UFR de mathématique et d'informatique
Université de Strasbourg

SIEMENS

Université de Strasbourg

Internship Report

Reactive trajectory planning for robotic operations in an unstructured environment simulated by Tecnomatix Process Simulate

Author:
Giulio Carpi Lapi

Supervisors:
Gautier Dumonteil
Guillaume Mouillet

August 27, 2025

Abstract

This report details the integration of a **reactive trajectory planning** system into **Tecnomatix Process Simulate**. The project's primary objective was to enable a **robotic arm** to **dynamically perceive and avoid obstacles** using real-time data from **RGB-D cameras**. The work involved developing a **multi-layered software solution** spanning **C#/WPF** for the user interface, **C++/CLI** for the managed/native bridge, and a **high-performance C++ core** for simulation and data processing. Key contributions include the implementation of a **custom simulation loop**, **event-driven point cloud capturing**, and a **two-stage filtering algorithm** to isolate dynamic obstacles by first removing the robot's own geometry and then filtering out the static environment. The system was successfully validated in a simulated **bin-picking scenario**, demonstrating its capability to detect unexpected objects and halt operations to prevent collisions, laying the groundwork for fully autonomous path replanning.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, **Gautier Dumonteil**, Kineo Components Developer, and **Guillaume Mouillet**, Product Owner of the ARK team, for their guidance, support, and mentorship throughout this internship. Their expertise and encouragement were instrumental in the successful completion of this project.

I also wish to extend my thanks to the entire **Kineo Team** and the **ARK team** at **Siemens Digital Industries Software** in Toulouse. The collaborative and welcoming environment made this a truly enriching experience.

Contents

1 Introduction

1.1 General Context and Motivation

The manufacturing industry is undergoing a profound transformation with **Industry 4.0**, integrating advanced technologies such as **Artificial Intelligence**, **Internet of Things**, **connectivity**, and **data analytics** to create smart, interconnected factories that enhance productivity, efficiency, and flexibility [1, 2, 3]. In this context, **simulation**, **digital twins**, and **Product Lifecycle Management (PLM)** are essential for optimizing processes. Simulation enables virtual modeling and testing of production systems to predict performance, identify issues, and reduce costs without real-world disruptions [4, 5, 6].

Digital twins—virtual replicas of physical assets—support real-time monitoring, predictive maintenance, and efficiency gains, minimizing downtime and improving quality [7, 8, 9]. PLM manages a product's entire lifecycle from design to disposal, integrating data and processes to streamline collaboration, cut errors, and speed time-to-market [10, 11, 12].

Automation is vital for manufacturing competitiveness amid global demands, as it lowers costs, boosts precision, enhances safety, and increases production capacity while enabling adaptability to market changes [13]. **Reactive trajectory planning** advances this by allowing robots to dynamically avoid obstacles in real-time, ensuring safe operations in unstructured environments and improving human-robot interactions [14]. Exploring **Research and Development** in this area yields benefits like optimized motion, reduced reliance on fixed paths, greater adaptability, and innovation in smart manufacturing [15].

1.2 Internship Framework

This internship was conducted within the **Kineo Team** at **Siemens Digital Industries Software** in Toulouse, France.

The work was carried out within the **Advanced Robotics Kinematics (ARK) team**, which is responsible for integrating and maintaining Kineo Components in **Tecnomatix Process Simulate**. Process Simulate is a key Siemens software for the 3D simulation, validation, and optimization of manufacturing processes [16]. Like all teams in the Kineo department the ARK team follows the **Scrum methodology**, organizing work into two-to-three-week sprints with daily stand-up meetings. Project progress and tasks were managed using **Polarion** [17], an application lifecycle management tool that facilitates efficient coordination and tracking. Version control was managed using **Git**, with **Git Extensions**.

1.3 Company and Technology Overview

Founded in 2001, **Kineo CAM** builds on over 15 years of research from the **Laboratory for Analysis and Architecture of Systems (LAAS)** and **French National Center for Scientific Research (CNRS)**. Its technology supports industries like automotive, aerospace, energy, shipbuilding, and manufacturing by optimizing robotic movements, simulating cable behaviour, and verifying assembly and disassembly processes.

Kineo's solutions include advanced software components and standalone applications for automatic motion and automatic path planning as well as collision detection. Integrated into modern **Computer-Aided Design (CAD)**, **Computer-Aided Manufacturing (CAM)**, **Computer-Aided Engineering (CAE)**, 3D digital mock-up, robotics systems, these tools enhance productivity by automating path planning and clash detection, ultimately saving customers time, costs, and resources.

1.3.1 Core Technology: Kineo SDKs and Process Simulate Integration

Kineo specializes in creating high-performance **Software Development Kits (SDKs)** for advanced robotics and geometric applications.

These SDKs are not standalone applications but are designed to be integrated into third-party software, providing powerful capabilities for path planning, collision detection, and simulation. A primary consumer of this technology is Siemens' own **Tecnomatix Process Simulate**, where Kineo's components are integrated to provide features for **Automatic Path Planning (APP)**. This project was developed as an extension of this core APP integration.

Kineo's portfolio comprises the following products:

- **KineoWorks**: Enables automatic computation of valid, collision-free trajectories for any kinematic system, addressing path planning challenges. Its pathfinding technology is a valuable tool for next-generation CAD and robotics systems with advanced motion planning and control capabilities.
- **Kineo Collision Detector (KCD)**: Facilitates spatial interference and collision checks between hierarchical assemblies of triangle mesh surfaces or polyhedrons. KCD provides generic mechanisms for handling geometrical objects, spatial collision testing, and reporting.
- **Kineo Flexible Cables**: Computes deformation, configuration, and stresses in compliant cables, such as pneumatic hoses and electrical cables. It is particularly suited for robotics simulation and control applications involving large deformations and high stresses in complex motion environments.

- **Kineo Nesting**: Optimizes the arrangement of irregular shapes within a defined space to minimize waste and maximize efficiency. Commonly used in additive manufacturing and cutting industries, it reduces costs and improves productivity through utilities like subnesting for delicate components and car trunk volume estimation.
- **KineoWorks Interact**: A rich Graphical User Interface toolkit that enables rapid development of 3D software applications, particularly suited for CAD and robotics applications.

Figure 1: The Kineo Libraries Architecture [18].

These components are built upon a layered architecture, as illustrated in Figure 1. At the base is **KineoUtility**, a foundational library providing core functionalities that all other Kineo components depend on. KineoUtility serves as the infrastructure layer, offering essential services including:

- **Mathematical Tools**: Classes for matrix operations (`CkitMat4` for 4×4 transformation matrices), vector calculations (`CkitVec3` for 3D vectors), and geometry utilities for angle conversions and interpolation.
- **Geometrical Model**: A comprehensive framework for representing 3D objects through the `CkitGeometry` class hierarchy, supporting various primitives (boxes, cylinders, spheres), polyhedrons, and assemblies with capability-based interfaces.
- **Kineo Runtime**: An introspection system enabling dynamic property access, notifications for loose coupling between objects, and support for cloning and serialization of complex object graphs.
- **Error Management**: Robust error handling through status codes, assertions for debug builds, and a logging system for debugging.

KineoUtility also acts as a bridge between KCD and other libraries through its collision toolkit concept. The main business libraries-**KineoWorks**, **Kineo Collision Detector (KCD)**, and **Kineo Flexible Cables**—all rely on this utility layer. The **KineoWorks Interact** libraries, used for building user interfaces, are layered on top, also depending on KineoUtility for their core functions. This modular design allows for robust and scalable development.

Acquired by **Siemens** in 2012, Kineo enhances Siemens' **Product Lifecycle Management (PLM)** software portfolio by integrating its motion planning and collision avoidance technologies into **Siemens**' **PLM Software Business Unit**. **Kineo**'s components are embedded in core **PLM** products such as **NX** (CAD/CAM/CAE), **Teamcenter** (PLM software), and **Tecnomatix** (digital manufacturing suite).

1.4 Project Objectives

The objective of this internship was to integrate a **reactive trajectory planner** into **Process Simulate**, enabling robotic arms to reactively avoid obstacles using real-time data from virtual **Red-Green-Blue-Depth (RGB-D)** cameras within the simulation environment. The goal was to demonstrate the potential for implementing such autonomy, with the understanding that this work relies on simulated sensors, not physical hardware. For instance, this technology would allow a stationary robotic arm to continue its primary task while dynamically avoiding mobile obstacles like **automated guided vehicles (AGVs)** or **inspection drones** that share its workspace. This capability is crucial for creating more flexible and collaborative manufacturing environments where human and robotic workflows can coexist safely and efficiently without being confined to rigidly separated zones.

To achieve this, a multi-stage process pipeline was envisioned. First, RGB-D cameras would capture snapshots to collect real-time scene data. This data would then be processed to extract depth information and generate a **point cloud** representing the surrounding environment. The point cloud would be filtered to remove the robot's own geometries and static objects, isolating dynamic obstacles. Given the limitations in Process Simulate's graphics engine, which does not support simultaneous playback of multiple operations, a custom simulation loop would be developed to enable concurrent simulation of robotic motions and dynamic elements. This was essential for testing realistic scenarios, such as a robotic arm performing a simple pick-and-place operation while a secondary operation moves an object (e.g., a box simulating a drone or AGV) to create potential collision risks. Collision detection would then be performed by comparing the filtered point cloud with the robot's geometries to identify potential obstacles. Finally, upon detection of a collision, a **path planning algorithm** would compute an alternative, collision-free trajectory for the robotic arm, ensuring safe and efficient operations in dynamic environments—potentially without interrupting the robot's movement.

2 Architectural Context and Design Patterns

The integration of Kineo's Automatic Path Planning (APP) technology into Process Simulate is built upon a sophisticated, multi-layered software-architecture. This design is essential for managing the complexity of a large-scale industrial feature that combines a high-performance native **C++ core** with a flexible, user-facing **C#/.NET** environment. This section details the foundational architecture of the APP component, outlining its layered structure and the key design patterns that ensure its robustness and extensibility.

