Research internship result

Intern: Martin Macé de Gastines

School: CYtech

Formation: ING2 GIA 2024/2025

School tutor: Naim Zoghlami

Foreign school: KAIST

Professor: Min H. Kim

Acknowledgements

I am thankful to Professor Min H. Kim who accepted me in his research lab, for his support and advice, I would also like to thanks Andréas Meuleman who informed me of this lab and contacted Min H. Kim for me, and everyone of the Visual Computing Lab for their warm welcome.

Table of contents

Introduction

The choice of this internship was motivated primarily by my desire to learn graphic programming, that i was practicing as a hobby for a few years. I got contact with Min H. Kim through Andréas Meuleman, with whom did a thesis. This internship in Korea was also a great occasion to discover a new culture, as I have never been in Asia before.

Goals

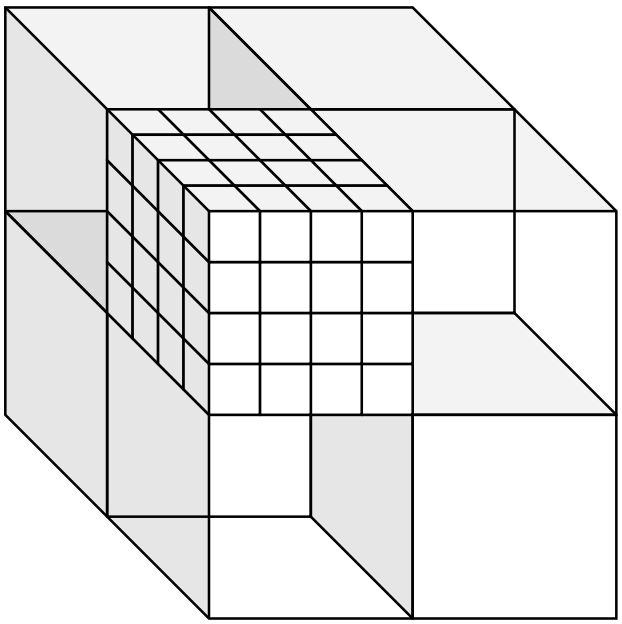
My mission given by my professor was to re-implement the techniques described in Texture Fusion[1] and Normal Fusion[2] using the Open3D library[3]. The texture fusion aswell as Normal fusion article base itself on the Voxel Hashing[4] reference implementation. The goal is to remove the dependancy on Direct3D 11 and Kinect to make it cross platform and use different models of cameras.

SLAM

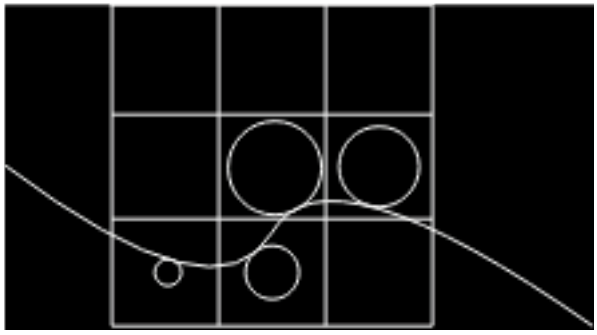
The techniques described in these articles are meant to solve the simultaneous localization and mapping(SLAM) problem. Given a series of depth or color images, you want to reconstruct the camera trajectory and the environment. This is a chicken and egg problem since you need the camera position to integrate depth and color into the geometry, and the geometry to accurately compute the camera position. To approximate a solution, we alternate the depth integration and the camera odometry. This allow the geometry to converge to the desired surface and because we compare each images to the geometry we're currently building, this limit the camera drift compared to the position estimation from successive images only.

Texture Fusion

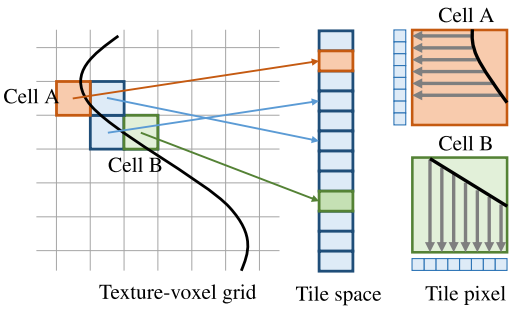
The texture fusion article describe a technique to do realtime high quality RGBD scanning. It reuse the data structure described in voxel hashing, which consist of a hashmap associating position with a block of voxel data.



This voxel data contain a truncated signed distance field(tsdf) [5] and color. We estimate the tsdf at any point by doing a trilinear interpolation on all 8 surrounding voxels. The signed distance field is mapping the distance to an implicit surface. When the value cross 0, we are crossing the surface.



