Augmented Sword

Virtual and Augmented Reality

Group 7:
Alberto Cunha (up201906325@edu.fe.up.pt)
Joana Mesquita (up201907878@edu.fe.up.pt)
Joaquim Monteiro (up201905257@edu.fe.up.pt)

Description

Calibration

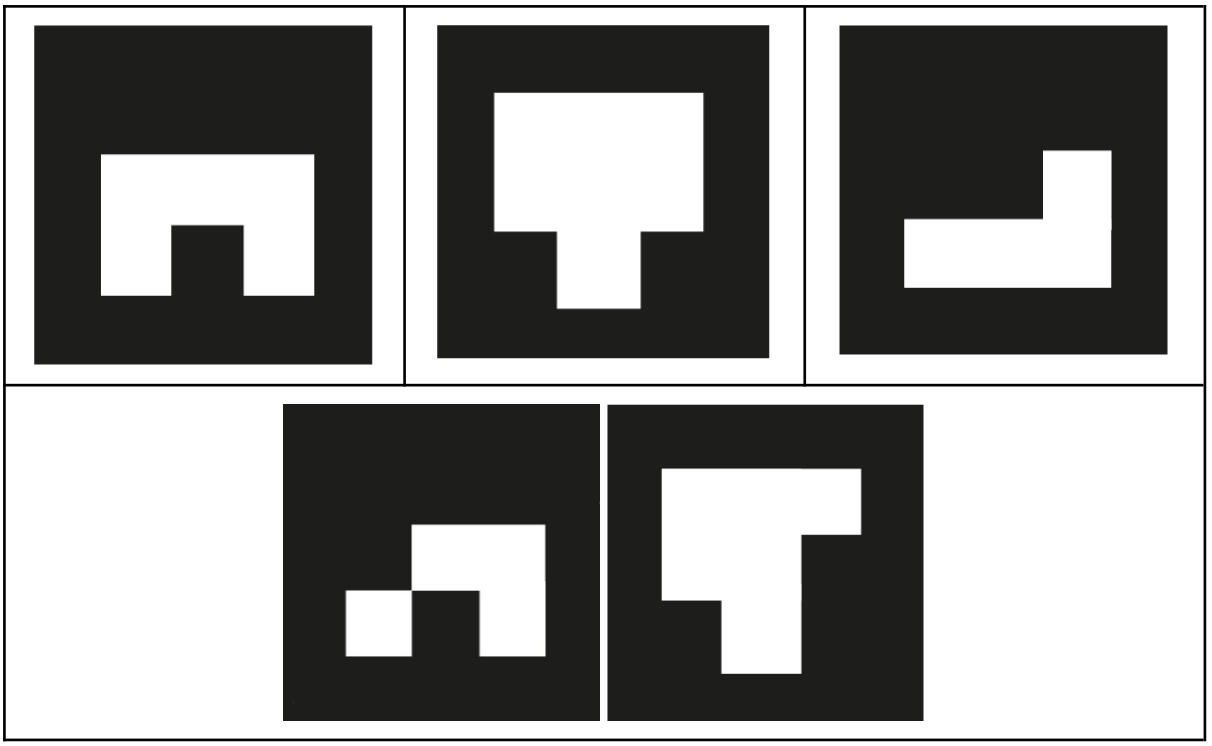
When the program is first executed, it tries to calibrate the camera using a checkerboard pattern. After this is completed, the calibration results are saved to disk, so this only has to be done once (per machine).

The calibration process consists of checking each camera frame for a checkerboard pattern. If the pattern is found, the frame is saved and the calibration process is suspended for a few seconds to allow the user to reposition the checkerboard image (so that there's enough variance for a good result).

After enough frames are collected, they're handed to the cv2.calibrateCamera function, which calculates the camera matrix, distortion coefficients, and the rotation and translation vectors.

Markers

For the assignment, we used the following markers:

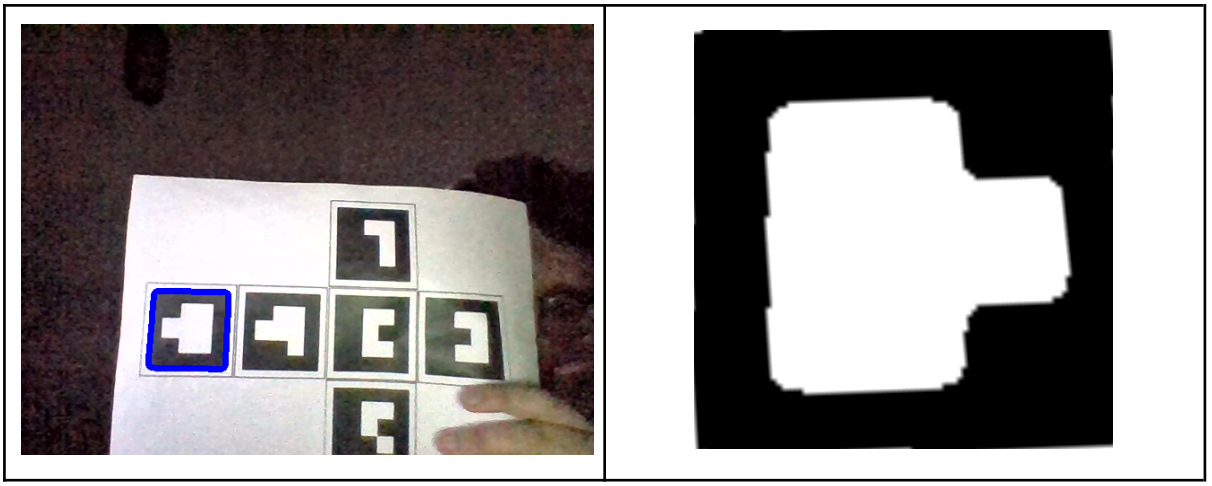


With these markers, we then built two marker swords, one in the shape of a cube with 5 markers and another flat with 2 markers as shown below.

