서강대학교

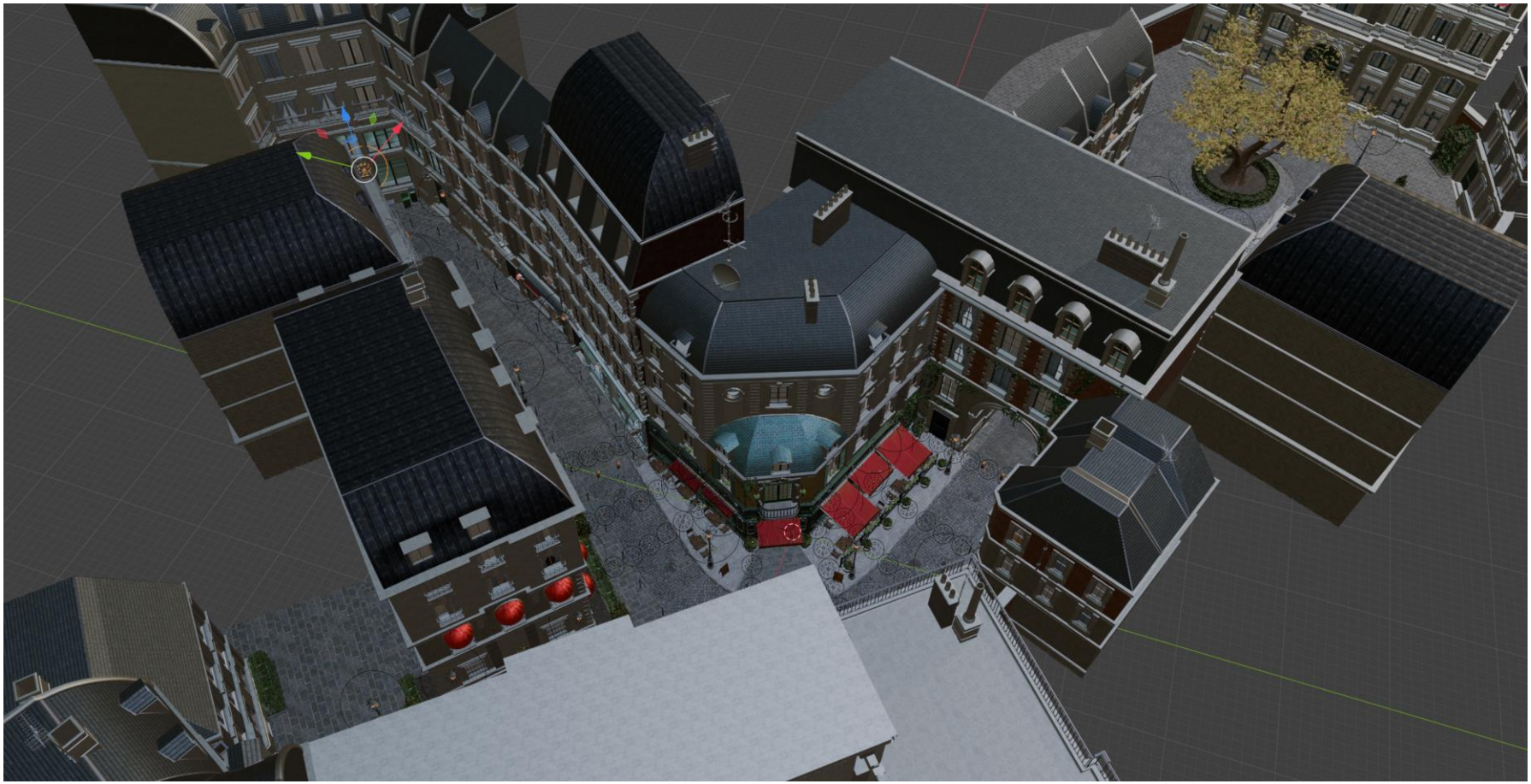
HW3보고서

〔2024.12.23〕

구현 항목	구현 여부
(방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전	O
(방법-2) deferred rendering 방식으로 변경한 버전	O
(방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전	O
(방법-4) stencil test와 depth test를 고려한 최적화 버전	O

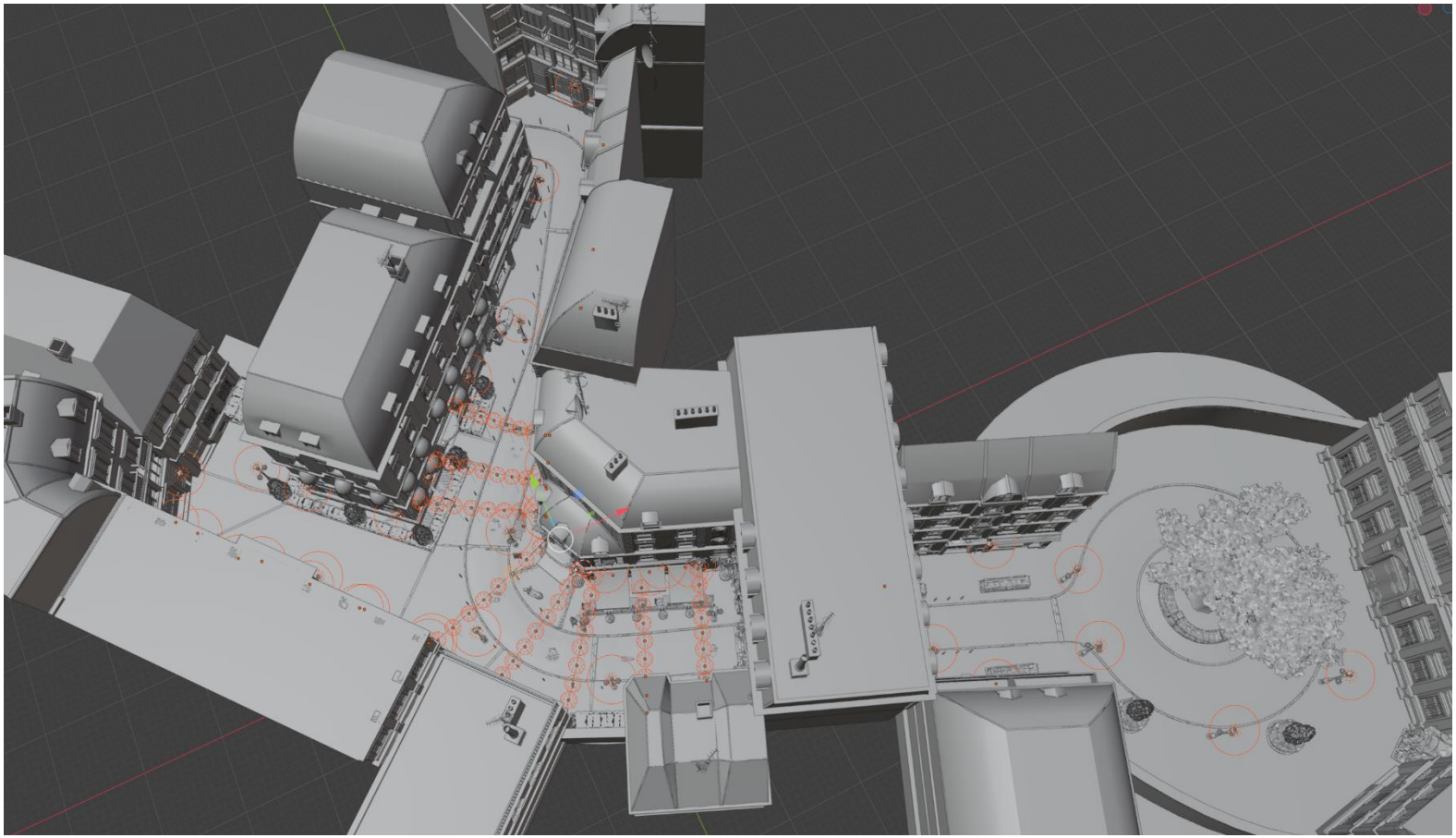
조작키	구현 여부
w / s	앞 / 뒤 이동
a / d	좌 / 우 이동
q / e	상 / 하 이동
t	렌더링 모드 스위칭
1, 2, 3, 4, 5	View 1 ~ 5 로 이동
마우스 클릭 후 드래그	상하좌우 회전

- **BistroExterior**



- BistroExterior data를 glTF 포맷으로 가공하여 사용하였다.

