Solid Environment Reconstruction on the GPU

Rouslan Dimitrov

May 15, 2007

Senior year research project at Jacobs University Bremen
as a formal requirement for a Bachelor degree in
Electrical Engineering and Computer Science

Abstract

Solid environment reconstruction performed entirely on the GPU is a relatively unexplored topic as of the date of writing. This paper presents a multi-pass algorithm that converts raw data from a range sensor or a stereo camera into a quad-mesh. A threshold parameter that governs the allowed deviation from the sample points is introduced. In this way, the number of generated faces can loosely be controlled and changed dynamically. The algorithm keeps a quad-tree as an internal data structure, which results in generation of square faces only. Although the face count can be minimized by using more complex polygons, this is of little use in practice as typically the solid environment is used for collision detection and such systems are very efficient for quadrilateral surfaces.

Acknowledgments

Special thanks to Gernot Ziegler for being my valuable friend and mentoring my progress in graphics programming.
A warm thank you to Galina Stefanova and Mariana Rashkova for offering me moral support throughout both my difficult and happy moments.

1 Introduction

The goal of the presented reconstruction algorithm is to build a solid environment as a quad-mesh from a depth-map. It is developed specifically for execution on a programmable GPU that supports a form of feedback (eg. frame buffer objects), floating point textures and branching.

There are several pre-processing stages that build a point in 3D with an associated normal for every pixel of the input depth-map. Using the implicit connectivity information of the depth-map, smooth quadrilateral faces are generated in image space and then transformed into local 3D space.

A likely application of such reconstruction algorithms is in mobile robots that need to maintain a model of the surroundings, which is to be used for path-finding, for example. Because they have a limited supply of energy and a power-efficient computer, any algorithm used should run in minimal processing time. The presented algorithm takes altogether a few milliseconds on a low-end graphics card, while completely freeing the main CPU for other tasks.

2 Related Work

Our algorithm is related to surface simplification methods because the input data can be triangulated and then simplified. There are many well-known such methods that are developed for the CPU. Garland et al [3] give a detailed overview of available surface simplification methods and the ones related to our goal generally fall into five categories.

2.1 Surface Simplification

Vertex Decimation Soucy et al. [7] describe a method to iteratively remove a vertex from the generated mesh with high efficiency. In general, however, it is difficult to parallelize such an algorithm, as after a removal of a vertex, new triangles are generated and a metric is updated based on which the next vertex is chosen. Another drawback of this class of algorithms for our purpose is that vertex decimation preserves topology, while we are interested in merging closely spaced regions.

Edge Contraction Garland et al. [3] propose an algorithm based on edge contraction that simplifies a given mesh by iteratively removing edges. Although their approach is serial in nature, too, it has the benefit of merging initially unconnected regions. Parallelization on local domains at the cost of reduced quality seems possible, but will result in doing locally optimal rather than global edge selection. The decision is based on additional information stored per vertex and the process of edge contraction after an edge is selected is mostly a local operation.

Vertex Clustering This class of algorithms relies on finding a bounding box around the model and assigning its vertices on a grid. Then each cluster gives a single vertex, whose position is determined based only on the vertices inside that cluster. An octree implementation can typically improve performance. This method is easily parallelizable and will be thoroughly investigated. A drawback [3] is loose error bounds and lack of control.

Surface Sampling [1] Ziegler et al. propose a novel algorithm for sampling a surface and generating a point cloud from it. Their algorithm runs entirely on the GPU and is similar to vertex clustering. Again, a grid is built within the bounding box of the model, but it's filled not with vertices, but with a sample if the given surface passes through the volume of the cluster. This algorithm is very similar to the one proposed.

Probability Liu et al [6] solve the surface simplification problem with a probabilistic approach. They iteratively minimize an error metric, which is based on the assumption that the distance between a sample and a plane follows a Gaussian distribution. Then, a model is iteratively refined by adding or removing planes. Although their findings produce very good results, the running time of the algorithm is far from real time.

All of the above mentioned algorithms were carefully investigated, but none of them lends itself directly to a reasonable port to the GPU. Their serial nature allows for parallelization of local space domains, which in turn leads to complications and cache misuse. In addition, they solve a more difficult problem by operating on arbitrary meshes, while we can assume a regular one made out of quads.

However, important ideas are gathered from the above descriptions - the proposed algorithm uses data clustering and maintains a loose error metric, which governs how surfaces are split. It makes local decisions in parallel and is based on image processing techniques. Probabilistic approaches were not considered.

2.2 Image Processing

The problem of quickly finding a rough solid representation of the depth map can potentially be solved entirely with image processing techniques.

Region Detection Ziegler et al. [2] propose an algorithm for detecting regions running solely on the GPU. Although the original application is the detection of empty regions in sparse matrices in PDE solvers, the algorithm suits our needs as well. The performance is real-time, and although the quads will have a side length of power of 2, it offers a nice trade off.

A modification of this algorithm is used to detect flat surfaces in image space, which are then transformed to robot space in 3D.

3 Overview

The main idea of our method is to operate mainly on generated surface normals because smooth surfaces have nearly parallel normals. As normals start to deviate more than the allowed threshold from a current average, the considered region is split.

