# ДОДАТОК А

## Лістинг коду для CoppeliaSim

```
from coppeliasim_zmqremoteapi_client import RemoteAPIClient
import random
import threading
import time
import numpy as np
import cv2

# --- CoppeliaSim Connection ---
client = RemoteAPIClient('localhost', 23000)
sim = client.getObject('sim')
print('[INFO] Connected to CoppeliaSim')

#Threat lock
lock = threading.Lock()

# Handles
manipulation_sphere = sim.getObject('/IRB140/manipulationSphere')
vision_sensor       = sim.getObject('/ROBOT_CAMERA_DETECTION')

# Home pose
HOME_POS = [-0.320, 0.210, 0.620]
HOME_ORI = [0, 0, -90]
with lock:
    sim.setObjectPosition(manipulation_sphere, sim.handle_world, HOME_POS)
    sim.setObjectOrientation(manipulation_sphere, sim.handle_world, HOME_ORI)

# Constants
PIXEL_TO_WORLD = 0.0005  # meters per pixel
data_params = {
    'MAX_SPEED':    0.5,     # m/s
    'HEIGHT_OFFSET':0.2,     # m above object
    'FRAME_SLEEP':  0.05,    # s between steps
    'GRASP_OFFSET': 0.05     # m above object to grasp
}

# Primitives and color HSV ranges
shape_constants = {
    'Cuboid':   sim.primitiveshape_cuboid,
    'Cylinder': sim.primitiveshape_cylinder,
    'Sphere':   sim.primitiveshape_spheroid
}
# only pick these colors (trash/gray excluded)
color_ranges = {
    'Green':  ((50,100,100),(70,255,255)),
    'Yellow': ((20,100,100),(30,255,255)),
    'Blue':   ((100,100,100),(130,255,255))
}

# Bins for each color
bin_positions = {
    'Green':  [0.325, 1.250, 0.450],
    'Yellow': [-0.025, 1.250, 0.450],
    'Blue':   [-0.900, 1.250, 0.450]
}

# track only handles
spawned = []

# Spawn loop: create objects periodically, track only non-gray
```

```
# and map handles to their color
spawned = []
shape_color_map = {}
def spawn_loop():
    while True:
        # choose shape and random color including gray
        name, prim = random.choice(list(shape_constants.items()))
        colors = {
            'Green': (0,1,0),
            'Yellow': (1,1,0),
            'Blue': (0,0,1),
            'Gray': (0.5,0.5,0.5)
        }
        color_name, color_rgb = random.choice(list(colors.items()))
        try:
            with lock:
                h = sim.createPrimitiveShape(prim, [0.15]*3, 0)
        except Exception as e:
            print(f"[WARN] spawn_loop: {e}, retrying...")
            time.sleep(0.5)
            continue
        if h != -1:
            with lock:
                sim.setObjectPosition(h, sim.handle_world, [0.225,-
0.55,0.54])
                sim.setObjectInt32Param(h, sim.shapeintparam_static, 0)
                sim.setObjectSpecialProperty(h,
                    sim.objectspecialproperty_collidable|
                    sim.objectspecialproperty_measurable)
                sim.setObjectInt32Param(h, sim.shapeintparam_respondable, 1)
                sim.setShapeColor(h, None,
sim.colorcomponent_ambient_diffuse, color_rgb)
            # track only if not gray
            if color_name != 'Gray':
                spawned.append(h)
                shape_color_map[h] = color_name
            print(f"[INFO] Spawned {name} with color {color_name}
(handle={h})")
        time.sleep(20)

# Move sphere toward target at speed scaled by dt target at speed scaled by
dt
def move_towards(target, dt):
    with lock:
        pos = sim.getObjectPosition(manipulation_sphere, sim.handle_world)
    vec = np.array(target) - np.array(pos)
    dist = np.linalg.norm(vec)
    if dist < 1e-3:
        return
    step = min(data_params['MAX_SPEED']*dt, dist)
    newp = (np.array(pos) + vec/dist*step).tolist()
    with lock:
        sim.setObjectPosition(manipulation_sphere, sim.handle_world, newp)

# Pick and place: attach, return home, go to bin, release, return home
def pick_and_place(obj, color):
    # attach
    with lock:
        sim.setObjectInt32Param(obj, sim.shapeintparam_static, 1)
        sim.setObjectParent(obj, manipulation_sphere, True)
    # return home with object
    while True:
        with lock:
            sp = sim.getObjectPosition(manipulation_sphere, sim.handle_world)
```

```
        if np.linalg.norm(np.array(sp)-np.array(HOME_POS))<0.01:
            break
        move_towards(HOME_POS, data_params['FRAME_SLEEP'])
        time.sleep(data_params['FRAME_SLEEP'])
    # move to bin
    binp = bin_positions[color]
    target_bin = [binp[0], binp[1], binp[2]+data_params['GRASP_OFFSET']]
    while True:
        with lock:
            sp = sim.getObjectPosition(manipulation_sphere, sim.handle_world)
