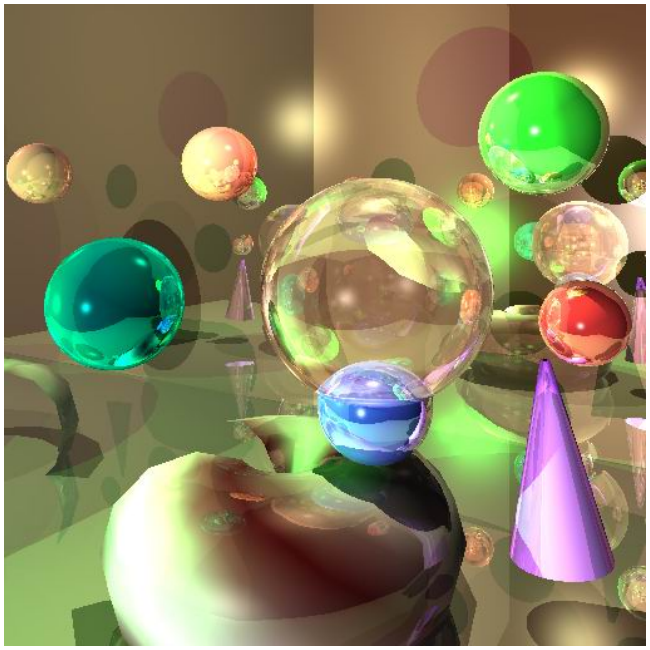
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

ISTD 50.017 Graphics & Visualization

Spring Semester 2016 (Term 7)



Assignment 5 : Ray Tracing

In this assignment, you will greatly improve the rendering capabilities of your ray caster by adding several new features. First, you will improve the shading model by recursively generating rays to create reflections and shadows. Second, you'll add procedural solid texturing. Finally, Implement supersampling to fix aliasing problems.

1. Getting Started

Note that this assignment is non-trivial. Please start as early as possible. An updated parser, sample solution, and new scene files for this project are included in the assignment distribution. Run the sample solution **a5soln** as follows:

./a5soln -input scene10_vase.txt -size 300 300 -output output01.bmp -shadows

or

./a5soln -input scene10_vase.txt -size 300 300 -output output01.bmp -shadows -jitter -filter

These will all generate an image named output01.bmp with different effects. We'll describe the rest of the command-line parameters later. When your program is complete, you will be able to render this scene as well as well as the other test cases given below.

2. Summary of Requirements

As mentioned above you'll be adding several new features to your ray caster to improve its realism and performance.

New features

- Recursive ray tracing for shadowing and reflections.
- Procedural solid textures.
- Supersampling for antialiasing.

We go into more detail about these features below in the implementation notes.

Your code will be tested using a script on all the test cases below. Make sure that your program handles the exact same arguments as the examples below. For your convenience, you can assume that “-jitter”, “-shadows” and “-filter” are always given. That is you can ignore these arguments and always run your ray-tracer with these features enabled.

3. Starter Code

For this assignment, part of the starter code is your own from Assignment 4. We updated some files and you need to merge your code with these updated files. Before start merging, make sure you have a back up copy of your code in case you damage your own files by accident during the merge. Now copy all files from the new starter code to your own code and overwrite everything.

There's a new scene parser to handle new material types. If you had to modify **SceneParser.cpp/hpp** for assignment four, remember to make the same modifications to the new scene parser we provide for you. Also, there is updated **Mesh.cpp/hpp** to accelerate intersection test speed. To help build procedural shaders, we provide you with some procedural noise routines. Finally, we give you a new **Material** class to handle new materials. Please replace your own **Material** with the new one.

SceneParser::getBackgroundColor(Vector3f dir) now takes the ray direction as an argument. Please update your main loop accordingly.

