Task 1

a)

The running time of the loop dominates the running time of the rasterize-function, but there is also quite a bit of unaccounted run time (When running at 1\*normal resolution). When running at 5\*resolution, 15322 of 15922 ms are used on work, but the whole process takes:

real 0m27.586s

user 0m27.036s

sys 0m0.136s

Which means that about half of all the work is done elsewhere.

a.1) Thus, we check the time that is used for the entire program, from the beginning of main, until the end. The resulting output is:

Output from ratio 1920x1080:

Rendering an image on the CPU..

Loading '../input/spheres.obj' file...

Rendering image... 0/703 complete.

Finished!

Processing work queue =1635 ms

Entire rasterization =1704 ms

Writing image to '../output/sphere.png'...

With main: 2538

As we can see, the entire program uses 2.5 seconds, while the rasterization only uses 1.7 ms on the rasterization. This means that the program cannot get faster than 0.8ms, even with an infinite speedup

b)

We use openMP to parallelize the main loop. Since we have not been diving deep enough in the program to properly see how unbalanced the workload is,

we try both static and dynamic scheduling on a smaller dataset.

Depending on how unbalanced the work is, it might also be useful to divide the list into small chunks:

#pragma omp for schedule(dynamic, 1), or larger #pragma omp for schedule(dynamic, BIGGER\_NUMBER)

Output from ratio 1920x1080:

Rendering an image on the CPU..

Loading '../input/spheres.obj' file...

STATIC:

Processing work queue =9635 ms

Entire rasterization =9695 ms

With main: 10512

DYNAMIC (chunks of 3):

Processingwork queue =1626 ms

Entire rasterization =1673 ms

With main: 2259

DYNAMIC (chunks of 5):

Processingwork queue =860 ms

Entire rasterization =931 ms

With main: 1518

DYNAMIC (chunks of 1):

Processingwork queue =732 ms

Entire rasterization =798 ms

With main: 1348

It becomes quite self-evident that the static distribution takes longer, even longer than the unthreaded version. Thus, we may assume that the amount of work needed for each task can be quite different. There does not seem to be a monotonic growth with the chunk-size, but dynamic, with size 1 seems to be the best choice

(I just realized that the ratios were given in the wrong order, when i looked at the output-file....)

(The error does not seem to have affected the runtime or the distribution of the time spent on rendering the image. )

The image looks quite bad, a lot of it is not properly coloured, and some of the colours are smeared everywhere. Additionally, the result changes each time, which might imply race-conditions.

c)

I was somewhat stuck, since I thought the errors came from race-conditions from the depthBuffer or the frameBuffer, from when the threads tries to read/write from them. Just to see what would happen, I made the mesh and transformed mesh into copies instead of references, just to see what happened. They are references to a shared variable, which is dangerous, but I thought the resource was read-only, Which would make it safe. I expected to see an increase in runtime, as the need for copies would increase. Even though this also happened, the rendering did no longer have any errors

There is still supposed to be a race-condition in the line:

if( pixelDepth >= -1 && pixelDepth <= 1 && pixelDepth < depthBuffer.at(y \* width + x)) {

depthBuffer.at(y \* width + x) = pixelDepth;

Since depthBuffer is shared, there should be a possibility of some spheres being possible to see through, but the image might be so big that two threads writing to the same buffer at the same time becomes a rare occurrence.

Solving this race-condition is quite a bit more difficult, as it requires some synchronization for comparing a value from the depth-buffer with a local one, while writing back the smallest one of those, and also ensuring that if a new thread starts writing to the buffer, it will overwrite the current value. Since the lecturer said we can ignore this, we will.

d)

The results for static and dynamic are already documented. So I will simply mention those from guided:

UNTHREADED

Processing work queue =2345 ms

Entire rasterization =2402 ms

With main: 3192

GUIDED (1)

Processing work queue =591 ms

Entire rasterization =653 ms

With main: 1452

When comparing this to the unthreaded, we get a speedup (Of the total program):

static speedup: 0.3

dynamic speedup: 2.3

guided speedup: 2.1

When only comparing the parallelized part, we instead get:

static speedup: 0.2

dynamic speedup: 3.2

guided speedup: 3.9

When comparing scheduling strategies, there is no guarantee for which one will be faster, as static has less overhead, but no load-balancing. Guided and dynamic have more load balancing. With more load-balancing, there is also more overhead. Since this task involves something similar to a heap-structure saved as an array, the earlier elements will be larger, and thus require more computing. This means that the first thread gets a lot more work than all the others. If we have used the structure of the work-queue to redistribute the jobs, static might have been the fastest.

