

Graphics Assn #4

Introduction

20160785 양준하, 20160463 성해빈

이번 과제는 지난 Assn3을 바탕으로 하여 둘 게임에 3D shading을 추가했다. Illumination Source, Reflection, Shading method 등을 구현하였다.

Background

Illumination Source

Directional light

directional light는 모든 위치에 같은 방향의 빛을 쏘며, 거리에 따른 빛의 세기 감소가 없다.

Point light

point light는 광원에 대한 방향의 빛을 쏘며, 거리에 따른 빛의 세기 감소가 있다.

Reflection

diffuse reflection

어떤 빛이 표면에 반사될 때, surface normal과 비슷한 각도로 들어온 빛일수록 세기가 강력하다. Illumination equation은 $I = I_p k_d (\bar{N} \cdot \bar{L})$ 이다. k_d 는 diffuse coefficient, \bar{N} 은 surface normal vector, \bar{L} 은 광원과 fragment 사이의 벡터이다.

specular reflection

어떤 빛이 표면에 반사될 때, surface normal을 법선으로 하는 반사각에 가까운 빛일수록 세기가 강력하다. 이 특성은 shiny한 표면일수록 더욱 강하게 나타난다. Illumination equation은 $I = I_p k_s O_s (\bar{R} \cdot \bar{V})^n$ 이다. k_s 는 specular coefficient, O_s 는 specular color, n 은 specular lobe coefficient, \bar{R} 은 \bar{L} 가 surface normal에 대해 reflect된 벡터이고, \bar{V} 는 eye와 fragment 사이의 벡터이다.

ambient reflection

ambient light는 모든 위치에 대해서 일정한 빛을 내게 만드는 것이다. 들어온 빛이 없어도 모든 위치에 대해 가짜 빛을 내게 만드는 방법이다. Illumination equation은 $I = I_p + I_a k_a$ 이다.

Shading method

Phong shading, Normal Mapping, Flat shading가 있는데, 모두 Implementation 항목에서 함께 설명하겠다.

Implementation

전반적인 구조는 Assn3과 동일하다. 매트릭스 연산은 모두 glm으로 수행하였다.

Develop Environment

개발환경은 Visual Studio 2017 C++, 주어진 freeglut와 glew, glm 0.9.9.2, winsdk 10.0.17134, opengl 4.6 이다. 빌드는 Release로 해야하는데, STL의 사용이 많아 debug로 빌드하면 각종 Assertion들이 난무해서 게임속도가 심각하게 느려져서 정상적인 플레이가 불가능 할 수도 있다.

Shading

이번 과제에서 구현하는 쉐이딩의 구조를 나눠보자면, 일단 폴리곤을 쉐이딩하는 shading method로 Flat 쉐이팅, Normal mapping, Phong 쉐이팅 3가지를 사용하였다. 각각의 3가지에 대해서 ambient, diffuse, specular의 reflection model을 모두 적용하였다. 광원은 과제에서 요구한대로 1개의 directional light와 2개의 point light가 존재한다.

Reflection model

앞서 말했듯 reflection model은 ambient, diffuse, specular를 모두 사용한다.

```
color = amb_r;
for(int i = 0; i < 3; i++)
{
    vec4 xx = light[i].pos[3] == 0 ? light[i].pos : light[i].pos - vpos;
    vec3 L = normalize((xx).xyz); //vector of light
    vec3 E = normalize(-vpos.xyz); //vector of eye (since eye is in origin)
    vec3 R = normalize(-reflect(L,vnormal));

    float katt = light[i].pos[3] == 0 ? 1 : 5.0/length(light[i].pos - vpos);

    vec4 dif_r = vicolor*light[i].diffuse * katt * max(dot(vnormal,L), 0.0); // NL is negative : backside
    vec4 spc_r = vicolor*light[i].specular * katt * pow(max(dot(R,E), 0.0), 0.8);
    color += clamp(dif_r, 0.0, 1.0) + clamp(spc_r, 0.0, 0.5);
}

color = clamp(color, 0.0, 1.0);
color[3] = vicolor[3];
```

다음 코드는 뒤에 설명할 여러 glsl 코드들에서 (거의) 공통적으로 나타나는데, 폴리곤 내부의 fragment들에 대한 normal의 계산이 일단 이루어졌다고 가정하고 설명을 하겠다.

