3D Programming: Assignment 1 Report

André Fonseca, Samuel Gomes, Tiago Almeida

June 4, 2017

1 Introduction

For our first assignment, we had to implement a global light model algorithm based on the Whitted ray tracer. We had to parse the NFF files in order to load a scene, calculate not only the primary rays, but also some objects intersections using implicit and explicit formulas and determine the image colors using the tracer reflected and refracted rays.

2 Scene creation and storage

In this assignment, the scene was created by reading a NFF (Neutral File Format) and storing the information it provides in a scene class that contains all elements of a scene.

2.1 NFF parsing

The NFF file was parsed by reading a line at a time with an input buffer. Then, the line is converted into a string parser and its values parsed one by one to variables.

According with the first character (or characters), the line will be processed to create a different scene element. For elements that require multiple lines (such as the camera) multiple lines are read before proceeding to parse the next scene element. Also, some lines have optional arguments. To scan these, a character peel must be performed to unsure the end of the line hasn't been reached.

2.2 The scene

The scene itself is composed of the following:

Camera The most complex structure of the scene, the camera is defined by 3 the vectors: eye (camera position), center (where the camera looks) and up (the camera orientation). Also, the camera contains a vertical field of view (FoVY), a near and far Z value (not used in this version) and finally a viewport size in x and y resolution.

Background Color The default color of the scene when no object is displayed.

Light Array An array of lights each with a position and color (white as default).

Material Array An array of different materials, each one with particular color, diffuse and specular refraction coefficient, shininess, transmittance and index of refraction.

Object Array An array of objects that exist on the scene. Each one has a position, an associated material and some virtual methods to be implemented in subclasses. The subclasses are spheres, planes or triangles.

3 Intersections Calculation

The objective of the intersection calculations is to find a factor t to multiply the direction of the tested ray in order to find an intersection point. If this factor cannot be calculated, it can be concluded that no intersection is found. Some intersections were computed based on implicit formulas relative to basic shapes. The shapes that were considered for this work include infinite planes, spheres and triangles. The triangles intersections are very useful because on a larger scale, they can be used to calculate intersections with triangle meshes.

3.1 Planes

The planes intersections are calculated based on the implicit equation depicted on Equation 1.

$$Ax + By + Cz + D = 0 \tag{1}$$

3.1.1 Checking Intersections

The intersection checking consists on computing the projection of a point on the plane in the planes normal. If the absolute value of the dot product between the ray direction and the plane normal is lower than an epsilon value, it can be concluded that the ray is parallel to the plane and the procedure is aborted.

Otherwise, the parameter t is calculated by using the expression of Equation 2. The parameter is then then tested. If it is negative the ray points on the opposite direction of the plane and the procedure is once again aborted. If the parameter can be calculated, it is applied to the ray expression to find the intersection point.

$$-\frac{rayInitPoint \dot{} normal + d}{denom} \tag{2}$$

3.1.2 Computing the normal vector

The normal vector is computed by the normalized cross product between the vectors inherent to the plane points.

3.2 Spheres

The sphere intersection is very interesting because of the checks that are performed to exclude the invalid points. This calculation is based on the implicit definition of the sphere given in Equation 3.

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = R^2 \tag{3}$$

3.2.1 Checking Intersections

First the ray is tested to see if it lies inside of the sphere (it's distance to the center is lower than the the radius of the sphere). Then, a validation is made to see if the ray points to outside the sphere (the point is outside the sphere and the angle is between the directions of the eye and the ray is negative). Next, and before the t parameter is fully calculated, the determinant of the second degree equation is checked for a negative root, in which case the calculation is discarded. Finally, the equation is solved and t is applied to the ray expression to find the intersection point.

3.2.2 Computing the normal vector

The normal vector in a point is defined by the center and that point. It is calculated in that manner.

3.3 Triangles

The triangle intersections are calculated based on the area defined by the barycentric coordinates of the ray point. A point can be described with respect to it's barycentric coordinates depicted in Equation 4.

$$P(\alpha, \beta, \gamma) = \alpha * a + \beta * b + \gamma * c \tag{4}$$

3.3.1 Checking Intersections

Firstly, validations similar to the plane intersection are made. Then, a check is done to see if the areas defined by the intersection point and the triangle points are all higher than 0 and lower than their sum. In case they are, the point is colliding the triangle.

3.3.2 Computing the normal vector

The normal vector is computed in a similar way to the plane normal vector.

4 Ray Tracing

4.1 Primary Ray

The primary ray is the ray cast from the camera origin to each one of the screen pixel coordinates. To correctly obtain these vector given each corresponding x and y pair, two steps are required. First, the x, y and z vectors of the camera in world space must be calculated. Then, the height and width of the screen in pixel size must be converted into world size width and height given the focal distance and the field of view in the Y axis. With this information, the x and y vectors can be obtained by using the corresponding coordinate as a percentage in total pixels, reentering by subtracting 0.5 and then multiplying for the pixel width or height in world coordinates and finally the corresponding eye vector. The z vector is simply obtained by multiplying the focal distance by the negative eye z vector. The normalized sum of all three vectors give the final primary ray in world space.

