Universidad de Córdoba

Máster en Inteligencia Computacional e Internet de las Cosas

Markerless augmented reality application

Autor: Alberto Fernández Merchán
Asignatura: Realidad Virtual y Aumentada

Profesores: Manuel Jesús Marín Jiménez
Enrique Yeguas Bolívar

Índice

1. Introducción **2**

2. Metodología **3**
2.1. Detección de keypoints y extracción de descriptores . . . 3
2.2. Emparejamiento de descriptores entre imágenes . . . 4
2.3. Filtrado de coincidencias incorrectas . . . 5
2.4. Localización del objeto . . . 6
2.5. Visualización del contorno y presentación de información . . . 6
2.6. Sustitución de la región con un parche de otra imagen . . . 9
2.7. Sustitución de la región con un vídeo . . . 9

3. Resultados experimentales **11**
3.1. Patch y Vídeo . . . 14

4. Limitaciones **15**

5. Conclusiones **15**

Índice de figuras

1. Imagen modelo del cuadro *El caminante sobre un mar de nubes* . . . 2
2. Funcionalidad de Patch con el detector AKAZE. . . . 14
3. Frame del vídeo incrustado en el cuadro con el detector SIFT. . . . 14

Índice de cuadros

1. Resultados de los diferentes detectores con las diferentes escenas . . . 11
2. Resultados de los tres detectores en las escenas 1, 2 y 3 . . . 12
3. Tabla con las imágenes de la Escena 7 con diferentes detectores . . . 13
4. Comparación de detectores AKAZE, ORB y SIFT en las escenas 5 y 6 . . . 13
5. Tiempo medio de ejecución de cada uno de los descriptores . . . 14

1. Introducción

La Realidad Aumentada se utiliza cada vez más en sectores como el entretenimiento, la medicina o la industria aeroespacial. En esta práctica nos centraremos en su aplicación en el mundo del arte, concretamente en museos, donde puede servir para mostrar información adicional sobre los cuadros o incluso proyectar escenas animadas en su interior.

Para desarrollar la aplicación utilizaremos el lenguaje de programación C++ junto con la librería OpenCV [1], que proporciona una amplia variedad de herramientas útiles para el procesamiento de imágenes.

El objetivo del proyecto es detectar el cuadro *El caminante sobre el mar de nubes (1818)* (Figura 1), pintado por Caspar David Friedrich, en diferentes imágenes en las que aparezca la obra. Para ello, utilizaremos la técnica de homografía y probaremos diversos descriptores, con el fin de captar los *keypoints* que nos permitirán identificar la posición y orientación del cuadro en cada escena.

Figura 1: Imagen modelo del cuadro *El caminante sobre un mar de nubes*

Los descriptores que emplearemos serán dos binarios (AKAZE y ORB) y uno basado en números reales para calcular las distancias (SIFT).

La aplicación es capaz de detectar el cuadro en varias imágenes con diferentes escenas y de construir una *bounding box* que delimita la región donde se ha identificado. Además, en la esquina superior izquierda de la imagen se muestra un cuadro de texto que indica el nombre de la obra, el año de creación y el autor.

También ofrece la opción de reemplazar la región detectada por otra imagen utilizando la opción *–patch*. En lugar de una imagen, también se puede optar por colocar un vídeo en dicha región con la opción *–video*.

En general, las opciones que ofrece la aplicación son las siguientes:

- **--model** «**ruta**»
  Especifica la ruta al modelo de imagen que contiene la obra *El caminante sobre el mar de nubes* para su detección en las escenas.

- **--scene** «**ruta**»
  Define la ruta al archivo de imagen que contiene la escena en la que se desea detectar la obra. Esta imagen se utilizará para realizar la comparación con el modelo de referencia.

- **--patch «ruta»**
  Permite reemplazar la región detectada del cuadro por una imagen específica. El parámetro ¡ruta¿debe señalar el archivo de imagen que se usará como sustituto.

- **--video «ruta»**
  Especifica la ruta a un archivo de vídeo. En este caso, la aplicación podrá aplicar la detección de la obra en cada uno de los fotogramas del vídeo.

