3D Real-time Reconstruction

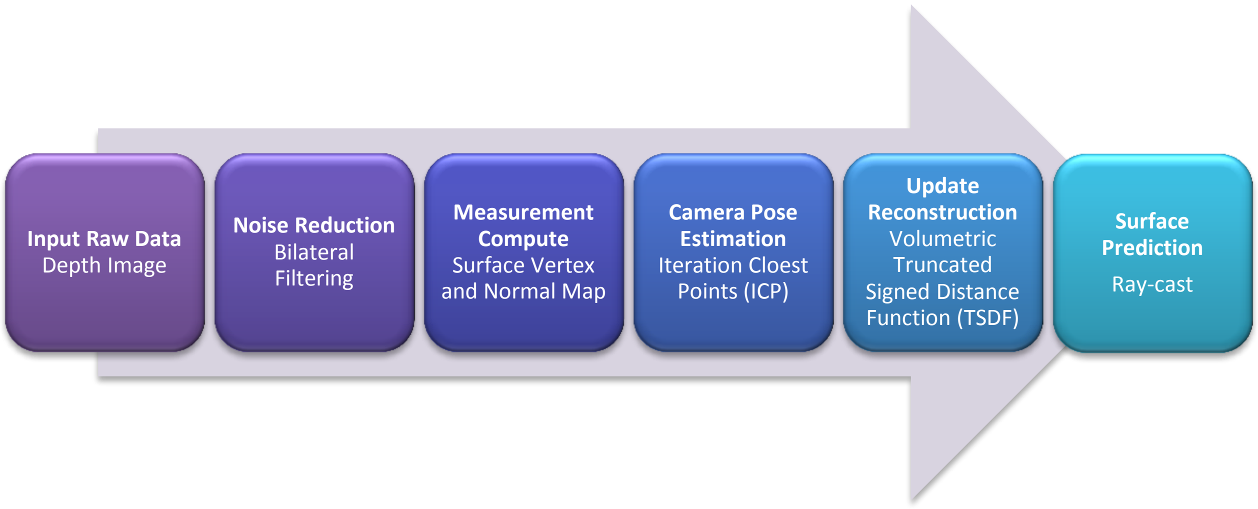
Jiakai Zhang, Hao Liu, Yu Xu

New York University

Contents

1. Introduction

We want to re-implement the paper1 from Microsoft Research to reconstruct the cloud points of the whole indoor scene using Kinect. The pipeline is as follow. Each step will be described separately.



1. Input raw data – depth Image

The Figure 1 shows the raw data from Kinect which is RGB Image and Depth Image.

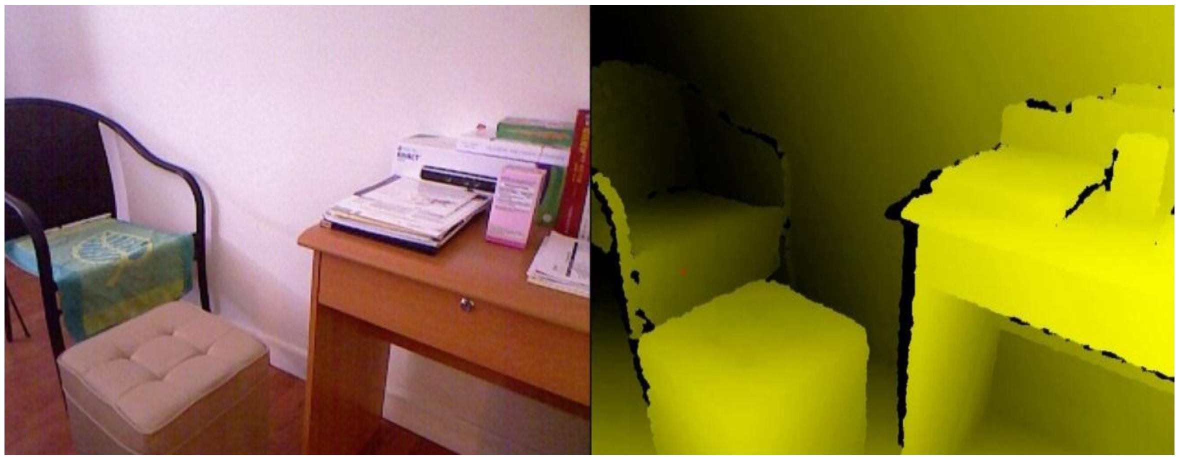


Figure 1 the Raw Data

The Kinect Camera has 30 FPS. The resolution for the depth image is 640 by 480. The two main problems of the raw depth map are noise and imperfect. Specifically, the depth data is missing at the edge of objects.

