University of Lisbon

Master in Computer Science and Engineering

3D Programming

Author:
Pedro Rodrigues ist83548
Maria Duarte ist86474
Yu Cheng ist97282

Professor:
Prof. Joao Antonio Madeiras Pereira

Group 11
19 of April of 2020

Contents

Chapter 1

Introduction

This report explains all the techniques and implementations done for assignment 1 of the curricular unit 3D Programming. It focuses on the implementation of the ray tracing algorithm that allows to render tridimensional images at a very high photorealistic level. The idea behind this algorithm is to simulate the rays of light that exist in the real world. The ray origin is the observer and which ray is treated individually, creating new refracted or reflected rays, considering the materials it intercepts.

Chapter 2

T. Whitted Ray-Tracer

2.1 Blinn-Phong model illumination

As taught in theoretical classes, The Blinn-Phong model is a slight modification of the Phong reflection model and incorporates three components for the calculation of color: Ambient color component, Diffuse color component and Specular color component and it can be described by the equation

$$C_\lambda = C_{amb_\lambda} + C_{light_\lambda} k_{dif_\lambda} (\hat{n} \cdot \hat{L}) + C_{light_\lambda} k_{s_\lambda} (\hat{h} \cdot \hat{n})^n$$

In our implementation, we first calculate the diffuse direction and we multiply it with color of light and diffuse color of the material intercept. Next, we check if the material has specular reflection component, if it occurs, we sum the previous calculated color with the product of color of light and specular color of the material multiplied with the halfway vector elevated by the shyness of material.

2.2 Multiple source lights

In the previous model, the calculation of the color of point is being applied to each light source.

2.3 Hard Shadow

In our implementation, we render the pixel with local color component if it's not in shadow area, otherwise it s nor rendered and has a black color. To test if the point intercept is in the shadow, we shoot a shadow ray which has as origin, the point intercept + bias, as direction, it travels to the point of light source.
If it intercept any object during the trip, it's in shadow, otherwise it' s not.

2.4 Mirror Reflection

For the reflection, we adopted the mirror reflection model, from the normal vector of the intersection point and the direction of the incident light, we can calculate the direction of the reflected light and it can be given by r=2(L·n)$n - L$

2.5 Mirror Refraction

For the refraction, we have to determinate two case: the incident ray is inside the object(medium1) or outside(medium0), and according the situation we swap the index of refraction and reverse the direction of normal vector. Next, we calculate Vt which is the x component of the incident ray.
One thing to pay attention is that to have is trasmittance of the light, we have to ensure that sin0(Vt * eta) is not equal to 1. By the Snell's law, we know the angle of the refracted ray, so we calculate the x component of the refracted ray and add it to the reverse of the normal vector of the intersection point, therefore we obtain the direction of the refracted ray.

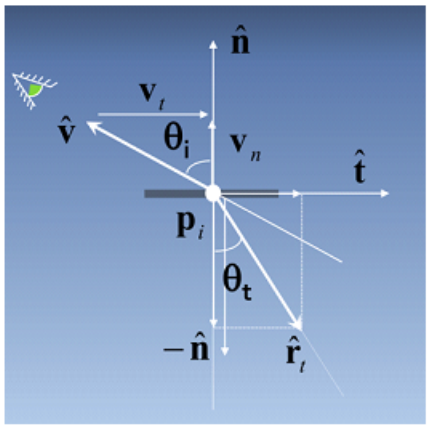

Figure 2.1: mirror refraction

2.6 Ray-Geometry Intersections

2.6.1 Sphere

To calculate if there is an intersection with a sphere, we implemented the algorithm taught in class. The first step of this algorithm is to calculate the distance (d) between the ray origin and the sphere's center, $d2 = (xc - xo)^2 + (yc - yo)^2 + (zc - zo)^2$. Then knowing that a sphere can be written in a quadratic equation 0, and that b represents the inner product between the ray direction and the vector oc, we calculate b, $b = xd(xc - xo) + yd(yc - yo) + zd(zc - zo)$. Then we check if $d^2 > r^2$ and b ¡ 0, if this is true there is no intersection and the algorithm ends. If it's false we need to calculate R, $R = b^2 - d^2 - r^2$ and check if $d^2 > r^2$. If it's true (origin is outside) then we calculate the smallest root, $t = b - \sqrt{R}$. If it's false (origin inside) we calculate the positive root,$t = b + \sqrt{R}$.