- **--save-video «ruta»**
  Si se está utilizando un archivo de vídeo, esta opción permite guardar el vídeo procesado en una ruta especificada. Se genera un nuevo archivo de vídeo con la obra detectada o reemplazada en cada fotograma.

- **--descriptor «AKAZE/ORB/SIFT»**
  Permite seleccionar el descriptor a utilizar para la detección de los *keypoints*. Los descriptores disponibles son:

  - **AKAZE**: Descriptor binario.
  - **ORB**: Descriptor binario.
  - **SIFT**: Descriptor basado en valores reales.

- **--experimental «numero de ejecuciones»**
  Permite ejecutar el proceso varias veces para obtener una media de tiempo de ejecución. Esta opción se utilizará para comprobar qué método es más rápido.

- **--help**
  Muestra un mensaje con todas las opciones disponibles y su descripción.

2. Metodología

En esta sección comentaremos como se desarrolla el programa y los diferentes pasos que toma para detectar la región donde aparece el cuadro que hemos marcado como **modelo**.

2.1. Detección de keypoints y extracción de descriptores

En este paso, se realiza la detección de puntos clave (*keypoints*) y la extracción de descriptores, que permiten realizar el emparejamiento entre las imágenes de la escena y el modelo. En esta práctica, se utilizan varios algoritmos de detección de puntos clave y extracción de descriptores, tales como ORB (Oriented FAST and Rotated BRIEF) AKAZE (Accelerated KAZE) y SIFT (Scale-Invariant Feature Transform); los dos primeros binarios y el último trabaja con números reales.

La función principal que se utiliza para detectar puntos clave y calcular los descriptores es `detectAndCompute()` de OpenCV. Esta función toma una imagen como entrada y devuelve dos objetos: un conjunto de puntos clave y los descriptores correspondientes. Los descriptores son vectores que representan las características locales alrededor de los puntos clave.

- **ORB [2]:** El algoritmo ORB es una combinación de los métodos FAST (para la detección de puntos clave) y BRIEF (para la extracción de descriptores binarios). Es eficiente y rápido, lo que lo hace adecuado para aplicaciones en tiempo real. Su principal limitación es que no es tan robusto ante cambios drásticos de iluminación como otros algoritmos.

- **SIFT [3]:** El algoritmo SIFT es adecuado para detectar características en diferentes escalas y es resistente a transformaciones de escala y rotación. Sin embargo, SIFT es más costoso computacionalmente en comparación con ORB y no es tan rápido.

- **AKAZE [4]:** AKAZE es una mejora de KAZE, que detecta puntos clave en una escala de imágenes, pero de manera más eficiente y rápida. Al igual que SIFT, AKAZE es invariante a transformaciones de escala y rotación, pero es más rápido en comparación.

El código que se usa en la práctica para aplicar estos descriptores es el siguiente

```
void rva_calculaKPsDesc(
    const cv::Mat &img,
    std::vector<cv::KeyPoint> &keypoints,
    cv::Mat &descriptors,
    const std::string &type
)
{
    cv::Ptr<cv::Feature2D> detector;

    if (type == "ORB") {
        detector = cv::ORB::create();
    } else if (type == "SIFT") {
        detector = cv::SIFT::create();
    } else if (type == "AKAZE") {
        detector = cv::AKAZE::create();
    } else {
        std::cerr << "Tipo de detector no reconocido: " << type << std::endl;
        return;
    }

    detector->detectAndCompute(img, cv::noArray(), keypoints, descriptors);
}
```

Al utilizar estos métodos, se puede comparar el modelo con la escena y encontrar las coincidencias entre ellos, lo que es el siguiente paso en el proceso.

2.2. Emparejamiento de descriptores entre imágenes

El siguiente paso es el de emparejar los *keypoints* del modelo y de la imagen de escena. Para ello utilizaremos la clase *DescriptorMatcher* de *OpenCV*, que nos permite aplicar algoritmos de emparejamiento por fuerza bruta.

