# Simulation of scanning and reconstruction of 3D objects in virtual environment

Boris Sliz[*]
*Supervised by: Martin Madaras*[†]

Institute of Computer Engineering and Applied Informatics
Slovak University of Technology
Slovakia

## Abstract

The development of 3D scanners in recent years enabled object scanning with high precision. In the process of 3D reconstruction, the first step is to scan the object with different points of view, while these scans are aligned. Despite the significant advances of 3D scanners, their acquisition is still costly, and capturing correct scans is time-consuming.

In this paper, we focus on the simulation of the process of scan acquisition, alignment, and reconstruction in the virtual environment. Our work consists of the implementation of multiple approaches to simulate 3D scanning. We use a simulated robotic hand manipulator to achieve points where we capture virtual scans. Reconstructed virtual scans can provide information on whether we captured all parts of the scanning object, or we need additional data. We also simulate a realistic spawn of multiple parts into the bin, where we use the virtual scanner to capture the scene. Virtual scans of randomly spawned objects are useful as testing data for localization purposes.

**Keywords:** Virtual scanning, 3D reconstruction, Kinematics

## 1 Introduction

With the development of new technologies, 3D scanning and 3D object reconstruction became very popular. The latest 3D scanners are capable of capturing millions of 3D points. The precision of these points can be high, but there are still some limitations, for example, when scanning objects are shiny or reflective.

Acquired data from these scanners can be used for a variety of applications. It is prevalent in the production of CGI movies or video games, where it might be easier to scan and reconstruct objects than create them with modeling software. Another widespread usage is in preserving cultural heritage, where objects are reconstructed and viewed by people from anywhere in the world through virtual reality. Since the improvement of precision in 3D scanners, they became more prevalent in medicine, mostly in dentistry, where doctors can create new teeth based on reconstructed 3D scans.

Most of the 3D scanners use time of flight [13] scanning methods. These scanners consist of 2 parts - laser and receiver. The laser casts rays, and the receiver captures when that ray hits the object. Depth is calculated based on the time from casting to receiving ray. Another popular method is called triangulation scanning [1]. These scanners have laser and camera. Laser, camera, and a point where the laser is received create a triangle. Distance between laser and camera and the angle between laser and point are known. When the camera receives a point on the object, we can calculate the angle between the camera and this point. From this, it is possible to easily calculate all sides of the triangle, and the angle axis of this triangle represents depth. These scanners are usually scanning objects that are rotating on a rotary table, which creates an automated scanning process.

The basic approach to automated scanning and reconstruction of 3D objects is based on a scanning object with the use of a rotary table combined with a robotic arm manipulator. This is also possible with capturing many 2D images that can also create a point cloud, but the result is not entirely accurate. From captured point clouds can be object then reconstructed into a 3D model. These reconstructed objects are not always ideal. This is usually because there are not enough scanned points of some parts of the object, and users then need to start this whole process over again with new points so that they cover all parts of the object.

### 1.1 Motivation

This paper focuses on the reconstruction of a 3D object from multiple scans, while we simulate this whole process in a virtual environment. Firstly the user picks points, where virtual scans should be captured. These points are then reached with a robotic arm manipulator with a virtual scanner as its end effector. These scans are then aligned and reconstructed. Simulating this process in a virtual environment can help in evaluating whether picked points

[*]boris.sliz1@gmail.com
[†]martin.madaras@gmail.com

can capture enough parts to reconstruct the whole scanning object.

One of the requirements of our system is to be able to simulate any model of a robotic manipulator. This enables us to easily switch between different robots, which allows fast and simple testing. All that is necessary is to add new models of robotic parts and setup basic parameters (joint angle rotation, length, etc.).

Capturing vast amounts of real scans can be expensive and time-consuming, so virtual scanning can also help in an easy acquisition of data to test algorithms. Scans that we create through computer simulation are also cheaper, faster, and way more flexible. They are ideal for large in analysis systems (e.g., Monte Carlo), where algorithms need to run thousands of times.

We also utilize physics simulation to spawn parts into a bin randomly. Automated bin picking solutions need to localize these parts so a robotic manipulator can pick them. This robot needs to see these parts, and it needs to recognize their location. We can then use these virtual scans to test the localization of these parts.

