3D Rendering

150015686

April 2018

Introduction

The aim of this project is to create a tool to be used to interpolate between a number of different face models with constant mesh by providing a helpful user interface for choosing the face models to be interpolated and also to display the resulting model.

Face repository

The first step of this project was to ingest the given data set and convert it into a format that could be easily used later on.

The data is provided in a format that requires some processing to extract the actual shape and texture of the model: an average shape and texture is defined, upon which offsets need to be added to arrive to the final model.

More precisely:

$$V_i = V_0 + wv_i * Voff_i$$

$$T_i = T_0 + wt_i * Toff_i$$

Where V_i and T_i are the vertices and colours of face i, V_0 and T_0 are the vertices and colours of the average face, and wv_i and wt_i are the weights with which the vertex and texture offsets are needed to be multiplied.

All faces are read and processed at startup and saved in memory for easy access.

Rendering

The first step in rendering the models on the screen was set up the boilerplate necessary to give access to the means of drawing on a window. For this project, we used Java Swing to create a component and override it's `paintComponent(Graphics g)` method, giving us access to the `Graphics` object which can be used to draw on the window.

Wireframe

To test that the face was loaded correctly, we first just drew the triangles on the screen as a wireframe. The resulting render was satisfactory (see Renders)) confirming that the faces were loaded appropriately.

During the implementation of the wire frame renderer, we noticed the following facts about the given model:

- the face was centered at (0,0) on the x and y axes
- the scale of the face was quite large, with the values of the vertices being around the order of 10^5
- the face was aligned with the xy plane

These factors had to be taken into account for the final render:

- the screen space had to be translated, as the default settings has the 0,0 point on the top left of the screen

- the face had to be scaled down, so that it could fit into screen space (which was set to only 500 px).
- due to the face's alignment, a quick ortographic projection could be achieved by just ignoring the z component of the vertices.

Colouring

The next step was to add colour to the drawn triangles. We defined the colour of each of the triangles to be constant on its surface (no interpolation) and to be equal to the average colour of the vertices defining the triangle. With this we got the first colour render. And with this, the next issue to solve was discovered. Some of the triangles were being drawn in the wrong order, leading to artifacts and overall a bad render quality.

Using depth information

Fortunately, this is a problem that has been already solved in many different ways. One of these ways is to use something called the Painter's algorithm. The algorithm naively solves the problem by drawing the triangles in the order of their distance from the projection plane (z=0 in our case) from furthest to nearest. This approach is known to have issues when drawing complex scenes, as it orders entire triangles from furthest to nearest and therefore there is the possiblity of drawing intersecting triangles incorrectly.

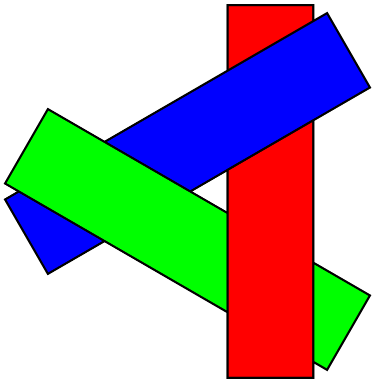


Figure 1: Scene configuration impossible to render with Painter's algorithm

The result (see Renders) was satisfactory, allowing us to move to the next step.

Lighting

The lighting for the scene uses Phong's reflection model with a directional light in the $+z$ direction (same direction as the one the camera is facing). The resulting lighting looks accurate, but could benefit greatly from interpolation between the vertices, especially in areas of high curvature magnitude, such as the nose or the chin.

An ambient source of light was added as well, to make the scene more bright. With this addition, the resulting values of the computation had to be clamped to the correct range, as some areas of the image, especially those perpendicular to the viewing direction (e.g the forehead) were oversaturated and caused exceptions with the Graphics backend.

Additionally, the colour and direction of lights can be configured, as specified in the Configuration section.

Transformations

For all this time, the render only drew one face in the middle of the screen. But the final project requires a number of other elements on the screen.

For this, we took inspiration from the Processing graphics library and designed a renderer that allows the user to apply transformations to the scene, draw elements on the screen and also save and restore transformations with the help of a stack.

