ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Department of Electrical, Electronic and Information Engineering
"Guglielmo Marconi" - DEI
Master Degree in Automation Engineering

Simulation of a Palletizing Robot in NVIDIA Omniverse Isaac Sim with a Computer Vision System

Master Degree Thesis Internship Report
in Autonomous and Mobile Robotics

Supervisor:
Gianluca Palli

Presented by:
Silvia Cagnolati

Co-Supervisors:
Alex Pasquali
Davide Bonvicini

Academic year - 2025/2026
Session III

Abstract

This thesis presents the design and implementation of a robotic palletizing simulation framework developed in NVIDIA Omniverse Isaac Sim. The work addresses the gap between high-level pallet layout generation and the low-level execution constraints of an industrial manipulator operating in a cluttered environment. In the proposed workflow, pallet configurations are provided by an external planning module, while the simulated robotic system is responsible for perception, grasping, optional package reorientation, collision-aware transfer, and stable placement on the pallet.

The developed environment acts as a digital twin of an industrial palletizing cell. It includes a KUKA manipulator model, a surface gripper, a simulated overhead RGB camera, JSON-driven order generation, and a physics-based execution pipeline. The perception module estimates the pose of incoming packages through classical image processing, while motion execution is governed by an RMPflow-based controller with different operating modes for free-space transfer and contact-sensitive phases. Two task-specific modules complement the controller: a package reorientation routine for packages that must be rotated before placement, and a placement strategy that combines pallet-layout post-processing with directional insertion logic to reduce lateral impacts during dense stacking.

The main contribution of the thesis is the integration of these components into a coherent manipulation pipeline that maps industrial order data into a complete perception-to-placement loop. The thesis does not claim a full experimental benchmark on real hardware; rather, it demonstrates that a structured simulation framework can be used to study the interaction between packing logic, perception uncertainty, motion generation, and contact-aware placement before deployment in a physical cell.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The automation of logistics and manufacturing systems has increased the need for robotic systems that can execute manipulation tasks with high repeatability, predictable cycle times, and limited human supervision. Among these applications, palletizing remains especially relevant because it lies at the intersection of combinatorial planning, physical interaction, and industrial throughput constraints. A robotic palletizer must not only place each package at the correct target pose, but also preserve stack stability, respect workspace and kinematic limits, and avoid destabilizing already placed packages during insertion [4, 2, 10].

These requirements make palletizing a multi-layer problem. At the high level, a packing planner determines which packages must be placed, in which order, and at which final poses on the pallet. At the execution level, however, the robotic system must translate that abstract packing solution into feasible motions, robust grasping actions, perception updates, and contact-aware placement behaviors. This thesis focuses on the execution layer. More precisely, it addresses the problem of how to realize a pallet layout produced by an external planner inside a realistic industrial simulation while accounting for perception uncertainty, robot kinematics, and contact-sensitive placement.

To address this problem, the work develops a digital twin of an industrial palletizing cell in NVIDIA Omniverse Isaac Sim. The environment reproduces the main elements of a real workstation: a KUKA manipulator mounted on a pedestal, a pallet, two working tables, an overhead RGB camera, and a surface gripper used to grasp carton-like packages. The simulator provides rigid-body physics, collision handling, and scene composition capabilities that are suitable for studying the interaction between motion

generation, contact, and pallet stability [5, 8, 9]. In this context, simulation is not used merely for visualization, but as an engineering environment in which perception, control, and manipulation strategies can be evaluated before deployment on physical hardware.

The packing orders used in the simulation are not generated online by the robotic controller. They are provided as JSON datasets derived from an external optimization or reinforcement-learning pipeline. Each order specifies the package dimensions, the desired final pallet pose, and the orientation identifier required by the downstream manipulation logic. The role of the robotic system developed in this thesis is therefore to execute those orders safely and consistently, not to solve the packing problem itself.

Within this scope, the thesis makes four main contributions:

- it develops an execution-oriented simulation framework that connects industrial order data, scene generation, sensing, and robot control inside Isaac Sim;
- it implements a perception pipeline based on an overhead RGB camera and lightweight image processing to estimate package position and orientation before grasping;
- it defines a state-machine-based manipulation architecture that coordinates grasping, optional package reorientation, transfer, and final placement under an RMPflow-based controller [3, 1];
- it introduces a placement strategy composed of two clearly separated modules: a pallet-layout regularization algorithm that enforces minimum spacing between packages, and a directional insertion module, that selects the safest approach direction during final placement.

The thesis was carried out in collaboration with Proteo Engineering, which provided the industrial context of the project and realistic order datasets. This industrial grounding is important because it keeps the work focused on a practical objective: not merely simulating a generic pick-and-place task, but reproducing the operational logic of a palletizing cell with sufficient fidelity to support future transfer to a physical setup.

Throughout the thesis, the manipulated objects are referred to as *packages*. The term *box* is retained only where it appears in code identifiers, file names, or figure captions closely tied to the implementation.

From a systems perspective, the complete workflow can be summarized as the following sequence:

1. **Dataset.** Dataset-driven order generation provides the target package poses and orientation identifiers.

2. **Perception.** The perception module estimates the pose of each incoming package on the picking table.

3. **Pick.** The robot executes the grasping action based on the estimated package pose.

4. **Optional Reorientation.** A reorientation stage is activated when the required pallet pose differs from the stable incoming pose.

5. **Placement Strategy.** The target layout is regularized and a collision-aware insertion direction is selected.

6. **Robot Execution.** The robot executes transfer and deposit motions under the state-machine and RMPflow control logic.

The remainder of the thesis develops the proposed framework progressively, moving from the general problem setting to the individual system components and finally to the overall experimental discussion. Chapter 2 introduces the palletizing problem, motivates the use of simulation, and presents the structure of the Isaac Sim environment adopted in this work. Chapter 3 then focuses on the modeling and integration of the surface gripper, while Chapter 4 presents the motion-generation framework and its relation to operational-space and reactive control principles. Chapter 5 is devoted to package reorientation, and Chapter 6 discusses the placement strategy by separating the spacing algorithm from the directional insertion logic. After that, Chapter 7 describes the perception pipeline, and Chapter 8 presents the state-machine architecture that coordinates the complete manipulation cycle. The results obtained with the developed system are reported in Chapter 9, whereas Chapter 10 concludes the thesis by summarizing the main limitations of the current work and outlining possible future developments.

Chapter 2

Palletizing and Simulation Environment

2.1 Robotic Palletizing and 3D Bin Packing Problem

Palletizing is a fundamental operation in modern logistics and industrial automation. The process consists of arranging packages or containers on a pallet according to a predefined spatial configuration in order to facilitate transportation, storage, and distribution. Automated palletizing systems are widely deployed in manufacturing plants, warehouses, and distribution centers, where they enable high-throughput handling of products while reducing human labor and improving operational safety.

In a typical industrial palletizing cell, a robotic manipulator receives packages from an incoming conveyor or staging area, grasps them using an appropriate end-effector, and places them onto a pallet according to a specific stacking pattern. The pallet configuration must satisfy several practical requirements, including mechanical stability, efficient use of pallet volume, and compliance with weight distribution constraints.

The design of pallet layouts is closely related to the well-known *Three-Dimensional Bin Packing Problem (3D-BPP)*. The 3D-BPP is a classical combinatorial optimization problem that consists of placing a set of three-dimensional objects inside a container of finite volume in such a way that the available space is utilized as efficiently as possible. In its general formulation, the problem involves determining the position and orientation of each object within the container while avoiding overlaps and respecting geometric

constraints.

Formally, given a container of dimensions (L, W, H) and a set of n rectangular items with dimensions (l_i, w_i, h_i), the goal is to find a placement configuration such that:

- All items lie completely inside the container boundaries;
- No two items overlap in space;
- The overall packing objective is optimized.

Typical optimization criteria include maximizing the number of items packed, maximizing volume utilization, minimizing the height of the final stack, or minimizing unused space.

The 3D bin packing problem is known to be NP-hard, meaning that no polynomial-time algorithm exists to solve all instances optimally. As a consequence, practical palletizing systems rely on heuristic methods, approximation algorithms, or learning-based approaches capable of generating near-optimal packing configurations within acceptable computational time.

In industrial palletizing scenarios, the packing problem is further complicated by additional physical constraints that go beyond pure geometry. For example, packages must often satisfy:

- Stability constraints to prevent stacked items from collapsing,
- Weight distribution constraints to avoid excessive load concentration,
- Orientation constraints imposed by product labeling or fragility,
- Accessibility constraints for robotic manipulation.

For these reasons, modern palletizing systems must combine geometric packing strategies with robotic manipulation capabilities that can physically realize the computed layout.

In recent years, machine learning approaches have emerged as promising solutions for tackling complex packing problems. In particular, Reinforcement Learning (RL) techniques have been explored as a way to automatically learn packing strategies through interaction with a simulated environment. In such approaches, an agent iteratively learns how to place items in a container by maximizing a reward function that captures packing efficiency and stability.

Within the context of this project, the packing configuration is generated by an external Reinforcement Learning algorithm that determines the optimal order composition and the placement arrangement of packages on the pallet. The robotic system developed in this work is responsible for executing these packing plans within a realistic simulation environment.

The manipulation pipeline must therefore translate high-level packing instructions into a sequence of physical actions performed by a robotic manipulator. This involves detecting incoming packages, grasping them reliably, potentially reorienting them to match the required orientation, and finally placing them in the target position on the pallet.

A critical challenge in robotic palletizing is ensuring that the manipulation process does not compromise the stability of previously placed packages. Even small impacts or misalignments during placement can propagate through the stacked structure and cause objects to shift. Consequently, robotic control strategies must carefully regulate motion speed, contact forces, and approach trajectories to maintain the integrity of the stack.

To study and validate such manipulation strategies without risking damage to physical equipment, simulation environments play a crucial role. High-fidelity robotic simulators allow researchers and engineers to prototype robotic behaviors, test perception algorithms, and evaluate control policies under controlled yet realistic physical conditions.

For this reason, the present project relies on a physics-based simulation environment capable of accurately reproducing the dynamics of an industrial palletizing cell.

2.2 Robotics Simulation for Industrial Manipulation

Simulation has become an essential tool in modern robotics research and development. Before deploying algorithms on real robotic hardware, it is often necessary to evaluate their behavior in a virtual environment that reproduces the physical characteristics of the real system.

Robotic simulation environments provide several advantages:

- Safe testing of control strategies without risk of damaging hardware;
- Rapid prototyping of robotic applications;

- Reproducibility of experiments under controlled conditions;
- Possibility to generate large amounts of synthetic data for training perception systems.

In the context of robotic manipulation, simulators must provide accurate modeling of rigid body dynamics, contact forces, frictional interactions, and sensor behavior. These elements are essential for ensuring that the simulated robot behaves in a manner consistent with real-world physics.

Furthermore, simulation platforms enable the creation of *digital twins*, which are virtual replicas of real industrial systems. A digital twin reproduces the kinematic structure, physical parameters, and operational constraints of a real robotic cell. By mirroring the behavior of the physical system, it allows developers to test control strategies and manipulation pipelines before transferring them to real hardware.

In palletizing applications, digital twins are particularly valuable because they allow the evaluation of packing strategies and robot trajectories without interrupting production lines or risking damage to products.

2.3 NVIDIA Omniverse Isaac Sim

The simulation environment used in this project is NVIDIA Omniverse Isaac Sim, a high-fidelity robotics simulation platform designed for developing, testing, and validating robotic systems in physically accurate virtual environments. Official technical details of Isaac Sim are provided in the NVIDIA documentation [5].

Isaac Sim is built on top of the NVIDIA Omniverse platform, a real-time 3D collaboration and simulation ecosystem that leverages Pixar's Universal Scene Description (USD) framework. This architecture allows complex scenes to be composed using a modular hierarchical structure in which each element of the environment is represented as a node within a scene graph. The Omniverse-USD integration is documented in the NVIDIA and OpenUSD references [5, 8].

The platform provides a comprehensive set of tools for robotics development, including:

- PhysX-based rigid body dynamics for realistic physics simulation;

- GPU-accelerated computation for high-performance real-time simulation;
- Synthetic data generation tools for computer vision and AI training;
- Integration with ROS and ROS2 middleware;
- Support for a wide range of simulated sensors such as RGB cameras, depth cameras, and LiDAR.

One of the main advantages of Isaac Sim is its ability to integrate perception, motion planning, control, and physics simulation within a single unified framework. This makes it possible to develop complete robotic pipelines that closely resemble real industrial deployments.

The simulator relies on the NVIDIA PhysX physics engine to compute rigid body dynamics and contact interactions. PhysX models physical phenomena such as gravity, collision detection, friction, restitution, and joint constraints, enabling accurate simulation of robotic manipulation tasks involving object contact and stacking.

Additionally, Isaac Sim leverages GPU acceleration to simulate complex scenes containing many interacting objects while maintaining real-time performance. This capability is particularly important in palletizing scenarios, where multiple packages interact through contact forces during stacking.

2.4 Scene Architecture Representation

The simulated palletizing cell developed in this project represents a digital twin of an industrial robotic workstation. The environment includes all the key elements typically present in a real palletizing system, including the robotic manipulator, working surfaces, package staging areas, and the destination pallet.

The main components of the simulated scene include:

- A robotic manipulator mounted on a pedestal,
- A package arrival area where boxes are spawned,
- A reorientation table used to rotate packages when required,
- A destination pallet where boxes are stacked,

- Dynamically generated packages based on dataset specifications.

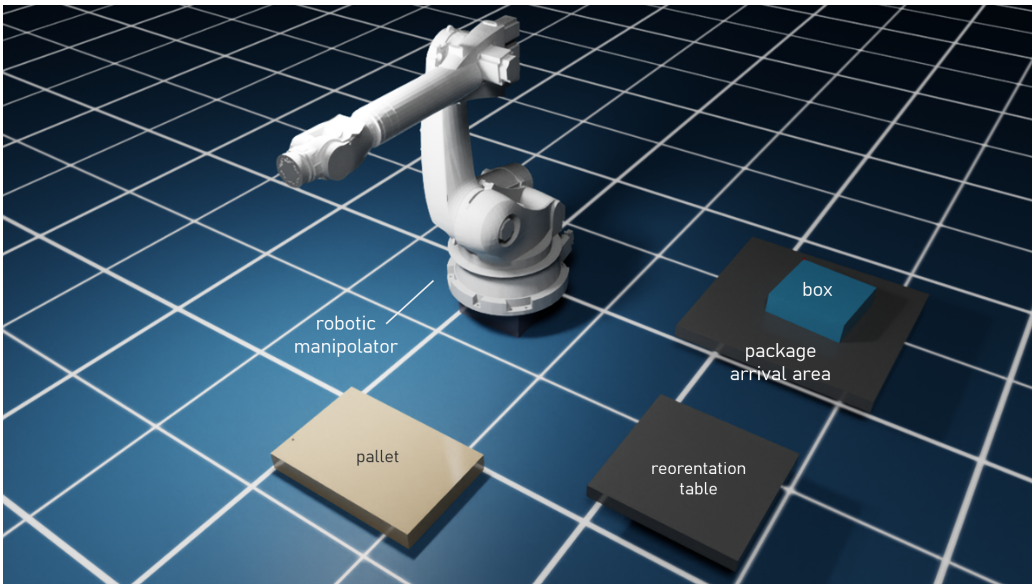


Figure 2.1: Scene architecture inside Isaac Sim.

All objects within the scene are represented as USD primitives with explicitly defined geometric, visual, and physical properties. The use of USD allows complex scenes to be organized hierarchically, enabling modular composition and efficient scene management. Further details on the USD scene representation model are provided in the OpenUSD documentation [8].

Each element of the environment is associated with specific physical attributes such as mass, inertia tensors, friction coefficients, and collision geometries. These parameters are essential for ensuring realistic interaction between the robot and the manipulated objects.

Static elements such as tables and support structures are configured as fixed rigid bodies that act as stable contact surfaces. Dynamic elements such as packages are instead modeled as fully simulated rigid bodies subject to gravity, friction, and collision forces.

This modeling strategy allows the simulation to reproduce the dynamic behavior of real packages during grasping, transport, and placement operations.

2.5 Dataset-Driven Generation of Packages

A central component of the simulation framework is the dynamic generation of packages based on structured data retrieved from an external dataset.

Each entry in the dataset corresponds to a specific box belonging to a palletizing order.

For each package, the dataset provides the following information:

- Geometric dimensions (l, w, h),
- Unique identifier (ID),
- Target orientation,
- Final placement coordinates on the pallet.