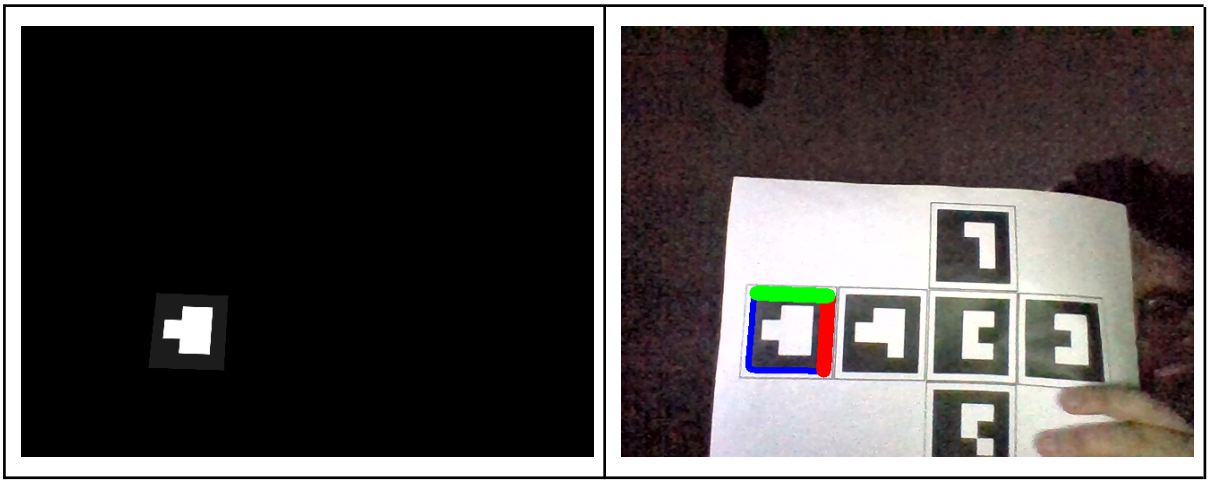
Marker Recognition

We capture real-time video using cv2.VideoCapture() to get the frames of the webcam. We then modify it, converting it to grayscale, blurring, and thresholding the image.

We would then find the contours of the marker. Then, we used a function to filter the contours found. The function would look for markers that had a square-like shape and return the contour with the greatest area. To find the squares, it would look for contours with an area inside a specific interval, contours that had 4 points and whose sides had similar lengths. It would also warp the perspective of the marker to get a frontal view of it.



The warped perspective would then be used in the next step. Template matching was used to identify which marker we found. To improve the performance, we used an XOR filter. We computed the result of an XOR operation between the images of the markers and the warped perspective of the recognized marker, then summed the result and if the percentage of XOR was below a certain threshold it would not compute the template matching. The warped marker would be rotated and matched with the markers. The function would return the best marker match, the rotation of the maker, and the result.



Reality Augmentation

Having completed the marker recognition, we proceeded to do the sword augmentation.

First, we created the sword by dividing it into three parts (the hilt, the base, and the blade) and defining all their points in arrays, saving them as global variables. After that, we created the functions that will allow us to draw pyramids or boxes from arrays of points using the openCV methods “cv2.draw_contours” and “cv2.line”.

In order to draw the sword on top of the markers, its points need to be transformed to align with the angle of the marker in relation to the camera. As such, we have to calculate the projected points from the marker.

To do this, we first defined the object point coordinates for the markers, taking into account their rotation. These points, along with the marker's image points and the camera's matrix and distortion coefficient (calculated during calibration), were used to obtain the rotation and translation vectors of the marker through the cv2.solvePnP function.

Afterward, we calculated the sword's rotation for each marker, based on what side of the sword they represented. Then, to place it in its proper position, we had to calculate the translation necessary to position it at the center of the cube/flat surface. For this, we calculated the center of the object points of the marker and subtracted a value from z (-1.5 for the cube marker and -0.5 for the flat one) so that the center would better match that of the marker sword. After we subtracted this array by the center of the sword parts after applying the rotation and then made sure that the rotation in y was 0 since we wanted to maintain the height of all the parts of the sword to keep them on top of each other. For the marker on top of the cube, since the sword was rotated in Z in a different way from the other markers, we had to also rotate the object point vector to match.

After all the transformations are applied, each part of the sword is then drawn in the proper coordinates.

Others

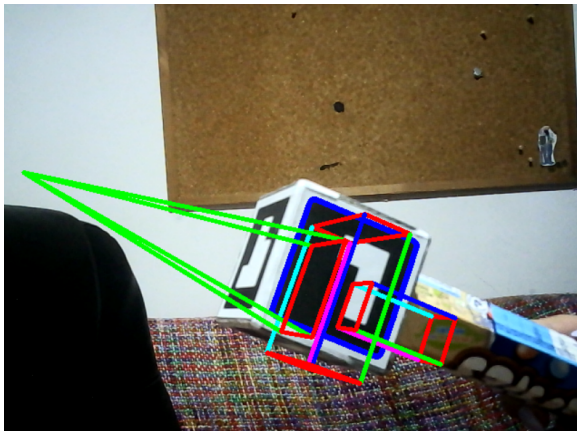
We experimented with using OpenGL to draw a more complicated sword model. While we were able to overlay the 3D graphics on the camera feed, we ran into issues while trying to make the sword match the marker's movement, so we removed it. Its code can be found in `renderer.py` and the commented out parts of `__init__.py`.

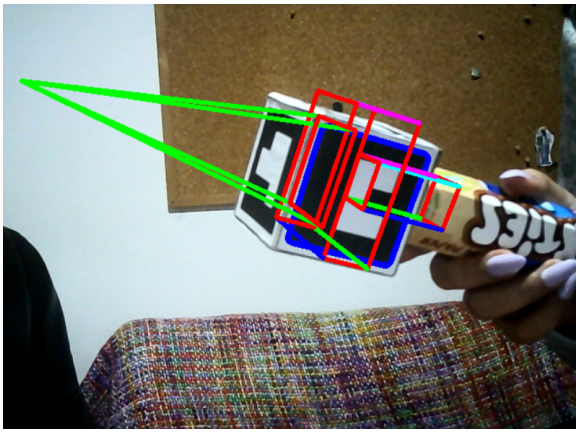
Results

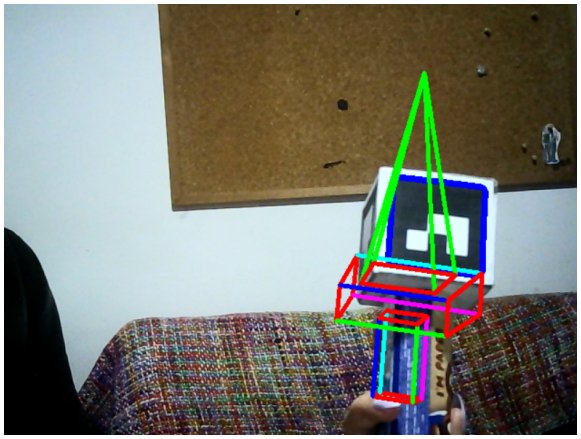
We were able to achieve all the objectives of the assignment using the methods described above and obtained the results seen below.

