CSE333: Computer Graphics

Assignment-04

Deepanshu Dabas, 2021249

Ans 1) Used lecture slide formula for calculating value beta and gamma,

Also made changes to add triangle class implementation in CMake
Necessary conditions were handled correctly using if else loops. Norn
were also set by creating constructor for same.

```
    Vector3D getNormal() const {return normal;}
    float getParameter() const {return t;}
    bool setParameter(const float par, const Object *obj);
    void setNormal(const Vector3D& n) {normal = n;}
    // Set New Normal
    bool didHit() const {return hit;}
```

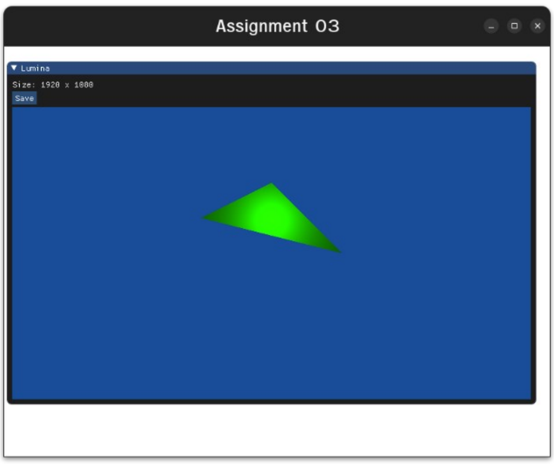
```
// triangle.cpp
#include "triangle.h"

bool Triangle::intersect(Ray &r) const
{
    // Utilize barycentric coordinates for ray-triangle test
    // e + td = f(u,v)
    Vector3D e = r.getOrigin();
    Vector3D d = r.getDirection();
    /*
        For a triangle with vertices a, b, and c intersection will occur when
        e + td= a+ beta(b-a) + gamma(c-a) , where beta and gamma are barycentric coordinates
    */
    glm::mat3 A(a[0] - b[0], a[0] - c[0], d[0],
                a[1] - b[1], a[1] - c[1], d[1],
                a[2] - b[2], a[2] - c[2], d[2]);
    /*     _                   _
          | xa-xb, xa-xc, xd |
       A= | ya-yb, ya-yc, yd |
          |_za-zb, za-zc, zd_|

             _                   _
            | xa-xe, xa-xc, xd |
       beta= | ya-ye, ya-yc, yd |
            |_za-ze, za-zc, zd_|
             -----------------
                   |A|
```

```
             _                   _
            | xa-xe, xa-xc, xd |
       beta= | ya-ye, ya-yc, yd |
            |_za-ze, za-zc, zd_|
             -----------------
                   |A|
              _                   _
             | xa-xb, xa-xe, xd |
       gamma= | ya-yb, ya-ye, yd |
             |_za-zb, za-ze, zd_|
              -----------------
                    |A|
             _                      _
            | xa-xb, xa-xc, xa-xe |
         t=  | ya-yb, ya-yc, ya-ye |
            |_za-zb, za-zc, za-ze_|
             -----------------
                    |A|
    */
    glm::mat3 B(a[0] - e[0], a[0] - c[0], d[0],
                a[1] - e[1], a[1] - c[1], d[1],
                a[2] - e[2], a[2] - c[2], d[2]);
    glm::mat3 C(a[0] - b[0], a[0] - e[0], d[0],
                a[1] - b[1], a[1] - e[1], d[1],
                a[2] - b[2], a[2] - e[2], d[2]);
    glm::mat3 D(a[0] - b[0], a[0] - c[0], a[0] - e[0],
                a[1] - b[1], a[1] - c[1], a[1] - e[1],
                a[2] - b[2], a[2] - c[2], a[2] - e[2]);
```

```cpp
    glm::mat3 D(a[0] - b[0], a[0] - c[0], a[0] - e[0],
                a[1] - b[1], a[1] - c[1], a[1] - e[1],
                a[2] - b[2], a[2] - c[2], a[2] - e[2]);

    double detA = glm::determinant(A);
    double beta = glm::determinant(B) / detA;
    double gamma = glm::determinant(C) / detA;
    double t = glm::determinant(D) / detA;
    if (beta <= 0 || gamma <= 0 || beta + gamma >= 1)
    {
        /*
            Intersection is inside the triangle iff
            beta> 0, gamma > 0 and beta + gamma < 1
            Otherwise the ray has hit the plane outside the triangle
        */
        return false;
    }
    else
    {
        Vector3D normal = crossProduct(a - b, a - c);
        normal.normalize();
        // calculate the normal of the triangle by taking the cross product of two sides of the
        triangle
        r.setParameter(t, this);
        r.setNormal(normal);
        // Update the ray's intersection point and normal
        return true;
    }
}
```

```cpp
//Triangle.h
#ifndef _TRIANGLE_H_
#define _TRIANGLE_H_

#include "object.h"
#include "ray.h"
#include "vector3D.h"
#include "color.h"
#include <glm/glm.hpp>

class Triangle : public Object
{
private:
    Vector3D a,b,c;

public:
    Triangle(const Vector3D& p1, const Vector3D& p2,const Vector3D& p3, Material* mat):
        Object(mat)
    {

        isSolid = true;
        a=p1;
        b=p2;
        c=p3;
        // initialize the triangle object constructor here with the given points and material
    }

    virtual bool intersect(Ray& r) const;
};
#endif
```

