Università
Ca'Foscari
Venezia

3D Software Render (*CM0470-2*)

Academic Year 2020-2021

Campus
Scientifico
Via Torino 155
Venezia 30172

–

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Assignment 3:
3D Pipeline Parallelized

Student

Camoli Filippo
871380

Assignment Description

Parallelize the rendering pipeline.

Decide on what level to introduce **concurrency** (fragment, scanline, triangle, object) in order to maximize throughput (and **minimize contention**). And explain *reasons* and *synchronization requirements*.

Have your library create a *user-specified number of worker-threads* and *distribute the load among them.*

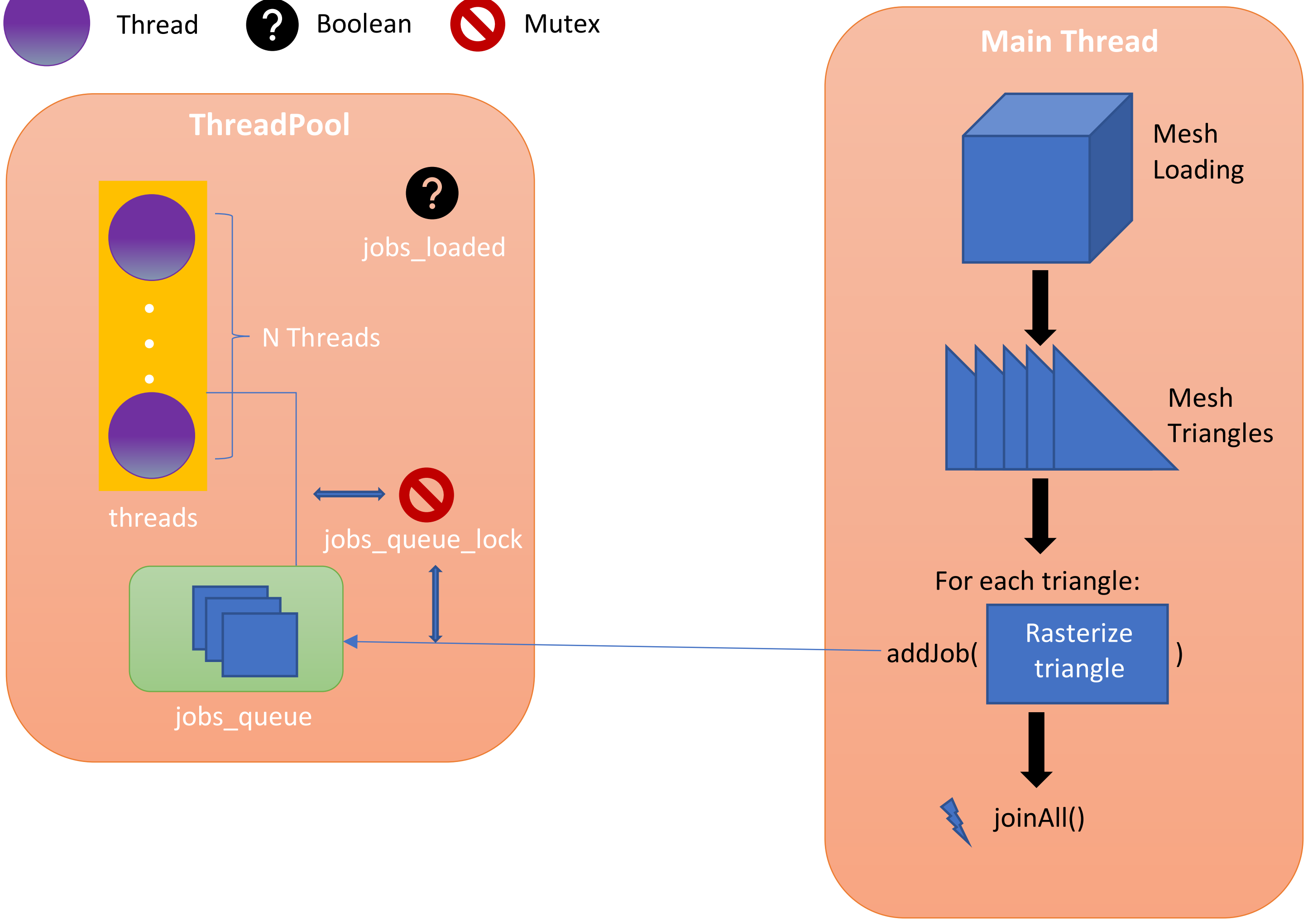
Solution: Parallelization per **Triangle**

