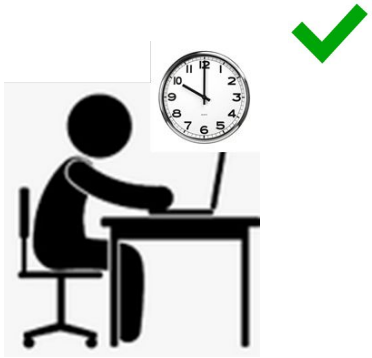
Universidad Intercontinental de la Empresa

Asignatura

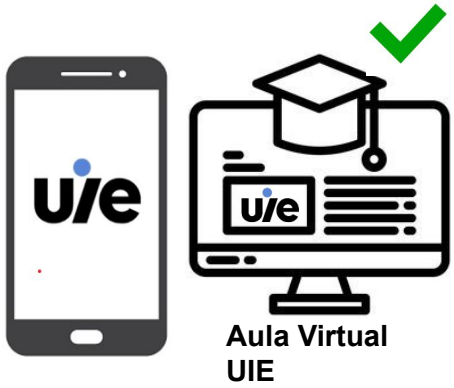
Computación gráfica

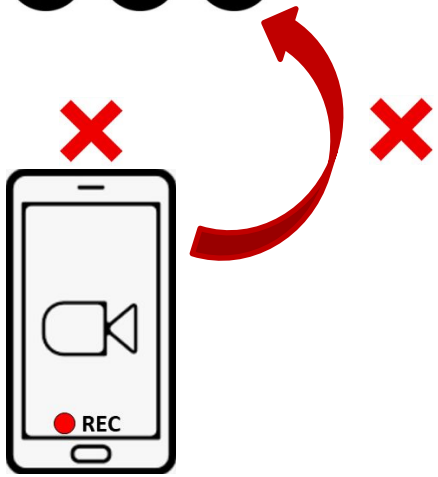
Profesor

David Cereijo Graña




Recordar


Aula Virtual
UIE


REC


Recordar

Participar

Universidad Intercontinental de la Empresa

Sesión 14

Unidad IV

Aplicaciones de la computación gráfica

Tema 4.3

Construcción computacional de escenas de realidad aumentada

Realidad aumentada sin marcadores

Ventajas y desventajas de la RA basada en marcadores

Ventajas:

- Algoritmo de detección menos costoso computacionalmente.
- Robustez frente a cambios en la iluminación.

Desventajas:

- No funciona correctamente si el marcador se encuentra parcialmente solapado.
- Necesidad de un patrón de imagen, poco estético, generalmente en **B&N** y de una forma predeterminada (cuadrado).
- Falta de coherencia con los objetos del mundo real.

Los marcadores son una buena forma de iniciarse en el uso de la **RA**, pero para una experiencia completa se requiere el uso de RA sin marcadores.

Ventajas y desventajas de la RA sin marcadores

Ventajas

- Puede ser utilizada para detectar objetos del mundo real.
- Funciona incluso si el objeto está parcialmente tapado.
- Puede tener cualquier forma y textura (excepto las texturas sólidas o de gradiente suave).

Desventajas

- Los objetos del mundo real carecen de form y estructura interna fijas y conocidas.
- Basada en algoritmos de reconocimiento de imagen y de detección de objetos, lo cual implica mayores necesidades de procesamiento, lo cual lo puede hacer inviable para determinados sistemas.

Funcionamiento de la RA sin marcadores

De modo general, el funcionamiento de una aplicación de RA sin marcadores comprende los siguientes pasos:

1. **Captura de imagen/vídeo:** la cámara del dispositivo captura el entorno en tiempo real.
2. **Preprocesamiento:** se aplican diversas técnicas para mejorar la imagen, reducir el ruido y ajustar parámetros como el brillo y el contraste.
3. **Detección de características:** se identifican puntos de interés en la imagen.
4. **Descripción de características:** se genera un descriptor o vector que representa la información local alrededor de cada característica detectada.
5. **Emparejamiento de características:** se buscan correspondencias entre puntos clave en diferentes fotogramas.
6. Estimación de la pose: se calcula la posición y orientación de la cámara respecto al entorno.
7. **Renderizado de objetos virtuales:** los objetos 3D se renderizan y alinean en el espacio 3D del entorno real.
8. **Visualización en tiempo real:** se muestra la escena real con los elementos virtuales superpuestos de manera coherente.

Detección y descripción de características

- **Detección de características:** proceso de identificar puntos de interés en una imagen que son únicos, repetible y robustos frente a transformaciones como rotación, cambio de escala, iluminación y perspectiva.

  El objetivo de esta fase es encontrar ubicaciones específicas en la imagen que puedan ser utilizadas para tareas posteriores, como el emparejamiento y seguimiento.

  Algunos ejemplos de características son: esquinas, bordes, blobs y puntos de alto contraste.

- **Descripción de características:** proceso que consiste en generar un descriptor o vector que representa de manera única la información local alrededor de cada característica detectada.

  Su objetivo es facilitar la comparación y emparejamiento de características entre diferentes imágenes o fotogramas.

