TU WIEN Informatics

Noisy Change Detection in 3D Scans

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Informatics

by

Nikolas Kaipel
Registration Number 12140073

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Stefan Ohrhallinger, Mag.rer.soc.oec. PhD

Vienna, November 13, 2025

Nikolas Kaipel	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Nikolas Kaipel

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 13. November 2025

Nikolas Kaipel

Acknowledgements

I would like to express my sincere gratitude to several people who supported me throughout the completion of this Bachelor's thesis.

First and foremost, I want to thank my supervisor, Stefan Ohrhallinger, for his guidance, constructive feedback, and support in completing this thesis.

I also wish to thank my family for their support and encouragement throughout working on this thesis.

Abstract

This thesis introduces a noise-aware, GPU-accelerated framework for 3D change detection with interactive performance that leverages a specialized voxel-tower scene representation. To handle noise inherent in measurements from depth-sensors such as the Kinect v2, we propose uncertainty-aware ellipsoids derived from empirical sensor noise models. A highly parallel, multi-stage CUDA pipeline is presented, implementing an efficient binning algorithm to rasterize these ellipsoids into the voxel-tower structure on the GPU. This approach achieves significant speedups and better scalability compared to prior CPU-based approaches. We evaluate the framework on both synthetic and real-world data, measuring runtime performance and robustness to sensor noise. The complete system is shown to convert depth images to the voxel-tower representation and detect changes in real time. By explicitly modeling sensor uncertainty and utilizing GPU parallelism, the proposed approach delivers a significantly more robust and scalable foundation for future real-time 3D change detection systems.

Contents

CHAPTER 1

Introduction

The detection of geometric changes in three-dimensional environments constitutes a central challenge in the fields of computer vision and robotics. As 3D sensor technology such as LiDAR has become increasingly accessible, widely used and precise, the need for reliable, real-time change detection has grown. The ability to identify and quantify spatial changes over time is fundamental to a wide range of modern applications—ranging from autonomous navigation and robotic manipulation to augmented reality and infrastructure monitoring.

Despite substantial progress in 3D sensing and reconstruction, real-time 3D change detection remains a difficult problem. The primary challenges arise from the massive volume of 3D data, the inherent noise and uncertainty in depth sensor measurements, and the high computational cost of spatial comparisons. Traditional geometric representations such as point clouds or polygonal meshes, while widely used, often suffer from high memory consumption and limited scalability for large-scale or high-frequency comparisons. Efficient change detection therefore requires both a compact spatial representation and a computational framework capable of exploiting modern parallel hardware.

This thesis builds upon the work of Oliver Kubicek [1], who proposed a solution to the problem of spatial representation by introducing a voxel-based scene representation for 3D change detection from depth images. In his approach, the 3D scene is discretized into a two-dimensional grid of vertical voxel towers. Each grid cell maintains a skip-list of depth intervals representing either regions that are empty but seen by the camera (“seen intervals”) or surface geometry (“surface intervals”). The remaining regions are "unseen intervals", which are implicitly given and not explicitly stored. This representation enables efficient comparison between two scene states by directly analyzing differences in the z-axis intervals of corresponding voxel towers. While Kubicek’s implementation demonstrated the conceptual validity of this method, its sequential rasterization and interval generation on the CPU are computationally limited. Moreover, the system’s

performance is affected by the presence of depth sensor noise, making it challenging to differentiate between real and spurious changes caused by noise.

The primary objective of this thesis is to address these limitations by re-implementing and extending the system with a focus on computational efficiency, robustness to noise, and scalability. The implementation provided in this thesis is a fully parallelized rasterization pipeline implemented in CUDA, which transforms depth images into the voxel tower representation through a sequence of optimized GPU kernels. This parallel architecture enables the processing of millions of geometric primitives in parallel, significantly reducing execution time compared to a CPU-based implementation.

In addition to the performance improvements, this work introduces several methods to improve the robustness against measurement noise and uncertainty inherent to depth sensors. Most notably, the rasterization process has been generalized from quads to ellipsoid primitives for the surface representation. Each ellipsoid encodes the uncertainty distribution of a depth measurement, with its radii being derived from the expected noise characteristics of the sensor. This modification enhances the resilience to noise in the surface rasterization process.

Together, these contributions result in a system that combines the compactness of interval-based voxel representations, the throughput of massively parallel GPU computation, and the robustness of uncertainty-aware surface modeling. The resulting system not only demonstrates a significant speedup over prior CPU implementations but also achieves improved resilience to noise and more consistent detection of genuine geometric changes.

CHAPTER 2

Related Work

This chapter reviews the foundational work this thesis, covering three primary areas: change detection in 3D scans, noise modeling in depth sensors, and GPU-accelerated computation using CUDA. First, we summarize the data structure and voxel tower framework introduced, which provides the structural basis for efficiently representing and comparing 3D scenes, used in this paper. Then, several papers are described that address the problem of 3D change detection and how their methods compare to the approach taken in this thesis. Next, we discuss Köppel's empirical noise models for depth sensors, which are used in this thesis to model the noise in the data that was captured using a Kinect v2 sensor. Finally, we outline key CUDA concepts relevant to the design of the GPU-based rasterization pipeline developed in this thesis. Together, these components form the context for the enhancements and contributions presented in subsequent chapters.

2.1 Change Detection in 3D Scans

Kubicek [1] introduces the voxel-based framework for real-time change detection in 3D scenes captured via depth images which this paper is based on. The core contributions from Kubicek are a sparse voxel grid aligned with the x-y plane, where each grid cell, termed a *voxel tower*, stores a skip list of non-overlapping intervals along the z-axis. This voxel tower representation draws from the Discrete Depth Structure (DDS) proposed by Radwan et al. [2], which discretizes a projection plane into a 2D grid with each cell storing sorted bounding depth intervals that encapsulate surfaces from projected point splats as cylinders, processed through an efficient GPU pipeline involving fragment counting, projection, sorting, and blending for near-linear construction time. These intervals are labeled as either *Surface* (representing observed geometry) or *Seen* (representing camera-visible space), with *Unseen* regions implicitly defined by the absence of intervals. The pipeline begins by generating quads from the depth image in camera space, back-

projecting them to world coordinates via camera parameters, and rasterizing them into the *voxel towers*: *Surface* intervals from quad projections onto towers, and *Seen* intervals from four quads making up a pyramidal frustum for each pixel, connecting that pixel to the camera origin. Change detection compares two voxel grids by identifying *Added* and *Removed* surfaces. This is achieved by direct comparison of voxel towers of the two voxel grids. For each interval in the voxel tower, a matching interval in the voxel tower at the same position in the other voxel grid is sought. The work from Kubicek provides the foundational voxel tower structure and rasterization backbone. This paper enhances it with noise-aware primitives to mitigate the impact of noise in real-world scenes, enabling more robust interval matching across misaligned grids due to sensor variance. In addition, an efficient binning algorithm on the GPU is implemented to speed up the rasterization of the primitives.

2.2 Change Detection

While voxel-based representations are common in change detection due to their ability to discretize captured 3D data, the exact approach varies based on scale and environment. For example, Aljumaily et al. [3] proposed a two stage change detection algorithm in urban environments. The first stage processes point clouds and assigns each point to the $1m^3$ cubes, or voxels. Then, these cubes are classified into different classes, such as ground or vegetation. The second stage, the change detection, compares these classification of voxels across time and categorizes them into *no-change*, *added objects*, *removed objects* or *noise*.

Schachtschneider [4] et al. introduce a method for change detection in urban environments involving multiple steps. Firstly, multiple scans from different epochs are aligned to achieve sub-centimeter accuracy. Secondly, the points in the point cloud are segmented into objects using a region growing algorithm. Thirdly, a 3D occupancy grids is constructed by tracing each LiDAR ray to classify voxels as *free*, *occupied*, or *unseen* across all measurements. Changes are detected by tracking and comparing theses three states for each voxel over multiple epochs. Additionally, the changes are categorized as permanent or temporary.

Another voxel-based approach is taken by Gehrung [5] et al. They propose an Octree, where the distribution of points in each voxel is modeled using Gaussian kernels. Changes are detected by calculating the intersection between two Gaussian kernels of corresponding voxels in different epochs. The result is a *Delta Octree*, that encodes added and removed geometry.

Aijazi et al. [6] propose a method for detecting and updating changes in LiDAR point clouds for automatic 3D urban cartography. The approach begins by classifying the point cloud into permanent and temporary objects, removing the temporary ones. The point cloud is then mapped onto a 3D evidence grid, where each cell's score is computed based on attributes such as occupancy volume, surface normals, laser reflectance intensity and RGB values. Changes are detected by comparing successive grids using custom similarity

functions, allowing categorization into additions (e.g., new constructions), removals (e.g., demolitions), or modifications.