4. Implementation Notes

4.1. Recursive Rays

After merging, you will add some global illumination effects to your ray caster. Because you cast secondary rays to account for shadows, reflection and refraction, you can now call it a ray tracer. You will encapsulate the high-level computation in a **RayTracer** class that will be responsible for sending rays and recursively computing colors along them.

To compute cast shadows, you will send rays from the visible point to each light source. If an intersection is reported, the visible point is in shadow and the contribution from that light source is ignored. Note that shadow rays must be sent to all light sources. To add reflection (and refraction) effects, you need to send secondary rays in the mirror (and transmitted) directions, as explained in lecture. Refer to the document for more details. The computation is recursive to account for multiple reflections (and or refractions).

1. Make sure your **Material** classes store refraction index, which are necessary for recursive ray tracing.

2. Create a new class **RayTracer** that computes the radiance (color) along a ray. Update your main function to use this class for the rays through each pixel. This class encapsulates the computation of radiance (color) along rays. It stores a pointer to an instance of **SceneParser** for access to the geometry and light sources. Your constructor should have these arguments (and maybe others, depending on how you handle command line arguments):
   RayTracer(SceneParser* s, int max bounces, ...);
   where max bounces is the maximum number of bounces (recursion depth) a light ray has. The main method of this class is **traceRay** that, given a ray, computes the color seen from the origin along the direction. This computation is recursive for reflected (or transparent) materials. We therefore need a max bounces to prevent infinite recursion. The maximum recursion depth which is passed as a command line arguments to the program with

-bounces max bounces.
Try that with the solution. **traceRay** takes the current number of bounces (recursion depth) as one of its parameters.
Vector3f traceRay(Ray& ray, float tmin, int bounces, Hit& hit) const;

3. Now is a good time to make sure you did not break any prior functionality by testing against scenes from the last assignment as well as the new scenes. You can do this by setting max_bounces=0. Check that you get the same results except that the specular component is removed.

4. (Optional) Add support for the new command line arguments: **-shadows**, which indicates that shadow rays are to be cast, and **-bounces**, which control the depth of recursion in your ray tracer.

5. Implement cast shadows by sending rays toward each light source to test whether the line segment joining the intersection point and the light source intersects an object. If there is an intersection, then discard the contribution of that light source. Recall that you must displace the ray origin slightly away from the surface, or equivalently set **tmin** to some ε.

6. Implement mirror reflections for reflective materials by sending a ray from the current intersection point in the mirror direction. For this, we suggest you write a function:
Vector3f mirrorDirection(const Vector3f& normal, const Vector3f& incoming);
Trace the secondary ray with a recursive call to **traceRay** using modified recursion depth. Make sure that **traceRay** checks the appropriate stopping conditions. Add the color seen by the reflected ray times the reflection color to the color computed for the current ray. If a ray didn't hit anything, simply return the background by **SceneParser::getBackgroundColor(dir)**.

7. Simple refraction. Cast refraction rays for transparent materials (**if Material::refractionIndex>0**). Air has refraction Index 1. You can assume you always start in air. The starter code keeps track of current refraction index as an additional argument to **traceRay**. This will not work properly for transparent objects that are inside other transparent objects. Choose your own implementation if you would like. We suggest you write a function:
bool transmittedDirection(const Vector3f& normal, const Vector3f& incoming, float index_n, float index_nt,Vector3f & transmitted);
index_n and **index_nt** are refraction indices of the current object and the object the ray is going into. Refer to here for a complete explanation.
Let **d** be the direction of the incoming light ray, **N** be the normal. Note that if the ray is currently inside the object, **d**．**N** will be positive.
Let n and n_t be the refraction indices of the two mediums. The refracted direction t is

$$\mathrm{t}=\frac{n(\boldsymbol{d}-\boldsymbol{N}(\boldsymbol{d}\cdot\boldsymbol{N})}{n_t}-\boldsymbol{N}\sqrt{1-\frac{n^2(1-(\boldsymbol{d}\cdot\boldsymbol{N})^2)}{n_t^2}}$$