=> We can not expect one strategy to be faster than the others, only more or less robust.

Task 2

a)

Number of devices on the lab PC = 1

b)

I pipe the output to a .txt-file, so it is written there istead...

Task 3

a)

INITIAL assumption

-----------------------------|

| 0-> tot items | med: (26^d) (Long if large depth)

-----------------------------|

| --------------------------|

| 0 -> Mesh count | short: (each cluster of objects only contains a handful of them)

| --------------------------| [ 1.. 20]

| | -----------------------|

| |0 -> vertexCount/3| med: Depends on the triangle count [1.. 200]|

| | -----------------------|

| | | --------------------|

| | | |minX -> maxX| med: The largest spheres take 1/10 of the picture =>

| | | --------------------| [1..1000]|

| | | | -----------------|

| | | | minY -> maxY| med: Exactly the same as for minX -> maxX

| | | | -----------------|

When probing with the standard input:

Number of items to be rendered: 703

meshCount 5

meshes[0].vertexCount 2880

I tried to print the differences between max x and min x, but atom crashed.

I managed to open it with gedit, and the result was: allways less than 15 with 1080 x 1920. Thus, the actual result is closer to:

Input from the bash

-----------------------------|

| 0-> tot items | med: (exponential with depth)

-----------------------------|

| --------------------------|

| 0 -> Mesh count | short: (constant)

| --------------------------|

| | -----------------------|

| |0 -> vertexCount/3| long: (Constant)

| | -----------------------|

| | | --------------------|

| | | |minX -> maxX| short: (quadratic with resolution)

