Exam Part 1

Texture mapping using linear transformations.

IAML 2020

The first part of the exam involves the use of geometric transformations in a simple computer vision application. The assignment consists of the following exercises:

1. **Transformation library:** You will create a library for handling general perspective transformations in two dimensions.
2. **Affine transformations:** You will implement code to find an affine transformation between sets of points.
3. **Texture mapping:** You will implement parts of a computer vision application that performs texture mapping using a triangle mesh. You will use your tranformation library to achieve this.

Hand-in

At the end of the course, you have to hand in the following elements:

- **Code:** You have to hand in the complete source code used for the assignments including any files provided by us. This makes it easy for us to test whether your implementations actually work. Upload the code as a **zip** archive (don't use rar).
- **Report:** You must combine your writings for each exam part into a single report document in **pdf** format. The reports for each part should be *approximately* 2-3 pages in length (normal pages of 2400 characters). Use clear formatting for exam parts and exercises. The structure of the report and content requirements is listed for each exercise.
- **Extra files:** Include all pictures, videos, and other files produced in a separate **zip** archive.

Each exam exercise includes a checklist with exact requirements for what must be included in the final hand-in. For the official exam assignment, you must group the individual parts and hand them in before Friday, May 22nd, at 23:59.

Feedback You can get preliminary feedback on your code and report for this part of the exam if you hand it in before Wednesday, March 11th, at 23:59. You may find the "**Part 1**" submission under the **Exam Assignments** section on LearnIT. We will not provide additional feedback after this deadline.

Requirements

We provide a number of minimum requirements to pass the exam. Use the list at the end of each exercise to check that you meet the requirements. To make the requirements as clear as possible, we mark tasks with one of the following symbols:

- The task is required.
- You have to implement at least one of the specified tasks.
- The task is entirely optional. You should only focus on these after you completed the other tasks.

Additionally, the symbol Ⓜ is added to master student only tasks, i.e. **bachelor students can ignore those tasks**.

You should also observe the following guidelines:

- **Be precise:** When explaining your approach, implementation, and results, be as precise as possible.
- **Describe the method:** Describe your overall method (perhaps supported by figures showing individual steps) and what your assumptions are. Your results **should be reproducible** from this description.
- **Use the theory:** Use theoretical concepts and terminology from the course to explain your solutions and findings.
- **Evaluate your results:** Discuss and reflect on your results. Explain possible reasons for the outcome.
- **Remember that you are learning:** We don't expect you to be fluent in all the material. Solve the tasks to the best of your abilities and show that you tried.
- **Use the available resources:** Use the course TAs as much as you need. If you are stuck, ask for hints or a simple deliberation on the meaning of a specific exercise. Although we cannot give you specific answers, we might be able to nudge you in the right direction.

The final grade (passed/failed) will be based on an overall assessment of your ability to fulfil the requirements. For questions regarding the general course requirements and differences between bachelor and master studens, refer to the official course descriptions.

Collaboration

We suggest you work together in groups of three to four people. Although the exam is strictly individual, we allow and even encourage a certain level of cooperation. Specifically, you should follow these rules:

1. Discussions on approaches and solutions are allowed.
2. Helping each other with coding is allowed.
3. Using a common code base is not allowed.
4. Writing the report together is not allowed.

Just to make it absolutely clear, copying text and code from each other or other sources in the final assignment is not allowed and will be considered to be plagiarism.

Exam Part 1.1

Transformation library

Your first task is to create a library for creating and combining arbitrary linear transformations in two dimensions and applying them to two-dimensional points and images. You will use the library in the next exercise. You must elaborate on your design decisions and implementation details in the report.

Task 1: *Implementation*

Python allows many different approaches. You may decide how you want to approach this problem. You may use a traditional object-oriented approach but it isn't necessary.

Below is a list of requirements that your library must fulfil.

