Back Sault Simulation using OpenCV and Unity

Yavuz Selim Kaya

1. Introduction

This technical document presents a real-time blob detection and visualization system that integrates Python (with OpenCV) and Unity using socket communication. The system is designed to detect moving objects (blobs) in a video stream and visualize them simultaneously in a Unity scene.

The Python component captures frames form a video source, processes each frame using computer vision techniques, and extracts blob coordinates. These coordinates are serialized in JSON format and transmitted over a TCP socket to a Unity application. Unity receives the data frame by frame and renders the detected points as part of a live simulation.

This project requires working knowledge of:

- Python (OpenCV, JSON handling, socket programming)
- C# scripting in Unity
- TCP/IP networking concepts
- Real-time data visualization

2. System Requirements

- Windows 10 operating system or later versions
- Python 3.10 or later versions
- OpenCV, JSON, Time, socket libraries
- Unity 2022.3 or later versions
- C#
- Minimum 4GB RAM

3. System Architecture

The system adopts a three-tier design in which a Python-based **Frame Processor** (leveraging OpenCV) ingests each video frame, detects the visible blobs, and converts their centroid coordinates into JSON. Inside the same process, a lightweight **TCP Gateway** streams these JSON strings over a persistent socket on "127.0.0.1:2002", ensuring continuous delivery at the target frame rate. On the client side, a **Visualization module** written in C# for Unity established the socket connection, reconstructs the point objects from the incoming stream, and renders them in the scene in real time.

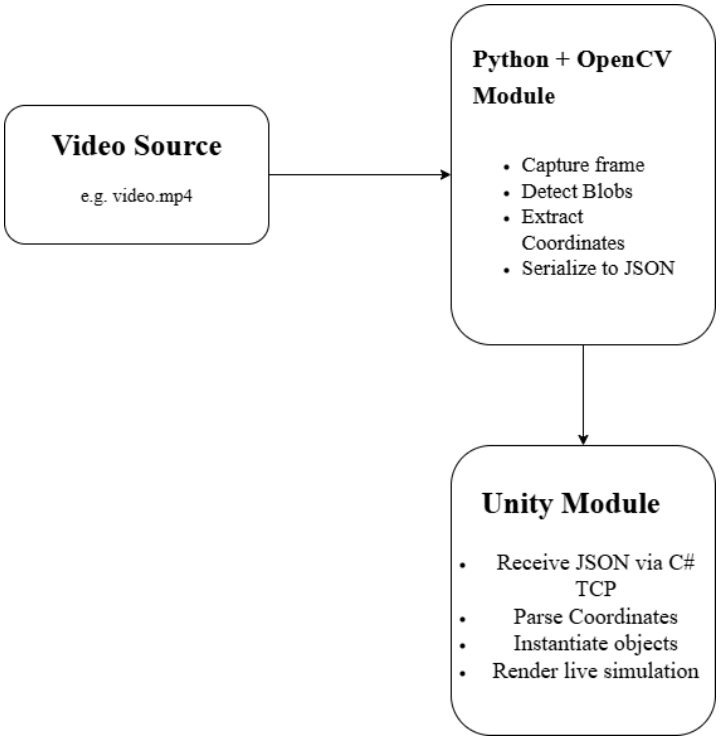

Figure 1

The end-to-end data path begins with frame capture from the *back_sault* image set. Each JPEG is thresholded at a value of 60, after which the resulting contours are distilled into centroid coordinates. Exactly seven points are required from every frame, so if the head marker is missing the algorithm interpolates it from shoulder to hip positions before packing the data. The coordinates are embedded in a JSON object of the form ***{"points": […]}*** and terminated with a single line-feed character. Unity's background thread gathers the byte stream, separates message by \n, deserializes them with ***JsonUtility*****,** and queues them for the main thread. During every update cycle, the main thread dequeues one message refreshes the corresponding GameObjects and redraws the skeletal LineRenderer segments.

