3D Programming: Assignment 2 Report

André Fonseca, Samuel Gomes, Tiago Almeida

June 4, 2017

1 Sampling Architecture

In order to facilitate the application of a variety of sampling techiques to create soft phenomena like anti-aliasing, soft shadows or depth of Field, a new sampling architecture was developed. A *Sampler* class is the superclass of several types of sampler like *RandomSampler*, *RegularSampler* or *JitteredSampler*. The only difference between each sampler class is the way they create samples. Some other mechanisms are common among those types of classes like the shuffling of the samples (method *shuffleSamples*) to avoid the creation of undesired graphical patterns or the obtainment of samples after their creation (method *nextSample*). The full architecture is depicted on Fig. 1.

2 Sampling Applications

2.1 Anti-Aliasing

Anti-aliasing is a very simple method to create. Instead of one ray for each pixel, multiple rays are casted. The offsets for each primary ray in a pixel are given by the current anti-aliasing sampler. After calculating the resulting color for each ray, the value is averaged defining the final pixel color. Algorithm 1 shows this simple procedure.

2.2 Soft Shadows

To achieve soft shadows in the rendering at first we tried sampling the lights position N times using multi-jittering. The image results of this approach didn't work quite what we expected, the whole scene was visibly darker and it was very slow. We ended up creating a MultiJitteredSampler object for each light, and in the ray-tracer shadow feeler calculation, we sample the position of of each light once within an area. With this approach we got soft shadows with different tones of gray.

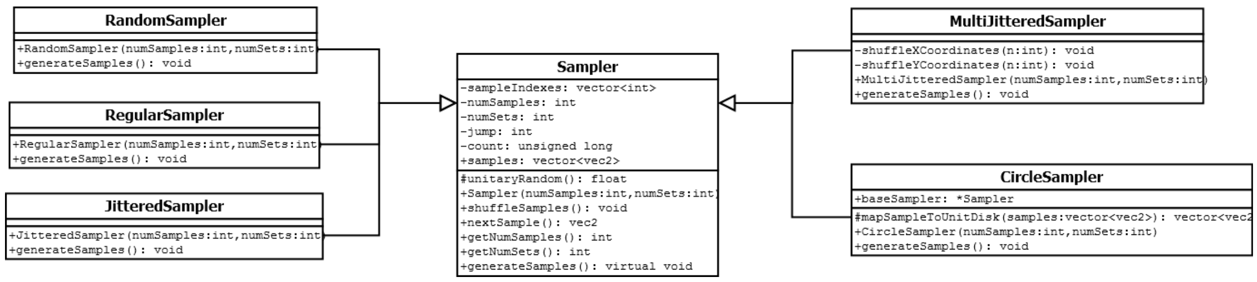
RandomSampler
+RandomSampler(numSamples:int,numSets:int)
+generateSamples(): void
RegularSampler
+RegularSampler(numSamples:int,numSets:int)
+generateSamples(): void
JitteredSampler
+JitteredSampler(numSamples:int,numSets:int)
+generateSamples(): void
Sampler
-sampleIndexes: vector<int>
-numSamples: int
-numSets: int
-jump: int
-count: unsigned long
+samples: vector<vec2>
#unitaryRandom(): float
+Sampler(numSamples:int,numSets:int)
+shuffleSamples(): void
+nextSample(): vec2
+getNumSamples(): int
+getNumSets(): int
+generateSamples(): virtual void
MultiJitteredSampler
-shuffleXCoordinates(n:int): void
-shuffleYCoordinates(n:int): void
+MultiJitteredSampler(numSamples:int,numSets:int)
+generateSamples(): void
CircleSampler
+baseSampler: *Sampler
#mapSampleToUnitDisk(samples:vector<vec2>): vector<vec2
+CircleSampler(numSamples:int,numSets:int)
+generateSamples(): void

Figure 1: Class UML diagram of the sampling architecture.

Algorithm 1 Anti-Aliasing

```
1: $color \leftarrow (0,0,0)$
2: for i=0; i< samplerAA.nSamples; i++ do
3:     $offset \leftarrow$ samplerAA.nextSample()
4:     $Ray \leftarrow$ calculatePrimaryRay($x$, $y$, $offset$)
5:     $color = color + rayTrace(Ray)$
6: end for
7: $color \leftarrow color/$samplerAA.nSamples
```

2.3 Depth Of Field

The implementation of the DOF included not only the creation of a new kind of sampler, called the *CircleSampler*, in order to map samples to a unit disk; but also the creation of the thin lens camera itself.