사용되는 보조벡터가 3가지가 있는데, L은 현재 fragment에서 광원으로의 벡터이다. E는 현재 fragment에서 눈(eye)로의 벡터이다 R은 specular shading을 위해 사용되는 것으로, 이론적 배경에서 설명하였다.

katt는 거리에 따른 밝기계수로, 원래 분모에 2차항이나 3차항까지 넣을 수도 있으나 현재는 간단하게 1차항의 length를 바로 사용하였다.

그 다음 diffuse를 통한 계산결과가 dif_r인데, `vicolor * light[i].diffuse * katt` 는 단순히 색상의 베이스와 밝기를 지정해주는 계수이다. 중요한 부분은 `max(dot(vnormal,L), 0.0)` 인데, diffuse의 핵심인 반사면의 법선과 광원 ray의 내적을 해서 반사율을 계산한다. 이 때 max가 있는 이유는 음수를 거르기 위해서인데, 음수가 나오는 것은 법선과 광선이 반대방향이라는 것이고, 다른말로 빛으로 부터 반대편이 되는 것이다. 따라서 그냥 0으로

두면 된다.

spc_r도 비슷하다. `pow(max(dot(R,E), 0.0), 0.8)` 는 sin의 제곱에 대한 근사이며, 0.8은 얼마나 물체가 번들번들한지에 대한(shinyness) 파라미터다.

여기서 diffuse와 specular의 계수가 `light[i]`에 들어있어 의아할수도 있지만, 이는 게임이 한정적이므로 구현의 편의상 그런 material 속성을 물체가 아니라 광원에 귀속시킨 것으로 추후에 해결해야할 점이다.

Shading Polygons

사용되는 opengl program(ver, frag shader의 조합)은 3가지로, 각각 Flat, Normal mapping, Phong에 대응한다. 각각은 폴리곤을 쉐이딩하는 고유한 방법들이지만, 위에서 설명한 reflection model은 모두 똑같이 계산한다.

Phong

제일 간단한 Phong의 경우 버텍스가 각각 갖고있는 노말을 interpolation하고, interpolation된 normal을 각 fragment들이 위에서 설명한 reflection model에 그대로 대입만 하면 끝난다.

이 때 매우 중요한 이슈가 있는데, 일단 첫번째는 더이상 projection matrix와 model-view matrix를 같이 관리하면 안된다는 것이다. 당연하지만 광원처리는 3D 세계에서 일어나야하기 때문에 projection은 제외하고 model-view만 적용된 상태에서 계산해야한다. 따라서 모든 버텍스 쉐이더는 이제 uniform으로 projection과 modelview를 따로 받는다.

다음으로 중요한 이슈는 버텍스의 노말도 transform되어야한다는 것이다. rotation만 적용하려면 upper 3x3 matrix를 따로 빼올수도 있지만 단순히 trasnform될 노말벡터의 w값을 0으로 주는것으로 해결된다.

이런 주의점이 모두 적용된 Phong shading의 버텍스 쉐이더는 다음과 같다.