These approaches are all using some form of voxel-based representation but the data structure they use are fundamentally different to the one described in the previous section and used in this thesis.

2.3 Noise Modeling in Depth Sensors

Köppel [7] developed empirical noise models for the KinectV2 and Phab2Pro depth sensors, characterizing both lateral (X and Y) and axial (Z) noise. The models predict noise as a function of pixel position (x, y), depth z, and sensor rotation. Lateral noise is captured by separate cubic polynomials $\sigma_x(z)$ and $\sigma_y(z)$ in depth, while axial noise employs a quadratic form $\sigma_z(x, y, z)$. The combined model ensures 90% of real measurements fall within predicted bounds. This thesis adopts Köppel's KinectV2 coefficients to parameterize ellipsoid primitives that represent surface geometry in the voxel grid construction pipeline. By modeling measurement uncertainty as ellipsoids with semi-axes derived from these noise estimates, the approach accounts for sensor inaccuracies during rasterization, leading to increased robustness to noise for the *Surface* intervals.

2.4 CUDA Programming

NVIDIA's CUDA [8] is a parallel computing API that allows developers to utilize the computational power of GPUs for general-purpose processing. CUDA exposes a threading model, where the user writes functions called *kernels*, which are then automatically executed by thousands of threads on the GPU, organized into blocks and grids. CUDA provides synchronization primitives like atomics and barriers, and access to the different memories on the GPU, like global, shared, texture, and constant memory. This facilitates massive parallelism with good performance for general-purpose tasks. The introduction of CUDA has revolutionized high-performance computing in multiple domains, such as scientific simulation, image processing and machine learning. This thesis leverages CUDA to implement an efficient, six-stage pipeline for real-time voxel grid construction from depth images, replacing CPU-bound rasterization with GPU parallelism.

CHAPTER 3

Method

This chapter presents the methodology developed to construct noise-aware voxel grids from depth images and to perform robust change detection between 3D scans. It begins by describing the scene-generation process, introducing the voxel tower data structure and detailing the transition from quad-based to ellipsoid-based rasterization to incorporate sensor noise. Next, the chapter outlines the rasterization of *Seen* intervals via triangle processing. Then, we introduce a six-stage GPU-accelerated pipeline for efficient voxel grid construction, using a parallel binning algorithm. Finally, the chapter explains the change detection algorithm, which compares two voxel grids while accounting for spatial misalignments caused by sensor noise. Together, these methods form the complete pipeline enabling interactive, noise-resilient change detection in 3D environments. The code for the full implementation is available on GitLab.

3.1 Scene Generation from Depth Image

The major part of the change detection process is the conversion from the depth image of the scanned scene into a special data structure introduced by Kubicek [1]. The two inputs required for the conversion are the depth image and the camera parameters of the camera that was used to record the scene.

3.1.1 Voxel Grid Data Structure

As a first step in the change detection pipeline, the depth image is converted to a special voxel-based representation introduced in the work of Kubicek [1]. This data structure allows for efficient storage, querying, and comparison of the 3D scene data. The voxel grid is stored as a two-dimensional grid aligned with the x-y plane. Each cell in this grid, also called a *Voxel Tower*, is a square with a fixed side length s and maintains a list of intervals, which represent the space along the z-axis of that cell, as illustrated in

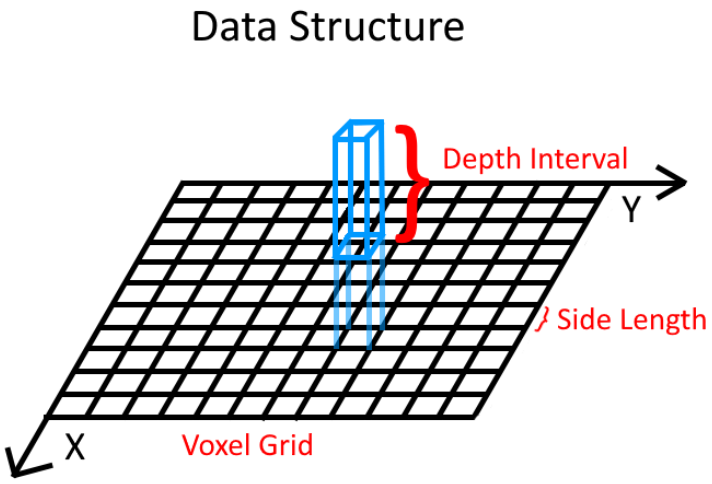

Figure 3.1: Illustration of the data structure: a 2D voxel grid storing a depth interval.

Figure 3.1. The alignment of the grid with the ground plane is chosen because scenes are usually less complex along the z-axis (vertical axis). The voxel grid allows for precise representation along the z while having a coarser representation along the x and y axes, allowing for more efficient storage. There are two types of intervals, which are stored as follows:

1. *Surface* intervals represent the physical surfaces of objects derived from the depth image.
2. *Seen* intervals define all regions that were visible to the camera.

The remaining 3D space, which is not covered by either interval type, is considered *Unseen*, which is implicitly given by the *Surface* and *Seen* intervals, and therefore not explicitly stored.

A key benefit of this data structure, compared to simply comparing two raw depth images, is its ability to represent a combined scene from multiple depth images. To achieve this, each new depth image is rasterized into the same voxel grid. Depth intervals generated from subsequent depth images are merged with the existing intervals in the grid. The resulting structure thus captures the combined representation of the full scene.

3.1.2 Rasterization of Ellipsoids

For the conversion of the depth image to the voxel grid data structure, Kubicek [1] generates a list of quads from the depth image, which is then rasterized into the voxel

grid data structure. A quad is defined by four vertices corresponding to the four corners. To get from the 2D depth image to the world coordinates, the quad is first constructed in image space and then consequently back-projected into world space using the camera's extrinsic parameters. The quads are tested for where they intersect the voxel towers and appropriate *Surface* and *Seen* intervals are inserted.

In this thesis, we replace quads as the primary primitive for the rasterization of surface intervals with ellipsoids. The main motivation for introducing ellipsoids as the primary primitive for surface intervals is to provide a representation of the uncertainty inherent in depth measurements. Rather than treating each depth value as a single, precise point, the ellipsoid representation captures the spatial extent of possible real-world positions, given the measurement uncertainty.

Each ellipsoid is defined by its center position (e_x, e_y, e_z) and semi-axis lengths (e_a, e_b, e_c), which are orthogonal and aligned with the camera coordinate axes (Figure 3.2). The ellipsoid equation in standard form is given by:

$$\frac{(x-e_x)^2}{e_a^2} + \frac{(y-e_y)^2}{e_b^2} + \frac{(z-e_z)^2}{e_c^2} = 1 \tag{3.1}$$

The semi-axis lengths of each ellipsoid are derived from a sensor-specific noise model. In this implementation, data from a Kinect v2 camera are used, and the noise characteristics are modeled following Köppel [7]. The model expresses lateral (x, y) and axial (z) noise based on the pixel position and depth value.

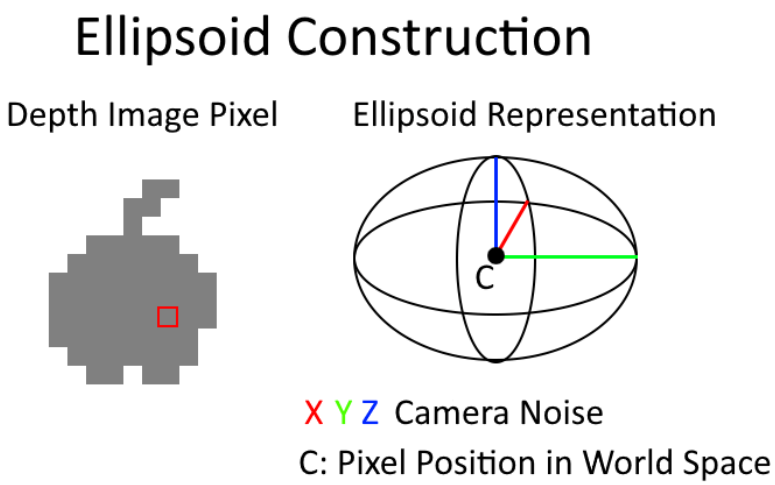

Figure 3.2: Illustration of the ellipsoid construction: Depth image pixel to ellipsoid.