- **BistroExterior**

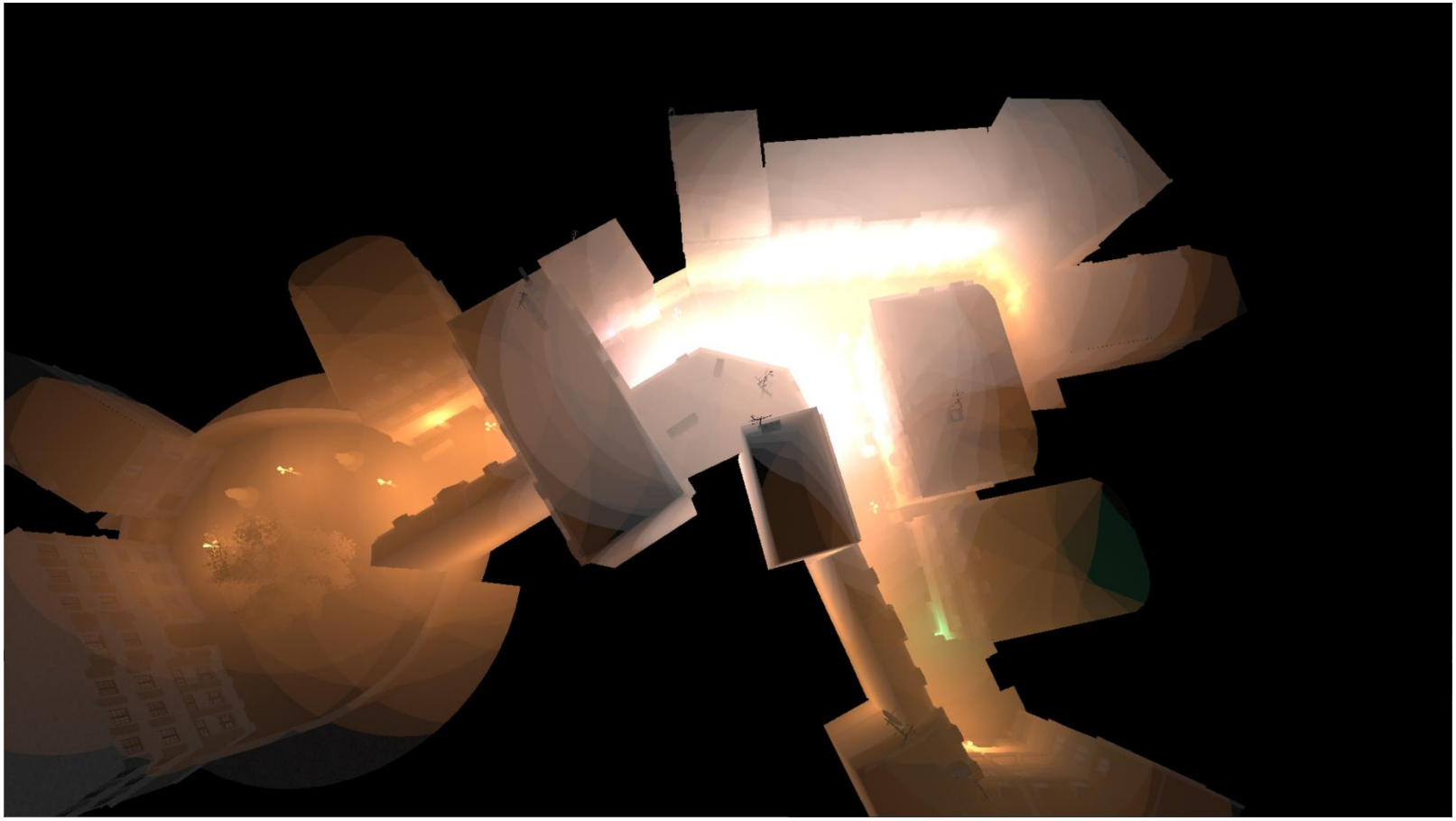


- Blender 툴을 이용해 위의 사진과 같이 광원을 직접 추가하여 glTF 파일로 export하였다. (사진 상의 주황색 원)

- **BistroExterior**

데이터 구성	개수
삼각형	2,829,226
점광원	110

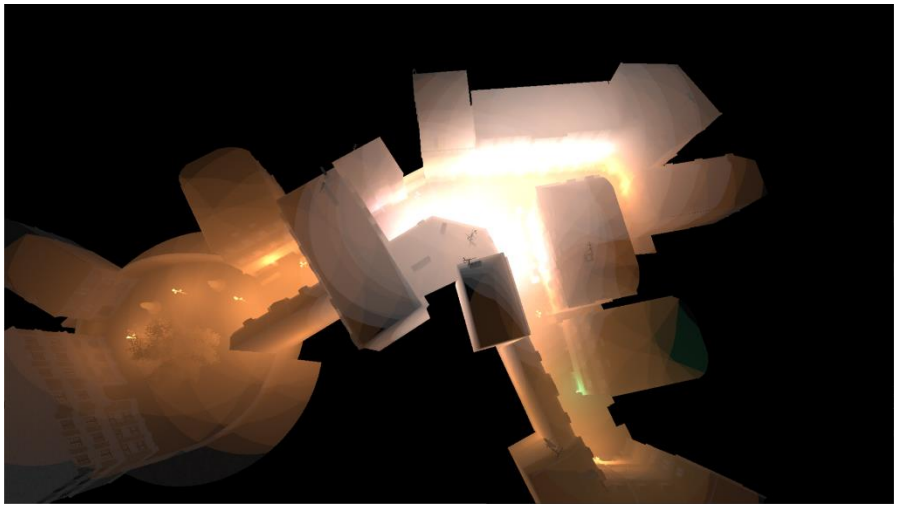
- **(방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전**



- 위의 사진은 각 점 광원의 radiance 나타낸 것이다.
  (사진에서 경계가 제대로 보일 수 있도록 attenuation이 5/256일 때만 렌더링)

