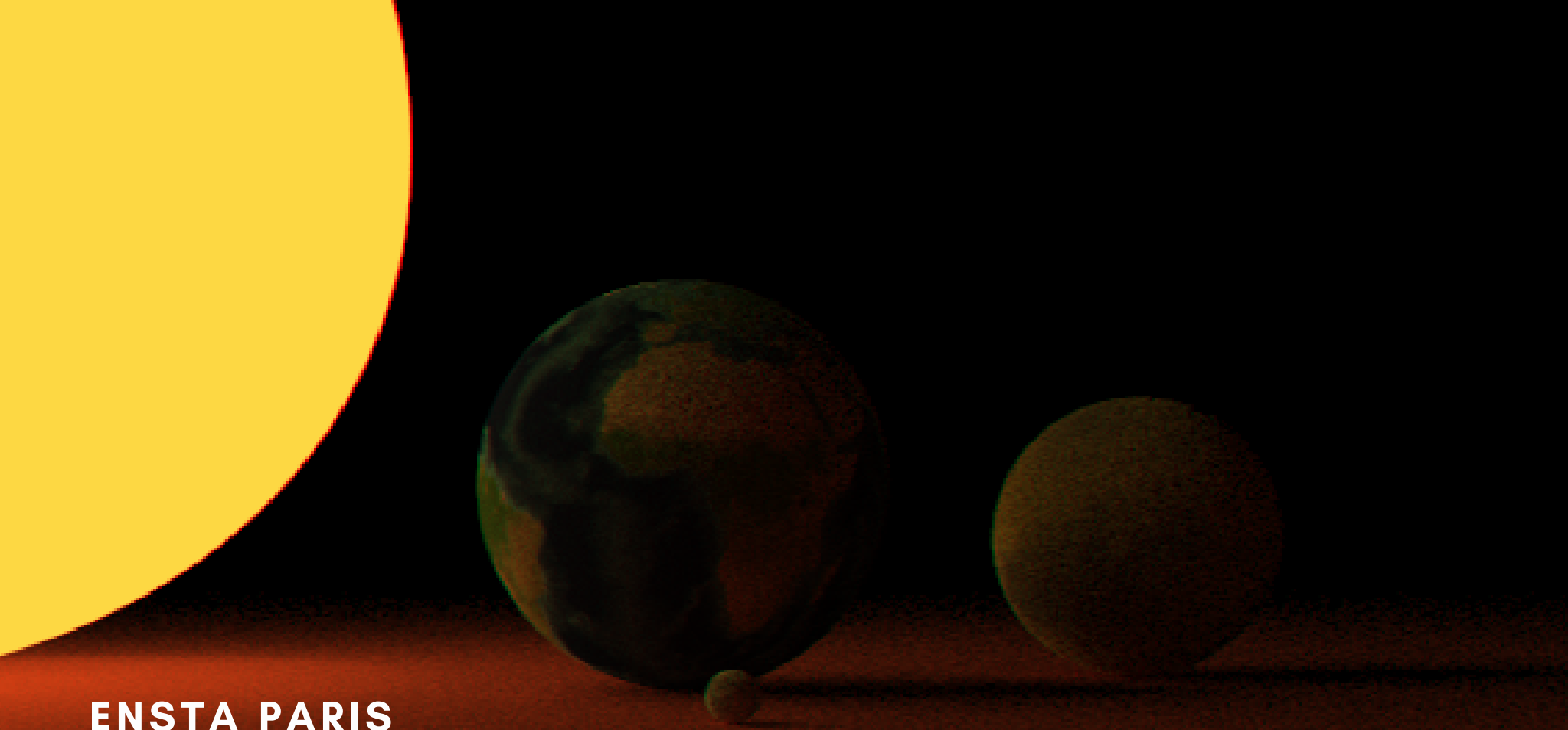


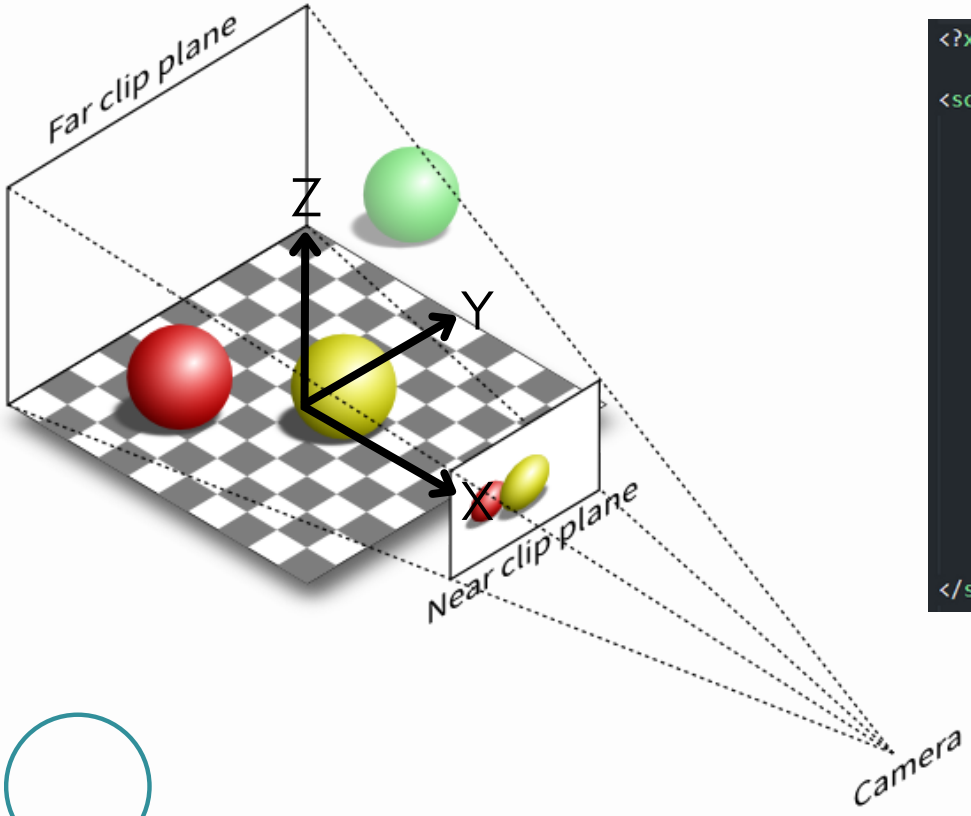
