Interactive graphics
Final project

Lorenzo D'Auria (1918917)
Gallotta Roberto (1890251)

1 Introduction

1.1 Overview

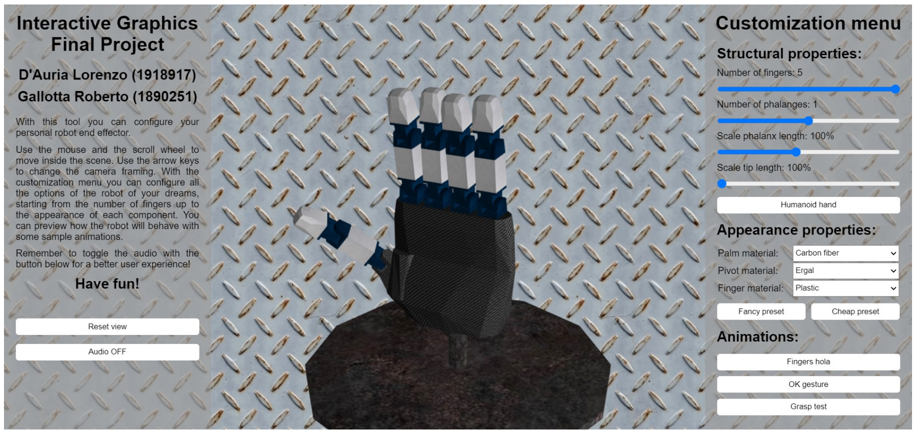
In this project we developed an interactive interface for configuring a robot end effector based on some fixed options. The user can see in real time the render of the end effector and modify the design with buttons and sliders.

The user can select a humanoid hand with a thumb, or a simple robotic gripper with fingers. The number of fingers and the number of phalanxes forming each finger can be changed, as well as the length of the phalanxes and the length of the finger's tips. The user can also change the end effector appearance choosing from different materials for the palm, the pivots and the fingers. Animations and sounds are also present to enhance the user experience.

This project uses only standard WebGL and the two libraries presented during the course: `MVnew.js` and `InitShaders.js`.

1.2 User interface

The window appears to the user as such.



Interactive Graphics Final Project

D'Auria Lorenzo (1918917)
Gallotta Roberto (1890251)

With this tool you can configure your personal robot end effector.

Use the mouse and the scroll wheel to move inside the scene. Use the arrow keys to change the camera framing. With the customization menu you can configure all the options of the robot of your dreams, starting from the number of fingers up to the appearance of each component. You can preview how the robot will behave with some sample animations.

Remember to toggle the audio with the button below for a better user experience!

Have fun!

Customization menu

Structural properties:
Number of fingers: 5
Number of phalanges: 1
Scale phalanx length: 100%
Scale tip length: 100%

Appearance properties:
Palm material: Carbon fiber
Pivot material: Ergal
Finger material: Plastic

Animations:

Figure 1: View at startup

The left bar contains a simple explanation of the tool with two buttons: one for activating the sounds and one for resetting the view to the initial configuration. On the right side of the window are present all the customization options already mentioned, subdivided in structural and appearance properties. The choice of the materials can be modified also by applying one of the two presets available for a cheap or a fancy configuration.

The user can move in the scene using mouse drag and the scroll wheel and can change the camera framing with the four arrow keys. The model is normally moving with an idle animation, but with the three buttons on the bottom some special animations can be previewed. Based on the configuration chosen, some animations may be disabled due to inconsistency or bad aesthetic results. This will be explained in greater details in the upcoming sections.

If the audio is activated the user will ear an ambient music playing and some dedicated sounds effects while changing configurations; we personally think that with them the user experience is more engaging.

2 Scene design

2.1 Parts design

The first step of this project consisted in the 3D modeling of all the elements in the scene since no preexisting models were used. With the use of the CAD software *Fusion 360* we created the base, the two palms (humanoid and robotic), the phalanx, the finger tip and the thumb models that were saved as `.stl` files.

A `.stl` file contains the vertex coordinates and the triangles definitions from the vertices, so to use these values inside our program, we had to extract and translate the model definition from the `.stl` file to the standard definition seen during the course with the `vec4()` and `triangle()` function calls. From the files it was also possible to extract the normals definition of each triangle that will be used for the light computation.

2.2 Scene setting

On the back of the scene, there is a fixed background of the size of the canvas on which a texture was applied. A directional light source lights all the scene except for the background, which depends only on the texture color. The center of the scene coincides with the center of the spherical joint of the wrist, while the camera center can be moved with the arrow keys.

The end effector model representation depends on the projection matrix computed with the call of the `perspective()` function and on the model view matrix computed for each piece starting from the `lookAt()` function that models the camera movements.