During the simulation, packages are instantiated dynamically based on this dataset. To better emulate realistic industrial conditions, the base pose of each box is perturbed through controlled randomization.

The spawning position is randomly displaced within a predefined interval in the horizontal plane, while the orientation may also be perturbed by small angular deviations. These perturbations simulate real-world variability in package arrival conditions and allow the perception and manipulation modules to be tested under non-ideal scenarios.

The dataset-driven spawning mechanism provides several key advantages:

- Scalability to large palletizing orders,
- Easy modification of package properties,
- Deterministic reproducibility of experiments,
- Direct integration with packing algorithms.

As a result, the simulation environment becomes not only a visualization tool but also a powerful experimental platform for evaluating robotic manipulation strategies and packing solutions generated by external optimization algorithms.

Chapter 3

Surface Gripper Model

In robotic manipulation tasks such as palletizing, the design of the end-effector plays a fundamental role in determining the reliability and efficiency of the grasping process. Industrial palletizing systems frequently rely on specialized grippers capable of handling packages of different sizes, shapes, and surface properties. Depending on the application, grippers may rely on mechanical fingers, vacuum suction, magnetic forces, or adhesive contact mechanisms.

Within the context of this project, the gripper is modeled as a *surface gripper* implementing a contact-based attachment logic. This abstraction allows the simulation to replicate grasping behavior without explicitly modeling complex finger geometries or vacuum systems. Instead of relying on mechanical closure, the surface gripper establishes a physics-based attachment when the gripper surface comes sufficiently close to an object surface.

This approach is particularly suitable for simulation environments where the primary objective is to evaluate manipulation strategies rather than to model the exact mechanical design of the gripper hardware.

3.1 Surface Gripper Concept

A surface gripper can be interpreted as a simplified grasping device that secures objects by establishing a contact-based adhesion constraint between the gripper surface and the manipulated object. Rather than enclosing the object using articulated fingers, the gripper detects nearby surfaces and generates forces that keep the object attached to the end-effector.

From a physical perspective, the grasp is maintained through two types of forces:

- **Coaxial forces**, acting along the normal direction of the contact surface, responsible for maintaining the object attached to the gripper.
- **Shear forces**, acting tangentially to the contact surface, responsible for resisting lateral sliding between the gripper and the object.

These forces approximate the behavior of real industrial gripping technologies such as vacuum suction cups or adhesive grippers, where the object is retained through surface adhesion rather than mechanical enclosure.

In simulation, the surface gripper is implemented as a logical component associated with the robot's end-effector. When activated, the component continuously checks whether a candidate object surface lies within a predefined sensing distance. If this condition is satisfied, an attachment constraint is generated between the gripper and the object, effectively creating a temporary rigid relationship governed by configurable force limits.

This modeling approach significantly simplifies the manipulation pipeline while still preserving the essential physical characteristics required for realistic pick-and-place operations.

3.2 Gripper Configuration Parameters

The behavior of the surface gripper is governed by a set of configurable parameters associated with the gripper primitive (prim) in the simulation scene. These parameters determine the sensing range of the gripper, the strength of the attachment forces, and the retry logic applied during grasp attempts.

The main configuration parameters include:

- `MAX GRIP DISTANCE`
  This parameter defines the maximum allowed distance between the gripper surface and the object surface for a grasp attempt to occur. If the object lies within this activation range, the gripper may attempt to establish a physical attachment.

  The parameter effectively defines the sensing envelope of the gripper and determines how close the robot must move before a grasp can be initiated.

- `COAXIAL FORCE LIMIT`
  This value specifies the maximum force that the gripper can apply

along the gripping axis, which corresponds to the direction normal to the contact surface. If the forces required to maintain the grasp exceed this threshold, the attachment is considered unstable and the object may detach.

This parameter therefore models the holding strength of the gripper along the main attachment direction.

- `SHEAR FORCE LIMIT`
  The shear force limit defines the maximum tangential force that the grasp can resist before sliding occurs. This parameter approximates the frictional stability of the contact interface between the gripper and the object surface.

  Higher shear limits result in more stable lateral interactions during transport.

- `RETRY INTERVAL`
  This parameter defines the time interval between consecutive grasp attempts if the initial attempt fails. The retry mechanism allows the system to automatically re-attempt attachment when the robot slightly adjusts its pose relative to the object.

These parameters allow the gripper behavior to be tuned according to the characteristics of the manipulated objects and the desired robustness of the grasp.

3.3 Integration with the Robot Model

In order to integrate the surface gripper with the robot model, a physical connection must be established between the robot end-effector and the gripper component.

Within the simulation environment, this connection is implemented through a six-degree-of-freedom (6-DOF) joint that attaches the gripper primitive to the robot tool frame. The joint defines the spatial relationship between the robot and the gripper and allows the gripper to inherit the motion of the end-effector.

In practice, the implementation creates a USD joint primitive connecting the robot tool frame and the gripper prim. The joint is configured with six

constrained axes corresponding to the three translational and three rotational degrees of freedom. Limit APIs are applied to each axis in order to restrict motion and ensure that the gripper remains rigidly attached to the robot.

The resulting structure can be interpreted as a rigid extension of the robot end-effector that carries the grasping logic.

3.4 Implementation of the Surface Gripper

The surface gripper used in this project is implemented through a dedicated Python class that interfaces with the simulation environment. The class encapsulates all operations required to create the gripper primitive, attach it to the robot, configure its parameters, and control its grasping behavior during runtime.

At initialization, the class retrieves the current simulation stage and creates a surface gripper primitive at a specified location within the robot tool hierarchy. The gripper is then connected to the robot tool frame through a dynamically created D6 joint.

The initialization process performs the following steps:

1. Acquire the simulation stage and the surface gripper interface.
2. Create a surface gripper schema object associated with the specified prim path.
3. Attach the gripper to the robot tool frame through an attachment relationship.
4. Create a six-degree-of-freedom joint between the robot tool and the gripper.

The creation of the D6 joint is an important step, as it establishes the rigid mechanical connection between the robot and the gripper component. Each axis of the joint is associated with limit attributes that effectively prevent unwanted motion, ensuring that the gripper follows the robot end-effector without additional degrees of freedom.

3.5 Gripper Control Interface

Once the gripper has been created and configured, it can be controlled through a runtime interface provided by the simulation framework.

The interface exposes a set of methods that allow the robotic controller to activate or deactivate the grasping mechanism during execution.

The most important control operations include:

- `close()`
  When this method is invoked, the surface gripper begins searching for nearby object surfaces within the configured gripping distance. If a suitable contact is detected, the gripper attempts to establish an attachment governed by the configured force limits.

- `open()`
  This method releases the currently attached object by removing the contact constraint. Once the attachment is removed, the object is again governed exclusively by the physics simulation and can move freely under gravity and contact forces.

- `is_closed()`
  Returns the current state of the gripper, indicating whether an object is currently attached.

- `is_open()`
  Indicates whether the gripper is currently inactive and not holding any object.

These methods are used by the manipulation controller to coordinate grasping and release operations during the pick-and-place pipeline.

3.6 Advantages of Surface Gripper Modeling

The adoption of a surface gripper abstraction offers several advantages in the context of robotic manipulation simulation.

First, the approach significantly reduces the geometric complexity of the gripper model. Explicitly simulating articulated fingers would require accurate collision meshes, contact modeling between fingers and objects, and detailed trajectory planning for finger closure.

Second, the surface gripper abstraction provides a robust and computationally efficient method for simulating grasping behavior in pick-and-place tasks where objects are typically manipulated through planar contact surfaces.

Third, the force-limited attachment mechanism allows the simulation to capture realistic grasp failure conditions. If the forces applied during manipulation exceed the configured limits, the object may detach from the gripper, replicating real-world gripping limitations.

Finally, the simplified grasping model allows the research effort to focus on higher-level aspects of the manipulation pipeline, such as motion planning, perception integration, and palletizing strategies.

For these reasons, the surface gripper provides an effective compromise between physical realism and implementation simplicity within the simulated palletizing environment.

3.7 Comparison with Alternative Gripper Models

Several different gripper technologies are commonly used in industrial robotic manipulation tasks. The choice of gripper model in a simulation environment depends on the desired balance between physical realism, computational complexity, and implementation effort. For palletizing applications in particular, multiple grasping technologies can be considered, each with different modeling implications.

3.7.1 Mechanical Finger Grippers

Mechanical grippers with articulated fingers are among the most widely used robotic end-effectors in industrial manipulation tasks. These devices grasp objects through physical enclosure using two or more fingers that close around the object. The stability of the grasp is determined by geometric constraints and frictional contact forces between the fingers and the object.

Simulating such grippers with high fidelity requires detailed geometric modeling of the finger structure and accurate collision detection between the finger surfaces and the manipulated object. Additionally, the motion planning system must explicitly control the opening and closing trajectories of the fingers while ensuring collision-free interactions.

Although this approach offers a high degree of physical realism, it also significantly increases the complexity of the simulation. The manipulation controller must manage additional degrees of freedom associated with finger articulation, and the physics engine must resolve multiple simultaneous contact points.

For these reasons, mechanical gripper models are typically used in simulation when the detailed study of grasp mechanics is required.

3.7.2 Vacuum Grippers

Vacuum suction grippers are extremely common in palletizing and packaging automation systems. These devices rely on negative pressure generated by suction cups to create an attachment between the gripper surface and the object.

From a simulation perspective, accurately modeling vacuum suction requires representing the pressure differential and the resulting normal forces acting on the contact surface. This can involve fluid modeling or simplified pressure-force approximations depending on the level of physical realism required.

Vacuum grippers are particularly effective when manipulating objects with flat and rigid surfaces, such as cardboard boxes or plastic containers. However, their performance can degrade when objects have irregular surfaces, porous materials, or insufficient contact area.

While vacuum grippers are realistic for palletizing scenarios, implementing them in simulation can require additional modeling complexity that is not always necessary when the objective is primarily to evaluate robotic motion and placement strategies.

3.7.3 Magnetic Grippers

Magnetic grippers represent another category of manipulation devices that rely on magnetic attraction to hold objects. These grippers are commonly used in applications involving ferromagnetic materials such as steel plates or metallic components.

In simulation environments, magnetic grasping can be modeled by introducing attractive forces between the gripper and the object based on distance and magnetic field strength. However, this approach is generally

limited to applications where the manipulated objects possess appropriate magnetic properties.

Since the palletizing scenario considered in this project involves cardboard boxes and packaged goods, magnetic grippers are not applicable to the manipulation task.

3.7.4 Surface Grippers

The surface gripper model adopted in this project can be considered an abstraction that captures the essential behavior of adhesion-based grasping systems without requiring detailed modeling of the physical gripping mechanism.

Instead of explicitly simulating suction cups or finger closure, the surface gripper establishes a force-limited attachment when a suitable contact surface is detected within a predefined activation distance. This mechanism approximates the effect of adhesive or suction-based gripping while maintaining a simple implementation.

The main advantages of this approach include:

- Reduced geometric complexity, since no articulated finger mechanisms need to be modeled.
- Simplified motion planning, as the controller does not need to manage additional finger joints.
- Robust grasping behavior in pick-and-place scenarios involving planar contact surfaces.
- Efficient computation within the physics simulation environment.

While the abstraction sacrifices some degree of physical realism, it provides a practical compromise that is well suited for the objectives of this project.

3.7.5 Rationale for the Selected Approach

Considering the requirements of the palletizing simulation, the surface gripper model was selected as the most appropriate compromise between realism and implementation complexity.

The primary goal of the project is to evaluate the manipulation pipeline and the interaction between motion planning, grasping logic, and object placement within the palletizing workflow. Detailed modeling of the internal mechanics of the gripper is therefore not strictly necessary.

By adopting the surface gripper abstraction, the simulation focuses on the key aspects that influence palletizing performance, including:

- object detection and alignment,
- robot trajectory generation,
- collision avoidance,
- stable object placement on the pallet.

This approach allows the simulation to remain computationally efficient while still capturing the essential physical interactions required for realistic manipulation behavior.

Consequently, the surface gripper provides an effective and scalable solution for implementing the pick-and-place pipeline within the simulated palletizing environment.

Chapter 4

Motion Planning and Control Logic

Robotic manipulation tasks such as palletizing require sophisticated motion planning and control strategies capable of coordinating multiple objectives simultaneously. A robotic manipulator must not only move efficiently between task locations but must also ensure safety, precision, and physical stability when interacting with objects and operating near obstacles.

In industrial palletizing scenarios, the robot repeatedly performs pick-and-place operations involving objects located on different work surfaces, such as input tables, orientation stations, and pallets. These operations require the robot to generate trajectories that are both efficient and safe while ensuring accurate positioning during grasping and placement.

Within the context of this project, motion generation is implemented using the **Riemannian Motion Policies (RMP)** framework provided by the NVIDIA Isaac Sim environment. RMPflow enables the composition of multiple motion behaviors defined in different task spaces, producing a unified control command that respects the priorities of all active objectives.

This chapter describes the theoretical principles underlying the adopted control strategy, the role of task-space policies in robotic motion synthesis, and the practical extensions introduced in the project to support adaptive motion behavior across different operational regions of the workspace.

4.1 Motion Planning in Robotic Manipulation

Motion planning for robotic manipulators involves determining a sequence of robot configurations that move the end-effector from an initial pose to a

desired target pose while respecting a set of constraints.

These constraints may include:

- avoiding collisions with objects or environmental structures,
- respecting joint limits and mechanical constraints,
- maintaining stable grasp configurations,
- ensuring smooth and dynamically feasible motion.

Traditional motion planning methods often rely on geometric path planning algorithms such as Rapidly-exploring Random Trees (RRT) or Probabilistic Roadmaps (PRM). While these approaches are effective for finding collision-free paths, they typically operate at a higher planning level and do not directly incorporate the dynamic interactions between multiple control objectives.

In contrast, policy-based motion generation frameworks compute control actions directly by combining multiple motion policies defined in different spaces. This approach allows the robot to react continuously to changes in the environment while maintaining smooth motion behavior.

RMPflow belongs to this class of reactive control frameworks and is particularly suitable for manipulation tasks requiring real-time adaptation.

4.2 Operational Space Control

A key concept underlying modern robotic control architectures is the distinction between *configuration space* and *task space*.

The configuration space of a robot is defined by the set of joint variables describing the robot's internal configuration:

$$q = (q_1, q_2, ..., q_n) \tag{4.1}$$

where each component corresponds to a joint position.

In many manipulation tasks, however, the desired behavior of the robot is specified in task space rather than joint space. For example, the objective may be to move the end-effector to a specific Cartesian position or orientation.

Operational Space Control (OSC) provides a framework for generating control commands that directly regulate task-space quantities. The relationship between joint velocities and end-effector velocity is given by the robot Jacobian:

$$\dot{x} = J(q)\dot{q} \tag{4.2}$$

where:

- x represents the task-space coordinates,
- q represents the joint configuration,
- $J(q)$ is the Jacobian matrix.

By manipulating this relationship, controllers can generate joint accelerations that produce desired motions in task space while respecting the robot's kinematic structure.

The RMPflow framework builds upon these principles by allowing multiple task-space policies to be defined simultaneously and combined through a mathematically consistent procedure.

4.3 RMPflow: Conceptual Overview

RMPflow is a motion generation framework that uses differential geometry to combine multiple motion policies into a unified control law.

Rather than defining a single global controller, RMPflow allows individual control objectives to be expressed as independent policies defined on different manifolds. Each policy describes a desired motion behavior for a specific task.

Formally, an RMP is represented as a pair:

$$(\mathbf{a}_i(\mathbf{x}), G_i(\mathbf{x})) \tag{4.3}$$

where:

- $\mathbf{a}_i(\mathbf{x})$ represents the desired acceleration in task space,
- $G_i(\mathbf{x})$ is a Riemannian metric that determines the importance and directional weighting of the policy.

The metric plays a critical role in defining how strongly the policy influences the final motion command.

Each policy operates within its own task space manifold $\mathcal{T}_i$. Through the use of Jacobian transformations, these policies are mapped into the robot configuration space and combined through a structured propagation process.

The resulting global control command is obtained by solving the optimization problem:

$$\mathbf{a}^* = \arg\min_{\mathbf{a}} \sum_i \|J_i\mathbf{a} - \mathbf{a}_i\|^2_{G_i} \tag{4.4}$$

where:

- J_i is the Jacobian mapping from configuration space to the i-th task space,
- $\mathbf{a}_i$ is the desired acceleration generated by the local policy,
- G_i defines the metric weighting the policy.

