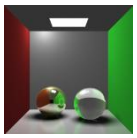


2019/2020, 4th quarter

INFOGR: Graphics

Practical 1: Ray Tracing

The assignment:

The purpose of this assignment is to create a ray tracer. You have plenty of choice in the execution of this assignment. In its most basic form, you create a 2D ray tracer. You can extend this in various ways, including a full 3D ray tracer.

Before we dive into the details, here are the formalities related to submission:

- Your code should compile and run out-of-the-box. To prevent disappointments, consider testing the package you are about to hand in on a different machine. If your code fails this requirement, points will be deducted, and we may not be able to grade your work at all.
- Please **clean** your solution before submitting (i.e. remove all compiled files and intermediate output). To do this, run `clean.bat` (included with the template), while Visual Studio is closed (VS tends to lock some files). After this you can zip the solution directories and send them over. If your zip-file is more than 10 megabytes in size something went wrong (not cleaned properly).
- Add a readme.txt which provides some information about your project. State how the work was distributed between you and your partner, what features you implemented (don't make us guess), how the application is operated (if this is not trivial) and what sources / materials / documents you used.
- Submit using Teams. If this is not possible, an alternative will be provided.
- You can work on this assignment alone, or with a partner. Larger groups are **not** allowed. There is no penalty for working alone.

Grading:

If you implement the minimum requirements, and stick to the above rules, you get a 6. Implement additional features to obtain additional points (up to a 10).

From the base grade of 6, we deduct points for a missing readme, a solution that was not cleaned, a solution that does not compile, or a solution that crashes.

Retaking?

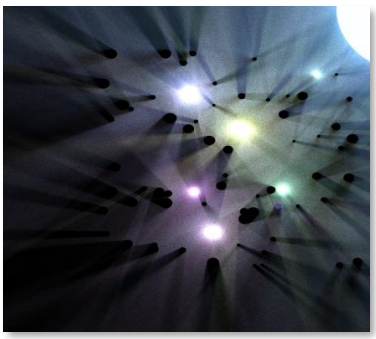
If you are retaking INFOGR, and passed the assignments last year, please notify me at bikker.j@gmail.com. In this case you may use your grades from last year. *Older results can not be used*.

Deadline:

The deadline for this project is **Thursday, May 28, 2020, 23:59h.** If you miss the deadline, you can use one of the extended deadlines: up to 12 hours later at the cost of 0.5 points, or up to 24 hours later at the cost of 1.0 points.

High-level Outline

For this assignment you will implement a 2D ray tracer. To get an impression of what this means, have a look at the following renders:



The left scene consists of circles which block light, and an omnipresent floor, which visualizes light density at each point on 2D space. Inside the circles the light is absent. Outside the circles, the light is the sum of the contributions of all light sources, considering occlusions and distance to each light source.

We can make the scene more interesting by adding other shapes, such as lines. We can also texture the floor, and add other materials for the primitives: e.g., a glass sphere would focus light, and a mirror plane would reflect light. This turns out to be non-trivial to implement though.

For the basic functionality we have the following bare necessities:

- a floating point frame buffer
- a loop over the pixels of the screen
- a ray class that we use to establish visibility of each light source
- for each pixel, a loop over the light sources, in which we estimate the 'potential contribution' of each light
- for each pixel, for each light, a loop over the primitives, to check if a ray intersects the primitive, in which case the primitive is an occluder
- code to intersect a ray with a 2D primitive
- code to turn the final floating point colors to integer pixel colors.

In pseudo-code:

```
for each pixel
  pixelColor = { 0, 0, 0 }
  for each light
    ray.O = pixelPosition()
    ray.D = normalizedDirectionToLight()
    ray.t = distanceToLight()
    bool occluded = false
    for each primitive p
      if (ray.intersects( p )) occluded = true
    if (!occluded)
      pixelColor += light.color * lightAttenuation( distanceToLight() )
  screen.Plot( x, y, ToRGB32( pixelColor ) )
```

A correct translation of this code to C# inside the template yields a program that satisfies the minimum requirements. *Note that there is no performance requirement for this application.*

The Fun Starts

That being said… With the basics in place there is still a ton to improve. Here is a list:

- Speed up the application. Processing of individual pixels is completely independent, so it is trivial to multi-thread the code. Once the code is fast enough you can start animating the lights for some interesting animations. If multi-threading is not enough, consider rendering at half the resolution. Specify in your readme.txt what you did to improve performance.
- Improve the visual fidelity. Point-sampled lights yield jagged shadow edges. This is however not the case if when you estimate the visibility of each light source using multiple shadow rays, which originate on multiple positions within a single pixel. This is the basic idea of anti-aliasing.
- Add area lights. Point lights yield hard shadow edges. An area light casts soft shadows. To implement this, cast rays to random positions on the perimeter of each area light. Visibility is again the average of the result of each occlusion query.
- Decouple screen resolution from world size. Chances are that you expressed all coordinates in pixel coordinates. This is not ideal; resizing the window or switching to a 4k monitor effectively make your app useless. Express all coordinates in a range of e.g. -1 … +1 over x and y, and translate these coordinates to screen coordinates. As a bonus, you can now easily zoom in and out.
- Texture the floor. Some cool wood, sand or grass should do the job.
- Have a cool scene. A box and two circles does the job for the minimum requirements, but where's the fun in that? Setup something awesome.

