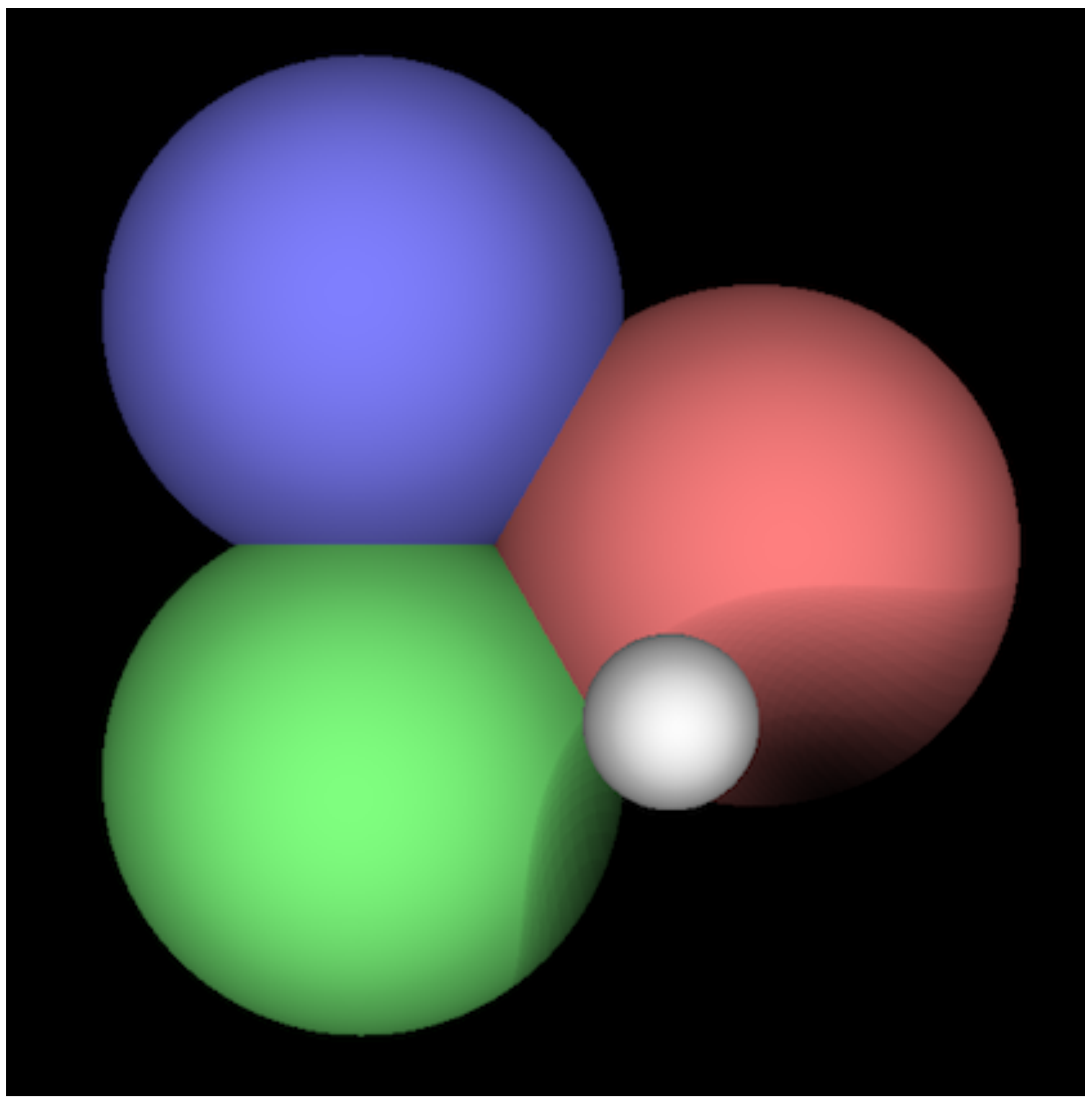
Project 3: vtray

Assigned:	4/11
Beta Version Due:	4/25 at 8 am
Final Version Due:	5/2 at 11:59 pm

GitHub Invitation Link (https://classroom.github.com/assignment-invitations/f7211a4632342e36b3cf9ce2cb3c8c5d). Accept the GitHub invitation, wait until you get an email saying the import is complete, and then clone the git repository to your local machine.

