

Ministry of Education and Science of Ukraine
NATIONAL UNIVERSITY OF KYIV-MOHYLA ACADEMY
Department of Informatics of the Faculty of Informatics

**USE OF AUGMENTED REALITY TO BUILD AN INTERACTIVE
INTERIOR ON THE IOS MOBILE PLATFORM**

**Text part to thesis in the specialty
"Software Engineering"**

Supervisor of the thesis
Ph.D. art. Off. Frankiv O. A.

(signature)

" ____ " _____ 2022

Made by student Babii V. K.

" ____ " _____ 2022

Kyiv 2020

Ministry of Education and Science of Ukraine
NATIONAL UNIVERSITY OF KYIV-MOHYLA ACADEMY
Department of Informatics of the Faculty of Informatics

Approved
Head of Department of Informatics,
Associate Professor, Ph.D.
_____ S. S. Gorokhovskiy
(signature) " _____ 2022.

INDIVIDUAL TASK

for thesis

student Babii Veronika 4th year of the Faculty of Informatics

TOPIC: USE OF AUGMENTED REALITY TO BUILD AN INTERACTIVE
INTERIOR ON THE IOS MOBILE PLATFORM

The content of the PM to the thesis:

Abstract

Introduction

Section 1. Analysis of the problem.

Section 2. Theoretical information.

Section 3. Description of practical research.

Conclusions and analysis of opportunities for further development

List of sources

Glossary

Applications

Date of issue " ____ " _____

2022

Supervisor _____

Task received _____

Schedule of thesis

Topic: Use of augmented reality to build an interactive interior on the iOS mobile platform.

No p/p	Name of the stage of the thesis	Deadline for the stage	Note
1.	Getting a task for coursework.	October 2021	
2.	Review of literature on the topic of work.	November - December 2021	
3.	Analysis of programs that solve the problem.	January 2022	
4.	Research on possible approaches.	January - February 2022	
5.	Writing preprocessing part – point cloud and 3D ply model creation.	February 2022	
6.	Writing postprocessing part – 3D mesh reconstruction.	February 2022	
7.	Writing the text part of the thesis.	April - May 2022	
8.	Thesis presentation creation.	May 2022	
9.	Thesis statement.	May 2022	

Content

<i>Abstract</i>	<i>5</i>
<i>Introduction</i>	<i>6</i>
<i>Section 1. Analysis of the subject area and setting the task of course work</i>	<i>8</i>
1. Analysis of the problem.	8
2. Analysis of the features of scanning applications and existing analogues of the application.	9
3. Analysis of possible approaches to create 3D model in iOS environment.	11
4. Setting the task of course work	12
<i>Section 2. Theoretical information</i>	<i>13</i>
1. Depth estimation.	13
2. True Depth camera.	13
3. LiDAR scanner.	14
4. ARKit.	15
5. Point cloud.	15
6. Rendering pipeline.	16
7. Metal Shading Language.	17
8. File formats.	18
9. 3D surface reconstruction.	21
10. Open3D library.	24
<i>Section 3. Description of practical research</i>	<i>25</i>
1. Practical research on possible approaches to create 3D model.	25
2. Point cloud creation.	29
3. Surface reconstruction of the point cloud.	32
<i>Conclusions and analysis of opportunities for further development</i>	<i>37</i>
<i>List of used literature</i>	<i>38</i>
<i>Glossary.....</i>	<i>41</i>
<i>Applications.....</i>	<i>42</i>
Appendix A. Points placement preparation	42
Appendix B. Coordinates unprojection	43
Appendix C. More points accumulation in case of camera rotation.....	44
Appendix D. Saving point cloud model to a .ply file.....	45
Appendix E. Common surface reconstruction pipeline. [26]	46

Abstract

The thesis is devoted to the study of possibilities of the 3D object creation from the real-life surroundings on the Apple iOS platform.

The research is based on the development of a mobile application for the iOS operating system for 3D model creation of the surroundings via the help of the LiDAR Scanner.

Keywords: mobile application development, iOS operating system, Swift programming language, augmented reality, ARKit, RealityKit, MetalKit, depth data, LiDAR Scanner, point cloud, mesh, 3D model, shader, CPU, GPU, open3D library, mesh reconstruction, Metal shading language, normals estimation, point cloud downsampling, BPA algorithm, Poisson algorithm, objects formats: .obj, .ply, .dae, .usdz.

Introduction

Augmented reality technologies swiftly enter all spheres of everyday life due to the popularity of AR/VR market, the rapid development of technological capabilities, and the convenience of the usage of augmented reality technologies in order to simplify everyday tasks. Virtualization brings the everyday life into the mobile devices and makes a lot of things available from anywhere in the world.

One of the popular topics nowadays is creation of 3D models of real-world objects and environments for the further use. With the help of the Apple devices' cameras and the brand-new LiDAR scanner technology it is possible to obtain detailed information about the depth of the objects at the distance up to 5 meters. That makes it possible to create a precise 3D model of the real-life environments, such as premises, apartments or streets.

This technology has many potential use cases, such as 3D models creation of the apartments for virtual navigation in them, the placement of another virtual models inside the scanned virtual environments, convenient storage of detailed plans of the rooms or apartments, creation of the 3D models of technical rooms for virtual training inside them, and so on. The wide range of applications of this solution and the rapid development of AR technologies allow to judge about the relevance of this approach and its usefulness in both everyday life and business solutions.

Scientific significance of the thesis lies in the conduction of the comprehensive research on all possible ways to create 3D model with the help of mobile device camera using iOS technologies available as per 2022 year. Practical significance is in the development of the app that can create a 3D point cloud model out of scene depth information and the script to reconstruct a full-fledged 3D model from the point cloud in .ply format.

The purpose of the thesis is to study ways to create interactive 3D model and to develop an iOS mobile application based on the chosen approach.

Tasks of the thesis:

1. Research on the technologies and approaches on how to create 3D model from the camera image.
2. Compare possible solutions and explore chosen one.
3. Develop an iOS app with the implementation of the chosen approach.

The object of the research is an approach for obtaining a 3D model with the camera of a mobile device using modern solutions available on the iOS platform.

The subject of the study is a mobile iOS application for creation an interactive 3D model by scanning surroundings via mobile device camera.

The practical importance of the work is the possibility of further use of the developed application in order to obtain a 3D model from the depth information from the LiDAR Scanner or to conduct further research based on the provided solution.

Thesis consists of three sections.

The first section studies the problem of creation of 3D model on the mobile device, analysis of scanning applications and setting the task of course work.

The second section is devoted to theoretical information about the Lidar scanner, its capabilities to get information about the depth in the scene, to point cloud creation, Apple frameworks and possible solutions that can solve the problem of 3d model creation.

The third section describes the process of the application implementation that can gain depth information and form a full-fledged 3D model from it.

Section 1. Analysis of the subject area and setting the task of course work

1. Analysis of the problem.

Replicating the structure and appearance of a real-life object by creating its 3D model is a hot topic and became even hotter with the evolution of AR/VR sphere in the last years and game industry during the past decade. Quality of game meshes and virtual surrounding are getting more advanced every year and computing power of the mobile devices too. This opens a wide range of opportunities for instant use of scanned real-life objects or surroundings in a games or projects, without the necessity to hire a 3D modelers team in order to recreate a specific object. Created 3D model can be immersed into the game virtual reality directly after its scanning, or otherwise - scanned surroundings can be used as game map instantly after scanning - without the need in the post-processing.

3D object scanning has a great potential and become more applicable in different spheres of everyday life or corporate business. For example, scanned flats can be successfully used in such spheres as interior design or real estate. Recreation of the flats as an interactive 3D models would ease the life of realtors, because customers can see real sized rooms and all details inside them, and even navigate themselves inside these flats' models.

In order to successfully enter the market, this solution of a clear 3D model creation, firstly, should be possible with basic iOS device for the use of ordinary users. Secondly, the time it takes to create the model should be within the appropriate limits. The format of the model should be suitable for further use, and the quality of the model should be high enough to see the details, but at the same time the size of the generated file should not exceed the adequate limits. In this paper we will consider approaches to meet these requirements.

2. Analysis of the features of scanning applications and existing analogues of the application.

During the last few years, due to the rapidly growing computational power of mobile devices and constant improvements of their cameras, on the market appear more and more applications that open the new possibilities of these devices. One such popular direction is the work with the augmented reality on mobile devices, and especially an immersion of the virtual objects into the real environment, or, conversely, the creation of virtual interactive models from the real environment.

