Universidad Nacional de San Agustín de Arequipa

Escuela Profesional de Ciencia de la Computación

Computación Gráfica

Laboratorio 5

Presentado por:
Zavalaga Orozco, Rushell Vanessa

Docente :
Rolando Jesus Cárdenas Talavera

1. Introducción

El presente informe tiene como objetivo implementar un sistema de detección de la trayectoria de un objeto (moneda) en caída libre utilizando procesamiento de video en tiempo real con OpenCV. Se utiliza la cámara web para capturar el movimiento y procesar cada frame para identificar el objeto y trazar su trayectoria.

2. Explicación del Procedimiento

En esta sección se detalla el flujo del procedimiento desarrollado para la detección y seguimiento de la trayectoria del objeto. Cada etapa del procesamiento de imágenes es acompañada por su respectiva implementación en el código, mostrando cómo se aplican técnicas como la conversión a escala de grises, umbralización, operaciones morfológicas y cálculo del centro del objeto en movimiento. Estas acciones permiten identificar con precisión la posición del objeto en cada cuadro del video y construir su trayectoria completa.

2.1. Captura del Video

Se permite al usuario seleccionar el origen del video: ya sea desde la cámara web (modo en vivo) o desde un archivo de video local. La opción se maneja mediante una variable `flag`, y se utiliza la clase `cv::VideoCapture` para abrir el flujo correspondiente.

```
video.open(0);  // Webcam
video.open(".../video.mp4"); // Archivo local
```

2.2. Preprocesamiento de la Imagen

a. Conversión a Escala de Grises

El video se convierte a escala de grises para simplificar el procesamiento.

```
escalaGris(frame);

void escalaGris(Mat &frame) {
    cvtColor(frame, frame, COLOR_BGR2GRAY);
}
```

b. Binarización

Se aplica un umbral para transformar la imagen a blanco y negro. Píxeles con valor mayor al umbral se vuelven blancos (255), los demás negros (0).

```
my_binarization(frame, umbral);

void my_binarization(Mat &frame, int umbral) {
    Mat binarizado = frame.clone();
    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            if (frame.at<uchar>(i, j) > umbral) binarizado.at<uchar>(i, j) = 255;
   else binarizado.at<uchar>(i, j) = 0;
        }
    }
 frame = binarizado.clone();
}
```

2.3. Operaciones Morfológicas

Estas operaciones mejoran la detección del objeto y eliminan ruido.

a. Dilatación

La dilatación es una operación morfológica que expande las regiones negras (que en este caso representan al objeto detectado). Esta expansión ayuda a consolidar las formas y llenar posibles huecos que hayan quedado tras la binarización.

El proceso de dilatación personalizado se implementa mediante la función `my_Dilate`, la cual recorre cada píxel de la imagen y verifica, mediante la función auxiliar `hit`, si se debe aplicar la dilatación en esa posición. Si la condición se cumple, se llama a `dilating` para modificar los píxeles de acuerdo al estructurante definido.

```
void my_Dilate(Mat &frame) {
    Mat dilated = frame.clone();

    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            if (hit(frame, estructurante, i, j)) {
                dilating(dilated, i, j);
            }
        }
    }
    frame = dilated.clone();
}
```

La función `hit` verifica si en la vecindad del píxel actual (definida por la forma del estructurante) existe al menos un píxel negro (`0`). Si esto ocurre, indica que se debe aplicar la dilatación en dicha posición.

```
bool hit(Mat frame, Mat estructurante, int x, int y) {
    for (int i = x; i < x + estructurante.rows; i++) {
        for (int j = y; j < y + estructurante.cols; j++) {
            if (i >= frame.rows || j >= frame.cols) continue;
            if (estructurante.at<uchar>(i - x, j - y) == 0 &&
                frame.at<uchar>(i, j) == 0) {
                return true;
            }
        }
    }
    return false;
}
```