## 2 Related work

There are many different ways to simulate 3D scanning, for example, getting points in the scene with raycasting or unprojecting data from the depth buffer. Raycasting approach [6] is based on algorithms that calculate the intersection of a ray cast from a virtual camera and objects in the scene. Camera casts $R_0,...,R_n$ rays in shape of planar cone and intersections of these rays are saved as points into the point cloud. These rays are then easily altered, which can simulate outliers created by real scanners. One option changes the angle of ray right after it is cast. The second option changes the position from where the ray is cast.

The approach using depth buffer is saving points in scene that are visible by the camera. Lidar OpenGL simulator [19] uses depth buffer to create a point cloud from a camera's point of view. This simulator is capturing only one object, and the result is very dependent on cutting planes of the camera. It uses additive noise in postprocessing - small random values are added to random points.

Robotic simulators like Gazebo [11] or RoboDK [9] can simulate the movement of the robotic hand manipulator with the implementation of inverse kinematics. These simulators usually use Jacobian inverse methods [4] to calculate inverse kinematics. The approximation method of inverse kinematics like Gradient descent [16] is not completely precise, but it can simulate imperfections of real robotic manipulators. This method first creates simulated rotation with all joints to all possible angles and then moves by chosen learning rate to a direction that gets end effector closest to the target. These robots are usually defined in URDF (unified robot description format) [14]. This format is an XML file that stores information about robot. It can define how many joints robot has, how much each of them weights, etc. These values can help create a realistic simulation. In virtual scanning simulations, where also the robotic hand is used, it is common to use also rotary table [3]. This table is useful when the robot is not able to reach behind the object that we want to scan. A combination of the robotic hand and rotary table allows capturing almost every part of the object, which is necessary in order to get good results in reconstruction.

Mentioned simulators also use physics engines so they can simulate collisions with high precision. One of the most popular physics engines is Bullet [2]. This engine calculates collisions of basic rigid bodies, but it also supports soft bodies and multi bodies (robot). Most of the precision of object collision behavior is because of the inertia matrix and center of mass, which define the object's behavior. UDRF files are containing these pieces of information.

## 3 Virtual scanning

We create our virtual simulation of scanning and reconstruction in OpenGL, where we capture scans through depth buffer values and camera parameters. We also use a method that modifies the basic OpenGL rendering pipeline. This modification happens in the fragment shader, where we store depth in 2 color channels and intensities of points in the remaining channel. To test another method of virtual scanning, we used raycasting in Unity to create point clouds.

### 3.1 Depth buffer

In this method, we implement a basic rendering OpenGL pipeline. In the virtual scene, the camera represents a virtual scanner, and scans are created based on data from the depth buffer. This data represents linear values in range 0-1. Later we will need to unproject camera coordinates to world coordinates, and for this, depth needs to be transformed based on near/far clipping planes. We can get this transformation with the following equation:

$$depth = n * f / (f + n - d * (f - n)) \tag{1}$$

where $d$ is value from depth buffer, $n$ is near clipping plane and $f$ is far clipping plane.

Clipping planes need to hug scanning objects or scenes as tight as possible because if the clipping plane would be much bigger than it is necessary, then the resulting point cloud would have depth values in the small range - which results in a flat point cloud.

To create a whole point cloud after the adjustment of the depth value, we need to find corresponding world coordinates. These coordinates can be acquired by unprojecting camera space. To do this, we use the camera's projection matrix and the corresponding value from the depth buffer.

To get the final point cloud, we save all points from unprojection into a custom .ply file. Point cloud captured like this is perfect (see Figure 1); it does not have any outliers, noise, or distortion for now, which could in the real world happen only in ideal conditions. Figure 1 represents one scan of the object. Later on, we will describe capturing multiple scans and aligning them for reconstruction.

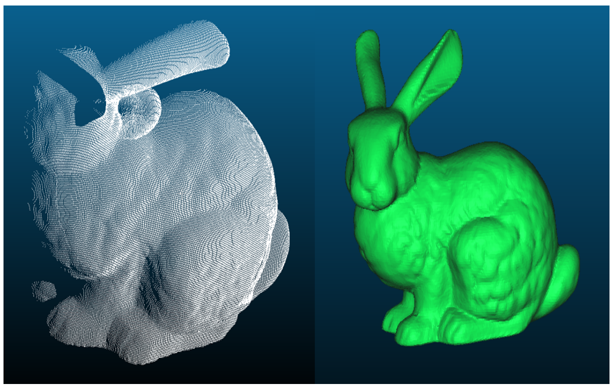
Figure 1: Virtual scan (left) and scanning object (right).