- ⚠ The library **exposes an easy to use API** that hides the underlying array operations.
- ⚠ The library is implemented as a single-file Python module. Use the file `transformations.py` for your implementation.
- ⚠ The library uses Numpy for array operations.
- ⚠ It is possible to create the following types of transformations: **Identity, rotation, translation, scaling, arbitrary (i.e. by providing a matrix).**
- ⚠ Transformations can be **inverted**, i.e. you must provide a function or method for performing the inversion.
- ⚠ Transformations can be **combined**.
- ⚠ Transformations can be **applied to two-dimensional points** arranged as a matrix of column vectors. Homographic transformation must be handled in the library as well.
- ⚠ Transformations can be **applied to images**. You may use `cv2.warpPerspective` to achieve this.
- ⚠ Manually implement image transformations using backward-mapping.

Note: You are not allowed to use OpenCV for anything except for loading and displaying images (see report requirements for details).

Hints: The following are simply helpful suggestions to keep in mind writing the implementation.

- When transforming images, use pixel indices as coordinates and acquire the backward transform using your library's inverse function/method.

- It will be useful later if your api for combinations can be chained or supports a variable number of input transforms. Ideas for approaches:
  - If your combination function returns the new transformation, they can be chained by function calls, i.e. `trans1.combine(trans2).combine(trans3)`.
  - The function `reduce` in Python's `functools` module would allow the following syntax: `reduce([trans1, trans2, trans3], combine)`.
  - You may use method overloading to make the combination syntax simpler, e.g. `trans1 @ trans2 @ trans3`.
- Use vector-operations as much as possible. Refer to the exercises for inspiration.
- Remember that images in Numpy/OpenCV format use y as the first coordinate and x as the second.

Task 2: *Report*

Your report should follow the structure suggested here:

1. ⚠ **Approach:** What is the overall design of your library? Why did you choose this design and did you consider other possibilities? Give a few examples of how to create transformations and combine them.
2. ⚠ **Implementation details:** How did you implement combining transformations and application to points and images?
3. ⚠ Ⓜ **Backward mapping:** OpenCV's methods for transforming images all use *backward mapping*. Explain, using course theory, what backward mapping is and why it is used.
4. ⚠ **Image tests:** In the file `test_transformations.py`, create a test application that transforms an input image and writes the output to a file. Use the application to demonstrate the effect of at least two different transformations. Each of the two transformations must consist of a combination of three transformations.

   Include before/after images of each example in the report and explain the results.
5. ⚠ **Manual tests:** Use one of the previously created transformations to transform an arbitrary point. Write code for this in `test_transformations.py`. Then calculate the transformation manually (including combining the parts) and compare.

Checklist

- ☐ The file `transformations.py` with your library implementation.
- ☐ The file `test_transformations.py` with your test code.
- ☐ Report section as specified.
- ☐ Two pairs of before/after images.

Exam Part 1.2

Affine estimation

Remember from the lecture the general form of an affine transformation:

$$\begin{bmatrix} a_1 & a_2 & t_x \\ a_3 & a_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{1}$$

This system of linear equations has *six* unknowns and thus requires *six* equations to have a unique solution. Given three 2d point correspondences, we have exactly six equations, two for each pair of points. We may visualise the points as two triangles. Visually, we can understand an affine transformation as the transformation between two arbitrary triangles. You will explore this directly in the next part.

The two equations for each pair of points can be written using regular algebra as

$$\begin{aligned} x' &= a_1 x + a_2 y + t_x \\ y' &= a_4 x + a_5 y + t_y. \end{aligned} \tag{2}$$

In matrix form, this can be written as

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ t_x \\ a_3 \\ a_4 \\ t_y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}. \tag{3}$$

You may want to calculate the matrix-vector product yourself to verify this. Stacking the equations for three points yields

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ t_x \\ a_3 \\ a_4 \\ t_y \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{bmatrix}, \tag{4}$$

where the matrix is square and thus is solvable by using matrix inversion.

Task 1: *Implementation*

- ⚠ Add a new function `learn_affine(points_source, points_target)` to your library that finds the affine transformation between two triangles. Use the matrix inverse method to solve the system.