- **(방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전**

광원 경계 제한 O

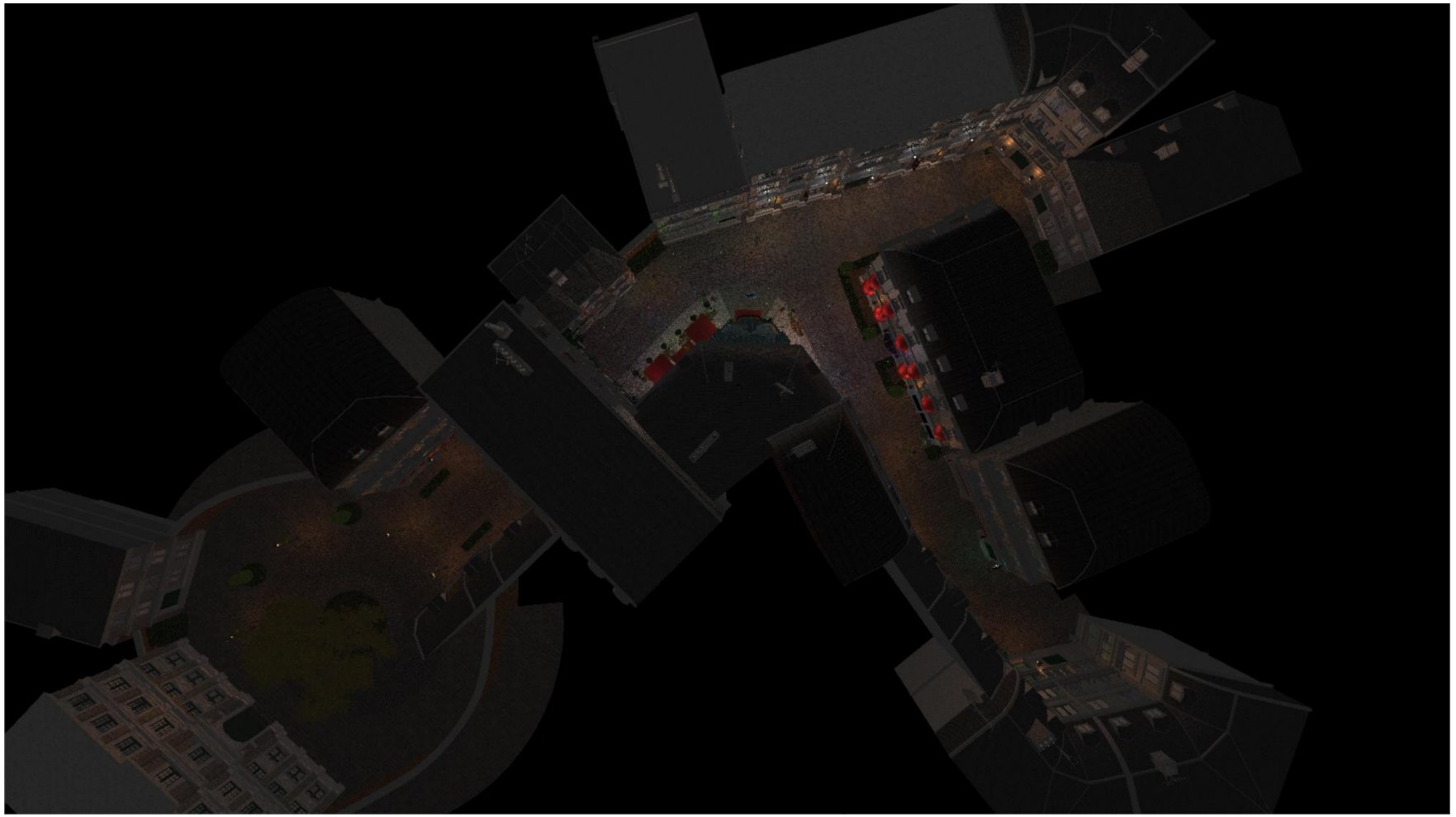


광원 경계 제한 X

- (방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전

View 1

- (방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전



View 2

- (방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전

View 3

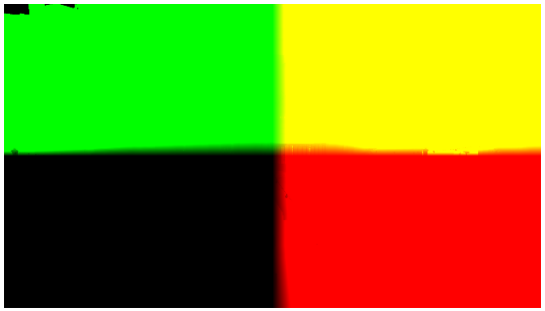
- (방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전

View 4

- (방법-1) forward rendering 코드에서 지역 점 광원으로 광원만 변경한 버전

View 5

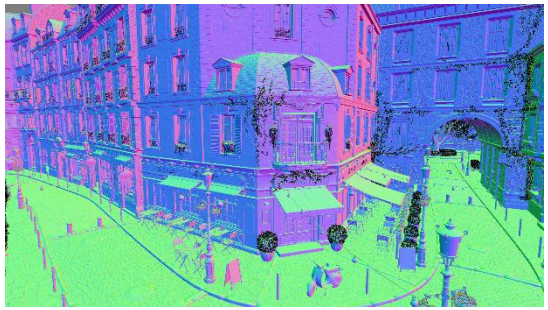
- **(방법-2) deferred rendering 방식으로 변경한 버전**



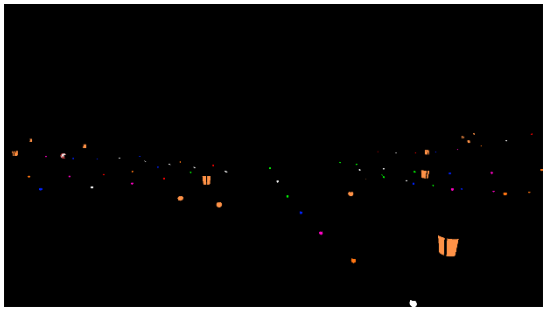
Position

MetallicRoughness

Albedo



Normal



Emissive

- 위의 사진은 Geometry pass의 color attachments이다.

- **(방법-2) deferred rendering 방식으로 변경한 버전**

geometry_pass.frag

```
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gAlbedo;
layout (location = 2) out vec3 gNormal;
layout (location = 3) out vec3 gMetallicRoughness;
layout (location = 4) out vec3 gEmissive;

uniform sampler2D uAlbedoTexture;
uniform sampler2D uNormalTexture;
uniform sampler2D uMetallicRoughnessTexture;
uniform sampler2D uEmissiveTexture;
uniform vec2 uMetallicRoughnessFactors;
```

