INFO9012-1: Parallel Programming
Project: Raycasting

March 14, 2025

Introduction. The project is personal or by group of two. The objective of the project is to get a practical knowledge of parallel programming on a small example.

The goal is to modify a raycasting program to make use of parallelism, in particular with multicore computers. A sequential version of the program is provided, with the source, on the eCampus space of the course. In principle, you do not have to bother with raycasting specific details. All language and GUI library aspects (e.g., interacting with the keyboard, rendering) are already demonstrated in the provided sequential program. According to the pedagogical commitment (*engagement pédagogique*) of the course, you must have access to a computer under Linux with X11. The code is library-independent enough to run on a variety of systems with minimal effort. A Raspberry Pi 3 with a Debian-based distribution is sufficient to run and experiment with the software.[1] Virtualization is another possibility, but the performance gain from parallelism might be unexpected and surprising. Alternatively, several ms8xx computers are available in the Montefiore building, as well as remotely using ssh. Again, thanks to the fact that the software only uses the X11 library, remote ssh execution will work with minimal effort.

It is not asked to improve the running times by modifying the raycasting algorithm.[2] The work essentially focuses on parallelism aspects only.

Please do not change the indentation of the existing code, adopt our coding indentation. You might like your indentation better, but we do have many programs to read, and adapting to each of your indentation preferences takes time.

A brief explanation of raycasting. Created in the late 1980s, raycasting is a fast pseudo-3D rendering technique that uses a 2D map (represented by a 2D array).

For each horizontal screen pixel, a ray is cast from the player until it hits a wall (represented by a value greater than 0 in the 2D array). For a screen resolution of 1920×1080, this means that 1920 rays are cast. Once the ray hits a wall, a vertical line is drawn on the screen at the corresponding horizontal coordinate. The height of the vertical line is determined by the distance between the player and the wall, the closer the wall, the taller the line.

An example of a single ray being cast is shown in Figure 1. A complete rendering example is shown in Figure 2.

[1] At the current time, it is possible to build a full working RP3-based computer for less than 50 €, to which you would just need to add a keyboard/screen/mouse to have a full working environment.

[2] Although there are numerous opportunities to do so, please do not waste time on improving the computation itself, since we will not evaluate that in any way.

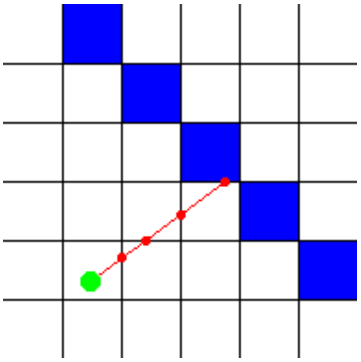

Figure 1: Demonstration of a single ray being cast, where the green disc is the player, the red line is the ray, the white cells are empty places, and the blue cells are walls. (Image from Lode Vandevenn)

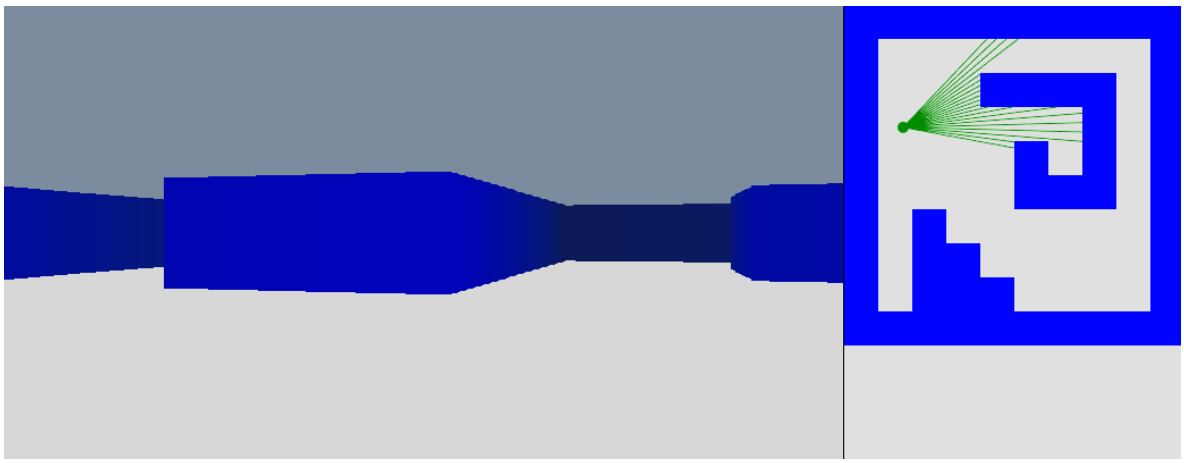

Figure 2: The final raycasting rendering is shown on the left. The 2D representation (with the rays cast) is shown on the right. An interactive live demo is available at `https://huth.me/raycast`.

