Augmented Reality Project

Verifying Arguments

The first task was making sure the images we were using had been loaded correctly

```
// Check if the images are loaded
// *** TODO: COMPLETAR ***

// Change resolution of the image model to half
cv::resize(img_model, img_model, cv::Size(), 0.5, 0.5);

// Resize the patch to the size of the model
if (use_patch)
    cv::resize(img_patch, img_patch, img_model.size());
```

For this, we used the `empty()` method from `cv::Mat`, that returns `true` if and only if the array (our image, in this case) has any elements. We do all the required checks and report any errors accordingly:

```
// Check if the images are loaded
if (img_model.empty())
{
    cerr << "Error: failed to load model image from file: " << model_path;
    return -1;
}
if (use_patch && img_patch.empty())
{
    cerr << "Error: failed to load patch image from file: " << patch_path;
    return -1;
}
```

We also need to check that the specified `video2` or `index-cam` exists and can be open correctly

```
// If use video2, create the video capture
cv::VideoCapture cap2;
if (use_video2) {
        // Check if video2_path is a the webcam index and open it
        // *** TODO: COMPLETAR ***
    }
    else
        // Open video file to display
        // *** TODO: COMPLETAR ***

    if (!cap2.isOpened()) {
        cout << "Error: video not loaded: " + video2_path << endl;
        return -1;
    }
}
```

For this check, we define the variables `use_video2` and `use_webcam`:

```
// Second video argument is available?
string video2_path = parser.get<string>("video2");
string webcam_path = parser.get<string>("index-cam");

// Video2 has priority over patch
// Static video has priority over camera
bool use_video2 = !video2_path.empty();
bool use_webcam = !webcam_path.empty();
bool use_patch = !patch_path.empty() && !(use_video2 || use_webcam);
```

This allows us to know if the user is using the static video or the webcam. Now we need to check if we have access to the specified video source. We split the checks in two parts: one for the static video and another one for the webcam. In order to open a video from a path and capture it, we use `cv::VideoCapture::open("path/to/video/filename.format")`. `open` is only one of the functionalities that `cv::VideoCapture` offers, and we will be using some others in this project.

Once we open the video, `cv::VideoCapture` offers another method `cv::VideoCapture::isOpened` to check that the source was correctly opened.

Similarly, if we want to use a webcam as the video source, `cv::VideoCapture.open` offers an overload that can accepts the index (`int`) of the webcam that we want to use.

The code for the checks is very similar, differing only in the error messages we log:

```
VideoCapture video2_source;
// If you a static video as the video2 source...
if (use_video2)
{
    video2_source.open(video2_path);
    if (!video2_source.isOpened())
```

```
    {
        cerr << "Error: Failed to open video file: " << video2_path
             << ". Please check that the file exists and you have read privileges to
open it." << endl;
        return -1;
    }
}
// If you use the webcam as the video2 source...
else if (use_webcam)
{
    int webcam_idx = stoi(webcam_path);
    video2_source.open(webcam_idx);
    if (!vide2_source.isOpened())
    {
        cerr << "Failed to open the camera " << webcam_path
             << ". Please check that the webcam is properly connected." << endl;
        return -1;
    }
}
```

The original template did not make any differences between these cases. If we wanted to use `use_video2` to handle the `use_webcam` case, and subsequently the `video2_path` to handle the `webcam_path`, we would need to check if the `video2_path` argument was specified as a number and then interpret it as such. For this approach, we could use something like the following code:

```
// assuming, for example, that video2_path = "2";
try
{
    int webcam_idx = std::stoi(video2_path);
    std::cout << "The string is a number: " << webcam_idx << std::endl;
}
catch (const std::invalid_argument& e)
{
    std::cout << "The string is not a number: " << e.what() << std::endl;
}
```

However, we found that this approach was not as clear as the one we proposed.

