LiU-ITN-TEK-A--20/045-SE

A System for Procedural Camera Movements for Navigation in Astrographics

Emma Broman

Ingela Rossing

2020-08-20

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--20/045-SE

A System for Procedural Camera Movements for Navigation in Astrographics

The thesis work carried out in Medieteknik
at Tekniska högskolan at
Linköpings universitet

Emma Broman
Ingela Rossing

Norrköping 2020-08-20

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

Linköping University | Department of Science and Technology
Master's thesis, 30 ECTS | Media Technology
202020 | LIU-ITN/LITH-EX-A--2020/045--SE

A System for Procedural Camera Movements for Navigation in Astrographics

Ett system för procedurell generering av kamerarörelser för navigering inom astrografik

Emma Broman
Ingela Rossing

Supervisor : Lovisa Hassler, Emil Axelsson
Examiner : Alexander Bock

External supervisor : Claudio Silva

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

This master thesis covers the automatic generation of camera motions for simplified navigation in OpenSpace, an open-source astrovisualization software. The project was conducted at the Tandon School of Engineering at New York University in Brooklyn, New York, in collaboration with the American Museum of Natural History.

The start phase of the project consisted of research and analysis of the characteristics of appropriate paths in OpenSpace as well as the needs of current OpenSpace pilots and users. The resulting characteristics were then translated to use cases and properties for the automatic camera flight system. Based on these use cases two sub-systems were developed: one for specifying details on the path to be created and one that creates a path from that specification. The paths are created using spline mathematics, where the control points of the spline are generated based on information about the scene and target objects. The spline curve is used for defining algorithms for controlling the speed and orientation interpolation along the path. Two example curve types were implemented, with different strategies for rotation interpolation and spline creation. The first curve is based on the assumption that the path is to be created between two visual targets in scene. It creates a zooming out motion that is especially useful when for example moving between planets in the solar system. The second curve type is more general and has the main focus of avoiding collision with objects in the scene. This makes it useful for creating paths to targets that are close to other objects, such as spacecraft orbiting a planet. The system can later be extended with more alternative types.

The result is a system that is capable of interpreting a specification from a user, creating a path based on that specification and moving the camera along that path. Instructions are provided through the system in the form of a list of targets that can be specified with varying levels of detail. The resulting system provides a good foundation for automatic camera motions in OpenSpace. Some work remains to solve challenges related to the scale and sparsity of the environment. There is also room for future extensions and improvements, especially regarding the control of the speed of the camera.

Acknowledgments

This project was possible thanks to the international collaboration between the OpenSpace partners, for which we are endlessly grateful. A special thanks to Anders Ynnerman from Linköping University and and Claudio Silva from New York University who have created the opportunity of our internship.

Special thanks also has to be given to everyone supporting us during the project. A big thanks to Jonathas Costa from New York University and Micah Acinapura and Carter Emmart from the Amercian Museum of Natural History. Your support on site, experience and opinions have been extremely valuable to us throughout the project. The same goes for our supervisors Emil Axelsson and Lovisa Hassler and our examiner Alexander Bock, who have provided support from our university in Sweden. Thank you for all your help, valuable feedback throughout the project and for many interesting meetings and discussions.

We would also like to thank everyone at VIDA at NYU, where we have been welcomed warmly by the whole department. Thank you all for making our stay in New York full of joy. Lastly we would like to thank Ann Borray who has helped us with everything from administration to plants and never failed to leave us with smiles on our faces.

Ingela Rossing and Emma Broman

Contents

Abstract iii

Acknowledgments iv

Contents v

1 Introduction 1
- 1.1 OpenSpace . . . 1
- 1.2 Motivation . . . 1
- 1.3 Aim . . . 2
- 1.4 Research Questions . . . 2
- 1.5 Delimitations . . . 3

2 Related Work 4
- 2.1 Navigation in OpenSpace . . . 4
- 2.2 Scale Difference and Precision . . . 5
- 2.3 Other Systems for Astrographics and Visualization . . . 6
- 2.4 Automated Camera Control and Motion Planning . . . 7
- 2.5 Splines for Path Planning and Motion Control . . . 7

3 Theory 9
- 3.1 Camera Mathematics . . . 9
- 3.2 Interpolation and Splines . . . 11

4 Defining Appropriate Camera Motions 15
- 4.1 Freedom . . . 15
- 4.2 Motion Sickness . . . 15
- 4.3 Understanding Motion in the Universe . . . 18

5 Use Cases 20
- 5.1 Specifying Motion Details . . . 20
- 5.2 Camera Behavior at a Target . . . 21

6 System Implementation 22
- 6.1 Path Specification . . . 22
- 6.2 Motion Along the Path . . . 25
- 6.3 Searching the Scene Graph . . . 29
- 6.4 Rendering the Path . . . 29

7 Path Generation 30
- 7.1 Generating a Spline Curve . . . 30
- 7.2 Camera Orientation Interpolation . . . 32

8 Results 35

8.1 Providing Input 35
8.2 Curve Type - Zoom Out Overview 36
8.3 Curve Type - Avoid Collision 37
8.4 Further on the Look-at Orientation Interpolation 38

9 Discussion **40**
9.1 Results 40
9.2 Method 42

10 Conclusion **46**
10.1 Future Work 48

Bibliography **49**

1 Introduction

This thesis project is part of the development of the open-source astrovisualization software OpenSpace [1, 2]. The goal is to extend the current navigation system with automatic camera motions for travel between destinations in the scene.

1.1 OpenSpace

OpenSpace uses data from NASA and other public sources to provide an interactive 3D visualization of the entire known universe and allows for real-time exploration of data from observations, simulations and space mission planning and operations. The software is being developed as a collaboration between NASA Goddard's Community Coordinated Modeling Center, the American Museum of Natural History, Linköping University, Tandon School of Engineering at New York University and the Scientific Computing and Imaging Institute at the University of Utah.

OpenSpace is implemented using C++17 and OpenGL 3.3 and above and is designed to be run on several different platforms, ranging from tablets to more immersive applications like projected domes. The software is currently being used both as a tool in research as well as for exhibitions and presentations about space missions and the cosmos for the general public. The mission of OpenSpace and use cases for astrographics visualization are further discussed by Bock et al. [1, 2].

1.2 Motivation

Navigation in virtual environment applications is a widely researched subject and a range of different approaches for camera control have been used in applications such as video games and visualization. In many cases, the navigation is done with some degree of interaction from the user, by mapping the parameters of the camera (e.g. position, orientation) to the dimensions of some input device. This sort of direct control can result in erratic and unnatural movements if the navigation is done by an inexperienced user, which can potentially induce motion sickness or disorientation for the observer. The risk is increased further when OpenSpace is displayed in an immersive environment, where the observer has the sense of being present inside the virtual world. The continuity and smoothness of the camera motion in terms of translation, rotation and speed, are important factors to preserve the sense of immersion and avoid motion sickness.

An on-going challenge is to automatically control a virtual camera, without input from a user. Existing approaches to compute camera paths are often inspired by methods from the field of robotics, which often rely on simulations and AI. Aspects related to cinematics and observer experience must be taken into account, since the criteria on a camera path are more complex than those of the motion of a robot. Within astrographics, there is also the challenge that comes with the large scale differences and distances that the camera must traverse. For

example, we want to be able to seamlessly navigate from the outer edges of the universe to the surface of a specific planet. How to do this automatically and also create an interesting and pleasing experience for the observer in an environment as sparse as space, is a problem that remains to be solved.

A system for automatically generating camera movements would be of great assistance to a person trying to navigate the scene in OpenSpace. Currently, the user has to manually navigate using an input device, such as a mouse and keyboard. The navigation scheme of OpenSpace prevents the user from getting lost when traveling in the universe. The camera movement is limited to a combination of orbital rotations around a chosen target, like a planet or spacecraft, and a linear motion towards or away from this target. This makes it difficult to create interesting and visually pleasing motions, something which is essential when creating content to be shown to the public. Preparing these shows is also time consuming and currently mainly performed by experts. If navigation could be done with the click of a button it would enable for less experienced user to prepare and host shows. It would also extend the possibilities for running the software in varying applications on different displays. For example, when viewing OpenSpace through a head-mounted display (HMD) it is currently close to impossible to navigate, since the web-based GUI does not work for these displays. This would no longer be an issue if a camera path could be predefined and started by pressing a button.

