Assignment 2 – How to build a world in 3D

Assignment due date: Monday, October 23, 8:30am

Hand-in to be submitted at the start of lecture,
code to be submitted on the mathlab server by the above due date
This assignment can be completed individually, or by a team of 2 students

Student Names (Last, First)

Student #1:

Student #2:

Student numbers

Student #1:

Student #2:

Student UtorIDs

Student #1:

Student #2:

We hereby affirm that all the solutions we provide, both in writing and in code, for this assignment are our own. We have properly cited and noted any reference material we used to arrive at this solution, and have not shared our work with anyone else.

_____ _____

Student 1 signature ***Student 2 signature***

(note: 3 marks penalty if any of the above information is missing)

Assignment 2 – How to build a world in 3D

At this point we are ready to tackle the problem of rendering a scene in 3D. This requires understanding of the complete process of image formation, object representation and manipulation using affine transformations, coordinate frames and conversions, and projection.

Assignments 2 and 3 will be dedicated to helping you strengthen and refine your understanding of these topics by building a fully operational ray tracer. Within a few weeks your code will be rendering high quality scenes with realistic illumination!

Learning Objectives – after completing this assignment you should be able to:

Set up a scene using canonical objects and affine transformations, illuminated by point light sources.

Set up a camera, and determine the correct transformation that determines which points in the scene map to which pixels in the image.

Determine the parameters of a ray going from the camera's center of projection, through any pixel in the image, and into the scene.

Set up the equations that give the intersection between rays from the camera and objects in the scene. Along with the process to determine local geometry including surface normal.

Use local geometry at an intersection between an object and a ray to evaluate the components of the Phong illumination model. Explain what each of the components contributes to the appearance of an object.

Understand, explain, and simulate the basic geometry of light that yields shadows and global specular reflection via recursive ray tracing.

Skills Developed:

Implementing the full image formation pipeline – from objects in a scene to pixel colours

Using affine transforms to manipulate objects and create scenes, and to easily handle ray-object intersections for deformed objects.

Handling illumination effects such as shadows and global specular reflections.

Developing and debugging software that works with light rays, objects, transforms, and images.

Reference material:

Lecture notes for camera projections, transformations, and simple ray tracing.

The detailed comments in the starter code. The starter code ***already provides a lot of the simple building blocks you will need***. Don't waste time implementing functionality already there.

Your TA and instructor! Don't get stuck! Seek help early!

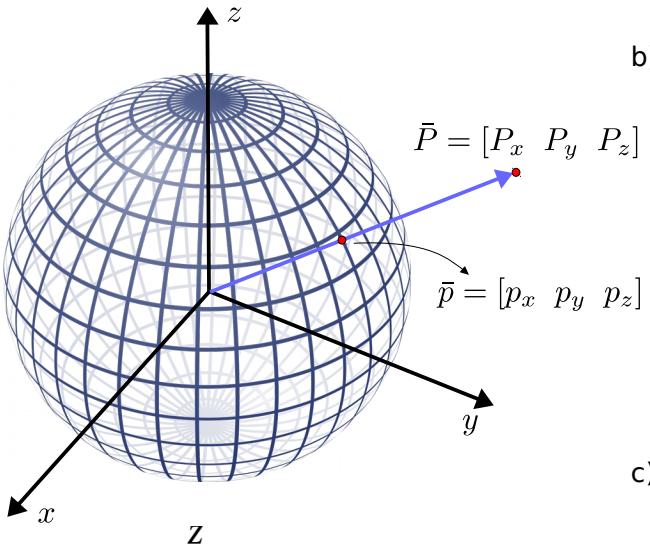
Assignment 2 – How to build a world in 3D

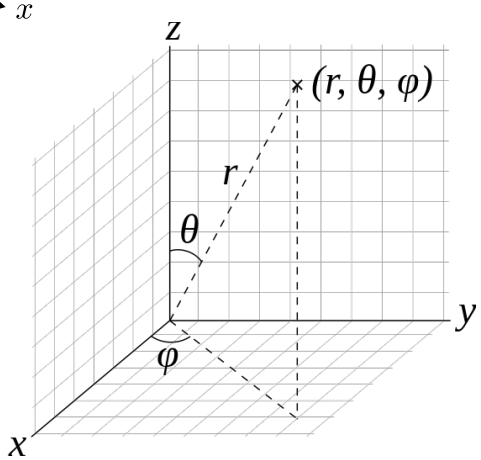
Part 1 – Image formation, projection, and coordinate frames

As we have seen in lecture, the standard model for image formation in computer graphics consists of a pinhole camera with a flat image plane. The projection of a scene point onto the image plane is obtained by finding the intersection between the ray from the center of projection to the point, and the image plane.

