High Performance Programming and Systems 2025

Exam Problem

Synopsis: Optimising and parallelising a ray tracer, including storing scenes as files.

Preamble

This document consists of 13 pages including this preamble; make sure you have them all. Your solution is expected to consist of a *short* report in PDF format, as well as a `.zip` or `.tar.gz` archive containing your source code, such that it can be compiled immediately with `make` (i.e. include the non-modified parts of the handout as well). The report must have the specific structure outlined in section 7.

Make sure to read the entire text before starting your work. There are useful hints at the end. You should also carefully read the code handout, including the header files. They contain normative instructions on what un-implemented functions are supposed to do.

The following rules apply to the exam.

- The exam is *strictly individual*. You are not allowed to communicate with others about the exam in any way.
- Do not share or discuss your solution with anyone else until the exam is finished. Note that some students have received a timeline extension. Do not discuss the exam until an announcement has been made on Absalon that the exam is over.
- Posting your solution publicly, including in public Git repositories, is in violation of university rules on plagiarism.
- If you believe there is an error or inconsistency in the exam text, *contact one of the teachers*. If you are right, we will announce a correction.

- The specification may have some intentional ambiguities in order for you to demonstrate your problem solving skills. These are not errors. Your report should state how you resolved them.
- Your solution to the exam problem is evaluated holistically. You are graded based on how well you demonstrate your mastery of the learning goals stated in the course description, including your ability to write correct C programs with appropriate error detection. You are also evaluated based on the elegance and style of your solution, including compiler warnings, inconsistent indentation, whether you include unnecessary code or files in your submission, and so on.

1 Introduction

For this exam you are given a fully operational ray tracer similar to the one presented in the week 7 exercises. You will be asked to parallelise the ray tracer, implement an accelerated variant, and implement additional programs related to its operation, as well as to analyse and measure the performance of the implemented programs. You are not required to understand the mathematics underlying the computer graphics, nor is it particularly useful. Our focus is on machine-level issues.

2 Bounding Volume Hierarchies

The main bottleneck in ray tracing is finding collisions between rays and objects in the scene. In a naive ray tracer, this is done by iterating through *all* objects and picking the closest hit. To speed this up, all practical ray tracers use a spatial tree structure that subdivides the space and makes it faster to find collisions by checking bounding boxes of the subdivisions, very similarly to *k*-d trees and octrees. The structure you will implement is a *bounding volume hierarchy* (BVH).

2.1 Structure

A BVH is a tree. Each node stores the following information:

1. An axis-aligned bounding box (*AABB*) describing a region of space. This bounding box must enclose the bounding boxes of all children of this node.
2. *One of* the following:
   a) An object.
   b) Two child nodes, called the *left* and *right* child.

An illustration is shown in fig. 1.

2.2 Determining hits

To find a hit between a ray and a BVH, we do as follows.

1. Check if the ray intersects the AABB of the node. If it does not, then there is no hit.

2. If the AABB contains an object, then check for a hit between the ray and the object as usual, with `object_hit()`.

3. Otherwise, if the AABB contains child nodes, then recursively check for a hit in both the left and right child. If a hit is found in both, then pick the closest hit.

The big advantage of the BVH is due to step 1, as it allows us to ignore large swathes of the space.

2.3 Construction

A BVH is constructed from a non-empty sequence of objects. Just as with k-d trees, there are many *valid* ways to construct a BVH. Good performance depends on the tree being reasonably balanced. The following algorithm works decently, and is very similar to the one for k-d tree construction you are familiar with from the assignments. You are not expected to come up with a better algorithm.

1. If the sequence contains only a single object, then construct a BVH node that contains that object and its AABB, computed with `object_aabb()`. Then return that node.

2. Otherwise, sort the objects by their minimum position along an axis (x, y, or z). Make sure you do not always use the same axis.

3. Split the sorted sequence of objects in half along the median element.

4. Construct child nodes for the two parts of the sequence. Construct and return a new node that has these two as its children, and whose AABB is their closing AABB (`aabb_enclosing()`).

