GPU-based Ray Tracing

Daniel Bencic

July 2020

Abstract

In comparison to rasterization, ray tracing does focus more on the realistic simulation of light transport. This enables rendering of effects like shadows, reflection or refraction and creating photorealistic images. However the algorithm is very computational intensive, because for every pixel in the scene a ray has to be traced for intersections with all objects in the scene. Since this doesn't affect offline rendering very much, it is mainly used for rendering scenes offline, like rendering animated movies. There is currently a lot of research, mainly by game development companies and GPU manufacturers, to make ray tracing possible in real-time.

The aim of this paper was, to improve the performance of the execution of the ray tracing algorithm, to make the rendering of low resolution images possible in real-time. To achieve this the ray tracing algorithm was mapped to a GPU device using OpenCL. The generated image is displayed on the screen in real-time using OpenGL. The result is an interactive ray tracer, that can render images with a resolution of 800 x 800 pixels at more than 144 frames per second.

Introduction

Ray tracing is a technique to render a 3D scene onto a 2D image plane. It's an alternative to the, in todays computer graphics more commonly used, rasterization method.

The idea behind the ray tracing alogrithm is, to reverse the process of creating an image in the real world. Rather than tracing millions of rays from the light source and calculating if a ray hits a point on the image plane, a ray from every pixel of the image plane is shot into the scene (Figure 1). It can then be calculated if this particular ray hits an object and the color of the corresponding pixel can be determined [1]. One can see that this algorithm is very straightforward compared to the rasterization method. If additional concepts like reflection or refraction of rays are used, high quality, photorealistic images can be rendered [3].

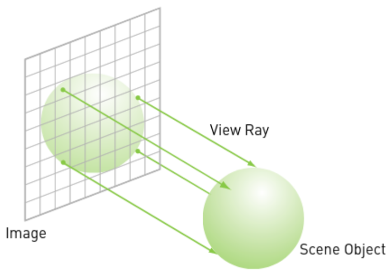


Figure 1: Concept of ray casting [2]

For simplicity reasons only the basic case of ray tracing - also called ray casting - is handled in this paper. That means the scene only contains primitive objects (e.g. spheres), no light source and rays are only traced to the first intersection. Let n be the width and m be the height of an image plane. Let c denote the number of primitive objects in the scene. Even the basic ray tracing algorithm has a time complexity of:

$$O(n * m * c)$$

As indicated by this formula, rendering large scenes in high resolutions with a CPU-based ray tracer can take quite some time. However, it turns out, that the ray tracing problem can easily be parallelized. This paper shows an implementation approach of a ray tracer, that uses the highly parallel nature of a GPU device to speed up the rendering process of a 3D scene, such that rendering of small low resolution scenes becomes possible in real-time.

Related Work

Purcell et al. [4] showed a way to implement the ray tracing algorithm on graphics hardware. They mapped the algorithm to the GPU by using programmable shaders for computations and textures to pass data and state between shader-calls.

Proposed Approach

Program Structure

This section gives an overview of how the ray tracing algorithm is transfered to the GPU. Figure 2 shows the top level program flow and if a part of the program is executed on the CPU or the GPU.

After starting the program, GLFW is initialized and an OpenGL context is created. Next the system is queried for OpenCL GPU devices and one device is selected. After setting up the external APIs, all necessary OpenGL objects are getting created as part of the initialization of the renderer module. The renderer module is described in more detail in section "Displaying the Scene". To prepare the computations on the GPU device the needed OpenCL objects are created by the ray tracer module. This step is explained in section "Computation". Then, the scene can be transfered to the GPU. As the scene does not change throughout the execution of the program this can be done before the rendering-loop to reduce memory access between CPU and GPU. In the rendering-loop first the wold parameters are queried from the renderer module and written to the buffer on the GPU. Afterwards the OpenCL kernel on the GPU is executed to render the scene using the ray tracing algorithm. The rendered image is read from the GPU and displayed on the screen. Finally the input from mouse and keyboard is processed as shown in section "Interactive Camera".

