CSC4140 Assignment VI

Computer Graphics

April 9, 2022

Ray Tracing

This assignment is 10%(with 2 extra credit) of the total mark.

Strict Due Date: 11:59PM, April 22^{th}, 2022

Student ID:

Student Name:

This assignment represents my own work in accordance with University regulations.

Signature:

Overview

You will implement the core routines of a physically-based renderer using a pathtracing algorithm. This assignment reinforces many of the ideas covered in class recently, including ray-scene intersection, acceleration structures, and physically based lighting and materials. By the time you are done, you'll be able to generate some realistic pictures (given enough patience and CPU time). You will also have the chance to extend the assignment in many technically challenging and intellectually stimulating directions.

Basically, this project has 5 tasks, worth a total 100 points. The tasks are as follows:

- Part 1: Ray Generation and Scene Intersection (20 points)
- Part 2: Bounding Volume Hierarchy (20 points)
- Part 3: Direct Illumination (20 points)
- Part 4: Global Illumination (20 points)
- Part 5: Adaptive Sampling (20 points)

You will also need to read these articles:

- Experiments, Report and Deliverables
- Optional Extra Credit Opportunities (20 points)

It will be also very helpful to read this document on CGL vectors library:

- CGL Vectors Library Documentation

In particular, please read the Experiments, Report, and Deliverables page before beginning the project. There are many deliverables for this project, so please plan accordingly. Several parts of this assignment ask you to compare various methods/implementations, and we don't want you to be caught off guard!

Running the Executable

The executable, **pathtracer**, must be run with a COLLADA file (.dae) when you invoke the program. COLLADA files use an XML-based schema to describe a scene graph (much like SVG). They are a hierarchical representation of all objects in the scene (meshes, cameras, lights, etc.), as well as their coordinate transformations.

There are many command line options for this project, which you can read about below. Use these between the executable name and the dae file.

For example, to simply run the regular GUI with the *CBspheres_lambertian.dae file and 8 threads*, you could type:

```
./pathtracer -t 8 ../dae/sky/CBspheres_lambertian.dae
```

Unlike previous assignments, we've provided a windowless run mode, which is triggered by providing a filename with the *-f* flag. This is useful for rendering if you're ssh-ed into an instructional machine, for example, or if you want to render on your own machine without opening the application window.

If you wanted to save directly to the *spheres_64_16_6.png* file with 64 samples per pixel, 16 samples per light, 6 bounce ray depth, and 480x360 resolution, you might rather use something like this:

```
./pathtracer -t 8 -s 64 -l 16 -m 6 -r 480 360 -f
 spheres_16_4_6.png ../dae/sky/CBspheres_lambertian.dae
```

Note: We recommend running with **4-8 threads almost always** – the exception is that you should use *-t* 1 **when debugging with print statements**, since printf is not guaranteed to be thread safe. (However, cout/cin are thread safe) Furthermore, if you are using a virtual machine, **make sure that you allocate multiple CPU cores to it**. This is important in order for your *-t* flag to work. In practice, it's good to assign $N - 2$ cores to the virtual machine, where$N$ is the number of your physical machine's number of CPU cores or hyperthreaded cores.

Speeding up Rendering Tests High quality renders take a long time to complete! During development and testing, don't spend a long time waiting for full-sized, high quality images. **Make sure you use the parameters specified in the writeup for your report, though!**

Here's some ways to get quick test images:

- Render with fewer samples!
- Utilize the cell rendering feature - start a render with R, then hit C and click-drag on the viewer. The pathtracer will now only render pixels inside the rectangular region you selected.
- Set a smaller window size using the -r flag (example: *./pathtracer -t 8 -s 64 -r 120 90 ../dae/sky/CBspheres_lambertian.dae*). Zoom out until the entire scene is visible in the window, then start a render with R.

Using the Executable and Graphical User Interface (GUI)

Command line options

Flag and parameters	Description
-s <INT>	Number of camera rays per pixel (default=1, should be a power of 2)
-l <INT>	Number of samples per area light (default=1)
-t <INT>	Number of render threads (default=1)
-m <INT>	Maximum ray depth (default=1)
-f <FILENAME>	Image (.png) file to save output to in windowless mode
-r <INT> <INT>	Width and height in pixels of output image (if windowless) or of GUI window
-p <x> <y> <dx> <dy>	Used with the -f flag (windowless mode) to render a cell with its upper left corner at [x,y] and spanning [dx, dy] pixels.
-c <FILENAME>	Load camera settings file (mainly to set camera position when windowless)
-a <INT> <FLOAT>	Samples per batch and tolerance for adaptive sampling
-H	Enable hemisphere sampling for direct lighting
-h	Print command line help message

Moving the Camera (in Edit and BVH mode)

Mouse	Action
Left-click and drag	Rotate
Right-click and drag	Translate
Scroll	Zoom in and out
Spacebar	Reset view

Keyboard Commands

Key	Action
E	Mesh-edit mode (default)
V	BVH visualizer mode
$\leftarrow$/$\rightarrow$	Descend to left/right child (BVH viz)
$\uparrow$	Move up to parent node (BVH viz)
R	Start rendering
S	Save a screenshot
$-$/$+$	Decrease/increase area light samples
[/]	Decrease/increase camera rays per pixel
$<$/$>$	Decrease/increase maximum ray depth
C	Toggle cell render mode
H	Toggle uniform hemisphere sampling
D	Save camera settings to file

Cell render mode lets you use your mouse to highlight a region of interest so that you can see quick results in that area when fiddling with per pixel ray count, per light ray count, or ray depth.

