Voxel Based 3D Mapping

Created by:
Nándor Szalma & Tamás Szekeres

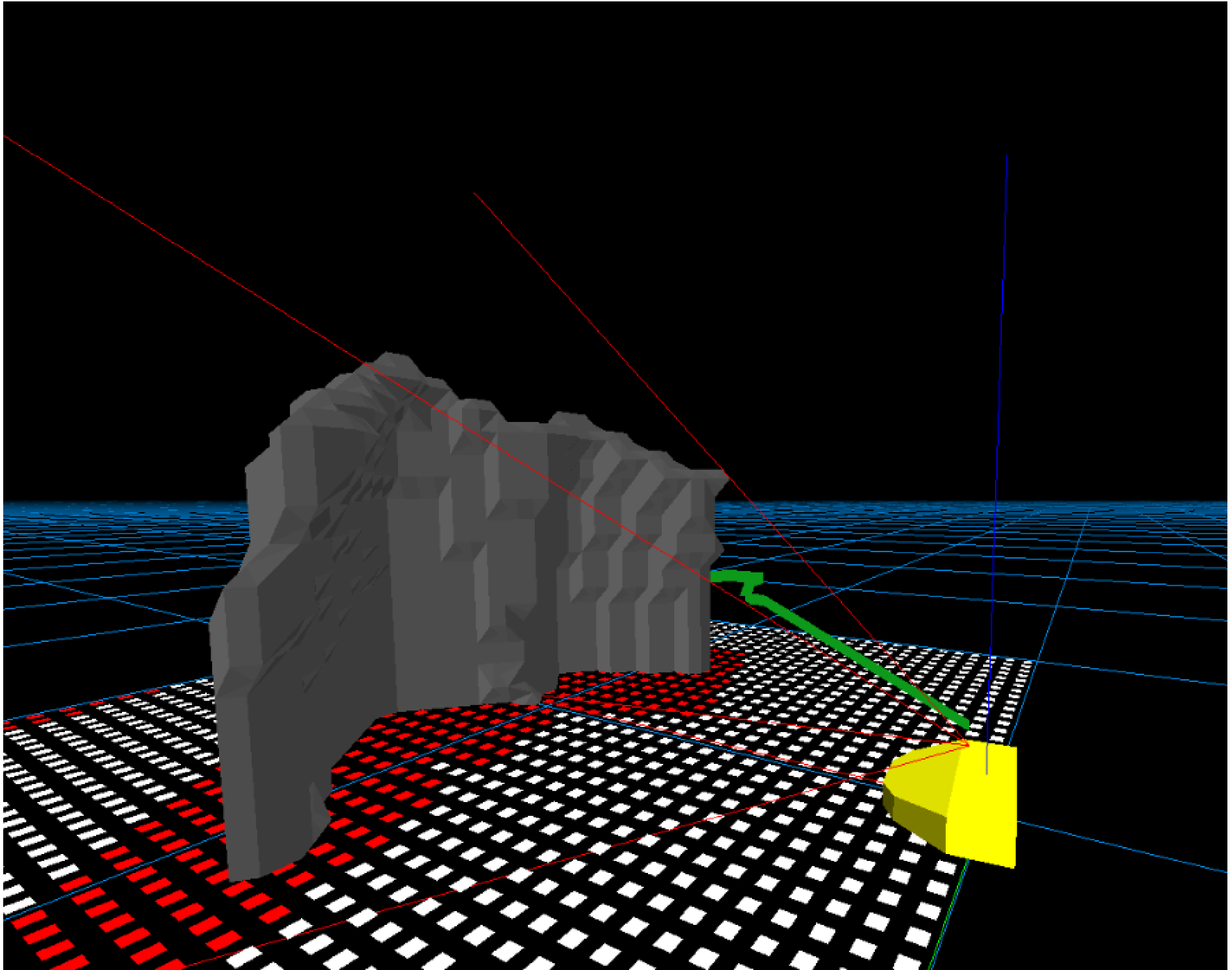


Table of Contents

1. Introduction 3
    1.1 Open Source 3

2. Hardware 3
    2.1 3D printed parts 4
    2.2 Electronic parts 4

3. Tracking 5
    3.1 Orientation 5
    3.2 Position 5

4. VoxelMapper 5
    4.1 Voxels 5
    4.2 Controls 5

4.2. Depth frame processing 6
    4.2.1 Multi-threaded compression on the robot 6
    4.2.2 Frame decompression 6
    4.2.3 Frame projection 6
    4.2.4 Chunk update 7
    4.2.5 Visualizing the chunks 7
        4.2.5.1 Optimizations 7
    4.2.6 2D map for navigation 8