Computation

To map the ray tracing algorithm to the GPU the algorithm needs to be specified in greater detail. The input data of the algorithm consists of three main parameters:

1. Size of the image plane
2. Camera parameters

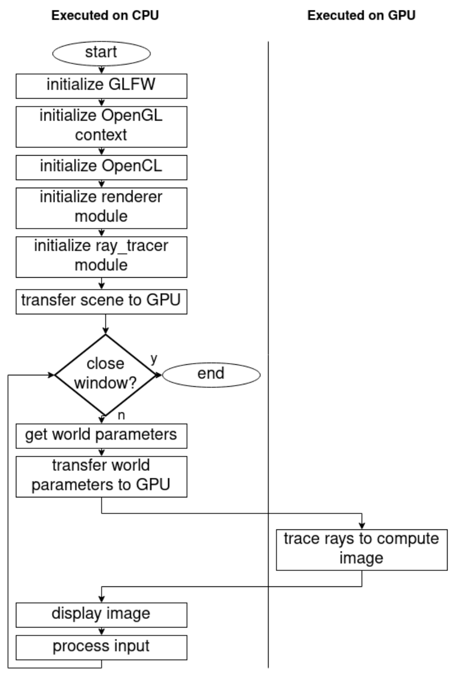


Figure 2: Program flow

3. Objects in the scene

The task of a ray tracer consists of the following subtasks, that need to be executed for every pixel of the image plane:

1. Generate ray through the center of the pixel
2. Trace ray to the first intersection
3. Shading of the pixel

As one can see, every pixel is treated for itself, so there are no dependencies between the pixel-tasks and they can be executed highly parallel. Due to the fact that every pixel-task performs the same calculations on different data, ray tracing is implemented as a data-parallel algorithm on the GPU. Every pixel-task can be seen as a map operation from pixel-coordinates to a RGB color (Figure 3). The subtasks of a pixel-task are executed in one kernel. They logically group together and form a pipeline, where each subtask depends on the previous one. A more fine-grained parallelism would therefore only increase the communication overhead.

The ray tracer module is responsible for setting up the computation on the GPU with OpenCL. The input

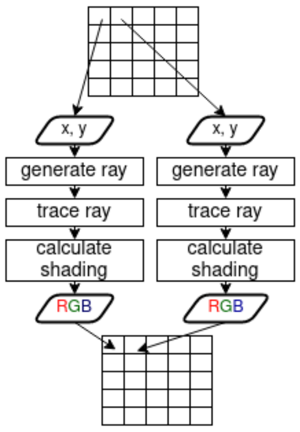


Figure 3: Mapping pixel-coordinates to RGB color

data for the algorithm does not get modified throughout kernel execution. This fact allows for storing all input data in constant memory. The module creates two input buffers, one for the objects contained in the scene and one for the world parameters (camera parameters and image plane size). Both are declared __constant. It also creates a global output buffer, which stores one RGB value for every pixel. The size of the output buffer is calculated by

$$size = width * height * 3$$

OpenCL uses the concept of global work-items and local work-groups. The global work-items size for the ray tracing algorithm is two dimensional and defined by the width and height of the image plane. The local work-group size should be a multiple of the GPU devices SIMD width to efficiently execute tasks as a wavefront [5]. The ray tracer queries this size and factorizes it into a two dimensional work-group size (Listing 1).

```
void calc_work_group_dims(const
    size_t max, size_t *dim0,
    size_t *dim1)
{
     uint mod;
     *dim0 = sqrt(max);
     while(mod = max % *dim0)
          --*dim0;
     *dim1 = max / *dim0;
}
```

Listing 1: SIMD width factorization

Displaying the Scene

For displaying the rendered image to the screen a textured quad is used. It is composed out of two triangles and spans the whole window.

The renderer module does create a vertex array object and a vertex buffer object, containing the four vertices for the textured quad. It also creates an OpenGL texture, which is used to get the generated image into the OpenGL pipeline, after it has be retrieved from the ray tracer module. The texture is sampled to the quad in the fragment shader to display the result on the screen.

