Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Автоматики і комп'ютеризованих технологій
(повна назва)

Кафедра Комп'ютерно-інтегрованих технологій, автоматизації та робототехніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

перший (бакалаврський)
(рівень вищої освіти)

«Розроблення додатку–тренажера маніпулятора для захоплення об'єктів на конвеєрі»

(тема)

Виконав:
здобувач 4 року навчання,
групи АКТАКІТ–21–3
Максим ДАНИЛЕНКО
(власне ім'я, прізвище)

Спеціальність 151 Автоматизація та комп'ютерно інтегровані технології
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Автоматизація та комп'ютерно інтегровані технології
(повна назва освітньої програми)

Керівник асистент Данило БЛИЗНЮК
(посада, власне ім'я, прізвище)

Допускається до захисту
Зав. кафедри КІТАР _____ Ігор НЕВЛЮДОВ
(підпис) (власне ім'я, прізвище)

2025 р.

Я, Даниленко Максим Маркович, як здобувач вищої освіти ХНУРЕ, розумію і підтримую політику закладу із академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Я не використовував штучний інтелект для підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

"9" червня 2025 р. Максим ДАНИЛЕНКО

Харківський національний університет радіоелектроніки

Факультет Автоматики і комп'ютерно-інтегрованих технологій

Кафедра Комп'ютерно-інтегровані технології, автоматизація та робототехніка

Рівень вищої освіти перший (бакалаврський)

Спеціальність 151 – Автоматизація та комп'ютерно-інтегровані технології
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Автоматизація та комп'ютерно-інтегровані технології
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

« 28 » квітня 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Даниленку Максиму Марковичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення додатку-тренажера маніпулятора для захоплення об'єктів на конвеєрі

Затверджена наказом по університету від 19.05.2025 № 390 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 09.06.2025 р.

3. Вихідні дані до роботи

3.1 Загальна концепція та призначення тренажера

3.2 Архітектура проекту та використані технології

3.3 Опис основних функціональних модулів

3.4 Алгоритм роботи автоматизованої лінії

4. Перелік питань, що потрібно опрацювати в роботі

4.1 Дослідження структури додатків Blender та підключення власних модулів.

4.2 Розробка логіки автоматизованої роботи виробничої лінії на базі Python-скриптів.

4.3 Дослідження та застосування принципів інверсної кінематики для анімації рухів

4.4 Організація взаємодії між модулями

4.5 Оцінка стабільності та відновлення автоматизованої роботи після переривання

4.6 Розглянути питання пов'язані з охороною праці

4.7 Оформлення пояснювальної записки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Демонстраційний матеріал представлений у форматі презентації PowerPoint (*.ppt) – 14 с. формату А4

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Приміт-ка
1	Аналіз і дослідження технологій для реалізації кваліфікаційної роботи	01.05.25	виконано
2	Моделювання необхідних візуальних моделей віртуальної виробничої лінії	11.05.25	виконано
3	Проектування та інтегрування необхідних модулів керування віртуальної виробничої лінії у середовище розробки Blender	03.06.25	виконано
4	Тестування та вдосконалення працездатності віртуальної виробничої лінії	06.06.25	виконано
5	Оформлення пояснювальної записки	09.06.25	виконано
6	Подання роботи на перевірку Інтернет-сервісом StrikePlagiarism	12.06.25	виконано
7	Оформлення пояснювальної записки	12.06.25	виконано
8	Подання роботи на рецензію	13.06.25	виконано
9	Подання роботи на підпис зав. кафедри	15.06.25	виконано
11	Подання кваліфікаційної роботи в ЕК	18.06.25	виконано

Дата видачі 28.04.25

Здобувач _____ Максим ДАНИЛЕНКО
(підпис) (власне ім'я, прізвище)

Керівник роботи _____ асистент Данило БЛИЗНЮК
(підпис) (власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 92 с., 4 табл., 65 рис., 26 джерел.

BLENDER, PYTHON, ІНВЕРСНА КІНЕМАТИКА, ВІРТУАЛЬНИЙ ТРЕНАЖЕР, МАНІПУЛЯТОР, КОНВЕЄР, ЗАХОПЛЕННЯ ОБ'ЄКТІВ, АВТОМАТИЗАЦІЯ, МОДЕЛЮВАННЯ, СИМУЛЯЦІЯ, РОЗЛИВ РІДИНИ, ПАКУВАННЯ

Об'єкт розробки – процес автоматизованого виконання виробничих операцій, зокрема захоплення, переміщення та укладання об'єктів маніпулятором, інтегрованим у конвеєрну систему.

Предмет розробки – програмна реалізація логіки керування та візуалізації роботи маніпулятора у віртуальному тренажері на базі Blender для моделювання повного виробничого циклу.

Метою даної кваліфікаційної роботи є покращення доступності, зрозумілості та гнучкості навчального процесу у сфері автоматизованого управління та робототехніки за рахунок розробки інтерактивного віртуального тренажера, який моделює повний цикл виробничого процесу – від надходження тари до вивезення продукції – із візуалізацією роботи маніпулятора, логіки автоматизованого захоплення об'єктів та його взаємодії з конвеєрною системою.

Область застосування – розроблений віртуальний тренажер призначений для використання у навчальному процесі з підготовки фахівців у галузі автоматизованого управління та робототехніки. Він дозволяє студентам у доступній формі вивчати повний цикл виробничого процесу, включаючи взаємодію маніпулятора з конвеєрною системою, логіку автоматизованого захоплення об'єктів та послідовність технологічних операцій. Тренажер може бути використаний як у класичному аудиторному навчанні, так і для самостійної роботи або дистанційних курсів.

ABSTRACT

Explanatory note: 92 pp., 4 table, 65 figures, and 26 references.

BLENDER, PYTHON, INVERSE KINEMATICS, VIRTUAL TRAINER, MANIPULATOR, CONVEYOR, OBJECT GRIPPING, AUTOMATION, SIMULATION, MODELING, LIQUID FILLING, PACKAGING

The object of development – the process of automated execution of production operations, including the grasping, moving, and placing of objects by a manipulator integrated into a conveyor system.

The subject of development – the software implementation of control logic and visualization of manipulator operation in a virtual training simulator based on Blender for modeling the complete production cycle.

The aim of this qualification work is to improve the accessibility, clarity, and flexibility of the educational process in the field of automated control and robotics by developing an interactive virtual training simulator that models the full production cycle – from container supply to product dispatch – with visualization of manipulator operation, automated object grasping logic, and its interaction with a conveyor system.

Application area – the developed virtual simulator is intended for use in the educational process for training specialists in the field of automated control and robotics. It enables students to study the full production cycle in an accessible format, including the manipulator's interaction with the conveyor system, automated object grasping logic, and the sequence of technological operations. The simulator can be used both in traditional classroom settings and for self-study or distance learning courses.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

КІТАР – комп’ютерно–інтегрованих технологій, автоматизації та робототехніки;

НДР – науково–дослідна робота;

ПЗ – програмне забезпечення;

ХНУРЕ – Харківський національний університет радіоелектроніки.

ROS – Robot Operating System

API – application programming interface

RTS – Robotics System Toolbox

SCADA – Supervisory Control and Data Acquisition

ІК – інверсна кінематика

ВСТУП

У сучасних умовах динамічного розвитку автоматизованих систем керування та робототехнічних комплексів спостерігається зростання потреби у засобах моделювання, тестування та навчання, орієнтованих на взаємодію технічних пристроїв у віртуальному середовищі. Серед таких засобів особливу увагу привертають роботизовані маніпулятори, які знаходять широке застосування в промислових процесах, системах логістики, а також у сфері наукових досліджень. Для підвищення ефективності їх використання необхідним є створення інструментів, що дозволяють здійснювати попередню візуалізацію та перевірку алгоритмів взаємодії з об’єктами та конвеєрними системами.

Застосування програмних тренажерів надає можливість проводити повноцінне тестування керуючих алгоритмів без необхідності використання фізичного обладнання, що значно знижує витрати на дослідницькі етапи. Крім того, віртуальні моделі є ефективним інструментом для навчання фахівців, які працюють у галузі мехатроніки, автоматизації та робототехніки. Це відкриває додаткові можливості для підвищення якості підготовки персоналу, який здатен працювати з подібними системами у реальних виробничих умовах.

Суттєвим фактором у створенні подібних програмних рішень є досвід розробки та аналізу SCADA-систем, які надають потужні засоби для візуалізації процесів, збору даних та управління у реальному часі. В межах попереднього дослідження було проведено порівняльний аналіз функціональних можливостей сучасних SCADA-пакетів, зокрема InTouch, iFix та WinCC [1], що дозволило окреслити низку принципів, які можуть бути ефективно перенесені у віртуальні тренажери: модульність, гнучкість архітектури, візуальна наочність, масштабованість та підтримка різноманітних сценаріїв керування. Отримані результати стали концептуальним підґрунтям при формуванні вимог до створення програмного середовища, здатного імітувати роботу маніпулятора у синхронізації з конвеєрною системою.

Розроблення програмного забезпечення у вигляді віртуального тренажера маніпулятора сприяє глибокому дослідженню процесів захоплення об’єктів, їх транспортування та синхронізації з конвеєрними лініями. Подібний підхід дозволяє оптимізувати логіку управління, виявити потенційні помилки або неузгодженості в алгоритмах, а також удосконалити архітектуру системи ще до її фізичного впровадження.

Метою даної кваліфікаційної роботи є покращення доступності, зрозумілості та гнучкості навчального процесу у сфері автоматизованого управління та робототехніки за рахунок розробки інтерактивного віртуального тренажера, який моделює повний цикл виробничого процесу – від надходження тари до вивезення продукції – із візуалізацією роботи маніпулятора, логіки автоматизованого захоплення об’єктів та його взаємодії з конвеєрною системою.

Об’єкт розробки – процес автоматизованого виконання виробничих операцій, зокрема захоплення, переміщення та укладання об’єктів маніпулятором, інтегрованим у конвеєрну систему.

Предмет розробки – програмна реалізація логіки керування та візуалізації роботи маніпулятора у віртуальному тренажері на базі Blender для моделювання повного виробничого циклу.

Для досягнення поставленої мети передбачено виконання таких завдань:

- провести аналітичний огляд існуючих рішень у галузі програмних тренажерів для роботизованих систем;
- дослідити принципи функціонування маніпуляторів та конвеєрних систем;
- обґрунтувати вибір середовища розробки та допоміжних інструментів;
- побудувати архітектуру програмного забезпечення та розробити його інтерфейс;
- реалізувати функціонал симуляції руху, захоплення та сортування об’єктів;
- провести тестування роботи додатку та здійснити аналіз ефективності розробленої системи.

Кваліфікаційна робота виконана згідно з вимогами ДСТУ 3008 – 15, а також

на основі методичних вказівок [2] та навчального посібника з дипломного проектування [3].

1 АНАЛІЗ ІСНУЮЧИХ СИСТЕМ ТРЕНУВАННЯ РОБОТ – МАНІПУЛЯТОРІВ

1.1 Огляд програм–симуляторів маніпуляторів

У сфері робототехніки та автоматизованих систем важливою складовою є створення та тестування віртуальних моделей роботів до їхньої фізичної реалізації. Це дозволяє попередньо оцінити ефективність алгоритмів керування, перевірити кінематику, динаміку та взаємодію з об’єктами довкілля. Одним із популярних інструментів для цього є MATLAB/Simulink, що надає спеціалізований інструментарій – Robotics System Toolbox. Також розглянемо Gazebo + ROS та V–REP/CoppeliaSim. Спочатку розглянемо можливості MATLAB/Simulink, переваги та обмеження в контексті моделювання роботів–маніпуляторів.

MATLAB/Simulink Robotics System Toolbox (RST) – це потужний інструмент для моделювання, симуляції та розгортання робототехнічних систем, зокрема маніпуляторів і мобільних роботів (рис. 1.1). Він забезпечує інтегроване середовище для розробки, що поєднує в собі алгоритми, візуалізацію та можливості генерації коду.

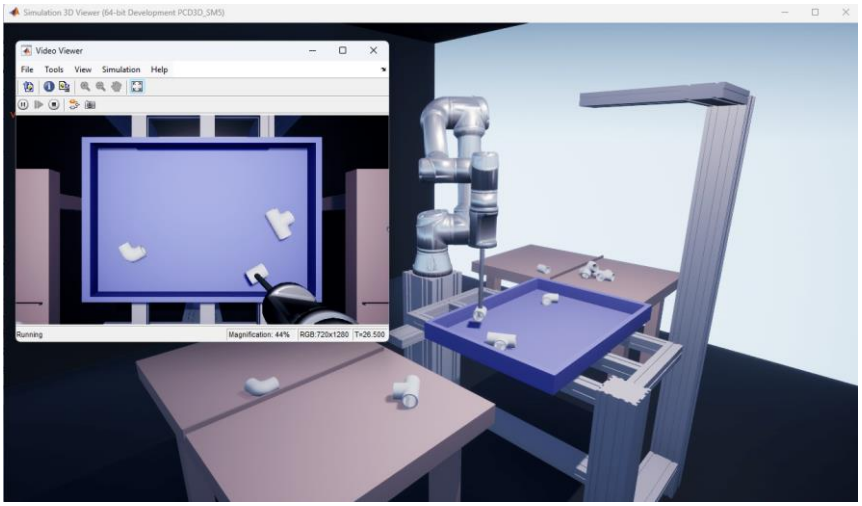

Рисунок 1.1 – Перегляд 3D симуляції маніпулятора [4]

RST дозволяє створювати точні моделі роботів за допомогою об'єкта rigidBodyTree, який представляє структуру робота як дерево жорстких тіл. Це дає змогу виконувати обчислення прямої та зворотної кінематики, аналізувати динамічну поведінку системи та планувати траєкторії руху. Інструменти для перевірки зіткнень, планування шляхів і генерації траєкторій забезпечують гнучкість у розробці складних алгоритмів.

Користувачі можуть імпортувати моделі роботів у форматі URDF або створювати власні моделі, використовуючи Simscape Multibody. Це спрощує процес моделювання та дозволяє швидко переходити від концепції до реалізації (рис. 1.2).

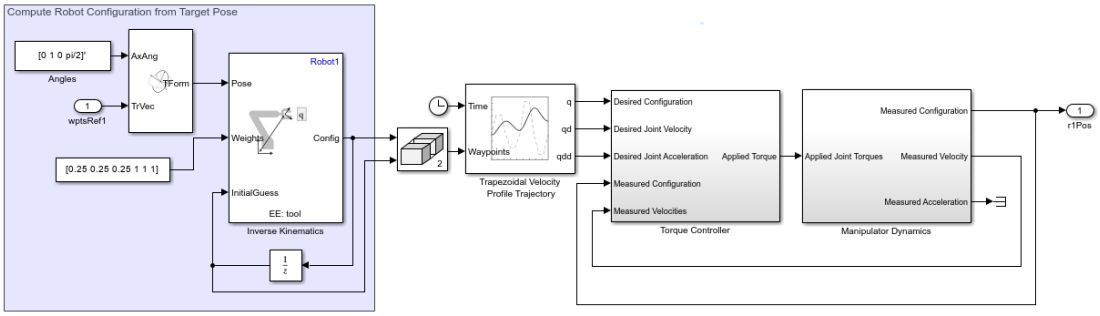

Рисунок 1.2 – Контроль динаміки та кінематики маніпулятора [5]

Крім того, за допомогою MATLAB Coder та Simulink Coder можна генерувати C/C++ код для розгортання на апаратних платформах, таких як Kinova Gen3 та Universal Robots UR серії.

Переваги:

– поєднання MATLAB і Simulink забезпечує зручність у розробці та тестуванні алгоритмів;

– можливість створення та імпорту моделей роботів спрощує процес розробки;

– інтеграція з Gazebo та Unreal Engine дозволяє тестувати системи в реалістичних умовах;

– автоматичне створення коду для розгортання на апаратних платформах прискорює процес впровадження.

Недоліки:

– освоєння всіх можливостей RST може вимагати значного часу та зусиль;

– RST є пропрієтарним програмним забезпеченням, що передбачає необхідність придбання комерційної ліцензії для повноцінного використання;

– спільне моделювання (ко-симуляція) може вимагати специфічного налаштування системного середовища та встановлення додаткових програмних компонентів.

Загалом, MATLAB/Simulink Robotics System Toolbox є потужним інструментом для розробки та тестування робототехнічних систем, що забезпечує гнучкість і ефективність у процесі розробки. Однак, варто враховувати вартість та потенційні складнощі при освоєнні цього середовища [6-8].

Gazebo у зв'язці з ROS (Robot Operating System) є одним із найпотужніших і найпопулярніших інструментів для моделювання робототехнічних систем у реальному часі. У своїй суті Gazebo є 3D–симулятором, який дозволяє створювати реалістичне середовище з фізикою, світлом, колізіями та текстурами. Він використовується для відлагодження поведінки роботів ще до їхнього запуску у фізичному світі (рис. 1.4). У свою чергу, ROS – це фреймворк, який надає гнучку інфраструктуру для розробки програмного забезпечення для роботів, зокрема публікацію та підписку на повідомлення, роботу з сервісами, контролерами, сенсорами тощо.

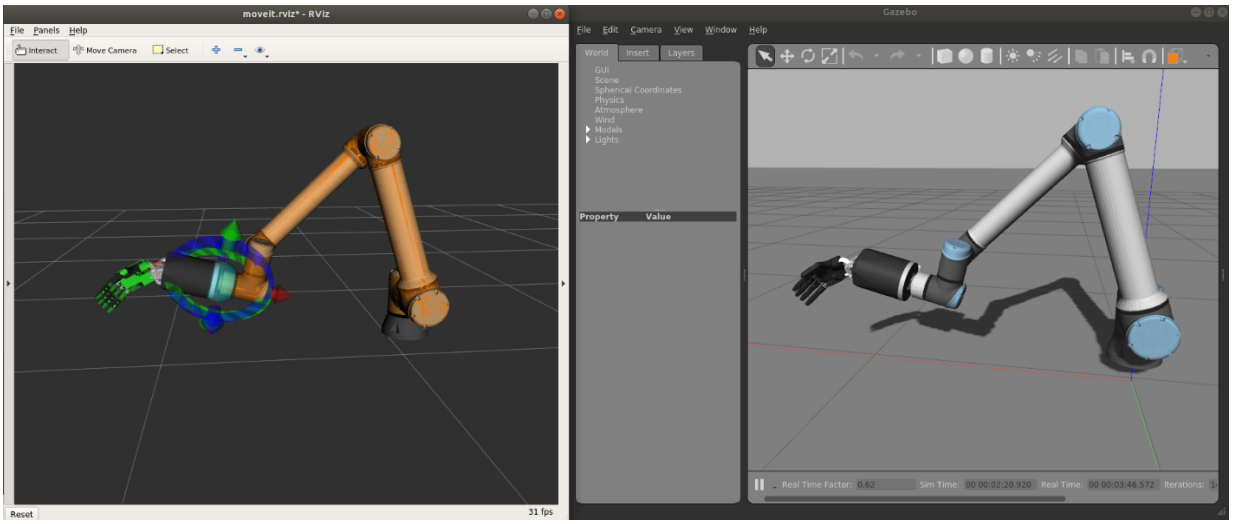

Рисунок 1.3 – Симуляція поведінки маніпулятора у середовищі Gazebo [9]

Комбінація Gazebo з ROS дозволяє розробникам створювати не просто віртуальні макети роботів, а повноцінні системи, які поводяться максимально наближено до реальності. У Gazebo можна змоделювати самого робота (рис. 1.5), наприклад маніпулятор або мобільну платформу, вказати його геометрію, масу, інерцію та точки з'єднання.

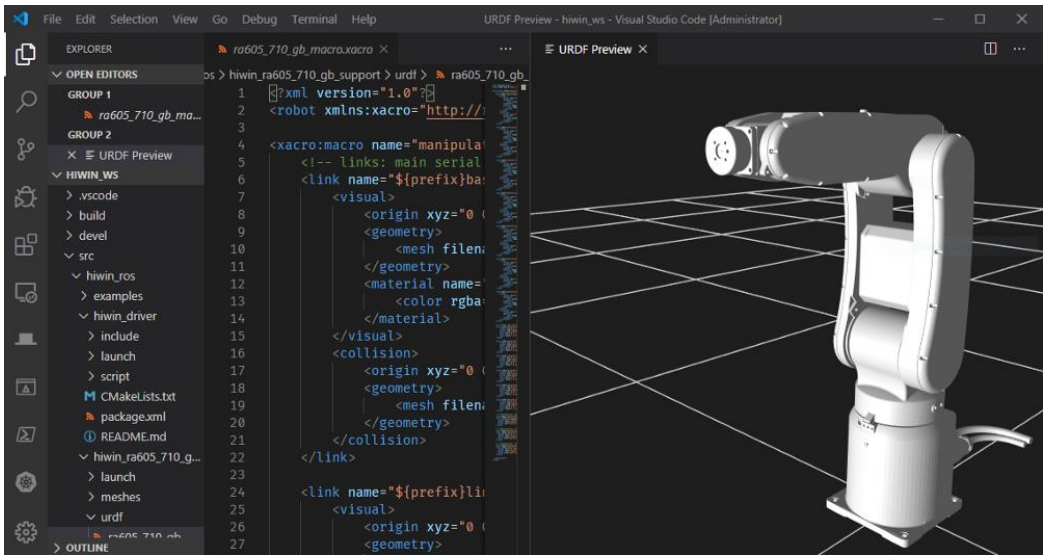

Рисунок 1.4 – Вікно Visual Code для URDF Preview налаштування ROS [10]

Також легко додаються різноманітні сенсори – камери, лідáри, IMU, ультразвукові датчики – які у симуляції працюють так само, як і в реальних роботах. ROS забезпечує інтеграцію цієї симуляції з програмною логікою: він приймає сигнали від сенсорів, обробляє їх через алгоритми контролю, й повертає команди до Gazebo у вигляді рухів, дій чи зворотних ефектів.

Однією з головних переваг цієї зв’язки є підтримка реального часу. Це означає, що симуляція може бути синхронізована зі справжнім часом або працювати в прискореному/сповільненому режимі, що дуже зручно для тестування довготривалих сценаріїв. Наприклад, можна симулювати переміщення маніпулятора вздовж конвеєра, спостерігаючи, як змінюються дані сенсорів, і як програмний модуль реагує на зміну положення об’єктів.

Ще одним плюсом є гнучкість налаштування. Усе описується у форматах URDF або SDF, які легко редагуються. Крім того, система підтримує численні

плагіни, які додають нову функціональність без потреби переписувати ядро Gazebo. Це може бути, наприклад, плагін керування диференціальними колесами, обробка зображень з камери, або модуль, що імітує вплив шуму у вимірах сенсорів. ROS–інтерфейс дозволяє з'єднати такі модулі у складні ланцюжки взаємодії.

Також важливо згадати про величезну спільноту користувачів, яка постійно ділиться прикладами, шаблонами роботів, параметрами симуляції та готовими world–сценами. Наприклад, існує величезна база open–source моделей: UR5, TurtleBot, Husky, та багато інших. Багато виробників роботів випускають офіційні пакети з підтримкою Gazebo+ROS.

Втім, ця система не позбавлена недоліків. Насамперед слід відзначити доволі високий поріг входу. Інсталяція та налаштування зв'язки Gazebo з ROS потребує глибшого розуміння як файлової структури ROS, так і структури world–файлів у Gazebo. У новачків часто виникають проблеми зі сумісністю версій (наприклад, ROS Noetic не завжди стабільно працює з Gazebo 11 без додаткових патчів). Також можуть з'являтися труднощі при інтеграції сторонніх моделей, зокрема коли URDF–файли містять нестандартні елементи або коли Gazebo не підтримує специфічні типи з'єднань .

Ще один важливий нюанс – обмеження продуктивності. Попри те, що Gazebo є досить потужним рушієм, при симуляції складних сцен із десятками об'єктів і великою кількістю сенсорів можуть виникати проблеми із швидкодією, фрізи або втрати кадрів. Це особливо критично для задач, де важлива висока частота оновлення (наприклад, точне позиціювання або робота із зворотним зв'язком).