2.1 A Layered Architecture for APP

The APP integration within Process Simulate employs a layered architecture to separate concerns, promoting modularity, maintainability, and performance. The key layers and their corresponding software components, illustrated in Figure 2, are:

- **Presentation Layer (C#):** This top-most layer handles the user interface (UI) and user interactions. It is primarily implemented in the `Engineering` solution, which defines the public API and UI components like the dialog for this project.
- **Application/Service Layer (C# & C++/CLI):** This layer orchestrates workflows and bridges the UI with the core engine. Key solutions include `OlpApi`, which provides high-level robotics services, and `Implementation`, the critical C++/CLI bridge that connects the managed and native worlds.
- **Domain/Business Logic Layer (C++):** This is the high-performance native core where computationally intensive algorithms reside. The `AppPathPlan` solution, which handles path planning and collision detection, and the `OperationSimServer` solution, which manages the simulation execution, are central to this layer.
- **Infrastructure/Utility Layer (C++ & C++/CLI):** This foundational layer provides common services and data structures. It includes core utilities within `AppPathPlan` and UI infrastructure from the `DnApplicativeInfra` solution.

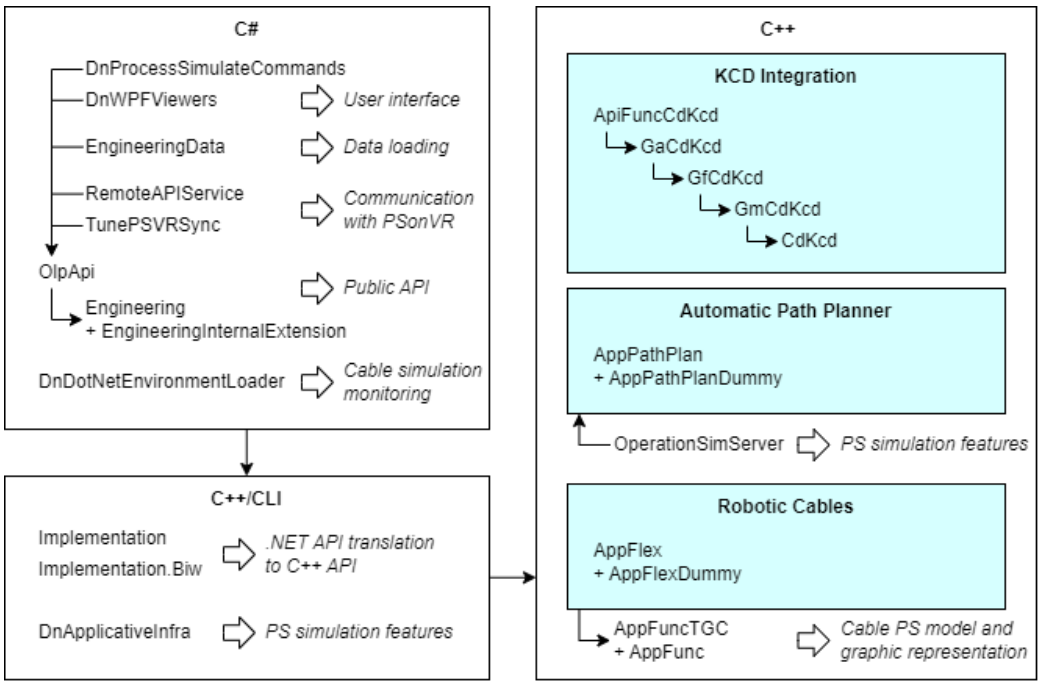


Figure 2: Process Simulate general architecture, illustrating the layered design [19].

2.2 Foundational Design Patterns in the APP Architecture

The APP framework is built on a set of established software design patterns that ensure a clear separation of concerns. Understanding these was crucial for integrating new functionality correctly.

1. **The Command Pattern:** User actions are not executed directly but are encapsulated as command objects (e.g., `TxBasePlanningCommand` in `AppPathPlan`). This decouples the UI from the execution logic, allowing operations to be managed, queued, and run in different contexts.

2. **The Bridge Pattern:** The `Implementation` solution is a perfect example of the Bridge pattern, decoupling the C# interfaces from their native C++ implementations. This allows the C# API and the C++ core to evolve independently.

3. **The Observer Pattern:** The system uses observers to broadcast notifications when data changes, for example, to update the UI when the selection changes or when a simulation event occurs in `OperationSimServer`.

4. **The Factory Pattern:** The `ITxPathPlanningContext` interface in the `Engineering` solution acts as a factory for creating various path planning queries (e.g., `CreateBasicPlanningQuery`). This centralizes object creation and decouples client code from concrete class instantiations.

3 Implementation Across Architecture Layers

To provide a clear overview of the solution's structure, Figure 3 presents a UML class diagram illustrating the key components and their interactions across the different architectural layers. The diagram shows the flow of control, starting from the user's action in the C# UI, passing through the C++/CLI bridge, and down to the native C++ core engine where the simulation and perception logic is executed.

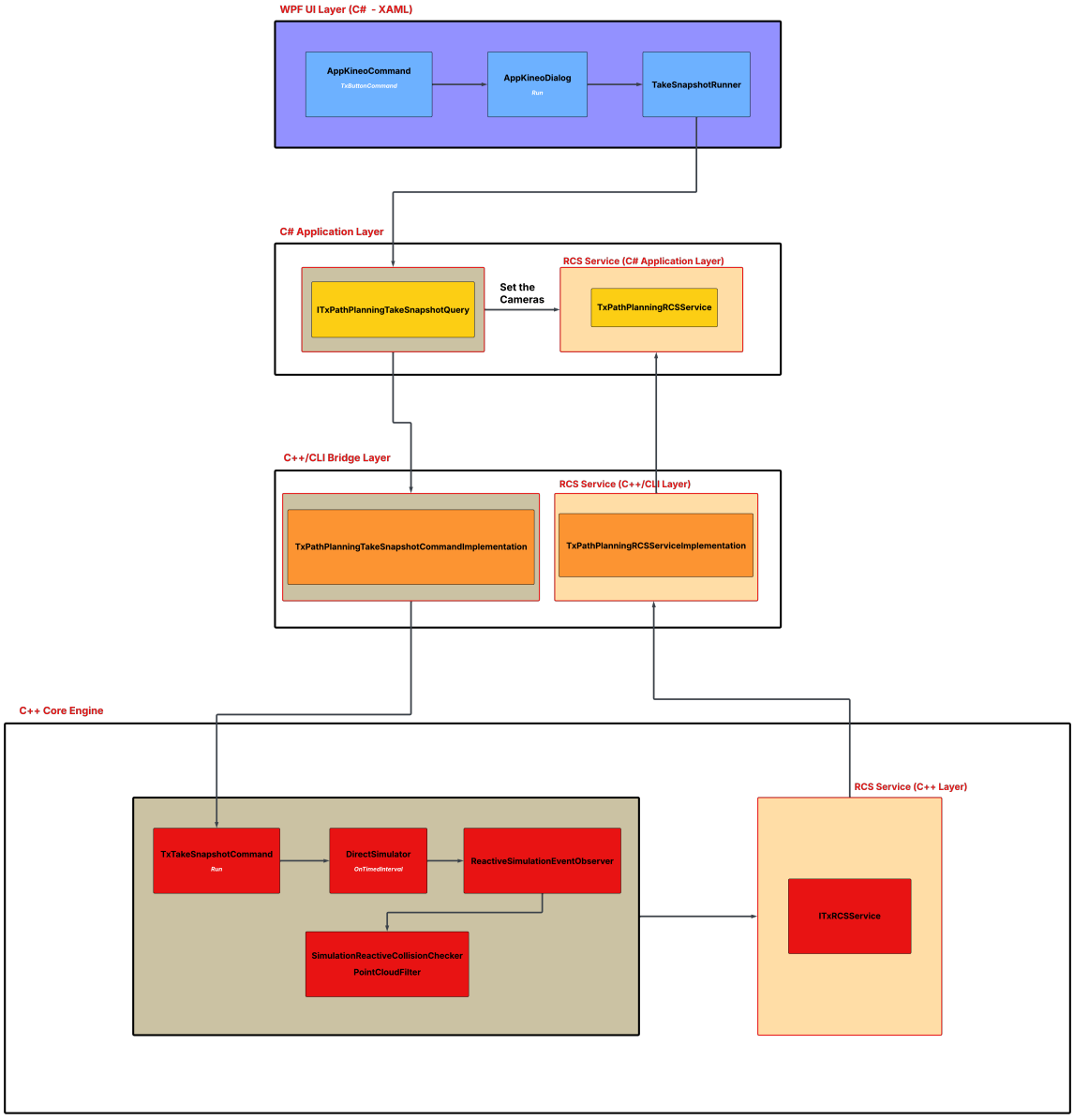


Figure 3: UML Class Diagram of the Reactive Planning Feature.

This section details my specific contributions at each layer of the architecture, following the data flow from the user's initial action in the UI down to the core C++ processing engine.

3.1 C# Application and UI Layer (`Engineering`, `OlpApi`, and `APP_API_Commands`)

This is the highest level, where the user workflow is initiated and orchestrated.

3.1.1 User Interface Design and Human-Computer Interaction

The `APPKineoDialog`, implemented using **Windows Presentation Foundation (WPF)**, serves as the primary user interface for the feature. As shown in Figure 4, it was designed to balance functionality with usability, following Process Simulate's UI design guidelines.