This surface can be exploited directly with raycasting or converted to a mesh via some techniques like marching cubes or dual contouring. The advantages of tsdf is that the implicit surface is easier to manipulate than a regular mesh. We can access a specific parts of the surface simply by accessing the voxel block at these coordinates. It also gives smoothed surface instead the facetted surface you would get with a mesh. The improvement over voxel hashing come from storing tiles of n^2 colors instead of just one color per voxel. This allow for a much higher resolution than the per voxel color technique for the same memory requirements. The tiles for each voxel maps the color on the surface to a side of the enclosing cube, similar to TileTrees[6], with the difference that we are are assuming the surface flat enough that all pixels can be mapped to one tile.



Open3D

Open3D[3] is an open source library. It provide some data structures and algorithms to work with 3D data. It support CPU and GPU computation with CUDA. The library is mainly designed as a python library with an API similar to numpy, but it is also possible to use the C++ API. The library is centered around the Tensor type, that represent a multidimensional matrix. They can be created either on the CPU or GPU, and subsequent operations will be their device. Another core type is the hashmap, which map some tensors of the same size to some indices, which can be used to access attribute data in another tensor. A lot of algorithms are also provided, like camera odometry and global registration.

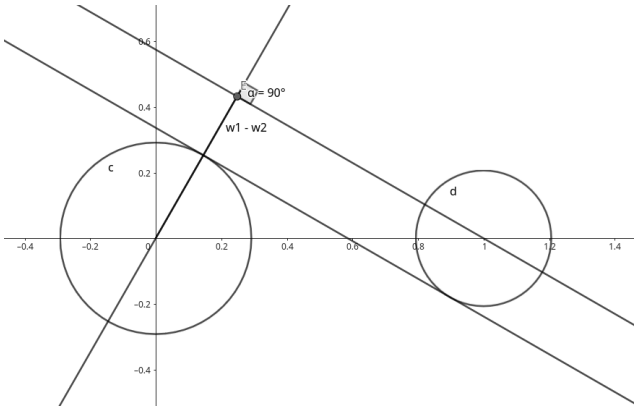
Mission

Data structure

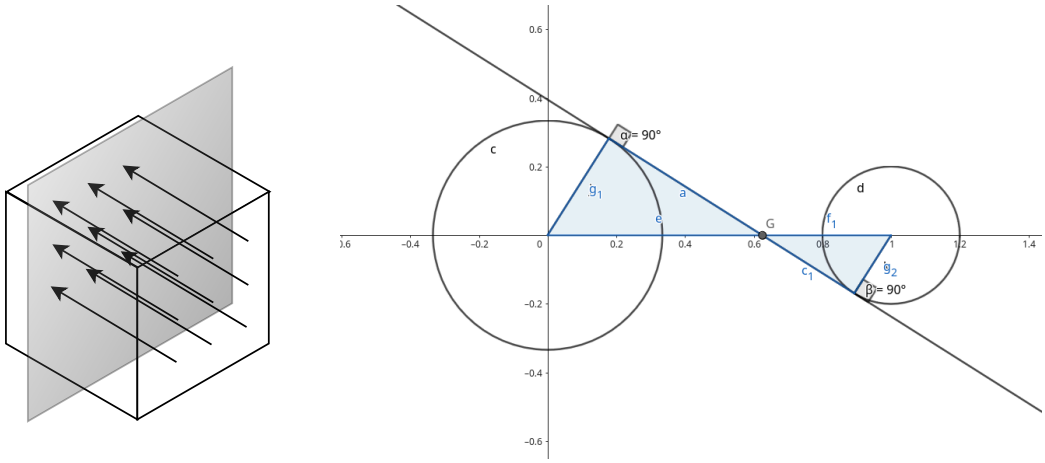
To implement the voxel hashmap in Open3D, I reused an existing data struture "VoxelBlockGrid" that is composed of a hashmap that associate 3D positions to blocks of user defined voxel data, which is similar to the one described in Voxel Hashing[4]. I extended the class to add a pool of tile, and specified an index into the heap in the voxel data. An index of -1 mean no associated tile. To insert new tiles I increment a counter and set the index to the previous value, simulating a stack. There is nothing yet to remove a tile but assuming the reconstructed geometry converge, you don't significantly lose memory. Thanks to C++ inheritance, any method that doesn't involve color can be directly used on my new class, for example depth integration,mesh extraction and raycasting(if you are not querying color).