Previously, the creation of a model of the scanned object was only possible with massive and expensive scanners, which were mainly used only in large enterprises. That is not convenient for everyday usage of an average customer.

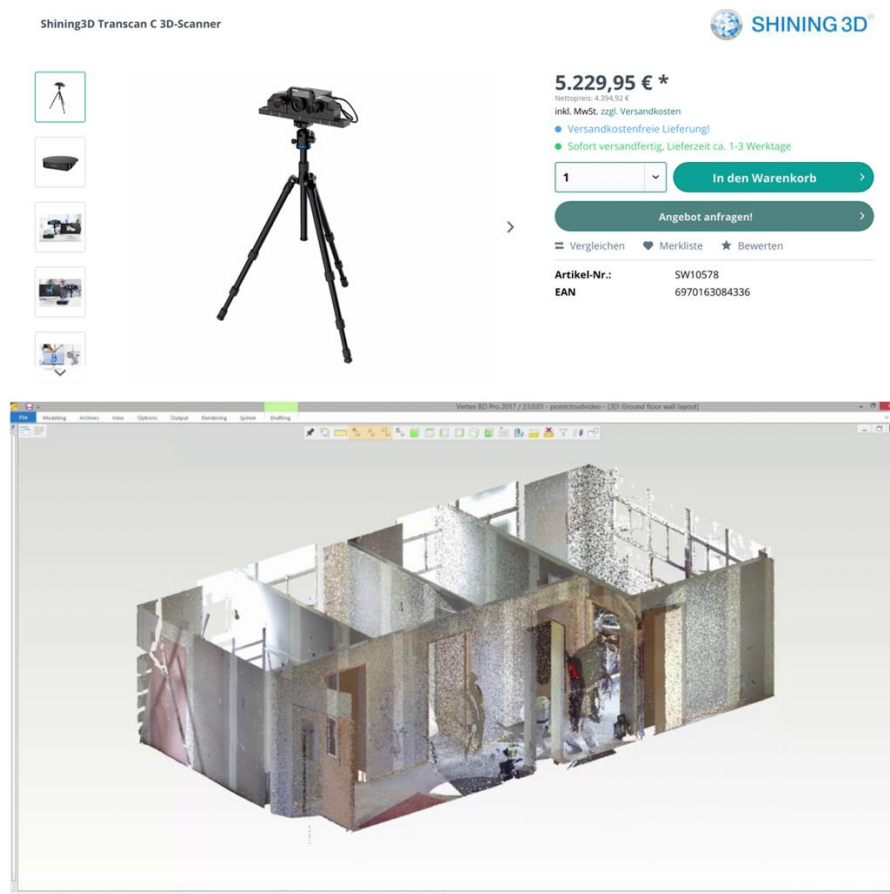


Figure 1. Shining 3D scanner and result of its work.

With the introduction of the LiDAR Scanner in the cameras of the new generations of Apple devices, the massive scanners have been put on the back burner. Since the LiDAR scanner has made it possible to obtain clear information about the depth to the object at a greater distance. This innovation opened a wide range of opportunities for more studies and development in this area for developers and to the greater use of developed applications for users.

There are already applications on the market that use LiDAR sensors along with the device's cameras to create 3D model of objects, such as Polycam and 3D Scanner.

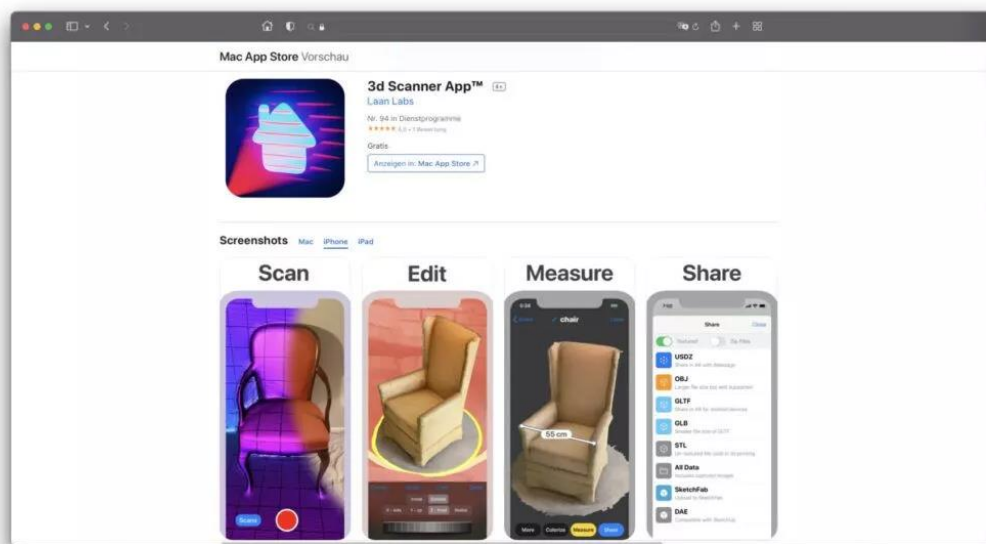


Figure 2. 3D Scanner iOS application.

The problem in the mentioned above apps is that they use custom approaches with the usage of the hard computations and integrations that might involve a solid team of developers, such solutions are mostly written in C++ programming language and such projects as theirs are closed private B2B solutions. Therefore, these apps either fully paid, or include purchases inside an app.

The research will focus on finding an uncomplicated solution that does not require a large budget or a team of developers to create and maintain it, and on Apple iOS devices market, and therefore the native iOS technologies and

approaches that might provide solution for the creation of the 3D model via scanning real life object.

3. Analysis of possible approaches to create 3D model in iOS environment.

The computing power of today's devices allows for the effective implementation of augmented reality elements in both iOS and Android solutions. The native frameworks of both platforms support the latest features in the field of AR. But still, Apple devices are superior to Android in terms of the speed of development of their device's hardware and especially the camera. From 2020 Apple began to add high-tech LiDAR scanners to their devices that allow to work better with the depth of the real-world objects at a greater distance. In this paper, the emphasis will be placed on the study of possible approaches to work with the augmented reality on the Apple platform using the new LiDAR scanner, namely the creation of a 3D model of the scanned environment.

Apple provides two frameworks for working with augmented reality - ARKit and RealityKit. ARKit is SDK that provides and processes sensor data necessary for AR experiences to work. While RealityKit is a higher-level SDK that provides some game engine functionality for AR apps, like input, multiplayer, audio, etc.

RealityKit framework offers Object Capture API that is able to create 3D model from photos taken from many different angles using a photogrammetry. It analyses the overlap area between different images to match up landmarks, and then produces a 3D model of the photographed object. API only works on macOS, so it's not an option for the research because processing cannot be done on a mobile device.

ARKit framework offers three approaches.

First one is Scene Reconstruction API. It doesn't provide an ability to export created mesh with the object texture, therefore it's not suitable for proper 3D model creation.

Second possible approach to look at is an .arobject creation. API recognizes features of the user's environment and use them to trigger the appearance of virtual content. Created .arobject model doesn't contain any 3D geometry information and cannot be displayed in any way later.

Third approach is a point cloud creation by presenting a visualization of the physical environment by placing points based a scene's depth data. This solution meets goals of the work as with this solution it is possible to gather structured point cloud, that later can be reconstructed in proper 3D mesh.

4. Setting the task of course work

After analyzing popular trends on the market, technological capabilities of modern devices and software development trends, it was decided to investigate the possibility of creating virtual environments for the iPhone mobile platform based on the new generation of devices with a LiDAR scanner.

The task of the thesis is to investigate the possibilities of creating a 3D model on a mobile device using native iOS solutions and to create a corresponding application based on a chosen approach.

Section 2. Theoretical information

1. Depth estimation.

Our eyes estimate depth by comparing the image obtained by our left and right eye. The minor displacement between both viewpoints is enough to calculate an approximate depth map. We refer to the pair of images our eyes receive as a stereo pair. This, combined with our lens with variable focal length, and general "seeing things" experience, allows us to have a seamless 3D vision.

In 3D computer graphics and computer vision, a depth data is a map of per-pixel data containing depth-related information.