The renderer provides the basic transformations:

- scale
- translate
- rotate in all 3 axes

These are then used to compose the final UI and also allows us to display more than just one face on the screen.

To showcase the rotation transformations, the composed face is rotated by 10 degrees on the Y axis. For further tinkering see the Configuration section.

The lighting is applyed after all the transformations, therefore always keeping the light direction as defined.

Rotation has been exposed to the user using the following key bindings: - `J` and `L` to rotate around the Y axis - `I` and `K` to rotate around the X axis

Face composition

The face composition works in two steps.

The face compositor holds in memory a vector of the same size as the number of faces. Each value will be in the range $[0, 1]$. Then the compositor can create a new face using this vector firstly dividing the vector by the sum of its components to ensure that the sum of all the components is 1 and then creating a new face by multiplying the face data (vertices and texture values) with the created weight and then summing them together.

$$F_{composed} = \sum_{i<no\_faces}^{i=0} \frac{C_i}{\sum_{j<no\_faces}^{j=0} C} * F_i$$

Where $F_{composed}$ is the resulting face, F_i is one of the given faces and C is the composition vector.

To actually set these values the user is presented with a triangle, with each point of the traingle being annotated with a face and its id. When the user clicks on the triangle, the weights of the faces present are adjusted by converting the click of the user into barycentric coordinates relative to the triangle using the following formulae:

$$\begin{aligned}\lambda_1 &= \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{\det(T)} = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)}\,, \\ \lambda_2 &= \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{\det(T)} = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)}\,, \\ \lambda_3 &= 1 - \lambda_1 - \lambda_2\,.\end{aligned}$$

where (x_i, y_i) are the positions of the i^{th} triangle, (x, y) is the position of the mouse click and $\lambda_1, \lambda_2, \lambda_3$ are the resulting weights for the 3 faces. As required by the compositor, the resulting weights are between 0 and 1.

The user is then allowed to change the faces at the vertices of the triangle as such:

- to change the currently selected face (the one whose id is in square brackets), press left or right arrow.
- to change the currently selected index, press A or D.

Configuration

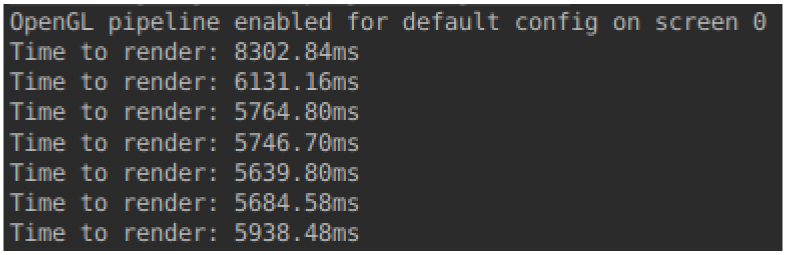
To expose some of the functionality and allow for customisation, a config file can be found (`app.properties`) with the following configurable properties:

- `offsetAmplificationFactor` (default: `1`) - a factor used to amplify the vertex offsets when generating the faces. This was used to amplify the differences between faces, which can make the result of the interpolation more visible
- `dataPath` (default: `data\`) - the path to the folder with the data files
- `faceCount` (default: `4`) - the number of faces to be loaded (min: `3`, max: `199`)
- `ambientColor` (default: `0.2, 0.2, 0.2`) - the colour of the ambient in RGB components between 0 and 1
- `directLightDirection` (default: `0,0,1`) - the direction of directional light (x,y,z). The direction will be normalized
- `directLightColor` (default: `1,1,1`) - the colour of the directional light in RGB components between 0 and 1
- `scaleX` (default: `1`) - scale factor to be applied in the x direction
- `scaleY` (default: `1`) - scale factor to be applied in the y direction
- `scaleZ` (default: `1`) - scale factor to be applied in the z direction
- `translateX` (default: `0`) - translation to be applied in the x direction
- `translateY` (default: `0`) - translation to be applied in the y direction
- `translateZ` (default: `0`) - translation to be applied in the z direction
- `rotateX` (default: `0`) - rotation to be applied on the x axis (in degrees)
- `rotateY` (default: `-10`) - rotation to be applied on the y axis (in degrees)
- `rotateZ` (default: `0`) - rotation to be applied on the z axis (in degrees)

Performance

By this point it became quite obvious that rendering the final UI at a decent FPS is but a dream, but nontheless some performance improvements were attempted, driven by the use of a profiler.