Finalmente, la función `dilating` se encarga de modificar los píxeles vecinos del punto donde se detectó un *hit*, expandiendo así la región negra:

```
void dilating(Mat& frame, int x, int y) {
    for (int i = x; i < x + estructurante.rows; i++) {
        for (int j = y; j < y + estructurante.cols; j++) {
            if (estructurante.at<uchar>(i - x, j - y) == 0 &&
                esValido(frame, i, j)) {
                frame.at<uchar>(i, j) = 0;
            }
        }
    }
}
```

En conjunto, estas funciones simulan el comportamiento de la dilatación de OpenCV pero de manera manual, permitiendo al estudiante comprender detalladamente el mecanismo interno de esta operación morfológica.

b. Erosión

La erosión es otra operación morfológica utilizada para eliminar ruido blanco (píxeles con valor alto o 255) alrededor del objeto detectado. Además, ayuda a definir con mayor precisión los bordes del objeto, eliminando pequeñas imperfecciones o regiones aisladas no deseadas.

En esta implementación personalizada, se utiliza la función `my_Erode`, la cual recorre cada píxel de la imagen binarizada y evalúa si el estructurante encaja completamente sobre una región negra. Esta verificación se realiza mediante la función auxiliar `fit`. Si se cumple esta condición, se considera que el píxel debe mantenerse negro, de lo contrario se convierte en blanco, lo que representa la erosión de esa parte del objeto.

```
void my_Erode(Mat &frame) {
```

```
    Mat eroded(frame.rows, frame.cols, CV_8UC1, 255);

    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            if (fit(frame, estructurante, i, j)) {
                eroded.at<uchar>(i + 1, j + 1) = 0;
            }
        }
    }
    frame = eroded.clone();
}
```

La función `fit` valida si el estructurante se ajusta completamente a una zona negra en la imagen. Es decir, verifica que todos los píxeles cubiertos por el estructurante coincidan con píxeles negros (`0`) en la imagen original. Si encuentra algún píxel blanco (`!= 0`) en una posición donde el estructurante requiere negro, devuelve `false`.

```
bool fit(Mat frame, Mat estructurante, int x, int y) {
    for (int i = x; i < x + estructurante.rows; i++) {
        for(int j = y; j < y + estructurante.cols; j++) {
            if (i >= frame.rows || j >= frame.cols) return false;
            if(estructurante.at<uchar>(i - x, j - y) == 0 &&
               frame.at<uchar>(i,j) != 0) {
                return false;
            }
        }
    }
    return true;
}
```

En resumen, esta operación reduce el tamaño de las regiones negras (el objeto), ayudando a eliminar imperfecciones pequeñas que podrían ser interpretadas como movimiento erróneo. En conjunto con la dilatación, permite refinar la silueta del objeto detectado.

Ambas operaciones utilizan un estructurante definido como una cruz 3x3:

```
uchar estructura[] = { 255, 0, 255,
                       0,   0, 0,
                       255, 0, 255 };
```

2.4. Detección del Objeto

Se identifica el centroide del objeto calculando el promedio de coordenadas de los píxeles negros detectados:

```
Point currentPoint = identifyPoint(frame);

Point identifyPoint(const Mat& frame) {
    int sumaX = 0, sumaY = 0, contador = 0;
    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            Vec3b pixel = frame.at<Vec3b>(i, j);
            if (pixel[0] == 0 && pixel[1] == 0 && pixel[2] == 0) {
                sumaX += j;
                sumaY += i;
                contador++;
            }
        }
    }

    if (contador <= 100) return Point(-1, -1);
    cout << "SumaX " << sumaX << " sumaY " << sumaY << endl;
 return Point(sumaX / contador, sumaY / contador);
}
```

Si el número de píxeles detectados es insuficiente (<100), se considera ruido y se descarta.

2.5. Seguimiento de la Trayectoria

Si el punto detectado es válido, se guarda en un vector y se dibuja una línea entre el punto actual y el anterior:

```
camino.push_back(currentPoint);
drawline(frame, prevPoint, currentPoint);
```

Esto permite visualizar en tiempo real el recorrido del objeto.