En función del tipo de descriptor utilizado (por ejemplo, ORB, SIFT o AKAZE), se debe elegir el emparejador adecuado. Los descriptores como ORB o AKAZE generan vectores binarios, por lo que se utiliza el emparejador `cv::DescriptorMatcher::BRUTEFORCE_HAMMING`, que mide la distancia de Hamming entre vectores. Por otro lado, los descriptores como SIFT generan vectores de tipo `float`, y en estos casos se usa `cv::DescriptorMatcher::BRUTEFORCE`, que emplea la distancia euclídea (L2) para comparar descriptores.

Una vez definido el tipo de emparejador, se aplica el método `match()` para obtener los emparejamientos directos. Este proceso garantiza que los puntos emparejados entre las imágenes sean lo más fiables y coherentes posible, lo que influye directamente en la estimación de la homografía y el posterior reconocimiento del objeto.

El código que se encarga de esta etapa es el que se recoge en la función `rva_matchDesc()`.

```
void rva_matchDesc(cv::Mat &descriptors1, cv::Mat &descriptors2,
                   std::vector<cv::DMatch> &matches, const std::string &type){
    cv::Ptr<cv::DescriptorMatcher> matcher;
    // Elegir el tipo de emparejador según el tipo de descriptores
    if (type == "ORB" || type == "AKAZE")
        matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::BRUTEFORCE_HAMMING);
    else
        matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::BRUTEFORCE);
    // Realizar el emparejamiento
    matcher->match(descriptors1, descriptors2, matches);
}
```

2.3. Filtrado de coincidencias incorrectas

Una vez obtenidas las coincidencias entre los descriptores de ambas imágenes, es normal que algunas de ellas sean erróneas o no estén alineadas geométricamente. Para mejorar la fiabilidad, es necesario realizar un filro de estos emparejamientos. Esto se logra mediante el cálculo de una homografía usando el algoritmo `RANSAC` (*Random Sample Consensus*), que permite identificar las correspondencias coherentes con una transformación proyectiva.

La función `cv::findHomography()` de OpenCV permite calcular la homografía entre dos conjuntos de puntos, devolviendo una máscara binaria que indica qué emparejamientos son considerados coherentes respecto a la geometría estimada. Aquellos que no cumplan dicha coherencia son descartados.

En el siguiente fragmento de código se muestra cómo se implementa este filtrado. Primero se extraen los puntos clave correspondientes a partir de los emparejamientos iniciales. Luego se calcula la homografía usando `RANSAC`, y finalmente se seleccionan sólo aquellos emparejamientos que son consistentes con dicha homografía.

```cpp
void rva_filterMatches(const std::vector<cv::DMatch> &matches,
                       const std::vector<cv::KeyPoint> &keypoints_model,
                       const std::vector<cv::KeyPoint> &keypoints_scene,
                       std::vector<cv::DMatch> &good_matches)
{
    /*
    Obtienes los puntos correspondientes a partir de los matches iniciales.
    Calculas la homografía usando findHomography() con RANSAC.
    OpenCV devuelve una máscara (inliers_mask) que indica qué matches son inliers (coherentes con la homografía).
    Los matches que no cumplan esa coherencia geométrica se descartan.
    */
    if (matches.empty())
    {
        std::cerr << "Error: No hay matches para filtrar." << std::endl;
        return;
    }

    // Extraer los puntos de cada match
    std::vector<cv::Point2f> points_model, points_scene;
    for (const auto &match : matches)
    {
        points_model.push_back(keypoints_model[match.queryIdx].pt);
        points_scene.push_back(keypoints_scene[match.trainIdx].pt);
    }

    // Calcular homografía con RANSAC para obtener inliers
    cv::Mat inlier_mask;
    double default_threshold = 3.0;
    cv::findHomography(points_model, points_scene, cv::RANSAC, default_threshold, inlier_mask);

    // Filtrar los matches usando la máscara
    for (size_t i = 0; i < matches.size(); ++i)
        if (inlier_mask.at<uchar>(i))
            good_matches.push_back(matches[i]);
}
```

2.4. Localización del objeto

Una vez filtradas las correspondencias incorrectas, el siguiente paso consiste en localizar el objeto de interés en la imagen de escena. Esto se logra mediante la estimación de una transformación geométrica, concretamente, una homografía, que permite proyectar las esquinas del modelo sobre la imagen de escena.