        if np.linalg.norm(np.array(sp)-np.array(target_bin))<0.01:
            break
        move_towards(target_bin, data_params['FRAME_SLEEP'])
        time.sleep(data_params['FRAME_SLEEP'])
        # release
    with lock:
        sim.setObjectParent(obj, -1, True)
        sim.setObjectInt32Param(obj, sim.shapeintparam_static, 0)
    # intermediate return home
    while True:
        with lock:
            sp = sim.getObjectPosition(manipulation_sphere, sim.handle_world)
        if np.linalg.norm(np.array(sp)-np.array(HOME_POS))<0.01:
            break
        move_towards(HOME_POS, data_params['FRAME_SLEEP'])
        time.sleep(data_params['FRAME_SLEEP'])
    # rotate 90° about Z at home
    with lock:
        sim.setObjectOrientation(manipulation_sphere, sim.handle_world,
                                 [HOME_ORI[0], HOME_ORI[1], HOME_ORI[2] +
np.pi/2])
    # move to bin for final placement
    binp = bin_positions[color]
    target_bin = [binp[0], binp[1], binp[2] + data_params['GRASP_OFFSET']]
    while True:
        with lock:
            sp = sim.getObjectPosition(manipulation_sphere, sim.handle_world)
        if np.linalg.norm(np.array(sp)-np.array(target_bin))<0.01:
            break
        move_towards(target_bin, data_params['FRAME_SLEEP'])
        time.sleep(data_params['FRAME_SLEEP'])
    # release to box
    # (object already released)
    # return home after sorting
    while True:
        with lock:
            sp = sim.getObjectPosition(manipulation_sphere, sim.handle_world)
        if np.linalg.norm(np.array(sp)-np.array(HOME_POS))<0.01:
            break
        move_towards(HOME_POS, data_params['FRAME_SLEEP'])
        time.sleep(data_params['FRAME_SLEEP'])

    while True:
        with lock:
            sp = sim.getObjectPosition(manipulation_sphere, sim.handle_world)
        if np.linalg.norm(np.array(sp)-np.array(HOME_POS))<0.01:
            break
        move_towards(HOME_POS, data_params['FRAME_SLEEP'])
        time.sleep(data_params['FRAME_SLEEP'])

# Camera loop: detect only valid colors, choose nearest to center

def camera_loop():
    prev = time.time()
```

```
    while True:
        now = time.time(); dt = now-prev; prev=now
        try:
            with lock:
                img_bytes, res = sim.getVisionSensorImg(vision_sensor)
        except:
            time.sleep(0.1)
            continue
        img =
np.frombuffer(img_bytes,dtype=np.uint8).reshape((res[1],res[0],3))
        bgr = cv2.cvtColor(img,cv2.COLOR_RGB2BGR)
        hsv = cv2.cvtColor(bgr,cv2.COLOR_BGR2HSV)

        # build list of candidates
        cands=[]
        for color,(lo,hi) in color_ranges.items():
            mask=cv2.inRange(hsv,np.array(lo),np.array(hi))

cnts,_=cv2.findContours(mask,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
            for c in cnts:
                M=cv2.moments(c)
                if M['m00']==0: continue
                cx=int(M['m10']/M['m00']); cy=int(M['m01']/M['m00'])
                # approximate world xy
                dx=(cx-res[0]//2)*PIXEL_TO_WORLD
                dy=-(cy-res[1]//2)*PIXEL_TO_WORLD
                approx=[HOME_POS[0]+dx,HOME_POS[1]+dy]
                if not spawned: continue
                # nearest obj handle
                obj=min(spawned, key=lambda h:
np.linalg.norm(np.array(sim.getObjectPosition(h,sim.handle_world)[:2])-
np.array(approx)))
                op=sim.getObjectPosition(obj,sim.handle_world)
                tgt=[op[0],op[1],op[2]+data_params['GRASP_OFFSET']]
                cands.append((color,c,(cx,cy),tgt,obj))
        if cands:
            # pick best by image-center distance
            best=min(cands,key=lambda x:abs(x[2][0]-res[0]//2)+abs(x[2][1]-
res[1]//2))
            color,cnt,cent,tgt,obj=best
            move_towards(tgt,dt)
            if
np.linalg.norm(np.array(sim.getObjectPosition(manipulation_sphere,sim.handle_
world))-np.array(tgt))<0.02:
                pick_and_place(obj,color)
                spawned.remove(obj)
        else:
            move_towards(HOME_POS,dt)

        cv2.imshow('cam',bgr)
        if cv2.waitKey(1)&0xFF==ord('q'): break
    cv2.destroyAllWindows()

# entry point
if __name__=='__main__':
    with lock: sim.startSimulation()
    threading.Thread(target=spawn_loop,daemon=True).start()
    camera_loop()
    with lock: sim.stopSimulation()
    print('[INFO] Simulation stopped')
```