## 3.2 Color buffer

In this method, we are modifying fragment shader to store depth in the color buffer. The length of a light ray in the scene represents depth. This ray is cast from the camera position, so depth value represents the distance between camera and fragment. This is a similar method as it is in time of flight 3D scanners, but instead of the time of ray, we can use the length of ray since it is easier and possible in the virtual environment. To store depth value, we use two 8-bit channels.

Basic fragment shader calculates color in RGB channels. We override GB channels with a 16-bit depth value. This depth can then be acquired and transformed in the same way as it was when we were using the depth buffer. This way, we can get XYZ world coordinates and create a virtual point cloud. There is still R channel free, and we use it to store intensities of points. These intensities in point clouds represent the brightness of each point. In regular environments (without too many different lights), this brightness is getting lower when points are further from the camera. To simulate this, we use attenuation of light, which we cast from the camera position, so if it is further from the camera, its attenuation is lower. We represent intensity as R color multiplied by constant, linear, and quadratic attenuation. Figure 2 portrays the point cloud of the same object with the same camera configuration, as seen in Figure 1, but using a color buffer with intensities.

Results from scans using depth buffer and color buffer are almost identical, but using light rays to calculate depth is more robust, and it more simulates real 3D scanners. It is more accurate for objects that are further from the camera since depth buffer is only accurate when objects are close to the camera.

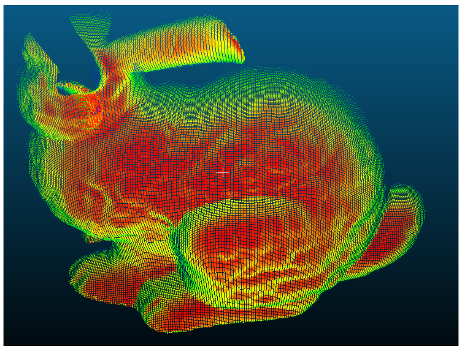
Figure 2: Virtual scan with intensities.

Both of these methods are fast because they do not require any hard computation when acquiring point clouds. This means that they both can be used in real-time applications.

These virtual scanners are in a virtual scene, and they are attached to the robotic hand manipulator, which is described in the next section. In this virtual environment, the user can pick multiple scanning points and multiple parameters. These parameters include noise, distortion, point cloud density, and alignment method.

## 3.3 Raycasting

To achieve world coordinates in the scene and create a point cloud, we can also use raycasting. We use Unity engine [18] because of its efficient and accurate implementation of raycasting. In Unity, we also create a scene with a robotic manipulator where users can pick points, where virtual scanner acquires point cloud. This point cloud is created by casting rays to all directions in cameras view and saving XYZ world coordinates after each ray hits the object. Resulting scan (see Figure 3) is very similar to the method where we use light rays and color buffer, but raycasting is computationally more expensive.

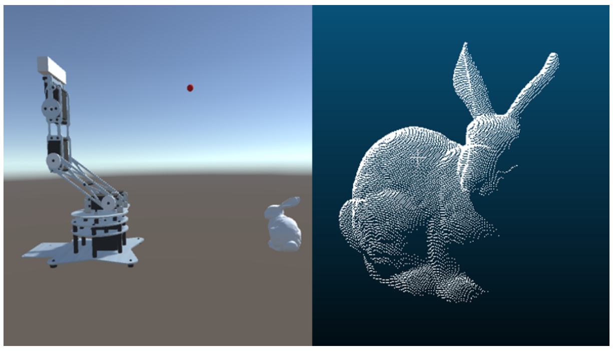
Figure 3: Scene in unity (left) and virtual scan (right).

## 3.4 Normals

The last part of the virtual scanning pipeline is a reconstruction, and to do that, our scans need to have normals. For each point in a point cloud, we pick $k$ nearest neighbors, that serve as points to estimate normals. We pick these neighboring points in the loop where we go through each point and look in all directions. From these points, we remove outliers. As an outlier, we identify points that have higher than a specified distance from the referenced point. After the removal of outliers, we can compute the centroid of these points. Neighboring points and centroid are input data for singular value decomposition (SVD), and the left singular vector from SVD then represents the estimated normal.