La homografía se calcula a partir de los *keypoints* emparejados y ya filtrados mediante el algoritmo `RANSAC`, de forma que solo se utilicen correspondencias fiables (aunque en la práctica se han mantenido tanto los puntos filtrados como los no filtrados para poder realizar comparaciones).

Una vez estimada la homografía, se define un conjunto de puntos que representan las esquinas del objeto en su imagen original. Estos puntos se transforman utilizando la función `cv::perspectiveTransform()`, obteniendo así su ubicación proyectada en la imagen de escena.

El siguiente código muestra este proceso paso a paso:

```
void rva_localizaObj(const cv::Mat &img1, const cv::Mat &img2,
                     std::vector<cv::KeyPoint> &keypoints1,
                     std::vector<cv::KeyPoint> &keypoints2,
                     std::vector<cv::DMatch> &matches,
                     cv::Mat &homography,
                     std::vector<cv::Point2f> &pts_im2)
{
    // Usamos los matches ya filtrados, no es necesario recalcular descriptores aquí
    std::vector<cv::Point2f> obj_points;
    std::vector<cv::Point2f> scene_points;

    // Obtener los puntos correspondientes de los matches
    for (size_t i = 0; i < matches.size(); i++)
    {
        obj_points.push_back(keypoints1[matches[i].queryIdx].pt);
        scene_points.push_back(keypoints2[matches[i].trainIdx].pt);
    }

    // Calcular la homografía entre las imágenes utilizando los matches (ya filtrados
    ↪  previamente)
    homography = cv::findHomography(obj_points, scene_points, cv::RANSAC);

    // Definir las esquinas del objeto (modelo) en img1
    std::vector<cv::Point2f> obj_corners(4);
    obj_corners[0] = cv::Point2f(0, 0);
    obj_corners[1] = cv::Point2f((float)img1.cols, 0);
    obj_corners[2] = cv::Point2f((float)img1.cols, (float)img1.rows);
    obj_corners[3] = cv::Point2f(0, (float)img1.rows);

    // Transformar las esquinas usando la homografía calculada
    std::vector<cv::Point2f> scene_corners(4);
    cv::perspectiveTransform(obj_corners, scene_corners, homography);

    // Almacenar las coordenadas de las esquinas transformadas
    pts_im2 = scene_corners;
}
```

2.5. Visualización del contorno y presentación de información

Una vez localizada la posición del objeto en la imagen de escena, se procede a dibujar su contorno y a mostrar información relevante sobre su detección.

Para visualizar el contorno del objeto se utiliza la función `cv::line()` de OpenCV, que permite conectar las esquinas proyectadas mediante líneas y formar así una *bounding box*. La función implementada `rva_draw_contour` recibe como parámetros la imagen, los puntos transformados, un color y el grosor de línea. En caso de que no haya al menos cuatro puntos, se lanza un mensaje de error y no se realiza ninguna operación.

Para dibujar la *bounding box* se ha utilizado este código:

```
void rva_draw_contour(cv::Mat image, std::vector<cv::Point2f> points,
                      cv::Scalar color, int thickness)
{
    // Verificar que haya al menos 4 puntos para dibujar el contorno
    if (points.size() < 4)
    {
        std::cerr << "Error: Se requieren al menos 4 puntos para dibujar un contorno." <<
        ↪  std::endl;
        return;
    }

    // Dibujar líneas entre los puntos en orden
    for (size_t i = 0; i < points.size(); i++)
        cv::line(image, points[i], points[(i + 1) % points.size()], color, thickness);
}
```

Además del contorno, también se ha implementado una función para mostrar información del cuadro sobre la imagen. Para ello se utiliza la función `cv::putText()`, que permite renderizar texto sobre una imagen con control sobre el tamaño, posición, grosor y tipo de fuente.

