Universiteit Utrecht

Ray Tracing Fundamentals

# Fantastic Awesome Ray Tracer

November 11, 2018

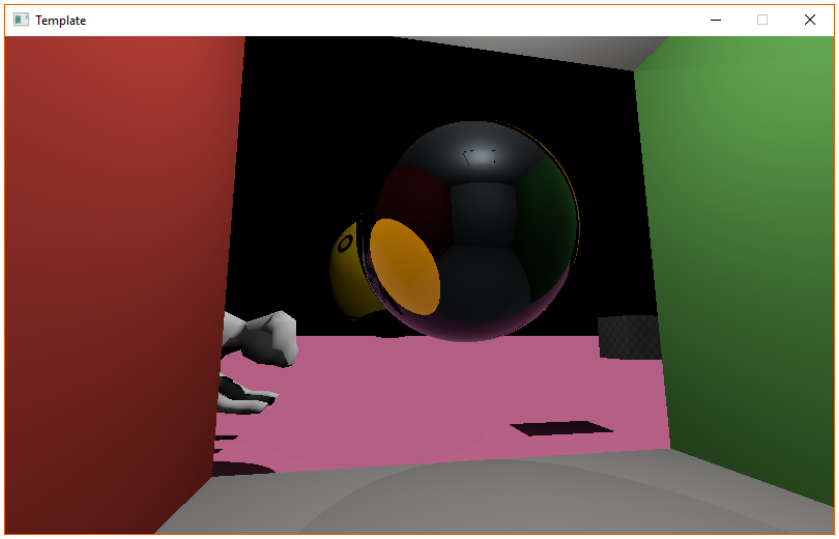
*Student:*
Mathijs Molenaar
5958970

Claudio Freda
6030769

*Teacher:*
Jacco Bikker

*Course:*
Advanced Graphics

*Course code:*
INFOMAGR




**Abstract**

In the first assignment for the INFOMAGR course we work on developing a Whitted-style ray tracer. It is able to render spheres, planes, and meshes, to shade accurately specular, glossy, refractive materials, and to map textures onto objects. The raytracer is implemented on the GPU using OpenCL.

# 1 Introduction

Computer graphics have two major use cases: real-time and Computer Generated Images.

Real-time graphics have been almost exclusively using rasterization. 3D triangle meshes are projected to 2D using a projection matrix. Afterwards they are rasterized on a 2D grid: the output image. The advantage of this approach is that it is very performant, especially with the hardware acceleration in modern graphics cards. The downside of rasterization by itself is that it is only capable of rendering direct lighting. Various algorithms exist to add shadows (i.e. shadow mapping) and global illumination (voxel cone tracing), but both are only coarse approximations.

When real-time frame rates are not a requirement, graphics programmers often defer to ray or path tracing. Ray tracing tries to model light rays intersecting with geometry. Because light rays that do not reach the camera are of no interest, the traversal is reversed. Light rays are traced out of the camera. In Whitted style ray tracing shadows are created by sending shadow rays/lines (rays with a start and end point) towards every light source. This algorithm can be further extended to path tracing, which takes indirect lighting into account and handles area light sources. This can be used to create very convincing images at the cost of performance.

# 2 General Architecture

We have opted for a C++ implementation as it allows for high performance and multi-threading (when done right). All geometry, material and lighting information is stored in the scene class. We decided not to use virtual functions in these classes because of performance concerns. Instead we have different classes for each type of geometry: spheres, planes and triangles. This makes it easy to loop over all geometry without a performance penalty. Since we wont be looping over materials, we opted for a different approach. Materials store a type enum and contain type specific data in a union. We did the same thing for lights because tracing shadow rays is way more expensive than a possible branch misprediction.

For every shape type we have defined an intersection functions with both rays and lines. Rays are needed for the main ray trace, while lines are needed for shadow rays. The scene class contains a ray intersect function that loops over all geometry and performs intersections tests on them. In addition it also contains functions to check for any intersections with a ray or line, which is used for shadows. The Whitted shading function takes care of the actual shading. It reads the material of the geometry that was intersected, sends out shadow and refraction/reflection rays and calculates the resulting colour.

# 3 OpenCL

Whitted style raytracing is a very computationally intensive process, but it is embarrassingly parallel. Since we had enough time left we decided that we would like to port our ray tracer to the GPU. The choice was between modern OpenCL 2.1+, OpenCL 1.2 and CUDA.