Лістинг програми камери

```
main_gui.py
import ttkbootstrap as tb
from ttkbootstrap.constants import *
from PIL import Image, ImageTk
import cv2
import os
import threading
import shutil
import sys
from ultralytics import YOLO
from plastic_classifier import classify_crop, retrain, model

CLASSES = ["PET", "HDPE", "PVC", "OTHER", "TRASH"]
DATASET_DIR = "dataset"
TRAINING_INPUT_DIR = "Training"

LANGUAGES = {
    "en": {
        "camera": " Camera",
        "dataset": " Dataset",
        "training": " Training",
        "restart": " Restart Camera",
        "extract_all": "Detect all objects",
        "save_as": "Save as",
        "retrain": " Retrain",
        "place_images": "Place images in folder:",
        "distribute": " Distribute by class",
        "n_neighbors": "n_neighbors:"
    },
    "uk": {
        "camera": " Камера",
        "dataset": " Датасет",
        "training": " Навчання",
        "restart": " Перезапустити камеру",
        "extract_all": "Розпізнавати всі об'єкти",
        "save_as": "Зберегти як",
        "retrain": " Перенавчити",
        "place_images": "Розмістіть зображення у папці:",
        "distribute": " Розподілити за класами",
        "n_neighbors": "n_сусідів:"
    }
}

class App(tb.Window):
    def __init__(self):
        self.language = "uk"
        self.lang = LANGUAGES[self.language]
        super().__init__(themename="darkly")
        self.title("Plastic Sorting System")
        self.geometry("1280x720")

        self.model = YOLO("yolov8n.pt")
        self.cap = cv2.VideoCapture(0)
        self.running = True
        self.last_crop = None

        os.makedirs(TRAINING_INPUT_DIR, exist_ok=True)

        self.lang = LANGUAGES[self.language]
        self.lang_switch_frame = tb.Frame(self)
```

```
            self.lang_switch_frame.pack(pady=5)

              tb.Button(self.lang_switch_frame, text='GB', bootstyle='info-
outline',                       width=4,                        command=lambda:
self.switch_language('en')).pack(side='left', padx=5)
              tb.Button(self.lang_switch_frame, text='UA', bootstyle='info-
outline',                       width=4,                        command=lambda:
self.switch_language('uk')).pack(side='left', padx=5)

            self.notebook = tb.Notebook(self)
            self.notebook.pack(fill="both", expand=True)

            self.clear_interface()
            self.create_camera_tab()
            self.create_dataset_tab()
            self.create_training_tab()

            self.update_frame()

        def create_camera_tab(self):
            self.camera_extract_all_objects = tb.BooleanVar(value=False)
            self.camera_tab = tb.Frame(self.notebook)
            self.notebook.add(self.camera_tab, text=self.lang["camera"])

            self.video_label = tb.Label(self.camera_tab)
            self.video_label.pack(pady=(10, 5))

               self.info_label  =  tb.Label(self.camera_tab,  text="Ожидание
распознавания...", font=("Helvetica", 12), bootstyle="info")
            self.info_label.pack(pady=(0, 10))

            self.save_buttons_frame = tb.Frame(self.camera_tab)
            self.save_buttons_frame.pack()

                              restart_btn     =     tb.Button(self.camera_tab,
text=self.lang["restart"], bootstyle="warning", command=self.restart_camera)
            restart_btn.pack(pady=10)

                         extract_checkbox    =    tb.Checkbutton(self.camera_tab,
text=self.lang["extract_all"],       variable=self.camera_extract_all_objects,
bootstyle="info")
            extract_checkbox.pack(pady=(0, 5))

            for idx, label in enumerate(CLASSES):
                            self.bind(str(idx   +   1),   lambda   e,   l=label:
self.save_latest_crop(l))
                                btn    =    tb.Button(self.save_buttons_frame,
text=f"{self.lang['save_as']}  {label}",  bootstyle="outline",  command=lambda
l=label: self.save_latest_crop(l))
                btn.pack(side="left", padx=5, pady=5)

        def create_dataset_tab(self):
            self.dataset_tab = tb.Frame(self.notebook)
            self.notebook.add(self.dataset_tab, text=self.lang["dataset"])

                        self.class_selector    =    tb.Combobox(self.dataset_tab,
values=CLASSES, bootstyle="dark")
            self.class_selector.set(CLASSES[0])
            self.class_selector.pack(pady=10)
                self.class_selector.bind("<<ComboboxSelected>>",  lambda  e:
self.refresh_images())

                              retrain_btn     =     tb.Button(self.dataset_tab,
```

```
text=self.lang["retrain"], bootstyle="success", command=retrain)
            retrain_btn.pack(pady=10)

             self.canvas = tb.Canvas(self.dataset_tab, background="#222222",
highlightthickness=0)