The proposed algorithm has multiple interconnected but independent processing stages. They can be grouped into six main classes:

Filter input noise, deal with uncertainties and reduce the size of the input data, so that later stages run faster with little loss of quality.

Transform the input data from various sensors into a well defined form - a point-list texture that maps every depth value to a point in 3D.

Convert Convert the point-list texture to a normal map + z channel.

Build the quad-tree and detect flat surfaces.

Locate faces in image space by traversing the tree.

Project the image space regions into local 3D space.

3.1 Filter and Reduce Size

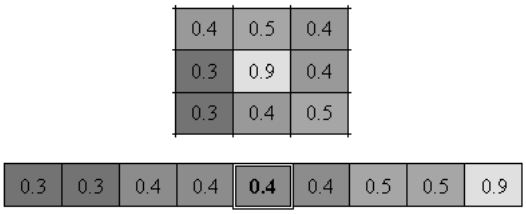


Figure 1: Filtering 3x3 portion of the depth map

The input data is passed on to a median filter with a 3×3 kernel. In addition, every 3×3 pixels contribute to one pixel of the final image. This in effect will reduce the depth map size by a factor of 9 and is expected to significantly reduce noise and erroneous pixels, which typically occur in sensor data.

Consider the 3×3 subimage of the input depth map shown in figure 1. After the median filtering step, we will replace all 9 pixels with just one. The middle sample of depth 0.9 is erroneous since it is very unlikely that there is such a small hole in the object. The first step is to sort in ascending order the 9 depth samples as shown in figure 1. Then the middle one (hence the term median filtering) is chosen as a representative for the whole region.

In the implementation, an unrolled bubble sort is used to determine the five smallest samples, out of which the biggest one is chosen as output.

3.2 Transform

This step is sensor-specific, though most sensors and simulators should fall into two general categories. To ease the discussion, let's consider a laser sensor that has 2 parameters - θ and ϕ, which specify the angle which the laser makes with the local horizontal and vertical axes. The sensor uses an interferometer to determine the distance to the object that reflects back the laser beam. Note that in this way, a straight wall in front of the robot will appear spherical in the depth-map.

There are two straightforward ways to control the parameters:

- Fill a pixel at normalized coordinates (θ, ϕ) of single channel texture with the currently measured depth. In other words, the sensor offsets the parameters by $\Delta\theta$ for each $n \cdot \Delta\phi, n = 1, 2, 3...$

- Another way to alter the parameters is in a ray-tracer-like manner, where θ and ϕ are varied non-uniformly. Instead, imagine a grid, a distance p from the sensor. Then the length of every $\overrightarrow{op}$, $|op|$ is stored at the grid intersection which $\overrightarrow{op}$ crosses. See figure 2.

To make the problem concrete, we need a couple of parameters:

- θ_{max} - the field of view, or the maximum deviation in radians the sensor makes.
- res - the resolution of the texture/grid used as output of the sensor and input of the current stage.

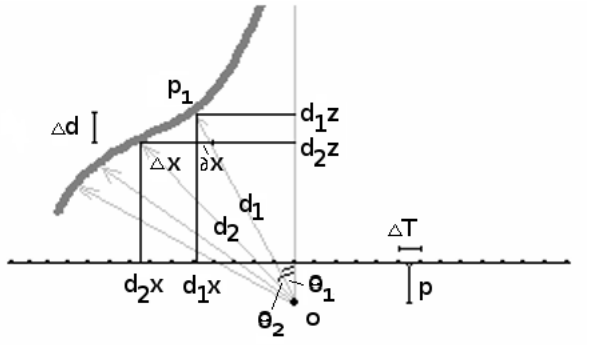


Figure 2: Transformation

As seen in figure 2, we would like to store the coordinates of every p_i in robot space. To simplify the problem, let's work in 2D - the concept is easily extended to 3D. By knowing θ_i then:

$$\begin{aligned} d_{iz} &= d_i \cos(\theta_i) \\ d_{ix} &= -d_i \sin(\theta_i) \end{aligned} \tag{1}$$

Then, (d_{ix}, d_{iz}) are stored in the (1-dimensional, in this case) array-like container, at position i. This step is important, as we locate the flat faces in depth-map space, in terms of i, and then need to transform them to robot space.

Finding θ_i is a little involved in the ray-tracer-like case:

- const $\Delta\theta$

$$\theta_i = \frac{\theta_{max}}{res} i$$

- varying $\Delta\theta$

$$\theta_i = \arctan(\frac{i \cdot \Delta T}{p}) \tag{2}$$

where p is the distance from the grid to the local origin. ΔT can be specified as an input parameter, however it has unclear and more importantly

dependent on p physical meaning, so we need to reexpress it in terms of our known parameters. Thus,

$$\begin{aligned} \frac{res}{2}\Delta T &= p\tan(\frac{\theta_{max}}{2}) \\ \Delta T &= \frac{2p}{res}\tan(\frac{\theta_{max}}{2}) \end{aligned} \tag{3}$$

Substituting in (2)

$$\theta_i = \arctan(i \cdot \underbrace{\frac{2}{res} tan(\frac{\theta_{max}}{2})}_{c})$$

Here c is constant and must be computed only once.

