Andrew Thompson u0879848

Part 1:

- For the first part, I started with a simple for loop that goes through each of the spheres, returning red if the ray intersects with any of them. Returns bgcolor otherwise

```
{
    Vector3f pixelColor = bgcolor;

    // TODO: implement ray tracing as described in the homework description
    float t0, t1;
    for (int i = 0; i < spheres.size(); i++) {
        if (spheres[i].intersect(rayOrigin, rayDirection, t0, t1)) {
            pixelColor = Vector3f(1.0, 0.0, 0.0);
            return pixelColor;
        }
    }

    return pixelColor;
}
```

- To return the colors of the spheres intersected, I had to determine which sphere was in front. I did this by keeping track of the smallest t0 returned by intersect() and the sphere associated with it. After going through all spheres, I return the color of the closest sphere.

```
{
    Vector3f pixelColor = bgcolor;

    // TODO: implement ray tracing as described in the homework description
    float t0, t1, minDist = 0.0;
    int closestSphere = -1;

    for (int i = 0; i < spheres.size(); i++) {
        if (spheres[i].intersect(rayOrigin, rayDirection, t0, t1)) {
            //If this is the first sphere intersected, make it the closestSphere
            if (closestSphere == -1) {
                minDist = t0;
                closestSphere = i;
            }
            //Update the closestSphere if it is closer to the origin than the current closestSphere
            else {
                if (t0 < minDist) {
                    minDist = t0;
                    closestSphere = i;
                }
            }
            pixelColor = spheres[closestSphere].surfaceColor;
        }
    }

    return pixelColor;
}
```

Andrew Thompson u0879848

Part 2:

- Using the same for loop from above to find the closestSphere, I then add 3 new variables to find the effect of the light on the spheres:

```
    Vector3f intersectionPoint = Vector3f(0.0, 0.0, 0.0);
    Vector3f lightDirection = Vector3f(0.0, 0.0, 0.0);
    bool lightBlocked = false;
```

- Once I know the closestSphere and the closest distance minDist (which is essentially the point visible from origin), I store the location of where the ray hit the closestSphere in intersectionPoint. Using this point as the new origin, I check if a ray from the point to each light intersects a sphere. If there are no intersections, the light contributes 0.333 * surfaceColor to the pixel, creating shadows.

```
    if (closestSphere != -1) {
        //The coordinates of the point of intersection closest to the origin.
        intersectionPoint = rayOrigin + (minDist * rayDirection);

        //For each of the lights, add 0.333 * surfaceColor if there is not a sphere in the way
        for (int light = 0; light < lightPositions.size(); light++) {
            lightDirection = lightPositions[light] - intersectionPoint;
            lightDirection.normalize();

            //Check if there is a sphere in the way, setting lightBlocked to true if so
            for (int i = 0; i < spheres.size(); i++) {
                if (spheres[i].intersect(intersectionPoint, lightDirection, t0, t1)) {
                    lightBlocked = true;
                }
            }

            //Add the light's contribution if it is not blocked above
            if (!lightBlocked)
                pixelColor += (0.333 * spheres[closestSphere].surfaceColor);

            //Reset the lightBlocked status
            lightBlocked = false;
        }
    }
    else {
        //Uses bgcolor if there was never an intersection with a sphere
        pixelColor = bgcolor;
    }

    return pixelColor;
}
```

Part 3:

- To implement the diffuse shading, the only additions I had to make to part 2 was a way to compute the surface normal at point P, and to put in the equation described in the pdf. For starters, I defined a new variable surfaceNormal:

```
Vector3f surfaceNormal = Vector3f(0.0, 0.0, 0.0);
```

- I then computed the surface normal using the point P (intersection point) and the center of the sphere in question (closestSphere):

```
if (closestSphere != -1) {
    //The coordinates of the point of intersection closest to the origin.
    intersectionPoint = rayOrigin + (minDist * rayDirection);
    surfaceNormal = intersectionPoint - spheres[closestSphere].center;
    surfaceNormal.normalize();
```

- I now have L (lightDirection), N (surfaceNormal), diffuseColor (spheres[closestSphere].surfaceColor), and kd (1). With this information, I simply define diffuse using the equation in the pdf:

```
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();

    // TODO: implement diffuse shading model
    resColor += 0.333 * kd * std::max(L.dot(N), 0.0f) * diffuseColor;
    return resColor;
}
```