                        self.scrollbar    =    tb.Scrollbar(self.dataset_tab,
orient="vertical", command=self.canvas.yview)
            self.scrollable_frame = tb.Frame(self.canvas)
                    self.scrollable_frame.bind("<Configure>",   lambda   e:
self.canvas.configure(scrollregion=self.canvas.bbox("all")))
             self.canvas.create_window((0, 0), window=self.scrollable_frame,
anchor="nw")
            self.canvas.configure(yscrollcommand=self.scrollbar.set)

            self.canvas.pack(side="left", fill="both", expand=True)
            self.scrollbar.pack(side="right", fill="y")

            self.refresh_images()

        def create_training_tab(self):
            self.extract_all_objects = tb.BooleanVar(value=True)
            self.n_neighbors_label = None
            self.training_tab = tb.Frame(self.notebook)
                                        self.notebook.add(self.training_tab,
text=self.lang["training"])

                              label_info     =     tb.Label(self.training_tab,
text=f"{self.lang['place_images']}     {os.path.abspath(TRAINING_INPUT_DIR)}",
font=("Helvetica", 10))
            label_info.pack(pady=10)

                         self.progress    =    tb.Progressbar(self.training_tab,
bootstyle="striped", maximum=100)
            self.progress.pack(pady=10, fill="x", padx=20)

                       self.log_text   =   tb.Text(self.training_tab,   height=8,
background="#111", foreground="#0f0")
                    self.log_text.pack(padx=10,   pady=(10,   0),   fill="both",
expand=True)

                        self.n_neighbors_label   =   tb.Label(self.training_tab,
text=f"{self.lang['n_neighbors']}   -",   font=("Helvetica",   9,   "italic"),
bootstyle="secondary")
            self.n_neighbors_label.pack(pady=(0, 5))

              self.extract_all_checkbox  =  tb.Checkbutton(self.training_tab,
text=self.lang["extract_all"],              variable=self.extract_all_objects,
bootstyle="info")
            self.extract_all_checkbox.pack(pady=(5, 5))

                             train_button    =    tb.Button(self.training_tab,
text=self.lang["distribute"],                                bootstyle="primary",
command=self.run_training_threaded)
            train_button.pack(pady=10)

        def run_training_threaded(self):
            threading.Thread(target=self.run_training, daemon=True).start()

        def run_training(self):
            self.progress['value'] = 0
                 self.log("[•]  Начинается  распределение  изображений  по
классам...")
            self.update_idletasks()
```

```
            if not hasattr(model, "classes_") or len(model.classes_) < 1:
                    self.log("[!] Классификатор ещё не обучен. Попробуйте
нажать 'Перетренировать'.")
                 tb.messagebox.showerror("Ошибка", "Классификатор не обучен.
Добавьте изображения и нажмите 'Перетренировать'.")
                return

            # Автоматическая подстройка количества соседей
            if hasattr(model, 'n_neighbors') and hasattr(model, '_fit_X'):
                model.n_neighbors = min(3, len(model._fit_X))
                    self.log(f"[•] n_neighbors автоматически установлен в
{model.n_neighbors}")
                         self.n_neighbors_label.config(text=f"n_neighbors:
{model.n_neighbors}")

                files = [f for f in os.listdir(TRAINING_INPUT_DIR) if
f.lower().endswith(('.jpg', '.png'))]
            total = len(files)
            if total == 0:
                self.log("[!] Нет изображений в папке Training.")
                return

            self.log(f"[•] Найдено {total} изображений для классификации.")

            for i, file in enumerate(files):
                full_path = os.path.join(TRAINING_INPUT_DIR, file)
                img = cv2.imread(full_path)
                if img is None:
                          self.log(f"[!] Не удалось прочитать изображение:
{file}")
                    continue

                self.log(f"[•] Обработка файла: {file} ({i+1}/{total})")
                results = self.model(img)[0]
                if not results.boxes:
                        self.log(f"[!] Объекты не найдены на изображении:
{file}")
                    continue
                for j, box in enumerate(results.boxes):
                        if not self.extract_all_objects.get() and j > 0:
                            break
                        x1, y1, x2, y2 = map(int, box.xyxy[0])
                        crop = img[y1:y2, x1:x2]
                        label = classify_crop(crop)
                           self.log(f"[✓] Объект {j+1} классифицирован как:
{label}")

                        class_dir = os.path.join(DATASET_DIR, label)
                        os.makedirs(class_dir, exist_ok=True)