The lateral noise components are modeled as follows:

$$\sigma_x(z) = a_1 z^3 + a_2 z^2 + a_3 z + a_4$$

$$\sigma_y(z) = b_1 z^3 + b_2 z^2 + b_3 z + b_4$$

, where z is the measured depth and a_i and b_i are empirically fitted coefficients determined by Köppel [7].

The axial noise, in contrast, depends not only on depth, but also on lateral position:

$$\sigma_z(x, y, z) = c_1x^2 + c_2y^2 + c_3z^2 + c_4xy + c_5yz + c_6xz + c_7x + c_8y + c_9z + c_{10}$$

, where x and y are the pixel position, z is the depth to the object, and c_i are, again, empirically determined coefficients [7].

The semi-axes are set as

$$\begin{aligned} e_a &= \sigma_x(z_p), \\ e_b &= \sigma_y(z_p), \\ e_c &= \sigma_z(x_p, y_p, z_p) \end{aligned}$$

, where z_p is the depth value at the pixel position (x_p, y_p).

Additionally, we need to project the ellipsoid's position into world space. Given the pixel coordinates (x_p, y_p) and the depth value at that pixel (z_p), we can use the camera's intrinsic and extrinsic parameters to back-project into the world's coordinate system. First, the pixel is transformed into the camera coordinate system using the camera's intrinsic parameters:

$$x_c = \frac{z_p(x_p - c_x)}{f_x}, \quad y_c = \frac{z_p(y_p - c_y)}{f_y}, \quad z_c = z_p$$

, where x_c, y_c and z_c are the positions in camera space, f_x and f_y are the focal lengths and c_x and c_y are the optical center of the camera.

Then, the ellipsoid center position (e_x, e_y, e_z) in world space is calculated as follows:

$$\begin{bmatrix} e_x \\ e_y \\ e_z \end{bmatrix} = R \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} + \mathbf{t} \tag{3.2}$$

, where R is the rotation matrix representing the camera's orientation, and $\mathbf{t}$ is the translation vector, which corresponds to the camera's position.

With position and semi-axes determined, each ellipsoid can now be rasterized. The essential problem of rasterization is identifying which voxel towers intersect the ellipsoid volume and to solve for the depth intervals (z-values) of those intersections (Figure 3.3). For an ellipsoid centered at (e_x, e_y, e_z) with semi-axes (e_a, e_b, e_c), we check whether any of the voxel tower's corners intersect the ellipsoid.

Let $\mathbf{A}, \mathbf{B}, \mathbf{C}$ be the camera's basis vectors in world space. For a voxel tower corner t_x, t_y, we search for the distance d_z from the ellipsoid center, such that the point $\mathbf{p} = (t_x, t_y, e_z + d_z)$ lies on the ellipsoid's surface. We form the vector from the center to

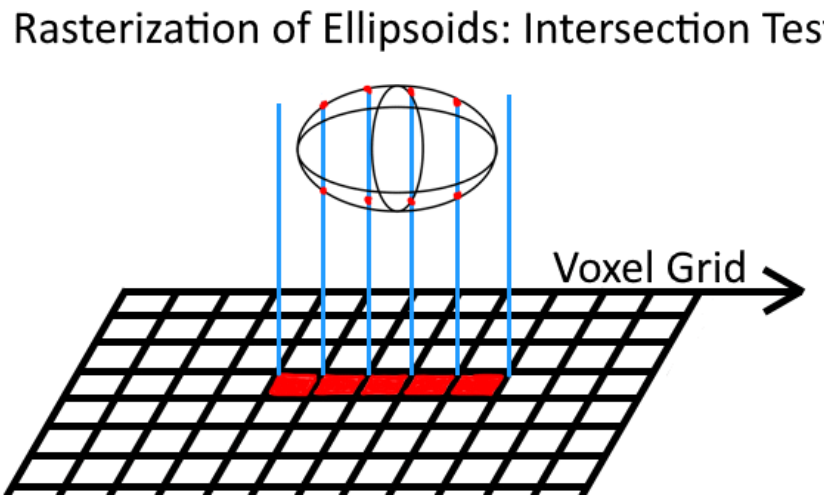

Figure 3.3: Illustration of the ellipsoid rasterization: intersection test.

this point $\mathbf{v} = [t_x - e_x, t_y - e_y, d_z]^T$. Then, this vector is projected onto the camera's local axes so that we obtain the local coordinates:

$$\begin{bmatrix} x_{local} \\ y_{local} \\ z_{local} \end{bmatrix} = \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \end{bmatrix} \mathbf{v} = \begin{bmatrix} A_x(t_x - e_x) + A_y(t_y - e_y) + A_z d_z \\ B_x(t_x - e_x) + B_y(t_y - e_y) + B_z d_z \\ C_x(t_x - e_x) + C_y(t_y - e_y) + C_z d_z \end{bmatrix}$$

For the $\mathbf{A}$ axis, let U_0 denote the constant part $A_x(t_x - e_x) + A_y(t_y - e_y)$ and $U_1 = A_z$ be the coefficient for d_z.

Similarly, we define V_0, V_1 for axis $\mathbf{B}$ and W_0, W_1 for axis $\mathbf{C}$.

Therefore,

$$\begin{aligned} \mathbf{A} \cdot \mathbf{v} &= U_0 + U_1 d_z \\ \mathbf{B} \cdot \mathbf{v} &= V_0 + V_1 d_z \\ \mathbf{C} \cdot \mathbf{v} &= W_0 + W_1 d_z \end{aligned}$$

.

Substituting these into the ellipsoid equation (Equation 3.1) we get

$$\frac{(U_0 + U_1 d_z)^2}{e_a^2} + \frac{(V_0 + V_1 d_z)^2}{e_b^2} + \frac{(W_0 + W_1 d_z)^2}{e_c^2}$$

To get the standard quadratic form $K_a d_z^2 + K_b d_z + K_c$, we first expand the quadratic terms and then sum up terms, which yields the coefficients:

$$K_a = \frac{U_1^2}{e_a^2} + \frac{V_1^2}{e_b^2} + \frac{W_1^2}{e_c^2}$$
$$K_b = 2\left(\frac{U_0 U_1}{e_a^2} + \frac{V_0 V_1}{e_b^2} + \frac{W_0 W_1}{e_c^2}\right)$$
$$K_c = \frac{U_0^2}{e_a^2} + \frac{V_0^2}{e_b^2} + \frac{W_0^2}{e_c^2} - 1$$

The discriminant $D = K_b^2 - 4K_a K_c$ determines the number of intersections. If $D \geq 0$, the tower corner intersects the ellipsoid at:

$$dz_{min} = \frac{-K_b - \sqrt{D}}{2K_a}$$
$$dz_{max} = \frac{-K_b + \sqrt{D}}{2K_a}$$

Otherwise, the corner does not intersect the ellipsoid.

If none of the tower corners intersect the ellipse, the ellipse might be entirely contained in the tower. Thus, an additional containment test is performed to cover this case. To do this, the semi-axes of the ellipsoid are projected into world space, which yields the semi-axes $proj_a, proj_b$.

Then, we check if the bounding box defined by

$$[e_x - proj_a, e_x + proj_a] \times [e_y - proj_b, e_y + proj_b]$$

is fully contained inside the voxel tower's bounding box. In that case, the full z-range of the ellipsoid is added as a depth interval to the tower.

This approach produces reliable intersection detection while being computationally efficient, but may miss overlaps in a few special cases. This is a trade-off between speed and accuracy and is a deliberate choice, which could be adapted. To achieve maximal accuracy, a mathematically exact intersection test for tower area and ellipsoid would be required, which would increase the computational cost.

3.1.3 Rasterization of Triangles

To rasterize the *Seen* intervals, which represent all regions visible to the camera, this implementation uses triangles. The approach is the same as the one introduced by Kubicek [1], but instead of rasterizing quads, triangles are rasterized. In Kubicek's implementation, two of the quads' vertices used for the rasterization of *Surface* are the camera position, meaning, the quads are actually triangles. Because Kubicek used

quads for the rasterization of *Surface*, the same code was reused for rasterizing the *Seen* intervals, even though the quads are triangles in this case. In this implementation, the quads for the *Surface* are replaced with ellipsoids, so the code reuse does not apply anymore. Thus, dedicated triangle rasterization is implemented for the *Seen* intervals, which is very efficient and simple due to the planar nature of the triangles.