deferred_0.frag

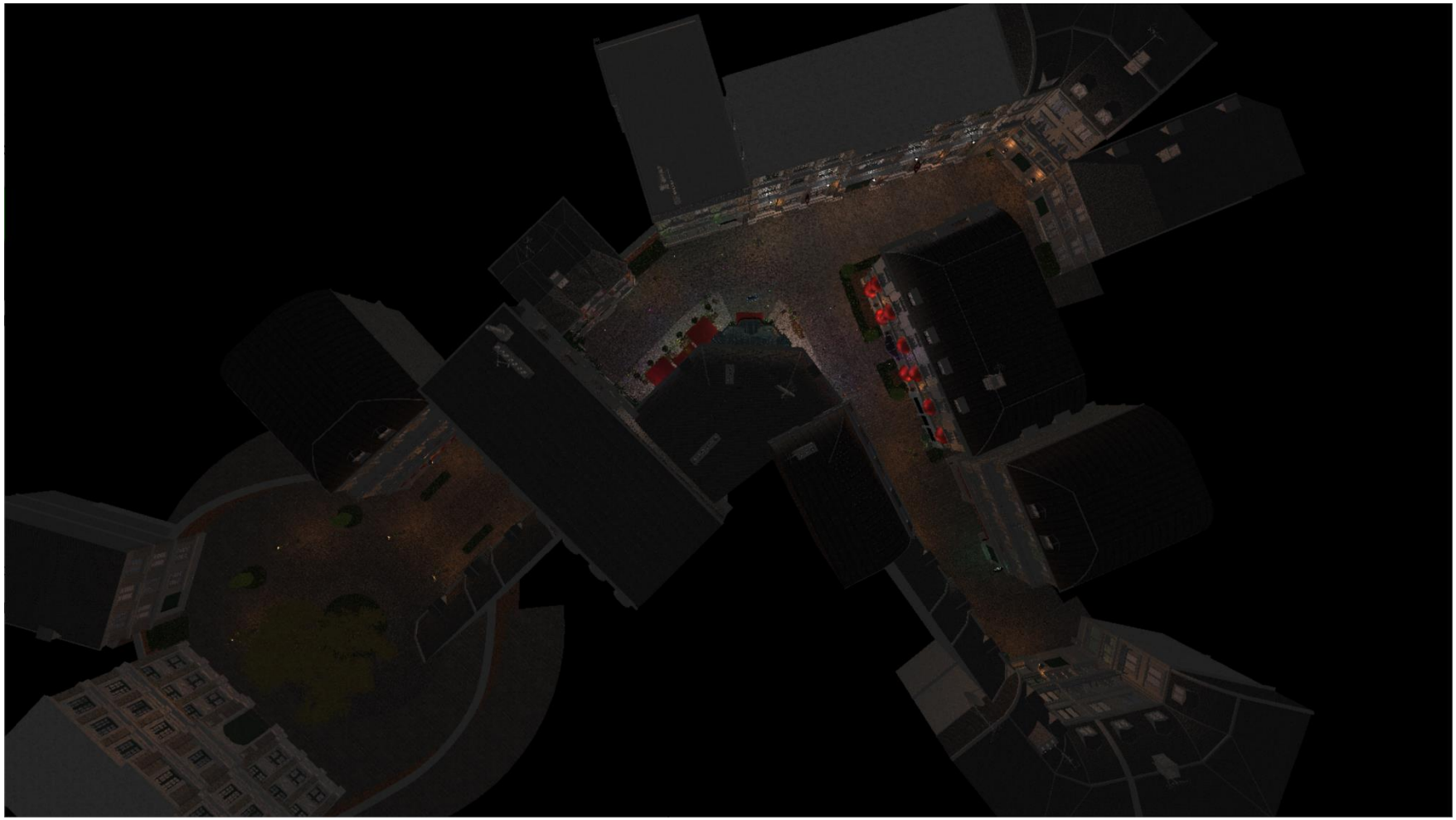
```
uniform LIGHT uLight[NUMBER_OF_LIGHTS_SUPPORTED];
uniform sampler2D uPositionTexture;
uniform sampler2D uAlbedoTexture;
uniform sampler2D uNormalTexture;
uniform sampler2D uMetallicRoughnessTexture;
uniform sampler2D uEmissiveTexture;
```

- Lighting pass에서는 위의 color attachments를 바탕으로 PBR 쉐이딩을 수행한다.

- (방법-2) deferred rendering 방식으로 변경한 버전

View 1

- (방법-2) deferred rendering 방식으로 변경한 버전



View 2

- (방법-2) deferred rendering 방식으로 변경한 버전

View 3

- (방법-2) deferred rendering 방식으로 변경한 버전

View 4

- (방법-2) deferred rendering 방식으로 변경한 버전

View 5

- **(방법-2) deferred rendering 방식으로 변경한 버전**

(방법-1)

(방법-2)

- Forward rendering 결과물과 거의 동일한 수준의 렌더링 결과가 나타났다.

- **(방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전**

Model.cpp

```
if (li.intensity <= 300)
{
    constant = 1.0f;
    linear = 0.7f;
    quadratic = 1.8f;
}
else
{
    constant = 1.0f;
    linear = 0.22f;
    quadratic = 0.20f;
}
light.lightAttenuationFactors[0] = constant;
light.lightAttenuationFactors[1] = linear;
light.lightAttenuationFactors[2] = quadratic;
light.lightAttenuationFactors[3] = 1.0f;

light.lightOn = 1;

float lightMax = std::fmaxf(std::fmaxf(light.color[0], light.color[1]), light.color[2]);
light.radius = (-linear + std::sqrtf(linear * linear - 4 * quadratic * (constant -
  (256.0 / 5.0) * lightMax))) / (2 * quadratic);
```

deferred_1.frag

```
struct LIGHT {
    vec4 position;
    vec3 color;
    bool lightOn;
    vec4 lightAttenuationFactors;
    float radius;
};
```

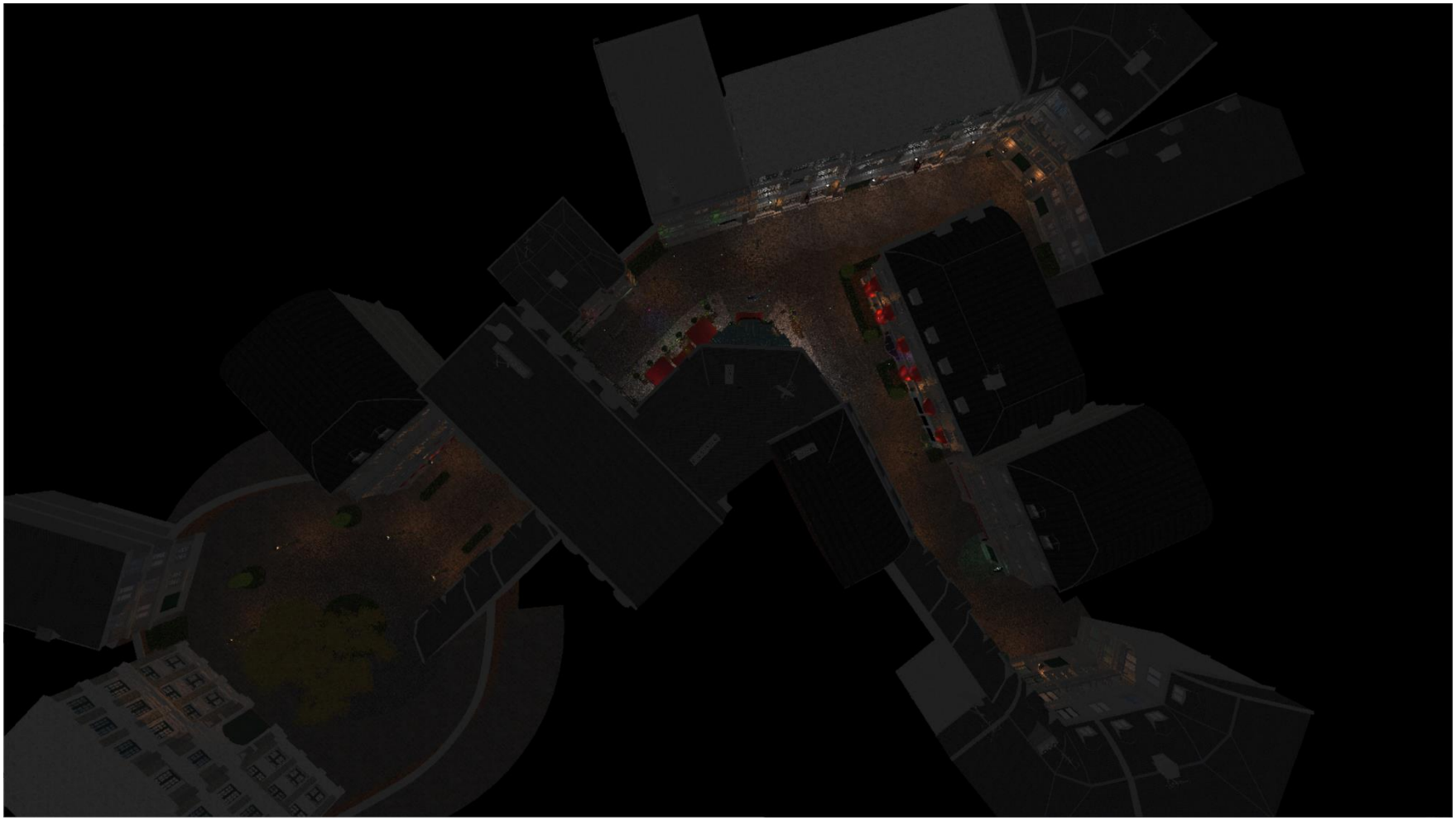
```
if (uLight[i].position.w != zerof) { // point light source
    lightDirectionEC = uLight[i].position.xyz - fragPos;
    float distance = length(lightDirectionEC);
    if (distance > uLight[i].radius)
        continue;
```