Methods and Algorithms

Generating a random Scene

To evaluate different kind of scenes the renderer module creates a scene by randomly generating the specified number of spheres. To create the spheres random values for position, color and size are calculated. For the position, real numbers for x- and y-coordinates are calculated by the following formula for real random values:

$$f(b) = -1 + 2 * (rand()/RAND\_MAX) * b,$$

where b is the positive and negative bound. The cameras default location is $(0.0, 0.0, 0.0)^T$ and it is looking in direction of the negative z-axis. So the value for the z coordinate of the sphere is the negated result of

$$f(b) = (rand()/RAND\_MAX) * b$$

This formula for calculating real positive random values is also used for determining random RGB values and the size of the sphere.

Generating Rays

To get a more realistic image, the rays do not have their origin in the center of each pixel, but they originate at the position of the camera and travel throught the center of a pixel into the scene.

A ray is represented with the parametric equation of a line [1]:

$$R(t) = R_0 + t * R_d$$

The two points in space needed to define a ray are first computed in camera space. The origin of the ray is known, because it is equivalent to the position of the camera, which is always $(0.0, 0.0, 0.0)^T$. To calculate the direction of the ray, the center of the pixel needs to be calculated. The distance between the position of the camera and the image plane is defined to be 1. That means the z-value of the pixel-center is -1. For calculating the x- and y-value of the pixel-center, the raster coordinates are first getting transformed to NDC space (ranging from -1.0 to 1.0):

$$NDC_x = 2 * \frac{x + 0.5}{width} - 1$$

$$NDC_y = 1 - 2 * \frac{y + 0.5}{height}$$

Afterwards the pixel is scaled according to the field of view. Since the distance from the camera orign to the image plane is 1, the scaling factor can easily computed with

$$scale = tan(\frac{fov}{2})$$

The last correction that needs to be done is, scaling the x-coordinate based on the aspect ratio, to not only have squared pixels. The pixel-center is then given by:

$$CAM_x = (2 * \frac{x + 0.5}{width} - 1) * tan(\frac{fov}{2}) * \frac{width}{height}$$

$$CAM_y = (1 - 2 * \frac{x + 0.5}{width}) * tan(\frac{fov}{2})$$

$$CAM_z = -1$$

By transforming R_0 and R_d to world space with the camera-to-world matrix, substracting R_0 from R_d and normalizing R_d, the complete ray equation in world space is known.

Tracing Rays

Tracing a ray means finding the first object it intersects with, as it travels through the scene. To achieve this every object in the scene has to be checked for intersection with the ray. If the ray intersects an object, a reference to this object is stored until a closer intersected object is found. Since the objects in the scene are only spheres (implicit objects) the calculation of a ray-sphere intersection can be done fast with an algebraic solution.

The implicit equation for the surface of a sphere centered at $S_c = (X_c, Y_c, Z_c)^T$ and the radius S_r is:

$$(X_s - X_c)^2 + (Y_s - Y_c)^2 + (Z_s - Z_c)^2 = S_r^2$$

To find out the intersection point between the ray and the sphere the left side of the equation gets substituted with the ray equation.

$$||R_0 + t * R_d - S_c||^2 = S_r^2$$

This equation can be simplyfied to a second-degree polynomial equation in terms of t:

$$A * t^2 + B * t + C = 0$$

with

$$A = ||R_d||^2 = 1$$

$$B = 2 * R_d^T (R_0 - S_c)$$

$$C = ||R_0 - S_c||^2 - S_r^2$$

It can then easily be solved with the quadratic equation. If the equation has one or two solutions, the value of t represents the distance to the intersection point(s), since the direction vector R_d is normalized. As the final step t is stored if it is smaller than the currently smallest distance to an intersected sphere.