## 3.5 Noise and distortion

To simulate 3D scanners correctly, the resulting point cloud should have some noise and distortion. To simulate noise, we are using Gaussian noise. It modifies normally distributed random points by a small value. To simulate distortion, we multiply XY coordinates of each point by radial and tangential distortion coefficients. This whole process can be seen in Figure 5; the scan on the left is shown with simulated Gaussian noise and on the right with simulated distortion. Distortion in Figure 5 is scaled up for illustration. Scanners usually do not have distortions that high.

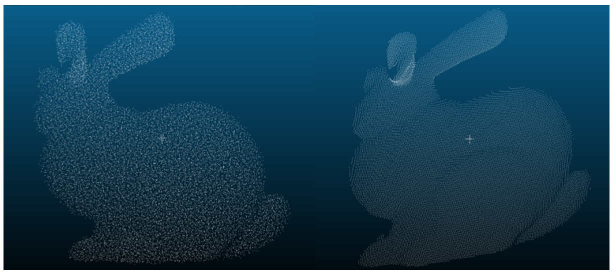
Figure 4: Point cloud with noise (left) and with distortion (right).

## 3.6 Alignment and reconstruction

To align virtual scans, we compute initial transformations with combination with ICP algorithm [15]. The first method that computes initial transformation takes the rotation of the model that we scan and the transformation of the camera. For the camera, we need its position and inverse rotation so that we can transform the scan to the initial rotation.

Keypoint matching is the second method that we use to get the initial transformation. This method contains four steps:

- Find keypoints
- Create descriptors of keypoints
- Match keypoints
- Calculate transformation

To find keypoints in a point cloud, we use SHOT algorithm [12]. This algorithm also works as a descriptor for keypoints, so it is holding all information about each keypoint (rotation, scale, etc.). Keypoints of two corresponding point clouds match when their descriptors are equal or very similar. There is a lot of outliers in these matches. For an outlier removal, we use RANSAC algorithm [7]. RANSAC returns four corresponding points with information about the transformation between these points. Alignment of all of the point clouds returns one union point cloud that can be reconstructed.

To reconstruct the aligned union point cloud, we use Poisson reconstruction [10]. In Figure 5 this whole process can be seen, from aligning two corresponding point clouds to reconstructing the union point cloud.

# 4 Robot manipulator

Our implementation of a robotic manipulator had to be generic so that it would be possible to add any robot model with any type of joints. First part is devoted to its initialization and second part to its positioning and movement.

## 4.1 Robot model

For every part of the robot model, it is necessary to set if it is a base, joint, or end effector. The base of the robot is static and always stays in the same position and rotation. Joints are usually able to rotate only in one direction. The end effector can be dynamic, so it is acting as a joint, or it can be static. For example, welder robots usually have dynamic end effectors, but when we connect a 3D scanner to a robot, it acts as an end effector, so it is static.

After setting up a robot in the virtual environment (Figure 6), we implement kinematics that will define its position and movement.

## 4.2 Kinematics

After setting up the robot model, we needed to implement its movement. The goal is to get end effector to position that is picked by the user. Forward kinematics checks where is the position of an end effector, and inverse kinematics takes care of the movement of joints. Forward kinematics is simple because it always has only one correct solution, inverse kinematics usually have multiple. To calculate forward kinematics, we need to find positions of joints based on their rotations. Forward kinematics assumes that we know the length of all joints, the rotation of each joint, and the position of the base. Based on these