| | | --------------------|

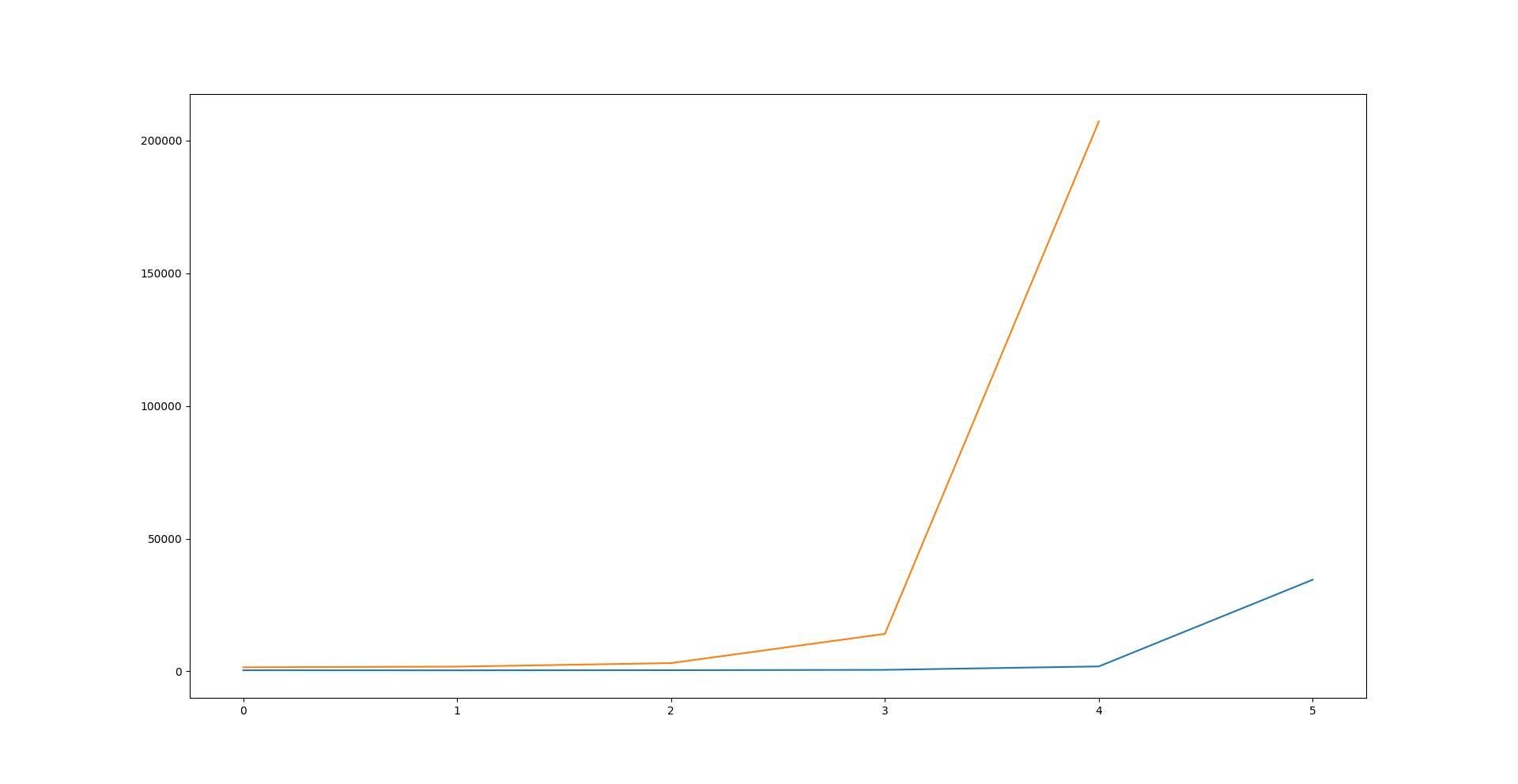
| | | | -----------------|

| | | | minY -> maxY| short: (quadratic with resolution)