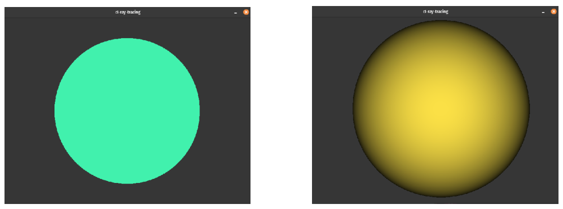
Shading Pixels

The final step in generating a rendered image with ray tracing is shading the corresponding pixel based on the result of tracing a ray. If the ray does not intersect any object in the scene the pixel gets shaded with the color of the background of the scene. If the ray does intersect with one or more spheres, the color of the closest hit sphere is the basis for shading the pixel. If every pixel is just shaded with this color the look of the sphere is a completely flat circle (Figure 4). To add the effect of a real sphere the angle at the intersection point is calculated. The normal at the point of intersection is given by

$$N_h = P_h - S_c$$

The basis color can then be adjusted with a shading factor:

$$sh = max(0.0, N_h^T (-R_d))$$



(a) Flat circle look (b) 3D sphere look

Figure 4: Improved shading

Interactive Camera

An interactive camera is used to test the performance of the algorithm, by experiencing the rendered result in real-time. The renderer module holds a camera object, which has members to store the position, the view direction and a vector, which represents the up direction of the camera coordinate system. A camera-to-world matrix can then be calculated by first finding the right vector of the camera coordinate system with the cross product of a temporary up vector $TMP_U = (0.0, 1.0, 0.0)^T$ and the front vector:

$$C_R = TMP_U \times C_F$$

The camera can only be rotated around the y- and x-axis. Because of that the temporary up vector and the front vector of the camera always span a plane perpendicular to the right vector of the camera coordinate system. After the right vector is known, the correct up vector of the camera coordinate system can be calculated as

$$C_U = C_R \times C_F$$

Finally a look-at matrix can be formed, that represents the camera-to-world transformation:

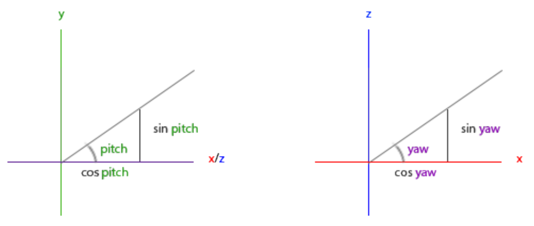
$$\begin{bmatrix} C_Rx & C_Ux & C_Fx & C_Px \\ C_Ry & C_Uy & C_Fy & C_Py \\ C_Rx & C_Ux & C_Fx & C_Px \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

This matrix is calculated every frame and sent to the GPU as a world parameter. Making the camera interactive is achieved by processing mouse and keyboard input. Keyboard input is used to move the camera around in the horizontal plane, while mouse input rotates the camera around the y- and x-axis. Movement of the camera is achieved by changing the position of the camera in direction of C_F and C_R, based on the pressed key (W = front, A = left, D = right, S = back):

```
// sp = speed of camera
switch (dir) {
    case front:
        mult_vec(&d, sp);
        add_vec(&r->cam.pos, &d);
        break;
    case left:
        mult_vec(&ri, sp);
        sub_vec(&r->cam.pos, &ri)
            ;
        break;
    case right:
        mult_vec(&ri, sp);
        add_vec(&r->cam.pos, &ri)
            ;
        break;
    case back:
        mult_vec(&d, sp);
        sub_vec(&r->cam.pos, &d);
        break;
}
```

Listing 2: Camera movement

For rotating the camera, first the changes in x- and y-direction of the mouse are calculated. The change in y-direction is then used to adjust the pitch (rotation around the x-axis). It is also checked, whether the pitch angle is greater than 89.9 degrees or lower than -89.9 degrees to prevent a gimble lock. The change in x-direction changes the yaw angle and therefore rotates the camera around the y-axis.