La función `rva_mostrarInfo` calcula la mejor escala para el texto en función del tamaño del contorno proyectado. Primero determina el área disponible para el texto (a partir del rectángulo delimitador), y luego ajusta el tamaño del texto para que todas las líneas proporcionadas quepan de forma legible en dicha área. También se añade un fondo semitransparente blanco detrás de cada línea para asegurar que el texto sea legible.

Para mostrar la información se ha usado este código:

```cpp
void rva_mostrarInfo(cv::Mat &image, const std::vector<cv::String> &info,
                     const std::vector<cv::Point2f> &vertices){
    if (vertices.size() < 4 || info.empty())
        return;
    float min_x = vertices[0].x, max_x = vertices[0].x;
    float min_y = vertices[0].y, max_y = vertices[0].y;
    for (const auto &v : vertices){
        min_x = std::min(min_x, v.x);
        max_x = std::max(max_x, v.x);
        min_y = std::min(min_y, v.y);
        max_y = std::max(max_y, v.y);
    }

    float box_width = max_x - min_x;
    float box_height = max_y - min_y;
    float line_spacing_factor = 0.2f;
    float margin_x = box_width * 0.05f;
    float margin_y = box_height * 0.05f;
    float available_height = box_height - 2 * margin_y;
    float max_line_height = available_height / info.size();
    float font_scale_height = max_line_height / 10.0f;
    float font_scale_width = 10.0f;
    for (const auto &line : info){
        int baseline = 0;
        cv::Size text_size = cv::getTextSize(line, cv::FONT_HERSHEY_SIMPLEX,
        1.0, 1, &baseline);
        float candidate_scale = (box_width - 2 * margin_x) / (float)(text_size.width + 1e-6);
        if (candidate_scale < font_scale_width)
            font_scale_width = candidate_scale;
    }
    float font_scale = std::min(font_scale_height, font_scale_width);
    float start_x = min_x + margin_x;
    float start_y = min_y + margin_y;
    for (size_t i = 0; i < info.size(); i++){
        cv::Point2f text_pos(start_x, start_y + i * (max_line_height * line_spacing_factor));

        int baseline = 0;
        cv::Size text_size = cv::getTextSize(info[i], cv::FONT_HERSHEY_SIMPLEX, font_scale, 1,
        ↪  &baseline);
        cv::Rect background_rect(
            text_pos.x - 2,
            text_pos.y - text_size.height - 2,
            text_size.width + 4,
            text_size.height + baseline + 4
        );
        background_rect &= cv::Rect(0, 0, image.cols, image.rows);
        if (background_rect.area() <= 0) continue;
        cv::Mat roi = image(background_rect);
        cv::Mat overlay;
        roi.copyTo(overlay);
        cv::Mat white_rect(roi.size(), roi.type(), cv::Scalar(255, 255, 255));
        double alpha = 0.5;
        cv::addWeighted(white_rect, alpha, overlay, 1.0 - alpha, 0.0, overlay);
        overlay.copyTo(roi);
        cv::putText(image, info[i], text_pos, cv::FONT_HERSHEY_SIMPLEX, font_scale,
                    cv::Scalar(0, 0, 0), 1, cv::LINE_AA);
    }
}
```

2.6. Sustitución de la región con un parche de otra imagen

Una vez localizada la posición del objeto en la imagen de la escena, es posible sustituir esa región por un parche proveniente de otra imagen.

Para ello, se aplica la misma matriz de homografía estimada previamente para proyectar el parche sobre la imagen de escena. El proceso consta de varios pasos, todos implementados en la función `rva_dibujaPatch`.

Primero, se utiliza `cv::warpPerspective()` para deformar el parche de acuerdo a la homografía, adaptando su forma y orientación a la región del cuadro detectado en la imagen destino. Para que solo se sobrescriba la zona de interés, se genera una máscara blanca del mismo tamaño que el parche original, la cual también se transforma con la misma homografía.

Por último, se copia el parche deformado sobre una copia de la imagen original usando la máscara proyectada, lo que hace que se reemplace, únicamente, la región correspondiente.