The CUDA option was immediately discarded because of the lack of Nvidia hardware on our side. Unfortunately both Apple and Nvidia have decided to drop modern OpenCL support as a way of promoting their own libraries (Metal and CUDA). Luckily, unlike CUDA, OpenCL can also be run on integrated graphics and CPUs with SSE support. In the end we chose OpenCL 1.2 because we would like to run our ray tracer on the fastest GPU available in the system. This also prevents us from having to create a second code path for cases where OpenCL/OpenGL interoperability is not available.

With OpenCL2.1, Khronos has brought a subset of C++14 to the GPU. The version of OpenCL we are supporting does not have this feature and requires us to program in C99. Since we didn't go overboard on Object Oriented Programming in our C++ implementation, porting to C99 was not that big of a challenge.

The biggest difference with the CPU implementation is that OpenCL does not allow for recursion. So instead we created our own stack to keep track of which rays to trace and what to do with the output. Right now every work item traces one ray which may lead to under utilization. According to AMD CodeXL's OpenCL profiler, the vector unit utilization is about 90%. With the limited time available we decided to not pursue this subject any further and instead focus on adding new features. When we are to add path tracing there will be a higher need for load balanced as a result of thread divergence.

In our initial OpenCL implementation we would ray trace the scene and copy the results back to the CPU. The CPU would than convert the image to a uint32 format and use SDL to plot the results. This proved to be a significant bottleneck, and we can do better! We expect that the primary graphics adapter always support OpenCL 1.2 (all 3 major vendors do). All major vendors also support OpenCL/OpenGL interoperability.

This allows us to use an OpenGL texture as an OpenCL texture. By rendering to such a texture we can completely avoid the CPU/GPU copy. It has the added advantage of free linear interpolation (for possible upscaling). It also allows us to output in a floating point format, automatically adding support for monitors with more than 8 bits of precision per color channel. In the end this gave us a 5x speed up (250fps vs 50fps) compared to the previous implementation that used SSE for float to int conversion.

# 4 Reflective materials

We have implemented simple reflectivity for spheres. A material can be reflective (fully specular) or glossy (blended diffuse/specular), with a specularity value.

When a ray hits a reflective surface, a new ray, reflected upon the surface normal, is spawned. The result of that ray is then colored by the reflective material color.

For glossy materials, the same is valid but the value is blended with a standard diffuse pass.


Figure 1: rendering of a glossy sphere

# 5 Refraction and absorption

We implemented refraction and light absorption for glass-like materials, working only on spheres.

When a ray hits, first of all we calculate the reflectance in the intersection point using Schlick's approximation. Then we calculate a reflective ray starting from the point of intersection.

Then we spawn a refractive ray that is checked for intersection inside of the sphere with the `intersectRayInsideSphere` function. This function is very similar to the standard ray-sphere intersection, but assumes that the ray starts inside of the sphere, thus inverting the sign of the Pythagoras's theorem addend at the end. After the intersection we spawn a second refractive ray coming out of the sphere.

At the end, the resulting colour is calculated as a weighted sum of the reflective ray and the final refractive ray results, using $reflectance$ and $(1 -$

*reflectance*) as weights. The refractive ray is multiplied by the absorption in Beer's law.

We decided to ignore possible further reflections inside of the sphere, as they would have a pretty small influence on the final result. When considering the future evolution of the raytracer into a path tracer, calculating additional reflections is probably the way to go.

With a little bit of additional time, we would have liked to implement refraction for meshes, but considering that it would be very expensive without a BVH, we decided to wait until the second assignment to do it.