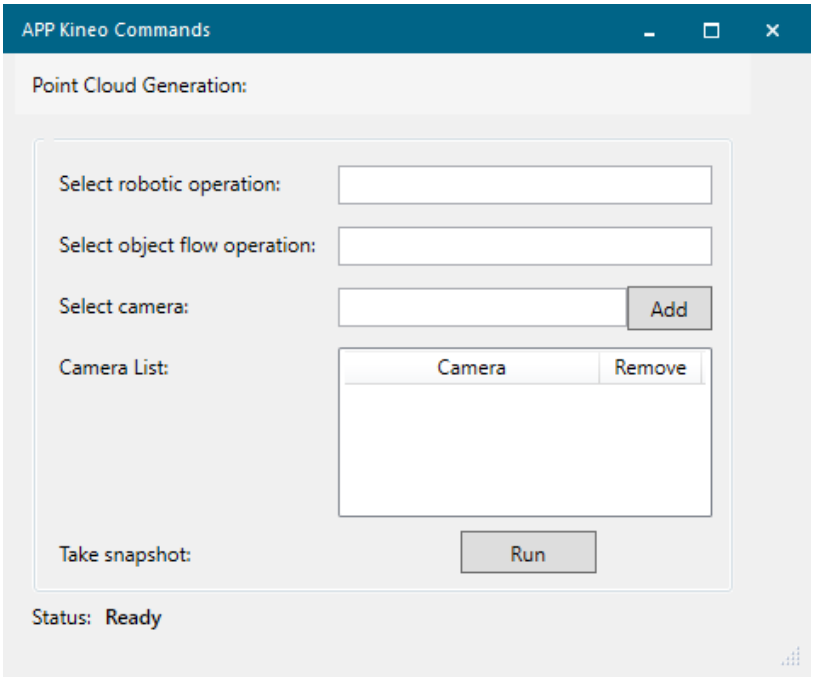


Figure 4: The dialog interfaces in Process Simulate.

Dialog Structure and Layout: The dialog employs a **Grid-based layout system**, ensuring a responsive and visually consistent interface. It is organized into three distinct sections: an operation selection area, a dynamic camera list, and a real-time status display.

Interactive Components:

- **Robotic Operation Selection:** Allows selection of the robotic motion to be monitored.
- **Object Flow Operation Selection:** Enables selection of moving objects whose interaction with the robot will be analyzed.
- **Multi-Camera Management:** Supports adding multiple RGB-D cameras to capture the scene from different angles.

User Workflow and Interaction Flow: The user can pick operations directly from the 3D scene, build a camera list dynamically, and monitor operation status in real-time. The dialog includes comprehensive state management to prevent concurrent operations and provide visual feedback through color-coded status messages (blue for processing, green for success, red for errors).

3.1.2 Command Orchestration

- **Command Pattern Implementation (`APPKineoCommand.cs`):**
  A `TxButtonCommand` was created to launch the dialog from the Process Simulate toolbar. This command encapsulates the initialization logic for the UI and the associated runner and context objects.
- **Operation Runner (`TakeSnapshotRunner.cs`):** The `TakeSnapshotRunner` class orchestrates the process on the C# side. It gathers the user's inputs and constructs the `ITxPathPlanningTakeSnapshotQuery`, which formally defines the request for the backend.

3.2 C# and C++/CLI Bridge (Implementation Solution)

This layer translates the request from the managed C# world into an action that the native C++ engine can understand.

- **API Contract Definition:** The new `ITxPathPlanningTakeSnapshotQuery` interface was created in the **Engineering** solution to establish a clear data contract for the feature. This interface includes a `List<TxViewCamera> Cameras` property to support the multi-camera functionality.
- **C++/CLI Wrapper (`TxPathPlanningTakeSnapshotCommandImplementation`):** In the **Implementation** solution, the implementation relied on this C++/CLI `ref class`. Its role is to receive the managed `ITxPathPlanningTakeSnapshotQuery` from the C# layer, unbox the relevant data, and instantiate the corresponding native C++ command, `TxTakeSnapshotCommand`.
- **Bidirectional Bridge (`TxPathPlanningRCSServiceBridgeImpl`):** This existing C++/CLI bridge was extended to enable communication from C++ back to C#. Native implementations for `processMultiplePointCloud` and `exportPointCloudToXYZ` were added. These functions are called by the C++ engine and delegate their work to the C# `TxPathPlanningRCSService`, effectively marshalling the captured point cloud data from the native world back to the managed world for processing.

3.3 The Native C++ Core Engine (AppPathPlan and OperationSimServer)

This is the high-performance layer where the core logic is executed.

- **Native Command (`TxTakeSnapshotCommand`):** This C++ class in `AppPathPlan` is the backend entry point for the feature. Its `run` method orchestrated the entire process. It initialized `DirectSimulator`, a custom simulation loop designed to provide fine-grained control over the operation's execution, and attached a `ReactiveSimulationEventObserver` to it.
- **Event-Driven Data Capture (`ReactiveSimulationEventObserver`):** This class was the heart of the C++ implementation. It hooked into the `DirectSimulator`'s time steps, acting as an observer. Its `onTimedInterval` method was called periodically, and within it, the following logic was implemented:
    1. Call the `ITxRCSService` (via the C++/CLI bridge) to capture snapshots from all active cameras using the `processMultiplePointCloud` method.
    2. Instantiate a custom `PointCloud` geometry object to hold the aggregated data.
    3. Invoke the `PointCloudFilter` to process the raw data, first removing the robot's geometry and then the static environment.
    4. Trigger the `SimulationReactiveCollisionChecker` to check for collisions between the robot and the filtered (dynamic) point cloud.
    5. Set a flag to stop the simulation if a collision is detected.
- **Custom Geometry (`PointCloud`):** To represent the captured sensor data within the Kineo environment, a new `PointCloud` class that inherits from the `IGeometry` interface, was implemeted. This class wrapped a native Kineo `CkcdGeometryPointCloud`, allowing the dynamic sensor data to be treated as a standard geometric object for collision detection by the Kineo Collision Detector.
- **Custom Simulator (`DirectSimulator`):** A key challenge in this project was that Process Simulate's standard simulation engine did not support the concurrent playback of multiple, independent operations, such as a robotic motion and a separate object flow. To overcome this limitation, the `DirectSimulator` was implemented. This custom simulation engine provided the fine-grained control necessary to create a dynamic, reactive test environment.
    1. **Path Aggregation:** It first processes the primary robotic operation, converting it into a time-parameterized Kineo path object (`CkwsPath`) and calculating its total duration.

2. **Time-Stepped Loop:** The simulator enters a main loop that advances time in discrete steps (e.g., 5 ms).
3. **Pose Interpolation:** At each time step ‘t‘, it calculates the precise pose of the robotic system by querying the `CkwsPath` for its configuration at that instant. Simultaneously, if an object flow operation is provided, it identifies the two waypoints that bracket time ‘t‘ and interpolates the object's transformation matrix to determine its position. It then updates the poses of both the robot and the dynamic object in the 3D scene.
4. **Observer Invocation:** After updating the scene, it invokes the `onTimedInterval` method of the attached `ReactiveSimulationEventObserver`, triggering the entire perception pipeline (snapshot, filtering, and collision check).
5. **Reactive Control:** The loop continues until the operation is complete or until the observer signals that a collision has been detected, at which point the simulation is halted.

This approach effectively synchronizes the robot's movement with the dynamic obstacle and the perception system, enabling a realistic simulation of reactive scenarios.

4 Core Algorithms

The dynamic environment perception feature is powered by a series of core algorithms that process raw sensor data into a usable format for collision detection. This section details the methodology behind the **point cloud generation**, the **multi-stage geometric filtering pipeline**, the **collision detection mechanism** and the **path planning algorithms**.

4.1 Depth Data Processing and Point Cloud Generation

The initial stage of the pipeline converts the 2D depth data from the virtual RGB-D cameras into a 3D point cloud in the world coordinate system.

4.1.1 Depth Buffer and Camera Parameters

The depth data is captured as a **depth buffer**, represented as a 2D array of depth values stored in a 1D float list in row-major order. Each value in the buffer corresponds to the distance from the RGB-D camera to an object at a specific pixel (u, v) in the image plane.

To convert this data into 3D coordinates, intrinsic camera parameters are first computed. The **focal length** f is derived from the camera's **field of view (FOV)** and image dimensions (width W and height H):

$$f = \frac{\sqrt{W^2 + H^2}}{2 \tan\left(\frac{\text{FOV}}{2}\right)}.$$

The **principal points** (c_x, c_y), representing the image center, are calculated as:

$$c_x = \frac{W - 1}{2}, \quad c_y = \frac{H - 1}{2}.$$

4.1.2 Pixel to 3D Point Conversion

The conversion from 2D pixel coordinates to 3D points requires understanding the **camera's coordinate system**. In Process Simulate, the camera reference frame follows a specific convention, as illustrated in Figure 5.

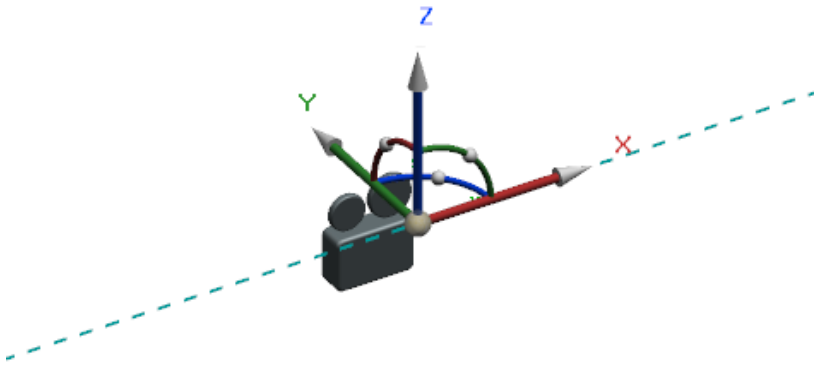


Figure 5: Camera reference frame in Process Simulate.

With this coordinate system established, for each pixel (u, v) in the depth buffer, a 3D point in camera space is computed as follows:

1. Retrieve the depth value d from the depth buffer at index $v \times W + u$:

$$d = \text{depthBuffer}[v \times W + u].$$

Note that this d value represents the distance along the camera's X-axis.