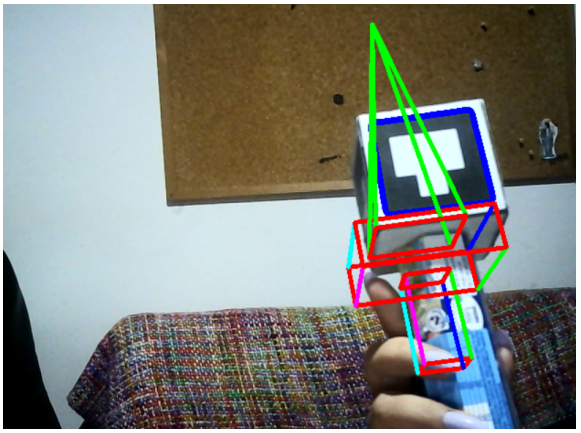
Results for flat sword marker

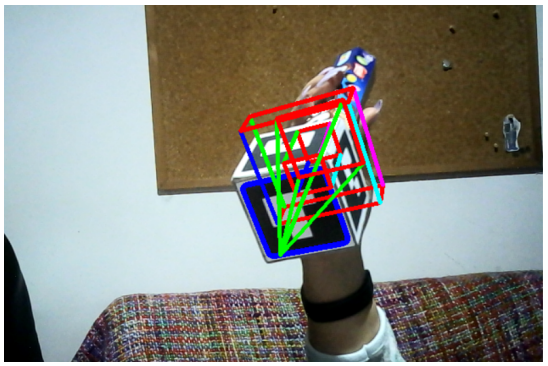
Results for cube sword marker











Annex

User's Manual

To run the program, Python >=3.10 and OpenCV 4 are required.

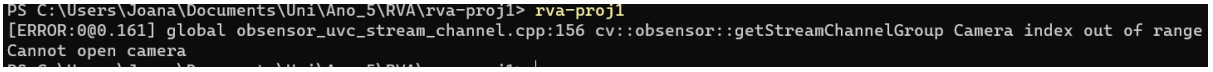
If these are already installed, the program can be executed by running `python -m rva-proj1` from the `src` folder.

Alternatively, the program can be installed with pip (which will install the necessary dependencies):

```
python -m venv venv
source venv/bin/activate  # venv/Scripts/Activate.ps1 if on Windows
pip install .
```

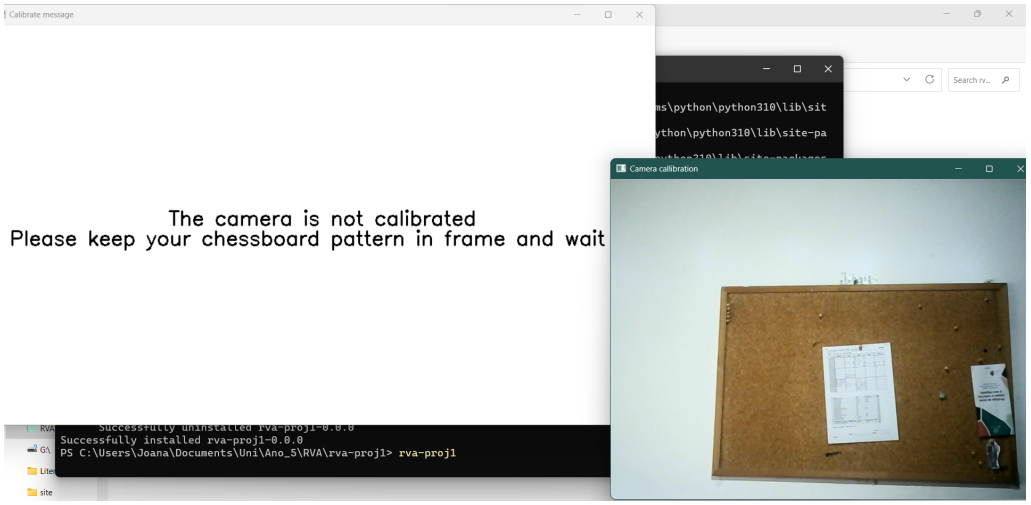
After this, the program can be executed by running `rva-proj1`.

Note: If running the program brings up a "Cannot Open Camera" error as seen below, verify that you have a camera connected to the computer.



Calibration

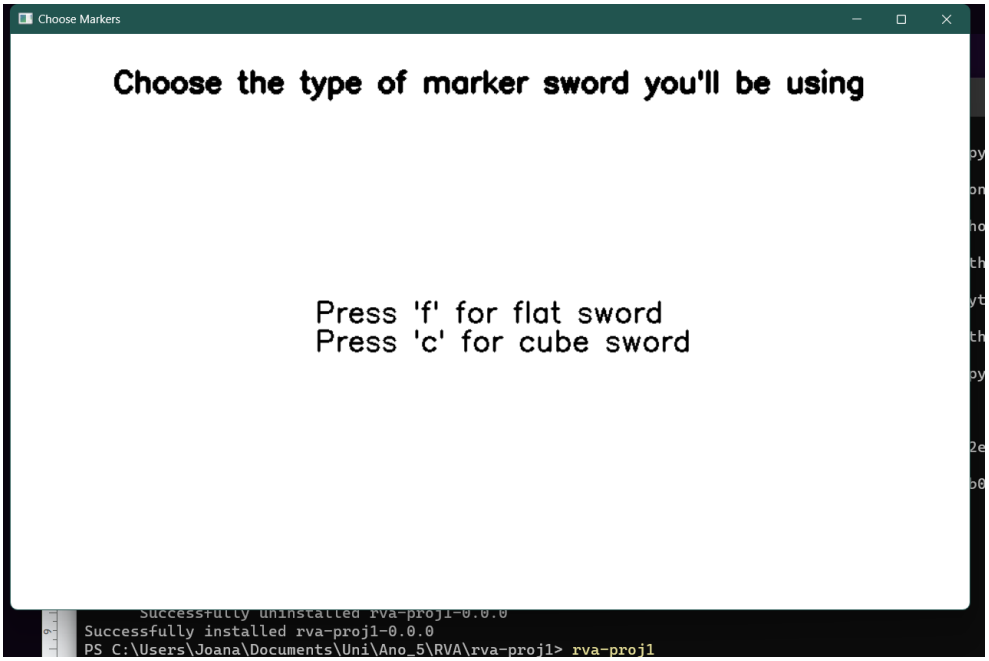
Once the application is running, it'll check if the user has already calibrated their camera. If not, it'll open the following windows to allow the user to perform the calibration otherwise it will skip calibration using the data stored instead



To perform the calibration, simply hold a chessboard pattern in front of the camera until the program finishes calibrating and closes these windows.