For each pixel in the depth image, a quad is generated and back-projected into world space. The back-projection follows the same process as for ellipsoids: pixel coordinates are first transformed to camera space using intrinsics, then rotated and translated to world space (3.2). Using the four vertices of that back-projected quad, the following four triangles are created: $(v_0, v_1, \mathbf{c})$, $(v_1, v_2, \mathbf{c})$, $(v_2, v_3, \mathbf{c})$, and $(v_3, v_0, \mathbf{c})$, where v_0, v_1, v_2, v_3 are the back-projected quad vertices and $\mathbf{c}$ is the camera position. Together, these four triangles form a pyramidal frustum, representing the visible space for that pixel.

During rasterization, each assigned triangle is processed for each voxel tower. For each voxel tower, we test whether any triangle intersects this tower. First, each corner of the tower t_x, t_y is checked for overlap with the triangle's projection onto the x-y plane using a barycentric coordinate test.

Let the triangle vertices without the z component be $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2 \in \mathbb{R}^2$, and define the following vectors:

$$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_2, \quad \mathbf{v} = \mathbf{p}_0 - \mathbf{p}_2, \quad \mathbf{w} = [t_x - \mathbf{p}_{2x}, t_y - \mathbf{p}_{2y}]^T$$

We can then calculate the determinant, which yields twice the signed area of the triangle:

$$\det = \mathbf{u}_y \mathbf{v}_x - \mathbf{u}_x \mathbf{v}_y$$

Now, the weights of the barycentric coordinates are

$$\begin{aligned} w_0 &= \frac{\mathbf{u}_y \mathbf{w}_x - \mathbf{u}_x \mathbf{w}_y}{\det} \\ w_1 &= \frac{\mathbf{v}_x \mathbf{w}_y - \mathbf{v}_y \mathbf{w}_x}{\det} \\ w_2 &= 1 - w_0 - w_1 \end{aligned}$$

, where $w_0 + w_1 + w_2 = 1$.

The point is inside the triangle if all three weights are non-negative. Having the weights, the z value can be determined using barycentric interpolation:

$$z = w_0 z_0 + w_1 z_1 + w_2 z_2,$$

where z_i is the z value of the ith triangle vertex.

Using this z value, the *Seen* interval is inserted into the corresponding voxel tower.

If none of the tower corners lie inside the triangle, the rasterizer performs a fallback check, considering whether all triangle vertices lie inside the voxel tower's bounding box, meaning the triangle is fully contained in the tower.

For all triangle vertices $v_i, i \in 0, 1, 2$, and tower side length of s, we check if

$$t_x - s \leq v_{ix} \leq t_x + s$$

and

$$t_y - s \leq v_{iy} \leq t_y + s$$

. If the conditions are true for all three vertices, the triangle is within the voxel tower. The depth interval is then given by

$$[z_{min}, z_{max}] = [min(z_0, z_1, z_2), max(z_0, z_1, z_2)]$$

.

3.2 GPU-Accelerated Voxel Grid Construction Pipeline

To achieve interactive performance in processing depth images and constructing voxel grids, the implementation uses GPU acceleration. The pipeline consists of six different stages, each implemented as a CUDA kernel. In combination, the kernels implement a parallel binning algorithm designed to maximize parallelism and minimize synchronization overhead. Each voxel tower corresponds to one *bin* in this implementation.

CPU Setup

Before GPU execution, the CPU prepares the necessary data structures and transforms the primitives from pixel space to world coordinates. The CPU setup includes the allocation of required GPU buffers, the conversion from the depth image to primitives and a one-time initialization of the CUDA framework. Each pixel in the depth image is converted into two types of primitives: the ellipsoid that represents surface geometry and models the uncertainty of measurements, and triangles, which are used for the rasterization of the *Seen* intervals. Both primitives are transformed from camera to world coordinates using the camera's intrinsic and extrinsic parameters. Simultaneously, the minimum and maximum x and y coordinates across all generated primitives are tracked to define voxel grid bounds and memory allocation sizes. The setup is done on the CPU because the computational cost is negligible in comparison to the rest of the processing pipeline.

Stage 1: Range Precomputation

The first stage determines the spatial bounds of each geometric primitive and identifies which voxel towers it might intersect. Each primitive is handled by one CUDA thread in parallel, computing its axis-aligned bounding box (AABB) and projecting that box onto

the voxel grid. For ellipsoids, the bounding box is computed using the center position and semi-axis lengths projected into world space, similar to how it is done in the containment check described in Section 3.1.2. For triangles, the AABB is derived from the minimum and maximum coordinates across all three vertices. The resulting bounding boxes are mapped to integer index ranges of voxel grid indices, expressing the range of voxel towers that the bounding box covers.

Stage 2: Primitive Counting

In the second stage, the algorithm counts how many primitives overlap each voxel tower. Using the precomputed ranges, each thread increments an atomic counter corresponding to the towers within the primitive's bounds. The result is a one-dimensional array, *bin_counts*, stored in a row-major order, where the number of intersecting primitives for a voxel tower at position x, y, is stored at $y \times w_{grid} + x$, where w_{grid} is the number of columns or the width of the voxel grid.

Stage 3: Prefix Scan and Memory Allocation

In the third stage, a parallel prefix-scan computes the cumulative offsets for each voxel tower's interval storage. This stage is implemented using the Thrust library's optimized scan operations, which leverage GPU-specific optimizations for maximum performance. The prefix-scan yields, again, a one-dimensional array, *bin_offsets*, with the same ordering and indexing, but the value stored is the offset into the global interval storage for that voxel tower.

The final prefix scan results are used to calculate the total number of primitive-voxel intersections, which is used to allocate the buffers that hold the interval data and the primitive index buffer. This has the benefit of requiring only a one-time allocation for all of the interval data, which is more efficient than alternatives such as dynamic memory allocations, which can be challenging on GPUs.

Each interval comprises two floating-point values: depth start and depth end. The interval type is encoded in the sign bit of the end value, where a positive end value maps to *Surface* intervals and a negative value to *Seen* intervals. The intervals are stored in an *structure of arrays* (SOA) approach, with three arrays holding the begin, end and type values, respectively. The SOA layout was chosen because the interval storage was by far the largest memory consumer, and it turned out to be a bottleneck on the tested hardware. The SOA approach allows for tight packing of the 8 bit integer types, which leads to a strongly reduced memory footprint and also improved runtime performance on the tested hardware.

Stage 4: Primitive Index Assignment

The fourth stage populates the primitive index buffer with the actual primitive indices that intersect each voxel tower. This stage uses the same binning approach as the counting

stage, but instead of just incrementing counters, it writes the primitive indices to the appropriate slots in the global primitive index buffer. The kernel assigns one thread to each primitive, which iterates through the voxel range that was computed in stage 1. For each voxel tower in the primitive's range, it atomically increments a fill counter to obtain a unique slot index, then writes the primitive index to that slot in the global primitive index buffer. The slot address is computed as $bin\_offsets[vid] + slot$, where vid is the voxel tower index, $bin\_offsets[vid]$ is the starting offset for that voxel tower's section of the global buffer, and $slot$ is the atomic counter value. This creates a compact, contiguous list of primitive indices for each voxel tower, enabling efficient access during the subsequent rasterization stage. The atomic operations ensure thread safety when multiple primitives are being assigned to the same voxel tower simultaneously.

Stage 5: Parallel Rasterization

The fifth stage performs the actual rasterization, converting primitives into depth intervals for each voxel tower. The kernel launches one block per voxel tower, with each block processing all primitives assigned to that tower. Each block uses shared memory to store intermediate interval results for efficient, local memory accesses. The kernel first initializes buffers in shared memory for storing interval data (begin, end, and type values) and a counter for the number of intervals generated. For each primitive assigned to the voxel tower, the kernel calls either the rasterization algorithm for triangles or for ellipsoids depending on the primitive's type. The aforementioned counter stored in shared memory is atomically incremented to safely append the generated intervals to the shared memory buffer. After all primitives have been processed, the intervals are copied from shared memory to the global interval storage arrays, and the final interval count is stored with the voxel tower.

Stage 6: Merging

The final stage merges overlapping and adjacent intervals within each voxel tower to create a compact, non-redundant representation. This stage is crucial for reducing memory usage and improving query performance, as the raw rasterization output often contains many overlapping intervals that can be combined. The kernel processes each voxel tower independently, using a two-phase approach. First, it copies all intervals from the global storage into shared memory and sorts them by their begin values using a parallel in-place bitonic sorting algorithm [9]. The bitonic sorting algorithm was chosen because of its relatively simple implementation, while meeting the performance requirements for this use case. After sorting, the kernel performs a sequential merge pass that combines overlapping intervals of the same type. The final result is a compact array of non-overlapping intervals for each voxel tower.