Blob

Un blob es una región de una imagen que presenta una homogeneidad en ciertas propiedades visuales, y que se diferencia significativamente del fondo o de otras regiones de la imagen.

Algunos blobs comunes son:

- **Regiones de alta intensidad:** zonas brillantes sobre un fondo oscuro.
- **Regiones de baja intensidad:** zonas oscuras sobre un fondo brillante.
- **Regiones de color uniforme:** áreas con un color predominante que difiere del entorno.
- **Regiones de textura uniforme:** áreas con patrones de textura que contrastan con otras partes de la imagen.

Características de los Blobs

Para que un blob sea útil debe cumplir las siguientes características:

- **Invarianza a transformaciones:** los blobs son invariantes frente a la escala, rotación e iluminación, es decir, deben poder detectarse independientemente de su tamaño en la imagen, su orientación y los cambios en las condiciones de iluminación.
- **Repetibilidad:** la detección debe ser consistente en diferentes condiciones de captura y en distintas visualizaciones de la misma escena.
- **Localización precisa:** aplicaciones como el seguimiento de objetos (*tracking*) y la estimación de la pose requieren la identificación exacta de la ubicación del blob en la escena.

Detección de esquinas

En una imagen digital, una **esquina** es un punto donde se cruzan dos o más **bordes**, y se caracteriza por presentar variaciones significativas en la intensidad en direcciones perpendiculares.

En la RA sin marcadores, las esquinas se utilizan a menudo como puntos de referencia estables que el sistema puede rastrear a lo largo del tiempo.

Algunos de los principales algoritmos de detección de esquinas son:

- Algoritmo de Harris
- Algoritmo de Shi-Tomasi
- Algoritmo FAST

Algoritmo de Harris

El algoritmo de Harris es un método ampliamente utilizado en visión artificial para la detección de esquinas en imágenes.

El algoritmo se basa en la **matriz de auto-correlación** o **matriz de estructura**, **M**, que mide cómo cambia la intensidad de la imagen en un área vecina o vecindario alrededor de cada píxel.

La **matriz de autocorrelación** captura la variación de la intensidad en diferentes direcciones y se define como:

$$M = \begin{bmatrix} \sum w(x,y) I_x^2 & \sum w(x,y) I_x I_y \\ \sum w(x,y) I_x I_y & \sum w(x,y) I_y^2 \end{bmatrix}$$

Donde:

- I_x e I_y: derivadas de la imagen en las direcciones **x** e **y**, respectivamente.
- **w(w,y)**: función de ventana que define el vecindario alrededor del píxel de interés.

Algoritmo de Harris: Interpretación de la matriz de autocorrelación

La matriz de autocorrelación **M** es simétrica, y sus autovalores λ_1 y λ_2 representan la variación de la intensidad en las direcciones principales del vecindario.

En función del valor de los autovalores, se distinguen tres situaciones:

- **Región plana:** cuando ambos autovalores son pequeños ($\lambda_1 \approx 0$ y $\lambda_2 \approx 0$), indica que no hay cambios significativos en ninguna dirección.
- **Borde:** cuando uno de los autovalores es grande y el otro es pequeño ($\lambda_1 \gg \lambda_2$ o $\lambda_2 \gg \lambda_1$), indica que hay un cambio significativo en una dirección.
- **Esquina:** cuando ambos autovalores son grandes ($\lambda_1 \gg 0$ y $\lambda_2 \gg 0$) indica que hay cambios significativos en dos direcciones.

Algoritmo de Harris: Función de respuesta de Harris

Para evitar el cálculo explícito de los autovalores, Harris propuso una función de respuesta **R** que combina los autovalores:

$$R = \det(M) - k \cdot (\mathrm{tr}(M))^2$$

Donde:

- $\det(M) = \lambda_1 \lambda_2$: es el determinante de **M**.
- $\mathrm{tr}(M) = \lambda_1 + \lambda_2$: es la traza de **M** (suma de los elementos de la diagonal principal).
- **k:** es un parámetro empírico (generalmente entre $0.04 \leq k \leq 0.06$).

Interpretación de la función de respuesta:

- $R \gg 0$: indica una esquina.
- $R \approx 0$: indica una región plana.
- $R < 0$: indica un borde.

Algoritmo de Harris: Funcionamiento

El algoritmo de Harris se lleva a cabo en los siguientes pasos:

1. **Cálculo de derivadas:** se calculan I_x e I_y aplicando operadores de Sobel u otros filtros, para obtener las derivadas de la imagen en **x** e **y**.
2. **Cálculo de los productos de las derivadas:** se calculan I_x^2, I_y^2 e I_xI_y.
3. **Aplicación de la función de ventana:** se aplica una ventana para sumar los productos de las derivadas en el vecindario de cada píxel, obteniendo la matriz **M**.
4. **Cálculo de la respuesta R:** se calcula la función de respuesta, **R**, para cada píxel.
5. **Umbralización y supresión de no máximos:** se aplica un umbral para seleccionar los puntos con respuestas significativas, y se realiza una supresión de los no máximos para localizar de forma precisa las esquinas.