The goal of this project is to gain practice designing and implementing an application using threads. You will be implementing a multi-threaded ray-tracing rendering engine named `vtray` that produces images like the following.

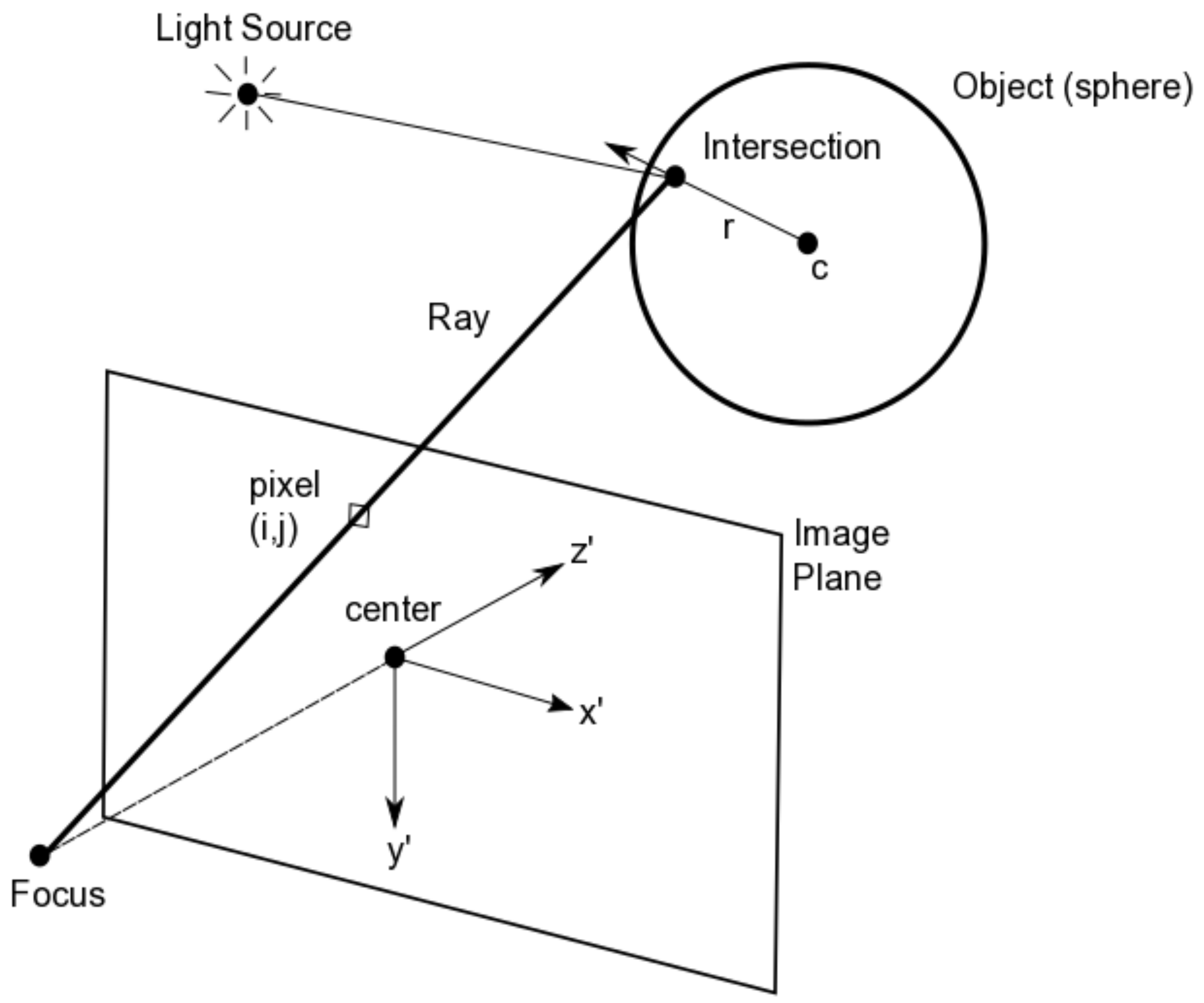


Introduction

Ray tracing (https://en.wikipedia.org/wiki/Ray_tracing_(graphics)) is a technique for photo-realistic rendering of virtual scenes. It is commonly used in the film industry (http://graphics.pixar.com/library/RayTracingCars/paper.pdf) and is based on a simplified physics-based model of low light interacts with objects in a scene.

