CMPE – 492

Senior Design Project II

FINAL REPORT

SURGEVISION

TEAM MEMBERS

HAKAN UCA

DENİZ POLAT

ABDULLAH DOĞANAY

ONUR USLU

Table of Contents

1. Introduction

1.1 Project Overview

SurgeVision is a next-generation AR/VR surgical planning and training platform designed specifically for hip replacement procedures. It leverages real-time point cloud processing, 3D visualization, and immersive interaction to assist surgeons in preoperative planning and to serve as an educational tool for medical trainees. The system transforms MRI and CT scans into point clouds, aligns them with real-world scan data, and presents the result in AR/VR environments such as Meta Quest 3.

1.2 Problem Statement

Traditional surgical planning techniques rely heavily on static 2D imagery and manual measurements, which may lead to inaccuracies and inefficiencies. Complex anatomy, motion artifacts, and limited interactivity increase the likelihood of surgical errors. There is a need for a more intuitive, accurate, and interactive planning tool to reduce errors and improve surgeon preparedness.

1.3 Objectives and Goals

- **Precision Enhancement**:
  - Convert medical imaging into accurate 3D point clouds.
- **Immersive Visualization**:
  - Use AR/VR tools for interactive model exploration.
- **Real-Time Performance**:
  - Maintain >25 FPS even under load for seamless experience.
- **Educational Utility**:
  - Enable replay, difficulty levels, and performance tracking for trainees.
- **Data Security**:
  - Use TLS encryption for safe handling of patient data.

1.4 Scope of the Project

The project covers:

- Point cloud generation from MRI/CT and AR/VR devices.
- Data simplification and registration.
- Real-time visualization on AR/VR headsets.
- User interaction via hand gestures or controllers.
- Integration into clinical and educational workflows.

1.5 Key Features and Functionalities

- **MRI Module**: Converts .nii to .ply format.
- **Scan Module**: Captures ARCore-based scans for real-time 3D data.
- **Simplification Module**: Downsamples and cleans point clouds.
- **Surface Model Module**: Generates surface mesh from point data.
- **Registration Module**: Aligns MRI and scanned data using ICP/CPD.
- **Visualization Module**: Displays final output in AR/VR or monitor.
- **User Interface Module**: Enables user interactions with gestures or controllers.
- **Tracking Module**: Monitors user/device motion to update the environment.

1.6 Methodology Overview

The system follows a modular, microservices-based architecture and is developed using C++, Unity, and Android SDKs. MRI and scan data are independently converted to point clouds, simplified, aligned, and visualized in Unity. Each module was developed, tested, and integrated incrementally using unit, integration, and system testing strategies.

2. System Design

2.1 System Architecture

SurgeVision is built on a modular microservices architecture to support real-time, low-latency operation. It follows an event-driven model where each step in the pipeline—from MRI upload to AR/VR rendering—is triggered by the previous one. The system supports both AR and VR interactions through Unity and ARCore, offering flexibility across devices.

2.2 Modules Overview

2.2.1 MRI Module

- Converts .nii files into .ply point clouds.
- Ensures valid input, handles conversion errors.

2.2.2 Scan Module

- Uses ARCore to capture 3D environment data.
- Transforms it into point cloud format.
- Synchronizes with MRI data later in pipeline.

2.2.3 Simplification Module

- Applies downsampling and outlier removal.
- Prepares data for real-time visualization.
- Maintains key anatomical features.

2.2.4 Surface Model Module

- Converts simplified point clouds into surface meshes.
- Uses the Marching Cubes algorithm for high-resolution rendering.

2.2.5 Registration Module

- Performs alignment between MRI and scan point clouds.
- Uses ICP (rigid) and CPD (non-rigid) registration.
- Targets <5mm alignment accuracy.

2.2.6 Visualization Module

- Renders 3D models in Unity-based AR/VR scenes.
- Provides real-time updates and multi-perspective viewing.

2.2.7 User Interface Module

- Manages menus, controls, and visual overlays.
- Supports gesture-based and controller-based interaction.

2.2.8 Tracking Module

- Continuously tracks motion of headset, hands, or tools.
- Dynamically updates point cloud positioning in real-time.

2.3 Hardware and Software Requirements

2.3.1 Hardware Specifications

- **AR/VR Headsets**: Meta Quest 3.
- **Mobile Devices**: ARCore-enabled smartphones.
- **High-performance PCs**: For MRI preprocessing and testing.

2.3.2 Software and Tools Used

- **Languages**: C++, C#, Kotlin/Java.
- **Libraries**:
    - **PCL** (Point Cloud Library): Processing and visualization.
    - **FAISS, FLANN**: Fast nearest-neighbor search.
    - **Eigen**: Matrix computations.
    - **Unity + ARCore**: AR/VR interaction and rendering.
- **Frameworks**: MVVM architecture for separation of concerns.
- **Compression**: Zlib/znzlib.
- **Security**: TLS for secure data transport.

3. Design and Implementation Details

3.1 Low-Level Design Decisions

3.1.1 Data Acquisition and Preprocessing

To ensure that both the medical scans and the live AR/VR captures feed smoothly into our registration pipeline, we apply a four-step preprocessing workflow:

- **Image Import & Validation**

We support both DICOM series (CT/MRI) and compressed NIfTI files. Upon loading, each slice is checked for completeness and consistent voxel spacing. Any missing or corrupted image triggers an alert so the user can re-scan or re-export before proceeding.

- **Point-Cloud Conversion**

From each validated 2D slice, we perform a simple intensity threshold to separate anatomy from background, then sample the resulting contours into a set of 3D points. The result is an initial "raw" point cloud that accurately captures the organ boundaries.