[1] Richard A Newcombe, Shahram Izadi, Otmar Hilliges et al. KinectFusion : Real-Time Dense Surface Mapping and Tracking. *IEEE International Symposium on Mixed and Augmented Reality, 2012*

2. Noise reduction – bilateral filtering (Jiakai Zhang)

The raw depth data from the Kinect is pretty noisy. It's hard to use for camera tracking. If we apply the Phong-shading to represent the normal map, the noisy normal vectors make the objects irregularity.

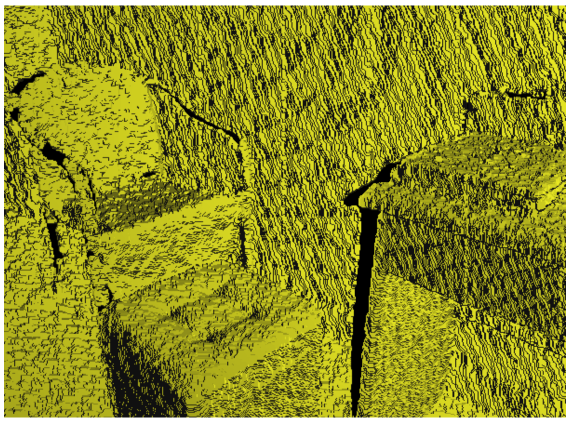


Figure 2 Normal Map - no bilateral filtering

Thus we implement a bilateral filtering[2] which is used to smooth the depth image and remove noise while still preserving edges. The details of this algorithm shows on this Web Page.

2.1 Describe

The main numerical formula is as following:

$$g(x,y)=\frac{\sum_{i,j} D(x,y)w(x,y,i,j)}{\sum_{i,j} w(x,y,i,j)}$$

Which the $w(x,y,i,j)$ is

$$w(x,y,i,j)=\exp(-\frac{(x-i)^2+(y-j)^2}{2\sigma_d^2}-\frac{\left\|D(x,y)-D(i,j)\right\|^2}{2\sigma_r^2})$$

Function $D(x,y)$ is the depth value of the pixel position (x, y). Function c returns a weight based on the distance from the center of the filter. Function returns a weight based on the similarities of the two depth values. Pixels that are closest to the center of the filter and are similar to the center depth value receive a higher weight. The (i, j) pairs indicate the filter radius. The filter radius is 1which means the rang of (i, j) is [-1, 0, 1]. Thus the center (x, y) has 8 neighbors.

Because we know the range of Euclidean distance $(x-i)^2+(y-j)^2$ .To accelerate computing, the Gaussian kernel for Euclidean distance between center and neighbors can be pre-computed. When we use them, we just look up the data by neighbor ID.

[2] C. Tomasi and R. Manduchi, "Bilateral Filtering for Gray and Color Images", *Proceedings of the 1998 IEEE International Conference on Computer Vision, Bombay, India.*

2.2 Scale

The pre-processing image is 640x480x1=307200. We create 16x16 threads per block and 40x30 blocks to parallelize filtering.

To accelerate the reference of the neighbors, we adopt the texture memory. The filtering can benefit from textures because calculations are generally performed in pixels where local weighted sum are considered, and neighbor pixels need to access each other's depth value. A spatially local cache works better for this than a simple linear memory cache. What's more, we'd like to run the filter k times.

2.3 Performance

The table 1 shows the runtime of bilateral filtering.

	1	**2**	**3**	**4**	**5**
Iterantions_num	1	1	1	2	2
Guassian_delta	10	5	10	10	10
Euclidean_delta	11	20	11	11	11
Filter_radius	1(9 cells)	1(9 cells)	2(25 cells)	1(9 cells)	2(25 cells)
Run Time (GPU)/ms	**18.71**	**18.66**	**49.01**	**37.41**	**98**
Run Time (CPU)/ms	**23**	**23**	**54**	**47**	**112**

Table 1 Run time of bilateral filtering

The figure 3 shows the result by choosing different parameters of filtering**.** The image order is the same as the table.

