UNIVERSIDAD EUROPEA DEL ATLÁNTICO

GRADO EN

Ingeniería Informática

Reconocimiento de Cartas mediante Técnicas Clásicas de Visión Artificial

Trabajo realizado por

Natalia Cruz Babbar

Profesor

Jose Manuel Breñosa Martínez

SANTANDER - 28 de noviembre, 2025

1. HARDWARE: Características técnicas y requisitos

1.1. Sistema de Captura

Componente	Especificación	Justificación
Dispositivo	Tablet Xiaomi Redmi Pad SE	Dispositivo móvil con cámara integrada de calidad suficiente para captura de video en tiempo real
Cámara	Cámara integrada de tablet Resolución: HD (1280x720) FPS: 30	Resolución adecuada para detectar detalles de cartas. No requiere cámara profesional.
Software de streaming	IP Webcam (Android) Protocolo RTSP	Permite convertir la tablet en cámara IP accesible desde red local. Streaming de baja latencia.
Conexión	Red Wifi local: 172.20.10.8:8080 RTSP URL: rtsp://172.20.10.8:8080/h264.	Conexión inalámbrica para flexibili de posicionamiento de la cámara

1.2. Superficie de Trabajo

Elemento	Especificación	Justificación
Tapete	Cartulina verde Material: Papel cartón	Superficie uniforme de color verde para facilitar segmentación por color HSV. Color verde permite alta diferenciación con cartas blancas.
Iluminación	Luz ambiental controlada. Sin sombras directas	Iluminación uniforme para minimizar variaciones de color y facilitar detección de contornos.

Ventajas del hardware seleccionado: El uso de una tablet como cámara IP ofrece portabilidad, bajo costo y facilidad de configuración. No requiere hardware especializado y permite realizar todo el procesamiento en el ordenador principal. La conexión RTSP proporciona streaming de video eficiente con latencia mínima.

2. SOFTWARE: Stack Tecnológico y Justificación

2.1. Lenguaje de Programación

Python 3.13 - Seleccionado por las siguientes razones:

1. Amplio ecosistema de librerías de visión artificial (OpenCV, NumPy)
2. Sintaxis clara y legible que facilita el mantenimiento del código
3. Excelente para prototipado rápido y desarrollo iterativo
4. Gran comunidad y documentación disponible
5. Rendimiento adecuado para procesamiento de video en tiempo real

2.2. Librerías Principales

Librería	Versión	Uso en el Proyecto	Justificación
OpenCV	4.12.0	• Captura de video RTSP • Procesamiento de imágenes • Detección de contornos • Template matching • Transformaciones geométric	Librería estándar de visión artificial. Optimizada en C++ para alto rendimiento. Inclu todas las funciones necesarias para procesamiento clásico.
NumPy	1.26.0	• Operaciones matriciales • Manipulación de arrays • Cálculos numéricos	Base fundamental para operaciones con imágenes Python.
Collections	Built-in	• Gestión de historial de clasificaciones (deque)	Proporciona estructuras de datos eficientes para suavizado

2.3. Arquitectura del Software

El proyecto sigue una **arquitectura modular** dividida en componentes especializados:

Módulo	Archivo	Responsabilidad
Configuración	config/settings.py	Centraliza todos los parámetros configurables (HSV, ROI, umbral
Preprocesamiento	src/vision/preprocessing.py	Segmentación de cartas, corrección de perspectiva, detección de
Template Matching	src/vision/template_matching.py	Carga de templates, correlación cruzada, matching multi-escala
Clasificación	src/vision/classification.py	Pipeline de clasificación completo, validación por reglas lógicas
Scripts de calibración	scripts/1_calibrar_hsv.py	Herramienta interactiva para ajustar valores HSV del tapete

Scripts de captura	scripts /2_capturar_imagenes_referencia.py	Captura de imágenes de referencia para crear templates
Scripts de templates	scripts/3_crear_templates.py	Extracción interactiva de templates de valores y palos
Script principal	scripts/5_clasificar_realtime.py	Sistema de reconocimiento en tiempo real con interfaz gráfica