- **Noise Filtering**

To remove scanning artifacts and isolated outliers, we apply two complementary filters in sequence:

1. A **statistical filter** that discards points whose local neighborhood distance is significantly higher than average.
2. A **radius-based filter** that removes small clusters of points not supported by any nearby neighbors.

- **Downsampling for Performance**

Finally, we apply a coarse **voxel-grid downsampling** to reduce the total point count by roughly 60–70%, while preserving geometric fidelity in high-curvature areas. This step dramatically lowers memory usage and accelerates subsequent registration without perceptible loss of detail. Using marching cubes and downsampling algorithms.

3.1.2 Point Cloud Registration and Visualization

To align and present multi-source point clouds in real time, we employ a four-stage workflow:

- **Rigid Registration**

We begin with a coarse alignment using a voxelized variant of the Iterative Closest Point (ICP) algorithm. By downsampling both source and target clouds to a 5 mm grid, we rapidly converge to within ~10 mm accuracy, providing a good starting point for finer registration.

- **Fine-Tuning Alignment**

From the coarse result, we switch to full-resolution ICP on the original point sets, iterating until the change in mean squared error drops below 0.001 mm^2. This yields sub-5 mm precision without introducing undue computational delay.

- **Quality Assessment**

After convergence, we compute the root-mean-square error (RMSE) and maximum point-to-plane residual. Points with residuals above a user-configurable threshold are highlighted in the UI, allowing immediate identification of poorly aligned regions.

- **Real-Time Rendering**

The final, registered cloud is streamed into the Unity engine, where our PointCloudRenderer batches points into GPU buffers for >25 FPS. Overlays such as color-mapped error heat-maps, coordinate axes, and adjustable clipping planes help users inspect alignment interactively.

3.2 User Interface and Interaction Design

3.2.1 User Experience Design

To deliver an intuitive, responsive interface across both desktop and AR/VR environments, we employ a four-part UX strategy:

- **Modular MVVM Architecture**
    - **Model** holds scan data, processed point clouds, and user preferences.
    - **ViewModel** manages state, exposes commands (e.g. “Start Scan,” “Toggle Heatmap”) and performs lightweight transformations.
    - **View** binds directly to ViewModel properties, enabling clear separation of concerns and easy unit testing.
- **Consistent Component Library**
    - A shared set of reusable UI elements (cards, sliders, menus) styled with our custom theme ensures visual coherence.
    - Dynamic layouts adapt seamlessly between desktop windows and headset overlays, maintaining readability and touch/controller reach.
- **Real-Time Data Binding**
    - Two-way binding updates UI metric registration error, frame rate, session status—in under 50 ms, so users always see the latest system state.
    - Asynchronous data streams (via Unity’s AsyncOperation) prevent UI freezes during heavy processing.

- **Performance-First Responsiveness**
    - Heavy tasks (e.g., point-cloud filtering) run on background threads, while UI updates are throttled to 60 Hz to keep controls snappy.
    - Lightweight animations and minimal draw calls ensure a steady >25 FPS, even with large point-cloud datasets.

3.2.2 Interactivity with AR/VR Systems

To provide seamless, hands-free control and feedback in immersive environments, we structured our AR/VR interactivity into four core components:

- **InteractionHandler**

Unifies input from controllers, hand tracking, and voice. All raw events (button presses, pinch/release gestures, spoken keywords) are normalized into high-level commands such as "Select Point," "Rotate View," or "Toggle Heatmap."

- **Gesture Mapping**

Leverages Unity's XR Interaction Toolkit to interpret pinch-and-hold for point picking, grab-and-drag for cloud translation, and two-finger swipe for orbiting the camera. Custom thresholds ensure reliable detection even in low-light or fast-motion scenarios.

- **FeedbackDisplay**

Renders real-time overlays—registration error heat-maps, frame-rate counters, session timers—onto world-locked canvases. This keeps critical metrics anchored near the anatomy, so users never lose context while moving their head or tools.

- **Spatial Anchoring & Calibration**

At startup, the system runs a quick QR-code or marker-based calibration to align virtual point clouds with the physical workspace. The **DepthSensorHandler** then continuously refines this anchor using SLAM, compensating for drift and ensuring spatial correspondence throughout the session.

3.3 Data Management and Security

To respect patient privacy and meet our non-functional requirements—while **not** persisting any user data in a database—the system enforces the following:

- **Ephemeral In-Memory Processing**

All MRI/CT scans and AR/VR depth frames are loaded directly into RAM. No point clouds, images, or patient identifiers are ever written to disk or stored in a database; once the application or session ends, all data structures are securely wiped.

- **TLS-Secured Communication**

All communication between clients, servers, and headsets is secured with TLS 1.2 or higher, using mutual certificate validation. This guarantees that scan uploads, registration commands, and visualization streams remain confidential and tamper-proof.

- **Host-Level Access Control**

We rely on the hospital or research-lab's existing authentication (e.g., OS login or network VPN) rather than managing user credentials in the app itself. This offloads user-account management to the secure environment and eliminates the need for an internal user-database.

- **In-Memory Audit Logging**

Operational events—such as "scan loaded," "ICP alignment started/finished," and "session closed"—are captured in a transient, in-RAM log buffer that contains no patient identifiers or raw image data. When the application exits, the entire buffer is purged, ensuring that no personally identifiable information persists beyond the session.

- **Data Minimization & Anonymization**

Any metadata required during a session (e.g., scan dimensions, timestamp) is kept only in transient memory and never linked to a patient's personal information. Direct identifiers are never imported or displayed, ensuring compliance with GDPR/HIPAA principles without needing a database.

This approach guarantees that **no** user or patient data is stored beyond the life of a single session, while all communication channels remain fully encrypted and integrated into the secure host environment.