Algoritmo de Harris: Implementación con Python y OpenCV

```python
import cv2
import numpy as np

BLOCKSIZE = 2
KSIZE = 3
K = 0.04
UMBRAL = 0.01
COLOR_RESALTE = [0, 0, 255] # Color rojo

imagen = cv2.imread('cubo.png')
imagen_grises = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
imagen_grises = np.float32(imagen_grises)
esquinas = cv2.cornerHarris(imagen_grises, BLOCKSIZE, KSIZE, K)
esquinas = cv2.dilate(esquinas, kernel: None)
imagen[esquinas > UMBRAL * esquinas.max()] = COLOR_RESALTE
cv2.imshow( winname: 'Esquinas de Harris', imagen)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Algoritmo de Harris: Ventajas y limitaciones.

Ventajas:

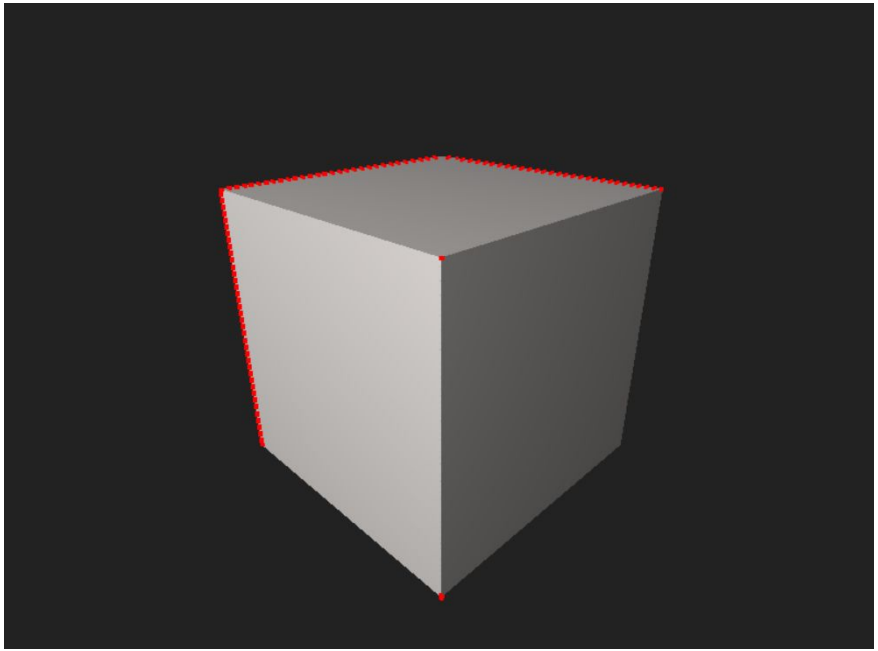
- **Robustez:** es resistente al ruido y a cambios en iluminación y contraste.
- **Eficiencia:** aunque es computacionalmente más costoso que otros algoritmos, es suficientemente eficiente como para ser utilizado en aplicaciones en tiempo real.
- **Invarianza:** es parcialmente invariante rotaciones y cambios de escala pequeños

Limitaciones:

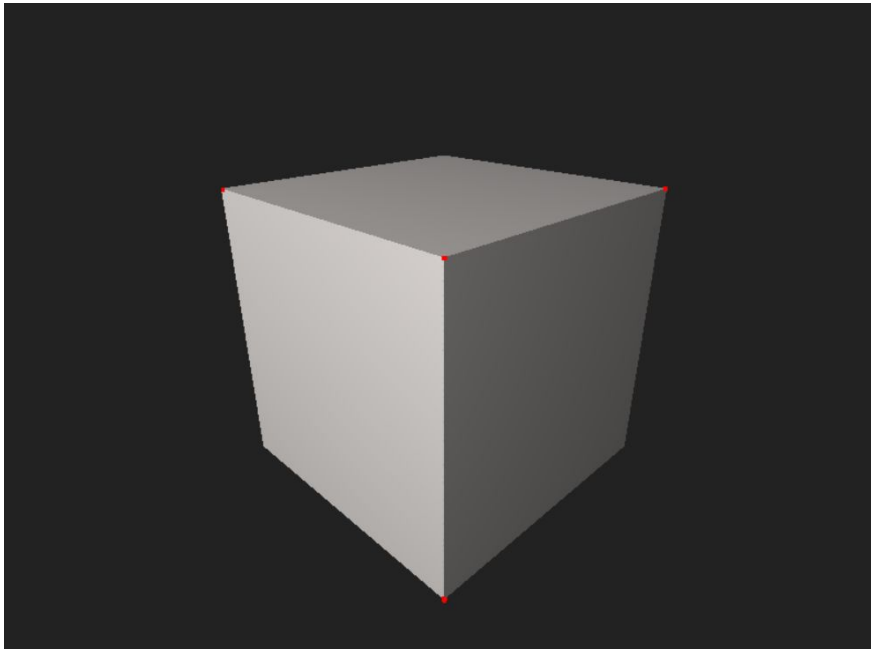
- **Escala:** no es invariante a cambios de escala grandes.
- **Sensibilidad:** la elección de los parámetros de operación puede afectar significativamente a los resultados, y puede requerir el ajuste para diferentes imágenes.

Algoritmo de Harris: Ejemplo 1

k = 0.04



k = 0.06



Algoritmo de Harris: Ejemplo 2

Algoritmo de Shi-Tomasi

El **algoritmo de Shi-Tomasi**, también conocido como *Good Features to Track* (buenas características para seguir), es una mejora del algoritmo de Harris.