2.6. Visualización Final de la Trayectoria

Una vez terminada la grabación, se genera una imagen negra sobre la cual se dibuja toda la trayectoria de la moneda:

```
Mat trajImage(frameSize, CV_8UC3, Scalar(0, 0, 0));
line(trajImage, camino[i - 1], camino[i], Scalar(0, 255, 0), 2);
```

2.7. Salida Visual y Consola

Se utilizan `imshow` para mostrar los resultados en ventanas:

```
imshow("Original", frame);
imshow("Video", frame);
imshow("Trayectoria final", trajImage);
```

Además, se emplean mensajes en consola para informar al usuario sobre el proceso y los puntos detectados.

3. Capturas de pantalla

A continuación, se muestran distintas etapas del procesamiento en tiempo real, donde se puede observar cómo se detecta y rastrea el objeto (una moneda) mediante técnicas de procesamiento de imágenes.

Figura 1: Entrada inicial del sistema (captura de la cámara)

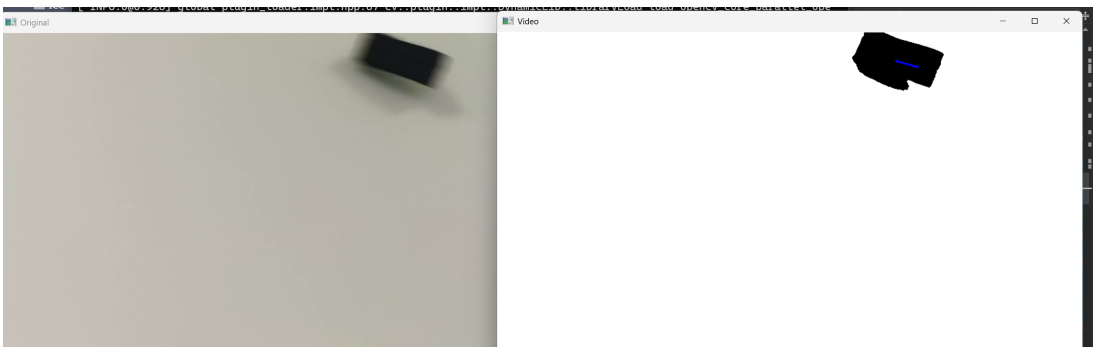

Figura 2: Procesamiento binario del objeto con técnicas morfológicas

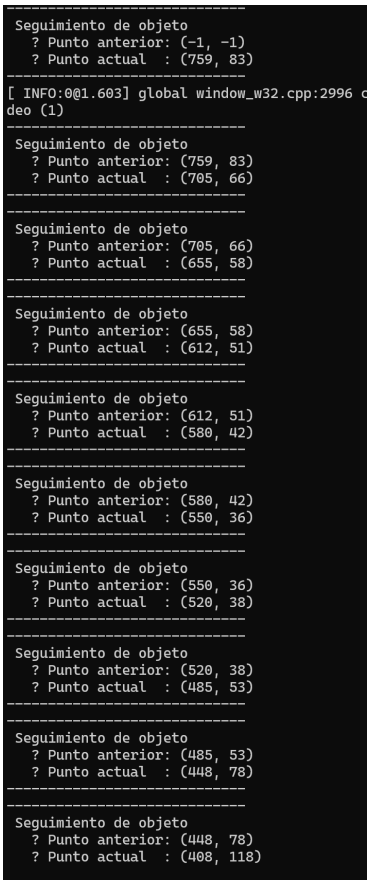

Figura 3: Seguimiento del objeto en terminal - coordenadas detectadas

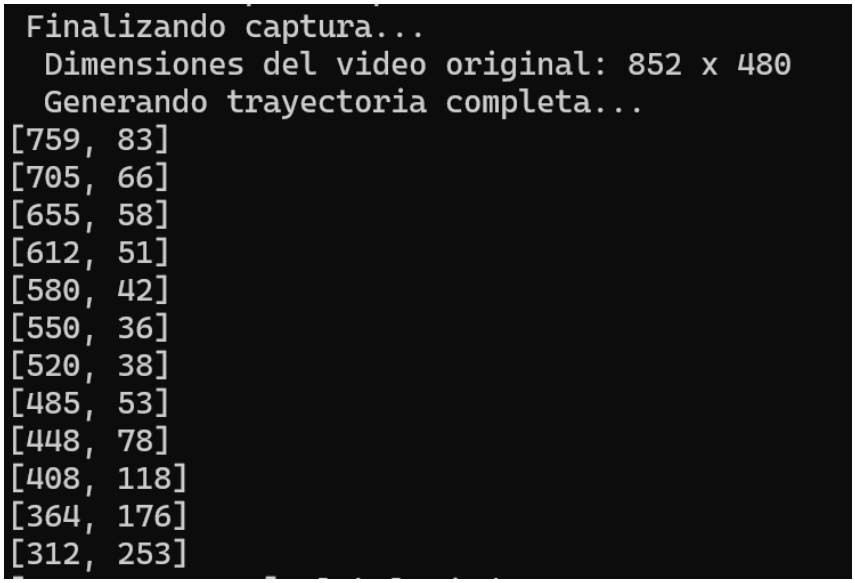

Figura 4: Actualización en tiempo real de los puntos detectados

4. Resultado Obtenido

Finalizado el proceso de captura y análisis, el sistema genera una imagen con la trayectoria completa del objeto.