2.3 On-load routine

When the web page is loaded, the `init()` function is called. In this function all the preliminary operations are done. First the sounds are initialized, then the WebGL canvas is configured with special care for the window size. The three buffers for points, normals and texture coordinates are populated locally and then passed to the GPU. Locations of the uniform variables that may change during the execution are stored to be reused inside the render function. Textures, event listeners and light are configured and the model in the scene is initialized in a standard configuration. At this point the animation cron job is started and the execution will continue looping inside the render function.

2.4 Render function and shaders

In the render loop is handled the particular behavior for the background with the uniform variable `uBack` used by the shaders. The projection and the model view matrices are then updated based on the actual camera position and orientation. At this point the whole model is drawn by traversing the hierarchical tree and the loop restarts.

The shaders are located inside the `.html` file. In the vertex shader, the value of `gl_Position` is computed using the projection and the model view matrices. In order to use a per-fragment for the light computations, the vectors used in the Phong model are computed and passed to the fragment shader. In case of a vertex of the background the `gl_Position` is computed using only the canvas size and no light computation is done.

In the fragment shader, the `fColor` first is computed using the texture sampler, then if the fragment doesn't belong to the background, the Phong model light computations are done and the color contribution of light is used by multiplying it by the `fColor` value.

In the `.html` file we also defined the Javascript scripts to load, the image resources and the audio sources.

3 Hierarchical model

3.1 The base node structure

Before introducing the hierarchical model, we present the base node structure we implemented in order to have a very flexible and easily updatable code.

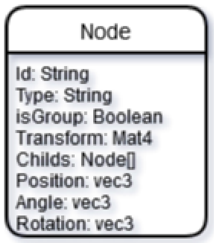


Figure 2: The node structure

We now motivate each attribute of the node:

- `id`: Each node has an unique id in the scene, which allows fast retrieval using the `findNode(id, element)` function. Since we can update the structure at the user's request, we defined the function `updateID(id, inc)` that handles the correct update of the nodes' ids in the tree.
- `type`: Each node is associated to a single model type. This allowed us to write a generic render function that would work for all model types.
- `isGroup`: This boolean variable determines if a node is to be considered as a group node or not. A group node is a node that should not be rendered. This is used mainly for handling the fingers in the robotic hand (with the group identified by `appendages`). Originally we had groups also for the *Falange* and *Falangina* types, but discarded them since handling the alignment of the pieces when scaling them resulted disadvantageous.
- `transform`: The current transform matrix of the node.
- `childs`: The list of child nodes.
- `position`: The position of the node, expressed as a vector of three values (position in the X, Y and Z coordinates).
- `angle`: The angle of the node, expressed as a vector of three values (around the X, Y and Z axis).
- `rotation`: The additional angular rotation of the node, expressed as a vector of three values (around the X, Y and Z axis). Note that this differs from the `angle` value in the following way:

$$\texttt{Total node rotation} = \texttt{angle} + \texttt{rotation}$$

3.2 The hierarchical tree

With the node structure well defined, we can then look at how we implemented the full hierarchical model. We preface this by reminding that we implemented a dynamic hierarchical model, with a customizable number of fingers and phalanxes and a customizable range of scaling for the *Falange* and *Falagina* phalanxes. This obviously added a layer of complexity that we will show later how it was handled.

The generic hierarchical model we implemented can be shown in the following diagram.

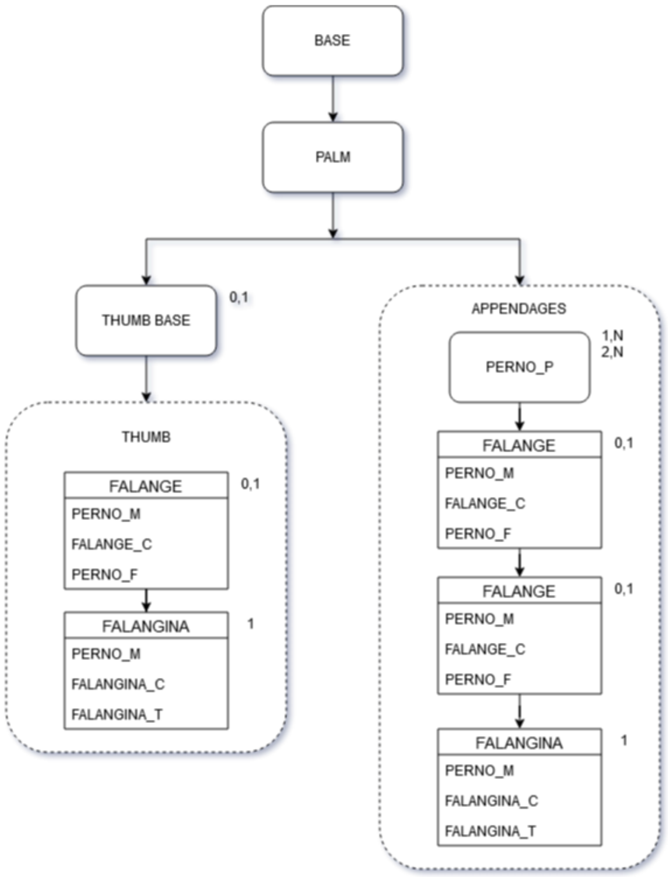


Figure 3: The generic hierarchical model; dotted lines represent a group, numbers represent the possible occurrences of a node type/archetype, arrows represent dependency