Shi y Tomasi observaron que el criterio de Harris puede, en algunos casos, identificar puntos que no son óptimos para el seguimiento, de modo que propusieron una modificación basada en una interpretación más directa de los autovalores de la matriz **M**.

Al igual que el algoritmo de Harris, el algoritmo de Shi-Tomasi calcula la matriz **M** para cada píxel, utilizando las derivadas $\mathbf{I_x}$ e $\mathbf{I_y}$, pero en lugar de calcular la función de respuesta, **R**, calculan directamente los autovalores de **M**. El criterio de selección de esquinas de Shi-Tomasi consiste en seleccionar los puntos donde el menor de los autovalores es mayor que un umbral establecido:

$$\mathbf{\min(\lambda_1,\lambda_2) > umbral}$$

Este criterio asegura que ambos autovalores sean grandes, garantizando que el punto tenga variaciones significativas en ambas direcciones y, por tanto, sea una buena esquina.

Algoritmo de Shi-Tomasi: Ventajas sobre el algoritmo de Harris

El algoritmo de detección de esquinas de Shi-Tomasi presenta las siguientes ventajas frente al algoritmo de Harris.

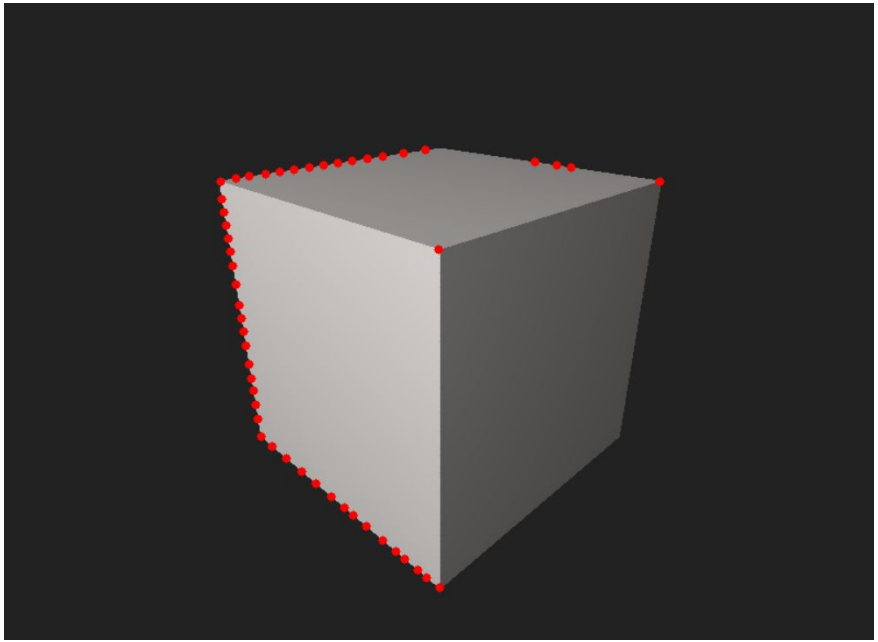
- **Mayor consistencia:** al no depender de k, se mejora su consistencia frente a diferentes imágenes y condiciones.
- **Mejor seguimiento:** al enfocarse en el mínimo de los autovalores, se garantiza que las esquinas seleccionadas tengan variaciones significativas en ambas direcciones, lo que las hace más adecuadas para el seguimiento.

Algoritmo de Shi-Tomasi: Implementación con Python y OpenCV

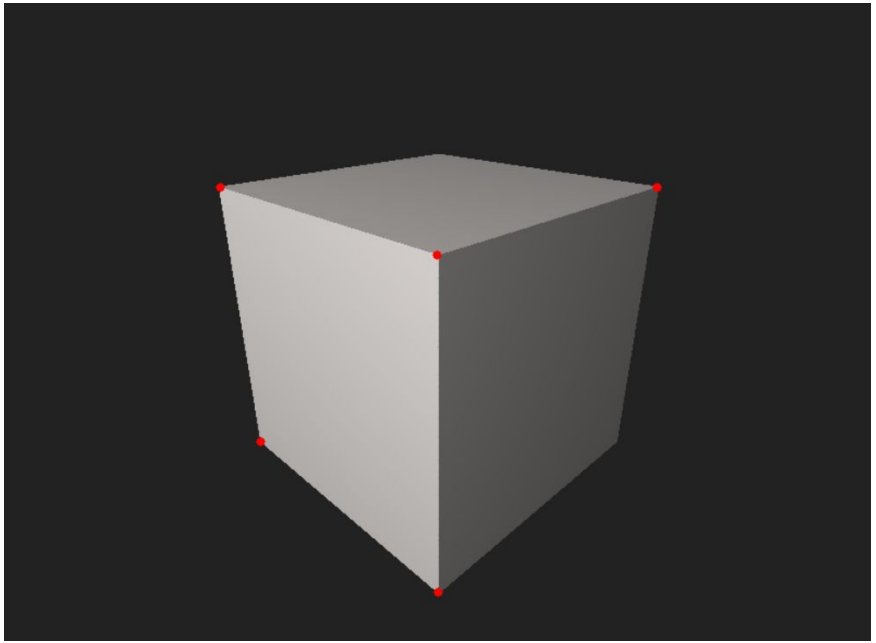
```python
import cv2
import numpy as np