OpenSpace is dependent on a high frame rate to run high resolution stereoscopic rendering in real time. Maintaining a high frame rate also allows for the software to be run on less powerful computers. This is an aspiration of the OpenSpace developers, since it makes the software available for a larger group of users.

1.3 Aim

The aim of the project is to create a system for generating camera paths for navigation in OpenSpace using spline mathematics. The user shall be able to provide instructions for the path, which will then be used to compute a suitable path for the camera. As an example, a possible set of instructions might be: "Fly me to the Moon in ten seconds, then go to Jupiter and stop for 5 seconds and lastly go to Mars".

An objective is to simplify navigation for exploration in OpenSpace as an aid for the pilot during presentations held in the software, as well as when recording movies. The automated paths will hopefully provide enough support for less experienced pilots to agree to hosting space shows. The solution should also provide enough flexibility to support the ambitions of experienced navigators and presenters. Furthermore, the automatic motions can reduce visual artifacts that can occur during manual camera control, like erratic movements or misnavigation and relieve both novice and experienced pilots in their task.

1.4 Research Questions

Apart from creating the paths, part of the challenge also lies in how to define the specifications for the motion. This should be done in a way that provides enough flexibility for the user, while still being simple enough for inexperienced users. With this in mind, the following research questions have been specified for this project:

1. What aspects characterizes a good path, with respect to the experience of the observer?
2. How can spline mathematics be used to automatically generate a camera path that provides a cinematic experience for the viewer? What challenges does the large scale of the virtual space environment entail?

3. How can the instructions on the desired path be specified? What degree of freedom should the navigator have when specifying these instructions?

An observer is a person viewing the application through some sort of display, hence experiencing the camera motion. The navigator, or the pilot, is the person steering the camera in the application. An OpenSpace session might have many observers, for example during a presentation in a dome theater, but only one pilot.

1.5 Delimitations

Some choices have been made to simplify the implementation and decision making process. Firstly, no graphical user interface (GUI) will be developed for specifying camera path instructions or constraints. This is outside the scope of this project and will be left as future work. Users will be able to provide input to the camera path generation system through the Lua scripting API that exists within OpenSpace, which will be extended with the required functionality.

Also, the user will not be able to edit a path once it has been generated. The focus will be on generating satisfactory camera paths immediately, without feedback from the user. Editing of the paths could be a possible extension of this project later on.

Thirdly, the path computation shall be limited to use only existing scene information in OpenSpace. No extensive pre-processing of the scene will be performed to generate new information. This limited the number of available methods to use. For example, several path finding methods used in AI and robotics use grid-based approaches for finding collision free paths.

Lastly, the simulation time of the application will be disabled. This means that the positions of the scene objects are assumed to be static, which simplifies computations regarding collisions and end positions for the camera path. The system can later be adapted to compensate for simulation time.

2 Related Work

Automated camera control and path planning is of importance within fields like robotics, computer games and visualization. Multiple applications of automated camera motions using different approaches exists. Only a few are combined with large scale virtual environments, which comes with its own challenges. Some other systems focusing on astrographics exists, with varying navigation options. A few have included automatic travel to a planet, while others also include functionality to design camera paths that can later be played.

This chapter reviews some previous work of relevance to this project, such as other systems for astrovisualization, previous approaches for automated camera control and usage of splines for motion planning. In addition, the current methods for navigation within OpenSpace are introduced, as well as some related prior thesis works.

2.1 Navigation in OpenSpace

Traveling in the universe is easiest when you know where you are going. The environment is very sparse and without aid a pilot might easily get lost in the emptiness of the universe. To address the difficulties with navigation in space Bock et al. adapted an orbital-based navigation scheme for navigation in OpenSpace [1]. The idea is that all interaction is performed in relation to a selected anchor, such as a celestial body. The pilot can steer the camera around the anchor while keeping it centered in the view or change the camera orientation using yaw, pitch and roll rotations (see section 3.1). Changing the camera orientation can lead to a sense of disorientation. To get reoriented the pilot can choose to re-target the camera aim towards the anchor. When doing so, a smooth interpolation between the current and target orientation occurs. This is also the case when a new anchor is selected. Once the aim has been rotated the pilot navigates in relation to the new anchor and can steer to the target by zooming in, or applying an automatic linear flight. The latter is the closest to any "automatic" navigation that currently exists within OpenSpace.

Another possible method of navigation is to instantly move the camera to a previously saved instance of camera properties, a *navigation state*. A navigation state includes all information that is required to reset the camera to that state at a later time: the camera position, orientation, anchor node, the current frame of reference and possibly a time stamp for the simulation.

The user can also navigate in the temporal domain by controlling the speed of the simulation or manually entering a desired date and time. As described by Bock et al., the time within OpenSpace is represented as a number of seconds past the 2000-01-01 epoch in double floating point precision [1]. Control of the simulation time is an important feature in OpenSpace, since part of the data the visualization is defined in relation to a point in time, like the position of celestial bodies, satellites or spacecraft.

OpenSpace has been a software that requires some experience to navigate smoothly and is mainly navigated using a computer mouse and keyboard. With a focus on bringing space to

the public, steps have been taken to allow inexperienced users to navigate in the software. Part of this work has been done through student projects. A tangible multi-touch interface for exhibitions was developed by Jonathan Bosson in 2017 [3]. In 2018 Hanna Johansson and Sofie Khullar developed a more user friendly interface built upon the new functionality [4]. Their solution allowed creation of a story with a set of targets which are then displayed as icons of the planet in a bookmark field. Clicking one of the icons triggers an automatic movement of the camera along a straight line to the new target. The user is then free to explore the new target and its moons, but is restricted to move too far away from the current target since that often lead to the visitors getting lost in space.

Collision Handling

The orbital navigation scheme handles collision with the current anchor node by preventing the camera to go closer than a certain distance to the anchor. Apart from this no collision handling is performed, meaning that it is fully possible to move the camera through other objects that are not chosen as anchor. The above mentioned approach is often sufficient, since the risk of colliding with other nodes in the scene is very low due to the sparseness and scale of the universe.

Most objects in the scene have a bounding sphere that is represented by a double precision value for the radius of the sphere. The scene in OpenSpace can hence be seen as a world of spheres. This representation is convenient for computing collision information, but the difficulty lies in determining what objects to check collision with. At the time of this project, OpenSpace has no implemented collision detection system. This means that to check whether a camera position collides with some bounding sphere in the scene, all nodes in the scene (the entire known universe) must be tested for collision in a brute force fashion, until one or none is found. In the worst case this leads to a time complexity of $\mathcal{O}(n)$, where n is the number of nodes in the scene.

Session Recording

OpenSpace currently has a system that can record a session in the software, save it to a file to then be play backed at a later time. The session recording provides less flexibility compared to a manual flight, but is useful when for example recording a movie in OpenSpace. This system does however require manual navigation during the session recording. Furthermore, information about the state of the camera for every frame in the recording must be saved. This potentially leads to large amounts of data having to be saved and stored on disk.

2.2 Scale Difference and Precision

A challenge related to visualization of astronomical data is how to represent the large scale differences with sufficient precision. The full scale of the observable universe covers about 10^{27} orders of magnitude, while our solar system have a magnitude scale of approximately 10^{13}. In between there is interstellar, galactic and intergalactic data. In addition, much smaller objects such as spacecraft or features on a planet surface must also be represented with sufficient precision.

To address the problem of the large scale range of the observable universe Axelsson et al. proposed a *dynamic scene graph* for OpenSpace [5]. The idea is to dynamically assign the frame of reference during run-time. By dynamically attaching the camera to the closest scene graph node of interest, the numerical precision can be maximized for the most important objects in the scene. This allows for navigation over scales and distances much larger than normally permitted by floating point precision.