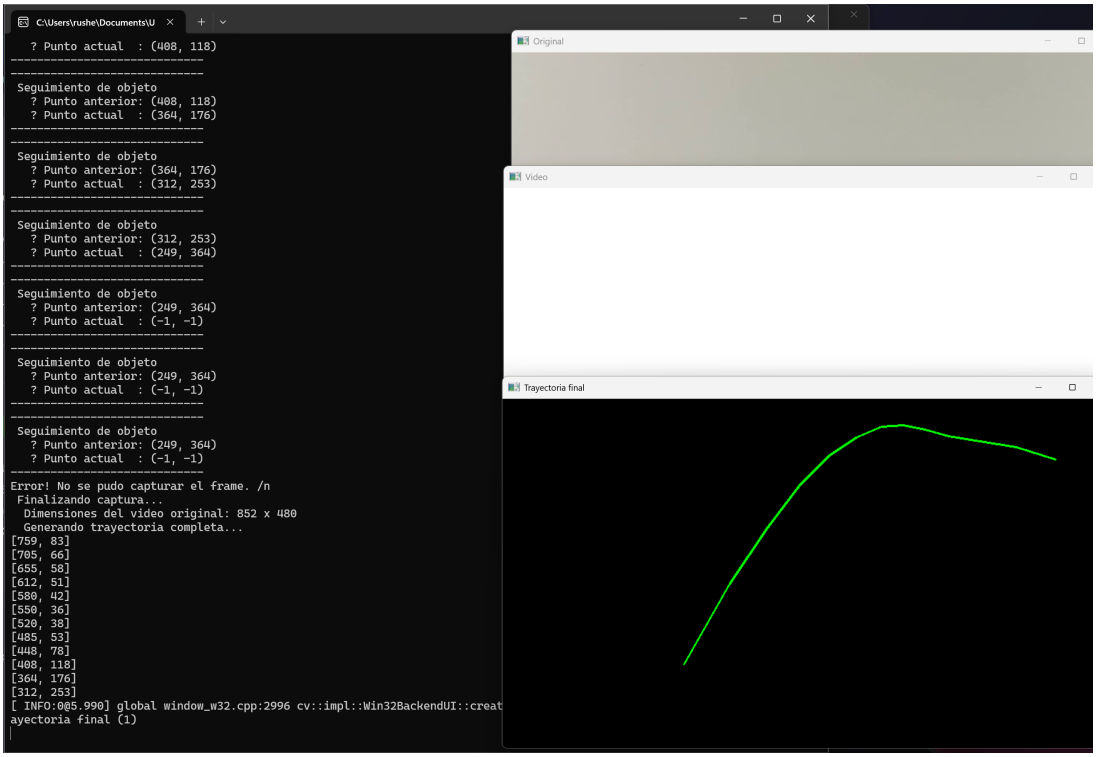

Figura 5: Visualización completa del entorno con la trayectoria sobrepuesta

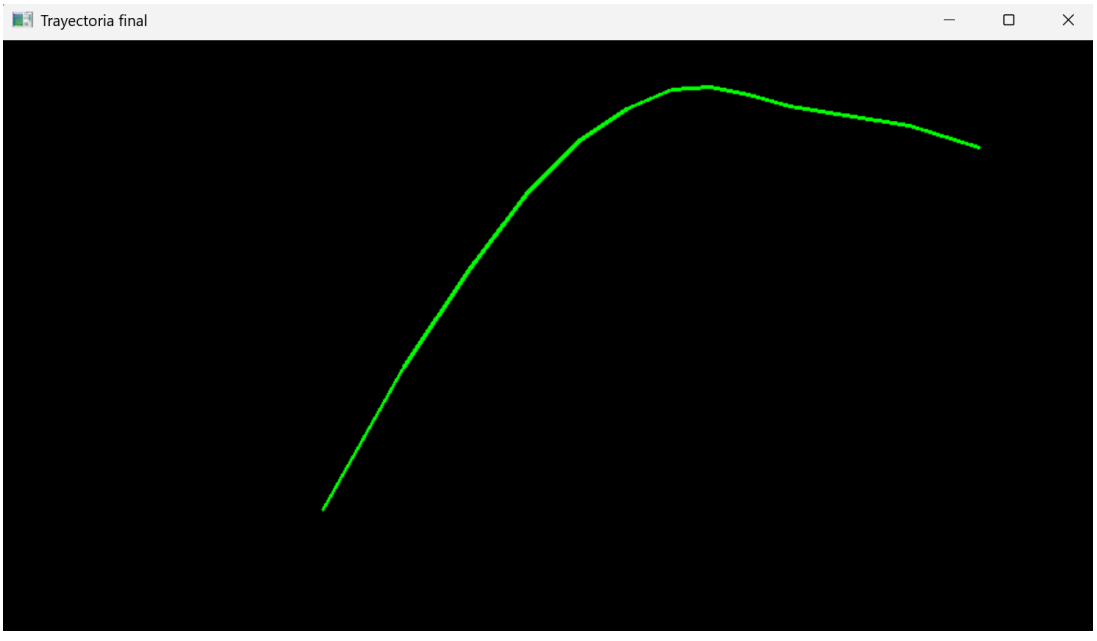

Figura 6: Trayectoria final generada por el sistema

5. Código Fuente

A continuación, se incluye el código implementado para realizar la detección y seguimiento del objeto.