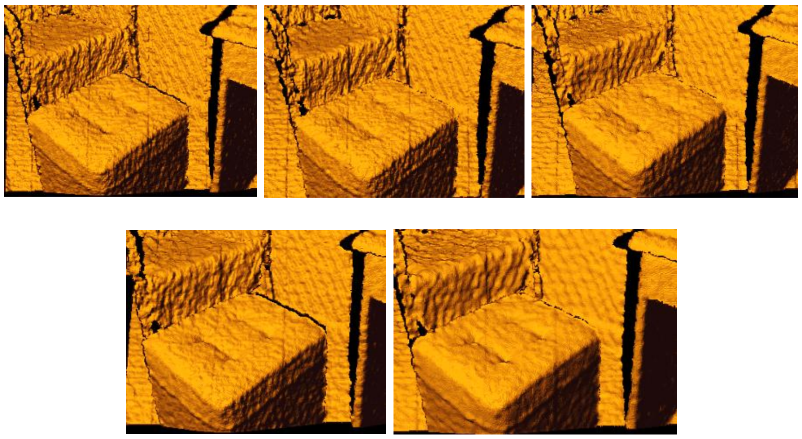


Figure 3 bilateral filtering process results

3. Measurement compute - surface vertex and normal map (Jiakai Zhang)

Given the intrinsic calibration matrix K (of the Kinect infrared camera), a specific depth measurement is projected as a 3D vertex in the camera's coordinate space as follows;

$$v(\mathbf{u}) = D(\mathbf{u})\mathbf{K}^{-1}(\mathbf{u},1), \mathbf{u} = (x, y)$$

The K matrix should be like $K = \begin{bmatrix} 1/f & 0 & 0 \\ 0 & 1/f & 0 \\ 0 & 0 & 1 \end{bmatrix}$, but actually in case of the specific Kinect camera, the projected matrix K should be revised. Thus the final 3D position of vertices is calculated by the following equation:

$$\begin{cases} v.x = (u.x - c.x) * D(\mathbf{u}) / f.x \\ v.y = (u.y - c.y) * D(\mathbf{u}) / f.y \\ v.z = D(\mathbf{u}) \end{cases}$$

By using neighboring re-projected vertices, we can calculate the normal vectors.

$$v(\mathbf{u}) = dv_i(\mathbf{u}) \times dv_j(\mathbf{u})$$

The process time of surface vertex (640x480x1=307200) and normal map (640x480x1=307200) computing via **GPU** is 1.8ms, which that via **CPU** is 55ms.

4. Coloring the vertices and Phong shading (Jiakai Zhang)

By using the RGB image from the Kinect, we set the color of all vertices and implemented the Phong shading. The following figure 4 shows the effect of different light positions and colors. The main problem is to find the corresponding color value for each vertex. It can be seen as camera calibration problem. The figure 4 shows the result of coloring and Phong shading.