                                                                filename     =
f"{os.path.splitext(file)[0]}_obj{j+1}.jpg"
                        dest_path = os.path.join(class_dir, filename)
                        cv2.imwrite(dest_path, crop)

                        self.log(f"[→] Сохранено: {dest_path}")

                self.log(f"[→] Перемещено в: {dest_path}")
                self.progress['value'] = int((i + 1) / total * 100)
                self.update_idletasks()

             self.log("[✓] Распределение завершено! Все файлы разложены по
папкам.")
               tb.messagebox.showinfo("Готово", "Файлы перенесены в папки
```

```
классов.")

        def log(self, message):
            self.log_text.insert("end", message + "\n")
            self.log_text.see("end")

        def update_frame(self):
            if not self.running:
                return

            ret, frame = self.cap.read()
            if not ret:
                self.after(100, self.update_frame)
                return

            frame = cv2.resize(frame, (800, 600))
            results = self.model(frame)[0]
            self.last_crop = None
            label_text = "Нет бутылки"

            for j, box in enumerate(results.boxes):
                if not self.camera_extract_all_objects.get() and j > 0:
                    break
                cls_id = int(box.cls[0])
                conf = float(box.conf[0])
                x1, y1, x2, y2 = map(int, box.xyxy[0])
                crop = frame[y1:y2, x1:x2]
                label = classify_crop(crop)
                self.last_crop = crop
                label_text = f"{label} ({conf:.2f})"
                cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 255), 2)
                            cv2.putText(frame,   label,   (x1,   y1   -   10),
cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 0), 2)

            self.info_label.config(text=label_text)

            rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            img = Image.fromarray(rgb)
            imgtk = ImageTk.PhotoImage(image=img)
            self.video_label.imgtk = imgtk
            self.video_label.config(image=imgtk)

            self.after(100, self.update_frame)

        def save_latest_crop(self, label):
            if self.last_crop is not None:
                folder = os.path.join(DATASET_DIR, label)
                os.makedirs(folder, exist_ok=True)
                count = len(os.listdir(folder))
                path = os.path.join(folder, f"{label}_{count}.jpg")
                cv2.imwrite(path, self.last_crop)
                print(f"[+] Сохранено: {path}")

        def refresh_images(self):
            for widget in self.scrollable_frame.winfo_children():
                widget.destroy()

            cls = self.class_selector.get()
            folder = os.path.join(DATASET_DIR, cls)
            files = [f for f in os.listdir(folder) if f.endswith(".jpg") or
f.endswith(".png")]

            for i, file in enumerate(files):
                img_path = os.path.join(folder, file)
```

```
                    img = Image.open(img_path)
                    img.thumbnail((120, 120))
                    img_tk = ImageTk.PhotoImage(img)

                    frame = tb.Frame(self.scrollable_frame)
                    frame.grid(row=i // 6, column=i % 6, padx=5, pady=5)

                    label = tb.Label(frame, image=img_tk)
                    label.image = img_tk
                    label.pack()

                                del_btn   =   tb.Button(frame,   text="Удалить",
bootstyle="danger-outline",      width=10,      command=lambda      p=img_path:
self.delete_image(p))
                    del_btn.pack(pady=2)

            def delete_image(self, path):
                os.remove(path)
                self.refresh_images()

            def clear_interface(self):
                for tab in self.notebook.winfo_children():
                    tab.destroy()

            def restart_camera(self):
                self.log("[⟳] Перезапуск камеры...")
                self.cap.release()
                self.cap = cv2.VideoCapture(0)
                self.running = True

            def on_close(self):
                self.running = False
                self.cap.release()
                self.destroy()

            def switch_language(self, lang_code):
                self.language = lang_code
                self.lang = LANGUAGES[self.language]
                self.clear_interface()
                self.create_camera_tab()
                self.create_dataset_tab()
                self.create_training_tab()

        if __name__ == "__main__":
            app = App()
            app.protocol("WM_DELETE_WINDOW", app.on_close)
            app.mainloop()
```