The result is a globally consistent acceleration command that respects the combined objectives of all policies.

4.4 Potential Fields and Reactive Control

Many motion policies used in robotic control can be interpreted as potential-field-based behaviors. In this formulation, desired motions are generated by defining artificial potential functions whose gradients drive the robot toward desirable configurations and away from undesirable ones.

For example, an attractive potential guiding the end-effector toward a target position may be defined as:

$$U_{target} = \frac{1}{2}k_p\|p_{ee} - p_{target}\|^2 \tag{4.5}$$

The resulting acceleration is obtained from the negative gradient of the potential:

$$a_{target} = -\nabla U_{target} \tag{4.6}$$

Similarly, repulsive potentials can be used to prevent collisions with obstacles.

RMPflow generalizes this idea by embedding these potential-based behaviors into Riemannian task spaces, allowing the policies to be combined in a mathematically consistent way.

4.5 Control Objectives in the Palletizing Task

In the palletizing scenario considered in this project, multiple control objectives must be satisfied simultaneously.

The most relevant objectives include:

1. **Target Approach**

   The robot must move its end-effector toward predefined grasping or placement poses.

2. **Obstacle Avoidance**

   The manipulator must maintain safe distances from surrounding objects such as tables, pallets, and previously placed boxes.

3. **Joint Limit Enforcement**

   Joint angles must remain within mechanical limits to prevent unrealistic configurations.

4. **Smooth Trajectory Generation**

   Robot motion must remain continuous and physically plausible, avoiding abrupt changes in velocity or acceleration.

Each of these objectives is encoded as a separate RMP within the RMPflow controller.

4.6 Target Tracking Policy

The target tracking policy is defined in Cartesian task space and drives the end-effector toward a desired pose.

The desired acceleration can be expressed as:

$$a_{target} = -K_p(p_{ee} - p_{des}) - K_d\dot{p}_{ee} \tag{4.7}$$

where:

- K_p is the proportional gain controlling convergence speed,
- K_d is the damping coefficient ensuring stable motion.

Within RMPflow, this policy is associated with a metric that determines its relative importance compared to other policies.

4.7 Collision Avoidance Policies

To prevent collisions with environmental objects, obstacle avoidance policies generate repulsive accelerations when the robot approaches dangerous regions.

These policies typically operate on distance-based task spaces defined between robot links and obstacles.

The resulting acceleration increases rapidly as the robot approaches an obstacle, effectively pushing the motion away from the collision region.

Such policies are particularly important in palletizing scenarios where the robot frequently operates in close proximity to surfaces such as the pallet and the picking table.

4.8 Zone-Dependent Gain Scheduling

One of the key contributions of this project is the introduction of a spatially adaptive motion policy that adjusts controller behavior based on the current region of the workspace.

The workspace is partitioned into two main categories:

- **Critical Zones**
  Regions near the picking table, the orientation table, and the pallet where precise motion is required.
- **Transit Zones**
  Open regions of the workspace where the robot moves between task locations.

When the robot operates in transit zones, controller gains are tuned to favor faster motion and longer step sizes.

Conversely, when the end-effector approaches a critical zone, the controller transitions to a high-precision mode characterized by lower velocities and stronger positional accuracy.

This adaptive strategy allows the robot to maintain efficiency during long-distance motions while still ensuring safe and accurate manipulation near contact surfaces.

RMP Flow Config Path	**Parameter**	**Fast**	**Slow**	**Unit**
`joint_velocity_cap_rmp`	`max_velocity`	8.0	4.0	rad/s
`joint_velocity_cap_rmp`	`velocity_damping_region`	1.5	1.5	rad/s
`target_rmp`	`accel_p_gain`	1400	1500	$1/s^2$
`target_rmp`	`accel_d_gain`	500	550	$1/s^2$
`target_rmp`	`metric_alpha_length_scale`	0.05	5.0	m
`axis_target_rmp`	`accel_p_gain`	4500	2500	$1/s^2$
`axis_target_rmp`	`accel_d_gain`	500	500	$1/s^2$
`axis_target_rmp`	`metric_scalar`	30	100	scalar

Table 4.1: Main RMPflow parameters distinguishing the fast and slow operating modes.

This table summarizes the main RMPflow parameters used in the implementation to distinguish the two operating modes. The slow-mode values are those explicitly assigned in `rmpflow_utils.py`, while the fast-mode entries are left blank because, in the current implementation, fast mode uses the default planner configuration loaded at initialization.

This adaptive speed management mechanism guarantees an optimal balance between operational efficiency and safety: transfers without load or with the box grasped far from surfaces are executed at the maximum speed permitted by the robot, while critical phases of contact, manipulation and deposit are slowed down to minimise the risks of box damage, accidental collisions or grasping failures.

4.9 Practical Outcomes and Performance

Experimental evaluation of the control architecture demonstrated several advantages.

First, the robot produces smooth trajectories when moving through unconstrained regions of the workspace. The combination of multiple RMP policies ensures that the generated motions remain physically plausible and dynamically stable.

Second, when approaching grasping or placement locations, the controller automatically reduces motion speed and increases positional accuracy. This behavior significantly improves the reliability of the grasping and placement operations.

Third, the adaptive gain scheduling mechanism reduces overall task completion time by allowing the robot to move quickly during transit phases while maintaining careful motion near critical areas.

Overall, the proposed motion planning strategy provides an effective balance between efficiency, precision, and safety within the simulated palletizing environment.

Chapter 5

Package Reorientation

In robotic palletizing workflows, packages often arrive in a standardized orientation that is not necessarily compatible with the optimal packing configuration required for pallet construction. As a result, an intermediate manipulation stage is required in which the robot adjusts the orientation of the package before the final placement operation.

This stage is referred to as *package reorientation*. It represents a geometric transformation applied to the package pose so that its orientation matches the configuration specified by the pallet packing strategy.

In the considered scenario, all packages initially arrive on the picking conveyor in a stable horizontal configuration. Their dimensions satisfy the ordering:

$$d_x \geq d_y \geq d_z \tag{5.1}$$

meaning that the largest face of the package always lies parallel to the ground plane. This configuration guarantees maximum stability during transport and simplifies the picking operation since the robot always approaches the top surface of the package.

However, the final pallet configuration may require packages to be placed in different orientations in order to satisfy several packing objectives:

- maximize pallet volume utilization,
- enable interlocking patterns between layers,
- reduce pallet height,
- ensure structural stability of stacked packages.

Consequently, the manipulation pipeline includes a dedicated reorientation phase in which the robot modifies the orientation of the package before placing it onto the pallet.

5.1 Data-Driven Orientation Encoding

Rather than computing the desired package orientation dynamically during execution, the project adopts a *data-driven orientation strategy*. The orientation of each package is determined by a packing algorithm and stored within a dataset that describes the full pallet configuration.

Each package is therefore associated with a discrete identifier called `orient_id`. This identifier specifies which orientation the package must assume before being placed onto the pallet.

The mapping between orientation identifiers and the corresponding rotations is defined through a dictionary of Euler angles:

```
orientations_euler = {
    0: [0, π/2, 0],
    1: [π/2, 0, π/2],
    2: [π/2, 0, 0],
    3: [-π/2, 0, 0],
    4: [π/2, π/2, 0],
    5: [π/2, 0, π/2],
}
```

Each entry represents a rotation defined by three Euler angles corresponding to rotations around the X, Y, and Z axes of the robot reference frame.

Using discrete orientation identifiers offers several advantages:

- deterministic execution of pallet configurations,
- direct compatibility with external packing algorithms,
- minimal computational overhead during runtime.

The packing solver simply outputs the required `orient_id`, and the robotic controller applies the corresponding rotation without needing to recompute orientation decisions online.

5.2 Quaternion Representation

Although Euler angles are convenient for defining discrete orientations, they are not well suited for interpolation or control in robotic systems due to issues such as singularities (gimbal lock) and discontinuities in rotation trajectories.

For this reason, all Euler rotations are converted internally into quaternion representation before being used by the motion controller.

Formally, for a given orientation identifier, the target quaternion is computed as:

$$q_{target} = \text{EulerToQuat}(\phi, \theta, \psi) \tag{5.2}$$

where (ϕ, θ, ψ) correspond to the Euler angles retrieved from the orientation mapping.

In the implementation, this conversion is performed using the utility function:

`euler_angles_to_quats()`

provided by the Isaac Sim numerical utilities.

The resulting quaternion is then normalized:

$$q = \frac{q}{||q||} \tag{5.3}$$

Normalization ensures that the quaternion represents a valid rotation and avoids numerical drift during repeated transformations.

5.3 Orientation Generation Logic

The orientation logic is implemented through a dedicated function that computes the quaternion corresponding to the required `orient_id`. The function also considers the `bundling_axis`, which determines how packages are arranged within the pallet layer.

Two main bundling configurations are supported:

- **Bundling along the Z-axis**

  Packages are aligned according to the vertical stacking axis. This configuration corresponds to the general palletizing case, where packages are stacked in standard orientations.

- **Bundling along the Y-axis**

  Packages are arranged laterally along the pallet width. This configuration is used for vertical placements, where packages are meant to be oriented vertically or in special packing patterns.

Depending on the selected bundling axis, the system uses a different mapping of Euler angles for the orientation identifiers.

The orientation computation function returns two different quaternions:

- q – the orientation used during the reorientation stage,
- q_{place} – the orientation used during the final placement on the pallet.

This distinction is necessary because the robot may approach the package from different directions depending on the manipulation phase.

If the package orientation corresponds to a configuration where the robot operates from above (i.e. when the Euler angle around the Y axis equals $\pi/2$), the same quaternion can be used for both manipulation and placement.

Otherwise, the robot switches to a default overhead orientation during the placement phase to guarantee stable vertical insertion onto the pallet surface.

5.4 Reorientation Procedure

The reorientation process follows a structured manipulation pipeline composed of several stages:

1. The robot grasps the package from its top surface while it lies in the default horizontal orientation.
2. The end-effector transports the package to a dedicated reorientation table located within the workspace.
3. The package is positioned at a predefined pose on the table.
4. The robot executes a controlled rotational motion to transform the package orientation toward the desired quaternion q_{target}.

The rotation is executed by minimizing the quaternion error between the current end-effector orientation and the desired target orientation:

$$q_{error} = q_{target} \otimes q_{current}^{-1} \tag{5.4}$$

where $\otimes$ denotes quaternion multiplication.

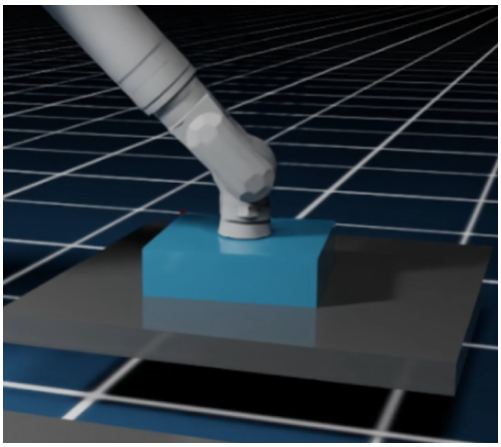
The controller generates angular velocity commands proportional to this error until the rotational deviation becomes sufficiently small.

The convergence condition is defined as:

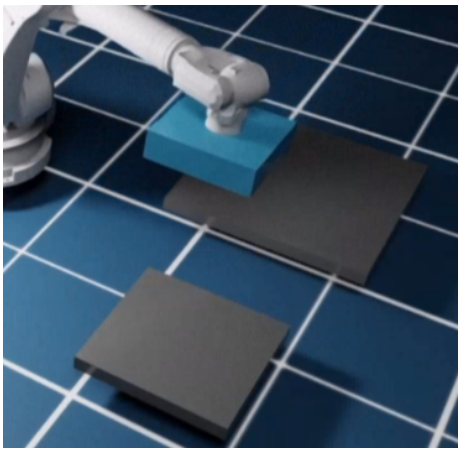
$$\|q_{error}\| < \epsilon \tag{5.5}$$

where ϵ represents a tolerance threshold defining acceptable orientation accuracy.

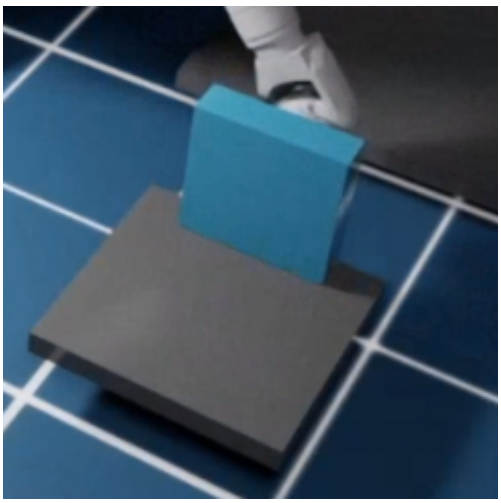
Once this condition is satisfied, the package is considered correctly oriented and the manipulation pipeline proceeds to the pallet placement stage.



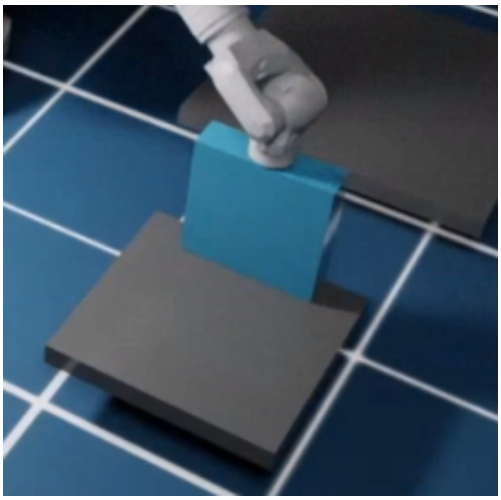
(a) Picking box in horizontal configuration.



(b) Transport box to reorientation table.

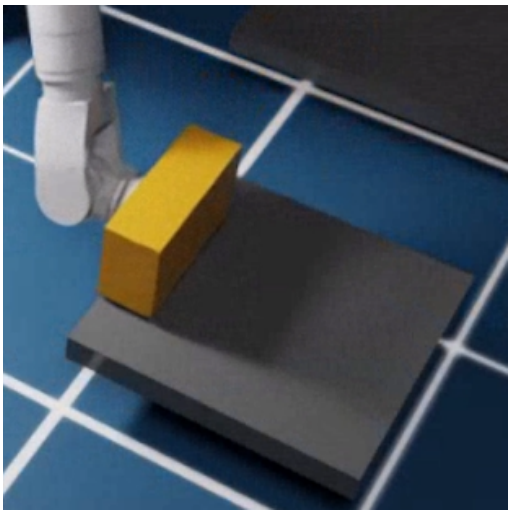


(c) Rotate box in vertical configuration.



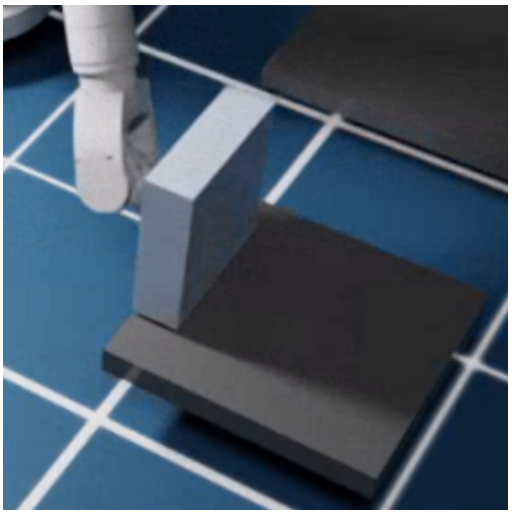
(d) Picking box in vertical configuration.

Figure 5.1: Reorientation phase with a box with `orient_id = 1` and `bundling_axis = y`

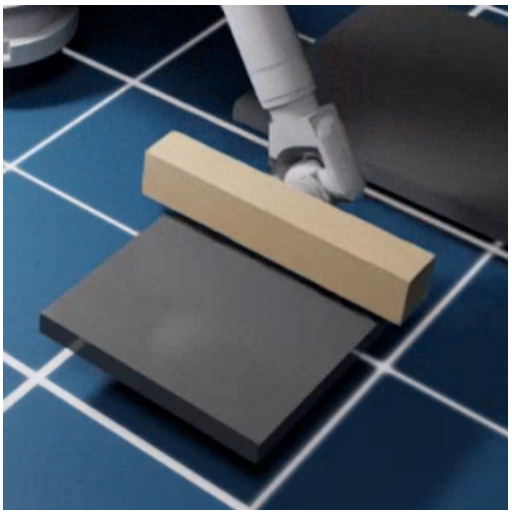


(a) Rotate the box with `orient_id = 4` and `bundling_axis = y`

(b) Rotate the box with `orient_id = 0` and `bundling_axis = y`



(c) Rotate the box with `orient_id = 3` and `bundling_axis = y`



(d) Rotate the box with `orient_id = 3` and `bundling_axis = z`

Figure 5.2: Reorientation phase with different orientations and bundling axes.