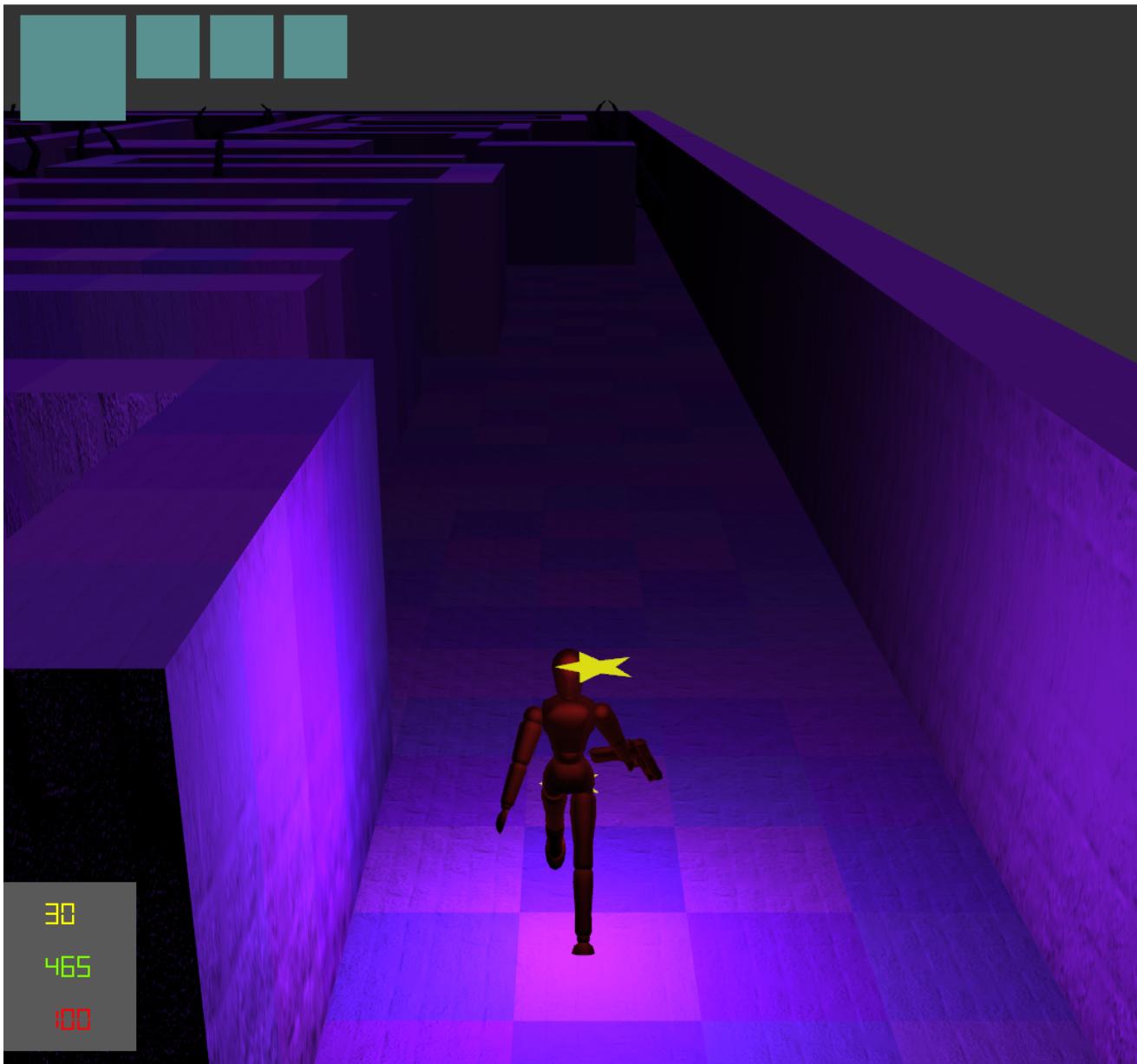
```
#version 330 core
in vec4 position;
in vec3 normal;

uniform mat4 projection;
uniform mat4 modelview;
uniform mat4 normaltrans;

out vec3 vnormal;
out vec4 vpos;

void main()
{
    vnormal = normalize(normaltrans * vec4(normal, 0.0)).xyz;
    vpos = modelview * position;
    gl_Position = projection * vpos;
}
```

노말의 transform을 위해 적용될 매트릭스는 그냥 modelview를 그대로 써도 되지만, 혹시 몰라서 따로 normaltrans라는 매트릭스를 받게 했다. 위에서 설명했듯 실제 rasterize될 gl_Position은 projection까지 적용되지만, 쉐이딩에서 사용될 3D 좌표는 model과 view transform까지만 적용하고 넘긴다(vpos)



Normal Mapping

Normal mapping의 경우에는 fragment의 노말을 직접 노말맵 텍스쳐에서 샘플링해온다는 것이 핵심이다. 그럼 일단 텍스쳐 좌표(UV)가 필요하지만, 여전히 버텍스의 노말도 필요하다. 왜냐하면 텍스쳐가 입혀질 plane이 어떻게 누워있는지도 당연히 고려해야하기 때문이다.