ENSTA PARIS

Ray Tracing

LACERDA Filipe
GUIMARAES Eduardo

GENERAL PROBLEM

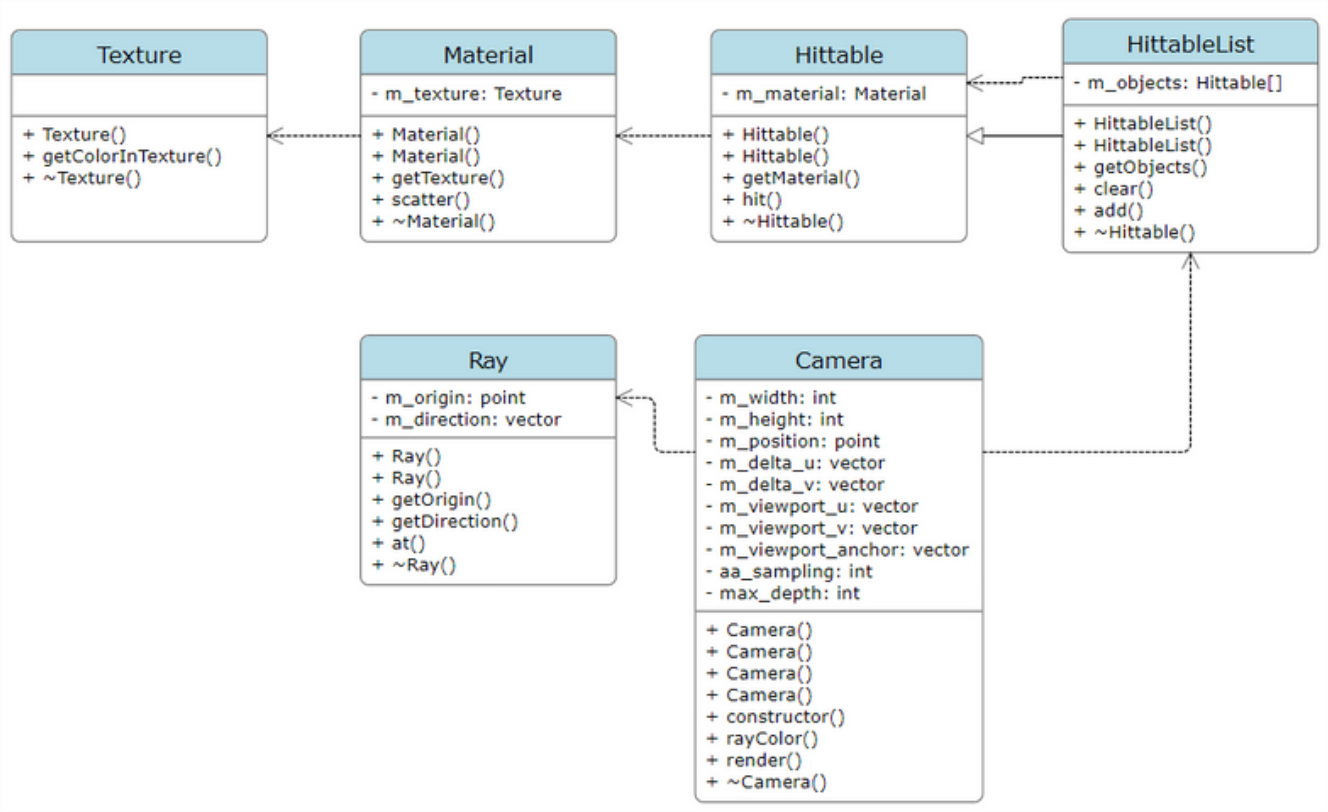
We can draw the base of our graphic motor as:



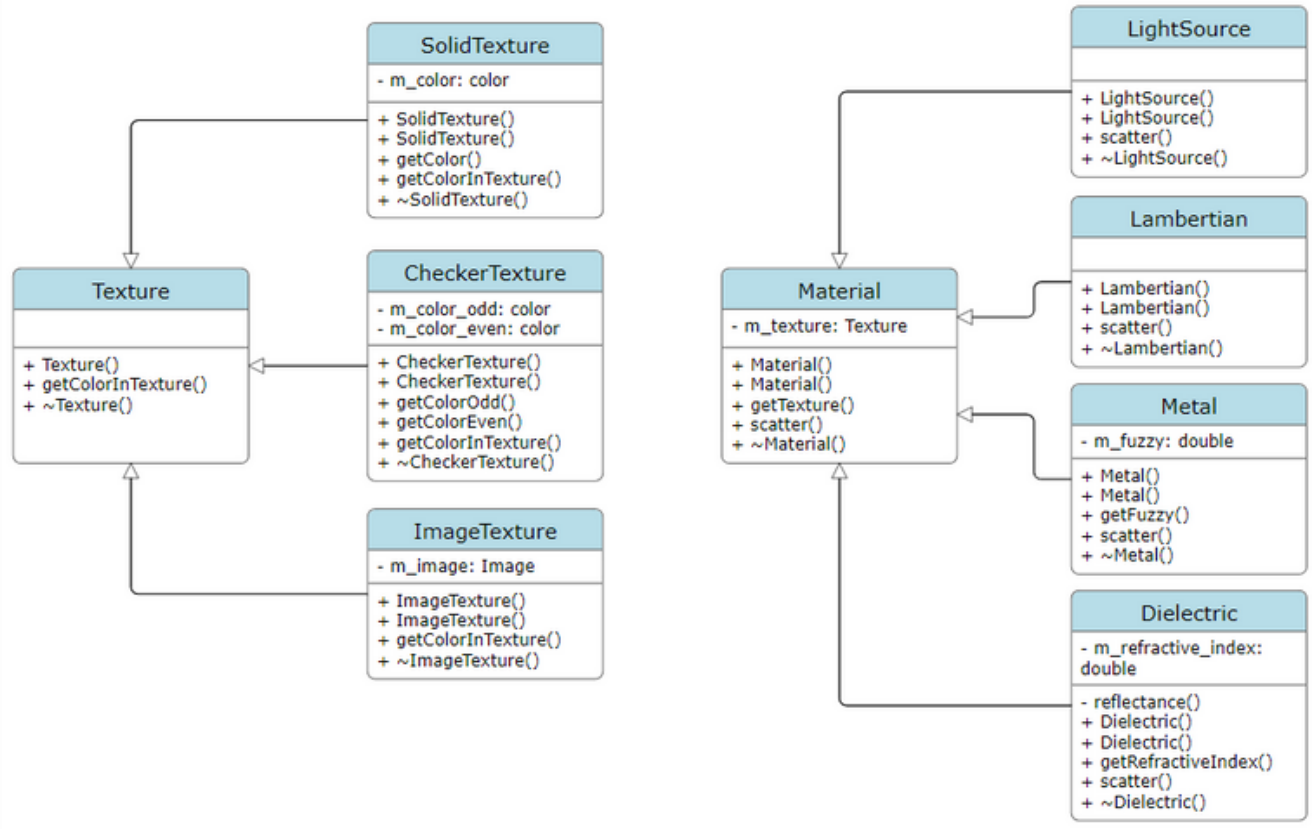

```xml
<?xml version="1.0" encoding="utf-8"?>

<scene>
    <object geometry="sphere">
        <center>
            <x>1.0</x>
            <y>-1.0</y>
            <z>1.2</z>
        </center>
        <radius>0.38</radius>
        <material appearance="light" texture="solid">
            <albedo>
                <r>1.0</r>
                <g>1.0</g>
                <b>1.0</b>
            </albedo>
        </material>
    </object>
</scene>
```