El código que se ha utilizado es el siguiente:

```
void rva_dibujaPatch(const cv::Mat &scene, const cv::Mat &patch, const cv::Mat &homography,
↪  cv::Mat &output)
{
    if (homography.empty())
    {
        std::cerr << "Error: La matriz de homografía está vacía. No se puede dibujar el patch."
        ↪  << std::endl;
        return;
    }

    // 1. Proyectar el patch
    cv::Mat warped_patch;
    cv::warpPerspective(patch, warped_patch, homography, scene.size(), cv::INTER_LINEAR,
    ↪  cv::BORDER_CONSTANT);

    // 2. Crear máscara blanca del tamaño del patch original
    cv::Mat patch_mask = cv::Mat::ones(patch.size(), CV_8UC1) * 255;

    // 3. Proyectar la máscara igual que el patch
    cv::Mat warped_mask;
    cv::warpPerspective(patch_mask, warped_mask, homography, scene.size());

    // 4. Copiar a la salida usando la máscara alineada
    output = scene.clone();
    warped_patch.copyTo(output, warped_mask);
}
```

2.7. Sustitución de la región con un vídeo

De forma parecida a la sustitución con una imagen estática, es posible reemplazar la región del objeto detectado por los fotogramas de un vídeo, logrando así un efecto de realidad aumentada en tiempo real.

Para ello, se lee el vídeo fotograma a fotograma utilizando `cv::VideoCapture`. Cada uno de estos cuadros se redimensiona al tamaño del objeto original y se proyecta sobre la imagen de la escena mediante la homografía previamente calculada, utilizando la función `rva_dibujaPatch`.

La región del objeto se actualiza en cada iteración del bucle, generando la ilusión de que el vídeo está superpuesto en la escena.

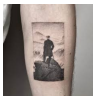
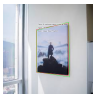
El sistema también permite capturar capturas de pantalla presionando la tecla `s`, así como guardar la salida como un nuevo vídeo si se habilita la opción correspondiente. En ese caso, se inicializa un objeto `cv::VideoWriter` para escribir cada *frame* resultante en un archivo de salida en formato MP4 utilizando el códec H.264.

Finalmente, se muestra una barra de progreso que permite seguir el procesamiento del vídeo y, en cualquier momento, el usuario puede interrumpir el proceso presionando una tecla.

```
            // Obtener información del video
            int totalFrames = static_cast<int>(cap2.get(cv::CAP_PROP_FRAME_COUNT));
            int currentFrame = 0;
            cv::Mat frame, output;
            cv::VideoWriter video_writer;
            bool video_initialized = false;
            while (cap2.read(frame))
            {
                if (frame.empty())
                {
                    std::cerr << "Frame vacío. Finalizando..." << std::endl;
                    break;
                }
                img_scene.copyTo(output);
                cv::Mat resized_frame;
                cv::resize(frame, resized_frame, img_model.size());
                rva_dibujaPatch(output, resized_frame, H, output);
                // Mostrar el resultado en la ventana
                cv::imshow("AugmentedReality-Video", output);

                // Inicializar el VideoWriter solo la primera vez
                if (save_video && !video_initialized)
                {
                    video_writer.open(save_video_path, cv::VideoWriter::fourcc('H', '2', '6',
                    ↪  '4'), 30, output.size(), true); // codec para guardar en .mp4 --> h264
                    if (!video_writer.isOpened())
                    {
                        std::cerr << "Error al abrir el archivo de video para guardar." <<
                        ↪  std::endl;
                        break;
                    }
                    video_initialized = true;
                }
                // Escribir el frame en el archivo de video si la opción de guardar está activada
                if (save_video)
                    video_writer.write(output);

                // Mostrar barra de progreso
                currentFrame++;
                showProgressBar(currentFrame, totalFrames);

                // Esperar 30ms para la tecla (y actualizar la barra de progreso sin bloquear el
                ↪  video)
                int key = cv::waitKey(30);
                if (key == 's' || key == 'S')
                    screenshot(scene_path, descriptor_type, img_scene); // guarda captura de
                    ↪  pantalla
                else if (key >= 0)
                    break; // Salir si se presiona alguna tecla
            }
```

