Ray Tracer

Maksim Levental
CAP 4730

January 31, 2014

Contents

1 Introduction **3**
- 1.1 Intersection . . . 3
  - 1.1.1 Sphere . . . 3
  - 1.1.2 Triangle . . . 4
- 1.2 Shading . . . 4
  - 1.2.1 Diffuse . . . 4
  - 1.2.2 Specular . . . 4
  - 1.2.3 Lighting . . . 5

2 Implementation **5**
- 2.1 `surfaces.h` . . . 5
  - 2.1.1 `sphere` class . . . 5
  - 2.1.2 `triangle` class . . . 5
- 2.2 `ray_tracer.cpp` . . . 6
  - 2.2.1 `makeCheckImage` . . . 6
  - 2.2.2 `keyPressed` . . . 6
  - 2.2.3 `main` . . . 6

3 Testing **7**
- 3.1 Only ambient . . . 7
- 3.2 Only diffuse . . . 8
- 3.3 Only specular . . . 9
- 3.4 Specular and diffuse . . . 10
- 3.5 Final . . . 11

4 Conclusion **11**

Abstract

Ray tracing is a basic method of rendering a three dimensional scene on a viewing plane. Light rays are modeled as vectors with origin at every pixel in viewing plane and direction. These rays are then tested for intersection against objects in the scene. Intersection tests are unique to each figure type. Shading is effected by comparing various dot products between the normal to the surface of the object, the intersection ray, and the light source vector. Ambient light is a global constant. Efficiency and numerical stability is poor.

1 Introduction

Ray tracing is a way to image objects in a virtual scene. That is to say if one wishes to represent a collection of 3D objects, for each of which one has an implicit representation, on a 2D image plane. Its efficacy relies on the idea that light can be modeled by ray/geometric optics. The basic idea is to treat every pixel in the image plane as a source for such an optical ray and then to simulate the effects of that ray's interaction with objects in the virtual scene.

1.1 Intersection

The simplest interaction the ray could have with an object is that it might intersect that object. Testing for such an intersection requires having an implicit representation for that object. To be precise a ray has the parametric vector representation

$$\vec{f}(t) = \vec{P} + t\vec{D}$$

where $\vec{P}$ is the origin of the ray, $\vec{D}$ is the direction of the ray, and $\vec{f}$ ranges through all the vectors to points on the line as t ranges from 0 to $+\infty$. Implicit representations of objects in the scene are a collection $g(\vec{x})$[1] such that

$$g(\vec{x}) = 0$$

for all $\vec{x}$ that correspond to points on that object.

1.1.1 Sphere

For the case of a sphere [3] with center $\vec{c}$ and radius r

$$g(\vec{x}) = g(\vec{f}(t)) = \left\| \vec{P} + t\vec{D} - \vec{c} \right\|^2 - r^2 = 0.$$

Equivalently define $\vec{v} = \vec{P} - \vec{c}$ then if

$$t = -(\vec{v} \cdot \vec{D}) \pm \sqrt{(\vec{v} \cdot \vec{D})^2 - (\vec{v}^2 - r^2)}$$

is real the ray intersects the sphere.

[1] We interpret g as any process for testing whether $\vec{x}$ is a "root".

1.1.2 Triangle

For the case of a triangle the intersection test is more complex [2]. Define for each ray r

$$\pi_r = (\vec{D}, \vec{D} \times \vec{P}) \equiv (\vec{U}_r, \vec{V}_r)$$

and

$$\pi_i \odot \pi_j = \vec{U}_i \cdot \vec{V}_j + \vec{U}_j \cdot \vec{V}_i.$$

For each edge e_i of the triangle assert π_{e_i}. Then the ray intersects the triangle iff

$$\pi_r \odot \pi_{e_i} \geq 0 \ \forall i \text{ and } \exists j \ \pi_r \odot \pi_{e_j} \neq 0$$

or

$$\pi_r \odot \pi_{e_i} \leq 0 \ \forall i \text{ and } \exists j \ \pi_r \odot \pi_{e_j} \neq 0.$$

1.2 Shading

A slightly more complex interaction that a ray could have with an object is specular reflection/refraction. If there is a light source (ambient light is discussed blow) then the degree to which the ray is refracted can be used to appropriately "shade" the surface using the Phong lighting model.