Color integration

The first strategy I used for color integration is to project an image on the geometry and for each pixel of the image, set a pixel on the surface tile. I used the builtin raycasting function, and from there I was able to compute the voxel position, uv, direction and allocate tiles if needed using only tensor operations. I realized this method have a few drawbacks, for example it will leave gaps if the image projected have a lower resolution than the tiles. On my next attempt I followed the Texture Fusion reference implementation: we determine "zero-crossing voxels" , voxels on which the tsdf come across a value of 0. This is all the voxels that contain a part of the implicit surface. Then we detemine the major axis of the surfaces's normal to maximize the usage of the texture tile. One convenient property of tsdf is that assuming a planar surface, you can easily compute one component of the normal with two value along an axis.



With a distance of 1 unit, the normal component along the axis is simply the difference between the two tsdf values. In our case, considering a cube of 8 tsdf values, we compute for all 3 axis the average of the normal component along 4 edges, then normalize the vector of these 3 components to get an estimation of the normal. Then we project each pixel position into the surface.

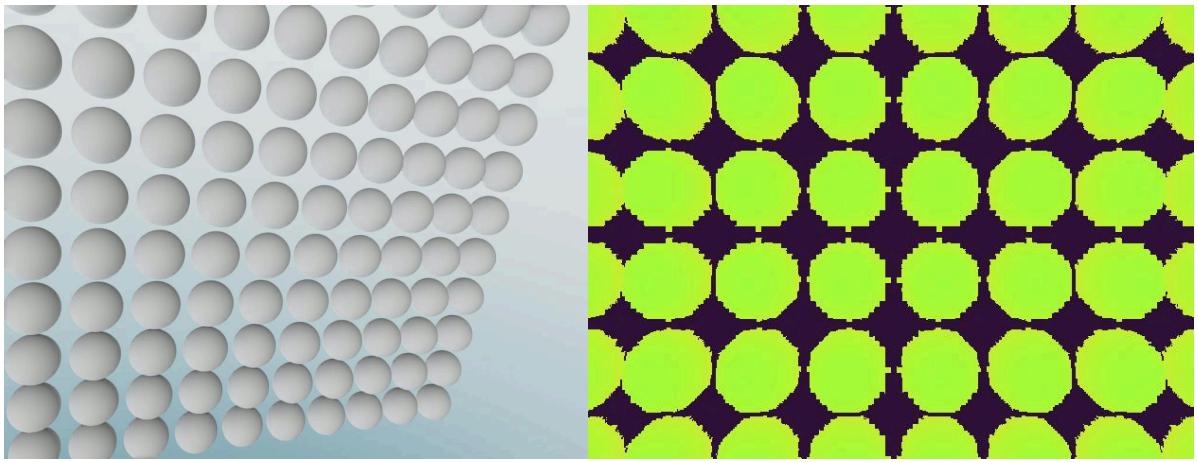


For this we also assume a planar surface. The distance from either sides and the tsdf values are proportional so the intersection can be expressed as $\frac{w_1}{w_1+w_2}$. We take as values a bilinear interpolation of 4 values on both faces. The surface point is then unprojected on the camera frame and we can directly fetch the pixel on the image. To implement this in Open3D, tensor operations were not convenient and efficient enough. I had to defines a custom kernel. A kernel in Open3D is a low level function that can call CPU or GPU code. It first dispatch between CPU and GPU implementation then by the type of data used. Kernels often support multiple data types through templates, and often implement the CPU and GPU algorithm with the same code. The way they do it is by creating a single header used by both code. The code used notably the ParallelFor function provided by Open3D, that run code in parallel either using multithreading on CPU or on a CUDA kernel on GPU, on top of that you have to use the preprocessor several times to overcome the differences between C++ and CUDA. The color integration works using two passes: The first one go over all voxels and select zero crossing voxel facing the camera. Their positions are stored in a buffer using an atomic counter. If they don't have a tile associated yet, it get a new tile index by incrementing the tile pool atomic counter. The second pass go over all pixels of all selected voxels and blend their color with the color of the source image.