일단 normal mapping의 버텍스 쉐이더는 다음과 같다.

```
(중략)
in vec2 tex;
out vec2 uv;
void main()
{
    vnormal = normalize(normal);
    uv = tex;
    vpos = modelview * position;
    gl_Position = projection * vpos;

}
```

거의 똑같지만 vnormal은 더 이상 트랜스폼되지 않는다. (순수한 모델공간을 유지해야하는데 그 이유는 밑에서 설명) 그리고 물론 uv도 out된다.

Normal mapping의 fragment 쉐이더는 다음과 같다.

```
uniform sampler2D texsam;

mat3 ss(vec3 v)
{
    mat3 m;
    m[0] = vec3(0, -v[2], v[1]);
    m[1] = vec3(v[2], 0, -v[0]);
    m[2] = vec3(-v[1], v[0], 0);
    return m;
}
void main()
{
    color = ambient;

    vec3 samnorm = normalize(texture(texsam, uv).xyz * 2 - vec3(1,1,1));
    vec3 zv = vec3(0,0,-1);
    vec3 cv = cross(zv, vnormal);

    mat3 sm = ss(cv);
    mat3 R = mat3(1.0) + sm + sm*sm*(1.0/(1+dot(zv, vnormal)+0.001));

    samnorm = mat3(normaltrans)*R*samnorm;
}

(후략)
```

일단 텍스쳐가 눕혀질 평면의 (모델공간상) 법선은 당연히 vnormal이 된다. 하지만 텍스쳐가 원래 z축을 보고 누워 있었기에 (텍스쳐맵을 열어보면 blue, 즉 z가 우세하다) z축을 vnormal로 회전시키는 회전행렬을 직접 구해야한다. 계산된 회전행렬은 R이고, 실제 texture()함수를 통해 샘플링된 samnorm에 곱한후 원래 필요한 normaltrans까지 곱하면 fragment를 위한 최종 노말벡터를 얻게 된다. 그 이후는 똑같다.



Flat

Flat은 복잡한 쉐이딩 계산을 삼각형에 대해 딱 한번만 계산하므로, 불연속성을 대가로 매우 빠른 속도를 얻는다. 일단 fragment 쉐이더에 있던 계산은 모두 버텍스 쉐이더로 넘어오고, fragment shader는 버텍스 쉐이더의 결과를 그대로 내보내기만 하면된다.

버텍스 쉐이더에서 계산해야하는 것은 노말벡터로 diffuse와 specular를 계산하는 똑같은 절차지만, 이 때 쓰는 노말벡터가 interpolation된 것이 아니라 그냥 버텍스 노말 자체를 바로 쓴다는 것이 차이다. 또한 out color에 'flat' 키워드를 붙혔는데, 이 경우에 계산이 3개의 버텍스에서 각각 이뤄진 후 색깔이 interpolation되는 것이 아니라 셋 중에 하나(대표 버텍스)에서만 계산되고 모든 내부 fragment에 대해 똑같이 적용한다.

```

flat out vec4 ocolor;

void main()
{
    vec3 vnormal = normalize(normaltrans * vec4(fnormal, 0.0)).xyz;
    vec4 vpos = modelview * position;
    gl_Position = projection * vpos;

    (중략)
    ocolor[3] = vicolor[3];
}

```

추가기능 : 이 때 버텍스의 노말을 그냥 원래 메쉬에 있는 것을 쓸 수도 있지만, 좀 더 정확하게 하기 위해 실제 삼각형의 법선을 계산하여 사용하였다. 즉 메쉬를 로딩할때 삼각형을 이루는 변 3개중 2개를 골라 외적한 결과를 저장해놓고 vnormal이 아닌 fnormal에 attribute 설정을 해놓았다. 이 부분에 대한 코드는 CShaderManager::M_LoadMesh()에 구현되어 있으니 참고하면 된다.



Illumination Sources

실제 광원은 uniform으로 전달되는데, 다음과 같다

```
struct SLight
{
    vec4 pos;
    vec4 diffuse;
    vec4 specular;
};

uniform SLight light[3];
```

다시한번 말하지만 여기서 `diffuse`와 `specular`의 계수가 `light[i]`에 들어있어 의아할수도 있지만, 이는 게임이 한정적이므로 구현의 편의상 그런 `material` 속성을 물체가 아니라 광원에 귀속시킨 것으로 추후에 해결해야 할 점이다.

