Polytech Sorbonne – EI5-SE

High Performance Computing

Project – Motion Detection

1 Introduction

In this project, you will work on a streaming application that detects and tracks moving objects from a video sequence. A working application, called "Motion", is given to you. You will have to speedup Motion thanks to the different levels of parallelism and optimization methods you learned in this class.

First you need to clone the repository of the Motion project:

```
git clone --recursive https://gitlab.lip6.fr/parallel-programming/motion-eise.git
```

The Motion project uses CMake in order to generate a Makefile: follow the README instructions to compile the code.

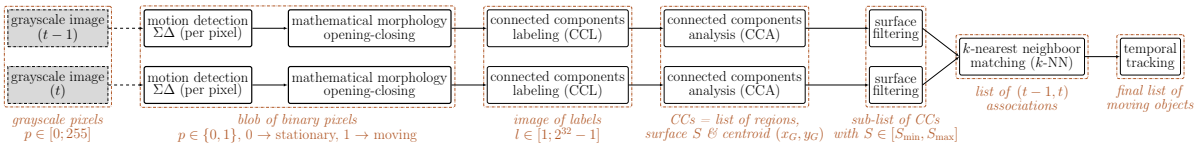

Figure 1: Motion detection and tracking processing tasks graph. In *italic*: the output of each processing.

Fig. 1 presents the different algorithms used to detect moving objects and to track them over time. To make it work, two strong assumptions are made: 1) the camera is fixed, 2) the light intensity is constant over time. First, an image is read from a camera (or a video sequence) and then it is converted in a grayscale image. Then, the $\Sigma\Delta$ algorithm is triggered. This algorithm is able to detect if a pixel is moving over time. It returns a binary image, if a pixel value is 0, then it means that it is not moving. Else, if a pixel value is 1, then it means that it is moving. After that, morphology algorithms are applied. This is a post-processing to regroup moving pixels together and eliminate isolated pixels. Then, from a binary image, a connected components labeling (CCL) algorithm is performed. The later, gives the same label to a group of pixels that are connected to each other. CCL returns an image of labels where $l = 0$ means no object and $l > 0$ means a moving object. From this image of labels, some components are extracted (CCA): for each object the center of mass (x_G, y_G), the bounding box $([x_{\min}, x_{\max}, y_{\min}, y_{\max}])$ and the surface S are extracted. Depending on their surface, the objects are filtered ($S_{\min} < S < S_{\max}$).

From two images at $t-1$ and t, a matching algorithm determines which objects are the same in the two different images (mainly according to their distance). At the end, the identified objects are tracked to have a constant identifier over time.

This graph of tasks is then repeated until the video sequence is over. It is not mandatory to understand perfectly each algorithm. The purpose of this project is to work on a streaming application, representative of a real application, and to perform optimizations on the **motion detection** and **morphology** tasks.

1.1 Run Motion

To run the code you will need some input videos. You can download a videos collection on Moodle (see the "Artifacts" section) or from this web link: `http://www.potionmagic.eu/~adrien/data/traffic.zip`. First, unzip the `traffic.zip` and from the `build` directory run the code with the following command:

```
./bin/motion2 --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
              --flt-s-min 2000 --knn-d 50 --trk-obj-min 5 --vid-out-play --vid-out-id
```

You should see a window with a top view of a highway and some moving cars (see Fig. 2) and you should see green bounding boxes around the cars.