Scene editor and GUI function tester

This project ships with an optional scene editor and function tester called Visual Debugger. In the scene editor, you can view the scene hierarchy, move objects, modify the BSDF, lighting, etc. In the function tester, we have setup two tests: one for *sample_L* with lights, one for ray-triangle or ray-sphere intersection. These two can be very useful to validate whether you have implemented correct ray-scene intersection functions.

You can uncomment line 25 of *src/application/visual_debugger.cpp* to enable this feature:

// #define ENABLE_VISUAL_DEBUGGER

This will appear as a separate pop-up window. You can take a look at how these GUI based testing functions are written using ImGUI in *src/application/visual_debugger.cpp* and add your own testing functions there. This can be very helpful for tracking down bugs in your project.

If you want to move a light in the scene, you need to move the light itself, as well as its corresponding mesh with an emmissive BSDF. The structure of our project currently requires the light as well as the mesh for the light to be separate. If you only move one of these, your scene will appear incorrectly lit. This bug also causes our hemisphere sampling image does not correspond with importance sampling image. If you encounter this, don't worry about it.

dae Files for Debugging

We have provided two dae files for debugging purposes, dae/simple/cube.dae and dae/simple/plane.dae, which render a cube and a plane respectively. Please feel free to edit these in Blender/directly from the file to help with debugging.

Basic Code Pipeline

What happens when you invoke pathtracer in the starter code? Logistical details of setup and parallelization:

1.The *main()* function inside main.cpp parses the scene file using a *ColladaParser* from *collada/collada.h*.

2. A new *Viewer* and *Application* are created. *Viewer* manages the low-level OpenGL details of opening the window, and it passes most user input into *Application*. *Application* owns and sets up its own *pathtracer* with a camera and scene.

3. An infinite loop is started with *viewer.start()*. The GUI waits for various inputs, the most important of which launch calls to *set_up_pathtracer()* and *PathTracer::start_raytracing()*.

4. *set_up_pathtracer()* sets up the camera and the scene, notably resulting in a call to *PathTracer::build_accel()* to set up the BVH.

5. Inside *start_raytracing()* (implemented in *pathtracer.cpp*), some machinery runs to divide up the scene into "tiles," which are put into a work queue that is processed by *numWorkerThreads* threads.

6. Until the queue is empty, each thread pulls tiles off the queue and runs *raytrace_tile()* to render them. *raytrace_tile()* calls *raytrace_pixel()* for each pixel inside its extent. The results are dumped into the pathtracer's *sampleBuffer*, an instance of an *HDRImageBuffer* (defined in image.h).

Most of the core rendering loop is left for you to implement.

1. Inside *raytrace_pixel()*, you will write a loop that calls *camera->generate_ray(...)* to get camera rays and *est_radiance_global_illumination(...)* to get the radiance along those rays.

2. Inside *est_radiance_global_illumination*, you will check for a scene intersection using *bvh->intersect(...)*. If there is an intersection, you will accumulate the return value in *Spectrum L_out*.

1. adding the BSDF's emission with *zero_bounce_radiance* which uses *bsdf->get_emission()*,

2. adding global illumination with *at_least_one_bounce_radiance*, which calls *one_bounce_radiance*

(which calls a direct illumination function), and recursively calls itself as necessary.

You will also be implementing the functions to intersect with triangles, spheres, and bounding boxes, the functions to construct and traverse the BVH, and the functions to sample from various BSDFs.

Approximately in order, you will edit (at least) the files

1. pathtracer/pathtracer.cpp (PART 1, 3, 4, 5)
2. pathtracer/camera.cpp (PART 1)
3. scene/triangle.cpp (PART 1)
4. scene/sphere.cpp (PART 1)
5. scene/bvh.cpp (PART 2)
6. scene/bbox.cpp (PART 2)
7. pathtracer/bsdf.cpp (PART 3)

You will want to skim over the files:

1. pathtracer/ray.h
2. pathtracer/intersection.h
3. pathtracer/sampler.h
4. pathtracer/sampler.cpp
5. util/random_util.h
6. scene/light.h
7. scene/light.cpp

since you will be using the classes and functions defined therein.

In addition, *ray.h* contains a defined variable **PART**. This is currently set to 1. You may use this variable if you like to test separate parts independently. For example, you can write things like if *(PART != 4) return false;* to easily "revert" parts of your code. **In particular, you will need to set** *PART* **to 5 to get your rate sampling images in Part 5**.