Rule 2 is performed for all i, or in the case of a depth-map, for all pixels. Practically, the point list is stored in a 3-channel float texture, where for every $p_{i,j}$ from the input depth-map, a transformed $(P_{x_{i,j}}, P_{y_{i,j}}, P_{z_{i,j}})$ is stored at location (i, j)

3.3 Convert

To find the normal for every input pixel p_i, we need to find the local directional derivatives:

$$\overrightarrow{n_i} = \nabla(p_i) = (S_{x_i}, S_{y_i}, -1) \tag{4}$$

Let's find S_{x_i} in 2D. Although we could use the current pixel and one neighbor, this would produce asymmetric results. Another approach is to judge the local slope on the left and right neighbors of p_i. Thus,

$$S_{x_i} = \frac{\Delta d_{x_i}}{\Delta x_i} \tag{5}$$

If the results from the previous stage are already available, we can compute (5) directly. However, there is a trade off between the number of stages and their complexity. It might be reasonable (depending on the input depth-map resolution) to combine the *transform* and *convert* stages and omit one cycle through the graphics pipeline.[1] In the rest of this sub-section, a stand-alone *convert*-stage is derived.

Let's project the two depths involved in (5) into local 3D space and perform the calculation there:

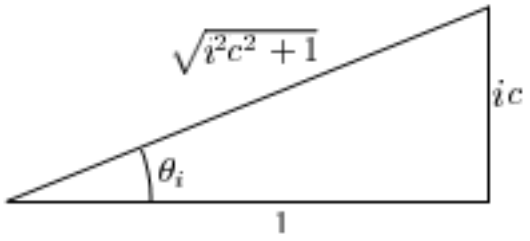
$$S_{x_i} = \frac{d_{i+1}\cos(\theta_{i+1}) - d_{i-1}\cos(\theta_{i-1})}{-d_{i+1}\sin(\theta_{i+1}) + d_{i-1}\sin(\theta_{i-1})} \tag{6}$$

Finding

$$\cos(\theta_i) = \cos(\arctan(i \cdot c))$$

for every input pixel is expensive, so we need to simplify the expression.

[1] In addition, the transformed depth-map might not even be needed for some uses of the algorithm, such as feature detection, rather than surface reconstruction.



Using $\theta_i = \arctan(i \cdot c_i)$, in the figure above, we set the opposite side of θ_i to $i \cdot c$ and the adjacent side to 1. Thus we see that

$$\cos(\theta_i) = \frac{1}{\sqrt{i^2c^2+1}} \tag{7}$$

$$\sin(\theta_i) = \frac{i \cdot c}{\sqrt{i^2c^2+1}} \tag{8}$$

After substituting in (6), we need to normalize (4)

$$\hat{n} = \frac{\overrightarrow{n_i}}{||\overrightarrow{n_i}||}$$

and store it in the output texture.

3.3.1 Approximation

Taking four divisions and four square roots per normal is computationally expensive and if $|\theta_{max}| \ll 1$, equation (6) can be significantly simplified. Assume (figure 2):

- $\Delta x \approx \Delta x + \delta x$
- $\cos(\theta_i) \approx 1$
- $d_i \approx d_{i+1}$

Although the second approximation is true only for very small θ_i, the region detection algorithm is relatively insensitive to the error produced here. Rather than computing the normals based on the projection of the depth onto the z-axis, we take the depth as it is. This results in spherical elongations of $d_{i,j}$ and normals being tilted away from the center of the depth-map.

In the following discussion, we use:

$$\Delta T \approx \frac{\theta_{max} p}{res} \tag{9}$$

Eq. (6) is significantly simplified:

$$S_{x_i} \approx \frac{\Delta d_i}{\Delta x_i} \tag{10}$$

By similar triangles,

$$\frac{\Delta x_i}{d_i} = \frac{\Delta T}{p}$$

$$\Delta x_i = \frac{d_i \Delta T}{p} \tag{11}$$

$$\Delta x_i \approx \frac{\theta_{max} d_i}{res} \tag{12}$$

Substituting in (10) gives,

$$S_{x_i} = \frac{\Delta d_i}{d} \cdot \underbrace{\frac{res}{\theta_{max}}}_{\text{const}} \tag{13}$$

Effectively we reduce the four divisions and square roots to one division.

3.4 Build

After the *transform* and *convert* stages, we have a point list of points in local 3D coordinates with associated normals stored in an array with preserved connectivity information. Assuming continuous surfaces, the 3D neighbors of $p_{i,j}$ are stored in the array at indices $(i-1,j),(i+1,j),(i,j-1),(i,j+1)$, for all $0 < i,j < res$.

Using this property, we build a quad-tree with the following invariant:

Quad-tree invariant 1 *Every node of the quad-tree stores the averaged normal and distance of its four children. In addition, it contains the number of faces detected below it in the tree.*

In this way, the root of the tree contains the total number of faces to be generated. All leaves are chosen to be individual pixels from the normal- and depth maps and their face count is initialized to 1. Pixel (i,j) is grouped with pixels $(i+1,j),(i,j+1),(i+1,j+1)$ and all leaves are unique. Figure 3 depicts this visually.