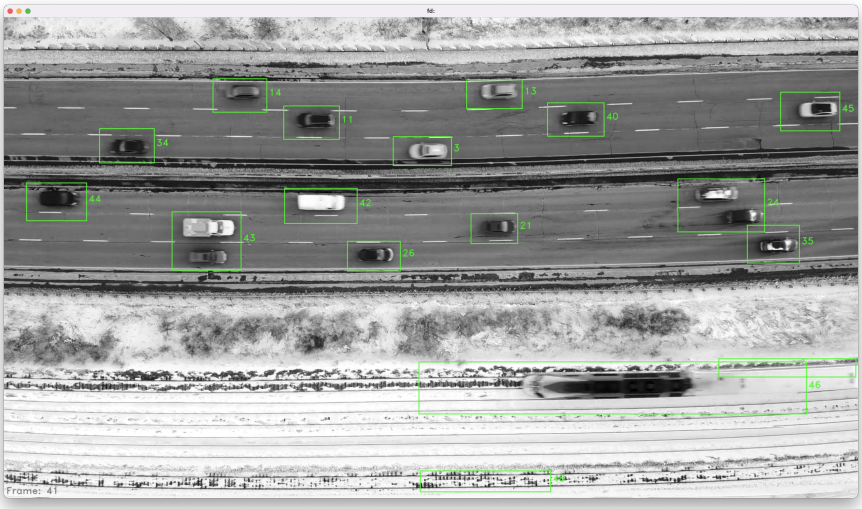

Figure 2: MOTION screenshot (with `--vid-out-play --vid-out-id` parameters).

1.2 Architecture of the Project

MOTION is mainly a C-style project but it is compiled in C++ to work with MIPP. The sources are located in the `src` folder, and there are 2 sub-folders:

- `common`: contains implementations of the processing tasks,
- `main`: contains source files that correspond to a final binary executable.

The headers are located in the `include` folder.

1.3 Motion Detection: Sigma-Delta Algorithm (ΣΔ)

The motion detection problem consists in separating moving and static areas in each frame. At each instant, each pixel must be tagged with a fixed/moving binary identifier. When the camera is fixed, such detection can be performed using the time differences computed for each pixel.

The following notations apply:

- t : current instant of time, used to identify the frames,
- I_t: grayscale source image at time t,
- I_{t-1}: grayscale source image at time $t-1$,
- M_t: background image (mean image),
- O_t: grayscale difference image,
- V_t: image of variance (standard deviation) computed for each pixel,
- L_t: binary label image (motion/background), $L_t(x) = \{0,1\}$ or $L_t(x) = \{0,255\}$ to encode {background, movement},
- x: the current pixel with (i,j) coordinates.

Most of motion detection techniques in an image sequence $I_t(x)$ are based on an estimate of the modulus of the temporal gradient $|\frac{\partial I}{\partial t}|$. If the light intensity of the scene vary slowly (= is constant between two consecutive images), then a significant variation in the pixel grayscale (above a threshold) between two images will imply that there is movement at that point.

The ΣΔ algorithm assumes that the noise level can vary at any point. To achieve this, the pixel grayscale is modeled by a mean $M_t(x)$ and a variance (standard deviation) $V_t(x)$. If the difference between the current image and the background image is greater than N times the standard deviation, then movement occurs. The value of N is a parameter. **In this project, N is always set to 2.**

This is a motion detection system based on the estimation of static background statistics using ΣΔ modulation: an iterative analog/digital conversion method that increments or decrements the digitized

Algorithm 1: Sigma-Delta ($\Sigma\Delta$).

[Part #1: mean computation]
foreach *pixel* x **do** // Step #1: M_t estimation
  if $M_{t-1}(x) < I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) + 1$
  if $M_{t-1}(x) > I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) - 1$
  otherwise do $M_t(x) \leftarrow M_{t-1}(x)$
[Part #2: difference computation]
foreach *pixel* x **do** // Step #2: O_t computation
  $O_t(x) = |M_t(x) - I_t(x)|$
foreach *pixel* x **do** // Step #3: V_t update and clamping
  if $V_{t-1}(x) < N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) + 1$
  if $V_{t-1}(x) > N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) - 1$
  otherwise do $V_t(x) \leftarrow V_{t-1}(x)$
  $V_t(x) \leftarrow \max(\min(V_t(x), V_{\max}), V_{\min})$
foreach *pixel* x **do** // Step #4: $\hat{L}_t$ estimation
  if $O_t(x) < V_t(x)$ **then** $\hat{L}_t(x) \leftarrow 0$
  else $\hat{L}_t(x) \leftarrow 1$

value by one unit according to the result of the comparison between the analog value and the current digitized value.

The algorithm initialization for $t = 0$ is the following: $M_0(x) \leftarrow I_0(x)$ and $V_0(x) \leftarrow V_{\min}$. Then, the algorithm is applied to the images from $t = 1$. The $V_{\min}$ and $V_{\max}$ constants are used to restrict the possible values of V_t. Typically, $V_{\min} = 1$ and $V_{\max} = 254$. The complete algorithm after initialization is shown in Alg. 1.

