karansher / computer-graphics-ray-tracing Public

forked from alecjacobson/computer-graphics-ray-tracing

Computer Graphics Assignment about Ray Tracing

0 stars 26 forks

Star Watch

Code Issues 26 Pull requests Actions Projects Security Insights

master

This branch is up to date with alecjacobson/computer-graphics-ray-tracing:master.

rarora7777 Delete README.html ... on Dec 8, 2019 42

View code

Computer Graphics – Ray Tracing

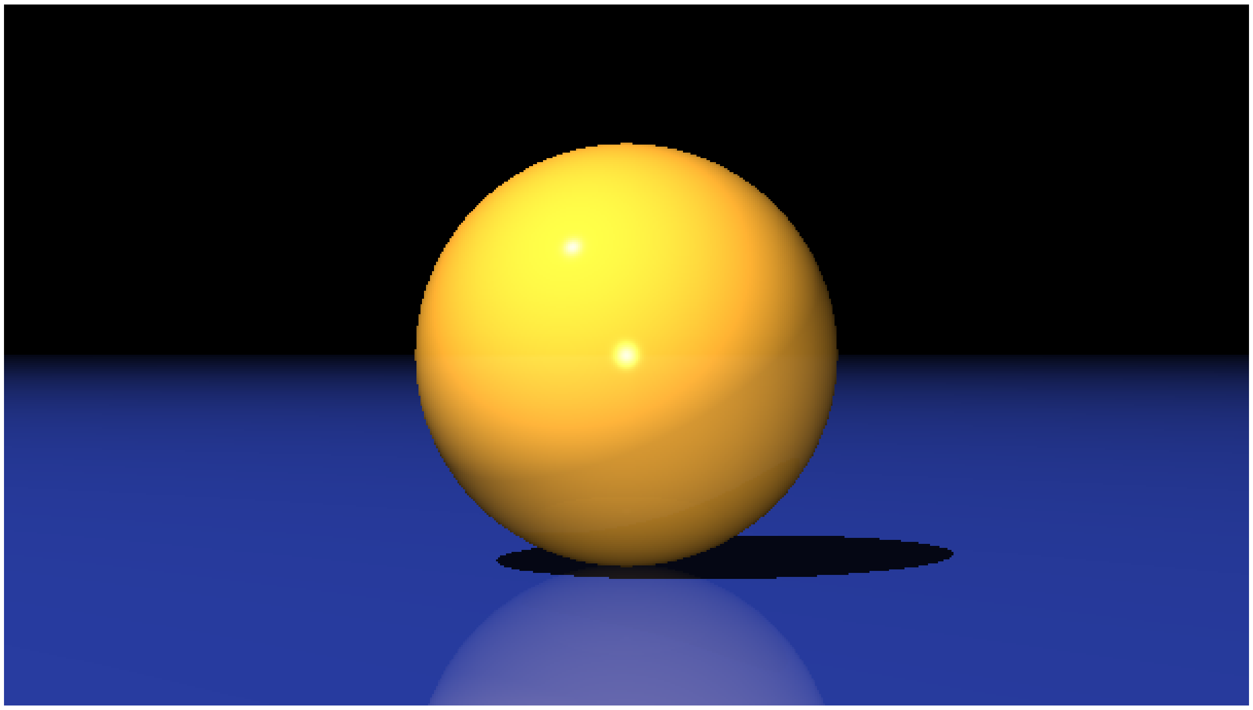
> **To get started:** Clone this repository and its submodule using
>
> ```
> git clone --recursive http://github.com/alecjacobson/computer-graphics-
> ray-tracing.git
> ```
>
> **Do not fork:** Clicking "Fork" will create a *public* repository. If you'd like to use GitHub while you work on your assignment, then mirror this repo as a new *private* repository: https://stackoverflow.com/questions/10065526/github-how-to-make-a-fork-of-public-repository-private

Background

Read Sections 4.5-4.9 of *Fundamentals of Computer Graphics (4th Edition)*.

Many of the classes and functions of this assignment are borrowed or adapted from the previous ray casting assignment.

Unlike that assignment, this ray tracer will produce *approximately* accurate renderings of scenes illuminated with light. Ultimately, the shading and lighting models here are *useful* hacks. The basic recursive structure of the program is core to many methods for rendering with global illumination effects (e.g., shadows, reflections, etc.).



Floating point numbers

For this assignment we will use the `Eigen::Vector3d` to represent points and vectors, but *also* RGB colors. For all computation (before finally writing the .ppm file) we will use double precision floating point numbers and `0` will represent no light and `1` will represent the brightest color we can display.