NUMERO_ESQUINAS = 50
CALIDAD = 0.01
DISTANCIA_MINIMA = 10
COLOR_RESALTE = [0, 0, 255]  # Color rojo

imagen = cv2.imread('imagenes/cubo.png')
imagen_grises = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
esquinas = cv2.goodFeaturesToTrack(imagen_grises, NUMERO_ESQUINAS, CALIDAD, DISTANCIA_MINIMA)
if esquinas is not None:
    esquinas = np.int32(esquinas)
    for esquina in esquinas:
        x, y = esquina.ravel()
        cv2.circle(imagen, center: (x, y), radius: 4, COLOR_RESALTE, -1)
else:
    print("No se detectaron esquinas.")

cv2.imshow( winname: 'Algoritmo de detección de esquinas de Shi-Tomasi', imagen)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Algoritmo de Shi-Tomasi: Ejemplo 1

CALIDAD = 0.01



CALIDAD = 0.2



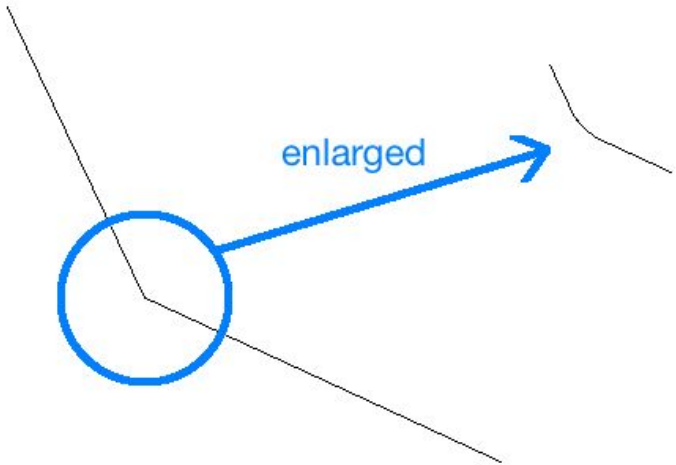
Algoritmo de Shi-Tomasi: Ejemplo 2

Limitaciones de la detección de esquinas

A la hora de analizar una imagen, se busca que el patrón a detectar sea invariante de aspectos como la escala, la rotación o la iluminación.

Un problema de la detección de esquinas, es que si ampliamos la imagen, lo que antes era detectado como una esquina, ahora puede ya no serlo.

Uno de los grandes problemas de la detección de esquinas es que no es invariante ante la escala.



Algoritmo SIFT (Scale Invariant Feature Transform)

El algoritmo **SIFT (*Scale-Invariant Feature Transform*)** es un método que permite detectar y describir características en imágenes.

A diferencia de los algoritmos de detección de esquinas, como los algoritmos de Harris y Shi-Tomasi, que son sensibles a cambios de escala y, en menor medida, a rotaciones, SIFT es invariante a escala, rotación y parcialmente invariante a cambios de iluminación y perspectiva.

Esto hace que el algoritmo SIFT sea muy adecuado para el emparejamiento de imágenes, el reconocimiento de objetos y la reconstrucción 3D.

SIFT es un algoritmo patentado cuyo uso comercial está sujeto a licencia.

Algoritmo SIFT: Funcionamiento

El algoritmo SIFT se lleva a cabo en cuatro etapas principales:

1. **Detección de extremos en el espacio-escala (scale-space extrema detection)**
2. **Localización de puntos clave (keypoint localization).**
3. **Asignación de orientación (orientation assignement)**
4. **Generación del descriptor de puntos clave (keypoint description)**

Algoritmo SIFT: Detección de extremos

En esta etapa, el objetivo es identificar puntos de interés en múltiples escalas para lograr invariancia a escala.

- **Construcción de la pirámide de escalas:** se construye una pirámide de escalas aplicando filtros Gaussianos con diferentes desviaciones estándar (σ) a la imagen original, generando versiones suavizadas de la imagen en distintas escalas.
- **Cálculo de la diferencia gaussiana:** a continuación, se calcula la Diferencia Gaussiana (DoG), restando imágenes suavizadas adyacentes en la pirámide, lo que ayuda a detectar blobs o regiones de interés.
- **Identificación de puntos clave:** los puntos clave candidatos se identifican buscando máximos y mínimos locales en las imágenes DoG, comparando cada píxel con sus vecinos en el espacio y en escala.

Los píxeles identificados como extremos locales en este espacio-escala se consideran como puntos clave potenciales.

Algoritmo SIFT: Localización de puntos clave

Una vez identificados los puntos clave candidatos, es necesario refinar su posición para mejorar la precisión y eliminar detecciones inestables.