Task 2: *Report*

1. ⚠ **Tests:** Implement the following test in `test_affine.py`. Create a compound transformation using your library. Then transform three points using the transformation. Finally, infer the affine transformation using the function `learn_affine()` and compare the resulting transformation matrix with the one you created.
2. ⚠ Ⓜ **Number of solutions:** Does all combinations of points produce valid solutions? If yes, provide an argument why, otherwise provide a counter-example and explanation.

Checklist

- ☐ The file `transformations.py` with the `learn_affine()` function added.
- ☐ The file `test_affine.py` with your test code.
- ☐ The specified report sections.

Exam Part 1.3

Texture mapping

Figure 1: Screenshots from the application you will develop. Notice how manipulation of the triangle mesh allows deformation of the input texture.

You will now use your library to create a texture mapping application. Specifically, you will be mapping a texture (just an image) to a triangle mesh in two dimensions. A two-dimensional triangle mesh is simply a set of triangles that define the shape of a polygon. An example is shown in Figure 2. Each triangle describes the shape locally.

The shape of the mesh can be changed by moving the points (typically called vertices) of the mesh. This changes the shape of each individual triangle in the mesh. As we discovered in the last exercise, changes to a triangle's shape can be completely described by an affine transformation.

The goal of texture mapping is to transform pixels in a texture according to the shape changes of its associated mesh. In our case, this transformation can be described by the transformations of the individual mesh triangles. We apply the affine shape transformation of a triangle to the texture in the region covered by the triangle. An example of this is shown in Figure 1.

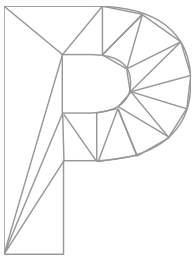

Figure 2: Example of a triangle mesh outlining the shape of the letter "P".

Each triangle has a position and shape in the original texture. The shape of the triangle is then changed (in this application by using the mouse) but the texture points remain stationary. The texture thus has to be transformed to the shape of the new triangle. The method can be described as follows. For each triangle in the mesh the following operations are performed:

1. The affine transformation between the original and deformed triangle is found.
2. The whole image is transformed according to the affine transformation.
3. The image is masked such that only the part covered by the deformed triangle is shown.

4. This triangular image region is added to the final image.

Thus, the final image is created by transforming each image patch according to the triangle deformation. The steps for a single triangle are visualised in Figure 3.

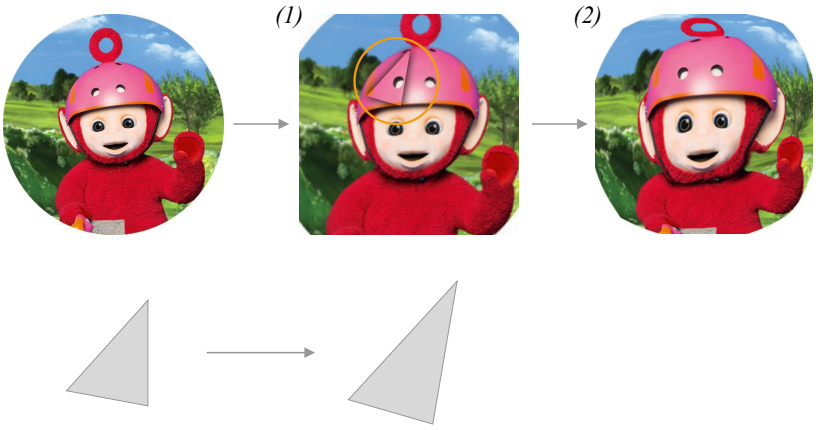

Figure 3: Mapping technique. First, transform the image according to the affine transformation between the original and deformed triangle (1). Next, mask the image such that only the part covered by the transformed triangle is used (2).

Task 1: *Texture mapping implementation*

We provide a skeleton for the application in `texture-map.py`, complete with a number of functions. It already allows manipulation of a triangle mesh using a mouse and keyboard. Read the documentation of the main function for details.

