# CMSC 435/634: Introduction to Computer Graphics

## Assignment 1
## Ray Tracing
## Initial submission September 11, 2018
## Due September 20, 2018 @ 11:59pm

### Before you start

You will be doing your work this semester in a class git repository. Before you get started on any of the assignments, you should fetch yourself a copy of your personal repository following these directions. Your personal class respoitory on the UMBC GL/Linux systems is `/afs/umbc.edu/users/a/d/adamb/pub/435/`*`your-user-name`*`.git`

To remotely clone over ssh execute: `git clone gl.umbc.edu:/afs/umbc.edu/users/a/d/adamb/home/public/435/`*`your-user-name`*`.git`

To use the `Eigen::Vector3d`, add `-I/usr/local/include/eigen3/` to your compilation flags and include `<Eigen/Dense>`.

To make sure you do not have last-minute problems, you must commit and push **something** to your repository by Tuesday, September 11th. This can be anything from an update to the readme to the completed project.

## The Assignment

For this assignment, you must write a C or C++ program that will render triangles using ray tracing and the barycentric coordinate intersection method. The input is in a simple text format called NFF, containing information about the view and objects in the scene. You will read an NFF scene from a file or stdin, and ray trace the single image described there. Rays that hit an object should be rendered in the triangle color, while rays that do not hit any object should use the background color of the scene. Do not try to include any of the more advanced ray tracing features you may read about (shadows, reflection, refraction, lights, etc.), we will get to those in the next assignment. Your output should be an image file in PPM format.

You can run my program here:

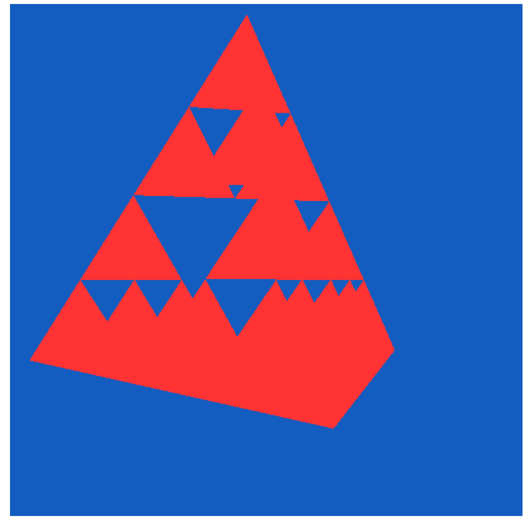
```
~adamb/public/hide/hide input.nff output.ppm
```

### Input

For the base assignment, you should be able to trace `tetra-3.nff`, which is checked into your `assn1` directory. Additional NFF format scenes can be generated using a set of programs called the 'Standard Procedural Databases'. A copy of these programs may be found in `~adamb/public/spd3.14/`. While NFF format is relatively simple, it does contain many features we will not be using in this assignment. You should be able to *read* any NFF format file, but ignore anything you do not implement. For the basic assignment, you should at least handle the "`v`" viewing specification, "`b`" background color, "`f`" object material specification (just the `r g b` color part, ignore the rest), and "`p`" polygon specification. "`c`" and "`pp`" are all multi-line, so you will at least need to recognize enough of those to know how much to skip. See here for more detail on the commands your program should support

**Please note: angle should be the *field of view* for the entire image, the angle subtended from the left edge of the image to the right edge. Additionally, pixels should be square. The documentation for .nff above adopts other conventions.**

Split any polygon with more than three vertices into a fan of triangles: given vertices 0,1,2,3,4, the triangles would be (0,1,2), (0,2,3), (0,3,4)



Since the SPD programs produce their output on stdout, if your ray tracer takes its input on stdin, you can pipe them together:

```
~adamb/public/spd3.14/tetra | ./hide
```

Since that has 4096 triangles and can be slow to raytrace, using the -s option can yield a simpler models. For example, tetra-3.nff with 64 triangles was generated with

```
~adamb/public/spd3.14/tetra -s 3 > tetra-3.nff
```

## Output File Format

The PPM format is one of the simplest image formats available. See the man page for 'ppm' on the university GL server for a description.