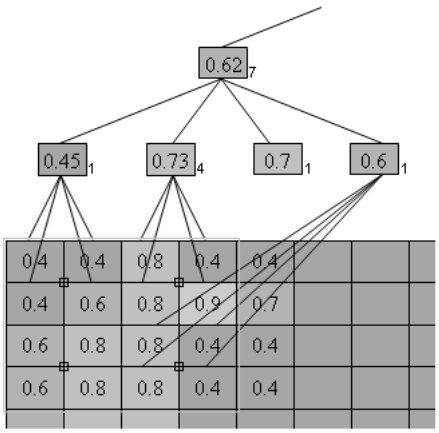


Figure 3: Part of the generated quad-tree

There are several ways of how to determine whether the children of a node belong to one region. Here, we use a decision step based on a computationally friendly pseudo-error metric *threshold*, modeled after Ziegler et al. [2]. There are two necessary conditions for merging regions:

- Each children contains exactly 1 region
- Let $\overrightarrow{n}_{avg}$ and d_{avg} be the averaged normal and distance of the children. Then, they form one region iff for every child i,

$$\overrightarrow{n}_{avg} - \overrightarrow{n}_i < \overrightarrow{n}_{threshold} \tag{14}$$

$$d_{avd} - d_i < d_{threshold} \tag{15}$$

Equation (15) is only needed if the depth map contains two parallel surfaces separated by distance greater than $d_{threshold}/2$ and we want to depict a sharp discontinuity at their edges. This criterion, however, increases face count in surfaces nearly parallel to the viewing direction and is not taken into account any further.

$\overrightarrow{n}_{threshold}$ can be roughly thought of as a measure of how much the normals within one region are allowed to vary from one-another. Thus the parameter has a direct impact on the number of faces generated. Typically, we need the same splitting criteria in every direction, so we let $\overrightarrow{n}_{threshold} = (n_t, n_t, n_t)$. Next to each node in figure 3, the number of regions below it is shown for $n_t = 0.2$. In this case the 16 depicted pixels contribute to 7 regions.

The following special cases should clarify the idea:

- *if* $n_t = 1$, there is only one face generated
- *if* $n_t = 0$, there are $res \times res$ faces generated

We note that the height of the tree is $\log_2 res$ and at depth i, there are 4^i nodes. Practically, the tree is stored in a mipmap fashion as known in computer graphics and is generated in bottom-up manner (figure 4).

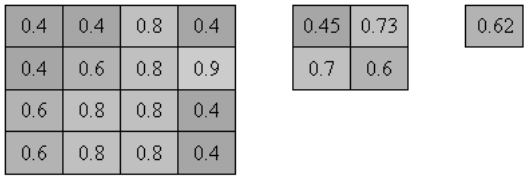


Figure 4: The quad-tree stored in mipmap fashion

3.5 Locate

After the quad-tree is generated it contains all the necessary information to locate the square regions in image space. Each region is defined by four vertices $v_k = (i_k, j_k),\ k = 1 \ldots 4$, where (i_k, j_k) are texture coordinates in the depth-map. While traversing the tree, however, keeping track of one vertex and a side length is more efficient. After a region is located with a bottom-left corner at (i, j) and side length a, assuming it is in quadrant I, the other vertices:

$$\begin{aligned} v_0 &= (i, j) \\ v_1 &= (i + a, j) \\ v_2 &= (i + a, j + a) \\ v_3 &= (i, j + a) \end{aligned}$$

The root of the tree contains the number of regions n. The tree is traversed n times and each traversal is assigned an index $i = 1 \dots n$. Let's give every region an identifier - $id = 1 \dots n$ The basic idea is that every node is assigned a range of ids that are below it. Thus, the range of the root is $1 \dots n$ and the range of the other nodes is more confined. Nodes that have one region below (or equivalently, range of 1) represent the quads that are saved.

The following C-like code illustrates the idea. Initially, `range_low = 0`, `depth = 0` and the routine is called once for every `index` $= 1 \dots n$. The last line

```
proc locate(int index, int range_low, int depth, ...)
{
   if this node has 1 child
   {
      region.side_length = res / 2^(depth);
      store region;
      stop;
   }
   for each child c
   {
      if range_low <= index < rangle_low + c.regions_below
          locate(index, range_low, depth+1, ...);
      else
          range_low = range_low + c.regions_below;
   }
   error "tree is inconsistent"
}
```

of code should never be reached because it signals that the quad-tree invariant is violated.

For the sake of clarity, determining the bottom-left corner of the current region is omitted from the code. It is a 2D vector $\overrightarrow{v}_{bl}$ passed as fourth argument, which is initially $(0, 0)$. Depending on which child we recurse into, $\overrightarrow{v}_{bl}$ is modified differently. There are four cases:

node of recursion	new $\overrightarrow{v}_{bl}$ passed
bottom-left node	$\overrightarrow{v}_{bl}$
bottom-right node	$\overrightarrow{v}_{bl} + (a, 0)$
top-left node	$\overrightarrow{v}_{bl} + (0, a)$
top-right node	$\overrightarrow{v}_{bl} + (a, a)$

where $a = res/2^{depth}$ is half of the size of the rectangle the current node covers.

In practice, the algorithm is implemented non-recursively for speed gains even if the target hardware supports recursion. The implementation of the increment of $\overrightarrow{v}_{bl}$ should not use branching because if-else statements are expensive on current graphics processors. Instead a constant array of four vectors $offset[4] = \{(0,0), (1,0), (0,1), (1,1)\}$ is used and is indexed with the child index $c \in [0 \dots 3]$. Then $\overrightarrow{v}_{bl} + a \cdot offset[c]$ is passed as the new bottom-left corner.