The application automatically keeps track of the original mesh points and the deformed mesh points. The mesh is generated automatically from points using `cv2.SubDiv2D` [1].

[1] Link to a detailed tutorial with implementation examples.

The application is comprised of the following classes:

- `TriangleMesh` contains all the functionality for creating and modifying the underlying mesh of triangles.
- `MeshGUI` is responsible for updating the display and handling user events.
- `TextureMap` is responsible for mapping the texture given a specific `TriangleMesh`.

You will only have to change methods in `TextureMap`.

1. ⚠ **Transform individual triangles:** Implement `transform_patch()`.
   (a) Find the affine transformation between `triangle_base` and `triangle_transformed` using your tranformation library.
   (b) Apply the transformation to the entire input texture (available in `self.texture`).
   (c) Return the transformed image.
2. ⚠ **Mask the transformed images:** Implement `mask_patch()`. You transformed the whole image for every triangle and you now need to perform the masking step.

(a) Create an empty image for the result using `np.zeros()`. Use `dtype=np.uint8` and the same size as the input `patch`.

(b) Draw the `triangle` by using `cv2.fillPoly()`[2]. Be aware that the function expects a list of arrays of points as input (see examples).

(c) Use OpenCV to mask the `patch` image with your newly created mask. Use Exercise 2 for inspiration if necessary.

[2] Examples of use can be found here. API docs can be found here.

3. ⚠ **Update all triangles:** Implement `update_patches()`. This method should iterate through each index to be updated and apply the transformation and masking steps from above.

(a) Iterate over `indeces`.

(b) For each index, get the original and transformed triangles by calling `self.mesh.get_mapping_points(i)`.

(c) Apply the `transform_patch()` and `mask_patch()` functions.

(d) Save the resulting image in `self.patches[index]`.

4. ⚠ **Perform the texture mapping:** Implement `get_transformed()`. Using the previously implemented functions, create a complete texture mapping.

(a) Create an empty image to draw individual patches to. It should have the same size as `self.texture`.

(b) Iterate over `self.patches`.

(c) For each patch, merge it with the empty image using a suitable arithmetic function from OpenCV[3].

(d) Return the final image.

[3] A short overview of possible methods can be found here.

You should now have a working application.

Task 2: *Improvements*

In the following, choose one of the suggested improvements and implement it.

1. ⚠ **Effective transformations:** Instead of transforming the whole image for each individual triangle, it is much more efficient to cut out only a rectangle covering the associated triangle. With this method, you have to translate the texture to account for the offset introduced by cutting the texture. Luckily, your transformation library already supports combining transformations. Use the method `triangle_properties` in `TextureMap` to get the position and size of a triangle.

2. ⚠ **Caching:** At every iteration of `MeshGUI.loop()`, all patches are transformed and masked and combined, even if nothing changes. Even when dragging points, only the triangles that use the point are changed. Add a way of caching the patches and the complete image returned by `get_transformed()`. The method

`update` in `TextureMap` is already called whenever the underlying `TriangleMesh` changes. Use it to implement the caching functionality.

Task 3: *Report*

This exercise was very code heavy

1. ⚠ **Evaluation of initial implementation:** How well does your implementation work? If you encountered major issues, explain them here. You should also include at least three example pictures of the application. Finally, record a video of you using the application. Use the built-in recording functions.
2. ⚠ **On improvements:** Explain which improvement challenge you chose (its okay to do both) and what approach you used for solving it. Evaluate your results. Did the changes result in significant performance improvements? Did you encounter new issues? Record a new video to demonstrate the improved application.

Checklist

- ☐ The file `texture-mapping.py` with all the changes you made, i.e. you do not need to create a separate file for the *improvements* task.
- ☐ Three example pictures of the application in action.
- ☐ A video demonstrating that the application works.
- ☐ A video demonstrating the improved version of the application.
- ☐ Report section as specified.