- Attenuation이 5/256 이하일 때는 조명 쉐이딩 계산을 수행하지 않도록 광원의 영향 범위를 계산하여 uniform 변수로 추가하였다.

- (방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전

View 1

- (방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전



View 2

- (방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전

View 3

- (방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전

View 4

- (방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전

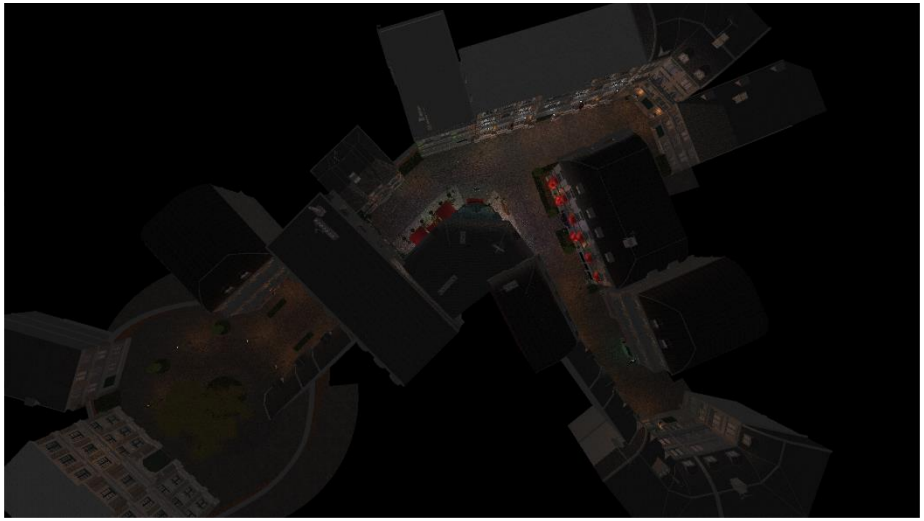
View 5

- (방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전

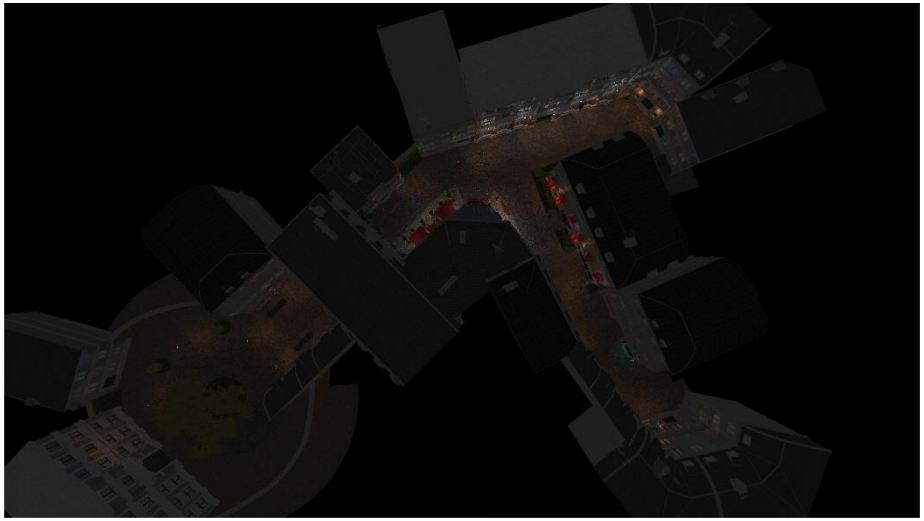
(방법-2)

(방법-3)

- **(방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전**



(방법-2)



(방법-3)

- **(방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전**

(방법-2)

(방법-3)

- (방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전

(방법-2)

(방법-3)

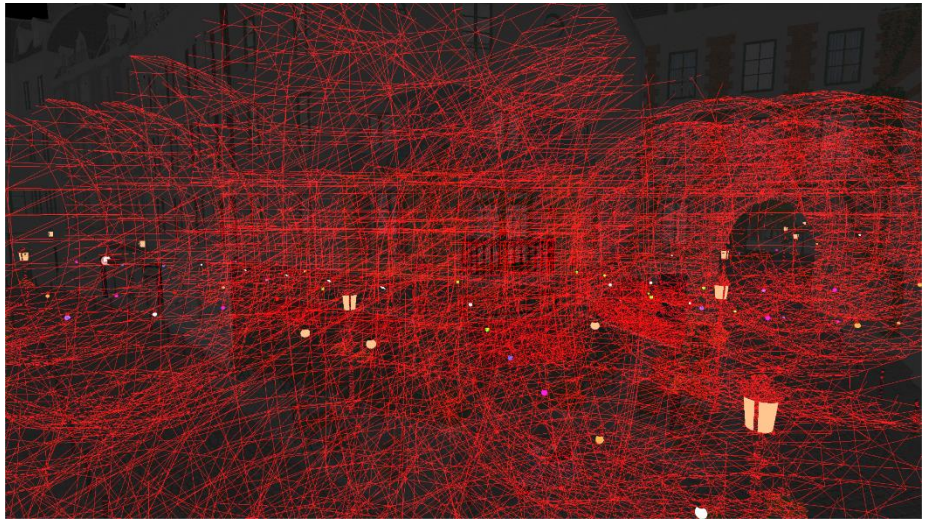
- **(방법-3) 각 광원에 해당하는 구의 반경을 고려하는 버전**

(방법-2)

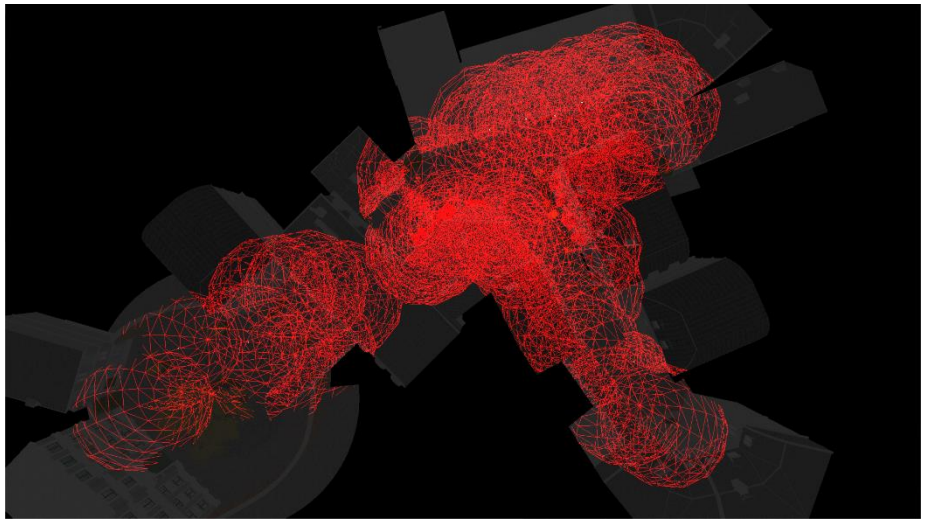
(방법-3)

- 5/256 이하의 attenuation factor에 대해서는 광원에 대한 쉐이딩을 수행하지 않았기 때문에 전체적으로 렌더링 결과물이 어두워졌다.