However, this is not the only common projection type in computer graphics. Other common projections include *cylindrical and spherical.* As a quick example, the retina in the human eye is roughly spherical. Spherical projection in particular is central to applications such as VR panoramic Imagery (see https://www.youtube.com/watch?v=DNT0OvsjK6Q).

a) [2 marks] Given a camera with a spherical image surface (and for simplicity, assume it is in front of the center of projection, just like we did for the flat image plane in lecture). Give equations to obtain the coordinates of point $\bar{p}$ which is the projection of a scene point $\bar{P}$ onto the spherical image surface. The radius of the sphere is f, the focal length.



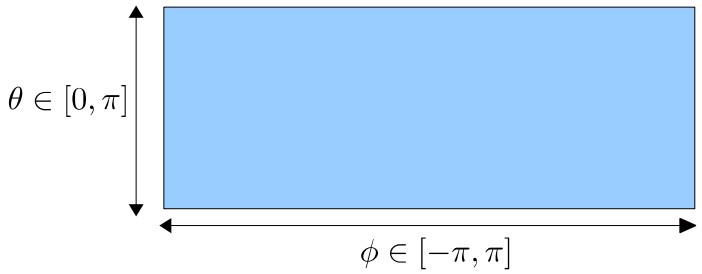


Standard representation for spherical coordinates (source: Wikipedia)

b) [2 marks] If we were to store the spherical image plane in memory, to render images onto it, we'd need to convert $\bar{p}$ into a 2-value index into our spherical image array. Since the sphere has known radius, we can use spherical coordinates θ, ϕ to do this.

Provide the expressions for the components of $\bar{p}$ in terms of spherical coordinates f, θ, ϕ and show how to solve for θ, ϕ

c) [2 marks] Consider the 2D array representing the spherically-projected scene:

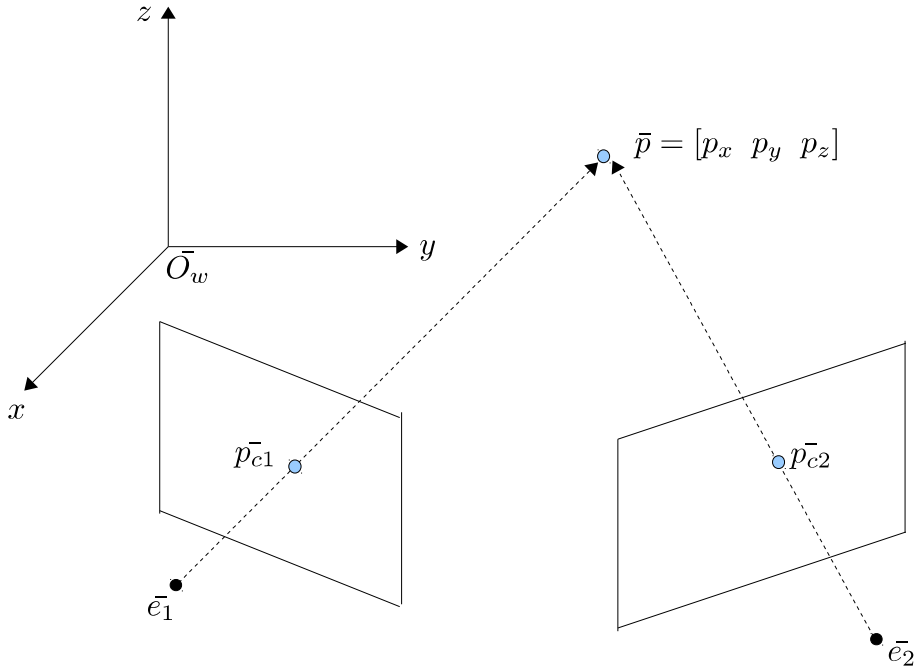


If we displayed this plane on screen as a flat image, we would see distortion along specific regions. Show where these would be. Why does this happen?

Attach work for this part immediately after this page

Part 1 (cont.)

Let's have a look at a pair of stereo-cameras, such a pair may be used, for example, to film 3D movies.