Li et al. tried to solve the problem with navigation in large-scale environments such as the universe by implementing a graphical user interface to aid the navigator [6]. They suggested a three-dimensional mini map with logarithmic scale to give the navigator a sense of the current position in the universe, as well as visual cues that provides an insight in the current scale and context. Similarly to Axelsson et al. they mention the importance of an adaptive reference frame, but also how the speed sensitivity for the input device should be adapted to the current situation during navigation. Travel over intergalactic distances requires a differently scaled speed compared to travel within a solar system.

The interface by Li et al. includes a few models for aided navigation. One takes the observer to a selected target by smooth interpolation along a line, similarly to the current approach in OpenSpace. Another is a path-based model, where the user specifies a series of landmarks that are then interpolated between. The path planning and visualization can be done in logarithmic scale. Since context specific details of the path are hardly visible in the linear space, the logarithmic scale gives the user a better sense of the details of the path. However, the visual shape of the path will be different in the logarithmic and linear scale.

2.3 Other Systems for Astrographics and Visualization

Throughout the years several softwares have been developed with the focus of visualizing astronomical data. In addition to OpenSpace, some other systems that can be used for creating content for dome theaters or digital planetariums are Digistar[1], Uniview [7] and Gaia Sky [8]. Space Engine[2] is a newly launched software for simulating the universe and is aimed for home use.

Digistar was released in 1983 and while it could only project dots and lines at that time, newer versions are used in dome theaters today. Using an XBox controller, navigation can be done in a way that imitates the control scheme of an airplane.

Uniview is used for interactive space shows and film recording. Navigation is done using a similar orbital scheme as OpenSpace is using. Uniview also lets the user automatically fly to a new target along a straight line. However, this kind of motion is not suitable for every situation and users are advised to use the feature with caution.

Gaia Sky is another open-source software for real-time 3D visualization of astronomical data. The system includes a number of different camera modes[3], including a focus based mode that appears to be similar to the orbital navigation scheme. In addition, Gaia Sky contains a system that lets a user set up key frames for the camera, called the Camrecorder. The key frames are then interpolated between using simple spline mathematics. With some knowledge about the behavior of the interpolation this provides a lot of flexibility for designing exact paths for the camera beforehand. However, the system leaves all control to the designer and provides no assistance in creating interesting paths or navigating in real time in the sparse environment.

Space Engine aims to run on powerful personal computers and also support stereoscopic rendering. A difference to the others is that while the content is aimed to be science-based, it prioritizes to make the environments more visually pleasing. The environment includes procedurally generated textures and objects in uncharted areas.

[1] About Digistar: `www.es.com/Digistar` [accessed 19 June 2020]

[2] About Space Engine: `http://spaceengine.org` [accessed 21 June 2020]

[3] Documentation on the different camera modes in Gaia Sky: `gaia-sky.readthedocs.io/en/latest/Camera-modes.html` [accessed 19 June 2020]

2.4 Automated Camera Control and Motion Planning

An early example of an attempt of automating camera control is Blinn's framework for computing a camera configuration to show a planet and space probe on given screen coordinates using a look-at transformation [9]. However, Blinn's work was limited to the specific situation of showing two objects on screen on given coordinates to create interesting shots for space movies. Later on attempts have been made to generalize his framework using principles from cinematography. For example, in 1996 Christianson et al. tried to formalize the principles for camera placement into a declarative language [10] .

There are more aspects to automatic camera control than just determining where to place the camera. Drucker et al. attempted the creation of a procedural framework with the aim of addressing the problem of combining different camera control paradigms in one system [11]. Drucker and Zeltzer later extended the work and developed the system CamDroid, due to problems with constructing generalizable procedures [12]. CamDroid is a constraint based framework and the approach is to handle camera control by encapsulating camera tasks, like panning between objects or following a target. In another study, Drucker and Zeltzer applied their task based system for a virtual museum tour [13]. The constraint in this case is a path for the camera, which is computed using the A* algorithm and a connectivity graph for all points of interest in the virtual museum.

Nieuwenhuisen and Overmars used probabilistic roadmaps to compute a smooth free-flying camera path in a virtual environment between a given start and end position [14]. Their solution was implemented in a C++ Library named CAVE (CAmeras in Virtual Environments). The roadmap is used to find a shortest path with linear segments, which is later smoothed using circular arcs. A similar approach was presented by Geraerts and Overmars, in which a collision free *corridor map* was computed from a graph based on Voronoi diagrams [15]. The corridors were then used to compute a shortest collision free path using traditional path finding methods. Both of these methods produces promising results in environments with clear pathways, such as the virtual city used in the corridor map experiments. However, explicit storage of every point of the path is required, as well as an extensive pre-processing step. Garaerts and Overmars also addressed the problem of avoiding dynamic obstacles and compared the performance of two strategies: either updating the corridor during run-time, or letting the obstacles apply repelling forces to the camera.

Most of the approaches mentioned above, as well as many others, were included in a review by Christie et al. in 2008 [16]. The review covers some of the key issues in camera control and classifies the approaches based on interactivity and degree of automation. Many of the approaches are based upon contributions from robotics or AI and often include some sort of pre-processing step to compute information about the the scene, which is something to be avoided for this project. Also, they often require a large number of iterations to find an optimal solution, which can potentially introduce unnecessary computational overhead.

2.5 Splines for Path Planning and Motion Control

Within the field of animation and robotics it is common to use splines, or piece-wise parametric curves, for defining motion. In robotics, smoothness of the path is important for satisfying physical constraints, such as minimizing the distance or letting the robot drive as fast as it can without loosing its grip while turning. While the constraints in this application are determined by perception and visual aspects, the limitations for the curve have similarities.

Choi et al. applied Bézier curves and optimization techniques to efficiently solve a path planning problem for autonomous vehicles [17]. Starting from linear segments, they combined cubic Bézier spline segments to simulate a smooth motion with corridor constraints. However, apart from the constraining corridors their approach included no method for dealing

with possible collisions. Deshmukh et al. took the problem a step further when they used a Bézier spline to plan the motion for a robot following a moving target [18]. Their method included dynamically updating the curve based on the movement direction of the target object. Both the above mentioned approaches point out desirable properties of Bézier splines for motion planning, one thing being the smoothness they achieve due to a few number of turning points [18].

Amamra et al. used *Pythagorean Hodograph* splines for a camera path planning problem for a three-dimensional visualization environment [19]. The path was planned based on a known set of viewpoints and the properties of the splines results in a smooth path with minimized curvature.

3 Theory

The challenge of creating the camera paths requires some fundamental understanding of mathematical concepts related to virtual cameras and animation. This chapter introduces some of the relevant theory for this project, such as orientation representation, interpolation and splines. Additional important concepts and practices for virtual and real-time cameras are covered in the book *Real-Time Cameras: A Guide for Game Designers and Developers* by Mark Haigh-Hutchinson [20].

3.1 Camera Mathematics

When used for navigation, the camera within OpenSpace is essentially represented by two properties: its *position* and *orientation*. The camera position is defined as a point in world coordinates that describes where the camera is physically located in the scene, given in a specific frame of reference. The camera orientation is represented using a *quaternion*, but as explained by Haigh-Hutchinson there are several different ways to represent the rotation of a camera, like transformation matrices and Euler angles [20].

Representing Orientation

Quaternions are widely used within the area of computer graphics to represent three-dimensional rotations [20]. They originate from the theory of complex numbers and are generally represented in the form:

$$q = q_0 + q_1\boldsymbol{i} + q_2\boldsymbol{j} + q_3\boldsymbol{k}, \tag{3.1}$$

where q_{0-3} are real numbers and $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{k}$ can be interpreted as unit vectors pointing along three spatial axes. That is, a quaternion is essentially a four-dimensional vector with one scalar part and one vector part: $q = (q_0, q_1, q_2, q_3) = (q_0, \boldsymbol{q})$. When using quaternions for describing orientations, the scalar part q_0 specifies the amount of rotation, while the vector part $\boldsymbol{q}$ represent the angle of rotation. Specifically, the quaternion q can be written as:

$$q = \begin{pmatrix} cos(\theta/2) \\ v_x\, sin(\theta/2) \\ v_y\, sin(\theta/2) \\ v_z\, sin(\theta/2) \end{pmatrix}, \tag{3.2}$$

where θ is the angle of rotation and $(v_x, v_y, v_z)^T$ is the rotation axis.