- And finally, I call diffuse whenever I increment pixelColor, which is when there are no spheres blocking the light:

```
//Add the light's contribution if it is not blocked above
if (!lightBlocked)
    pixelColor += diffuse(lightDirection, surfaceNormal, spheres[closestSphere].surfaceColor, 1.0);
```

- Next, for the complete Phong shading model, I am only lacking the parameter V, which I define as againstRay and compute below:

```
//Calculate V, the vector pointing against the ray
againstRay = rayOrigin - intersectionPoint;
againstRay.normalize();
```

- I now have every parameter needed to pass to phong, which is defined as the summation of the diffuse and specular components for each light. I do this similarly to previous steps: I just add what is returned by phong to the pixelColor

```
//Add the light's contribution if it is not blocked above
if (!lightBlocked) {
    pixelColor += phong(lightDirection, surfaceNormal, againstRay, spheres[closestSphere].surfaceColor,
        Vector3f::Ones(), 1.0, 3.0, 100);
}
```

- To define phong, I first get the diffuse component by calling diffuse with the appropriate parameters. I then find the specular component by first finding the reflection vector R as defined in the lecture slides, and then plugging it into the equation (also defined in the slides): $\mathbf{E_S = I_i \cdot C_S \cdot k_S \cdot max(\cos \beta, 0)^h}$

```
{
    Vector3f resColor = Vector3f::Zero();
    Vector3f R = Vector3f::Zero();

    // TODO: implement Phong shading model
    resColor += diffuse(L, N, diffuseColor, kd);

    R = 2 * N * (N.dot(L)) - L;
    resColor += 0.333 * specularColor * ks * pow(std::max(R.dot(V), 0.0f), alpha);

    return resColor;
}
```

- And these were the only changes needed to implement phong shading. My final and complete implementation of trace is below:

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

    // TODO: implement ray tracing as described in the homework description
    float t0, t1, minDist = 0.0;
    int closestSphere = -1;
    Vector3f intersectionPoint = Vector3f(0.0, 0.0, 0.0);
    Vector3f lightDirection = Vector3f(0.0, 0.0, 0.0);
    Vector3f surfaceNormal = Vector3f(0.0, 0.0, 0.0);
    Vector3f againstRay = Vector3f(0.0, 0.0, 0.0);
    bool lightBlocked = false;

    for (int i = 0; i < spheres.size(); i++) {
        if (spheres[i].intersect(rayOrigin, rayDirection, t0, t1)) {
            //If this is the first sphere intersected, make it the closestSphere
            if (closestSphere == -1) {
                minDist = t0;
                closestSphere = i;
            }
            //Update the closestSphere if it is closer to the origin than the current closestSphere
            else {
                if (t0 < minDist) {
                    minDist = t0;
                    closestSphere = i;
                }
            }

        }
    }
```

```
    if (closestSphere != -1) {
        //Calculate P, the point of intersection closest to the origin
        intersectionPoint = rayOrigin + (minDist * rayDirection);
        //Calculate N, the surface normal
        surfaceNormal = intersectionPoint - spheres[closestSphere].center;
        surfaceNormal.normalize();
        //Calculate V, the vector pointing against the ray
        againstRay = rayOrigin - intersectionPoint;
        againstRay.normalize();


        //For each of the lights, add 0.333 * surfaceColor if there is not a sphere in the way
        for (int light = 0; light < lightPositions.size(); light++) {
            lightDirection = lightPositions[light] - intersectionPoint;
            lightDirection.normalize();

            //Check if there is a sphere in the way, setting lightBlocked to true if so
            for (int i = 0; i < spheres.size(); i++) {
                if (spheres[i].intersect(intersectionPoint, lightDirection, t0, t1)) {
                    lightBlocked = true;
                }
            }

            //Add the light's contribution if it is not blocked above
            if (!lightBlocked) {
                pixelColor += phong(lightDirection, surfaceNormal, againstRay, spheres[closestSphere].surfaceColor,
                    Vector3f::Ones(), 1.0, 3.0, 100);
            }

            //Reset the lightBlocked status
            lightBlocked = false;
        }
    }
    else {
        //Uses bgcolor if there was never an intersection with a sphere
        pixelColor = bgcolor;
    }

    return pixelColor;
}
```