Floating point numbers $\neq$ real numbers, they don't even cover all of the rational numbers.

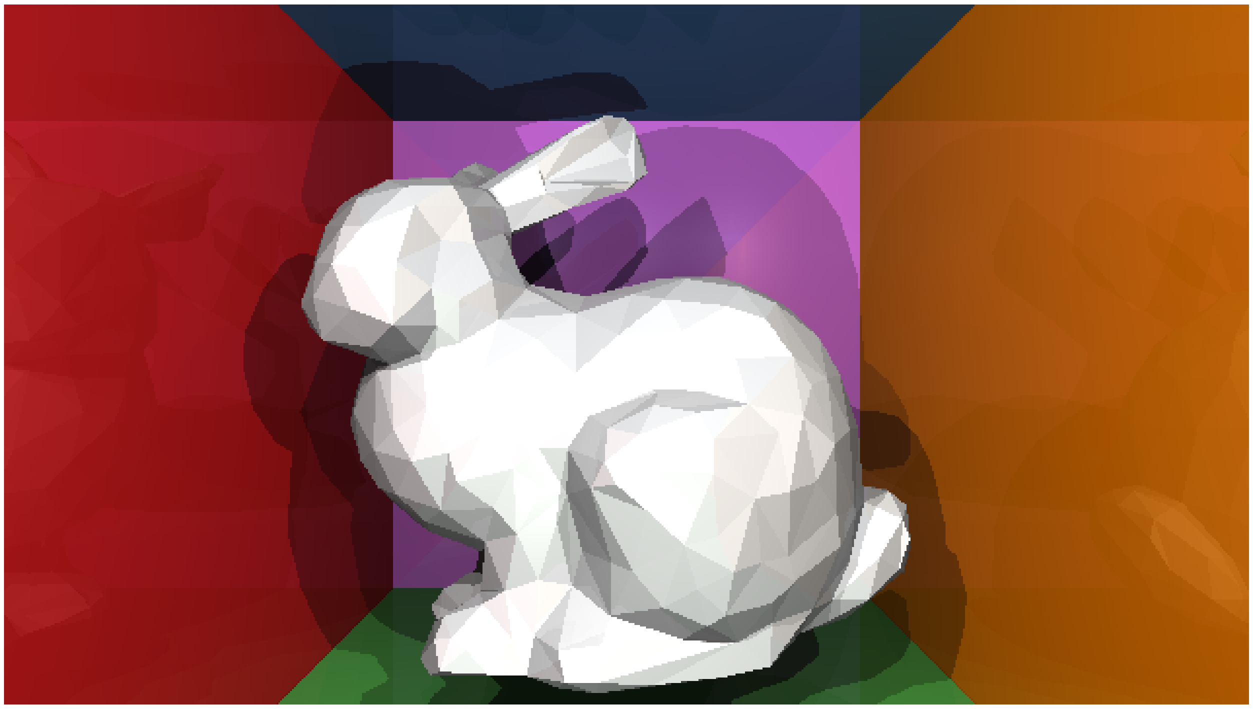
This creates a number of challenges in numerical method and rendering is not immune to them. We see this in the need for a fudge factor to discard ray-intersections when computing shadows or reflections that are too close to the originating surface (i.e., false intersections due to numerical error).

> **Question:** If we build a ray and a plane with floating point coefficients, will the intersection point have floating point coefficients? What if we consider rational coefficients? What if we consider a sphere instead of a plane?
>
> **Hint:** Can we *exactly* represent $1/3$ as a `double`? Can we represent $\sqrt{2}$ as a rational?

Dynamic Range & Burning

Light obeys the superposition principle. Simply put, the light reflected of some part of an objects is the *sum* of contributions from light coming in all directions (e.g., from all light sources). If there are many bright lights in the scene and the object has a bright color, it is easy for this sum to add up to more than one. At first this seems counter-intuitive: How can we exceed 100% light? But this premise is false, the 1.0 does not mean the physically brightest possible light in the world, but rather the brightest light our screen can display (or the brightest color we can store in our chosen image format). High dynamic range (HDR) images store a larger range beyond this usual [0,1]. For this assignment, we will simply *clamp* the total light values at a pixel to 1.

This issue is compounded by the problem that the Blinn-Phong shading does not correctly conserve energy as happens with light in the physical world.



> **Question:** Can we ever get a pixel value *less than zero*?
>
> **Hint:** Can a light be more than off?
>
> **Side note:** This doesn't stop crafty visual effects artists from using "negative lights" to manipulate scenes for aesthetic purposes.