`pos`의 `w`값이 0이면 directional light로 간주한다. 문서에서 요구한 것처럼 1개의 약한 directional light와 2개의 플레이어를 따라다니는 point light가 있는데, **추가기능 : 원래 고정되어야하지만 light source를 시간에 따라 조금씩 움직이게 만들었다. directional light의 경우에는 방향이 조금씩 돌아가며, point light 2개는 플레이어 주위를 천천히 공전한다. 또한 그 point의 위치를 노란 별로 출력하였다.**

CShaderManager

`CShaderManager`는 셰이더와 폴리곤, 메쉬, 텍스쳐 등 GPU 이용을 전반적으로 관리하는 Singleton클래스로, 처음에 한번 초기화 되며 그 이후에 `CGraphics`에서 종종 참조된다. 자세한 설명은 `Assn3`에 나와있으니 설명하기 바란다. 차이점은 `obj`를 로드하여 각각 `submesh`들을 하나하나의 폴리곤으로 `VBO`와 `VBA`를 생성해서 관리하게 했다는 점이다. 자세한 코드는 생략한다.

Shaders

실제로 우리가 `assn4`에서 사용한 vertex/fragment 쉐이더의 조합은 3개로 정확히 Phong, normal mapping, flat에 대응한다.

- Phong : 프로그램은 `prg3`이며 `ver_shd.glsl`과 `frag_shd.glsl`을 사용한다
- Normal Mapping : 프로그램은 `prg4`이며 `ver2.glsl`과 `frag_shd_map.glsl`을 사용한다
- Flat : 프로그램은 `prg5`이며 `verp.glsl`과 `frag_shdp.glsl`을 사용한다.

Game

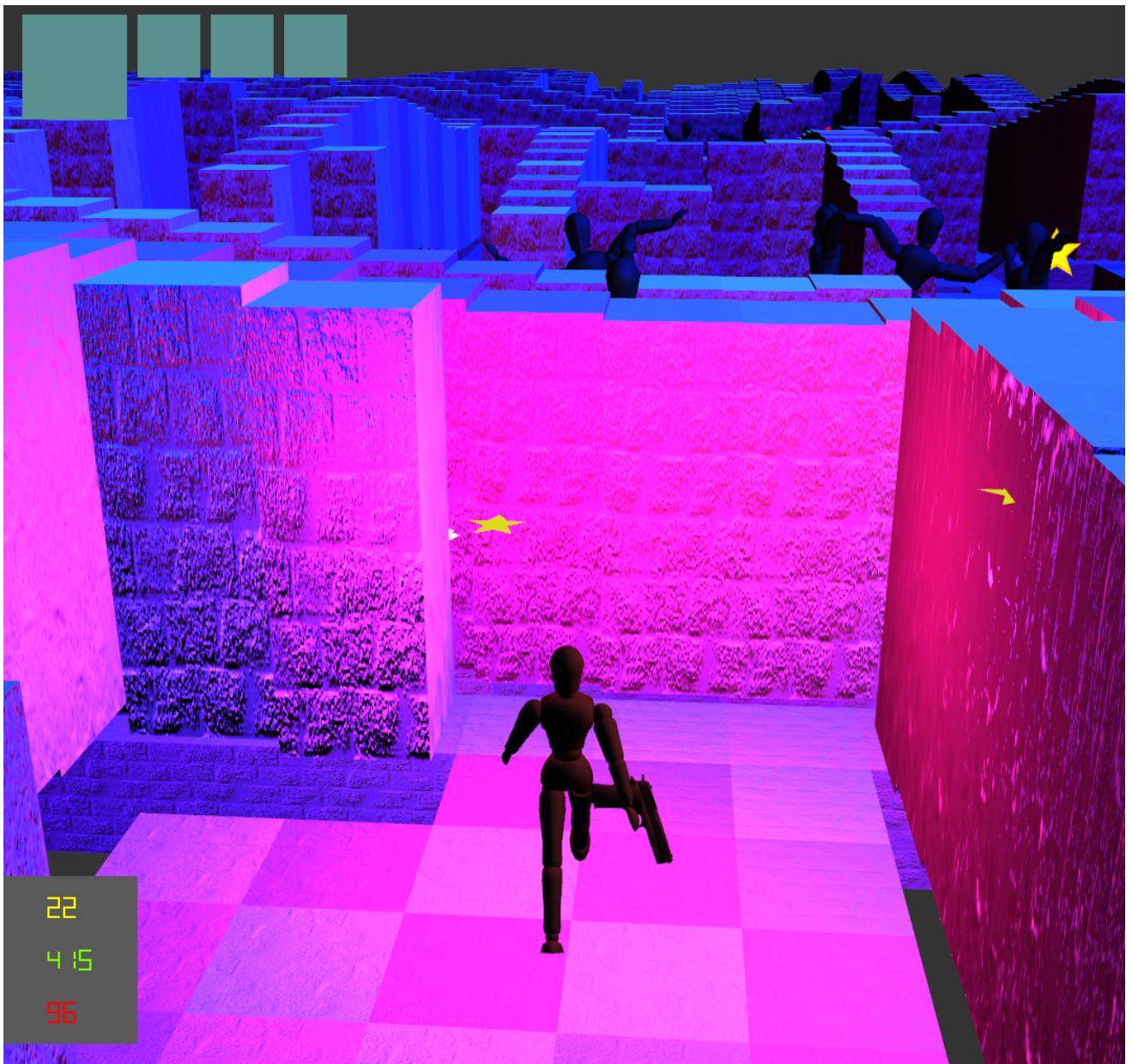
게임플레이 자체는 `Assn3`과 거의 동일하지만 쉐이딩과 관련된 두 가지 추가 조작이 생겼다.