The provided code. The provided code is in C++. To compile it on Linux, you need `g++`, and `xlib` (e.g. provided by the Debian package libx11-dev); a `Makefile` is provided, so in principle, you just need to type `make` to compile. The program works on the ms8xx network, and it is possible to run remotely, even with the graphical interface (you will have to configure any X11 server on your machine, and use X11 forwarding with ssh). It might be possible to run using Windows Subsystem for Linux (WSL), but we do not guarantee support for that.

Run the program and interact with it The program must be run with three arguments, the width and height of the window, and a file path providing IPs and ports to play with other players, e.g.

```
./raycasting 800 600 ips1.txt
```

The following keys can be used to interact with the program:

- the upper and lower arrow keys are used to move forward and backward;
- the left and right arrow keys are used to turn left and right;
- the escape key is used to exit the program.

The IPs file provided must be formatted as follows:

```
LISTENING_PORT
IP1 PORT1
IP2 PORT2
...
```

where the first line contains the port on which your program will receive the positions of the other players. The subsequent lines contain the IPs and the ports to which your position will be sent. Two example files are provided (see the *ips1.txt* and *ips2.txt* files).

As an example, you can run 2 instances of this program that communicate together on the same computer by having a first file called *ips1.txt* that contains:

```
12345
127.0.0.1 12346
```

and a second one called *ips2.txt* that contains:

```
12346
127.0.0.1 12345
```

You can then run the instances by typing (each line in its own terminal):

```
./raycasting 800 600 ips1.txt
./raycasting 800 600 ips2.txt
```

This example is visually shown in Figure 3.

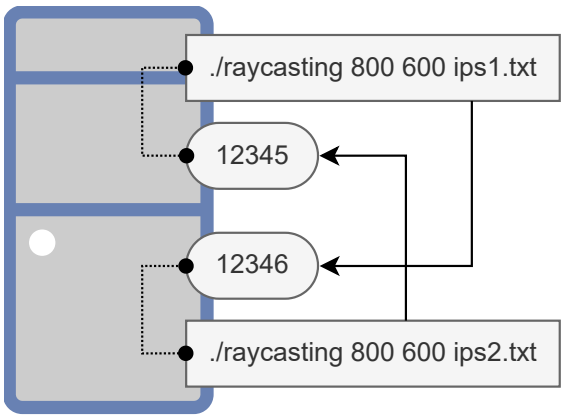

Figure 3: Visual representation of what happens when starting the program twice, once with *ips1.txt*, and once with *ips2.txt*.

Objectives of the project. The sequential program is usable and the interaction is working. However, no use is made of the multicore architecture of modern computers, resulting in a lower frame count per second (especially at higher resolution).

The objectives of the project are to learn how to

1. use the available cores on the computer in order to increase the amount of computation per unit of time, i.e., to increase the number of frames per second (fps);

2. understand that the structure of the program has to be conceived with parallelism in mind, e.g., functions should be thread safe;

Your mission. The base project you are given contains five folders: one named sequential and four folders named 1, 2, 3, and 4 (one folder for each step detailed below).

The sequential folder contains a sequential version of the program and is only there for you to keep a copy of the base sequential program so that you can easily restart from a clean program during one of the steps (if you need to). You are not asked to modify it, and it will not be evaluated.

The other folders (1 to 4) are exact copies of the sequential folder, which means that you will start each step from the base sequential program. You are free to modify any file for each step; however, you should be able to implement all solutions by only modifying the main.cpp, Raycaster, DoubleBuffer, WindowManager, and UDPReceiver files (.cpp and .h files included and all steps included).

When creating threads, mutexes, etc., you are allowed to use the `C`-style functions to create them (pthread_create, etc.), but it is recommended to use the `C++` tools for that[3]. When creating a thread, do not forget to gracefully exit it at the end of the program[4].