3.4 Integration and Testing Strategies

Our testing pipeline—defined in the SurgeVision Test Plan—combines automated unit tests, staged integration checks, and real-world system validations:

1. **Unit Testing**
    - **C++ Core Modules** use Google Test to verify every class (e.g. MedicalDataLoader, StatisticalFilter, ICPAligner) against known inputs and edge cases.
    - **C# Interface Components** employ NUnit to exercise ViewModels, service wrappers, and utility functions.
    - A GitHub Actions workflow runs these suites on every pull request, enforcing ≥ 80 % code coverage before merge.
2. **Integration Testing**
    - **Pipeline Scenarios** ("Import → Preprocess → Register", "AR Stream → Simplify → Visualize") are executed via scripted tests that compare intermediate outputs (point counts, RMSE metrics) against golden datasets.
    - Continuous integration spins up a headless Unity instance to confirm that the client libraries and rendering engine interoperate correctly.
3. **Performance & Stress Testing**
    - Using Unity Profiler and custom logging hooks, we verify that a 10 million-point cloud can be registered in ≤ 2 s and rendered at ≥ 25 FPS on target hardware (desktop GPU and Meta Quest 3).
    - Repeated load cycles detect memory leaks and ensure stability over multi-hour sessions.
4. **System & Acceptance Testing**
    - End-to-end workflows are tested on both Windows desktops and standalone VR headsets to confirm UI commands, calibration routines, and error-reporting all function without unhandled exceptions.
    - Clinical users perform a scripted User Acceptance Test (UAT), exercising features like point picking, heat-map toggles, and alignment validation, providing feedback on usability and reliability.

5. **Regression & Release Management**
    - All test results are aggregated in Jira and SonarQube dashboards.
    - A nightly build that passes every test becomes the Release Candidate for the next sprint.
    - Critical bugs block promotion; only builds with zero high- or critical-severity regressions are tagged for deployment.

This layered approach—unit, integration, performance, system, and acceptance testing—ensures that every SurgeVision release meets our functional requirements and delivers a robust, secure experience in both desktop and immersive environments.

4 Testing and Validation

4.1 Testing Methodology

To ensure functional correctness, usability, and robustness of the SurgeVision system, a layered testing strategy was implemented covering unit, integration, system, performance, and acceptance testing.

4.1.1 Unit Testing

Each module was tested in isolation using automated frameworks (Google Test for C++ and NUnit for C#). The goal was to validate method-level logic, boundary conditions, and exception handling. Example cases include validation of .nii to .ply conversion (MRI Module), and filtering effectiveness in point cloud simplification.

4.1.2 Integration Testing

Integration tests validated the data flow across modules such as MRI → Simplifier → Surface Model → Visualization. Emphasis was placed on ensuring point cloud accuracy and proper registration under 5mm deviation using both ICP and CPD algorithms.

4.1.3 System Testing

System-level tests simulated real-world usage such as "MRI Upload → Scan → Register → Visualize". These tests were conducted on Meta Quest 3 and high-performance PCs, assessing end-to-end stability and interaction flows.

4.1.4 Performance Testing

The system was benchmarked under high load to confirm it meets the goal of ≥25 FPS during AR/VR rendering and ≤2-second registration for 10 million-point clouds. Unity Profiler and Android Logcat tools were used to monitor FPS, memory usage, and responsiveness.

4.1.5 User Acceptance Testing (UAT)

Conducted with medical students and clinicians, UAT assessed usability, clarity, and interaction quality. Participants performed predefined tasks, with feedback collected via Likert-scale surveys and structured observation forms.

4.1.6 Beta Testing

Real-world testing took place in simulated hospital settings, allowing domain experts to use the system in authentic workflows. Insights gathered here informed several usability and robustness enhancements prior to final release.

4.2 Test Cases and Results

Over 20 test cases were defined, covering both functional (e.g., MRI module output accuracy, UI response) and non-functional aspects (e.g., FPS stability). The system passed all critical-path scenarios. Test results were tracked using Jira and integrated into CI pipelines via GitHub Actions.

4.3 Test Environment

Hardware: Meta Quest 3, ARCore-enabled devices, high-performance PCs.

Software: Unity (v2021+), Android SDK, NUnit, Google Test.

Simulation Settings: Controlled lab setups and live environments were both used to simulate clinical use cases.

4.4 Defect Management and Bug Resolution

Defects were logged in Jira with detailed reproduction steps and severity tagging. Developers were assigned issues based on availability. Fixes were verified through regression tests and only builds with zero high-severity bugs were promoted to release.

5. Performance Analysis

5.1 System Performance Evaluation

5.1.1 FPS and Real-time Rendering

Under heavy loads (10M+ point clouds), SurgeVision consistently maintained rendering performance above 25 FPS on both desktop and Quest 3 environments. The Unity engine's GPU batching and point cloud streaming optimizations ensured smooth visual output.

5.1.2 Data Processing Efficiency

End-to-end processing—from MRI upload to final AR rendering—was achieved in under 5 seconds on high-end systems. Registration and simplification steps were parallelized to reduce latency, leveraging multi-threading libraries (OpenMP, TBB).

5.2 Stress and Load Testing

SurgeVision underwent repeated scanning and rendering cycles to test for memory leaks, heat buildup, and stability over extended periods. No critical faults were observed. The system handled point clouds exceeding 15M points without crashes or frame drops.

5.3 User Feedback and System Usability

Feedback from UAT and beta users highlighted high satisfaction in realism, ease of use, and system responsiveness. Improvements were made based on feedback, such as increasing gesture recognition reliability and adding heatmap-based visual aids for alignment accuracy.