Following solution is based on 2nd Assignment solution given by Teacher. In order to achieve concurrency with both following conditions:

- *User-specified number of worker-threads*
- *Distribute the load among them*

I take in account the usage of a Thread Pool that in my solution can be easy reduced in this diagram. All objects' names used in this diagram are 1:1 representation on what's written in my code.
Objects Legend:

🟦	A single job	▨	Multiple jobs	🟩	Queue (FIFO)	🟨	Vector
●	Thread	?	Boolean	🚫	Mutex		



Parallelization is developed in **triangle level**.
A ThreadPool object is instantiated into main function with argument the user specified number of thread which by default is given by `std::thread::hardware_concurrency()`:

```
ThreadPool tp(#threads);
```

At this point when ThreadPool class is instantiated, its constructor spawn *nThreads* defined by its only one parameter and set the boolean `jobs_loaded = false` . This boolean is used to giving a stop-thread-running condition to achieve the join of all threads (see later for more explanation).
When each thread is spawned, each reference is saved in a vector (of threads), used at the end of the render (i.e: in the main function in order to waiting render completion and printout the render result) when we want to join all threads by calling the `joinAll` function:

```
void joinAll(){
     for(std::thread &t : threads)
            t.join();
}
```

Each spawned thread run the following code. A while true statement is necessary due to the fact that we need to re-utilize each thread when it ends its job.
Two phases are present here

1- *Getting a job from the queue* (if any and if there's still work to do)
   Here we use a `unique_lock` on `jobs_queue_lock` mutex. This synchronization is necessary because we're operating with multiple threads (included the main one which insert the jobs, explained later) that are reading and writing in a shared (fifo) queue. So, after creating a function pointer initialized to null, we use a block statement where we're activating the lock and each thread waits until there's *something to do.* Every thread can continue its execution only if some conditions can allow that, or someone do a notification. These conditions says that we can try to get a job if the queue isn't empty or if it's empty but there's nothing to do (`jobs_loaded`) then we can execute next instructions which filtrates the two cases.
   If there's *nothing to do (empty queue and jobs are loaded)* then we exit from our infinite loop, otherwise we extract the next job from the queue.

2- *Executing the job*
   Once we exit from the block-lock-statement we can execute the new assigned job and notify to another thread that we exit from the strict zone.

```
{
     while(true){
            std::unique_ptr<std::function<void()>> scheduledJob { nullptr };
            {
                  std::unique_lock<std::mutex> lock(jobs_queue_lock);
                  jobs_condition.wait(lock, [this]() {
                        if(!jobs_queue.empty() || (jobs_loaded && jobs_queue.empty()))
                                    return true;
                              else
                                    return false;
                  });
                  if(jobs_loaded && jobs_queue.empty())
                        break;
                  scheduledJob = std::move(jobs_queue.front());
                  jobs_queue.pop();
             }
             (*scheduledJob)();
     }
}
```

Then the tp object (instantiated in main function) is passed by reference into *scene.render(),* to make things work, a ThreadPool instance pointer is added to every functions involved in the *scene.render()* call-tree. For each object it's called its respective pointer implementation render function where for each triangle in the object (mesh) we add a new anonymous function that contains the elaborated vertices and the rasterize function call:

```
for(const auto& t : mesh_) { //for each triangle in the mesh
     tpIstance->addJob([this, &t, &world, &view, &rasterizer](){
             auto v1 = t[0];
            auto v2 = t[1];
            auto v3 = t[2];
            transform(world,v1);
            transform(view,v1);
            transform(world,v2);
            transform(view,v2);
            transform(world,v3);
            transform(view,v3);
            rasterizer.render_vertices(v1,v2,v3, shader_);
     });
}
```

After all objects (ie: triangles) are loaded into the job queue we can call `tpIstance->jobsLoaded()` which is useful when we need to stop worker-threads from looping infinitely even if all jobs are done:

```
void jobsLoaded(){
     {
             std::unique_lock<std::mutex> lock(jobs_queue_mutex);
             jobs_loaded = true;
     }
     jobs_condition.notify_all();
}
```

Synchronization is required by thread's waiting condition that uses also this boolean. Notify all is useful in cases when all are waiting to do something and the queue is empty.
At this point the job (basically the rasterizer call function) will be added into *jobs_queue* by addJob function:

```
void addJob(std::function<void()> j){

     std::unique_ptr<std::function<void()>> jobReference =
                                 std::make_unique<std::function<void()>>(std::move(j));
     {
             std::unique_lock<std::mutex> addJobLock(jobs_queue_lock);
             jobs_queue.push(std::move(jobReference));
     }
     jobs_condition.notify_one();
}
```

A `void function` j is passed by argument, then moved into `jobs_queue` (as unique-smart-pointer to avoid dangling pointers to functions) only after locking the same `jobs_queue_lock` mutex since the queue is used by threads to pop jobs. Once it's inserted, we notify **one** waiting thread since it's reasonable to limit only to one thread per time job schedule.