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**



View 1



View 2

- (방법-3)에서 설정한 광원의 반지름만큼의 크기를 갖는 구체를 렌더링하고, 그 과정에서 Stencil buffer에 광원 내/외부를 판단할 수 있는 값을 기록한다.

- (방법-4) stencil test와 depth test를 고려한 최적화 버전

DeferredShading.cpp

```
// Pass 1: Stencil Pass
glEnable(GL_STENCIL_TEST);

glEnable(GL_DEPTH_TEST);
glDisable(GL_CULL_FACE);

glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 0);

glDrawBuffer(GL_NONE);

glStencilOpSeparate(GL_BACK, GL_KEEP, GL_INCR_WRAP, GL_KEEP);
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_DECR_WRAP, GL_KEEP);

glUseProgram(hShaderPrograms[STENCIL]);

for (int i = 0; i < NUMBER_OF_LIGHT_SUPPORTED; i++)
{
    glm::mat4 modelMatrix(1.0f);
    modelMatrix = glm::translate(modelMatrix, glm::vec3(lights[i].position[0], lights
      [i].position[1], lights[i].position[2]));
    modelMatrix = glm::scale(modelMatrix, glm::vec3(lights[i].radius));
    modelViewMatrix = viewMatrix * modelMatrix;
    modelViewProjectionMatrix = projectionMatrix * modelViewMatrix;
    modelViewMatrixInvTrans = glm::inverseTranspose(glm::mat3(modelViewMatrix));

    glUniformMatrix4fv(hShaderLocations[STENCIL]["uModelViewProjectionMatrix"], 1,
      GL_FALSE, &modelViewProjectionMatrix[0][0]);

    sphere.drawWithoutTextures();
}
```

stencil_pass.vert

```
#version 430

uniform mat4 uModelViewProjectionMatrix;

layout (location = 0) in vec3 aPosition;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

void main(void) {
    gl_Position = uModelViewProjectionMatrix * vec4(aPosition, 1.0f);
}
```

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

deferred_2.frag

```
        if (localScaleFactor > zerof) {
            float NdotL = max(dot(normalEC, lightDirectionEC), 0.0f);
            vec3 halfwayEC = normalize(lightDirectionEC - normalize(fragPos));

            vec3 radiance = uLight.color * localScaleFactor;
            float NDF = DistributionGGX(normalEC, halfwayEC, roughness);
            float G   = GeometrySmith(normalEC, viewEC, lightDirectionEC, roughness);
            vec3 F    = fresnelSchlick(max(dot(halfwayEC, viewEC), 0.0), F0);

            vec3 numerator    = NDF * G * F;
            float denominator = 4.0 * max(dot(normalEC, viewEC), 0.0) * max(dot(normalEC,
              lightDirectionEC), 0.0) + 0.0001;
            vec3 specular = numerator / denominator;

            vec3 kS = F;
            vec3 kD = vec3(1.0) - kS;
            kD *= 1.0 - metallic;

            colorSum += (kD * albedo / PI + specular) * radiance * NdotL;
        }
    }
    finalColor = vec4(colorSum, 1.0f);
```

구를 렌더링하며 각 광원에 대한 쉐이딩 계산 수행

final_pass.frag

```
#version 430
in vec2 vTexCoord;

layout (location = 0) out vec4 finalColor;

uniform sampler2D uColorSum;
uniform sampler2D uAlbedoTexture;
uniform sampler2D uEmissiveTexture;

void main(void) {
    vec3 colorSum = texture(uColorSum, vTexCoord).rgb;
    vec3 ambient = vec3(0.03f) * texture(uAlbedoTexture, vTexCoord).rgb;
    vec3 emissive = texture(uEmissiveTexture, vTexCoord).rgb;

    colorSum += ambient + emissive;

    finalColor = vec4(pow(colorSum, vec3(1.0/2.2)), 1.0f);
}
```