- **Interpolación:** se interpola la posición exacta del punto clave en el espacio-escala mediante una aproximación de Taylor de segundo orden.
- **Descarte de puntos con bajo contraste:** los puntos con bajo contraste se descartan, ya que son susceptibles al ruido, evaluando el valor de la función DoG en esa posición.
- **Descarte de puntos con bordes mal definidos:** se eliminan los puntos que se encuentran en bordes mal definidos.

Esto asegura que los puntos clave finales sean estables y robustos frente a variaciones en iluminación y ruido.

Algoritmo SIFT: Asignación de orientación

Para lograr invariancia a rotación, a cada punto clave se le asigna una o más orientaciones basadas en las propiedades locales de la imagen.

- **Cálculo de la magnitud y la dirección:** se calcula la magnitud y dirección del gradiente en un vecindario alrededor del punto clave, utilizando la escala correspondiente.
- **Construcción del histograma de orientaciones:** se construye un histograma de orientaciones (generalmente con 36 bins, cada uno cubriendo 10 grados) ponderado por la magnitud del gradiente y una ventana Gaussiana centrada en el punto clave.
- **Asignación del punto clave:** la orientación con el pico más alto en el histograma se asigna al punto clave. Si existen picos secundarios que superan un cierto umbral respecto al pico principal, se pueden crear puntos clave adicionales con esas orientaciones.

Esto permite que el algoritmo sea invariante a rotaciones de la imagen, ya que el descriptor se calculará respecto a esta orientación asignada.

Algoritmo SIFT: Generación del descriptor de puntos clave

En esta etapa, se crea un descriptor único y robusto para cada punto clave que captura la información local de la imagen.

- **Extracción de región:** se extrae una región alrededor del punto clave, orientada según la orientación asignada previamente.
- **Subdivisión de la región:** esta región se divide en una cuadrícula de subregiones (por ejemplo, 4×4), formando un total de 16 subregiones.
- **Cálculo del histograma:** en cada subregión, se calcula un histograma de gradientes con 8 orientaciones, resultando en un vector de características de 128 dimensiones (16×8).
- **Ponderación de gradientes:** los gradientes se ponderan por una ventana Gaussiana para dar mayor importancia a los píxeles cercanos al centro del punto clave.
- **Normalización del descriptor:** el descriptor se normaliza para reducir los efectos de cambios en iluminación y contraste.

El resultado es un vector distintivo y robusto que permite emparejar puntos clave entre diferentes imágenes de manera eficiente y precisa.

Algoritmo SIFT: Implementación con Python y OpenCV

```python
import cv2

NUMERO_ESQUINAS = 0  # 0 significa que detectará todas las esquinas posibles
CONTRASTE_THRESHOLD = 0.04  # Umbral de contraste para filtrar puntos clave débiles
EDGE_THRESHOLD = 10  # Umbral para filtrar puntos clave en bordes
SIGMA = 1.6  # Desviación estándar para el filtro Gaussiano
COLOR_RESALTE = (0, 255, 0)  # Verde

imagen = cv2.imread('imagenes/uie.png')
imagen_gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
sift = cv2.SIFT_create(nfeatures=NUMERO_ESQUINAS,
                       contrastThreshold=CONTRASTE_THRESHOLD,
                       edgeThreshold=EDGE_THRESHOLD,
                       sigma=SIGMA)
keypoints, descriptores = sift.detectAndCompute(imagen_gris, None)
imagen_con_keypoints = cv2.drawKeypoints(imagen, keypoints, outImage: None, COLOR_RESALTE,
                                         flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow( winname: 'Algoritmo SIFT', imagen_con_keypoints)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Algoritmo SIFT: Ejemplo

Algoritmo SURF (Speeded Up Robust Feature)

El principal inconveniente de **SIFT** es su elevado coste computacional, lo que lo hace lento y poco adecuado para aplicaciones en tiempo real. Esta complejidad se debe principalmente al uso de convoluciones Gaussianas en su proceso de detección de características.

El algoritmo **SURF** optimiza este aspecto reemplazando el filtro Gaussiano por un filtro de caja, que ofrece una aproximación eficiente del suavizado Gaussiano. Esta mejora convierte a SURF en un algoritmo más rápido y sencillo de calcular, haciéndolo adecuado para entornos que requieren un procesamiento más ágil.

Sin embargo, SURF también está sujeto a patentes, y su uso en OpenCV requiere llevar a cabo una compilación específica del entorno.

Algoritmo SURF: Implementación con Python y OpenCV

```python
import cv2

HESSIAN_THRESHOLD = 400  # Umbral de Hessian para filtrar puntos clave
NO_OCTAVAS = 4           # Número de octavas en la pirámide de escalas
NO_INTERVALOS = 2        # Número de intervalos por octava
COLOR_RESALTE = (0, 255, 0)  # Verde

imagen = cv2.imread('imagenes/uie.png')
imagen_gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
surf = cv2.xfeatures2d.SURF_create(hessianThreshold=HESSIAN_THRESHOLD,
                                   nOctaves=NO_OCTAVAS,
                                   nOctaveLayers=NO_INTERVALOS)
keypoints, descriptores = surf.detectAndCompute(imagen_gris, None)
imagen_con_keypoints = cv2.drawKeypoints(imagen, keypoints, outImage: None, COLOR_RESALTE,
                                         flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow( winname: 'Puntos Clave de SURF', imagen_con_keypoints)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Algoritmo SURF: Implementación con Python y OpenCV

```
cv2.error: OpenCV(4.10.0)
D:\a\opencv-python\opencv-python\opencv_contrib\modules\xfeatures2d\src\surf
.cpp:1028: error: (-213:The function/feature is not implemented) This
algorithm is patented and is excluded in this configuration; Set
OPENCV_ENABLE_NONFREE CMake option and rebuild the library in function
'cv::xfeatures2d::SURF::create'
```