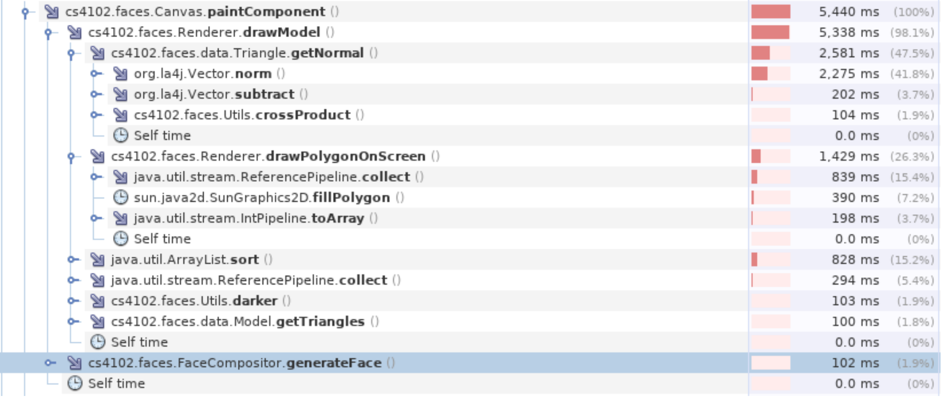
Before the start of the optimization, this is a snapshot of the usual time to render the scene:



```
OpenGL pipeline enabled for default config on screen 0
Time to render: 8302.84ms
Time to render: 6131.16ms
Time to render: 5764.80ms
Time to render: 5746.70ms
Time to render: 5639.80ms
Time to render: 5684.58ms
Time to render: 5938.48ms
```

Figure 2: Render time before optimization

And here is the result of profiling the drawing. Do note that the time is not relevant as profiling hits performance. We're mostly interested at the ratios:



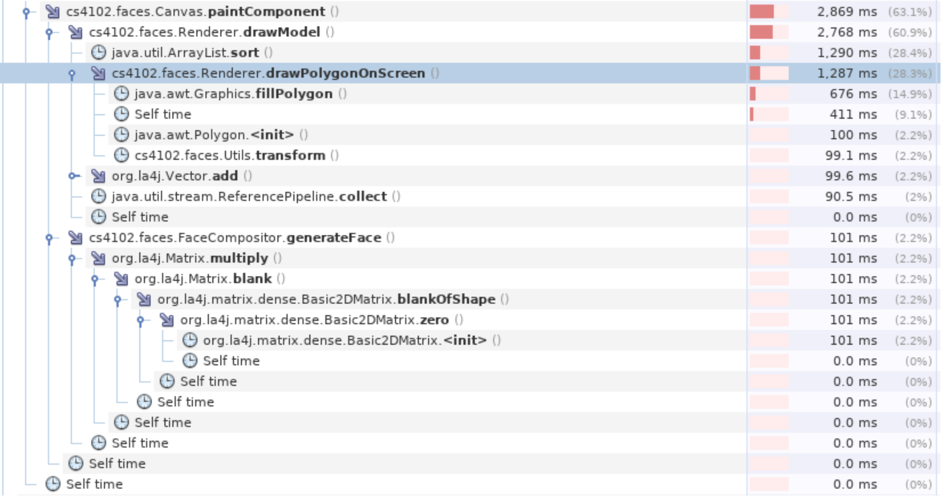
```
cs4102.faces.Canvas.paintComponent ()                                5,440 ms  (100%)
  cs4102.faces.Renderer.drawModel ()                                 5,338 ms  (98.1%)
    cs4102.faces.data.Triangle.getNormal ()                          2,581 ms  (47.5%)
      org.la4j.Vector.norm ()                                        2,275 ms  (41.8%)
      org.la4j.Vector.subtract ()                                      202 ms  (3.7%)
      cs4102.faces.Utils.crossProduct ()                               104 ms  (1.9%)
      Self time                                                        0.0 ms  (0%)
    cs4102.faces.Renderer.drawPolygonOnScreen ()                     1,429 ms  (26.3%)
      java.util.stream.ReferencePipeline.collect ()                    839 ms  (15.4%)
      sun.java2d.SunGraphics2D.fillPolygon ()                          390 ms  (7.2%)
      java.util.stream.IntPipeline.toArray ()                          198 ms  (3.7%)
      Self time                                                        0.0 ms  (0%)
    java.util.ArrayList.sort ()                                        828 ms  (15.2%)
    java.util.stream.ReferencePipeline.collect ()                      294 ms  (5.4%)
    cs4102.faces.Utils.darker ()                                       103 ms  (1.9%)
    cs4102.faces.data.Model.getTriangles ()                            100 ms  (1.8%)
    Self time                                                          0.0 ms  (0%)
  cs4102.faces.FaceCompositor.generateFace ()                          102 ms  (1.9%)
  Self time                                                            0.0 ms  (0%)
```