2. Compute the normalization factor t to account for the projection geometry:

$$t = \sqrt{\left(\frac{u - c_x}{f}\right)^2 + \left(\frac{v - c_y}{f}\right)^2 + 1}.$$

3. Calculate the 3D coordinates (X, Y, Z) in camera space, where the depth value maps to the X-coordinate due to the camera's reference frame:

$$X = \frac{d}{t}, \quad Y = \frac{(c_x - u) \times d}{f \times t}, \quad Z = \frac{(v - c_y) \times d}{f \times t}.$$

This process transforms each pixel's depth value into a 3D point relative to the camera's coordinate system, forming the basis for point cloud generation. To reduce computational load, a **sampling rate** (e.g., every 5th pixel) is applied, as implemented in the `ProcessDepthBuffer` method.

Coordinate frame note. Unlike the conventional computer vision frameworks such as ROS or OpenCV, where Z is forward (view), X is right, and Y is down/up, Process Simulate uses a CAD-oriented convention where the X-axis points along the viewing direction, Y points to the left, and Z points upward [20]. This explains why in our formulas, the depth value is assigned to the X-coordinate rather than the Z-coordinate. When integrating with external computer vision libraries or comparing with standard literature, this coordinate system difference must be carefully considered to ensure proper transformations.

4.1.3 Camera to World Transformation

The final step in generating the point cloud is to transform the 3D points from the camera's local coordinate system into the global world coordinate system used by Process Simulate. This is achieved using the camera's 4×4 absolute **transformation matrix**, which is retrieved from the simulation environment.

A **transformation matrix** is a powerful mathematical tool used in computer graphics to represent an object orientation (rotation) and position (translation) in space. By using **homogeneous coordinates**, a 3D point (X, Y, Z) is represented as a 4D vector $(X, Y, Z, 1)$. This allows both rotation and translation to be combined into a single matrix multiplication.

The general form of a transformation matrix (T) is:

$$T = \begin{pmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Where:

- A 3×3 **rotation matrix** $\mathbf{R}$ that defines the object's orientation.
- A 3×1 **translation vector** $\mathbf{t}$ that defines the object's position.

The transformation of a point P_{camera} in the camera's frame to a point P_{world} in the world frame is then a simple matrix-vector multiplication:

$$\begin{pmatrix} P_{world} \\ 1 \end{pmatrix} = T_{camera \to world} \times \begin{pmatrix} P_{camera} \\ 1 \end{pmatrix}$$

This operation correctly converts the point's coordinates from the camera's local reference frame to the global world reference frame.

Numerical Example: Let's consider a camera with the following transformation matrix $T_{camera \to world}$, which defines its position and orientation in the world. In this case, the camera is only translated, not rotated.

$$T_{camera} = \begin{pmatrix} 1 & 0 & 0 & 1000 \\ 0 & 1 & 0 & 500 \\ 0 & 0 & 1 & 200 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A point is captured and converted to the camera's local 3D coordinates system as:

$$P_{camera} = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix}$$

To transform this into world coordinates, we represent it in homogeneous coordinates and perform the multiplication:

$$\begin{pmatrix} P_{world} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1000 \\ 0 & 1 & 0 & 500 \\ 0 & 0 & 1 & 200 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 10 \\ 20 \\ 30 \\ 1 \end{pmatrix}$$

The resulting point in world coordinates is:

$$P_{world} = (10 + 1000, 20 + 500, 30 + 200) = (1010, 520, 230)$$

This process is repeated for every sampled point from the depth buffer to generate the final world-space point cloud. When multiple cameras are used, their individual point clouds are aggregated into a single, comprehensive cloud representing the scene from multiple viewpoints.

4.2 Two-Stage Point Cloud Filtering

To ensure the robot reacts only to unexpected, dynamic obstacles, a **two-stage filtering pipeline** is applied to the raw point cloud at each time step. This process is essential for eliminating false positives caused by the robot seeing itself or known static objects in the environment.

4.2.1 Stage 1: Robot Geometries Filtering

The first stage removes points that correspond to the robot's own body. Without this step, the robot would immediately detect a collision with itself and halt. The algorithm, implemented in the `PointCloudFilter` class, proceeds as follows:

1. **Kinematic State Acquisition:** At the current simulation time, the robot's kinematic configuration is retrieved. This provides the exact pose for every link in the robotic system.
2. **Bounding Volume Collection:** The algorithm traverses the robot's kinematic tree. For each geometric link, it retrieves its pre-existing **Oriented Bounding Box (OBB)**, which is used by the collision detection engine, and computes its position in world coordinates. These OBBs form a complete, albeit simplified, volumetric representation of the robot.
3. **Point-in-Volume Test:** Each point in the raw point cloud is tested against this set of OBBs. For a given point P_{world} and a link's OBB, the point is transformed into the OBB's local coordinate system: $P_{local} = T_{OBB}^{-1} \times P_{world}$.
4. **Filtering Logic:** The point is classified as belonging to the robot if its local coordinates (x, y, z) fall within the half-dimensions (h_x, h_y, h_z) of the OBB, with a small tolerance ϵ to account for surface variations and sensor noise:

$$|P_{local,x}| \leq h_x + \epsilon \quad \wedge \quad |P_{local,y}| \leq h_y + \epsilon \quad \wedge \quad |P_{local,z}| \leq h_z + \epsilon$$

Points that fall inside any of the robot's OBBs are discarded. All other points proceed to the next filtering stage.

4.2.2 Stage 2: Static Environment Filtering

After removing the robot's own geometry from the point cloud, a second, equally important filtering step is performed: **dynamic environment filtering**. The primary goal of this feature is to enable the robot to react to *unexpected* or *moving* obstacles. Static elements of the environment, such as part bins, fixtures, or the workpiece itself, are considered known and part of the planned operation. If these static objects were included in the final point cloud, the system would detect a collision as soon as the robot approached them to perform its task (e.g., picking a part from a bin), leading to a false positive and unnecessarily halting the operation.

To isolate only the dynamic elements, an algorithm, based on establishing a **static baseline** of the environment, was implemented in the `ReactiveSimulationEventObserver`:

1. **Baseline Snapshot:** Baseline Snapshot: At the beginning of the simulation (specifically, on the first observation event), the system captures a point cloud and performs the robot self-filtering described above. The resulting point cloud, representing the static scene, is stored as the `m_staticBaselinePoints`.
2. **Point-by-Point Comparison:** For every subsequent point cloud captured during the simulation, each point $P_{current}$ (already self-filtered) is compared against the entire baseline cloud.
3. **Proximity-Based Filtering:** A point $P_{current}$ is considered static if it is within a predefined tolerance distance, ϵ_{static} (e.g., 2 cm), of any point $P_{baseline}$ in the static baseline cloud.

$$\exists P_{baseline} \in \text{StaticBaseline} \quad \text{s.t.} \quad \|P_{current} - P_{baseline}\|_2 \leq \epsilon_{static}$$

   If a point is identified as static, it is discarded.
4. **Final Dynamic Point Cloud:** Only the points that are not close to any baseline point are retained. This final set represents only the dynamic obstacles in the scene and is the input for the collision detector.

4.3 Collision Detection

Once the dynamic point cloud is isolated, it is fed into the **Kineo Collision Detector** to check for interference with the robot. The `SimulationReactiveCollisionChecker` class orchestrates this process:

1. **Custom Geometry Creation:** The filtered list of dynamic points is encapsulated within a custom `PointCloud` geometry object, which implements the `IGeometry` interface. This allows the Kineo engine to treat the dynamic sensor data as a standard geometric primitive.

2. **Collision Set Definition:** A `CollisionSet` is created. This is a data structure that defines which pairs of objects should be checked against each other for collisions. In this case, it pairs the robot's mobile geometries (the "left-hand side") with the newly created `PointCloud` geometry (the "right-hand side"). This instructs the checker to test for collisions only between the robot and the dynamic obstacles.

3. **Collision Query:** At each time step, a `RoboticSystemCollisionCheckQuery` is executed. This query uses the robot's current pose and the defined collision set to perform the check. A **near-miss threshold** of 100mm is also configured. This is a safety margin that allows the system to detect when objects are close to colliding, not just when they are already intersecting, enabling more proactive responses.

4. **Result Analysis:** If the collision checker returns a result indicating a collision or a near-miss, a flag is set. This flag is read by the `DirectSimulator`, which then halts the simulation to prevent the impending collision.

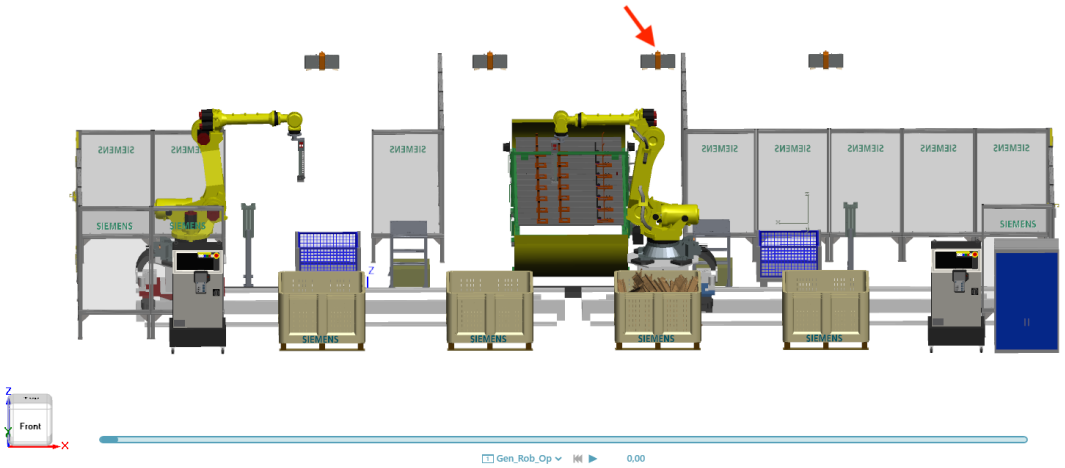
5 Validation and Results

The system was validated using a **robotic bin-picking scenario** within Process Simulate to test performance and the functional correctness of the algorithm pipeline.