Shading Switch

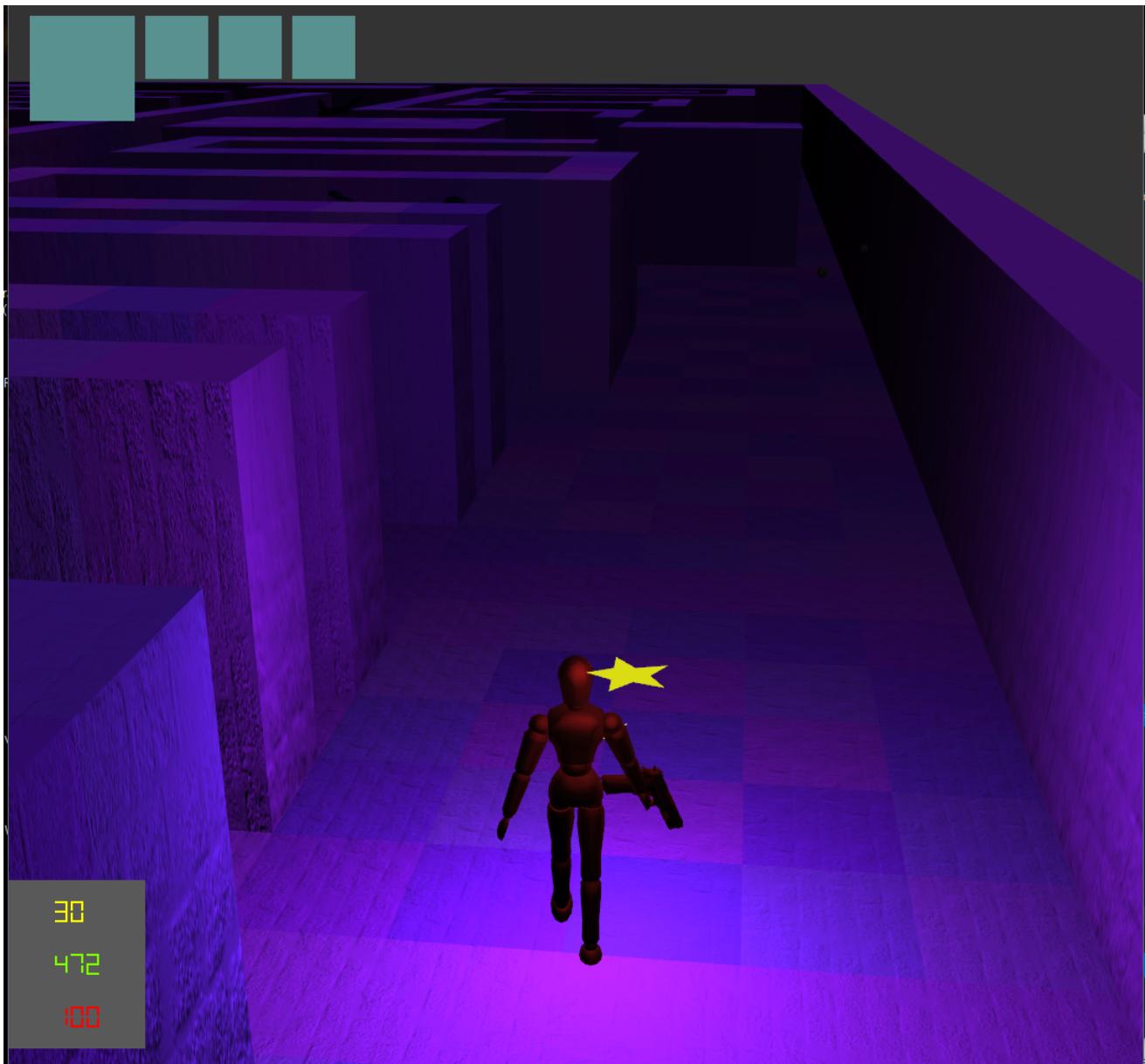
`f`키를 누르면 (한/영 조심) Phong + normal mapping과 flat 쉐이딩을 전환할 수 있다. 전자의 경우 플레이어와 적은 Phong으로, 벽은 normal mapping으로, 아이템은 쉐이딩을 적용받지 않는다. 후자의 경우 아이템을 제외한 모든 오브젝트가 flat 쉐이딩을 적용받는다.