3. HOJA DE RUTA DEL DESARROLLO

El desarrollo del proyecto siguió un proceso iterativo y sistemático dividido en fases:

FASE 1: Configuración del Entorno

Objetivos:

- Configurar tablet como cámara IP con app IP Webcam
- Establecer conexión RTSP desde Python
- Verificar captura de video en tiempo real
- Configurar estructura de directorios del proyecto

Resultado: Stream de video funcionando correctamente con latencia mínima

FASE 2: Calibración de Segmentación

Objetivos:

- Implementar script de calibración HSV interactivo (1_calibrar_hsv.py)
- Determinar rangos óptimos para detectar el tapete verde
- Ajustar parámetros hasta lograr segmentación precisa
- Documentar valores HSV finales en settings.py

Resultado: Valores HSV: H[35-105], S[153-255], V[0-255] - Segmentación exitosa del fondo

FASE 3: Captura de Dataset

Objetivos:

- Desarrollar script de captura de imágenes de referencia
- Capturar al menos 1 imagen por cada una de las 52 cartas
- Normalizar cartas mediante transformación de perspectiva
- Validar calidad de imágenes capturadas

Resultado: 52 cartas únicas capturadas con múltiples orientaciones (80+ imágenes totales)

FASE 4: Creación de Templates

Objetivos:

- Implementar herramienta interactiva de selección de ROI
- Extraer templates de valores (AS, 2-10, J, Q, K)
- Extraer templates de palos (♠ ♥ ♦ ♣)
- Procesar templates: binarización, limpieza morfológica, normalización

Resultado: 13 templates de valores + 4 templates de palos creados exitosamente

FASE 5: Implementación de Template Matching

Objetivos:

- Desarrollar sistema de carga de templates
- Implementar template matching con cv2.matchTemplate
- Agregar matching multi-escala para robustez
- Optimizar umbrales de confianza

Resultado: Sistema de matching funcional con TM_CCOEFF_NORMED

FASE 6: Detección de Color

Objetivos:

- Implementar detección de color rojo vs negro
- Probar método HSV (falló con rojos)
- Cambiar a método de ratio BGR (exitoso)
- Integrar detección de color en clasificación

Resultado: Detección de color fiable usando diferencias en canales BGR

FASE 7: Pipeline de Clasificación

Objetivos:

- Integrar preprocesamiento, template matching y validación
- Implementar reglas lógicas de consistencia (color vs palo)
- Desarrollar sistema de confianza con umbrales
- Optimizar ROI para capturar correctamente 10

Resultado: Pipeline completo con validación y alta precisión

FASE 8: Sistema en Tiempo Real

Objetivos:

- Desarrollar interfaz gráfica para clasificación en vivo
- Implementar detección de múltiples cartas
- Agregar estadísticas y métricas de rendimiento
- Optimizar para lograr 25-30 FPS

Resultado: Sistema funcionando en tiempo real con interfaz informativa

FASE 9: Testing y Refinamiento

Objetivos:

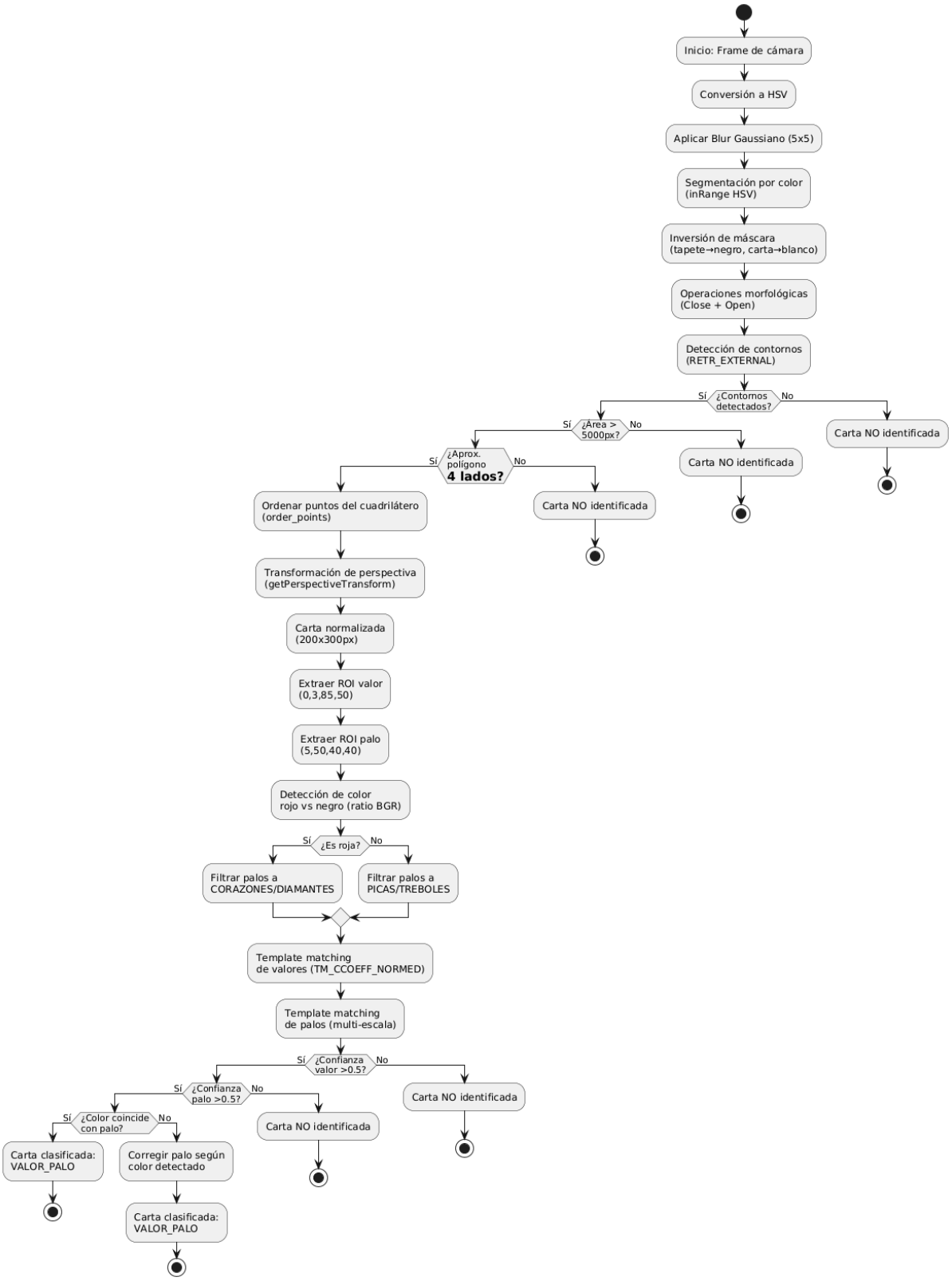
- Probar con todas las 52 cartas
- Identificar y corregir errores de clasificación
- Ajustar parámetros de ROI (especialmente para 10)
- Recrear templates problemáticos

Resultado: Tasa de éxito >95% en condiciones controladas

4. SOLUCIÓN IMPLEMENTADA

4.1. Diagrama de Decisión para Clasificación

El proceso de clasificación sigue el siguiente flujo de decisiones (formato PlantUML):



4.2. Secuencialización de Operaciones

A continuación se detalla cada operación del pipeline con sus parámetros y justificación:

Operación 1: Conversión de Espacio de Color

Función cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

Parámetros

- frame: Imagen BGR de entrada
- COLOR_BGR2HSV: Conversión a espacio HSV

Justificación HSV separa información de color (H), saturación (S) y brillo (V), facilitando la segmentación por color independiente

Operación 2: Filtro Gaussiano

Función cv2.GaussianBlur(hsv, (5,5), 0)

Parámetros

- Kernel: 5x5 píxeles
- Sigma: 0 (calculado automáticamente)

Justificación Reduce ruido de alta frecuencia en la imagen antes de la segmentación, mejorando la calidad de la máscara

Operación 3: Segmentación por Color

Función cv2.inRange(blurred, lower_hsv, upper_hsv)

Parámetros

- lower_hsv: [35, 153, 0]
- upper_hsv: [105, 255, 255]
- Salida: Máscara binaria

Justificación Crea máscara donde el tapete verde queda en blanco. Rangos calibrados específicamente para el tapete usado

