Andrew Fryzel u1070220

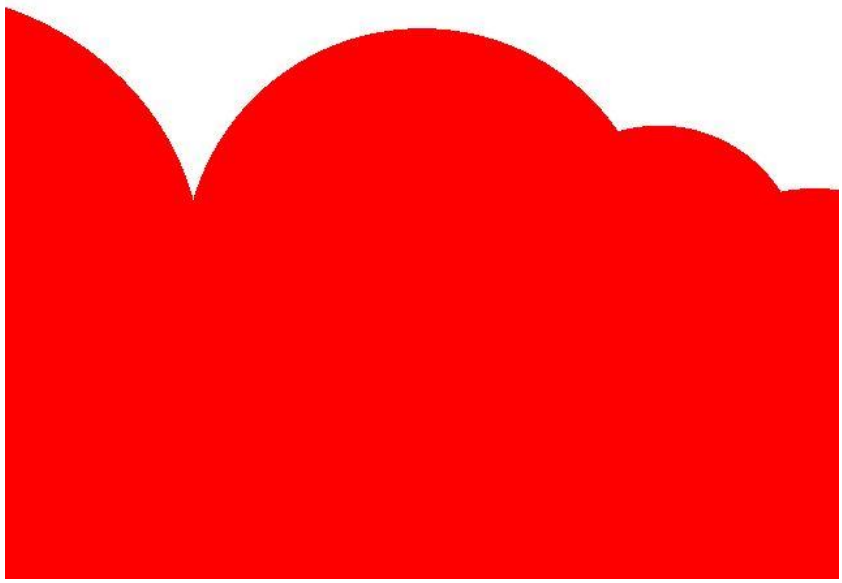
CS4600

Homework 5

Part 1: Ray Casting

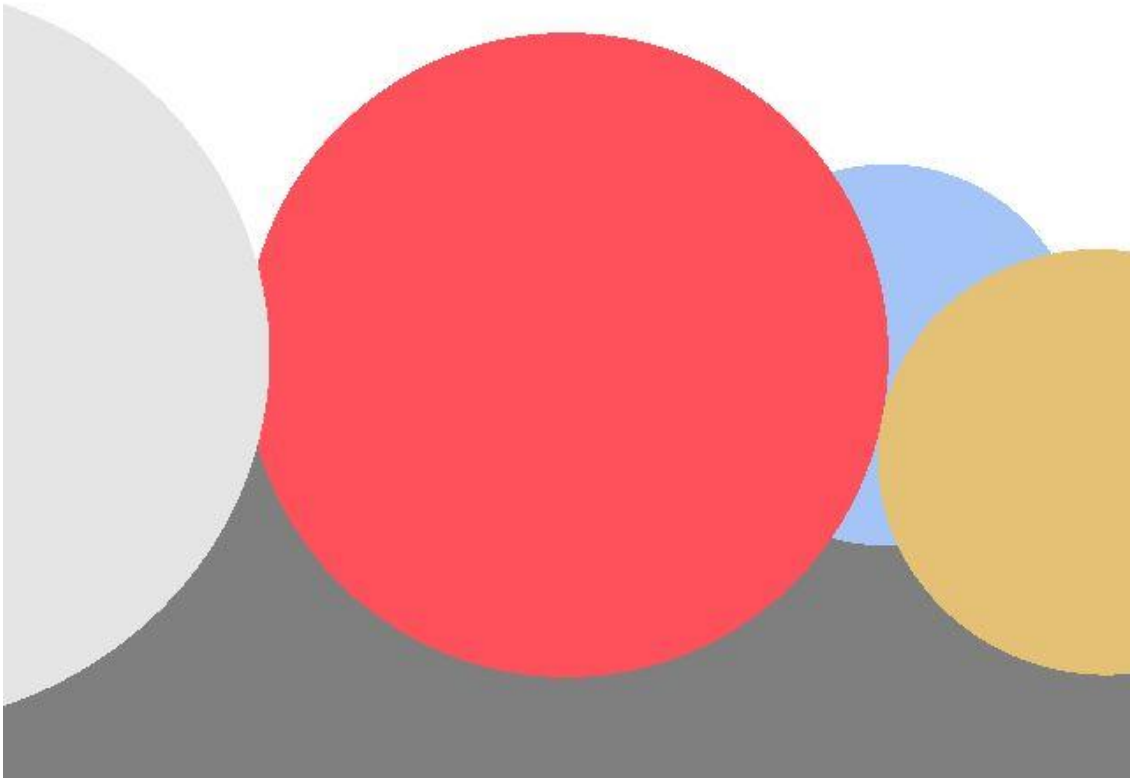
a) For the ray casting part of the assignment, I just followed the direction outlined in the assignment 5 document. I iterated through the spheres and if there was an intersection (bool intersectionCheck) then I changed the distance value to be I, which acts as a flag later when I checked if there had been an intersection.

```
Vector3f trace(
    const Vector3f& rayOrigin,
    const Vector3f& rayDirection,
    const std::vector<Sphere>& spheres)
{
    Vector3f pixelColor = bgcolor;
    Vector3f point{ 255, 0, 0 };

    float min = INFINITY;
    int distance = -1;
    float t0;
    float t1;


    for (unsigned int i = 0; i < spheres.size(); i++)
    {
        bool intersectionCheck = spheres[i].intersect(rayOrigin, rayDirection, t0, t1);
        // If "true" is returned, the function also returns the intersection points as distances along the ray(t0, t1, where t0 is the
        // first intersection and t1 the second one).

        if (min > t0&& intersectionCheck)
        {
            distance = i;
        }
    }
    if (distance != -1)
    {
        return point;
    }
    else
    {
        return bgcolor;
    }
}
```



b) For the second part of the assignment, the only thing that really needed to be changed was to alter the code to reflect the surface color of the specific sphere. This was done by changing the return to return the surfaceColor.

