Nightsky Simulation

Anfängerpraktikum

Thorsten Trinkaus

Supervisor
Rene Weinmann

Heidelberg, Germany, December 12, 2024

Faculty of Mathematics and Computer Science
Heidelberg University

Contents

1 Introduction **1**

2 Related Work **3**

3 Methods **5**

3.1 glMatrix 5

3.2 Data 5

3.3 Rendering Pipeline 6

3.4 Objects in the Scene 7

3.5 Scene Management 12

3.5.1 Scene Construction 12

3.5.2 Rendering the Scene 14

4 Results **23**

4.1 User Interface 23

4.1.1 Controlling the Star System 23

4.1.2 Controlling Star Signs 23

4.2 Interactions with Nightsky 24

4.2.1 Traversing Space 24

4.2.2 Travel between Star Systems 28

4.2.3 Drawing Star Signs 28

5 Conclusion **33**

6 Acknowledgement **35**

7 Apendix **37**

Bibliography **43**

1 Introduction

The Gaia spacecraft is a astrometric observatory built by the EADS Astrium and owned by the European Space Agency (ESA). Since its launch in 2013, Gaia's aim is to create a very precise three-dimensional map of our Milky Way galaxy. During its runtime, Gaia not only recorded the position of over thousand million stars, but also their motions, luminosity, temperature and composition. It is focused on delivering a better understanding of the origin and history of our own galaxy. The data collected by the observatory so far has been released in three batches.

With this project we wanted to use this data, more precisely the third data release, to create an application that visualizes our night sky. To make the visualization as portable as possible, we opted for building a browser based application. We used WebGL as the JavaScript API for rendering three-dimensional scenes within the browser. This also means that we exclusively used HTML, CSS and JavaScript to create our visualization, further improving its portability as it can be deployed as a static webpage. To achieve a good balance between level of detail and performance, we extracted the 10.000 brightest stars rather than using the entire dataset.

The project started out as a group project for a beginner practical at the Visual Computing Group of the University of Heidelberg. This report will only cover my contributions to the project. My task was to develop a JavaScript framework that could handle everything related to the WebGL rendering pipeline and the scene management. This includes parsing the data provided by other team members, building the scene and finally rendering the scene to a browser canvas. As the project progressed, we expanded the visualization to not only render the Gaia data, but also to display star signs in three-dimensional space. To further enhance the user experience, I also added the ability to draw custom star signs and travel between star systems to view the visualization from the perspective of different systems.

2 Related Work

The Gaia data releases are very popular. Naturally, the data has been visualized before this project. A great visualization is Gaia Sky [SJMS19]. This software from the Zentrum für Astronomie at the University of Heidelberg, is available for Windows, Linux and macOS. It not only maps positions and distances, but also the motions of billions of stars. Through its ease of use and level of detail, the open source project can be used for both scientific research and public engagement. Instead of improving on the level of detail, my project will focus on portability. My application can be deployed as a static webpage and does not need to be installed by users.

For work directly related to my project, see the Anfängerpraktikum by Eric Benz and the Anfängerpraktikum by Jan Vomstein, as they continued to build upon the foundation of my framework. Among other things, Eric Benz added star motion to the application, while Jan Vomstein was responsible for all the textures I used.

3 Methods

3.1 glMatrix

glMatrix [JM] is the only JavaScript framework I used to create this application. It is specifically designed to drastically improve WebGL by performing vector and matrix operations. The framework is highly optimized and therefore not only provides simple vector and matrix operations, but also improves the performance of the project.

3.2 Data

At the beginning of this chapter, I would like to briefly discuss the data I used, even though I was not directly responsible for gathering the different datasets. In order to make the visualization possible, I had to parse the given datasets and extract all the needed data.

This application uses three different datasets:

- **Gaia Data-Release** - I was provided with pre-filtered versions of the third Gaia data release, which contain the information for the 100, 1.000, 10.000 and 100.000 brightest stars from the data release. As a compromise between performance and detail, I decided to use the list of the 10.000 brightest stars. For the position of the stars, the pre-filtered list already contains x,y,z coordinates relative to our sun. For the calculation of the color values, the list only provides the Gaia color index $G_{BP} - G_{RP}$, which represents the difference between the magnitudes measured by the blue photometer (BP) and the red photometer (RP). Initially, I tried to convert this magnitude to RGB magnitudes by applying the equations proposed by Cardiel et al. (2021) [Car+21]. In the end, I decided to use a different method because these RGB magnitudes can not be translated to RGB values [0,1] directly. In search of a good alternative, I found the algorithm by Tanner Helland [Hel] for converting tem-

perature values (in Kelvin) to RGB values [0,1]. To compute the effective temperature of the stars only from their $G_{BP} - G_{RP}$ indices, I use the equation introduced by Jordi et. al. (2010) [Jor+10]. This method is not perfect either, as the equation only works for $G_{BP} - G_{RP}$ indices smaller than 1.5. For indices greater or equal to 1.5, I use the same approach as Gaia Sky [SJMS19] by simply using linear interpolation to map the indices to effective temperatures.

- **Star Signs** - To simulate our night sky, we also wanted to visualize the star signs visible from Earth in three-dimensional space. It was not possible to extract the stars that make up our star signs from the Gaia data, such that we needed a second list of all the stars that make up the star signs and their positions relative to the sun. The list that was provided to me contains much more information about these stars, but all I use for this application is the position relative to the sun and a HIP value, which is a identifier used to associate the stars from this list with the specific star signs.

- **Connections** - This list contains all the connections that make up our star signs. Each connection consists of the two stars that are connected, identified by their HIP values, and the name of the corresponding star sign.

Furthermore, one of the other group members worked on generating textures for the application. All the textures I use in the Nightsky visualization were created in this process. Since they were not designed with any specific star, planet or any other celestial object in mind, I randomly assign them to my objects.

3.3 Rendering Pipeline

In the introduction in Chapter 1, I noted that I chose WebGL as the basis for this project. WebGL is a web based graphics API derived from OpenGL ES 2.0, optimized to be used in web browsers. The WebGL rendering pipeline handles everything required to display a three-dimensional scene on a two-dimensional screen. In order to achieve the level of portability, WebGL sacrifices some of the customization capabilities offered by OpenGL. This section will outline the different stages of the rendering pipeline and the differences to OpenGL. The first step in the pipeline is the vertex shader. Buffers, variables and uniforms are made available to the vertex shader via JavaScript. The vertex shader performs basic transformations for the individual vertices. It passes on interpolated attributes to the next stages in the

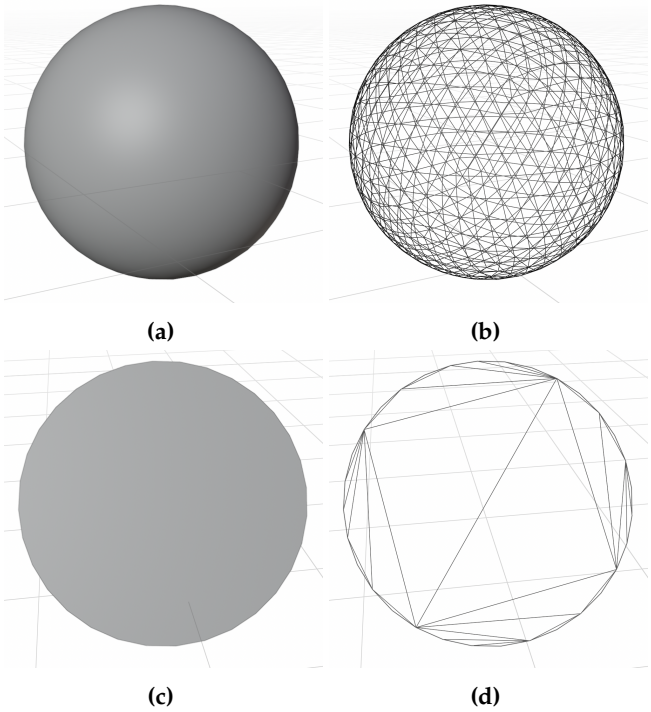


Figure 3.1: The models I used and their meshes: Sphere [(a) and (b)] and circle [(c) and (d)].

pipeline. Unlike OpenGL, the tessellation and geometry shaders in WebGL are not programmable. After assembling the vertices into primitives and the rasterization, which converts them into fragments, the next programmable stage in the pipeline is the fragment shader. Here, the individual fragments are processed and their color values computed. Per-fragment operations such as depth testing determine, which fragments should be written to the frame buffer. Finally, the pixel data is written to a canvas element in the browser. To conclude this section, the things I need to provide to the pipeline are the buffers, variables, uniforms, textures, vertex shaders and fragment shaders. Everything else is handled by WebGL itself.

3.4 Objects in the Scene

The first object I added to my application was the camera. The camera is positioned and rotated through the use of event listeners that keep track of the user's mouse and keyboard inputs. Its purpose is to provide and update the view matrix for the rendering pipeline according to the user's inputs.