Не менш вагомим є питання обмеженої графіки. У порівнянні з ігровими рушіями на кшталт Unreal Engine чи Unity, Gazebo виглядає менш привабливо у візуальному плані. Його графіка функціональна, але доволі базова, що іноді обмежує сферу використання, наприклад, у презентаційних проектах або під час розробки інтерфейсів доповненої реальності.

Попри ці недоліки, Gazebo у поєднанні з ROS залишається найбільш збалансованим безкоштовним рішенням для симуляції робототехніки, яке можна використовувати як для навчання, так і для наукових досліджень або підготовки

програмного забезпечення до реального розгортання. Його гнучкість, розширюваність і підтримка ROS–екосистеми дозволяють створювати масштабовані, адаптивні симуляції – від простого маніпулятора до складної групи автономних роботів [11-14].

CoppeliaSim (раніше відомий як V–REP) – це потужне інтегроване середовище для моделювання, візуалізації, контролю та автоматизації робототехнічних систем (рис. 1.6). Його відрізняє модульна архітектура та унікальний підхід до симуляції, що дозволяє реалізовувати навіть найскладніші сценарії навчання та тестування роботів. Завдяки мультиагентній симуляції, система дозволяє одночасно керувати кількома незалежними об'єктами, які можуть взаємодіяти один з одним у режимі реального часу.

На відміну від інших систем, які в основному орієнтовані на фізичну точність або графіку, CoppeliaSim надає баланс між швидкістю, гнучкістю сценаріїв та інтеграцією з зовнішніми API. Він підтримує багатомовне програмування, включаючи LUA (вбудований), Python, Java, C/C++, Matlab, Octave та інші мови через віддалені API або ROS–інтерфейси. Це дозволяє інтегрувати його з іншими симуляторами або системами керування.

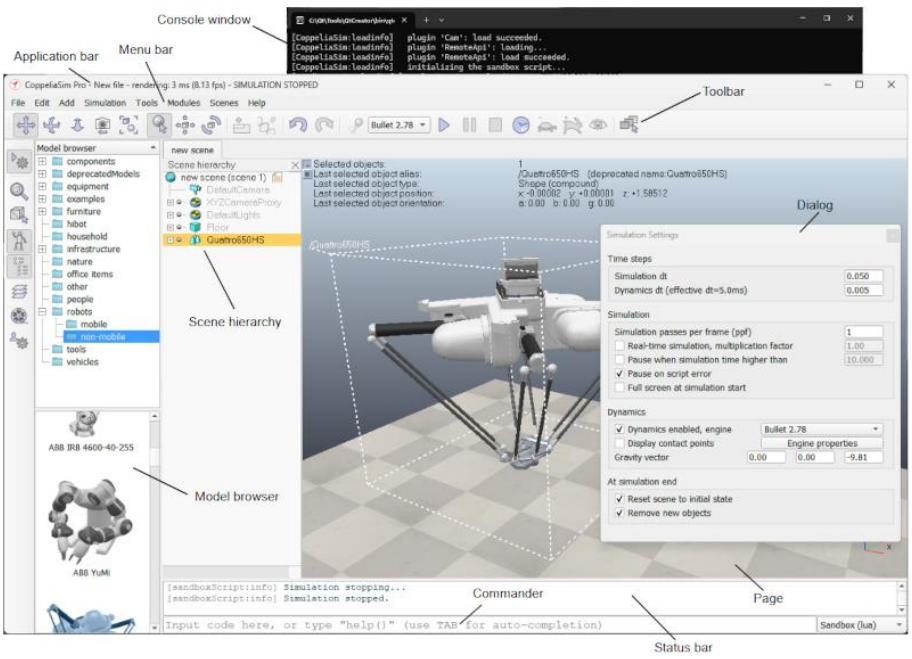

Рисунок 1.5 – Вікно середовища CoppeliaSim [15]

Однією з ключових можливостей CoppeliaSim є використання “scene hierarchy” – ієрархічної структури об'єктів, де кожен об'єкт (робот, сенсор, актор, поверхня, камера тощо) є окремим вузлом, який можна переміщати, клонувати або зв'язувати з іншими елементами. Це забезпечує високий ступінь гнучкості при створенні навчальних або виробничих сценаріїв. Наприклад, можна змоделювати взаємодію маніпулятора з конвеєром, де сенсори відслідковують появу об’єктів, а скрипти LUA контролюють момент захоплення.

Іншим важливим аспектом є візуалізація. CoppeliaSim забезпечує реалістичну графіку з динамічними тінями, освітленням і симуляцією камери, що дозволяє не лише бачити поведінку роботів, але й реалізовувати симуляцію сенсорів, наприклад камери або лідара. Дані з цих сенсорів можна використовувати для навчання моделей машинного навчання або перевірки алгоритмів обробки зображень у реальному часі.

Платформа також підтримує фізичні рушії, такі як Bullet, ODE, Vortex, Newton, що дозволяє обирати між швидкістю та фізичною точністю в залежності від задачі. Наприклад, для тренування нейромереж можна використовувати швидкий рушій з простими колізіями, а для прецизійного тестування – більш точний з урахуванням сили, тертя, маси та моментів інерції.

Переваги:

– повна гнучкість сценаріїв – можливість реалізовувати як прості, так і складні інтерактивні сцени з елементами ШІ;

– вбудоване середовище програмування – LUA–скрипти дозволяють реалізовувати логіку прямо всередині симуляції;

– мультиагентна симуляція – декілька роботів або систем можуть взаємодіяти в одній сцені незалежно;

– API на багатьох мовах – легка інтеграція з Python, ROS, MATLAB, Java та іншими системами;

– можливість запису та відтворення сценаріїв – підходить для навчання та повторного тестування;

– платформа незалежна – працює на Windows, Linux та macOS.

Недоліки:

– висока складність для новачків – ієрархія сцен, скрипти, логіка взаємодії потребують певного досвіду;

– пропрієтарна модель – безкоштовна версія обмежена функціоналом, для повного доступу потрібна ліцензія (особливо для використання в комерційних цілях);

– іноді перевантажений інтерфейс – складні сцени можуть уповільнювати навігацію або викликати зависання;

– менш активна спільнота, ніж у Gazebo або ROS – важче знайти конкретні приклади або швидку допомогу.

У підсумку, CoppeliaSim – це універсальний інструмент для проектування робототехнічних систем і навчання алгоритмів, який пропонує безліч можливостей для розробників, дослідників і студентів. Його сильна сторона – сценарії навчання, завдяки яким можна моделювати цілі курси дій з логікою, затримками, сенсорними подіями та зворотним зв'язком [16-17].

Порівняння середовищ можна представити вигляді табл. 1.1.

Таблиця 1.1 – Порівняння середовищ для моделювання робототехнічних систем

Критерій	MATLAB/Simulink Robotics System Toolbox	Gazebo + ROS	CoppeliaSim
1	2	3	4
Основне призначення	Моделювання, аналіз, симуляція та розгортання робототехнічних систем	Реалістична симуляція роботів з інтеграцією ROS	Візуалізація, сценарії навчання, мультиагентна симуляція

Продовження таблиці 1.2

1	2	3	4
Підтримка ROS	Часткова (через ROS Toolbox)	Повна інтеграція	Часткова (через плагіни)
Фізичні рушії	Simscape Multibody	ODE, Bullet, DART, Simbody	Bullet, ODE, Vortex, Newton
Підтримка сенсорів	Так (через моделі та симуляцію)	Так (через додатки)	Так (вбудовані моделі сенсорів)
Графіка та візуалізація	Обмежена (переважно 2D/3D графіки)	Реалістична 3D–графіка	Реалістична 3D–графіка
Підтримка реального часу	Так	Так	Так
Ліцензування	Комерційна ліцензія	Відкрите програмне забезпечення	Безкоштовна для освіти, комерційна ліцензія
Критерій	MATLAB/Simulink Robotics System Toolbox	Gazebo + ROS	CoppeliaSim
Платформи	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS
Крива навчання	Помірна	Висока	Помірна
Інтеграція з апаратним забезпеченням	Так (через підтримку конкретних платформ)	Так (через ROS)	Обмежена
Підтримка спільноти	Висока (MathWorks)	Дуже висока (відкрита спільнота ROS)	Помірна

1.2 Можливості Blender для моделювання та контролю роботів

Blender – це потужне та безкоштовне програмне забезпечення з відкритим кодом, яке широко використовується для 3D–моделювання (рис. 1.7), анімації та візуалізації. Хоча спочатку Blender не був призначений для робототехніки, його гнучкість та розширюваність дозволили інтегрувати інструменти для моделювання та контролю роботів.

Рисунок 1.7 – Можливості Blender у моделюванні роботів [18]

Blender є надзвичайно потужним середовищем для створення тривимірних моделей, і ця функціональність ідеально підходить для розробки роботизованих систем (рис. 1.8).

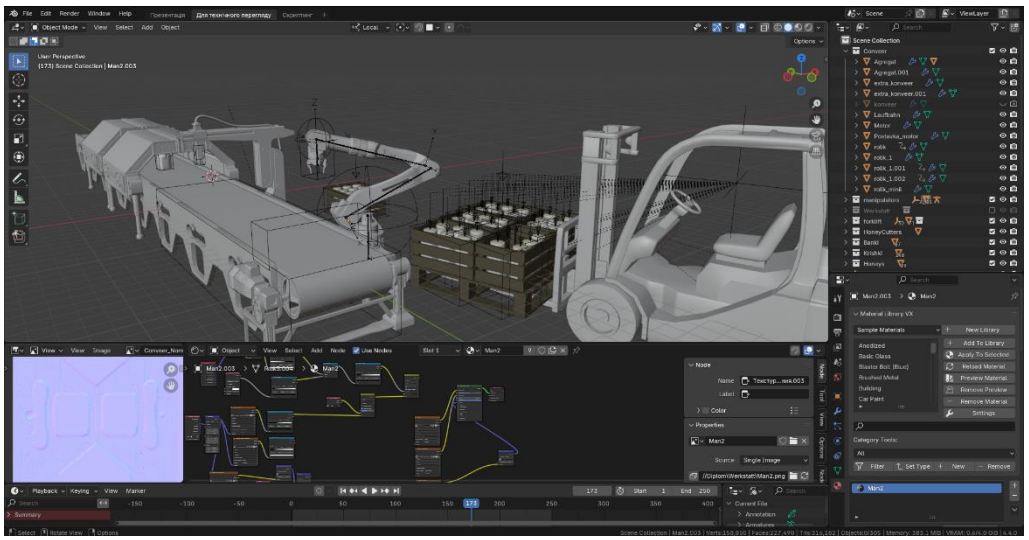

Рисунок 1.8 – Вікно середовища Blender з відкритим проектом

За допомогою Blender можна моделювати кожен компонент робота окремо – основу, суглоби, приводи, датчики, захоплювачі – та об'єднувати їх у єдину структуру з точною геометрією та масштабом. Важливою перевагою є наявність високоточних інструментів побудови, таких як Boolean–операції, криві Bézier, симетричне дзеркалювання (Mirror), модифікатори для генерації складних форм (Subdivision Surface, Solidify) тощо.

Особливу цінність для робототехнічного моделювання має система ригінгу (rigging), яка дозволяє створювати кістяки для об'єктів, подібно до того, як створюють скелети персонажів у комп'ютерній графіці. Для маніпуляторів це означає можливість задати зв'язки між ланками та створити структуру, що імітує реальні ступені свободи (DOF). У комбінації з системою інверсної кінематики (IK), користувач може керувати не окремими ланками, а кінцевим ефектором (наприклад, захоплювачем), задаючи йому цільову точку у просторі (рис. 1.9). Blender автоматично розраховує положення суглобів для досягнення цієї точки, що надзвичайно спрощує створення рухів і симуляцію сценаріїв захоплення об'єктів.

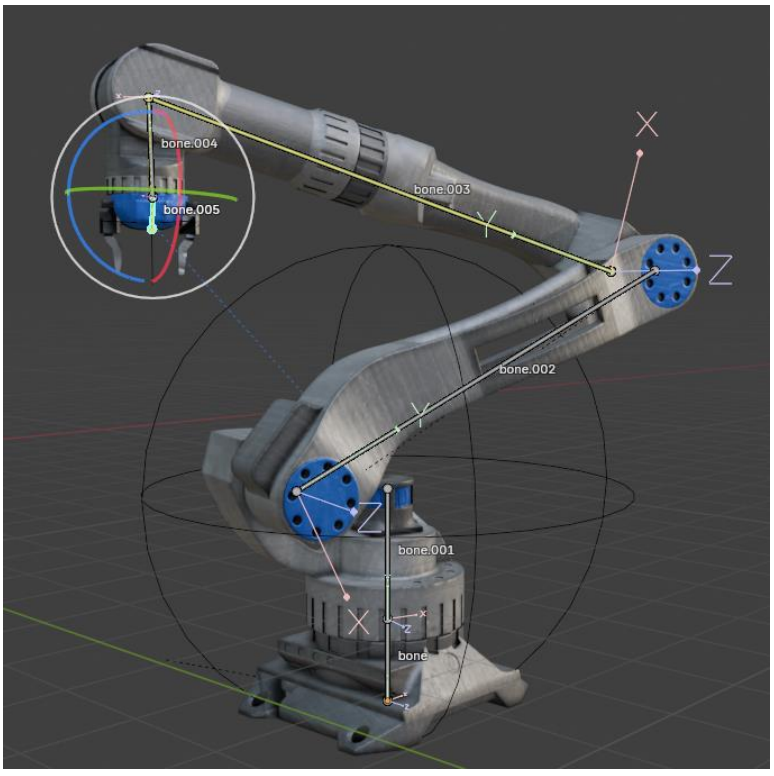

Рисунок 1.9 – Скелетування за допомогою кісток та інверсної кінематики

Blender також підтримує анімацію на основі ключових кадрів (keyframes), що дозволяє створювати плавні послідовності руху маніпулятора. Це корисно не лише для візуалізації, але й для тестування логіки руху, перевірки плавності переходів, оцінки досяжності позицій. Такі анімації можна виводити як відео (рис. 1.10), використовувати для навчальних демонстрацій або експортувати координати суглобів для подальшого використання в програмному коді реального робота.

Рисунок 1.10 – Відображення ключових кадрів на часовій шкалі проекту

Фізичні симуляції у Blender базуються на рушії Bullet Physics, який підтримує моделювання жорстких тіл (rigid bodies), м’яких тіл (soft bodies), тканини, рідин і колізій. У контексті робототехніки особливо важливо те, що кожен об’єкт у сцені може бути призначений як активний чи пасивний учасник фізичної взаємодії. Це дозволяє симулювати падіння, пересування по конвеєру, зіткнення з перешкодами, силу тяжіння та інші впливи.

Для маніпуляторів, наприклад, це дозволяє перевірити, чи зможе робот захопити об’єкт без його зсування, як змінюється траєкторія при зіткненні, чи стабільно утримується вантаж. Крім того, можливе створення складних сценаріїв (рис. 1.11), наприклад, падіння об'єкта на конвеєрі з його подальшим

перехопленням маніпулятором. Фізика працює у реальному часі або з прискоренням, що робить Blender зручним інструментом для початкової симуляції.

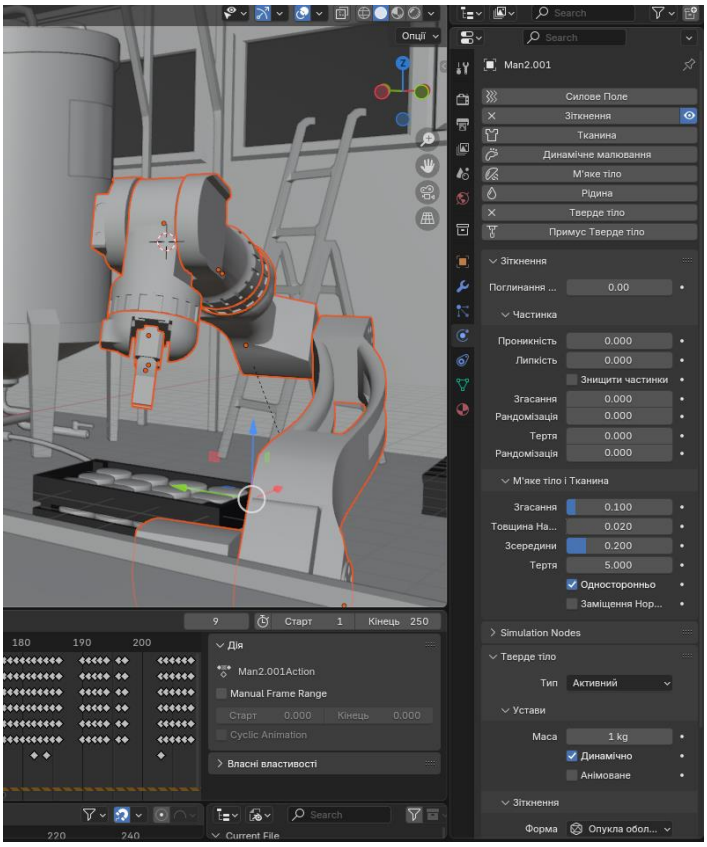

Рисунок 1.11 – Вікно налаштування фізичних властивостей

Blender має два вбудованих рушії для рендерингу: Cycles (фотореалістичний) та Eevee (реального часу). Обидва підтримують глобальне освітлення, реалістичні матеріали, відблиски, тіні, а також візуалізацію зі складними ефектами. Це дозволяє створювати надзвичайно привабливі сцени, які можна використовувати як для презентацій проектів, так і для навчання, симуляції або демонстрації поведінки роботів в різних умовах.

Користувач може створити сцени, які симулюють цех, конвеєрну лінію, лабораторію або будь–яке інше виробниче середовище. Завдяки камерам, що розміщуються в просторі, можна отримати візуалізацію з різних кутів, включаючи перспективу самого робота або оператора. Це особливо корисно для імітації зору робота, тестування алгоритмів розпізнавання образів або створення датасетів [19] .

1.3. Огляд існуючих додатків для Blender

Інтеграція Blender з зовнішніми інструментами: від анімації до керування реальним роботом

Blender, хоч і не створений спеціально для робототехніки, має надзвичайно гнучку архітектуру та потужний API на мові програмування Python. Саме завдяки цьому він може використовуватись не лише для 3D–моделювання, а й як повноцінне середовище для візуального програмування та симуляції керування роботами.

У Blender все – від руху об'єктів до зміни матеріалів або реакції на події – можна контролювати через Python. Це дозволяє створювати скрипти автоматизації, де, наприклад, маніпулятор рухається вздовж заданої траєкторії, реагує на появу об’єкта або змінює свою поведінку в залежності від параметрів середовища. Такі скрипти можуть бути простою заміною або навіть прототипом ПЗ для реального контролера.

Blender дозволяє зчитувати координати, швидкості, кути обертання (Euler або Quaternion), а також відстежувати взаємодії з об’єктами у сцені (зіткнення, торкання, наближення тощо).

Приведемо приклад, який ілюструє практичне використання Blender для візуального створення траєкторії руху маніпулятора з подальшим експортом координат для використання у реальному роботі:

Приклад: Створення та експорт траєкторії руху маніпулятора в Blender Розробити базовий рух маніпулятора, який переміщає захоплювач із початкової точки в простір до цільового об’єкта (наприклад, куба на конвеєрі), та експортувати координати цього руху у файл для подальшого використання у ROS–контролері.

Завдання: Розробити базовий рух маніпулятора, який переміщає захоплювач із початкової точки в простір до цільового об’єкта (наприклад, куба на конвеєрі), та експортувати координати цього руху у файл для подальшого використання у ROS–контролері.

Кроки: Створення моделі: У Blender моделюється спрощений маніпулятор з

3 або 4 ланками (armature + mesh). Кожна ланка прив'язується до кістки (Bone), формуючи ієрархію суглобів.

Ригінг та інверсна кінематика: Застосовується Inverse Kinematics (IK) до кінцевого ефектора (захоплювача), що дозволяє задавати рух лише однієї точки – а Blender автоматично розраховує, як мають змінюватися кути обертання в суглобах.

Анімація: За допомогою ключових кадрів створюється анімація, у якій маніпулятор переміщується від початкової позиції до точки над кубом, опускається, затискає його (імітація), і повертається назад.

Python–скрипт для експорту: Створюється скрипт, який зчитує положення кінцевого ефектора в кожен кадр анімації, і записує ці дані у .csv (рис. 1.13).

Отриманий .csv файл імпортується у Python–скрипт під ROS, який через контролери подає ці координати як Waypoints для реального маніпулятора.

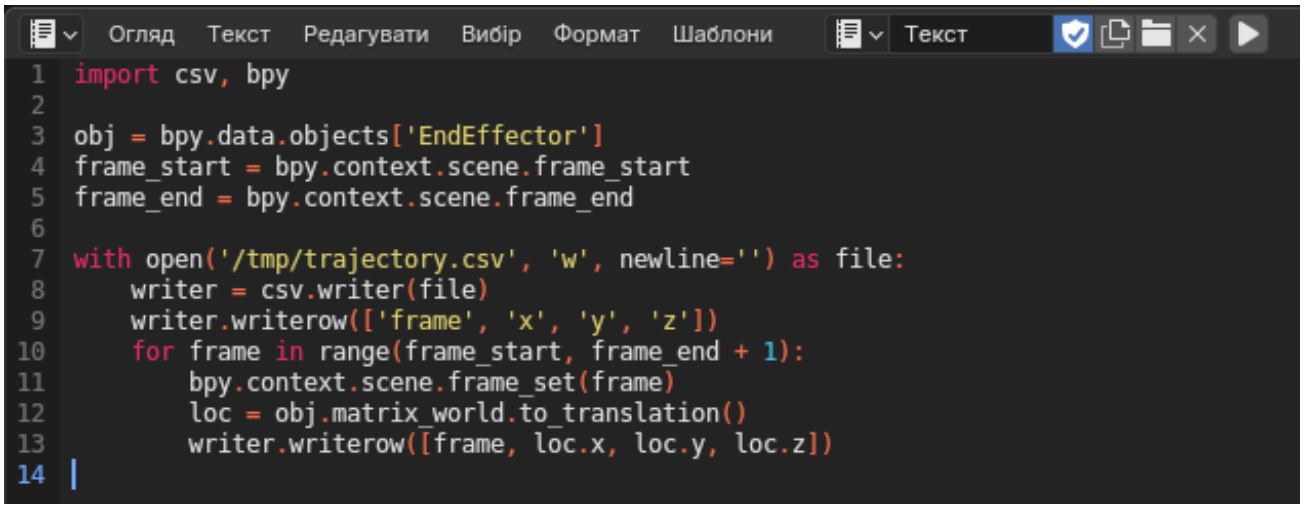

Рисунок 1.12 – Код на Python для зчитування кінцевого ефектора

Цей приклад показує, як Blender може бути використаний як візуальний редактор руху, де замість складного програмування в консолі, інженер задає рух «від руки» і лише експортує дані у зручному форматі для реального застосування.

Оскільки Blender має зручну систему анімації (ключові кадри, інверсна кінематика, криві руху), його дуже зручно використовувати для візуального створення траєкторій руху маніпулятора. Замість програмування координат вручну або використання складних пакетів типу MoveIt! для ROS, можна просто

«прокласти» шлях руху в Blender – візуально, інтуїтивно.

Після цього – зчитати позиції суглобів або координати кінцевого ефектора (наприклад, кожні 0.1 секунди) і експортувати їх у CSV, JSON або інший формат для подальшого використання у контролерах реального робота.

Один з найцікавіших напрямків – керування реальним роботом через Blender. Це стало можливим завдяки плагінам, як–от:

MarIOnette – це безкоштовний додаток для Blender, розроблений ентузіастами, який дозволяє перетворити Blender із суто графічного редактора на інтерфейс керування реальними роботами (рис. 1.14). Він відкриває можливість транслювати рухи 3D–моделі на апаратний пристрій наприклад, серводвигуни або маніпулятори.