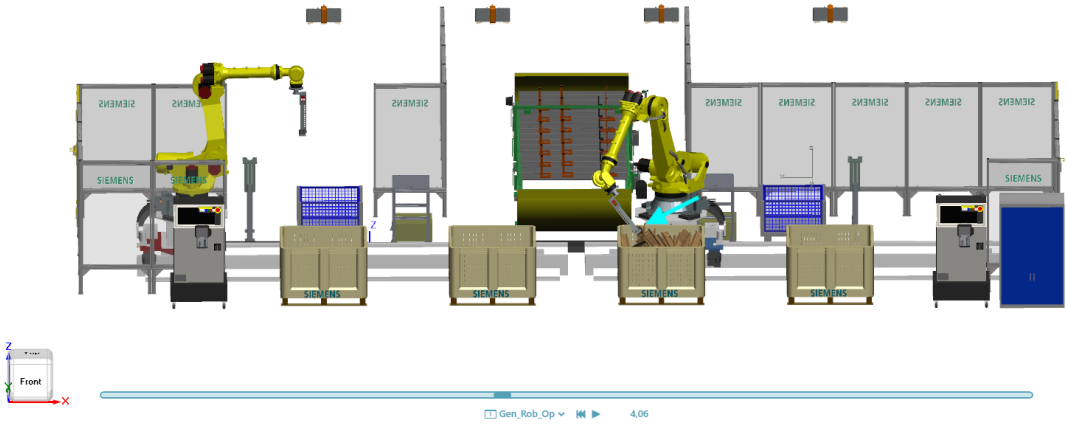
5.1 Test Scenario and Data Acquisition

The validation environment is a simulated bin-picking cell, which includes two robotic arms on independent rails, a set of part bins, and four downward-facing RGB-D cameras. The test case involves an operation where the rightmost robot picks an object from a full bin and places it on a conveyor belt.

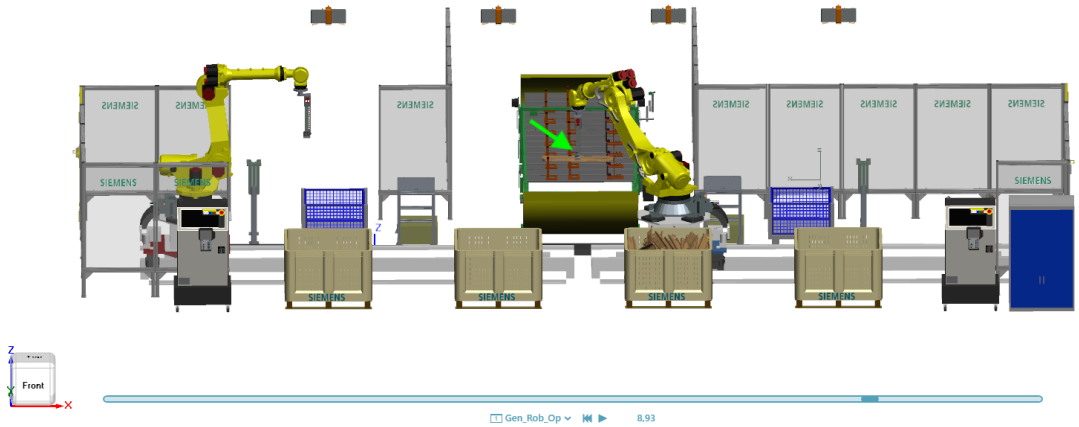
Figure 6 illustrates the key phases of this operation. The sequence begins with the initial state (a), where a red arrow indicates the active camera providing data to the perception system. The robot then descends to pick the target object from the bin, an action highlighted by the blue arrow in (b). The robot then places the object onto the conveyor belt, shown by the green arrow in (c), and finally returns to its initial position to complete the cycle.



(a) Initial state before the operation begins.



(b) The robot arm descends into the bin to pick the target object.



(c) The robot places the object on the conveyor belt.

Figure 6: Sequence of the robotic bin-picking operation in Process Simulate, used for validation.

The complete trajectory planned for this operation is shown in Figure 7.

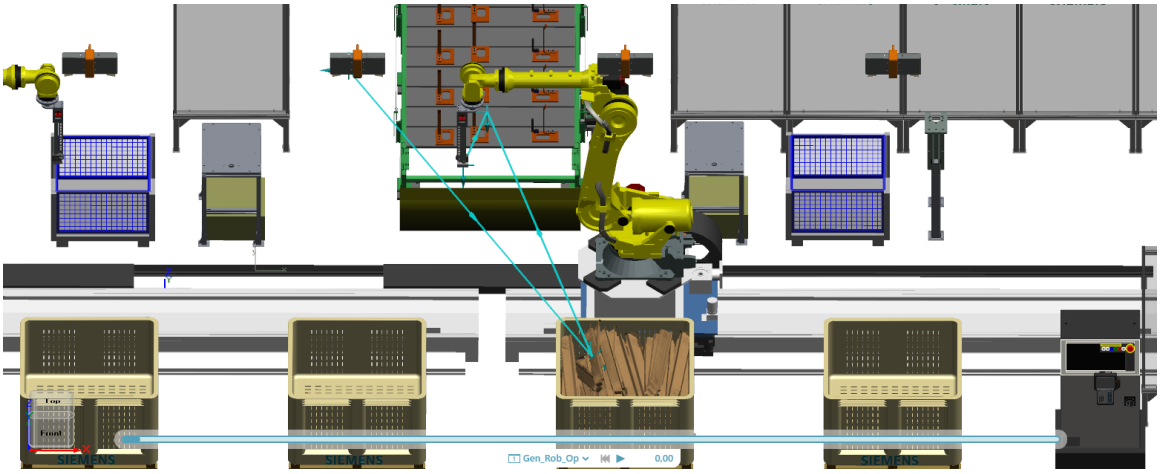
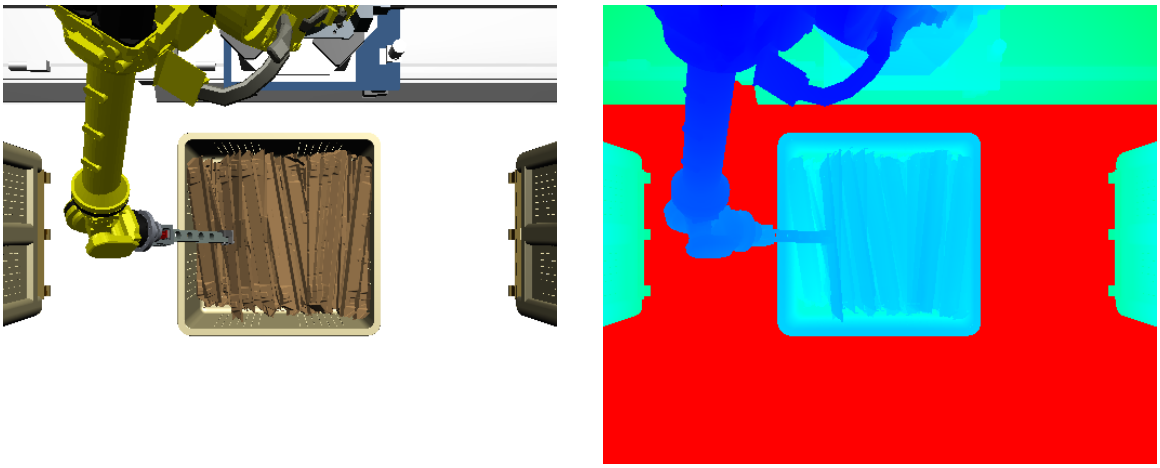


Figure 7: The robot's planned trajectory (blue) for the pick-and-place operation in Process Simulate.

As shown in Figure 8, the camera captures both standard color (RGB) images and depth maps. The depth map is the critical input for the perception algorithm, as it provides the distance information required to reconstruct the 3D environment.



(a) RGB output at initial state.

(b) Depth map output at initial state

Figure 8: Sample RGB and Depth data captured from a single virtual camera in Process Simulate.

To validate the system's reactive capabilities, a dynamic obstacle was introduced into the simulation. A simple box (highlighted by the red arrow in Figure 9), moved by an **Object Flow operation**, was used to simulate an interfering agent, such as an AGV or an inspection drone. The trajectory of this box was intentionally designed to intersect with the robot's path during its pick-and-place cycle. This setup creates a predictable collision event, allowing for a clear assessment of the perception system's ability to detect the obstacle and halt the robot's motion to prevent impact.

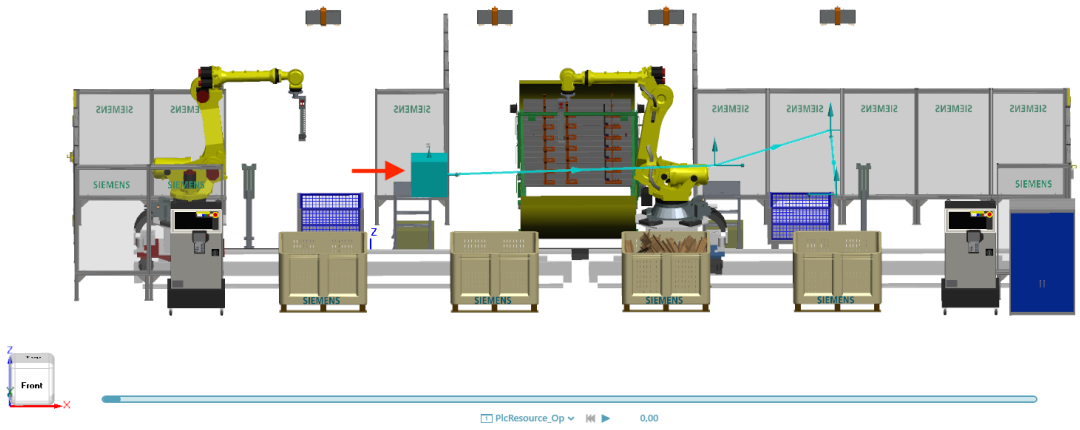


Figure 9: The dynamic obstacle (box) and its planned trajectory (blue) in Process Simulate.