In the MOTION project, a naive $\Sigma\Delta$ implementation is given to you:

- Header: in the `include/c/motion/sigma_delta/sigma_delta_compute.h` file,
- Source: in the `src/common/sigma_delta/sigma_delta_compute.c` file.

See the `sigma_delta_compute` function.

1.4 Mathematical Morphology

In this project, we consider squared elements B of size 3×3. Let X be the set of pixels associated with the B element. There are two basic operations: the *dilation* of X noted $\delta_B(X)$ and the *erosion* of X noted $\epsilon_B(X)$. The application of mathematical morphology operators is similar to filtering operators (stencils or convolutions), but with non-linear operations.

For binary images, *dilation* consists in computing a `OR` on the B neighborhood in the source image and writing it to the destination image. Conversely, *erosion* consists in computing a `AND` on the neighborhood. So, if a point in the neighborhood is 1, the *dilation* produces a 1 (since `x OR 1 == 1`), thus dilating the binary connected component. Conversely, if only one pixel is 0 in the B neighborhood, the *erosion* will produce a 0 (since `x AND 0 == 0`), thus eroding the connected component.

Erosion is used to reduce noise in images: if we consider that a small group of pixels is the noise that we're trying to remove, then applying *erosion* with a B element of size 3×3 will make any group of pixels with a radius smaller than its size disappear.

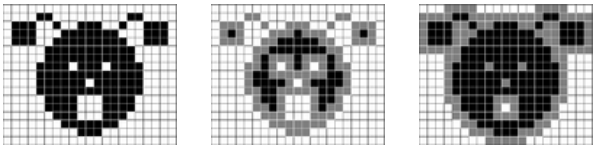

Figure 3: Left: the initial binary image. Center: *eroded* image with a 3×3 squared element: the gray pixels are removed. Right: *dilated* image with a 3×3 squared element: the gray pixels are added. Source: Wikipedia.

Let r be the radius and $d = 2r + 1$ the diameter of a squared element B, then an *erosion* of radius r removes, to any connected component, a thickness of r pixels of contour while a *dilation* of radius r adds

a thickness of r pixels to the contour (see Fig. 3, note that in the figure the logic is reversed: pixels at 1 are black while pixels at 0 are white).

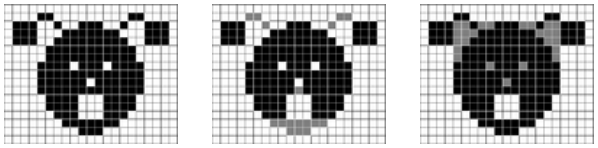

Figure 4: Left: the initial binary image. Center: *opened* image with a 3×3 squared element: the gray pixels are removed. Right: *closed* image with a 3×3 squared element: the gray pixels are added. Source: Wikipedia.

From these two operators, two others can be defined: the *closing* $\phi_B(X) = \epsilon_B(\delta_B(X))$ and the *opening* $\gamma_B(X) = \delta_B(\epsilon_B(X))$. *Closing* reduces (or even completely close) holes in connected components, while *opening* does the opposite, enlarging these same holes (see Fig. 4, note that in the figure the logic is reversed: pixels at 1 are black, while pixels at 0 are white).

One of the advantages of *opening* and *closing* is that they preserve the (discrete) size of the regions, unlike *erosion*, which reduces it, or *dilation*, which increases it. Depending on requirements, either a *closing* or an *opening* can be chosen. As these operators are idempotent, applying them several times does not change the result (which will be identical to that obtained after a single application). On the other hand, they can be chained (*opening* and then *closing* or *closing* and then *opening*) to improve the result image (noise reduction, filling holes, ...). By gradually increasing their radius, we obtain sequential alternating filters, which are particularly effective for removing noise.

In the MOTION project, naive 3×3 mathematical morphology implementations are given to you:

- Header: in the `include/c/motion/morpho/morpho_compute.h` file,
- Source: in the `src/common/morpho/morpho_compute.c` file.

See the `morpho_compute_opening3` and `morpho_compute_closing3` functions.

2 Assignments

2.1 Tasks Graph Simplification

A working system is implemented in the `src/main/motion2.c` file. First you have to read this code and to understand how it works. When compiled, the `src/main/motion2.c` source file produces the `motion2` executable binary.