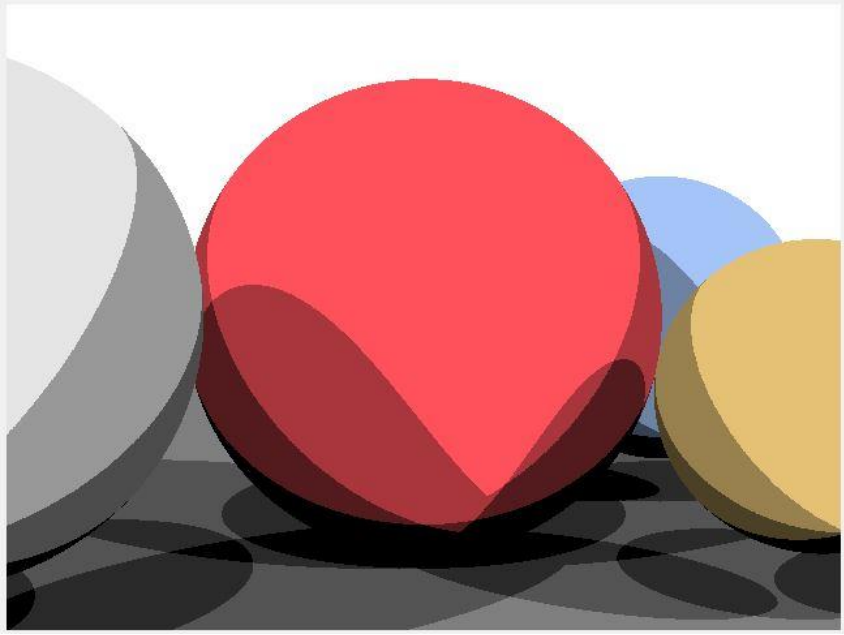
```
Vector3f trace(
    const Vector3f& rayOrigin,
    const Vector3f& rayDirection,
    const std::vector<Sphere>& spheres)
{
    Vector3f pixelColor = bgcolor;
    Vector3f point = Vector3f::Zero();

    float min = INFINITY;
    int distance = -1;
    float t0;
    float t1;


    for (unsigned int i = 0; i < spheres.size(); i++)
    {
        bool intersectionCheck = spheres[i].intersect(rayOrigin, rayDirection, t0, t1);
        // If "true" is returned, the function also returns the intersection points as distances along the ray(t0, t1, where t0 is the
        // first intersection and t1 the second one).

        if (min > t0&& intersectionCheck)
        {
            distance = i;
        }
    }
    if (distance != -1)
    {
        return point += spheres[distance].surfaceColor;
    }
    else
    {
        return bgcolor;
    }
}
```



Part 2: Shadow Rays

This section of the assignment also just followed the assignment instructions. I added the second for loop to iterate thought both the lightPositions and the spheres. This double iteration validates that the point isn't being intersected by any of the other spheres that could block the light. If there isn't an intersection then we can use the color of the sphere, otherwise, the bgcolor is used.