2.6.2 Infinity plane

To calculate if there is an intersection between a ray and a plane, we use a point that defines the plane (p0), a normal with the plane (N) and the ray equation $o + d \cdot t = p$. Using the equation that defines the plane, $(p - p0) \cdot N = 0$ and the ray equation, we can derive an equation to find the distance between the plane and the ray, $t = \frac{(o-p0)\cdot N}{N \cdot d}$. From this equation we can conclude that if there is no intersection because the ray and plane

are parallel, there's also no intersection if $t < 0$, because in this case the intersection happens behind the ray origin.

2.6.3 Triangles

To calculate intersections with triangles we used the Tomas Möller, Ben Trumbore "Fast, Minimum Storage Ray/Triangle Intersection" algorithm. As taught in class an arbitrary point p on the plane can be written by using its Barycentric Coordinates (α, β, γ), $p = \alpha a + \beta b + \gamma c$ with $\alpha + \beta + \gamma = 1$.
From that equation we can derive an equivalent one with only two independent Barycentric Coordinates $p(\beta, \gamma) = a + \beta(b - a) + \gamma(c - a)$ and using the ray equation we just need to solve $o + d \cdot t = p(\beta, \gamma)$ to calculate the intersection.

2.6.4 Axis Aligned Bounding Box

To calculate intersections between a ray and AABB we used the Kay and Kajiya Algorithm taught in class.
The goal of this algorithm is to find the largest entering t-value and the smallest exiting t-value for each direction, in the end of the algorithm we save the smallest exiting t-value (*tDist*) and the largest entering t-value (*tProx*) and we evaluate if $tProx > tDist$ or if $tDist < 0$ are true, if that's the case then there is no intersection.

Chapter 3

Stochastic sampling technique

3.1 Anti-aliasing

Anti-aliasing is a technique meant to help an image became smoother, by diminishing jaggies - stairstep-like lines. It helps to increase the resolution of an image. In this project we implemented antialiasing using the jittered method.
Before we were using a single ray for every pixel, but since a pixel contains an infinite number of points, and they may not have the same color, we need to take into account the different colors each point may have. So, what we did was, for each single pixel, we created N random samples, and created a ray for each one of those samples. With this, after calculating the color for every point in that pixel we calculated our final color by doing $color = color\ /\ N^2$. So, now for that pixel the color it has, takes into consideration the color for each of its random distributed N points.

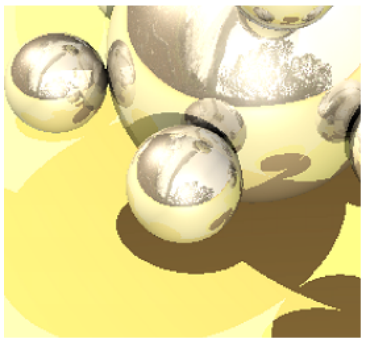

Figure 3.1: with anti-aliasing

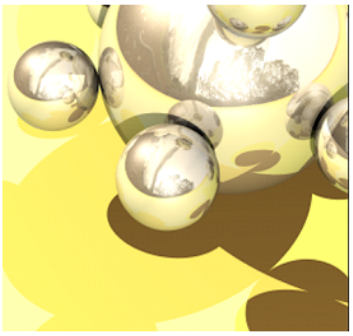

Figure 3.2: without anti aliasing

3.2 Depth of field

The Depth of Field effect is meant to simulate the way a camera or the human eye works by only focusing in a focus plane.
To achieve this effect first we need to have the effect of anti aliasing working, and for each ray generated for same x and y we generate different origin points in lens, taking into account the lens aperture defined in the NFF file. Using the coordinates x and y of the view plane and the focal ration defined in the NFF file we calculate the coordinates of the focal plane.
*focalPlane=viewPlane*focalRatio* With the origin and the destination of the ray defined

we have everything to calculate the ray direction that will be used in the calculation of the color. The final color will be the average of the sum of the colors of each ray.