4.2 Ray Casting

The rayCasting function is very simple. Using a ray with an initial location and direction, we test for intersections with all objects in the scene and save the intersection and the object intersected point with the lowest positive distance t from initial point and in the ray's direction. This information is saved in references passed as arguments and true is returned. If there is no intersection, the function returns false.

4.3 Diffuse and Specular Component

Each trace calculates the color in the intersected point as a sum of multiple components. The two first components, the diffuse and specular depend on the position of the light source. As such, for each light source in the scene, a color contribution is calculated. The diffuse

component, is the light that directly hits the object. The smaller the angle between the light ray and the traced ray (the eye), the higher the diffuse contribution. However, the specular component is the light that is reflected from the light source to the eye and, as such, its higher when the angle between the light reflection vector (calculated using the *glm::reflect*) and the eye vector is small. Also, this value is then amplified by the objects shininess property.

4.4 Hard Shadows

Since the diffuse and specular components depend on the light source, it the path from the intersection point to each light is obstructed, instead of adding them a shadow should be cast (no color contribution is added by these components). To confirm if this happens, is as simple as testing for an intersection starting from the intersection point and in the light's direction. A small value in the light's direction is added to the initial point to avoid self-shadowing problems caused by floating point errors in calculations.

4.5 Reflection

The reflection direction is calculated using the *glm::reflect(direction, normal)* method (which returns the opposite direction to the one inputted). The ray intersection point and the reflection direction are used to build the reflected ray. The color contributions of posterior reflected rays are then recursively calculated, until the reflected rays do not intersect objects. Their color contributions are then summed to the pixel colors.

4.6 Refraction

The refraction direction is calculated with the procedure *refract*. The refraction coefficient is adapted to the position of the rays initial points. If the rays are inside the object, the refraction coefficient if the inverse of as if it was outside.

After the refraction direction is calculated, the rays color contributions are then measured in a similar way to the refraction ones.

5 Extra - Anti-aliasing

For the extra we chose to do Anti-aliasing, because the first scene we ray traced contained aliasing artifacts. We thought anti-aliasing would be challenging and also improve the final image.

The algorithm we implemented was Supersampling. It augments the image resolution which creates more pixel samples. It then projects those samples (it averages the pixel colors with their neighbors colors). The result is a lower resolution frame with less aliasing artifacts. The results obtained by the application of this method are depicted on Figures 1 and 2.

Appendix

Table 1: Times to render scenes with a depth of 6

Antialiasing	**1X**	**2x**
Scene	**Seconds to render**	
balls_low	17.15	71
balls_medium	97.73	419.38
balls_high	7992.80	14319.85
mount_low	34.06	130.88
mount_high	4064.21	13679.77
mount_very_high	78774.54	NA