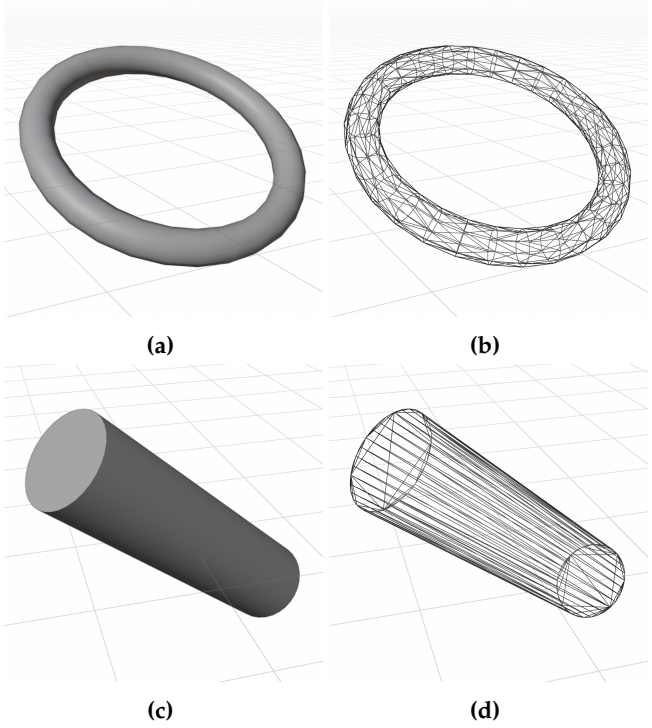


Figure 3.2: The models I used and their meshes: Torus [(a) and (b)] and cylinder [(c) and (d)].

In an empty scene, there is nothing for my camera to capture. The next step, of course, was to conceptualize the objects, that should be rendered in the visualization. To get used to WebGL, but also to create a reference point for potential users, I designed a miniature versions of our solar system in the center of the scene. Later, I added the option to travel between star systems, which are visualized in the same way. Since many of the objects in such a scene share common properties, I decided on using an object oriented approach:

- **CelestialBody** - First, I designed a base class that contains all the common attributes and methods for any object in the scene (e.g. position, scale, color, model, texture, world matrix, etc.).

The base class itself uses a class for handling the models:

- **Model** - In order to visualize anything in the context of WebGL, I needed some sort of triangulated mesh that can be rendered with the rendering

pipeline. To achieve good shading, it is also important that these models provide good normal vectors for the vertices of the meshes. I decided on pre-generating simple models and using those throughout the project. This saved me the pain of generating the meshes and normal vectors at runtime. For generating, I used blender [Ble], which provides not only meshes and normal vectors for smooth shading, but also texture coordinates that enable the use of textures. An instance of the *Model* class saves the list of vertices, normal vectors and texture coordinates for such a model and creates a WebGL vertex buffer, which is used later to rendering the model. Normally, I would need to load the model separately for each object that uses it. This class allows for one instance to be used by multiple *CelestialBody* instances at once. With this, a model only needs to be loaded once.
Visualizations of the models I use can be seen in Figure 3.1 and Figure 3.2.

Extending *CelestialBody*, there are five classes, each representing a specific kind of object:

- **Star** - This is the center piece of the scene. The *Star* class represents the star of the current star system (e.g. the sun of our solar system). Instances of this class are always positioned at the center (0,0,0) and are designed not to move. Therefore, this class is treated as a singleton, such that there always is only one instance in the scene. It serves two purposes. First, just like in a real star system, it is the center of the system and other objects orbit around it. More importantly, it serves as the light source of the simulation. For this, a *Star* instance has six camera objects, which are used to create a three-dimensional shadow map. This process is described in more detail in Section 3.5.2.
  Model - Instances of the *Star* class use a model of a sphere (Figure 3.1a and Figure 3.1b).

- **OrbitingObject** - This class represents objects that orbit around either the *Star* or an other instance of this class (e.g. the moon orbits the earth, which orbits the sun). Initially, I tired to implement a gravity system where the position of each object is computed by the masses and velocities of all objects. I quickly discarded this idea, as it turned out to be a numerical nightmare. Even if tuned right, the different objects would slowly drift apart over time. Instead, I used the polar coordinate system to move the instances in a circle around the objects they orbit. The cartesian coordinates are used to compute the angle

between the x-axis and the position of the instance relative to the origin of its orbit. The radius of the orbit is calculated by taking the distance from the instance to the orbited object. New cartesian coordinates are computed by increasing the calculated angle and translating the coordinates back from the polar coordinate space.

Let $\vec{p_{\text{orb}}} = \begin{bmatrix} x_{\text{orb}} \\ y_{\text{orb}} \\ z_{\text{orb}} \end{bmatrix}$ be the position of the orbited object. Further let $\vec{p_0} = \begin{bmatrix} x_0 \\ 0 \\ z_0 \end{bmatrix}$ be the starting position of the object relative to the orbited object. To convert these cartesian coordinates into the polar coordinate system relative to the orbited object, I compute the angle between x-axis and the position of the object:

$$\theta_0 = \text{arctan2}(z_0, x_0) = \begin{cases} \arctan(\frac{x_0}{z_0}) & \text{for } x_0 > 0 \\ \arctan(\frac{z_0}{x_0}) + \pi & \text{for } x_0 < 0, z_0 > 0 \\ \pi & \text{for } x_0 < 0, z_0 = 0 \\ \arctan(\frac{z_0}{x_0}) - \pi & \text{for } x_0 < 0, z_0 < 0 \\ \frac{\pi}{2} & \text{for } x_0 = 0, z_0 > 0 \\ -\frac{\pi}{2} & \text{for } x_0 = 0, z_0 < 0 \\ 0 & \text{else} \end{cases} \tag{3.1}$$

The next step is to update this angle by a angular velocity ω which is based on the objects orbit speed:

$\theta_1 = (\theta_0 + dt * \omega) \; mod \; (2 * \pi)$, where dt is the time past since the last update. (3.2)

Still missing for polar coordinates is the radius:

$$r = \sqrt{{x_0}^2 + {z_0}^2} \tag{3.3}$$

Finally, new cartesian coordinates can be computed using r and θ_1:

$$\vec{p_1} = \begin{bmatrix} x_1 \\ 0 \\ z_1 \end{bmatrix} = r * \begin{bmatrix} \cos(\theta_1) \\ 0 \\ \sin(\theta_1) \end{bmatrix} \tag{3.4}$$

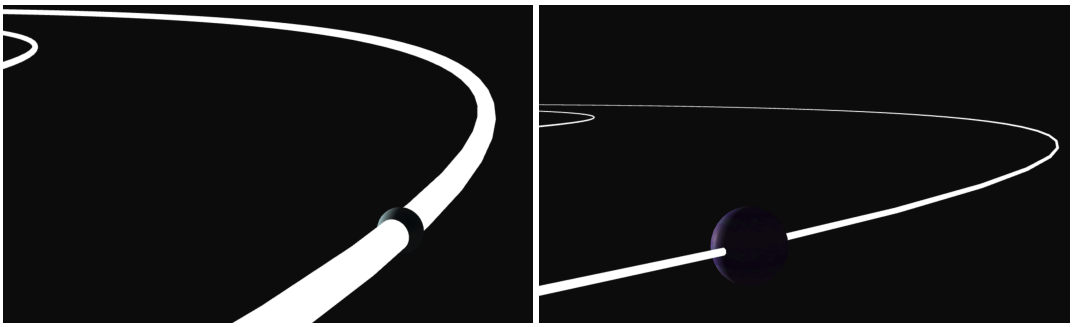
This method is fast but limits the position of the object to the x-z-plane. To enable full three-dimensional positioning, the rotation around the origin of the orbit is saved separately and applied to the world matrix before rendering.
Model - Instances of the *OrbitingObject* class also use the model of a sphere (Figure 3.1a and Figure 3.1b).

- **StaticOrbit** - This class represents the orbits themselves. It is designed to create more visual clarity by rendering the orbits of the *OrbitingObject* class.
  Model - Instances of the *StaticOrbit* class use a model of a torus (Figure 3.2a and Figure 3.2b) with a large outer radius and a small inner radius. The model is scaled and rotated to fit the orbit of the *OrbitingObject* instance. Unlike a sphere, scaling a torus creates a problem. Because the scale also affects the inner radius, larger orbits appear bulky. To prevent this, I added an alternative to the *StaticOrbit* class, by dynamically adding short cylinders around the orbit instead of using a single model. I will explain this together with the *Connector* class.

- **BackgroundStar** - This class represents the stars from the Gaia data release and the stars from the star sign list. They are positioned relative to the original position of the current star system (with our solar system at (0,0,0)).
  Model - I could have used the same sphere model, I used before, but, in an attempt to minimize the required vertices for more than 10.000 stars, I decided to use the model of a circle (Figure 3.1c and Figure 3.1d) instead. To still give the illusion of a full sphere, the circles are rotated before rendering, such that they always face the camera object. This is possible, because *BackgroundStar* objects are rendered without light and without textures, just using a solid color (In Section 3.5.2, I explain the render programs in more detail).

- **Connector** Each instance of this class represents a connection between two instances of the *BackgroundStar* class. It is positioned between the two stars and is used to visualize a connection for the star signs.
  Model - The model I used for the *Connector* class is a cylinder model (Figure 3.2c and Figure 3.2d). The model is scaled in length to fit the distance between the two stars and is rotated to face one of them.