Synchronization per target access using a Mutex per pixel

This section briefly explain how synchronization is used into render scanline function since it's the only portion of code that uses global resources as z_buffer and target_t shared memory among threads.
The following solution is simple but a little bit expensive in creating a mutex matrix as showed inside the `set_target` function called by Main thread before loading objects into the scene (and start polling threads):

```
std::vector<std::vector<std::mutex>> temp(height);
for(int i = 0; i < height; i++){
     temp[i] = std::vector<std::mutex>(width);
}
pixelMutex.swap(temp);
```

Following code is a snippet/partial code from `render_scanline` function since it's almost the same.

```
for (; x != std::min(width,xr+1); ++x) {
     const float ndcz = interpolatef(ndczl,ndczr,w);
     Vertex p = interpolate(vl,vr,w);
     perspective_correct(p);
     pixelMutex[y][x].lock();
     if ((z_buffer[y*width+x]+epsilon) >= ndcz){
            z_buffer[y*width+x] = ndcz;
            pixelMutex[y][x].unlock();
            target[y*width+x] = shader(p);
            w -= step;
     } else
            pixelMutex[y][x].unlock();
}
```

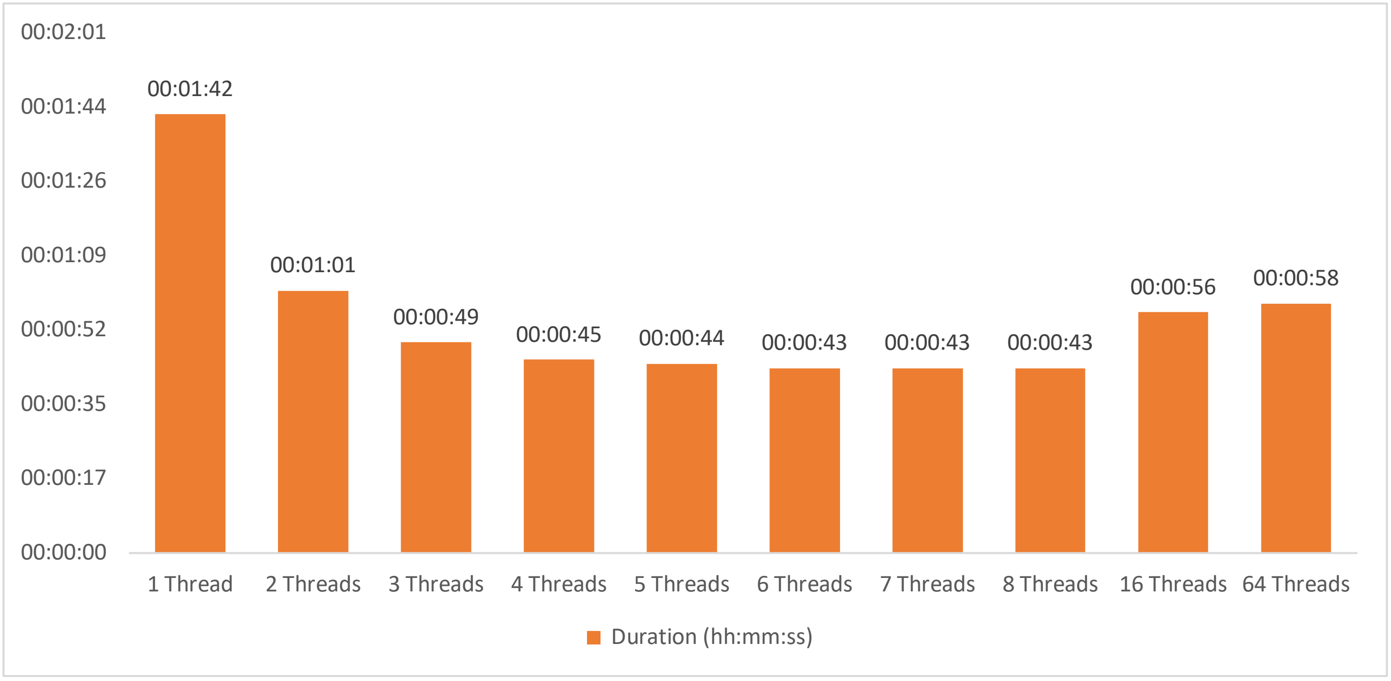
In order to minimize contention and avoid variable w updating erroneously, some instruction as interpolating and correcting vertices calls are done before entering the contention zone. This is because we need to check some info into z_buffer but we need the guarantees that the z_buffer value in the *y,x* position will be consistent when we want to updated it into z_buffer. Lock and unlock are performed in order to atomically perform involved z_buffer's operations since conditional wait variables complicates code complexity. Else-branch ensure that mutex is unlocked when we skip a certain pixel shading.