Some devices have two cameras separated by a small distance, such as iPhone 7 and 8, to capture images from different viewpoints. These two images form a stereo pair that is used to compute depth information. iOS devices with a back-facing dual camera or a front-facing TrueDepth camera can capture this depth data. [1]

2. True Depth camera.

The TrueDepth refers to front-facing cameras with a dot projector in the Apple devices. It provides depth data in a real time that allows to determine the distance of a pixel from the front-facing camera. Camera projects an infrared light pattern in front of itself and images that pattern with an infrared camera. By observing how the pattern is distorted by objects in the scene, the capture system can calculate the distance from the camera to each point in the image. [2]

TrueDepth cameras were first introduced in the iPhone X in November 2017. They support Portrait Mode photography and Face ID facial recognition on the iPhone X and later, and the 3rd generation iPad Pro and later.

Dual cameras on earlier models, such as the iPhone 7 Plus and 8 Plus, were able to be used to calculate depth data to support some of the features, such as Portrait Mode, but this only was applicable to the rear-facing cameras. [3]

Apple AVFoundation framework introduced depth data capture for photos and video in iOS 11. The data it provides is suitable for many apps but may not

meet the needs of those that require greater precision depth. Starting in iOS 15.4, it is possible to access the LiDAR camera on the supporting hardware that offers high-precision depth data suitable for use cases like room scanning and measurement. [4]

3. LiDAR scanner.

A good alternative to multiple cameras is to use sensors that can infer distance, such as LiDAR, which stands for Light Detection And Ranging. It uses pulsed laser to send out pulses of light, and receiver to pick them up that gives accurate depth information. LiDAR can measure a variable distances to surrounding objects up to 5.0 meters away. It operates at nano-second speed — from 0.2 to 5 ns — that means there are hundreds of millions of pulses per second. [5]

In 2020, Apple Inc. released the iPad Pro 2020 and the iPhone 12 Pro with novel build-in LiDAR sensors. The technology uses a similar method to its famous Face ID face recognition, but somehow differs in a lot of ways. LiDAR Scanner is composed of two modules with its lens mounted overlapping one another. Consisting of a transmitter and a receiver sensor, with the first emits a series of points in the infrared which are detected by the sensor. The depth mapping LiDAR is capable of, is made purposefully for a larger range depth mapping over a wider range, instead of a more in-depth and finer measurements for scanning a face. [6]

In recent years many Android devices were equipped with iToF's that can be useful in ARCore apps. Apple's LiDAR is basically a direct Time-of-Flight (dToF) sensor. The main difference between direct Time-of-Flight and indirect Time-of-Flight image sensors is in that iToF sensor sends out continuous and modulated light and measures the phase of the reflected light for calculating a distance to an object, whereas dToF sensor sends out short pulses of light that last just a few nanoseconds and then measures the time it takes for some of the emitted

light to come back. iToF measures the phase shift, while dToF measures the direct time of flight. So dToF has a considerably higher accuracy than iToF. [7]

Devices such as the iPhone 12 and 13 Pro and Pro Max, and iPad Pro can use the LiDAR Scanner to calculate the distance of real-world objects from the user. In world-tracking experiences on iOS 14, ARKit provides a buffer that describes the objects' distance from the device in meters.

4. ARKit.

If `frameSemantics` of the `ARWorldTrackingConfiguration` for `ARSession` is set to `.sceneDepth` or `.smoothedSceneDepth` - ARKit provides a the depth buffer (`depthMap`) as a `CVPixelBuffer` on the current frame's `sceneDepth` or `smoothedSceneDepth` property, depending on the enabled frame semantics. Every pixel in the depth buffer maps to a region of the visible scene, which defines that region's distance from the device in meters.

The raw depth values in `sceneDepth` can create the impression of a flicker effect, but the process of averaging the depth differences across frames - via `.smoothedSceneDepth` option - smooths the visual effect. [8]

5. Point cloud.

A 3D point cloud is a set of points in three-dimensional space. The point cloud represents the surface of an object in 3D. Each point consists of three coordinates that uniquely identify its location. Additional information such as RGB color values and surface normal can be embedded as point attributes.

According to depth information from the device's LiDAR Scanner for every distance sample in the session's periodic depth reading, information about every pixel in `depthMap` such as its position and depth is recorded and can be linked with appropriate pixel color information from `capturedImage`. Each vertex calculates its x and y location in the camera image by converting its position in the one-dimensional vertex array, to a 2D position in the depth texture. Therefore object's point cloud is created.

6. Rendering pipeline.

3D point cloud is made up of vertices. Each vertex refers to a point in 3D space, made up of x, y and z values. These vertices will be read in by a 3D renderer and then passed to the GPU, where shader functions will process them to create the final image or texture to be sent back to the CPU and displayed on the screen.

Rendering - the processing of an outline image using color and shading to make it appear solid and three-dimensional. The entire process from importing a model's vertices to generating the final image on your screen, is commonly known as a rendering pipeline. The rendering pipeline is a list of commands sent to the GPU, along with resources such as vertices, materials and lights that make up the final image.

All computers have a Central Processing Unit (CPU) that drives the operations and manages the resources on a computer. And a Graphics Processing Unit (GPU) that is a specialized hardware component that can process images, videos and massive amounts of data really fast. Because it has a highly parallelized architecture, specialized in doing the same task repeatedly, and with little or no data transfers, it's able to process larger amounts of data. A CPU, on the other hand, can't handle massive amounts of data really fast, but it can process many sequential tasks really fast.

Also, the CPU typically only has a handful of cores while the GPU has hundreds — even thousands of cores. With more cores, the GPU can split the problem into many smaller parts, each running on a separate core in parallel. At the end of processing, the partial results are combined and the final result returned to the CPU.

GPU cores have special circuitry for processing geometry and are often called shader cores that writes a whole frame at a time to fit the entire rendering window and then proceed to rendering the next frame as quickly as possible to maintain a good frame rate. Each frame consists of commands that CPU issues to

the GPU. These commands are being wrapped in a render command encoder. Command encoders are being organized by command buffers that stand in a command queue. [10]

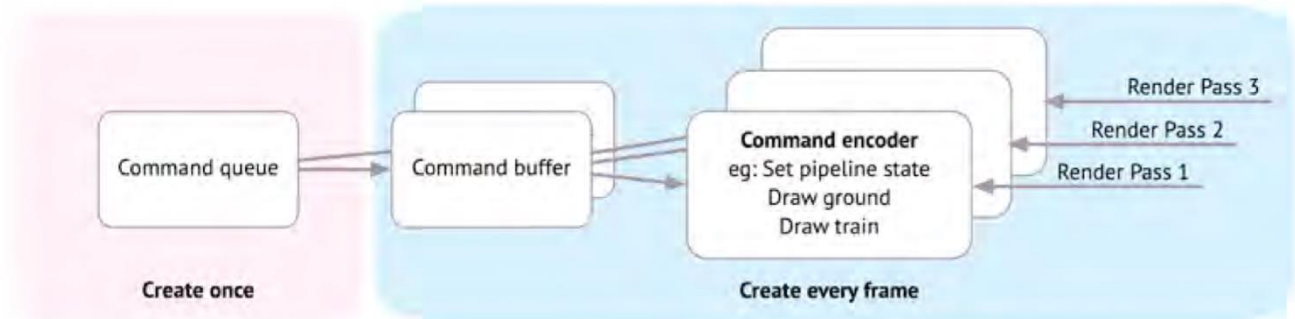


Figure 3. Rendering pipeline.

7. Metal Shading Language.

After the point cloud vertices are passed to the GPU, shader can process the vertices via shader functions in order to create the final image to be displayed. Shader functions are small programs that run on the GPU that are written in the Metal Shading Language.

Metal was announced at WWDC 2014 and was initially made available only on A7 or newer GPUs. Apple created a new language to program the GPU directly via shader functions. This is the Metal Shading Language (MSL) based on the C++11 specification. A year later at WWDC 2015, Apple announced two Metal sub-frameworks: MetalKit and Metal Performance Shaders (MPS). The API has continued to evolve, and WWDC 2017 introduced an exciting new version of the API: Metal 2. Metal 2 adds support for Virtual Reality (VR), Augmented Reality (AR) and accelerated machine learning (ML), among many new features.

In Metal, code that runs on GPUs is called a shader, because historically they were first used to calculate colors in 3D graphics. There are two types of shader functions in metal - vertex and fragment. The vertex function is where

you usually manipulate vertex positions, and the fragment function is where you specify the pixel color.