5. Path finding 8
    5.1 Criteria of the selection 8
    5.2 Choosing the best algorithm 8
    5.3 How the A* works? 9

6. Conclusions 9
    6.1 Issues 10
    6.2 Further improvements 10

7. Sources 10

1. Introduction

The robot is able to reconstruct a 3D map from the position, orientation and depth map provided by the Kinect. The robot is remote controlled, tracks It's own position and sends the sensor data to a remote computer for processing.

1.1 Open Source

The source code was developed using git and available here:
gitlab.com/nandee95/voxelmapper
gitlab.com/nandee95/voxel-mapper-onboard

2. Hardware

Since the whole robot is larger than our 3D printer It's constructed from two halves. The two control wheels and the planetary wheel is placed in a triangular shape for stability. The Kinect's weight sits on the two rear wheels. The battery pack moves the center of inertia closer to the center of the triangle.

○ 2.1 3D printed parts

Name	Print time	Material
Body (2 pcs)	2 x 6h 30m	PET-G
Cover (4 pcs)	2 x 1h 50m + 2 x 1h 35m	PET-G
Battery pack (complete)	5h	PLA
Kinect support (bottom + top)	2h 47m + 2 x 30m	PLA
Planet wheel bell	2h	PLA
Battery charger	3h 8m + 2h 44m	PLA + PET-G
Fixing belts	43m	PLA
Motor supports	2 x 54m	PLA
HAL Sensor supports	17m	PLA
Raspberry Pi supp. pillars	43m	PLA
Hinges	45m	PLA
Other tests and fail prints	~5h	PLA + PET-G
	39h 40m	

2.2 Electronic parts

Name	Pieces	Price (HUF)
MPU-9250	1	*resused*
AS5600	2	1900
Any direction wheel	1	360
HC-SR04	2	620
HC-SR04	1	*reused*
Servo wheel	2	1000
100A 5S BMS	1	1150
Battery spring	5 pair	150
Power button	1	500
IR proximity sensor	4	660
18650 battery cells	5	*salvaged*
Step-down (battery - 12V)	1	700
Step-down (battery - 12V)	1	1490
Battery connectors	1 set	260
		8790

3. Tracking

The robot needs to know its position and orientation to be able to project the captured images into a 3D space.

3.1 Orientation

The orientation data is acquired from the MPU9250 sensor which contains a compass, accelerometer and gyroscope. The sensor is driven by a library ported from Arduino.

3.2 Position

The positioning is performed by two AS5600 contactless potentiometers. Since both using the same i2c address(which can't be changed by any way) one is wired to the hardware i2c and the other one uses a software i2c. We glued a magnet to each wheel and those sensors are able to tell how much they turned. From the perimeter and distance of the wheels we can calculate where those wheels will end up after rotating. By averaging those positions we can calculate a displacement vector and this vector can be added to the global position. This calculation loses precision by the longest time steps we take so they are performed in their own thread about 100 times a second.

4. VoxelMapper

VoxelMapper is a C++ application we developed to visualize the scanned 3D map. It's using mődern OpenGL for the graphics and OpenCL for the gpu calculations.

4.1 Voxels

The voxel word comes from "volume pixel" which means the world is defined by a 3D array. The array consists of ones where there is volume and zeros for air. Each voxel is 50x50x50mm in our world. The world is split into chunks for efficient rendering and updating. Each chunk is 1x1x1m and right above the floor. This means there are 20x20x20=800 sample points in a cubic meter.

4.2 Controls

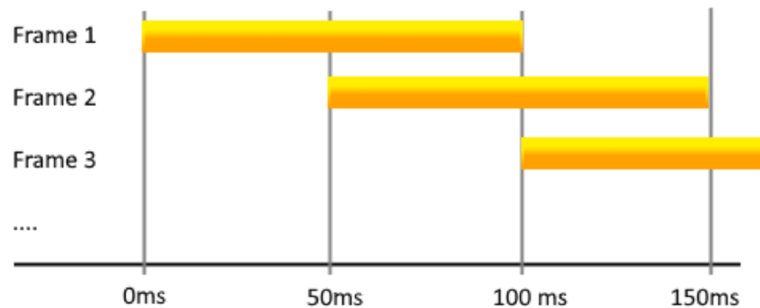
The camera can turned by right clicking on the window area and moved via WASD. Left click starts the path finding algorithm. The robot can be remote controlled by the arrow keys. C brings the depth map to the center of the screen. F5 forces to recalculate all the meshes on the map (mainly for testing purposes). F6 clears all of the chunks. F11 toggles fullscreen. Escape closes the application.