Рисунок 1.13 – Прев'ю принципу роботи плагіну Marionette [19]

Основна ідея. При наявності 3D–моделі робота в Blender можна задати його положення вручну, через інверсну кінематику (IK), або навіть анімацію. І в реальному часі, ці положення передаються на фізичний пристрій – тобто реальний робот починає повторювати рухи, які створені у Blender.

Це можливо завдяки тому, що MarIOnette:

– зчитує положення об'єктів у сцені (особливо кісток або об'єктів із назвами типу servo1, arm_link2 тощо);

– трансформує ці дані в набір координат/кутів;

– передає ці дані через серійний порт (UART/USB) або мережевий сокет (TCP/WebSocket);

– на іншій стороні ці дані приймає мікроконтролер (наприклад, Arduino, ESP32, Raspberry Pi), який і керує мотором або маніпулятором.

Що тобі потрібно:

– Blender (версія 3.0+);

– плагін MarIOnette (у вигляді .py або .zip–додатка) – встановлюється через Preferences > Add–ons > Install;

– реальний пристрій із серійним інтерфейсом (наприклад, Arduino із серводвигунами);

– скрипт–приймач на стороні пристрою, який читає дані із серійного порту та обертає сервоприводи.

Приклад роботи у Blender:

– створюється об'єкт, наприклад servo1, який відповідає куту першого суглоба;

– створюється анімація: цей об'єкт повертається з 0° до 90°;

– MarIOnette перехоплює ці дані в реальному часі та відправляє рядок: servo1:90\n.

Приклад коду для Arduino який читає рядок з порту, перевіряє чи це команда для servo1, і змінює кут на заданий (рис. 1.15).

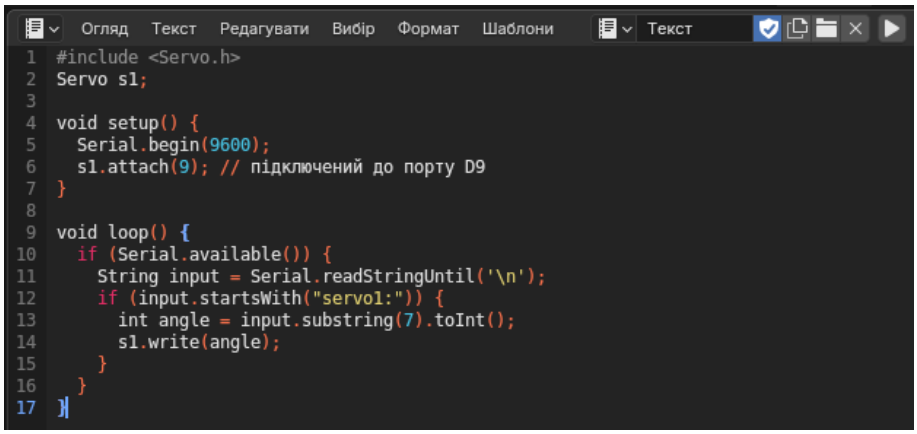

Рисунок 1.14 – Приклад коду для Arduino

Така технологія дає змогу керувати реальними роботами "від руки" – без складного програмування траєкторій; навчати роботів рухам, використовуючи Blender як HMI (інтерфейс людина–машина); створювати демонстраційні сценарії без дорогого обладнання; тестувати кінематику робота ще до завершення розробки повного алгоритму

Серед основних переваг додатку варто відзначити: відсутність потреби у глибоких знаннях ROS або C++; можливість візуального контролю дій робота в Blender; відкритий програмний код з підтримкою Python; та універсальність у використанні з апаратним забезпеченням, що має серійний інтерфейс обміну даними.

Також є деякі обмеження а саме: Не підходить для складної зворотної кінематики чи точного контролю в реальному часі; потрібно самому писати приймальний код на мікроконтролері; немає стандартів підключення[19].

Blender ROS Bridge – це неофіційний набір інструментів, що дозволяє інтегрувати Blender із системою ROS (Robot Operating System). Завдяки такому поєднанню відкривається можливість об’єднати візуально–орієнтоване середовище Blender з потужною інфраструктурою ROS для управління реальними або симульованими роботами.

Інакше кажучи, це спосіб, як використовувати Blender не лише для 3D–моделювання, а як візуальний frontend або навіть симулятор для ROS–сумісного робота.

Принцип роботи. У Blender створюється сцена з моделлю робота (маніпулятор, мобільна платформа тощо).

За допомогою Python API в Blender ти можеш зчитувати/оновлювати положення об’єктів, орієнтацію, сенсорні дані тощо.

З іншого боку, ROS надає топіки для передавання даних, наприклад /joint_states, /cmd_vel, /sensor_data тощо.

Між Blender і ROS використовується міст – Python–скрипти, які через бібліотеку rospy або rosbridge (через WebSocket) обмінюються даними.

Тобто, наприклад, коли в Blender потрібно зробити анімацію руху

маніпулятора, ці кути можна транслювати в ROS, а там – на апаратного робота. І навпаки: коли сенсор робота бачить перешкоду, ця інформація може бути передана назад у Blender і відобразитись у сцені.

Переваги використання Blender з ROS э візуалізація та налагодження: Blender дає змогу легко візуалізувати стан робота: положення суглобів, розміщення об'єктів, перешкоди, зони досяжності. Це робить налагодження системи більш інтуїтивним – ти бачиш, що «бачить» або «робить» робот, не лише у коді чи в терміналі, а у 3D.

Створення власних симуляцій: На відміну від Gazebo, Blender дозволяє створити свою власну сцену з високоякісною графікою, анімаціями, освітленням і навіть фізикою. Це дає змогу створити, наприклад, навчальний симулятор або презентацію для інвесторів, яка буде виглядати набагато краще, ніж у звичайному ROS GUI.

Blender API повністю Python–орієнтований. А оскільки ROS також має Python–бібліотеку (rospy), написання моста між ними – просте й зрозуміле. Усі дані (позиція, швидкість, команда, статус) можна обробляти прямо у Blender.

Відсутність обмежень налаштування дозволяє створити власний GUI у Blender, власну логіку, власні скрипти – не прив'язуючись до обмежень класичних ROS–інструментів. Наприклад, керувати роботом через інтерактивні панелі в Blender, або запускати анімації через ROS–події.

Недоліки та обмеження

– неофіційний статус та підтримка: На відміну від Gazebo, який офіційно підтримується ROS, Blender ROS Bridge є результатом ініціативи спільноти розробників з відкритим вихідним кодом. Тобто не можливо отримати готову документацію, підтримку чи стабільні оновлення. Доведеться шукати рішення на форумах або GitHub;

– відсутність готової фізики та сенсорів: У Gazebo є готові сенсори (лідар, IMU, камери), які автоматично генерують симуляцію даних. У Blender доведеться створювати власну логіку або підключати зовнішні емулятори;

– потреба у власних скриптах: Для взаємодії із ROS потрібно самостійно

створювати мости на основі rospy, що вимагає глибшого розуміння архітектури ROS. Для забезпечення передачі кута повороту сервопривода до візуального середовища Blender необхідно реалізувати програмний модуль, що здійснює підписку на ROS-топік /joint_states та виконує синхронізацію отриманих даних із трансформаціями відповідного об'єкта сцени.

– Обмеження щодо роботи в реальному часі: середовище Blender не адаптоване для функціонування в умовах жорсткого (детермінованого) реального часу, що ускладнює його застосування для задач із високими вимогами до затримки та передбачуваності виконання. [20].

Порівняння додатків можна представити вигляді табл. 1.2.

Таблиця 1.2 – Порівняння додатків для Blender для робототехніки

Критерій	MarIOnette	Blender ROS Bridge	Custom Python Scripts (експорт траєкторій)
1	2	3	4
Призначення	Пряме керування реальними пристроями через серійний порт	Інтеграція з ROS через WebSocket або rospy	Експорт координат/анімованих рухів для подальшого використання в ROS або мікроконтролері
Тип з'єднання	UART / Serial (USB)	WebSocket або TCP (через rosbridge або rospy)	Файли (CSV, JSON) або вручну сформовані команди
Переваги	Простий у налаштуванні; підходить для навчання; працює в реальному часі	Глибока інтеграція з ROS; можливість двостороннього обміну даними; високий рівень налаштування	Візуальне програмування; гнучкість формату; немає потреби у прямому зв’язку

Продовження таблиці 1.2

1	2	3	4
Недоліки	Потребує Arduino/серійного приймача; обмежене масштабування	Неофіційний статус; потребує знань ROS; складна синхронізація	Не підходить для роботи в реальному часі; лише сценарії без підключення до Інтернету
Критерій	MarIOnette	Blender ROS Bridge	Custom Python Scripts (експорт траєкторій)
Реалізація	Blender додаток (.zip), інтерфейс для керування в сцені	Скрипти Python з підключенням до ROS (зазвичай через rosbridge)	Простий Python–експортер координат із Blender
Тип управління	Керування окремими сервоприводами (кутами)	Управління рухами через ROS–топіки (наприклад, /joint_states)	Управління позицією або станами через заздалегідь розраховану траєкторію
Рівень складності	Низький	Середній – Високий	Низький – Середній
Сфера застосування	Прототипування, демонстрації, навчання	Дослідження, симуляції, робототехнічні проекти з ROS	Початкове навчання, експерименти з анімацією, підготовка рухів

2 РОЗРОБЛЕННЯ ДОДАТКУ–ТРЕНАЖЕРА ДЛЯ КЕРУВАННЯ МАНІПУЛЯТОРОМ

2.1 Загальна концепція та призначення тренажера

Розробка віртуальних тренажерів у галузі автоматизації виробництва набуває все більшої актуальності через стрімкий розвиток робототехніки, збільшення складності виробничих процесів та необхідність попереднього тестування алгоритмів без залучення дорогого фізичного обладнання. У зв’язку з цим було поставлено завдання створити програмне середовище, яке дозволяло б імітувати повноцінну роботу маніпулятора на умовній виробничій лінії.

Основна ідея створення даного додатку–тренажера полягає у моделюванні циклічної роботи маніпулятора, що виконує захоплення, заливку, переміщення та сортування об’єктів – зокрема, банок з рідиною – у заданій послідовності. У рамках симуляції враховано такі етапи, як контроль наявності об'єкта в зоні роботи, вирівнювання, автоматичне захоплення, транспортування та складування.

Проект реалізовано як набір Blender–додатків, які дозволяють налаштувати середовище симуляції відповідно до потреб користувача. Blender обрано як платформу не випадково – це потужне середовище для 3D–моделювання і анімації, що має підтримку Python API, що дозволяє створювати власні сценарії та інтерактивну логіку поведінки об’єктів. Таке рішення дозволяє візуалізувати кожну дію маніпулятора у режимі реального часу, а також створювати сценарії з різними рівнями складності – як у ручному, так і в автоматичному режимах.

Призначення тренажера полягає в наступному:

– освітнє застосування. Додаток може бути використаний студентами для вивчення принципів роботи автоматизованих систем, маніпуляторів, логіки сортування і транспортування об’єктів. Це дозволяє отримати практичні навички роботи з логікою роботи виробничих ліній без фізичних витрат;

– інженерна перевірка алгоритмів. Перед впровадженням програмного

забезпечення у фізичну систему, можна протестувати його логіку у віртуальному просторі, оцінити поведінку маніпулятора в різних умовах (наприклад, зміщення об’єктів, збій послідовності тощо);

– підготовка операторів. Тренажер дозволяє створити сценарії, де майбутній оператор виробничої лінії може натренуватися працювати з системою без ризику пошкодження обладнання або продукції;

– швидке налагодження логіки. У середовищі Blender можна оперативно змінити позиції, геометрію або властивості об’єктів та моментально перевірити, як це вплине на весь процес. Це значно пришвидшує процес розробки.

Таким чином, тренажер виконує одразу кілька функцій: навчальну, тестову, інженерну та демонстраційну. Завдяки використанню Blender як основної платформи, розробник отримує уніфіковане середовище з широкими можливостями для модифікації, інтеграції додаткових модулів та створення сценаріїв будь–якої складності.

Особливістю цього проекту є також розбиття логіки на окремі додатки – кожен з яких відповідає за свою підсистему. Такий підхід дозволяє ізольовано тестувати окремі компоненти системи (наприклад, модуль заливки, сортування або руху конвеєра), а потім інтегрувати їх у єдину керовану логіку.

2.2 Архітектура проекту та використані технології

Розробка тренажера розпочиналась із постановки основних цілей, серед яких – створення системи, що не лише виконує програмну логіку у фоновому режимі, а й наочно демонструє віртуальну модель реального виробничого процесу. Було поставлено завдання реалізувати повноцінний візуалізований макет, у якому об’єкти мали б реалістичну поведінку, взаємодіяли б між собою, а також дозволяли б користувачу на будь–якому етапі зупинити процес, проаналізувати його хід і, за потреби, внести зміни.

Саме з огляду на такі вимоги було обрано середовище Blender як основну платформу реалізації. Цей інструмент забезпечує не лише 3D–візуалізацію, а й

широкі можливості інтеграції програмної логіки, взаємодії між об’єктами, анімації та користувацького інтерфейсу, що дозволяє будувати інтерактивні симуляційні моделі високого рівня деталізації.

У процесі вибору платформи для реалізації тренажера було розглянуто декілька варіантів. Зокрема, часто використовувані серед студентів середовища Unity або Unreal Engine, які переважно орієнтовані на розробку інтерактивних ігрових додатків. Проте в контексті моделювання точних, керованих виробничих процесів ці інструменти виявились менш зручними, особливо у частині роботи з фізичними об’єктами, контролем анімацій та паралельних дій.

Натомість середовище Blender продемонструвало низку переваг, які зробили його доцільним вибором для реалізації проекту. По–перше, воно має вбудовану підтримку мови програмування Python, що дозволяє створювати власні аддони, оператори та сценарії для прямого керування об’єктами сцени. По–друге, Blender забезпечує зручне моделювання 3D–об’єктів, створення анімацій і – що особливо важливо – миттєву візуалізацію результатів роботи коду у реальному часі. Це критично при реалізації синхронізованих процесів, таких як рух конвеєра, заливка, обертання, переміщення та сортування об’єктів.

Крім того, середовище надає широкі можливості для налаштування інтерфейсу користувача. Було реалізовано окрему панель керування безпосередньо в області 3D Viewport, куди винесено ключові елементи управління: поля для введення кількості об’єктів, кнопки запуску, зупинки, виклику операторів заливки рідини, переміщення тощо. Такий підхід дозволяє повністю абстрагувати користувача від необхідності взаємодії з програмним кодом, що значно підвищує зручність як у процесі тестування, так і під час демонстрації роботи системи.

Архітектура проекту

Уся структура побудована модульно. Кожен окремий логічний процес винесено у свій Python–додаток:

– automation_controller.py – модуль який запускає автоматичний цикл відносно вже прописаних функцій та параметрів: переміщення банок, заливка, захоплення, закручування кришок і укладання в коробки;

– conveyor.py – відповідає за імітацію руху по конвеєрній стрічці. Реалізовані перевірки на зіткнення з об’єктами, обмеження руху і навіть візуальні ефекти з текстурою;

– liquid_filler.py – модуль, що керує процесом заливки рідини. Він перевіряє, чи є банка під соплом, і регулює швидкість наповнення;

– manipulator.py – тут описано, як маніпулятор знаходить банки, вирівнюється до них, захоплює, переносить і розміщає в коробки. Також реалізовані перевірки на зіткнення з об’єктами, обмеження руху;

– Forklift.py – моделює поведінку віртуального навантажувача, що вивозить заповнені коробки і підвозить нові. Він теж діє автоматично за умов досягнення ліміту по банках.

Такий підхід до побудови архітектури системи, за якого кожна функціональність реалізована у вигляді окремого модуля або оператора, дозволяє не лише підтримувати порядок у структурі коду, а й забезпечує можливість індивідуального тестування та налагодження кожного елемента окремо. Це особливо важливо на етапі розробки та перевірки коректності логіки. Розподіл функціоналу на незалежні компоненти дає змогу ізолювати певну ділянку логіки, не впливаючи на роботу інших частин системи. Наприклад, можливо протестувати лише сценарій заливки рідини, не активуючи при цьому етапи сортування або транспортування. Аналогічно, є змога відпрацьовувати алгоритм обертання кришки, залишаючи інші механізми – зокрема, конвеєр – у неактивному стані. Це суттєво підвищує стабільність розробки, а також спрощує виявлення та виправлення потенційних помилок.

Використані технології:

– Blender 4.4.0 – основне середовище для 3D–візуалізації, Python–скриптингу і створення додатків;

– Python 3.10 (вбудований у Blender) – мова програмування для написання логіки. Усі оператори та інтерфейси реалізовано через bpy – це офіційний API Blender для взаємодії з усіма внутрішніми об’єктами, сценами, властивостями та інтерфейсом. Він дозволяє безпосередньо з Python–коду змінювати розташування

моделей, запускати анімації, створювати нові об’єкти, а також додавати користувацькі панелі керування в інтерфейс Blender. Зокрема, завдяки конструкції bpy.data.objects.get("назва") можливо отримати конкретний об’єкт на сцені – наприклад, банку, маніпулятор або контейнер – після чого виконати з ним необхідні дії: змінити координати, обертання, прикріпити до іншого об’єкта, або задати новий стан.

Саме завдяки bpy було реалізовано логіку, що працює в режимі реального часу: об’єкти реагують на події, дії виконуються поступово за заданим алгоритмом, а зміни відображаються у вікні сцени на кожному кроці. Такий механізм дозволяє досягти високої інтерактивності моделі та створити ілюзію безперервного, «живого» виробничого процесу.

Модуль bpy поєднує в собі гнучкість мови Python із прямим доступом до 3D–функціональності Blender, що дає змогу керувати практично будь–якою дією, яку зазвичай виконують вручну – включно з трансформацією об'єктів, запуском анімацій, створенням ключових кадрів тощо. Завдяки цьому досягнута висока продуктивність розробки та точне відтворення сценаріїв автоматизації.

Mathutils – використовується для реалізації координатних обчислень у тривимірному просторі було використано модуль mathutils, зокрема клас Vector, який є стандартним інструментом для роботи з векторами у середовищі Blender. У системі Blender позиції об’єктів, напрямки руху, обертання, зміщення та інші трансформації не оперують окремими числовими значеннями, а реалізуються через тривимірні вектори, які мають компоненти по осях X, Y та Z. Використання Vector дає змогу виконувати арифметичні операції над координатами, обчислювати відстані між об’єктами, напрямки, кутові зсуви, а також реалізовувати обертання чи трансляцію з урахуванням просторового контексту. Це дозволяє побудувати точні алгоритми взаємодії між об’єктами, синхронізувати їхнє положення, а також реалізувати складні сценарії поведінки, які враховують як геометрію сцени, так і динаміку процесу. Наприклад, коли маніпулятор вирівнюється до банки, потрібно порахувати напрям, у якому він має зрушити – для цього віднімається одна позиція від іншої:

```
direction = target.location - obj.location
```

Результатом операції віднімання координат є об’єкт типу Vector, який вказує напрямок, у якому має здійснюватися рух. Далі цей вектор нормалізується – тобто приводиться до одиничної довжини – з метою забезпечення сталої швидкості переміщення. Це реалізується через команду direction.normalize(). Після цього положення об’єкта оновлюється додаванням до його координат нормованого напрямку, помноженого на значення швидкості:

```
obj.location += direction * speed.
```

Такий підхід дозволяє точно та стабільно керувати переміщенням об’єктів у сцені.

bpy.props – використовується для створення керованих параметрів у сцені який дозволяє оголошувати змінні, значення яких можна задавати або змінювати через інтерфейс користувача. До таких параметрів належать, зокрема, швидкість руху об’єктів, кількість банок у партії, поточний статус процесу, а також інші службові властивості, необхідні для контролю за виконанням автоматизованих дій.

Завдяки цьому підходу проект отримав високу гнучкість і масштабованість. Стало можливим додавання нових типів об’єктів, зміна логіки дій, а також потенційна інтеграція з альтернативними засобами керування – наприклад, з клавіатури або зовнішніх пристроїв – без необхідності змінювати основну архітектуру системи[22-23].

2.3 Опис основних функціональних модулів

У процесі розробки тренажера структура проекту була поділена на окремі функціональні модулі (додатки), кожен з яких відповідає за конкретну частину логіки: керування конвеєром, заливку рідини, роботу маніпулятора,

обслуговування складу тощо. Такий підхід дозволив зробити систему більш гнучкою, зручною для тестування й подальшого розширення.

Модулі реалізовані за допомогою Python–скриптів з використанням API Blender, що дає змогу керувати об’єктами сцени в реальному часі. У цьому підрозділі описано призначення кожного аддона та наведено ключові фрагменти коду, які реалізують основну логіку проекту.

2.3.1 conveyor.py – Модуль керування рухом конвеєра

Модуль conveyor.py відповідає за імітацію руху банок по конвеєрній стрічці у віртуальному середовищі Blender. Його головне призначення – забезпечити логіку поступового переміщення об’єктів (банок з рідиною) з одного кінця лінії до іншого, при цьому враховуючи обмеження простору, наявність перешкод та взаємодію з іншими об'єктами сцени.

Основні завдання модуля:

– переміщення банок вперед/назад по осі Y та візуальний ефект руху стрічки (через зміщення текстури);

– перевірка наявності перешкод (маніпулятор, кінець стрічки);

– вивід попереджень у випадку блокування руху;

– керування через панель користувача та клавіатуру;

– реалізація переміщення банок вперед/назад по осі Y та візуальний ефект руху стрічки (через зміщення текстури).

За рух конвеєра відповідає метод move(), який знаходиться у класі ConveyorBase. Саме він відповідає за фактичне пересування об’єктів типу krishka (банки) уздовж осі Y. Це означає, що кожна банка зсувається на певну відстань назад (оскільки в моїй сцені конвеєр рухається в мінус Y), а коефіцієнт KRISHKA_SPEED_MULTIPLIER дозволяє гнучко регулювати швидкість. Також реалізовано зсув текстури стрічки по осі Y . Це зроблено через зміну значення offset на Mapping Node в матеріалі Rubber. Це було зроблено для візуального ефекту синхронного руху банки і стрічки (рис. 2.1).

```
        # Двигаємо банку
        krishka.location.y -= distance * KRISHKA_SPEED_MULTIPLIER

    # Двигаємо текстуру
    if mapping_node:
        offset = mapping_node.inputs[1].default_value
        offset[1] += distance * TEXTURE_SPEED_MULTIPLIER
        mapping_node.inputs[1].default_value = offset
```

Рисунок 2.1 – Рядок коду, який відповідає за рух конвеєра та текстури стрічки

Реалізація перевірки наявності перешкод та кінець стрічки. Для уникнення ситуацій, коли банки можуть "проходити" крізь інші об’єкти або зіштовхуватись з маніпулятором, у модулі було реалізовано перевірку на наявність перешкод перед виконанням руху. Така перевірка забезпечує логічність поведінки системи й дозволяє зберігати коректність просторового розміщення об'єктів у сцені.