  As mentioned before, I also use this class to dynamically generate orbits, overcoming the problem with the *StaticOrbit* class. In the process, placeholder *CelestialBody* objects are placed in a circle around the orbit. These



(a) (b)

Figure 3.3: The orbit of a *OrbitingObject* far away from its *Star* with the use of *StaticOrbit* (a) and dynamic orbit, using *Connector* objects (b).

placeholder objects are then connected using the *Connector* class. The more placeholders, the smaller the cylinders become in length, which results in a cleaner orbit but also increases the vertex count. After removing the placeholders, the orbit is complete, created from a collection of *Connector* objects. The resulting torus is customizable, as the inner radius can be set through scaling the radii of the individual cylinders. See Figure 3.3 for a comparison between a *StaticOrbit* object and the dynamic version, using *Connector* objects.

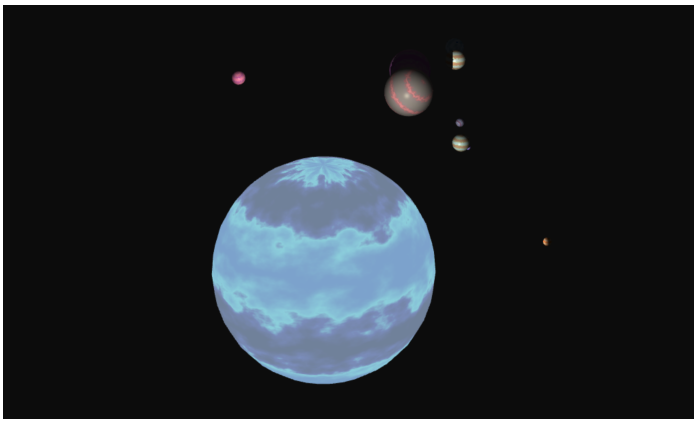
3.5 Scene Management

3.5.1 Scene Construction

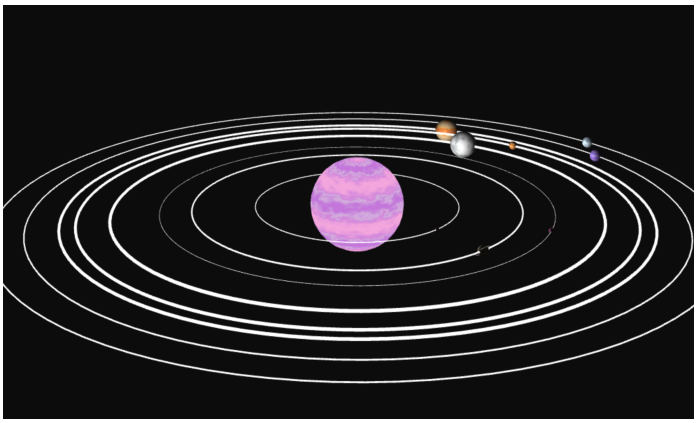
Now that I explained all the important components, I can finally talk about the scene itself. To build the scene for any star system, the application follows the same steps:

1. The star is added to the center of the scene.

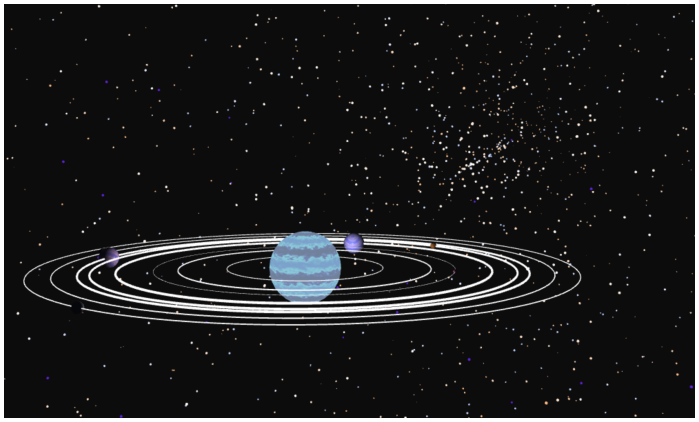
2. The objects that orbit the star are added.



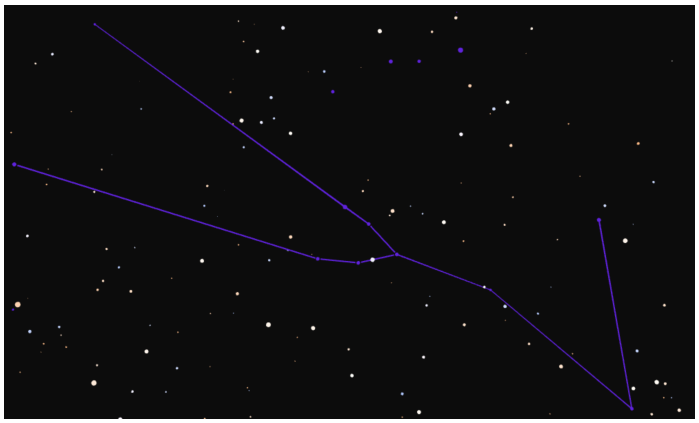
3. The orbits themselves are added.



4. The scene is filled with the stars in the background.



5. Visible star signs are added.



3.5.2 Rendering the Scene

The last step is to render the scene to the canvas. There are 4 shader programs used each frame to render the Nightsky scene. Most of the scene can be rendered using flat shading without light computations, as most objects are either stars in the background that are far away, or connections and orbits, which are artificial objects and therefore do not interact with any possible light source. The shader program used for flat shading is fairly basic.

The vertex shader computes the vertex position by transforming it from object space to clip space, using world, view and projection matrices. The world matrix depends on the object, which is rendered. It transforms the position of the object from a local object space to the world space of the scene. This includes scaling, rotation and translation. The view matrix stays consistent for each object but depends on the camera of the scene. Its purpose is to transform the position of the

vertices from the world space to the camera space, with the camera as the origin. Finally, the projection matrix transforms the vertices from the camera space to clip space, which is a normalized coordinate space where objects can be projected onto the screen. This matrix only depends on the camera's field of view, the aspect ratio of the canvas to which is rendered and the near and far clipping distances for the view frustum.

$$gl\_Position = M_{\text{Perspective}} * M_{\text{View}} * M_{\text{World}} * vertex\_Position \tag{3.5}$$

Additionally, the vertex shader passes the texture coordinates to the fragment shader. Together with a RGBA [0,1] color value, these coordinates are used to set the color of the fragment.

$$gl\_FragColor = \begin{cases} 0.5 * Texture\_Color + 0.5 * Flat\_Color, & \text{if a texture is used} \\ Flat\_Color, & \text{else} \end{cases} \tag{3.6}$$

See Figure 3.4 for an example of flat shading.

Instead of rendering everything with a flat color, I wanted to make the solar system stand out by using phong shading and shadow casting. Since there is just one star in the center, I only need to handle a single, stationary light source in the center of the scene. The vertex shader mostly stays the same, especially Equation 3.5. Additionally to providing the texture coordinates to the fragment shader, it also provides the vertex position and vertex normal in world space. The normal vector is transformed using the transpose of the inverse of the world matrix: $M_{Normal} = [M_{World}{}^{-1}]^T$. The fragment shader is more complex and needs more information. Ambient color is the base color of the rendered object, representing indirect light that bounces around the scene. This ensures that the object is visible even if the object is obscured by shadows. Diffuse color is the color of the object under direct exposure to light. It simulates matte lighting, where the surface reflects light equally in all directions. Specular color is the color of the shiny highlights caused by reflections on the surface. While ambient and diffuse colors are defined by the rendered object, I use the ambient color of the light source as the specular color. The phong exponent controls the sharpness of the specular highlights and therefore represents the shininess of the object's surface. Ambient, diffuse and specular coefficients control the intensity of these colors. For its computations, the fragment shader also needs the positions of the light source and the camera. To

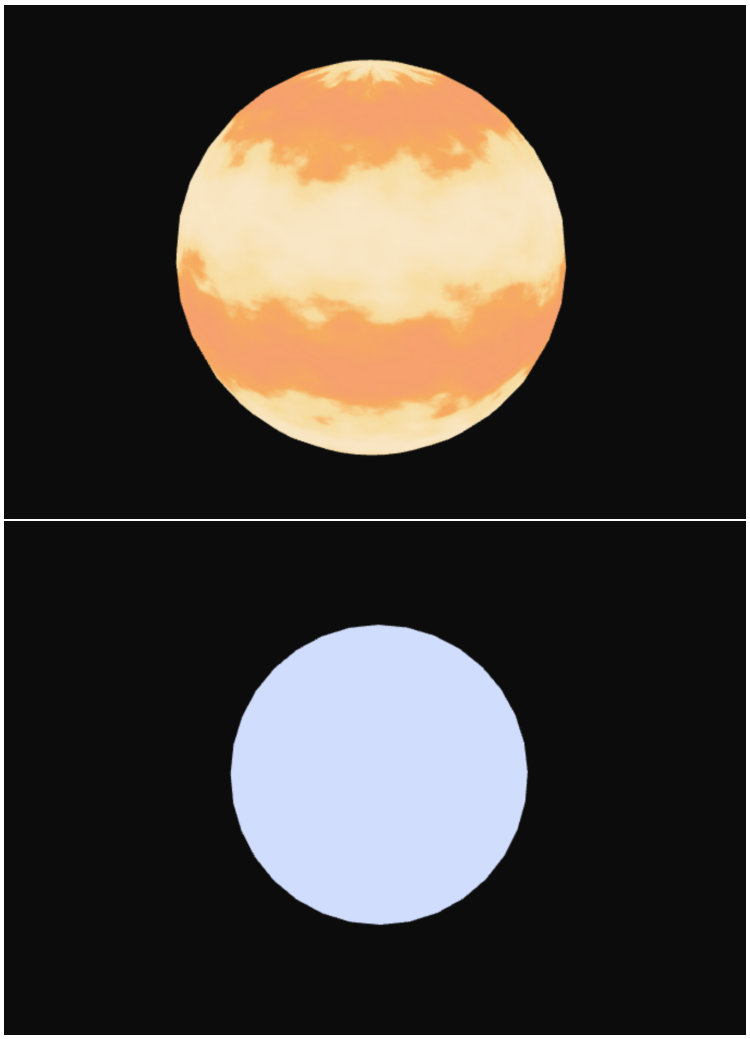


Figure 3.4: Two objects rendered using flat shading, one with and one without a texture.