In addition, before storing the region in the code listing, all vertices $\overrightarrow{v}_k = \overrightarrow{v}_{bl}$+`region.side_length`$\cdot offset[k],\ k = 0 \dots 3$ need to be computed.

3.6 Project

After the previous stage, the vertex coordinates of the quads are known in terms of texture coordinates (i, j) in the depth-map texture. We use the map $T : \mathbf{R}^2 \rightarrow \mathbf{R}^3$ between every pixel in the depth-map and its 3D projection in local space. The T operator is rather simple - it does a lookup at position (i, j) in the point-list created in the *transform* stage. The result is a point in 3D that is used as a vertex of the projected quad.

It should be noted that at this point, the determined regions reside in the graphics card memory. They can be either downloaded to the CPU or directly used for visualization (using vertex buffer objects, for example).

4 Results and Discussion

To demonstrate the developed algorithm, a sample application in C using OpenGL was implemented. The program uses frame buffer objects which capture the output of a stage into a texture and the same texture is fed as input to the next stage. There is an additional stage that renders the generated surface on the screen, using vertex buffer objects. Data is transferred to the graphics card only once at initialization and never sent back. All shaders are written in GLSL and the program is optimized by using the latest OpenGL extensions that are supported on the utilized NVIDIA GeForce 6200. If-else statements are eliminated wherever possible by using interpolating polynomials, constant offset arrays or conditional writes.

4.1 Normals Generation

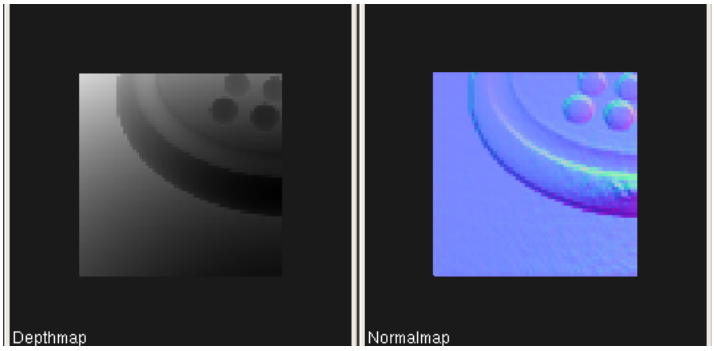


Figure 5: Input depth-map and corresponding normal-map

On the left side of figure 5 is the sample depth-map[2] that was extensively used during development. It shows a button laying on a flat surface. Because the button is round and has some fine detail (four semi-spheres) as can be seen in the upper-right part of the image, it is hard to depict it accurately with the axis-aligned squares that are inherent for the algorithm proposed. Thus, in a sense it is a worse-than-average scenario.

[2]Texture reference: Cristóbal Vila. http://www.etereaestudios.com/training_img/solitario/solitario_en_01.htm

On the right, the generated color-coded normals can be seen[3]. For visualization, the following direct mapping was used[4]:

$$(R, G, B) = (n_x, n_y, n_z) \tag{16}$$

Note that the spherical distortion of the depth-map is removed (in the *transform* stage) and the flat surface is covered with approximately the same normal color. The blockness on the bottom-right can be attributed to low precision for nearby objects (dark depth-map colors).

4.2 Overall Performance

Figures 7 and 8 show 3D reconstructions of the depth map for several values of *threshold*.

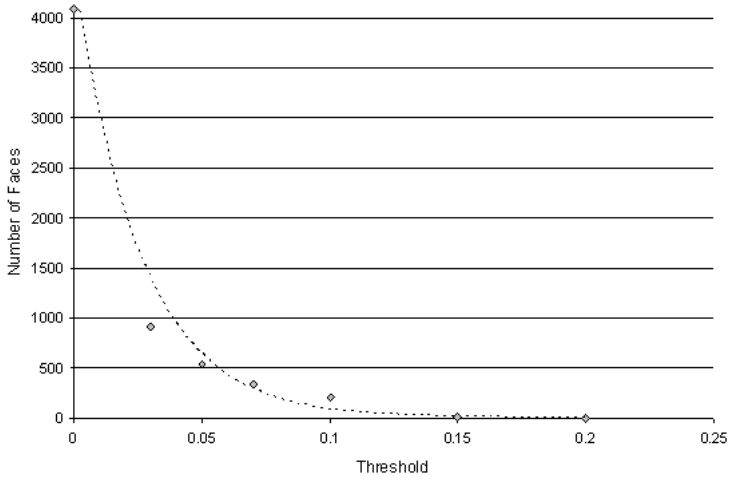


Figure 6: Effect of *threshold* on the face count

The *threshold* is very important for the degree of subdivision. Figure 6 shows the exact relationship for the current depth-map. The equation of the fitting line is of the form $y(x) = e^{ax},\ a \ll 0$ and clearly shows an exponential relationship.