Algoritmo FAST (Features from Accelerated Segment Test)

Aunque **SURF** es más rápido que **SIFT**, sigue sin ser suficientemente rápido como para ser utilizado en aplicaciones en tiempo real. El algoritmo **FAST (*Features from Accelerated Segment Test*)** permite llevar a cabo una detección rápida de los puntos de interés. Su principal ventaja radica en su alta velocidad y simplicidad, lo que lo hace ideal para aplicaciones en tiempo real como la realidad aumentada y el seguimiento de objetos.

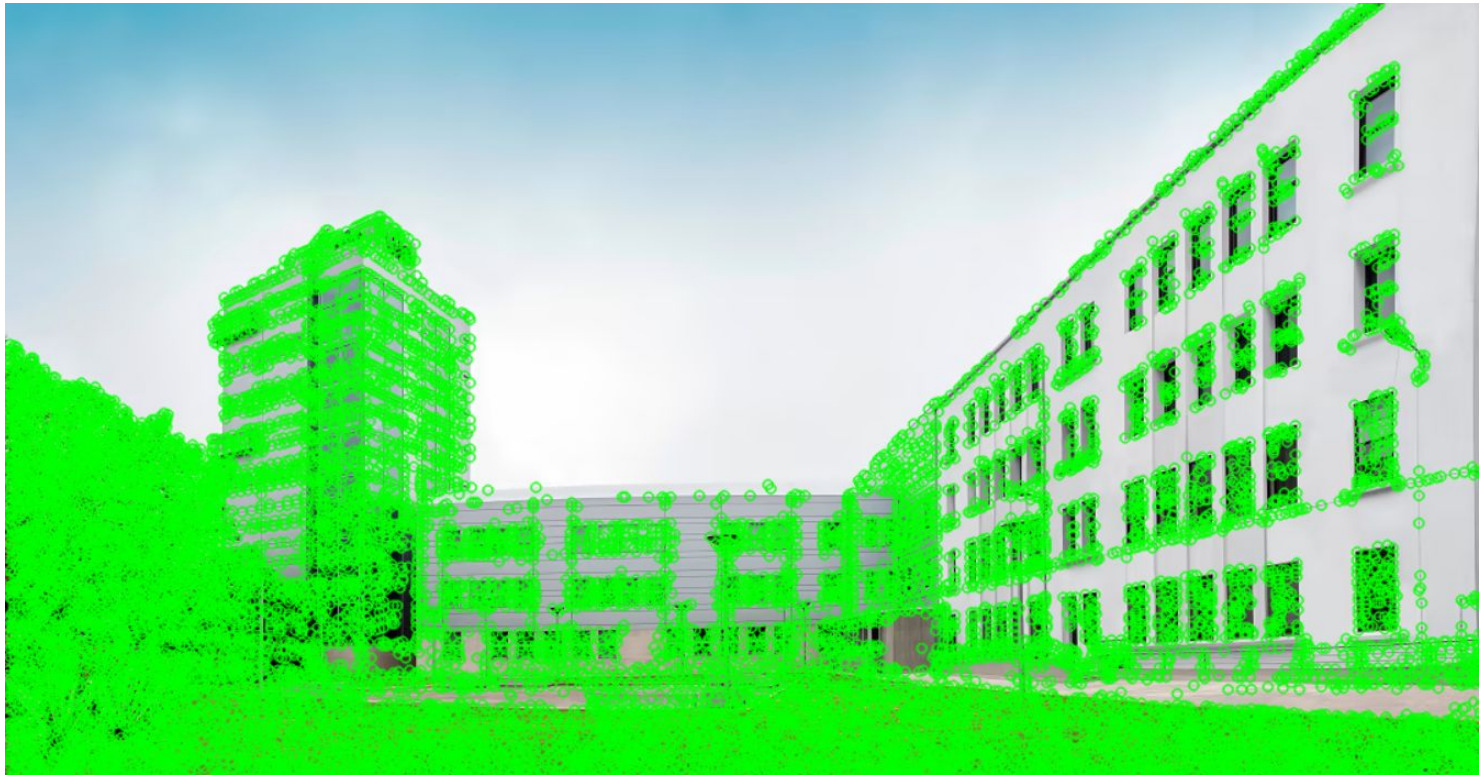
- Se examina cada píxel en una imagen en escala de grises y se compara con un círculo de 16 píxeles alrededor de él.
- Un píxel se considera un punto clave si al menos un número consecutivo de **n** píxeles en el círculo son significativamente más claros o más oscuros que el píxel central, según un umbral definido.
- Finalmente, se aplica supresión de no máximos para refinar los puntos clave, eliminando detecciones redundantes cercanas.