5.5 Contact-Aware Rotation Strategy

The reorientation operation must be executed carefully to avoid collisions between the robot manipulator and the surrounding environment.

A key design consideration concerns the interaction between the rotating package and the reorientation table.

If the package were placed at the center of the table during rotation, the robot wrist or forearm could intersect the table surface while executing the rotational motion.

To avoid this issue, the package is intentionally positioned near the edge of the reorientation table.

This configuration provides several benefits:

- increased clearance for the robot wrist during rotation,
- reduced risk of collision with the table surface,
- improved manipulability of the robot arm.

The robot first lowers the package to a safe height above the table edge while maintaining a small vertical clearance. The rotational motion is then executed in free space while keeping the package close to the support surface.

This strategy minimizes the risk of dropping the package while still guaranteeing sufficient clearance for safe rotational motion.

5.6 Advantages of the Reorientation Module

The introduction of a dedicated reorientation stage provides several important advantages within the palletizing pipeline.

First, it decouples the grasping operation from the final placement orientation. The robot can always pick packages using the same stable top-down grasp while adjusting the orientation later in the process.

Second, the use of discrete orientation identifiers ensures that pallet configurations generated by external packing algorithms can be executed deterministically within the robotic system.

Third, the quaternion-based rotation control guarantees smooth rotational trajectories that avoid singularities and maintain stable robot motion.

Finally, the modular structure of the reorientation process simplifies the overall manipulation architecture. Each stage of the pipeline focuses on a specific geometric transformation, improving system robustness and maintainability.

For these reasons, the reorientation module acts as a deterministic geometric transformation component within the palletizing pipeline, ensuring that each package reaches the pallet in the exact orientation required by the packing strategy.

Chapter 6

Collision Avoidance and Directional Placement Strategy

The final deposit phase is the most contact-sensitive part of the palletizing cycle. At this stage the robot is no longer operating in free space: it must insert a grasped package into a partially filled pallet while preserving the arrangement already created by previous placements. In the implemented system this problem is addressed through two distinct, complementary components that must not be conflated.

The first component is a **placement margin algorithm**: its purpose is to post-process an ideal pallet layout and obtain a realizable arrangement with a controlled minimum spacing between boxes. The second component is a **direction-selection module**: this module does not alter the layout itself; instead, it determines how the robot should approach the final pose during the insertion phase.

6.1 Two Complementary Roles

The distinction is central to the logic of the thesis.

- The placement margin algorithm acts on the *target layout*. It reads the planned box arrangement from JSON, separates boxes that are too close, preserves metadata such as orientation identifiers, and produces a modified layout that is more robust for downstream execution.
- The direction-selection module acts on the *approach motion*. Given the target box and the boxes already present on the pallet, it determines

whether the final motion should be purely vertical or should follow a specific horizontal or diagonal insertion corridor.

This split is also consistent with the software architecture: some problems are easier to solve once, at the layout level, while others depend on the local contact geometry encountered during the actual placement of each box.

6.2 Placement Margin Algorithm

The algorithm defines a `PalletSeparator` class used to convert an ideal pallet arrangement into a separated one. The need for this module arises from a practical limitation of dense packing solutions: even if a layout is geometrically valid, adjacent boxes with zero or near-zero clearance can generate unstable contact resolution in the physics engine once the robotic cell tries to execute the plan.

Each box is represented by the lower corner and the box dimensions,

$$(x, y, z, l, w, h),$$

which is sufficient for reasoning about layer membership, lateral spacing, and pairwise overlaps. The algorithm keeps the vertical coordinate unchanged and focuses on the horizontal plane, because the intent is to regularize placement feasibility without rewriting the stacking order provided by the upstream planner.

The layout regularization follows three principles:

1. **Layer-wise processing.** Boxes are grouped by height and separated within the same layer, which is where lateral conflicts matter most.

2. **Minimal-displacement correction.** For each conflicting pair, the algorithm computes how much extra gap is needed along X or Y and applies the smaller correction, splitting the displacement according to the available margin toward the pallet borders.

3. **Stack consistency.** After separation, vertical-alignment constraints are re-applied so that upper boxes remain coherently supported by the lower ones whenever the original layout encoded a stack.

This second step can be summarized in compact form. Let $g_{\min}$ denote the required minimum lateral clearance, and let g_x and g_y be the current

clearances between two conflicting packages along the pallet axes. The required corrections are

$$\Delta_x = \max(0, g_{\min} - g_x), \qquad \Delta_y = \max(0, g_{\min} - g_y). \tag{6.1}$$

The algorithm then selects the axis requiring the smallest feasible displacement,

$$k^* = \arg\min_{k \in \{x,y\}} \Delta_k, \tag{6.2}$$

and applies the correction along that axis while distributing the displacement according to the residual free space available toward the pallet boundaries. This expression does not change the implementation; it only makes explicit the minimal-displacement logic already described above.

The implementation also includes a feasibility-recovery strategy. If the requested minimum distance cannot be achieved within the real pallet footprint, the algorithm temporarily introduces a *virtual pallet margin*. This enlarges the effective workspace during the correction stage and allows difficult layouts to be regularized before being written back to JSON.

6.3 Direction Selection

The second component addresses a different question: once a target pallet pose is known, from which side should the robot approach it?

The direction-selection module evaluates a discrete set of candidate insertion directions on the pallet plane:

```
DIRECTIONS = {
    "Center", "Nord", "Nord_Est", "Est", "Sud_Est",
    "Sud", "Sud_West", "West", "Nord_West"
}
```

The label `Center` represents a purely vertical placement, whereas the other directions correspond to horizontal or diagonal insertion corridors. The selected direction is later consumed by the waypoint-generation logic in the robot controller, which computes the appropriate pre-placement pose.

6.4 Geometric Representation and Contact Reasoning

For the selection of the direction, each box is modeled as an axis-aligned rectangular prism. For directional reasoning, only the projection on the pallet plane is required. Let B_t be the target box and B_o a neighboring box already placed on the pallet. Their planar rectangles are denoted by

$$R_t = (x_t^{\min}, x_t^{\max}, y_t^{\min}, y_t^{\max}), \qquad R_o = (x_o^{\min}, x_o^{\max}, y_o^{\min}, y_o^{\max}). \tag{6.3}$$

The module first verifies whether the two boxes overlap along the vertical axis. If they do not belong to the same relevant height range, they are ignored. Otherwise, the planar relationship is analyzed to determine which sides of the target region are obstructed.

To make this test conservative, the algorithm uses *inflated* planar rectangles. Instead of reasoning on the nominal box footprint only, each footprint can be expanded by a small amount:

$$\delta_{\text{inflation}} = 10 \text{ mm}. \tag{6.4}$$

This buffer allows the system to detect near-collision situations before they become problematic in the simulator.

Once the relevant contacts have been identified, the algorithm estimates how much free space is available along each candidate insertion direction. Conceptually, a ray is cast from the target-box center in the opposite direction of the intended motion. The algorithm then computes the distance to the first obstacle, where obstacles include both neighboring boxes and pallet boundaries.

The underlying geometric primitive is a ray–axis-aligned-bounding-box intersection test. The output of this analysis is a corridor length, that is, the available straight-line distance along which the package can travel before colliding with an obstacle. If $\mathcal{D}_{\text{adm}}$ denotes the subset of admissible candidate directions and $L(d)$ the corresponding collision-free corridor length, the preferred insertion direction is

$$d^* = \arg\max_{d \in \mathcal{D}_{\text{adm}}} L(d). \tag{6.5}$$

This formal expression makes explicit the criterion already used by the implementation: among all directions compatible with the local contact configuration, the selected one is the direction providing the longest feasible insertion corridor.

Once the direction has been selected, the module computes the corresponding start point,

$$p_{\text{start}} = c + Ld, \tag{6.6}$$

where c is the target center, d is the normalized insertion direction, and L is the estimated corridor length.

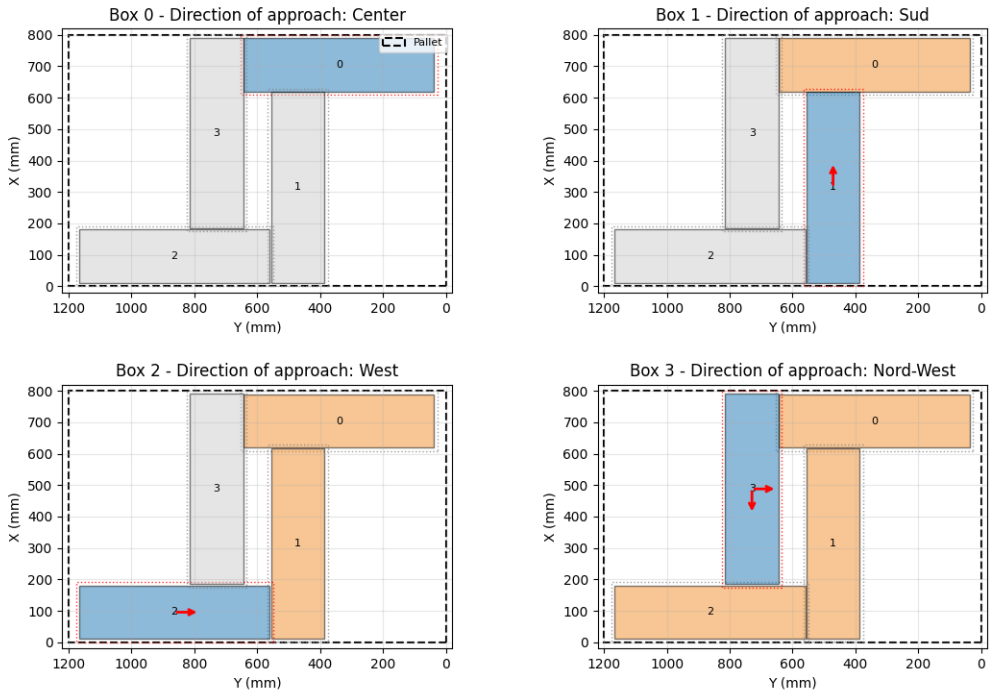


Figure 6.1: Example of directional placement analysis. The arrow denotes the insertion direction selected for the final approach.

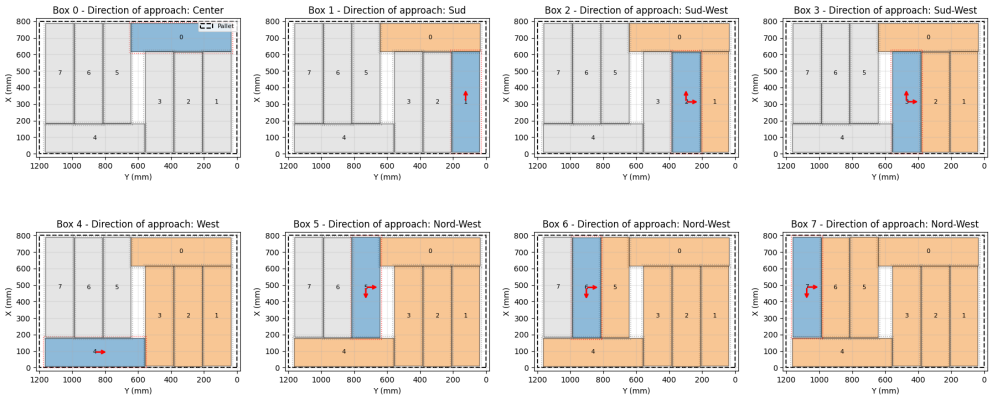


Figure 6.2: Example of directional placement analysis of a more complex scenario.

Figures 6.1 and 6.2 highlight that the final insertion direction is not chosen in isolation, but with respect to the packages that have already been placed around the target region. Their positions define the admissible contacts and constrain the available free corridor. Moreover, the contact analysis can detect multiple simultaneous contacts on the same neighboring package; this makes it possible to validate non-axis-aligned approaches as well, so that the package can follow a diagonal placement trajectory whenever this is the most compatible solution with the local arrangement.

6.5 Adaptive Clearance Margin

In addition, the system uses a small clearance margin between adjacent boxes whenever direct face-to-face contact would otherwise occur:

$$\delta \in [1 \text{ mm}, 10 \text{ mm}] \tag{6.7}$$

This offset is intentionally small: it does not change the macroscopic structure of the pallet, but it reduces the probability of repeated contact resolution, interpenetration artifacts, and lateral pushes generated during deposit.

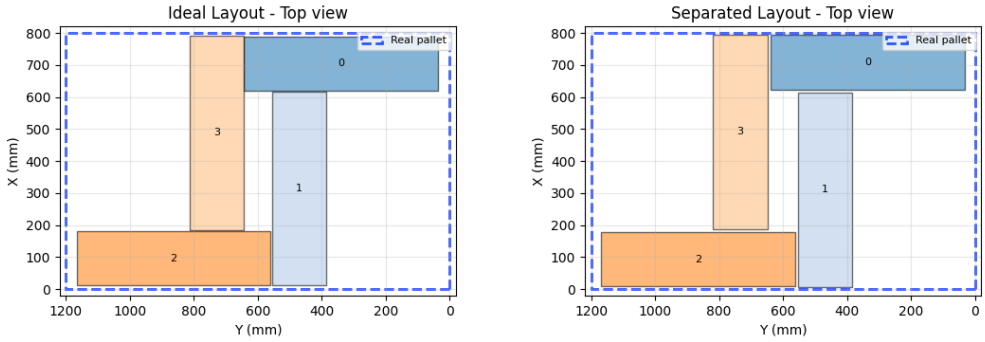


Figure 6.3: Example of an additional clearance margin introduced to increase placement robustness.

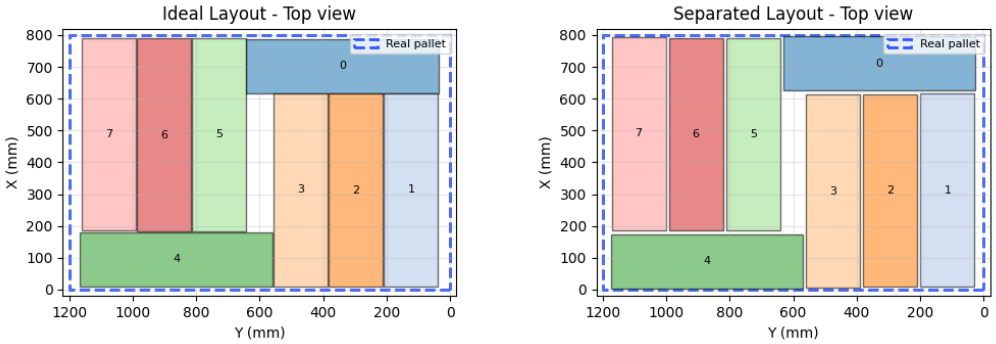


Figure 6.4: Example of an additional clearance margin of a more complex scenario.

Figures 6.3 and 6.4 illustrate how this auxiliary clearance margin preserves the overall pallet structure while reducing the probability of repeated side contacts during deposit.

6.6 Integration with Robot Execution

The placement strategy is tightly connected to the control architecture described in the previous chapters. The output of the adaptive margin process regularizes the target pallet layout before execution, whereas the output of the direction selection algorithm is used by the waypoint computation inside the state-machine logic. The resulting workflow is therefore two-level:

- geometric regularization of the target arrangement before execution;
- contact-aware choice of the insertion direction during final placement.

This division of responsibilities is one of the strengths of the implementation, because it avoids delegating every placement difficulty to the motion

controller alone.

The proposed strategy is deterministic, interpretable, and easy to debug. It is well suited to the goals of this thesis, which are centered on building a reliable execution framework for simulated palletizing. At the same time, the approach remains rule-based and relies on box-shaped geometric abstractions, manually selected margins, and discrete candidate directions. These assumptions are acceptable in the present context, but they also define the main limits of the method and motivate some of the future developments discussed in the conclusion.