5.2 Performance-Driven Camera Configuration

A series of tests were run to measure the snapshot duration in milliseconds for various camera resolutions, designed to reflect common configurations of RGB-D depth cameras available on the market, such as the **Microsoft Kinect v2** (512x424 resolution, 70° FOV) [21], **Intel RealSense D435** (up to 1280x720 resolution, 87° horizontal FOV) [22], **Azure Kinect DK** (640x576 narrow mode or 1024x1024 wide mode, variable FOV from 75° to 120°) [21], and **Orbbec Femto Bolt** (640x576 narrow mode or 1024x1024 wide mode, up to 120° horizontal FOV) [23]. The configurations tested included resolutions ranging from 512x424 to 1920x1920 and FOVs from 35° to 91°, covering standard, high-definition, and specialized industrial cameras. The results, depicted in Figure 10, show a clear trade-off between resolution and performance.

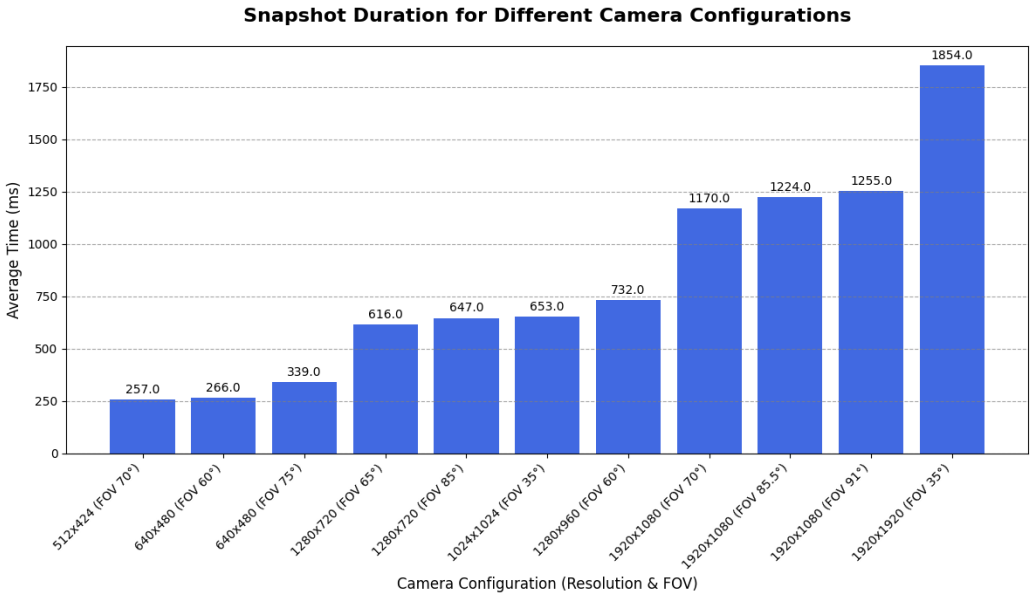
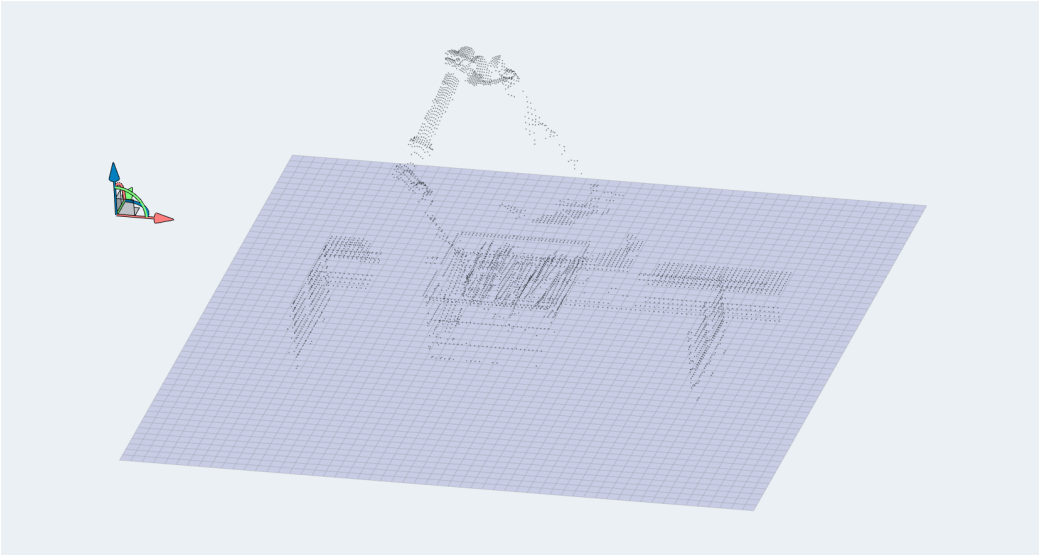


Figure 10: Performance benchmark of snapshot duration for various camera configurations.

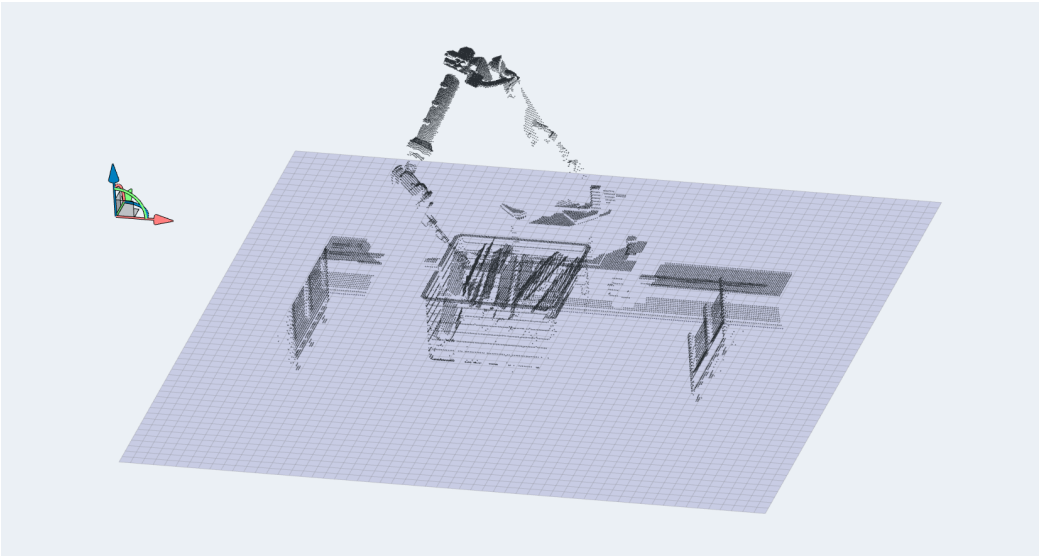
Based on this analysis, a **resolution** 512×424 with a 70° **FOV** was selected. This configuration provides a good balance between performance and detail, generating a point cloud of approximately 3,700 points from an initial depth buffer of 217,088 values, which is sufficient for accurate collision detection. Snapshot times averaged 250 ms, enabling 5 captures per second in simulation—adequate for reactive applications.

5.3 Point Cloud Generation

The first step is the generation of a raw point cloud from the camera's depth data, following the conversion process detailed in Section 4.1. Figure 11 shows the resulting point cloud before any filtering, visualized from a side view in the KineoWorks Interact Demonstrator.



(a) Lower density point cloud (sampling rate: 5).



(b) Higher density point cloud (sampling rate: 2).

Figure 11: Point clouds at different levels of detail, visualized in the KineoWorks Interact Demonstrator.

The two subfigures illustrate the output using sampling rates of 5 (a) and 2 (b) respectively. The clouds contain points corresponding to the robot's arm (visible entering the bin), the bin itself, and the parts within it. This demonstrates the successful conversion of depth data into a 3D world representation.

5.4 Robot Geometries Filtering

To illustrate the effectiveness of the **robot geometries filtering algorithm** (as detailed in the Section 4.2.1), we present visualizations from the bin-picking test scenario. Figure 12 shows the robot's bounding boxes overlaid on the scene, highlighting the volumetric regions used for point exclusion. This is followed by the resulting filtered point clouds at different levels of detail (LOD), demonstrating how the algorithm removes points corresponding to the robot's structure while preserving environmental data.