Механізм реалізовано шляхом створення умовної точки попереду об’єкта, що рухається, яка перевіряється на потрапляння в межі колайдера (маніпулятора чи кінця стрічки). Для цього використовуються допоміжні функції def is_point_in_aabb(point, obj), def is_colliding(obj1, obj2, tx=0.3, ty=0.3, tz=0.3), def get_world_bbox_corners(obj), def is_aabb_overlap(obj1, obj2) що реалізують геометричну перевірку на основі координат (рис. 2.2 – рис. 2.4).

```
#Перевіряє, чи знаходиться точка point всередині габаритного куба (AABB — Axis-Aligned Bounding Box) об’єкта obj.
def is_point_in_aabb(point, obj):
    bbox = get_world_bbox_corners(obj)
#Отримується список координат всіх 8 кутів прямокутного габаритного об'єму (bounding box) об'єкта у світових координатах.
    min_corner = Vector((min(v.x for v in bbox), min(v.y for v in bbox), min(v.z for v in bbox)))
    max_corner = Vector((max(v.x for v in bbox), max(v.y for v in bbox), max(v.z for v in bbox)))
#Обчислюються мінімальні та максимальні координати по X, Y, Z — тобто кути "ящика" bounding box, в якому знаходиться об’єкт.
    return (
        min_corner.x <= point.x <= max_corner.x and
        min_corner.y <= point.y <= max_corner.y and
        min_corner.z <= point.z <= max_corner.z
    )
#Повертає True, якщо точка point лежить всередині bounding box об'єкта. Інакше — False.
```

Рисунок 2.2 – Функція def is_point_in_aabb(point, obj)

```
def is_colliding(obj1, obj2, tx=0.3, ty=0.3, tz=0.3):
    #Перевіряє, чи знаходяться два об’єкти досить близько один до одного — умовна "колізія".
    return (abs(obj1.location.x - obj2.location.x) < tx and
            abs(obj1.location.y - obj2.location.y) < ty and
            abs(obj1.location.z - obj2.location.z) < tz)
#Якщо відстань між об’єктами по кожній осі менша за заданий поріг (tx, ty, tz) — вважається, що об’єкти "зіштовхнулися".
```

Рисунок 2.3 – Функція def is_colliding(obj1, obj2, tx=0.3, ty=0.3, tz=0.3)

```python
def get_world_bbox_corners(obj):
    return [obj.matrix_world @ Vector(corner) for corner in obj.bound_box]
#Отримує координати всіх 8 кутів bounding box у світових координатах.

def is_aabb_overlap(obj1, obj2):
    bbox1 = get_world_bbox_corners(obj1)
    bbox2 = get_world_bbox_corners(obj2)
#Перевіряє, чи перетинаються два габаритних прямокутника (bounding boxes)

    min1 = Vector((min(v.x for v in bbox1), min(v.y for v in bbox1), min(v.z for v in bbox1)))
    max1 = Vector((max(v.x for v in bbox1), max(v.y for v in bbox1), max(v.z for v in bbox1)))
 #Обчислення мінімального та максимального кута bounding box першого об'єкта.

    min2 = Vector((min(v.x for v in bbox2), min(v.y for v in bbox2), min(v.z for v in bbox2)))
    max2 = Vector((max(v.x for v in bbox2), max(v.y for v in bbox2), max(v.z for v in bbox2)))
#Обчислення мінімального та максимального кута bounding box другого об'єкта.

    return (
        min1.x <= max2.x and max1.x >= min2.x and
        min1.y <= max2.y and max1.y >= min2.y and
        min1.z <= max2.z and max1.z >= min2.z
    )
```

Рисунок 2.4 – Функція def get_world_bbox_corners(obj), def is_aabb_overlap(obj1, obj2)

У разі виявлення перешкоди рух припиняється, а в інтерфейсі користувача відображається відповідне попередження. Це дозволяє зберігати інформативність і запобігає некоректному виконанню алгоритму.

Приклад логіки виведення повідомлень наведено нижче (рис. 2.5):

```python
#Вивід повідомлення якщо перешкода та обмеження, та комбінації кнопок щоб вийти з перешкоди
for obj in grabbers + ends:
    if is_point_in_aabb(probe, obj):
        if not wm["obstacle_shown"] or wm["last_blocked_direction"] != direction:
            wm["obstacle_shown"] = True
            wm["last_blocked_direction"] = direction
            global warning_text

            if obj.name.startswith("grabber_collider"):
                warning_text = (
                    "Попереду маніпулятор!\n"
                    "Якщо рух вперед – натисніть ← ↑ ↓ ;\n"
                    "Якщо рух назад – натисніть ← ↓ ↑ "
                )

            elif obj.name.startswith("end"):
                warning_text = (
                    "Обмеження по руху!\n"
                    "Якщо рух вперед – натисніть ← ↑ ↓ ;\n"
                    "Якщо рух назад – натисніть ← ↓ ↑ "
                )
            else:
                warning_text = "Перешкода!"

            bpy.ops.wm.call_menu(name="WM_MT_obstacle_warning")

        # Блокуємо рух тільки якщо знову у ту сторону
        if wm["last_blocked_direction"] == direction:
            return True  # стоп тільки в цю сторону
```

Рисунок 2.5 – Фрагмент логіки виведення повідомлення о перешкоді

Завдяки такому підходу було досягнуто взаємодію з оточенням у реальному часі, а також забезпечено адаптивність поведінки об’єктів залежно від ситуації в сцені. Подібна перевірка є необхідною частиною моделювання динаміки

виробничої лінії, оскільки дозволяє враховувати логіку просторових обмежень та уникати анімаційних помилок.

Реалізація керування через панель користувача та клавіатуру. Керування рухом об'єктів по конвеєру було реалізовано як через інтерфейс користувача, так і за допомогою клавіатурного вводу, що забезпечує зручність у тестуванні та демонстрації роботи системи.

Для взаємодії через інтерфейс було створено окрему панель керування у вікні 3D View (розділ Sidebar), яка дозволяє змінювати швидкість, запускати або зупиняти рух, а також переміщувати банки вперед або назад за допомогою кнопок:

– регулювати швидкість;
– запускати/зупиняти рух;
– керувати напрямом (вперед/назад);
– отримувати попередження про перешкоди. (рис. 2.6).

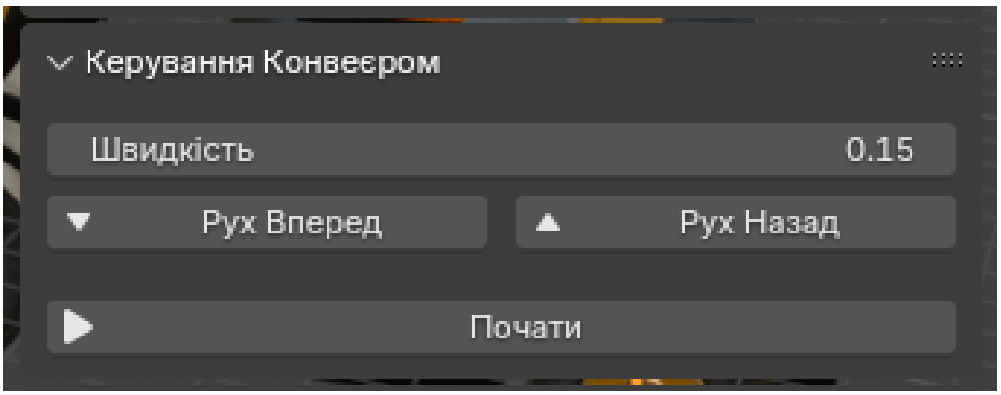

Рисунок 2.6 – Інтерфейс керування конвеєром

Інтерфейс було побудовано на базі класу VIEW3D_PT_conveyor_control, з використанням елементів UI, таких як layout.prop, layout.operator, layout.label

Крім того, було реалізовано модальний оператор, який дає змогу керувати рухом безпосередньо з клавіатури. Завдяки обробці подій клавіш W, S та стрілок ↑, ↓, користувач може ініціювати рух банок по стрічці у відповідному напрямку. Подібне рішення особливо зручне під час налагодження або демонстрацій, коли доступ до панелі може бути обмежений (рис. 2.7).

```python
class VIEW3D_PT_conveyor_control(bpy.types.Panel):
    bl_label = "Керування Конвеєром"
    bl_idname = "VIEW3D_PT_conveyor_control"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "Керування Лінією"

    def draw(self, context):
        layout = self.layout
        scene = context.scene

        layout.enabled = not scene.auto_line_running

        if not scene.auto_line_running:
            layout.prop(scene, "conveyor_speed", text="Швидкість")
        else:
            layout.label(text=f"Швидкість: {scene.conveyor_speed:.2f}")

        row = layout.row()
        row.operator("wm.move_conveyor_forward", text="Рух Вперед", icon='TRIA_DOWN')
        row.operator("wm.move_conveyor_backward", text="Рух Назад", icon='TRIA_UP')

        layout.separator()

        if scene.conveyor_running:
            layout.operator("wm.stop_conveyor_control", text="Зупинити", icon='PAUSE')
        else:
            layout.operator("wm.start_conveyor_control", text="Почати", icon='PLAY')
```

Рисунок 2.7 – Фрагмент коду для виведення елементів UI

Також у модальному режимі встановлюється таймер, який через короткі проміжки часу оновлює позиції об’єктів у сцені, забезпечуючи плавність анімації. Такий підхід дозволив створити інтерактивну систему керування, яка реагує на дії користувача в реальному часі (рис. 2.8).

```python
class WM_OT_MoveForward(bpy.types.Operator):
    bl_idname = "wm.move_conveyor_forward"
    bl_label = "Рух Вперед"

    def execute(self, context):
        ConveyorBase.move(context.scene.conveyor_speed, direction='forward')
        return {'FINISHED'}

class WM_OT_MoveBackward(bpy.types.Operator):
    bl_idname = "wm.move_conveyor_backward"
    bl_label = "Рух Назад"

    def execute(self, context):
        ConveyorBase.move(-context.scene.conveyor_speed, direction='backward')
        return {'FINISHED'}

class WM_OT_ConveyorModal(bpy.types.Operator):
    bl_idname = "wm.conveyor_modal"
    bl_label = "Conveyor Modal"

    def modal(self, context, event):
        scene = context.scene

        # ESC / ПКМ для выхода
        if event.type in {'ESC', 'RIGHTMOUSE'}:
            return self.cancel(context)

        if not scene.conveyor_running:
            return self.cancel(context)

        # Кнопки управління
        if event.type in {'W', 'UP_ARROW'}:
            self.up_pressed = event.value == 'PRESS'
            return {'RUNNING_MODAL'}

        if event.type in {'S', 'DOWN_ARROW'}:
            self.down_pressed = event.value == 'PRESS'
            return {'RUNNING_MODAL'}

        # Таймер - рухаємо
        if event.type == 'TIMER':
            speed = scene.conveyor_speed * 0.5
            if self.up_pressed:
                ConveyorBase.move(speed, direction='forward')
            if self.down_pressed:
                ConveyorBase.move(-speed, direction='backward')

            return {'RUNNING_MODAL'}

        return {'PASS_THROUGH'}
```

Рисунок 2.8 – Фрагмент коду модального оператора для руху конвеєра

2.3.2 liquid_filler.py – Модуль заливки рідини

Модуль liquid_filler.py є одним із етапів моделювання виробничого процесу є заповнення тари рідиною. Для реалізації цього процесу було створено окремий модуль, який відповідає за логіку позиціювання сопла над банкою, перевірку наявності ємності під ним, імітацію потоку рідини та поступове наповнення.

Для того щоб уникнути помилок і візуального "зливання у повітря", було реалізовано функцію is_krishka_under_nozzle_support(). Вона перевіряє, чи знаходиться банка у допустимих межах координат під конструкцією з назвою nozzle_support.

Використано порівняння координат по трьох осях (X, Y, Z) з допусками, які визначають, чи банка в правильному положенні (рис. 2.9).

```
def is_krishka_under_nozzle_support():
    nozzle_support = bpy.data.objects.get("nozzle_support") # Отримуємо об'єкт сопла
    if not nozzle_support:
        return False  # Якщо об'єкт не знайдено — повертаємо помилкове значення

    # Перебираємо всі об'єкти, що починаються з "krishka"
    for obj in bpy.data.objects:
        if obj.name.startswith("krishka"):

                # Обчислюємо відстань між об'єктом та соплом по кожній осі
            dx = abs(nozzle_support.location.x - obj.location.x)
            dy = abs(nozzle_support.location.y - obj.location.y)
            dz = nozzle_support.location.z - obj.location.z

             # Якщо відстані менші за допустимі межі — вважається, що банка під соплом
            if dx < 0.14 and dy < 0.14 and 0.35 < dz < 0.6:
                return True
    return False # Якщо жодна банка не під соплом — повертаємо False
```

Рисунок 2.9 – Функція is_krishka_under_nozzle_support().

Подібний принцип також застосовано у функції find_attached_honey_cutter(), яка знаходить дочірній об’єкт банки, що імітує поверхню рідини – це об’єкт із назвою, що починається з honey_cutter.

Функцію find_attached_honey_cutter() можна побачити на рис. 2.10.

```python
def find_attached_honey_cutter():
    nozzle_support = bpy.data.objects.get("nozzle_support")
    if not nozzle_support:
        return None

    for krishka in bpy.data.objects:
        if not krishka.name.startswith("krishka"):
            continue

        dx = abs(nozzle_support.location.x - krishka.location.x)
        dy = abs(nozzle_support.location.y - krishka.location.y)
        dz = nozzle_support.location.z - krishka.location.z

        if dx < 0.14 and dy < 0.14 and 0.35 < dz < 0.6:
            # Є банка під соплом - шукаємо її дитини з honey_cutter
            for child in krishka.children:
                if child.name.startswith("honey_cutter"):
                    return child
    return None
```

Рисунок 2.10 – Функція find_attached_honey_cutter()

Сам процес заливки реалізовано через клас–оператор WM_OT_FillHoney. Це модальний оператор, який виконується в циклі з певною затримкою, симулюючи поступове заповнення банки.

Перед початком роботи оператор перевіряє:

– чи є банка під соплом;

– чи знайдено об’єкт honey_cutter;

– чи вже не заповнено банку повністю (якщо об'єкт піднятий вище певного рівня).

Після цього запускається процес заливки:

– створюється анімація руху потоку рідини (honey_stream), який "опускається";

– одночасно піднімається об'єкт honey_cutter, симулюючи зростання рівня рідини в банці;

– обчислюється відсотковий прогрес (fill_progress) – він виводиться в інтерфейсі (рис. 2.11).

```python
# Оператор заливки рідини
class WM_OT_FillHoney(bpy.types.Operator):
    bl_idname = "wm.fill_honey"
    bl_label = "Залити Рідину"

    _timer = None
    _cutter = None
    _stream = None
    _stream_start_z = 0
    _stream_returning = False

    def execute(self, context):
        if not is_krishka_under_nozzle_support():
            self.report({'ERROR'}, "Немає банки під соплом!")
            return {'CANCELLED'}

        cutter = find_attached_honey_cutter()
        if not cutter:
            self.report({'ERROR'}, "Об'єкт 'honey_cutter' не знайдено!")
            return {'CANCELLED'}

        if cutter.location.z > 2.0:
            self.report({'WARNING'}, "У цій банці вже є рідина!")
            return {'CANCELLED'}

        self._cutter = cutter  # зберігаємо объект
        stream = bpy.data.objects.get("honey_stream")
        if stream:
            self._stream = stream
            self._stream_start_z = stream.location.z
        context.scene.fill_progress = 1
        self._timer = context.window_manager.event_timer_add(0.02, window=context.window)
        context.window_manager.modal_handler_add(self)
        return {'RUNNING_MODAL'}

    def modal(self, context, event):
        if event.type == 'TIMER':
            if self._cutter and self._cutter.location.z < MAX_FILL_HEIGHT:
                # Двигаємо потік вниз
                if self._stream:
                    target_z = self._stream_start_z - 4
                    if self._stream.location.z > target_z:
                        self._stream.location.z -= context.scene.fill_speed / 5

                # Поднимаємо рідину
                self._cutter.location.z += context.scene.fill_speed / 10

                # Оновлюємо прогресс
                percent = (self._cutter.location.z / MAX_FILL_HEIGHT) * 100
                context.scene.fill_progress = int(min(percent, 100))

            else:
                # Завершення: збросити потік та прогресс
                if self._stream:
                    self._stream.location.z = self._stream_start_z

                context.window_manager.event_timer_remove(self._timer)
                context.scene.fill_progress = 0
                return {'FINISHED'}
```

Рисунок 2.11 – Класс–оператор WM_OT_FillHoney для заливки рідини

До 3D–панелі інтерфейсу було додано окрему секцію "Заливка Рідини", у якій користувач може:

– налаштовувати швидкість заливки;

– запускати процес за допомогою кнопки;

– бачити прогрес заливки у вигляді слайдера;

– рухати сопло вліво/вправо та вгору/вниз (рис. 2.12).

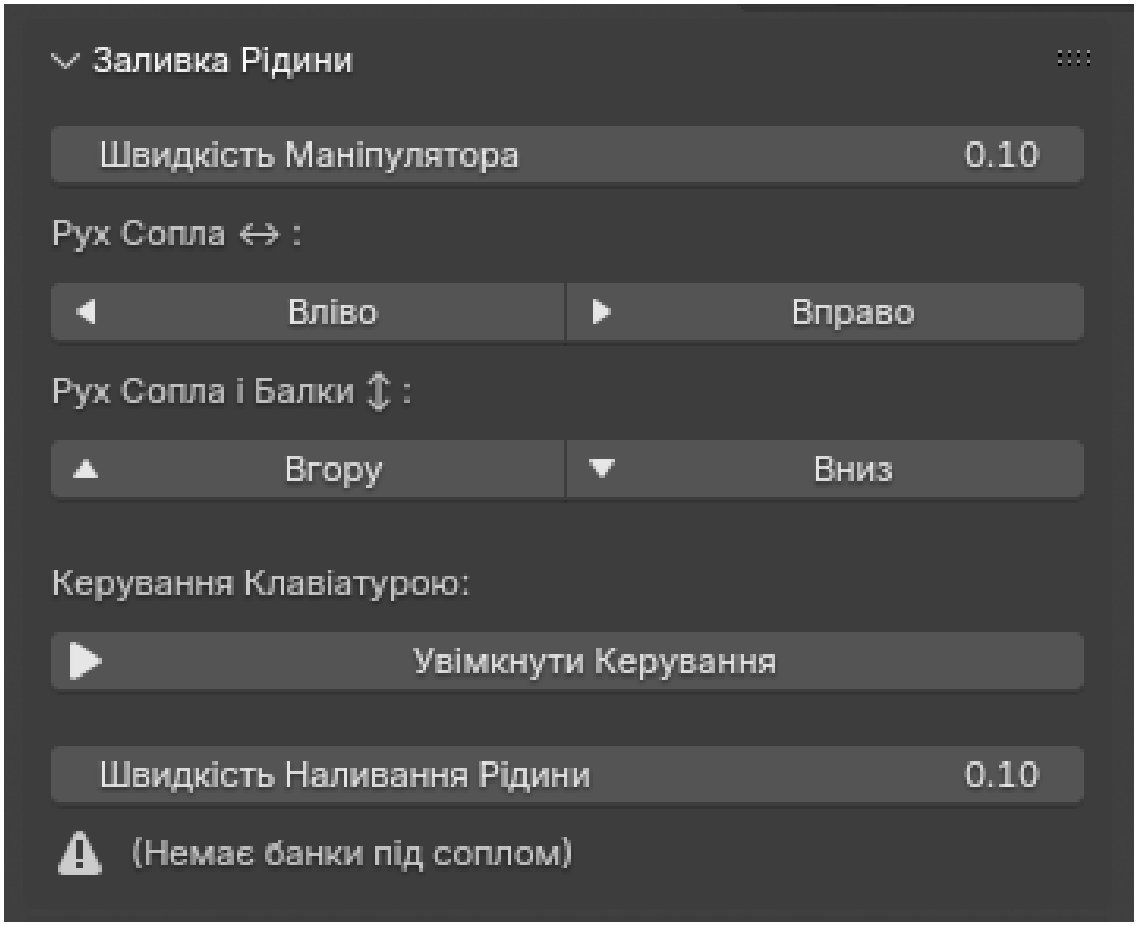

Рисунок 2.12 – Інтерфейс додатку "Заливка Рідини"

Також реалізовано альтернативний спосіб керування – через клавіатуру, що зручно для швидкої симуляції.

За допомогою клавіш WASD або стрілок можна переміщати сопло, а керування активується відповідним оператором WM_OT_StartKeyboardControl(bpy.types.Operator).

Реалізований оператор WM_OT_StartKeyboardControl(bpy.types.Operator) можна побачити на рис. 2.13 – рис. 2.14 .

```python
# Модальне керування клавішами
class WM_OT_StartKeyboardControl(bpy.types.Operator):
    bl_idname = "wm.start_keyboard_control"
    bl_label = "Увімкнути Керування Клавіатурою"

    _timer = None

    def invoke(self, context, event):
        if context.scene.keyboard_control_running:
            self.report({'INFO'}, "Керування вже увімкнене")
            return {'CANCELLED'}

        wm = context.window_manager
        self._timer = wm.event_timer_add(0.02, window=context.window)
        wm.modal_handler_add(self)
        context.scene.keyboard_control_running = True
        return {'RUNNING_MODAL'}

    def modal(self, context, event):
        nozzle = bpy.data.objects.get("nozzle")
        nozzle_support = bpy.data.objects.get("nozzle_support")
        speed = context.scene.manipulator_speed

        # якщо флажок вимкнен - то виходимо
        if not context.scene.keyboard_control_running:
            return self.cancel(context)

        if event.type == 'ESC':
            return self.cancel(context)

        if event.type in {'W', 'UP_ARROW'} and event.value == 'PRESS':
            if nozzle and nozzle.location.z + speed <= NOZZLE_MAX_Z:
                nozzle.location.z += speed
                if nozzle_support:
                    nozzle_support.location.z += speed

        if event.type in {'S', 'DOWN_ARROW'} and event.value == 'PRESS':
            if nozzle and nozzle.location.z - speed >= NOZZLE_MIN_Z:
                nozzle.location.z -= speed
                if nozzle_support:
                    nozzle_support.location.z -= speed

        if event.type in {'A', 'LEFT_ARROW'} and event.value == 'PRESS':
            if nozzle_support and nozzle_support.location.x - speed >= NOZZLE_SUPPORT_MIN_X:
                nozzle_support.location.x -= speed

        if event.type in {'D', 'RIGHT_ARROW'} and event.value == 'PRESS':
            if nozzle_support and nozzle_support.location.x + speed <= NOZZLE_SUPPORT_MAX_X:
                nozzle_support.location.x += speed

        return {'PASS_THROUGH'}

    def cancel(self, context):
        wm = context.window_manager
        if self._timer:
            wm.event_timer_remove(self._timer)
        context.scene.keyboard_control_running = False
        if hasattr(wm, "keyboard_control_operator"):
            if hasattr(wm, "keyboard_control_operator"):
                delattr(wm, "keyboard_control_operator")
```

Рисунок 2.13 – Оператор WM_OT_StartKeyboardControl(bpy.types.Operator) для керування маніпулятор заливки рідини

```python
    def cancel(self, context):
        wm = context.window_manager
        if self._timer:
            wm.event_timer_remove(self._timer)
        context.scene.keyboard_control_running = False
        if hasattr(wm, "keyboard_control_operator"):
            if hasattr(wm, "keyboard_control_operator"):
                delattr(wm, "keyboard_control_operator")

        return {'CANCELLED'}


# Оператор для вимкнення керування
class WM_OT_StopKeyboardControl(bpy.types.Operator):
    bl_idname = "wm.stop_keyboard_control"
    bl_label = "Вимкнути Керування Клавіатурою"

    def execute(self, context):
        context.scene.keyboard_control_running = False  # модальний оператор сам завершиться
        return {'FINISHED'}
```

Рисунок 2.14 – Оператор вимкнення керування

2.3.3 manipulator.py – Модуль роботи маніпулятора

Для реалізації маніпуляцій із об'єктами в межах виробничої лінії було створено окремий модуль manipulator.py, який відповідає за керування маніпулятором, його вирівнювання по об’єктах, захоплення елементів та взаємодію з коробками для пакування.

Цей модуль реалізує функціонал, що дозволяє контролеру автоматично визначати позицію банок і кришок, підводити маніпулятор до об'єкта з урахуванням відстані та висоти, здійснювати захоплення, обертання кришок і подальше укладання в ящик.