6. Challenges and Limitations

6.1 Technical Challenges

Real-time ICP Precision:

During registration of live ARCore scans with preprocessed MRI point clouds, the ICP algorithm occasionally failed to converge under 5mm accuracy when the user's environment lacked sufficient surface texture or contained reflective artifacts (e.g., glass walls in the hospital room). This forced us to fine-tune the voxel grid and implement fallback recovery logic using CPD.

Multithreaded Pipeline Synchronization:

Synchronizing concurrent threads for point cloud simplification, registration, and visualization introduced race conditions, especially when MRI data was large. We encountered deadlocks when Unity's main thread was blocked by preprocessing tasks, requiring a redesign using asynchronous coroutines and message passing.

ARCore Device Inconsistencies:

Different Android smartphones produced slightly different point cloud densities and noise profiles due to varying ARCore SDK versions and hardware specs, impacting registration quality. This necessitated device-specific calibration routines.

6.2 User Interaction Challenges

Gesture Misdetection in Low Light:

In dimly lit operating rooms, the Quest 3 hand tracking system struggled to detect fine gestures like "pinch-and-hold," resulting in unintended selections or missed interactions. One user reported being unable to rotate the model during a training session, prompting us to introduce a fallback option using controller input and adaptive gesture timeouts.

Misalignment During Spatial Anchoring:

During calibration, if the headset wasn't facing the marker directly or the user moved too early, spatial anchors were misaligned, leading to a significant mismatch between the virtual anatomy and physical space. This led to user frustration and retrials.

Learning Curve in VR UI:

Medical professionals unfamiliar with VR controls found it unintuitive to use the heatmap toggle and session replay tools at first. We had to simplify the UI hierarchy and introduce contextual voice prompts.

6.3 Hardware and Software Constraints

Meta Quest 3 Memory Bottleneck:

While loading 10M+ point clouds for complex anatomical scans, the Meta Quest 3 frequently ran out of memory and crashed the Unity application. We had to aggressively downsample high-resolution models on the server before streaming to the headset and redesign our renderer to use GPU instancing and batching.

Unity Version Compatibility:

Upgrading Unity for better performance often broke compatibility with existing ARCore plugins or introduced unexpected rendering bugs in the Quest 3 headset. Each upgrade required retesting the full rendering pipeline.

Limited GPU Parallelism on Mobile:

Unlike desktop GPUs, the mobile GPU on Quest 3 had limited support for compute shaders, forcing us to rewrite high-performance visualizations (e.g., clipping planes and color-mapped overlays) using optimized vertex buffers instead.

6.4 Future Improvements and Enhancements

AI-Assisted Registration and Landmark Detection:

To reduce reliance on rigid ICP/CPD algorithms and operator fine-tuning, we plan to integrate a CNN-based model that detects anatomical landmarks automatically and aligns point clouds using learned correspondences. This would enable faster, more robust alignment even when input data is noisy or partially occluded.

Multi-User Collaboration in Shared AR Space:

Enabling multiple surgeons or trainees to interact with the same virtual anatomy in a synchronized shared session (via networked AR) could vastly improve training realism. This would require real-time data streaming and consistent spatial mapping across devices.

Persistent Cloud Storage Integration:

Storing anonymized session data, user preferences, and metrics securely in the cloud could enable long-term tracking of trainee performance and progress analysis—pending proper GDPR-compliant design.

7. Conclusion

7.1 Summary of Achievements

SurgeVision successfully achieved its goal of creating a real-time, secure, and immersive AR/VR platform for surgical planning and training. The system accurately converted MRI and AR scans into point clouds, performed efficient simplification, and aligned data using hybrid registration methods. Real-time visualization on Meta Quest 3 and desktop platforms was implemented with an intuitive interface using gestures or controllers.

Key accomplishments include:

- Sub-5mm point cloud registration accuracy
- Real-time rendering at ≥25 FPS with large datasets
- A gesture-based, responsive UI design
- Full TLS-secured operation with no persistent PII

7.2 Impact on Surgical Planning and Training

SurgeVision provides a significant improvement over traditional 2D surgical planning tools by introducing immersive, 3D visualizations. This helps surgeons better understand patient anatomy before operations, reducing risk and improving confidence. Trainees benefit from an interactive, simulation-based learning platform that supports hands-on practice without patient involvement.

Pilot testing with clinicians and medical students revealed that SurgeVision enhances spatial understanding and supports decision-making. The ability to manipulate 3D models, inspect alignment heatmaps, and simulate procedures offers a novel educational experience that complements theoretical learning.

7.3 Final Thoughts and Recommendations

SurgeVision demonstrates how real-time AR/VR visualization can revolutionize the surgical workflow. It is a scalable, secure, and high-performance system that bridges clinical, educational, and technological domains.

For future versions, we recommend:

- Adding AI-driven automation in segmentation and registration
- Supporting shared multi-user sessions for collaborative planning
- Expanding compatibility to other surgical disciplines beyond orthopedics
- Introducing cloud storage with GDPR-compliant encryption for longitudinal studies

With these additions, SurgeVision can evolve into a full-fledged surgical assistance platform, improving safety, efficiency, and training outcomes in modern healthcare.