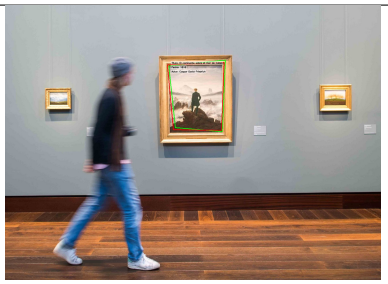
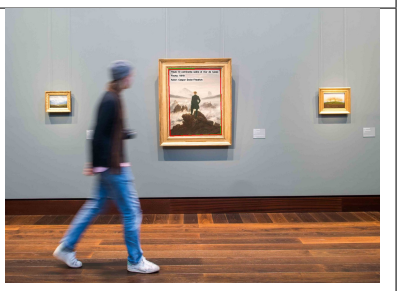
3. Resultados experimentales

Finalmente, en esta sección veremos los resultados obtenidos con los diferentes descriptores en las escenas que hemos seleccionado. Estos resultados se pueden ver en la Tabla 1.

	Scene 1	Scene 2	Scene 3	Scene 4	Scene 5	Scene 6	Scene 7	Scene 8	Scene 9	Scene 10
ORB										
AKAZE										
SIFT										

Cuadro 1: Resultados de los diferentes detectores con las diferentes escenas

Destacamos las escenas 1, 2 y 3, que se puede ver en la Tabla 2, donde se detecta el cuadro con los tres algoritmos de selección de *keypoints*, sin embargo, `SIFT` funciona mejor que los otros dos delimitando correctamente el marco del cuadro. Los descriptores binarios funcionan peor ya que muestran deformaciones en la bounding box.

AKAZE	ORB	SIFT
Escena 1 con AKAZE	**Escena 1 con ORB**	**Escena 1 con SIFT**
Escena 2 con AKAZE	**Escena 2 con ORB**	**Escena 2 con SIFT**
		
Escena 3 con AKAZE	**Escena 3 con ORB**	**Escena 3 con SIFT**

Cuadro 2: Resultados de los tres detectores en las escenas 1, 2 y 3

También querría destacar la escena 7 (Tabla 3), ya que es una imagen que no pertenece al cuadro original, sino a un dibujo realizado digitalmente. El detector SIFT es capaz de detectar el cuadro, mientras que los detectores binarios no pueden determinar una bounding box.

Aunque la bounding box se sale de los limites del cuadro, vemos que sí ha sido capaz de detectarlo.

También destacamos las escenas 5 y 6, que se pueden ver en la Tabla 4 donde ninguno de los tres algoritmos ha conseguido detectar nada.

Esto puede ser debido a que en la escena 5 la imagen es un tatuaje en blanco y negro y puede que no haya conseguido detectar los suficientes keypoints y, en la escena 6, el cuadro está demasiado inclinado y posiblemente tampoco haya podido filtrar los suficientes keypoints.

AKAZE	ORB	SIFT
Escena 7 con AKAZE	**Escena 7 con ORB**	**Escena 7 con SIFT**

Cuadro 3: Tabla con las imágenes de la Escena 7 con diferentes detectores

Escena 5 - AKAZE	**Escena 5 - ORB**	**Escena 5 - SIFT**
Escena 6 - AKAZE	**Escena 6 - ORB**	**Escena 6 - SIFT**

Cuadro 4: Comparación de detectores AKAZE, ORB y SIFT en las escenas 5 y 6

Para evaluar el tiempo de procesamiento, se realizaron 10 ejecuciones de cada algoritmo sobre la escena 1, calculando el tiempo medio de ejecución. Los resultados, presentados en la Tabla 5, muestran que el algoritmo SIFT es el que presenta un mayor tiempo de cómputo. Este comportamiento es esperable, ya que utiliza descriptores basados en números reales, que requieren mayor capacidad de procesamiento. En contraste, ORB es el algoritmo más eficiente en términos de tiempo, gracias al uso de descriptores binarios. AKAZE, que también emplea descriptores binarios, presenta un tiempo intermedio, aunque ligeramente superior al de ORB.