Автоматичне вирівнювання. Було реалізовано функції align_to_object() яка відповідає за точне позиціювання маніпулятора відносно цільового об’єкта. Це дає змогу досягти реалістичної симуляції підведення граббера до банки чи кришки (рис. 2.15).

```
def align_to_object(controller, target, offset_z):
    controller.location.x = target.location.x
    controller.location.y = target.location.y - 0.10
    controller.location.z = target.location.z + offset_z
```

Рисунок 2.15 – Функція align_to_object()

Також було введено функцію align_to_free_box_point() для точного позиціювання захопленої банки у вільну комірку ящика, об’єкт box відповідає за вільне місце у ящику, так було реалізовано щоб маніпулятор поміщав продукцію згідно виміряним місцем щоб уникнути зіткнення з іншою банкою (рис. 2.16).

```
def align_to_free_box_point(ctrl, offset_z):
    scene = bpy.context.scene
    for point in bpy.data.objects:
        if point.name.startswith("box."):
            has_banka = any(child.name.lower().startswith("krishka") for child in point.children)
            if not has_banka:
                align_to_object(ctrl, point, offset_z)  # контролер вирівнюється к обєкту
                scene.aligned_to_box = True
                return True
    return False
```

Рисунок 2.16 – Функція align_to_free_box_point()

Таке вирівнювання дозволяє з високою точністю розміщувати маніпулятор у зоні дії, враховуючи офсети по висоті та глибині.

Захоплення об'єктів. У межах моделювання дій маніпулятора важливою частиною є захоплення об'єкта, що дозволяє передавати банку або кришку з однієї точки в іншу. Для цього було реалізовано оператор WM_OT_GrabBanka, який виконує логічне “прикріплення” вибраного об’єкта до контролера маніпулятора.

Алгоритм захоплення побудовано за принципом зміни ієрархії об’єктів у сцені Blender: об'єкт, який потрібно захопити (наприклад, кришка), стає дочірнім елементом об'єкта–маніпулятора. Це реалізується наступним чином (рис. 2.17).

```
439
440 class WM_OT_GrabBanka(bpy.types.Operator):
441     bl_idname = "wm.grab_banka"
442     bl_label  = "Захопити Кришку"
443     def execute(self, context):
444         ctrl  = context.scene.manipulator_controller
445         banca = None
446         for obj in bpy.data.objects:
447             if obj.name.startswith("banka") and obj.parent == ctrl:
448                 banca = obj
449                 break
450         if not banca:
451             banca = find_nearest_object("banka")
452 
453         if banca and ctrl:
454             banca.parent = ctrl
455             banca.matrix_parent_inverse = ctrl.matrix_world.inverted()
456         return {'FINISHED'}
457
```

Рисунок 2.17 – Оператор WM_OT_GrabBanka

(banca.parent = ctrl) встановлює зв’язок “батько–дитина” між маніпулятором і об’єктом. Таким чином, при будь–якому русі або обертанні маніпулятора, прив’язаний об’єкт автоматично повторює його рухи.

(matrix_parent_inverse) забезпечує збереження поточної позиції об’єкта після встановлення нового батьківського зв’язку. Інакше після прив’язки об’єкт міг би зміститися або некоректно повернутися.

Оператор WM_OT_GrabBanka спочатку перевіряє, чи є банка або кришка у визначеній зоні (наприклад, поруч із маніпулятором), і лише після цього виконує прив’язку. Це дозволяє уникнути помилок, коли користувач намагається захопити об’єкт, що фізично недоступний.

Також реалізовано можливість візуального контролю – користувач бачить, що об’єкт “приєднався” до маніпулятора, оскільки він починає рухатися разом із

ним. В інтерфейсі відображається активний стан, який інформує, що об’єкт утримується.

Захоплення є ключовим етапом у логіці роботи всієї лінії, оскільки без цього неможливо виконати обертання кришки або перенесення банки до місця пакування. У поєднанні з іншими операціями (вирівнювання, обертання, відпускання), ця частина модуля забезпечує повноцінний цикл дій маніпулятора.

Обертання кришки. Після успішного захоплення кришки наступним етапом є її обертання для імітації процесу закручування на банку з рідини. Для цього у модулі manipulator.py було реалізовано окремий модальний оператор WM_OT_RotateBanka – modal(self, context, event) який забезпечує поступове обертання об’єкта навколо осі Z з контрольованою швидкістю.

Оператор запускається вручну з панелі керування або програмно після перевірки умов (наявність кришки, банка з рідиною тощо). Під час виконання оператор працює в циклі з таймером, поступово обертаючи кришку (рис. 2.18).

```
def modal(self, context, event):
    if event.type == 'TIMER':
        if self._rot < 120:
            ang = (self.speed/100) * (math.pi/180)
            self.banca.rotation_euler.z -= ang
            center = self.grab.matrix_world.to_translation()
            rot = mathutils.Matrix.Rotation(ang, 4, 'Z')
            mat = mathutils.Matrix.Translation(center) @ rot @ mathutils.Matrix.Translation(-center)
            self.grab.matrix_world = mat @ self.grab.matrix_world
            self._rot += self.speed/100
        else:
            # зберегти мирову трансформацію кришки
            mw = self.banca.matrix_world.copy()

            # відвяжемо від грабера
            self.banca.parent = None
            self.banca.matrix_world = mw

            # привяжемо до банки
            if self.jar:
                self.banca.parent = self.jar
                self.banca.matrix_parent_inverse = self.jar.matrix_world.inverted()
                self.banca.matrix_world = mw  # снова применить точную позицию

            # сбросимо обертання граббера
            self.grab.rotation_euler.z = 0

            # трішки піднять контроллер
            self.jar.parent = self.ctrl
            self.jar.matrix_parent_inverse = self.ctrl.matrix_world.inverted()

            collider = bpy.data.objects.get("grabber_collider")
            if collider:
                if context.scene.grabber_collider_initial_z == 0.0:
                    # зберегти поточну положення в МИРОВиХ координатах
                    context.scene.grabber_collider_initial_z = collider.matrix_world.translation.z

                # Створюємо нову матрицю зі зміщенням униз на 2
                new_world_matrix = collider.matrix_world.copy()
                new_world_matrix.translation.z = context.scene.grabber_collider_initial_z - 2.0

                if collider.parent:
                    parent_inv = collider.parent.matrix_world.inverted()
                    collider.matrix_local = parent_inv @ new_world_matrix
                else:
                    collider.matrix_world = new_world_matrix



            return self.cancel(context)
```

Рисунок 2.18 – Модальний оператор WM_OT_RotateBanka – modal

У цьому фрагменті:

– self.banca – це об’єкт кришки, яка була захоплена;

– rotation_euler.z – кут повороту навколо вертикальної осі;

– ang – кут, на який об’єкт обертається за одну ітерацію (визначається як константа або змінна швидкість).

Процес обертання побудовано таким чином, щоб створити візуально плавну анімацію. Завдяки використанню модального оператора обертання виконується в реальному часі, а не миттєво. Це дозволяє користувачеві спостерігати, як кришка поступово фіксується на банці, що візуально наближає поведінку до реального виробничого процесу.

Окрім кришки, у деяких випадках обертається також і граббер (захват маніпулятора), що створює додатковий ефект механічного зусилля. Це реалізовано шляхом синхронного обертання батьківського об'єкта або допоміжного елемента, що візуально передає обертання.

Також оператор перевіряє, чи вже було досягнуто повного кута обертання (наприклад, 360° або інший заданий поріг). У разі завершення обертання оператор самостійно припиняє свою роботу, після чого об’єкт можна вважати “закрученим”.

Завдяки такій реалізації було досягнуто комбінації анімації, логіки і взаємодії з об'єктами. Обертання стало не лише візуальним ефектом, а функціональним етапом у послідовності дій виробничої симуляції.

Розміщення банки в коробку. Після того як банка була заповнена рідиною, накрита кришкою та відповідно обернута, виконується наступний логічний етап – переміщення готової банки в коробку для подальшого пакування.

Для реалізації цього процесу було створено оператор WM_OT_PlaceJarInBox, який відповідає за визначення вільного місця в коробці, позиціювання банки та її коректну інтеграцію до структури ящика.

Оператор WM_OT_PlaceJarInBox можна побачити на рис. 2.19.

```
# Знайти найближчу ящик
nearest_box = find_nearest_free_box()
if not nearest_box:
    self.report({'WARNING'}, "Ящик не знайдено!")
    return {'CANCELLED'}


# Зберігаємо світову матрицю банки та кришки
mw_jar = jar.matrix_world.copy()
mw_banka = None
banka_obj = None
for child in jar.children:
    if child.name.lower().startswith("banka"):
        banka_obj = child
        mw_banka = child.matrix_world.copy()

# Відв'язуємо банку від маніпулятора
jar.parent = None
jar.matrix_world = mw_jar

# Якщо є кришка — перезберігаємо позицію та прикріплюємо до банку знову
if banka_obj and mw_banka:
    banka_obj.parent = jar
    banka_obj.matrix_world = mw_banka

# Прив'язуємо до скриньки
jar.parent = nearest_box
jar.matrix_parent_inverse = nearest_box.matrix_world.inverted()
spawn_point = bpy.data.objects.get("SpawnPoint")
```

Рисунок 2.19 – Оператор WM_OT_PlaceJarInBox

На початку роботи оператор здійснює пошук коробки, об’єкти якої у сцені мають префікс назви box. Далі зчитується перелік точок розміщення (позицій у середині коробки), які представлені як об’єкти–пустишки (empty) із заданими координатами. Для кожної точки перевіряється, чи не зайнята вона вже банкою. Це реалізовано шляхом аналізу наявності колізій або прив’язаних об’єктів у відповідному місці.

Після того як знайдено вільну позицію, виконується:

а) розрив попереднього зв’язку між банкою та маніпулятором;

б) переміщення банки до цільової точки розміщення:

1) jar.location = target_position.location;

2) jar.rotation_euler = target_position.rotation_euler.

Встановлення нового батьківського зв’язку, при якому банка стає дочірнім об’єктом коробки. Це необхідно для збереження структури при переміщенні коробки або створенні динамічних груп об’єктів у сцені:

```
jar.parent = box
```

Перед зміною батьківського зв’язку виконується збереження matrix_parent_inverse, що дозволяє уникнути небажаного зсуву об’єкта під час прив’язки до нової ієрархії.

Оператор також містить додаткову логіку, яка гарантує:

– що одна позиція в коробці не буде зайнята двома банками одночасно;

– що банка не буде переміщена в коробку, якщо вона ще не повністю готова (наприклад, не має кришки або рівень рідини менший за мінімальний поріг);

– що банка розташовується чітко у межах коробки, без візуального перетинання її стінок.

Таким чином, оператор WM_OT_PlaceJarInBox завершує цикл обробки банки, імітуючи останню фазу виробничого процесу – пакування готової продукції. Його реалізація дозволила досягти структурованості сцени, логічної завершеності симуляції та підготовки об’єктів до вивантаження або подальшої обробки.

Інтерфейс керування маніпулятором. Для забезпечення зручності роботи з маніпулятором у модулі було реалізовано окрему інтерактивну панель керування, яка розташована у бічному меню інтерфейсу Blender (вкладка Sidebar, розділ "Manipulator Control").

Ця панель містить основні інструменти, що забезпечують повноцінну взаємодію з об'єктами та середовищем:

– переміщення маніпулятора по осях X, Y, Z, із візуальним контролем координат та врахуванням обмежень робочої зони;

– активація клавіатурного режиму керування, який дозволяє швидко переміщати маніпулятор за допомогою клавіш W, A, S, D, а також стрілок;

– функції автоматичного вирівнювання маніпулятора відносно банки, кришки або вільної точки у коробці;

– можливість захоплення та відпускання об'єктів, що дозволяє моделювати процес перенесення елементів;

– операція обертання кришки, яка активується окремою кнопкою й забезпечує симуляцію закручування (рис. 2.20).

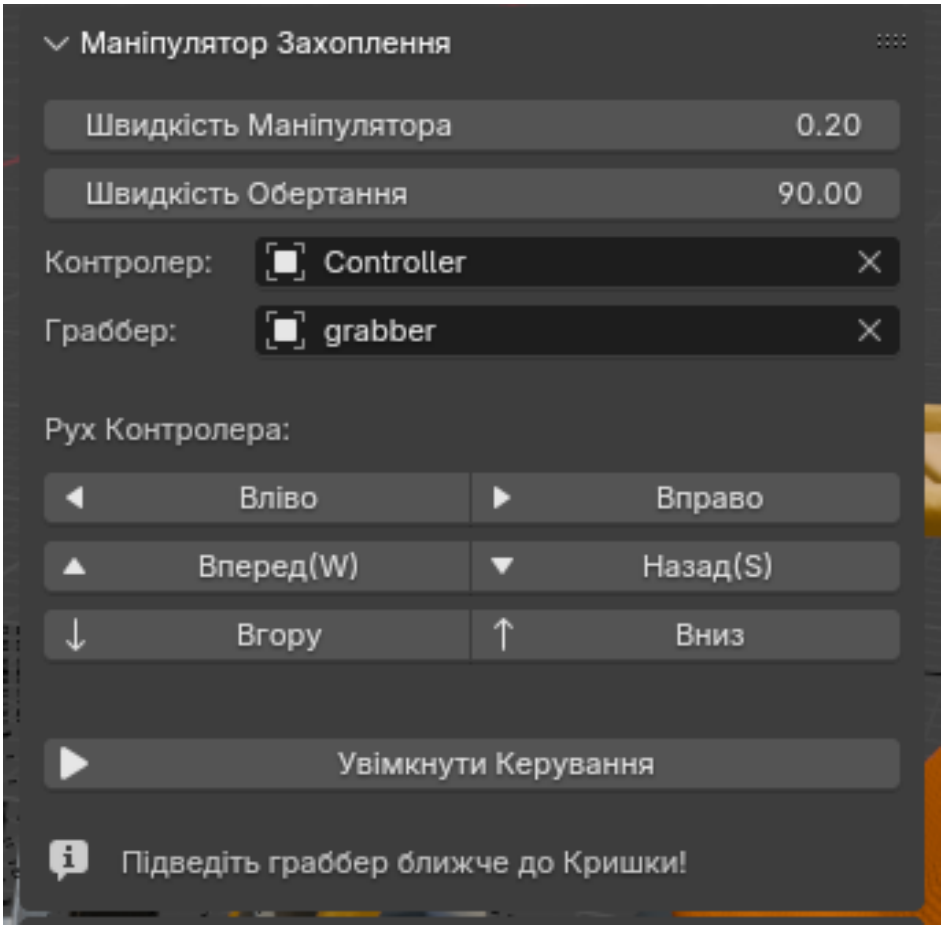

Рисунок 2.20 – Інтерфейс керування маніпулятором захоплення

Інтерфейс було реалізовано через клас VIEW3D_PT_manipulator_panel(), який формує окрему вкладку у боковому меню 3D View. Безпосереднє наповнення інтерфейсу здійснюється у методі def draw(self, context), де за допомогою функцій макету (layout) додаються кнопки керування, повзунки та інші елементи взаємодії з користувачем Панель дозволяє запускати оператори вирівнювання, захоплення,

обертання, а також активувати ручне або автоматичне керування маніпулятором (рис. 2.21)

```
def draw(self, context):
    layout = self.layout
    scene = context.scene

    layout.enabled = not scene.auto_line_running

    if not scene.auto_line_running:
        layout.prop(scene, "manipulator_speed", text="Швидкість Маніпулятора")
        layout.prop(scene, "rotation_speed", text="Швидкість Обертання")
        layout.prop(scene, "manipulator_controller", text="Контролер")
        layout.prop(scene, "grabber_object", text="Граббер")
    else:
        layout.label(text=f"Швидкість Маніпулятора: {scene.manipulator_speed:.2f}")
        layout.label(text=f"Швидкість Обертання: {scene.rotation_speed:.2f}")
        layout.label(text=f"Контролер: {scene.manipulator_controller.name if scene.manipulator_controller e
        layout.label(text=f"Граббер: {scene.grabber_object.name if scene.grabber_object else '–'}")


    layout.separator()
    layout.label(text="Рух Контролера:")
    row = layout.row(align=True)
    row.operator("wm.move_manipulator_left", text="Вліво", icon='TRIA_LEFT')
    row.operator("wm.move_manipulator_right", text="Вправо", icon='TRIA_RIGHT')
    row = layout.row(align=True)
    row.operator("wm.move_manipulator_forward", text="Вперед(W)", icon='TRIA_UP')
    row.operator("wm.move_manipulator_backward", text="Назад(S)", icon='TRIA_DOWN')
    row = layout.row(align=True)
    row.operator("wm.move_manipulator_up", text="Вгору", icon='SORT_DESC')
    row.operator("wm.move_manipulator_down", text="Вниз", icon='SORT_ASC')
    layout.separator()
    #layout.operator("wm.reset_manipulator_state", text="🔄 Скинути Стан")
    layout.separator()
    if not scene.manipulator_keyboard_control_running:
        layout.operator("wm.start_manipulator_control", text="Увімкнути Керування", icon='PLAY')
    else:
        layout.operator("wm.stop_manipulator_control", text="Вимкнути Керування", icon='PAUSE')
    layout.separator()
    ctrl = scene.manipulator_controller
    if not ctrl:
        layout.label(text="❗ Виберіть контролер!")
        return
```

Рисунок 2.21 – Реалізація інтерфейсу через VIEW3D_PT_manipulator_panel() а саме def draw(self, context)

Панель дозволяє запускати оператори вирівнювання, захоплення, обертання, а також активувати ручне або автоматичне керування маніпулятором. В залежності від поточної ситуації в сцені, інтерфейс динамічно адаптується, показуючи лише ті дії, які можливі саме у цей момент.

У структурі інтерфейсу реалізовано три основні логічні гілки:

– якщо до маніпулятора вже прикріплено банку з кришкою – активується опція вирівнювання до коробки (wm.align_to_box) та кнопка “Помістити банку” (wm.place_jar_in_box), за умови, що відстань до ящика не перевищує допустимий поріг;

– якщо в руці лише кришка – і її підводять до банки – при наближенні маніпулятора до банки стає доступною опція вирівнювання (wm.align_to_jar). Після

цього відбувається перевірка наявності рідини в банці: через різницю координат між базою банки та об'єктом honey_cutter визначається, чи банка заповнена. Якщо так – з’являється кнопка “Закрутити кришку” (wm.rotate_banka); якщо ні – користувача попереджають про відсутність рідини;

– якщо кришка ще не захоплена – інтерфейс перевіряє, чи маніпулятор наближено до доступної кришки. При відповідній відстані активуються команди для вирівнювання (wm.align_to_banka) та захоплення (wm.grab_banka). Якщо ж маніпулятор розташовано надто далеко – виводиться інформаційне повідомлення про необхідність наближення (рис. 2.22).

```
# якщо банка в руці і к ній прикручена кришка
if attached_krishka:
    has_kryshka = any(child.name.lower().startswith("banka") for child in attached_krishka.children)
    if has_kryshka:
        box = find_nearest_object("box")
        if box and (box.location - ctrl.location).length < BOX_PLACE_DISTANCE:
            layout.separator()
            layout.label(text="📦 Ящик поруч:")
            layout.operator("wm.align_to_box", text="🎯 Вирівнятись до Ящика")
            if box and (box.location - ctrl.location).length < 0.15:
                layout.operator("wm.place_jar_in_box", text="📦 Помістити банку")

        else:
            layout.label(text=" 🔍 Ящик далеко або не знайдено!")

# кришка в руці – підводимо до банки
elif attached_kryshka:
    if krishka:
        dist = (krishka.location - ctrl.location).length
        if dist < PLACE_DISTANCE:
            layout.operator("wm.align_to_jar", text="🎯 Вирівняти по Банці")

            dist = (krishka.location - ctrl_offset).length
            if dist < MIN_DIS:
                # перевірка діапазану між катерром і банкою
                has_honey = False
                if krishka:
                    base_z = krishka.matrix_world.translation.z
                    for child in krishka.children:
                        if child.name.lower().startswith("honey_cutter"):
                            z_world = child.matrix_world.translation.z
                            delta_z = z_world - base_z
                            if 0.9 <= delta_z <= 1.1:
                                has_honey = True
                                break

                if has_honey:
                    layout.operator("wm.rotate_banka", text="🔄 Закрутити Кришку")
                else:
                    layout.label(text=" У цієї банки нема меду")

        else:
            layout.label(text="Підведіть граббер ближче до Банки!", icon='INFO')
    else:
        layout.label(text="⚠ Банка не знайдена! Додайте нову.")

# кришка ще не захвачена
else:
    kryshka = find_nearest_object("banka")
    if kryshka:
        dist = (kryshka.location - ctrl_offset).length
        if dist < CATCH_DISTANCE:
            layout.operator("wm.align_to_banka", text="🎯 Вирівняти до Кришки")

            dist = (kryshka.location - ctrl_offset).length
            if dist < MIN_DIS:
                layout.operator("wm.grab_banka", text="🤚 Захопити Кришку")
        else:
            layout.label(text=" Підведіть граббер ближче до Кришки!", icon='INFO')
    else:
```

Рисунок 2.22 – Фрагмент коду VIEW3D_PT_manipulator_panel() для виводу необхідних кнопок залежачи від ситуації

Завдяки такій структурі, користувачу не потрібно вручну визначати порядок дій – панель самостійно керує логікою взаємодії, надаючи лише актуальні опції. Це суттєво підвищує зручність використання симулятора і зменшує кількість помилкових дій.

2.3.4 forklift.py – Модуль керування навантажувачем та обробки ящиків

У рамках моделювання виробничої лінії необхідно було врахувати не лише процеси заливки, захоплення та переміщення банок, але й імітувати логістичну частину – управління пакуванням і транспортуванням готової продукції. Для цього було реалізовано окремий програмний модуль forklift.py, який виконує функцію віртуального навантажувача.

Цей модуль відповідає за кілька важливих завдань:

– контроль за заповненістю ящиків банками з рідиною;

– визначення моменту, коли потрібно замінити повні коробки на нові;

– візуалізацію процесу доставки порожніх ящиків у сцену;

– автоматичне поповнення запасів кришок, що необхідні для подальшого складання продукції.

Таким чином, forklift.py доповнює логіку симуляції, дозволяючи змоделювати повний виробничий цикл, включаючи підготовку до наступних етапів. Його робота не тільки візуально збагачує сцену, а й підтримує стабільність роботи інших модулів, які потребують наявності вільних коробок або достатньої кількості кришок.

Загальна ідея полягає в тому, що після заповнення усіх доступних ящиків система активує віртуальний навантажувач, який очищує сцену від старих банок і доставляє нові порожні контейнери, готові до подальшої роботи. Це дозволяє уникнути “зупинки лінії” у випадках, коли всі місця для пакування зайняті.