4.3 Median Filter and Reduction

Median filtering is a well known filtering method and descriptions can be found elsewhere. For our purposes, it effectively reduces isolated pixels that deviate from their surrounding ones. Figure 9 shows how after filtering and reduction, the writing in black is completely removed[5]. This, however, is ideal case as the writing has width of at most 2 pixels most of the time. In some cases, median

[3]Steps in-between: transform the depth-map (left) to local 3D space, assign a normal to each point, aggregate the normals to the normal map (right).

[4]The typical $(R, G, B) = 0.5 \times (n_x, n_y, n_z) + (0.5, 0.5, 0.5)$ was not used in favor of better contrast.

[5]Actually, the left image is the original, sent through a low-pass, which is typical for reducing high-frequency (single-pixel, in our case) noise.

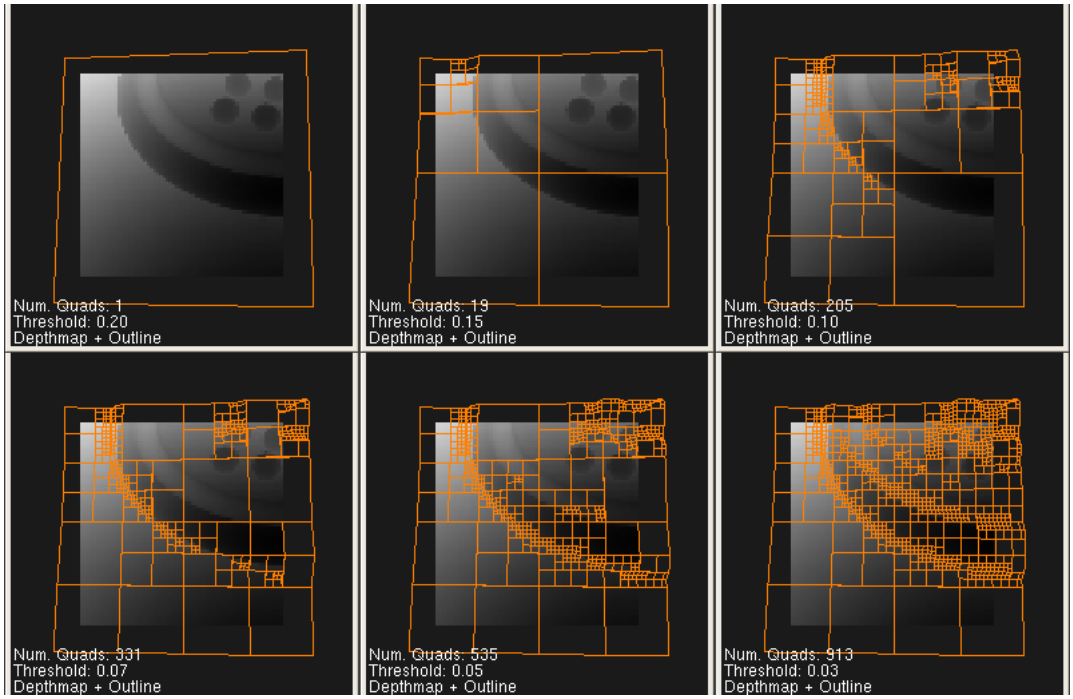


Figure 7: Reconstructed surface, front view

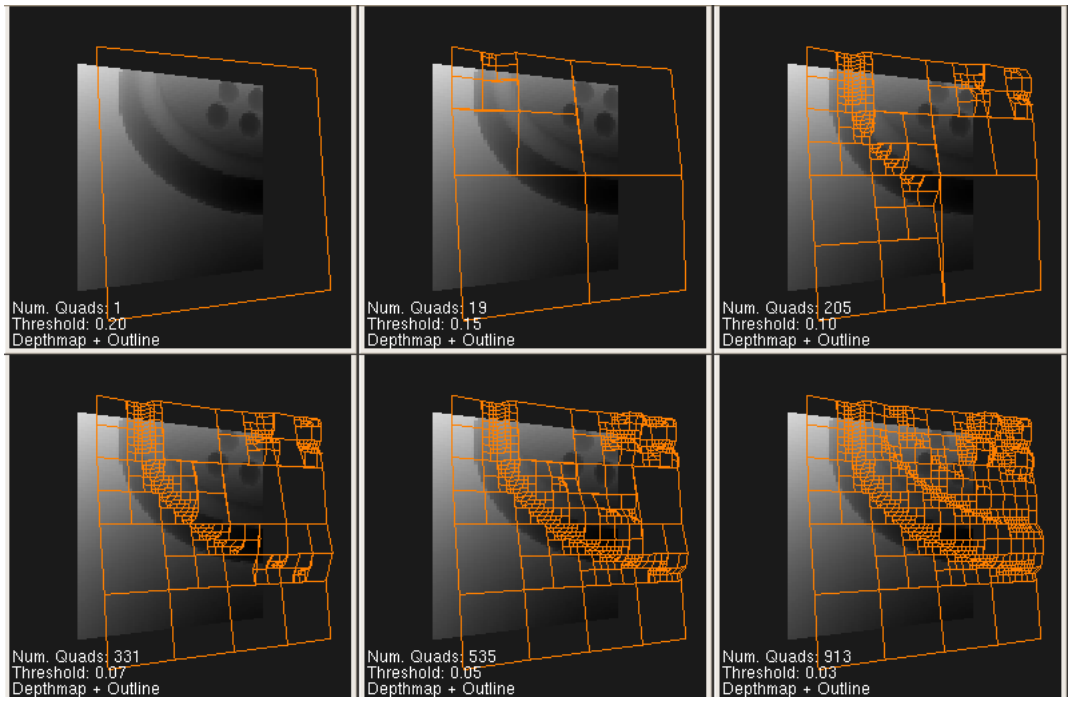


Figure 8: Reconstructed surface, at 25^o degrees

filtering introduces errors, but they typically can be neglected. For example, consider looking at an edge of a cube. If median filtering is performed on such a scene, the edge will be rounded off. Sensors, however, typically introduce relatively greater errors, so this effect can be neglected.