As you know, photons of light emanating from a source and striking an object can be absorbed, reflected, or refracted depending on the material properties and photon wavelengths. The accumulations of photons at a detector are what determine the color seen. We can simplify this model using the ray-tracing idea from optics, treating the light as a ray, and reversing the order from source-object-detector to detector-object, source. In our program we will ignore refraction and combine absorption and reflection into a simple formula. More sophisticated ray tracers include these components as well as secondary reflections (reflections of reflections), optical transfer functions, etc., and can render quite realistic scenes (http://hof.povray.org/).

Consider the following illustration. A simple object, a sphere in this case, is illuminated by a light source. A camera is observing this scene, defined as an image plane with a focus some perpendicular distance from the center of the image plane. The camera is oriented in a right hand coordinate system with a normal vector z' as indicated.



The image plane is divided into N x M pixels, each with a center in the global coordinate system. For each pixel, a *primary* ray is cast from the focal point through the pixel center into the scene. If the ray does not intersect any object in the scene then the color is set to an RGB value of (0,0,0) indicating no light. If the ray intersects the object a *shadow* ray is cast from the intersection to each light source to determine the color at the pixel. If the shadow ray does not intersect any other object in the scene then the color depends on the light intensity, the object material, and the dot-product between the surface normal and normalized shadow ray vector. In our simple ray tracer the material properties are simply an RGB value indicating the color of the object, and the Lambert coefficient of the surface, a number between 0 and 1 determining how much of the light is reflected. If the shadow ray does intersect an object then the color contribution from that light source is none. The color contributions from multiple light sources is additive and scaled by the maximum intensity in the rendered image (autoexposure). In pseudo-code, the color at a non-shadow intersection from a single light source is:

```
scale = dot_product(surface_normal, shadow_ray)*surface_lambert
pixel_color = scale*light_intensity*surface_color;
```

For simplicity, our scene descriptions will only include spheres and planes as objects. A sphere is defined by a center, radius, color, and Lambert coefficient. A plane is defined by a center, a normal, a color, and a Lambert coefficient. The camera is defined by a center, normal, focal depth, number of pixels in each direction, and the size of each pixel. Objects, light sources, and the camera parameters will be read from a configuration file in JSON (http://www.json.org/) format, producing a PNG formatted image of the scene.

For more background on ray tracing see one of the many tutorials available (https://www.google.com/search?q=ray%20tracing%20tutorial).

Concurrency

Ray tracing is computationally expensive, so it is a good candidate for leveraging hardware concurrency (parallelism). Since each primary ray is independent, this is relatively easy. However, some care must be taken in the design to minimize the overhead of thread creation and lock contention, while still maintaining properly ordered memory access. **You must use C++11 threads** to implement your concurrency rather than Qt's threading components (`QThreads`). You can reuse and adapt any of the threading example code from the lectures.

Important Note: You should have a working version without threading completed prior to adding concurrency. This will make debugging much easier.

Scene and Camera File format

The scene and camera are specified in a JSON file with the following key-value pairs.

1. A single key "objects", value JSON array of objects where each object has key-values

- key "type", value string equal to either "sphere" or "plane"
- if type is sphere then there are key-values
  - key "center", value JSON object with keys "x", "y", and "z" with double values
  - key "radius", value double
  - key "color", value JSON object with keys "r", "g", and "b" with integer values
  - key "lambert", value double
- if type is plane then there are key-values
  - key "center", value JSON object with keys "x", "y", and "z" with double values
  - key "normal", value JSON object with keys "x", "y", and "z" with double values
  - key "color", value JSON object with keys "r", "g", and "b" with integer values
  - key "lambert", value double

2. A single key "lights", value JSON array of lights where each object has key-values

- key "location", value JSON object with keys "x", "y", and "z" with double values
- key "intensity", value double