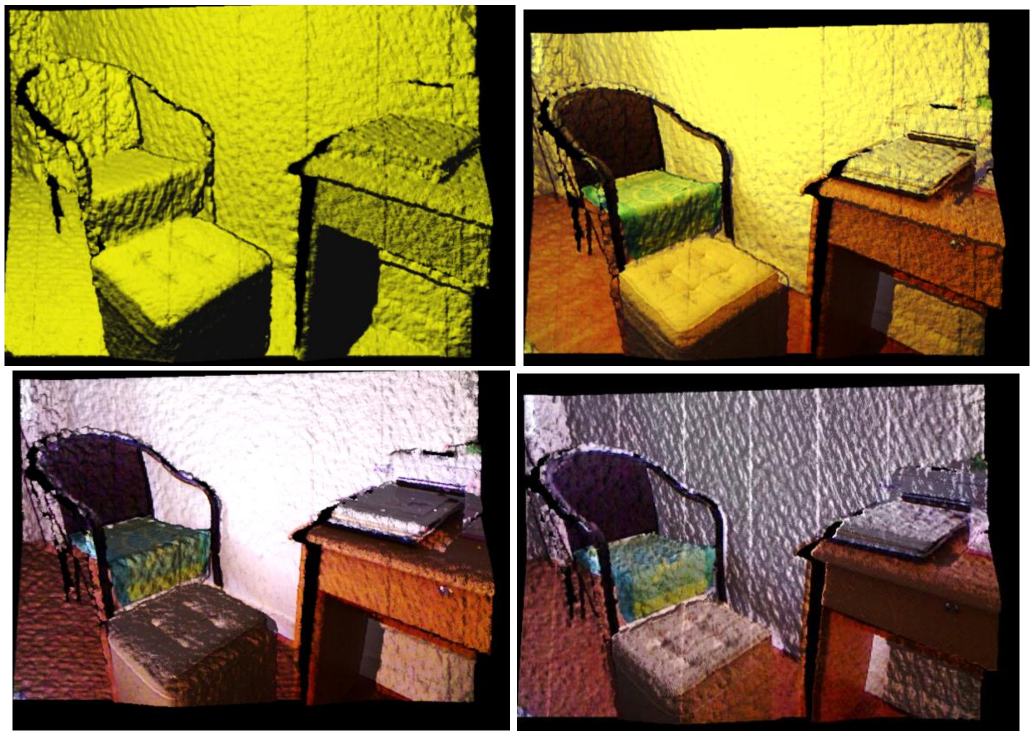


Figure 4

Even though the OpenGL programming is not a parallel part, it's very useful for debugging and data visualization. And it's also time-consuming to implement. More specifically, it contains a virtual trackball implantation, lighting, vertices object buffer for rendering, and CUDA graphics map resources and so on.

5. Camera Pose Estimation – ICP (Yu Xu, Jiakai Zhang, Hao Liu)

The input of ICP is the consecutive cloud points and normal vectors in different frames. The output is the 6DOF transformation matrix T which indicates the pose of camera.

5.1 Describe

Iterative closet Point Algorithm has multiple versions. According to the features of Kinect Data which is small differences between consecutive frames and the available data of normal vectors, the projective data association is better to find the correspondence points (ICP)[3]. The details are as following:

We set the first frame correspond to the global coordinates so other frames should align to the first frame. So we align points of the kth frame to points of (k-1)th frame, align points of the (k-1)th frame to points of the (k-2)th frame….align points of the 2nd frame to points of 1st frame. Then every frame will be aligned to the first frame.

In order to align two frames, we need to find corresponding points between two frames. We need a rule to find corresponding points so that the energy function converges well. Once we have found corresponding points, we need to generate a transform matrix by these points. After finding the corresponding oriented points, we should minimize the point-to-plane error metric (Energy Function)[4], defined as the sum of squared distances between each point in the current frame and the tangent plane at its corresponding point in the previous frame.

$$E(T_{g,k}) = \sum_{\Omega(\mathbf{u})\neq null} \left\| (T_{g,k} V_k(\mathbf{u}) - \hat{V}_{k-1}(\hat{\mathbf{u}}))^T \hat{N}_{k-1}(\hat{\mathbf{u}}) \right\|_2$$

So we encounter a nonlinear least square problem. We should transform this problem to linear least square problem by approximation so that we can solve it quickly and update our data real time. The contradiction is if we choose all points to solve least square problem by SVD, it will cost a lot of time so that we cannot do "real time" updating our data. If we choose fewer points, we can solve least square problem fast but we cannot get satisfactory transform matrix to align two frames well. How to make it? We need find a balance. As the optimized method mentioned in [4], it can solve as a 6-by-n linear system using SVD algorithm (solve on CPU).

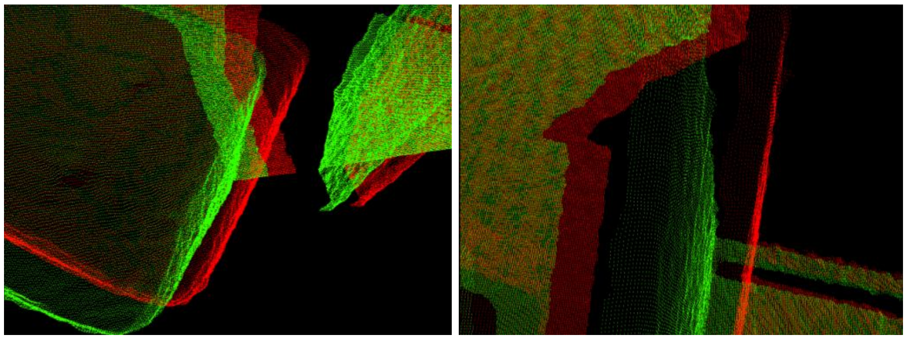
5.2 Scale

We need to check 640x480=307200 pairs of points and solve 6-by-n linear system, which n is around 2000.