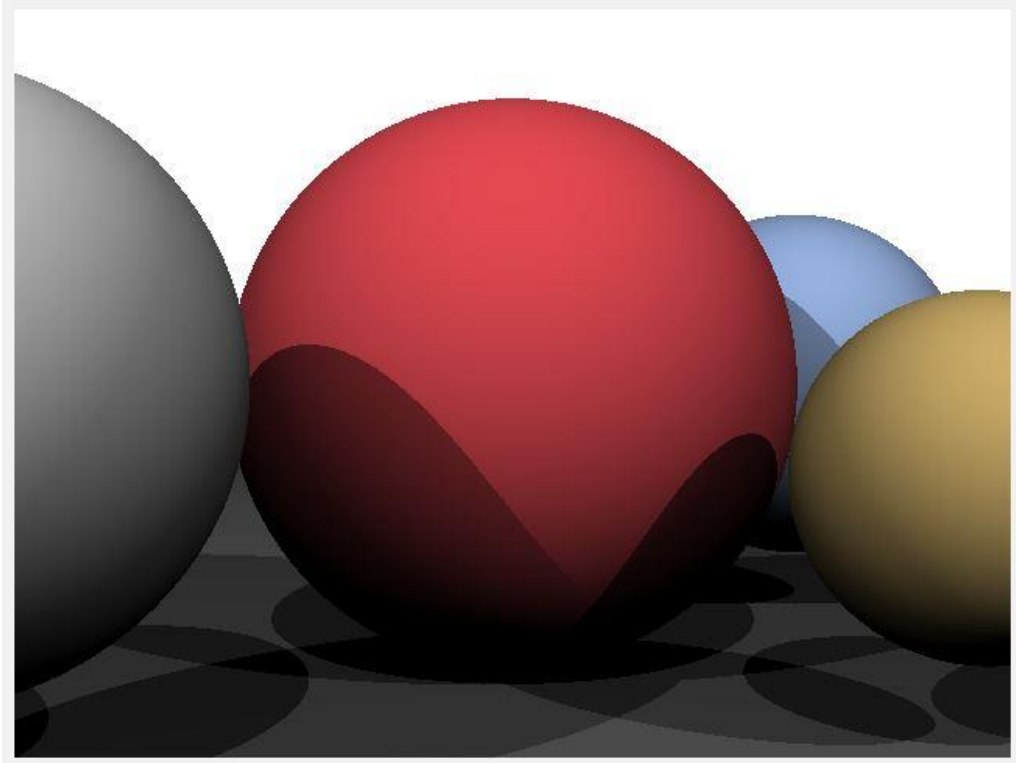
```
Vector3f pixelColor = bgcolor;
Vector3f point = Vector3f::Zero();
Vector3f locationOnHit;
float min = INFINITY;
int distance = -1;
float t0;
float t1;

for (unsigned int i = 0; i < spheres.size(); i++)
{
    bool intersectionCheck = spheres[i].intersect(rayOrigin, rayDirection, t0, t1);
    // If "true" is returned, the function also returns the intersection points as distances along the ray(t0, t1, where t0 is the
    // first intersection and t1 the second one).

    if (min > t0&& intersectionCheck)
    {
        min = t0;
        distance = i;
        locationOnHit = rayOrigin + (min * rayDirection);
    }
}

float t00;
float t11;

for (unsigned int j = 0; j < lightPositions.size(); j++)
{
    Vector3f lights = lightPositions[j] - locationOnHit;
    lights.normalize();
    bool intersectionCheck = false;

    for (unsigned int k = 0; k < spheres.size(); k++)
    {
        if (spheres[k].intersect(locationOnHit, lights, t00, t11))
        {
            intersectionCheck = true;
        }
    }

    if (!intersectionCheck)
    {
        point += 0.333 * spheres[distance].surfaceColor;
    }
}

if (distance != -1)
{
    return pixelColor = point;
}
else
{
    return bgcolor;
}
```



Part 3: Illumination Models

a) For the diffuse function, I referenced the assignment instructions and specifically that "we add 0.333 * kd * max(L.dot(N), 0) * diffuseColor". First, I use an if condition to make sure that the light isn't being blocked and that the direction vector DOT normal value == 0. If these conditions are met, then I used the previous formulas to calculate the resColor vector.

```
// diffuse reflection model
Vector3f diffuse(const Vector3f& L, // direction vector from the point on the surface towards a light source
    const Vector3f& N, // normal at this point on the surface
    const Vector3f& diffuseColor,
    const float kd // diffuse reflection constant
)
{
    Vector3f resColor = Vector3f::Zero();
    // Have to make sure that the light isn't blocked
    if (L.dot(N) > 0)
    {
        //From an intersection point P we cast rays to all lights (as in Task 2),
        //and if the light is not blocked, we add 0.333 * kd * max(L.dot(N), 0) * diffuseColor
        resColor = 0.333 * kd * L.dot(N) * diffuseColor;
    }

    return resColor;
}
```



b) For the second part, the phong code wasn't too challenging once I understood it. I reference the lecture 14 slides, specifically slide 11. The slide gives the formula $R = 2N(N \cdot L)-L$. I used this formula in the phong function to calculate R, then I find the diffuse value by passing in the appropriate parameters. If rDOTv > 0 then then I raise the dot product value to the power of alpha. If it isn't greater than 0, I simply set maximum to 0. I wasn't completely confident if I needed to set this to 0, if I could have done something more clever, but this implementation seems to work. The diffuse value, the maximum value, and the formula give in the assignment, are used to calculate the resColor using the phong method.

```
// Phong reflection model
Vector3f phong(const Vector3f& L, // direction vector from the point on the surface towards a light source
    const Vector3f& N, // normal at this point on the surface
    const Vector3f& V, // direction pointing towards the viewer
    const Vector3f& diffuseColor,
    const Vector3f& specularColor,
    const float kd, // diffuse reflection constant
    const float ks, // specular reflection constant
    const float alpha) // shininess constant
{
    Vector3f resColor = Vector3f::Zero();

    // TODO: implement Phong shading model
    //slide 11 lecture 14 R = 2N (N · L) -L
    Vector3f R = 2 * N * N.dot(L) - L;

    Vector3f diffused = diffuse(L, N, diffuseColor, kd);

    float maximum;

    R.dot(V) > 0 ? maximum = std::pow(R.dot(V), alpha) : maximum = 0;

    resColor = diffused + 0.333 * ks * specularColor * maximum;


    return resColor;
}
```