To add some extra incentive (beyond joy and honor), the first four items of the above list each score an additional 0.5 points when implemented correctly, potentially raising your grade to an 8. Clearly indicate implemented features in your readme!

The Fun Continues

If you are looking for a bit of a challenge, consider the following quite hard bonus assignments. You probably will have to do some research for these.

- For ultimate performance, translate the code to a shader and run the whole thing on the GPU. Two bonus points if you manage this. Warning: very hard with the knowledge you have at this point of your education.
- Add normal mapping to the floor. Look up the technique and figure out a way to add it to a 2D ray tracer. One bonus point for a good-looking implementation.
- Take the ray tracer to 3D, but without spoiling the 2D look. Have a camera that always looks down on a relatively flat world, but use real spheres and planes instead of lines. The easiest way to tell that you were successful is that the floor can now be reflective: this was not possible in the pure 2D case. Two bonus points if you pull this off.
- Rays are not just useful for light propagation. They are great for physics too. To demonstrate this, add one or more moving balls to the scene that respond correctly to collisions with the geometry. One bonus point for a basic physics demo. Two bonus points for a little game that uses physics and 2D ray tracing. Pinball perhaps?

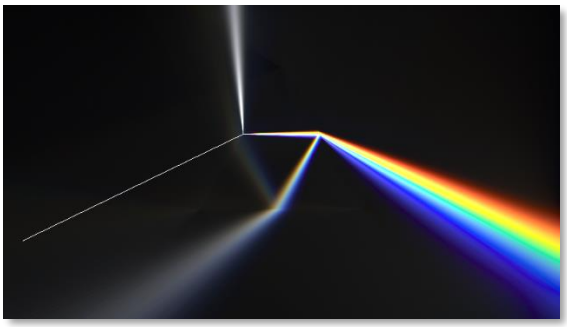
If you're still reading you probably already scored a 10. There is one more set of challenges though.

For Honor

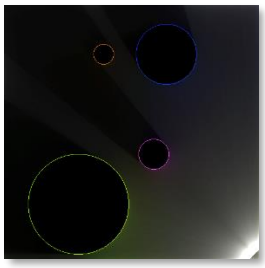
If you need a serious challenge, implement one of the following:

1. Add reflection and refraction to the 2D ray tracer. Important: do not go to 3D for this. The idea is that a glass sphere focuses light, while mirror objects reflect light. To calculate images for this in 2D you will need to start light paths at the light sources instead of at the pixels. Distribute a very large number of particles ('photons') through the scene to get a good estimate of the average light density at each pixel. The image below is an example of 2D caustics resulting from glass circles.

   If you feel particularly adventurous, add spectral rendering. This time each particle has a wavelength, which affects the index of refraction of many materials, such as crystal. This yields pretty rainbows:



2. Create a 2D path tracer. This time, every point on the 2D plane is not only lit by the light sources, but also by light bouncing off of the other geometry in the scene. To estimate this indirect light, send a few (e.g. 16) rays in random directions to find nearby geometry. When you find nearby geometry, calculate how much energy this geometry reflects towards the original 2D point. Research the topic of path tracing for this.



Specials

Like last year, we have some special rewards this year.

1. Fastest ray tracer

Use every trick from the book to make your 2D ray tracer run as quickly as possible. GPGPU, multithreading, SIMD, even RTX; everything is on the table. For fairness, we need to specify the scene: use a 5x5 square room with three non-overlapping circles of radius 1 in it, and add four point lights, one in each corner of the room. Performance will be measured on my 2-core (4 thread) desktop with a RTX 2080 GPU. Win the award for **one bonus point (up to a 10)**. And glory. *Lots of glory.*

2. Smallest ray tracer

Reduce the size of your 2D ray tracer to as few characters as possible. The resulting ray tracer must meet the minimum requirements of this assignment. You may implement the tiny tracer in a programming language of your choice. Win the award for **one bonus point (up to a 10)** and pride.

2a: reward 2 interpreted as ‘smallest source file’.
2b: reward 2 interpreted as ‘smallest executable’.

And Finally,

Don’t forget to have fun; make something beautiful!

May the Light be with you,

- *Jacco.*