Performance's comparison

In this section we'll see how multi-threading is useful in this kind of stress-intensive rasterization benchmark with 1 Million of cubes. For laziness I've used the same cube at the same position in the scene. In order to achieve a true experience each object is passed by copy to *scene.addObject(…)* function since it's the only way to look like 1Mln different cubes. For sure we're going to see only one cube in the scene/render.
Note that the Multi-threading optimization is on Triangle level but Object loading it's performed by *main thread*. This is important to notice because in the first phase of the run the main thread is loading objects (1 cube has 12 triangles so they are ~12Mln jobs effectively) and putting them into the queue, meanwhile the *spawned threads of the Thread Pool* are going to start their lifecycle during this phase so, in the initial phase, it looks like n+1 threads concur, but we only care and look only threads in the render phase.
Measurements are made using Xcode's performance analysis, for this reason results will be altered since this program will trace and record system's component usage during the render execution. Due to HW limitations, meaningful tests will be up to 4 (physical) threads, in 5+ threads' tests might get performance degradation due to expensive mutex concurrency and mid-high contention.
HW-SW Configuration test: *MacOS 11, Xcode 12, Intel i5-2500s (2nd gen.) @ 2.7GHz, 4 Physical Threads.*

Time Comparison

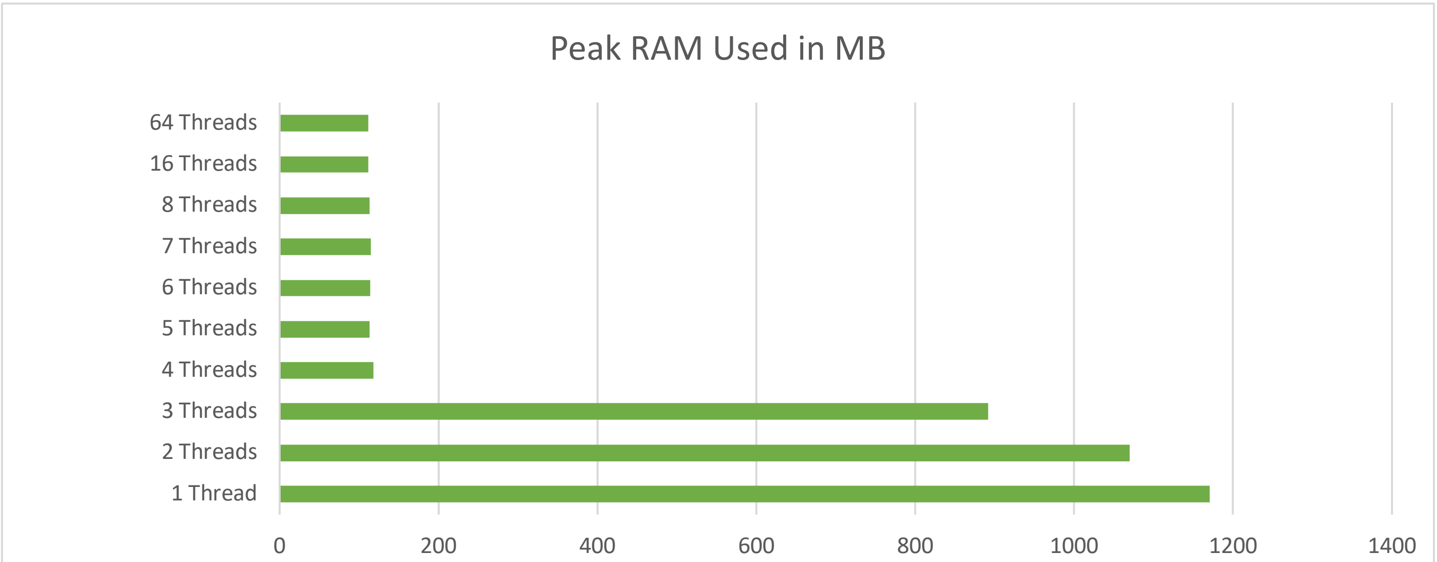
This graphs show the entire test duration *in hours : minutes : seconds* in respective thread configuration.