8. References

1. **ARCore Documentation** – Google Developers.
   https://developers.google.com/ar

2. **Besl, P. J., & McKay, N. D. (1992).** "A Method for Registration of 3-D Shapes." *IEEE Transactions on Pattern Analysis and Machine Intelligence.*
   [DOI:10.1109/34.121791] (for ICP)

3. **Myronenko, A., & Song, X. (2010).** "Point Set Registration: Coherent Point Drift." *IEEE Transactions on Pattern Analysis and Machine Intelligence.*
   [DOI:10.1109/TPAMI.2009.86]

4. **Point Cloud Library (PCL) Documentation.**
   https://pointclouds.org/

5. **FAISS – Facebook AI Similarity Search.**
   https://github.com/facebookresearch/faiss

6. **FLANN: Fast Library for Approximate Nearest Neighbors.**
   Muja, M. & Lowe, D. G. (2009). "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." *VISAPP (1)*.
   https://www.cs.ubc.ca/research/flann/

7. **Unity XR Interaction Toolkit Documentation.**
   https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@latest

8. **TLS 1.2 Specification** – IETF RFC 5246.
   https://tools.ietf.org/html/rfc5246

9. **HIPAA Compliance Summary** – U.S. Department of Health and Human Services.
   https://www.hhs.gov/hipaa/for-professionals/index.html

10. **GDPR Summary** – EU GDPR Information Portal.
    https://gdpr-info.eu/

11. **MVVM Pattern Overview** – Microsoft Docs.
    https://learn.microsoft.com/en-us/archive/msdn-magazine/2009-february/wpf-in-silverlight-the-mvvm-pattern

12. **NIfTI File Format Specification** – Neuroimaging Informatics Technology Initiative. https://nifti.nimh.nih.gov/nifti-1

13. **OpenMP API Specification.**
https://www.openmp.org/specifications/

14. **Intel Threading Building Blocks (TBB).**
https://www.intel.com/content/www/us/en/developer/tools/oneapi/threading-building-blocks.html

15. **Marching Cubes Algorithm:** Lorensen, W.E., & Cline, H.E. (1987). "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." *ACM SIGGRAPH.*

9. Appendices

9.1 Glossary of Terms

Term	Definition
AR (Augmented Reality)	A technology that overlays digital content (e.g., 3D models, data) onto the physical world in real-time using devices like smartphones or AR headsets.
ARCore	Google's platform for building AR experiences on Android devices, enabling motion tracking, environmental understanding, and light estimation.
DICOM (Digital Imaging and Communications in Medicine)	A standard for storing and transmitting medical imaging information. Commonly used for CT and MRI data.
FAISS (Facebook AI Similarity Search)	A library for efficient similarity search and clustering of dense vectors, often used in high-dimensional point cloud tasks.
FLANN (Fast Library for Approximate Nearest Neighbors)	A library for performing fast approximate nearest neighbor searches in high-dimensional spaces.
FPS (Frames Per Second)	A measurement of how many frames (images) are rendered per second in a visual application. A higher FPS implies smoother visualization.
GPU (Graphics Processing Unit)	A hardware component designed for high-speed image processing and rendering, essential for AR/VR performance.

Term	Definition
HIPAA (Health Insurance Portability and Accountability Act)	A U.S. regulation that sets the standard for protecting sensitive patient health information.
ICP (Iterative Closest Point)	A rigid point cloud registration algorithm that minimizes the distance between corresponding points in two point sets.
Mesh	A 3D object composed of vertices, edges, and faces that define its shape, typically generated from point clouds using algorithms like Marching Cubes.
MRI (Magnetic Resonance Imaging)	A non-invasive medical imaging technique used to visualize detailed internal structures of the body, particularly soft tissues.
MVVM (Model-View-ViewModel)	A software architecture pattern that separates data (Model), user interface (View), and logic (ViewModel), promoting maintainable code.
NIfTI (Neuroimaging Informatics Technology Initiative)	A file format (.nii) used for storing MRI/CT data, particularly in neuroimaging applications.
OpenMP / TBB	Parallel programming libraries for C++ that enable efficient use of multicore CPUs. OpenMP is compiler-based; TBB is template-based.
PCL (Point Cloud Library)	An open-source library for 2D/3D image and point cloud processing, including filtering, registration, segmentation, and surface reconstruction.

Term	Definition
PII (Personally Identifiable Information)	Any data that could potentially identify a specific individual (e.g., name, ID, medical record). Must be protected under GDPR/HIPAA.
Point Cloud	A collection of 3D data points representing the external surface of an object or environment, typically used in 3D modeling and analysis.
Real-Time Rendering	The process of generating images or frames in real-time, typically at ≥25 FPS for smooth AR/VR experiences.
SLAM (Simultaneous Localization and Mapping)	A computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location.
TLS (Transport Layer Security)	A cryptographic protocol ensuring secure communication over a network, commonly used in web and application security.
Unity	A cross-platform game engine used to build interactive 2D and 3D experiences, including AR/VR applications.
Voxel Grid Downsampling	A method to reduce the number of points in a point cloud by grouping them into a regular 3D grid (voxels) and keeping one representative point per voxel.
XR (Extended Reality)	A collective term for AR, VR, and Mixed Reality (MR) technologies.

Term	Definition
Zlib/Znzlib	Compression libraries used for efficient data storage and transmission, commonly applied in handling medical image formats.

9.2 Code Snippets and Algorithm Descriptions

- **Oculus Integration**

  Oculus Integration refers to incorporating Meta (Oculus) VR hardware into development platforms like Unity. It provides tools, prefabs, and APIs for accessing VR input, hand tracking, headset tracking, and rendering functionalities for immersive VR and Mixed Reality experiences. We used the Oculus SDK to customize locomotion, interactions, and environment settings, enabling full control over the VR experience. It's essential for creating apps that leverage Oculus-specific features like passthrough or spatial anchors.

```
public class OculusAccess : MonoBehaviour
{
    void Update()
    {
        // 1. Headset Position & Rotation
        if (XRDevice.isPresent)
        {
            Vector3 headPos = InputTracking.GetLocalPosition(XRNode.Head);
            Quaternion headRot = InputTracking.GetLocalRotation(XRNode.Head);
            Debug.Log("Headset Pos: " + headPos + ", Rot: " + headRot.eulerAngles);
        }

        // 2. Left Controller
        Vector3 leftPos = InputTracking.GetLocalPosition(XRNode.LeftHand);
        Quaternion leftRot = InputTracking.GetLocalRotation(XRNode.LeftHand);
        Debug.Log("Left Controller Pos: " + leftPos);

        // 3. Input Example: Check if trigger is pressed
        if (OVRInput.Get(OVRInput.Button.PrimaryIndexTrigger))
        {
            Debug.Log("Primary Trigger Pressed");
        }

        // 4. Hand Tracking (if enabled in Oculus settings)
        if (OVRInput.IsControllerConnected(OVRInput.Controller.Hands))
        {
            Vector3 handPos = OVRInput.GetLocalControllerPosition(OVRInput.Controller.Hands.Left);
            Debug.Log("Left Hand Pos (Tracking): " + handPos);
        }
    }
}
```