gui_dataset_viewer.py

```
import os
import tkinter as tk
from tkinter import ttk, messagebox
from PIL import Image, ImageTk
from plastic_classifier import retrain

DATASET_DIR = "dataset"
CLASSES = ["PET", "HDPE", "PVC", "OTHER", "TRASH"]

class DatasetViewer(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Dataset Manager")
        self.geometry("800x600")
```

```
        self.selected_class = tk.StringVar(value=CLASSES[0])
        self.image_list = []
        self.current_index = 0

        self.create_widgets()
        self.load_images()

    def create_widgets(self):
        top_frame = tk.Frame(self)
        top_frame.pack(side=tk.TOP, fill=tk.X)

             class_menu = ttk.OptionMenu(top_frame, self.selected_class,
self.selected_class.get(), *CLASSES, command=self.on_class_change)
        class_menu.pack(side=tk.LEFT, padx=10, pady=10)

            retrain_btn = ttk.Button(top_frame, text="Retrain Classifier",
command=self.retrain_model)
        retrain_btn.pack(side=tk.LEFT, padx=10)

        self.image_label = tk.Label(self)
        self.image_label.pack(expand=True)

        control_frame = tk.Frame(self)
        control_frame.pack(side=tk.BOTTOM, pady=10)

                prev_btn = ttk.Button(control_frame, text="<< Prev",
command=self.show_prev_image)
        prev_btn.grid(row=0, column=0, padx=5)

                  del_btn = ttk.Button(control_frame, text="Delete",
command=self.delete_current_image)
        del_btn.grid(row=0, column=1, padx=5)

                next_btn = ttk.Button(control_frame, text="Next >>",
command=self.show_next_image)
        next_btn.grid(row=0, column=2, padx=5)

    def load_images(self):
        folder = os.path.join(DATASET_DIR, self.selected_class.get())
             self.image_list = sorted([os.path.join(folder, f) for f in
os.listdir(folder) if f.endswith((".jpg", ".png"))])
        self.current_index = 0
        self.show_image()

    def show_image(self):
        if not self.image_list:
            self.image_label.config(image="", text="No images found.")
            return
        path = self.image_list[self.current_index]
        img = Image.open(path)
        img.thumbnail((1280, 720))
        img_tk = ImageTk.PhotoImage(img)
        self.image_label.img_tk = img_tk
        self.image_label.config(image=img_tk)

    def on_class_change(self, *_):
        self.load_images()

    def show_next_image(self):
        if not self.image_list:
            return
        self.current_index = (self.current_index + 1) % len(self.image_list)
        self.show_image()
```

```
    def show_prev_image(self):
        if not self.image_list:
            return
        self.current_index = (self.current_index - 1) % len(self.image_list)
        self.show_image()

    def delete_current_image(self):
        if not self.image_list:
            return
        path = self.image_list.pop(self.current_index)
        try:
            os.remove(path)
                                messagebox.showinfo("Deleted",     f"Удалено:
{os.path.basename(path)}")
        except Exception as e:
            messagebox.showerror("Error", f"Ошибка удаления: {e}")
        if self.current_index >= len(self.image_list):
            self.current_index = max(0, len(self.image_list) - 1)
        self.show_image()

    def retrain_model(self):
        retrain()
        messagebox.showinfo("Success", "Классификатор переобучен!")

if __name__ == "__main__":
    app = DatasetViewer()
    app.mainloop()
```