Figure 3: Profiling before optimization

We know that we cannot optimize the actual drawing, as it is done by the awt library. But we see that a large proportion of the time is taken by calculating normals, and more accurately by calculating the magnitude of a vector. Additonally the collect call is used to transform the vertices of a triangle into scren space using a matrix multiplication.

From here we see that the weak link of the pipeline is the linear algebra library. Due to the fact that it is designed to work for any matrices and vectors, some operations have some overhead which quickly sums up when we are doing a few hundred thousand multiplications. Therefore extended the given matrices and vectors to improve performance for our use case which is 4x4 matrices and 4-vectors, while also being compatible with the rest of the library.

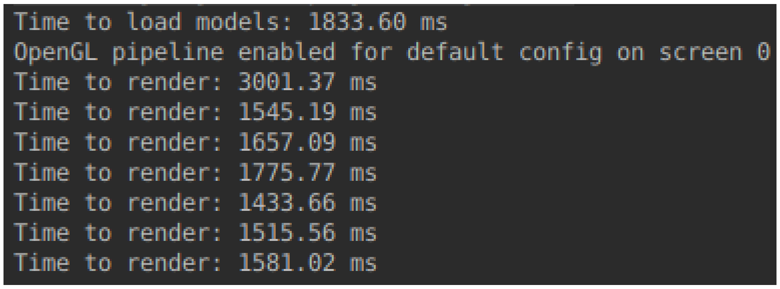
After this optimization step has been completed, here is the result of the profiling:



```
cs4102.faces.Canvas.paintComponent ()                                2,869 ms  (63.1%)
  cs4102.faces.Renderer.drawModel ()                                 2,768 ms  (60.9%)
    java.util.ArrayList.sort ()                                      1,290 ms  (28.4%)
    cs4102.faces.Renderer.drawPolygonOnScreen ()                     1,287 ms  (28.3%)
      java.awt.Graphics.fillPolygon ()                                 676 ms  (14.9%)
      Self time                                                        411 ms  (9.1%)
      java.awt.Polygon.<init> ()                                       100 ms  (2.2%)
      cs4102.faces.Utils.transform ()                                 99.1 ms  (2.2%)
    org.la4j.Vector.add ()                                            99.6 ms  (2.2%)
    java.util.stream.ReferencePipeline.collect ()                     90.5 ms  (2%)
    Self time                                                          0.0 ms  (0%)
  cs4102.faces.FaceCompositor.generateFace ()                          101 ms  (2.2%)
    org.la4j.Matrix.multiply ()                                        101 ms  (2.2%)
      org.la4j.Matrix.blank ()                                         101 ms  (2.2%)
        org.la4j.matrix.dense.Basic2DMatrix.blankOfShape ()            101 ms  (2.2%)
          org.la4j.matrix.dense.Basic2DMatrix.zero ()                  101 ms  (2.2%)
            org.la4j.matrix.dense.Basic2DMatrix.<init> ()              101 ms  (2.2%)
            Self time                                                  0.0 ms  (0%)
          Self time                                                    0.0 ms  (0%)
        Self time                                                      0.0 ms  (0%)
      Self time                                                        0.0 ms  (0%)
    Self time                                                          0.0 ms  (0%)
  Self time                                                            0.0 ms  (0%)
Self time                                                              0.0 ms  (0%)
```

Figure 4: Profiling after optimization

And here is the average rendering time for the scene:



```
Time to load models: 1833.60 ms
OpenGL pipeline enabled for default config on screen 0
Time to render: 3001.37 ms
Time to render: 1545.19 ms
Time to render: 1657.09 ms
Time to render: 1775.77 ms
Time to render: 1433.66 ms
Time to render: 1515.56 ms
Time to render: 1581.02 ms
```

Figure 5: Render time after optimization

As you can see, the optimization reduced the time for rendering the scene from ~6 seconds per frame to 1.5 seconds per frame. While this is not close to real time, it is still a lot better.