Operación 4: Inversión de Máscara

Función cv2.bitwise_not(mask_fondo)

Parámetros

- mask_fondo: Máscara del tapete
- Salida: Máscara de cartas

Justificación Invierte la máscara para que las cartas queden en blanco y el fondo en negro, preparando para detección de contorno

Operación 5: Cierre Morfológico

Función cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)

Parámetros

- Kernel: 3x3 píxeles
- Operación: MORPH_CLOSE (dilatación + erosión)

Justificación Rellena pequeños huecos dentro de las cartas detectadas, uniendo

regiones fragmentadas

Operación 6: Apertura Morfológica

Función cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

Parámetros

- Kernel: 3x3 píxeles
- Operación: MORPH_OPEN (erosión + dilatación)

Justificación Elimina ruido pequeño fuera de las cartas, limpiando la máscara de píxeles aislados

Operación 7: Detección de Contornos

Función cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

Parámetros

- RETR_EXTERNAL: Solo contornos externos
- CHAIN_APPROX_SIMPLE: Compresión de puntos

Justificación Encuentra los bordes de las cartas. RETR_EXTERNAL ignora contornos internos innecesarios, mejorando rendimient

Operación 8: Filtrado por Área

Función cv2.contourArea(contour) >= MIN_CONTOUR_AREA

Parámetros

- MIN_CONTOUR_AREA: 5000 píxeles

Justificación Descarta contornos pequeños causados por ruido o sombras. 5000px corresponde aproximadamente al tamaño míni

Operación 9: Aproximación Poligonal

Función cv2.approxPolyDP(contour, epsilon, True)

Parámetros

- epsilon: 0.03 * perímetro
- closed: True

Justificación Simplifica el contorno a un polígono. epsilon=3% del perímetro permite detectar cuadriláteros incluso con bordes liger

Operación 10: Ordenamiento de Puntos

Función order_points(approx)

Parámetros

- approx: 4 puntos del cuadrilátero
- Método: Suma/diferencia de coordenadas

Justificación Ordena los 4 vértices (TL, TR, BR, BL) para asegurar transformación de perspectiva correcta. Usa criterio geométrico

Operación 11: Transformación de Perspectiva

Función cv2.getPerspectiveTransform(pts1, pts2) cv2.warpPerspective(frame, M, (200,300))

Parámetros

- pts1: Puntos de la carta detectada
- pts2: Puntos destino [0,0], [200,0], [200,300], [0,300]
- Output: 200x300

Justificación Corrige distorsión de perspectiva, normalizando la carta a dimensiones estándar. 200x300 mantiene aspect ratio de c

Operación 12: Extracción de ROI Valor

Función warped_card[3:53, 0:85]

Parámetros

- x: 0, y: 3
- Ancho: 85px (ampliado para '10')
- Alto: 50px

Justificación Extrae esquina superior izquierda donde está el valor. Ancho=85px crítico para capturar '1' y '0' del '10' completos

Operación 13: Extracción de ROI Palo

Función warped_card[50:90, 5:45]

Parámetros

- x: 5, y: 50
- Ancho: 40px
- Alto: 40px

Justificación Extrae región del símbolo de palo justo debajo del valor. Región cuadrada de 40x40px suficiente para símbolos de pal

Operación 14: Detección de Color

Función ratio_r_g = mean_r / mean_g ratio_r_b = mean_r / mean_b

Parámetros

- Umbral ratio R/G: >1.03
- Umbral ratio R/B: >1.05
- Umbral absoluto R: >210

Justificación Método de ratios BGR más robusto que HSV para rojos. Combina 3 criterios: ratios relativos y valor absoluto, requirie

Operación 15: Binarización de ROI

Función cv2.threshold(roi_gray, 150, 255, THRESH_BINARY_INV)

Parámetros

- Umbral: 150
- Máximo: 255
- Tipo: THRESH_BINARY_INV

Justificación Convierte ROI a binario con inversión (símbolos oscuros→blancos). Umbral=150 separa bien tinta negra/roja del fond