The *CircleSampler* simply maps each sample returned by another base sampler to a unit disk. Samples are mapped from the set [0,1] in a square to the set [-1,1] on a disk. It uses the concentric map algorithm [1]. The concentric map divides the circle into regions which then serve to map ranges of values as indicated by the expression bellow:

A new camera architecture was developed. The main camera class *Camera*, has two subclasses called *PinHoleCamera* and *ThinLensCamera*. The differences between the two types of camera is the way they calculate the primary rays (overriding the method *calcPrimaryRay*). While the *PinHoleCamera* calculates the primary rays using simply the perspective camera properties just like what was already implemented, the *ThinLensCamera* places the origin of the rays on a circle lens, through the use of the *CircleSampler* samples multiplied by the lens radius; and calculates their direction as the diference between the camera coordinates of the view plane points (each *pixelPoint*) and the circle samples (each *lensPoint*).

3 Uniform Grid

3.1 Bounding Boxes

To implement the Uniform Grid acceleration structure the first step was to create the Bounding Box class to be used in the algorithm. We changed the Objects classes that we had from the previous assignment, in a way that upon creation, their bounding box is also calculated and stored. The intersection algorithm used is the AABB -Kay and Kajiya which gives the tProx, the entrance intersection point and tDist, the exit intersection point. After using this acceleration structure, we noticed speedups of over 300, as can be seen on Table 1. A comparison of the execution of the ray tracer with and without using the uniform grid can be seen of Fig. 2.

3.2 Grid setup

The uniform grid setup is divided in three steps: The creation of the bounding box, the creation of the grid cells and finally filling those cells.

[1]More information about concentric map can be consulted on chapter 9 of the book "Ray Tracing From the Ground up" by Kevin Suffern

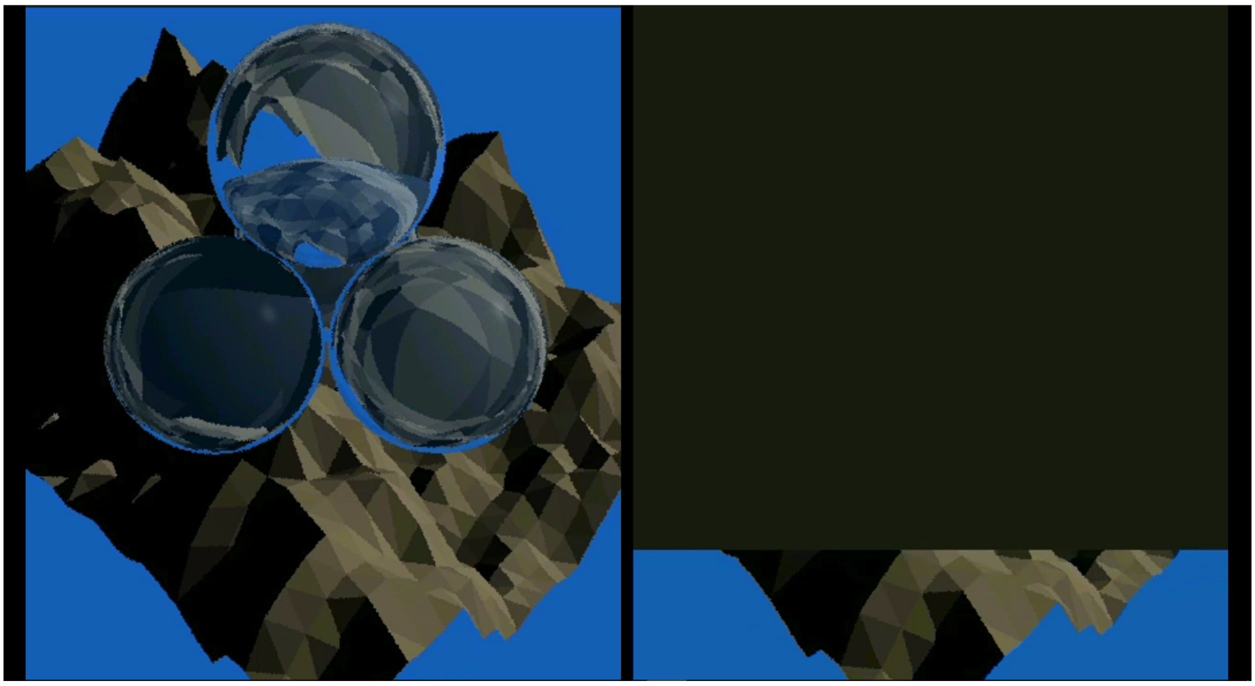
Figure 2: Comparison between the execution of the ray tracer with and without using the uniform grid for the mount_high test case. When the algorithm using the grid finishes (left image), the other is still processing (right image).