check if the fragment is obscured by the shadow of another object, it also needs a cubemap texture with depth information. A cubemap texture is a texture that consists of six two-dimensional square textures. These images make up the six sides of a cube. Texture coordinates used to access the cubemap are three-dimensional vectors, representing the direction from the center of the cube. In our case, the six sides represent the six directions from the light source, such that the light source is in the center of the cube. Accessing the texture with a directional vector results in the normalized depth of the nearest object to the light source in this direction. The shader uses the following steps to compute the fragment color:

1. Normalize the transformed normal vector:

$$N = \frac{trans\_normal}{|trans\_normal|} \tag{3.7}$$

2. Compute and normalize the vector from the vertex to the light source:

$$L = \frac{light\_pos - vert\_pos}{|light\_pos - vert\_pos|} \tag{3.8}$$

3. Compute the Lambertian reflectance and ensure it is non-negative:

$$l = \max(N \cdot L, 0) \tag{3.9}$$

   The Lambertian reflectance is the property that defines an ideal diffusely reflecting surface. If this value is zero, the light is coming from behind the surface of the object, in which case the specular contribution to the fragment color is zero.

4. Compute the reflection vector of the light around the normal:

$$R = 2 * (L \cdot N) * N - L \tag{3.10}$$

5. Compute and normalize the vector from the vertex to the camera:

$$V = \frac{cam\_pos - vert\_pos}{|cam\_pos - vert\_pos|} \tag{3.11}$$

6. Compute the angle between the reflected light and the viewing direction and ensure it is non-negative. With this angle, compute the specular component:

$$\alpha = \max(R \cdot V, 0) \tag{3.12}$$

$$specular = \alpha^{phong\_exp} \tag{3.13}$$

7. Compute the depth from the light source to the vertex and normalize it, such that it is a value between 0 and 1:

$$depth = \frac{|vert\_pos - light\_pos| - near\_clip}{far\_clip - near\_clip} \tag{3.14}$$

$near\_clip$ and far_clip are the clipping distances of the cameras used to create the different sides of the cubemap. Consequently, the depth values of the cubemap start with 0 at $near\_clip$ and end with 1 at far_clip.

8. Get the depth value in the direction the light is going. If this value (plus some bias to counteract planefighting) is at least as big as the calculated depth of the vertex, light directly hits the fragment. The last step is to put everything together to get the fragment color:

$$Amb\_Color = \begin{cases} 0.5 * Texture\_Color + 0.5 * Amb\_Color, & \text{if a texture is used} \\ Amb\_Color, & \text{else} \end{cases} \tag{3.15}$$

$$Dif\_Color = \begin{cases} 0.5 * Texture\_Color + 0.5 * Dif\_Color, & \text{if a texture is used} \\ Dif\_Color, & \text{else} \end{cases} \tag{3.16}$$

$$gl\_FragColor = \begin{cases} k_A * Amb\_Color \\ +k_D * l * Dif\_Color \\ +k_S * specular * Spe\_Color, & \text{if the fragment is illuminated} \\ k_A * Amb\_Color, & \text{else} \end{cases} \tag{3.17}$$

k_A, k_D and k_S are the coefficients.

See Figure 3.5 for an example of phong shading.

How is the cubemap texture for depth testing generated? There are six sides of the cube that represent the different directions from the star in the center and

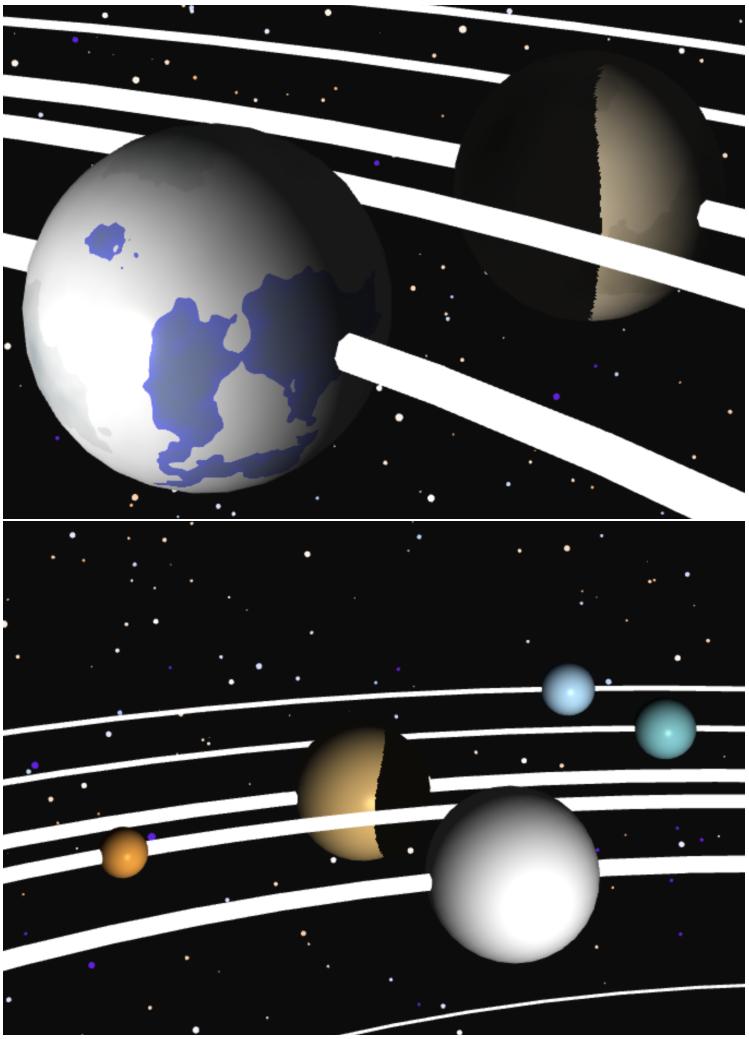


Figure 3.5: Objects rendered using phong shading, with and without textures.

need to be rendered individually. For this, the star object has six cameras, covering the different directions. The third shader program is used six times, once for each camera, but instead of rendering to the canvas, it renders one of the cube faces. The vertex shader again stays mostly the same, with the view matrices being determined by the cameras of the star and providing the fragment shader with vertex position in world space. The fragment shader takes the vertex position, light position and clipping distances of the camera used. It computes the normalized depth values and sets the fragment color to this value.

$$depth = \frac{|vert\_pos - light\_pos| - near\_clip}{far\_clip - near\_clip} \tag{3.18}$$

$$gl\_FragColor = rgba(depth, depth, depth, 1) \tag{3.19}$$

The last shader program enables interactions with the scene. It renders objects that should be clickable with an unique id as the fragment color. This is the simplest of the four programs. It sets the vertex position as usual and sets the provided id as the fragment color. After rendering all the clickable objects, it is possible to identify the object underneath the mouse cursor by checking the color of the pixel underneath. However, each color channel of the rgba [0,1] value only covers a small range of unique numbers. In order to guarantee a id for every object in the scene, the id is split cross all four color channels, one byte per channel:

$$\begin{aligned} red &= (id \gg 0)\ \&\ 0xFF \\ green &= (id \gg 8)\ \&\ 0xFF \\ blue &= (id \gg 16)\ \&\ 0xFF \\ alpha &= (id \gg 32)\ \&\ 0xFF \\ gl\_FragColor &= rgba(\frac{red}{255}, \frac{green}{255}, \frac{blue}{255}, \frac{alpha}{255}) \end{aligned} \tag{3.20}$$

These four programs are used each frame for generating the visualization. Each frame follows the same order:

1. Compute $dt = current\_frametime - previous\_frametime$ and use it to update the camera and all the updatable objects in the scene.
2. As it is computationally expensive to render all stars in the background, the application filters out the stars that are not visible to the camera.
3. Use the objects that make up the current star system to generate the cubemap.
4. Render all clickable objects and check if one is selected at the moment.
5. The final step is to render both flat and shaded objects with the respective shader program.
6. Repeat for the next frame.