3. A single key "camera" with key-values

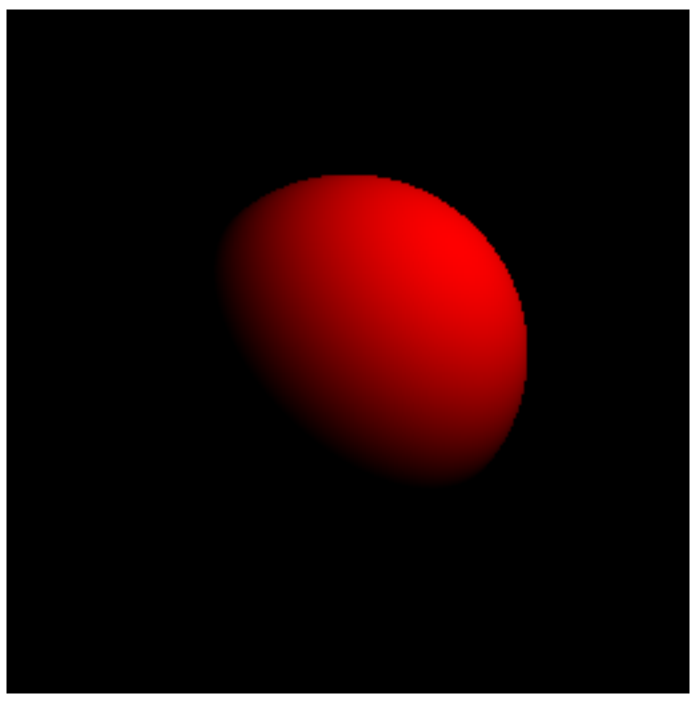
- key "center"

- key "normal"
- key "focus"
- key "size", values JSON integer array of size 2
- key "resolution", values JSON double array of size 2

The following is an example input file (`scene0.json`) with a single sphere, light, and camera.

```
{
    "camera": {
        "center": { "x": 0, "y": 0, "z": 0},
        "focus": 10,
        "normal": { "x": 0, "y": 0, "z": 1},
        "resolution": [0.01, 0.01],
        "size": [256,256]
    },
    "lights": [
        {
            "intensity": 1,
            "location": { "x": 5, "y": -5, "z": 0}
        }
    ],
    "objects": [
        {
            "center": { "x": 0, "y": 0, "z": 5},
            "color": {"b": 0, "g": 0, "r": 255},
            "lambert": 1,
            "radius": 1,
            "type": "sphere"
        }
    ]
}
```

When run through vtray this should produce the following image.



Error Handling

When parsing the JSON file:

- valid spatial coordinates are any double
- valid radii are non-negative floating point numbers
- valid Lambert coefficients are floating point numbers in [0,1]
- valid colors components are integers in [0,255]
- valid light intensities are non-negative floating point numbers
- valid size values are any positive integer
- valid resolution are any positive double

Your program should produce an error if the input file does not meet the above specifications.

Module Specifications

You will be using Qt for the JSON parsing and PNG output, but the rest of the code should be yours. In this assignment you will need to select, design, implement, and test your own modules. These decisions should be documented in a **plain text** file called README.txt in the top-level of the repository.

Program Specifications

The program is a single executable named vtray (vtray.exe on Windows) that takes three command-line arguments: an optional number of rendering threads to use (defaults to 1), the JSON file name that contains the scene and camera definition, and the output image file name. For example on a Unix system, to render the scene described by the file scene.json into the file named scene.png using two rendering threads one would run (> is the prompt):

```
> ./vtray -t 2 scene.json scene.png
```

As another example, on a Windows system, to render the scene described by the file scene.json into the file named scene.png using the default number of rendering threads, one could run (> is the prompt):

```
> .\vtray.exe scene.json scene.png
```

If any error occurs during processing (cannot open files, parsing errors, etc.) the program should print a message that begins with "Error" to **standard error** and exit with `EXIT_FAILURE`. Otherwise, it should write the output file and exit with `EXIT_SUCCESS`.

Unit and Integration Testing

For each module of code you should have a set of unit tests covering its functionality using Catch. These tests should be built as part of your overall project using CMake as described below. A file named `unittests.cpp` is provided with examples that should be replaced by your code. You may divide your unit tests into multiple files -- simply create a new cpp file that includes `catch.hpp` and implements the tests and add the file name the the `test_src` list in `CMakeLists.txt`.