Chapter 7

Computer Vision and Perception Pipeline

Reliable perception of incoming packages is a prerequisite for the entire manipulation pipeline. Before the robot can move toward a package, it must estimate the package position on the input table and recover the orientation required to build a consistent grasping pose. In the implemented system, this task is handled by a computer vision algorithm integrated into the main scenario class used by the Isaac Sim extension.

The design choice adopted in this thesis is intentionally pragmatic. Instead of using a deep neural detector, the perception pipeline relies on a fixed top-down camera geometry and a lightweight classical image-processing workflow. This choice is appropriate for the considered setting, where the object class is known, the camera is stationary, and the main source of uncertainty is the randomized placement of packages on the spawn table.

At a functional level, the perception module maps an RGB image into the grasping information required by the robot controller:

$$(\hat{\mathbf{p}}_W, \hat{\theta}) = \mathcal{V}(I_{\mathrm{RGB}}, K, T_{WC}), \tag{7.1}$$

where I_{RGB} is the camera image, K is the intrinsic matrix, T_{WC} is the camera pose in the world frame, $\hat{\mathbf{p}}_W$ is the estimated package center in world coordinates, and $\hat{\theta}$ is the estimated in-plane orientation angle. The implemented pipeline is composed of seven sequential steps:

1. ROI extraction;
2. HSV conversion;

3. Otsu thresholding;

4. morphological filtering;

5. contour detection;

6. pose estimation;

7. pixel-to-world conversion.

The following sections discuss these stages in the same order.

7.1 Simulation of Industrial Variability

To avoid an unrealistic deterministic setup, each spawned package is perturbed before perception takes place. In the current implementation the random translation is bounded by

$$\Delta x, \Delta y \in [-0.1, 0.1] \text{ m}, \tag{7.2}$$

while the random orientation around the vertical axis is bounded by

$$\Delta\theta \in [-10^\circ, 10^\circ]. \tag{7.3}$$

These perturbations are generated directly in the main code before the vision stage. Their purpose is not to model every industrial disturbance in detail, but to prevent the robot from relying on a single nominal spawn pose and to force the perception module to estimate the package pose online.

7.2 Overhead Camera Configuration

The perception system uses a simulated RGB camera mounted above the picking table in a top-down configuration. In the current implementation, the camera is placed at a height of 3.0 m above the spawn area and initialized with a resolution of 3840×2160 pixels and an acquisition frequency of 60 Hz. The camera orientation is specified as a quaternion corresponding to a downward-looking pose.

The perspective projection model is used, and the camera parameters are configured to provide a field of view that covers the entire spawn table with sufficient margin. The camera is set to capture RGB images only, without

depth or segmentation channels, to keep the perception pipeline simple and focused on the core task of pose estimation.

Representative full-frame acquisitions are shown in Figures 7.1 and 7.2.

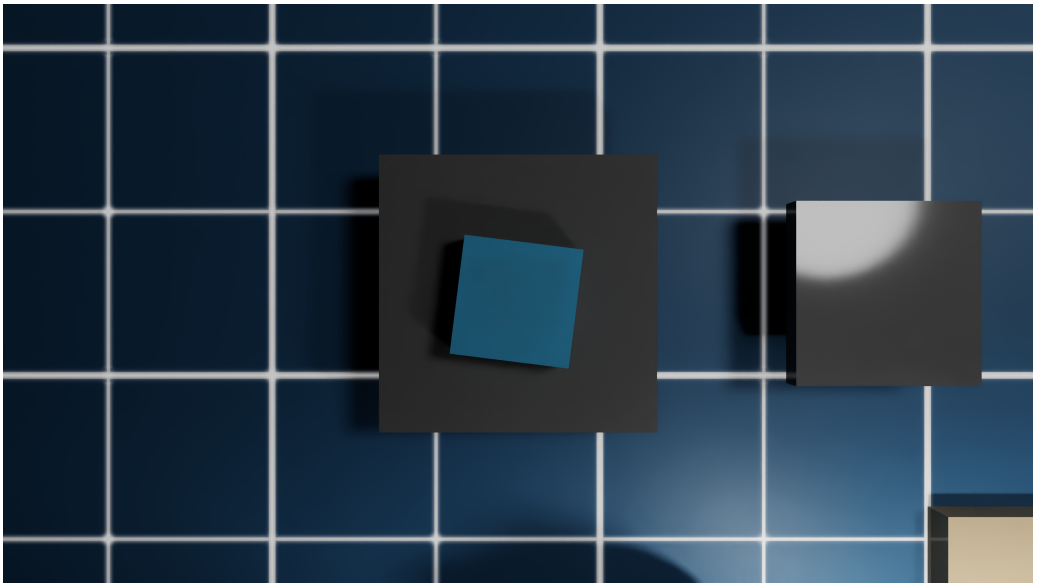


Figure 7.1: Example of the full RGB frame captured by the overhead camera. The package is randomly placed on the spawn table, and the perception pipeline must estimate its pose from this image.

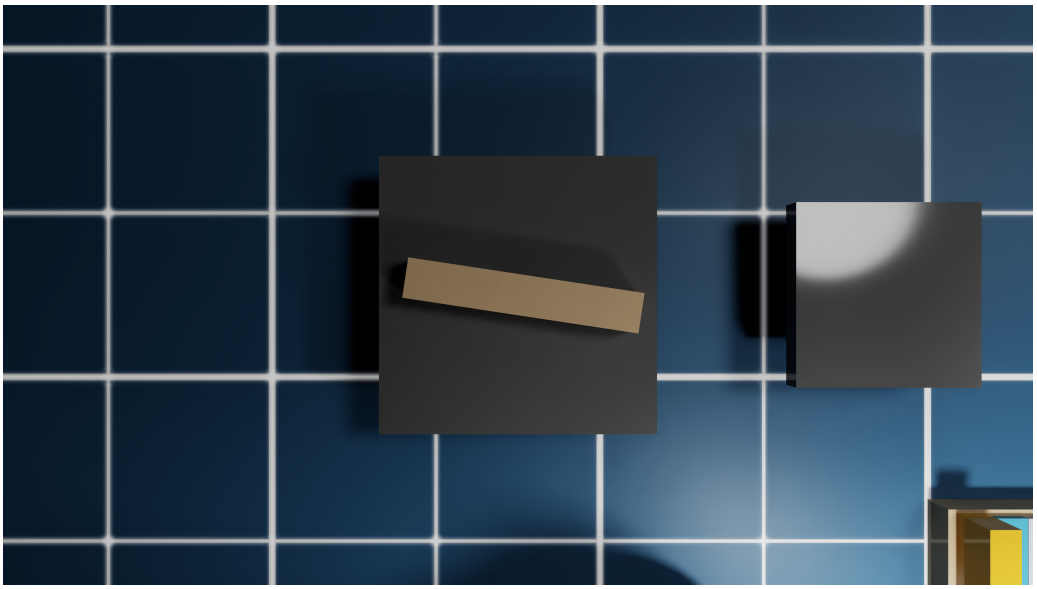


Figure 7.2: Example of the full RGB frame captured by the overhead camera with a different pose.

Once the sensor has been initialized, the intrinsic matrix is read directly

from Isaac Sim:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{7.4}$$

This choice is important because it avoids manually hard-coding projection parameters and keeps the vision model consistent with the simulated sensor configuration provided by the platform [5].

7.3 Region of Interest Extraction

The full camera frame contains much more than the spawn area. To reduce the influence of irrelevant background regions, the algorithm extracts a region of interest defined by fixed margins proportional to image width and height. In practice, the ROI is chosen so that the detection logic operates only on the central portion of the image corresponding to Table A, which is the table in which packages are spawned (picking table).

This step has two benefits. First, it reduces the probability of false detections caused by unrelated scene elements. Second, it simplifies the subsequent segmentation stage because the statistics of the region of interest are more stable than those of the full frame.

Examples of the cropped ROI are reported in Figure 7.3.

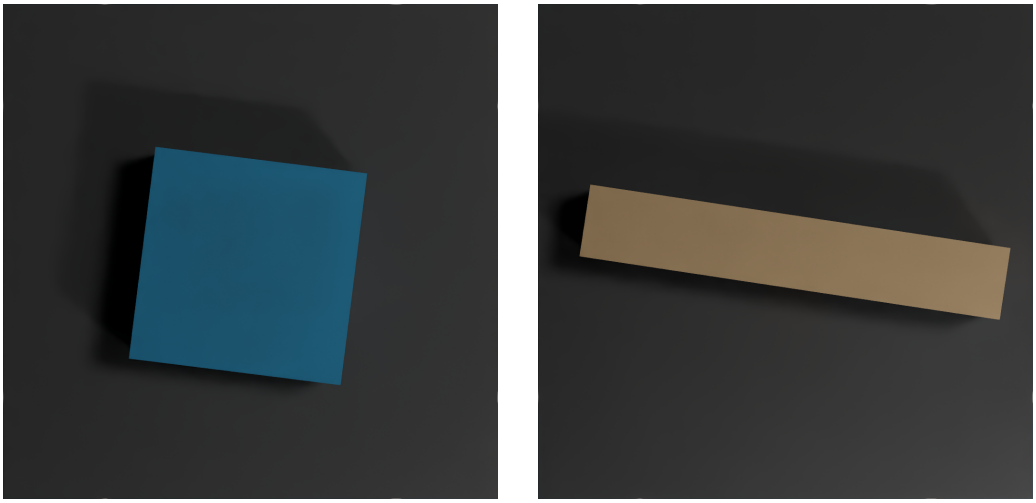


Figure 7.3: Region of interest extracted around the spawn table of different boxes in different poses.

7.4 Color Segmentation and Binary Mask Generation

After ROI extraction, the cropped RGB image is converted to HSV color space and the value channel is extracted. The value image is then smoothed with a Gaussian filter and thresholded using Otsu's method [7]. In the implementation, the Otsu threshold is further increased by a configurable strictness parameter, which makes the segmentation more selective. If the resulting mask contains too many foreground pixels, the mask is inverted to preserve a consistent object-background interpretation.

After thresholding, the algorithm applies morphological opening and closing with an elliptical structuring element. These operations remove isolated noise pixels and fill small gaps in the segmented region. The overall image-processing logic follows standard OpenCV operators and conventions [6]. Representative masks are shown in Figure 7.4.

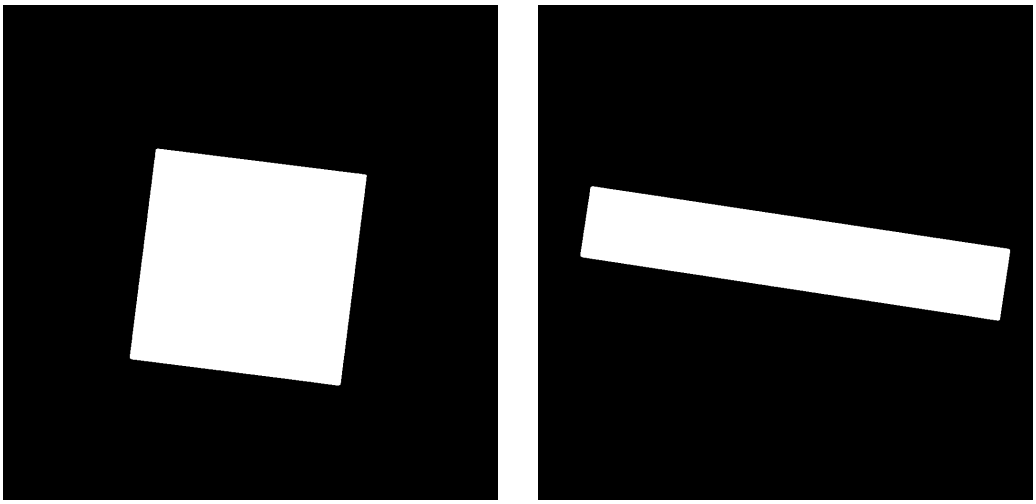


Figure 7.4: Binary masks obtained after segmentation and morphology of the different boxes.

7.5 Contour Detection and Pose Estimation

Contours are extracted from the binary mask, and the largest contour above a minimum area threshold is selected as the candidate package. The pose estimate is then derived from the minimum-area bounding rectangle computed by `cv2.minAreaRect`. This rectangle provides three key quantities:

- the pixel coordinates of the rectangle center;

- the rectangle side lengths;
- the in-plane rotation angle.

The major axis of the rectangle is used to estimate package orientation. The angle is normalized to remain within a compact interval, which reduces discontinuities and simplifies the subsequent conversion to a grasping quaternion. In practice, the implementation uses the direction of the left edge of the detected rectangle to measure the package orientation angle in the image plane. The same direction is then drawn on the demonstrative output image, so that the estimated angle becomes immediately visible and can be checked qualitatively during debugging and validation. The intermediate and final overlays are shown in Figures 7.5 and 7.6.

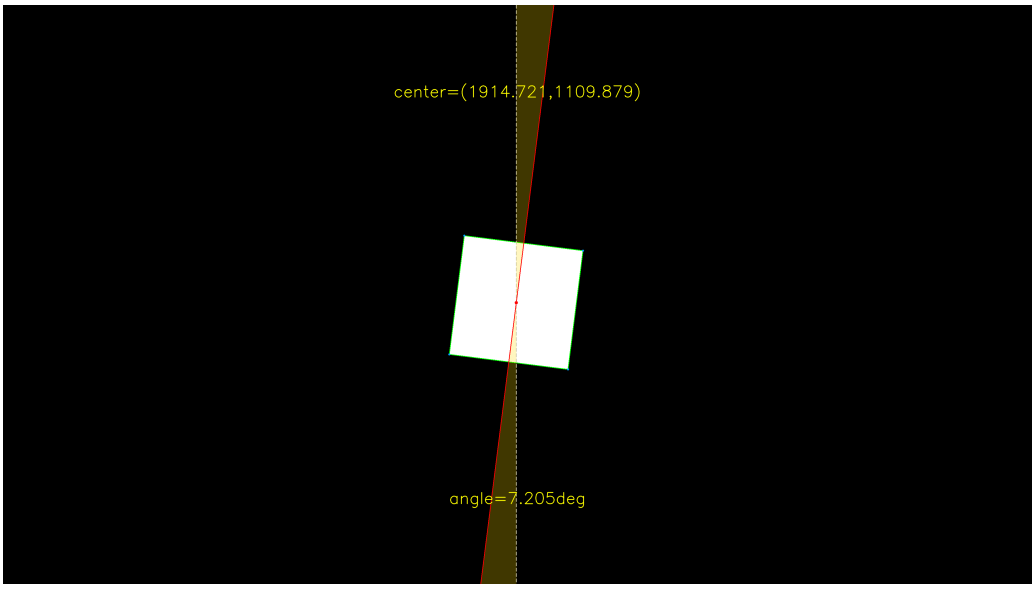


Figure 7.5: Detection overlay on the segmented mask, showing the estimated center, bounding box, and orientation angle cues.

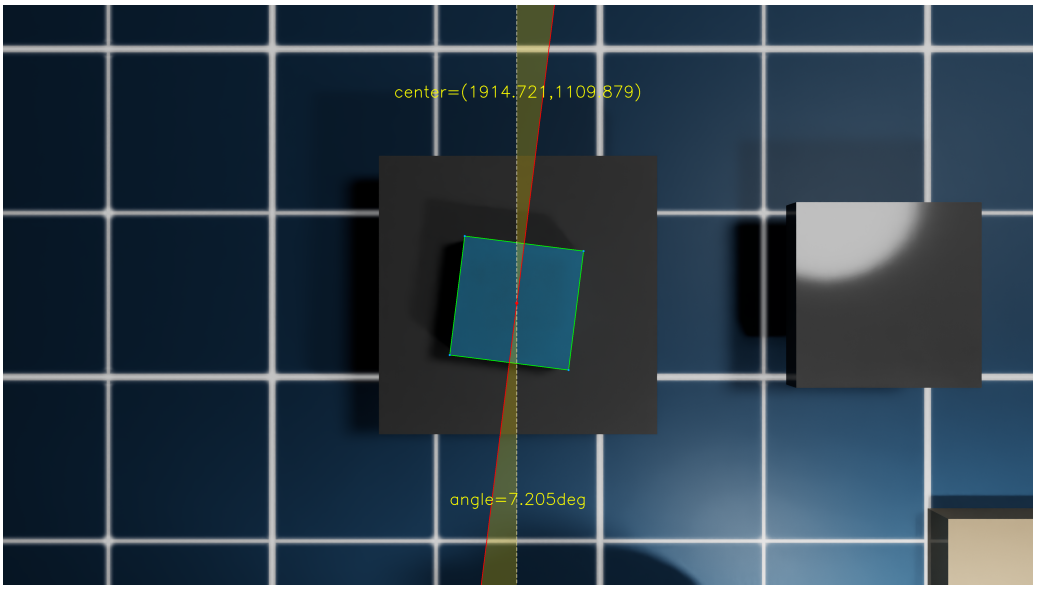


Figure 7.6: Detection overlay on the RGB image, showing the estimated center, bounding box, and orientation angle cues.