In order to pass created function to the processing on the GPU, you need to assign it to the pipeline descriptor. It holds all sorts of information that the GPU needs - the vertex and fragment functions that you just created and such information as which pixel format it should use and whether it should render with depth. Then from the pipeline descriptor, pipeline state is being created for its use of the GPU. [11]

After final image of point cloud dots is positioned on the scene, in order to save scanned object as a point cloud 3D model for further processing, its coordinates should be converted from scene space to the world space and saved with its colors. While user camera is being rotated, pixel buffer gains more information about the scanned object. Therefore, the world space points should be copied from the buffer and appended to an array, this process is repeated for each frame in order to create 3D point cloud from all sides. Then, this point cloud vertices array should be written to a file with all necessary accompanying data.

8. File formats.

There are a number of standard 3D file formats that can be used for convenient object storing.

1) .obj - Wavefront OBJ format developed by Wavefront Technologies.

Contains only a single model. Materials, such as textures and surface properties, can be specified using an accompanying .mtl file.

Format:

- mtlname - name of the accompanying .mtl file that holds the material and texture file names for the model;
- g - starts a group of vertices;
- v - vertex;
- vn - surface normal - a vector that points orthogonally;

- vt - uv coordinate for textures;
- usemtl - name of a material providing the surface information that is defined in the accompanying .mtl file;
- f - faces. Each face has three elements consisting of a vertex/texture/normal index;
- s - smoothing.

The .mtl file contains the model's materials that describe how the 3D renderer should color the vertex.

Format:

- newmtl material_1 - group that contains all of the vertices;
- Kd - diffuse color of the surface;
- Ka: ambient color;
- Ks: The specular color reflected from a highlight;
- Other properties, such as ao, subsurface, metallic, specularTint, roughness, anisotropicRotation, sheen, sheenTint, clearCoat, clearCoatGloss.

3) USD is a format devised by Pixar, which can hold massive scenes with textures, animation and lighting information. There are various file extensions:

- .usd - a Universal Scene Description (USD) file that consists of assets or links to assets. The file can contain mesh geometry, shading information, models, cameras and lighting;
- .usdz - a single archive file that contains all the files - not just links;
- .usda - the USD file in text format;
- .usdc - the USD file in binary format. [12]

4) .glTF - GL Transmission Format is a relatively new open sourced format that supports static models, animation, and moving scenes.

A GLB file (.glb), which stands for “GL Transmission Format Binary file”, is a standardized file format used to share 3D data. Precisely, it can

contain information about 3D scenes, models, lighting, materials, node hierarchy and animations.

The GLB format is a version of the GLTF file. GLB format is a binary file format while the GLTF is based on JSON (JavaScript Object Notation). The GLB locates all of the elements of a 3D scene, including materials, node hierarchy and cameras in one single compressed file. In comparison, the GLTF file requires external processing file formats, such as for textures, shaders and animation data. [13][14]

5) STL is a file format native to the stereolithography CAD software created by 3D Systems. STL has several backronyms such as "Standard Triangle Language" and "Standard Tessellation Language". It is widely used for rapid prototyping, 3D printing and computer-aided manufacturing. STL files describe only the surface geometry of a three-dimensional object without any representation of color, texture or other common CAD model attributes. The STL format specifies both ASCII and binary representations. [15]

5) PLY is a computer file format known as the Polygon File Format or the Stanford Triangle Format. It was principally designed to store three-dimensional data from 3D scanners. The data storage format supports a relatively simple description of a single object as a list of nominally flat polygons. A variety of properties can be stored, including color and transparency, surface normals, texture coordinates and data confidence values. There are two versions of the file format, one in ASCII, the other in binary. [16]

```

ply
format ascii 1.0          { ascii/binary, format version number }
comment made by Greg Turk { comments keyword specified, like all lines }
comment this file is a cube
element vertex 8          { define "vertex" element, 8 of them in file }
property float x          { vertex contains float "x" coordinate }
property float y          { y coordinate is also a vertex property }
property float z          { z coordinate, too }
element face 6            { there are 6 "face" elements in the file }
property list uchar int vertex_index { "vertex_indices" is a list of ints }
end_header                { delimits the end of the header }
0 0 0                    { start of vertex list }
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3                { start of face list }
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0

```

Figure 4. PLY file format. [17]

The .ply format is commonly used to store point cloud data. In order to get 3D mesh from the point cloud, it needs to be reconstructed via surface reconstruction algorithms that run on the point cloud to generate a watertight surface over points.

9. 3D surface reconstruction.

In many scenarios the goal is to generate a dense 3D geometry, i.e., a triangle mesh. However, from a depth sensor we only obtain an unstructured point cloud. To get a triangle mesh from this unstructured input it's necessary, firstly, to preprocess the input point cloud and then perform surface reconstruction on it. [18]

Preprocessing:

- Voxel downsampling is often used as a pre-processing step for many point cloud processing tasks. It uses a regular voxel grid to create a uniformly downsampled point cloud from an input point cloud. The algorithm operates in two steps:
 - Points are bucketed into voxels.
 - Each occupied voxel generates exactly one point by averaging all points inside.

- Normal estimation. Surface normals are important properties of a geometric surface. Oriented normals are required for many reconstruction algorithms. The most popular approach is to orient every point's normal perpendicular to its face surface towards camera position.

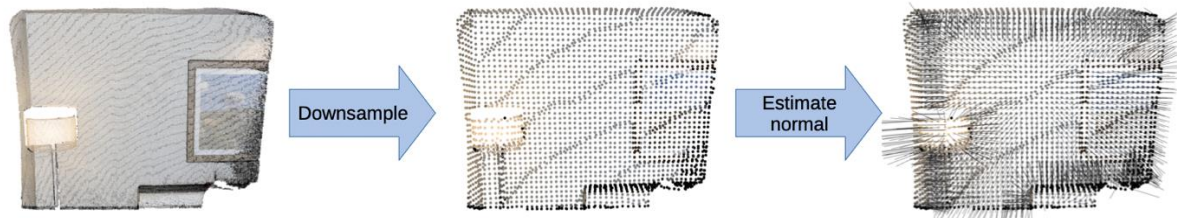


Figure 5. A simple 3D data processing task: load a point cloud, downsample it, and estimate normals.

Algorithms for surface reconstruction:

- Alpha shapes algorithm - is a generalization of a convex hull of a point cloud that is the smallest convex set that contains all points. [19]

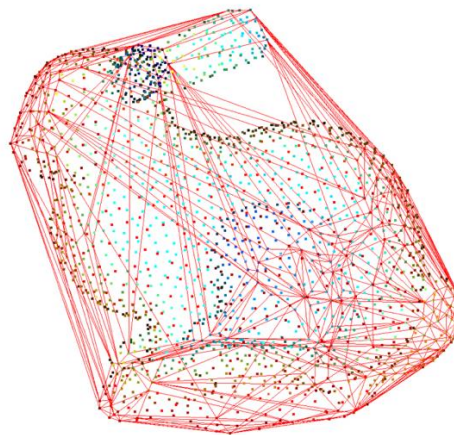


Figure 6. Convex hull. [19]

- The Ball Pivoting algorithm (BPA) is a surface reconstruction method which is related to alpha shapes. The idea behind the BPA is to simulate the use of a virtual ball to generate a mesh from a point cloud. Algorithm process is similar to rolling a tiny ball across the point cloud “surface”. This tiny ball is dependent on the scale of the mesh, and should be slightly larger than the average space between points. When a ball is

dropped onto the surface of points, the ball will get caught and settle upon three points that will form the seed triangle. From that location, the ball rolls along the triangle edge formed from two points.

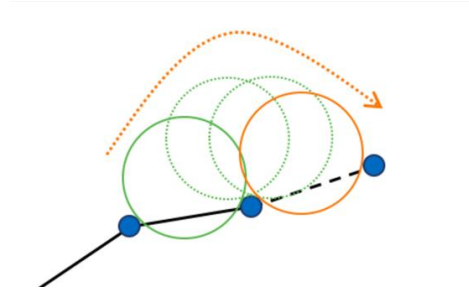


Figure 7. Pivoting ball. [20]

The ball then settles in a new location: a new triangle is formed from two of the previous vertices and one new triangle is added to the mesh. As we continue rolling and pivoting the ball, new triangles are formed and added to the mesh. The ball continues rolling and rolling until the mesh is fully formed. This algorithm assumes that the point cloud has normals.