Whitelist

There are many ways to "multiply" two vectors. One way is to compute the component-wise multiplication: $\mathbf{c} = \mathbf{a} \circ \mathbf{b}$ or in index notation: $c_i = a_i b_i$. That is, multiply each corresponding component and store the result in the corresponding component of the output vector. Using the Eigen library this is accomplished by telling Eigen to treat each of the vectors as "array" (where matrix multiplication, dot product, cross product would not make sense) and then using the usual `*` multiplication:

```
Eigen::Vector3d a,b;
...
// Component-wise multiplication
Eigen::Vector3d c = (a.array() * b.array()).matrix();
```

The `.matrix()` converts the "array" view of the vector back to a "matrix" (i.e., vector) view of the vector.

Eigen also has a built in way to normalize a vector (divide a vector by its length): `a.normalized()`.

README.md

Tasks

`src/Plane.cpp`, `src/Sphere.cpp`, `src/Triangle.cpp`, `src/TriangleSoup.cpp`, `src/first_hit.cpp`, `src/viewing_ray.cpp`, `src/write_ppm.cpp`

See the previous ray casting assignment.

`PointLight::direction` in `src/PointLight.cpp`

Compute the direction to a point light source and its *parametric* distance from a query point.

`DirectionalLight::direction` in `src/DirectionalLight.cpp`

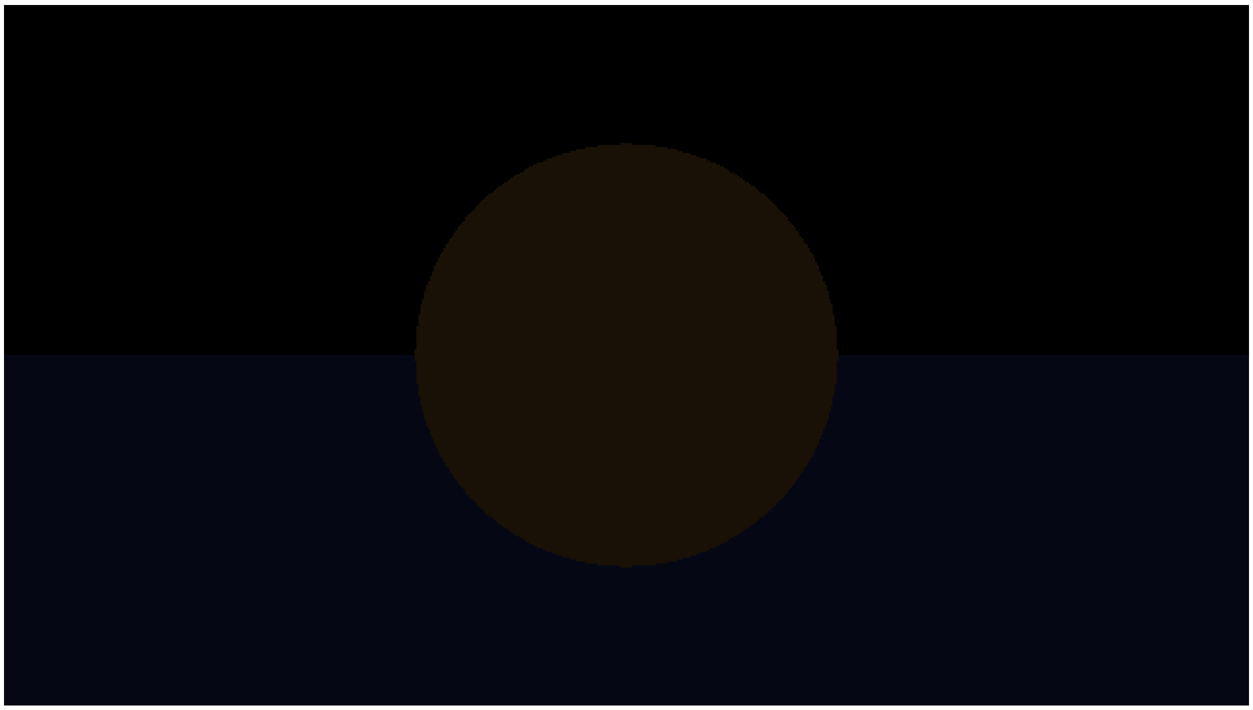
Compute the direction to a direction light source and its *parametric* distance from a query point (infinity).

`src/raycolor.cpp`

Make use of `first_hit.cpp` to shoot a ray into the scene, collect hit information and use this to return a color value.