Контроль заповненості ящиків Було реалізовано class OBJECT_PT_forklift_panel() а саме у def draw(self, context), який визначає, скільки ящиків у сцені вже заповнено банками (krishka), і скільки залишаються вільними. Для цього аналізується наявність дочірніх об'єктів у кожному контейнері з іменем

box. Якщо хоча б один внутрішній об'єкт присутній – ящик вважається зайнятим. Це дає змогу оцінювати стан системи пакування в реальному часі (рис. 2.23).

```
# Рахуємо усі банки та позиції
box_positions = [obj for obj in bpy.data.objects if obj.name.startswith("box")]

occupied = 0
for box in box_positions:
    for child in box.children:
        if child.name.startswith("krishka") and child.type == 'MESH':
            occupied += 1
            break

total_boxes = len(box_positions)
free = max(total_boxes - occupied, 0)
layout.label(text=f"Вільно: {free} із {total_boxes}")
```

Рисунок 2.23 – Фрагмент OBJECT_PT_forklift_panel(), def draw(self, context)

Доставка нових ящиків і очищення повних. Реалізовано анімовану симуляцію дій навантажувача – об'єкта Forklift, який виконує очищення ящиків і заміну на нові. Його поведінка керується оператором OBJECT_OT_process_crates, що реалізує трьохфазний алгоритм:

– фаза 1 – від'їзд назад. Об'єкт Forklift переміщується назад по осі Y, ніби вирушаючи за новими порожніми ящиками. Рух реалізовано через зміну координат у глобальному просторі;

– фаза 2 – очищення заповнених ящиків. Після завершення фази руху виконується очищення кожного об'єкта box від усіх дочірніх елементів (банок, кришок). Це моделює процес вивезення заповненої продукції;

– фаза 3 – повернення назад. Після очищення навантажувач повертається у вихідну позицію, створюючи ілюзію доставки нових порожніх ящиків, готових до подальшого використання (рис. 2.24).

```
_timer = None
_start_location = None
_step = 0
_direction = 1
_start_time = 0
_deleted = False

def modal(self, context, event):
    if event.type != 'TIMER':
        return {'PASS_THROUGH'}

    obj = bpy.data.objects.get(FORKLIFT_NAME)
    if not obj:
        self.report({'ERROR'}, f"Объект {FORKLIFT_NAME} не знайдено")
        return {'CANCELLED'}

    #Фаза 1: від'їзд назад
    if self._phase == "depart":
        local_step = obj.matrix_world.to_3x3() @ Vector((0, -0.15, 0))
        obj.location += local_step
        self._step += 1
        if self._step >= 70:
            self._phase = "cleanup"

    #Фаза 2: видалення об'єктів
    elif self._phase == "cleanup":
        delete_nested_objects()
        self._phase = "return"

    #Фаза 3: повернення назад
    elif self._phase == "return":
        local_step = obj.matrix_world.to_3x3() @ Vector((0, 0.15, 0))
        obj.location += local_step
        self._step_return += 1
        if self._step_return >= 70:
            context.window_manager.event_timer_remove(self._timer)
            return {'FINISHED'}

    return {'RUNNING_MODAL'}
```

Рисунок 2.24 – Оператор OBJECT_OT_process_crates, що реалізує трьохфазний алгоритм

Поповнення кришок. Модуль також відповідає за підтримання запасів кришок (banka) у зоні зберігання. У спеціально відведеному об'єкті boox перевіряється, скільки кришок знаходиться у кожній вертикальній стопці. Якщо кількість менша за 10, система автоматично створює відсутню кількість копій, дотримуючись фіксованого кроку по осі Z (z_offset). Функція fill_stacks() забезпечує поповнення заздалегідь визначених позицій на місці їх знаходження (рис. 2.25).

```python
#Отримання об'єкта контейнера boox, в якому зберігаються кришки.
def fill_stacks(box_name="boox", cap_prefix="banka", z_offset=0.19):
    box = bpy.data.objects.get(box_name)
    if not box:
        print(f"[!] Box '{box_name}' не знайдено")
        return

#Пошук шаблону кришки (banka) для копіювання.
    cap_template = None
    for obj in bpy.data.objects:
        if obj.name.startswith(cap_prefix):
            cap_template = obj
            break
    if not cap_template:
        print(f"[!] Не знайдено шаблон кришки з префіксом '{cap_prefix}'")
        return

    collection_name = "Krishki"
    if collection_name in bpy.data.collections:
        target_col = bpy.data.collections[collection_name]
    else:
        target_col = bpy.data.collections.new(collection_name)
        bpy.context.scene.collection.children.link(target_col)

#Кожна кришка, що знаходиться в межах boox, групується по XY-координатах, щоб сформувати вертикальні стопки.
    stacks = {}
    for obj in bpy.data.objects:
        if obj.name.startswith(cap_prefix) and is_inside_box(obj, box):
            xy = round_xy(obj.location)
            stacks.setdefault(xy, []).append(obj)

#Якщо в стопці менше ніж 10 кришок — створюються копії і розміщуються одна над одною з кроком z_offset.
#Нові об’єкти додаються в колекцію Krishki.
    for xy, caps in stacks.items():
        caps.sort(key=lambda c: c.location.z)
        count = len(caps)
        missing = 10 - count
        if missing <= 0:
            continue

        top_z = caps[-1].location.z if caps else 0

        for i in range(missing):
            new_cap = cap_template.copy()
            new_cap.data = cap_template.data.copy()
            new_cap.location = Vector((xy[0], xy[1], top_z + z_offset * (i + 1)))
            target_col.objects.link(new_cap)  # добавляємо в коллекцію
```

Рисунок 2.25 – Функція def fill_stacks()

Інтерфейс користувача. Для взаємодії з модулем було реалізовано окрему панель у Sidebar інтерфейсу Blender через OBJECT_PT_forklift_panel у методі draw() . У ній користувачу надається доступ до наступних функцій:

– перегляд поточного стану заповненості ящиків;

– запуск анімації очищення та доставки нових ящиків;

– ручне поповнення кришок;

– інформування про помилки, наприклад, відсутність об'єкта boox у сцені (рис. 2.26).

```python
class OBJECT_PT_forklift_panel(bpy.types.Panel):
    bl_label = "Лічильник та Погружчик"
    bl_idname = "OBJECT_PT_forklift_panel"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = 'Керування Лінією'

    def draw(self, context):
        layout = self.layout
        scene = context.scene

        layout.enabled = not scene.auto_line_running

        # Рахуємо усі банки та позиції
        box_positions = [obj for obj in bpy.data.objects if obj.name.startswith("box")]

        occupied = 0
        for box in box_positions:
            for child in box.children:
                if child.name.startswith("krishka") and child.type == 'MESH':
                    occupied += 1
                    break

        total_boxes = len(box_positions)
        free = max(total_boxes - occupied, 0)
        layout.label(text=f"Вільно: {free} із {total_boxes}")

        if occupied >= 1:
            layout.operator("object.process_crates", text="Привезти вільні ящики", icon='ADD')
        else:
            layout.label(text=" Усі ящики вільні, немає сенсу приносити нові", icon='INFO')
        layout.separator()

        #Перевірка стопок у ящика boox
        box = bpy.data.objects.get("boox")
        if not box:
            layout.label(text="[!] Не знайдено ящик 'boox'")
            return

        stacks = {}
        for obj in bpy.data.objects:
            if obj.name.startswith("banka") and is_inside_box(obj, box):
                xy = round_xy(obj.location, precision=0.1)
                stacks.setdefault(xy, []).append(obj)

        all_filled = all(len(stack) >= 10 for stack in stacks.values())

        if all_filled and stacks:
            layout.label(text=" Всі стопки заповнені", icon='INFO')
        else:
            layout.operator("object.fill_stacks_from_base", text="Доповнити кришки", icon='ADD')
```

Рисунок 2.26 – Класс OBJECT_PT_forklift_panel та метод draw() для реалізації інтерфейсу та логіки кнопок

Кнопки активуються лише тоді, коли виконуються відповідні умови (наприклад, повна зайнятість усіх ящиків), що дозволяє уникнути помилкових дій [22-23].

2.3.5 Реалізація інверсної кінематики (ІК) для маніпулятора

Для забезпечення плавного та реалістичного руху маніпулятора у віртуальному середовищі було застосовано метод інверсної кінематики (ІК), який дозволяє керувати рухами кінцівок за допомогою цільового об’єкта. Замість ручного обертання кожної кістки окремо, за допомогою ІК достатньо перемістити

контрольну точку (target), після чого всі інші сегменти автоматично обчислюють свої положення, дотримуючись заданих обмежень.

Структура та налаштування

У сцені використовується арматура з декількома кістками, що моделює сегменти промислового робота. Кінцева кістка (Bone.005) є основною для застосування інверсної кінематики. До неї було додано обмежувач типу IK (Inverse Kinematics) через вкладку властивостей кісток.

Основні параметри налаштування:

– ціль (Target) – об’єкт– пустушка (Empty), який виступає в ролі контрольної точки для напрямку руху;

– довжина ланцюга (Chain Length) – 2, що означає: впливають дві кістки, починаючи від кінцевої;

– кількість ітерацій – 500, для досягнення високої точності обчислення;

– вплив позиції та обертання – встановлено на 1.0, що гарантує повне підпорядкування положенню цілі;

– використання кінця кістки – активовано;

– розтягування – дозволене, щоб забезпечити плавні рухи навіть при неідеальному розміщенні сегментів (рис. 2.27).

Після встановлення цілі та застосування IK–обмежувача, рух маніпулятора виконується через переміщення цільового об’єкта, що автоматично обчислює положення відповідних кісток. Завдяки цьому забезпечується реалістична поведінка механізму, подібна до роботи справжнього промислового робота. IK–система дозволяє:

– швидко позиціонувати граббер маніпулятора в потрібну точку;

– уникати неприродних вигинів;

– автоматично обчислювати оптимальні кути повороту кожного сегмента;

– працювати з декількома кістками без необхідності ручного контролю кожної з них.

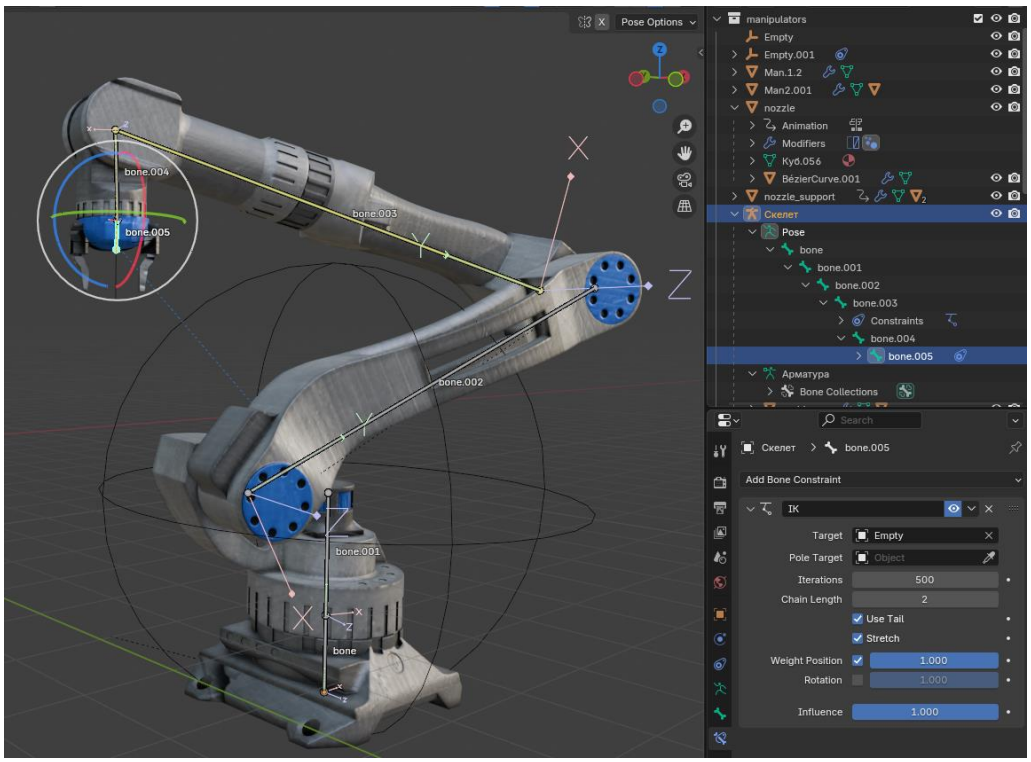

Рисунок 2.27 – Відкритий список з ієрархією кісток та модифікатору інверсної кінематики

Цільовий об’єкт IK може також змінювати своє положення під час виконання команд скрипта, наприклад, при вирівнюванні до банки або коробки (align_to_object). Таким чином, IK інтегрується у загальну систему координування рухів, що робить маніпулятор не просто анімованим, а функціонально керованим інструментом у складі тренажера.

Використання інверсної кінематики дало змогу досягти високої гнучкості та реалістичності руху маніпулятора без потреби в складних анімаціях. Це рішення є особливо ефективним для тренажерів, які імітують роботу роботизованих систем, де важливо забезпечити точність позиціонування, природність руху та зручність керування.

У процесі налаштування інверсної кінематики (IK) було виявлено, що при переміщенні цільового об’єкта кінцівка маніпулятора іноді виконує некоректне обертання навколо своєї осі, що призводить до неприродного згинання всієї руки. Така поведінка обумовлена відсутністю або неправильним позиціюванням додаткової направляючої осі – Pole Target (рис. 2.28).

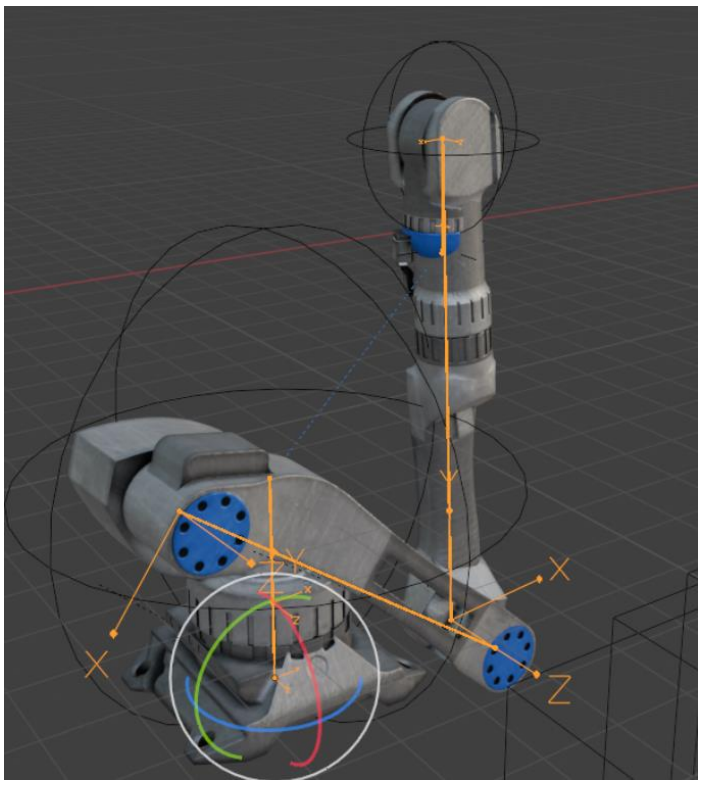

Рисунок 2.28 – Випадок некоректної поведінки ІК

Система інверсної кінематики (ІК) виконує обчислення, що дозволяють кінцевій точці ланцюга кісток досягти певної цілі $\vec{T}$, з урахуванням геометричних обмежень. Основною задачею є обчислення кута згину ланцюга, проте при відсутності орієнтовного вектора площини згину (наприклад, через відсутність Pole Target) виникає неоднозначність, що призводить до візуального викривлення.

Визначення напрямних векторів

Нехай:

– $\overrightarrow{B_0}$ – координати першої точки (основа плеча);

– $\overrightarrow{B_1}$ – координати суглоба (лікоть);

– $\overrightarrow{B_2}$ – координати кінця ланцюга (захват);

– $\vec{T}$– цільова точка, до якої потрібно дотягтись.

Обчислюються вектори:

$$\vec{A} = \overrightarrow{B_1} - \vec{B}_0 \tag{2.1}$$

$$\vec{B} = \overrightarrow{B_2} - \vec{B}_1 \tag{2.2}$$

Обчислення кута згину. Фізичний кут згину θ визначається за допомогою скалярного добутку:

$$\theta = \arccos\left(\frac{\vec{A} \cdot \vec{B}}{|\vec{A}| \cdot |\vec{B}|}\right) \tag{2.3}$$

Це значення описує поточне згинання між кістками.

Побудова площини згину. Площина, в якій відбувається згин, задається векторною нормаллю:

$$[\vec{n} = \vec{A} \times \vec{B}] \tag{2.4}$$

де $(\times)$ – векторний добуток. Вектор $(\vec{n})$ перпендикулярний до площини згину.

Визначення викривлення. При переміщенні цілі $\vec{T}$ площина згину може змінити орієнтацію, особливо якщо відсутній Pole Target. Щоб оцінити рівень викривлення, порівнюються нормалі до і після зміщення:

$$\delta = \arccos\left(\frac{\overrightarrow{n_1} \cdot \overrightarrow{n_2}}{|\overrightarrow{n_1}| \cdot |\overrightarrow{n_2}|}\right) \tag{2.5}$$

де $(\overrightarrow{n_1})$ – нормаль до згину до переміщення;

$(\overrightarrow{n_2})$ – нормаль після переміщення.

Якщо $\delta \approx \pi$ – відбулось перевертання площини, яке візуально сприймається як викривлення маніпулятора.

Використання Pole Target для стабілізації. Для уникнення геометричної неоднозначності вводиться точка орієнтації – Pole Target, розташована у просторі на координатах $(\vec{P})$. Вона задає вектор напрямку:

$$\vec{D} = \vec{P} - \overrightarrow{B_1} \quad (2.6)$$

І контролює орієнтацію площини згину:

$$\text{sign} = \text{sign}\left(\vec{n} \cdot \vec{D}\right) \quad (2.7)$$

Цей знак дозволяє визначити, чи площина згину спрямована у правильний бік. При його зміні система автоматично перестрибує на інший варіант згину, що і спричиняє ефект викривлення.

Система інверсної кінематики без Pole Target є математично недостатньо визначеною: площина згину обирається довільно, на основі попереднього положення або внутрішньої евристики Blender. Через це можливі раптові стрибки у положенні нормалі $(\vec{n})$, що призводить до візуального викривлення – скручування або «зламання» кінцівки.

Запровадження орієнтовної точки (Pole Target) дозволяє стабілізувати геометрію та уникнути викривлень навіть при складному русі цілі [24-25].

2.4 Алгоритм роботи автоматизованої лінії

У модулі automation_controller.py було реалізовано повністю автоматизований алгоритм керування виробничим процесом. Його логіка ґрунтується на послідовному виконанні станів (етапів), які змінюються за допомогою змінної _stage. Весь процес реалізовано у вигляді модального оператора Blender, що активується через таймер і циклічно оновлює стан системи в реальному часі.

Робота модуля побудована на основі раніше створених допоміжних аддонів, які відповідають за окремі функціональні частини автоматизованої лінії: рух конвеєра, заповнення банок рідиною, дії маніпулятора, сортування банок у коробки тощо. Зокрема, для керування заливкою рідиною, маніпуляцією з об’єктами, транспортуванням та виявленням об'єктів використовуються такі оператори як

fill_honey, grab_banka, rotate_banka, place_jar_in_box, які були попередньо реалізовані в окремих модулях системи.

Завдяки інтеграції з цими аддонами, головний контролер виконує роль координаційного центру, який послідовно активує відповідні частини системи згідно з виробничим сценарієм

Перед запуском автоматизованої лінії користувач задає необхідну кількість банок, які потрібно обробити. Для цього у графічному інтерфейсі Blender передбачене відповідне числове поле, також. Після цього натискається кнопка запуску процесу (Почати), яка активує оператор AutoOperator(bpy.types.Operator) (рис. 2.29).

```
69
70  class AutoOperator(bpy.types.Operator):
71      bl_idname = "wm.run_auto_operator"
72      bl_label = "Автоматичний Процес"
73
74      _timer = None
75      _stage = 0
76      _jar_counter = 0
77      _jar_total = 0
78      _move_count = 0
79      _target = None
80      _controller = None
81      _delay = 0
82      _rotated = False
83      _stopped_by_user = False
84
85      def modal(self, context, event):
86          if not context.scene.auto_line_running:
87              return self.cancel(context)
88
89          if event.type != 'TIMER':
90              return {'PASS_THROUGH'}
91
92          if self._delay > 0:
93              self._delay -= 1
94              return {'RUNNING_MODAL'}
```

Рисунок 2.29 – Оператор AutoOperator(bpy.types.Operator)

Від цього моменту управління передається системі автоматизації, яка далі функціонує без втручання користувача.

Увесь процес поділено на етапи, реалізовані у вигляді станів, що послідовно змінюються в залежності від поточної ситуації на сцені. Опис кожного з них наведено нижче.

Основні етапи роботи автоматизованої лінії. Було реалізовано повністю автоматизований процес керування виробничим циклом у середовищі Blender. Вся

взаємодія з користувачем організована через графічний інтерфейс – спеціальну панель AutoPanel, розташовану у правій частині вікна 3D View (рис. 2.30).

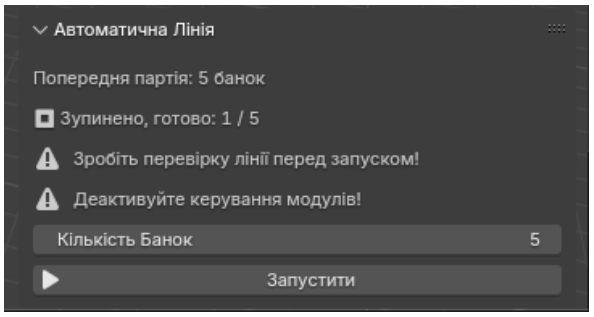

Рисунок 2.30 – Інтерфейс модуля автоматичної лінії

Інтерфейс дозволяє задавати ключові параметри запуску: зокрема, кількість банок, що мають бути оброблені, вводиться через числове поле, розташоване у верхній частині панелі. Після цього користувач активує процес, натискаючи кнопку «Запустити», що викликає модальний оператор AutoOperator (рис. 2.31).

```
class AutoPanel(bpy.types.Panel):
    bl_label = "Автоматична Лінія"
    bl_idname = "VIEW3D_PT_auto_line"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "Керування Лінією"

    def draw(self, context):
        layout = self.layout
        scene = context.scene

        if scene.auto_line_last_batch > 0:
            layout.label(text=f"Попередня партія: {scene.auto_line_last_batch} банок")

        if scene.auto_line_running:
            layout.label(text=f"Готово: {scene.auto_line_jar_done} / {scene.auto_line_total_jars_global}")
        elif scene.auto_line_total_jars_global > 0 and scene.auto_line_jar_done > 0:
            layout.label(text=f" Зупинено, готово: {scene.auto_line_jar_done} / {scene.auto_line_total_jars_global}")
            layout.label(text="Зробіть перевірку лінії перед запуском!", icon='ERROR')
            layout.label(text="Деактивуйте керування модулів!", icon='ERROR')
        else:
            layout.label(text="Зробіть перевірку лінії перед запуском!", icon='ERROR')
            layout.label(text="Деактивуйте керування модулів!", icon='ERROR')

        if not scene.auto_line_running:
            layout.prop(scene, "auto_line_total_jars", text="Кількість Банок")
        else:
            layout.label(text=f"Кількість Банок: {scene.auto_line_total_jars}")

        row = layout.row()
        if not scene.auto_line_running:
            row.operator("wm.run_auto_operator", text="Запустити", icon='PLAY')
        else:
            row.operator("wm.stop_auto_operator", text="Зупинити", icon='CANCEL')


class StopAutoOperator(bpy.types.Operator):
    bl_idname = "wm.stop_auto_operator"
    bl_label = "Зупинити Автомат"

    def execute(self, context):
        context.scene.auto_line_force_cancel = True
        return {'FINISHED'}
```

Рисунок 2.31 – Класс AutoPanel с логікою показу повідомлень та кнопок залежачи від ситуації

Після запуску система переходить у автоматичний режим і розпочинає поетапне виконання дій, які змінюються за допомогою змінної стану _stage. Цикл побудований як модальний оператор із вбудованим таймером, що періодично перевіряє умови на сцені та запускає відповідні дії у потрібний момент. Всі дії, включаючи переміщення конвеєра, заливку рідиною, захоплення, обертання та розміщення банок, відбуваються без участі користувача, але з можливістю зупинки або скидання стану системи через доступні елементи управління на панелі. Таким чином, було реалізовано зручний і гнучкий інтерфейс, який забезпечує запуск і контроль за складним виробничим процесом без необхідності прямої взаємодії з об’єктами сцени.

На початковому етапі, коли змінна стану _stage дорівнює 0, система перевіряє готовність до заповнення банки рідиною. Для цього викликається функція can_fill_honey(), яка аналізує положення аплікатора.