We first note that *Falange* and *Falangina* are archetypes of nodes, determing the triplet of nodes with type `perno_m`, `falange_c`, `perno_f` and `perno_m`, `falangina_c`, `falangina_t` respectively. The dependency between the two archetypes is that `perno_m` is the child of `perno_f`.

We now explain how the hierarchical model is composed for both the humanoid hand and the robotic one: both start with the same base, at the top of which we have either the humanoid palm or the robotic palm. In case of the humanoid hand, the palm includes the base for the thumb and the thumb, a group node that always contains a *Falangina* and can contain at most one *Falange*.

Both types of palms have, as child, the group node denoted by the "Appendages" id. This group is used to handle the fingers from the palm separately, especially when considering the different cases of removing/adding fingers when there is a thumb present or not, depending on the type of the hand. The Appendages group node contains all the fingers; a finger start with a `perno_p`, to which is always attached a *Falangina* archetype node and at most two *Falange* archetype nodes.

Note that for the humanoid hand, we have a minimum of one finger (other than the thumb), whereas for the robotic hand we have a minimum of two fingers.

3.3 Adding/removing nodes

Addition and removal of fingers is handled calling the corresponding functions `addFinger(node)` and `removeFinger(node)`, with the `node` being the group node Appendages by default.

Removing a finger simply removes the last child of the node `childs` and realigns the remaining fingers. We can remove a finger as long as we have enough fingers remaining afterwards (this is dependent on the type of hand). Adding a finger, instead, consists of building a new finger with the correct number of phalanxes, adding it to the node's `childs` and aligning the fingers.

We note two things before continuing:

- The alignment of fingers is done via the function `alignFingers(fingers)`, which follows a given preset for the robotic hand and computes the positions dynamically for the humanoid hand.
- Each finger consists of a `perno_p` node with an incremental id number with root "`finger`" (e.g.: "`finger0`"). Following archetype nodes will have an additional "`_x`" (with `x` being a number) and nodes will have an additional "`_x`" (with `x` being a number); so, for instance, the `perno_f` of the second *Falange* of the first finger will have "`finger0_2_3`" as id.

Adding and removing phalanxes proved to be slightly trickier. We remove a phalanx by finding the base node (of type `perno_p`) and the second phalanx (that can be either a *Falange* or a *Falangina*), we remove the first phalanx from the base and instead put the second phalanx. We then update the ids from the newly-first phalanx and make sure its placement is correct.

When adding a phalanx, we first find the base node and the next phalanx (the child of the base node), we create a new phalanx of type *Falange*, we make that as the child of the base node and set the previously first phalanx as child of the first new phalanx. We then update the ids from the second phalanx on and correct its position.

Both adding and removing a phalanx are constrained by the minimum and maximum number of phalanxes admissible (0 and 2, respectively). Adding and removing fingers are also constrained by a minimum of and a maximum: at any given point, a humanoid hand can have at most 4 fingers and at least 1, whereas a robotic hand can have at most 4 fingers and at least 2.

3.4 Rendering the tree

In order to render the hierarchical model correctly, we implemented the function `traverse(element)` similar to the one from Homework 2, but tweaked it for the new node structure. In particular, we note that, since we deal with different models, we update the matrix of normal vectors for each node. The model view matrix is handled as expected, updating it with the node's transform matrix and storing it in a stack, removing it after traversing the node's `childs`. As aforementioned, if a node is a group node, it's not rendered, albeit it updates the model view matrix.

The generic render function is `drawElement(element)`, which handles setting up the correct material (a call to `setupMaterial(type)` which updates the values related to the material with a lookup to the `materials` table) and the correct texture (a call to `useTexture(type)` that updates which texture to use during the rendering process via a lookup to the `materials` table) to use for the current node.

The actual drawing on the canvas is done via the `gl.drawArrays()` function, drawing `gl.TRIANGLES` with the points stored in `pointsArray` at the locations specified in the table `boundsMap`.

4 Texture and materials

4.1 Textures

In order to have a defined texture for each component of the scene, we first imported each model in *Blender*. Within it, we created the UV mappings, which can be found in the `Resources/UVs` folder of the project. From those, we extracted the texture coordinates for each model.