(a) Pitch (b) Yaw

Figure 5: Calculating the new camera direction [6]

Figure 5 shows how the the pitch and yaw angle influences the new direction (front) vector of the camera coordinate system. The new direction vector is calculated according to Listing 3 and stored in the camera object.

```
r->cam.dir.x =
     cos(rad(r->cam.yaw))
     *
     cos(rad(r->cam.pitch));
r->cam.dir.y =
     sin(rad(r->cam.pitch));
r->cam.dir.z =
     sin(rad(r->cam.yaw))
     *
     cos(rad(r->cam.pitch));
```

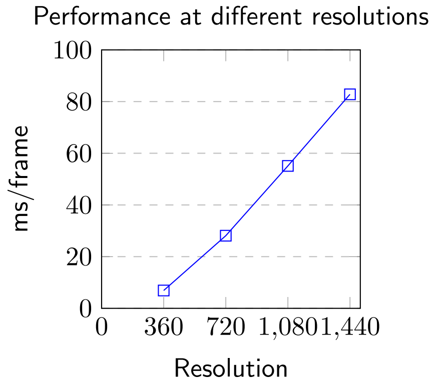
Listing 3: Camera rotation

Evaluation

This section shows different test cases, where the performance of the developed algorithm is analyzed. The performance is measured as the time in milliseconds needed to render a frame. For every testcase 1000 frames are rendered while the camera is moved around in the scene.

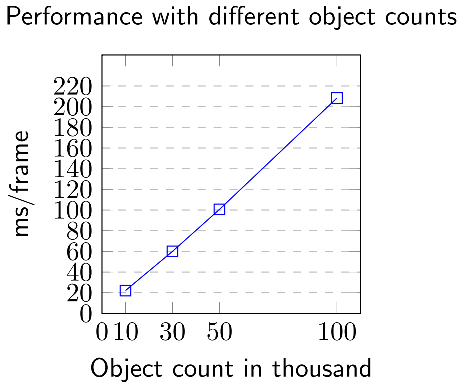
Resolution

The resolution is one of the parameters, which has a huge impact on the computations needed to render the scene, since a ray has to be generated and traced for every pixel. The following graph shows the result of different 16:9 resolutions. The value for 360p is limited to 6.9 ms/frame (144 FPS) by the refresh-rate of the monitor, which means the real value is even lower. As expected the performance is getting worse rapidly with increasing resolution, because doubling the resolution means four times more rays to trace.



Object Count

The second parameter affecting the rendering-performance is the amount of objects in the scene. For this test case the algorithm is tested with a fixed resolution of 800 x 800 with a rising count of objects in the scene.



The time to render a frame, as anticipated, is growing linearly with the the amount of objects in the scene. It can be seen, that for implicit objects the intersection calculations are cheap, and even with 10000 objects in the scene a 800 x 800 image can be rendered at 45 FPS.

Conclusion

This paper showed that the ray tracing algorithm can be speeded up by mapping it to the GPU. The algorithm can be parallelized in an easy way, because tracing different rays does not required communication or synchronization between tasks and the input data remains constant. Furthermore it showed, that real-time ray tracing for low resolution images with a lot of implicit objects is possible.

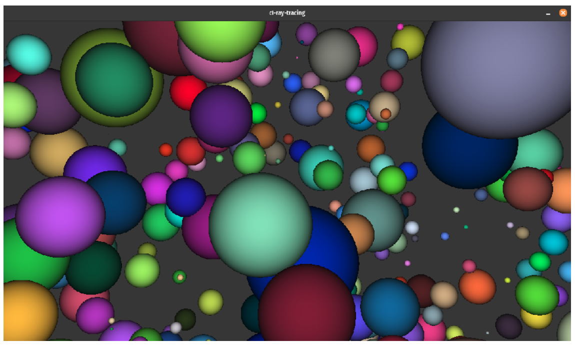


Figure 7: Ray tracing with many objects

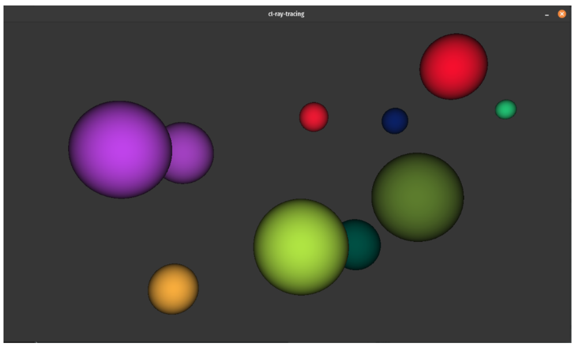


Figure 6: Basic ray tracing

The algorithm presented does only handle the simplest form of ray tracing. In further work a more advanded algorithm can be tested by adding light-sources, tracing rays further than the first object they hit, adding better shading models or tracing non-implicit surfaces like meshs.