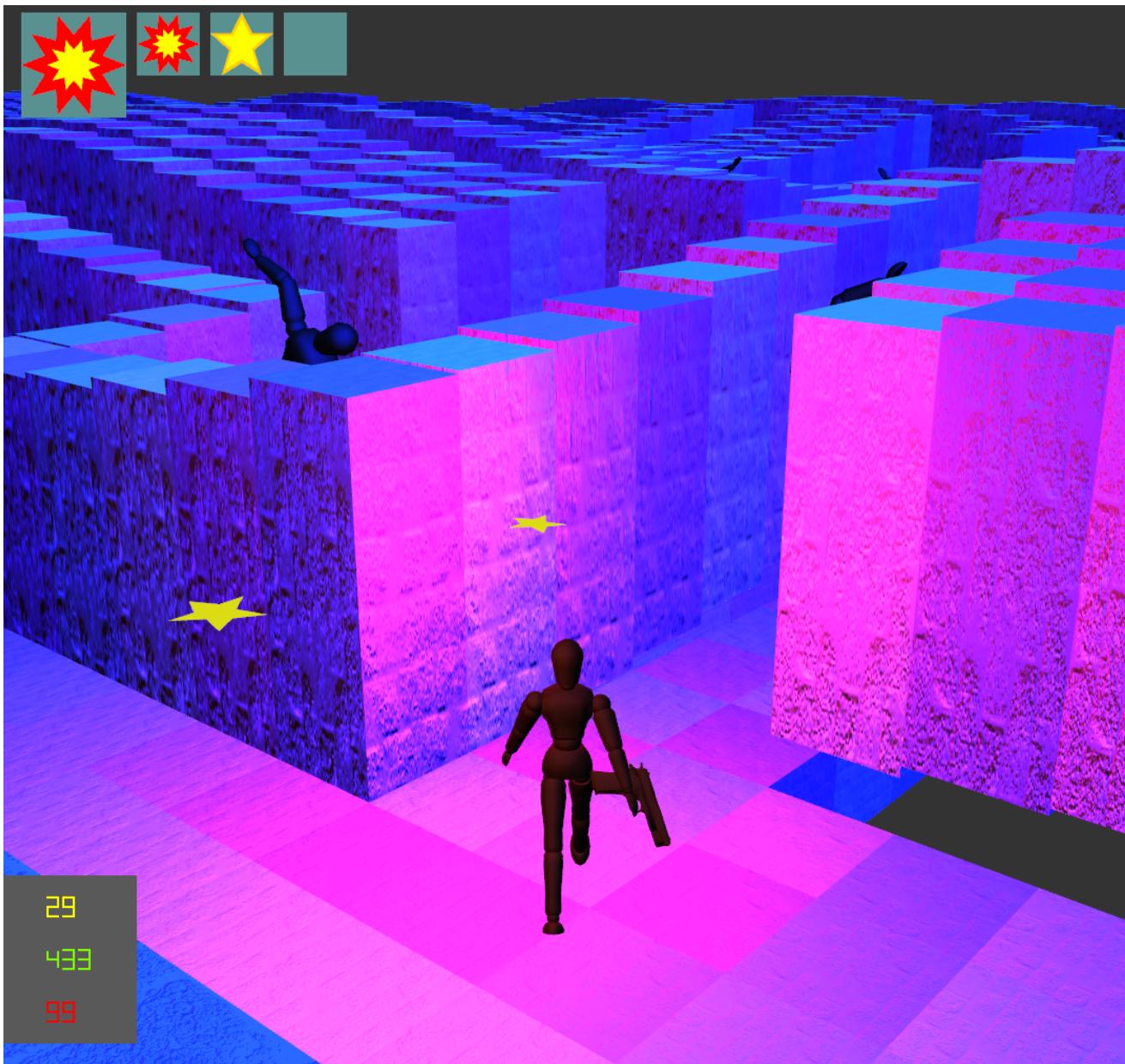
Party Mode

추가기능 : `p`키를 누르면 파티모드로 전환하는데, 원래 백색광이었던 광원이 화려한 색깔로 바뀌고 또 위치도 다이나믹하게 바뀐다. 추가로 벽들이 출렁출렁 움직인다.



Example





Discussion

Material Property

물체마다 diffuse, specular의 계수나 shininess 등을 각각 매겨 서로 다른 질감을 가진 것처럼 표현할 수도 있지만 아직 구현하지 않았다. 추후에 구현한다면 좀 더 쉐이딩이 실감나게 느껴질 것 같다.

Texture

현재도 Texture를 로딩해서 사용하고 있지만, 노말맵을 위해서만 쓰고 실제 RGB 맵핑은 하지 않는다. 할 수도 있었지만 쉐이딩과 같이 적용되면 제대로 작동하는지 헷갈릴 수도 있어서 일단 구현하지 않았다. 텍스쳐에 딱 맞는 노말맵이 지금처럼 있다면 언제라도 텍스쳐맵핑과 노말맵핑을 동시에 하여 실감나는 질감을 구현할 수 있을 것으로 기대한다.

Global Illumination

Global Illumination, 특히 그림자가 꼭 구현해보고 싶었는데 어려워 보여서 포기했다. 쉐이딩과 함께 광원효과를 책임지는 그림자는 나중에라도 꼭 구현해볼 것이다. 굴절이나 반사도 마찬가지다.