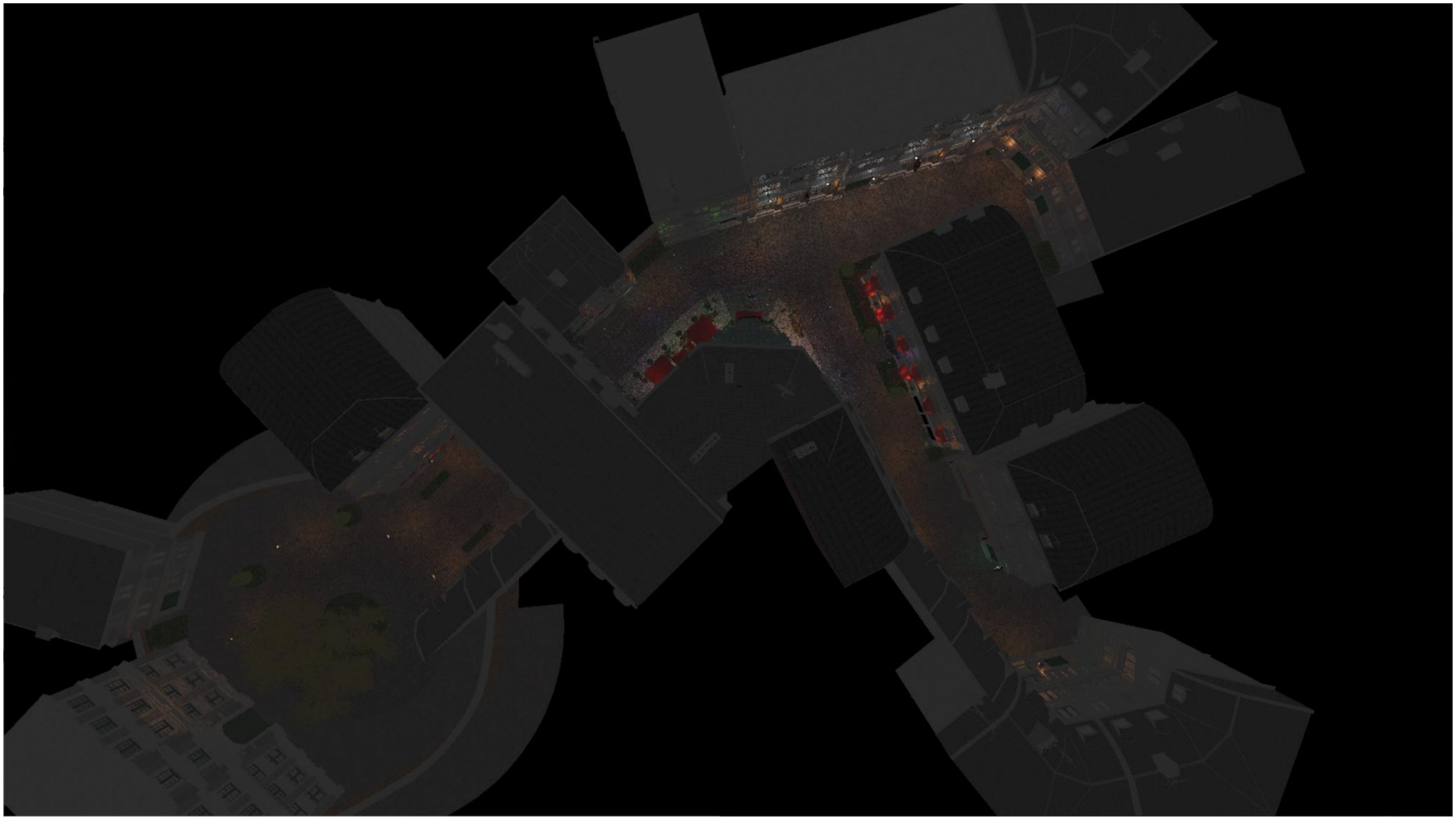
누적된 쉐이딩 결과 + α => 최종 색상 계산

- Stencil test를 통해 광원 내부에 있는 지점에 대해서만 쉐이딩 계산 후 색상을 누적한다.
- 최종 Pass에서는 ambient, emission 값을 더해주고 gamma correction을 수행한다.

- (방법-4) stencil test와 depth test를 고려한 최적화 버전

View 1

- (방법-4) stencil test와 depth test를 고려한 최적화 버전



View 2

- (방법-4) stencil test와 depth test를 고려한 최적화 버전

View 3

- (방법-4) stencil test와 depth test를 고려한 최적화 버전

View 4

- (방법-4) stencil test와 depth test를 고려한 최적화 버전

View 5

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

(방법-2)

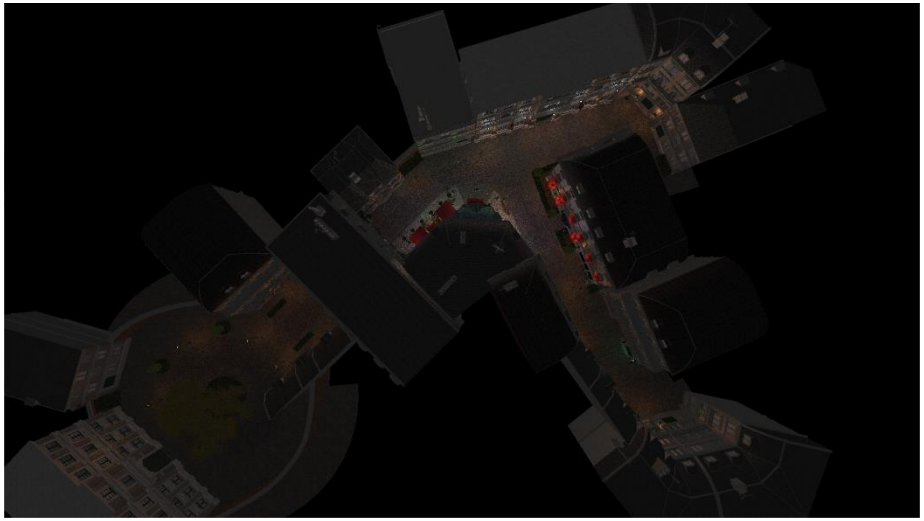
(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

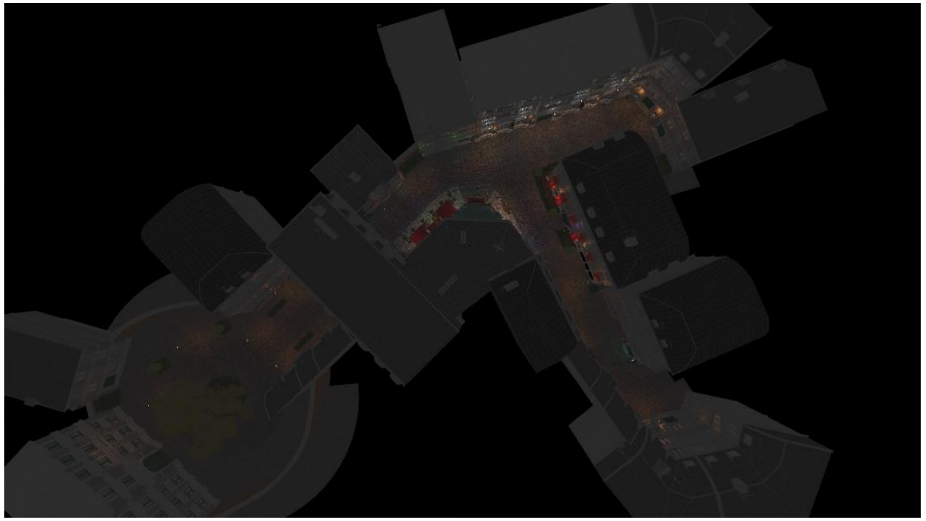
(방법-3)

(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

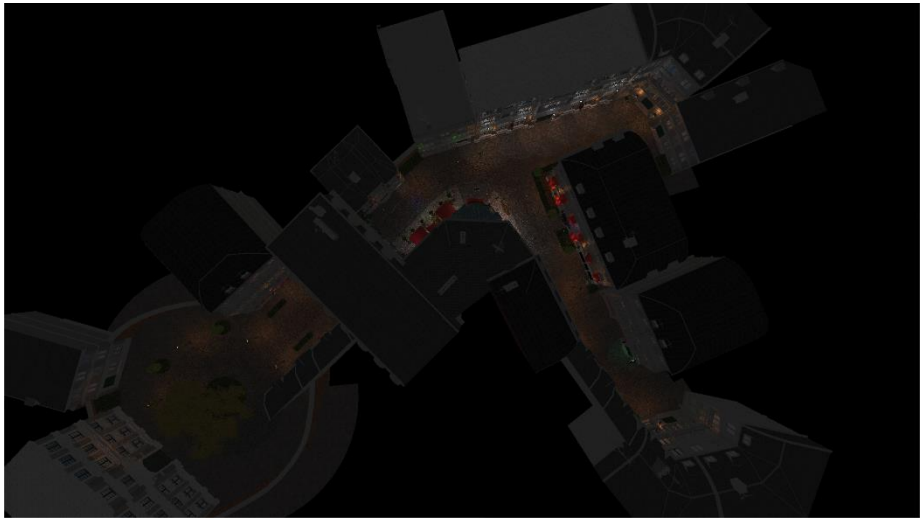


(방법-2)

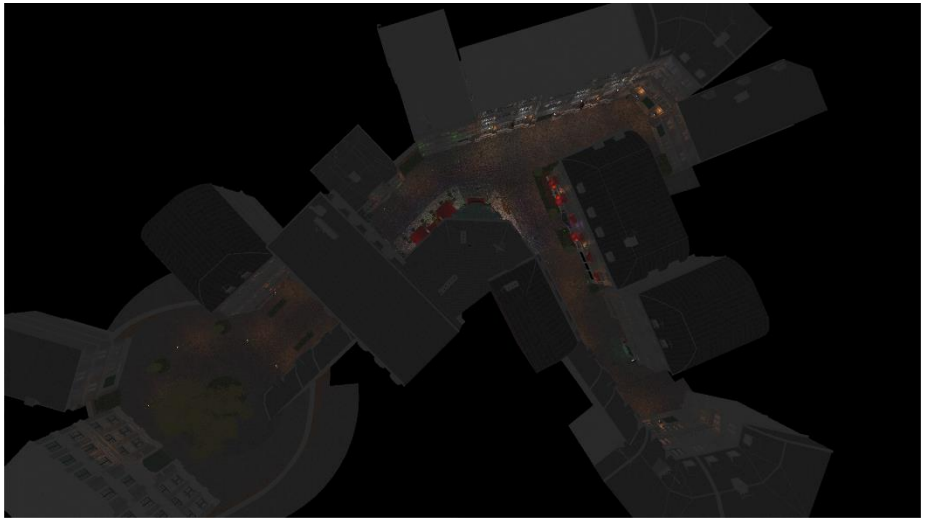


(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**



(방법-3)



(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

(방법-2)

(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

(방법-3)

(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

(방법-2)

(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

(방법-3)

(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

(방법-2)

(방법-4)

- **(방법-4) stencil test와 depth test를 고려한 최적화 버전**

(방법-3)

(방법-4)

- 전체적으로 (방법-2)와 (방법-3)이 혼합된 듯한 렌더링 결과물이 나타났다.