First, you need to decide whether there is refraction or not and return false if there is no refraction (total reflection). This is done by checking that the quantity under the square root is positive.
Then, if there is refraction, you should use the same **traceRay** function to get a color of the recursive ray.
Finally, you need to blend the reflection color and refraction color using Schlick's approximation to Fresnel's equation. We are going to compute a weight R for the reflection color and use $1-R$ for refraction color.
R is given by

$$R=R_0+(1-R_0)(1-c)^5, R_0=(\frac{n_t-n}{n_t+n})^2.$$
$$c=\begin{cases}abs(\boldsymbol{d}\cdot\boldsymbol{N}), & n\le n_t\\ abs(\boldsymbol{t}\cdot\boldsymbol{N}), & n>n_t\end{cases}.$$

4.2. Perlin Noise

Next you'll build materials using Perlin Noise, which lets you to add controllable irregularities to your procedural textures. Here's what Ken Perlin says about his invention:

"Noise appears random, but isn't really. If it were really random, then you'd get a different result every time you call it. Instead, it's "pseudo-random"—it gives the appearance of randomness. Its appearance is similar to what you'd get if you took a big block of random values and blurred it (ie: convolved with a Gaussian kernel). Although that would be quite expensive to compute."

So we'll use his more efficient (and recently improved) implemenation of noise translated to C++ in **PerlinNoise.{h,cpp}**.

PerlinNoise::octaveNoise is implemented for you. It computes

$$N(x, y, z) = noise(x, y, z) + noise(2x, 2y, 2z)/2 + noise(4x, 4y, 4z)/4 + \ldots,$$

where (x, y, z) is simply the global coordinate.

Ken Perlin's original paper and his online notes have many cool examples of procedural textures that can be built from the noise function. You will implement a simple **Marble** material. This material uses the **sin** function to get bands of color that represent the veins of marble. These bands are perturbed by the noise function as follows:

$$M(x, y, z) = sin(\omega x + aN(x, y, z))$$

Fill in the class Noise which calls the function **PerlinNoise::octaveNoise** with proper arguments to obtain $N(x, y, z)$. Then compute $M(x, y, z)$. The value of $M(x, y, z)$ will be a floating point number that you should clamp and use to interpolate between the two contained colors. Here's the constructor for **Noise**:

Noise(int octaves , const Vector3f & color1, const Vector3f & color2, float frequency, float amplitude);

where frequency is w and amplitude is a in the equation above.

Try different parameter settings to understand the variety of appearances you can get. Remember that this is not a physical simulation of marble, but a procedural texture attempting to imitate our observations.

4.3. Anti-aliasing

Next, you will add some simple anti-aliasing to your ray tracer. You will use supersampling and filtering to alleviate jaggies and Moire patterns.

1. Jittered Sampling.
   For each pixel, instead of getting a color with one ray, we can sample multiple rays perturbed randomly. You are required to subdivide a pixel into 3 × 3 sub-grids. This is equivalent to rendering an image with 3x resolution. Then, for each pixel at (i, j) in the high resolution image, instead of generating a ray just using (i, j), generate two random numbers r_i, r_j in range [−0.5, 0.5] to get $(i + r_i, j + r_j)$.

2. Gaussian blur.
   In the solution, we use the kernel K = (0.1201, 0.2339, 0.2931, 0.2339, 0.1201). First blur the image horizontally and then blur the blurred image vertically.
   To blur an image horizontally Each pixel color $I'(i, j)$ in the new image is computed as a weighed sum of pixels in the original image.
   $$I'(i, j) = I(i, j - 2)K(0) + I(i, j - 1)K(1) + I(i, j)K(2) + I(i, j + 1)K(3) + I(i, j + 2)K(4).$$
   Note that you probably want to allocate a new image to do the blurring to avoid bugs. If $(i, j - 2)$ is out of image boundary, for example, simply use $(i, 0)$ and so on.