There are three components to the light cast on the object: ambient, which is just a global constant energy added to all points, diffuse, which is a function of the normal to the surface and the direction from the point to the light source, and specular, which is a function of the point of view of the virtual camera and the direction of the light source.

1.2.1 Diffuse

Diffuse light is calculated as

$$I \, k_d \max(0, \vec{n} \cdot \vec{v}_l)$$

where I is a global illumination constant, k_d is the diffusion constant (to modulate how potent the diffuse lighting is), $\vec{n}$ is the normal to the surface, and $\vec{v}_l$ is the direction to the light source. Taking the maximum of 0 and $\vec{n} \cdot \vec{v}_l$ is done because the normal might be in the opposite direction of the source of light, if for example a point on a hidden section of the object is intersected by the viewing ray.

1.2.2 Specular

Specular lighting is typically computed as

$$I \, k_d \max(0, \vec{v}_r \cdot (2(\vec{n} \cdot \vec{v}_l) - \vec{v}_l))^n$$

but this is too computationally inefficient so the reflection ray, $2(\vec{n} \cdot \vec{v}_l) - \vec{v}_l)$ is approximated by the bisector vector of $\vec{l}$ and $\vec{r}$

$$\vec{v}_H = \text{bisector}(\vec{v}_l, \vec{r}) = \frac{(\vec{v}_l + \vec{v}_r)}{\|\vec{v}_l + \vec{v}_r\|}$$

Hence the specular shading gradient is computed as

$$I\, k_s \max(0, \vec{n} \cdot \vec{v}_H)^n$$

1.2.3 Lighting

Total lighting applied to the surface of the object is then

$$k_a I_a + I_w \Big(k_d \max(0, \vec{n} \cdot \vec{v}_l) + k_s \max(0, \vec{n} \cdot \vec{v}_H)^n \Big)$$

where k_a and I_a are constants ambient lighting constant that apply a warm hue to the scene.

2 Implementation

The ray tracer algorithm is implemented in C++ using the Boost library uBLAS [1] for the linear algebra components and the freeglut OpenGL api to actually draw the window.

2.1 `surfaces.h`

This defines the classes for the two main surface types "traced" by the ray tracer, spheres and triangles. Each class derives from the base class `surface`, which defines 3 pure virtual methods (`intersection_test`, `unit_normal`, `test`) that each subclass should implement, and the base member that all subclasses share `color`, which is of type `GLubyte`, a typedef from the freeglut library for `char`. `intersection_test` takes as arguments the ray origin and the ray direction and returns the distance of the intersection point from the viewing plane or -1 in case there is no intersection.

`surfaces.h` also defines typedefs for commonly used vectors from the uBLAS library, global constants for the light source direction, the viewing plane pixel array (`checkImage`), and shading constants. Finally `surfaces.h` defines 3 helper functions: `rotate_viewing_angle` is necessary for effecting rotations of the camera perspective, `cross_product` is the cross product in 3 dimensions (because that is absent from uBLAS), and `loadbar` is a loading bar that shows progress of rendering in the console.

2.1.1 `sphere` class

The constructor takes as arguments a `GLubyte[3]` RGB color, a `dbl_vec` center, and a radius.

2.1.2 `triangle` class

The constructor takes as arguments a `GLubyte[3]` RGB color, and 3 `dbl_vec` vertices in counter-clockwise order. That is to say the orientation of the triangle is right-handed, so if

the triangle vertices passed are $(0,1,0)$,$(0,0,0)$, and $(1,0,0)$ then the normal would be in the $\vec{k}$ direction. This is important for the shading algorithm.

Note `plucker_edgeXY[2]` in the `triangle` class is π_{e_i} discussed in 1.1.2 .

2.2 `ray_tracer.cpp`

This implements the actual ray tracing algorithm.

2.2.1 `makeCheckImage`

For every pixel in the viewport, each of which corresponds to the entries in the `checkImage` array defined in `surfaces.h`, a `long_double_vec` is created. Then for each object in the scene, pointers to which are all stored in the global `std::vector<surface*>` container, the intersection test for that class is run. If there is in fact intersection then that object is stored in a `std::map<double,surface*>` container with key equal to the distance from the viewing plane. This is so that if the ray intersects many objects the one with the smallest distance (key) will be the first element in the `map`.

If any objects are intersected then the shader method of the nearest one (first element in `map` as mentioned above) is called to color the corresponding entry in `checkImage`. Otherwise `checkImage` is set to black.