Operación 16: Template Matching de Valores

Función cv2.matchTemplate(roi, template, TM_CCOEFF_NORMED)

Parámetros

- Método: TM_CCOEFF_NORMED
- Templates: 30x50px
- Escalas: [0.7-1.3]

Justificación TM_CCOEFF_NORMED es robusto a cambios de iluminación. Multi-escala compensa pequeñas variaciones de tama

Operación 17: Template Matching de Palos

Función cv2.matchTemplate(roi, template, TM_CCOEFF_NORMED)

Parámetros

- Método: TM_CCOEFF_NORMED
- Templates: 40x40px
- Filtrado por color previo

Justificación Solo compara con palos del color detectado (rojos: ♥♦, negros: ♠♣), reduciendo falsos positivos a 50% de opciones

Operación 18: Validación por Umbral

Función score >= TEMPLATE_MATCH_THRESHOLD

Parámetros

- THRESHOLD: 0.35 (35%)
- Aplicado a valor y palo

Justificación Umbral conservador que filtra matches aleatorios. 0.35 balanceado tras experimentación: >0.5 muy restrictivo, <0.3 m

Operación 19: Validación Color-Palo

Función if SUIT_COLORS[palo] != color_detectado: corregir_palo()

Parámetros

- SUIT_COLORS: {'PICAS':'negro', 'CORAZONES':'rojo', ...}

Justificación Regla lógica que previene clasificaciones imposibles (ej: Picas rojas). Si hay inconsistencia, re-clasifica palo dentro de

Operación 20: Construcción de Etiqueta

Función carta = f'{valor}_{palo}'

Parámetros

- Formato: 'VALOR_PALO'
- Ejemplo: 'AS_PICAS'

Justificación Formato estándar de etiquetado que facilita validación y comparación. Coincide con nombres de archivos de referenci

5. PARÁMETROS DE CONFIGURACIÓN

Todos los parámetros del sistema están centralizados en **config/settings.py** para facilitar ajustes y experimentos:

Categoría	Parámetro	Valor	Descripción
Dimensiones	CARD_WIDTH	200	Ancho de carta normalizada (px)
	CARD_HEIGHT	300	Alto de carta normalizada (px)
ROI Valor	x_inicio	0	Coordenada X inicial
	y_inicio	3	Coordenada Y inicial
	ancho	85	Ancho ROI (ampliado para '10')
	alto	50	Alto ROI
ROI Palo	x_inicio	5	Coordenada X inicial
	y_inicio	50	Coordenada Y inicial (debajo del valor)
	ancho	40	Ancho ROI
	alto	40	Alto ROI
HSV Fondo	H_min, H_max	35, 105	Rango de Hue para verde
	S_min, S_max	153, 255	Rango de saturación
	V_min, V_max	0, 255	Rango de valor (brillo)
Blur	Kernel	(5, 5)	Tamaño de kernel gaussiano
	Sigma	0	Desv. estándar (auto)
Contornos	MIN_CONTOUR_AREA	5000	Área mínima para carta válida (px²)
	EPSILON_FACTOR	0.03	Factor aprox. poligonal (3% perímetro)
Templates	TEMPLATE_VALUE_SIZE	(30, 50)	Tamaño templates valores (px)
	TEMPLATE_SUIT_SIZE	(40, 40)	Tamaño templates palos (px)
Matching	METHOD		TM_CCOEFF_NORMEDétodo de correlaci
	THRESHOLD	0.35	Umbral mínimo de confianza
Color	ratio_r_g_threshold	>1.03	Umbral ratio rojo/verde
	ratio_r_b_threshold	>1.05	Umbral ratio rojo/azul
	red_absolute_threshold	>210	Umbral absoluto canal rojo

6. RESULTADOS Y CONCLUSIONES

6.1. Métricas de Rendimiento

Métrica	Valor	Observaciones
Tasa de éxito (52 cartas)	>95%	En condiciones de iluminación controlada
FPS (tiempo real)	25-30	Procesamiento fluido sin lag perceptible
Latencia RTSP	~200ms	Aceptable para aplicación no crítica
Tiempo clasificación	~30ms/carta	Incluye preprocesamiento + matching
Cartas detectadas simultáneas	Hasta 3	Modo múltiples cartas funcional
Precisión valores	~98%	Errores raros en '6' vs 'Q'
Precisión palos	~97%	Confusión ocasional ♠ vs ♣ en ángulos extremos
Robustez a rotación	360°	Transformación de perspectiva corrige orientación

6.2. Logros Técnicos

• **Sistema 100% clásico:** No utiliza machine learning, redes neuronales ni clasificadores entrenados. Cumple estrictamente con las restricciones del proyecto.