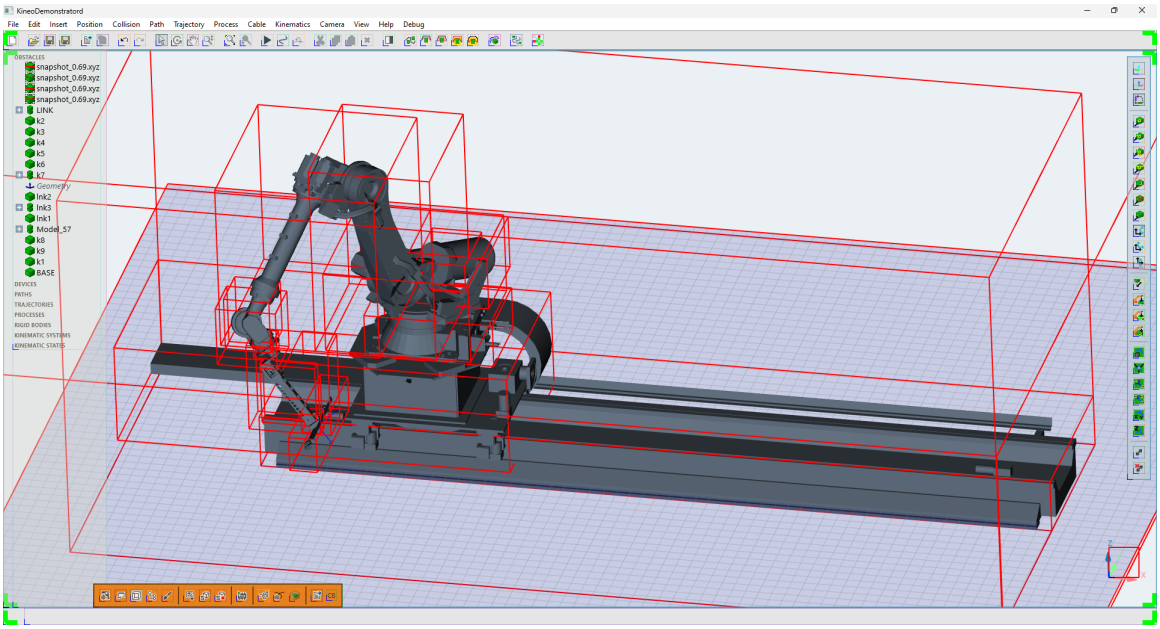
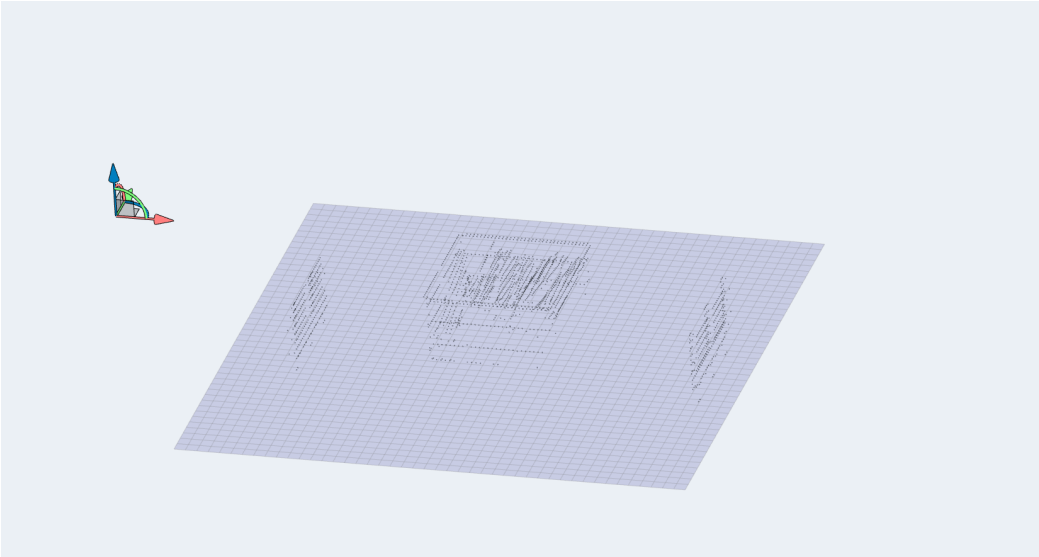


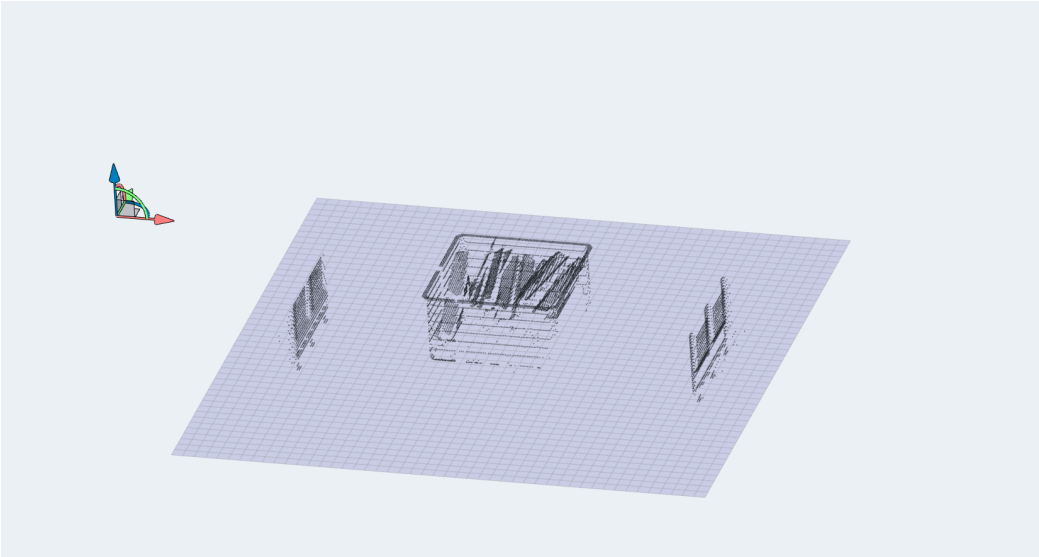
Figure 12: Device bounding box visualization.

Note: The image depicts the artifact bounds and position in the view and not the actual oriented bounding boxes used in the filtering process.

Figure 13 shows the resulting point cloud after the robot geometry filtering stage, visualized from a side view in the KineoWorks Interact Demonstrator. The two subfigures illustrate the output using sampling rates of 5 (a) and 2 (b) respectively. As intended, the points corresponding to the robot arm have been successfully removed, leaving only the points from the static environment, which includes the bin and the parts within it.



(a) Filtered lower density point cloud (sampling rate: 5)



(b) Filtered lower density point cloud (sampling rate: 2)

Figure 13: Filtered point clouds at different levels of detail, visualized in the KineoWorks Interact Demonstrator

5.5 Dynamic Obstacle Isolation

The final and most critical step in the perception pipeline is the isolation of dynamic obstacles. This is achieved by applying the static environment filtering algorithm detailed in Section 4.2.2. The effectiveness of this stage was validated under two conditions within the test scenario.

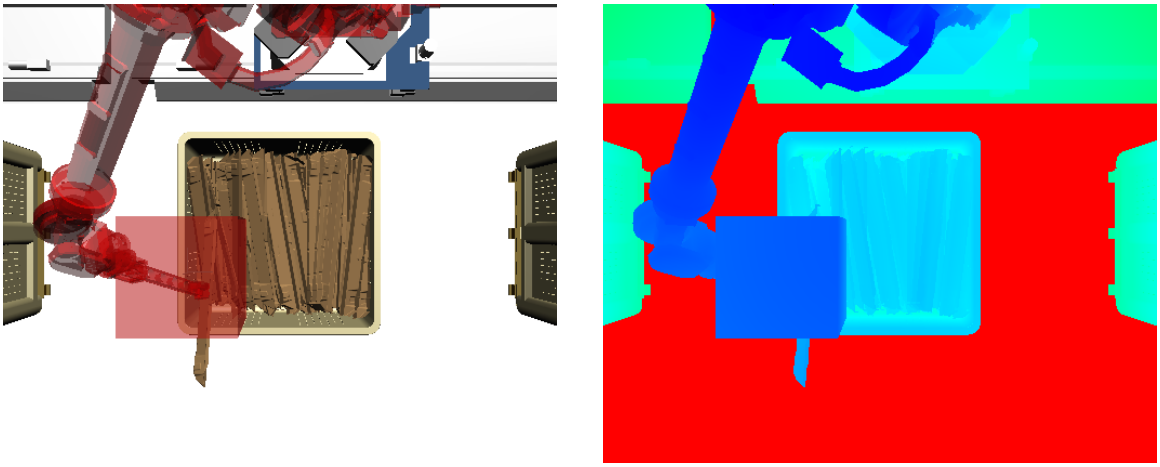
First, with no dynamic obstacle present, the system correctly identified that the point cloud (already filtered for the robot's geometry) matched the initial static baseline. As a result, all points corresponding to the bin and its contents were successfully removed. The final point cloud passed to the collision checker was empty, which is the desired outcome. This confirms that the system will not generate false-positive collisions with known, static parts of its workcell, allowing the robot to perform its programmed task without interruption.

Second, when the dynamic obstacle (the moving box) was introduced, the filtering process yielded a different result. As the box entered the camera's field of view, the points representing it were retained after the static baseline subtraction, as they had no corresponding points in the initial snapshot. The resulting output was a sparse but clean point cloud representing only the geometry of the unexpected, moving object. This isolated point cloud is the crucial input for the collision checker, enabling the system to react specifically and exclusively to dynamic changes in its environment.

5.6 Collision Detection Validation

Following the filtering stages, the system performs a **collision check** between the robot's geometry and this dynamic point cloud, as described in Section 4.3. As the robot's arm and the box converged, the `SimulationReactiveCollisionChecker` detected an imminent collision, triggering the stop condition in the `DirectSimulator` and halting the robot's motion.

Figure 14 provides a view from the camera at the exact moment the collision was detected. The RGB image (a) shows both the robot and the box highlighted in red and rendered transparently, which is Process Simulate's visual indicator for a collision. The corresponding depth map (b) captures the scene's geometry at that instant. This result provides definitive validation that the entire perception and reaction pipeline is functioning correctly.



(a) RGB output at the moment of collision.

(b) Depth map output at the moment of collision.

Figure 14: Sample RGB and Depth data captured from a single virtual camera in Process Simulate.

6 Discussion and Future Work

This project successfully laid the groundwork for reactive robotics in Process Simulate. However, several avenues for optimization and extension exist.

Performance Optimization of Filtering Algorithms: The current filtering algorithms, while functionally correct, could be optimized for performance in scenarios with very dense point clouds or complex robot geometries.

- **Robot Geometry Filtering:** The current implementation checks each point against the bounding box of every robot link, resulting in a complexity of roughly $O(N_{points} \times M_{links})$. This could be significantly improved by using a spatial partitioning data structure, such as a 3D grid or an Octree, to store the robot's geometry. This would reduce the number of point-in-volume tests to only those in the vicinity of the robot.
- **Dynamic Environment Filtering:** The static baseline filtering compares each new point to every point in the baseline, an $O(|PC_{current}| \times |PC_{baseline}|)$ operation. This can be optimized by structuring the baseline points in a k-d tree or a voxel hash grid. A k-d tree would reduce the search for the nearest neighbor to $O(|PC_{current}| \log |PC_{baseline}|)$, while a voxel grid could achieve near-amortized $O(|PC_{current}|)$ performance.