Capturing `video2`

The next assignment reading the `video2_source` frame by frame:

```
// Warp the patch to the object using OpenCV
if (use_video2) {
    // If use video2, read the frame and resize it to the size of the patch
    // *** TODO: COMPLETAR ***
}

if (!img_patch.empty())
{
    rva_dibujaPatch(img_scene, img_patch, H, img_scene);
}
```

We modified the original template a little bit, refactoring the call to `rva_dibujaPatch()`. For this, we create a variable to store the patch indistinctly, whether the patch was specified as a static image, or it is the current frame of `video2` or the webcam, we will store this image/frame in `patch` so we call `rva_dibujaPatch` only once. We also define `cv::Mat patch` outside the main loop so we don't waste memory or time creating the variable on every iteration.

On top of that, if the user is not using a source for `video2` and will simply be using a static image for the patch, we do not need to assign it to `patch` in every iteration of the `while` loop. Because of this, we declare the `cv::Mat patch` just before starting the `while` loop and then, in each iteration, we check if `patch` needs to be overwritten with the current frame of the video or webcam. In case it does, we overwrite it.

Following this logic, if `use_video2` and `use_webcam` are both `false`, `patch` will contain the static patch image and this value will be assigned only once during the entire execution of the code. Otherwise, with the input stream `video2_source`, opened, the code reads the next frame from the stream and stores it in the variable `patch`. The `>>` operator is the input operator for streams, and it reads data from the stream and stores it in the variable on its right-hand side:

```
Mat patch;
if (!img_patch.empty())
{
    patch = img_patch;
}
while (cap.read(img_scene))
{
    // <template code...>

    // When using a video2_source, read the frame and resize it to the size of the
patch
    if (use_video2 || use_webcam)
    {
        // Read the next frame from the input stream
        // (resize it so that it fits in our model!)
        video2_source >> patch;
        cv::resize(patch, patch, img_scene.size());
    }
    if (!patch.empty())
    {
        // Warp the patch to the object using OpenCV
```

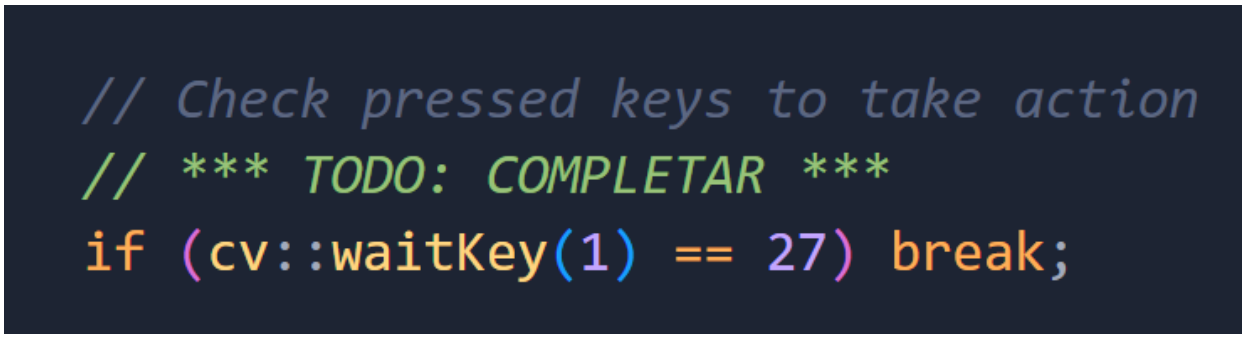
```
        rva_dibujaPatch(img_scene, patch, H, img_scene);
    }

    // <template code...>
}
```

Reacting to user input

The next task was making certain actions according to the user's input as follows:

- if the user presses `Q` or `Esc`, stop the execution.
- if the user presses `S`, take a screenshot.



In C++, `waitKey(1)` is typically used in Computer Vision applications to handle keyboard input. `waitKey(1)` is a function that waits for a specified number of milliseconds for a keyboard event to occur. In this case, we set the parameter to `1`, which means that we wait for 1 millisecond for a key press to happen. If a key is pressed during this time, `waitKey()` will return the ASCII code of the pressed key, so we only need to save it to a variable `onKeyPress` in order to take the appropriate action:

```
// Check for user input
int onKeyPress = waitKey(1);

// Exit the program if the user presses the 'q' or 'Esc' key
if (onKeyPress == 27 || onKeyPress == 'q')
{
    cout << "Execution terminated. Exiting..." << endl;
    break;
}
// Take a screenshot of the current scene if the user presses the 's' key
if (onKeyPress == 's')
{
    string filename = "../data/screenshots/screenshot_" +
to_string(++screenshots_cnt) + ".jpg";
    imwrite(filename, frame);
    cout << "Screenshot saved as " << filename << endl;
}
```