With e_1=[1 4 1]$_w$, e_2=[1 6 1]$_w$ (camera centers in world coordinates), and knowing That both cameras are looking at point p_l=[3 -2 3]$_w$, with up vector t=[0 0 1]$_w$, and both have a focal length f=1.

Assuming point ***p*** projects onto image location p_{c1}=[0.027836 -.309684 1.0]$_{im}$, and that ***p*** is at at *lambda*=-7.0191 units from e_1.

[2 marks] Obtain the ***world coordinates of point p***. Show each step of your work and explain what you are doing.

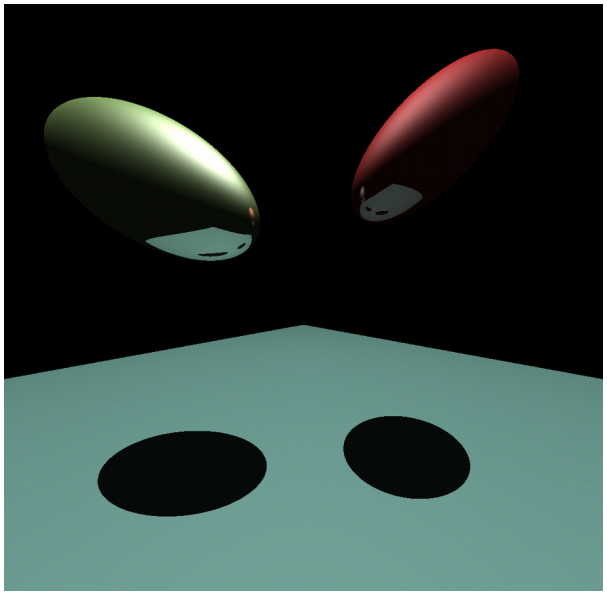
[3 marks] Obtain the coordinates of p_{c2}, the projection of ***p*** onto camera 2's image plane. Explain your work and show each step of the computation.

[4 marks] If we map all the points along the ray from e_1 to ***p*** onto the image plane of camera 2, they will form a line. Give the parametric equation of this line on camera 2's image plane, assuming the origin is at p_{c2}.

Note: *Please show how each of the vectors that make up the camera-to-world and World-to-camera coordinate conversion matrices were computed, as well as The matrices themselves. But* ***use a computer for the computations***.

(attach all work for this part ***immediately*** after this page)

Build your own Ray Tracer



Output of my solution for this assignment

Basic Ray Tracer [50 marks in total]

Your task is to implement a basic ray-tracer and render a simple scene using ray casting and the Phong shading model as described in lecture. The starter code sets up a scene with two spheres and a plane as shown above, illuminated by a single point light source. Your job is to render the scene by implementing the code fragments missing for object intersections, Phong illumination, and recursive ray casting.

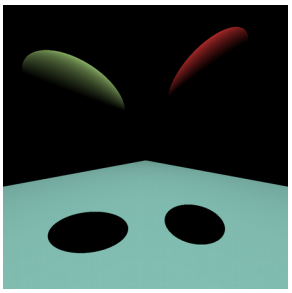
You will have to implement the following code fragments:

(a) ***[5 marks] Ray casting*** – computing a ray from the camera through each pixel and into the scene.
(b) ***[5 marks] Implement code for ray-sphere intersection***
(c) ***[5 marks] Implement code for ray-square intersection***
(d) ***[5 marks] Implement code for ray-cylinder intersection***
Note: All intersection code must work for affinely-deformed objects
(e) ***[5 marks] Compute the correct normals*** for affinely-deformed objects
(f) ***[10 marks] Phong illumination*** for a point light source (including ambient, diffuse, and specular components)
(g) ***[5 marks] Shadows***
(h) ***[10 marks] Reflection*** by recursive ray-casting

Note that your code will be auto-tested. So test every component thoroughly and for multiple different cases.