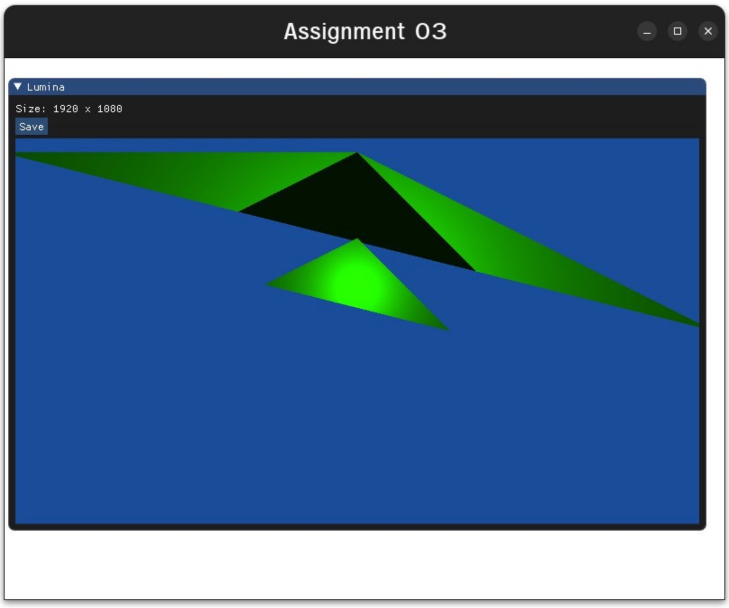
Ans 2) Blinn phong shading requires the calculation of ambient, diffu and specular components which can be directly calculated using moc theory implementation (Ref: Lecture Slides)



```cpp
    void setAmbient(const Color &amb) { ambient = amb; }
    Color getAmbient() { return ambient; }
    std::vector<LightSource *> getLightSourceList() { return lightSourceList; }
    // To get the list of light sources in the world in order to compute the shading
```

```cpp
    GLFWwindow *window = setupWindow(screen_width, screen_height);

    ImVec4 clearColor = ImVec4(1.0f, 1.0f, 1.0f, 1.00f);

    // Setup raytracer camera. This is used to spawn rays.
    Vector3D camera_position(0, 0, 10);
    Vector3D camera_target(0, 0, 0); //Looking down -Z axis
    Vector3D camera_up(0, 1, 0);
    float camera_fovy =  45;
    camera = new Camera(camera_position, camera_target, camera_up, camera_fovy, image_width, image_height);

    //Create a world
    World *world = new World;
    world->setAmbient(Color(1));
    world->setBackground(Color(0.1, 0.3, 0.6));

    Material *m = new Material(world);
    m->color = Color(0.1, 0.7, 0.0);
    m->ka = 0.1;
    m->kd = 0.9;
    m->ks=  0.7;
    m->n =  16;
    // Initialize the material properties, such as ka, kd, ks,n

    Object *triangle =new Triangle(Vector3D(2, 0, 0), Vector3D(0, 2, 0), Vector3D(-2, 1, 0), m);
    world->addObject(triangle);
    // Used 3D Calculator for visualising 3D coordinates, link for same https://www.geogebra.org/3d?lang=en f
    Object *Triangle1=new Triangle(Vector3D(10, 0, -3), Vector3D(0, 5, -3), Vector3D(-10, 5, -3), m);
    world->addObject(Triangle1);
    LightSource *light = new PointLightSource(world, Vector3D(0,1, 1), Color(1, 1, 1));
    world->addLight(light);
```

Ans 3: Shadows are implemented by checking the intersection point creating another triangle object of large size while maintaining distar from light source. Triangles are then placed on the plane while maintaining the same centroid for both triangles to create a shadow effect.