Figure 1 : Oculus Integration Access Code Snippet

- **Passthrough Logic**

  Passthrough logic enables a VR headset to display the real-world environment through its built-in cameras. In Unity, we used this feature to blend virtual elements with the real world, creating Mixed Reality (MR) applications where digital objects coexist with physical surroundings. We used in this application because our app requires user awareness of their real environment, such as product visualization, interactive learning, or safety-aware AR experiences.

```
public class PassthroughController : MonoBehaviour
{
    public OVRPassthroughLayer passthroughLayer;

    void Start()
    {
        if (passthroughLayer != null)
        {
            passthroughLayer.enabled = true;  // Enable Passthrough
            passthroughLayer.overlayType = OVROverlay.OverlayType.Overlay; // Optional
            passthroughLayer.edgeRenderingEnabled = true; // Optional outline
        }

        // Enable global Passthrough
        if (OVRManager.instance != null)
        {
            OVRManager.instance.isInsightPassthroughEnabled = true;
            Debug.Log("Passthrough enabled");
        }
        else
        {
            Debug.LogWarning("OVRManager not found in scene!");
        }
    }
}
```

Figure 2 : Passthrough Logic Access Code Snippet

- **Depth API**

  The Depth API allows developers to access depth information captured by a VR headset's sensors. This information describes how far objects are from the headset, enabling features such as real-world surface detection, occlusion handling, and 3D reconstruction.

  By converting depth maps into point clouds or meshes, we can interact with the real environment in a geometry-aware manner, enhancing realism in AR/VR applications.

- **Point Cloud**

  A point cloud is a collection of data points defined in 3D space. Each point represents a position (x, y, z) and may include additional properties like color or intensity.
  Point clouds are generated from depth sensors and LiDAR are used in 3D scanning and medical imaging**.**

- **Point Cloud Library**

  The Point Cloud Library (PCL) is a large-scale, open-source C++ library designed for 3D point cloud processing. It includes algorithms for filtering, feature estimation, segmentation, surface reconstruction, and registration of point clouds. PCL is widely used in academic and industrial applications involving robotics, autonomous vehicles, 3D modeling, and SLAM (Simultaneous Localization and Mapping).

Table 1. Performance Metrics of ICP, GICP, and CPD (+25° Rotation Test, 870,195 Points)

Algorithms	Iteration	Completion Time	RMSE	Translation Error	Rotation Error
ICP	100	1084.54 seconds	2.02086	0.0990939	0.0998959
GICP	100	5787.44 seconds	3.08329	0.105818	0.0395647
CPD	100	4771.01 seconds	1.31061	20.4549	23.3066

Figure 1. Visualization of the algorithm's output on a +25° rotated point cloud of 870,195 points.

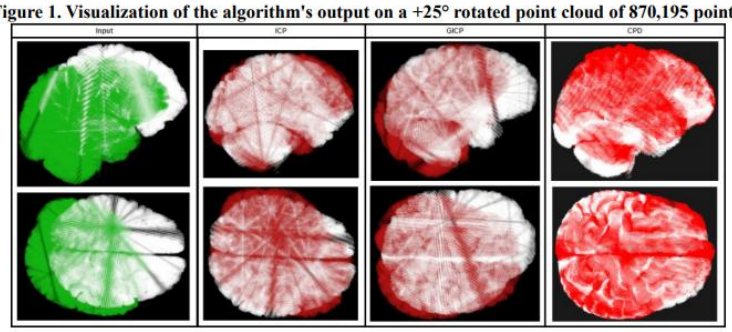

Figure 3 : Comparisons of Different Algorithm Libraries on High Point Clouds

Table 2. Performance Metrics of ICP, GICP, and CPD (+25° Rotation Test, 117,506 Points)

Algorithms	Iteration	Completion Time	RMSE	Translation Error	Rotation Error
ICP	100	136.909 seconds	2.49419	0.110609	0.155766
GICP	100	709.717 seconds	2.70053	0.145894	0.196832
CPD	100	603.615 seconds	1.80678	20.6976	22.1105

Figure 2. Visualization of the algorithm's output on a +25° rotated point cloud of 117,506 points.

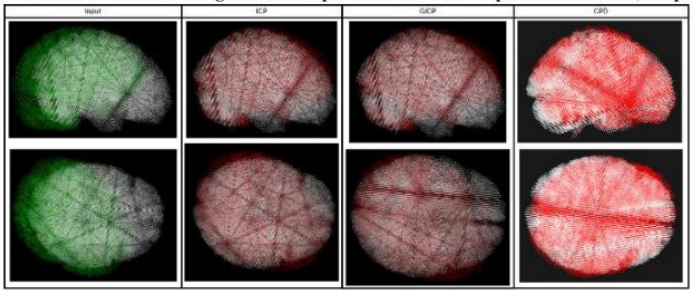

Figure 4 : Comparisons of Different Algorithm Libraries on Low Point Clouds

- **ICP Algorithm**

  The Iterative Closest Point (ICP) algorithm is used to align two point clouds by minimizing the distance between their corresponding points. It iteratively refines the transformation (rotation and translation) needed to best align a source cloud with a target cloud.

