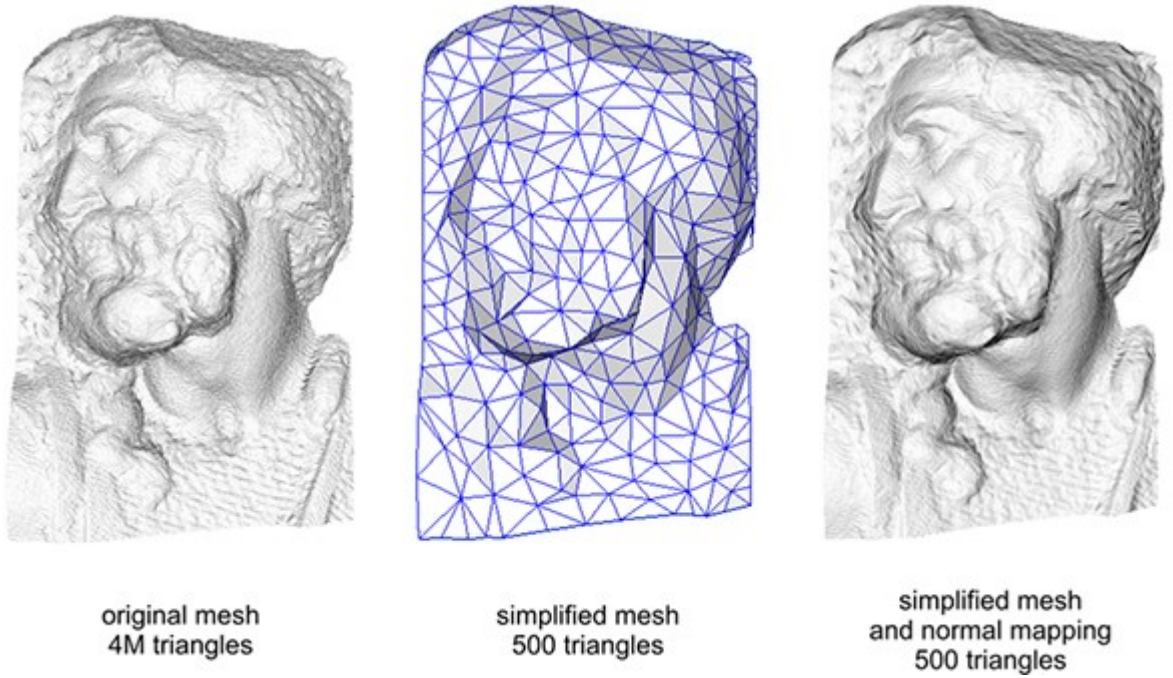
- 다음으로 노멀맵을 읽어들임

```
vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType_HEIGHT, "texture_normal");
```

Complex Objects

- 노멀 맵 적용 결과
  - 추가된 디테일을 확인 가능

  - 모델의 삼각형 개수를 확 줄이기 위해 normal mapping 사용을 고려할 수도 있음



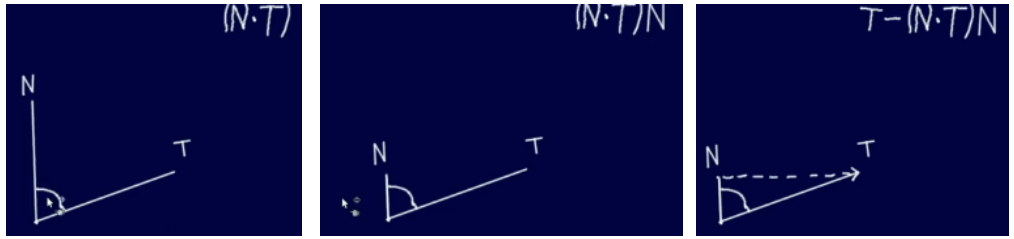
One Last Thing

- 다수개의 삼각형으로 구성된 복잡한 메시(mesh) 안에서 탄젠트 벡터를 계산할 때의 문제
  - 메시 내 삼각형들은 보통 vertex를 공유하는 형태로 구현됨
  - Normal vector처럼 tangent vector도 부드러운 표현을 위해 세 vertex별 값의 평균이 됨
  - 하지만 이 경우, 세 TBN vector는 서로 수직이 아닌 상태에 놓일 수 있고, 이는 TBN matrix가 직교가 아님을 의미

- Gram-Schmidt process를 이용한 해법
  - Tangent vector를 다시 N에 직교하도록 바꿔 주고, 여기에 따라 bitangent도 다시 계산
  - 이 교정에 따라 normal mapping의 결과가 크게 향상

```
vec3 T = normalize(vec3(model * vec4(aTangent, 0.0)));
vec3 N = normalize(vec3(model * vec4(aNormal, 0.0)));
// re-orthogonalize T with respect to N
T = normalize(T - dot(T, N) * N);
// then retrieve perpendicular vector B with the cross product of T and N
vec3 B = cross(N, T);

mat3 TBN = mat3(T, B, N)
```

VS



OpenGL Game Rendering Tutorial: Normal Mapping Mathematics - YouTube

ⓒ nvidia

마무리

마무리

- Advanced lighting의 두번째 시간으로, 아래와 같은 내용을 살펴보았습니다.
  - Shadow mapping
  - Point shadows
  - Normal mapping

- 다음 실습 시간에는 간단하게 예제 코드를 수정하고 실행해 볼 예정입니다.
  - LearnOpenGL 5.3.1.2의 문제 있는 shadow mapping 코드를 제대로 수정 (5.3.1.3처럼)
  - LearnOpenGL 5.3.2.1의 point_shadows에 PCF 적용 (5.3.2.2처럼)
  - LearnOpenGL 5.4의 normal mapping 코드에서 normal mapping 계산 부분을 다 삭제하여, normal mapping 적용 전후의 성능(Fraps 사용) 및 화질 비교

- 실습은 다음 시간이 마지막이며, 프로젝트 발표 이후에는 이론 수업만 진행됩니다.
  - Parallax mapping, HDR, bloom, deferred shading, SSAO 등은 렌더링시 optional한 요소이기 때문
  - 지금까지 해 왔던 코드 단위의 세세한 설명 대신, 개념 중심으로 보다 간략하게 강의 자료를 구성할 예정