5.1. main.cpp

```
#include <opencv2/opencv.hpp>
#include <vector>

using namespace cv;
using namespace std;


VideoCapture video;
uchar estructura[] = { 255, 0, 255, 0, 0, 0, 255, 0, 255 };
Mat estructurante(3, 3, CV_8UC1, estructura);

void printMat(const Mat &mat) {
    for (int i = 0; i < mat.rows; i++) {
        for (int j = 0; j < mat.cols; j++) {
            cout << (int)mat.at<uchar>(i, j) << " ";
        }
        cout << endl;
    }
}

bool esValido(const Mat& frame, int x, int y) {
    return (x >= 0 && x < frame.rows && y >= 0 && y < frame.cols);
}

void escalaGris(Mat &frame) {
    cvtColor(frame, frame, COLOR_BGR2GRAY);
}

void my_binarization(Mat &frame, int umbral) {
    Mat binarizado = frame.clone();
    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            if (frame.at<uchar>(i, j) > umbral) binarizado.at<uchar>(i,
   j) = 255;
      else binarizado.at<uchar>(i, j) = 0;
        }
    }
  frame = binarizado.clone();
}

bool fit(Mat frame, Mat estructurante, int x, int y) {
    for (int i = x; i < x + estructurante.rows; i++) {
        for(int j = y; j < y + estructurante.cols; j++) {
            if (i >= frame.rows || j >= frame.cols) return false;
            if(estructurante.at<uchar>(i - x, j - y) == 0 && frame.at<
   uchar>(i,j) != 0) {
                return false;
```

```
      }
        }
    }
    return true;
}

bool hit(Mat frame, Mat estructurante, int x, int y) {
    for (int i = x; i < x + estructurante.rows; i++) {
        for (int j = y; j < y + estructurante.cols; j++) {
            if (i >= frame.rows || j >= frame.cols) continue;
            if (estructurante.at<uchar>(i - x, j - y) == 0 && frame.at<
   uchar>(i, j) == 0) {
                return true;
            }
        }
    }
    return false;
}

void my_Erode(Mat &frame) {
    Mat eroded(frame.rows, frame.cols, CV_8UC1, 255);

    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            if (fit(frame, estructurante, i, j)) {
                eroded.at<uchar>(i + 1, j + 1) = 0;
            }
        }
    }
    frame = eroded.clone();
}

void dilating(Mat& frame, int x, int y) {
    for (int i = x; i < x + estructurante.rows; i++) {
        for (int j = y; j < y + estructurante.cols; j++) {
            if (estructurante.at<uchar>(i - x, j - y) == 0 && esValido(
   frame, i, j)) {
                frame.at<uchar>(i, j) = 0;
            }
        }
    }
}

void my_Dilate(Mat &frame) {
  Mat dilated = frame.clone();

    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            if (hit(frame, estructurante, i, j)) {
        dilating(dilated, i, j);
            }
        }
    }
  frame = dilated.clone();
}

void drawline(Mat& frame, Point p1, Point p2) {
```

```
  line(frame, p1, p2, Scalar(255, 0, 0), 2);
}

Point identifyPoint(const Mat& frame) {
    int sumaX = 0, sumaY = 0, contador = 0;
    for (int i = 0; i < frame.rows; i++) {
        for (int j = 0; j < frame.cols; j++) {
            Vec3b pixel = frame.at<Vec3b>(i, j);
            if (pixel[0] == 0 && pixel[1] == 0 && pixel[2] == 0) {
                sumaX += j;
                sumaY += i;
                contador++;
            }
        }
    }

    if (contador <= 100) return Point(-1, -1);
    cout << "SumaX " << sumaX << " sumaY " << sumaY << endl;
  return Point(sumaX / contador, sumaY / contador);
}

int main() {
  // 1 Capurar video desde webcam o archivo
    cout << "\n========================================\n";
    cout << "           LABORATORIO 5 - CS GRAFICA    \n";
    cout << "        Detecci n y Trayectoria de Objeto\n";
    cout << "========================================\n\n";
    bool flag = false;
    cout << ">>   Cmo   deseas capturar el video?\n";
    cout << "    [0] En vivo (Webcam)\n";
    cout << "    [1] Desde archivo (Upload)\n";
    cout << "    Selecci n: ";
    cin >> flag;

    if(flag) {
        cout << "\n Cargando video desde archivo...\n";
        video.open("C:/Users/rushe/Documents/Universidad/S7/Graphics/CS-