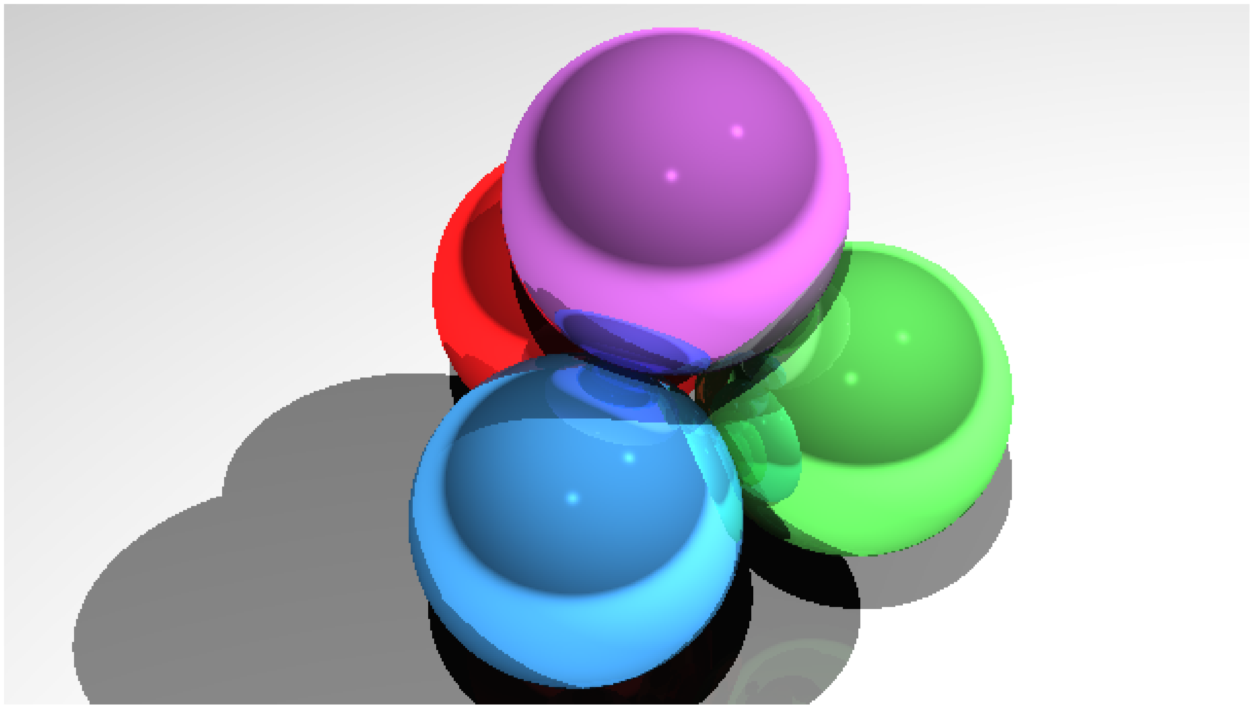
`src/blinn_phong_shading.cpp`

Compute the lit color of a hit object in the scene using Blinn-Phong shading model. This function should also shoot an additional ray to each light source to check for shadows.



`src/reflect.cpp`

Given an "incoming" vector and a normal vector, compute the mirror reflected "outgoing" vector.



`src/creative.json`

Be creative! Design a scene using any of the available Object types (spheres, planes, triangles, triangle soups), Light types (directional, point), Material parameters, colors (materials and/or lights), and don't forget about the camera parameters.

The .json format is rather straightforward. But you may find this validator useful.

> ### HW2 Solution
>
> If you don't trust your solutions to the files from HW2:

```
src/Plane.cpp
src/Sphere.cpp
src/Triangle.cpp
src/TriangleSoup.cpp
src/first_hit.cpp
src/viewing_ray.cpp
src/write_ppm.cpp
```

> You can use precompiled binaries (provided for linux, mac, and windows) using a the cmake command:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Debug -DHW2LIB_DIR=../lib/debug/linux/ ..
make
```

> This will use the library at `../lib/debug/linux/libhw2.a` instead of compiling the above files in `src/`.

> **Pro Tip:** After you're confident that your program is working *correctly*, you can dramatic improve the performance simply by enabling compiler optimization:
>
> ```
> mkdir build-release
> cd build-release
> cmake ../ -DCMAKE_BUILD_TYPE=Release
> make
> ```

Releases

No releases published

Packages

No packages published

Languages

● **C++** 97.5% ● **CSS** 1.9% ● **Other** 0.6%