3. Down-sampling. To get from the high-resolution image back to the original specified

resolution, average each 3 × 3 neighborhood of pixels to get back 1 pixel.

4. (Optional) Handle commandline arguments:
   - **-jittered** Enable jittered sampling.
   - **-filter** Enable Gaussian smoothing and down-sampling.

   If you choose to ignore these arguments, your program should always enable these two steps.

5. Test Cases

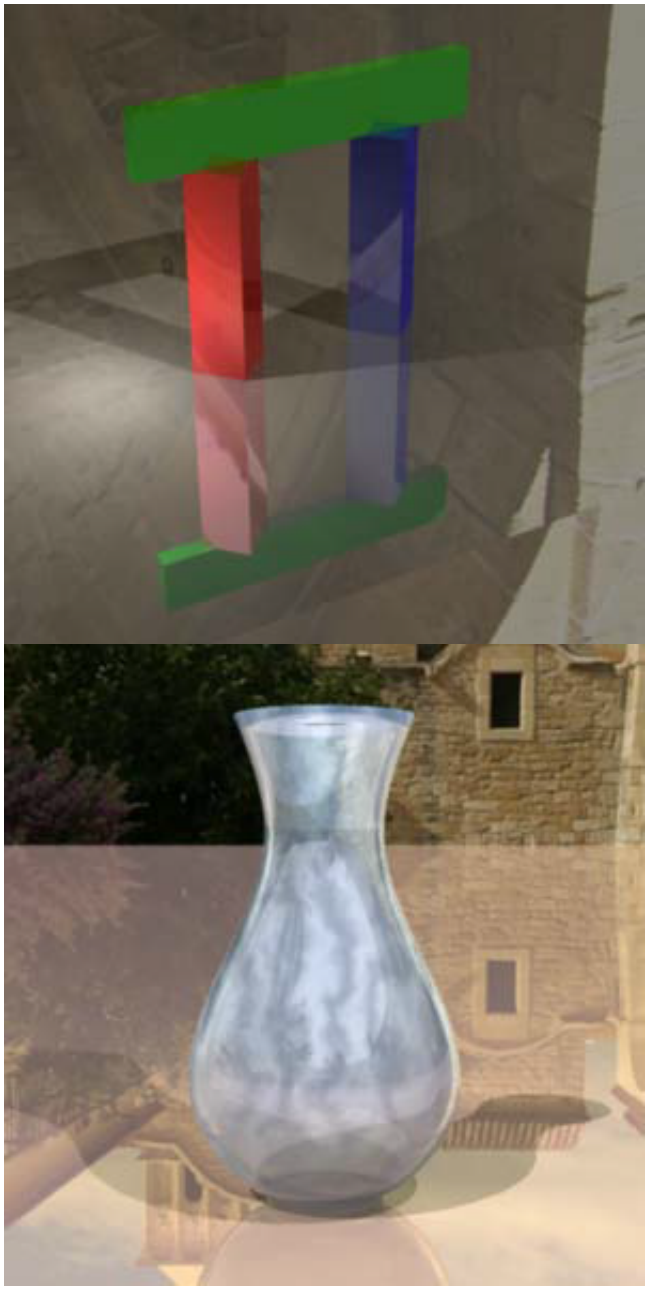
Your assignment will be graded by running a script that runs these examples below. Make sure your ray caster produces the same output (up to visual perception).