   GRAFICA/Lab5/video.mp4");
    //video.open("C:/Users/rushe/Pictures/Camera Roll/video4.mp4");
        if (!video.isOpened()) {
            cout << " Error: No se pudo abrir el video desde archivo.\n"
   ;
            return -1;
        }
        else {
            cout << " Video cargado correctamente.\n";
        }
    }
    else {
        cout << "\n Iniciando captura en vivo desde webcam...\n";
        cout << "   Presiona [ESC] para finalizar la grabaci n.\n";
        video.open(0);
    }

    cout << "\n Preparando el procesamiento de video...\n";
    Mat frame;
    Size frameSize;
```

```
    vector<Point> camino; // Almacenar puntos de trayectoria
Point prevPoint(-1, -1); // Punto previo para dibujar la trayectoria
    int maxX = 0, maxY = 0;

    video.read(frame);
    frameSize = frame.size();  // Tama o del video original

    if (frame.empty()) {
        cout << "Error! No se pudo capturar el frame. /n";
        return -1;
    }
    frameSize = frame.size();

    while(true) {
        video.read(frame);
        if (frame.empty()) {cout << "Error! No se pudo capturar el frame
  . /n"; break;}

    imshow("Original", frame);

        // 2 Prepocesamiento de la imagen
    // Escala de grises y binarizaci n
    escalaGris(frame);
    int umbral = 100;
    my_binarization(frame, umbral);

        // 3 Detecci n de movimiento
    // dilatacion y erosi n
    my_Dilate(frame);
        my_Erode(frame);

        cvtColor(frame, frame, COLOR_GRAY2BGR);
    // 4 deteci n de centro de objeto
    Point currentPoint = identifyPoint(frame);

        cout << "-----------------------------\n";
        cout << " Seguimiento de objeto\n";
        cout << "       Punto anterior: (" << prevPoint.x << ", " <<
  prevPoint.y << ")\n";
        cout << "       Punto actual  : (" << currentPoint.x << ", " <<
  currentPoint.y << ")\n";
        cout << "-----------------------------\n";


        if(prevPoint.x == -1 && prevPoint.y == -1) { prevPoint =
  currentPoint; }
        if (currentPoint.x != -1 && currentPoint.y != -1) {
            camino.push_back(currentPoint);
            if (currentPoint.x > maxX) maxX = currentPoint.x;
            if (currentPoint.y > maxY) maxY = currentPoint.y;
            drawline(frame, prevPoint, currentPoint);
            prevPoint = currentPoint;
        }

        imshow("Video", frame);
        if (waitKey(30) == 27) break; // Presionar ESC para terminar
    }
```

```
    cout << "\n Finalizando captura...\n";
    cout << "  Dimensiones del video original: " << frameSize.width << "
    x " << frameSize.height << "\n";
    cout << "  Generando trayectoria completa...\n";


    Mat trajImage(frameSize,CV_8UC3, Scalar(0, 0, 0));
    for (size_t i = 1; i < camino.size(); i++) {
        cout << camino[i-1] << endl;
        line(trajImage, camino[i - 1], camino[i], Scalar(0, 255, 0), 2);
    }

    imshow("Trayectoria final", trajImage);
    waitKey(0);

    cout << "\n Proceso completado.\n";
    cout << "   La trayectoria ha sido generada y mostrada.\n";
    cout << "   Cierra la ventana para salir.\n";
    cout << "\n Gracias por usar el sistema de seguimiento.\n";

    video.release();
    destroyAllWindows();
    return 0;
}
```

Listing 1: Código principal de dtección de trayectoria

6. Conclusiones

Durante el desarrollo de este proyecto, pude implementar con éxito un sistema que detecta y traza la trayectoria de un objeto en caída libre en tiempo real. Aplicando técnicas fundamentales de procesamiento de imágenes, como la escala de grises, la binarización, la dilatación y la erosión, logré identificar y seguir el objeto de forma eficiente. Este trabajo me permitió comprender mejor cómo interactúan estos procesos para facilitar la segmentación y el análisis de video en vivo. Además, noté que la calidad de la detección puede verse afectada por factores externos como la iluminación, por lo que considero importante mejorar ese aspecto para obtener resultados más precisos en futuras pruebas.