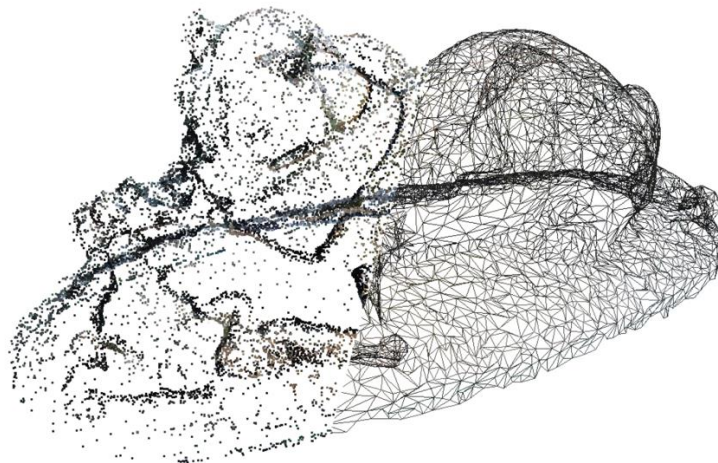


Figure 8. Point cloud surface reconstruction.

- The Poisson surface reconstruction method - solves a regularized optimization problem to obtain a smooth surface. It is known as an implicit meshing method, that's goal is to fit a watertight surface from the original point set by creating an entirely new point set representing an

isosurface linked to the normals. Therefore, the algorithm assumes that the point cloud has normals. [18]

Poisson surface reconstruction will also create triangles in areas of low point density. A low density value means that the vertex is only supported by a low number of points from the input point cloud. After density indication for each vertex, vertices and triangles that have a low support can be removed. [21]

Easily work with mesh reconstruction algorithms allows Open3D library.

10. Open3D library.

Open3D is an open-source library designed for processing 3D data. It was introduced by Qian-Yi Zhou, Jaesik Park and Vladlen Koltun – researchers at Intel Labs. It allows the use of a set of efficient data structures and algorithms for 3D data processing. Library provides a C++ interface and an easier Python-based interface.

Among library features - it enables 3D visualization and a physically based rendering (PBR) approach of computer graphics; the reconstruction of 3D scenes and alignment of surfaces; supports PyTorch and TensorFlow machine learning frameworks.

Module	Functionality
Geometry	Data structures and basic processing algorithms
Camera	Camera model and camera trajectory
Odometry	Tracking and alignment of RGB-D images
Registration	Global and local registration
Integration	Volumetric integration
I/O	Reading and writing 3D data files
Visualization	A customizable GUI for rendering 3D data with OpenGL
Utility	Helper functions such as console output, file system, and Eigen wrappers
Python	Open3D Python binding and tutorials

Figure 9. Open3D library included modules.

Section 3. Description of practical research

1. Practical research on possible approaches to create 3D model.

Approaches with the usage of native iOS technologies that were considered in order to create 3D model via scanning real life object:

1) Object Capture API in RealityKit framework.

API is possible to test via Photogrammetry Apple Sample Project. [22] RealityKit offers Object Capture API that can create 3D model from photos taken from many different angles using a photogrammetry. It analyses the overlap area between different images to match up landmarks, and then produces a 3D model of the photographed object. API only works on macOS, so not an option for this research as processing cannot be done on mobile device.

Hardware requirements:

- Mac for post-processing:
 - o Mac with M1 chip or Intel 16GB RAM and AMD 4GB VRAM GPU.
 - o macOS version is Monterey or later.
 - o XCode version is 13 or later.
- Device for taking pictures:
 - o iPhone or iPad with a dual-lens rear camera (and preferably a LiDAR scanner, although not required). [23]

Input: Object photos with depth information from different angles.

Output: Model in .usdz format.

Outtakes:

- Quality of produces result is high.
- Processing is taking approximately 3-5 minutes depending on the model and quantity of photos.
- Object processing is only possible on macOS devices, especially on the newest ones.



Figure 10. Result of Object Capture API.

2) ARKit Scene Reconstruction

In the sample project ARKit uses the LiDAR Scanner to create a polygonal model of the physical environment. The LiDAR Scanner quickly retrieves depth information from a wide area in front of the user, so ARKit can estimate the shape of the real world without requiring the user to move. ARKit converts the depth information into a series of vertices that connect to form a mesh. [24]

Steps:

- ARMeshGeometry object is being retrieved from current AR view.

```
@IBAction func exportMesh(_ button: UIButton) {
    let meshAnchors = arView.session.currentFrame?.anchors.compactMap({ $0 as? ARMeshAnchor })

    DispatchQueue.global().async {

        let filename = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0].appendingPathComponent("MyFirstMesh.obj")

        guard let device = MTLCreateSystemDefaultDevice() else {
            print("metal device could not be created")
            return
        }

        let asset = MDLAsset()

        for anchor in meshAnchors! {
            print("anchor", anchor)
            let mdlMesh = anchor.geometry.toMDLMesh(device: device)
            asset.add(mdlMesh)
        }
    }
}
```

Figure 11. Get ARMeshGeometry anchors.

- ARMeshGeometry anchors is being converted to MDLMesh anchors.
- MDLMesh anchors are being exported as .obj file.

```

extension ARMeshGeometry {
    func toMDLMesh(device: MTLDevice, transform: simd_float4x4) -> MDLMesh {
        let allocator = MTKMeshBufferAllocator(device: device)

        let data = Data.init(bytes: transformedVertexBuffer(transform), count: vertices.stride * vertices.count)
        let vertexBuffer = allocator.newBuffer(with: data, type: .vertex)

        let indexData = Data.init(bytes: faces.buffer.contents(), count: faces.bytesPerIndex * faces.count * faces.indexCountPerPrimitive)
        let indexBuffer = allocator.newBuffer(with: indexData, type: .index)

        let submesh = MDLSubmesh(indexBuffer: indexBuffer,
                                indexCount: faces.count * faces.indexCountPerPrimitive,
                                indexType: .uInt32,
                                geometryType: .triangles,
                                material: nil)

        let vertexDescriptor = MDLVertexDescriptor()
        vertexDescriptor.attributes[0] = MDLVertexAttribute(name: MDLVertexAttributePosition,
                                                            format: .float3,
                                                            offset: 0,
                                                            bufferIndex: 0);
        vertexDescriptor.layouts[0] = MDLVertexBufferLayout(stride: vertices.stride)

        return MDLMesh(vertexBuffer: vertexBuffer,
                       vertexCount: vertices.count,
                       descriptor: vertexDescriptor,
                       submeshes: [submesh])
    }
}

```

Figure 12. Export ARMeshGeometry as MDLMesh.

- After scanning texture information is not available. Attempted to assign appropriate color to the object of specific type specified by ARMeshClassification.



Figure 13. Mesh with colors applied by ARMeshClassification.

The sample project is presenting the mesh with colors on scene by leveraging the debug method, which is not publicly accessible - `arView.debugOptions.insert(.showSceneUnderstanding)`. It leverages RealityKit

for rendering AR content, which does not have a method to generate a mesh from the ARMeshGeometry and texture it.

Hardware requirements - device with LiDAR scanner.

Input: Object's depth data by LiDAR scanner.

Output: Model in .obj/.usdz format without color information.

Outtakes:

- Impossible to extract object mesh with texture.

3) Scanning And Detecting 3D Objects sample project by Apple.

Input: Depth information from regular camera.

Output: Model in .arobject format.

Process:

- Object is scanned from the different angles.
- It is possible to share the object as a file with .arobject extension.

Outtakes:

Received .arobject doesn't contain 3D geometry information and cannot be displayed in any way later. File in .arobject file is just a reference object with 2D and cloud points. It cannot be converted to any format that contains 3D information or be used later. .arobject file contains only the spatial feature information needed to recognize a scanned real-world object, it is not a displayable 3D reconstruction mesh of that object. [25]

4) Export object's **point cloud** as .ply model and create 3D mesh from it.

Present a visualization of the physical environment by placing points based a scene's depth data. Then form dense point cloud data to structured point cloud object. Solution is based on Displaying a Point Cloud Using Scene Depth sample project by Apple.

Hardware requirements: Apple device with LiDAR scanner.

Input: Depth information of the physical environment by the LiDAR scanner.

Output: Scanned object with texture as point cloud in .ply format without normals.

Outtakes:

- Good result quality.
- It is possible to create full-fledged 3D model from the gathered point cloud.

2. Point cloud creation.

To recreate physical surroundings as a point cloud model, depth information about each pixel on the current frame is needed to position them and camera capture data - to color each pixel from depth data buffer.

1. From the device's LiDAR scanner get frame's depthMap - depth data for each pixel from the frame; pixel buffer of the estimated distances from the device to the environment.