3.2.1 Bounding Box Setup

The grid needs a bounding box that defines the size of the entire grid and therefore encompasses all the objects of the scene. To calculate it, we compare the minimum values in all coordinates for each object and select the lower ones. We do the same proceeding for the maximum values but select the higher values instead of the lower ones.

3.2.2 Cells Creation

Now with the size of the grid given by the bounding box, we can calculate how many cells will be created. A variable parameter M is given when creating the grid to allow for flexible grid size. The higher the value of M, the bigger the grid. If $M = 0$, the grid will only have 1 cell. After calculating the number of cells, they are initialized in an array. The equation that calculates the grid size triplet is as follows:

$$nCells = worldSize \times M \times \sqrt[3]{\frac{nObjects}{worldSize.x \times worldSize.y \times worldSize.z}} + (1, 1, 1) \quad (1)$$

3.2.3 Filling the cells

After initializing the cells, they need to be filled with the objects that are inside the cell. In order to know which cells an objects belongs to, it is as simple as finding out which indexes the bounding box of the object is between. The object is then considered to be inside all the cells that define that limit (border cells inclusive).

To convert the bounding box of the object to an index, we executed algorithm 2 for both the minimum point and maximum point of the bounding box of each object:

Algorithm 2 World Position to Grid Index

```
1: procedure WORLDPOSTOIDX($point$)
2:     $relativePos \leftarrow point - gridBoundingBox.getMinPos()$
3:     $idx \leftarrow (0, 0, 0)$
4:     $idx.x \leftarrow \text{CLAMP}((relativePos.x/worldSize.x) \times nCells.x, 0, nCells.x - 1)$
5:     $idx.y \leftarrow \text{CLAMP}((relativePos.y/worldSize.y) \times nCells.y, 0, nCells.y - 1)$
6:     $idx.z \leftarrow \text{CLAMP}((relativePos.z/worldSize.z) \times nCells.z, 0, nCells.z - 1)$
7:     return $idx$
8: end procedure
9:
10: ▷ CLAMP return $arg_1$ if between $arg_2$ and $arg_3$ limits. It returns the transgressed limit
    otherwise.
```

3.3 Grid Traversal

The grid traversal is a straightforward process. In each step, the ray is intersected with the next cell. This is done by checking the intersection of the ray with equally spaced planes in all directions. The closest ray-cell intersection point (lowest t) is checked and the next cell for the traversal is identified accordingly (as being the one above, bellow, ahead or from the sides). The next ray-plane intersection is then checked (for that direction) and the boundaries of the grid are tested to avoid traversing out of the it.

Table 1: Comparison between no acceleration and uniform grid, acceleration raytracing with depth 6, time in seconds

Scene	**No acceleration**	**Uniform Grid**	**Speedup**
balls_low	17.15	12.85	1.3x
balls_medium	97.73	24.51	3.9x
balls_high	7992.80	79.90	100x
mount_low	34.06	55.88	None
mount_high	4064.21	127.27	31.9x
mount_very_high	78774.54	253.84	310.3x

4 Extra functionalities

4.1 Multi-jittering

After some research about the multiple sample techniques in existence, we decided to implement in our project the multi-jittered samples since they have both the advantages of jittered and n-rooks samples.

N-rooks are samples that guarantee that in the $n \times n$ grid, each possible x and y positions are sampled. After positioning in a very organized manner the samples in both the x and y coordinates are shuffled. This technique is 1D which means that there are only n samples in the $n \times n$ grid. This is not very practical in our architecture as every other sampling algorithm is 2D.