4 Results

In Chapter 3 I discussed the methods I used to create the Nightsky application. Here in Chapter 4, I want to visualize the results of the project. As the project is completely build on JavaScript, potential users do not need any special software. Once deployed as a static webpage, users only need a browser with WebGL support (pretty much any modern browser), keyboard and mouse. I deployed Nightsky at `https://thorsten-trinkaus.github.io/Nightsky/` using GitHub Pages [Git].

4.1 User Interface

The User Interface (UI) of the Nightsky application not only conveys useful information like the controls (Figure 4.1) or information about the object currently selected by the mouse cursor (Figure 4.2), but also provides a simple way to interact with some of the objects in the scene.

4.1.1 Controlling the Star System

As mentioned before in Chapter 3, there is a simulated star system in the center of the visualization. The UI allows the user to control this simulation. Users can pause (or unpause) the scene by pressing a button in the UI (Figure 4.1). A slider (also Figure 4.1) can be used to control the orbit speed of the *OrbitingObject* instances.

4.1.2 Controlling Star Signs

Upon starting the application, there are no visible star signs. The user can choose the star signs that should be displayed. For this, the UI (Figure 4.3 and Figure 4.4) provides a interactive list of all the star signs available to be visualized. Through checkboxes, the user decides which star signs to show and which to hide. Nightsky also allows users to draw their own star signs. For this, the UI (Figure 4.3) provides some options. First, there is a color picker, which enables colorful star sign designs. Secondly, the UI makes it possible for the user to export and import their custom

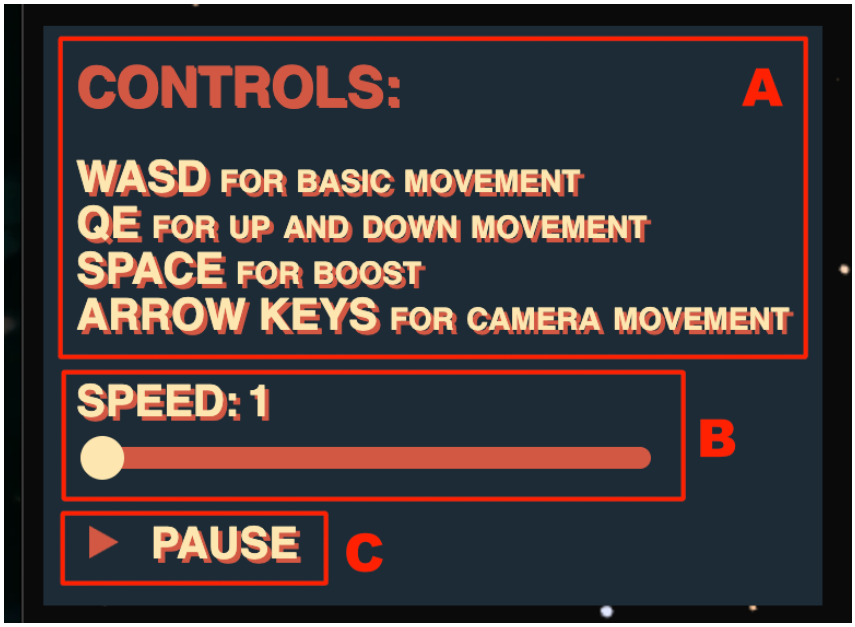


Figure 4.1: First UI panel - **A** Description for the controls - **B** Speed control for the star system - **C** (Un)Pause button for the star system

star signs to share the designs with others. A button exports the current state by copying it as a string to the clipboard of the user. To import star signs, the user needs to input this string and press the import button.

4.2 Interactions with Nightsky

Of course, the user is not limited to only using the User Interface. This section explores the possible ways to interact directly with the scene.

4.2.1 Traversing Space

After loading all the needed resources, the user is thrown right into our solar system. From there, the user can traverse the three-dimensional space by using keyboard inputs: *WASD*-Keys can be used for general movement, while the *QE*-Keys move the camera up and down. Rotations of the camera are possible, either through the *ARROW*-Keys or by holding down the left mouse button and moving the cursor around. As the distances between the objects in the scene can get quite large, the movement is accelerated while holding down the *SPACE*-Key.

Figure 4.2: Second UI panel - This panel provides some extra information about the selected object. The information depends on the object itself.

Figure 4.3: Third UI panel - A Input to import star signs - B Button to export star signs drawn by the user - C Color-Picker for the drawn star signs - D List of all available star signs

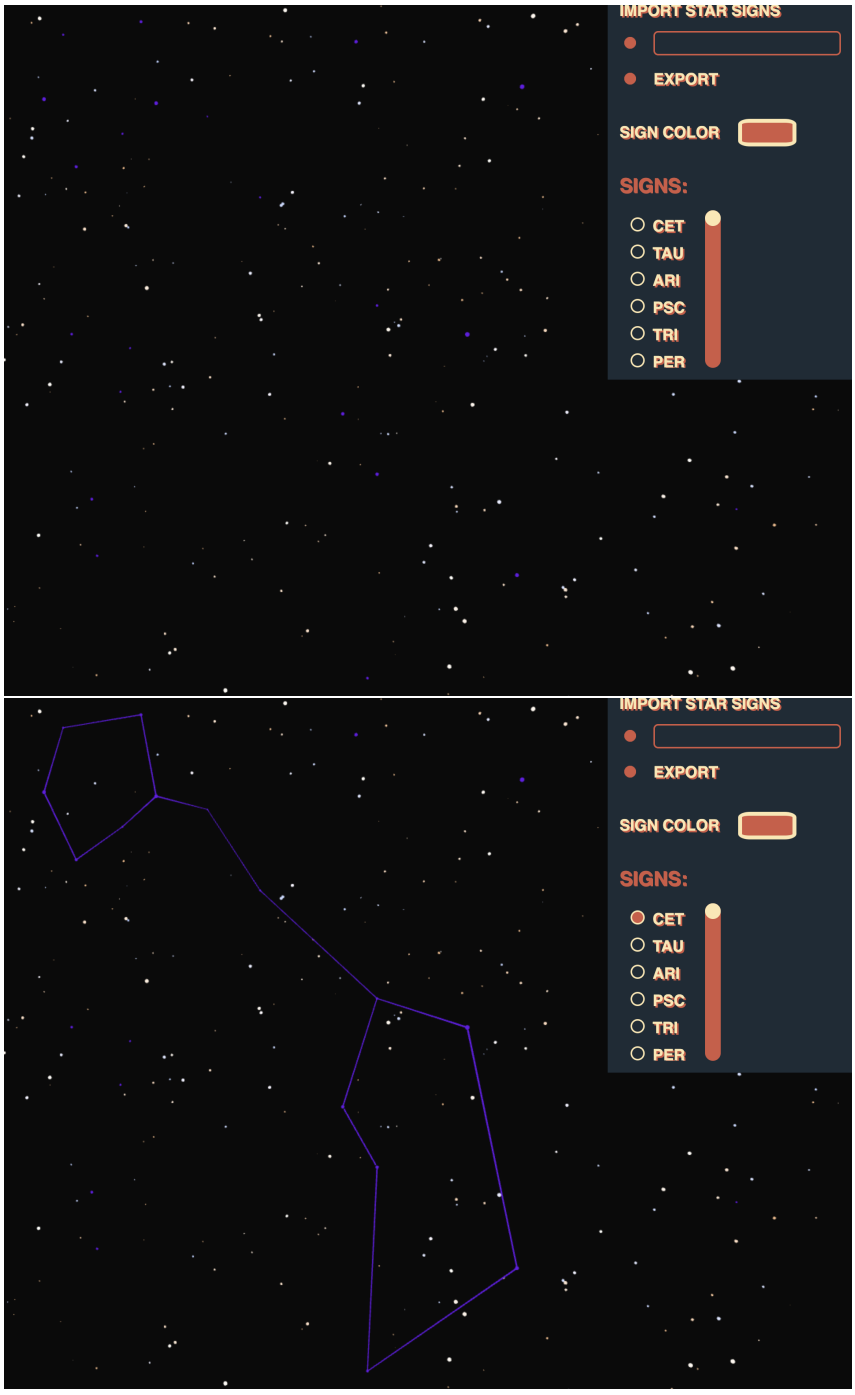


Figure 4.4: Example for selecting a star sign.

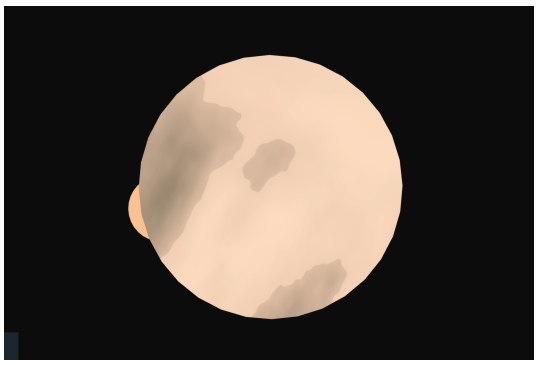
4.2.2 Travel between Star Systems

How would a specific star sign look from a different star system? To visualize this perspective, the user can select a star (with the mouse cursor) and press the *P*-Key. The camera will automatically zoom into the selected star and the scene will be rebuild from the perspective of the selected star. After this process is complete, the camera zooms out again. The *OrbitingObject* instances in the scene are generated at random, as the Gaia dataset does not provide information about orbits around the stars. The scene should still display all the visible star signs, but now from a new perspective. See Figure 4.5 and Figure 4.6 for an example.