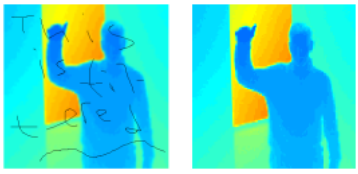


Figure 9: Median filtering on an image: source (left) and output (right)

4.4 Running Time

The complete running time of computing each surface in firgure 7 is shown below.[6] The input depth-map is 192×192 pixels and is reduced to 64×64 after the *filter and reduce* stage. The AGP transfer is not included in the timing.

Threshold	0.20	0.15	0.10	0.07	0.05	0.03
Time [ms]	2	3	4	4	5	7

4.5 Approximation

Section 3.3.1 presents a simplification of the algorithm, which pays off particularly if the *transform* stage is omitted. This means that the reconstructed surface is built by perturbing a sphere rather than a flat surface. Figure 10 shows a comparison with the complete algorithm.

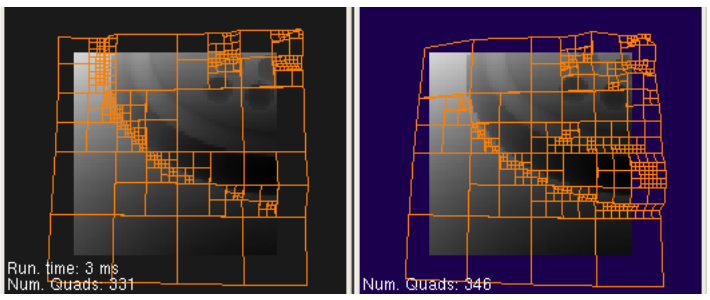


Figure 10: Comparison between the complete algorithm (left) and the approximation (right)

5 Future Work

The developed algorithm has three major weak spots:

[6] Tested on NVIDIA GeForce 6200 with 64-bit memory, working in AGP 4x mode.

- The reconstructed surface contains holes. Sometimes big quads neighbor several other quads on each side and it is likely that the smaller quads have different normals. This is a sufficient condition for surface discontinuity (figure 11). A straight-forward approach is to offset all vertices of the smaller quads that touch the side of the big quad in image space. For the current application of path-finding, this side effect is not very important because the bounding volume of the robot can be enlarged with the size of the biggest estimated surface crack. For other uses, however, a fix might be needed.

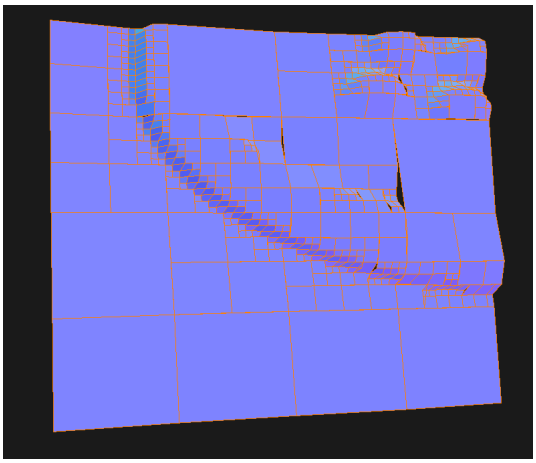


Figure 11: Surface discontinuity

- The reconstructed surface is continuous. Imagine that the a robot sensor creates the depth-map. The robot is located in a room with a single box in front of it. In this scenario the sensor will create a discontinuous depth-map. The proposed algorithm will connect the front face of the box with the back wall, which might not be correct. It is difficult, however, to determine whether such surfaces should be connected. One way is to discard the generated quad if it is almost parallel to the viewing direction, but this will often introduce erroneous cracks. Since finding proper heuristics is not trivial, this extension is left as a possible future work. Figure 12 depicts how discontinuous are currently handled.

- The generated quads are not planar. The generated mesh might need to be triangulized or the generated quads flattened before further use. For example, efficient collision detection systems project each face to 2D by dropping a coordinate. For non-flat faces this might produce errors, but in the typical case they will probably be unnoticable if the surface could be non-continuous (see the first item above).