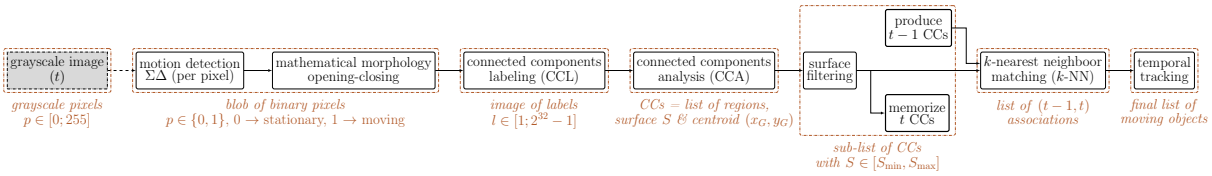

Figure 5: MOTION simplified tasks graph compared to Fig. 1. In *italic*: the output of each processing.

The tasks graph given in Fig. 1 performs two times the same computations for one frame. This can be avoided, and thus, the overall throughput in Frames Per Second (FPS) can be increased. A simplified tasks graph is given in Fig. 5. As you can see, some tasks have been removed and two new tasks have been introduced : *produce* and *memorize*. The latter will *memorize* the list of regions (CCs) at t to *produce* them at $t+1$ and thus avoiding useless re-computations.

In the `src/main/` folder you will create a new `motion.c` file that implements the tasks graph shown in Fig. 5. You can start from a copy-paste of the `motion2.c` file. **You will take care that the results of the newly produced `motion` executable are exactly the same than the ones from the `motion2` executable.** Refer to the next section (Sec. 2.2) for executable comparison and validation.

2.2 Code Validation and Debugging

`motion2` is the golden model. To compare the results of `motion2` and `motion` you need to generate the logs of `motion2` executable first (we do it for only 20 frames to execute faster):

```
./bin/motion2 --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
  --vid-in-stop 20 --flt-s-min 2000 --knn-d 50 --trk-obj-min 5 --log-path logs_ref
```

Secondly, you need to generate the logs of the `motion` executable:

```
./bin/motion --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
 --vid-in-stop 20 --flt-s-min 2000 --knn-d 50 --trk-obj-min 5 --log-path logs_new
```

Finally you need to compare the logs together:

```
diff logs_ref logs_new
```

If the later command returns nothing, it means that `motion2` and `motion` are equivalent (in term of features). This is good, your new implementation is correct! If not... it is time to debug :'(.

Note that in the following sections, you will modify some parts of the code that can have an impact on both `motion` and `motion2` executable. Then, it is strongly advised to carefully save the previous `logs_ref` folder for checking later!

2.2.1 Memory Leak

During the debugging process you may want to use sanitizer tools like `valgrind` or GCC/Clang `libasan`. This is a very good idea and it is likely that these tools will help you a lot. **But be aware that their are some memory leaks in the OpenCV library. Thus, when debugging, it is strongly advised to compile the code without the OpenCV library with the following commands**:

```
cmake .. -DMOTION_OPENCV_LINK=OFF
make -j4
```

Once the code is debugged, you can re-enable OpenCV (`-DMOTION_OPENCV_LINK=ON`) for a better visualization.

2.3 Coding Tools and Techniques

For the project you can use the tools and techniques we saw during this class. Here is a short list:

- CPU and GPU optimizations: Loop unrolling, reduction, cost of the operators, compiler optimization options, and so on
- CPU Single Instruction Multiple Data (SIMD)
  - **MIPP library**: `https://github.com/aff3ct/MIPP`
  - Intel SSE and AVX intrinsic functions: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`
  - ARM NEON intrinsic functions: `https://developer.arm.com/architectures/instruction-sets/intrinsics/`
- CPU multi-threading: **OpenMP** or threads POSIX
- GPU programming: **OpenCL**, OpenMP or CUDA

Using MPI is NOT recommend for this project, as we will not run the code on a cluster.

2.4 Code Performance Measurements

In this project, you will **focus on reducing the execution time** by using different types of parallelism and optimizations. To measure the execution time, you will not consider the video decoding time, the visualization time and the logs time.

MOTION comes with the `--vid-in-buff` parameter to hide the video decoding time: the frames are decoded and put into a buffer during the video initialization. This behavior is representative of a real embedded application, where there would be no video decoding. Instead, a camera would send frames at a fixed rate. Be careful, buffering frames can take a lot of memory (and time). Thus, we will limit this buffer to 100 frames.