With this information in mind, you are asked to implement the following four tasks:

[3]See `std::thread` (usage example), `std::mutex`, `std::atomic`.

[4]When pressing the escape key, all created threads should exit before the main thread. If you use `std::thread`, see `std::thread::join`.

1. Use OpenMP to improve the frame count per second. The computations related to the floor, ceiling, wall, and sprite casting are the heaviest in this program. Use OpenMP to parallelize each of them. This should be a few lines change in the code. It might take time for you to locate where to operate the changes, but if it needs more than a few lines, you are doing it wrong. When done correctly, you should see an increase in fps, especially at higher resolution.

2. In the sequential program, the image is first generated in a double buffer, then, using X11, it is displayed, and the keys pressed by the user are retrieved. Create a new thread to manage the display and keys so that the logic of the game and the interactions with the windowing system are separated.

3. Create a thread to parallelize the reception of the positions of the other players. To avoid busy waiting when no position is received, modify `UDPReceiver.cpp` on line 35 by replacing `MSG_DONTWAIT` by `MSG_WAITALL`. With this change, the socket becomes blocking and times out after 100 milliseconds (as specified on line 15) instead of returning immediately if it does not receive anything.

4. Create a thread to parallelize the position sending to the other players. To avoid sending the position unnecessarily, send it only when the player has moved. In order to achieve this, use a condition variable[5] that is notified by the main thread when the player has moved.

It is important to note that the tasks 2, 3, and 4 might not bring significant improvement in performance (or even none at all). Do not rely on this criterion alone to verify the correctness of your solutions. Remember to apply good practices that you have learned in the course. Also, feel free to use tools such as Helgrind to help you detect data races (note that it can detect false positives, especially with OpenMP).

Deliverables. Make sure your submitted work complies with the following instructions. **Otherwise, the grade will be impacted.**

1. If done in a group of two, only one of you must submit the work.

2. The folder containing the project must contain four folders named 1, 2, 3, and 4. You can leave or delete the sequential folder if you wish, it will be ignored anyway.

3. The folder must be named according to your IDs. For example, if your ID is `s20221234` and your partner's ID is `s20224321`, the folder should be named `s20221234_20224321`, and if you do the project on your own, it should be named `s20221234`.

4. Once your folder is named correctly, create a **zip** archive and name it the same way. Using the previous example, in the terminal, you can easily create this archive by typing:

```
zip -r s20221234_20224321.zip s20221234_20224321
```

5. Finally, upload this zip file on eCampus. Again, if done in a group of two, only one of you must submit it.

[5] See `std::condition_variable`.

Organization. The last sessions of the course are dedicated to help you tackle all tasks in this project. Please do not leave all the work for the last days, because it will increase the total amount of time you will have to invest in this project, and you will be alone to figure out the difficulties rather than benefiting from the help of the assistants.

Cores, hyperthreading, and all that. There are system dependent ways (i.e., it differs whether you are using Windows, Linux, or another system) to allocate a thread to a particular logical CPU. We do not ask you to use system dependent features, because they are obscure, and this would also make checking your code difficult. Leave the scheduling of the threads to the kernel, and use a macro to specify the number of cores the program should use.

Plagiarism. Cooperation (even between groups) is allowed, but we will have **no tolerance at all for plagiarism** (writing together, reusing/sharing code or text). Code available on the web that serves as a source of inspiration must also be properly referred in the code using comments. ChatGPT, Copilot, and the like are strictly prohibited.

Participation. Each student has to participate in the project, and do her/his fair share of work. Letting others in the group do all work will be considered as plagiarism.

Assignment. If something is unclear, if you find a mistake, or believe something must be better specified, notify us. This text is not perfect, we will improve it with your comments.

Important final remark. This project is not meant to be a huge amount of work, or to leave you alone to find, e.g., obscure options of OpenMP. Ask for help, come to the practical sessions on Fridays, use the Forum, request a Q/A zoom session, if needed.

Additional information Below are some links related to the algorithmic aspects of ray-casting. These are **not** mandatory and are only provided if you are curious about it.

- Lode Vandevenne's raycasting tutorial (from which the code is heavily inspired).
- Super Fast Ray Casting in Tiled Worlds using DDA. A clear explanation of Digital Differential Analyzer (DDA), the algorithm used to detect walls.
- Ray-Casting Tutorial For Game Development And Other Purposes by F. Permadi.