In order to take a screenshot and save it to our computer, we initialize a variable `int screenshots_cnt = 0` to keep count on the number of screenshots taken during the execution and save every on of them with a different name `screenshot_<screenshots_cnt>.jpg`. Every time the user presses `s`, we take a screenshot and save it using `cv:imwrite` after increasing `screenshots_cnt` by 1.

Implementing `rva_dibujaPatch`

`rva_dibujaPatch` takes three `cv::Mat` objects as inputs: `scene`, `patch`, and `H`, and a `cv::Mat` object `output` to hold the output of the process. `scene` represents the original image, or scene, on top of which we need to draw the `patch` image, and `H` is the homography matrix used to warp the `patch`.

The first thing we need to do is warping the patch into the scene, applying the homography transformation to `patch` using the `cv::warpPerspective()` function like we did in class.

We define a `getMask()` function to obtain a binary `cv::Mat` object that represents the mask of the warped patch, and we will overlay this mask into the scene, instead of the raw patch. The reason why this is necessary is that the process of warping `patch` using `H` results in a non-rectangular shape that does not fit perfectly into the destination image. In other words, the warped patch has some areas that fall outside of the boundaries of the destination image. To avoid drawing these areas onto the destination image, we need to create a mask that will indicate which pixels of the warped patch should be drawn onto the destination image and which pixels should be ignored.

```
Mat getMask(const Mat &warpedPatch)
{
    Mat mask, grayWarpedPatch;
    cvtColor(warpedPatch, grayWarpedPatch, COLOR_BGR2GRAY);
    threshold(grayWarpedPatch, mask, 0, 255, THRESH_BINARY);
    return mask;
}
```

To *"copy"* the contents of one image onto another one, we use the `cv::copyTo()` function instead of a simple assignment operator (=) because we want to selectively copy only the non-zero pixels of the warped patch onto the destination image.

By using `warpedPatch.copyTo(output, mask)` we are copying only the non-zero pixels of `warpedPatch` to output, as specified by the mask. This allows us to draw the warped image patch onto the destination image only where it overlaps with the destination image, while ignoring any pixels outside of the destination image boundaries. This will also remove any transparency effects between the scene and the patch, which was an error we faced during development.

The `rva_dibujaPatch` function is finally:

```
void rva_dibujaPatch(const Mat &scene, const Mat &patch, const Mat &H, Mat &output)
{
    auto size = scene.size();
    Mat warpedPatch;
    warpPerspective(patch, warpedPatch, H, size);

    Mat mask = getMask(warpedPatch);

    scene.copyTo(output);
    warpedPatch.copyTo(output, mask);
}
```

Here is an example obtained by this algorithm, which consists on a *PlayStation 4* game cover overlayed on the cover of *PlayStation 5* game box:

Passing different descriptors

The solution is able to use different descriptors, that use both binary and non-binary keypoints. The binary descriptors are listed in the provided code, they are:

- AKAZE
- BRISK
- ORB

The non-binary descriptors (descriptors that use non-binary keypoints) supported are:

- SIFT
- KAZE

With SIFT being the default option.

To pass a descriptor to the program, we use the `--desc` parameter and then specify the descriptor name with insensitive casing. If none is specified, SIFT will be used:

```
string str_to_descriptor_key(const string &input)
{
    if (input.empty())
        return "SIFT";
    auto descriptor = input;
    // case insensitive comparisons
    transform(descriptor.begin(), descriptor.end(), descriptor.begin(), ::toupper);

    if (descriptor == "AKAZE" || descriptor == "BRISK" || descriptor == "ORB" ||
descriptor == "KAZE" || descriptor == "SIFT")
        return descriptor;
    else
        return "";
}

// ...

// Get descriptor to use
auto desc_input = parser.get<string>("desc");
auto descriptor = str_to_descriptor_key(desc_input);
if (descriptor.empty())
{
    cerr << "Invalid descriptor string: " << desc_input << ". Accepted values are:
sift, akaze, kaze, orb and brisk (case insensitive)" << endl;
    return -1;
}
```

To perform the case-insensitive comparisons, we compare the uppercase input to a set of predefined keys, that will be used later on too. Of course, if an invalid descriptor is specified, we log the error message listing all supported descriptors to the user.