Choosing Markers

After the calibration is finished, the following window will appear asking which of the two types of marker swords is going to be used as seen below.

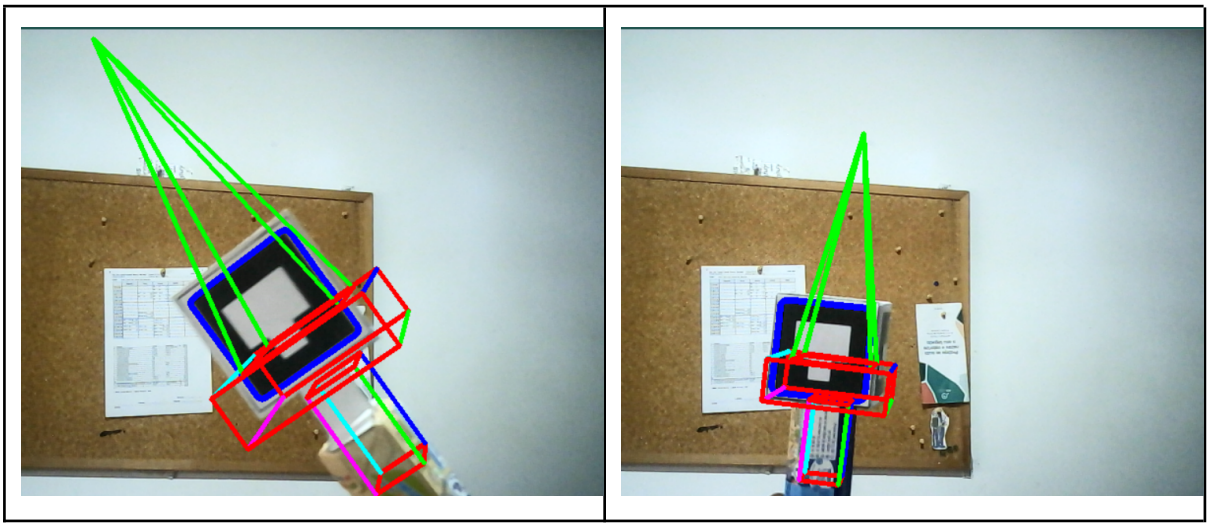


Pressing f will start the program for the flat marker sword and pressing c will start the program for the cube marker sword.

This window can also be closed to close the application by hitting the X in the corner or pressing Q on the keyboard.

Reality Augmentation

After the marker is chosen, the user can move it around and see the 3D sword on top of it as seen below.



Code

Note: these Python files should be placed in a `src/rva-proj1/` folder relative to the `pyproject.toml` file.

__main__.py

```
from . import main

main()
```

init.py

```
# from threading import Thread
# import time

import cv2
# import numpy

from .ui import Quit, choose_marker_type
from .calibration import callibrate_camera_with_cache
from .marker import GAUSSIAN_KERNEL_SIZE, define_obj_pts,
filter_contours, find_best_match, find_homography
# from .renderer import Renderer
from .sword import draw_sword_cube, draw_sword_flat


def main():
    # Initialize marker type as cube
    useFlatSword = False

    # Initialize camera
    cap = cv2.VideoCapture(0)

    # Check if camera opened successfully
    if not cap.isOpened():
        print("Cannot open camera")
        return

    # Calibrate camera
    _, cal_mtx, cal_dist, _, _ = callibrate_camera_with_cache(cap)
```

```
    try:
        # Choose marker type
        useFlatSword = choose_marker_type()
    except Quit as e:  # If user pressed 'q' key or closed the window
        print(e)  # Print exception message
        return  # Exit program

    # Initialize the OpenGL renderer in a separate thread
    # Renderer.window_size = size
    # render_thread = Thread(target=Renderer.run, name="render",
daemon=True)
    # render_thread.start()
    #
    # while True:
    #     try:
    #         window = moderngl_window.window()
    #         if window.config is None:
    #             time.sleep(0.1)
    #             continue
    #         break
    #     except ValueError:
    #         time.sleep(0.1)

    # Read until camera is closed
    while True:
        # if window.is_closing:
        #     break

        # Capture frame-by-frame
        ret, frame = cap.read()
        # If the frame is not read correctly, break the loop
        if not ret:
            print("Can't receive frame (stream end?). Exiting...")
            break

        # Send the frame to the OpenGL renderer
        # rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        # flipped_rgb_frame = numpy.flip(rgb_frame, 0).copy(order='C')
        # window.config.update_frame(flipped_rgb_frame)

        # Operations on the frame
        # Convert to grayscale
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
        # Blur image to remove noise
        gaussian_frame = cv2.GaussianBlur(gray_frame,
(GAUSSIAN_KERNEL_SIZE, GAUSSIAN_KERNEL_SIZE), 0)
        # Threshold image
        ret, threshold_frame = cv2.threshold(gaussian_frame, 0, 255,
cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        # Find contours
        contours, _ = cv2.findContours(threshold_frame, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        # Filter contours
        output = filter_contours(contours, threshold_frame)

        # found_homography = False

        # Draw contours and sword
        for contour, _, box in output:
            # Draw contours
            cv2.drawContours(frame, [contour], -1, (255, 0, 0), 5)
            # Find the best match for the marker found
            best_marker, rotation, _ = find_best_match(output)

            # If a marker was found
            if best_marker is not None:
                # Send the homography data to the OpenGL renderer
                # h, status = find_homography(box, rotation)
                # window.config.update_homography(h)
                # found_homography = True

                # Set object points for the marker's rotation
                objtp = define_obj_pts(rotation)

                # Convert both image points and object points to
float32
                objtp = objtp.astype('float32')
                imgp = box.astype('float32')

                # Solve PnP to get rotation and translation vectors
                _, rvecs, tvecs = cv2.solvePnP(objtp, imgp, cal_mtx,
cal_dist)

                # Draw sword depending on the marker type
                if useFlatSword:
```

```
                    draw_sword_flat(objtp, cal_mtx, cal_dist, rvecs,
tvecs, frame, best_marker)
                else:
                    draw_sword_cube(objtp, cal_mtx, cal_dist, rvecs,
tvecs, frame, best_marker)

        # if not found_homography:
        #     window.config.update_homography(None)

        # Display the resulting frame
        cv2.imshow("webcam", frame)

        # Press Q on keyboard to exit
        if cv2.waitKey(1) == ord('q'):
            break
        # Close window when user clicks X
        if cv2.getWindowProperty("webcam", 0) == -1:
            break

    # if not window.is_closing:
    #     window.close()

    # When everything done, release the capture
    cap.release()
    # Close all windows
    cv2.destroyAllWindows()
```