Within the Python realm, **FrameProcessor** handles image I/O, blob detection, and point numbering via the functions ***get_frame_coordinates()*** and ***prepare_numbered_coords(),*** while **SocketServer** binds the port and calls ***send_frames()*** to maintain the 30-FPS rhythm. On the Unity side, **SocketClient** relies on ***TcpClient*** and ***ReceiveData()*** routine to rebuild JSON messages, and the **Renderer** component-through ***UpdatePoints()*** and ***CreateLine()*** instantiates point prefabs and skeletal lines, keeping the visualization in sync with the data feed.

Communication occurs over TCP in little-endian order across a persistent connection on port 2002(this value can be changed in configuration). Each payload is UTF-8 encoded JSON that lists the seven numbered points; a single line feed(0x0A) marks the end of every message. Because each frame is roughly one kilobyte, the stream's bandwidth averages about 30kB/s at 30 FPS.

Transmission cadence is enforced in Python with ***sleep(1/FPS)***, which Unity's main thread deliberately consumes one queued frame per ***Update()*** call. This pairing guarantees that the simulation's visual tempo mirrors the source frame rate and prevents buffer overflows on either side.

If the head point is absent, it is estimated dynamically from the shoulder and hip positions. Should the socket connection break, a *Broken Pipe* is logged and the server closes the connection gracefully so that Unity can attempt a clean reconnection. Because every JSON message is self-contained, the loss of one frame doesn't compromise later frames, allowing the visualizer to continue unhindered once connectivity is restored.

4. Data Format Specification

```
{
  "points": [
    { "id": 0, "x": 123, "y": 456 },
    { "id": 1, "x": 234, "y": 567 },
    { "id": 2, "x": 345, "y": 678 },
    { "id": 3, "x": 456, "y": 789 },
    { "id": 4, "x": 567, "y": 890 },
    { "id": 5, "x": 678, "y": 901 },
    { "id": 6, "x": 789, "y": 912 }
  ]
}
```

This is the data format we used for sending data from Python socket to Unity TCP connection. All points have "id", "x" and "y" values. While image processing section, the algorithm may detect 6 or 7 points. If the algorithm detects 6 points, with interpolation algorithm fills the empty slot with interpolation techniques.

5. Software Overview

The system has 2 scripts. First is the Python script, which does image processing and a socket server. The second is the C# script, which takes the coordinates from the network and initializes in Unity.

5.1. Python Script

The Python script has two purposes. The first is detecting blobs in the images. Second is to send the blob coordinates with socket.

```python
import cv2
import os
import glob
import json
import socket
import time
import math

# Settings
INPUT_FOLDER = 'back_sault'
IMAGE_TYPE = '*.jpg'
THRESHOLD = 60
HOST = '127.0.0.1'
PORT = 2002
FPS = 30
TOTAL_POINTS = 7
reference_coords = None  # the points which taken coordinates from the first frame.
```

Figure 2

The first part of the code involves importing libraries and setting the basic constant variables that are used throughout the rest of the code. The socket uses the “127.0.0.1” (local host) with “2002” port. The result simulation in Unity will be 30 frames per second, and the back-sault video has seven blob points to detect. The OpenCV library uses the threshold value 60 to make it easier for the image processing algorithm. Reference coordinates variable, which is called “reference_coords” is defined for later.

```python
def get_frame_coordinates(image_path):
    frame = cv2.imread(image_path)
    if frame is None:
        return []

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(gray, THRESHOLD, 255, cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    coords = []
    for contour in contours:
        M = cv2.moments(contour)
        if M['m00'] != 0:
            cX = int(M['m10'] / M['m00'])
            cY = int(M['m01'] / M['m00'])
            coords.append({"x": cX, "y": cY})
    return coords
```

Figure 3

The function named “get_frame_coordinates” takes “image_path” as a parameter. The purpose of the function is to detect blobs in each frame and send coordinates to the “coords” array. The function returns a “coords” array.

```python
def interpolate_missing_point(coords):
    if len(coords) == 6:
        coords = sorted(coords, key=lambda p: (p["y"], p["x"]))
        max_gap = 0
        insert_index = len(coords)
        for i in range(len(coords) - 1):
            gap = coords[i+1]["y"] - coords[i]["y"]
            if gap > max_gap:
                max_gap = gap
                insert_index = i + 1

        p1 = coords[insert_index - 1] if insert_index > 0 else coords[0]
        p2 = coords[insert_index] if insert_index < len(coords) else coords[-1]

        interp_x = (p1["x"] + p2["x"]) // 2
        interp_y = (p1["y"] + p2["y"]) // 2
        coords.insert(insert_index, {"x": interp_x, "y": interp_y})
    return coords
```