The provided cmake file includes four tests with scenes of varying complexity and correctness (tests``scene*.json `). On your host, building/running the tests will run your vtray program on the scenes and should produce output files in the build` is also run to automatically verify the similarity of your output to that expected. Since slight implementation differences and rounding issues can prevent the images from being identical at he pixel level this might lead to false negatives, but it is a reasonable check that your images are correct.

Using CMake to build and test your software

The initial repository contains a skeleton `CMakeLists.txt` file. You will need to modify this file. See the sections at the top marked `EDIT`. In the appropriate section add any source files you create. Note the only required files are `vtray.cpp`, which implements `main`, and `untitests.cpp`, which sets up the test. How you name your files and partition your code is a design decision for you to make. However, you should stick to lower-case only filenames and the `.hpp` `.cpp` suffix convention. Be sure to add those files in the appropriate place in the cmake configuration file.

Grading Environment

The initial repository includes a Vagrantfile that sets up a virtual machine with minimal graphical interaction enabled. As in earlier projects this is a duplicate of the environment used for grading. This environment adds the ability to display images in a separate window from the shell to aid debugging inside the VM. After initializing the environment with `vagrant up` you will need to reboot the machine, e.g. `vagrant halt`, followed by `vagrant up` again. This need be done only once (and after any vagrant destroy).

This should allow all parts of the software and tests to be built and run. In the virtual machine this translates to the following:

```
> cd ~
> cmake /vagrant
> make
> make test
```

This treats the source directory as the shared host directory (`/vagrant`) and places the build in the home directory of the virtual machine user (`/home/ubuntu`). Using CMake on your host system will vary slightly by platform and compiler/IDE. To view the images in the VM you can use the `feh` program, e.g.

```
> feh /vagrant/tests/scene0.png
```

Note: just type Cntrl-C to exit the viewer.

On your host system you can use any image viewer you like. Note the VM uses only a single CPU, so while using threading should work, you cannot expect performace to scale well. As it is likely you host machine has more than 1 core you should see some performance improvement when running there.

As in previous projects the VM and provided CMakeLists.txt file is setup to run memory and code quality checks and code coverage analysis.

To run the coverage analysis in the VM and copy it back to the host filesystem

```
cd ~
cmake -DCOVERAGE=TRUE /vagrant
make
make coverage
cp -r Coverage_Report /vagrant
```

To run the memory check in the VM

```
cd ~
cmake -DMEMORY=TRUE /vagrant
make
make memtest
```

To run the code quality checks in the VM

```
cd ~
cmake -DTIDY=TRUE /vagrant
make
make tidy
```

Submission

Your submission must consist of a git repository containing **only** the source code and any supporting test and configuration files (Vangrantfile, CMakeLists.txt, tests directory and other files provided in the starter repository).

To submit your assignment, either the beta or final versions:

1. Tag the git commit that you wish to be considered for grading as "beta" or "final".

   ```
   git tag <tagname>
   ```

   If you want to have multiple versions of the tags, name them sequentially, i.e. beta, beta2, beta3, final, final2, final3, etc. We will consider the last one present in the repository when it is pulled for grading.

2. Push this change to GitHub

```
git push origin <tagname>
```

Be sure you have committed all the changes you intend to by the respective due dates. **Failure to complete these steps by the due date will result in no credit being assigned.**

Grading

There are 100 points allocated to this assignment distributed among the beta and final versions.

	Beta	Final
Correctness	15	35
Performance	-	20
Testing	10	10
Code Quality	-	10

- The correctness score is determined by the similarity of the output images to that expected. Your program should have no memory errors.
- The performance score is determined by the ratio of the time required to complete the rendering for increasing values of t, relative to t=1. This will be run on a multicore machine.
- The testing score is determined by the quality of your unit tests, that is the amount of coverage they test.
- The code quality score is determined by the code compiling cleanly with no warnings (at the same warning level as P0-P2) as well as a other criteria that will be provided as part of the beta grading.

Note: Some partial credit may be assigned based on evidence of work and good incremental development practice, as exhibited by your git commit history. Be sure to record your development progress via commits. At the very least this acts as a backup of your work.