marker.py

```
from pathlib import Path

import cv2 as cv
import numpy as np

GAUSSIAN_KERNEL_SIZE = 13
MAX_AREA_PERCENTAGE = 0.4
MIN_AREA_PERCENTAGE = 0.01
APPROXIMATION_ACCURACY = 0.05
MAX_WIDTH_RATIO = 0.2
MAX_HEIGHT_RATIO = 0.2
MAX_HEIGHT_WIDTH_RATIO = 1.25
```

```
MAX_XOR_PERCENTAGE = 40

#open sword markers
MARKER_DIR = Path(__file__).parent.joinpath("markers")
FRONT_SWORD_MARKER =
cv.imread(str(MARKER_DIR.joinpath("front_sword.png")),
cv.IMREAD_GRAYSCALE)
BACK_SWORD_MARKER =
cv.imread(str(MARKER_DIR.joinpath("back_sword.png")),
cv.IMREAD_GRAYSCALE)
LEFT_SWORD_MARKER =
cv.imread(str(MARKER_DIR.joinpath("left_sword.png")),
cv.IMREAD_GRAYSCALE)
RIGHT_SWORD_MARKER =
cv.imread(str(MARKER_DIR.joinpath("right_sword.png")),
cv.IMREAD_GRAYSCALE)
TOP_SWORD_MARKER = cv.imread(str(MARKER_DIR.joinpath("top_sword.png")),
cv.IMREAD_GRAYSCALE)

# sword markers size
MARKER_SIZE = len(FRONT_SWORD_MARKER)

# sword makers indexes
FRONT_SWORD = 0
BACK_SWORD = 1
LEFT_SWORD = 2
RIGHT_SWORD = 3
TOP_SWORD = 4

# create list with all the sword markers
MARKER_LIST = [FRONT_SWORD_MARKER, BACK_SWORD_MARKER,
LEFT_SWORD_MARKER, RIGHT_SWORD_MARKER, TOP_SWORD_MARKER]

ROTATION_90 = 0
ROTATION_180 = 1
ROTATION_270 = 2
ROTATION_0 = 3

# filter the contours found in a frame to return the contour of the
marker and its warped perspective
def filter_contours(contours, threshold):
    valid_contour = []
    biggest_area = 0
```

```
    current_box = []
    return_contour = []

    # calculates max and min area acceptable
    img_area = threshold.shape[0] * threshold.shape[1]
    max_area = MAX_AREA_PERCENTAGE * img_area
    min_area = MIN_AREA_PERCENTAGE * img_area

    for contour in contours:
        # gets area of contour
        area = cv.contourArea(contour)

        # filters per contour area
        if area > max_area or area < min_area:
            continue

        # get the approximate shape of the contour
        epsilon = APPROXIMATION_ACCURACY * cv.arcLength(contour, True)
        approx = cv.approxPolyDP(contour, epsilon, True)

        # only accepts shapes with 4 sides
        if len(approx) != 4:
            continue

        # gets box points
        box = np.squeeze(approx).astype(np.float32)
        p0, p1, p2, p3 = box[2], box[1], box[0], box[3]

        # gets sides lengths
        width_03 = np.sqrt(((p0[0] - p3[0]) ** 2) + ((p0[1] - p3[1]) **
2))
        width_12 = np.sqrt(((p1[0] - p2[0]) ** 2) + ((p1[1] - p2[1]) **
2))

        height_01 = np.sqrt(((p0[0] - p1[0]) ** 2) + ((p0[1] - p1[1])
** 2))
        height_23 = np.sqrt(((p2[0] - p3[0]) ** 2) + ((p2[1] - p3[1])
** 2))

        max_width = max(int(width_03), int(width_12))
        max_height = max(int(height_01), int(height_23))

        # evaluates if all sides are about the same size
```

```
        if (
            abs(width_03 - width_12) > MAX_WIDTH_RATIO * max_width
            or abs(height_01 - height_23) > MAX_HEIGHT_RATIO *
max_height
            or abs(max_height - max_width) > MAX_HEIGHT_WIDTH_RATIO *
min(max_height, max_width)
        ):
            continue

        # records current biggest square-like contour
        if area > biggest_area:
            biggest_area = area
            current_box = box
            valid_contour = contour

    # if there are valid contours, compute the warped perspective of
the marker, if not, return empty list
    if len(valid_contour) != 0:
        output_height = MARKER_SIZE
        output_width = MARKER_SIZE

        # get the points to warp the perspective of the marker
        input_pts = np.float32([current_box[2], current_box[1],
current_box[0], current_box[3]])
        output_pts = np.float32(
            [
                [0, 0],
                [0, output_height - 1],
                [output_width - 1, output_height - 1],
                [output_width - 1, 0]
            ]
        )

        # get warped marker
        transformation_matrix = cv.getPerspectiveTransform(input_pts,
output_pts)
        warped_marker_image = cv.warpPerspective(
            threshold, transformation_matrix,
            (output_width, output_height),
            flags=cv.INTER_LINEAR
        )

        # append values to return
```

```
        return_contour.append((valid_contour, warped_marker_image,
current_box))

    return return_contour

# Find the contour with the best match to the template
def find_best_match(output, use_xor_filter=True):
    min_result = 99999999
    best_marker, rotation = None, None

    # gets warped perspective of the marker
    _, marker_image, _ = output[0]

    # goes through every marker to see which one is the best match
    for m in range(len(MARKER_LIST)):
        # to identify the rotation of the marker
        for i in range(4):
            # rotates marker 90 degrees every loop
            marker_image = cv.rotate(marker_image,
cv.ROTATE_90_CLOCKWISE)

            # uses this filter to better the performance
            if use_xor_filter:
                # if the percentage of the 2 images that are the same
is below a certain value it does not do template matching
                xor = cv.bitwise_xor(marker_image,
MARKER_LIST[m]).sum()
                xor_percentage = xor / (marker_image.shape[0] *
marker_image.shape[1])
                if xor_percentage > MAX_XOR_PERCENTAGE:
                    continue

            # gets result of template matching
            result = cv.matchTemplate(
                marker_image, MARKER_LIST[m], cv.TM_SQDIFF_NORMED
            )

            # saves current best result
            if result < min_result:
                min_result = result
                best_marker = m
                rotation = i
```

```
    # returns the best_marker and the rotation of the marker in the
image
    return best_marker, rotation, min_result

# finds the homography between the template marker and the marker in
the box
def find_homography(box, rotation):
    marker_height = MARKER_SIZE
    marker_width = MARKER_SIZE

    # gets marker points
    marker_pts = np.float32(
            [
                [0, 0],
                [0, marker_height - 1],
                [marker_width - 1,  marker_height - 1],
                [marker_width - 1, 0]
            ]
        ).reshape(-1,1,2)

    # changes box points dependent on the marker's rotation and
computes the homography
    if rotation == ROTATION_90:
        box_pts = np.float32([box[1], box[0], box[3],
box[2]]).reshape(-1,1,2)
        return cv.findHomography(marker_pts, box_pts)
    if rotation == ROTATION_180:
        box_pts = np.float32([box[0], box[3], box[2],
box[1]]).reshape(-1,1,2)
        return cv.findHomography(marker_pts, box_pts)
    if rotation == ROTATION_270:
        box_pts = np.float32([box[3], box[2], box[1],
box[0]]).reshape(-1,1,2)
        return cv.findHomography(marker_pts, box_pts)
    if rotation == ROTATION_0:
        box_pts = np.float32([box[2], box[1], box[0],
box[3]]).reshape(-1,1,2)
        return cv.findHomography(marker_pts, box_pts)

# Define object points for each marker rotation
def define_obj_pts(rotation):
    if rotation == ROTATION_90:
        return np.array([[0,0,0],[0,1,0],[1,1,0],[1,0,0]])
```

```
    if rotation == ROTATION_180:
        return np.array([[0,1,0],[1,1,0],[1,0,0],[0,0,0]])
    if rotation == ROTATION_270:
        return np.array([[1,1,0],[1,0,0],[0,0,0],[0,1,0]])
    if rotation == ROTATION_0:
        return np.array([[1,0,0],[0,0,0],[0,1,0],[1,1,0]])
```