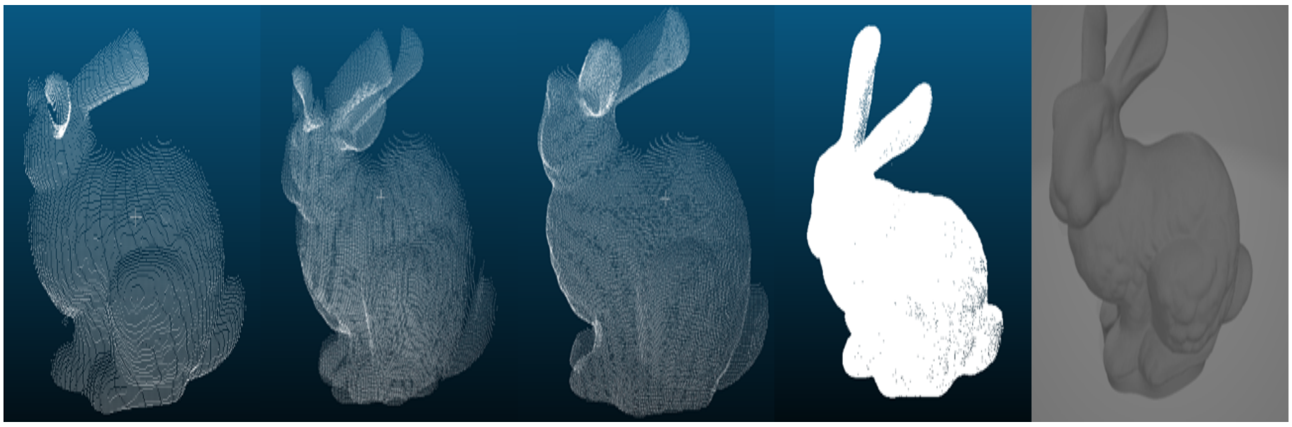

Figure 5: From left to right: single point cloud, 2 unaligned clouds, 2 aligned clouds, 35 aligned clouds, reconstructed aligned clouds.

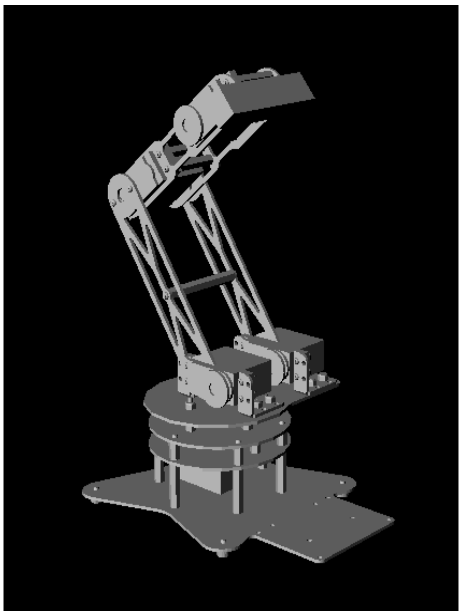

Figure 6: Robot hand in virtual scene.

assumptions, it is possible to calculate the position of the remaining joints with the following equation:

$$P_i = P_{i-1} + rotate(D_i, P_{i-1}, \sum_{k=0}^{i-1} \alpha_k) \tag{2}$$

$P$ represents the position of joints, $D$ is the length of each joint, and $\alpha$ are angles of previous joints. After iterating through all joints, we can get the position of the end effector.

In a virtual environment, it is also possible to just get the position of an end effector since its more comfortable, straight forward and has the same result.

To implement inverse kinematics, we used a Gradient descent algorithm. In kinematics, this algorithm firstly virtually rotates joint to all possible directions (if the joint can move in one angle, there are only two possible directions). Then Gradient descent checks which rotation made the robot closer to the target point, so we just compare the distance of two positions. This joint then rotates, and the value of this rotation is called the learning rate. This value is usually minimal because significant learning rates can miss the target position. Each joint rotates by a value of the learning rate, but it starts in a joint that is close to the end effector because this joint moves the whole robot least. If we start with the lowest joint in robot hierarchy, we risk missing the target point due to significant movement.

After the implementation of robot initialization and its movement, we had to add the possibility for the user to pick points where a virtual scanner captures the scans. We added a target point to a virtual scene that can be easily moved. Users can move this target around and pick multiple points. Except for these points, it is possible also to pick how many times will be object rotated and by how much. After points and rotations are picked, virtual scanning can start. A virtual scanner is located at the end of the robotic hand, so it is its end effector. This scanner will reach all picked points for each rotation. So let us say if we picked four points and ten object rotations by 20 degrees, then the simulation will capture 40 scans. Each robot calculates the radius of its reach so it can notify users if they pick unreachable points. After obtaining each scan, the computation of initial transformation and normals starts. These scans are being aligned on the separate thread while the process of scanning is still running. After alignment of each scan, it is added to the visualizer, so the user can see in real-time how the creation of union point cloud runs. Reconstruction starts when all scans are captured and aligned. After the finish of reconstruction, we add the resulting mesh to the visualizer, so it is possible to inspect it.

## 5 Virtual scanning in physics simulation

This chapter describes the implementation of a virtual scanner in a physics simulation that spawns parts into the bin. This simulation shows another possible usage of the synthetic point clouds, and it is to test localization algorithms. Localization is a method of locating objects in a point cloud. In industry, this is often used on the identification of the location of parts/objects that are randomly stored in a container or bin. We use this use case as a possibility to check if our virtual scanning methods can be used for multiple applications.