References

[1] A. S. Glassner, *An Introduction to ray tracing*. Academic Press, 1989.

[2] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley, 2011.

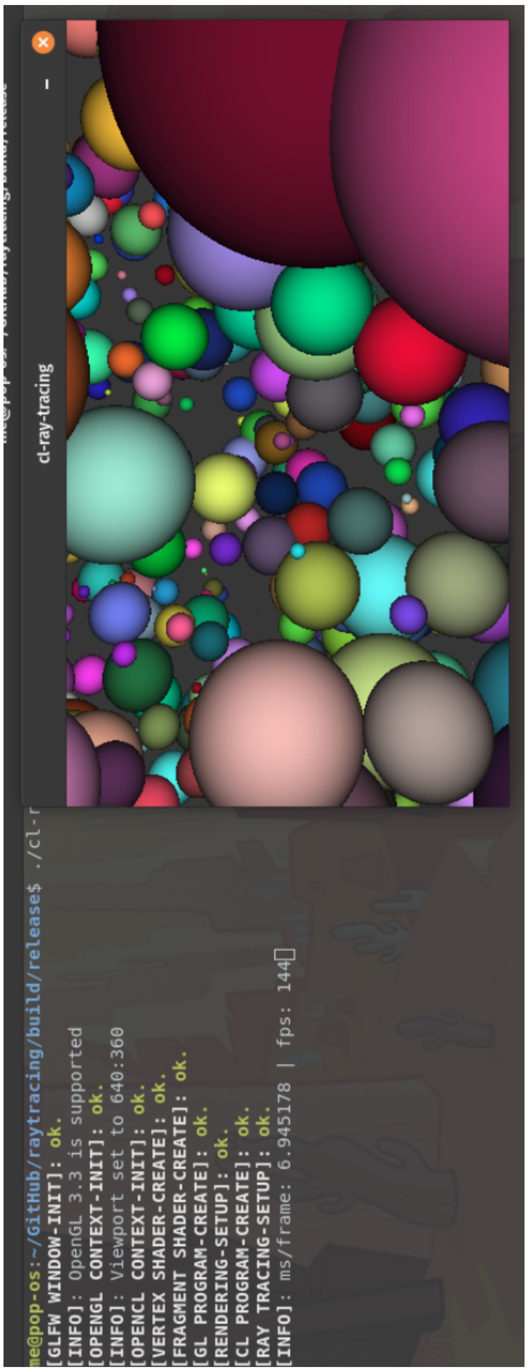
[3] R. Kuchkuda, "An introduction to ray tracing," *Theoretical Foundations of Computer Graphics and CAD*, p. 1039–1060, 1988.

[4] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, p. 703–712, 2002.

[5] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, 2015.

[6] J. de Vries, *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion.* Kendall & Welling, 2020.

Appendix



```
me@pop-os:~/GitHub/raytracing/build/release$ ./cl-r
[GLFW WINDOW-INIT]: ok.
[INFO]: OpenGL 3.3 is supported
[OPENGL CONTEXT-INIT]: ok.
[INFO]: Viewport set to 640:360
[OPENCL CONTEXT-INIT]: ok.
[VERTEX SHADER-CREATE]: ok.
[FRAGMENT SHADER-CREATE]: ok.
[GL PROGRAM-CREATE]: ok.
[RENDERING-SETUP]: ok.
[CL PROGRAM-CREATE]: ok.
[RAY TRACING-SETUP]: ok.
[INFO]: ms/frame: 6.945178 | fps: 144
```

Program execution