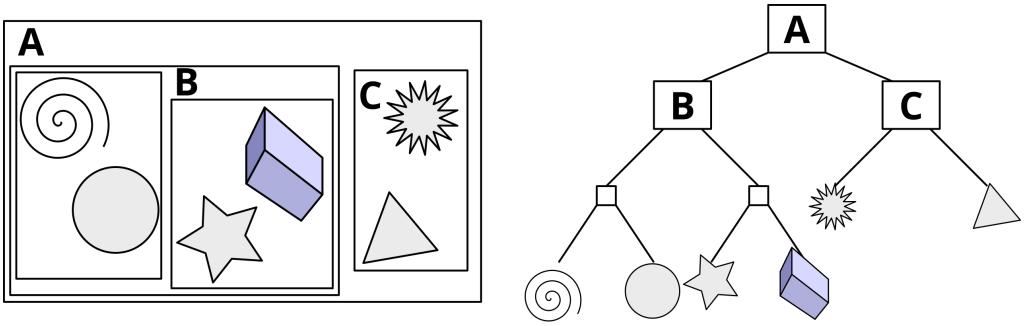

Figure 1: Bounding Volume Hierarchy in two dimensions. The left shows the geometric interpretation of a BVH, where the AABB if each cell encloses its children—in this case the AABBs of cells do not overlap, but in general they may. The right shows the tree structure of the BVH. By Schreiberx - Own work, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=17853864`.

3 Scene files

The ray tracer as given constructs the scene to be rendered in memory. In practice, real ray tracers read scene descriptions from files, where they may have been created by entirely different programs (such as 3D editors). As part of this exam you will implement storing and loading scenes from files. You are given a programming interface that you must implement (`scene_io.h`), but the details of the implementation (`scene_io.c`) are entirely up to you. In particular, you must design and implement the file format yourself.

The following function stores to a file a set of objects, the materials they reference, and the camera position and orientation:

```
int store_scene(char* fname,
                size_t num_objects, struct object *objects,
                struct vec lookfrom, struct vec lookat);
```

The `fname` is the name of the file to create. The function returns `0` on success and nonzero otherwise. *You* must decide on an appropriate representation in the file. The use of this function is demonstrated by the program `store_scene` (see section 4).

Apart from storing to a file, you must also be able to load the scene from a file. The programming interface for this is slightly more intricate.

A scene loaded from a file is represented by an opaque struct type:

```
struct loaded_scene;
```

A scene can be loaded from a file produced by `store_scene()` with the following function:

```
struct loaded_scene* load_scene(const char *);
```

Storing and loading a scene must result in a scene that is *equivalent* to the original scene, but it need not be *identical*. For example, the objects may appear in a different order in memory.

Since `struct loaded_scene` is opaque, users can only interact with it through the following functions:

```
// Retrieve the number of objects in a loaded scene.
size_t scene_get_num_objects(struct loaded_scene*);

// Retrieve the objects corresponding to a loaded scene.
struct object* scene_get_objects(struct loaded_scene*);

// Retrieve the camera position from a loaded scene.
struct vec* scene_get_lookfrom(struct loaded_scene*);

// Retrieve the camera target from a loaded scene.
struct vec* scene_get_lookat(struct loaded_scene*);

// Deallocate any memory referenced by a loaded scene.
void free_loaded_scene(struct loaded_scene*);
```

See the program `describe_scene` for an example of how they are used.

4 The programs

This section documents the behaviour of the programs included in the handout. Some of the programs will only be fully functional after one or more tasks have been completed. See also the appendices.

4.1 ray

A brute-force ray tracer that does *not* use a BVH. Initially sequential, but you are asked to parallelise it. Used as follows:

```
$ ./ray outfile nx ny ns scene
```

Here `outfile` is the name of the output image, `nx` is the width of the image, `ny` is the height of the image, `ns` is the number of samples, and `scene` is the name of the desired scene (`empty`, `irreg`, `nice`, `rgbbox`). If `scene` is left out, it defaults to `nice`.

Once Task B has been implemented, `scene` may also be the name of a scene file, as produced by `store_scene`.

4.2 ray_bvh

Used in the same way as `ray`, but is presumably faster.

4.3 store_scene

Stores a named scene in a file. Used as

```
$ ./store_scene outfile scene
```

where `scene` must be one of the known named scenes (`empty`, `irreg`, `nice`, `rgbbox`).

4.4 describe_scene

Describes the objects stored in a scene file. Used as

```
$ ./describe_scene outfile
```

where `outfile` is a file produced by `store_scene`.

5 Tasks

Read the comments in the code handout for details on how to implement the functions specified in the tasks below. None of the tasks depend on each other, and you do not have to implement them in the order given below.

5.1 Task A: Parallelise `ray.c`

The `ray.c` program in the handout is sequential. Parallelise it with OpenMP. Make any modifications necessary in order for the parallelisation to be correct and efficient.