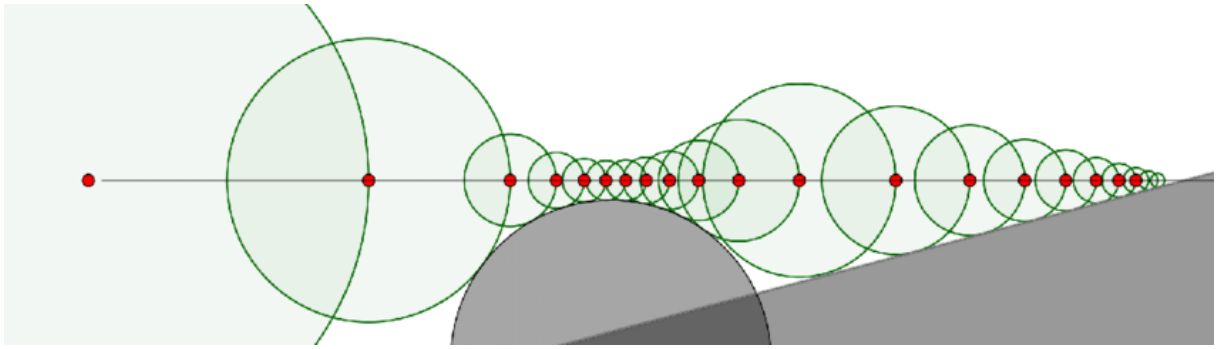
Raycasting

Raycasting is used to get an image of the current reconstructed geometry and color, to estimate the camera movement with the next frame. It does so by stepping ray on the space until we find a surface. The VoxelBlockGrid already had a method for raycasting, however it didn't had the subvoxel precision to read the correct pixel.

Here an example with a test scene:



The algorithm does sphere tracing, which mean advancing the ray by the tsdf value of the current position as it should not encounter a surface, then compute an intersection once it hit a negative tsdf value



I made a few adjustments to it, by switching to dual voxel coordinates to improve the accuracy and computing multiple intersections like a dichotomy. Some tradeoffs were made, for example stepping in empty blocks is done at a constant interval, and can cause artifacts when crossing chunk borders. A better but probably more expensive solution would be to use a digital differential analyzer(DDA) algorithm to directly step on blocks borders.

Texture Transfer

The depth integration modify tsdf values changing the implicit surface. This cause the projected colors to no longer be valid. A solution for that is to search a corresponding point along the normal on the old tsdf value. We can then transfer the color on the new surface. For that I take a point offset by the normal at some distance and do a dichotomy with the old values. This require duplicating the values and copying them before each depth integration.

Camera input

One of my task was to use a Realsense camera I had at my disposition. Fortunately, Open3D have an IO module that provide a way to read from a camera input, including Realsense cameras. At first I read input frames from ROSbag files, which are a recording of color and depth video. This gave me the ability to replay the same file for debugging purpose, and allowed me to use much higher quality sample. Then I added support for camera input once I had a working camera tracking.

End Result

All of this was tied together in a command line application I made from a modified Open3D example that performed SLAM on a VoxelBlockGrid with a color per voxel. I was able easily switch between this and my implementation. The reconstructed geometry was identical because there were no modifications to depth integration and camera tracking was only using depth information.

The texture was very blurry on both, which make the increased resolution of the 2nd technique not apparent. I supposed this was due to a missing part of the Texture Fusion paper, the local texture optimization, which distort the image to match the rendered image, avoiding a lot of the blur from camera artifacts and reconstruction imprecisions. Unfortunately, I started integrating this algorithm only in the last week, and didn't got far enough to be able to run it.

I also expected more performance considering the high end GPU I was working with (Nvidia TITAN RTX). The raytracing was barely running at 40 fps, despite being able to to do 10000's of step per ray on a flat grid. I suspect the hashmap being quite expensive, the lookup probably causing a lot of branching when traversing a grid, and so hurting parallelization. An idea could be to use a bitfield mask for fast traversal.

Conclusion

In conclusion, this internship was highly enriching in many different way: I had the opportunity to explore a new country and a different culture. I also discovered the field of research in computer science and developed some interest into it. My mission here taught me a lot about graphic programming and gave me a great challenge. I was able to dive deep into GPU programming and linear algebra to surpass myself and discover many more subjects related to visual computing.

Bibliography

[1] J. H. Lee, H. Ha, Y. Dong, X. Tong, and M. H. Kim, "TextureFusion: High-Quality Texture Acquisition for Real-Time RGB-D Scanning," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020.

[2] H. Ha, J. H. Lee, A. Meuleman, and M. H. Kim, "NormalFusion: Real-Time Acquisition of Surface Normals for High-Resolution RGB-D Scanning," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2021.

[3] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A Modern Library for 3D Data Processing," *arXiv:1801.09847*, 2018.

[4] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3D Reconstruction at Scale using Voxel Hashing," *ACM Transactions on Graphics (TOG)*, 2013.

[5] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, in SIGGRAPH '96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 303–312. doi: 10.1145/237170.237269.

[6] S. Lefebvre and C. Dachsbacher, "TileTrees," in *Symposium on Interactive 3D graphics and games (I3D 2007)*, in I3D '07 Proceedings of the 2007 symposium on Interactive 3D graphics and games. Seattle, United States: ACM, Apr. 2007, pp. 25–31. doi: 10.1145/1230100.1230104.