A red bounding box is drawn around the viewing plane as well.

2.2.2 `keyPressed`

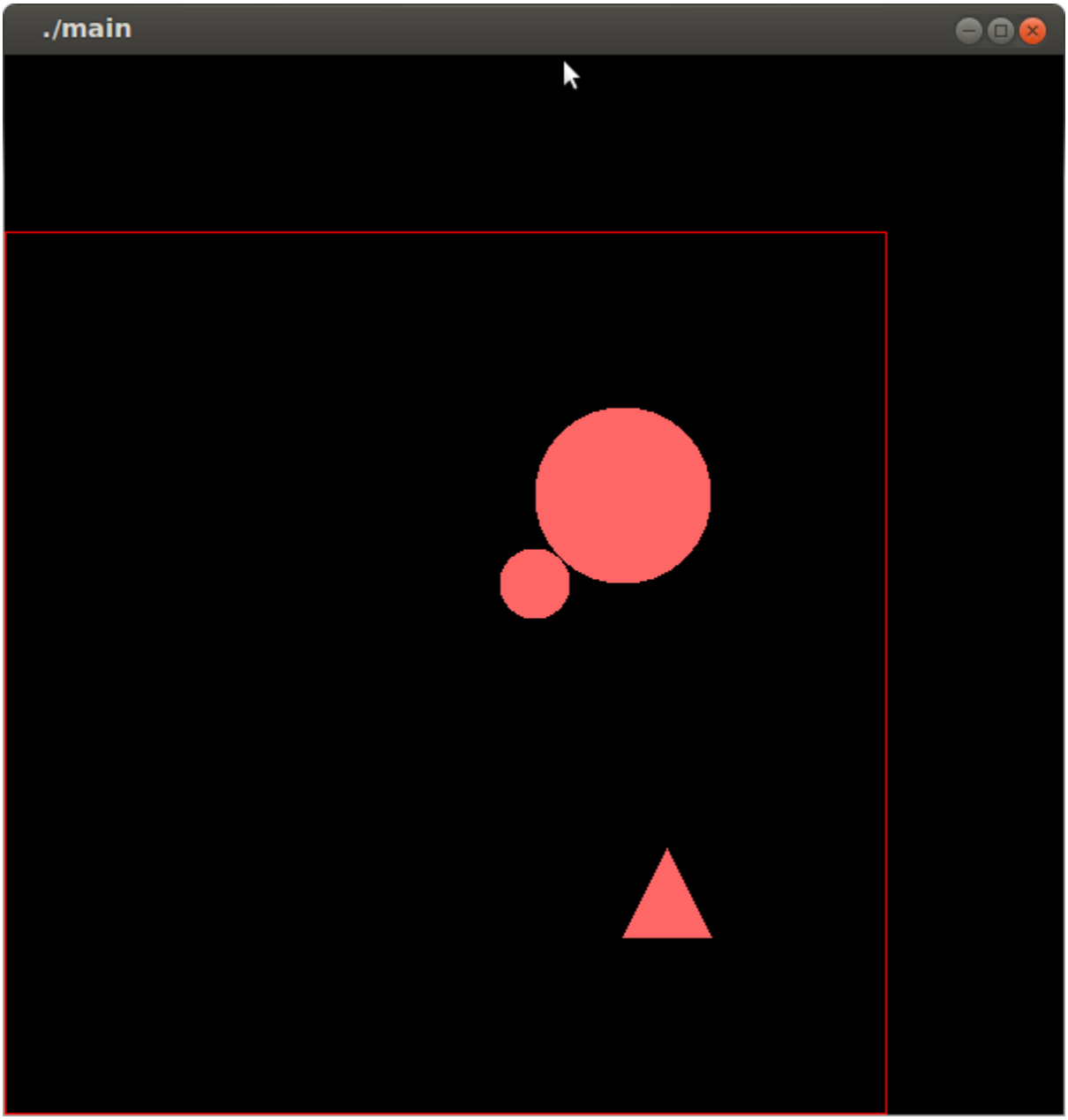
Rotations of the camera perspective upon key press are implemented. Pressing `d` on the keyboard rotates 10 degrees counter-clockwise around the z-axis and pressing `a` undoes this rotation. Pressing `w` on the keyboard rotates 10 degrees counter-clockwise around the y-axis and pressing `s` undoes this.