Hints: The parallelised program does not need to produce the *exact* same result as the original program.

5.2 Task B: Implement loading and storing of scenes

In `scene_io.h`, come up with a definition for `struct loaded_scene` and implement all functions. Defining the representation is your job. You are evaluated on the correctness and efficiency of your chosen representation.

Hints: If you find this task too overwhelming, implement a limited variant that assumes all objects in the scene are spheres and have the same lambertian material.

5.3 Task C: Finish implementation of `ray_bvh.c`

In `ray_bvh.c`, finish the implementation of the BVH. Parallelise the program to the best of your ability. Once implemented, `ray_bvh` is used identically to `ray`.

6 Code handout

You are allowed to modify the following files in the code handout. Do not modify any files not listed below. You are allowed to add new files, although it is not necessary in order to solve the exam. Do not remove the timing code from `ray.c` and `ray_bvh.c` or change its output format.

- `ray.c`
- `ray_bvh.c`
- `scene_io.c`
- `Makefile` (modifying this file is not required to solve any of the tasks, and you should not solve the tasks in a way that requires modification of this file).

7 Your Report

Your report should be no more than ten pages in length and must be structured exactly as follows:

Introduction: *Briefly* mention very general concerns, any ambiguities in the exam text and how you resolved them, and your own estimation of the quality of your solution. *Briefly* mention whether your solution is functional, and on what you base this assessment (e.g. testing). Do not include any information also covered by the specific questions below.

A section answering the following numbered questions:

1. The following questions pertain to Task A.
   a) Which modifications did you need to perform to `ray.c` in order to parallelise it, and did they change the behaviour of the program?
   b) Measure and show the weak and strong scaling of `ray` as you vary the number of threads. Explain how you chose the workloads (scene, resolution, sample count).
   c) Does the program exhibit good spatial locality?
   d) Describe a scene and an otherwise sensible way of parallelising the rendering loop with OpenMP that might not take good advantage of all available CPU cores, as well as how this can possibly be solved.
2. The following questions pertain to Task B.
   a) Describe the layout (preferably with a diagram) of how your implementation stores the `nice` scene in a file.
   b) Does your implementation preserve the original ordering of the objects and materials?
3. The following questions pertain to Task C.
   a) Measure and show the weak and strong scaling of `ray_bvh` as you vary the number of threads. Explain how you chose the workloads.
   b) Does the program exhibit good spatial locality, especially when compared with `ray`?

c) Construct a workload where `ray_bvh` performs better than `ray` and one where it performs worse, and explain why.
d) Why is parallelisation of the BVH construction more complicated than parallelising the rendering loop, and did you parallelise it in your implementation?
e) Describe why or why not parallelisation of the BVH construction might or might not be worth it.

Your answers must be as precise and concrete as possible, with reference to specific programming techniques and/or code snippets when applicable. All else being equal, **a short report is a good report**.

A Examples

1 Using data files

The following examples here will work once Task A and Task B have been implemented.

The following command stores a scene in a file.

```
$ ./store_scene nice.data nice
```

We can then ask for a description of the contents of the `nice.data` file. The precise output, such as ordering of objects, may depend on how you solve Task 2.

```
$ ./describe_scene nice.data
sphere
  centre=(4.000000,0.000000,-6.000000)
  material=lambertian(albedo=(0.800000,0.800000,0.800000))
xy_rectangle
  x0=-4.000000
  x1=4.000000
  y0=-5.000000
  y1=0.000000
  k=-5.000000
  material=metal(albedo=(0.000000,0.000000,1.000000), fuzz=0.900000)
yz_rectangle
  y0=-5.000000
  y1=0.000000
  z0=-5.000000
  z1=5.000000
  k=-4.000000
  material=metal(albedo=(1.000000,0.000000,0.000000), fuzz=0.900000)
yz_rectangle
  y0=-5.000000
  y1=0.000000
  z0=-5.000000
  z1=5.000000
  k=4.000000
```

```
  material=metal(albedo=(0.000000,1.000000,0.000000), fuzz=0.900000)
```

We can then render an image based on the data file.

```
$ time ./ray nice.ppm 500 500 10 nice.data
Reading scene from nice.data...
```

And if necessary, and we have ImageMagick[1] installed, we can then convert the PPM file to a PNG, which ought to be readable everywhere:

```
$ magick nice.ppm nice.png
```

[1] https://imagemagick.org