Figure 4

The function named “interpolate_missing_points” takes “coords” as a parameter. The purpose of the function is to complete missing points with interpolation.

```python
def prepare_numbered_coords(coords):
    global reference_coords

    if len(coords) < 6:
        return []

    if len(coords) == 6:
        coords = interpolate_missing_point(coords)

    if len(coords) != 7 and reference_coords is None:
        return []


    if reference_coords is None and len(coords) == 7:
        coords = sorted(coords, key=lambda p: (p["y"], p["x"]))
        reference_coords = coords.copy()
        print("Reference frame set.")
        numbered = []
        for i, p in enumerate(coords):
            numbered.append({ "id": i, "x": p["x"], "y": p["y"] })
        return numbered


    if len(coords) == 6 and reference_coords is not None:
        ref_shoulder = reference_coords[1]  # ID 1: shoulder
        ref_hip = reference_coords[2]       # ID 2: hip

        shoulder = min(coords, key=lambda p: euclidean(p, ref_shoulder))
        hip = min(coords, key=lambda p: euclidean(p, ref_hip))

        head_x = (shoulder["x"] + hip["x"]) // 2
        head_y = shoulder["y"] - abs(hip["y"] - shoulder["y"])
        coords.append({"x": head_x, "y": head_y})
        print("Head estimated and added.")
```

Figure 5

```python
    if len(coords) != 7:
        return []


    matched = []
    used_indices = set()

    for ref in reference_coords:
        min_dist = float('inf')
        closest = None
        closest_idx = -1

        for i, c in enumerate(coords):
            if i in used_indices:
                continue
            dist = euclidean(ref, c)
            if dist < min_dist:
                min_dist = dist
                closest = c
                closest_idx = i

        if closest is not None:
            matched.append(closest)
            used_indices.add(closest_idx)

    if len(matched) != 7:
        print("Matching error: not all points matched.")
        return []

    numbered = []
    for i, p in enumerate(matched):
        numbered.append({ "id": i, "x": p["x"], "y": p["y"] })
    return numbered
```

Figure 6

The function “prepare_numbered_coords” which given in Figure 5 and Figure 6 takes “coords” array as a parameter. The function checks coordinates have how many blobs with if statements. If it has 6 points to detect, the function calls “interpolate_missing_points” for complete the missing point or points with interpolation.

```
def send_frames():
    image_files = sorted(glob.glob(os.path.join(INPUT_FOLDER, IMAGE_TYPE)))
    if not image_files:
        print("Images not found")
        return

    print("Starting...")
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((HOST, PORT))
    s.listen(1)
    print("Waiting for Unity to connect...")
    conn, addr = s.accept()
    print(f"Connection established with: {addr}\n")

    try:
        for image_path in image_files:
            coords = get_frame_coordinates(image_path)
            numbered_coords = prepare_numbered_coords(coords)

            if not numbered_coords:
                print(f"Skipping frame: {os.path.basename(image_path)}")
                continue

            data = {"points": numbered_coords}
            json_str = json.dumps(data) + "\n"
            conn.sendall(json_str.encode('utf-8'))

            print(f"Sent {len(numbered_coords)} points for {os.path.basename(image_path)}")
            time.sleep(1 / FPS)

        print("All frames sent successfully.")
    except BrokenPipeError:
        print(" Unity connection lost.")
    finally:
        conn.close()
        s.close()
```

Figure 7

The function named “send_frames” takes nothing as parameters. The function works like the brain of the entire program. It starts the socket server, sorts images, and makes the Unity connection.

5.2. C# Scripts

```
using System;
using System.Collections.Generic;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using UnityEngine;
```

Figure 8

This piece of code imports needed classes.