Here is an example of a command line to use for the measurements:

```
./bin/motion2 --vid-in-buff --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 \
              --vid-in-stop 100 --flt-s-min 2000 --knn-d 50 --trk-obj-min 5
```

As a consequence, we will not take into account the whole execution time of the MOTION application. And, we will **focus on increasing its throughput** (= number of frames per second or FPS).

Of course, for debugging and validation purpose, you will need to use the logs (`--log-path`) and the visualization (`--vid-out-play --vid-out-id`) parameters.

It is possible to see the **average latency of each task by using the `--stats` parameter**. It can help you to see the impact of specific optimizations on a sub-part of the tasks graph. You can also try to run the code on different video sequences that have different resolutions. Optimizations can have a different impact on the throughput depending on the video resolution...

2.5 Code Optimizations

In this section, possible optimizations are given to guide you. The order of these optimizations is not mandatory, and it is possible to perform other optimizations. It is strongly recommended to draw inspiration from the optimizations seen in class. **Evaluation will be based on code performance. Please note: computations must be correct!** Refer to Sec. 2.2 for code validation.

2.5.1 "Standard" Optimizations

Here is a list of optimizations you should start with (not necessarily in the same order):

- Loop unrolling / loop fusion,
- Code vectorization (SIMD),
- Domain decomposition for multi-threading (`for-loop`),
- Porting $\Sigma\Delta$ and/or morpho to the GPU.

2.5.2 Operators Fusion

The mathematical morphology operators used are separable, factorizable, associative and commutative:

- An operator with a $d \times d$ 2D element is separable into two 1D operators with respective $d \times 1$ and $1 \times d$ elements and vice versa. For instance: $(3 \times 3) \leftrightarrow (3 \times 1) \circ (1 \times 3)$ and $(5 \times 5) \leftrightarrow (5 \times 1) \circ (1 \times 5)$ (see Fig. 6).
- Two 2D element operators of r radius give a 2D element operator of $2r$ radius and vice versa. For instance: $(3 \times 3) \circ (3 \times 3) \leftrightarrow (5 \times 5)$ (see Fig. 7).

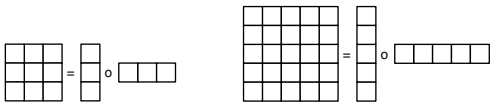

Figure 6: Left: decomposition of a (3×3) element into two (3×1) and (1×3) elements. Right: decomposition of a (5×5) element into two (5×1) and (1×5) elements.

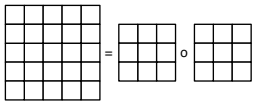

Figure 7: Decomposition of a (5×5) element of radius 2 into two (3×3) elements of radius 1.

2.5.3 Pipeline of Row Operators

Once each mathematical morphology operator has been optimized with the previous transformations, it is possible to optimize the sequence:

- Rather than applying one operator to an entire image and then doing the same with the next operator, it is possible to pipeline them (and to pipeline the detection with the post-processing).
- Image operators are split into "row" operators. These "row" operators are pipelined. This improves data persistence in caches close to the processor. This optimization is particularly effective in a multi-threaded context. This is known as *Cache Level Parallelism*, an improved version of *Thread Level Parallelism* (which is only concerned by computations).
- This optimization is applicable in scalar and SIMD. Note that it may require a prologue code and an epilogue code.

2.5.4 Computation and Memory Formats

In SIMD (but not only!), it is essential to maintain maximum parallelism for the computations efficiency:

- For binary mathematical morphology, instead of using one byte to store one pixel, it's possible to store 8 binary pixels per byte. This gives us 8 pixels for `uint8_t` elements, 32 pixels for `uint32_t` elements and 64 pixels for `uint64_t` elements.
- Finally, if data occupies less memory space (8 bits per byte versus 1 bit per byte), it is loaded/stored into memory more quickly. This optimization reduces processing time by reducing memory access times. FYI, this optimization is known as the *bit-packing* technique.