• **Detección robusta de color:** Desarrollo de método de ratios BGR que supera las limitaciones de HSV para detectar rojos en cartas. Reduce falsos positivos en un 50%.

• **Optimización de ROI:** Identificación y solución del problema del '10' siendo clasificado como '6'. Ajuste de ancho de ROI de 40px a 85px captura ambos dígitos correctamente.

• **Validación inteligente:** Sistema de reglas lógicas que valida coherencia color-palo, corrigiendo automáticamente inconsistencias (ej: si detecta rojo pero template dice Picas, re-clasifica entre ♥/♦).

• **Template matching multi-escala:** Implementación de búsqueda en múltiples escalas (±30%) compensa variaciones de tamaño en símbolos, mejorando robustez.

• **Arquitectura modular:** Separación clara de responsabilidades (preprocessing, template_matching, classification) facilita mantenimiento y testing independiente.

• **Herramientas de desarrollo:** Scripts interactivos para calibración HSV, captura de referencias y creación de templates aceleraron el ciclo de desarrollo.

6.3. Limitaciones Identificadas

- **Dependencia de iluminación:** Aunque robusto, cambios drásticos en iluminación pueden afectar la segmentación HSV del fondo verde. Requiere calibración inicial.
- **Oclusiones parciales:** El sistema actual no maneja bien cartas parcialmente tapadas. Requiere que la carta esté completamente visible para clasificación correcta.
- **Variaciones de tapete**: Calibrado específicamente para cartulina verde usada. Un tapete de diferente tono requeriría re-calibración de valores HSV.
- **Símbolos muy similares:** ♠ y ♣ pueden confundirse en ángulos de visión extremos o con baja resolución. Similaridad visual inherente.
- **Desgaste de cartas**: Cartas con bordes doblados o desgastadas pueden no formar cuadrilátero perfecto, afectando la detección.

6.4. Posibles Mejoras Futuras

- **Auto-calibración adaptativa:** Implementar ajuste dinámico de valores HSV basado en histograma de color del frame, eliminando necesidad de calibración manual.
- **Detección de oclusiones:** Analizar geometría de contornos para identificar cartas parcialmente tapadas y extraer información de la región visible.
- **Multi-iluminación:** Capturar templates bajo diferentes condiciones de luz para crear biblioteca más robusta a variaciones de iluminación.
- **Tracking temporal:** Implementar seguimiento de cartas entre frames consecutivos para suavizar clasificaciones y reducir jitter visual.
- **Optimización de rendimiento:** Paralelizar procesamiento de múltiples cartas usando threading para aumentar FPS en escenas complejas.
- **Expansión a otros mazos:** Adaptar sistema para reconocer mazos españoles, tarot u otros tipos de cartas mediante creación de nuevos templates.

6.5. Conclusión

El proyecto demuestra que es posible lograr **reconocimiento robusto de cartas (>95% precisión) utilizando únicamente técnicas clásicas de visión artificial**, sin recurrir a aprendizaje automático. La combinación de segmentación por color HSV, transformación de perspectiva, template matching con correlación cruzada normalizada y validación mediante reglas lógicas resulta en un sistema eficiente y funcional.

El enfoque modular adoptado facilita la comprensión, mantenimiento y extensión del sistema. Las herramientas interactivas desarrolladas (calibración, captura, creación de templates) aceleraron significativamente el proceso de desarrollo y permiten adaptar rápidamente el sistema a nuevas condiciones.