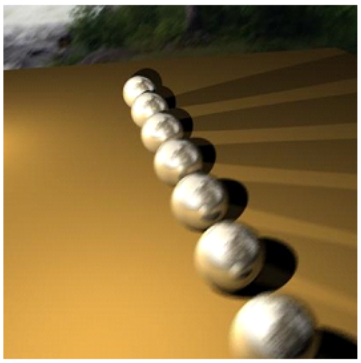

Figure 3.3: with depth of field

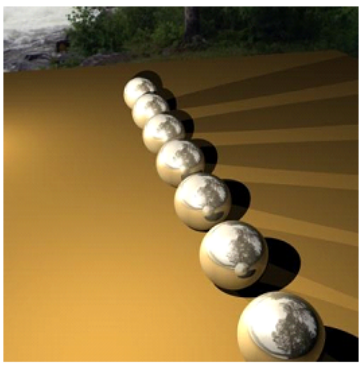

Figure 3.4: without depth of field

3.3 Soft shadows

In the previous chapter the hard shadow is being implemented, but since in real world the shadow won't be appearing so clear and delimited, the technique of shadow is beging applied to fulfill such requirement, and its core mecanism is to change the source light from a point to a area.
There are two cases for the implementation of soft shadow: with or without anti-aliasing. In the first case, since each pixel is subdivided in small points, from each point of the pixel a shadow ray will be shooted, but its direction will be pointing to a random point of area of light that is assumed to be infinite.
In our implementation, we concretize this technique by modifying the direction of shadow ray pointing it to a random point that is in a certain range from the source.
In the second case, we have N points that constitute the area of light and the intensity of each point of the light is being subdivided by N. From each pixel, shoot a shadow ray to each points of area light and sum the color obtained.
In our implementation, we didn' t create a structure that represent the area light, but by, for each light source, shooting N sub shadow ray that directs to a point that has a fixed distance from the source light, and if it's not in shadow, render the pixel with 1/N intensity of light source.

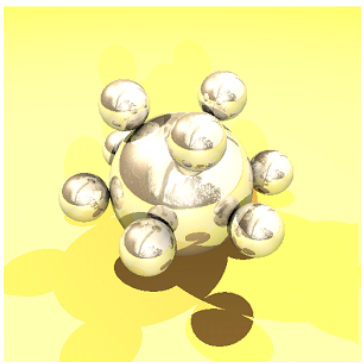

Figure 3.5: hard shadow

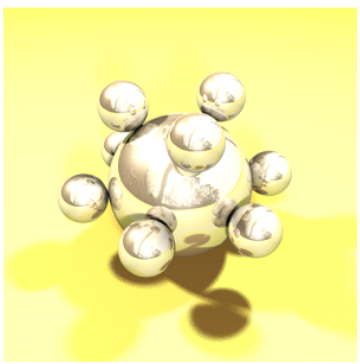

Figure 3.6: soft shadow with anti aliasing

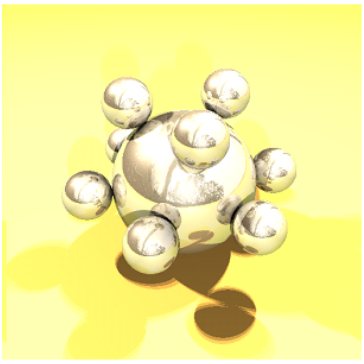

Figure 3.7: soft shadow without anti aliasing

Chapter 4

acceleration structure

4.1 Uniform Grid

The ray tracing algorithm is exhaustive, since each ray is intercepted with each object causing the rendering time to be proportional to the number of objects in the scene. Which makes it hopelessly insufficient for ray tracing large numbers of objects. A solution for this is the Uniform Grid, that drastically reduces the number of objects that each ray has to intercept.
A uniform grid has the shape of an axis-aligned box that contains a group of objects (the scene objects). The box is subdivided into a number of cells that also have the shape of an axis-aligned box and are all the same size. It basically is a uniform spatial division that divides part of the world space into a regular 3d cellular structure, hence the name. A ray that hits the grid only passes through certain cells and is only intersected with the objects in those cells, saving a lot of time. Not only that, but the cells are tested strictly in the order that the ray traverses them as t increases. This means, that the first object that the ray hits is the nearest one, and the process stops for that ray.

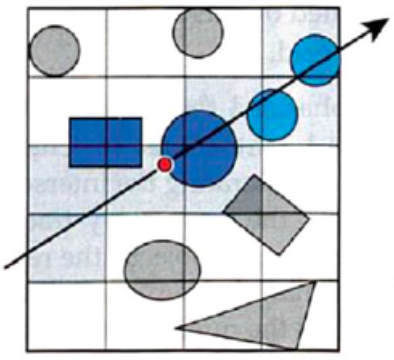

Figure 4.1: uniform grid

Now to construct the grid we had to create it, compute its bounding box and set up the cells.
The computations of the bounding box correspond to the union of the bounding boxes of all the objects. We got the min and max of the grid's bounding box with the functions *find_min_bounds(void)* and *find_max_bounds(void.*
Regarding the set up of the cells, the first thing we did was calculate the number of cells for each direction and then multiply all to get the total number of cells:
$s = pow(\frac{objects_number}{dim.x*dim.y*dim.z}, (1.f)/(3.f))$, with dim = max - min.