To create a PPM file, first you should store your image in an array of bytes in y/x/color index order:

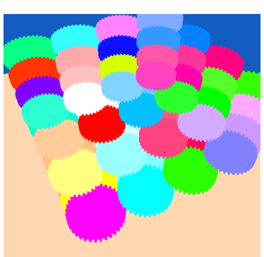
```
unsigned char pixels[HEIGHT][WIDTH][3];
```

When filling in this array, remember that it is in y/x order, not the more familiar x/y order. The final index is the color component, with r=0, g=1 and b=2. Color values range from 0 to 255. For example, this would store a floating point color value of .5 into the green component at x,y:

```
pixels[y][x][1]= .5*255;
```

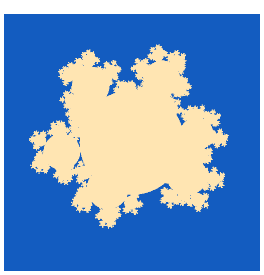
Once you've filled in the pixels array, actually writing the PPM file is quite simple:

```
FILE *f = fopen("hide.ppm","wb");
fprintf(f, "P6\n%d %d\n%d\n", WIDTH, HEIGHT, 255);
fwrite(pixels, 1, HEIGHT*WIDTH*3, f);
fclose(f);
```



## Extra Credit

For up to 25 points of extra credit, handle concave polygons. Splitting polygons into a fan of triangles only works for convex polygons. Instead, use the test-ray method to handle arbitrary concave polygons. Try the "gears" SPD program to generate a scene with some pretty complex polygons.



For up to 15 points of extra credit, implement spheres too. Try the `balls` SPD program for a scene with lots of spheres.

### Other people's code

Ray tracing is a popular rendering technique, and the internet contains lots of resources for ray tracers in general and things like ray-object intersection in particular. Other than the PPM snippet above, **YOU MAY NOT USE ANY OUTSIDE CODE**. All code that you use must be your own.

## Strategy

This is a big assignment. Start **NOW**, or you will probably not finish. No, really, I promise you will not be able to do it in the last two days. Even before we get to all of the details of the ray tracing itself, you can still start working on your file parsing.

Some implementation tips from a previous instructor.

## What to turn in

Turn in this assignment electronically by pushing your source code to your class git repository by 11:59 PM on the day of the deadline. Do your development in the proj1 directory so we can find it. Be sure the Makefile will build your project when we run '`make`' (or edit it so it will). Also include a `README.txt` file telling us about your assignment. **Do not forget to tell us what (if any) help did you receive from books, web sites or people other than the instructor and TA.**

Check in along the way with useful checkin messages. We will be looking at your development process, so a complete and perfectly working ray tracer submitted in a single checkin one minute before the deadline will NOT get full credit. Do be sure to check in all of your source code, Makefile, README, and updated .gitignore file, but no build files, log files, generated images, zip files, libraries, or other non-code content.

To make sure you have the submission process working, you must do at least one commit and push by the Saturday two weeks before the deadline.

(This page: www.csee.umbc.edu/~adamb/435/proj1.html)

# CMSC 435/634: Introduction to Computer Graphics

## Assignment 2
## Ray Tracing II
## Due October 2, 2018 @ 11:59 PM

### Before you start

This assignment builds on the previous one. If yours worked, I strongly recommend using your own code as a starting point for this assignment. If you were not able to get the first part of your ray tracer working, I will provide access to my program in `~adamb/public/hide` (after the late submission deadline for assignment 1). You could either use this code to figure out what was wrong with your assignment, or use it as a base for this one. I have also placed an executable for my program in `~adamb/public/shade`.