7.6 Pixel-to-World Coordinate Transformation

Once the center of the package has been found in image coordinates, the robot still needs a target in the world frame. The implemented conversion assumes a fixed top-down camera geometry. Given a detected pixel coordinate (p_x, p_y), the normalized image coordinates are computed as

$$x_n = \frac{p_x - c_x}{f_x}, \qquad y_n = \frac{p_y - c_y}{f_y}. \tag{7.5}$$

Assuming a known working height, these normalized coordinates are converted into metric offsets in the camera frame and then added to the camera world position. The sign of the vertical image coordinate is inverted so that image coordinates and world coordinates remain consistent under the top-down convention. The final output is a 3D point in the Isaac Sim world frame, used as the grasp target for the pick phase.

This mapping is intentionally simple. It is effective because the camera pose is fixed, the object lies on a known table plane, and the thesis focuses on a structured industrial setup rather than on fully general 6D pose estimation.

7.7 Orientation Reconstruction for Grasping

The detected in-plane angle is not used directly as a raw scalar. Instead, it is converted into a quaternion correction and combined with a base orientation associated with the robot end-effector. The implementation explicitly selects the smallest equivalent angular correction so that the manipulator does not perform unnecessary rotations before grasping the box.

This is one of the small but important engineering choices that improve motion quality. A vision module that returns a geometrically correct but rotationally inconsistent angle would still create avoidable complexity for the control layer. By normalizing the angle early, the perception pipeline provides cleaner input to the state machine.

7.8 Debug Outputs and Practical Role

The perception module saves debug images for the full RGB frame, the cropped ROI, the segmentation mask, and the final overlay. This is a valuable practical feature because it makes it possible to inspect failed detections without replaying the whole simulation.

Within the complete palletizing workflow, the output of the perception system is deliberately narrow but essential: the controller receives the estimated grasp point and the orientation needed to align the end-effector with the spawned package. Everything that follows, namely approach, grasp, lifting, optional reorientation, and pallet placement, depends on this first bridge between sensing and action.

The main strength of the implemented perception pipeline is therefore not generality, but suitability to the problem at hand. It is lightweight, explainable, and well aligned with the controlled geometry of the simulated palletizing cell. Its main limitation is that it relies on fixed viewpoints, hand-crafted segmentation logic, and package-like visual assumptions. Compared with learned detectors such as YOLO-based object detectors or pose-estimation networks such as PoseCNN, the implemented method is easier to interpret and debug, but it is less robust to strong appearance variability, clutter, and partial occlusions. Isaac Sim nevertheless provides a natural path toward these alternatives, because the same digital twin can be used to generate synthetic labeled images for future detector training without changing the current execution architecture.

Chapter 8

The State Machine Architecture of the Palletizing Robot

8.1 Introduction and System Architecture

The simulated palletizing system implemented in NVIDIA Isaac Sim is based on a software architecture centred around the `KukaAttachExample` class, which encapsulates the entire control logic of the KUKA KR210 robot over the life cycle of each individual box. The executive core of this architecture is the `update()` function, called at every simulation step by the Isaac Sim framework. It constitutes, in effect, the main body of the discrete-time control loop: at each invocation it receives the current time-step value, updates the state of the motion planner, and advances the finite state machine that governs the operational sequence of the robot.

The general structure of the `update()` method is organised into three logical macro-blocks executed in sequence at every call. In the first block, the operative positions depending on the current box are computed from the data read from the planning JSON file. In the second block the state machine proper is found, implemented through a chain of mutually exclusive conditional statements (`if`/`elif`) indexed by the integer variable representing the current state. In the third and final block the robot control is executed: the target positions are transmitted to the RMPflow planner, which generates the vector of joint actions, which are subsequently applied to the robot's kinematic model through the `ArticulationMotionPolicy` interface.

8.2 Data Structures and State Variables

Before describing the flow of the state machine, it is necessary to introduce the main state variables that govern its behaviour. The variable `self._step` is the integer counter that uniquely identifies the current state and is the sole index of the machine. The variable `self._boxes` counts the number of boxes already successfully palletised, and is incremented at the end of the operational cycle of each box. The boolean variable `self.box_spawned` signals whether the current box has already been materialised in the scene.

The verification of target-reaching conditions is delegated to instances of the `TargetChecker` class, of which the system instantiates three with differentiated tolerances:

- `self.target_checker` is used during precision phases (approach, deposit and rotation), with the most restrictive tolerances: positional threshold of 2 mm and an angular tolerance of 0.5°;
- `self.position_checker` is employed for intermediate transfers, with a positional threshold of 5 mm and an angular tolerance of 1°, requiring the condition to be satisfied for at least 3 consecutive steps;
- `self.box_checker` is reserved for the final quality assessment of the placement of all boxes.

The management of the planner's operating speed relies on the pair `self._rmpflow_fast` and `self._rmpflow_slow`, configured respectively for rapid movements in free zones and for slow movements during contact or deposit phases. The boolean flag `self._in_slow_mode` tracks the active modality, while `self._slow_for_placement` specifically indicates that the deceleration is motivated by a positioning operation.

8.3 The State Machine: Description of the Operational Flow

The state machine is implemented as a sequence of mutually exclusive conditional blocks, each identified by an integer value of `self._step` between 0 and 20. The transition from one state to the next occurs when the completion condition of the current state is satisfied, typically verified by one of the `TargetChecker` instances. The operational cycle for a single box comprises

21 states. At the completion of 21st state, the counter `self._boxes` is incremented, the flag `self.box_spawned` is reset, and `self._step` is brought back to 0, initiating the cycle for the next box.

It is important to note that the state machine is not strictly linear: at the end of step 5 and step 6 there are logical bifurcations that allow skipping steps 6–13 (relating to the orientation phase on the auxiliary table) when the box orientation is already correct for the deposit. In such cases, control passes directly to step 15, significantly shortening the cycle.

Summary Table of States

Step	**Description**	**RMPflow**	**Gripper**
0	Spawn box	—	—
1	Computer vision (position detection)	—	—
2	Move above the box to be picked	Fast	—
3	Descend and grasp the box	Slow	Close
4	Lift box (lift-off)	Fast	Closed
5	Safe intermediate position 1 (above spawn zone)	Fast	Closed
6	Safe intermediate position 2 (above orient. table)	Fast	Closed
7	Rotate box on orientation table	Slow	Closed
8	Release box on orientation table	Slow	Open
9	Step back to avoid collision	Slow	Open
10	Lift up to exit proximity zone	Slow	Open
11	Descend above box on orientation table	Slow	Open
12	Grasp box in new orientation	Slow	Close
13	Lift box above orientation table	Slow	Closed
14	Return to safe intermediate position 2	Fast	Closed
15	Transfer to intermediate position above pallet	Fast	Closed
16	Move above deposit position	Fast	Closed
17	Pre-deposit (proximal position)	Slow	Open
18	Deposit box at target position	Slow	Open
19	Step back above the pallet	Slow→Fast	Open
20	Return to safe intermediate position	Fast	Open

Table 8.1: Summary of the 21 states of the pick-and-place state machine.

Pseudocode of the State Machine

Algorithm 1 provides a compact pseudocode representation of the full pick-and-place state machine executed inside `update()` at every simulation step. The outer loop iterates over all boxes defined in the JSON dataset; the inner structure reflects the sequential – and conditionally shortened – chain of states described in Table 8.1.

Algorithm 1 Pick-and-Place State Machine for the Palletizing Robot

```
1:  procedure StateMachine(items, placements)
2:      while boxes < total_boxes do
3:          if step = 0 then                                   ▷ Spawn box
4:              SpawnBoxFromJson(boxes, dim, rand_offset); step ← 1
5:          else if step = 1 then                ▷ Computer vision detection
6:              frame ← GetCameraFrame
7:              (center, θ) ← DetectBoxFromTop(frame)
8:              pick_pos ← PixelToWorldTopdown(center); step ← 2
9:          else if step = 2 then                              ▷ Move above box
10:             ori_spawn ← ThetaToOrientation(θ)
11:             SetTarget(spawn_inter, ori_spawn)
12:             if position_reached then step ← 3
13:         else if step = 3 then                          ▷ Descend and grasp
14:             SwitchToSlowMode
15:             SetTarget(pick_pos, orient_spawn)
16:             if target_reached then
17:                 Gripper.Close
18:                 if gripper_closed then step ← 4
19:         else if step = 4 then                                  ▷ Lift box
20:             SetTarget(spawn_inter, orientation)
21:             if position_reached then step ← 5
22:         else if step = 5 then      ▷ Intermediate pose above spawn zone
23:             SwitchToFastMode
24:             SetTarget(intermediate_pos_1, orientation)
25:             if position_reached then
26:                 if orientation already correct for deposit then
27:                     step ← 15                    ▷ Bypass orientation table
28:                     else step ← 6
```

```
29:     else if step = 6 then      ▷ Intermediate pose above orient. table
30:         SetTarget(intermediate_pos_2, orientation)
31:         if position_reached then
32:             if only a flip is needed then
33:                 step ← 14                    ▷ Bypass re-grasp sequence
34:                 else step ← 7
35:     else if step = 7 then          ▷ Rotate box on orientation table
36:         SwitchToSlowMode
37:         SetTarget(table_pos_4, orientation_id)
38:         if target_reached then step ← 8
39:     else if step = 8 then          ▷ Release box on orientation table
40:         SetTarget(table_pos, orientation_id)
41:         if target_reached then
42:             Gripper.Open
43:             if gripper_open then step ← 9
44:     else if step = 9 then          ▷ Step back – horizontal retreat
45:         SetTarget(table_pos_0, orientation_id)
46:         if target_reached then step ← 10
47:     else if step = 10 then         ▷ Lift up from orientation table
48:         SetTarget(table_pos_1, place_orientation)
49:         if target_reached then step ← 11
50:     else if step = 11 then  ▷ Descend above box in new orientation
51:         SetTarget(table_pos_3, place_orientation)
52:         if target_reached then step ← 12
53:     else if step = 12 then         ▷ Grasp box in new orientation
54:         SetTarget(table_pos_2, place_orientation)
55:         if target_reached then
56:             Gripper.Close
57:             if gripper_closed then step ← 13
58:     else if step = 13 then             ▷ Lift above orientation table
59:         SetTarget(table_pos_1, place_orientation)
60:         if target_reached then step ← 14
61:     else if step = 14 then         ▷ Return to safe intermediate 2
62:         SwitchToFastMode
63:         SetTarget(intermediate_pos_2, place_orientation)
64:         if position_reached then step ← 15
65:     else if step = 15 then                      ▷ Move above pallet
```

```
66:             SetTarget(intermediate_pos, place_orientation)
67:             if position_reached then step ← 16
68:         else if step = 16 then                ▷ Move above deposit target
69:             SetTarget(place_inter, place_orientation)
70:             if position_reached then step ← 17
71:         else if step = 17 then          ▷ Pre-deposit at proximal position
72:             SwitchToSlowMode
73:             SetTarget(target_place_pre, place_orientation)
74:             if target_reached then
75:                 Gripper.Open
76:                 if gripper_open then step ← 18
77:         else if step = 18 then           ▷ Final deposit at target position
78:             SetTarget(target_place, place_orientation)
79:             if target_reached then
80:                 Gripper.Open
81:                 if gripper_open then step ← 19
82:         else if step = 19 then                ▷ Step back above the pallet
83:             SetTarget(place_inter, place_orientation)
84:             if target_reached then step ← 20
85:         else if step = 20 then       ▷ Return to safe position – next box
86:             SwitchToFastMode
87:             SetTarget(inter_pos, place_orient)
88:             if position_reached then boxes ← boxes + 1; step ← 0
89:         // Robot actuation – executed every simulation frame
90:         SetEndEffectorTarget(target_pos, target_ori)
91:     end while
92: end procedure
```

8.4 Detailed Description of Each Step

Step 0 — Box Spawning

Step 0 constitutes the entry point of the operational cycle for each box. The system first checks that the flag `self.box_spawned` is `False`, a condition indicating the absence of the box in the scene. If the condition is satisfied, the function `spawn_box_from_json()` is invoked, which materialises the 3D

model of the box in the simulation scene from the dimensions read from the JSON file, applying the previously sampled random translation and rotation offset. The flag `self.box_spawned` is set to `True` and the counter `self._spawn_wait_frames` is reset to zero. Unlike the other steps, step 0 does not provide for a waiting condition: the transition to the next state occurs immediately upon completion of execution.

Step 1 — Box Position Detection via Computer Vision

The second state is dedicated to detecting the position and orientation of the box through the computer vision system integrated in the simulator. The system first waits for a minimum of 10 frames to ensure that the simulation physics has stabilised the box's position. After this waiting period, an RGB frame is acquired from the overhead camera, and subsequently the image is analysed with computer vision algorithms to detect the box contour, determining its centre in pixels (`px`, `py`) and the planar rotation angle `theta`.

The pixel coordinates of the detected centre are then converted to three-dimensional world coordinates via the `pixel_to_world_topdown()` method, which performs the inverse projection from the overhead view to the box spawn plane, yielding the effective pick position `self._pick_pos`. The transition to step 2 occurs only if the detected centre is not `None`, ensuring robustness in case of detection failure.

Step 2 — End-Effector Movement Above the Box

In this state the robot is commanded to reach the intermediate position `spawn_inter`, placed directly above the pick position detected in the previous step, at a safe height that avoids collision with the box. Before setting the target, the gripper orientation for the pick phase is computed via `theta_to_orientation_spawn()`, which integrates the angle `theta` detected by computer vision with the world gripper orientation. Position checking is delegated to `self.position_checker`, allowing the robot to proceed only once it has stabilised in the neighbourhood of the target for the required number of frames.

Step 3 — Descent and Box Grasping

Step 3 represents the first critical phase of physical contact between robot and box. The system switches the RMPflow planner's operating mode to slow mode through the `switch_rmpflow_mode()` function, which reduces the maximum end-effector velocity to ensure a controlled and precise approach. The target is set to the effective pick position `self._pick_pos` with the orientation computed in step 2. The completion condition is managed by `self.target_checker`, which is more restrictive than `self.position_checker`: only upon its satisfaction is gripper closure commanded via `self.surface_gripper.close()`. The transition to step 4 occurs only when the gripper has confirmed its closed state (`is_closed()`), ensuring that the box is effectively grasped before proceeding with lifting.

Step 4 — Box Lift-Off

In this state the robot lifts the box vertically up to position `spawn_inter`, retracing in reverse the descent path of step 3. The slow mode is still activated in this step, and the deceleration ensures that the box is raised without jerks or oscillations that could compromise the grasp. Once `spawn_inter` is reached within the required tolerance, the state machine advances to step 5.

Step 5 — First Safe Intermediate Position (Above Spawn Zone)

This step moves the robot, with the box grasped, to the first global intermediate position `intermediate_pos_1`, located at coordinates $[-2.5, 2.5, 2.0]$ m, i.e. above the orientation table area. The objective is to create a high-altitude transit point that avoids interference with the physical structures present in the scene. In this step, the switch from slow mode to fast mode is also performed: the planner is replaced with `self._rmpflow_fast` and the `ArticulationMotionPolicy` is updated accordingly.

Upon reaching `intermediate_pos_1`, the state machine performs a logical bifurcation: if the `bundling_axis` is `'z'` and the `orient_id` is 0, or if the `bundling_axis` is `'y'` and the `orient_id` is 5 (predefined conditions for the boxes orientation), the box already has the correct orientation for deposit and does not need to pass through the orientation table. In this case

`self._step` is set directly to 15, skipping all steps from 6 to 14. Otherwise the machine proceeds linearly to step 6.