4.2.3 Drawing Star Signs

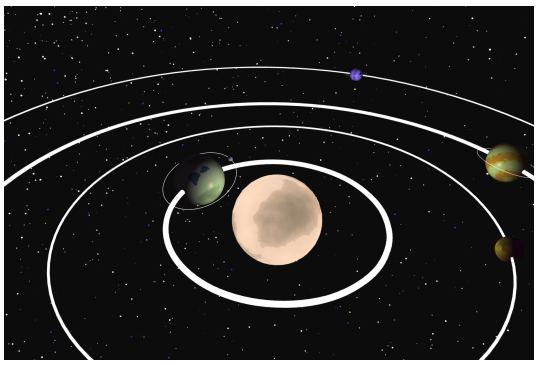
The application is not limited to showing established star signs. Users can draw their own custom star signs and look at them in three-dimensional space. A connection between two stars can be added by first selecting one of the stars and pressing the middle mouse button and then doing the same for the second star. The color of the new connector is defined by the current value of the color-picker. Added a wrong connection? Connectors drawn by the user can be removed by selecting the connector itself and pressing the *P*-Key. See Figure 4.7 for an example. In order to import another example directly into the application, this string can be used: */6753/2878/00ff9d/6311/4511/ff0059/3104/3263/002aff/4730/2849/ffc800*

(a)



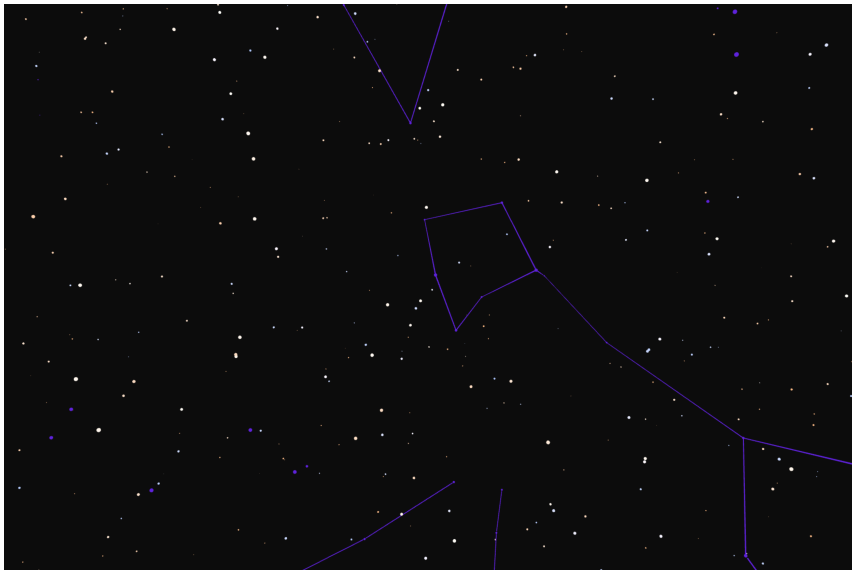
(b)

(c)



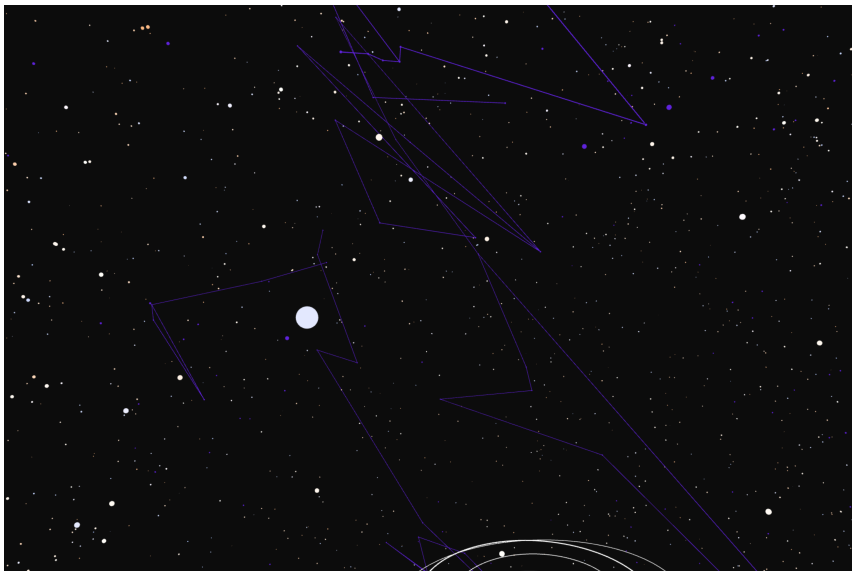
(d)

Figure 4.5: Process of swapping star systems. (a) - (c) shows the camera zooming in, while (d) shows the new star system.



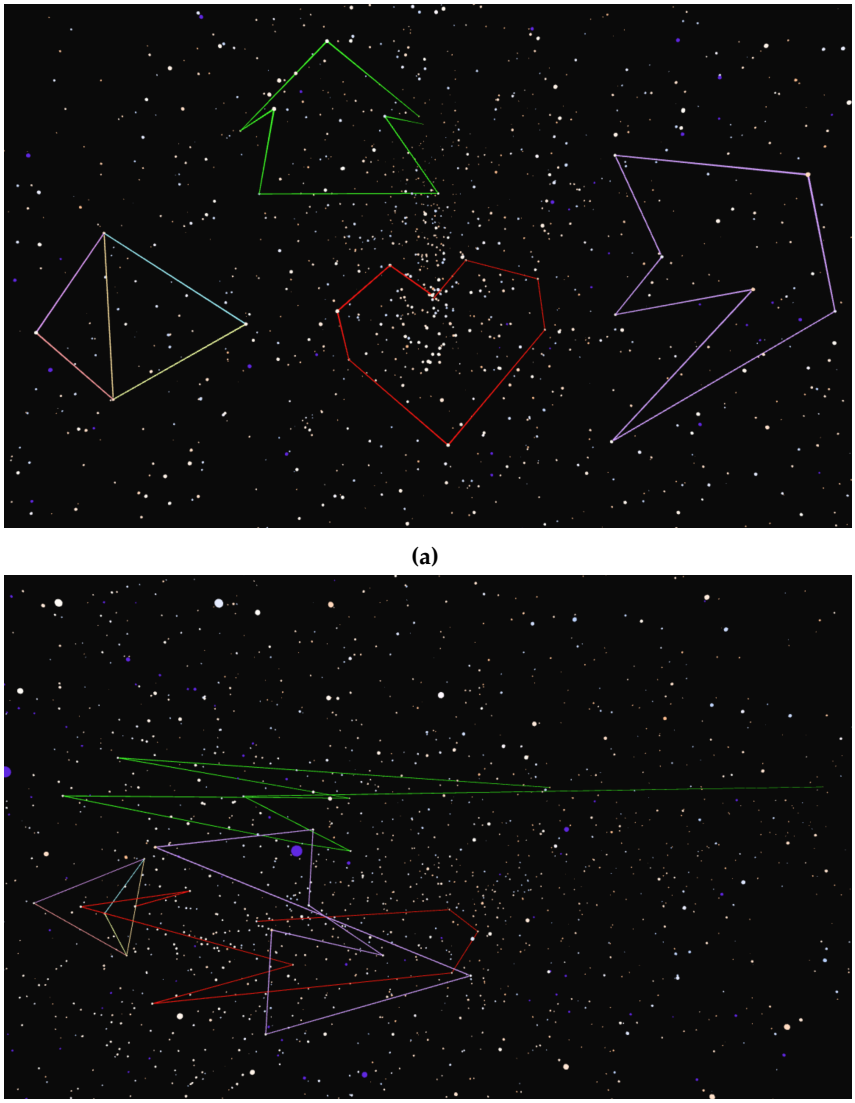
(a)

(b)



(c)

Figure 4.6: Star signs from different perspectives: (a) from our sun, (b) from a different point in our solar system, (c) from a different star system.



(a)

(b)

Figure 4.7: Examples for custom star signs.

5 Conclusion

In conclusion, I created a application, which visualizes not only the Gaia data release, but also our star signs in three-dimensional space. Users are able to traverse this scene and interact with the objects inside it. The feature to draw new star signs adds a reusability factor to the project. My application is not as detailed as other visualizations done before, but my solution is very lightweight, which makes it super portable, as this project will run in almost every browser. The object oriented approach makes it easy to build scenes different from the scenes I use in my application, which is important, because this project was further adapted and improved by other members of the group project.
At the end, I want to talk about some things I think could still be added to my framework. To get more visual depths, a bloom effect could be added, symbolizing the light emitted by all the stars in the scene. More realism could also be achieved through the use of ellipses instead of circles for the orbits. This would need some sort of implementation of Kepler's laws. Also, this project is limited by the prefiltered Gaia data release. A better version could improve the visualization, as the current version suffers from reddening of the stars further away from the sun.