The quaternion operations and their derivations are extensively covered by Hanson in the book *Visualizing quaternions* [21]. Hanson also describes some of the benefits of quaternions:

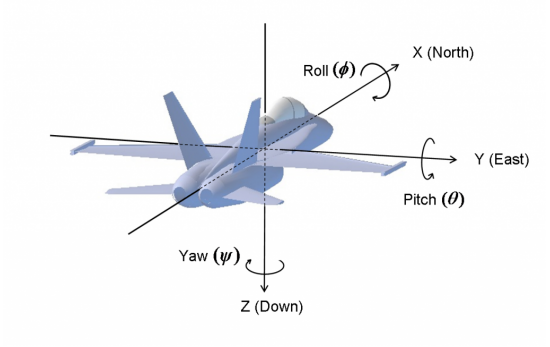

Figure 3.1: Illustration of Euler angles

- **Comparison:** By supporting computation of distances and similarities, quaternions allow for comparison between orientation states.
- **Interpolation:** A unique aspect of quaternions is that they support smooth interpolation between orientation states. Other representations, like matrices or Euler angles, have issues with this due to the occurrence of the so called gimbal lock [20].

A drawback of quaternions is the performance cost that comes with converting rotations from quaternions to matrix format. This cost is usually pretty small compared to the benefits that comes with using quaternions; especially the possibility of smooth interpolation.

Another way of representing the camera orientation is using Euler angles [20]. In this case, rotation is specified using three angles: *yaw, pitch* and *roll*, as shown in Figure 3.1. Each angle corresponds to the rotation around one of the axes of the object's local coordinate system. To compute the final rotation, the rotation matrices corresponding to each angle and axis are multiplied.

The Euler angle representation is more intuitive compared to quaternions when exposing the values for the rotation to the user. The angles are directly connected to the motion of air planes and thus more relatable for humans. There are cases within OpenSpace where Euler angles are used for rotation computations. In this case, the angle information is later used to compute the final orientation quaternion for the camera before it is updated.

Look-at

Apart from representing the orientation there are also a number of different methods available for computing a desired orientation. One common approach is to use a *look-at* computation, where either a quaternion or transformation matrix is constructed from a desired focal point of the camera [20]. First, the forward-vector for the view direction is computed as the difference between the current camera position and the focal point. Then, given an up-vector for the camera, the right-vector can be computed using a cross product. Once the right, up and forward vectors are obtained they can be used to construct the desired mathematical construct for the transformation.

As explained by Haigh-Hutchinson, the up-vector is often kept the same as the up direction of the world to maintain a consistent frame of reference for the observer and avoid rolling motions [20].

Easing Functions

Easing functions are curves that describe the rate of change of a parameter. In the case of animation they can, for example, be used to control the speed of an object. Easing functions are

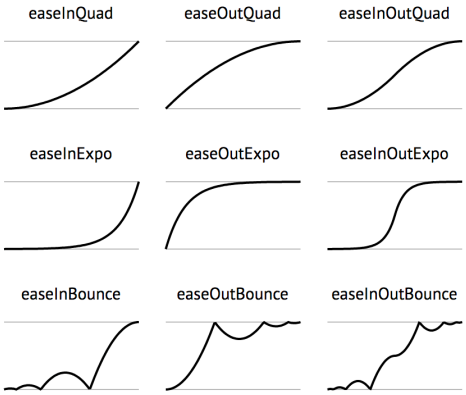

Figure 3.2: A few examples of easing functions. First row: quadratic rate of change. Second row: exponential rate of change. Last row: a special easing function used to create a bouncing special effect for the rate of change. From A. Sitnik and I. Solovev, `www.easings.net` [Accessed July 5 2020].

commonly used within animation to create more natural motions and transitions, since real objects almost never move with completely constant speed. When applied, easing functions make the interpolation of a parameter start and/or end with a continuous acceleration. They can also be used to create for example dampened or elastic motions.

Typically, an easing function is characterized by its mathematical function in combination with whether it eases in or out the parameter, or both. Easing in this case means that the derivative of the curve equals zero in the beginning and/or end of the curve, thus "easing" the rate of change. A few examples of easing functions are shown in Figure 3.2.

3.2 Interpolation and Splines

To compute intermediate values within the range of a set of discrete data points, interpolation is used. The simplest example is linear interpolation, where two values are proportionally mixed based on the interpolation parameter $t \in [0, 1]$. Using linear interpolation, any intermediate value between two data points v_1 and v_2 can be computed as

$$v(t) = (1 - t) \cdot v_1 + t \cdot v_2. \tag{3.3}$$

Notice how the proportions are set so that $v(t)$ exactly translates to v_1 for $t = 0$ and v_2 for $t = 1$. This is a characteristic of any method of interpolation.

Though simple, linear interpolation is powerful and widely used. In this project, a relevant version is the *spherical linear interpolation* (SLERP) introduced by Shoemake in 1985 [22]. It describes the interpolation between two points on a unit sphere and is important for smoothly interpolating between quaternions.

More advanced interpolation algorithms are often constructed from linear methods. An example is *spherical cubic interpolation*, also known as SQUAD, which is essentially a bi-linear construction of SLERP interpolations. The details of SQUAD and SLERP for quaternion interpolation are further described by Eberly [23]. Interpolation can also be performed using continuous curves between defined points, for example different *spline curves*.

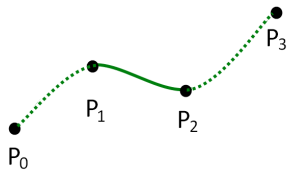

Figure 3.3: Illustration of the evaluation of a Catmull-Rom spline segment.

Splines are mathematical functions defined by piece-wise polynomials. They provide an efficient way of modeling complex curves and surfaces and are often used for interpolation problems, but also to define camera paths within the area of computer graphics [20].

Spline curves are constructed from a sequence of user specified ordered *control points* and possibly a number of tangent vectors. They can consist of just one single spline, or a series of composite spline segments. One segment of the curve typically include four (or fewer) consecutive control points. The segments can be independent of each other, but to achieve different levels of continuity they are often linked. The values t_i for the spline curve that correspond to evaluation of the start and end points of the segments are called *knots*. A *uniform* spline have knots distributed with equal distance, otherwise it is *non-uniform*. The spline either interpolates or approximates the control points based on the evaluation method, or *spline type*.

Many splines curves are examples of *Hermite* curves, which are defined by four control points and a cubic polynomial with the general form

$$Y = Ax^3 + Bx^2 + Cx + D, \tag{3.4}$$

where the constants A, B, C and D varies depending on the spline type. Cubic polynomials are often used within computer graphics and animation, since they can be combined to generate most desired curve shapes without introducing too costly computations.

Different spline types have different properties and the choice of spline type depends on the requirements for the curve. It might for example be a certain level of continuity, a constraining convex hull or simply the desired shape.

Catmull-Rom

Catmull-Rom splines are hermite curves where the tangent at each point $\boldsymbol{p}_i$ is computed as the difference between next and previous point on the spline. They are C^1 continuous with local control and pass through all control points. Evaluation of a curve segment is done using four consecutive control points, where the resulting curve will pass through the second and third point as shown in Figure 3.3.