sword.py

```
import cv2
import numpy

from .marker import BACK_SWORD, FRONT_SWORD, LEFT_SWORD, RIGHT_SWORD,
TOP_SWORD

# sword base points
BASE_SWORD_POINTS = numpy.float32([[-2.25,-0.25,-0.25],
[-2.25,0.25,-0.25], [-2.25,0.25,0.25], [-2.25,-0.25,0.25],
[-1.25,-0.25,-0.25], [-1.25,0.25,-0.25], [-1.25,0.25,0.25],
[-1.25,-0.25,0.25]]).reshape(-1,3)
# sword hilt points
HILT_SWORD_POINTS = numpy.float32([[-0.75,-0.75,-0.75],
[-0.75,0.75,-0.75], [-1.25,0.75,-0.75], [-1.25,-0.75,-0.75],
[-0.75,-0.75,0.75], [-0.75,0.75,0.75], [-1.25,0.75,0.75],
[-1.25,-0.75,0.75]]).reshape(-1,3)
# sword blade points
BLADE_SWORD_POINTS = numpy.float32([[2.25,0,0], [-0.75,0.5,0.5],
[-0.75,0.5,-0.5], [-0.75,-0.5,-0.5], [-0.75,-0.5,0.5]]).reshape(-1,3)

# sword components indexes
BASE_SWORD = 0
HILT_SWORD = 1
BLADE_SWORD = 2

# list with all the sword components
SWORD = [BASE_SWORD_POINTS,HILT_SWORD_POINTS,BLADE_SWORD_POINTS]

# draw parallelepiped for sword base and hilt
def drawBoxes(img, imgpts):

    imgpts = numpy.int32(imgpts).reshape(-1,2)
```

```
    # draw base
    img = cv2.drawContours(img, [imgpts[:4]],-1,(0,0,255),3)

    # draw pillars
    img = cv2.line(img, tuple(imgpts[0]), tuple(imgpts[4]),(255,0,0),3)
    img = cv2.line(img, tuple(imgpts[1]),
tuple(imgpts[5]),(255,255,0),3)
    img = cv2.line(img, tuple(imgpts[2]),
tuple(imgpts[6]),(255,0,255),3)
    img = cv2.line(img, tuple(imgpts[3]), tuple(imgpts[7]),(0,255,0),3)

    # draw top
    img = cv2.drawContours(img, [imgpts[4:]],-1,(0,0,255),3)
    return img

# draw pyramid for sword blade
def drawPyramid(img, imgpts):
    imgpts = numpy.int32(imgpts).reshape(-1,2)

    # draw highest point
    tip = tuple(imgpts[0])

    #draw other points
    for i in range(4):
        img = cv2.line(img, tip, tuple(imgpts[i + 1]), (0, 255, 0), 3)

    img = cv2.drawContours(img, [imgpts[1:]],-1,(0,0,255),3)
    return img


# get rotation matrix for each marker
def get_rotation(best_marker):
    # z rotation because our sword is tilted in the Z axis
    rotationMatrixZ = numpy.array([
        [numpy.cos(numpy.radians(-90)), -numpy.sin(numpy.radians(-90)),
0],
        [numpy.sin(numpy.radians(-90)), numpy.cos(numpy.radians(-90)),
0],
        [0, 0, 1]
        ], dtype=numpy.float32)

    # get rotation matrix for each marker
```

```
    if best_marker == FRONT_SWORD:
        rotationMatrix = numpy.array([
            [numpy.cos(0), 0, numpy.sin(0)],
            [0, 1, 0],
            [-numpy.sin(0), 0, numpy.cos(0)]
        ])
    if best_marker == BACK_SWORD:
        rotationMatrix = numpy.array([
            [numpy.cos(numpy.pi), 0, numpy.sin(numpy.pi)],
            [0, 1, 0],
            [-numpy.sin(numpy.pi), 0, numpy.cos(numpy.pi)]
        ])
    if best_marker == LEFT_SWORD:
        rotationMatrix = numpy.array([
            [numpy.cos(numpy.pi / 2), 0, numpy.sin(numpy.pi / 2)],
            [0, 1, 0],
            [-numpy.sin(numpy.pi / 2), 0, numpy.cos(numpy.pi / 2)]
        ])
    if best_marker == RIGHT_SWORD:
        rotationMatrix = numpy.array([
            [numpy.cos(-numpy.pi / 2), 0, numpy.sin(-numpy.pi / 2)],
            [0, 1, 0],
            [-numpy.sin(-numpy.pi / 2), 0, numpy.cos(-numpy.pi / 2)]
        ])
    if best_marker == TOP_SWORD:
        rotationMatrix = numpy.array([
            [1, 0, 0],
            [0, numpy.cos(numpy.pi / 2), -numpy.sin(numpy.pi / 2)],
            [0, numpy.sin(numpy.pi / 2), numpy.cos(numpy.pi / 2)]
        ])
        # this marker is on top and needs to rotate the other way
around
        rotationMatrixZ = numpy.array([
            [numpy.cos(numpy.radians(90)),
-numpy.sin(numpy.radians(90)), 0],
            [numpy.sin(numpy.radians(90)),
numpy.cos(numpy.radians(90)), 0],
            [0, 0, 1]
            ], dtype=numpy.float32)

    # combine rotation matrices
    rotationMatrix = numpy.dot(rotationMatrixZ, rotationMatrix)
    return rotationMatrix
```

```
# Calculate center of object points to align sword
def calculate_object_points_center(objtp, z_value,best_marker):
    # calculate center of object points
    objtp_center = numpy.mean(objtp, axis=0)
    # set z value so that sword is not glued to the marker
    # and instead the marker is in the middle of the sword
    objtp_center[2] = z_value

    # in the top marker since we rotate -90 degrees in Z we also need
to rotate the center points of the object
    if best_marker == TOP_SWORD:
        # rotation matrix for -90 degrees in Z
        rotation_matrix = numpy.array([[0, -1, 0],
                                       [1, 0, 0],
                                       [0, 0, 1]])
        # apply rotation
        objtp_center = numpy.dot(rotation_matrix, objtp_center)

    return objtp_center

# apply transformations for each section of the sword
def apply_transformations(objtp_center,rotationMatrix,mtx, dist, rvecs,
tvecs , frame, best_marker):

    for i in range(3):
        # get points for each section of the sword
        axisBoxes = SWORD[i] + numpy.array([1, 0.5, 0],
dtype=numpy.float32)

        # rotate sword
        axisBoxes = numpy.dot(axisBoxes, rotationMatrix)
        # get sword center
        axisBoxesCenter = numpy.mean(axisBoxes, axis=0)
        # calculate translation to align sword by subtracting the
center of the sword from the center of the object points
        translation = objtp_center - axisBoxesCenter

        if (best_marker == TOP_SWORD):
            # for top marker the axis rotate -90 degrees in the Z axis
so we have to stop the x axis translation instead
            translation[0] = 0
        else:
```

```
            # remove y axis translation because the sword is always on
the same height
            translation[1] = 0

        # apply translation
        axisBoxes += translation

        # project points
        imgpts, _ = cv2.projectPoints(axisBoxes, rvecs, tvecs, mtx,
dist)

        # draw sword
        if i == BLADE_SWORD:
            drawPyramid(frame, imgpts)
        else:
            drawBoxes(frame, imgpts)


# draw sword for flat configuration
def draw_sword_flat(objtp, mtx, dist, rvecs, tvecs , frame,
best_marker):

    # calculate center of object points to align sword
    objtp_center = calculate_object_points_center(objtp,
-0.5,best_marker)

    # calculate rotation matrix for each marker
    rotationMatrix = get_rotation(best_marker)

    # apply transformations and draw sword
    apply_transformations(objtp_center,rotationMatrix,mtx, dist, rvecs,
tvecs , frame, best_marker)


# draw sword for cube configuration
def draw_sword_cube(objtp, mtx, dist, rvecs, tvecs , frame,
best_marker):

    # calculate center of object points to align sword
    objtp_center = calculate_object_points_center(objtp, -1.5,
best_marker)

    # calculate rotation matrix for each marker
```

```
    rotationMatrix = get_rotation(best_marker)

    # apply transformations and draw sword
    apply_transformations(objtp_center,rotationMatrix,mtx, dist, rvecs,
tvecs , frame, best_marker)
```