## The Assignment

For this assignment, you will add mirror reflection and diffuse and Blinn-Phong specular lighting with shadows to your ray tracer. I have provided an example `teapot.nff` (spd3.14/teapot -s 3) that you can use for testing (In addition to the other spd models). Note that the teapot is made up of polygonal patches (abbreviation pp), so your program will need to support this input. You can simply ignore the normal information.

You will need to use additional parameters from the '`f`' lines in the nff file. The full set of '`f`' parameters is

`f` *`r g b Kd Ks e Kt ir`*

Where `r,g,b` is the base surface color, `Kd` is a coefficient to scale the overall diffuse contribution, `Ks` is the coefficient for both reflection rays and a Blinn-Phong glossy specular component with exponent `e`, `Kt` is the coefficient for refracted rays, and `ir` is the index of refraction.

In assignment 1, you needed to track the color at the closest intersection. For this assignment, you will also need to track the surface normal, intersection location, and additional surface parameters. Once you find that intersection, you will need to compute the surface shading. Given unit normal, `N`; unit vector from the surface intersection point toward the light, `L`; unit vector halfway between the light and ray, `H`; and light intensity of `1/sqrt(numLights)`, sum the contribution from each light. For each light, cast a shadow ray from the surface to the light. If that light is not shadowed, the diffuse and specular contribution from that one light would be computed as:

```
diffuse = max(0, dot(N,L))
specular = pow(max(0, dot(N,H)), e)
localColor.r += (Kd*r*diffuse + Ks*specular) * lightIntensity
localColor.g += (Kd*g*diffuse + Ks*specular) * lightIntensity
localColor.b += (Kd*b*diffuse + Ks*specular) * lightIntensity
```

Since the color of each reflection ray is the total color of whatever it hits, including its own diffuse, specular and reflection, compute the color in a recursive process:

```
Color trace(ray) {
  ...
  totalColor = localColor;
  if (Ks > 0 && !recursionLimit)
    totalColor += Ks*trace(reflectionRay)
  return totalColor
}
```

Stop the recursion if the number of bounces would be greater than 5. The easiest way to do this is to add data to each ray to track the "depth" of the ray---the number of bounces.

### Extra Credit

For up to 20 points of extra credit, implement stratified sampling ("jittering") for antialiasing by shooting multiple, jittered, rays per pixel. You should allow the user to specify the square root of the number of rays as a command line option (see getopt). The number of samples will be the square of the parameter (i.e. a parameter of 3 results in 9 samples), arranged in a jittered 3x3 grid). At right is the output of my program with:

```
./shade -j -s 3 teapot-3.nff stratified.ppm
```

For up to 25 points of extra credit, implement depth of field by shooting multiple rays per pixel. You should allow the user to specify the square root of the number of rays and aperture size as command line options. The focal plane should pass through the 'at' point specified in the nff file. Note that if implementing depth-of-field and stratified sampling bothe the lens and the image should be sampled for each ray. At right is the output of my program with:

```
./shade -a 0.5 -s 3 teapot-3.nff dof.ppm
```

For up to 20 points of extra credit, implement refraction too. If your ray tracer supports concave polygons, the gears nff files include refaction. I have also provided an edited tetra version of `teapot-3.nff` called `refract.nff` that has been edited to use for refraction testing.

For up to 20 points of extra credit, implement phong/smooth shading with the "-p" command line option. At right is the output of my program with:

```
./shade -p teapot-3.nff phong.ppm
```

## What to turn in

Turn in this assignment electronically by pushing your source code to your `assn2` GIT directory by 11:59 PM on the day of the deadline. We will be looking for multiple checkins documenting your development process.

As always, double check that you have submitted **everything** we need to build and run your submission, but **no** generated files (.o's, executables, or images). Be sure to include a Makefile that will build your project when we run '`make`', and a `readme.txt` file telling us about your assignment. Do not forget to tell us what (if any) help did you receive from books, web sites or people other than the instructor and TA.

(This page: www.csee.umbc.edu/~adamb/435/proj2.html)