Якщо аплікатор не знаходиться в робочому положенні (вище певного рівня по осі Z), то активується оператор move_conveyor_forward, який зміщує конвеєр на один крок уперед. Таким чином, система автоматично підводить банку до зони розливу, очікуючи момент, коли заливку буде дозволено (рис. 2.32).

```
try:
    if self._stage == 0:
        if not can_fill_honey():
            bpy.ops.wm.move_conveyor_forward()
        else:
            self._stage = 1
```

Рисунок 2.32 – Фрагмент AutoOperator(), stage 0

Після того як аплікатор досягне необхідної позиції, система переходить до стану _stage == 1, де запускається оператор fill_honey.

У цей момент запускається візуалізація процесу заливання рідини в банку, а також ініціалізується змінна fill_progress, яка буде поступово зменшуватися в ході анімації (рис. 2.33).

```
elif self._stage == 1:
    bpy.ops.wm.fill_honey()
    self._stage = 2
```

Рисунок 2.33 – Фрагмент AutoOperator(), stage 1

Коли заливка розпочата, система переходить до стану _stage == 2, де вона очікує завершення процесу. Це очікування реалізується шляхом перевірки, чи fill_progress дорівнює нулю. Поки цей параметр зменшується, система перебуває в режимі паузи (рис. 2.34).

```
elif self._stage == 2:
    if context.scene.fill_progress == 0:
        self._move_count = 0
        self._stage = 3
```

Рисунок 2.34 – Фрагмент AutoOperator(), stage 2

Після завершення заливки активується наступний стан – _stage == 3. На цьому етапі конвеєр додатково зсувається вперед, щоб перемістити банку ближче до маніпулятора. Після цього обчислюється орієнтація маніпулятора та ініціюється рух до контрольної точки вирівнювання, відомої як cap_align_point. Коли об’єкт досягає цієї позиції з допустимою точністю, викликається оператор align_to_banka, що відповідає за орієнтацію захватного пристрою відносно банки. Після невеликої затримки система готується до наступного кроку (рис. 2.35).

```
elif self._stage == 3:
    if self._move_count < 100:
        bpy.ops.wm.move_conveyor_forward()
        self._move_count += 1

        self._controller = context.scene.manipulator_controller
        self._target = self._get_target("banka")

        self._target = bpy.data.objects.get("cap_align_point")
        if self._controller and self._target:
            aligned = move_object_straight(self._controller, self._target.location, 0.4)
            if aligned:
                bpy.ops.wm.align_to_banka()
                self._delay = 5
                self._stage = 42
```

Рисунок 2.35 – Фрагмент AutoOperator(), stage 3

На етапі _stage == 4 виконується безпосередньо захоплення банки. Для цього застосовується оператор grab_banka, який активує механізм фіксації банки на маніпуляторі. Цей процес моделює фізичне захоплення об’єкта у реальному виробництві (рис. 2.36).

```
elif self._stage == 4:
    bpy.ops.wm.grab_banka()
    self._target = self._get_target("krishka")
    self._rotated = False
    self._stage = 5
```

Рисунок 2.36 – Фрагмент AutoOperator(), stage 4

Далі, при переході до стану _stage == 5, маніпулятор прямує до кришки. Після досягнення об’єкта, що відповідає за імітацію кришки, виконується перевірка: чи було успішно завершено заливання рідини та чи банка знаходиться у захваті. Якщо всі умови виконано, ініціюється оператор rotate_banka, який обертає банку, імітуючи процес закручування кришки (рис. 2.37).

```
elif self._stage == 5:
    if self._controller and self._target:
        aligned = move_object_straight(self._controller, self._target.location + Vector((0, -0.1, 2.2)), 0.4)
        if aligned and not self._rotated:
            bpy.context.view_layer.update()
            if is_close(self._controller.location, self._target.location + Vector((0, -0.1, 2.2))) and has_hon
                context.scene.auto_rotation_running = True
                bpy.ops.wm.rotate_banka()
                self._rotated = True
                self._delay = 5
                self._stage = 6
```

Рисунок 2.37 – Фрагмент AutoOperator(), stage 5

У наступному стані – _stage == 6 – задається пауза. Це зроблено для стабілізації та візуального завершення попередньої дії. Після завершення затримки система визначає найближчу вільну коробку серед об'єктів типу box та готується до переміщення (рис. 2.38).

```
elif self._stage == 6:
    self._delay = 140
    self._target = self._get_target("box")
    self._expected_rotation = None
    self._stage = 7
```

Рисунок 2.38 – Фрагмент AutoOperator(), stage 6

Під час стану _stage == 7 виконується рух маніпулятора до точки align_box_point. Ця точка визначає точну позицію над коробкою, в яку планується помістити банку. Коли координати збігаються з допустимою похибкою, викликається оператор align_to_box, який орієнтує банку для точного розміщення (рис. 2.39).

```
elif self._stage == 7:
    self._target = bpy.data.objects.get("align_box_point")
    if self._controller and self._target:
        aligned = move_object_straight(self._controller, self._target.location, 0.4)
        if aligned:
            bpy.ops.wm.align_to_box()
            self._delay = 5
            self._stage = 8
```

Рисунок 2.39 – Фрагмент AutoOperator(), stage 7

У стані _stage == 8 відбувається саме розміщення банки в коробку. Це реалізується через оператор place_jar_in_box, який додає банку до колекції об’єктів коробки. Також інкрементується внутрішній лічильник кількості оброблених банок. Якщо кількість досягла кратного значення (наприклад, 6), то додатково викликається fill_stacks_from_base – оператор, який відповідає за оновлення візуального стану коробки, зокрема за викладення кришок (рис. 2.40).

```
elif self._stage == 8:
    try:
        bpy.ops.wm.place_jar_in_box()
        self._jar_counter += 1
        context.scene.auto_line_jar_done += 1

        if self._jar_counter % 6 == 0:
            bpy.ops.object.fill_stacks_from_base()

        self._returning_home = True
        self._stage = 9

    except Exception as e:
        self.report({'ERROR'}, f"Помилка при розміщенні банки: {e}")
        return self.cancel(context)
```

Рисунок 2.40 – Фрагмент AutoOperator(), stage 8

На етапі _stage == 9 система перевіряє заповненість усіх доступних коробок. Якщо всі об’єкти типу box вже мають кришки, активується оператор process_crates, який символізує завершення заповнення і виконує фінальні операції, пов’язані з пакуванням (рис. 2.41).

```
elif self._stage == 9:
    boxes = [obj for obj in bpy.data.objects if obj.name.startswith("box")]
    occupied = 0
    for box in boxes:
        for child in box.children:
            if child.name.startswith("krishka"):
                occupied += 1
                break

    if occupied >= len(boxes):
        bpy.ops.object.process_crates()

    if self._jar_counter >= self._jar_total:
        self._returning_home = True
        self._stage = 10

    else:
        self._stage = 0 #початок нового циклу
```

Рисунок 2.41 – Фрагмент AutoOperator(), stage 9

Завершальним є стан _stage == 10, який означає завершення всієї партії. У цьому стані маніпулятор повертається до початкової позиції, параметри партії скидаються, і система готова до нового циклу (рис. 2.42).

```
    elif self._stage == 10:
        if not self._returning_home:
            def delayed_fill():
                try:
                    bpy.ops.object.fill_stacks_from_base()
                except Exception as e:
                    print("❌ Помилка при виклику fill_stacks_from_base:", e)
                return None

            bpy.app.timers.register(delayed_fill, first_interval=0.1)

            self.report({'INFO'}, f"✅ Автомат завершив роботу: {self._jar_total} банок виконано.")
            if not self._stopped_by_user:
                context.scene.auto_line_last_batch = self._jar_total
                context.scene.auto_line_total_jars_global = 0
                context.scene.auto_line_jar_done = 0

            return self.cancel(context)


except Exception as e:
    self.report({'ERROR'}, f"Автомат зупинено через помилку: {e}")
    return self.cancel(context)

if self._returning_home:
        done = move_object_straight(self._controller, self._start_position, 0.4)
        if done:
            self._returning_home = False

context.scene.auto_line_stage_last = self._stage

return {'RUNNING_MODAL'}
```

Рисунок 2.42 – Фрагмент AutoOperator(), stage 10

Система забезпечує повністю автономну логіку роботи на основі попередньо створених додатків для керування заливкою, конвеєром, захватом і маніпуляцією об’єктами. Завдяки використанню модального оператора, усі дії виконуються без участі користувача, що імітує поведінку справжньої автоматизованої виробничої лінії.

2.5 Тестування та верифікація логіки роботи

Для перевірки коректності роботи автоматизованої лінії було проведено серію тестів у середовищі Blender. Основною метою тестування було верифікувати, що логіка роботи, реалізована у модальному операторі AutoOperator, відповідає очікуваній поведінці в рамках заданого алгоритму.

Тестування здійснювалося інтерактивно, шляхом запуску повного виробничого циклу з різними параметрами. Було перевірено як мінімальні, так і граничні значення – від однієї до десяти банок. Для кожного випадку спостерігалося, чи система правильно виконує послідовність дій: просування конвеєра, зупинку в зоні заливки, подачу рідини, захоплення банки, обертання, транспортування до коробки та розміщення.

Особливу увагу було приділено перевірці умов переходу між станами (_stage). Зокрема, тестувалося, що кожен стан виконується лише тоді, коли попередній завершено коректно. Також проводилася верифікація роботи операторів, які виконують ключові функції: fill_honey, grab_banka, rotate_banka, place_jar_in_box тощо. Усі оператори успішно виконували свої функції при правильному положенні об'єктів на сцені.

Додатково було протестовано систему на стійкість до помилок сцени. Наприклад, у випадку відсутності об’єкта box або krishka, система не завершувала цикл примусово, а зупинялася або очікувала на коректну конфігурацію, що також є підтвердженням стабільності логіки.

Результати тестування засвідчили, що логіка системи працює відповідно до закладеного алгоритму. Автоматизована лінія коректно реагує на зміну станів,

виконує всі функції у заданому порядку та дозволяє досягти моделювання повного виробничого циклу без зовнішнього втручання. Це підтверджує працездатність розробленої моделі і її придатність для подальших експериментів та інтеграцій.

2.6 Практичні результати симуляції

У результаті симуляції в середовищі Blender було успішно реалізовано повний виробничий цикл автоматизованої лінії. Система виконувала поетапну обробку банок: від подачі на конвеєр до заповнення рідиною, закриття кришками та укладання у тару. Кожна дія системи керувалась через внутрішній стан _stage, що забезпечувало послідовне й логічне виконання сценарію. Поведінка всіх компонентів відповідала заданим умовам, а сама логіка роботи продемонструвала стабільність та узгодженість між усіма етапами.

Перед початком роботи у сцені були підготовлені об’єкти: банки, аплікатор, маніпулятор, кришки та коробки. На рисунку 2.43 представлено загальний вигляд сцени та екран користувача до запуску системи: об'єкти розміщено у вихідних положеннях, а маніпулятор перебуває у початковій позиції очікування (рис. 2.43).

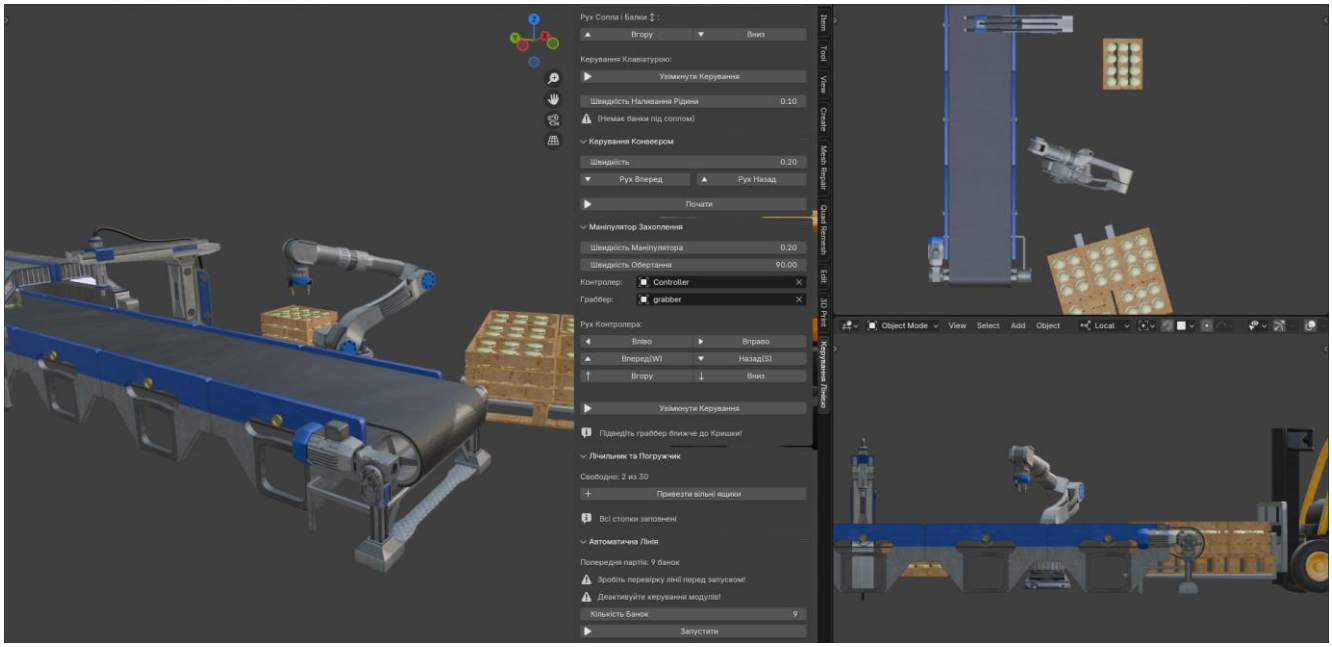

Рисунок 2.43 – Загальний вигляд сцени та екран користувача перед запуском модуля автоматичної лінії

Після введення користувачем кількості банок у відповідному полі інтерфейсу та активації кнопки «Запустити», був викликаний оператор AutoOperator, що запускає модальний цикл автоматизації. Починаючи з цього моменту, всі дії виконувались автоматично. На рисунку 2.44 продемонстровано початок заливки – банка точно позиціонується під аплікатором, після чого активується оператор fill_honey. Система очікує завершення процесу заливки відповідно до змінної fill_progress (рис. 2.44).

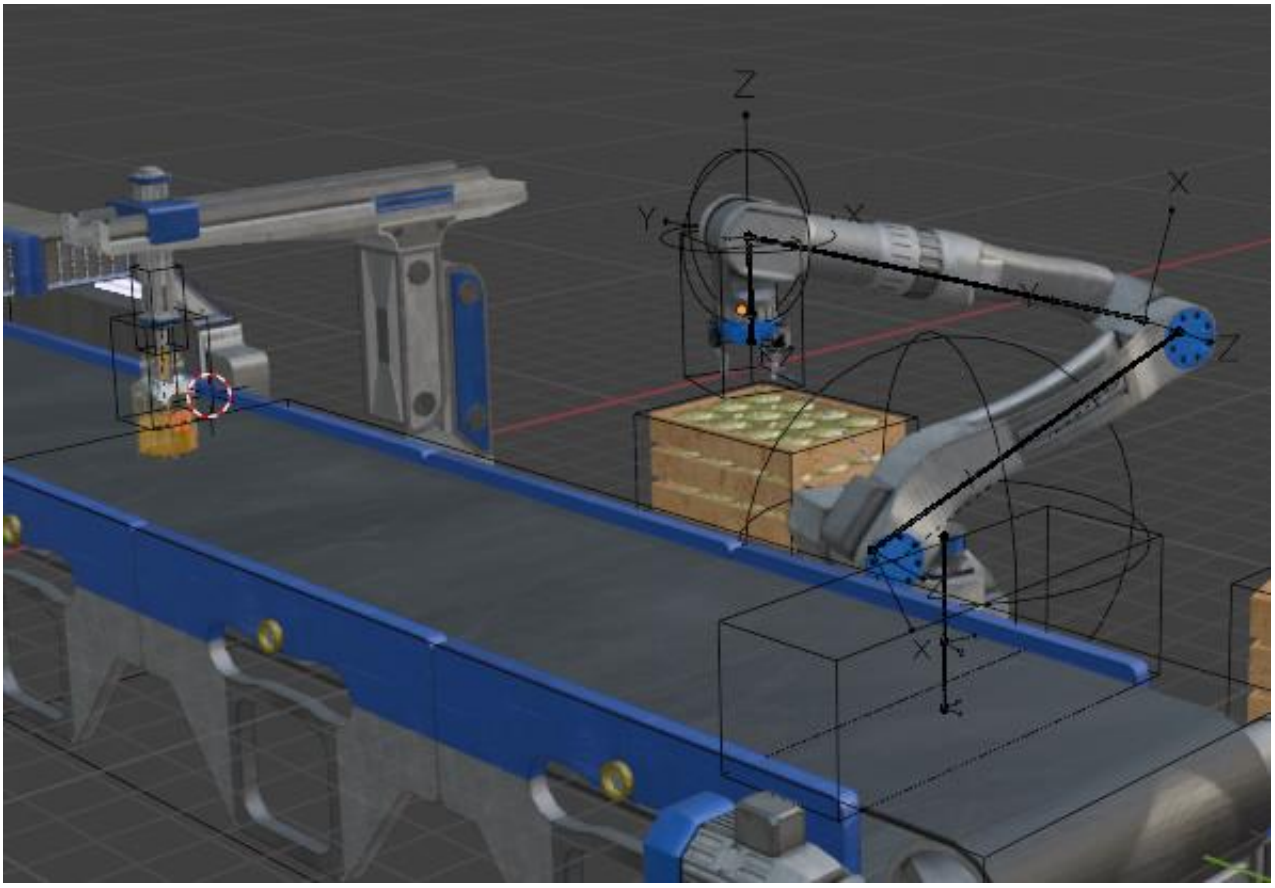

Рисунок 2.44 – Процес заливки рідини в банку

Після цього, згідно з внутрішньою логікою, конвеєр просувається вперед, і маніпулятор починає рух до кришки. Як видно на ілюстрації, маніпулятор вирівнюється відносно кришки, після чого виконується оператор grab_banka для її захоплення. Рухи виконуються з дотриманням точності позиціонування, що забезпечує плавність анімації та реалізм дії (рис. 2.45).

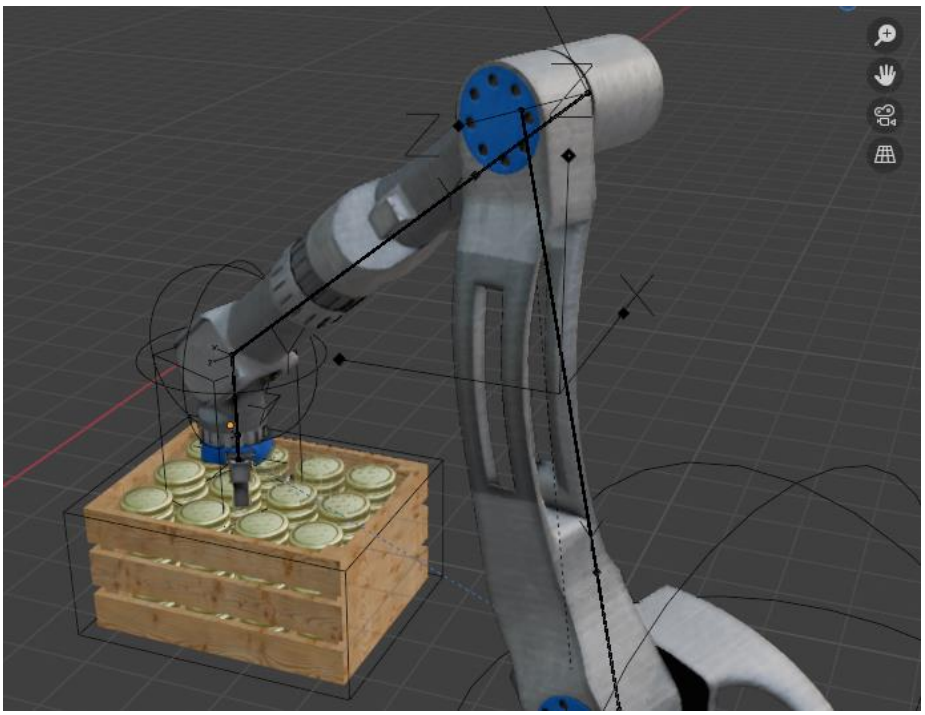

Рисунок 2.45 – Маніпулятор захоплює кришку

У наступній фазі маніпулятор транспортує кришку до зони закручування. При досягненні контрольної точки активується оператор rotate_banka, який ініціює обертання банки для імітації встановлення кришки. Цей етап дозволяє візуалізувати складну координацію рухів маніпулятора (рис. 2.46).

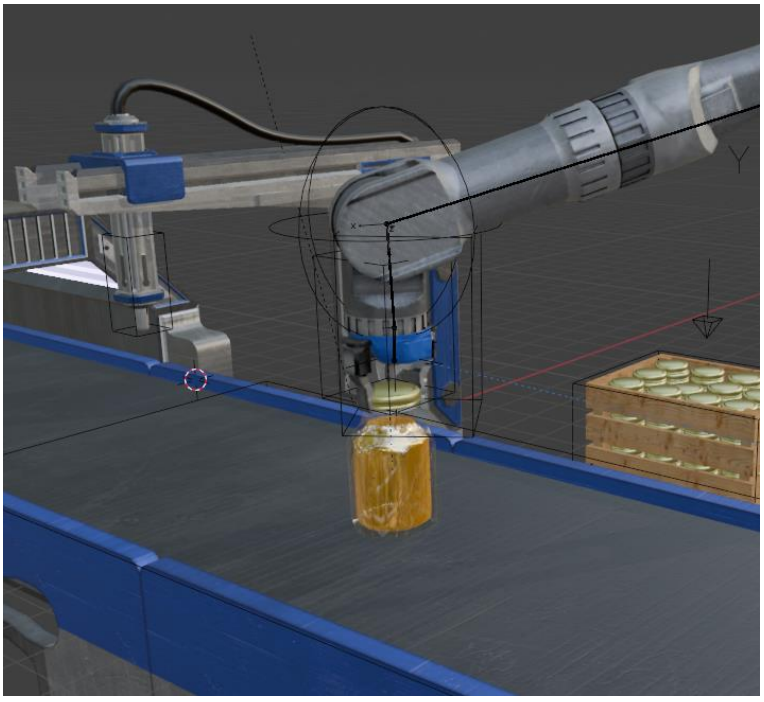

Рисунок 2.46 – Маніпулятор переносить кришку до банки та закручує її

Далі банка переміщується до найближчої вільної коробки, де за допомогою оператора place_jar_in_box об’єкт точно розміщується всередині контейнера. Після кожної шостої банки додатково активується оператор fill_stacks_from_base, який додає кришки у коробку для моделювання підготовки наступної партії (рис. 2.47).

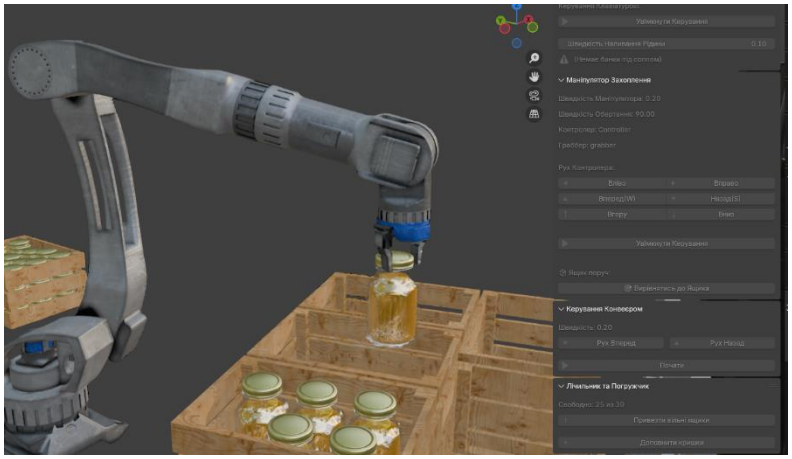

Рисунок 2.47 – Банка опускається в коробку

На рис. 2.48 наведено фінальний вигляд коробки з уже заповненими банками, що ілюструє успішне завершення виробничого циклу. Усі банки розміщено обережно та відповідно до логіки розташування. (рис. 2.48).