Step 6 — Second Safe Intermediate Position (Above Orientation Table)

The robot moves towards `intermediate_pos_2`, located at $[-0.5,\ 2.5,\ 2.0]$ m, i.e. directly above orientation table B. This waypoint is necessary to ensure a vertical approach to the table, avoiding oblique trajectories that could cause collisions with the table edge or with other objects in the scene. Similarly to step 5, upon reaching this waypoint a second bypass condition is verified: if the `bundling_axis` is `'z'` with `orient_id` equal to 4, or if the `bundling_axis` is `'y'` with `orient_id` equal to 2 (predefined conditions for the boxes orientation), the required rotation has already been performed and the machine jumps directly to step 14. Otherwise execution continues to step 7.

Step 7 — Box Deposit on the Orientation Table for Rotation

Step 7 manages the deposit of the box on the orientation table with the desired final orientation `orientation_id`, which corresponds to the orientation the box must have once placed on the pallet. The planner mode is brought to slow mode to ensure the precision required by the operation. The target is set to position `table_pos_4`, a proximal position to the table that precedes the actual deposit. The completion of this step prepares the robot for the box release in the subsequent step.

Step 8 — Box Release on the Orientation Table

In this state the robot positions itself at `table_pos`, the definitive deposit position on the table, maintaining the orientation `orientation_id`. Upon reaching the target, gripper opening is commanded via `self.surface_gripper.open()`. The transition to step 9 occurs only after confirmation of gripper opening (`is_open()`), ensuring that the box is effectively released and that the gripper attachment is deactivated before any subsequent robot movement.

Step 9 — Step Back to Avoid Collision with the Released Box

Immediately after the release, the robot must move away from the box while remaining in slow mode to avoid hitting it. Step 9 brings the end-effector to a position that represents a horizontal retreat from the release position. This manoeuvre is necessary to create sufficient lateral clearance to allow the subsequent vertical movement without interfering with the box just deposited.

Step 10 — Upward Movement to Exit the Box Proximity Zone

After the horizontal retreat, the robot rises vertically to position `table_pos_1`, maintaining the `place_orientation`. This vertical movement completes the exit sequence from the orientation table, bringing the gripper to a height sufficient to allow subsequent top-down approach movements without risk of collision with the deposited box.

Step 11 — Vertical Approach to the Box on the Orientation Table

Once the box has been released and the robot has moved away safely, the box must be retrieved in its new orientation. Step 11 brings the end-effector to a point above the box on the table, with the `place_orientation`. This position represents the beginning of the grasping descent, analogous conceptually to position `spawn_inter` in step 2, but referred to the orientation table.

Step 12 — Box Grasping in the New Orientation

Step 12 is the second critical point of physical contact. The robot descends to the effective pick position on the orientation table, with the placing orientation. Upon satisfaction of the precision condition by `self.target_checker`, gripper closure is commanded. The transition to step 13 is conditional on closure confirmation, exactly as in step 3. At the end of this step, the box is grasped in the correct orientation for the final deposit.

Step 13 — Box Lifting Above the Orientation Table

The robot rises with the grasped box to a safe distant position. This lifting frees the box from the table surface and brings it to a safe height for the subsequent transfer towards the pallet. Control is delegated to `self.target_checker`, ensuring movement precision.

Step 14 — Return to the Second Safe Intermediate Position

Step 14 brings the robot back to the intermediate position `intermediate_pos_2`, elevated above the orientation table, with the `place_orientation`. In this step the switch from slow mode to fast mode is performed: the planner `self._rmpflow` is replaced with `self._rmpflow_fast` and the policy is updated. This operation prepares the robot for fast transfers towards the pallet. Step 14 is also the arrival point of the bypass coming from step 6 (boxes that require rotation but are already correct in direction), guaranteeing flow consistency.

Step 15 — Transfer to the Intermediate Position Above the Pallet

The robot moves to `intermediate_pos`, located at $[0.0, 0.5, 2.0]$ m, i.e. above the pallet, with the `place_orientation`. This is the main transit point between the pick/orientation zone and the deposit zone. Fast mode ensures efficient transfer. Step 15 is also the arrival point of the bypass coming from step 5 (boxes already correctly oriented), ensuring that all paths converge at the same point before approaching the pallet.

Step 16 — Approach to the Deposit Zone (Above the Deposit Target)

The robot moves towards position `place_inter`, computed by `compute_waypoints()` as the approach point above the deposit position `target_place`, at a safety height. In this step the value of `direction_case` is logged, identifying the deposit approach direction chosen by the planning algorithm (typically one of the four cardinal directions relative to the pallet). Fast mode is maintained.

Step 17 — Pre-Deposit at the Proximal Position

Step 17 constitutes the first phase of the actual deposit. The system switches to slow mode and brings the robot to position `target_place_pre`, located 10 mm above the definitive deposit position. This pre-deposit position allows the robot to decelerate and stabilise before final contact with the deposit surface, reducing the risk of excessive impact. Upon satisfaction of the precision condition by `self.target_checker`, the step is completed.

Step 18 — Box Deposit at the Definitive Target Position

Step 18 represents the culminating moment of the operational cycle: the final deposit of the box on the position computed by the palletizing planner. The robot descends to `target_place`, the exact centre-of-mass position of the box on the pallet, maintaining the `place_orientation`. Upon reaching the target with the precision required by `self.target_checker`, gripper opening is commanded and the step is completed.

Step 19 — Step Back Above the Deposit Position

After the deposit, the robot rises to position `place_inter` to move vertically away from the deposit zone without interfering with the box just placed. In this step the planner is switched back to slow mode to ensure a controlled departure. This step is the exact mirror image of step 16: whereas in step 16 the robot descended towards the working zone, in step 19 it moves away from it.

Step 20 — Return to Safe Intermediate Position and Advance to Next Box

The last step of the cycle brings the robot to the general intermediate position `intermediate_pos`, returning it to a neutral and safe configuration. In this step the definitive switch from slow mode to fast mode is performed, analogous to steps 5 and 14. Upon satisfaction of the position condition by `self.position_checker`, the state machine executes the reset operations for the next cycle: `self._boxes` is incremented by one, `self.box_spawned` is set to `False`, and `self._step` is reset to 0. The operational cycle for the next box can then begin.

8.5 Robot Control: From State Machine to Actuation

At the end of the state machine execution, regardless of the current step, the `update()` function always executes the robot control block. This block constitutes the bridge between the logical planning of the state machine and the physical actuation of the robot in the simulation.

The control block is articulated in five main operations. First, the position and orientation of the robot base (`base_p`, `base_o`) are retrieved via the `get_world_pose()` method of the articulation. Second, the position (`tp`) and orientation (`to`) of the current target are read; the orientation quaternion is normalised to ensure numerical correctness of the computation. Third, `self._rmpflow.set_end_effector_target(tp, to)` is invoked, updating the end-effector objective in the RMPflow planner. Fourth, `self._rmpflow.update_world()` recomputes the obstacle avoidance configurations and `self._rmpflow.set_robot_base_pose(base_p, base_o)` updates the base position in the world reference frame. Finally, via `self._articulation_rmpflow.get_next_articulation_action(step)`, the vector of joint actions (joint positions and velocities) is computed, which is then applied to the robot's kinematic model via `self._articulation.apply_action(action)`.

At the end of the control block, `update()` records box and robot trajectories via `record_box_states()` and `record_robot_states()`, and updates the visual markers of the deposit vertices via `update_box_vertex_markers()`. This pattern ensures that, regardless of the state machine state, the robot always performs a planning and actuation step at every simulation frame, maintaining motion continuity.

8.6 Adaptive Management of the Planner's Operating Speed

A relevant aspect of the system architecture is the dynamic management of the RMPflow planner's operating speed. The system keeps in memory two planner instances configured with different parameters: `self._rmpflow_fast`, optimised for rapid transfers in obstacle-free zones, and `self._rmpflow_slow`, configured with reduced velocity limits for contact

and deposit operations.

Mode switches occur at precise points in the state machine, determined by the nature of the imminent operation. The switch to slow mode is activated at the beginning of approach, contact and deposit steps via the `switch_rmpflow_mode()` function, which first verifies that the switch has not already occurred recently (cooldown). The switch to fast mode instead occurs upon reaching safe intermediate positions far from obstacles, and consists in the direct replacement of the `self._rmpflow` instance with `self._rmpflow_fast` and the renewal of the `ArticulationMotionPolicy`.

8.7 Termination Condition and Palletizing Quality Evaluation

Before the state machine execution, the `update()` function verifies whether all boxes foreseen by the palletizing plan have been successfully placed, i.e. whether `self._boxes` $\geq$ `total_boxes`. If this is the case, the execution flow bypasses the state machine entirely and proceeds to a terminal block that performs the following operations: the robot is commanded to reach the safe intermediate position `intermediate_pos` as a rest posture; the trajectories of the boxes and the robot, recorded during the entire palletizing session, are serialised in JSON format and saved to a directory; finally, `self.box_checker.check_box_placement()` is invoked to compute the *reward* (number of boxes correctly placed within tolerance) and the *penalty* (number of boxes outside tolerance).

The evaluation system constitutes a quality supervision element for the entire simulated palletizing operation, providing a quantitative metric that can be used to assess the performance of the planning algorithm and the control system as a whole.

8.8 Concluding Remarks on the State Machine Architecture

The implemented state machine represents an effective solution for managing the operational complexity of a robotic palletizing system. Its sequential structure with conditional bifurcations allows boxes with different orientation requirements to be handled uniformly, reducing the average number of

operations required for simpler cases by means of the bypass of some steps.

The adaptive management of the planner's operating speed ensures safe behaviour during contact phases. The clear separation between the state machine logic and the robot control block, executed at every frame regardless of the current state, ensures continuity and stability of the robot's motion during state transitions.

Overall, the system implements a pick-and-place cycle articulated in 21 states, with the possibility of shortening it to 12 states for boxes that do not require re-orientation, for a total of 8 or 15 distinct end-effector movements depending on the complexity of the required orientation.

Chapter 9

Results

This chapter reports representative executions selected from the set of simulated orders. The study is based on 25 reference orders used as the common evaluation set for the analysis. Rather than including every generated result, the chapter reports only the most distinctive and representative cases for the project. This subset is sufficient to illustrate the main qualitative outcome of the thesis, namely the ability of the complete pipeline to execute different palletizing scenarios with stable motion, consistent target tracking, and readable package flow throughout the cell.

9.1 Aggregate Tracking Accuracy

Before discussing representative trajectories and screenshots, it is useful to summarize the global tracking performance measured over all 25 tested cases. Table 9.1 collects the mean and maximum tracking errors for both Cartesian position and end-effector orientation, together with the nominal acceptance limits adopted for the analysis.

Metric	**Mean**	**Max**	**Limit**
Position tracking error	**3.91 mm**	**5.68 mm**	**5 mm**
Orientation tracking error	**0.25°**	**0.85°**	**1°**

Table 9.1: Aggregate tracking metrics computed over the 25 reference test cases.

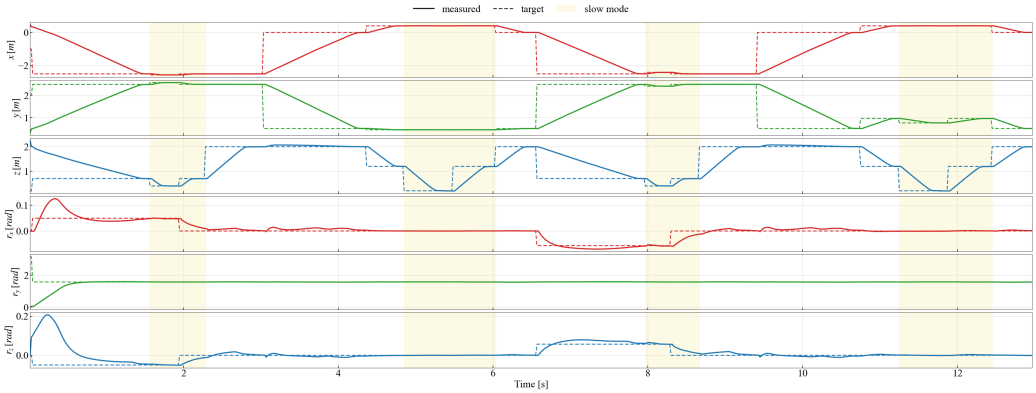
These results show that the proposed control architecture maintains a good level of tracking accuracy across the full evaluation set. In particular,

the mean position error remains below the 5 mm target, while the mean orientation error is substantially smaller than the 1° limit. The most relevant observation is that the worst-case orientation error still stays within specification, whereas the worst-case position error reaches 5.68 mm, thus exceeding the nominal threshold only slightly. Overall, the data indicate that the system is accurate and stable over the 25 analyzed cases, with the main residual criticality concentrated in a limited number of peak positional deviations.

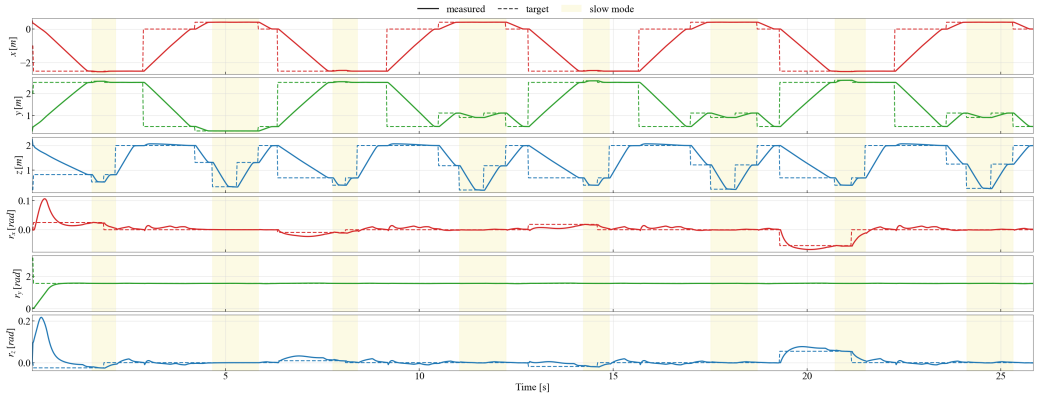
9.2 Robot Position and Orientation Profiles

Figure 9.2 summarizes the end-effector evolution over the full duration of some representative simulations. The solid curves correspond to the measured robot state, the dashed curves to the commanded targets, and the shaded intervals mark the phases in which the controller switches to slow mode for grasping, reorientation, or placement. Across the reported cases, the measured motion follows the reference with small deviations, while the orientation channels remain smooth even during the transitions associated with pick-and-place events.

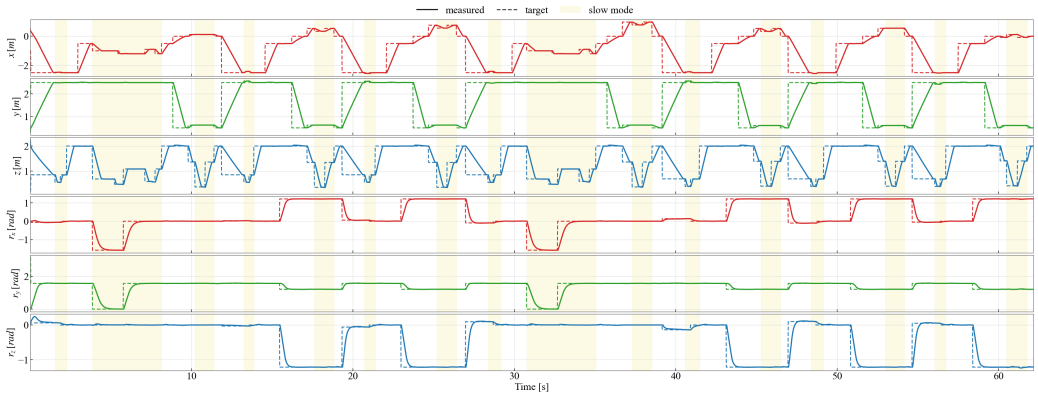
The shorter orders show that the controller can complete compact cycles without oscillations near the grasp and release phases. The more populated orders are more demanding because the same sequence must be repeated many times over a longer horizon, yet the plots still show regular convergence toward the targets and no visible accumulation of tracking errors. These profiles confirm that the state machine, the gain scheduling, and the RMPflow-based policy remain coherent for both simple and dense orders.