./a5 -input scene06_bunny_1k.txt -size 300 300 -output 6.bmp -shadows -bounces 4 -jitter -filter

./a5 -input scene10_sphere.txt -size 300 300 -output 10.bmp -shadows -bounces 4 -jitter -filter

./a5 -input scene11_cube.txt -size 300 300 -output 11.bmp -shadows -bounces 4 -jitter -filter

./a5 -input scene12_vase.txt -size 300 300 -output 12.bmp -shadows -bounces 4 -jitter -filter



6. Hints

- Incremental debugging. Implement and test one primitive at a time. Test one shading a time. Ambient, diffuse, specular, and then texture.
- Use a small image size for faster debugging. 64 × 64 pixels is usually enough to realize that something might be wrong.
- As usual, don't hesitate to print as much information as needed for debugging, such as the direction vector of the rays, the hit values, etc. When you get weird results, don't hesitate to use simple cases, and do the calculations manually to verify your results. Perhaps instead of casting all the rays needed to create an image, just cast a single ray (and its subsequent recursive rays).
- Modify the test scenes to reduce complexity for debugging: remove objects, remove light sources, change the parameters of the materials so that you can view the contributions of the different components, etc.
- Use **assert()** to check function preconditions, array indices, etc. See **cassert**.
- The "very large" negative and positive values for t used in the **Hit** class and the intersect routine can simply be initialized with large values relative to the camera position and scene dimensions. However, to be more correct, you can use the positive and negative values for infinity from the IEEE floating point standard.

- Parse the arguments of the program in a separate function. It will make your code easier to read.
- Implement the normal visualization and diffuse shading before the transformations.
- Use the various rendering modes (normal, diffuse, distance) to debug your code. This helps you locate which part of your code is buggy.
- You do not need to declare all methods in a class virtual, only the ones which subclasses will override.
- To avoid a segmentation fault, make sure you don't try to access samples in pixels beyond the image width and height. Pixels on the boundary will have a cropped support area.

7. Extra Credits (ISTD)

Most of these extensions require that you modify the parser to take into account the extra specification required by your technique. Make sure that you create (and turn in) appropriate input scenes to show off your extension.

7.1 Easy

Create a wood material or other procedural material that uses Perlin Noise.

Add more interesting lights to your scenes, e.g. a spotlight with angular falloff.

Render with depth of field (if you didn't do this last time).

7.2 Medium


Load or create more interesting complex scenes. You can download more models and scenes that are freely available online.


Bidirectional Texture Functions (BTFs): make your texture lookups depend on the viewing angle. There are datasets available for this online.


Bump mapping: look up the normals for your surface in a height field image or an normal map. This needs the derivation of a tangent frame. There many such free images and models online.


Render glossy surfaces.


Render interesting BRDF such as milled surface.


Uniform Grids. Create a 3D grid and "rasterize" your object into it. Then, you march each ray through the grid stopping only when you hit an occupied voxel. Difficult to debug.


Simulate dispersion (and rainbows). The rainbow is difficult, as is the Newton prism demo.


Make a little animation (10 sec at 24fps will suffice). E.g, if you implemented depth of field, show what happens when you change camera focal distance. Move lights and objects around.


Add motion blur to moving objects in your animation.


Add area light sources and Monte-Carlo integration of soft shadows.

7.3 Hard


Irradiance caching.


Subsurface scattering. Render some milk, jade etc.


Path tracing with importance sampling, path termination with Russian Roulette, etc.


Raytracing through a volume. Given a regular grid encoding the density of a participating medium such as fog, step through the grid to simulate attenuation due to fog. Send rays towards the light source and take into account shadowing by other objects as well as attenuation due to the medium. This will give you nice shafts of light.


Animate some water pouring into a tank or smoke rising.


Photon mapping with kd-tree acceleration to render caustics.

8. Submission

You are to write a README.txt (or optionally a PDF) that answers the following questions:

- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.

- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. *This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.*

- Did you do any extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what and how you did it.

- Got any comments about this assignment that you'd like to share? Was it too long? Too hard? were the requirements unclear? Did you have fun, or did you hate it? Did you learn something, or was it a total waste of your time? Feel free to be brutally honest; we promise we won't take it personally.

Submit your assignment online. Please submit a single archive (.zip or .tar.gz) containing:

- Your source code.

- A compiled executable built from your code name `a5`.

- The README file.

- Any additional files necessary to run your program.