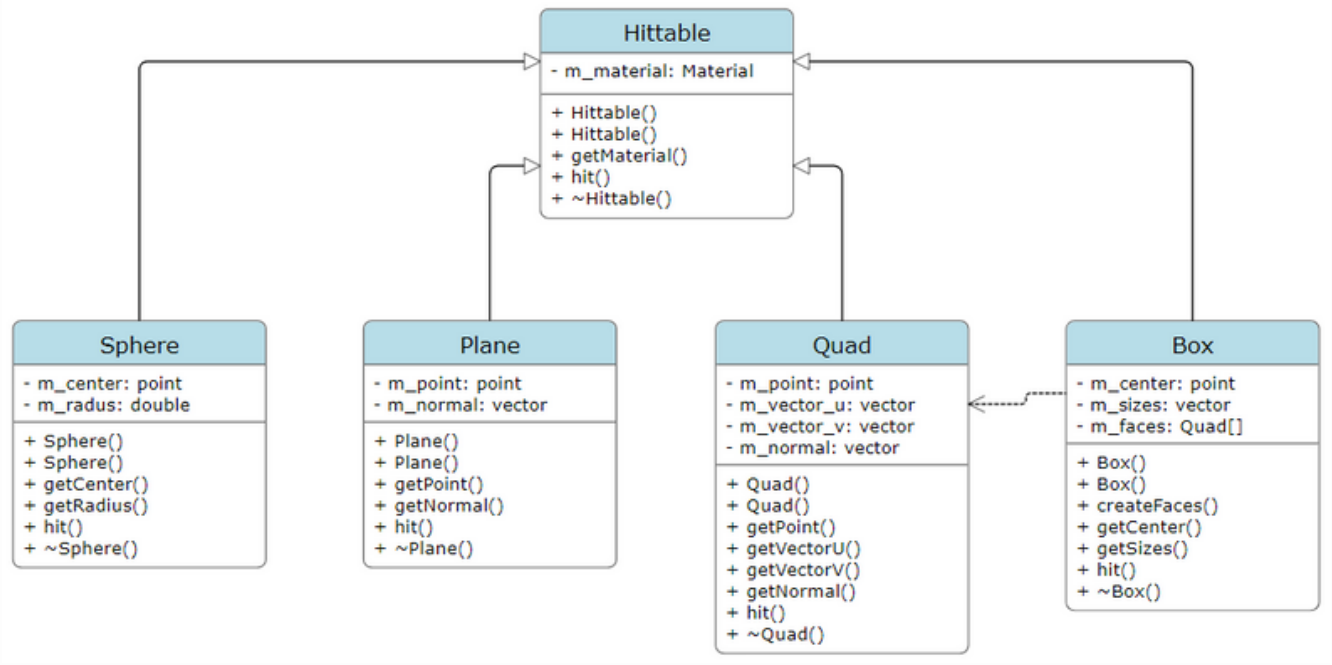
CLASS RELATIONS

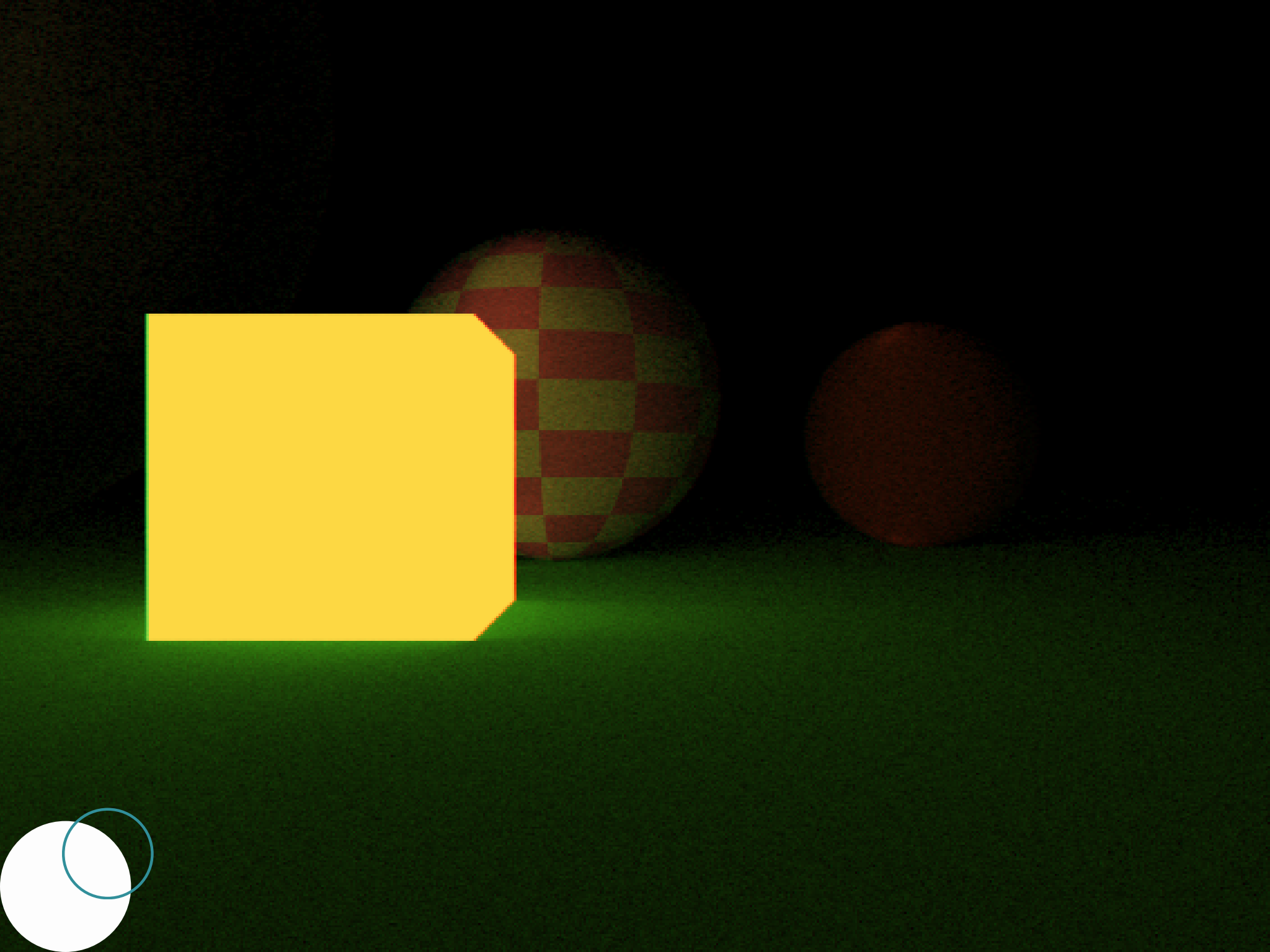


CLASS RELATIONS



CLASS RELATIONS





PERFORMANCE

For certains resolutions and anti-aliasing sampling numbers, our rendering can take several minutes. Where we could start?

- Data and functional parallelization;
- Different approaches to our tracing [2,3], from the statistical approach to anti-aliasing to something like a pre-treatment to ignore some of our rays;
- Code optimization of operations that are repeated multiple times during execution, like square roots and normalization.

FAST INVERSE ROOT

- Inverse Square Root is an expensive operation, made everytime a vector/ray is normalized;
- A faster iteractive version was used for Quake III:

```
inline float q_sqrt(float number){
    long i;
    float x2,y;
    const float threehalfs = 1.5f;

    x2 = number * 0.5f;
    y = number;
    i = * ( long * ) &y;  // evil floating point bit level hacking
    i = 0x5f3759df - (i >> 1); // what the ****?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ));  // 1st iteration
    y = y * ( threehalfs - ( x2 * y * y ));  // 2nd iteration

    return y;


}
```

PARALLELIZATION

OPTIONS:

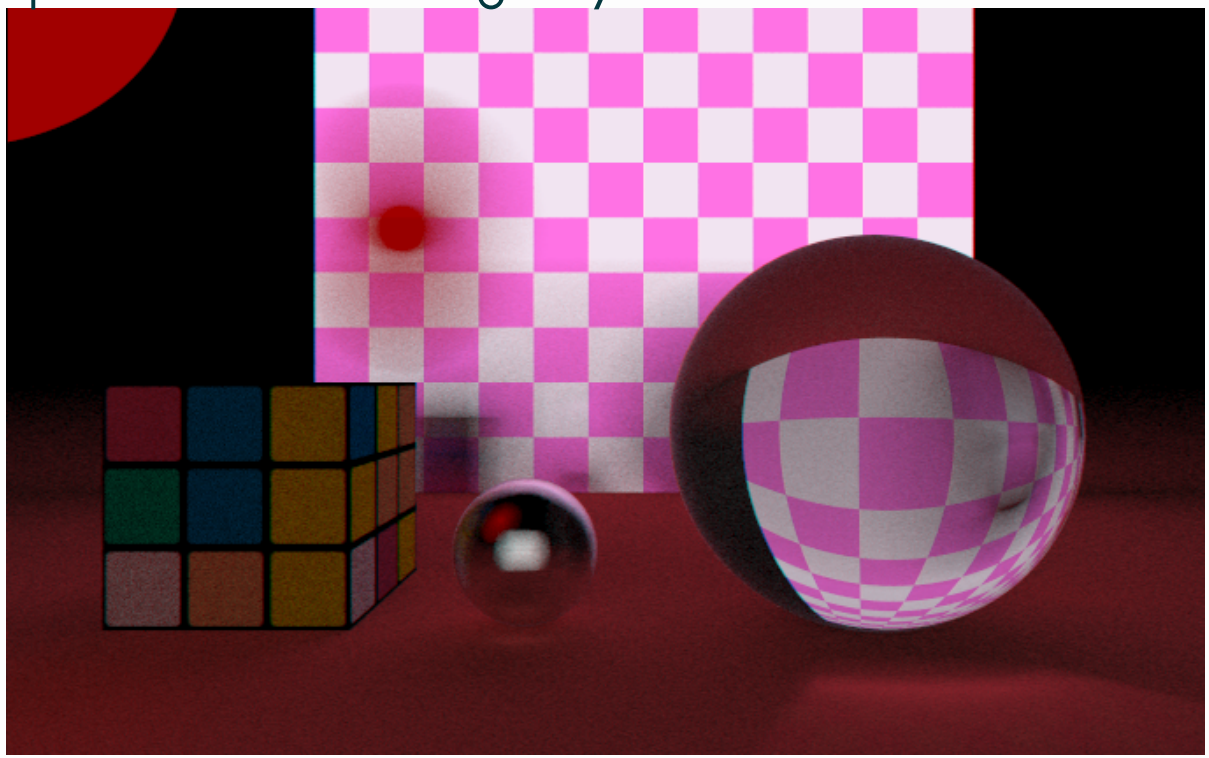
- GPU with CUDA: Fast, but big changes would be made to the code;
- OpenMPI: Task balance is not that trivial;
- OpenMP: We have a simple loop, so task balance can be made by the API's own scheduler.

OPENMP IT IS!

```
num_threads = omp_get_num_procs();//gets total number of threads
omp_set_dynamic(0);//lets code change number of max threds used in parallel block
omp_set_num_threads(num_threads);//sets number of threads used in a parallel block
#pragma omp parallel for private(i) schedule(dynamic,10)
for (j = 0; j < m_height; j++) {
    for ( i = 0; i < m_width; i++) {
```

PARALLELIZATION

- Images got at least 2x faster(the optimum parameters vary from sytem to system;
- Easy to mix it with MPI, in the case we have multiple multiprocessors in a single system.



PERFORMANCE

WHAT COULD BE DONE TO MAKE IT FASTER?

- The parallelization with CPU is nice, but using a GPU could make things even 10x faster [3]. We could also use an approach that mixes both [4];
- Even though we made the inverser square root faster, other ineficient operations remain (normal square root and antialiasing sampling for an example);
- Some smarter aproaches to raytracing could also be made, inspired by what they do in videogames, like leaving ray tracing to shadows and reflections and using rasterization for the rest;

References

[1] J. Q. Michael, “Parallel Programming in C with MPI and OpenMP” MC GRAW HILL, 2004.

[2] S. Peter, D. B. Trevor and H. Steve, "Ray Tracing in One Weekend Series", 2018-2023, free acess in https://raytracing.github.io/.

[3] A. Roger, “Accelerated Ray Tracing in One Weekend in CUDA”, 2018, free acess in https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/.

[4] M. Evgeny, “Practical Tips for Optimizing Ray Tracing”, 2023, free acess in https://developer.nvidia.com/blog/practical-tips-for-optimizing-ray-tracing/.