2.5.5 Combining SIMD and Multi-thread

The combination of SIMD and multi-thread parallelism means that memory accesses need to be highly optimized, as data will be consumed at very high speed. Here is a list a things you can do to reduce memory accesses:

- Reduce the number of memory accesses by operators fusion. We consider merging $\Sigma\Delta$ steps with morpho-math "row" operators.
- Maximize data persistence in caches to reduce transfer times by pipelining operators. Here again consider pipelining $\Sigma\Delta$ steps and morpho-math operators.
- Combine fusion and pipeline. The "ultimate" combination being the fusion/pipeline of all 1-bit morpho operators in SIMD×OpenMP.

2.5.6 Offload Computations to the GPU

Today GPUs are generally faster than CPUs to perform regular operations. Considering this, it could be very efficient to perform $\Sigma\Delta$ and the morpho on the GPU. However, do not forget that there is a latency to offload the computations on the GPU. Thus, it is better to avoid to do it too often. Here are some strategies you can consider to reduce the GPU latency:

- If you have a GPU that shares the RAM with the CPU, do not copy the data from the CPU to the GPU but use shared memory buffers instead : see `map` an `unmap` OpenCL features.
- Try to reduce the number of kernel invocations. For this you can perform recompute in the local workgroup and, for instance, you can perform one erosion and one dilation in the same kernel (see the operators fusion in Sec. 2.5.2).
- Maximize the use of the local memory instead of the global memory when possible. Remember that the local memory is much faster than the global memory.

2.5.7 Logical versus Binary Coding

There are two ways of coding mathematical morphology images. Either on `{0,1}`, or on `{0,255}`. The second choice is simpler, as it enables faster viewing and fine-tuning. The first choice allows boolean images to be stored on 1 bit. It is advised to use a conversion function from one format to the other to have both at the same time for debugging purposes.

2.5.8 CPU/GPU Heterogeneous Computations

Once you implemented a CPU and a GPU version you can compare the achieved FPS. Then, depending on the previous performance, you can split the image into 2 sub-parts, one to compute on the CPU and an other to compute on the GPU. This should increase the throughput even more. Be aware that this implementation will require additional synchronizations between the CPU and the GPU. And, you will need to recompute some data at the CPU/GPU frontiers to maintain the parallelism.

3 Due Report

The report is a document in which you will explain the optimizations you performed and you will show the performance improvement with plots and tables. You can write it in French or in English. But be aware that **the spelling and the care you take with the document will be an important part of the final grade**. It is strongly recommended to adopt the following organization (**in bold**: the most important sections):

1. Introduction / Presentation
2. **Description of the Optimizations**
3. Computer Architecture
4. **Experimentation Results**
5. Conclusion

Introduction is a section where you will present the project. You will explain its main purpose. It is not a "copy-paste" of this document. What is interesting is your vision/understanding of the project.

The description of the optimizations is an important part of the report, we expect that you explain the choices you made to improve the efficiency of the code. Thus, **it is strongly recommended to add some illustrative figures to facilitate the reader's understanding.** You can also add some small source code parts to show a specific technique. But be aware, you should not have too much source code on your report.

Before the experimentation section, you will describe your testbed: the computer architecture (CPU, GPU & memory).

In the experimentation section, you will mainly consider the performance in term of throughput (= FPS). For instance, you can observe:

- FPS depending on the number of cores (= speedup curves),
- FPS depending on the image sizes,
- FPS depending on the initial version versus the SIMD version,
- FPS depending on the CPU and the GPU versions,
- And so on.

To measure the FPS, you will prefer to use the throughput reported when calling MOTION with the `--stats` arguments, as it is more accurate. FPS gives the overall application performance, this is the most important metric. However, you can also comment improvements you made on one task ($\Sigma\Delta$ or Morphology) using the average latency in milliseconds. **Each time, for FPS or latency, you will not forget to specify your baseline: the initial performance of the `motion2` and `motion` executable without optimization.**

In the conclusion, you will recall the most important results of your project, as well as your overall point of view. Finally you will give some directions for future improvements.

List of things to avoid:

- Screenshots in general: please type the text in the document, images are generally heavy and ugly
- Unstructured document: use Microsoft Word, Libre Office, LaTeX, ...
- Start to write the report at the last time: you should start as soon as possible and complete it while you are working on the project

4 Work Submission

You will have to submit both the code and the report on Moodle. For the report, you will submit a "PDF" file named `report.pdf`. For the code you will send a "zip" file named `code.zip` without the binaries (delete the `build` folder) and the video sequences.