Advanced algorithms and programming methods 2

Year: 2020/2021

Assignment 3

3D Render Pipeline

January 2021

Marco Carfizzi 860149

Index

1 Introduction

The goal of this document is to describe the implementation for the third assignment of the course. Starting from the given code, the assignment has these main requests:

1. Parallelize the rendering pipeline.
2. Allow the creation of a user-specified number of worker-threads and distribute the load among them.

2 Analysis

The first issue I needed to address was to figure out on what level the parallelization should happen, in order to maximize throughput and minimize contention.

The candidate levels were fragment, scanline, triangle and object.

2.1 Choice of the level

What I basically did was a trial and error approach.

While the fragment and scanline solutions needed much work to ensure that the high number of shared resources weren't conflicting and so I decided to focus either on the triangle or the object levels.

The preferred solution was chosen by measuring the performance (in seconds) needed to render multiple times a scene composed by multiple objects.

Trying both the triangle and the object levels, the chosen solution was to apply the parallelization to the objects. The only contention happening was in the rasterizer render_scanline method, more in particular the problem was that the z_buffer or the target could be overwritten. More details will be given in the next section.

3 Implementation

The main idea is that inside *render* method of the *Scene* class, the program loads the single object render into a thread. The issue is that one scene could have more objects than threads, so this needs some form of synchronization.

There is also the need to manage the critical section of the *render_scanline* method with proper synchronization.

3.1 Scene

3.1.1 Render method

Inside the *Scene* class, I edited the *render* method to have a *std::vector* of threads called *v_threads*.

For each object, a new thread will be added to *v_threads* and this will call for the synchronizer to manage the situation. If there is no space in the thread pool, the addition of new threads is halted until another one has ended its execution.

When the threads are loaded, they will be joined and executed.

3.2 Object

3.2.1 Render method

The nested class *Object* has the wrapper for the *pimpl* of the *render* method and that's the one called by the scene.

I edited this method to call the synchronizer when an object has been rendered, so that a new thread can be freed and this is notified to the next one.

3.3 Rasterizer

As I already mentioned earlier, the changes made in the Scene/Object classes are not sufficient: the pimpl implementation calls for the *render_vertices* method from the rasterizer, which will call the *render_scanline* method. This last method will attempt the *z_buffer* test and eventually edit the z_buffer and the target values. If two or more threads enter at the same time in this section, the data might become inconsistent and print the wrong result.

One more note is that a *Synchronizer* instance is present, so that it's easier to call for the synchronization when the rasterizer is passed to the scene and to ease the setting of the threads number.

3.3.1 Variables

I added the following variables to the rasterizer:

- ◇ `std::mutex mtx_`

  a mutex used to manage the shared critical resources of the z_buffer and the target.

- ◇ `Synchronizer synchro`

  a *Synchronizer* instance.

3.3.2 Render_scanline method

To lock the critical area I used a simple standard mutex to ensure that the z_buffer test codelines are accessed in a mutually exclusive way by the threads:

```
std::unique_lock<std::mutex> lock(mtx_);
```

3.4 Synchronizer

I decided to implement the *Synchronizer* as a nested class inside the *Rasterizer* to have a coherent design.

3.4.1 Variables

```
        std::condition_variable cv_;
unsigned short int nThreads_ = std::thread::hardware_concurrency();
unsigned short int usedThreads_ = 0;
bool canAdd = true;
std::mutex mtx_;
```

The mutex and the *condition_ variable* are needed to implement the synchronization required for not having more active threads than supported.

The *canAdd* flag is used to check whether it is possible to add new threads or not, based on the fact that the number of used threads *usedThreads_* must be lower than the total number of setted threads *nThreads_* .

3.4.2 Methods

Beside the usual constructors, I implemented three methods:

- ◇ *inline void setNThreads(unsigned short int nThreads)*: Set thread number and returns a warning if high number of thread is given.
- ◇ *inline void adder()*: Set the *unique_ lock* for the mutex to wait that the *canAdd* flag is set to *true*. If it is possible, increase the number of threads in the pool and update the flag accordingly.
- ◇ *inline void remover()*: The number of threads in the pool is decreased, the flag is updated and a *notify* signal is sent to the next thread that needs to go into execution.

4 How to test

4.1 How to setup the main

If the user wants to set the number of used threads he can call the *set_ n_ threads* from the rasterizer. If no number is set, the program will default to the maximum number of hardware threads. To retrieve the actual number of threads the user can call the *get_ n_ threads* from the rasterizer.

To make some measurement of the performance of the pipeline, I implemented a for loop to load multiple objects into the scene. I additionally added a for loop to render multiple times the same scene. The values of objects and renders to perform can be set in the two global constants in the *main.cpp* file, right before the *main*.

4.2 How to build

I provided the project with a MakeFile, so all is needed is to open a terminal in the correct path and write

```
make
```

to produce a *main* executable file.

4.3 How to run

Use

```
./main
```

to start the program.

The resulting output is the last computed render, with an information about the seconds needed to perform the computations (this is done just for debug).