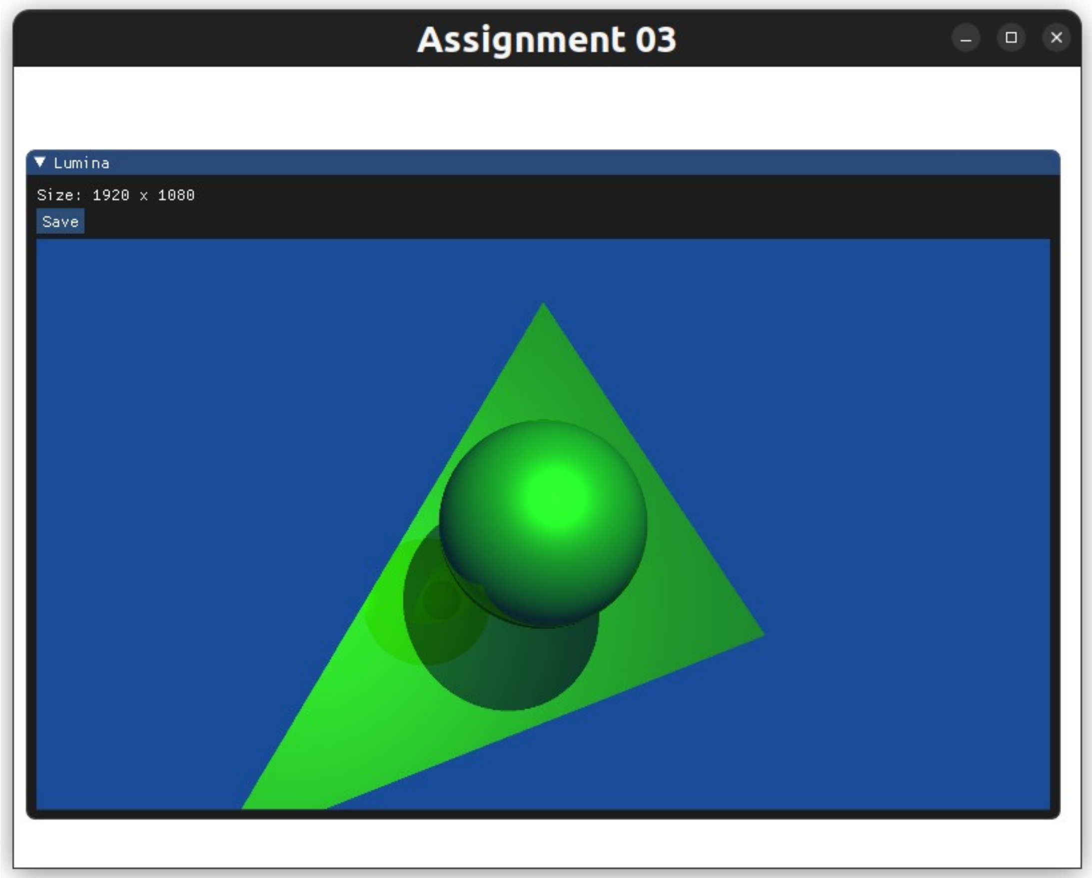

```cpp
#include "world.h"
#include "material.h"
#include <glm/glm.hpp>
#include <bits/stdc++.h>
Color Material::shade(const Ray &incident, const bool isSolid) const
{
    Vector3D normal = unitVector(incident.getNormal());
    // get the normal of the surface
    Vector3D l = incident.getDirection();
    // get the direction of the incident ray
    Vector3D v = l * (-1);
    // get the direction of the view ray
    LightSource *light = world->getLightSourceList()[0];
    Vector3D lightPosition = light->getPosition();
    // get the position of the light source
    Vector3D lightDirection = lightPosition - incident.getPosition();
    lightDirection.normalize();
    // get the direction of the light source from the intersection point
    Color intensity = light->getIntensity();
    // get the intensity of the light source
    Vector3D halfWayVector = (lightDirection + v);
    halfWayVector.normalize();
    // get the half way vector by normalizing the sum of the light direction and the view direction
    Color ambient(0), diffuse(0), specular(0);
    ambient = color * ka * intensity;
    diffuse = color * kd * intensity * glm::max(0.0, dotProduct(normal, lightDirection));
    specular = color * ks * intensity * pow(glm::max(0.0, dotProduct(normal, halfWayVector)), n);
    // initialize the ambient, diffuse and specular color
    Ray shadowRay(incident.getPosition() + lightDirection * 0.01, lightDirection);
    // initialize the shadow ray
    world->firstIntersection(shadowRay);
    if (isSolid)
    {
        if (shadowRay.didHit())
        {
            return ambient;
        }
    }
    return ambient + diffuse + specular;
}
```