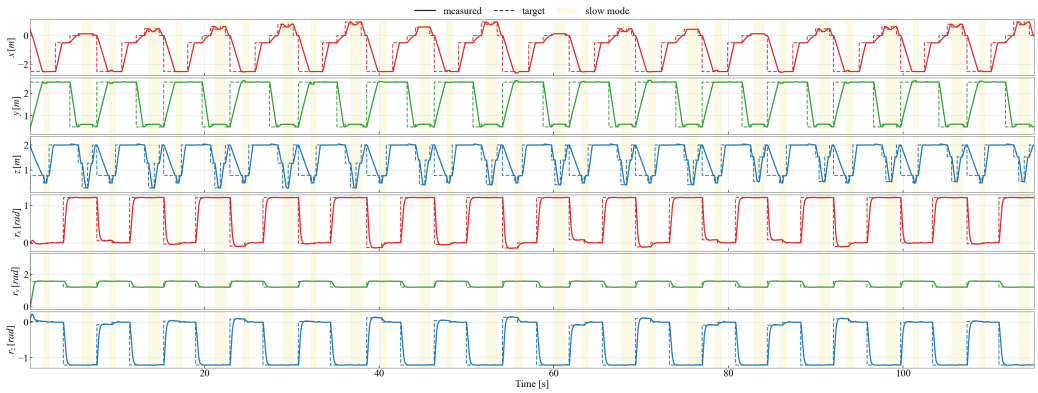
(a) Representative short order with two boxes.



(b) Representative short order with four boxes.



(c) Representative short order with seven boxes.



(d) Representative dense order with fifteen boxes.

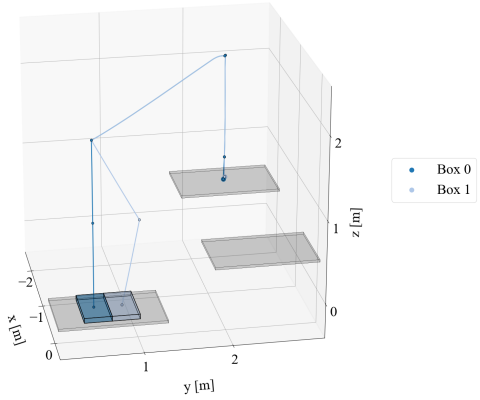
Figure 9.2: Cartesian position and orientation profiles during representative complete palletizing runs.

9.3 Package Trajectories for Representative Orders

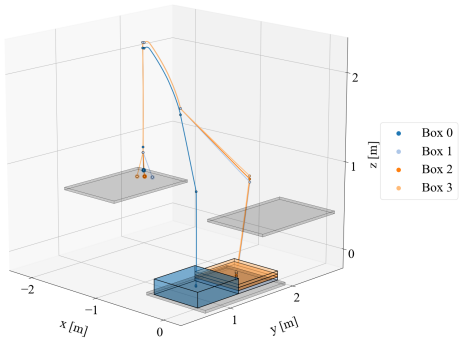
The trajectory plots of Figure 9.4 provide a compact overview of the entire execution of one order in a single image. Each colored line represents the

path followed by one package from its spawn position to the final pallet placement, including the intermediate transfer areas and, when needed, the reorientation table. For this reason, these figures are particularly useful to compare different order structures without replaying the full simulation.

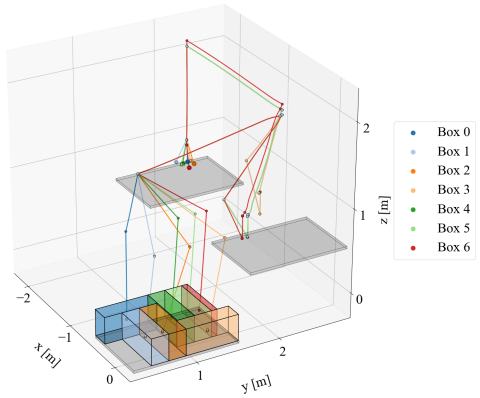
The reported examples highlight that the control framework behaves reliably across different order conditions. In simpler orders with few packages, the trajectories remain direct and regular, and the execution can be completed without requiring reorientation steps. In more populated orders, instead, the paths become richer and may include transitions through the reorientation area before final placement. This shows that the same control architecture can manage both straightforward palletizing sequences and more complex executions in which several packages require orientation adjustment before being deposited on the pallet.



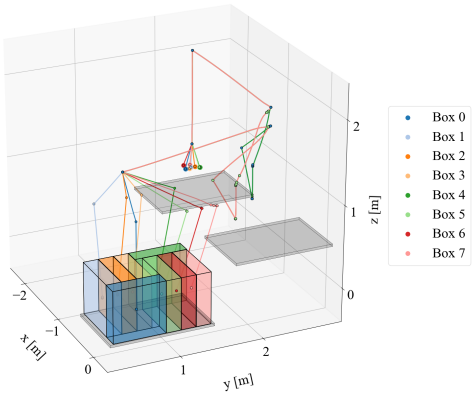
(a) Simple order with direct and regular placement trajectories.



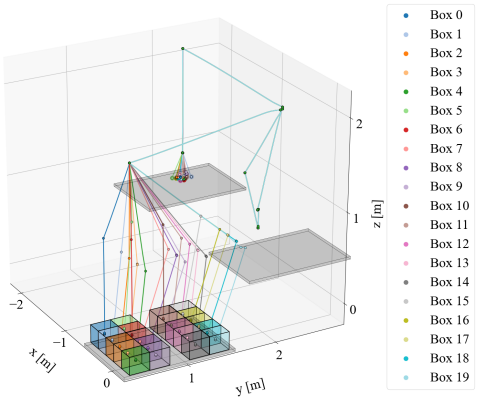
(b) Simple order with compact execution and no reorientation.



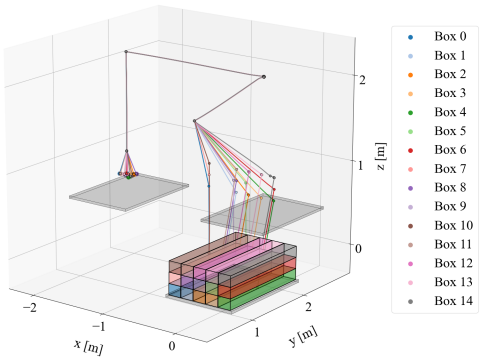
(c) Intermediate order with longer but still regular transfer paths.



(d) Intermediate order with stable transport and placement trajectories.



(e) Dense order with visible transitions through the reorientation area.



(f) Dense order with multiple packages requiring orientation adjustment.

Figure 9.4: Representative package trajectories for simple direct-placement orders and more populated orders requiring reorientation.

9.4 Realistic Visual Results of the Simulation

Beyond trajectory plots and state profiles, the simulation also provides realistic visual feedback of the full palletizing cell. The screenshots reported in Figures 9.6 were captured directly from Isaac Sim during representative executions and illustrate the digital twin under realistic lighting, materials, and scene composition.

These views complement the previous results in an important way. While the plots quantify motion regularity and controller behavior, the screenshots make it possible to assess the spatial coherence of the workstation, the robot posture during manipulation, the package arrangement on the pallet, and the overall realism of the simulated interaction between the manipulator, the working tables, and the handled boxes.

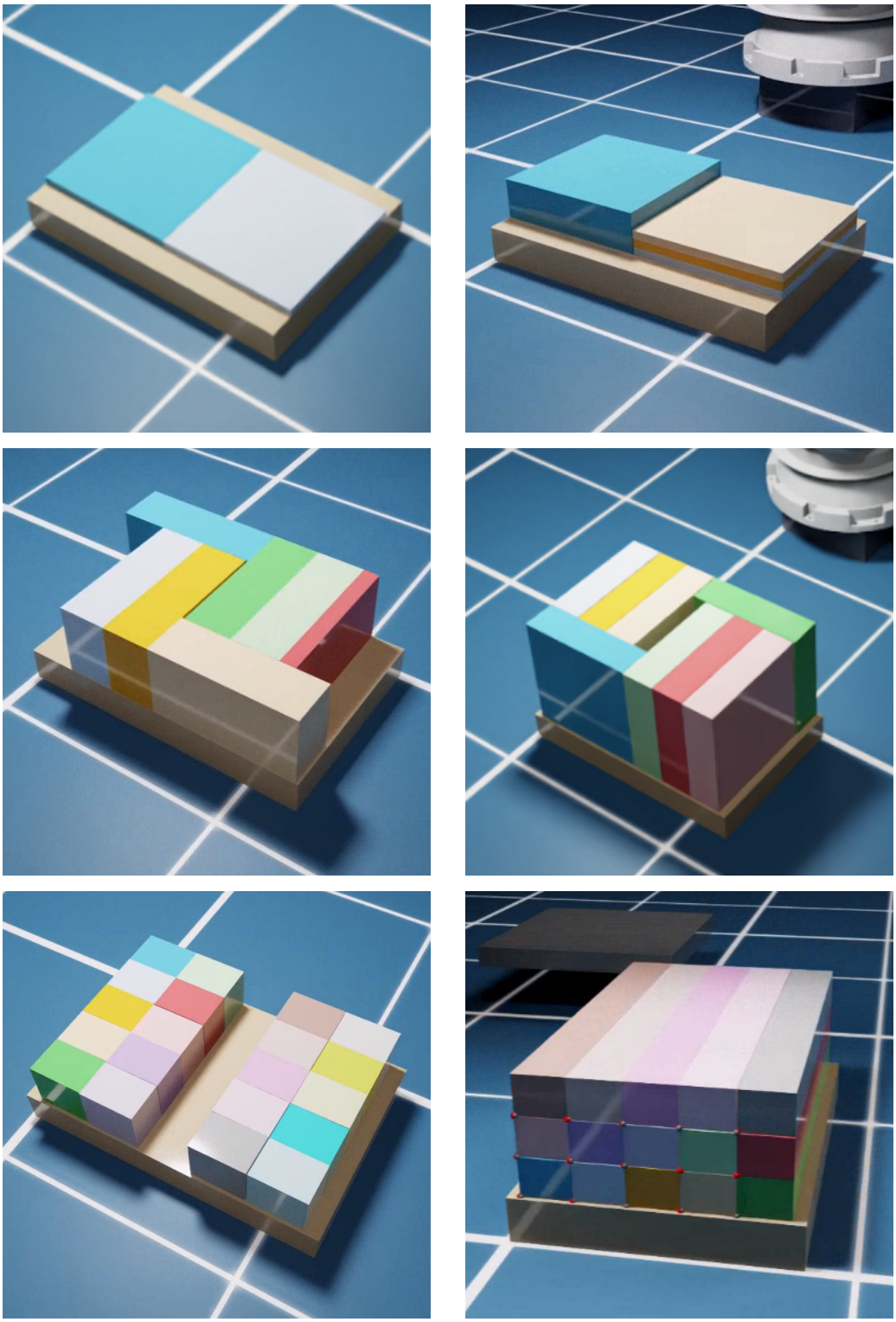


Figure 9.6: Representative realistic capture of the simulated palletizing cell during execution. The views highlight the visual realism of the digital twin and the consistency of the robot behavior across different manipulation situations.

Chapter 10

Conclusion

This thesis presented an execution-oriented simulation framework for robotic palletizing developed in NVIDIA Omniverse Isaac Sim. The core objective was not to solve pallet optimization from scratch, but to study how an externally generated pallet plan can be executed by a robotic manipulator under realistic constraints of sensing, kinematics, contact, and workspace geometry. From this perspective, the work occupies the intermediate layer between high-level packing logic and future industrial deployment.

The main contribution of the thesis is system integration. The developed framework combines JSON-driven order loading, a digital twin of the robotic cell, a surface gripper model, a top-down perception module, a state-machine-based manipulation strategy, an RMPflow-based motion controller, a package reorientation procedure, and a placement strategy that separates layout regularization from directional insertion reasoning. Considered individually, these components are established ideas; considered together, they form a coherent pipeline that transforms packing data into a complete perception-to-placement execution loop.

From an engineering standpoint, the work shows that a physically grounded simulation environment can be used to validate robotic manipulation logic before transfer to real hardware. The system manages randomized package spawning, estimates package pose from RGB images, converts that estimate into robot targets, performs grasping and transport, optionally reorients the package on an intermediate table, and finally places the package on the pallet while accounting for local collision risks. In this sense, the digital twin is not only a visual replica of the workstation, but also a practical development environment for testing interactions between sensing, planning, and contact-rich execution.

The thesis includes both qualitative observations and quantitative results, as discussed in the results chapter through aggregate tracking metrics and execution statistics collected over multiple simulated orders. At the same time, the work does not aim to present a fully exhaustive benchmark campaign. Rather, the reported evidence shows that the proposed modules can operate together in a stable and interpretable way, and that the chosen heuristics are sufficient to execute complete palletizing sequences in the considered setup. This is a meaningful result for a thesis centered on system integration and digital prototyping, while still leaving room for future studies based on broader statistical validation and more extensive comparative testing.

A further contribution lies in the explicit treatment of placement robustness. Instead of assuming that every upstream layout is directly executable, the work introduces a placement-margin regularization stage and a separate directional insertion analysis through `select_direction.py`. This separation makes the overall pipeline easier to interpret, debug, and extend. Likewise, the perception module deliberately privileges transparency and computational simplicity over generality, which is a defensible design choice in a structured industrial scenario.

A playlist collecting representative simulation videos for different palletizing orders is available at:

Playlist Link

These videos further support the qualitative conclusions of the thesis: the simulated system is able to execute different orders, adapt the manipulation sequence to the required orientations and intermediate transfers, and carry the palletizing process to completion across multiple representative scenarios.

Viewed as a whole, the thesis supports the broader idea that simulation-driven development is not merely a convenience but a practical engineering methodology for robotics [9]. A well-constructed digital twin makes it possible to test execution logic, inspect failure modes, and refine manipulation strategies before exposing a physical system to contact-rich tasks such as palletizing. For this reason, the framework also represents a plausible foundation for future sim-to-real transfer, provided that the remaining discrepancies between simulation and hardware are characterized explicitly.

10.1 Future Work

1. Experimental Benchmarking

The most immediate extension of this work is a structured experimental validation campaign. The current framework already contains the elements needed to measure execution success, placement error, tracking quality, and qualitative robustness over multiple orders. Turning these observations into a reproducible benchmark would significantly strengthen the scientific contribution of the project and would make it easier to compare alternative control and placement strategies on a common basis.

2. Learned Perception

A second natural development concerns perception. The current pipeline is well matched to a fixed top-view camera and package-like objects, but it would likely degrade in the presence of strong appearance variability, partial occlusions, or more heterogeneous object classes. For this reason, one promising direction is the integration of learned detection or pose-estimation models, for example YOLO-style detectors or PoseCNN-like architectures, trained on synthetic data generated inside Isaac Sim. Such an extension would preserve the current simulation infrastructure while increasing robustness beyond the structured assumptions adopted in this thesis.

3. Sim-to-Real Transfer

The long-term value of the framework lies in sim-to-real transfer. The digital twin already provides a meaningful environment for validating scene organization, perception geometry, waypoint structure, and manipulation logic. The next step would be to compare the simulated behavior with a real robotic palletizing cell and progressively calibrate the discrepancies in sensing, contact, timing, and gripper behavior. Only through that comparison will it be possible to assess which components of the current pipeline can be transferred directly and which require additional adaptation.

Bibliography

[1] Ching-An Cheng, Max Mühlbauer, Calvin Yang, Perry Liang, J. Zico Kolter, Jun Morimoto, and Byron Boots. Rmpflow: A geometric framework for generation of multi-task motion policies. *IEEE Transactions on Automation Science and Engineering*, 18(3):968–987, 2021.

[2] Samir Elhedhli, Fatma Gzara, and Berk Yildiz. Three-dimensional bin packing and mixed-case palletization. *INFORMS Journal on Optimization*, 1(4):323–352, 2019.

[3] Oussama Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[4] Silvano Martello, David Pisinger, and Daniele Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000.

[5] NVIDIA. Nvidia isaac sim documentation. `https://docs.omniverse.nvidia.com/isaacsim/latest/`. Accessed: 2026-03-13.

[6] OpenCV. Opencv documentation. `https://docs.opencv.org/`. Accessed: 2026-03-13.

[7] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

[8] Pixar Animation Studios. Openusd documentation. `https://openusd.org/release/index.html`. Accessed: 2026-03-13.

[9] Fred J. A. M. van Houten, Fumihiko Kimura, Kentaro Yamazaki, and Rainer Stark. Digital twin driven smart manufacturing. *CIRP Annals*, 69(2):533–556, 2020.

[10] Wenbin Zhu, Ying Fu, and You Zhou. 3d dynamic heterogeneous robotic palletization problem. *European Journal of Operational Research*, 316(2):584–596, 2024.