- Check for LiDAR availability in order to get access to the .sceneDepth and .smoothedSceneDepth options of the ARWorldTrackingConfiguration.frameSemantics.

```
if ARWorldTrackingConfiguration.supportsFrameSemantics(.sceneDepth) {  
    let configuration = ARWorldTrackingConfiguration()  
    configuration.frameSemantics = [.sceneDepth, .smoothedSceneDepth]  
    session.run(configuration)  
}
```

Figure 14. SceneDepth availability check.

- Get depthMap. After configuration of the .sceneDepth property - depthMap is accessible for the ARSession's currentFrame.

```
private func updateDepthTextures(frame: ARFrame) -> Bool {  
    guard let depthMap = frame.smoothedSceneDepth?.depthMap,  
          let confidenceMap = frame.smoothedSceneDepth?.confidenceMap else {  
        return false  
    }  
  
    self.depthTexture = self.makeTexture(from: depthMap, with: .r32Float, at: 0)  
    self.confidenceTexture = self.makeTexture(from: confidenceMap, with: .r8UInt, at: 0)  
  
    return true  
}
```

Figure 15. Get depthMap of the frame.

2. Assign pixel data from the camera image to each point from the depthMap.
 - Access ARKit camera feed - `capturedImage` property on frame — pixel buffer containing the image captured by the camera.

```
private func updateCapturedImageTextures(for frame: ARFrame) {
    let pixelBuffer = frame.capturedImage
    guard CVPixelBufferGetPlaneCount(pixelBuffer) >= 2 else {
        return
    }

    // Create two textures - Y and CbCr, from the provided frame's captured image.
    self.capturedImageTextureY = self.makeTexture(from: pixelBuffer, with: .r8Unorm, at: 0)
    self.capturedImageTextureCbCr = self.makeTexture(from: pixelBuffer, with: .rg8Unorm, at: 1)
}
```

Figure 16. Get capturedImage of the frame.

For every point in the depth map - check the corresponding pixel in the camera image and assign the pixel's color to the point by sampling that depth map value's position in the camera image. Each vertex calculates its x and y location in the camera image by converting its position in the one-dimensional vertex array, to a 2D position in the depth texture. Checking and assigning are done on the GPU side via shaders.

`capturedImage` represents data in the YUV format, while the GPU color format is RGBA, so it's necessary to convert the camera data from YUV to RGBA.

```
constant auto yCbCrToRGB = float4x4(float4(+1.0000f, +1.0000f, +1.0000f, +0.0000f),
                                       float4(+0.0000f, -0.3441f, +1.7720f, +0.0000f),
                                       float4(+1.4020f, -0.7141f, +0.0000f, +0.0000f),
                                       float4(-0.7010f, +0.5291f, -0.8860f, +1.0000f));
```

Figure 17. YCbCr translation matrix.

3. Place dot onto physical environment surface for each pixel from depthMap pixel buffer with pixel color.

- Draw dots on the surface. Points on the scene are being drawn using Metal shading language on the GPU side (appendix A).

For positioning of points its coordinates need to be translated from world to screen via projection matrix.

4. Repeating the same process and enlarging depth buffer via checking confidenceMap (depth confidence values) while camera is being rotated in order to create full 3D point cloud.
5. Create 3D model in .ply format from point cloud.
 - Scene coordinates of the points should be translated (unprojected) to the world space coordinates (appendix B).
 - Store world space points with its color in a MTLBuffer.
 - While camera is being rotated, buffer gains more information about the scanned object. Therefore, the world space points should be copied from the buffer and appended to an array, this process is repeated for each frame in order to create 3D point cloud from all sides (appendix C).
 - Extended point cloud vertices buffer should be written to a file with all necessary accompanying data (appendix D).

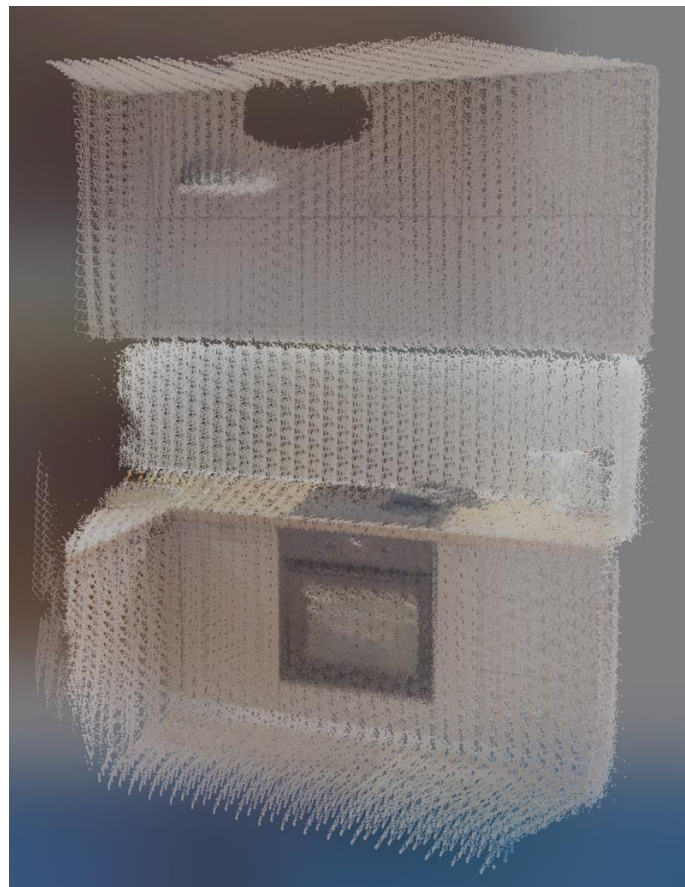


Figure 22. Result of 3D point cloud model creation in .ply file

3. Surface reconstruction of the point cloud.

For point cloud post-processing Open3D framework, namely Python API, was chosen because of its ease of use and extensive functionality.

Prerequisites: Open3D package is compatible with python versions: 2.7, 3.5 or 3.6.

Input: structured point cloud in .ply file without normals.

Output: .ply file of reconstructed 3D textured mesh.

Point cloud pre-processing and reconstruction process using Open3D library (appendix E):

- Load created point cloud in the .ply format to the script.

```
path = "/Users/veronika/Desktop/input_data.ply"
pcd = o3d.io.read_point_cloud(path)
```

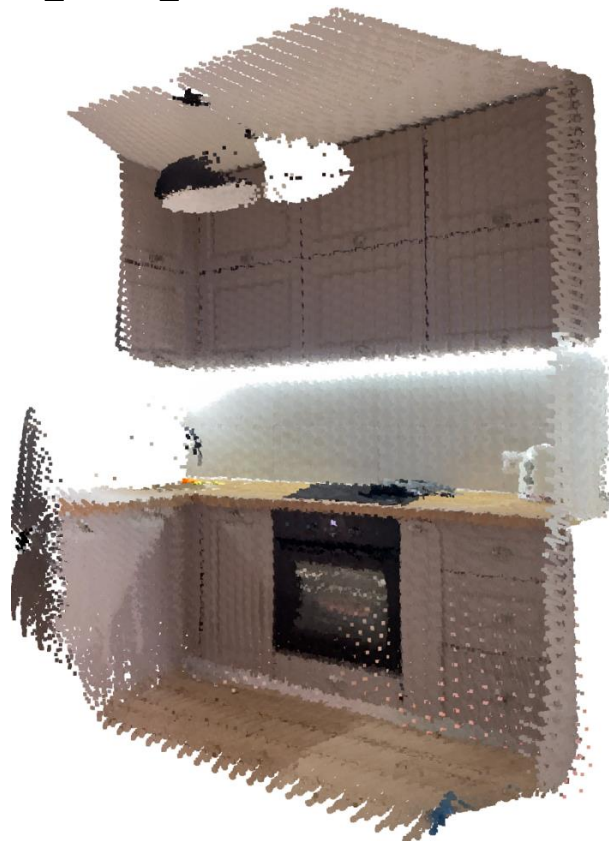


Figure 23. Visualization of initial point cloud.

- Downsample the point cloud with a voxel of size 0.01

```
downpcd = pcd.voxel_down_sample(voxel_size=0.01)
```




Figure 24. Visualization of the downsampled point cloud.

- Recompute the normal of the downsampled point cloud.

```
downpcd.estimate_normals(search_param=o3d.geometry.KDTreeSearch  
hParamHybrid(radius=0.1, max_nn=30))
```

Normals is needed for both BPA and Poisson surface reconstruction algorithms. Therefore, produced mesh without properly oriented normals is distorted.