In order to combine the jittered samples with the above described technique, we subdivide each cell of the $n \times n$ grid into a $n \times n$ sub-grid. Samples are then placed in the $n^2 \times n^2$ in a manner that satisfies the n-rooks condition. However, this placement also satisfies the

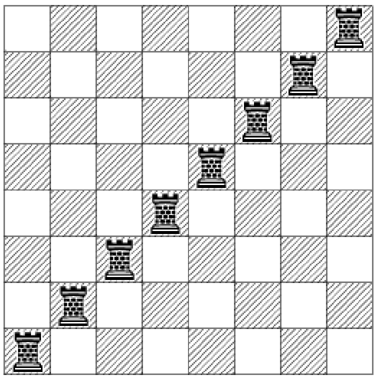

Figure 3: An example of the n-rooks algorithm using chess pieces.

multi-jittering condition for the $n \times n$ grid. Samples are then shuffled in their x and y coordinates such as in the n-rooks technique.

4.2 Mailboxes

Just as suggested in the theoretical slides, we also implemented a mailbox system to speedup the ray-tracing calculations. Every ray is created with an ID value that is obtained from a static ID generator.

Then, each object contains an ID of the last intersected ray as well as the t value ($o+t\times d = p$) of the intersection with that ray. Then, when calculating an intersection, if the ID of the ray is the same as the one locally stored, no calculations are required and the saved t value is used to quickly get the intersection point.

4.3 Plane Generator

In order to show anti-aliasing results, checkerboard planes are one of the best possible scenes to test. However, since the acceleration structures did not allow for the use of planes and we don't have support for textures, the only way to create such a plane was through multiple polygons. Since writing such a scene by hand would take an insurmountable time investment, we instead created a small program capable of creating such a scene adding it to the NFF file given some pairs of parameters such as **number of squares in x and y**, **size x and y** and finally **square color 1 and 2**. The plane is always axis-aligned with the XY axis.

Appendix

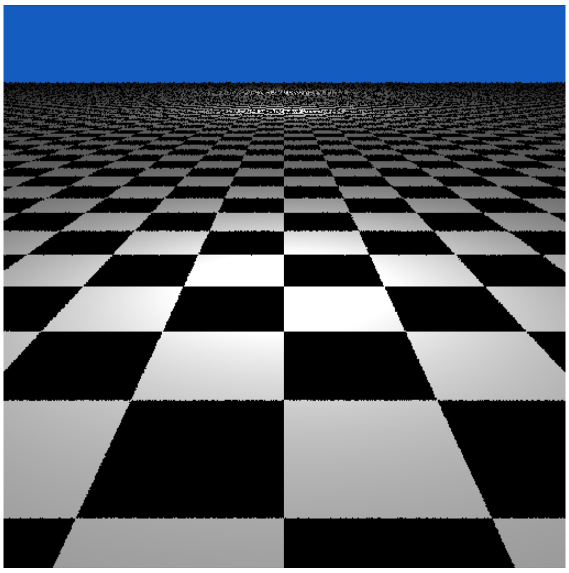
Figure 4: Plane generator scene with no anti-aliasing

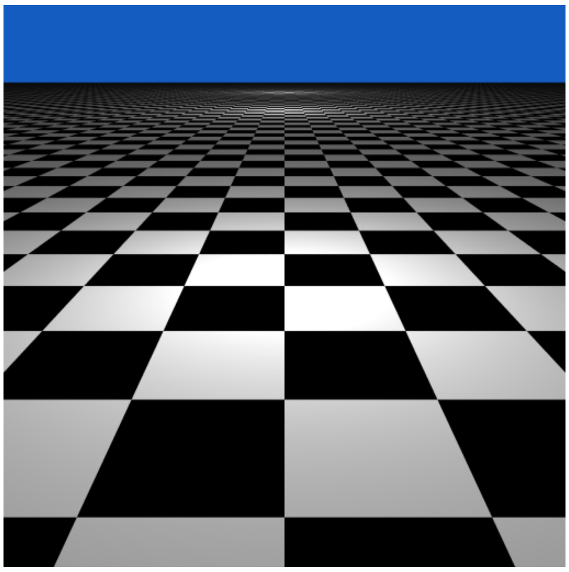
Figure 5: Plane generator scene with an 8x8 anti-aliasing