Рисунок 2.48 – Заповнена коробка

Під час тестування було змодельовано декілька варіантів запуску: з мінімальною кількістю банок (1), стандартною (6) та розширеною (12), що

дозволило оцінити гнучкість роботи алгоритму при різних вхідних параметрах. У всіх випадках система успішно проходила всі етапи, включно з перевіркою на наявність вільних коробок, відмову у разі їх відсутності та контроль логіки переходу між станами.

Окрім основної логіки автоматичного виконання, система передбачає можливість ручного зупинення процесу користувачем на будь–якому етапі. Це дає змогу як перервати виконання у разі потреби, так і відновити його пізніше без втрати прогресу, що особливо важливо під час тестування або зміни конфігурації сцени.

Поведінка системи при зупиненні залежить від того, на якому саме етапі було ініційовано зупинку. Якщо процес було зупинено до завершення заливки рідини (тобто до завершення stage < 3), система просто припиняє виконання, і при наступному запуску продовжує обробку з тієї кількості, яка залишилась. Наприклад, якщо з 5 запланованих банок було оброблено лише 3, то після перезапуску залишкові 2 банки буде завершено автоматично (рис. 2.49).

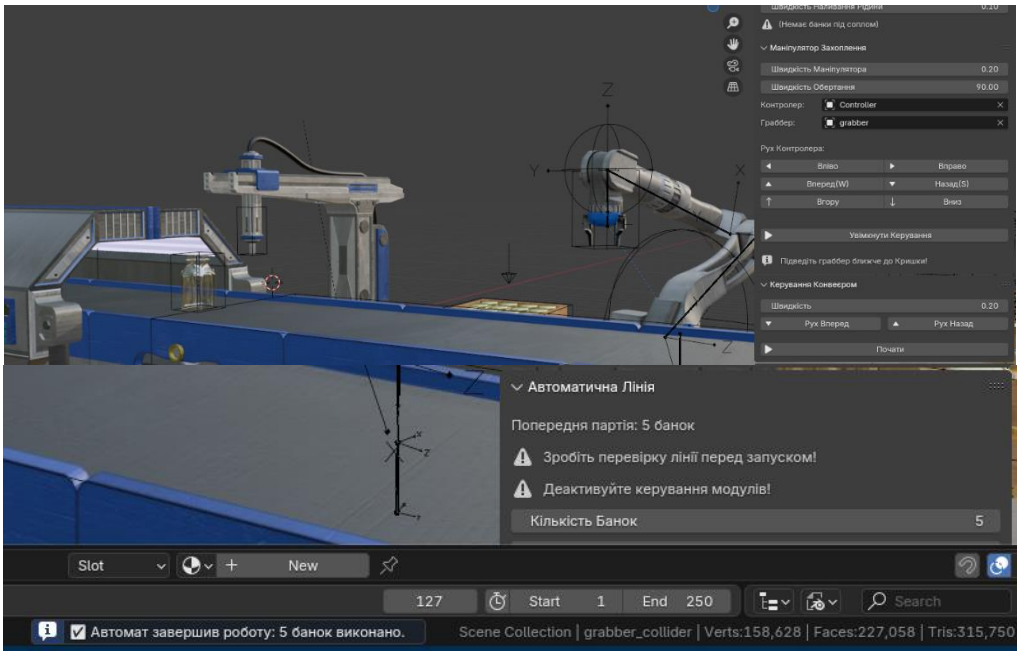

Рисунок 2.49 – Результат роботи модуля після зупинки

Натомість, якщо зупинка відбулась після завершення заливки (stage >= 3), тобто вже у фазі переміщення банки або її обробки маніпулятором, система фіксує

це як “зупинку після заливки” Ця логіка реалізована через змінну auto_line_manual_stop_after_fill, яка активується, якщо зупинка відбулась після заливки. Система інформує користувача про це повідомленням: "Зупинено після заливки – очікується ручне завершення." (рис. 2.50).

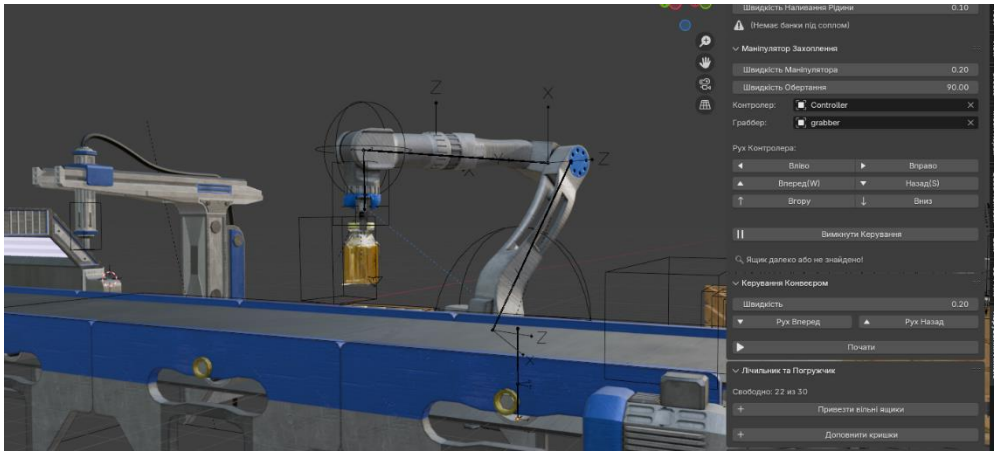

Рисунок 2.50 – Реакція алгоритму на зупинення після заливки

У такому випадку автомат переходить у режим очікування ручного завершення дії: користувач повинен вручну завершити укладання банки (наприклад, за допомогою відповідного оператора place_jar_in_box). Після цього банку буде зараховано як оброблену, і процес може бути продовжений з наступної банки. Такий підхід дозволяє зберегти цілісність циклу та уникнути подвійного обліку вже залитих банок (рис. 2.51).

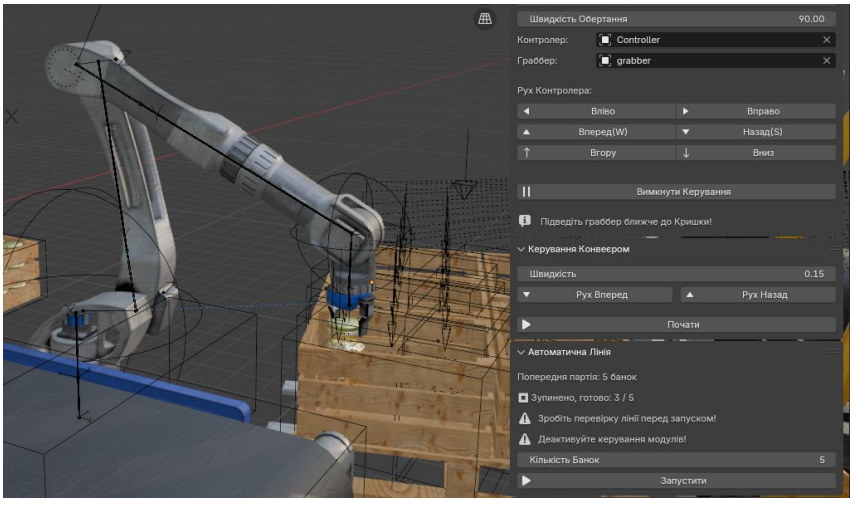

Рисунок 2.51 – Алгоритм добавляє готову банку у логіку модуля

Таким чином, механізм зупинки не тільки не порушує логіку виконання, а й забезпечує додаткову гнучкість, дозволяючи відновлювати процес із точно визначеної точки. Це особливо цінно при налагодженні, відлові помилок або в навчальних симуляціях виробничих процесів (рис. 2.52).

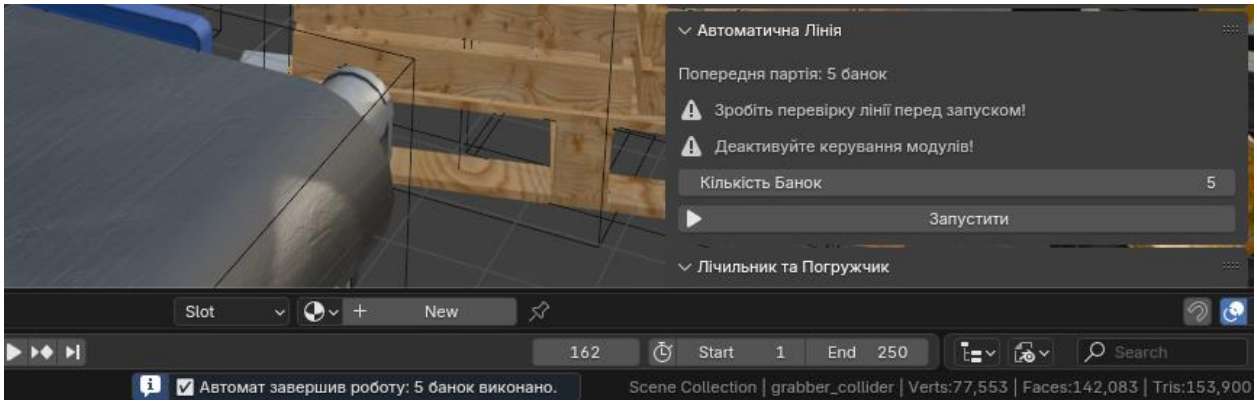

Рисунок 2.52 – Результат роботи модуля з зупинкою після заливки рідини

Практичні результати симуляції продемонстрували ефективність побудованої архітектури, стабільну роботу в рамках віртуального середовища, адаптивність до зміни умов сцени та повну відсутність потреби у ручному втручанні під час виконання. Це підтверджує, що реалізована система придатна до розширення, масштабування та інтеграції в складніші автоматизовані моделі на базі Blender.

2.7 Висновки по реалізації розділу

У цьому розділі було реалізовано та проаналізовано повноцінну автоматизовану модель виробничої лінії у середовищі Blender. Було створено низку функціональних модулів, які забезпечують керування ключовими етапами процесу: подача банок на конвеєрі, заливка рідини, встановлення кришок, транспортування та укладання готової продукції в коробки. Логіка управління побудована на основі модального оператора з використанням таймера, що дозволяє організувати динамічне й гнучке перемикання станів (_stage) відповідно до умов на сцені.

Інтерфейс користувача було реалізовано у вигляді окремої панелі, що забезпечує зручність налаштування параметрів симуляції, зокрема кількості об’єктів для обробки, та управління запуском/зупиненням системи. Передбачено гнучку логіку роботи при зупиненні: система здатна коректно відновити процес як до, так і після завершення окремих етапів, з урахуванням факту завершення заливки рідини.

У процесі тестування система продемонструвала стабільність, передбачуваність та повну відповідність закладеним алгоритмам. Була підтверджена можливість роботи з різною кількістю об’єктів, виявлена чітка обробка помилкових ситуацій, а також забезпечено інформування користувача про ключові стани процесу.

Таким чином, реалізовану систему можна вважати успішно виконаною моделлю виробничого процесу, яка поєднує як візуальне, так і логічне моделювання. Результати підтверджують ефективність використання Blender не лише як графічного редактора, а й як повноцінного інструменту для побудови та симуляції автоматизованих систем з можливістю програмного розширення.

3 ОХОРОНА ПРАЦІ

Однією з найважливіших складових будь–якої професійної діяльності, особливо в галузі ІТ та інженерії, є забезпечення належного рівня охорони праці та безпеки умов роботи. Організація безпечного робочого середовища є обов’язковою вимогою для запобігання шкідливому впливу виробничих факторів на здоров’я працівників, підвищення ефективності праці та уникнення аварійних ситуацій.

3.1 Загальні умови виконання роботи

Практична частина роботи здійснювалась у спеціалізованому приміщенні лабораторії, де розміщено персональні комп’ютери з моніторами на базі РК–дисплеїв. Площа лабораторного простору становить близько 30 м², а загальний об’єм – приблизно 100 м³. Згідно з діючими санітарними нормами, на одного працівника має припадати не менше 6 м² площі та 20 м³ об’єму повітря. В умовах даної лабораторії ці норми повністю дотримані, оскільки в приміщенні одночасно працювало не більше п’яти осіб.

3.2 Мікроклімат і освітлення

Мікрокліматичні умови контролювались згідно з нормативами ДСТУ. Температура повітря в робочій зоні підтримувалась у межах 23–25 °C, відносна вологість – 40–60%, швидкість руху повітря не перевищувала 0,1 м/с. Усі ці параметри відповідають рекомендованим показникам, сприяючи підтриманню комфортного фізіологічного стану під час виконання інтелектуальної роботи.

Щодо освітлення, у приміщенні використовувалось поєднання природного освітлення (через вікна загальною площею 9 м²) та штучного (освітлювальні прилади з яскравістю понад 300 лк). Це забезпечувало необхідний рівень видимості

при роботі з документами, екранами комп’ютерів та іншими дрібними об’єктами, що позитивно впливало на продуктивність праці.

3.3 Аналіз шкідливих виробничих факторів

У процесі виконання роботи були ідентифіковані та проаналізовані потенційні шкідливі фактори, які можуть виникати у середовищі програмно–інженерної діяльності:

– фізичні фактори: шум від роботи електронного обладнання, статична електрика, мікрокліматичні коливання;

– психофізіологічні фактори: тривале перебування в одній позі, напруження зорового аналізатора (робота за монітором понад 4 години на добу), монотонність праці, підвищене емоційне навантаження під час налагодження програмного коду або під час тестування;

– біологічні та хімічні фактори у лабораторії були відсутні.

Значна увага приділялась ергономіці робочого місця: стіл і стілець були регульовані по висоті, що дозволяло підтримувати правильну позу під час роботи; клавіатура і миша розміщувались на оптимальній висоті, що зменшувало навантаження на кисті рук.

3.4 Вентиляція та повітрообмін

У лабораторії передбачено систему природної вентиляції через вікна та щілинне провітрювання.

При необхідності застосовувалась примусова вентиляція для забезпечення відповідного повітрообміну.

Рівень вмісту вуглекислого газу в приміщенні не перевищував допустимі значення, що є свідченням адекватного повітряного обміну.

3.5 Електробезпека

Оскільки у процесі реалізації проекту використовувалось електронне обладнання (зокрема комп’ютери, контролери, плати Arduino), забезпечення електробезпеки мало пріоритетне значення. Вся апаратура працювала від мережі 220 В, була заземлена та під’єднана через стабілізатори напруги. Розетки і вимикачі знаходилися в справному стані, а доступ до них не був ускладнений. Крім того, для захисту від короткого замикання використовувались автоматичні вимикачі.

3.6 Пожежна безпека

В лабораторії наявні вогнегасники порошкового типу, розташовані у легкодоступному місці біля виходу. Проведено інструктаж з протипожежної безпеки, зокрема стосовно правил використання електроприладів і дій у випадку виникнення задимлення або пожежі. Усі працівники знайомі з розташуванням шляхів евакуації та планом дій у надзвичайних ситуаціях.

3.7 Висновок до розділу

На підставі аналізу умов праці встановлено, що всі ключові параметри робочого середовища відповідали нормативним вимогам щодо безпеки праці(табл. 3.1). У результаті реалізації комплексу заходів із забезпечення охорони праці було мінімізовано вплив шкідливих факторів, забезпечено комфортні умови для інженерної діяльності, що сприяло ефективному та безпечному виконанню поставлених завдань. Виявлені незначні фактори ризику не мали критичного впливу на здоров’я або працездатність.

Таблиця 3.1 – Ключові параменти робочого середовища

Параметр	Значення / Опис
Площа приміщення	30 м²
Об’єм приміщення	100 м³
Кількість працівників	До 5 осіб одночасно
Норматив на 1 особу	Не менше 6 м² площі та 20 м³ об'єму повітря
Температура повітря	23–25 °C
Вологість	40–60%
Швидкість повітря	≤ 0,1 м/с
Природне освітлення	Вікна площею 9 м²
Штучне освітлення	Освітлювальні прилади, ≥ 300 лк
Фізичні фактори	Шум, мікроклімат, статична електрика
Психофізіологічні фактори	Зорове навантаження, статика, монотонність, стрес
Хімічні/Біологічні фактори	Відсутні
Вентиляція	Природна + примусова при потребі
Електробезпека	Заземлення, стабілізатори, автоматичні вимикачі
Пожежна безпека	Вогнегасники, інструктаж, евакуаційний план

ВИСНОВКИ

У межах кваліфікаційної роботи бакалавра було реалізовано додаток–тренажер для моделювання роботи автоматизованої виробничої лінії у середовищі Blender. Система призначена для симуляції процесів заливки рідини в банки, встановлення кришок, сортування та пакування продукції. Основний функціонал було реалізовано через модульну архітектуру із застосуванням мови Python та API Blender.

У першому розділі було сформульовано мету та завдання роботи, визначено сферу застосування розробленої системи, а також проведено аналітичний огляд подібних технологій і принципів моделювання автоматизованих процесів. Особливу увагу приділено вибору середовища розробки – Blender, як безкоштовного та функціонального інструменту для інтеграції візуалізації з програмною логікою.

У другому розділі розглянуто основні технічні етапи розробки: створення 3D–моделей, конфігурація об’єктів сцени, їх анімація та взаємодія через спеціально розроблені аддони. Було визначено основні компоненти виробничої лінії, описано структуру даних, логіку циклічного процесу, та обґрунтовано вибір підходів до побудови маніпуляційних дій. Окрему увагу приділено моделюванню кінематики маніпулятора та формуванню алгоритмів переміщення банок.

У третьому розділі, розробленому в межах практичної частини, створено повноцінну програмну реалізацію симуляції. Автоматизована система керується через інтерфейс користувача, який дозволяє запускати, призупиняти або продовжувати обробку партії об'єктів. Було реалізовано поведінку системи при зупиненні як до, так і після ключових етапів (заливка, транспортування), що забезпечує гнучкість і стійкість логіки. Проведено повне тестування системи, а також верифікацію симуляції з різною кількістю об'єктів і у різних сценаріях. Отримані результати підтвердили коректність роботи як візуальної частини, так і програмної логіки.

Результатом роботи є гнучка, масштабована симуляційна модель автоматизованої лінії, яка може бути використана як тренажер, демонстраційний проект або основа для подальших досліджень. Розроблений функціонал продемонстрував ефективність поєднання Blender із Python для моделювання реальних виробничих процесів у віртуальному середовищі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Danylenko M. M. Comparative analysis of modern SCADA packages for production automation / M. M. Danylenko, S. V. Sotnik // International Journal of Academic Engineering Research (IJAER). – 2025. – Vol. 9. – 2. – pp. 26-34.

2 Методичні вказівки з підготовки кваліфікаційної роботи бакалавра для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 151 Автоматизація та комп'ютерно–інтегровані технології освітньої програми «Системна інженерія» / Упоряд.: І.Ш. Невлюдов, О.М. Цимбал, О.В. Токарєва, А.І. Бронніков. Харків: ХНУРЕ, 2022. – 66 с.

3 ДСТУ 3008–15. Документація. Звіти у сфері науки та техніки. структура та правила оформлення. Введ. 2015–06–22. К. Держстандарт України, 2017. – 29 с.

4 Intelligent Bin Picking System in Simulink [Електронний ресурс]. Режим доступу: https://www.mathworks.com/help/robotics/ug/intelligent-bin-picking-system-in-simulink.html (дата звернення: 22.04.2025).

5 Model and Control Robot Dynamics to Automate Virtual Assembly Line [Електронний ресурс]. Режим доступу: https://www.mathworks.com/help/simulink/slref/model-and-control-robot-dynamics-to-automate-virtual-assembly-line.html (дата звернення: 22.04.2025).

6 Robotics System Toolbox – MathWorks [Електронний ресурс]. Режим доступу: https://www.mathworks.com/products/robotics.html (дата звернення: 22.04.2025).

7 Corke P. I. Robotics, Vision and Control. – Springer, 2017. [Електронний ресурс]. Режим доступу: https://petercorke.com/books/robotics–vision–control–all–versions/ (дата звернення: 22.04.2025).

8 RigidBodyTree в MATLAB Docs [Електронний ресурс]. Режим доступу:https://www.mathworks.com/help/robotics/ref/rigidbodytree.html (дата звернення: 22.04.2025).

9 Gazebo — Dexterous Hand 2.2.4 documentation [Електронний ресурс].

Режим доступу: https://shadow-robot-company-dexterous-hand.readthedocs-hosted.com/en/2.2.4/user_guide/sim_gazebo.html (дата звернення: 23.04.2025).

10 Robot Operating System (ROS)integrated with Microsoft Azure service [Електронний ресурс]. Режим доступу: https://www.researchgate.net/figure/Robot-Operating-System-ROSintegrated-with-Microsoft-Azure-services-17_fig3_374532017 (дата звернення: 23.04.2025).

11 Gazebo Tutorials – Classic [Електронний ресурс]. Режим доступу: https://classic.gazebosim.org/tutorials

12 ROS Wiki – URDF [Електронний ресурс]. Режим доступу: https://wiki.ros.org/urdf (дата звернення: 23.04.2025).

13 ROS + Gazebo Integration Overview [Електронний ресурс]. Режим доступу: https://wiki.ros.org/gazebo_ros_pkgs (дата звернення: 23.04.2025).

14 Quigley M. та ін. ROS: an open–source Robot Operating System. – ICRA, 2009. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/233881999_ROS_an_open–source_Robot_Operating_System (дата звернення: 23.04.2025).

15 User interface [Електронний ресурс]. Режим доступу: https://manual.coppeliarobotics.com/en/userInterface.htm (дата звернення: 23.04.2025).

16 CoppeliaSim Overview [Електронний ресурс]. Режим доступу: https://www.coppeliarobotics.com (дата звернення: 23.04.2025).

17 CoppeliaSim User Manual [Електронний ресурс]. Режим доступу: https://manual.coppeliarobotics.com (дата звернення: 23.04.2025).

18 Curtis Holt, Robotic Design with Blender [Електронний ресурс]. Режим доступу: https://www.artstation.com/artwork/Z5lk0R (дата звернення: 23.04.2025).

19 Blender 4.4 Reference Manual [Електронний ресурс]. Режим доступу: https://docs.blender.org/manual/en/latest/ (дата звернення: 23.04.2025).

20 knee-koh/MarIOnette. Blender plugin for controlling Arduino-based microcontrollers over Serial. [Електронний ресурс]. Режим доступу:

https://github.com/knee-koh/MarIOnette (дата звернення: 23.04.2025).

21 geni-lab/ros_blender_bridge, A bridge between ROS and Blender [Електронний ресурс]. Режим доступу: https://github.com/geni-lab/ros_blender_bridge (дата звернення: 23.04.2025).

22 Python API documentation for Blender. [Електронний ресурс]. Режим доступу:https://docs.blender.org/api/current/index.html(дата звернення: 27.04.2025).

23 3.13.4 Documentation – Python [Електронний ресурс]. Режим доступу: https://docs.python.org/3/ (дата звернення: 27.04.2025).

24 Blender Developer Documentation, Inverse Kinematics – iTaSC Algorithm. [Електронний ресурс]. Режим доступу: https://developer.blender.org/docs/features/animation/ik/(дата звернення: 07.05.2025).

25 Blender Developer Documentation [Електронний ресурс]. Режим доступу: https://developer.blender.org/docs/ (дата звернення: .04.2025).

26 МВ до лаб. робіт з дисципліни «Основи охорони праці» для студентів усіх напрямів та форм навчання. / Упоряд.: Т.Є. Стиценко, В.А. Айвазов, О.В. Мамонтов. – Харків: ХНУРЕ, 2018.– 120 с.