The Catmull-Rom curve comes in different versions, with varying behavior depending on the parameterization of the curve. The parameter t_i corresponding to control point $\boldsymbol{p}_i$ can be computed as:

$$t_i = t_{i-1} + ||\boldsymbol{p}_i - \boldsymbol{p}_{i-1}||^{\alpha}, \tag{3.5}$$

where $t_0 = 0$ and the parameter $\alpha \in [0, 1]$. If $\alpha = 0$ the parameter values will be uniformly distributed, creating the *uniform* version. A *centripetal* parameterization is achieved with $\alpha = \frac{1}{2}$ and $\alpha = 1$ yields a *chordal* parameterization. A comparison of the three versions is shown in Figure 3.4.

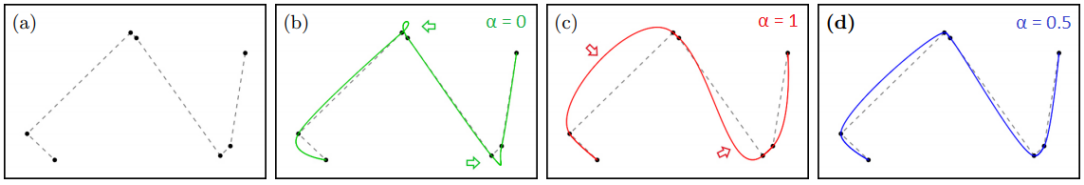

Figure 3.4: Catmull-Rom spline evaluation using the same control polygon (a) but different α values. Notice the self-intersection and overshoot in the short segments of the uniform version (b) as well as the rounded shape that is generated from the chordal version (c). Adapted from Yuksel et al. [24].

The effect of different α-values has been analyzed by Yuksel et al. [24]. According to their study, only the centripetal version is guaranteed to avoid self-intersections. Uniform distribution can introduce sharp turns as well as overshooting and self-intersections in shorter segments. Furthermore, the chordal version might deviate a lot from the control polygon. However, it generates a rounder curve that might be more suitable for motion control.

An extension of Catmull-Rom splines are *Kochanek-Bartels splines*. These include parameters for tension, bias and continuity to change the spline tangents and thus alter the curvature in the control points [20].

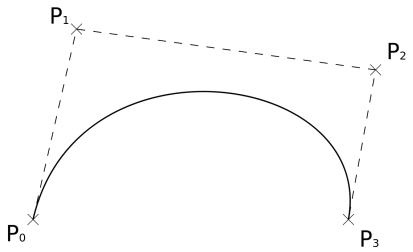

Figure 3.5: Cubic Bézier curve segment created from four control points, where $P_1 - P_0$ defines the tangent in the point P_0 and $P_3 - P_2$ makes out the tangent in P_3.

Bézier

Bézier curves are often chosen for their aesthetic shape and intuitive representation. They are guaranteed to stay within the convex hull of the control polygon. The cubic version uses four control points $\boldsymbol{p_0}, ..., \boldsymbol{p_3}$ and is evaluated by the equation

$$\boldsymbol{B}(t) = (1-t)^3\boldsymbol{p}_0 + 3(1-t)^2\boldsymbol{p}_1 + 3(1-t)t\boldsymbol{p}_2 + t^3\boldsymbol{p}_3, \tag{3.6}$$

with $0 \leqslant t \leqslant 1$. The curve shape is determined by the intermediate control points, which define the tangents in the start and end point of a segment (see Figure 3.5). The curve will pass through the start and final point of the segment, but generally not the intermediate points.

A composite Bézier curve, i.e. a series of connected Bézier segments, is C^0 continuous. However, higher order of continuity can be achieved by careful manipulation of the tangents. C^1 continuity requires parallel tangents in the knots, while C^2 continuity can be achieved if the knots also have matching curvature. Guaranteeing the continuity constraints introduce dependencies to intermediate control points of adjacent segments.

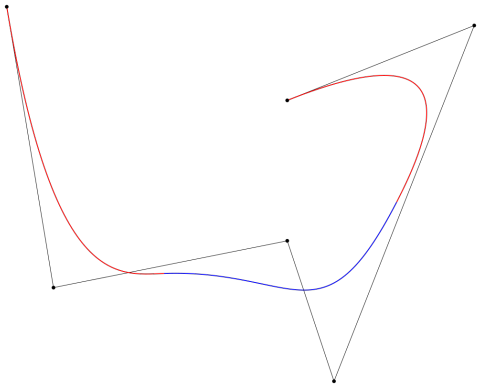

Figure 3.6: Example of a uniform B-spline, with marked out internal Bézier segments.

B-splines and NURBS

B-splines are a generalized version of the Bézier curves, composed of cubic Bézier segments (see Figure 3.6). They have built-in C^2 continuity, provide local control and can reproduce the same curves as Catmull-Rom and Bézier curves, while keeping the property of staying inside its convex hull. As for the Bézier curve, the B-spline is guaranteed to pass through the first and last control point but approximates the others. Furthermore, the order of the B-spline increases with the number of control points. This requires additional information to represent the spline. It can also be less intuitive to control the shape of the resulting curve.

Most Hermite curves are unable to represent conical shapes such as circles or spirals. A desirable property of non-uniform rational B-splines (NURBS) is their ability to do this.

Pythagorean-Hodograph Curves

A subgroup of hermite splines that has been gaining popularity for controlling CNC-machinery is Pythagorean Hodograph curves (*PH-curves*). They have geometric properties that enables exact computations of for example arc length and curvature. Other polynomial splines usually require these quantities to be approximated, using for example quadrature rules. In 2012, Gajny et. al. proved that PH-splines have desirable properties for path planning in real-time and concluded that they both provide better accuracy and might be up to almost 15 times faster than regular polynomial splines [25].

4 Defining Appropriate Camera Motions

Within camera mathematics and animation a good path is a smooth path that reaches a certain level of continuity. It should also avoid collisions with objects in the scene and guarantee frame coherency and that the destination is reached. The goal of this project also includes the semantic aspects of the motion. The creation of a motion that is aesthetically pleasing and provides a good experience for the user depends on several different aspects, like the artistic freedom provided to the person designing the path. Another key factor is avoiding motion sickness. This phenomenon is only partly understood, but seemingly depends on the degree of experienced immersion and how the rotation and speed of the camera is controlled. In addition, the field of astrographics comes with its own unique challenges for camera motion, related mainly to the sparsity and large scale of the environment.

4.1 Freedom

Making something aesthetically pleasing and meaningful is an artistic problem and is often better left to humans than computers. Whether it is creating a film sequence or traveling through space during an interactive show, OpenSpace is used to tell a story unknown to the software. The aim is to provide a tool for storytellers who want to communicate their personal narrative and create an experience for their audience. In many ways a "good path" is therefore one that the pilot can choose.

On the other hand, the freedom is useless with limited navigation skills and knowledge of the scene. Even an expert pilot is unlikely to know the current placement of the planets and the optimal way between them. An automatic path relieves the pilot from these expectations, which means limiting the freedom. The freedom should not, however, be completely removed. After all, a path with a random destination is not very interesting while "Take me to Earth" or "Go to this predefined view of the Grand Canyon" are both interesting cases.

The more specifications given, the less flexible the path generation becomes. By specifying the path too strictly the pilot can accidentally force the system to create a less pleasant path, for example by specifying a too short duration for the motion. A good tool for automatic paths has found a balance in providing automatic functionality while maintaining the possibility for precise artistic expression.

4.2 Motion Sickness

Avoiding motion sickness, or more specifically in this case *cybersickness*, is a key factor to what is considered a good path. Vision is one of the main sensory systems used for sensing motion. Most people have at some point experienced a strong sense of movement when seeing a train

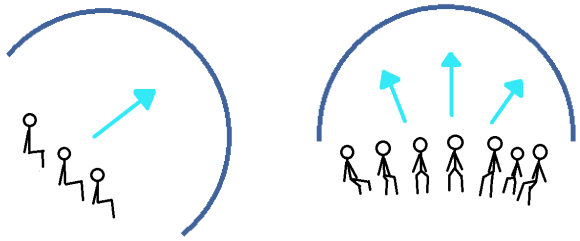

Figure 4.1: Illustration of a uni-directional (left) versus omni-directional (right) dome.

depart from a platform. Before the brain registers that it is actually the train that is moving the observer can experience a *visually induced motion sickness* (VIMS).