To calculate the keypoints, we modified the implementation seen in class to account for different descriptors as follows:

```
void rva_calculaKPsDesc(const Mat &img, vector<KeyPoint> &keypoints, Mat
&descriptors, const string &descriptorKey)
{
    auto descriptor_obj = get_descriptor(descriptorKey);

    // same as implementation seen in classes
    descriptor_obj->detectAndCompute(img, noArray(), keypoints, descriptors);
    Mat img_keypoints;
}
```

And implemented the method `get_descriptor(string descriptorKey)` of course. This method is very simple, it defines a dictionary with the supported keys and returns the appropriate OpenCV implementation, instead of hard-coding the implementation like we did in class:

```
Ptr<Feature2D> get_descriptor(const string &key)
{
    map<string, function<Ptr<Feature2D>()>> descriptor_map =
        {
            {"AKAZE", []()
             { return AKAZE::create(); }},
            {"BRISK", []()
             { return BRISK::create(); }},
            {"ORB", []()
             { return ORB::create(); }},
            {"KAZE", []()
             { return KAZE::create(); }},
            {"SIFT", []()
             { return SIFT::create(); }}};

    auto it = descriptor_map.find(key);
    return it->second();
}
```

Similarly, of course, we had to redefine our matcher since different descriptors will use different distances (for example, we cannot use a FLANN-based matcher for binary descriptors and keypoints). The method `rva_matchDesc` was modified accordingly and `get_matcher` was defined very similarly to `get_descriptor`:

```
Ptr<DescriptorMatcher> get_matcher(const string &key)
{
    map<string, function<Ptr<DescriptorMatcher>()>> matcher_map =
        {
            {"AKAZE", []()
             { return
DescriptorMatcher::create(DescriptorMatcher::BRUTEFORCE_HAMMING); }},
            {"BRISK", []()
             { return
DescriptorMatcher::create(DescriptorMatcher::BRUTEFORCE_HAMMING); }},
            {"ORB", []()
```

```
            { return
DescriptorMatcher::create(DescriptorMatcher::BRUTEFORCE_HAMMING); }},
           {"KAZE", []()
            { return DescriptorMatcher::create(DescriptorMatcher::FLANNBASED); }},
           {"SIFT", []()
            { return DescriptorMatcher::create(DescriptorMatcher::FLANNBASED); }}};
    auto it = matcher_map.find(key);
    return it->second();
}

void rva_matchDesc(const Mat &descriptors1, const Mat &descriptors2, vector<DMatch>
&matches, const string &descriptorKey)
{
    auto matcher = get_matcher(descriptorKey);

    // same as implementation seen in classes:
    vector<vector<DMatch>> knn_matches;
    matcher->knnMatch(descriptors1, descriptors2, knn_matches, 2);
    //-- Filter matches using the Lowe's ratio test
    const float ratio_thresh = 0.75f;
    for (size_t i = 0; i < knn_matches.size(); i++)
        if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance)
            matches.push_back(knn_matches[i][0]);
}
```

After executing tests with different video samples that used different camera angles, we arrived at some observations on how these detectors compared:

- SIFT was the second best in terms of accuracy for eliminating false-matches of keypoints. The result was very consistent, the patch image was almost at all times overlapping the scene correctly.
- BRISK and AKAZE performed similarly to SIFT in terms of accuracy, but failed to detect or eliminate a noticeable amount of false matches, which resulted on the edges being detected incorrectly sometimes. However, the results were acceptable and AKAZE performed faster than both BRISK and SIFT, while obtaining similar results for most camera angles.
- ORB performed the best in terms of speed among the feature detectors we tested, but it also was the one that failed to detect the most false matches. This resulted in a noticeable unstable result, specially when the camera was rotating, but the speed increase was more than double when compared to SIFT.
- KAZE performed the best in terms of accuracy. During our tests with multiple videos and angles, produced almost no false matches, resulting in an almost perfect overlap of the patch image on the scene for most camera angles. However, it performed extremely slow when compared to the other detectors, being more than 5 times slower than SIFT for some tests.