6 Acknowledgement

This work has made use of data from the European Space Agency (ESA) mission *Gaia* (https://www.cosmos.esa.int/gaia), processed by the *Gaia* Data Processing and Analysis Consortium (DPAC, https://www.cosmos.esa.int/web/gaia/dpac/consortium). Funding for the DPAC has been provided by national institutions, in particular the institutions participating in the *Gaia* Multilateral Agreement.

For more information about the *Gaia* mission and the data it collected, see Gaia Collaboration et al. [Gai+16] [Gai+23] [Gai+21] or Babusiaux et al. [Bab+23].

7 Apendix

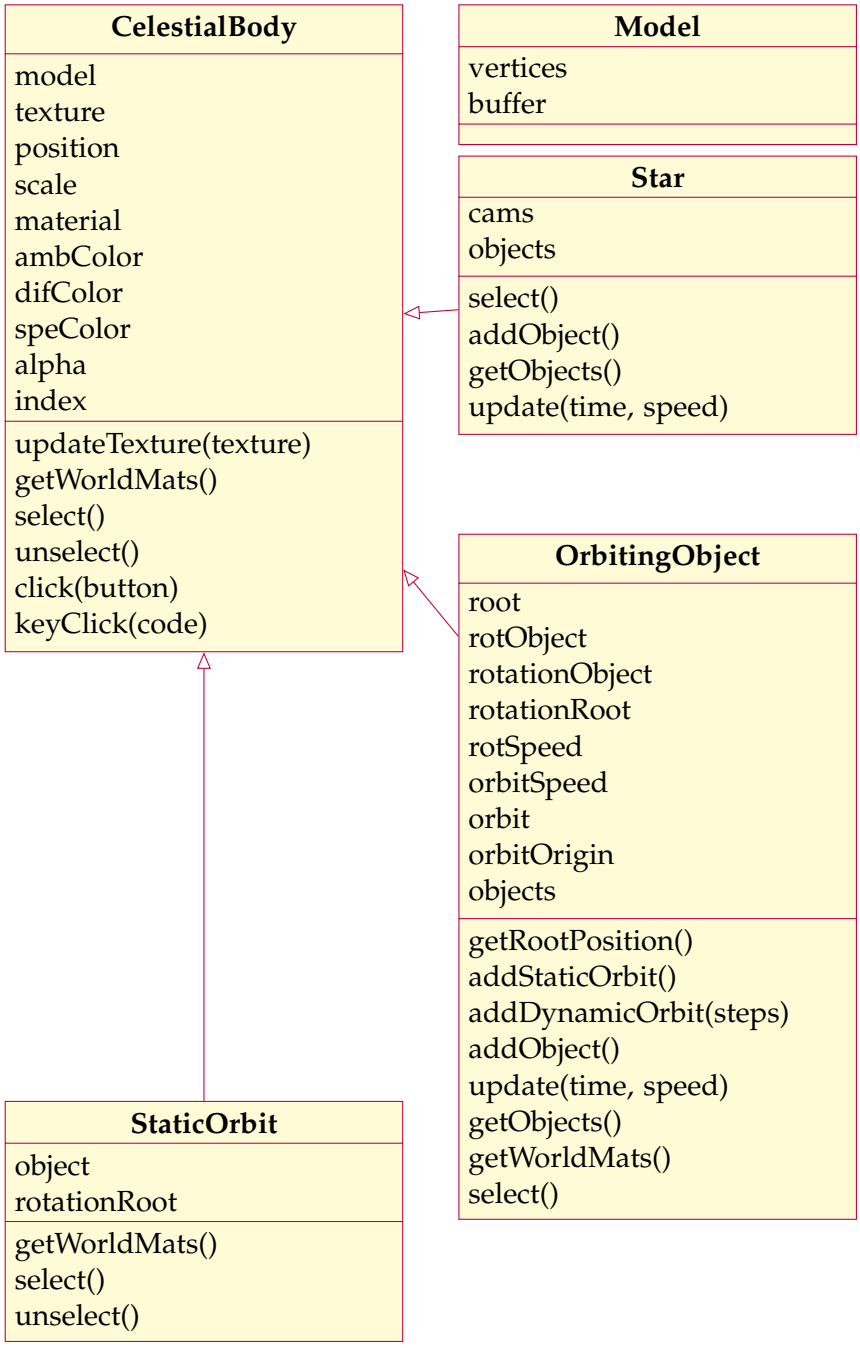


Figure 7.1: UML class diagrams part 1.