Even if the phenomenon is well known, the actual source of it is still not verified. The most common theory to explain the source of motion sickness is a confusion between sensory mismatch of the sensed stimuli and the perceived reality. Part of the field instead prefers to explain motion sickness as a cause of postural instability; an inability to keep a stable posture matching a wrongly perceived gravitational direction. A third more general version of this theory focuses solely on the perceived sense of the up direction. Bos et. al explain VIMS as a phenomenon that can occur when the motion is something the observer cannot anticipate [26]. This might be any movement which is perceived as unnatural because it is too erratic, changes too fast or in an unexpected direction. Hence, in terms of motion sickness a motion that is easy to anticipate is to prefer. Rebenitsch and Owen extends this theory by suggesting that scene context might be a neglected parameter [27]. Psychological aspects correlated to VIMS are still poorly understood. The few studies that have been conducted have found correlation between cybersickness and the scene content and its level of realism, exposure to similar experiences as well as the observers mental state due to anxiety or extreme hormone levels [28, 29, 30, 31].

The symptoms that occurs vary widely between individuals, making it difficult to find a specific measure for motion sickness. In more early studies the most popular measurement of the level of motion sickness was the *simulation sickness questionnaire* (SSQ), which maps a large number of symptoms to severity scores for the level of nausea, oculomotor and disorientation, as well as a combined total score. In recent years measures of physiological state and body sway have been introduced, but are usually still combined with a shorter questionnaire. The different measures in combination with varying displays makes it difficult to compare studies and find general knowledge of VIMS. A review on cybersickness by Rebenitsch and Owen unite findings within the cybersickness field and highlights the lack of general recommendation for designing applications with respect to VIMS [27]. To conclude some of the most important aspects appears to be related to navigation, rendering mode and screen type.

Considering Different Displays

Sense of immersion and presence is found to be negatively correlated to cybersickness, which might partly explain the big difference of VIMS caused by different displays [27, 32]. OpenSpace is not specific to one platform and the risk of motion sickness is significantly less for a monitor compared to more immersive displays such as the hemispherical (or omni-directional) cinema at the Hayden Planetarium[1] at the American Museum of Natural History or the stereoscopic (or uni-directional) dome at Visualiseringcenter C[2] in Norrköping, Sweden. Figure 4.1 shows the difference between these two dome types. The best way to ensure

[1] `www.amnh.org/exhibitions/permanent/hayden-planetarium` [Acessed 20 June 2020]
[2] `www.visualiseringscenter.se` [Acessed 20 June 2020]

a low sickness risk for a motion is testing in the specific environment and conditions beforehand. Ensuring a good camera motion on multiple platforms could require that adjustments can be made to match the different displays. For example, motions can be experienced as more extreme on a larger screen, which can be compensated for by reducing the speed. Different kinds of motions that are known to cause problems on some displays could also be avoided or limited.

As a last remark on displays, the experience varies between observers depending on their level of control and how they are seated in relation to the screen. A single pilot could prefer to have control over the steering while the audience are merely passengers who can not anticipate the motion as well, making them more prone to sickness. During the automatic path the pilot will also become a passenger, which might make their experience more equal.

Appropriate Speed and Rotation

Another aspect that has a large impact on the experience of a motion is the travel speed. A slow speed is less likely to make anyone sick, but it can lead to a less exciting experience. Going as fast as possible is considered the best speed in most path planning applications and has been been applied also for optimal camera motion in virtual environments [14]. The difficulty lies in knowing what is too fast, especially in a sparse scene such as the solar system where the perceived speed is less connected to physical laws. The perceived speed and sensation of movement depend largely on visual cues and changes in the scene. Without any visible objects close to the camera it can seem like the camera is not moving at all, even when traveling towards a target in a speed corresponding to more than light speed in reality. Staring at an unchanging scene can be both boring and confusing for an observer.

For a sparse scene, one approach that has been used is to approach an object with a speed that is logarithmic to the distance to the surface of the object [33]. This makes the size of the object on the display grow in a constant rate, under the assumption that the camera is looking and traveling straight towards the object. The object closest to the camera will in most cases have the main impact on what is considered a suitable speed and Li et al. have also decided to match the speed with the scale of where in the universe the travel occurs [6].

Angular rotation can lead to quick changes in the scene and easily induce motion sickness. In most real-time situations it is also desirable to rotate smoothly over time and possibly dampen the angular velocity at the start and end of a motion to enhance predictability [20]. Regarding the direction of rotation, Nieuwenhuisen and Overmars accentuate on the importance of allowing the viewer to anticipate cues about the future motion of the camera [14]. Their solution to the problem was setting the camera rotation to look at a future position along the path. A human spinning in reality would prefer to quickly set their eyes on a new target and then spin their body; ask any ballerina. For a camera paths in space it is not always possible to see the new target or any other object, making this behavior difficult to mimic. For the empty scenes, limiting the angular acceleration might be the best approach for reducing motion sickness.

Rebenitsch and Owen conclude in their review that existing studies have struggled to find any translational or single rotational axis more prone to induce discomfort than the others [27]. Their suggestion on some general guidelines for designing navigation with motion sickness in mind includes keeping speed low for both translation and rotation, as well as preferring a single rotational axis over combining them, which some studies have found to produce more sickness [34, 35]. In computer games and movies the roll rotation, where the view direction is used as rotation axis, is often avoided to keep a constant up direction. Flight simulations are one of few exceptions, where the roll is used in moderation to mimic the behavior of an airplane. Carter Emmart has long experience of navigating OpenSpace and shares that if there is a horizon, it is also desirable to keep its placement on the screen con-

stant while moving closer to the surface of a planet. Traveling in space is supposedly similar to a flight simulation and even if there is no gravity in space, a human might prefer to imagine so. It appears that humans have a strong sense for the downward direction. Apart from gravity, visual cues play an important part in deducing this direction and contradictions between the deduced and actual direction is linked to motion sickness. Bos et al. presents a real example of how the lack of visual cues in space lead to motion sickness [26]. Astronauts traveling between two separate rooms underestimated their own rotation and reported sensing motion sickness when arriving to something that looked as if it was tilted unnaturally. Furthermore, the paper includes one of the few attempts to express the level of motion sickness as a function where the measure is completely based on the angular velocity in relation to the perceived direction of gravity. It is unclear how much variability exists between individuals and how well the expression can be applied to a different environment, but it could provide a cue for an algorithm optimizing orientation interpolation of a camera path.

At the uni-directional dome theater in Visualiseringcenter C in Norrköping Alexander Bock, development lead of OpenSpace, has observed that some rotations in OpenSpace are more prone to induce motion sickness than others. In this environment it is not intuitive which combinations of rotations will induce motion sickness. Sometimes, applying a more complex rotation can make the motion more pleasant. Furthermore, the same camera motion that induces motion sickness in a dome theater might not create symptoms when viewed on a computer screen. This suggests that different options for rotational behavior is appropriate for cross-platform adaptation.

4.3 Understanding Motion in the Universe

Space visualization aims to help humans observe the universe and get an understanding of the data. A natural goal becomes expressing the content in a way that helps a human to grasp it, taking the abilities and limitations of the mind into consideration. The human mind has an attention span of a few seconds and can only remember simple behaviors of an object that is no longer visible. A good transition allows the observer to envision the scene to a level that is beyond separate images.

A challenge within space visualization is to give the observer a coherent frame of reference during the transition to a new target. The size of two objects compared to the large distances often makes it impossible to see both at the same time, making it difficult to compare the size of the objects and the distances between them. The exception is objects located in close proximity to each other, such as moons or spacecrafts orbiting a planet. A larger screen makes it more difficult to find small targets and it can be helpful to making it possible to predictable where the objects in the scene will appear. Diels et. al. suggest keeping the placement of the focus on the screen both static and centered [36]. When traveling between two planets, experienced OpenSpace pilots will sometimes choose to zoom out to a view of the entire solar system, such as the one in Figure 4.2. Even though the planets are too small to be visible in this view, their rendered orbits can help give a sense of the distance traveled. Gazing down at the solar system when traveling contributes to keeping something interesting in view for most of the motion and the observer is also given the chance to better understand the relationship between the planets.