SIFT offered the best balance between accuracy and speed, performing better than BRISK, AKAZE and ORB for most scenarios and being faster than KAZE and BRISK, and almost as fast as AKAZE. For this reason, we coded it as the default descriptor option when none is specified.

Nonetheless, it is important to note that the *"fastest"*, *"most accurate"* or *"best performing"* descriptor always depends on the specific application and hardware being used, as well as the quality and angles of images and videos themselves and the requirements of the problem. These conclusions are based on the experimentation for this scenario only.

Saving the execution result into an output video

An optional task was saving the video of the execution once it finishes. To achieve this, there are two approaches:

- Create the output video and then save every frame as we generate them.
- Collect all the frames generated during execution and then save them before exiting the program.

We will show how to implement the second one since the first one is a subproblem of the second one, so it will be a little more interesting.

First, we declare the following variable just before the main loop:

```
// For each video frame, detect the object and overlay the patch
std:vector<cv::Mat> frames;
// ...
while (cap.read(img_scene))
{
    // ...
}
```

`frames` will store all of the frames we generate during execution so we can then write them to a video file. To add a frame to our frames collection:

```
// For each video frame, detect the object and overlay the patch
std:vector<cv::Mat> frames;
// ...
while (cap.read(img_scene))
{
    // ...

    // Save each frame wot collect them all
    // We will write them later to an output video using a VideoWriter object
    Mat frame = img_scene.clone();
    // Save them with full/original scale
    cv::resize(frame, frame, img_model.size(), 1, 1);
    frames.push_back(frame);
}
```

This code saves each frame generated by overlaying the patch into the video stream by pushing a copy of the current frame to `frames`. It also resizes the current frame to the size of `img_model` before saving it. The purpose of resizing the frame is to make sure all frames have the same size as the original `img_model`.

The only step left is to save the frames into the video object:

```
const string codec = "MJPG";                    // Or "mp4v" for .mp4 files
const string outputFile = "../data/output.avi"; // Or "output.mp4" for .mp4 files
const int frameRate = 30;
// Initialize VideoWriter
VideoWriter videoWriter;
videoWriter.open(outputFile,
                 VideoWriter::fourcc(codec[0], codec[1], codec[2], codec[3]),
frameRate,
                 frames[0].size(),
                 true);
// Check if creating the video works
if (!videoWriter.isOpened())
{
    cerr << "Could not open the output video file for writing" << endl;
    return -1;
}

// Write all frames to the video file
for (const auto &frame : frames)
{
    videoWriter.write(frame);
}
cout << "Video saved to " << outputFile << endl;

// Release the video writer to close the output video file
videoWriter.release();
```

To save the video, we will use a `cv::VideoWriter` that we initialize using the following parameters:

- `outputFile`: the name of the output file to be written.
  - It is always `output.avi` for this example.
- `fourcc`: the four-character code of the *codec* to be used for encoding the video.
  - This is passed as a parameter to `fourcc()`, which returns the corresponding integer code.
  - This code is then passed to `VideoWriter::open()`.
  - We are using the *codec* `MJPG`, which allows us to save videos in the `AVI` format, but we also tested this code with `MP4V` to save `MP4` videos for example.
- `frameRate`: the number of frames per second to be written to the output video.
- `frames[0].size()`: the size of the frames to be written, which is set to the size of the first frame in the frames vector.
  - Note that we made sure all of the frames had the same size.
- `true` : a flag indicating whether the output video file should be opened in write mode.
  - We open it in write/overwrite mode.

We must then check that `videoWriter` has been created successfully, and if not, log the corresponding error message.

Finally, we write all the frames saved in the `frames` vector to the output video file using the `cv::VideoWriter.write()` function and dispose the output video file.