Figure 1: No antialiasing

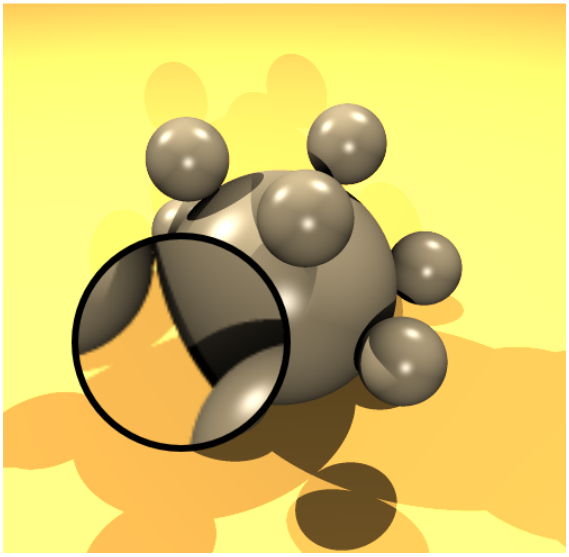

Figure 2: Antialiasing 3x

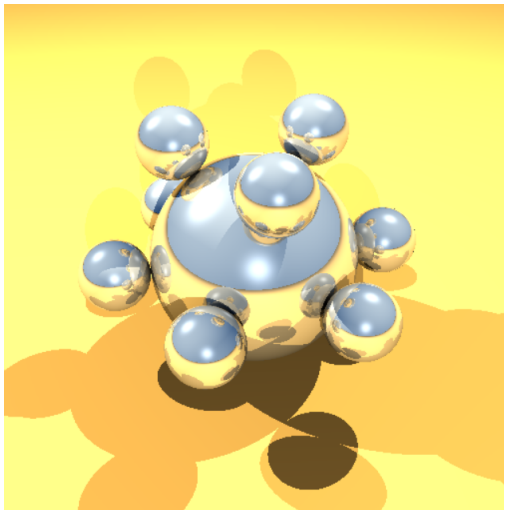

Figure 3: Scene Balls Low

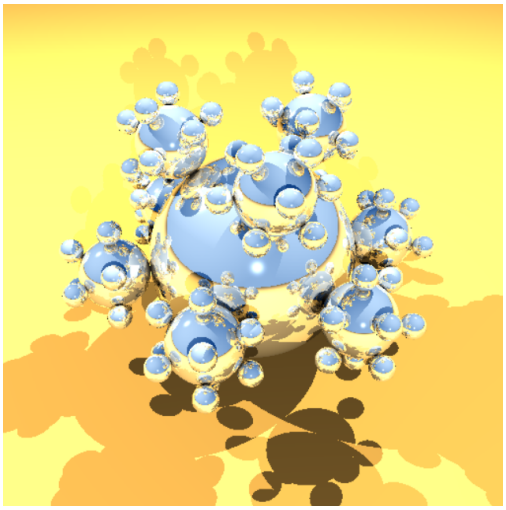

Figure 4: Scene Balls Medium

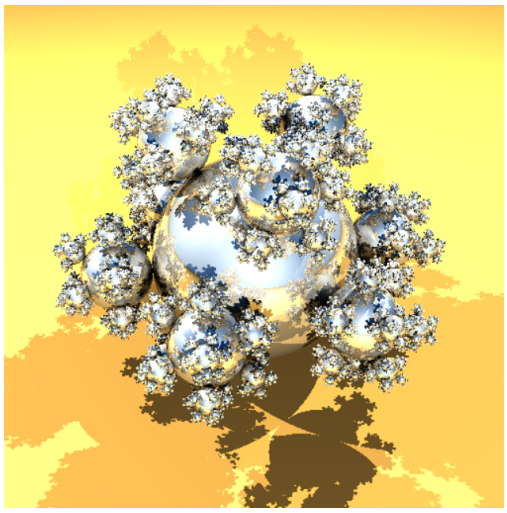

Figure 5: Scene Balls High

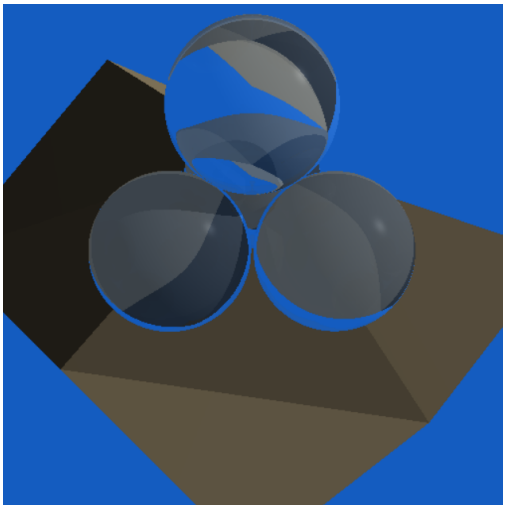

Figure 6: Scene Mount Low

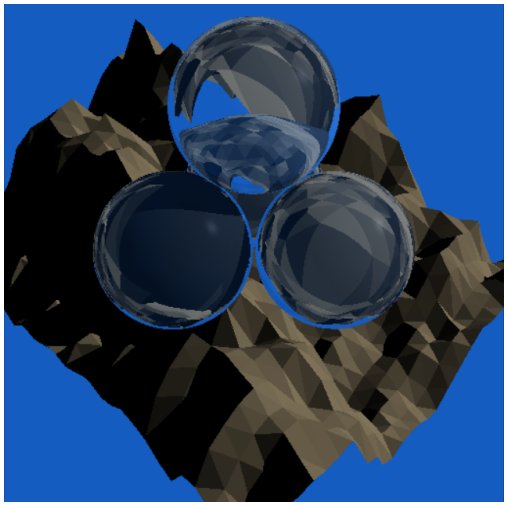

Figure 7: Scene Mount High

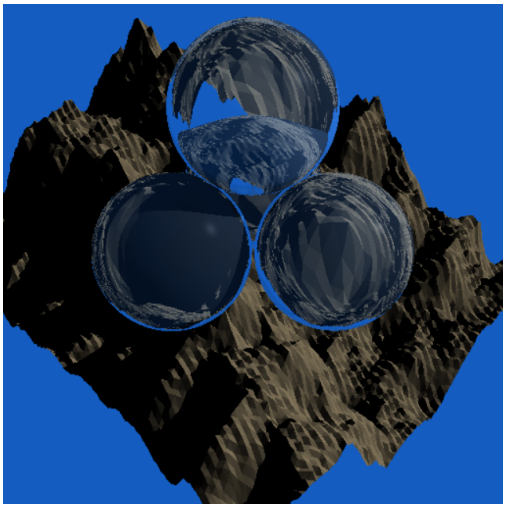

Figure 8: Scene Mount Very High