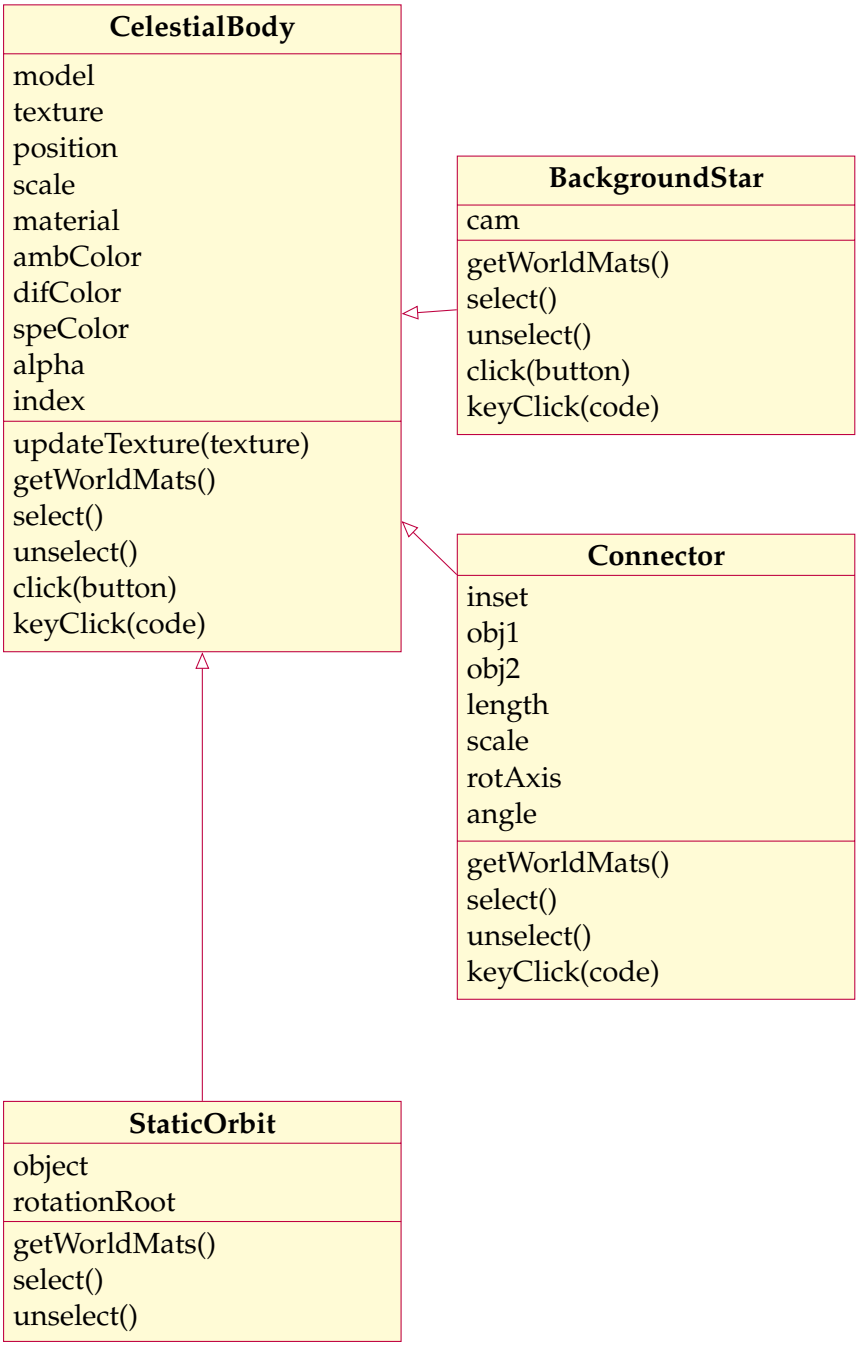


Figure 7.2: UML class diagrams part 2

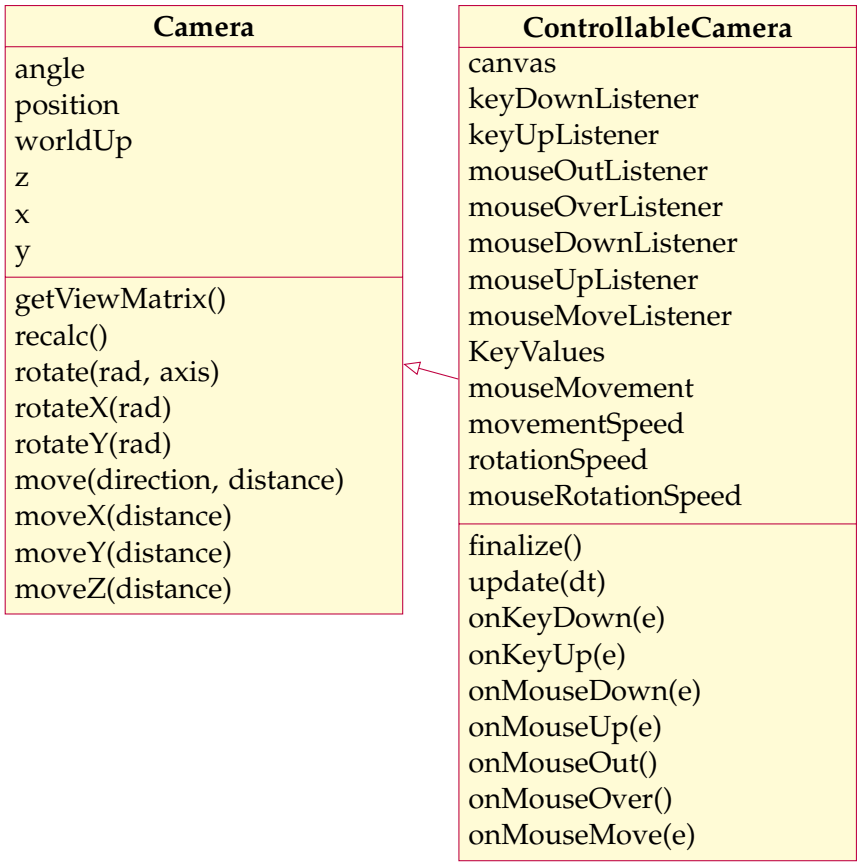


Figure 7.3: UML class diagrams part 3

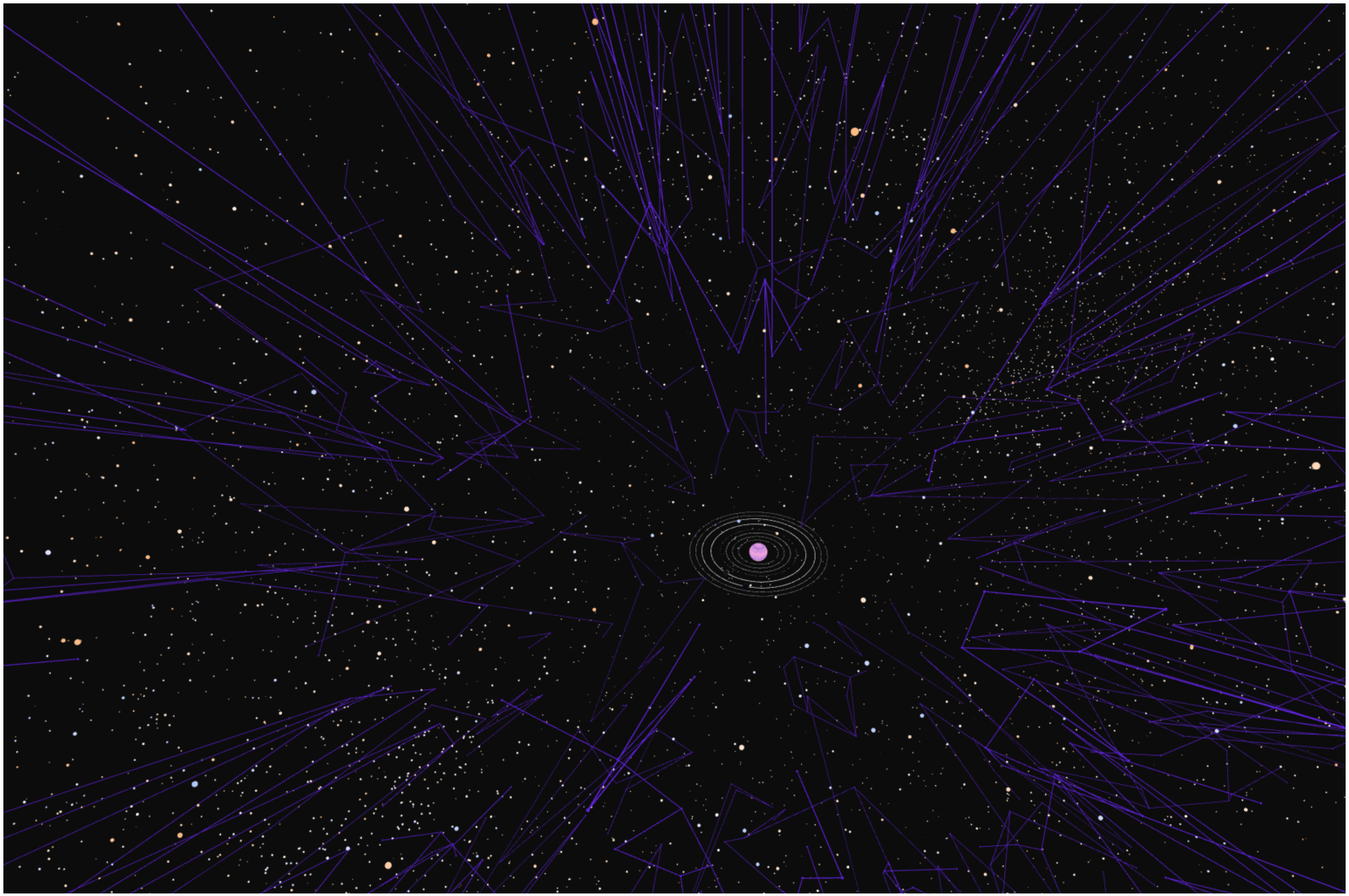


Figure 7.4: A picture of all the star signs, visible from outside our solar system.

Bibliography

[Bab+23] C. Babusiaux, C. Fabricius, S. Khanna, T. Muraveva, C. Reylé, F. Spoto, A. Vallenari, X. Luri, F. Arenou, M. A. Álvarez, F. Anders, T. Antoja, E. Balbinot, C. Barache, N. Bauchet, D. Bossini, D. Busonero, T. Cantat-Gaudin, J. M. Carrasco, C. Dafonte, S. Diakité, F. Figueras, A. Garcia-Gutierrez, A. Garofalo, A. Helmi, Ó. Jiménez-Arranz, C. Jordi, P. Kervella, Z. Kostrzewa-Rutkowska, N. Leclerc, E. Licata, M. Manteiga, A. Masip, M. Monguió, P. Ramos, N. Robichon, A. C. Robin, M. Romero-Gómez, A. Sáez, R. Santoveña, L. Spina, G. Torralba Elipe, and M. Weiler. "Gaia Data Release 3. Catalogue validation". *A&A* 674, A32, 2023, A32. arXiv: 2206.05989 [astro-ph.SR] (cit. on p. 35).

[Ble] Blender. *blender.org - Home of the Blender project - Free and Open 3D Creation Software — blender.org*. https://www.blender.org/. [Accessed 07-12-2024] (cit. on p. 9).

[Car+21] N. Cardiel, J. Zamorano, J. M. Carrasco, E. Masana, S. Bará, R. González, J. Izquierdo, S. Pascual, and A. Sánchez de Miguel. "RGB photometric calibration of 15 million Gaia stars". *Monthly Notices of the Royal Astronomical Society* 507:1, 2021, pp. 318–329. ISSN: 0035-8711. eprint: https://academic.oup.com/mnras/article-pdf/507/1/318/39767196/stab2124.pdf. URL: https://doi.org/10.1093/mnras/stab2124 (cit. on p. 5).

[Gai+16] Gaia Collaboration et al. "The Gaia mission". *A&A* 595, A1, 2016, A1. arXiv: 1609.04153 [astro-ph.IM] (cit. on p. 35).

[Gai+21] Gaia Collaboration et al. "Gaia Early Data Release 3. Summary of the contents and survey properties". *A&A* 649, A1, 2021, A1. arXiv: 2012.01533 [astro-ph.GA] (cit. on p. 35).

[Gai+23] Gaia Collaboration et al. "Gaia Data Release 3. Summary of the content and survey properties". *A&A* 674, A1, 2023, A1. arXiv: 2208.00211 [astro-ph.GA] (cit. on p. 35).

[Git] GitHub. *Creating a GitHub Pages site - GitHub Docs — docs.github.com*. https://docs.github.com/en/pages/getting-started-with-github-pages/creating-a-github-pages-site. [Accessed 21-11-2024] (cit. on p. 23).

[Hel] T. Helland. *How to Convert Temperature (K) to RGB: Algorithm and Sample Code — tannerhelland.com*. https://tannerhelland.com/2012/09/18/convert-temperature-rgb-algorithm-code.html. [Accessed 18-11-2024] (cit. on p. 5).

[JM] B. Jones and C. MacKenzie. *Toji/GL-Matrix: Javascript matrix and Vector Library for high performance webgl apps*. URL: https://github.com/toji/gl-matrix (cit. on p. 5).

[Jor+10] C. Jordi, M. Gebran, J. M. Carrasco, J. de Bruijne, H. Voss, C. Fabricius, J. Knude, A. Vallenari, R. Kohley, and A. Mora. "Gaiabroad band photometry". *Astronomy & Astrophysics* 523, 2010, A48. ISSN: 1432-0746. URL: http://dx.doi.org/10.1051/0004-6361/201015441 (cit. on p. 6).

[SJMS19] A. Sagristà, S. Jordan, T. Müller, and F. Sadlo. "Gaia Sky: Navigating the Gaia Catalog". *IEEE Transactions on Visualization and Computer Graphics* 25:1, 2019, pp. 1070–1079 (cit. on pp. 3, 6).