  We used ICP for object tracking, environment merging, or registering real-world scans with virtual models in AR/VR systems.

```cpp
using namespace std;
using namespace Eigen;

typedef vector<Vector3d> PointCloud;

// Find nearest neighbor in B for each point in A
vector<int> findNearestNeighbors(const PointCloud& A, const PointCloud& B) {
    vector<int> indices;
    for (const auto& a : A) {
        double minDist = numeric_limits<double>::max();
        int bestIndex = -1;
        for (int i = 0; i < B.size(); ++i) {
            double dist = (a - B[i]).squaredNorm();
            if (dist < minDist) {
                minDist = dist;
                bestIndex = i;
            }
        }
        indices.push_back(bestIndex);
    }
    return indices;
}
```

Figure 5 : Base ICP Algorithm Code Snippet

```cpp
// Compute best-fit transform between matched points
void bestFitTransform(const PointCloud& A, const PointCloud& B, Matrix3d& R, Vector3d& t) {
    Vector3d centroid_A = Vector3d::Zero();
    Vector3d centroid_B = Vector3d::Zero();

    for (int i = 0; i < A.size(); ++i) {
        centroid_A += A[i];
        centroid_B += B[i];
    }

    centroid_A /= A.size();
    centroid_B /= B.size();

    MatrixXd H = MatrixXd::Zero(3, 3);
    for (int i = 0; i < A.size(); ++i) {
        Vector3d a = A[i] - centroid_A;
        Vector3d b = B[i] - centroid_B;
        H += a * b.transpose();
    }

    JacobiSVD<MatrixXd> svd(H, ComputeFullU | ComputeFullV);
    R = svd.matrixV() * svd.matrixU().transpose();

    // Handle reflection
    if (R.determinant() < 0) {
        Matrix3d V = svd.matrixV();
        V.col(2) *= -1;
        R = V * svd.matrixU().transpose();
    }

    t = centroid_B - R * centroid_A;
}
```

Figure 6 : Base ICP Algorithm Code Snippet

```cpp
// Main ICP function
void ICP(const PointCloud& source, const PointCloud& target, Matrix3d& R_final, Vector3d& t_final, int maxIterations = 20, double tolerance = 1e-6) {
    PointCloud src = source;
    R_final = Matrix3d::Identity();
    t_final = Vector3d::Zero();

    for (int iter = 0; iter < maxIterations; ++iter) {
        vector<int> indices = findNearestNeighbors(src, target);

        PointCloud matched;
        for (int idx : indices) {
            matched.push_back(target[idx]);
        }

        Matrix3d R;
        Vector3d t;
        bestFitTransform(src, matched, R, t);

        // Apply transformation
        for (auto& point : src) {
            point = R * point + t;
        }

        // Update total transformation
        R_final = R * R_final;
        t_final = R * t_final + t;

        // Check for convergence
        double meanError = 0.0;
        for (int i = 0; i < src.size(); ++i) {
            meanError += (src[i] - matched[i]).norm();
        }
        meanError /= src.size();

        if (meanError < tolerance) break;
    }
}
```