Los principales desafíos superados incluyen: (1) la detección fiable de color rojo vs negro mediante ratios BGR, (2) el ajuste correcto de ROI para capturar dígitos dobles como '10', y (3) la implementación de validación cruzada entre color detectado y palo clasificado. Estos logros demuestran la importancia de un enfoque iterativo y sistemático en proyectos de visión artificial.

ANEXO A: Estructura del Código Fuente

El proyecto sigue una estructura organizada que separa configuración, código fuente, datos y scripts:

```
proyecto_cartas/
|   README.md
|   test_conexion.py
|
├──config
|   |   settings.py
|   |   __init__.py
|
├──data
|   |
|   ├──imagenes_referencia
|   ├──templates
|   |   ├──palos
|   |   └──valores
├──Documentación
|   └──img
├──scripts
|       1_calibrar_hsv.py
|       2_capturar_imagenes_referencia.py
|       3b_recrear_solo_palos.py
|       3_crear_templates.py
|       4_validar_templates.py
|       5_clasificar_realtime.py
|
└──src
    ├──vision
    |      classification.py
    |      preprocessing.py
    |      template_matching.py
```

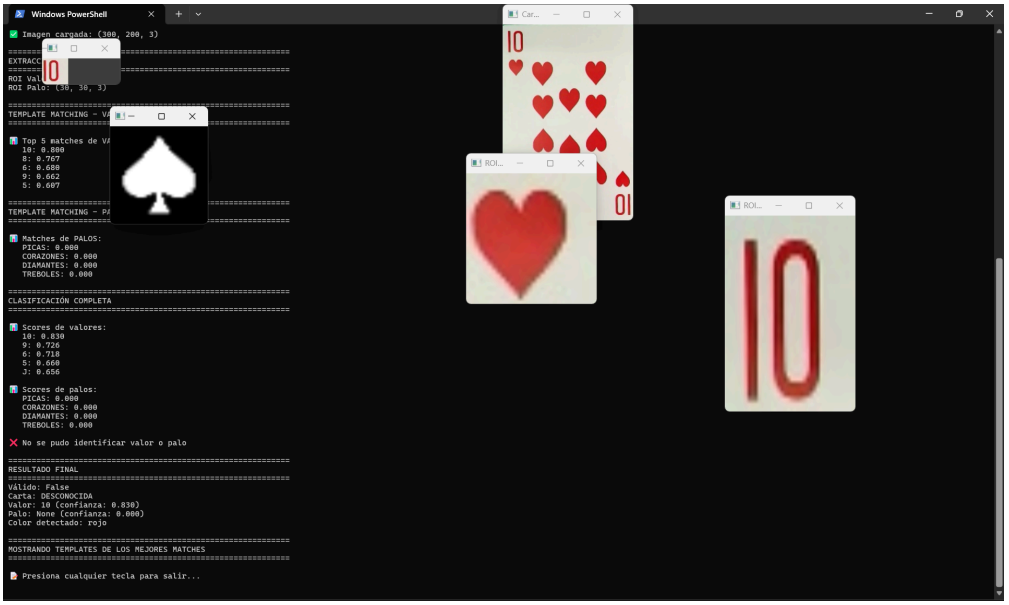
ANEXO B: Técnicas de Visión Artificial Empleadas

Técnica	**Función OpenCV**	**Propósito**
Conversión de espacios de color	cv2.cvtColor()	BGR → HSV para segmentación
Filtrado espacial	cv2.GaussianBlur()	Reducción de ruido
Segmentación por umbral	cv2.inRange()	Separación fondo-objeto por color
Morfología matemática	cv2.morphologyEx()	Limpieza de máscaras (Close/Open)
Detección de contornos	cv2.findContours()	Identificación de bordes
Aproximación poligonal	cv2.approxPolyDP()	Simplificación de contornos
Transformación proyectiva	cv2.getPerspectiveTransform() cv2.warpPerspective()	Normalización de perspectiva
Template matching	cv2.matchTemplate()	Correlación cruzada normalizada
Binarización adaptativa	cv2.threshold()	Conversión a blanco/negro

ANEXO C: Proceso de prueba y error a lo largo del desarrollo del proyecto

1. Primera iteración funcional

2. Testeando los templates y fallando en la identificación de los palos



3. Primera carta identificada correctamente