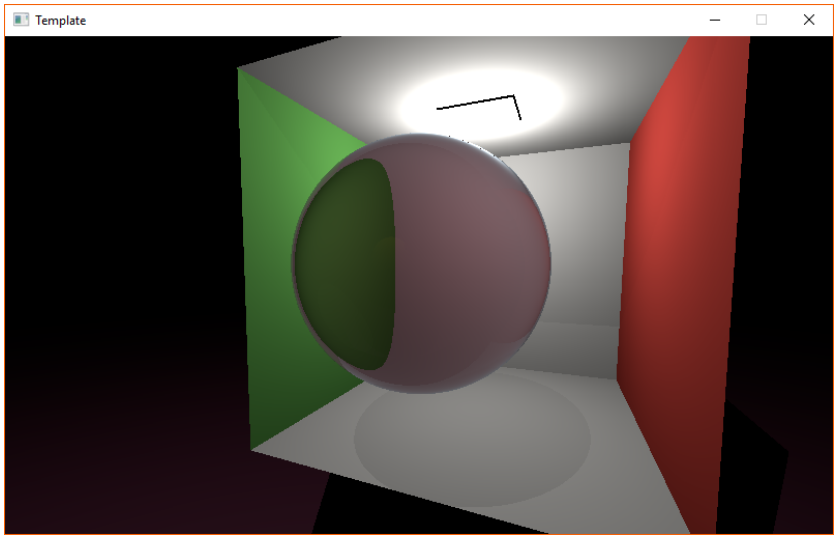



Figure 2: refractive effect with green-blue absorption, resulting in a red-coloured sphere

# 6 Mesh Support

We implemented a mesh loader using the Assimp library. Assimp supports many kinds of mesh formats, but we focused on `.obj` files. The mesh loader loads meshes as two buffers, one of vertices and one of triangles. The triangles are defined as three vertex indices, three normals, and three texture coordinates. It also loads a material, if available, from the .obj file, but we only support diffuse materials and textures while importing meshes, at least for now.

As for rendering the mesh, we implemented the Möller-Trumbore ray-triangle intersection test[3] and we check light and shadow rays against every triangle in the mesh. This becomes slow very quickly, but we know that the BVH implementation in the second assignment will improve that.

The resulting surface normal and texture coordinate (see below) are blended between the normals of the three vertices of the triangle using

barycentric coordinates. The ray-triangle test already calculates those we just had to do the blending.

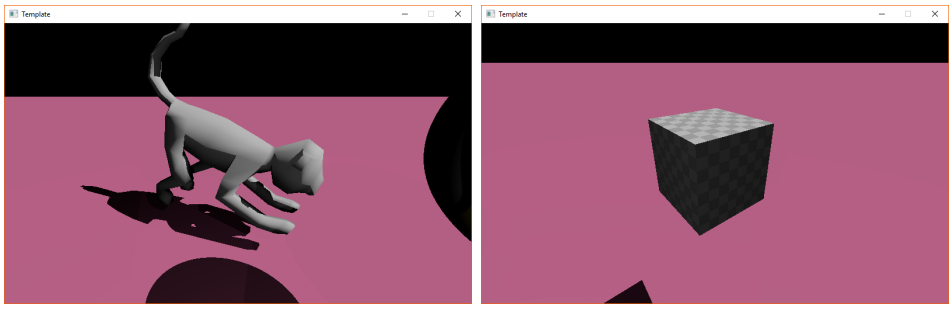


Figure 3: Rendering of a monkey and a textured cube

# 7 Texture Support

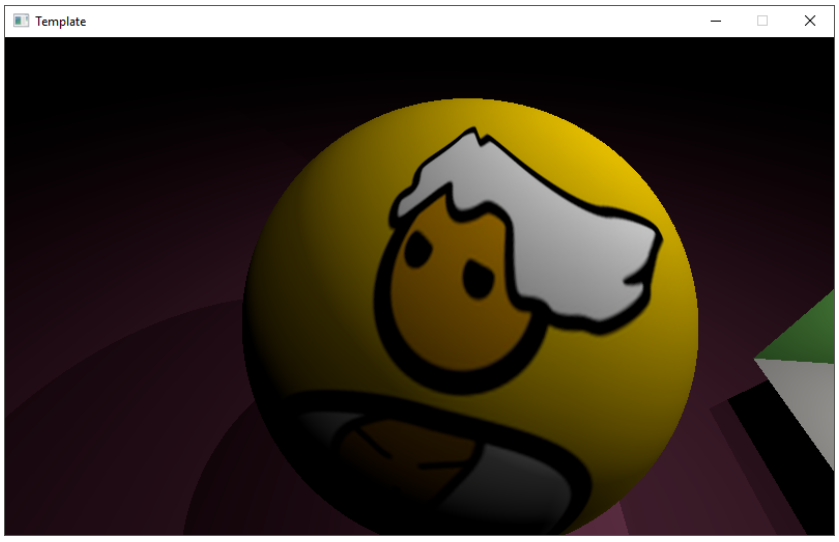
Texture support is a bit harder than on a CPU implemenation. The only way to pass a list of textures to OpenCL is by using an array of 2D images (`image2d_array_t`). This is stored in memory as a 3D texture so every sub-texture should be the same size (or smaller). An advantage of using OpenCL is that we can utilize the graphics hardware for texture filtering. We created a texture class which automatically rescales textures using FreeImage when they are loaded from disk. The texture class does not store the data itself. Instead the data is stored in a vector and the texture contains a index to it. This is useful because we need to have an index anyway for indexing in the image array in OpenCL. Additionally, we use an unordered map to ensure that we do not load the same texture twice.