It is worth noting that the visibility of solar systems and galaxies is limited by their planar shape. The disc formation of the galactic or orbital plane is not visible from all directions. A path inside this disc might limit or completely lose the benefit of visual cues along the way. Within our solar system, this kind of path also increases the risk that the orbit lines create confusion rather than insight. When the lines are too close to the camera they can appear curved on the display or move quickly. Another problematic aspect is that the visibility of the celestial bodies is compromised due to the lack of illumination in the scene. In our solar

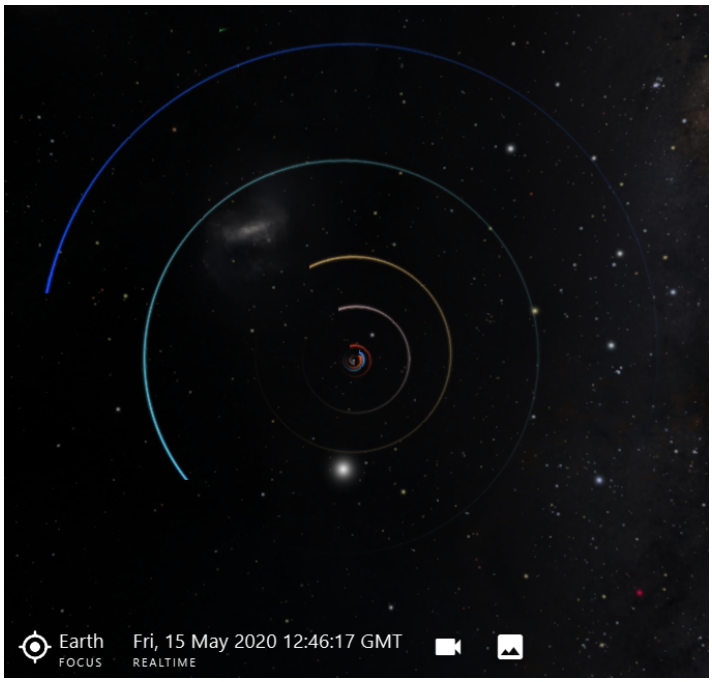

Figure 4.2: An overview of the solar system, in OpenSpace, from a perpendicular direction to the orbital plane. The colored lines are the orbital trails of the planets.

system, the Sun is the only source of light and it can be difficult to spot a planet in shadow against the dark background. To address this problem it might be useful to let the procedural curves take the light direction into account.

As a last note, the environment has big variations and best practices differ between paths. Any specialized solution is risking to be worse than the simpler path in unintended use cases. Creating too many assumptions of the behaviors based on relationships to a planetary system or light direction can potentially lead to strange movements. On the other hand, humans are very flexible at handling exceptions and can see a natural solution for specialized cases. Paths with less variation to circumstances might be less predictable and introduce risks of motion sickness.

5 Use Cases

To form the foundation for the design choices made in this project a number of use cases have been defined for the system. They have been identified based on discussions with OpenSpace experts and users, mainly during the pre-study phase but also continuously throughout the project.

The use cases identified for an automatically generated camera path can be summarized as:

1. Steer to a destination in one motion
2. Steer through multiple specified points or camera states in one motion
3. Steer through a sequence of separate motions

The first case covers the spontaneous choice to go somewhere from your current location. The current implementation combines orientation interpolation with manual steering. An algorithm that steers the camera might provide a smoother experience.

The second case is similar to the keyframed paths in Gaia Sky's Camrecorder system (see section 2.3) and was suggested as a way to allow the creation of a specific predefined path. The curve is forced to intersect all the points provided which gives the designer more freedom to force a specific shape, but also more responsibility. The design process of this path could occur during a preparation for a show or movie recording and provide designers with a way to test and adjust their camera paths until the resulting motion is satisfactory.

The last case is a useful tool for preparing a show. Having a predefined sequence of camera motions can be compared with a slide show and can be fine tuned before hand. While the other two use cases create one smooth motion, this case can allow the navigator to stop at a target before continuing to the next. The next motion could in practice be either of the two previous use cases.

5.1 Specifying Motion Details

Each motion can be further specified in addition to the target or intermediate targets. For example, when holding a presentation it could be useful to specify a desired duration of the motion, e.g. "Go to X in A seconds". The end state of the camera motion might also be specified further, for example by a desired distance to a planet or other target. It could even be specified as an exact coordinate and orientation configuration using a navigation state (see section 2.1).

Furthermore, when steering to a sequence of paths it might be of interest to provide information about whether the camera motion should stop at the intermediate destinations and perform a chosen behavior before continuing to the next. During the pause, the presenter may talk about the target or respond to questions from the audience and should be able to

navigate freely. However, in the case of a movie recording it might be of more interest to specify the exact duration of a pause.

Default Settings and Platform Specific Adaptations

As OpenSpace runs on different platforms, it should be possible to control some parameters that affect the quality of the camera motion depending on these platforms. An important example is the speed of the motion, since an observer is more sensible to fast changes when viewing the application in a dome compared to on a monitor. Another example, for similar reasons, is that it is often desirable to remove the roll when interpolating between rotations. Furthermore, it could be helpful to be able to set default behaviors such as curve type, speed settings and pause behaviors before hand. That way they do not need to be explicitly specified by the user every time a path is to be generated. For instance, the first use case ("Go to X") can happen many times during an OpenSpace session. The presenter or pilot usually wants the same behavior in most cases and it would be cumbersome to be specify these details for every individual session.

5.2 Camera Behavior at a Target

Upon arrival at a planet it is often desirable to stay at that planet for a while before leaving. A completely static scene is uninteresting to the audience and during OpenSpace shows an experienced pilot will, for example, create a slow orbital motion. Hence, when specifying a series of paths it is helpful if there is a way to define a behavior during a pause. Some examples of possible instructions for behavior at targets could be: "orbit the target, until button click" or "time lapse, for 20 seconds". To clarify, this pause behavior does not aim to take the camera to a specific place and is not defined as a use case for the automatically generated path. It is rather the opposite, a camera behavior for the time that the camera is not controlled by a played path. A sequence of camera behaviors would include both.

Using different pause behaviors can make a show more exciting and some behaviors could also be tailored for a specific platform. For example, the Hayden Planetarium at the American Museum of Natural History has an *omni-directional* dome theater, where the edges of the dome display is parallel to the floor the observers are seated in a circle and looking up at the display (see Figure 4.1). That is, the dome has no unified forward direction. A behavior requested by the personnel at the Hayden Planetarium is a move which lets audience in all directions see the planet as it travels around the room.

6 System Implementation

First of all, a system that allows users to specify details and constraints for a path to be created was implemented. This input is then used to generate a spline path matching that specification. This includes making choices about the aspects of the path that were not included in the specification. The path creation includes several different aspects and is covered in full in chapter 7. Finally, the camera is moved along the generated spline path. This should be done in a continuous manner and with a well defined speed that ensures a comfortable experience for an observer.

In addition to specifying, creating and traversing a path, functionality for handling the behavior during a stop in the path was also implemented. A simple method for rendering a generated path was implemented as well. Being able to see the path was important to get valuable insights of the behavior of the curves during the implementation, as well as for debugging.

Spline curves were chosen for the representation of the camera path due to their performance and stability. Knowing future positions allows for pre-processing and a reduced number of on-line computations. Splines also provide a compact representation compared to storing a camera state for each frame, which reduces expensive read and write operations.

6.1 Path Specification

To create a path, the system requires a specification from the user that describes the details of the desired motion. The implementation of the specification has been designed to support the use cases described in chapter 5. The path specification was implemented as a Lua table or JSON object which is passed to a path generating function in the Scripting API of OpenSpace.