Figure 25. Poisson reconstruction result without oriented normals.

- Orient computed normals towards camera location.

```
downpcd.orient_normals_towards_camera_location(camera_location  
=np.array([0., 0., 0.]))
```

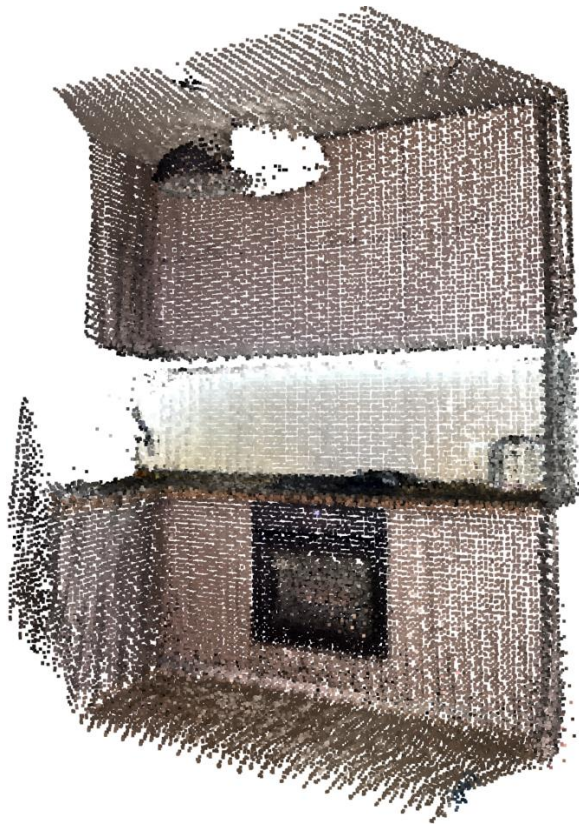


Figure 26. Visualization of the point cloud with oriented normals.

- Reconstruct the point cloud surface via Poisson algorithm.

```
with  
o3d.utility.VerboseContextManager(o3d.utility.VerboseLevel  
.Debug) as cm:  
    mesh, densities =  
o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(down  
pcd, depth=9)
```



Figure 27. Poisson reconstruction result.

Point cloud surface reconstruction via Open3D framework is also possible with BPA algorithm. But produced results are worse than ones via Poisson algorithm.



Figure 28. BPA point cloud reconstruction result.

- Identify and remove low density vertices.

```
vertices_to_remove = densities < np.quantile(densities, 0.01)  
mesh.remove_vertices_by_mask(vertices_to_remove)
```

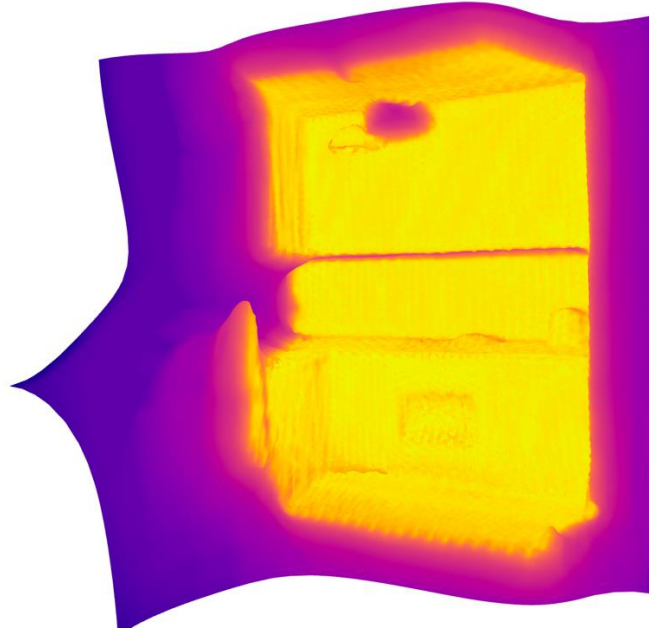


Figure 29. Low densities areas visualization.

- Save produced mesh to a file.

```
o3d.io.write_triangle_mesh("/Users/veronika/Desktop/poisson_output_mesh.ply", mesh)
```



Figure 30. Poisson mesh after low density area removal.

Conclusions and analysis of opportunities for further development

In the thesis, native iOS approaches for real-life object's 3D model creation were studied. The analysis of found solutions were carried in order to choose the most suitable approach.

The result of the research is an app for structured point cloud creation from scene depth information and Python script for surface reconstruction of the point cloud in order to get full-fledged 3D model of real-life object.

Further development opportunities:

Among the possibilities for further enhancement of the developed solution is rewriting point cloud post-processing script to C++ programming language and embedment of it in the current Swift project in order to carry out all the computations on the device. Therefore, app and algorithms optimizations should be considered.

List of used literature

1. Depth estimation [Electronic resource]: <https://beyondminds.ai/blog/depth-estimation/>
2. Capturing photos with depth [Electronic resource]:
https://developer.apple.com/documentation/avfoundation/cameras_and_media_capture/capturing_photos_with_depth
3. TrueDepth [Electronic resource]: <https://apple.fandom.com/wiki/TrueDepth>
4. Capturing depth using the LiDAR camera [Electronic resource]:
https://developer.apple.com/documentation/avfoundation/cameras_and_media_capture/capturing_depth_using_the_lidar_camera
5. Scene Reconstruction with a LiDAR scanner [Electronic resource]:
<https://medium.com/macoclock/arkit-911-scene-reconstruction-with-a-lidar-scanner-57ff0a8b247e>
6. How LiDAR differs from ‘TrueDepth’ Face ID [Electronic resource]:
<https://www.eyerys.com/articles/how-apples-lidar-sensor-differs-one-its-truedepth-face-id>
7. Time-Of-Flight and LiDAR sensors [Electronic resource]:
<https://www.eyerys.com/articles/time-flight-and-lidar-sensors-how-depth-sensing-cameras-go-mobile>
8. Creating a fog effect using scene depth [Electronic resource]:
https://developer.apple.com/documentation/arkit/environmental_analysis/creating_a_fog_effect_using_scene_depth
9. Displaying a point cloud using scene depth [Electronic resource]:
https://developer.apple.com/documentation/arkit/environmental_analysis/displaying_a_point_cloud_using_scene_depth
10. Begbie C. Queues, buffers and encoders / Caroline Begbie // Metal by Tutorials / Caroline Begbie., 2019. – P. 34.
11. Begbie C. How did Metal come to life? / Caroline Begbie // Metal by Tutorials / Caroline Begbie., 2019. – P. 23

12. Begbie C. 3D file formats / Caroline Begbie // Metal by Tutorials / Caroline Begbie., 2019. – P. 47
13. Everything you need to know about glTF files [Electronic resource]:
<https://www.marxentlabs.com/glTF-files/>
14. What is a GLB file? [Electronic resource]: <https://visao.ca/what-is-glb-file/>
15. STL (file format) [Electronic resource]:
[https://en.wikipedia.org/wiki/STL_\(file_format\)](https://en.wikipedia.org/wiki/STL_(file_format))
16. PLY (file format) [Electronic resource]:
[https://en.wikipedia.org/wiki/PLY_\(file_format\)](https://en.wikipedia.org/wiki/PLY_(file_format))
17. Polygon file format [Electronic resource]:
<http://paulbourke.net/dataformats/ply/>
18. Open3D – Surface reconstruction [Electronic resource]:
http://www.open3d.org/docs/latest/tutorial/Advanced/surface_reconstruction.html
19. Open3D – Point cloud [Electronic resource]:
<http://www.open3d.org/docs/latest/tutorial/Basic/pointcloud.html>
20. Point cloud to mesh [Electronic resource]:
<https://cs184team.github.io/cs184-final/writeup.html>
21. Generate 3D mesh from point clouds [Electronic resource]:
<https://towardsdatascience.com/5-step-guide-to-generate-3d-meshes-from-point-clouds-with-python-36bad397d8ba>
22. Creating a Photogrammetry command-line app [Electronic resource]:
https://developer.apple.com/documentation/realitykit/creating_a_photogrammetry_command-line_app/
23. Object Capture API [Electronic resource]:
<https://9to5mac.com/2021/06/09/hands-on-macos-12-brings-new-object-capture-api-for-creating-3d-models-using-iphone-camera/>

24. Visualizing and interacting with a reconstructed scene [Electronic resource]:

https://developer.apple.com/documentation/arkit/content_anchors/visualizing_and_interacting_with_a_reconstructed_scene

25. Scanning and detecting 3D objects [Electronic resource]:

https://developer.apple.com/documentation/arkit/content_anchors/scanning_and_detecting_3d_objects

26. CGAL 5.4 - Poisson Surface Reconstruction [Electronic resource]:

https://doc.cgal.org/latest/Poisson_surface_reconstruction_3/index.html

Glossary

AR – Augmented Reality

3D – three-dimensional.