Texture coordinates are calculated based on the shape type: for meshes, it's blended between the vertices as described in the previous section. For spheres, we used the standard uv coordinates for a sphere which are:

$$u = 0.5 + \frac{atan2(d_x, d_y)}{2\pi}$$

$$v = 0.5 - \frac{asin(d_y)}{2\pi}$$

We do not support textures on planes, as a plane is of infinite size and does not have a vector base to map textures to. We could define one such base, but it would defy the simplicity of the plane primitive. A plane-like object can be represented by a properly constructed mesh anyway.




# 8 Gamma Correction

A common pitfall for people that just started graphics programming is to overlook gamma correction. The sRGB colour gamut is the default color space in all operating systems and the only one in Windows. In our colour calculations we expect that if we multiply the luminance by two, the resulting brightness will be doubled as well. This is not the case when using the sRGB colour gamut.

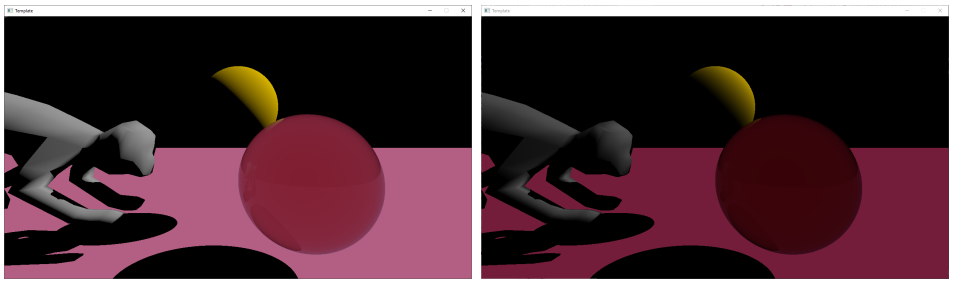


Figure 4: Gamma Correction (left: with, right: without)

The human eye does not perceive a increase or decrease in luminance in a linear fashion[1]. This is what the sRGB colour gamut corrects for. Digital cameras measure luminance but automatically convert and store it in a gamma corrected format. All of our colour calculations are performed on luminance values and as such, are linear. We can approximate the conversion to sRGB by taking the 2.2 log. A more precise conversion formula is provided by dice in the Moving Frostbite to PBR course notes[2] (page 87). To be completely gamma correct, we should ensure that the input images are also in a linear format. We use FreeImage for correcting textures using a 2.2 log

curve and Dice's accurate sRGB conversion for correcting the final image (listing 1).

Listing 1: Accurate linear to sRGB conversion

```
float3 accurateLinearToSRGB(float3 linearCol)
{
    float3 sRGBLo = linearCol * 12.92f;
    float3 sRGBHi = (pow(fabs(linearCol), 1.0f / 2.4f)
       * 1.055f) - 0.055f;
    return (linearCol <= 0.0031308f) ? sRGBLo : sRGBHi;
}
```

# 9 Conclusion

With only limited time available we managed to implement a fully featured Whitted style ray tracer. Furthermore, we were able to port it to the GPU using OpenCL and OpenGL interop. We are not adhering to the complete rendering equation. But we felt like adding gamma correction was an important building block for future development. Although we support planes and spheres, this support may drop in a future version as both primitives can be approximated using triangle meshes.

# References

[1] Cambridge in Colour, S. M. Understanding Gamm Correction, 2016.

[2] Electronic Arts Frostbite, Sébastien Lagarde, C. d. R. Moving Frostbite to Physically Based Rendering, 2014.

[3] Scratchapixel. Ray tracing, rendering a triangle.