OpenGL Optimization

현재 프로그래밍하면서 OpenGL의 최적화가 부족하다는 것을 깨달았다. redundant한 bind를 여러번 한다든지, 아니면 버퍼를 낭비하고 있다든지, uniform 전달을 쓸데없이 많이한다든지 하는 것들이다. 소팅을 잘 하면 해결할 수 있을 것 같지만 복잡한 문제이다.

Conclusion

결론적으로 이번 어싸인에서는 3D 렌더링의 꽃인 쉐이딩을 다양한 방식으로 구현해보았다. 다양한 쉐이더를 사용하여 오브젝트마다 다른 쉐이딩 모델을 적용하였으며 쉽게 전환할수도 있게 했다.

Reference

전반적인 OpenGL 사용법과 여러 OpenGL 함수에 대해서는 다음과 같은 레퍼런스를 참고 했다.

[The OpenGL Utility Toolkit \(GLUT\) Programming Interface API Version 3](#)

[OpenGL® 4.5 Reference Pages](#)

[Lighthouse3d.com](#)

[opengl-tutorial](#)

쉐이딩은 OpenGL 튜토리얼을 두루 참조했으며, 특별히 퍼온 곳은 없다 bmp의 로딩에 대해서는 다음 소스코드를 사용하였다. [ogl!](#)

Obj로더는 좀 더 나은 것으로 바꿨다. 출처는 [tinyobjloader](#)이다.