Open3D – library that supports development of software that deals with 3D data.

API – Application Programming Interface.

True Depth - Apple's camera system that replaces the front facing camera on the iPhone X and later.

LiDAR - Light Detection and Ranging - a remote sensing method.

Applications

Appendix A. Points placement preparation

```
vertex PointVertexOut pointVertex(uint vertexID [[vertex_id]],
                                   constant PointCloudUniforms &uniforms [[buffer(pointCloudUniforms)]],
                                   constant PointUniforms *pointUniforms [[buffer(pointUniforms)]] {

    // Get point data.
    const auto pointData = pointUniforms[vertexID];
    const auto position = pointData.position;
    const auto confidence = pointData.confidence;
    const auto sampledColor = pointData.color;
    const auto visibility = confidence >= uniforms.confidenceThreshold;

    // Animate and project the point.
    float4 projectedPosition = uniforms.viewProjectionMatrix * float4(position, 1.0);
    const float pointSize = max(uniforms.pointSize / max(1.0, projectedPosition.z), 2.0);
    projectedPosition /= projectedPosition.w;

    // Prepare for output.
    PointVertexOut out;
    out.position = projectedPosition;
    out.pointSize = pointSize;
    out.color = float4(sampledColor, visibility);

    return out;
}
```

Appendix B. Coordinates unprojection

```
/// Vertex shader that takes in a 2D grid-point and infers its 3D position in world-space, along with RGB and confidence
vertex void unprojectVertex(uint vertexID [[vertex_id]],
    constant PointCloudUniforms &uniforms [[buffer(pointCloudUniforms)]],
    device PointUniforms *pointUniforms [[buffer(pointUniforms)]],
    constant float2 *gridPoints [[buffer(gridPoints)]],
    texture2d<float, access::sample> capturedImageTextureY [[texture(textureY)],
    texture2d<float, access::sample> capturedImageTextureCbCr [[texture(textureCbCr)],
    texture2d<float, access::sample> depthTexture [[texture(textureDepth)],
    texture2d<unsigned int, access::sample> confidenceTexture [[texture(textureConfidence)]]]) {

    const auto gridPoint = gridPoints[vertexID];
    const auto currentPointIndex = (uniforms.pointCloudCurrentIndex + vertexID) % uniforms.maxPoints;
    const auto texCoord = gridPoint / uniforms.cameraResolution;
    // Sample the depth map to get the depth value
    const auto depth = depthTexture.sample(colorSampler, texCoord).r;
    // With a 2D point plus depth, we can now get its 3D position
    const auto position = worldPoint(gridPoint, depth, uniforms.cameraIntrinsicsInversed, uniforms.localToWorld);

    // Sample Y and CbCr textures to get the YCbCr color at the given texture coordinate
    const auto ycbcr = float4(capturedImageTextureY.sample(colorSampler, texCoord).r, capturedImageTextureCbCr.sample(colorSampler, texCoord.xy).rg, 1);
    const auto sampledColor = (yCbCrToRGB * ycbcr).rgb;
    // Sample the confidence map to get the confidence value
    const auto confidence = confidenceTexture.sample(colorSampler, texCoord).r;

    // Write the data to the buffer
    pointUniforms[currentPointIndex].position = position.xyz;
    pointUniforms[currentPointIndex].color = sampledColor;
    pointUniforms[currentPointIndex].confidence = confidence;
}
```

Appendix C. More points accumulation in case of camera rotation

```
private func shouldAccumulatePoints(in frame: ARFrame) -> Bool {
    let cameraTransform = frame.camera.transform
    return self.currentPointCount == 0
    || dot(cameraTransform.columns.2, self.lastCameraTransform.columns.2) <= self.cameraRotationThreshold
    || distance_squared(cameraTransform.columns.3, self.lastCameraTransform.columns.3) >= self.cameraTranslationThreshold
}

private func accumulatePoints(frame: ARFrame, commandBuffer: MTLCommandBuffer, renderEncoder: MTLRenderCommandEncoder) {
    self.pointCloudUniformsBuffer.pointCloudCurrentIndex = Int32(self.currentPointIndex)

    var retainingTextures = [self.capturedImageTextureY, self.capturedImageTextureCbCr, self.depthTexture, self.confidenceTexture]

    commandBuffer.addCompletedHandler { buffer in
        retainingTextures.removeAll()
        var i = self.pointsCpuBuffer.count
        while (i < self.maxPointsInCloud && self.pointsBuffer[i].position != simd_float3(0.0,0.0,0.0)) {
            let position = self.pointsBuffer[i].position
            let color = self.pointsBuffer[i].color
            let confidence = self.pointsBuffer[i].confidence
            if confidence == 2 { self.highConfidencePointsCount += 1 }
            self.pointsCpuBuffer.append(
                PointCPU(position: position,
                        color: color,
                        confidence: confidence))
            i += 1
        }
    }
}
```

Appendix D. Saving point cloud model to a .ply file.

```
static func writeToFile(pointsCpuBuffer: inout [PointCPU], highConfidenceCount: Int) throws {
    let fileName = "scan"

    let documentsDirectory = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0]
    let date = Date().description(with: .current)

    let plyFile = documentsDirectory.appendingPathComponent("\(fileName)_\(date).ply", isDirectory: false)
    FileManager.default.createFile(atPath: plyFile.path, contents: nil, attributes: nil)

    let format = "binary_little_endian"

    var headersString = ""
    let headers = [
        "ply",
        "comment Created by SceneX (IOS)",
        "format \(format) 1.0",
        "element vertex \(highConfidenceCount)",
        "property float x",
        "property float y",
        "property float z",
        "property uchar red",
        "property uchar green",
        "property uchar blue",
        "property uchar alpha",
        "element face 0",
        "property list uchar int vertex_indices",
        "end_header"]

    for header in headers { headersString += header + "\r\n" }
    try headersString.write(to: plyFile, atomically: true, encoding: .ascii)

    try writeBinary(file: plyFile, format: format, pointsCPUBuffer: &pointsCpuBuffer)
}
```

```
private static func writeBinary(file: URL, format: String, pointsCPUBuffer: inout [PointCPU]) throws -> Void {
    let fileHandle = try! FileHandle(forWritingTo: file)
    fileHandle.seekToEndOfFile()
    var data = Data()

    for point in pointsCPUBuffer {
        if point.confidence != 2 { continue }

        var x = point.position.x.bitPattern.littleEndian
        data.append(withUnsafePointer(to: &x) {
            Data(buffer: UnsafeBufferPointer(start: $0, count: 1))
        })

        var y = point.position.y.bitPattern.littleEndian
        data.append(withUnsafePointer(to: &y) {
            Data(buffer: UnsafeBufferPointer(start: $0, count: 1))
        })

        var z = point.position.z.bitPattern.littleEndian
        data.append(withUnsafePointer(to: &z) {
            Data(buffer: UnsafeBufferPointer(start: $0, count: 1))
        })

        let colors = point.color
        var red = self.arrangeColorByte(color: colors.x).littleEndian
        data.append(withUnsafePointer(to: &red) {
            Data(buffer: UnsafeBufferPointer(start: $0, count: 1))
        })

        var green = self.arrangeColorByte(color: colors.y).littleEndian
        data.append(withUnsafePointer(to: &green) {
            Data(buffer: UnsafeBufferPointer(start: $0, count: 1))
        })

        var blue = self.arrangeColorByte(color: colors.z).littleEndian
        data.append(withUnsafePointer(to: &blue) {
            Data(buffer: UnsafeBufferPointer(start: $0, count: 1))
        })

        var alpha = UInt8(255).littleEndian
        data.append(withUnsafePointer(to: &alpha) {
            Data(buffer: UnsafeBufferPointer(start: $0, count: 1))
        })
    }
    fileHandle.write(data)
    fileHandle.closeFile()
}
```

Appendix E. Common surface reconstruction pipeline. [26]