3.3 Change Detection

The change detection algorithm works by comparing two voxel grids generated from depth images captured at different time points. The core principle is to identify intervals that exist in one voxel grid but not in the other, which indicates changes in the scene geometry. The comparison is performed on a per voxel tower basis. The output of the change detection is another voxel grid, but instead of the *Surface* and *Seen* intervals, this voxel grid stores intervals that are either:

1. *Added* intervals, which represent surfaces present in the new scene but missing in the old scene.
2. *Removed* intervals, which represent surfaces present in the old scene but missing in the new scene.

Due to noise in the measurements of the camera, the two compared voxel grids often differ slightly in dimensions, and the same physical surface might end up rasterized in different voxel tower positions. These minor spatial misalignments prohibit straightforward per voxel tower comparison. To combat this issue, the comparison algorithm not only checks the corresponding voxel tower at the exact same position in the other voxel grid, but also checks neighboring voxel towers for trying to find the matching depth interval. The radius of voxel towers, in which the comparison algorithm searches for the matching depth interval, is a tunable parameter.

CHAPTER 4

Results

The results presented in this chapter evaluate the proposed voxel grid construction pipeline from depth images and the change detection algorithm. The pipeline was tested on both artificial and real-world data. The artificial data consists of noise-free depth images, serving the purpose of validating the conversion from depth images to the voxel grid data structure; in particular, the ellipsoid and triangle rasterization processes as well as the parallel binning algorithm. The real-world data, captured using a Kinect v2 sensor, is used to assess the robustness of the pipeline to sensor noise and inaccuracies during voxel grid construction and evaluate the change detection strategy. All experiments were conducted on an NVIDIA RTX 3070 GPU with 8 GB VRAM and an Intel Core i7-12700 CPU.

4.1 Artificial Data

The artificial data was generated to simulate scenes without noise to validate the conversion of the depth image to the voxel grid, in particular, the ellipsoid and triangle rasterization, as well as the general binning algorithm. Two models were used for the artificial data: an airplane and an angel statue. To get depth images from the 3D models, the tool *virtual-3d-scanner* [10] was used to simulate scanning a scene with a camera such as the Kinect v2.

4.1.1 Airplane Scene

Figure 4.1 depicts a modified depth image of the artificial airplane scene, output by the *virtual-3d-scanner* tool [10] and serving as input to the change detection algorithm. In the unmodified depth image, all values are close to black (0), resulting in the airplane being nearly invisible in the depth image. For the visualization, the values were scaled to enhance visibility. Also, the displayed image uses 8-bit color channels; the actual image

uses 16-bit color channels and contains finely varying depths. Pixels with infinite depth (black regions) are excluded from the primitive generation. The algorithm expects black to be infinite depth and white to be zero depth. Because some sensors might produce depth images where this spectrum is inverted, the implementation provides an option to invert the depth values when loading the depth image.

Figure 4.1: Depth image of the artificial airplane scene.

Figure 4.2 shows the 3D visualization of the voxel towers after rasterizing ellipsoids and triangles, showing the *Surface* intervals (red) and *Seen* intervals (blue), representing the airplane's surface and seen space, respectively. In the background, the x-y voxel grid aligned with the ground plane is rendered in gray. For each voxel tower in the voxel grid, the list of intervals for that voxel is rendered as cuboids with width and height matching those of the grid, and the depth corresponding to the interval's z-coordinates.

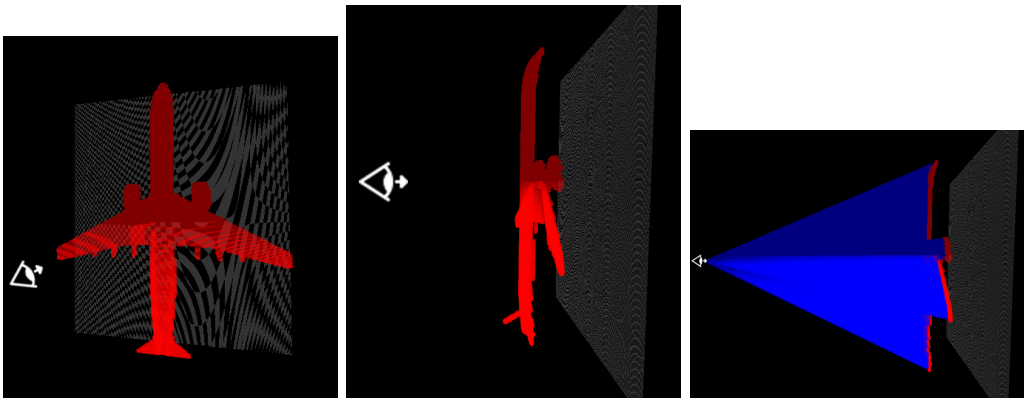

Figure 4.2: 3D visualization of voxel towers after rasterizing ellipsoids and triangles, showing the *Surface* intervals (red) and *Seen* intervals (blue), representing the airplane's surface and seen space, respectively.

Table 4.1 summarizes the runtime configuration for the airplane scene. The 320×240 depth image yields 35912 triangles and 8978 ellipsoids, totaling 44890 primitives.

Table 4.1: Runtime configuration of the airplane scene.

Depth Image	320x240
Side Length	1.000000
Voxel Grid	601 × 707 (424907 total voxels)
No. of Triangles	35912
No. of Ellipsoids	8978
Total Primitives Processed	44890
Maximum Intervals in a Tower	5
Average Intervals per Tower	0.45

As indicated in Table 4.1, the depth image has dimensions of 320x240. The conversion from depth image to primitives, the GPU initialization, and the data transfer each take just a few milliseconds (Table 4.2). The subsequent GPU pipeline takes a total of 107.98 ms to complete the rasterization of the primitives into the voxel grid and intervals. The dominant part of the rasterization runtime is the allocation of the buffers to store the intervals, which occurs because the system must first read back the results from the prefix scan to calculate the required buffer sizes and then allocates rather large buffers. In total, the entire process takes 120.49 ms, which is within the requirement of interactive performance.

Table 4.2: Chronological CPU and GPU timings for the airplane scene.

Phase	**Time**
1. Ellipsoid Construction	1.89 ms
2. GPU Initialization	4.29 ms
3. CPU → GPU Transfer	6.32 ms
4. GPU Pipeline (Total)	107.98 ms
Stage 1: Precompute Ranges	0.15 ms
Stage 2: Count Primitives	8.17 ms
Stage 3: Prefix Scan & Allocation	69.25 ms
Stage 4: Fill Bins	16.89 ms
Stage 5: Rasterize Bins	7.73 ms
Stage 6: Merge Intervals	5.79 ms
Total E2E (CPU + GPU)	**120.49 ms**

The memory usage turns out to be the greater bottleneck compared to runtime performance. The voxel grid and intervals together require 1759 MB of GPU memory (Table 4.3), while the primitive index buffer requires 872 MB. Together with the storage needed to store the primitives and buffers required by the binning algorithm, the total memory

usage accumulates to around 2838 MB. This is a substantial portion of the 8 GB of VRAM available on the NVIDIA RTX 3070. Increasing the voxel grid side density or increasing the size of the depth image quickly leads to out-of-memory errors, while the runtime performance scales well and would allow for processing depth images of larger size in interactive time constraints. Possible ways to reduce the memory usage and further improve the runtime performance are discussed in Section 4.2.1.

Table 4.3: GPU memory usage of the airplane scene.

Input Primitives (Triangles + Ellipsoids)	1.44 MB
Voxel Grid (Towers + Intervals)	1760 MB
Binning Buffers (counts, offsets etc)	4.86 MB
Primitive Index Buffer	872 MB
Total GPU memory allocated	2638 MB

4.1.2 Angel Statue Scene

The angel statue scene serves a similar purpose to that of the airplane scene. However, it contains more depth variation and structural detail compared to the simple airplane scene. Therefore, it was chosen as an additional test to validate the rasterization pipeline. Again, the model was generated using the virtual-3d-scanner [10] tool. The following image (4.3) shows the depth image of the angel statue scene produced by the virtual-3d-scanner. The depth image are, again, modified to show the model more visibly.

Figure 4.3: Depth image of the artificial angel scene.

Figure 4.4 shows the result of the rasterization pipeline, visualized in the same way as the airplane scene in Figure 4.2. The main difference between the angel statue scene and the airplane scenes is the chosen ellipsoid semi-axes lengths. Normally, the semi-axes

lengths are determined through the noise model of the sensor, as described in section 3.1.2. Naturally, for the artificial data, however, there is no noise in the data and thus no noise model to apply. Therefore, the semi-axes lengths were chosen arbitrarily. To test different ellipsoid sizes, the semi-axes lengths for the angel statue scene were chosen to be 1.0, while for the airplane scene they were chosen to be 3.0. The different ellipsoid sizes can be seen in the Figures 4.2 and 4.4, and explain the clearly different looking surface representation.