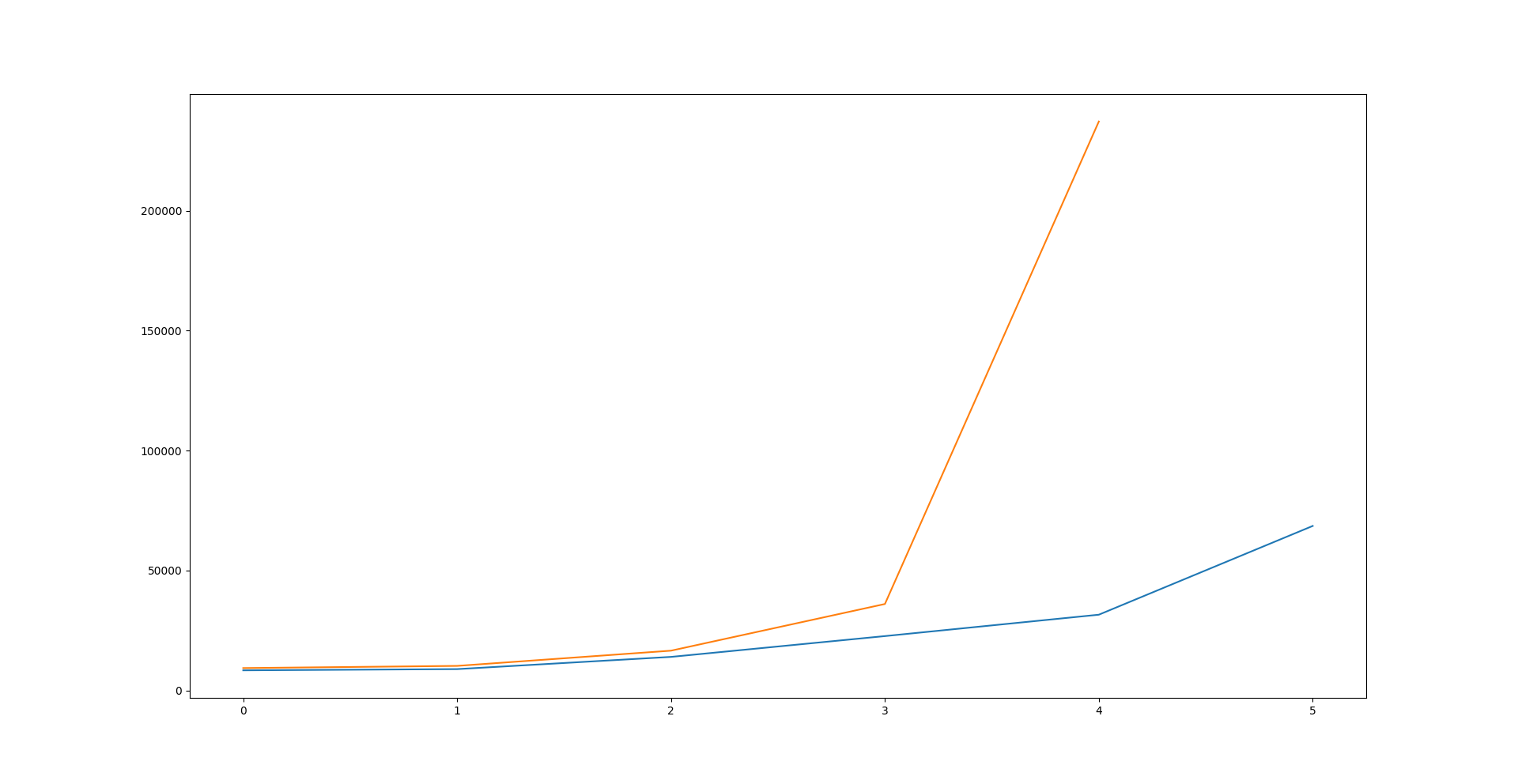
| | | | -----------------|

We decided to parallelize the rendering of the triangles. This would ensure spreading out the workload and having enough jobs that cores would not stay idle. The two innermost loops would then stay intact, but the three outer ones would be dismantled. Because we can launch a three-dimensional grid of thread blocks, this fits nicely with our three loops. The potential speedup will become larger as there are more tasks that can be paralelized. When ploting the time used for rendering at different depths, it becomes clear that the differnece between the CPU and the GPU.

Only the rasterisation (CPU in yellow, and time in ms) :