ui.py

```
import numpy as np
import cv2

# Exception class for quitting the program
class Quit(Exception):
    pass

# Function for UI to choose the type of marker sword
def choose_marker_type():

    # Create a white image
    width, height = 1000, 600
    white_color = (255, 255, 255)
    white_image = np.full((height, width, 3), white_color,
dtype=np.uint8)

    # Define center of the image for text
    x = (width - cv2.getTextSize("Choose the type of marker sword
you'll be using", cv2.FONT_HERSHEY_SIMPLEX, 1,3)[0][0] )// 2
    y = height // 2

    # Write text
    cv2.putText(white_image, "Choose the type of marker sword you'll be
using", (x, 60), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0), 3)

    # Define center of the image for text
    x = (width - cv2.getTextSize("Press 'f' for flat sword",
cv2.FONT_HERSHEY_SIMPLEX, 1,2)[0][0] )// 2

    # Write promp text
    cv2.putText(white_image, "Press 'f' for flat sword", (x, y),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0), 2)
```

```
    cv2.putText(white_image, "Press 'c' for cube sword", (x, y + 30),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0), 2)

    # Show image
    cv2.imshow("Choose Markers", white_image)

    # Wait for user input
    while True:
        key = cv2.waitKey(1)

        # Check if user pressed 'f' or 'c'
        if key == ord('f'):
            cv2.destroyAllWindows() # Close window
            return True
        if key == ord('c'):
            cv2.destroyAllWindows() # Close window
            return False
        # Check if user pressed 'q' or closed the window
        if key == ord('q'):
            raise Quit("User pressed 'q' key or closed the window") #
Raise exception
        if cv2.getWindowProperty("Choose Markers",0) == -1:
            raise Quit("User pressed 'q' key or closed the window") #
Raise exception


def calibrate_camera():

    # Create a white image
    width, height = 1000, 600
    white_color = (255, 255, 255)
    white_image = np.full((height, width, 3), white_color,
dtype=np.uint8)

    # Define center of the image for text
    x1 = (width - cv2.getTextSize("The camera is not calibrated",
cv2.FONT_HERSHEY_SIMPLEX, 1,2)[0][0] )// 2
    x2 = (width - cv2.getTextSize("Please keep your chessboard pattern
in frame and wait, p", cv2.FONT_HERSHEY_SIMPLEX, 1,2)[0][0] )// 2
    y = height // 2

    # Write text
```

```
    cv2.putText(white_image, "The camera is not calibrated", (x1, y),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0), 2)
    cv2.putText(white_image, "Please keep your chessboard pattern in
frame and wait", (x2, y + 30), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0),
2)

    # Show image
    cv2.imshow("Calibrate message", white_image)
```