Ans 4) Using theory from class, I implemented reflection and refracti using tir approximation(based on snell's law). Recursion is used for better approximations.

```
double eta = 1.35;
double eta_air = 1.0;

Vector3D refract(Vector3D d, Vector3D n, double refrac_index){
    double n_dot_d = dotProduct(d, n);
    Vector3D t = unitVector(((d - n*n_dot_d)/refrac_index) - (n*sqrt(1 - ((1 - pow(n_dot_d, 2))/pow(refrac_index, 2)))));
    return t;
}

bool tir_check(Vector3D d, Vector3D n, double refrac_index){
    double n_dot_d = dotProduct(d, n);
    double undertheroot = 1 - (pow(eta_air, 2)*(1 - pow(n_dot_d, 2)))/pow(refrac_index, 2);
    if(undertheroot < 0){
        return true;
    }
    return false;
}
```

```
Color World::shade_ray(Ray& ray, int count)
{
    firstIntersection(ray);
    if(ray.didHit()) {
        if (count <= 10) {
            if(ray.intersected()->isSolid == true){
                Vector3D n = ray.getNormal();
                Vector3D d = ray.getDirection();
                double n_dot_d = dotProduct(d, n);

                Vector3D r = unitVector(d - (2 * n_dot_d * n));
                Ray reflectedRay(ray.getPosition(), r);
                return ((ray.intersected())->shade(ray) + Color(0.3, 0.3, 0.3)*shade_ray(reflectedRay, count + 1));
            }
            else{
                Color k(0);
                double c;

                Vector3D n = ray.getNormal();
                Vector3D d = ray.getDirection();
                double n_dot_d = dotProduct(d, n);

                Vector3D r = unitVector(d - (2 * n_dot_d * n));
                Ray reflectedRay(ray.getPosition(), r);

                if(n_dot_d < 0){
                    Vector3D t = refract(d, n, eta);
                    c = -1 * n_dot_d;
```

```
                    double at = ray.getParameter();
                    k = Color(1.0, 1.0, 1.0);

                    Ray refractedRay(ray.getPosition(), t);

                    double r0 = pow((eta - 1.0), 2)/pow((eta + 1.0), 2);
                    double r_factor = r0 + (1 - r0)*pow((1-c), 5);
                    return k*(1 - r_factor)*shade_ray(refractedRay, count+1) + k*r_factor*shade_ray(reflectedRay, count+1);
                }
                else{
                    k = Color(1.0, 1.0, 1.0);
                    if(!tir_check(d, -1 * n, 1.0/eta)){
                        Vector3D t = refract(d, -1 * n, 1.0/eta);
                        c = dotProduct(t, n);
                        Ray refractedRay(ray.getPosition(), t);
                        double r0 = pow((eta - 1.0), 2)/pow(eta + 1.0, 2);
                        double r_factor = r0 + (1 - r0)*pow((1-c), 5);
                        return k*(1 - r_factor)*shade_ray(refractedRay, count + 1) + k*r_factor*shade_ray(reflectedRay, count
                        + 1);
                    }
                    else{
                        return k*shade_ray(reflectedRay, count+1);
                    }
                }
            }
        }
        return (ray.intersected()->shade(ray));
    }
    return background;
```


Assignment 03
Lumina
Size: 1920 x 1080
Save