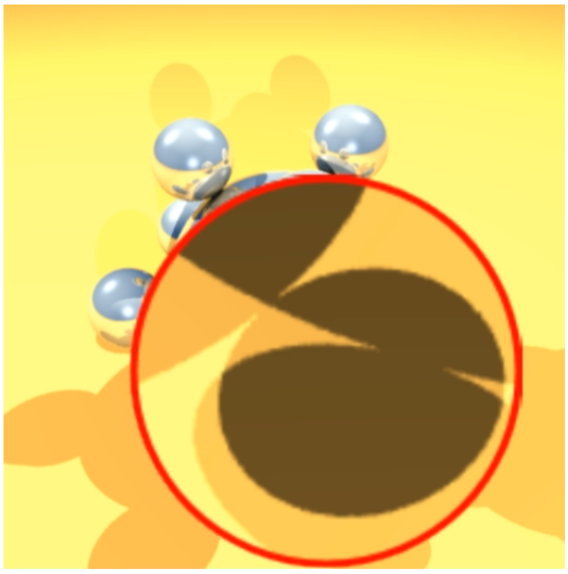
Figure 6: Hard shadows

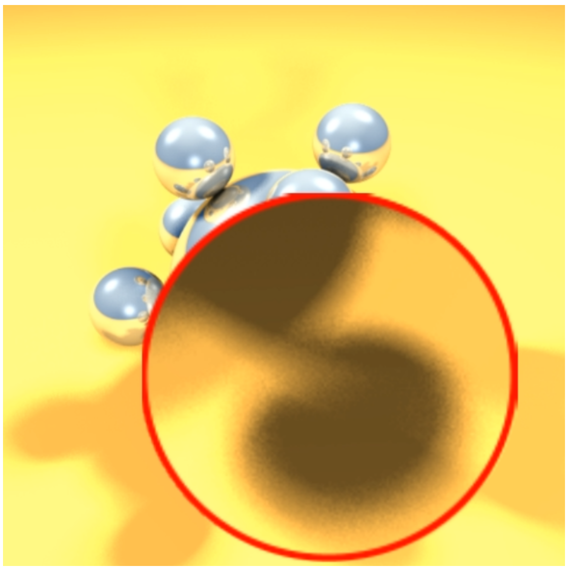
Figure 7: Soft shadows

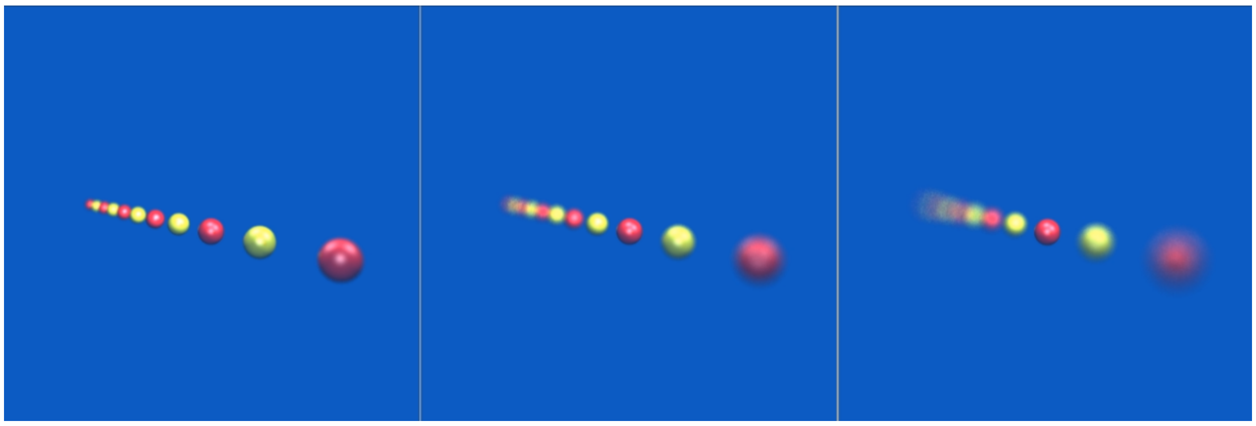

Figure 8: Depth of field example