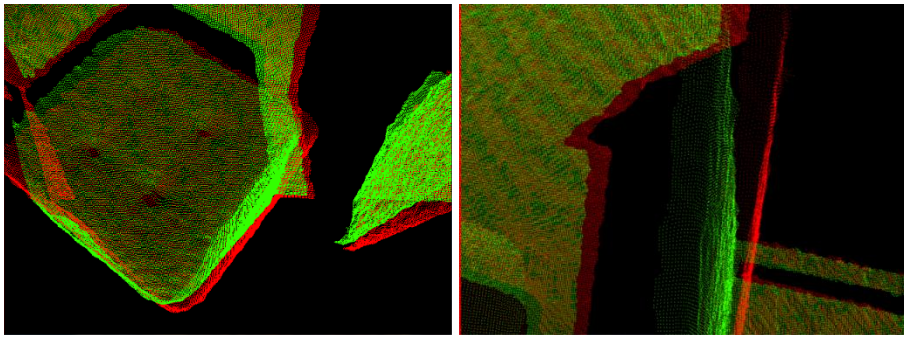
[3] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. *3D Digital Imaging and Modeling, Int. Conf., 2001*
[4] K. Low. Linear least-squares optimization for point-toplane ICP surface registration. *Technical report, TR04- 004, University of North Carolina, 2004*

5.3 Performance

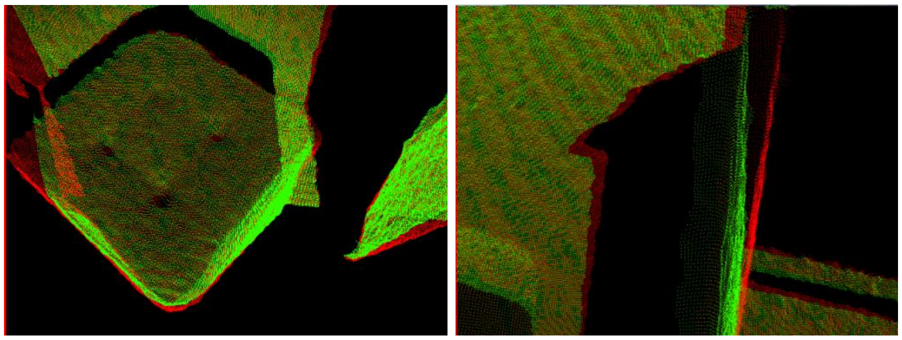
The figure shows the results of ICP. The two images are obtained from two different viewports but the same scene.



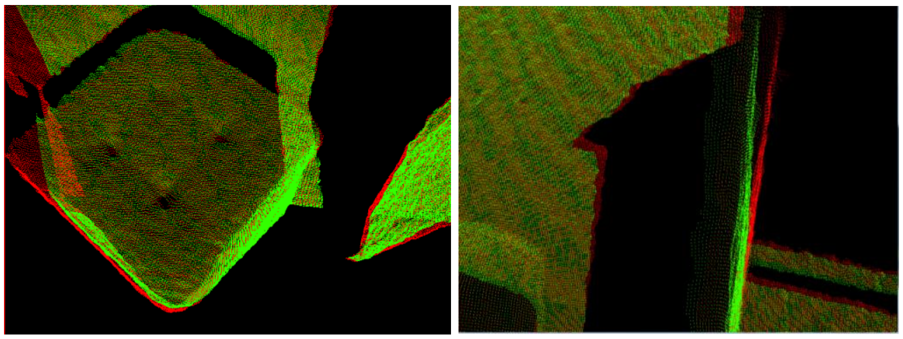
Before ICP (Error: 1102.125)