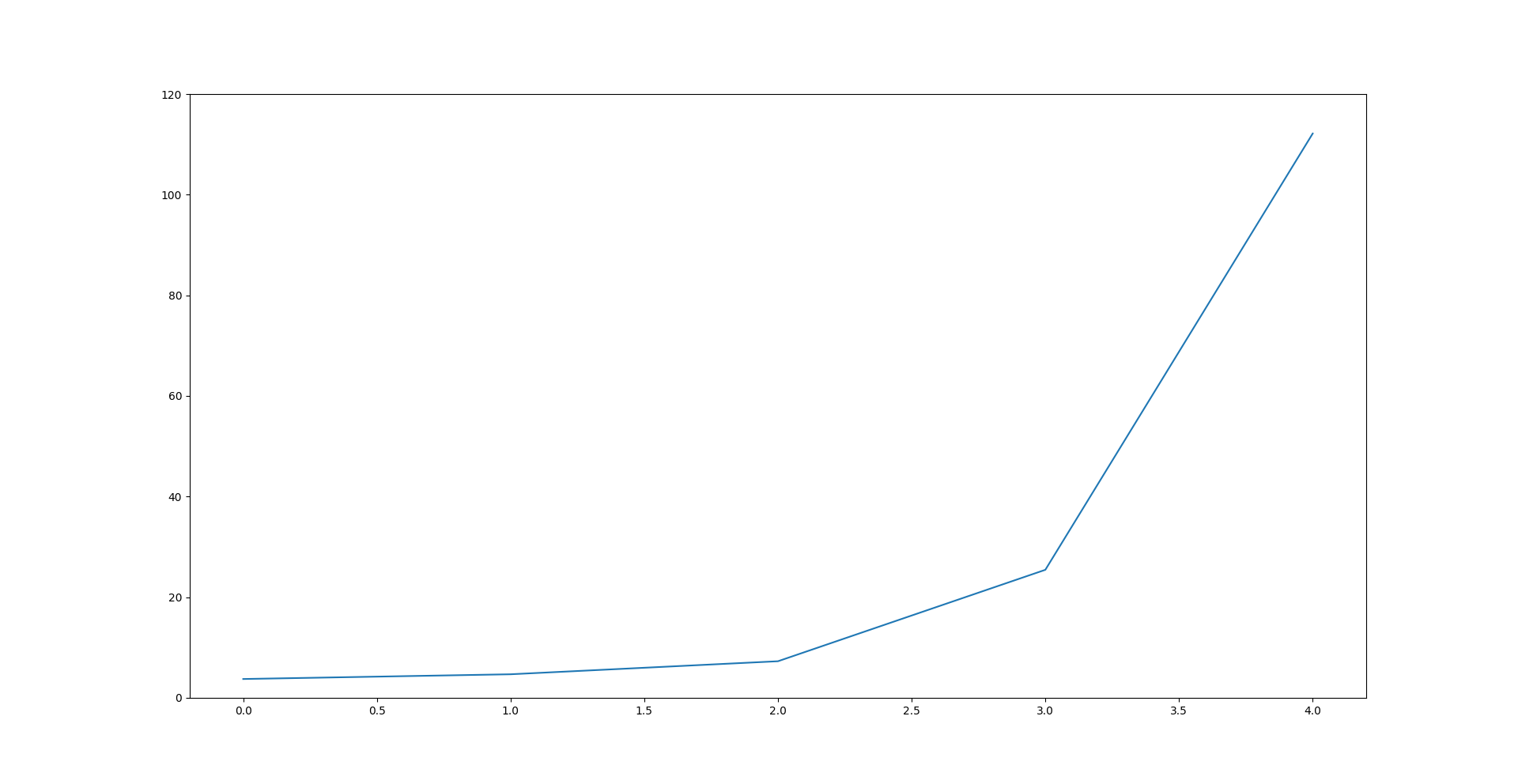


Rasterisation and saving (CPU in yellow, and time in seconds) :



We see that as long as the depth is suffichieltly large, the rasterisation completely dominates the runtime, but if the depth is 1, the result is that the rasterisatin is ~400ms, while the entire program roughly takes 8 seconds with a resolution of 5400 x 9600.

The resulting speedup is:



As we can see, the GPU is not all that much faster at smaller depths. It is a bit faster, since we still have a significant number of pixels to be set and a decent number of triangles. As we go towards larger depths, we move from a rather large speedup at depth 4 to an absurd one at the depth of 5 and 6 (6 is still running at one of the lab-PCs) .

In our implementation, we divide on the threads with regads to the number of triangles, resulting in a loss of processes equal to the number of items to be iterated, multiplied by the number of meshes in each item, but we did not have time to implement this, as making the code work took too much time.

After running the program on the CPU for a veeeeery long time, the result was 82 minutes, and 32 seconds of total. It is not included in the plot, as the values would completely dominate everything else, but the almost 50 times speedup still remains.

Tables with the numeric values of the runtimes : (The depths are 1,2,3,4,5 and 6, the resolution is 5400 x 9600 pixels, the ast measurement form the CPU never completed)

CPU

* Terminal output from time

1510 ms

1770 ms

3100 ms

14117 ms

207167 ms

4924581 ms

* measured form rasterise with std::chrono

0m9.373s

0m10.283s

0m16.628s

0m36.079s

3m57.153s

82m38.746s

GPU

* Terminal output from time

405 ms

379 ms

427 ms

555 ms

1847 ms

34517 ms

* measured form rasterise with std::chrono

0m8.438s

0m8.941s

0m14.036s

0m22.704s

0m31.619s

1m8.607s