Assignment 5: Ray tracing

CS 4600 Computer Graphics
Fall 2019
Ladislav Kavan

In this assignment we will explore basic ray tracing. This is an individual assignment, i.e., you have to work independently. All information needed to complete this homework is covered in the lectures and discussed at our Canvas Discussion Boards. You shouldn't have to use any textbooks or online resources, but if you choose to do so, you must reference these resources in in your final submission. It is strictly prohibited to reuse code or fragments of code from textbooks, online resources or other students -- in this course this is considered as academic misconduct (https://www.cs.utah.edu/academic-misconduct/). Do not share your homework solution with anyone -- this is also treated as academic misconduct in this course, even if nobody ends up copying your code.

The framework code is written in C++ with the following dependencies:

- C++ STL
- Eigen

The recommended IDE is Visual Studio 2017 Community Edition, which is available free of charge for educational purposes. The framework code provides precompiled dependencies for Visual Studio 2017. If you choose to use a different platform or IDE version it is your responsibility to build the dependencies and get the project to work.

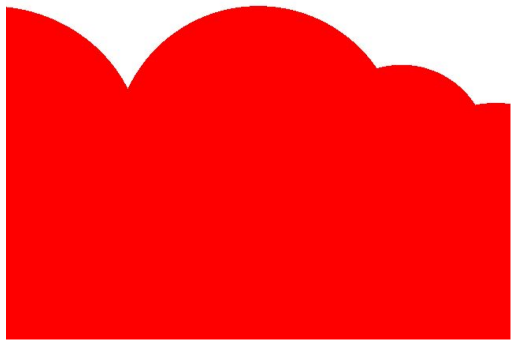
The assignment should be implemented inside the provided *main.cpp* file using the specified subroutines. No other source code / dependencies / libraries are needed or allowed for this assignment. The provided source code, after being successfully compiled, linked, and executed, will produce file "render.ppm" which contains solid black image. Your task will be to make this image much more interesting!

1 Ray casting (34 points)

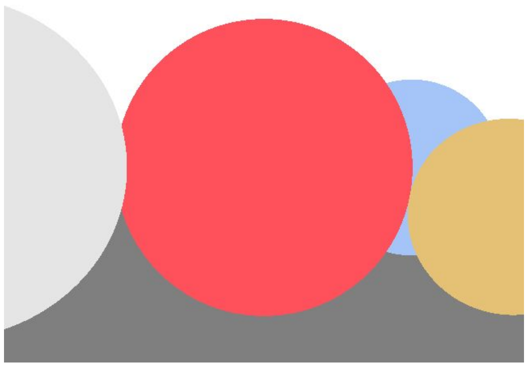
The provided function *render* sends a ray through all pixels of the rendered image. “Sending” a ray corresponds to calling function

```
Vector3f trace(
        const Vector3f &rayOrigin,
        const Vector3f &rayDirection,
        const std::vector<Sphere> &spheres)
```

which you have to implement. The ray to be traced is specified by its origin (rayOrigin) and direction (rayDirection). The “rayDirection” is a unit vector. This ray needs to be intersected against all objects in our scene -- in this homework, this is a list of spheres, represented by the input array “spheres”. You may want to use the provided function Sphere::intersect, which returns “true” if the ray intersected the sphere and “false” otherwise. If “true” is returned, the function also returns the intersection points as distances along the ray (t0, t1, where t0 is the first intersection and t1 the second one). In this first task, your *trace* function should return red color if the ray hit **any** sphere and return *bgcolor* otherwise (background color, white by default). The resulting image should look like this:



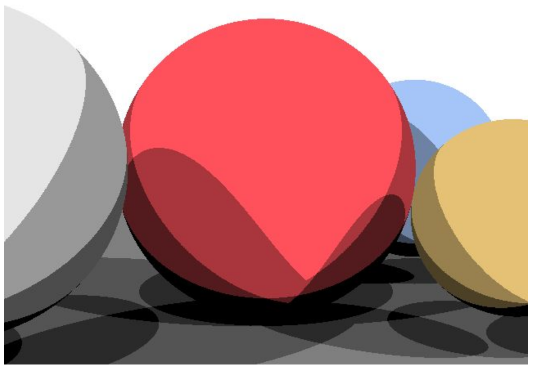
Save this image as “out001.ppm” and submit along with your solution. Next, we slightly improve this image by rendering each sphere with a different color. The color of each sphere is stored in Sphere::surfaceColor which will be the new return value of function *trace*. If no sphere is hit by the ray, the *trace* function should return the background color (*bgcolor*). The resulting image should look like this:



Please save this image as “out002.ppm” and submit along with your solution.

2 Shadow rays (33 points)

Next we will add light sources to our scene, so we get some shadows. In our testing scene, we have three point lights. Their positions are specified in array *lightPositions*. The key idea of ray tracing is that when the *trace* function detects that a ray hit a sphere at point P, we need to find out whether this point P is visible from each light. If it is visible, the light is shining at point P and contributes 0.333 * surfaceColor to the resulting pixel color. If the path of light between the light source and P is blocked by some other sphere, then there is no light from the light source that would reach point P and the contribution to the resulting pixel color is zero. The factor 0.333 is there because we have three light sources and we want the brightest color (1) if all light sources are visible. After you implement this, you should get an image that looks like this:



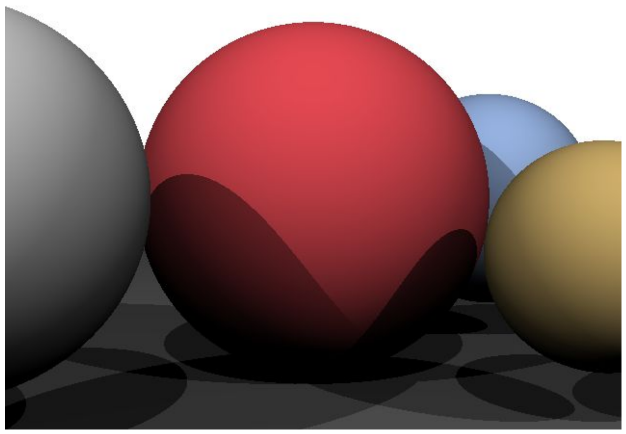
You can see that all of the three light sources are now casting shadows! Please save this image as “out003.ppm” and submit along with your solution.

3 Illumination models (33 points)

Finally, we will add illumination (or reflection) models to create more realistic images. We will start with diffuse shading. From an intersection point P we cast rays to all lights (as in Task 2), and if the light is not blocked, we add 0.333 * kd * max(L.dot(N), 0) * diffuseColor, where L is unit vector pointing from P to the light and normal N is unit vector of surface normal at P (the normal of a sphere is pretty easy to compute -- make sure not to forget to normalize the vectors so they are really unit!), kd is weight of the diffuse reflection (1). You should implement this by filling in the body of the function

```
Vector3f diffuse(const Vector3f &L,
        const Vector3f &N,
        const Vector3f &diffuseColor,
        const float kd)
```

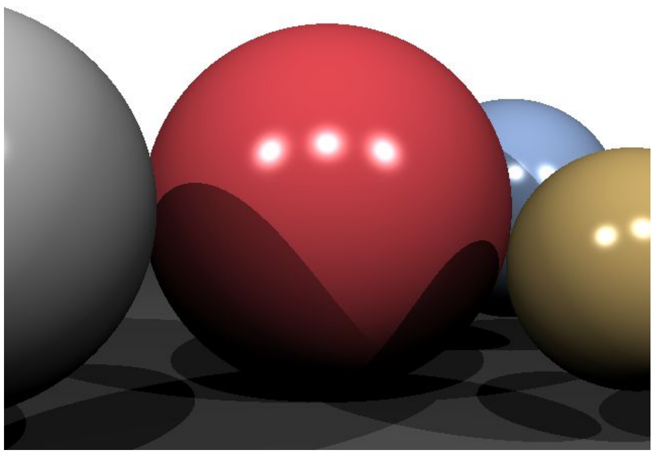
which you call with the appropriate parameters -- the diffuse color should be set to the "surfaceColor" of each sphere. This will lead to image that looks like this:



Please save this image as "out004.ppm" and submit along with your solution. Finally, we will add specular reflection according to the Phong reflection model. Your job is to implement and call function

```
Vector3f phong(const Vector3f &L,
        const Vector3f &N,
        const Vector3f &V,
        const Vector3f &diffuseColor,
        const Vector3f &specularColor,
        const float kd,
        const float ks,
        const float alpha)
```

where L is unit direction to the light, N is unit normal vector, V is unit normal vector pointing against the ray, diffuse color is the color of each sphere (Sphere::surfaceColor), specularColor is color of the light (in this homework we have white lights which means that you pass here just Vector3f::Ones()), kd is weight of the diffuse reflection (1), ks is weight of specular reflection (3) and alpha is shininess parameter (100). You can call your function *diffuse* to compute the diffuse component of the Phong reflection model, i.e., the function *phong* will only add the specular component (again scaled by 0.333 as with diffuse shading). At the end, you will get an image that looks like this:



where we can clearly see the specular highlights corresponding to the three lights in our virtual scene. Please save this image as “out005.ppm” and submit along with your solution.

4 Extra credit ideas

For extra credit, you can implement recursive ray tracing to produce mirror-like effects on reflective surfaces. As discussed in the lecture, the key idea is to continue tracing the ray after the first bounce: if the ray hits a reflective material, we compute the reflection direction and recursively call the ray tracing function. You may also design a new scene made out of spheres (or possibly different objects!). Another very nice extension is supporting area light sources, which allows generation of soft shadows (as opposed to hard shadows which we have done so far). This requires casting multiple rays for each light source. A more general version of this is called Distributed ray tracing (https://en.wikipedia.org/wiki/Distributed_ray_tracing).

5 Submission

When you're finished with Tasks 1, 2, and 3 you'll need to submit the following:

- Source code (you should only modify the main.cpp file). The main.cpp file is all we need -- please do not submit any other files, especially NOT .exe files and other files created by Visual Studio.
- PDF document describing what you did, screenshots / graphs are recommended. If you used any textbooks or online resources that may have inspired your way of thinking about the assignment, you must references these resources in this document
- Your resulting images "out001.ppm", "out002.ppm", ..., "out005.ppm"
- [optional] If you have succeeded with Task 4, submit the code "main-extra.cpp" and an image "out-extra.ppm" showing the result of your recursive Ray tracer, possibly on different scenes.

Please pack all of your files to a single ZIP file named Lastname_Firstname_HW5.zip

Please submit the ZIP file(s) via Canvas and in the comments specify how many late days you're using and have remaining (e.g. "Used 0 late days, 5 remaining.") Failure to comply with submission instructions may result in loss of points (up to zero in cases of severe negligence).

Late policy

Everyone has 5 late days to cover circumstances such as minor illness, competitions, family matters, etc. You can distribute your late days arbitrarily among assignments (e.g. you can use 0 late days on HW assignment 1 and all 5 late days on HW assignment 2, leaving no late days left for subsequent assignments). After using up all of your late days, a 0.1*X*number_of_late_days will be deducted from your points, where X is the pre-penalty number of points. Each late day is defined as 24 hours after the submission deadline on Canvas, there is no distinction between weekdays and weekends or holidays. Requests for exceptions to this policy (e.g., due to major medical conditions) must be made in writing, with a date and a signature of the student or their legal representative, and include documentation pertaining to the case (e.g., a note from the University Center for Disability & Access).