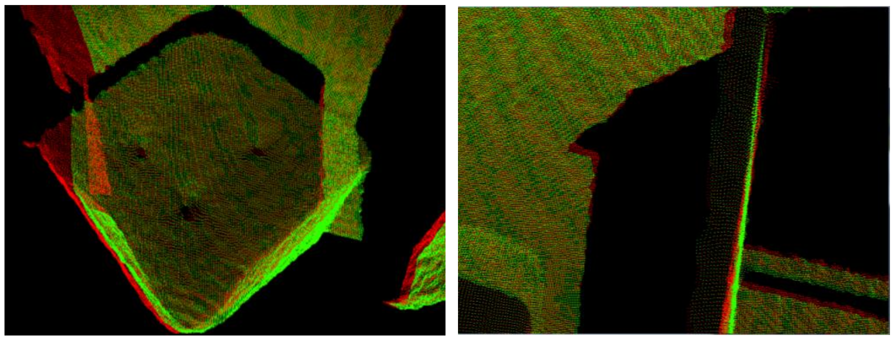


The 1st iteration (**Error** 238.798 **Time on CPU/GPU**: 8ms/2 ms)



The 2nd iteration (**Error** 65.941 **Time on CPU/GPU**: 5ms/1ms)


The 3rd iteration (Error 39.241 **Time on CPU/GPU**: 6ms/2ms)


The 4th iteration (Error 13.628 **Time on CPU/GPU**: 7ms/2ms)

6. Update reconstruction – TSDF (Hao Liu, Jiakai Zhang)

Once we know the position and rotation relations between frames, we can use TSDF to merge all frame depth map into one. Here we use truncated signed distance function (TSDF) to save merged data. TSDF actually a 3d tensor or I call it a cube, which represents the space we are measuring. The value of each volume in the cube is the distance to closest surface. And this distance is signed and truncated. If the volume is behind the surface in the view of camera, then we set distance a negative value. If the distance between volume and surface is too long, then we set the distance equal to 1 or -1. We use truncation to efficiently get parallel surfaces.

6.1 Describe

How to get TSDF cube?

(a) Compute local 'cube': for each frame we compute a 'cube' only from the depth map of this frame. Actually, we compute this cube but not save it, compute one volume and then update it into global cube. For each volume we find the corresponding point in the depth and use depth value as the position of nearest surface.

(b) Update the global TSDF cube: when we obtained the local 'cube', we need to merge this cube into global cube. We use weight merging instead of directly merging. We use two kind of weights: (i) weights of different volumes in local cube are different, the volume near and in front of the camera have high weight. (ii) weights of global cube and local cube are different, local cube has low weight to avoid noise.

Here is the mathematic formula to compute local cube:

$$C_{local}(p) = \Psi\left(\lambda^{-1}\left|\left|t_{g,k} - p\right|\right|_2 - R_k(x)\right)$$

$$\lambda = ||K^{-1}\dot{x}||_2$$

$$x = \lfloor \pi(KT_{g,k}{}^{-1}p) \rfloor$$

$$\Psi(\eta)\begin{cases}\min\left(1, \dfrac{\eta}{\mu}\right) * (-1), if\ \eta \geq -\mu \\ \qquad null, otherwise\end{cases}$$

6.2 Scale

The size of TSDF cube is 256x256x256=16.7M as the same as the size of cube weight.

Difficulties in TSDF?

(a) The size of TSDF cube: we should trade between accuracy and GPU memory. The cube is a 3D tensor, so its size increase very fast when we want to finer grid in our space.

(b) How to set the value of weight. This weight is very important in our TSDF merging process. A good weight should be corresponding to the relative position of volume and camera and in the same time be truncated for updating possible.

(c) It's very difficult to debug, you cannot efficiently to output your result even in CPU. We use Mathematics to output our zero contour surface of cube.

6.3 Performance

Threads/Block	**8,8**	**8,8**	**16,16**	**16,16**	**32,32**	**32,32**
Size of Cube	128,128,128	256,256,256	128,128,128	256,256,256	128,128,128	256,256,256
GPU Time/ms	**19.1**	**179.2**	**17.3**	**142.49**	**20.03**	**168.97**

7. Surface prediction – Ray casting[5] (Jiakai Zhang, Hao Liu)

After TSDF updating, we have the TSDF cube.