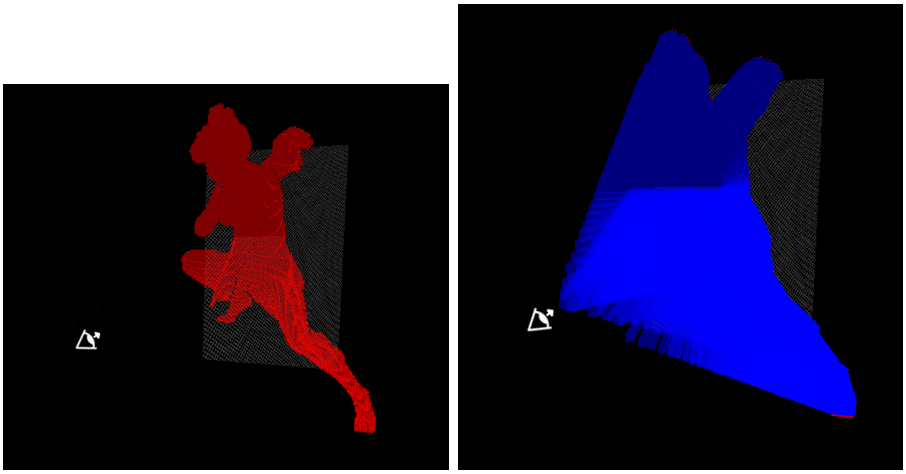

Figure 4.4: 3D visualization of voxel towers after rasterizing ellipsoids and triangles showing the *Surface* intervals (red) and *Seen* intervals (blue), representing the angel's surface and seen space, respectively.

The 320×240 depth image yields 32040 triangles and 8021 ellipsoids, totaling 40061 primitives (Table 4.4). This makes the angel scene similar in complexity to the airplane scene, when it comes to raw processing of primitives. The voxel grid size of 292 × 481 is determined from the bounding box of all back-projected ellipsoids in the scene. The min_x, max_x, min_y, and max_y values are computed during scene construction by taking the minimum and maximum x/y coordinates across all triangle vertices and ellipsoid extents. Then the grid dimensions are calculated as width $= \lceil \frac{max_x - min_x}{\text{side\_length}} \rceil$ and height $= \lceil \frac{max_y - min_y}{\text{side\_length}} \rceil$, where side_length denotes the side length of the voxel towers and is an adjustable parameter.

For the angel statue scene, the GPU pipeline consumes 97.54 ms for the rasterization. With the ellipsoid construction, GPU initialization and data transfer, the total runtime accumulates to 107.92 ms. The performance profile closely resembles that of the airplane scene, notably, also with regards to how much each phase of the pipeline contributes to the total runtime.

The memory usage of the angel scene matches that of the airplane scene closely. The two highest memory consumers are, again, the voxel grid containing the intervals data, and the primitive index buffer. The total memory usage for this scene amounts to around

Table 4.4: Runtime configuration of the angel statue scene.

Depth Image	320x240
Side Length	1.000000
Voxel Grid	292 × 481 (140452 total voxels)
No. of Triangles	32040
No. of Ellipsoids	8021
Total Primitives Processed	40061
Maximum Intervals in a Tower	8
Average Intervals per Tower	0.69

Table 4.5: Chronological CPU and GPU timings for the angel statue scene.

Phase	**Time**
1. Ellipsoid Construction	1.69 ms
2. GPU Initialization	1.88 ms
3. CPU → GPU Transfer	6.82 ms
4. GPU Pipeline (Total)	97.54 ms
Stage 1: Precompute Ranges	0.08 ms
Stage 2: Count Primitives	10.79 ms
Stage 3: Prefix Scan & Allocation	58.52 ms
Stage 4: Fill Bins	19.41 ms
Stage 5: Rasterize Bins	5.23 ms
Stage 6: Merge Intervals	3.50 ms
Total E2E (CPU + GPU)	**107.92 ms**

2203 MB (Table 4.6). Potential ways to reduce the memory usage are discussed in Section 4.2.1.

Table 4.6: GPU memory usage of the angel statue scene.

Input Primitives (Triangles + Ellipsoids)	1.28 MB
Voxel Grid (Towers + Intervals)	1468 MB
Binning Buffers (counts, offsets etc)	1.61 MB
Primitive Index Buffer	731 MB
Total GPU memory allocated	2203 MB

4.1.3 Change Detection With Synthetic Data

In order to test the change detection, a synthetic scene is used, which consists of two objects. First, the angel statue from Section 4.1.2, and secondly the *Suzanne* ape head. Figure 4.5 shows the *before* state: the head is in front of the angel, which means the

angel is partially occluded from the perspective of the viewer. For the *after* state of the scene, the head is removed, revealing the angel in its entirety again. The goal is for the change detection algorithm to correctly recognize and categorize these changes into *Added* and *Removed* intervals.

Figure 4.5: Depth image of the synthetic change detection test scene.

Figure 4.6 shows the rasterized version of the *before* scene on the left, and the *after* image on the right. In the *before* picture, it is clear to see that the part of the angel covered by the head is missing.

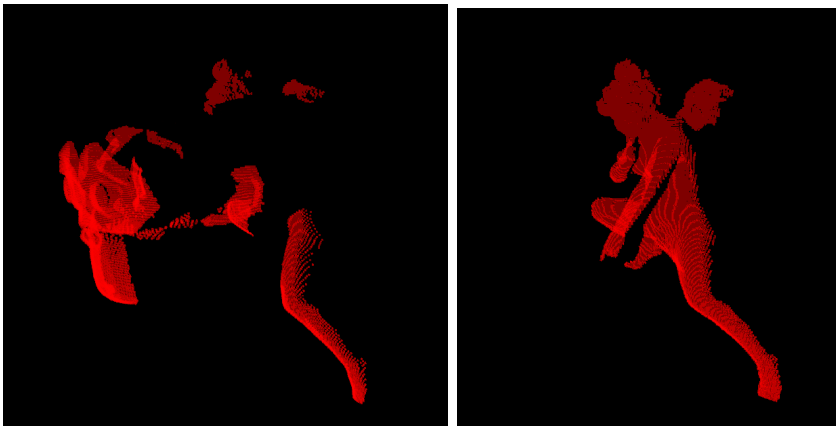

Figure 4.6: Rasterization before and after removing the occluding object.

Figure 4.7 depicts the resulting voxel grid after running the change detection algorithm. It displays *Added* intervals in red, while showing the *Removed* intervals in blue. The change detection algorithm appears to have correctly identified the ape head being removed, while also correctly identifying the part of the angel, which was previously covered, as added geometry.

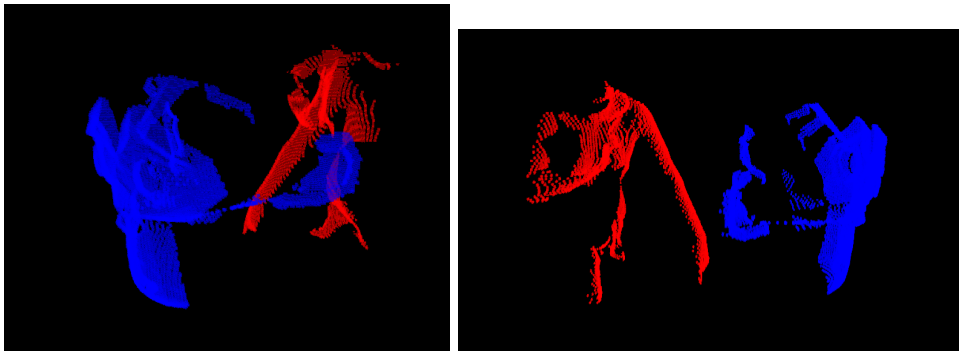

Figure 4.7: 3D Visualization of voxel towers after change detection, showing the *Added* intervals (red) and the *Removed* intervals (blue) representing the changes between the two scenes.

The runtime for the change detection follows similar patterns than the ones for the plane and angel scenes. The rasterizations of the *before* and *after* scenes take 120.84 ms and 99.08 ms, respectively (Table 4.7). After both scenes have been rasterized, the change detection algorithm takes 59.47 ms to find the changes. All in all, the entire process from the two depth images to the changes takes 279.39 ms.

Table 4.7: Chronological CPU and GPU timings for the synthetic change detection scene.