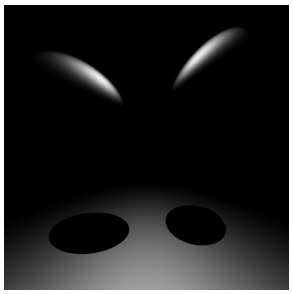
To demonstrate the working of your program, you must generate and submit along with your code the following images rendered at 512x512 pixels:

1. A scene signature where each pixel shows a unique color identifier for the first object hit (or background). This lets you test your ray casting/intersection code. You can generate this by making the ***ambient*** component to ***1.0***, and then setting the, ***diffuse***, ***specular, and global reflection*** components to zero.
   This image must be named: ***signature.ppm***



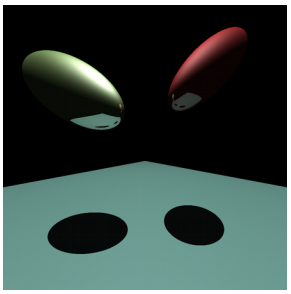
2. A rendered scene with only the diffuse and ambient components of the Phong model plus shadows. (set the ambient component for each object to ***0.1***)

   This image must be named: ***diffuse.ppm***



3. A rendered scene showing the specular component of the Phong model only (no ambient/diffuse). ***Note:*** Your highlights may look different from the image to the left (more spread out). That's normal, but they should be there, and have an *oval* shape. If your highlights are circles, something is wrong with the computation of transformed normals.

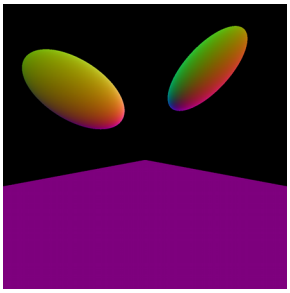
   This image must be named: ***specular.ppm***



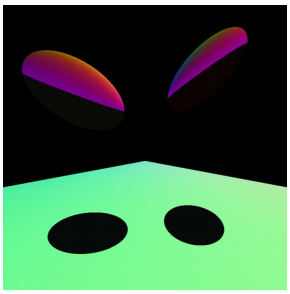
4. The final scene rendered with all components of the Phong model, including the global specular reflection Term.

   This image must be named: ***full.ppm***

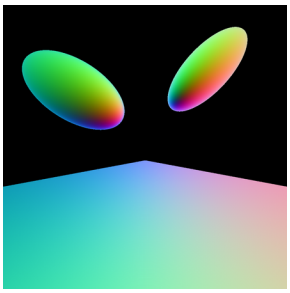
How to debug your code: Use Computer Graphics! You have a blank image, output to that image any quantities you need to look at. See the examples below, and make sure you understand what the images are telling you. The bottom line is: There is too much information going around inside the raytracer, printing values to the console will not be much help. An image can help you see what's going on if you use it well!



Example 1:
An image showing the components of the normal vector for each point on the objects in the scene.
R = (nx + 1)/2
G = (ny + 1)/2
B = (nz + 1)/2



Example 2:
An image showing the components of the ***r*** vector (perfect reflection direction) for each point in the scene



Example 3:
An image showing the components of the global reflection vector (the direction the ray hitting the objects is reflected toward).

You ***DO NOT*** have to submit images like the above, you are meant to use them to help you debug your code and make sure your computations are correct.

Remember, you can plot on the image pretty much any value being computed for a ray within your code! Use this extensively to ensure your code is working. It will save you time and help you find problems that are extremely hard to diagnose from printed values.

Once you notice something is not as it should, pick a pixel known to be in one of the spheres. Change the for loops so that the raytracer ***only*** processes that pixel. Print all values computed by your code (ray directions, intersection points, normals, and all vectors). Find and solve any problem(s). Start with a recursive depth of 1 and only increase it if everything else is working fine.

Compiling and running the starter code

Like all previous assignments. The raytracing code is designed to work on Linux. Therefore we expect you to do your work at the IC 406 lab. You can work remotely on Mathlab. This time the code will not produce a real-time visual display, so you do not have to worry about the graphical issues involved in working remotely over ssh. However, note that the code is fairly computationally intensive. You will be able to work faster if you are coding on a separate machine, as opposed to sharing the CPU with all other Mathlab users.

Use the included ***compile.sh*** script to compile the code.

Your final submission must compile and run without any form of glitch on Mathlab

Note: To encourage good coding practice, we will deduct up to

[10 marks] penalty - for badly designed code: that is, code lacking comments, confusing or unstructured code, or unnecessarily repetitive code.

Starter code in detail

utils.h, utils.c : Provide functions to manipulate points, vectors, matrices, and objects. This includes operations like dot products, matrix inversion, matrix-vector product, object creation and object transformations. There are parts in utils.c that you need to complete. Furthermore, you need to know the code in these functions since you will use these utilities extensively while building the raytracer.