2.2.3 `main`

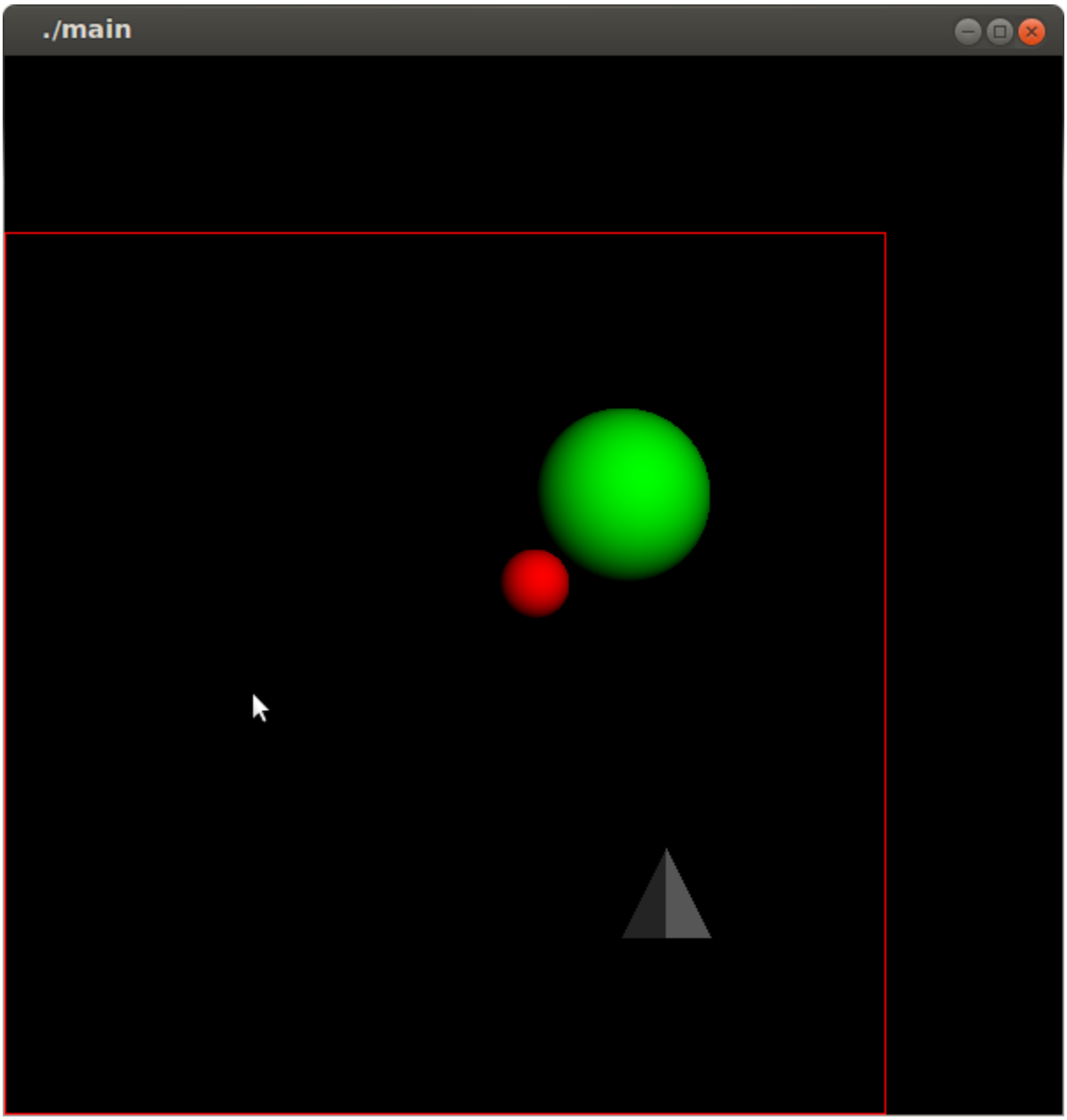
Objects are instantiated and added to the `std::vector<surface*> surfaces` container. `int_vec push` can be used to translated objects by adding to either the center, in the case of a `sphere`, or to all vertices, in the case of a `triangle`.