calibration.py

```
import time
from pathlib import Path

import numpy as np
import cv2 as cv

from .ui import calibrate_camera

CHESSBOARD_PATTERN = (6, 9)
SLEEP_INTERVAL = 2
TARGET_POINT_COUNT = 20

CRITERIA = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30,
0.001)


def callibrate_camera(cap: cv.VideoCapture):
    # prepare object points, like (0,0,0), (1,0,0), (2,0,0)
....,(6,5,0)
    objp = np.zeros((CHESSBOARD_PATTERN[0] * CHESSBOARD_PATTERN[1], 3),
np.float32)
    objp[:, :2] = np.mgrid[0:CHESSBOARD_PATTERN[0],
0:CHESSBOARD_PATTERN[1]].T.reshape(-1, 2)
    # Arrays to store object points and image points from all the
images.
    objpoints = []  # 3d point in real world space
    imgpoints = []  # 2d points in image plane.

    # Call UI message to explain how to calibrate the camera
    calibrate_camera()
```

```
    while len(objpoints) < TARGET_POINT_COUNT:
        ret, frame = cap.read()
        if not ret:
            raise Exception("Failed to calibrate camera: could not read
frame")

        gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
        # Find the chess board corners
        ret, corners = cv.findChessboardCorners(gray,
CHESSBOARD_PATTERN, None)
        # If found, add object points, image points (after refining
them)
        if ret:
            objpoints.append(objp)
            corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1,
-1), CRITERIA)
            imgpoints.append(corners2)
            # Draw and display the corners
            cv.drawChessboardCorners(frame, CHESSBOARD_PATTERN,
corners2, ret)

        cv.imshow('Camera callibration', frame)
        if cv.waitKey(1) == ord('q'):
            raise Exception("Failed to calibrate camera: canceled")

        if ret:
            time.sleep(SLEEP_INTERVAL)

    cv.destroyAllWindows()
    return cv.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],
None, None)


def callibrate_camera_with_cache(cap: cv.VideoCapture):
    path = Path.cwd().joinpath("camera_calibration.npz")
    if path.exists():
        print(f"Loading cached calibration values from '{path}'")
        data = np.load(path)
        ret, mtx, dist, rvecs, tvecs = data["ret"].item(), data["mtx"],
data["dist"], data["rvecs"], data["tvecs"]
    else:
        ret, mtx, dist, rvecs, tvecs = callibrate_camera(cap)
```

```
        np.savez(path, ret=ret, mtx=mtx, dist=dist, rvecs=rvecs,
tvecs=tvecs)

    return ret, mtx, dist, rvecs, tvecs
```

renderer.py

```
from pathlib import Path

import moderngl
from moderngl_window import geometry, WindowConfig
from moderngl_window.scene.camera import OrbitCamera
import numpy
from pyrr import Matrix44, Quaternion, Vector3


class Renderer(WindowConfig):
    gl_version = (3, 3)

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        # Load the sword
        self.scene =
self.load_scene(str(Path(__file__).parent.joinpath("models/minecraft_di
amond_sword.glb")))
        self.original_matrix = self.scene.matrix

        # Set up the camera
        # An OrbitCamera can be rotated with the mouse, it can be
switched to a regular Camera to have a fixed position
        self.camera = OrbitCamera(aspect_ratio=self.wnd.aspect_ratio,
radius=0.5, near=0.1, far=1000.0)

        # Prepare the geometry, texture and shaders to display the
camera feed
        self.quad = geometry.quad_2d(size=(2.0, 2.0))
        self.texture = self.ctx.texture(self.window_size, 3,
data=numpy.zeros((self.window_size[0], self.window_size[1], 3),
dtype=numpy.uint8))
        self.texture_program = self.ctx.program(
```

```
            vertex_shader="""
            #version 330
            in vec2 in_position;
            in vec2 in_texcoord_0;
            out vec2 uv;
            void main() {
                gl_Position = vec4(in_position, 0.9999999, 1.0);
                uv = in_texcoord_0;
            }
            """,
            fragment_shader="""
            #version 330
            uniform sampler2D tex;
            in vec2 uv;
            out vec4 f_color;
            void main() {
                f_color = texture(tex, uv);
            }
            """,
        )
        self.current_frame: numpy.ndarray | None = None
        self.frame_was_updated = False

        self.homography_adjustment = Matrix44.identity()
        self.homography_was_updated = False

    def render(self, time: float, frametime: float):
        self.ctx.enable_only(moderngl.DEPTH_TEST | moderngl.CULL_FACE)
        self.ctx.clear()

        # Display the camera frame as the background
        if self.current_frame is not None:
            if self.frame_was_updated:
                self.frame_was_updated = False
                self.texture.write(self.current_frame)

            self.texture.use()
            self.quad.render(self.texture_program)

        # Reposition the sword
        if self.homography_was_updated:
            self.scene.matrix = self.original_matrix *
self.homography_adjustment
```

```
        # Draw the sword
        self.scene.draw(self.camera.projection.matrix,
self.camera.matrix, time)

    def mouse_drag_event(self, x: int, y: int, dx, dy):
        self.camera.rot_state(dx, dy)

    def mouse_scroll_event(self, x_offset: float, y_offset: float):
        self.camera.zoom_state(y_offset)

    def resize(self, width: int, height: int):

self.camera.projection.update(aspect_ratio=self.wnd.aspect_ratio)

    def update_frame(self, frame: numpy.ndarray):
        self.current_frame = frame
        self.frame_was_updated = True

    def update_homography(self, homography: numpy.ndarray | None):
        if homography is not None:
            # TODO: self.homography_adjustment = ???
            ...
        else:
            # keep it visible if no homography found for now,
            # TODO switch to the bottom version to hide the sword if no
homography is found
            self.homography_adjustment = Matrix44.identity()
            # self.homography_adjustment =
Matrix44.from_scale(Vector3([0, 0, 0]))

        self.homography_was_updated = True
```

Others

pyproject.toml

```
[project]
name = "rva-proj1"
version = "0.0.0"
authors = [
    { name = "Alberto Cunha", email = "up201906325@edu.fe.up.pt" },
```

```
    { name = "Joana Mesquita", email = "up201907878@edu.fe.up.pt" },
    { name = "Joaquim Monteiro", email = "up201905257@edu.fe.up.pt" }
]
dependencies = [
    "moderngl>=5.8.2",
    "moderngl-window>=2.4.5",
    "opencv-python>=4.8.1.78",
]
requires-python = ">= 3.10"

[project.scripts]
rva-proj1 = 'rva_proj1:main'

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```