To create this simulation, we use physics engine Bullet [2], which can import objects as rigid bodies and compute their collisions. We use our virtual scanner on these parts so it can be used to test localization, which is a method of locating objects in a point cloud.

We implement this simulation in multiple environments, and one of them is the Gazebo simulator. This simulator uses two physics engines to calculate collisions, Bullet and ODE (Open Dynamics Engine) [8]. To spawn objects into the scene, we use C++ plugin that connects to the simulation. This plugin sends messages to a simulator with the specification of positions and orientations of objects. The behavior of these objects is based on their URDF or SDF (Simulation description format) files. SDF is an XML format that describes objects and environments for robot simulations, visualization, and control. These files specify information about the object's physics parameters, which include gravity, friction, and inertial parameters (inertia matrix, mass, and center of mass). The simulation spawns objects until parts have filled the bin. A virtual scan is created after the last part is spawned.

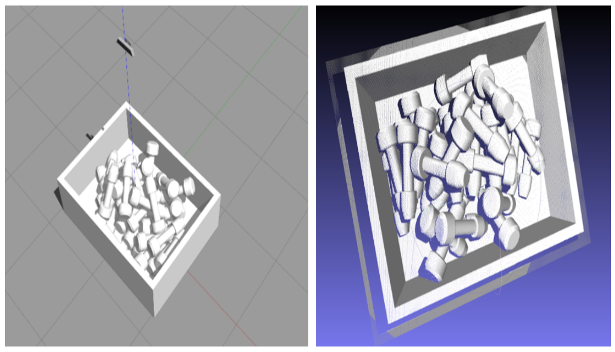
Figure 7: Bin filled with parts (left) and captured point cloud (right) in Gazebo.

To create a virtual scan in Gazebo, we use its simulated depth camera (see Figure 7). In the Gazebo scene, we can add and set up this camera and start the simulation. While simulation is running, it streams messages that can be captured by using C++ plugin. We use this plugin to read from these messages, obtain XYZ coordinates, and create a point cloud.

Collision simulation in Gazebo has high precision, but there is a significant overhead in communication between C++ and simulator, which makes simulation run very slow. To improve the speed of simulation, we also used the Bullet engine separately without the Gazebo simulator. Using the Bullet library and PyBullet [5], which is a Bullet Python wrapper, we managed to create the same simulation without communication overhead.

In Bullet, we first import the URDF file of the object. This file is then parsed and converted to Bullet format. The loaded mesh needs to be transformed into convex hull mesh since Bullet supports dynamic collisions only for basic shapes (cube, cone, etc.) and the convex hull of the mesh. Collisions for triangulated objects are allowed only for static objects. It is possible to use approximate convex hull decomposition [17] to decompose object into multiple smaller objects, where a combination of theirs convex hulls creates shape similar to original triangulation mesh.

To capture a virtual scan of the bin, we use PyBullet's OpenGL renderer. This renderer allows us to read values from the depth buffer. We use these values to create a virtual scan that captures parts that are inside the bin (see Figure 8).

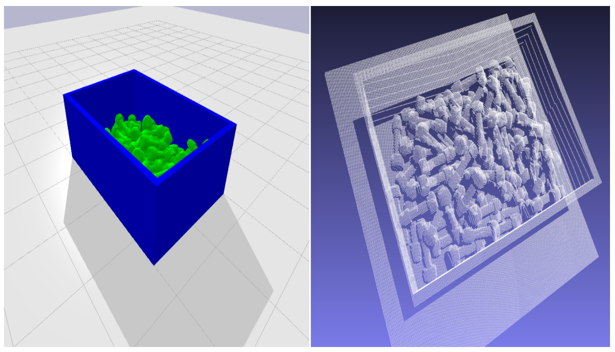
Figure 8: Bin filled with parts (left) and captured point cloud (right) using Bullet physics engine.

## 6 Experiments

The data used to analyze the virtual scanning and reconstruction is captured by Photoneo's PhoXi 3D scanner[1]. In this section, we compare the reconstruction of an object in a virtual environment with reconstruction using a PhoXi 3D scanner. To test localization, we create a random scene with parts inside the bin. The virtual scanner captures the point cloud of the bin and then uses localization to find parts.