7.1 Describe

We'd like to choose the particular camera position to cast ray to the volume of the TSDF cube. If we find the sign of the TSDF value changes, it means we find a point on the surface. And we calculate the normal vector by calculating the gradient of TSDF at this point. The formula is as follow:

$$R_{g,k}\hat{N}_k = \hat{N}^g_k(\mathbf{u}) = \eta(\nabla F(\mathbf{p})), \nabla F = [\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}]^T$$

[5] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive ray tracing for isosurface rendering. *In proceedings of Visualization, 1998. 2.4, 3.4, 3.4*

Some tricks in Ray casting:

(a) Two scales acceleration: Because our TSDF cube is truncated in distance, then we actually can use μ as the ray casting step-size. And once we find the signed change, we change our step-size to our cube precision and find the more accurate surface.

(b) Linear prediction: when we are on the signed change point in cube, instead of using the coordinate of the volume, we can use two volumes to do a linear regression to predict the surface position.

7.2 Performance

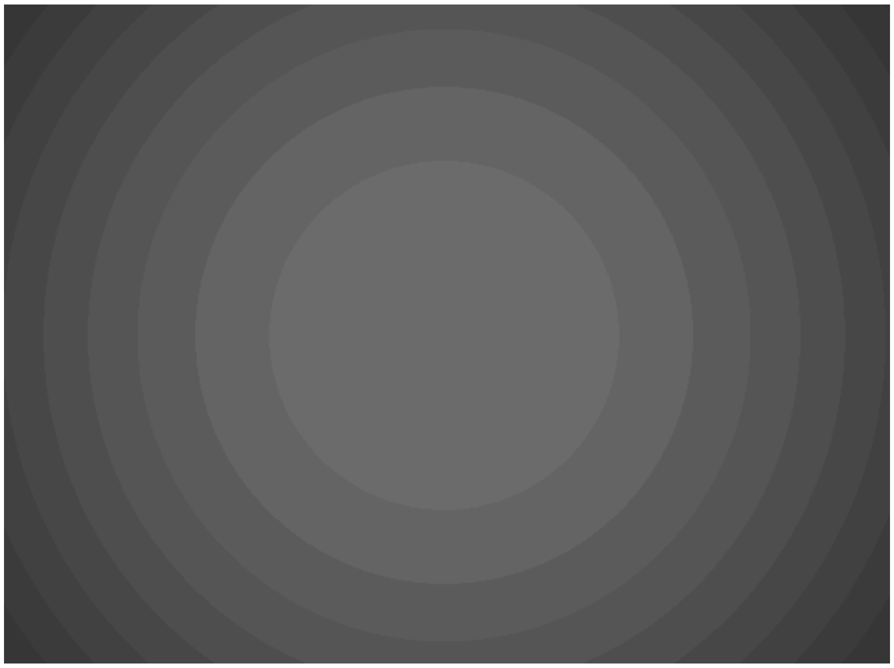


Figure 5 Steps in Ray tracing

Figure 5 shows that ray marching steps are drastically reduced by skipping empty space according to the minimum truncation μ (light color equals to 10 iterations and dark 50 = 6-8 speedup). Marching steps can be seen to increase around the surface interface where the signed distance function has not been truncated.

Threads/Block	**8,8**	**8,8**	**16,16**	**16,16**	**16,16**	**16,16**	**32,32**	**32,32**
Size of Cube	128*3	256*3	128*3	128*3	256*3	256*3	128*3	256*3
Truncated	Yes	Yes	Yes	No	Yes	No	Yes	Yes
GPU Time/ms	**16.1**	**135.5**	**12.84**	**127.6**	**101.2**	**521.2**	**16.8**	**145.3**

8. Intergrade the whole pipeline (Jiakai Zhang, Hao Liu, Yu Xu)

Currently, we are trying to integrate different parts. It's the most time-consuming part in our project. We'd like to finish it by

9. Code release

We’d like to keep the code private. Because we use CUDA language and Kinect, you need to have a NVDIA graph card and Xbox Kinect to run the code. In this project, we use the NVDIA GTX 460M graph card.

How to set up the environment?

9.1 OpenNI SDK for Kinect

Download and install the OpenNI SDK from the link. Do not plug in the Kinect before finish installing.

9.2 OpenCV SDK

Download and install the OpenCV SDK from the link.

9.3 OpenGL SDK

Download and install the OpenGL SDK (include glew) from the link.

9.4 CUDA SDK

Download and install the OpenGL SDK from the link.