As we can see when we're *using n+1* up to *2*n* physical threads, results are quite the same, maybe due to the Intel's Logical Threads solution. With 16 and 64 threads we can clearly see how performance degradation is consistent due to synchrinization costs but still better than 1-2 Threads configuration tests.

RAM Usage (Peak) Comparison

Total ram allocated for this benchmark is ≈ 1,32GB.
During the running test the peak ram usage is shown in the following graph:



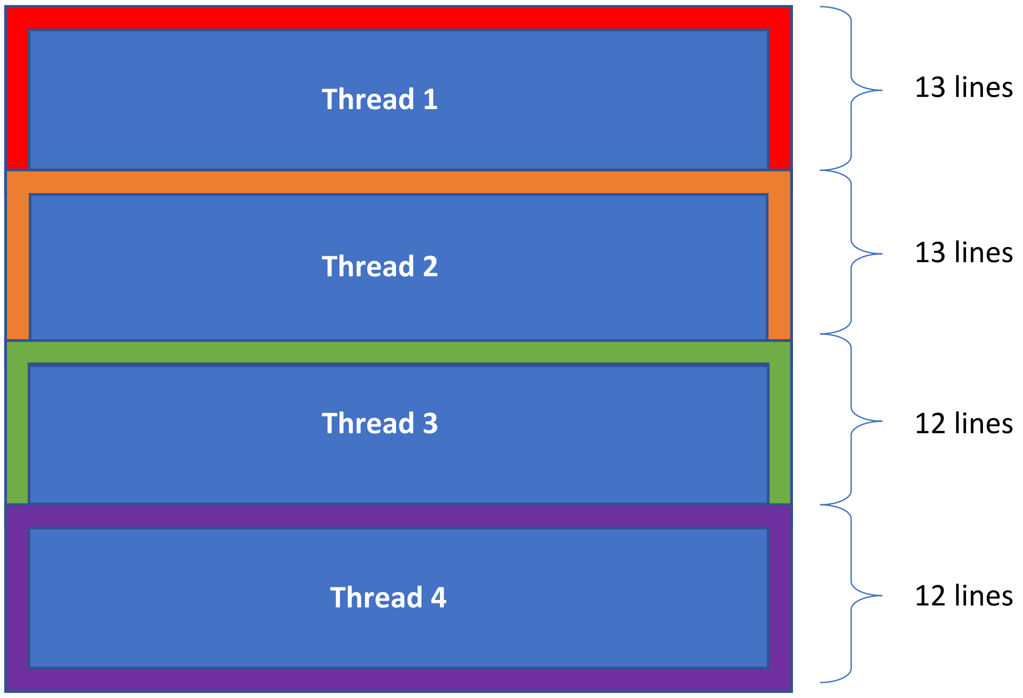
	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	16 Threads	64 Threads
🟩 Peak RAM Used in MB	1170,47	1070,11	891,4	117,7	113,5	114	114,9	113	111,5	111,8

These results show us how multi-threading can help us also maintaining low ram usage over running time. As we can see, when we choose more threads than the physical ones supported by our hardware, it's clear there's some stationarity.

Final considerations

When we want to handle with concurrency synchronization it's required. But there's a way to optimize the rasterization and the render in such a way that we don't even have to do synchronization with mutex locks also involving a Thread Pool solution.
The idea is to re-write the rasterization procedure. We first have to split the target (same as z_buffer) into n threads portion lines. Each thread will manage a line per cycle and draw the respective pixel based on triangle's information. Each thread will operate in their zone and they will read at the same time each triangle to rasterize so no writing conflicts with shared buffers.

Example with n = 4 threads, target's height = 50:



A pseudo code is given to understand the basic idea:

```
for (int j = 0, i = 0; j < nThreads && i < height; ++j, i += height/nThreads){
     threadsVector[j].do_work([](){
          renderLines(i, i + nThread);
     });
}
```

where *renderLines* is something like this:

```
void renderLines(int start, int end) {
      for (int y = start; y < end; y++) // for each row of its portion
            for (int x = 0; x < width; x++) // for each column
                  for (Triangle t : triangles) // for each triangle in the scene
                        if (insideTriangle(x, y, t)) // if x,y screen pixel is into t
                              draw (x, y);            // draw it
}
```