Using the modified `triangle` function, we added the UV coordinates for each face to the `texCoordsArray` list. That way, when we render a triangle, we use the correct portion of the texture. We tested this by rendering the scene using a default background texture and the UV textures for the model.

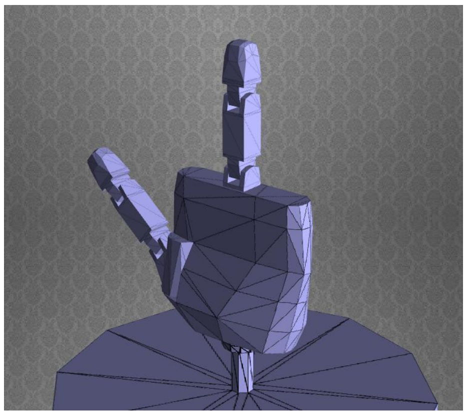


Figure 4: Testing the UV textures for the model; the width of the lines is caused by a stretching of the texture.

We then proceeded to assign a texture for each material. This was simply implemented by the `textures` table that would setup the `gl.TEXTURE_2D` from the source image (defined in the `.html` file) and assign it to the material key.

Then, when rendering the model, we switch which texture to use according to the desired material.

4.2 Materials

We defined a material as a class with the following light components properties:

- Ambiental
- Diffusive
- Specular
- Shininess

This class follows the requirements of the Phong model.
Each material is mapped to its name and each model piece is mapped to a material. This way, in a rendering cycle, we can update the necessary matrices in the fragment shader in order to obtain a different appearance.

The user has the possibility to choose which material to apply to the palm, the pivots and the phalanxes, though we note here that each node type could have a different material applied. Updating the material is done by simply updating the corresponding value in the `materials_map` table and the result is visible from the following render cycle.

5 Animations

The animations are treated inside the `animations.js` file. In the `init()` function executed on load, we set a cron job with the function `setInterval()` used to call every 50 ms the `updatePose()` function. This way, we ensure that the animations will always be displayed with a specific timing, no matter how fast the render loop is executed. On every call of `updatePose()` we increment the time instant variable `t` which is used to define in which part of the animation time line we are in.
The following four animations have been created:

- **Idle animation:** a loop of small movements to avoid a static scene, as this animation is always active if no other animation are in progress.
- **Fingers hola:** the robotic fingers are closed and reopen in progression.
- **OK gesture:** available only if the hand is humanoid, consist in the touch of the thumb and index finger.
- **Grasp test:** all the fingers are closed to grasp an hypothetical object.

Each animation is defined by a sequence of key frames which uniquely define what pose the model must have in a specific instant of time. The `pose` is an array of 18 elements. The first three elements are the roll, pitch and yaw angles (in degrees) of the spherical wrist, after that each finger has associated a triplet of values identifying the tilt angle of each of his phalanges. The last triplet are the tilt angles of the thumb.

When `updatePose()` is executed, based on the `actualAnimation` value, the two nearest key frames to `t` of the chosen animation sequence are selected, then the value of each element of `pose` is computed by doing a linear interpolation. Then the model relative positions and angles are updated following the `pose` value. This is done by updating the three values of `palm.rotation`, the value of `rotation[0]` for all the existing phalanges of all the fingers and the value of `rotation[1]` of the thumb (if the thumb exist). We note that, in this way if the actual model configuration needs less than the 18 values in the array, the non relevant ones are simply ignored.

Since the special animations are activated with buttons when another animation is in progress, we implemented a special behavior for the transition from the pose when the button was pressed (stored in `transPose`) and the first pose from where the chosen animations starts. This special behavior is activated by setting the `transition` variable to `true`, as this will change the standard computation of `pose` in the linear interpolation in a fixed amount of time from `transPose` to the first frame of the new animation.

Since the end effector can assume very different configurations, the defined animations sometimes can lead to an overlap of some elements, which is aesthetically unpleasant. To solve this problem after every configuration change, the function `checkValidAnimations()` is called. This function will check for each animation, based on the number of fingers and phalanges. If the values of the scaling of `falange_c` and `falangina_c` are allowed, enabling and disabling the button for the animation accordingly.

6 Sounds

We decided to add a background music and sound effects using the standard Javascript library for sounds. We defined the source of each sound in the `.html` file and, upon page loading, we initialize a table for all the sounds.

In order to maintain cross-platform usage, we had to start with the background music disabled (this was mainly due to Firefox policy, where sounds from pages have to be either from a trusted source or activated by the user). Toggling the audio on or off plays (looping) or pauses the music.

Each alteration to the end effector appearance has its own sound effect, launched by the corresponding event handling function.