Es necesario tener en cuenta que **FAST** solo detecta los puntos de interés, para calcular sus descriptores necesitamos utilizar después otros algoritmo como **SIFT** o **SURF**.

Algoritmo FAST: Implementación con Python y OpenCV

```python
import cv2

UMBRAL_DE_CORTE = 10  # Umbral para la detección de píxeles como esquinas
NON_MAX_SUPPRESSION = True  # Aplicar supresión de no máximos
COLOR_RESALTE = (0, 255, 0)  # Verde

imagen = cv2.imread('imagenes/uie.png')
imagen_gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
fast = cv2.FastFeatureDetector_create(threshold=UMBRAL_DE_CORTE, nonmaxSuppression=NON_MAX_SUPPRESSION)
keypoints = fast.detect(imagen_gris, None)
imagen_con_keypoints = cv2.drawKeypoints(imagen, keypoints, outImage: None, COLOR_RESALTE,
                                         flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow( winname: 'Algoritmo FAST', imagen_con_keypoints)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Algoritmo FAST: Ejemplo



Algoritmo BRIEF (Binary Robust Independent Elementary Features)

Aunque utilicemos **FAST** para detectar los puntos clave, seguimos necesitando otro algoritmo como **SIFT** o **SURF** para calcular los descriptores.

El algoritmo **BRIEF** es un algoritmo rápido y eficiente, utilizado junto con detectores de puntos clave como **FAST** para generar descriptores binarios que facilitan el emparejamiento rápido entre imágenes.

- **BRIEF** crea descriptores comparando pares de píxeles dentro de una ventana local alrededor de cada punto clave.
- Para cada par, se realiza una comparación de intensidades: si el píxel **A** es más brillante que el píxel **B**, se asigna un bit de valor **1**; de lo contrario, se asigna **0**.
- Este proceso genera un vector binario compacto (por ejemplo, **256** bits) que representa la configuración de intensidades en la región.

Algoritmo BRIEF: Implementación con Python y OpenCV

```python
import cv2

NUMERO_PUNTOS = 500  # Número máximo de puntos clave a detectar
COLOR_RESALTE = (0, 255, 0)  # Verde en formato BGR

imagen = cv2.imread('imagenes/uie.png')

imagen_gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
fast = cv2.FastFeatureDetector_create()
keypoints = fast.detect(imagen_gris, None)
keypoints = sorted(keypoints, key=lambda x: -x.response)[:NUMERO_PUNTOS]
brief = cv2.xfeatures2d.BriefDescriptorExtractor_create()
keypoints, descriptores = brief.compute(imagen_gris, keypoints)
imagen_con_keypoints = cv2.drawKeypoints(imagen, keypoints, outImage: None, COLOR_RESALTE,
                                         flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow( winname: 'Algoritmo BRIEF', imagen_con_keypoints)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Algoritmo BRIEF: Ejemplo

Algoritmo ORB (Oriented FAST and Rotated BRIEF)

El algoritmo **ORB** combina **FAST** y **BRIEF**, fue creado por OpenCV Labs, y es rápido, robusto y open source.

Su funcionamiento es el siguiente:

1. **Detección de puntos clave:** utiliza **FAST** para identificar puntos de interés de forma rápida y eficiente.
2. **Asignación de orientación:** calcula la orientación dominante de cada punto clave basado en los gradientes locales, garantizando la invariancia a rotación.
3. **Generación de descriptores:** emplea BRIEF rotado según la orientación asignada, creando descriptores binarios compactos que facilitan el emparejamiento rápido mediante operaciones de bits.

Algoritmo ORB: Implementación con Python y OpenCV

```python
import cv2

NUMERO_PUNTOS = 500  # Número máximo de puntos clave a detectar
NIVEL_PIRAMIDE = 8    # Número de niveles en la pirámide de escala
UMBRAL_CORTE = 31     # Umbral para la detección de características
COLOR_RESALTE = (0, 255, 0)  # Verde

imagen = cv2.imread('imagenes/uie.png')
if imagen is None:
    print("No se pudo cargar la imagen. Asegúrate de que el nombre y la ruta sean correctos.")
    exit()
imagen_gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
orb = cv2.ORB_create(nfeatures=NUMERO_PUNTOS, nlevels=NIVEL_PIRAMIDE, edgeThreshold=UMBRAL_CORTE)
keypoints, descriptores = orb.detectAndCompute(imagen_gris, None)
imagen_con_keypoints = cv2.drawKeypoints(imagen, keypoints, outImage: None, COLOR_RESALTE,
                                         flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow( winname: 'Puntos Clave de ORB', imagen_con_keypoints)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Algoritmo ORB: Ejemplo

Referencias bibliográficas

- Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L. (1993). *Introduction to Computer Graphics*. Addison-Wesley Publishing Company.
- Méndez, M. (2022). *Introducción a la graficación por computadora*. https://proyectodescartes.org/iCartesiLibri/PDF/GraficacionComputadora.pdf