Phase	**With Occluder**	**Without Occluder**
1. Ellipsoid Construction	2.81 ms	1.95 ms
2. GPU Initialization	1.69 ms	1.53 ms
3. CPU → GPU Transfer	6.47 ms	0.19 ms
4. GPU Pipeline (Total)	109.87	95.42 ms
Stage 1: Precompute Ranges	0.07 ms	0.01 ms
Stage 2: Count Primitives	12.42 ms	10.91 ms
Stage 3: Prefix Scan & Allocation	66.54 ms	57.86 ms
Stage 4: Fill Bins	21.29 ms	19.29 ms
Stage 5: Rasterize Bins	6.03 ms	5.40 ms
Stage 6: Merge Intervals	3.53 ms	1.94 ms
5. Change Detection	59.47 ms	
6. GPU → CPU Transfer (Readback)	0.43 ms	
Total E2E (CPU + GPU)	**120.84 ms**	**99.08 ms**

4.2 Real-World Data

To evaluate the pipeline on real-world data, depth images were captured using a Kinect v2 sensor. This section presents results for two scenarios: a single depth image for validating

the voxel grid construction in the presence of sensor noise and inaccuracies, and a pair of depth images to test the full change detection capability by introducing a change in the scene (adding a chair).

The following two images (Figure 4.8) show the scene that was recorded by Kubicek [1] with a Kinect v2 sensor. The image on the left shows the scene without a chair, while the image on the right shows the scene after adding a chair. The objective is for the pipeline to detect that the chair was added to the scene and not detect spurious changes due to noise.

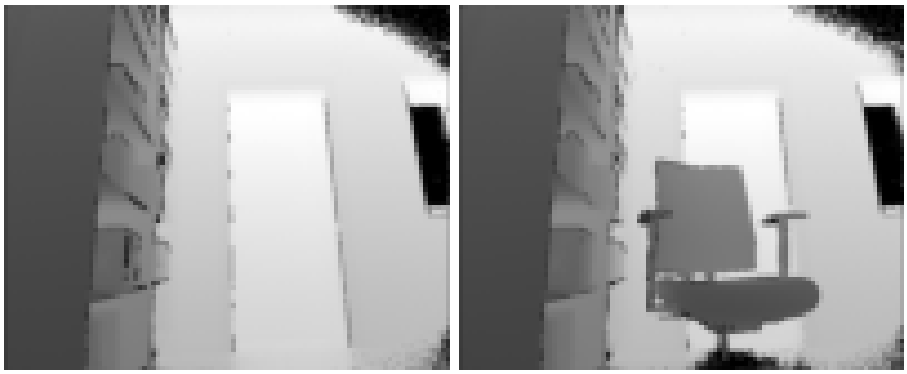

Figure 4.8: Depth images of the real-world scene without and with a chair.

The following visualization (4.9) shows the voxel towers after rasterizing ellipsoids, showing the *Surface* intervals representing the scene with the chair. The visualization here uses a gray-scale color map to show the depth values of the intervals, instead of the usual red color, to enhance the differentiability of the objects in the room.

The change detection, as per section 3.3, identifies the chair as added geometry (Figure 4.10), with *Added* intervals in red.

As described in the section 3.3, the change detection algorithm works by comparing two voxel grids pairwise on the voxel tower level. This real-world example underlines the issue mentioned in the change detection section regarding the use of two voxel grids of different dimensions. Even though the camera position and resolution are the same for both depth images, the resulting voxel grid for the scene without the chair measures 420x455, while the scene with the chair measures 420x437. The difference arises from the fact that the depth values at the border of the depth image are slightly different, which propagates through the noise-model to different ellipsoid semi-axes lengths and, in turn, different voxel grid sizes.

According to the Table 4.9, the performance characteristics of the real-world data closely align with the ones from the artificial data. The main observation for this example is the 197.73 ms runtime required for the voxel grid comparison. This computational cost arises from the need to check neighboring voxel towers for interval matches. The runtime scales with the radius of neighboring voxel towers to check. Choosing the radius is a

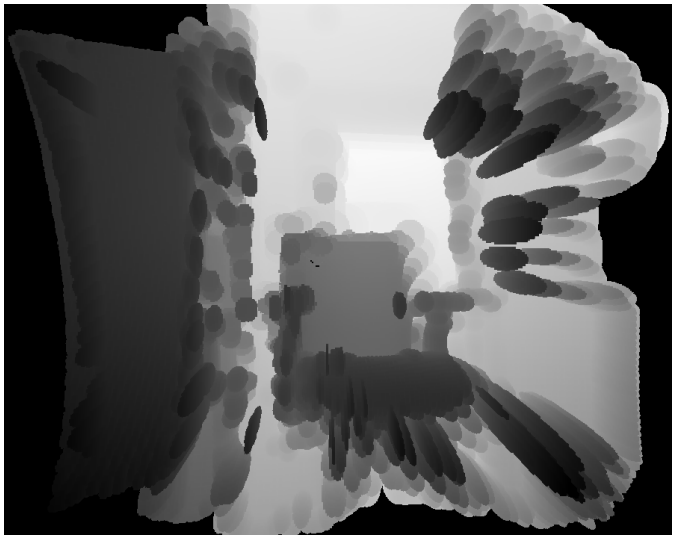

Figure 4.9: 3D Visualization of voxel towers after rasterizing ellipsoids, showing the *Surface* intervals representing the chair room scene's surface.

Table 4.8: Runtime configuration of the chair scene.

Parameter	**Without Chair**	**With Chair**
Depth image	100x82	100x82
Side length	1.000000	1.000000
Voxel Grid	420 × 455	420 × 437
No. of triangles	31720	31732
No. of ellipsoids	8111	8114
Total Primitives	39831	39846
Maximum Intervals in a Tower	22	22
Average Intervals per Tower	1.68	1.72

trade-off of speed versus the likelihood of having spurious, alone standing intervals in the resulting grid.

Even though the original depth image for the real-world data is just 100x82 compared to the 320x240 for the artificial data, the memory consumption is quite similar. This is because in the artificial depth images, a major part of the image was just black background, which is interpreted as no surface. Therefore, no primitives are generated for most of the images pixels. In comparison, the real data set, captures an indoor scene and all of the pixels turn out to be representing surface. This leads to a similar number of primitives for the artificial and real data. Another reason for why the memory consumption for the real-data are relatively larger is that the semi-axes lengths of the ellipsoids provided by the noise model are substantially bigger than the arbitrary ones used in the artificial data. Bigger ellipsoids lead to more voxel tower intersections, which,

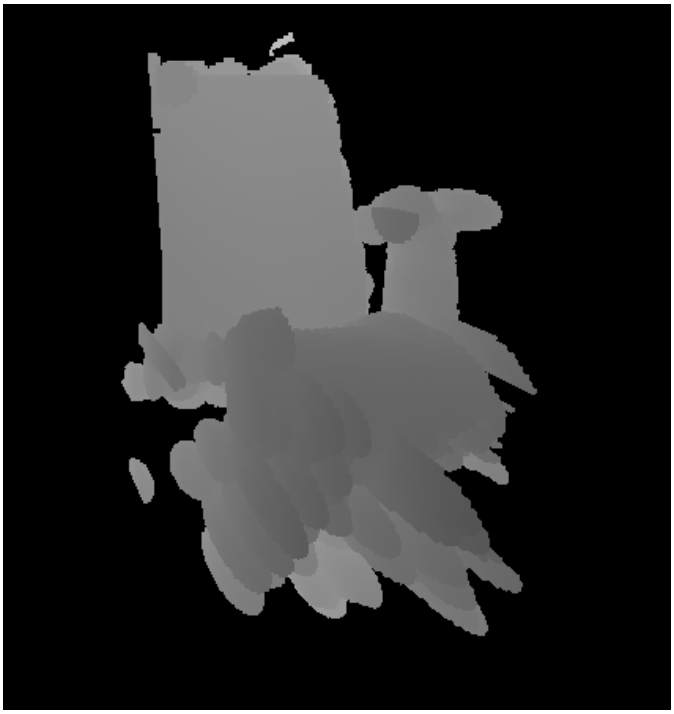

Figure 4.10: 3D Visualization of voxel towers after change detection, showing the *Added* intervals representing the changes between the two scenes.

in turn, lead to more intervals that need to be stored.

4.2.1 Potential Performance and Memory Usage Improvements

Given that the runtime performance and definitely memory usage could become a limiting factor in more complex and bigger scenes, the following outlines a few possible ways to improve the pipeline in that regard. As discussed in the previous sections covering the memory usage of the pipeline in the context of the different scenes, the biggest memory usage bottlenecks are the interval storage and the primitive index buffer storing 32 bit integers. The index buffer could in theory use less bits for the indices, depending on the number of primitives generated from the depth image. In the test cases presented in this chapter, 32 bit indices are required.