The profiling now shows an almost perfect result, given the approach we are taking with the time to render being split between the actual draw calls to the graphics backend and the sorting of the triangles.

The sorting of the triangles could be replaces with a different approach, such as z-buffering, but the Graphics library we are using doesn't support direct pixel manipulation.

For optimizing the system overall, a lot of the work could be passed on to the GPU, both for the actual rendering pipeline and for the face composition, as GPU's are optimized for matrix and vector manipulations.

Evaluation

The basic specification was fully achieved, with the system successfully composing 3 faces using a triangle as a user input. The composed face is displayed using flat shading and an ortographic projection.

The advanced specification was also implemented, with the user being able to select which faces to compose from and with the system remembering the weights of the previously used faces. The user is able to preview the faces between which he is interpolation, but not when selecting the face to be used in the interpolation. This was a design choice, as with the system's current performance, rendering 100 faces would take quite a long time.

Some of the highly advanced requirements have also been implemented, with the user being able to change the direction and colour of the light, the addition of ambient lighting, as well as the ability to transform the generated face by rotating, scaling and translating it on the screen using configuration parameters. All these transformations can be done at run time as well, but for the project only the x and y axis rotations have been bound to keys.

Conclusion

This was an interesting practical to implement. I am overall happy with the submission. I wish I had started the practical using a library like OpenGL, as it would be more close to what the state of the art is and also produces an a lot faster system.

Run instruction

Modify the config as you wish…

Only 10 faces have been provided in the submission. If you want to use more, please change the `dataPath` property.