RayTracer.h, RayTracer.c: These comprise the main code for the ray tracer. They follow the structure and flow of the ray tracer pseudo-code from the lecture notes.
You need to add code here to complete the ray tracer functionality.

svdDynamic.h, svdDynamic.c: This is a library used for matrix manipulation. You do not need to look at the code in any of these files.

Read carefully all the comments in the starter code – they describe in many places what you need to do. Parts of the code where work is needed are marked ***“TO DO”***.

What to submit:

- ***ALL*** your code. That means all .h and .c files as well as the compilation script.

- The images requested above (signature.ppm, diffuse.ppm, specular.ppm, and full.ppm).

- Completed *autotester_id.txt*

Submitting your work

You will submit a single, compressed file named:

RayTracer_studentNo1_studentNo2.tgz (e.g. RayTracer_11223344_55667788.tgz)

Example: From just outside your starter code directory (and ***after*** you remove any junk) do:

```
tar –cvfz RayTracer_11223344_55667788.tgz ./starter
submit –c cscd18f17 –a A2 –f RayTracer_11223344_55667788.tgz
```

General advice

- ***Start early:*** This is a fairly complex programming project, and requires both good C programming skills and understanding of the course material.

- ***Ask questions:*** Don't wait if you have problems. Ask your instructor or TA any questions related to the course material that arise while doing your work. If you have questions about the starter code, talk to the course instructor as well.

- ***Think, then code:*** The solution is reasonably short, but figuring out what the correct thing to do is requires thought and understanding. Do not start cobbling code together until you have understood what you should be doing.

- ***Have fun:*** Ray tracers are cool. After this assignment, you will understand how they work inside, and you will have your very own ray tracer to show off!

Marking Scheme	
Written problems	20 marks
Working code	50 marks (auto-tested)
In-class quiz	50 marks (individual)

Get Crunchy

Since you now have a working ray tracer, you may want to create a cool scene to show what it can do.

For bonus marks:

[5 marks] – Create a very cool scene. Use reflection and shadows cleverly, together with a sufficiently deep recursion, reflective surfaces can create interesting visual effects.

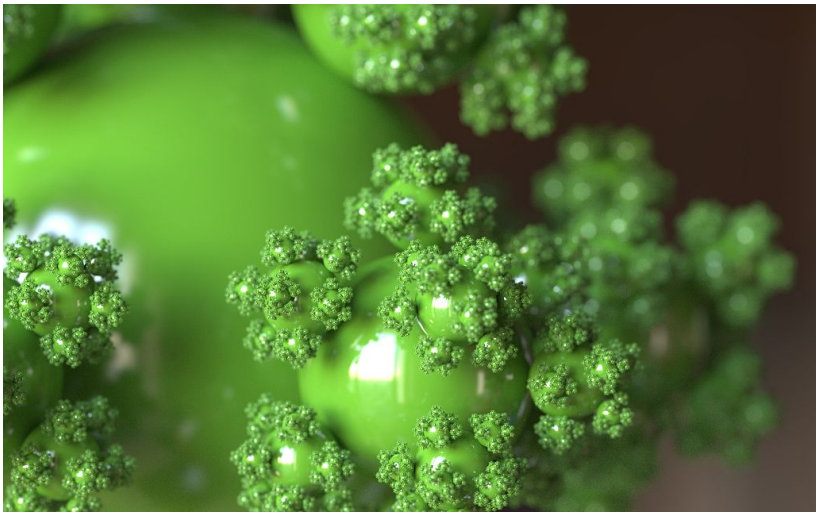
[10 marks] – Introduce procedural geometry.

Placing objects by hand can be tedious and limits your ability to create complex scenes (can you imagine how much time it would take to specify the position, size, and orientation of each leaf on every tree of a forest?)

Build functions that take a few input parameters and create interesting geometry algorithmically. For inspiration, have a look at ***structure synth***. A rendering engine that uses very simple scripts to quickly create complex geometry.

Apply what you've learned! (remember we built a snail shell in A1 from a simple parametric spiral, some circles, and transformations). Parametric curves can help you place objects along fairly complicated paths, and transforms will let you control their shape and orientation.

You could also look at L-systems – simple rule-based methods for producing structures that look like plants and trees.



Procedurally generated scene entirely made of spheres. A scene like this can easily be rendered (without the depth of Field effect) using your ray tracer
Image by Miles Bader, source: Wikipedia