Figure 7 : Base ICP Algorithm Code Snippet

9.3 Additional Photos From Application

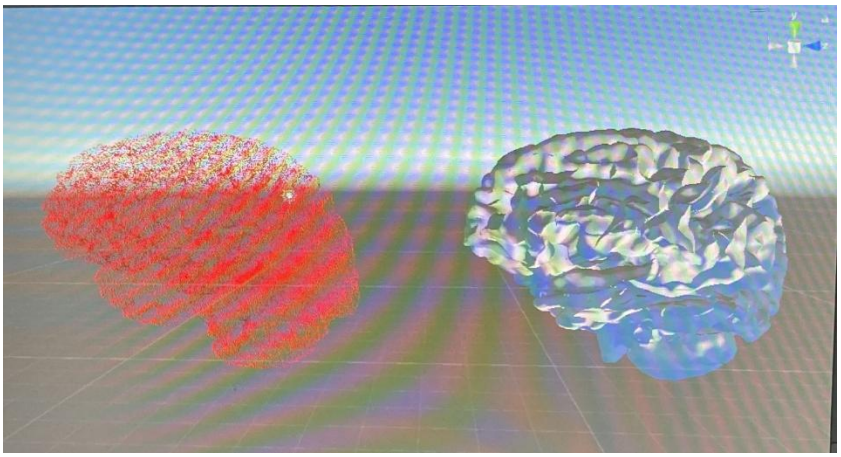

Figure 8 : Brain MRI 3D visualization and App's Point Cloud

Figure 9 : 3D Printed Skull for Presenting

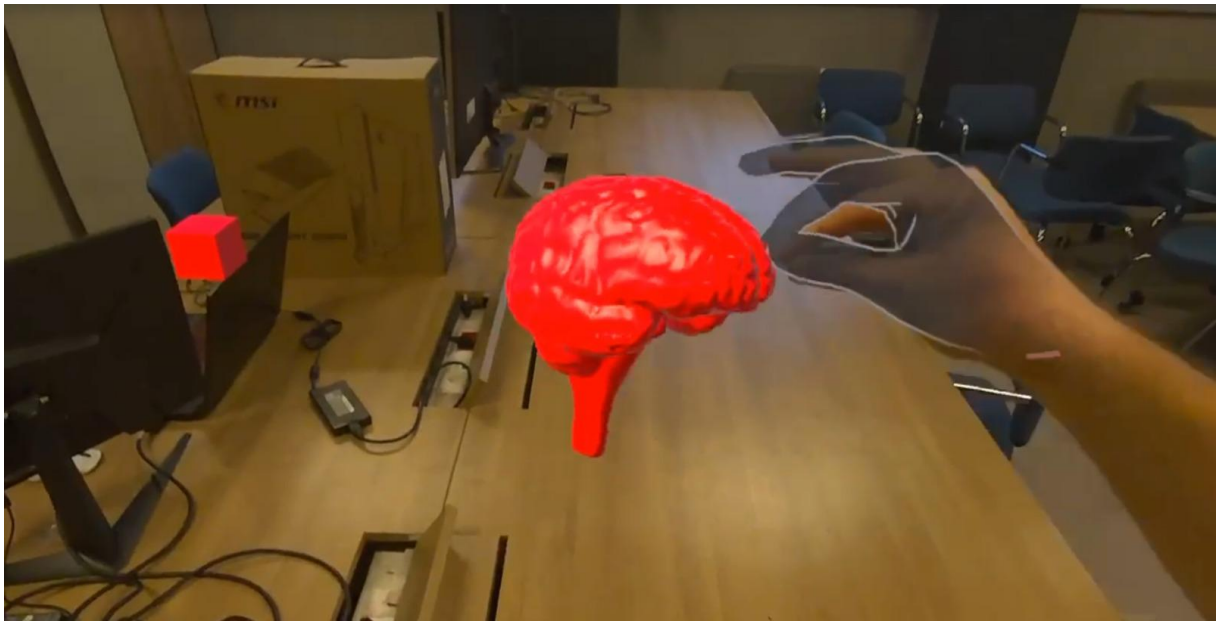

Figure 10 : Pinching and Moving the 3D Model of Brain

Figure 11 : Pinching and Moving the 3D Model of Brain

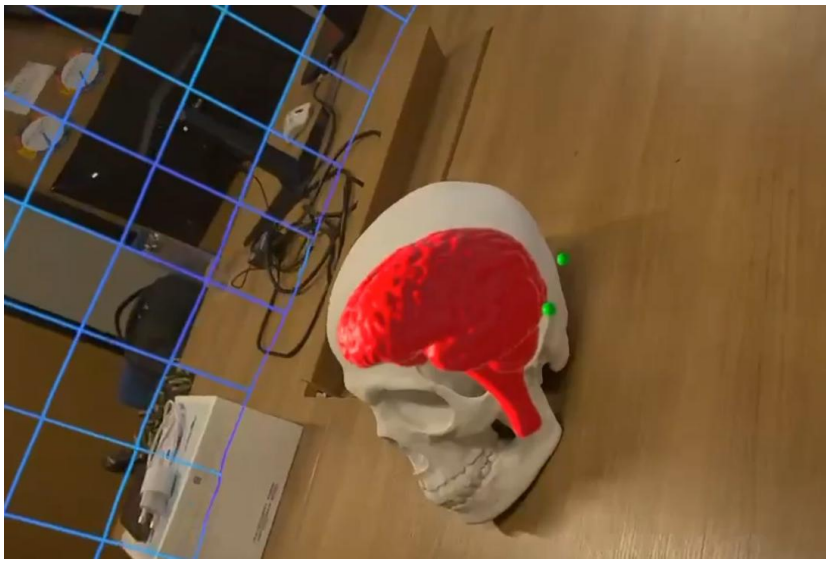

Figure 12 : Brain MRI 3D Model and Skull Matching in AR Environment (Side View)

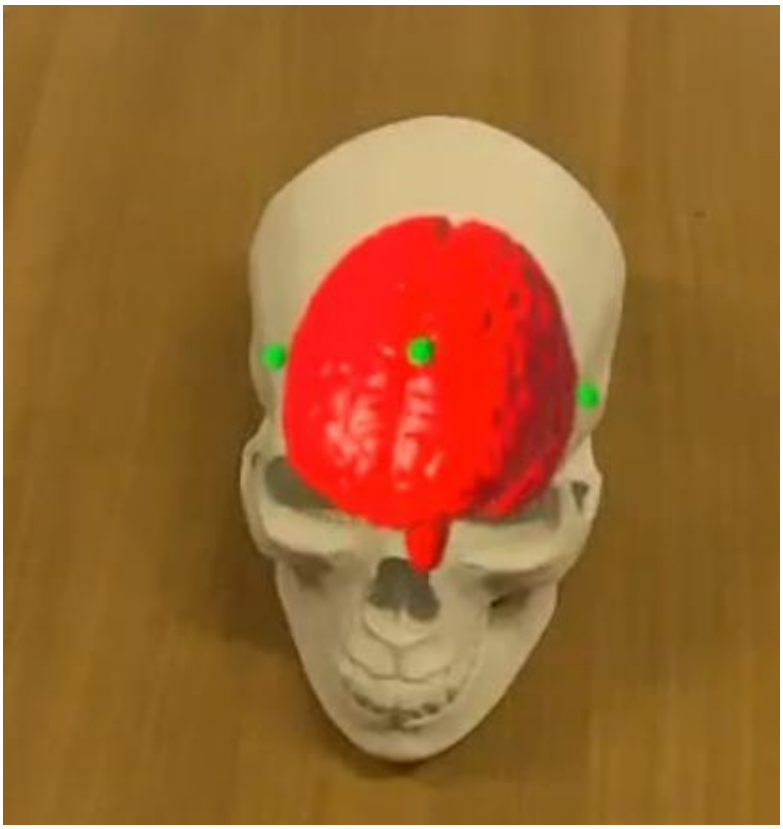

Figure 13 : Brain MRI 3D Model and Skull Matching in AR Environment (UpFront View)

Figure 14 : Skull in AR Environment