- Setting

하드웨어	상세
GPU	NVIDIA GeForce RTX 3080 TI
CPU	AMD Ryzen 9 5900X
메모리	32GB
해상도	1920 * 1080

```
Current Shader Program : BASIC
FPS: 102.16
FPS: 49.21
FPS: 49.12
FPS: 49.31
FPS: 49.12
FPS: 49.21
FPS: 49.07
FPS: 49.16
Current Shader Program : DEFERRED_0
FPS: 115.88
FPS: 156.69
FPS: 158.21
FPS: 157.06
FPS: 157.06
FPS: 157.21
Current Shader Program : DEFERRED_1
FPS: 181.27
FPS: 186.25
FPS: 186.81
FPS: 187.06
FPS: 184.45
FPS: 188.43
Current Shader Program : DEFERRED_2
FPS: 211.16
FPS: 226.55
FPS: 226.77
FPS: 226.77
FPS: 226.77
FPS: 226.10
```

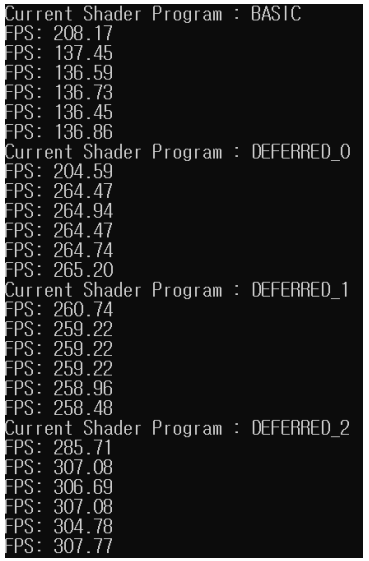
- FPS 비교(LIGHTS = 110)

방법	평균 FPS
(방법-1)	49.196
(방법-2)	157.246
(방법-3)	184.6
(방법-4)	226.592

* OpenGL vsync로 인해 모니터의 주사율에 맞게 프레임이 고정되는 현상이 있어 wglext 를 이용해 vsync를 해제하고 측정함.

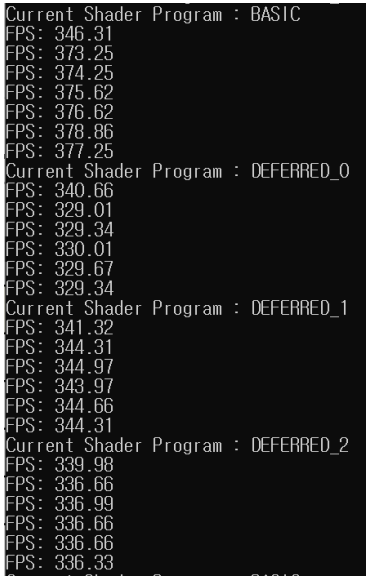
- FPS 비교(LIGHTS = 50)

방법	평균 FPS
(방법-1)	136.816
(방법-2)	264.764
(방법-3)	259.472
(방법-4)	302.268



- FPS 비교(LIGHTS = 10)

방법	평균 FPS
(방법-1)	375.72
(방법-2)	329.474
(방법-3)	343.914
(방법-4)	337.39



* OpenGL vsync로 인해 모니터의 주사율에 맞게 프레임이 고정되는 현상이 있어 wglext 를 이용해 vsync를 해제하고 측정함.

(LIGHTS = 110)

방법	평균 FPS
(방법-1)	49.196
(방법-2)	157.246
(방법-3)	184.6
(방법-4)	226.592

- (방법-1): Depth Test가 fragment shader 이후에 수행됨에 따라 렌더링 지점과 관련 없는 삼각형까지 광원에 대한 쉐이딩을 수행하게 되어 광원의 개수가 많아질수록 성능이 떨어진다. 실험에서는 가장 좋지 못한 성능을 보였다.
- (방법-2): 화면에 렌더링되는 지점에 대해서만 광원에 대한 쉐이딩을 수행하지만, 직접적으로 영향을 주지 않는 광원에 대해서도 계산을 수행하므로 비교적 성능이 좋지 않다.
- (방법-3): attenuation이 작을 것으로 판단되는 경우에는 아예 광원에 대한 쉐이딩을 수행하지 않도록 하여 (방법-1)보다 성능이 좋아졌다. 하지만, 분기로 인해 GPU 병렬성이 떨어지게 되어 큰 성능 향상을 이루진 못했다.
- (방법-4): 광원 영향 범위의 크기를 갖는 구체를 렌더링하고, stencil test를 이용해 해당 지점에만 광원에 대한 쉐이딩을 수행함으로써 (방법-3)의 병렬성 문제를 해결하였다. 이를 통해 가장 높은 성능을 보이며, 가장 성능이 좋지 못한 (방법-1)에 비해서는 4.5배의 향상을 이루었다.

(LIGHTS = 50)

방법	평균 FPS
(방법-1)	136.816
(방법-2)	264.764
(방법-3)	259.472
(방법-4)	302.268

(LIGHTS = 10)

방법	평균 FPS
(방법-1)	375.72
(방법-2)	329.474
(방법-3)	343.914
(방법-4)	337.39

- 하지만 광원이 적을 때(LIGHTS = 10)에는 오히려 렌더링을 한 번만 수행하는 (방법-1)의 경우가 가장 좋은 성능을 보였다. 그 다음으로는 렌더링을 2번 수행하며 비교적 최적화된 (방법-3), 그리고 렌더링을 3번 수행하지만 GPU 병렬성이 좋은 (방법-4), 마지막으로 렌더링을 2번 수행하지만 최적화된 요소가 없는 (방법-2) 순으로 좋은 성능을 보였다.

- 광원이 적절하게 많을 때에는 (방법-1)이 가장 좋지 않은 성능을 보이고 (방법-4)가 가장 좋은 성능을 보였지만, (방법-2)와 (방법-3)의 성능이 큰 차이를 보이지 않았고, 오히려 (방법-3)의 성능이 더 떨어지는 결과가 나타나기도 했다. 이는 분기로 인해 GPU 병렬성이 떨어져 오히려 성능이 하락한 것으로 해석된다.

- 카메라 이동/회전 기능 추가 : w/a/s/d/q/e + 마우스 드래그 / Camera.hpp
- 렌더링 모드 스위칭 기능 추가: t
- glTF 로딩 & 렌더링 기능 추가: Model.cpp, Model.h
- Phong Shading -> PBR 쉐이딩 변경