If it is possible to coarsen the spatial resolution in the x and y dimensions, the single biggest improvement to memory usage is to increase the side-length of the voxel towers. For example, the airplane scene requires 2838 MB of GPU memory with a side-length of 1.0 (Table 4.3); A doubling of the side-length to 2.0 drops the memory usage to 750 MB. Increasing it further to 4.0, the required memory amounts to 207 MB. This means that with each doubling of the side-length, the memory usage is reduced by a factor of approximately 3.7.

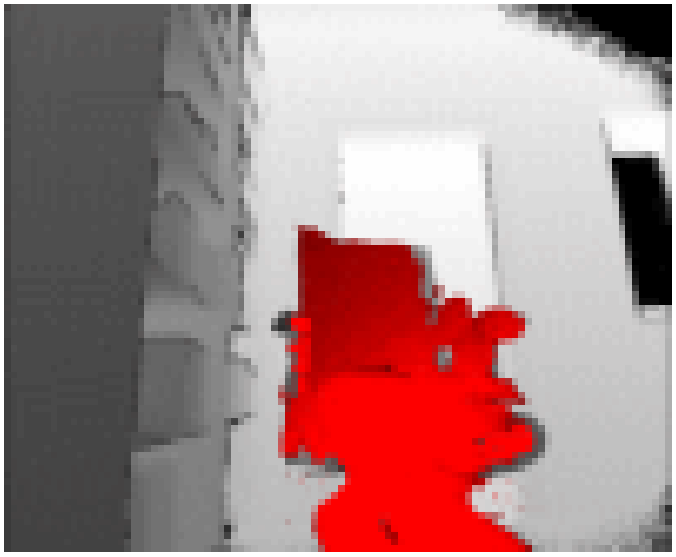

Figure 4.11: Voxel grid of containing changes overlapped with the original depth image.

Table 4.9: Chronological CPU and GPU timings for the chair scene.

Phase	**Without Chair**	**With Chair**
1. Ellipsoid Construction	1.72 ms	1.71 ms
2. GPU Initialization	2.23 ms	1.90 ms
3. CPU → GPU Transfer	6.24 ms	0.31 ms
4. GPU Pipeline (Total)	123.35	111.88 ms
Stage 1: Precompute Ranges	0.06 ms	0.03 ms
Stage 2: Count Primitives	9.80 ms	9.19 ms
Stage 3: Prefix Scan & Allocation	72.39 ms	66.37 ms
Stage 4: Fill Bins	20.94 ms	19.21 ms
Stage 5: Rasterize Bins	9.87 ms	8.88 ms
Stage 6: Merge Intervals	10.28 ms	8.20 ms
5. Change Detection	197.73 ms	
6. GPU → CPU Transfer (Readback)	0.43 ms	
Total E2E (CPU + GPU)	**133.55 ms**	**115.80 ms**

Alternatively, if increasing the side-length is not an option, the next best option is to reduce the precision of the interval positions. Right now, the positions are stored as 32 bit floats. Reducing these to 16 bit would reduce the memory usage of the interval storage, which makes up the largest part of the memory requirement, by a factor of 1.8.

A possible optimization to avoid the memory usage bottleneck with no loss of precision is streaming the memory from the CPU to the GPU as needed. This would allow for processing larger scenes that don't fit into the GPU memory at once. But it may come

Table 4.10: GPU memory usage of the chair scene.

Memory Component	Without Chair	With Chair	Comparison Result
Triangles + Ellipsoids	1.27 MB	1.28 MB	-
Towers + Intervals	1845.47 MB	1703.02 MB	0.08 MB
Binning Buffers	2.19 MB	2.19 MB	1.46 MB
Primitive Index Buffer	919.68 MB	848.45 MB	-
Total	**2768.62 MB**	**2554.94 MB**	**1.54 MB**

at the cost of increased runtime.

The biggest runtime consumer appears then to be the allocation of GPU buffers, which is done after the prefix scan. The performance bottleneck caused by the buffer allocations arises primarily from the high-latency synchronization required by the memory copy operations. The prefix scan by itself takes around 1 ms on all tested scenes, while the entire stage takes around 60-80 ms. After the prefix scan is done, the result is read back from the GPU. This information is required to allocate buffers of the proper size. The performance of this stage could be improved by avoiding the read back from the GPU and using persistently allocated buffers.

The second biggest performance consumer is the filling of the bins. This refers to the stage in which primitives are assigned to the voxel towers. The bottleneck here is the usage of global atomic operations. This could be reduced by using atomic operations in shared-memory instead, or utilizing warp-level synchronization.

CHAPTER 5

Conclusion

This thesis has addressed the critical challenges in 3D change detection with interactive performance by extending and optimizing a voxel-based scene representation originally proposed by Kubicek [1]. By introducing a fully parallelized GPU-accelerated pipeline implemented in CUDA, replacing quad-based surface primitives with uncertainty-aware ellipsoids, the proposed system achieves significant advancements in computational efficiency, noise robustness, and scalability. These contributions enable the reliable processing of large-scale depth images from commodity sensors like the Kinect v2, transforming raw 2D depth data into compact 3D voxel towers that allow precise spatial comparisons.

The experimental evaluation on both artificial and real-world datasets validates the efficacy of these enhancements. On noise-free artificial scenes, such as the planar surface and angel statue models, the pipeline demonstrates accurate rasterization of *Surface* and *Seen* intervals, with end-to-end processing times under 500 ms, well within interactive constraints. Memory consumption, peaking at approximately 2.8 GB on an NVIDIA RTX 3070, highlights scalability limits for more complex, highly detailed scenes. A more coarse voxel grid, by increasing the side lengths, reduces the memory consumption substantially.

In real-world scenarios, the ellipsoid representation effectively models sensor-specific noise characteristics to mitigate spurious interval generation. The addition of a chair to an indoor scene was detected as added geometry with minimal false positives, demonstrating the system's resilience to depth inaccuracies and minor misalignments between voxel grids.

Despite the successful improvements, there remain multiple opportunities for further, big improvements. For instance, performance and memory consumption can be improved further. In particular, the current synchronous GPU-CPU transfers, CPU-based primitive generation as well as the CPU comparison represent leave a lot of room for optimization.

Memory efficiency might also be improved by halfing the precision of interval storage or on-demand streaming of memory.

Overview of Generative AI Tools Used

To improve the correctness and clarity of the writing in this thesis, I used Gemini 2.5 Flash for text refinement. The prompt I used was "Review the following text and look for spelling and grammar errors. Also, suggest alternative words and phrases to improve clarity. Ensure that the original meaning is not altered."

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] O. Kubicek, *Change detection in 3D scans*. Bachelor's thesis, Technische Universität Wien, 2025.

[2] M. Radwan, S. Ohrhallinger, and M. Wimmer, "Fast occlusion-based point cloud exploration," *The Visual Computer Journal*, vol. 37, p. 13, Sept. 2021.

[3] H. Aljumaily, D. F. Laefer, D. Cuadra, and M. Velasco, "Voxel change: Big data–based change detection for aerial urban lidar of unequal densities," *Journal of Surveying Engineering*, vol. 147, no. 4, p. 04021023, 2021.

[4] J. Schachtschneider, A. Schlichting, and C. Brenner, "Assessing temporal behavior in lidar point clouds of urban environments," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLII-1/W1, pp. 543–550, 2017.

[5] J. Gehrung, M. Hebel, M. Arens, and U. Stilla, "A fast voxel-based indicator for change detection using low resolution octrees," *ISPRS Annals of the Photogrammetry Remote Sensing and Spatial Information Sciences*, vol. IV-2/W5, pp. 357–364, 05 2019.

[6] A. K. Aijazi, P. Checchin, and L. Trassoudaine, "Detecting and updating changes in lidar point clouds for automatic 3d urban cartography," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. II-5/W2, pp. 7–12, 2013.

[7] T. Köppel, *Extracting Noise Models – considering X / Y and Z Noise*. Bachelor's thesis, Technische Universität Wien, 2017.

[8] NVIDIA Corporation, "Cuda c++ programming guide." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2025. Accessed on 25. October 2025.

[9] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), (New York, NY, USA), p. 307–314, Association for Computing Machinery, 1968.

[10] Salingo, "Virtual 3D Scanner: Generate RGB-D image and point cloud from 3D mesh." https://github.com/Salingo/virtual-3d-scanner, 2021. Accessed: October 20, 2025.