main.py

```
import cv2
import os
from ultralytics import YOLO
from plastic_classifier import classify_crop, save_crop, retrain

model = YOLO("yolov8n.pt")
TARGET_CLASS = "bottle"

LABEL_KEYS = {
    ord('1'): 'PET',
    ord('2'): 'HDPE',
    ord('3'): 'PVC',
    ord('4'): 'OTHER',
    ord('5'): 'TRASH',
    ord('r'): 'RETRAIN'
}

SAVE_DIR = "dataset"
for label in LABEL_KEYS.values():
    if label != 'RETRAIN':
        os.makedirs(os.path.join(SAVE_DIR, label), exist_ok=True)

def main():
    cap = cv2.VideoCapture(0)

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        results = model(frame)[0]

        for box in results.boxes:
            cls_id = int(box.cls[0])
```

```
            cls_name = model.names[cls_id]
            if cls_name != TARGET_CLASS:
                continue
            conf = float(box.conf[0])
            x1, y1, x2, y2 = map(int, box.xyxy[0])
            crop = frame[y1:y2, x1:x2]
            label = classify_crop(crop)

            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 255), 2)
            cv2.putText(frame, f"{label} ({conf:.2f})", (x1, y1 - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 255), 2)

        cv2.putText(frame, "1: PET  2: HDPE  3: PVC  4: OTHER  5: TRASH  R:
Retrain  ESC: Exit",
                    (10, 25), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (200, 255, 200),
2)

        cv2.imshow("YOLOv8 Plastic Classifier", frame)
        key = cv2.waitKey(1)

        if key == 27:  # ESC
            break
        elif key in LABEL_KEYS:
            if LABEL_KEYS[key] == 'RETRAIN':
                retrain()
            else:
                for box in results.boxes:
                    cls_id = int(box.cls[0])
                    cls_name = model.names[cls_id]
                    if cls_name != TARGET_CLASS:
                        continue
                    x1, y1, x2, y2 = map(int, box.xyxy[0])
                    crop = frame[y1:y2, x1:x2]
                    save_crop(crop, LABEL_KEYS[key])

    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

plastic_classifier.py