4.2. Depth frame processing

4.2.1 Multi-threaded compression on the robot

The depth frame is 640x480 and each pixel is a 2 byte unsigned integer. Each depth frame is consists of 600 kb of data. The frame needs to be compressed but we can't use pre-existing image formats since image components are usually consists of a single byte. We ended up using zip compression which takes about 100ms on the highest performance settings. This would result in less than 10 fps so we needed to further improve the system. The program starts 3 threads just for compression after receiving each frame from the Kinect the data gets pushed into a queue and one thread wakes up to preform the compression task. The compression still takes the same time but we are able to do multiple compression simultaneously. The absolute highest fps we can reach this way is about ~25 fps but after introducing sensor readings and processing it dropped down to ~20 fps.

As a result the 12 Mbps network traffic(uncompressed) is reduced to 2Mbps(compressed).



4.2.2 Frame decompression

The frame is packed together with the sensor reading and gets sent to the desktop application via sockets. The decompression takes about ~4ms on a modern hardware so no threading or further processing needed.

4.2.3 Frame projection

The frame as it is gets uploaded into the GPU (into an OpenCL buffer). We can think about the frame as a collection of distance data in different angles from the camera. The kernel program that runs on the GPU calculates those angles converts it into a normal vector and multiplies It with the actual distance(pixel value) which results in world coordinates. Since our graphics is consists of small blocks those coordinates needs to be divided by the block size(50mm) to get grid coordinates. If we think about it this information not only tells us that there is something in this direction and this far but we also know that there is free space between the camera and this point in space.

The points of the line needs to be rasterized so we know which elements in the grid needs to be turned into air. Bresenham's line algorithm runs in the GPU and returns the grid coordinates between the point and the camera. The final grid's values are the following:

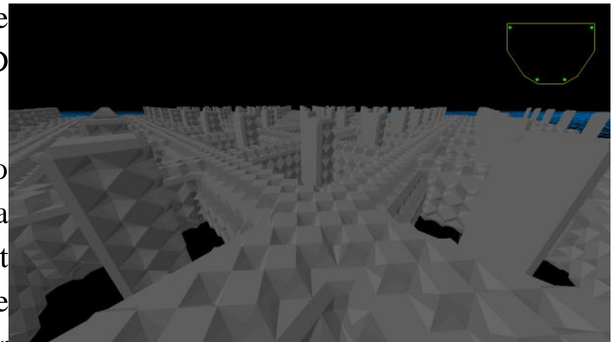
-1 → turned into air
0 → remains unchanged
1 → turned into solid

The results are stored in a 3D array(held in GPU memory).

4.2.4 Chunk update

This information needs to be transformed into our chunk system. A different kernel receives the projected data and 25(5x5) chunks around the robot. It's only job is to perform the changes we calculated earlier on the chunks.

4.2.5 Visualizing the chunks



The chunks are still defined with a 3D array. The marching cubes algorithm is used to generate a closed 3D mesh.

This mesh still exists in an OpenCL buffer we need to pass it to OpenGL for rendering. This step involves a GPU-GPU transfer. Without proper shading we couldn't tell the difference between individual triangles on the screen so the kernel also computes the surface normal for each triangle which is later used for some basic direction based shading in the shader.

4.2.5.1 Optimizations

This is by far is the most expensive step in the pipeline. I needed to perform multiple optimizations to reach near real-time performance. The steps required in this step is increased rapidly by increasing the grid resolution.

At first all of the chunks were updated after each frame but it was improved by reporting from the kernel which chunk actually changed and only performing the update step on those chunks.

The chunks were updated one after another in the GPU. This can by improved since the GPU operations not actually blocking the current thread. The updates were split into two parts. At first I cycle through the chunks that needs to be updated and start the GPU operation in their own command queue. In the second step I wait for the operation to finish if It's not already finished and download the data which is needed by the CPU. This only made the execution about 6x faster.

The algorithm contains a lookup table which is by itself greatly improves the execution time but storing this table for each GPU thread uses a lot of VRAM. It was moved into a global buffer and only stored in the memory once.