Path Replanning and Full Autonomy: The original scope of the project included not only detecting obstacles but also automatically replanning the robot's trajectory to avoid them. While the current implementation successfully achieves the "sense and react" phase by halting the robot, the "plan and act" phase remains a critical next step for future work. The existing architecture was designed specifically to support this extension. Figure 15 illustrates the envisioned reactive control loop as a state machine.

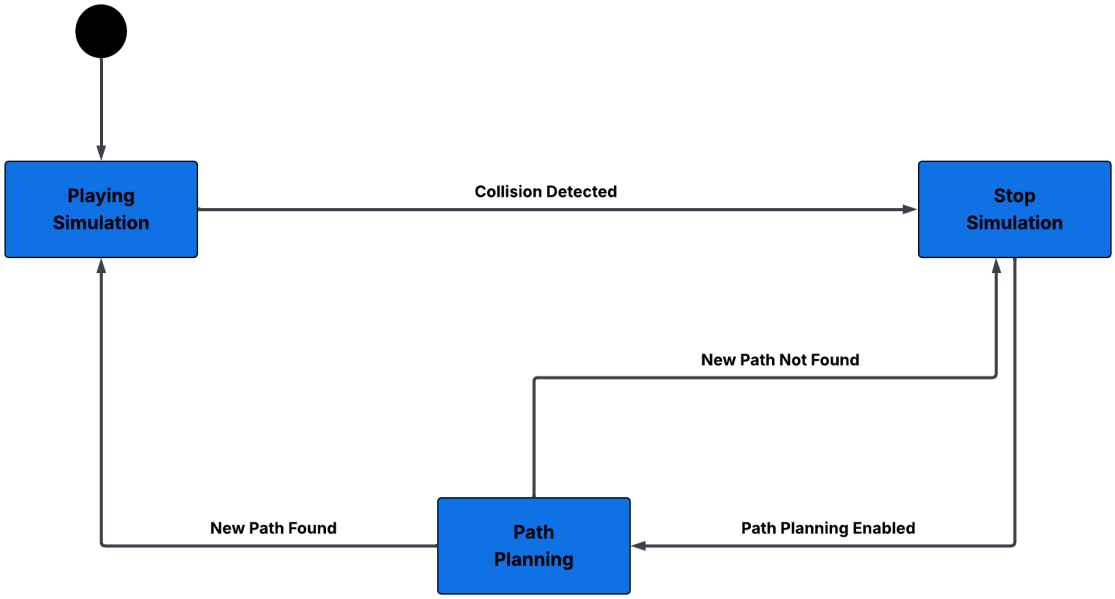


Figure 15: State machine for the complete reactive control loop.

The envisioned workflow would be as follows:

1. Upon detecting an imminent collision, the `DirectSimulator` would pause the robot's motion instead of stopping completely.
2. The isolated dynamic point cloud, which represents the obstacle, would be passed to the **KineoWorks** path planner as a temporary, "forbidden" geometry.
3. A new path planning query would be initiated, seeking a collision-free trajectory from the robot's current configuration to its original destination, now taking the new obstacle into account.
4. If a valid path is found, the `DirectSimulator` would be updated with this new trajectory and resume the simulation, allowing the robot to fluidly navigate around the obstacle.

Implementing this functionality would complete the reactive control loop, transforming the system from a safety-stop mechanism into a truly autonomous and intelligent robotic solution capable of adapting to unstructured environments in real time.

7 Conclusion

This internship project successfully demonstrated the integration of a reactive perception system for robotic arms within Siemens Tecnomatix Process Simulate. By architecting a solution that spans the C# UI, a C++/CLI bridge, and a high-performance C++ core, the project achieved its primary objective: enabling a simulated robot to perceive its environment in real-time using virtual RGB-D cameras and react to unexpected obstacles.

Key achievements include the development of a custom simulation loop for fine-grained temporal control, an event-driven pipeline for data capture, and a robust two-stage filtering algorithm. This algorithm effectively isolates dynamic obstacles by first removing the robot's own geometry and subsequently subtracting a pre-established static environmental baseline. The system's capability to detect and react to unforeseen objects was successfully validated in a bin-picking scenario, where the robot correctly halted its operation to avoid a collision.

While the work establishes a solid foundation for advanced autonomous robotics within the Siemens digital manufacturing ecosystem, it also highlights a significant constraint of the simulation environment: the performance of the underlying virtual camera snapshot operation in Process Simulate. The average snapshot time of 250 ms, while adequate for this proof-of-concept, is a limitation imposed by the simulation's rendering engine. It is important to note that this bottleneck would not be present with physical hardware, as modern real-world RGB-D cameras operate at much higher frame rates.

Nevertheless, the project proves the feasibility of the proposed architecture and provides the essential "sense and perceive" components required for future work in automatic path replanning, paving the way for more flexible and intelligent industrial automation.

List of Acronyms

AGV Automated Guided Vehicle

AI Artificial Intelligence

APP Automatic Path Planning

ARK Advanced Robotics Kinematics

CAD Computer-Aided Design

CAE Computer-Aided Engineering

CAM Computer-Aided Manufacturing

CLI Common Language Infrastructure

CNRS Centre national de la recherche scientifique (French National Centre for Scientific Research)

FOV Field of View

IoT Internet of Things

KCD Kineo Collision Detector

LAAS Laboratoire d'analyse et d'architecture des systèmes (Laboratory for Analysis and Architecture of Systems)

LOD Level of Detail

OBB Oriented Bounding Box

PLM Product Lifecycle Management

RGB-D Red-Green-Blue-Depth

SDK Software Development Kit

UI User Interface

WPF Windows Presentation Foundation

References

[1] IBM. What is Industry 4.0? https://www.ibm.com/think/topics/industry-4-0.

[2] McKinsey & Company. What are Industry 4.0, the Fourth Industrial Revolution, and 4IR? https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-are-industry-4-0-the-fourth-industrial-revolution-and-4ir#, 2022.

[3] SAP SE. What is Industry 4.0? https://www.sap.com/products/scm/industry-4-0/what-is-industry-4-0.html.

[4] Patrick Ruane, Patrick Walsh, and John Cosgrove. Using simulation optimization to improve the performance of an automated manufacturing line. *Procedia Computer Science*, 217:630–639, 2023. Accessed: 2025-07-31.

[5] FlexSim Software Products, Inc. Manufacturing Simulation. https://www.flexsim.com/manufacturing-simulation.

[6] Visual Components. Manufacturing simulation: how it works and why you should do it? https://www.visualcomponents.com/what-is-manufacturing-simulation.

[7] Sean Camarella, Michael P. Conway, Kevin Goering, and Mark Huntington. Digital twins: The next frontier of factory optimization. https://www.mckinsey.com/capabilities/operations/our-insights/digital-twins-the-next-frontier-of-factory-optimization, 2024.

[8] Mohsen Attaran and Bilge Gokhan Celik. Digital twin: Benefits, use cases, challenges, and opportunities. *Decision Analytics Journal*, 6:100165, 2023. Accessed: 2025-07-31.

[9] Matterport LLC. Industry 4.0: Guide to Smart Manufacturing with Digital Twin Technology. https://matterport.com/it/learn/digital-twin/manufacturing?srsltid=AfmBOorqBU2r1gVnj6xvwrJRYfuSFcR6i2GCi9XHi1rqAV4LheywZnD_.

[10] SAP SE. What is product lifecycle management (PLM)? https://www.sap.com/products/scm/plm-r-d-engineering/what-is-product-lifecycle-management.html.

[11] Oracle Corporation. What is PLM (Product Lifecycle Management)? https://www.oracle.com/scm/product-lifecycle-management/what-is-plm/.

[12] Heather Krebsbach. What is PLM (Product life cycle management)? https://www.atlassian.com/agile/product-management/plm.

[13] Oliver Munro. Automation in Manufacturing: Uses, Examples, & Trends. https://www.unleashedsoftware.com/blog/automation-in-manufacturing, 2023.

[14] Apan Dastider, Hao Fang, and Mingjie Lin. Retro: Reactive trajectory optimization for real-time robot motion planning in dynamic environments. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024. Accessed: 2025-07-31.

[15] Alexander Dahlin and Yiannis Karayiannidis. Trajectory scaling for reactive motion planning. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 5242–5248, 2022. Accessed: 2025-07-31.

[16] Siemens Digital Industries Software. Tecnomatix Process Simulate. https://plm.sw.siemens.com/en-US/tecnomatix/process-simulate-software. Accessed: 2025-07-31.

[17] Siemens PLM Software. Polarion. https://polarion.plm.automation.siemens.com/products/overview. Accessed: 2025-07-31.

[18] Siemens Digital Industries Software. *Kineo SDK Documentation*, 2025. Internal documentation, not publicly accessible. Accessed August 2025.

[19] Kineo Wiki. *Process Simulate general architecture*, 2025. Internal documentation, not publicly accessible. Accessed August 2025.

[20] Tully Foote and Mike Purvis. Standard Units of Measure and Coordinate Conventions. https://www.ros.org/reps/rep-0103.html, 2010. Accessed: 2025-08-08.

[21] Gregorij Kurillo, Evan Hemingway, Mu-Lin Cheng, and Louis Cheng. Evaluating the accuracy of the azure kinect and kinect v2. *Sensors*, 22(7):2469, 2022.

[22] Intel Corporation. Intel® realsense™ depth camera d435 - product specifications. https://www.intel.com/content/www/us/en/products/sku/128255/intel-realsense-depth-camera-d435/specifications.html, 2018. Accessed: 2025-08-13.

[23] Orbbec. Femto bolt hardware specifications. https://www.orbbec.com/documentation/femto-bolt-hardware-specifications. Accessed: 2025-08-13.