To test possible use of virtual scanning and reconstruction, we need to compare our virtual results with some object that is captured and reconstructed with real scanners. We have available two PhoXi 3D scanners with a rotary table to use for testing. In the first step, we simulate the

[1]https://www.photoneo.com/phoxi-3d-scanner/

scanning process in our application, where we choose two points in the virtual scene. From these points, a virtual scanner captures multiple scans of object placed on the scene. The number of these scans depends on how many rotations of the model we select.

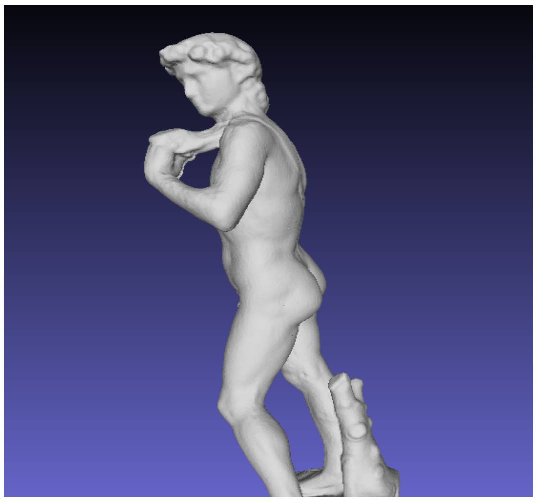
Figure 9: 3D model of David by Michelangelo reconstructed with 20 virtual scans.

Figure 9 shows a reconstructed object from two different views with ten rotations, so it is created from 20 scans. Since this model meets the requirements (it does not have any missing parts), we can move to the real scanners. To get the correct comparison, these scanners need to copy the pose of virtual scanners as much as possible. The relative position to the model and up vector of the virtual scanners needs to be set similarly for real scanners. This would be easy with the robotic hand since it gets scanner to this pose automatically. By using calibration with a checkerboard, we managed to place scanners to pose of the scanners from the virtual environment. When the pose of the scanners is set, we just need to choose the number of object rotations, and scanning can start.

Figure 10 shows a reconstructed object from virtual scans (left) and reconstructed object from real scans (right). This comparison shows that simulating the whole process of 3D reconstruction can help us predict results that we get from real 3D scanners.

To check if it is possible to use a virtual scanner for another application usage, we capture a virtual scan of the bin with randomly spawned parts and afterward run localization on this scan. To test object localization, we use Photoneo's 3D Localization SDK[2]. This algorithm is able to find a specified 3D objects in the virtual point cloud that we captured. Figure 11 shows the localization of the object in a virtually created point cloud.

[2]https://www.photoneo.com/3d-scanning-software/#localization/

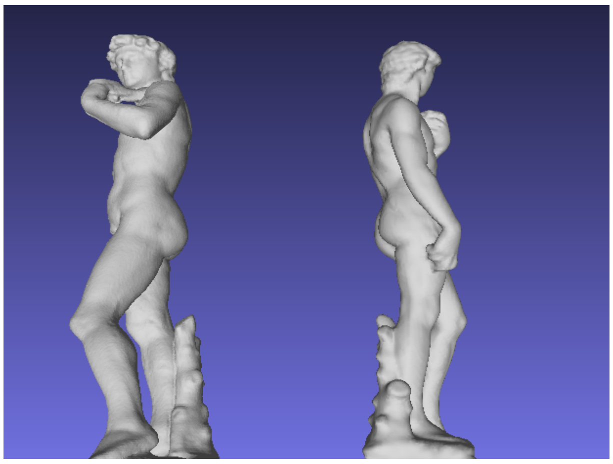
Figure 10: Object reconstructed from virtual (left) and real (right) scans.

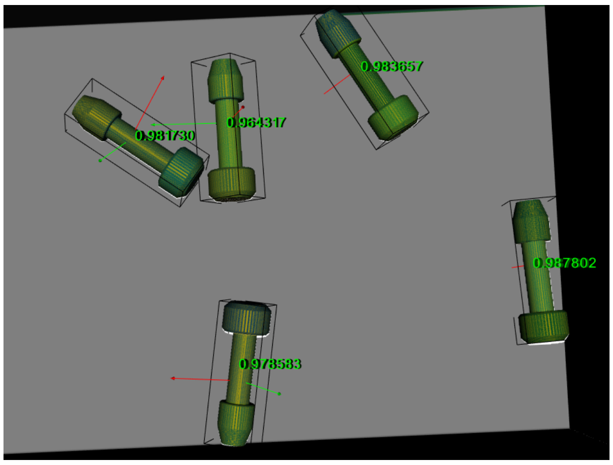

Figure 11: Reference object localized in the virtual scan.