At first the chunk update took about ~150 ms for just a few chunks but I was able to reduce that time to 10ms to 25ms (depending on how much chunk updated).

4.2.6 2D map for navigation

The path finding algorithm needs to know where is safe for the robot to move. The third kernel takes the bottom 5 layers and projects it to the floor. Since the robot has multiple dimensions not just a simple dot also checks for any solid points in a radius. The resulting map shows where is safe to move for the robot.

5. Path finding

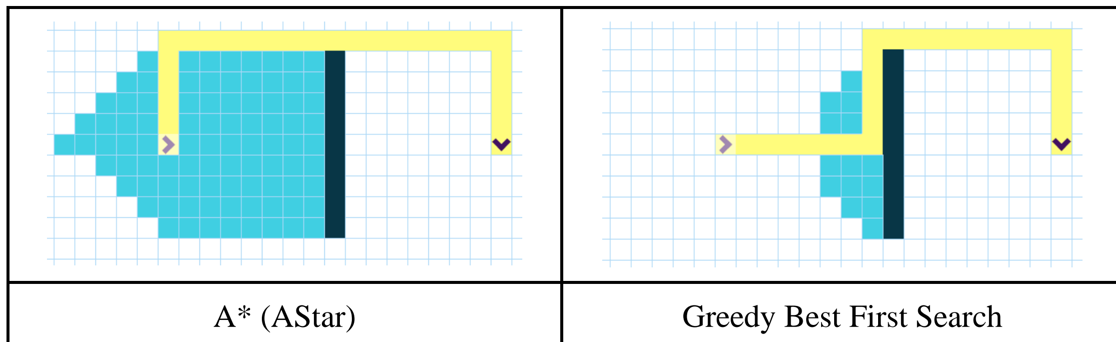
We used the A* algorithm for pathfinding. It's the most widespread algorithm for pathfinding these days. The best thing in this algorithm, that it always finds the shortest path and runs faster than the Dijkstra algorithm.

○ 5.1 Criteria of the selection

- Turn less
- Always find the path if it possible
- Make sure that the found path is the shortest

○ 5.2 Choosing the best algorithm

Although there are several good algorithm on the internet, it didn't take long to reduce the list to two types of algorithms. Those were the **A*** and the **Greedy Best First Search** (GBFS). Both of them are successors of the Dijkstra algorithm. Whats makes them better, that they have Heuristics. Unlike Dijkstra's algorithm where you mostly brute force the search, these techniques apply weights on each individual node that tells you which way to go. The difference between the two, that **A*** follows a well restricted rule set meanwhile the **GBFS** pushes forward until it bumps into obstacle then starts seeking for a way to continue just like **A***. In summary, if you want the shortest path and the runtime not matters the A* is the answer. If you need an algorithm that runs fast (even realtime) but a bit inaccurate, use **GBFS**. It is noteworthy that the **GBFS** makes more turn and can be mislead with promising dead-end mazes.



A* (AStar)	Greedy Best First Search

- ### 5.3 How the A* works?

A* is an **Informed Search Algorithm**, or a **best-first search**, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost in our situation the distance. It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its reach the given goal.

At each iteration, **A*** needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* uses three weights **G**, **H** and **F** to select the path that minimizes the cost to the goal. **G** is the distance from the start. **H** is the distance to the goal. **F** is the sum of the previous two. When **A*** steps forward it always select the node with the lowes **F** value. For more information about the algorithm see the **Introduction to A*** from **Amit**. His explanation is far the best in my opinion.

6. Conclusions

6.1 Issues

The compass is not able to give real time results and the whole map gets distorted.

6.2 Further improvements

- The projection kernel and the merging kernel could be combined together to save memory.
- Trying different, more responsive compass.
- Positioning from the already discovered map.
- Calibrating the depth frame.
- Use of a faster single board computer.
- Linear interpolation between the vertices for more smooth surfaces.
- Reducing the grid size.
- Autonomous mapping.

7. Sources

Those are the main sources that lead us through the development process.

Bresenham's algorithm python implementation:
https://www.geeksforgeeks.org/bresenhams-algorithm-for-3-d-line-drawing/

Marching cubes:
https://www.youtube.com/watch?v=M3iI2l0ltbE
http://paulbourke.net/geometry/polygonise/

Pathfinding demonstrations:
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
https://www.gamedev.net/articles/programming/artificial-intelligence/a-pathfinding-for-beginners-r2003/