3 Testing

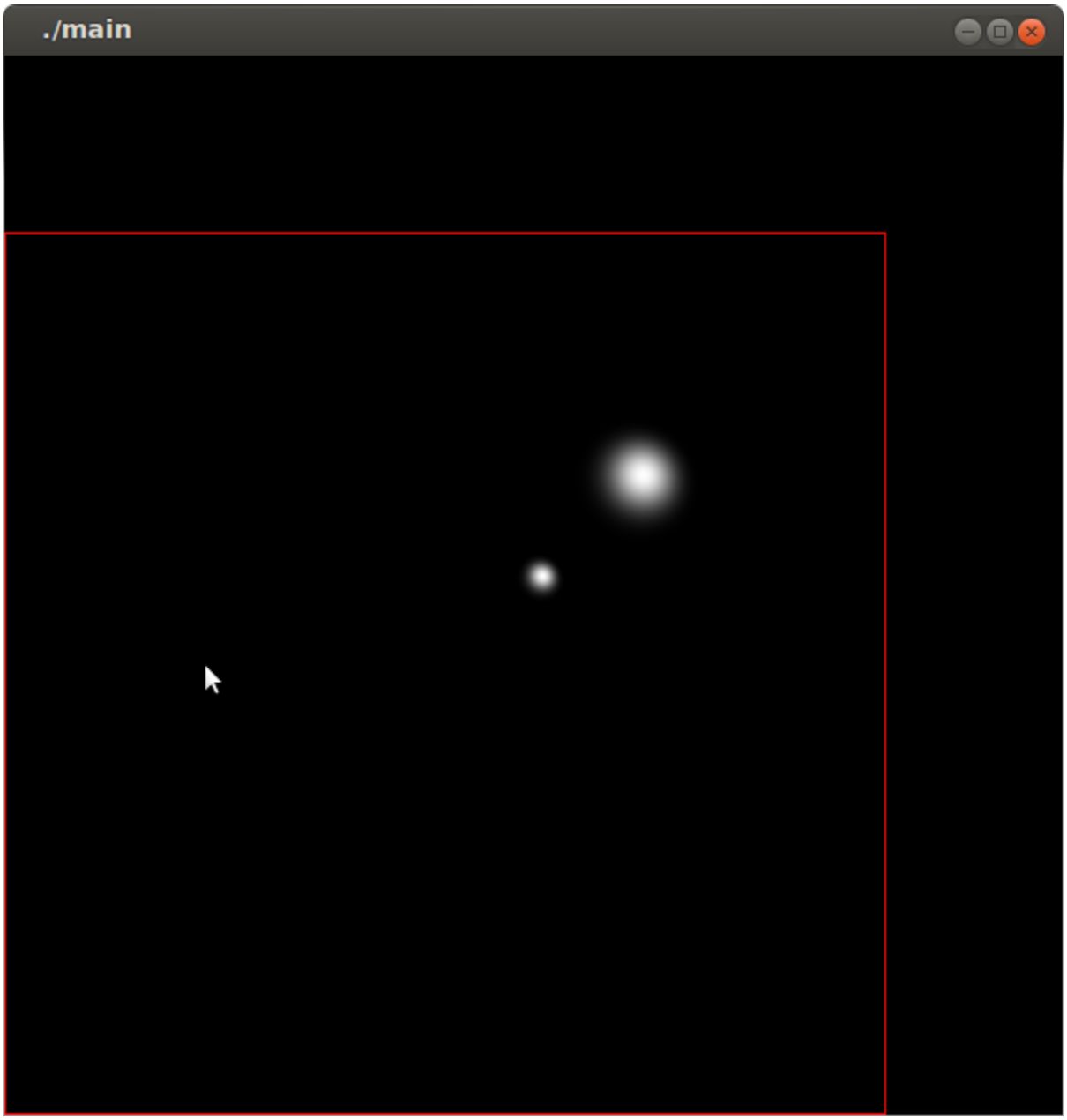
3.1 Only ambient



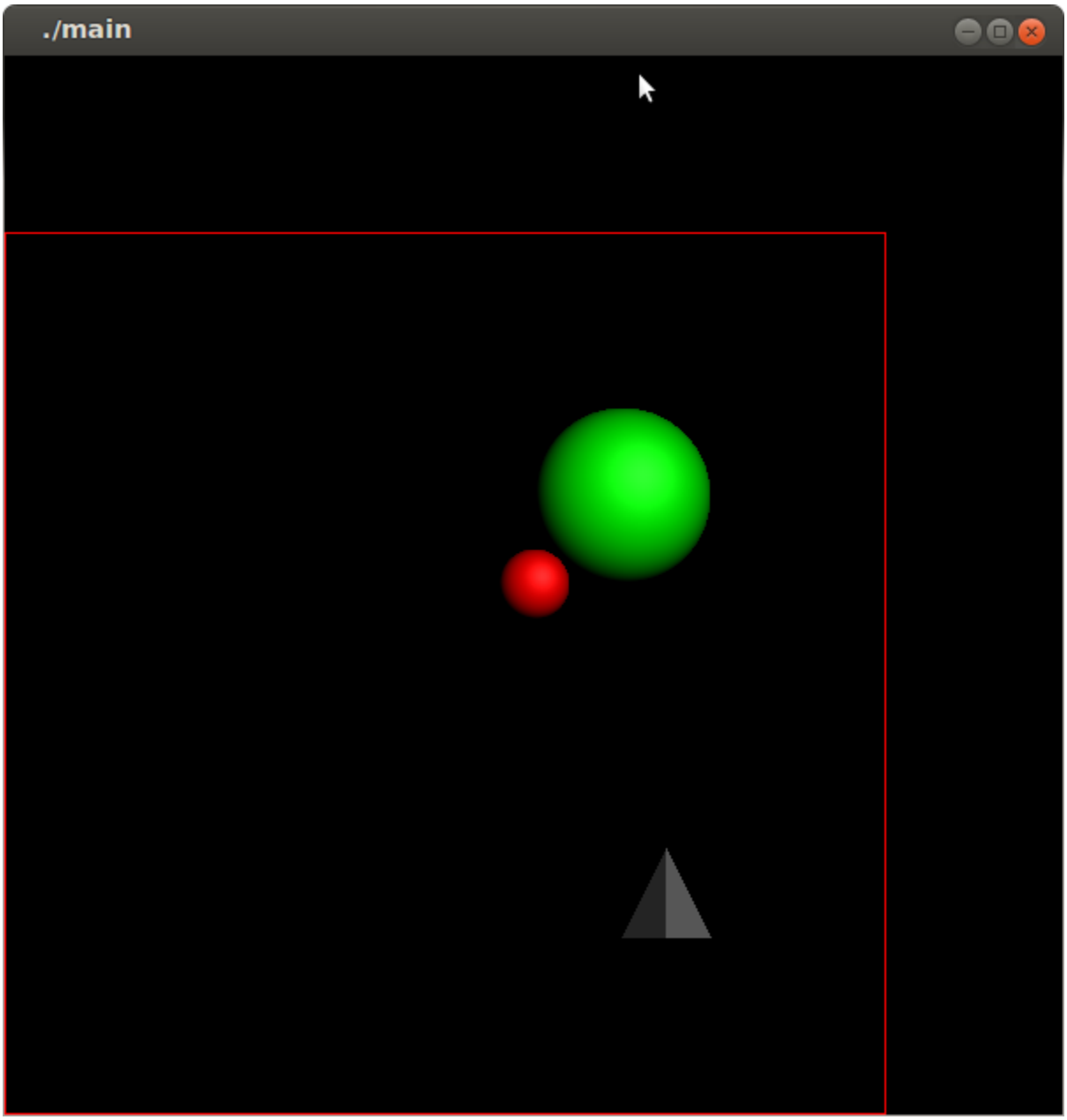
3.2 Only diffuse



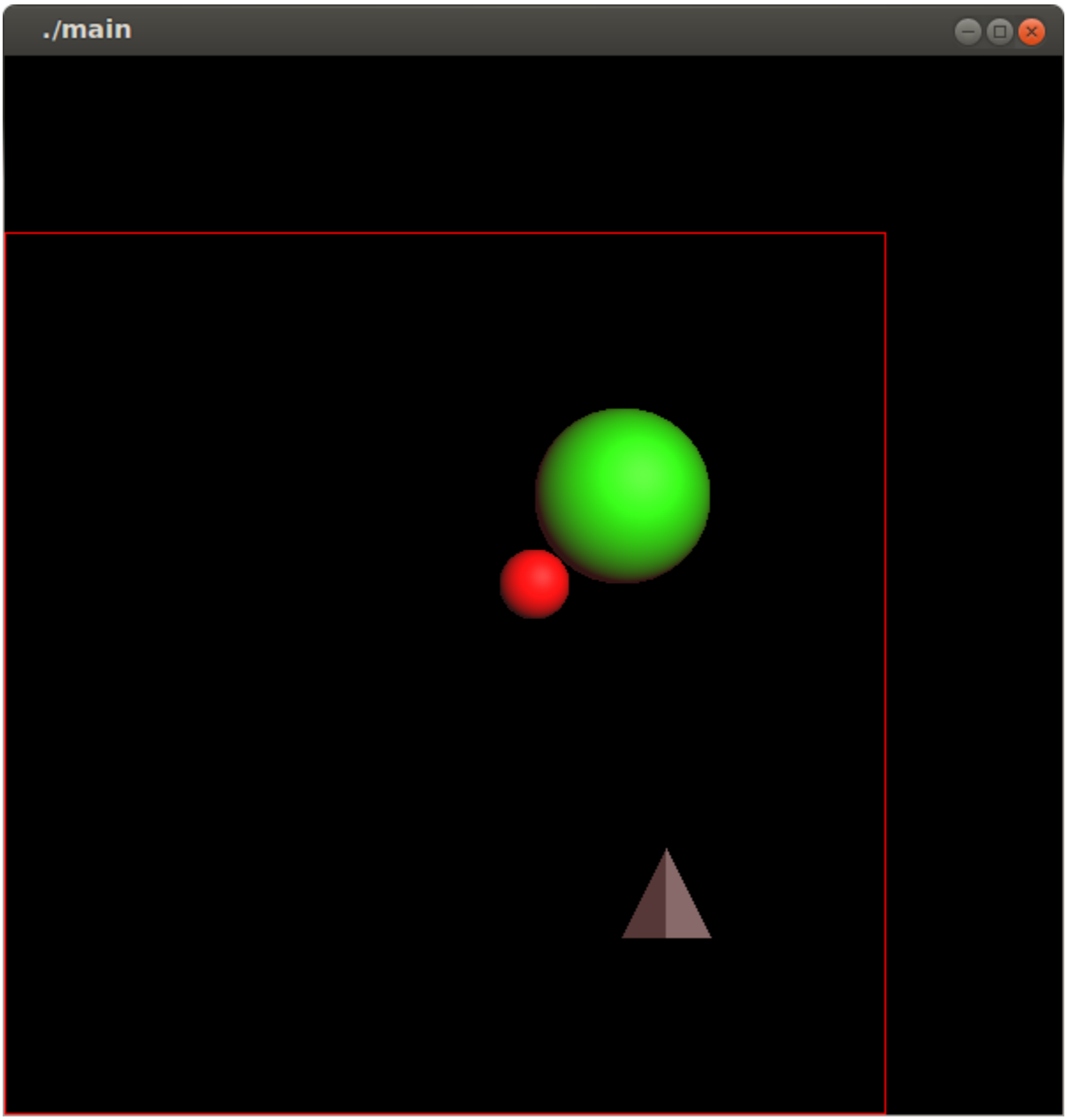
3.3 Only specular



3.4 Specular and diffuse



3.5 Final



4 Conclusion

While ray tracing is a simple algorithm to implement it is not numerically stable. For example when implementing rotations it was necessary to change the intersection test in the `triangle` class. `double diff` had to be introduced because the naive check checked whether

```
abs(norm_plucker) == ( abs(sidetestAB)+abs(sidetestBC)+abs(sidetestCA) )
```

but this ceased to work because of accumulated floating point error. Hence checking that they were within an interval of each other.

Furthermore testing rays emanating from every pixel in the viewing plane is very inefficient because most of the scene is empty. **The addition of the tetrahedron to the scene did noticably slow rendering but that's simply because said tetrahedron is small. Given the complexity of the intersection test for the `triangle` class it is obvious that were the tetrahedron bigger it would greatly slow the rendering.**

References

[1] *BOOST C++ Libraries*. `http://www.boost.org`.

[2] Nikos Platis and Theoharis Theoharis. Fast ray-tetrahedron intersection using plucker coordinates. *Journal of Graphics Tools*, 8(4):37–48, 2003.

[3] Wikipedia. Ray tracing (graphics) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Ray%20tracing%20(graphics)&oldid=589687998`, 2014. [Online; accessed 28-January-2014].