```
import os
import cv2
import joblib
import numpy as np
from glob import glob
from sklearn.neighbors import KNeighborsClassifier

MODEL_PATH = "plastic_model.pkl"
CLASSES = ['PET', 'HDPE', 'PVC', 'OTHER', 'TRASH']

def extract_features(img):
    img = cv2.resize(img, (64, 64))
    return img.flatten()

def train_model():
    X, y = [], []
    for label in CLASSES:
        path = os.path.join("dataset", label)
        os.makedirs(path, exist_ok=True)
        for file in glob(f"{path}/*.jpg"):
            img = cv2.imread(file)
            if img is None:
```

```
                continue
            X.append(extract_features(img))
            y.append(label)
    if len(X) == 0:
        return None
    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(X, y)
    joblib.dump(model, MODEL_PATH)
    return model

def load_model():
    if os.path.exists(MODEL_PATH):
        return joblib.load(MODEL_PATH)
    else:
        return train_model()

model = load_model()

def classify_crop(crop):
    feat = extract_features(crop)

    # Подстройка n_neighbors под количество обученных примеров
    if hasattr(model, '_fit_X') and len(model._fit_X) > 0:
        model.n_neighbors = min(3, len(model._fit_X))
    else:
        return "UNKNOWN"

    try:
        return model.predict([feat])[0]
    except Exception as e:
        print("[!] Ошибка классификации:", e)
        return "UNKNOWN"

def save_crop(img, label):
    os.makedirs(f"dataset/{label}", exist_ok=True)
    count = len(os.listdir(f"dataset/{label}"))
    filename = f"dataset/{label}/{label}_{count}.jpg"
    cv2.imwrite(filename, img)
    print(f"[+] Saved: {filename}")

def retrain():
    global model
    model = train_model()
    print("[*] Model retrained.")
```

# ДОДАТОК Б

Демонстраційний матеріал

| Формат | Позиція. | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кільк. | Примітки |
|---|---|---|---|---|---|
| | | | Текстові документи | | |
| А4 | 1 | ГЮІК ХХХХХХ.049 ПЗ | Пояснювальна записка | 54 | с. |
| | | | | | |
| | | | | | |
| | | | Інші документи | | |
| | 2 | | Демонстраційний матеріал | 16 | с. |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

| Змін. | Арк | Номер докум. | Підпис | Дата | ГЮІК ХХХХХХ.049 ВД | | | |
|---|---|---|---|---|---|---|---|---|
| Розроб. | | Савенко Д.О. | | | Розроблення системи автоматизації для сортування пластмасових виробів з використанням маніпулятора із 6 ступенями вільності<br>Відомість кваліфікаційної роботи | Літера | Арку | Аркуш |
| Перевір. | | Ромашов Ю.В. | | | | Н | 1 | 1 |
| | | | | | | Кафедра КІТАР ХНУРЕ | | |
| Н.контр. | | Демська Н.П. | | | | | | |
| Затв. | | Невлюдов І.Ш. | | | | | | |