$nx = trunc(m * dim.x * s) + 1$,
$ny = trunc(m * dim.y * s) + 1$,
$nz = trunc(m * dim.z * s) + 1$,
$totalCells = nx * ny * nz$,
where m is a multiplying factor that allows us to vary the number of cells, and +1 is to guarantees the number of cells is never zero.
After this, and creating all the cells, we add the objects to their corresponding cells. So, for each object, we start by computing the indexes that contain the max and min coordinates of the object's box. After computing the*ixmin, iymin,izmin,ixmax,iymax* and *izmax*, we added the object to all the cells that it overlaps, being the index of the cell equal to *ix+nx*iy+nx*ny*iz* , with *ix=[ixmin,ixmax], iy=[iymin,iymax], iz=[izmin,izmax].*
Regarding the ray intersection of the grid, we started by checking if the ray intersects the bounding box of the grid. If not, the algorithm stops. Otherwise, we calculate the starting cell. To do this, we check if the ray starts inside the Grid, if so then we calculate the cell that contains the ray origin, doing:
*cellPos.x=clamp(intray.origin.x-bboxmin.x*nx / dim.x, 0, intnx- 1)* .
We do this as well for direction y and z.
If the ray doesn't starts inside the Grid, we find the cell where the ray hits the grid from the outside, doing: *cellPos.x=clamp((int)hitPoint.x-bbox→min.x*nx / dim.x, 0, (int)nx-1)*, where *hitPoint= ray.origin+ray.direction*t* with t being the distance we got from the ray interception with the grid's bounding box. Again, we do this for every direction.
After having the starting cell, we do the transverse set up. We start by calculating *vector dt =Vector(max.x-min.x / nx, max.y-min.y / ny, max.z-min.z / nz)*
that represents the ray parameter increments across the cell in the x, y and z directions. Then we calculate *tnext, step and stop*, by doing: If *ray.direction > 0* then
*tnext.x=min.x+ cellPos.x+ 1*dt.x; step.x=1; stop.x=nx*
Else then *tnext.x=min.x+ nx- cellPos.x*dt.x; step.x=-1; stop.x=-1.*
And if *ray.direction.x == 0* then *tnext.x= +INFINITY.* We do this for every direction.
Finally, the only thing left is the transversal loop, where we get the object for the current cell, find the one who is closest, save that *minDistance* and check if the hit object is confined to the current cell by doing *minDistance<tnext.x* and the same for y and z direction. We also check if *cellPos.x==stop.x* (we do the same for direction y and z) to check if the ray is about to exit the grid.
In case the ray in question is a shadow ray, the process is the same, we simply do not save the distance or the object, we just care if there is or not an interception.
The grid traversal algorithm that we used was Amanatides and Woo algorithm.

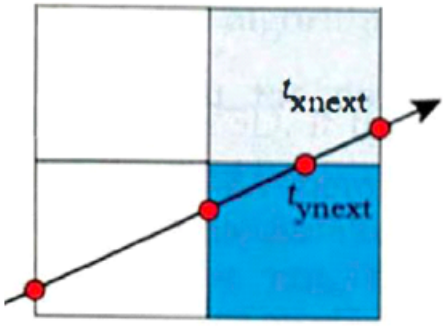

Figure 4.2: Tranversing the cells

Chapter 5

Statistics

Control Keys:
Key a – On/Off antialiasing
Key s – On/Off soft shadows
Key d – On/Off depth of field
Key g – On/Off grid
Execution Times (File: balls_medium.p3f)

Execution Times	GRID	SOFT SHADOWS	ANTIALIASING (N=4)	DEPTH OF FIELD
≅1.03s	NO	NO	NO	-
≅14.45s	NO	NO	YES	NO
≅14.82s	NO	NO	YES	YES
≅2.23s	NO	YES	NO	-
≅16.51s	NO	YES	YES	NO
≅16.84s	NO	YES	YES	YES
≅0.77s	YES	NO	NO	-
≅12.01s	YES	NO	YES	NO
≅12.20s	YES	NO	YES	YES
≅1.75s	YES	YES	NO	-
≅14.50s	YES	YES	YES	NO
≅13.99s	YES	YES	YES	YES

Figure 5.1: Executions Time