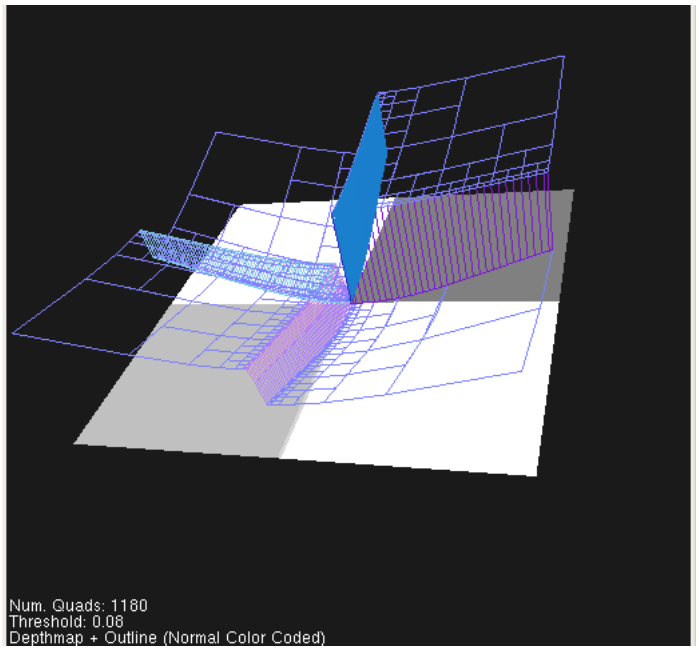


Figure 12: Reconstruction of a discontinuous depth-map

References

[1] Ziegler G., Tevs A., Theobalt C., Seidel P., 2006. *GPU Point List Generation through Histogram Pyramids*, Max Planck Saarbruecken.

[2] Ziegler G., Dimitrov R., Theobalt C., Seidel P., 2006. *RealTime QuadTree Analysis using HistoPyramids*, Max Planck Saarbruecken.

[3] Garland M., Heckbertt P, 1997. *Surface Simplification Using Quadric Error Metrics*, Carnegie Mellon University.

[4] Yu Z., Wong H. *An Efficient Adaptive Simplification Method for 3D Point-based Computer Graphics Models*, ASM

[5] Nevado M., Bermejo J., Casanova E. *Obtaining 3D Models of Indoor Environments with a Mobile Robot by Estimating Local Surface Directions*, Robotics and Autonomous Systems.

[6] Liu Y., Emery R., Chakrabarti D., Burgard W., Thrun S. sdfd *Using EM to Learn 3D Models of Indoor Environments with Mobile Robots*, Carnegie Mellon University.

[7] Soucy M., Laurendeau D. *Multiresolution surface modeling based on hierarchical triangulation*, Computer Vision and Image Understanding.

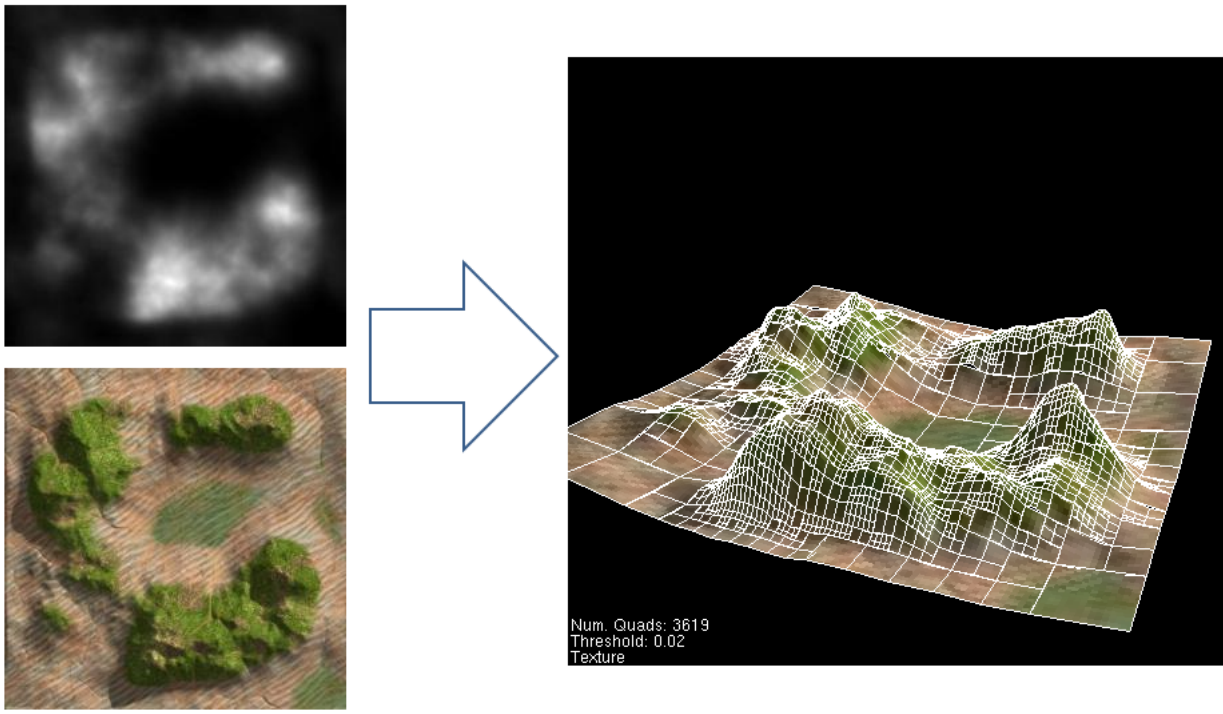


Figure 13: Reconstructed surface.
Texture reference: Fagerlund, M. http://www.hypeskeptic.com/mattias/Landscaper/