In the project folder run:

```
./gradlew run
```

Renders from different stages of implementation

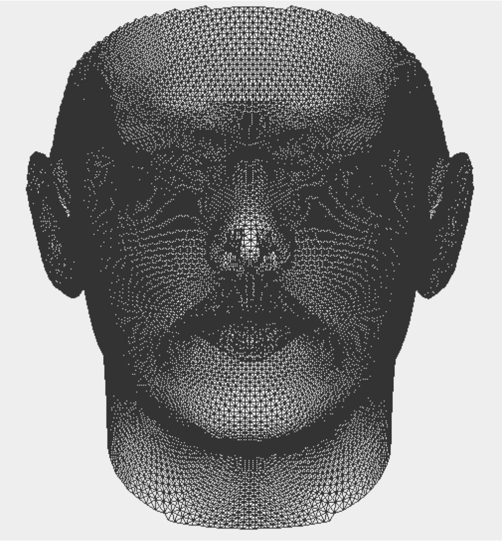


Figure 6: Wireframe render

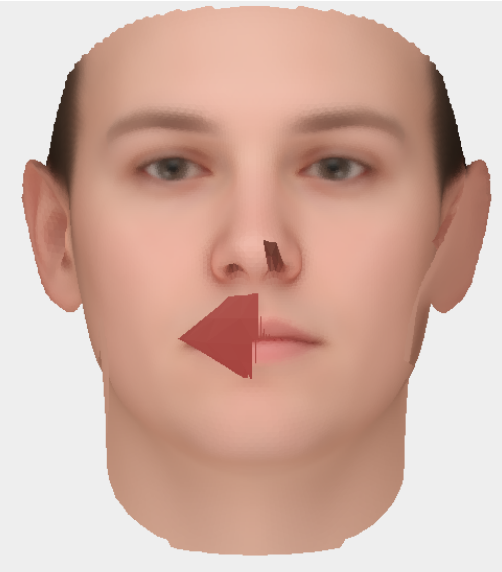


Figure 7: Colour render

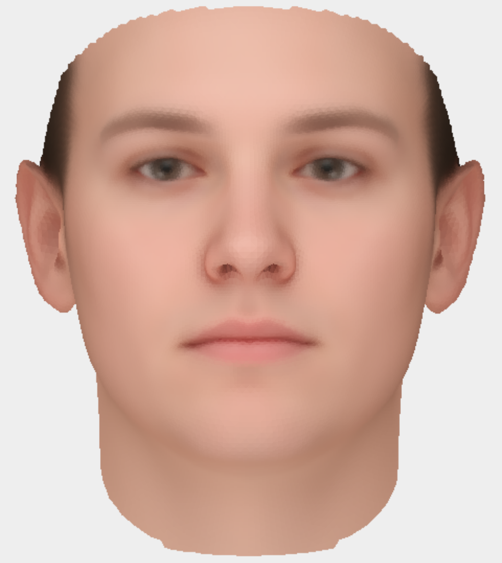


Figure 8: Painter's algorithm render

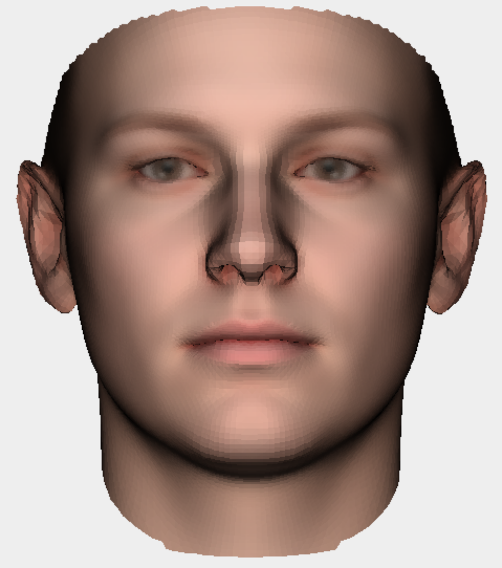


Figure 9: Lighting-enabled render

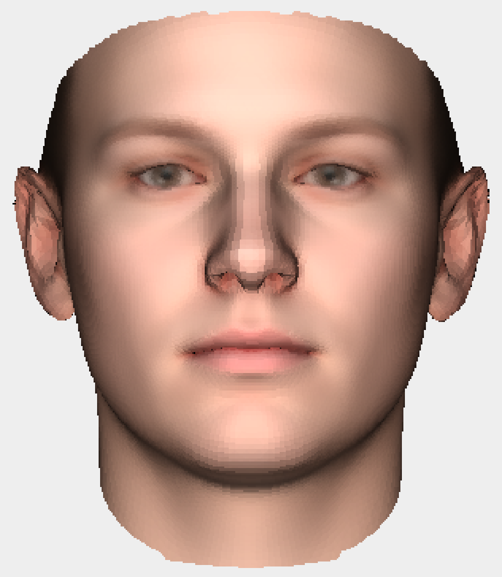


Figure 10: Ambient light added

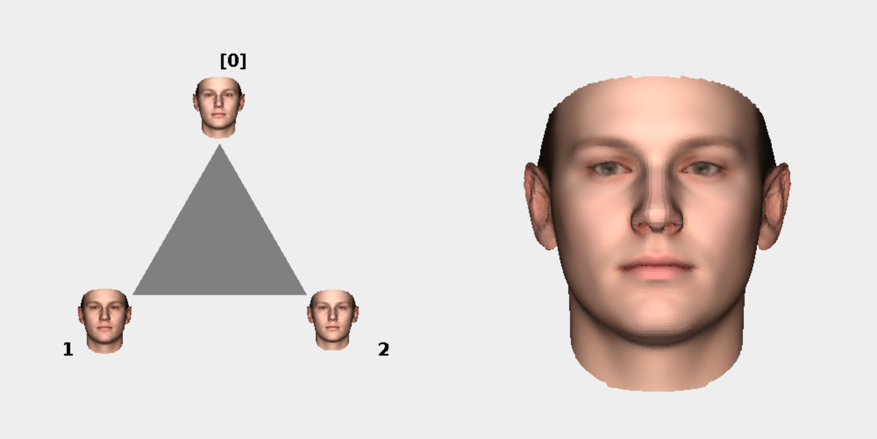


Figure 11: Final UI