## 7 Conclusion

This paper describes how we can simulate the process of virtual scanning and how we can use it. We presented multiple approaches of virtual scanning. In one of them, we use the basic OpenGL rendering pipeline, where we create point clouds from depth buffer or modified color buffer. To test and compare other methods, we also implement a virtual scanner in the Gazebo (depth camera) and Unity (raycasting) engines. In the second part, we focus on the simulation of a robotic hand manipulator, which we use to capture scans from multiple points.

Most of the existing solutions on simulating 3D scanners focus only on the creation of synthetic point clouds. We propose a solution that combines the simulation of a robotic hand, virtual scanning, alignment, and reconstruction. This simulation can be used as a test environment before the use of real systems. It can help to speed up the process of setting up robot, where we need to pick enough points to capture point clouds that can successfully reconstruct object.

To show another application use of a virtual scanner, we create a physics simulation where we spawn multiple parts into the bin. This simulation uses Bullet physics engine to compute collision detection. After this bin is filled with parts, we can create virtual scans. These scans can then be used to test the localization of these parts and their gripping with the robotic hand.

## 8 Future work

There are many possible ways of improvement in the case of the virtual scanner. We simulate real scanners by adding noise and distortion in the postprocessing part, but to create a more realistic virtual scanner, it would be necessary to add a complete simulation of scanner physics and also to consider all parameters on the scene, for example, the material of objects.

## References

[1] David Acosta, Olmer García, and Jorge Aponte. Laser triangulation for shape acquisition in a 3d scanner plus scan. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA'06)*, volume 2, pages 14–19. IEEE, 2006.

[2] Adrian Boeing and Thomas Bräunl. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 281–288, 2007.

[3] Theodor Borangiu, Anamaria Dogar, and Alexandru Dumitrache. Modeling and simulation of short range 3d triangulation-based laser scanning system. *Proceedings of ICCCC*, 8:190–195, 2008.

[4] Samuel R Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17(1-19):16, 2004.

[5] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*, 2016.

[6] Malvin Danhof, Tarek Schneider, Pascal Laube, and Georg Umlauf. A virtual-reality 3d-laser-scan simulation. *BW-CAR— SINCOM*, page 68, 2015.

[7] Konstantinos G Derpanis. Overview of the ransac algorithm. *Image Rochester NY*, 4(1):2–3, 2010.

[8] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 4397–4404. IEEE, 2015.

[9] Beata Jakubiec. Application of simulation models for programming of robots. In *Proceedings of the International Scientific Conference. Volume V*, volume 283, page 292, 2018.

[10] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006.

[11] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[12] Gentao Liu, Xiangming Wen, Wei Zheng, and Peizhou He. Shot boundary detection and keyframe extraction based on scale invariant feature transform. In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, pages 1126–1130. IEEE, 2009.

[13] Aongus McCarthy, Robert J Collins, Nils J Krichel, Verónica Fernández, Andrew M Wallace, and Gerald S Buller. Long-range time-of-flight scanning sensor based on high-speed time-correlated single-photon counting. *Applied optics*, 48(32):6241–6251, 2009.

[14] Michael Mortimer, Ben Horan, Matthew Joordens, and Alex Stojcevski. Searching baxter's urdf robot joint and link tree for active serial chains. In *2015 10th System of Systems Engineering Conference (SoSE)*, pages 428–433. IEEE, 2015.

[15] Aleksandr Segal, Dirk Haehnel, and Sebastian Thrun. Generalized-icp. In *Robotics: science and systems*, volume 2, page 435. Seattle, WA, 2009.

[16] Gaurav Tevatia and Stefan Schaal. Inverse kinematics for humanoid robots. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 1, pages 294–299. IEEE, 2000.

[17] Daniel Thul, Lubor Ladicky, Sohyeon Jeong, and Marc Pollefeys. Approximate convex decomposition and transfer for animated meshes. *ACM Transactions on Graphics (TOG)*, 37(6):1–10, 2018.

[18] Unity Technologies. Unity 2019.1.6.

[19] John O Woods and John A Christian. Glidar: An opengl-based, real-time, and open source 3d sensor simulator for testing computer vision algorithms. *Journal of Imaging*, 2(1):5, 2016.