Por último, también se muestran dos *bounding boxes* en cada imagen: una verde, que corresponde a la detección utilizando los matches filtrados, y otra roja, que representa la detección con los matches sin filtrar. En general, observamos que la utilización de matches filtrados no produce grandes variaciones en las escenas seleccionadas. De hecho, en algunas imágenes ambas *bounding boxes* se solapan casi por completo.

	Media (ms)
ORB	52.4279
AKAZE	139.286
SIFT	143.478

Cuadro 5: Tiempo medio de ejecución de cada uno de los descriptores

En la Tabla 2 se puede apreciar que, en la escena 1 con el detector ORB, la bounding box verde se solapa con la roja, siendo esta última la más visible. En las demás escenas, el uso de matches filtrados ayuda a reducir el área delimitada, aunque siguen apareciendo ciertas deformaciones, especialmente con detectores como AKAZE y ORB.

3.1. Patch y Vídeo

En cuanto a la funcionalidad de superponer otra imagen sobre la región detectada, vemos que funciona correctamente. Teniendo los mismos problemas que con la bounding box (ver Figura 2).

Figura 2: Funcionalidad de Patch con el detector AKAZE.

Para insertar el vídeo, hemos utilizado un enfoque similar al de colocar el parche, insertando cada frame de forma individual en la región de interés mediante la misma función utilizada para las imágenes estáticas. Un frame del resultado se puede ver en la Figura 3.

Figura 3: Frame del vídeo incrustado en el cuadro con el detector SIFT.

4. Limitaciones

Durante el desarrollo de este trabajo se han detectado varias limitaciones tanto en el uso de los detectores como en el proceso de emparejamiento de características. Uno de los principales problemas observados es la sensibilidad de los detectores a cambios de iluminación, rotación o escala. Aunque detectores como SIFT y AKAZE están diseñados para ser invariantes a transformaciones, en la práctica, las detecciones pueden verse afectadas por condiciones no controladas en las escenas.

Otra limitación importante es la deformación de las bounding boxes obtenidas a partir de los matches. En algunos casos, estas regiones presentan una forma irregular o un área excesiva, lo que dificulta una interpretación precisa de la localización del objeto o escena detectada. Esto se hace especialmente notorio con detectores binarios como ORB y AKAZE.

Finalmente, cabe destacar que el rendimiento de cada detector también depende del contenido específico de cada escena. En escenas más homogéneas o con menos puntos clave relevantes, los detectores pueden fallar en encontrar coincidencias significativas, comprometiendo la calidad de la detección.

5. Conclusiones

Los resultados obtenidos en este trabajo permiten concluir que la selección del descriptor influye en el equilibrio entre la velocidad de ejecución y la calidad de la detección. En general, los detectores basados en descriptores binarios como ORB y AKAZE presentan tiempos de procesamiento mucho más reducidos, siendo especialmente útil en aplicaciones donde el rendimiento en tiempo real es prioritario. Sin embargo, esta ganancia en velocidad puede venir acompañada de una menor precisión en la detección y una mayor deformación en las transformaciones obtenidas.

Por otro lado, el uso de SIFT, aunque computacionalmente es más costoso, ofrece una detección más robusta y estable, especialmente en escenas con transformaciones más complejas o condiciones menos controladas.

Además, el filtrado de matches ha demostrado ser una herramienta útil para mejorar la fiabilidad de las detecciones, aunque su impacto no siempre es significativo. En algunas escenas, las bounding boxes resultantes con y sin filtrado son prácticamente idénticas, mientras que en otras, el filtrado ayuda a reducir el área y mejorar la localización.

Referencias

[1] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[2] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International Conference on Computer Vision*, pp. 2564–2571, 2011.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, pp. 91–110, 2004.

[4] P. F. Alcantarilla and T. Solutions, "Fast explicit diffusion for accelerated features in nonlinear scale spaces," *IEEE Trans. Patt. Anal. Mach. Intell*, vol. 34, no. 7, pp. 1281–1298, 2011.