A path specification consists of a list of *instructions*. Each instruction represents one continuous motion, an independent segment, of the final path. Depending on the use case an instruction contains a varying amount of information. The minimal information is a specification of the camera destination. The duration for the motion might also be specified.

Apart from instructions about where to go, the specification also includes information about whether the path should stop between intermediate segments. This can be done for the complete path or per instruction. Information about the stop also include the behavior to be applied in the stop, and possibly the duration until the path shall be resumed. See section 6.1 for more details. Furthermore, the possibility to add a specific start state for the camera was added. This feature increases the flexibility of the system and lets the user create the same motion every time the same specification is provided to the system. Note, however, that a requirement for creating the exact same motion is that the date and time of the simulation of the following runs matches the first.

The information of each instruction in the path specification is used to generate a set of *waypoints* that the path will pass through. A waypoint stores information about the state of the camera, position and orientation, in that point. It also includes a reference to the scene graph node associated with that camera state. This allows the system to make conclusions and choices for the path with respect to that position of the node, its size, and so on. Once a series of waypoints have been generated from the path specification the actual camera path can be created. See chapter 7 for details.

The use cases covered in chapter 5 translate to two different types of motions for the system: a target-based motion and a motion with multiple waypoints. The second motion requires a detailed specification about the points or camera states that the camera shall pass through during the motion. For the target-based motion the objective is instead to steer the camera from its current state to a specific target in the scene. This can possibly leave a lot of freedom to the system, since not every detailed about the desired target must be specified.

A target is a representation of where the user wants the camera to go. For a single destination four possible target options were implemented:

1. A node in the scene
2. A node in the scene, with a specified height above its bounding sphere
3. A position in relation to a node
4. A navigation state (see section 2.1)

The options are ordered with increasing level of detail. In the latter two cases the desired end position for the camera can be identified and computed directly. In the other, the choice of the exact position is left to the system.

Choosing an End State for the Camera

If no target waypoint can be deduced from an instruction, the system must compute a final position for the path segment. Depending on the input, information about the end state for the camera might be partly defined or not at all. However, the minimal information is a specified node in the scene to compute the end state in relation to the camera.

For the camera orientation it is assumed that the specified node should be in view. Therefore the end orientation of the camera is always set to face the node in question, by computing the view direction as the vector between the center position of the node and the final camera position. The distance from the end position to the node center is computed by a constant times the radius of the bounding sphere of the node. If a desired height is specified, the distance is set to the radius plus the height value.

Choosing the camera position is a bit more problematic. The end position is required to create the spline curve of the path, but it might be a challenge to choose a suitable end state before the shape of the curve is defined. If the end state is not well defined in regards to the shape of the curve the path might make an odd detour to reach the specified position. The position also affects the final orientation, since it is used to create the view vector for the camera. Choosing an end state that yields an orientation similar to the initial orientation can be useful for minimizing the angular speed required for the interpolation. However, the outcome depends on the chosen interpolation method. Moreover, keeping interesting objects in view could be more important than reducing the amount of rotation.

There is no light in space apart from the sunlight and the objects in the scene are hardly visible from the side facing away from the Sun. The position was computed to always be on

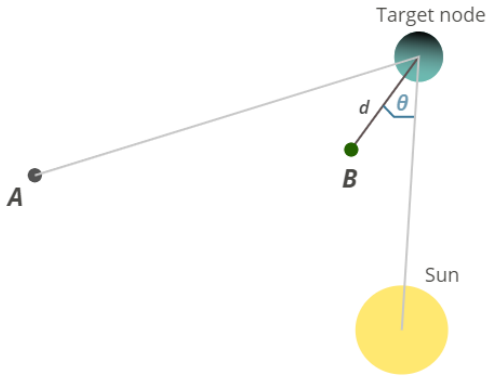

Figure 6.1: Computation of the default end position for the camera so that it shows the illuminated side of the target. ***A*** is the start position for the camera. ***B*** is the resulting end position, computed from position ***A***, the positions of the Sun and target node, an angle θ and a distance d to the center position of the node.

the illuminated planet by computing the direction form the position of the Sun to the target node. To create a more interesting view with an increased sense of depth, a slight rotation around the target node was added to compute the final position, as illustrated in figure 6.1. The rotation axis was computed as the cross product of the vectors from the target node to the start camera position and Sun position respectively.

A special problematic case is when the targeted node is close to another node in the scene. An example is the *International Space Station* (ISS) which is orbiting close to the Earth. If the end position is generated to be between the two nodes it leaves little room for the path to nicely approach the target. It also makes it difficult to avoid collision with the nodes without applying sophisticated collision handling methods. Furthermore, scene graph nodes are sometimes placed on the surface of another node. In this case, if a path is created from the node on the surface the system must avoid to create a target position that is inside or very close to the larger node.

To address the problems above, the target position of the camera was computed differently if the targeted node is in close proximity to another node in the scene. The end state is then computed based on the center position of the nearby node so that it is located behind the targeted node (see Figure 6.2). This reduces the risk of collision when leaving or approaching the target. The nearby node is found by comparing the center point against all other relevant nodes in the scene, or until a proximate node is found. The node is considered close if the position lies within the sphere represented by the center of the other node and a proximity radius, which is defined as a constant times the radius of the bounding sphere of the node.

Default Duration

In addition to the final position, the duration for the path is allowed to be determined by the system if no value is specified. To avoid getting a long duration for large distances the duration was computed as the natural logarithm of the distance for the path. For travels within the solar system (order of magnitude 10^{13}) this yields a duration of up to approximately 30 seconds, while travels over the intergalactic scale (approximately 10^{24}) will take about a minute. This was considered a suitable choice for monitors, but as discussed in chapter 4 the experienced speed differs between applications. Hence, an user-defined parameter was added to allow the user to scale the duration.

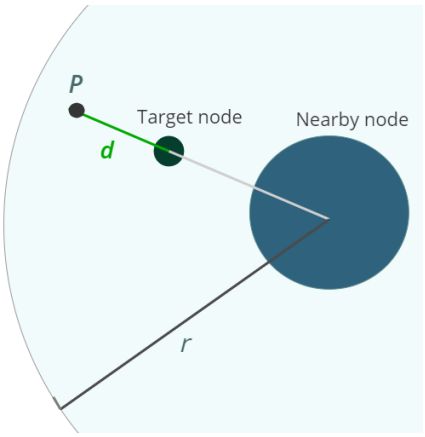

Figure 6.2: Computation of the end position when the target node is close to another node in the scene (within the proximity radius r of a node). $\boldsymbol{P}$ is the resulting end position, computed in the direction given by the vector between the target and nearby node center with a distance d from the target node center.

Setting an appropriate duration can also be rephrased as setting an appropriate speed for the motion. For a path with a constant length, the time it takes to traverse the path is inversely proportional to the average speed. This relation is important to consider when defining methods of controlling the camera speed (see section 6.2).

Pauses

The intermediate stops in the path is represented using one entry of pause information between every pair of path segments. Each entry contains information about whether the path should stop at the target and possibly for how long. If no duration is specified, the path is paused until the user chooses to resume. The pause information is always created using some set default values, but can be changed by the user through the path specification.

Additionally, each pause entry contains information about the desired behavior during the stop. A pause behavior is a predefined camera motion in relation to the target node, that moves the camera in a certain way each frame. Two different behaviors were implemented:

1. free navigation, which just switches back to the normal orbital navigation scheme
2. a simple orbit motion that moves the camera in an orbit around the target node

The implemented pause is a break in the predefined path. A consequence of this is that once the path is continued, the succeeding path segment needs to be recomputed if the start state of the path segment has changed.

6.2 Motion Along the Path

The camera is moved along a path segment-wise using a forward-stepping method. Every frame, the displacement is computed using the time step dt and a value for the current speed of the camera. The total distance traveled along the path is then computed by summing the resulting displacement with a stored value for the previously traversed distance. This gives an approximation of the current arc length along the spline, which can be used to compute a position on the curve if the spline is parameterized by arc length. For consistency, the camera orientation is also interpolated based on the traveled distance, rather than time.

The numerical method chosen for stepping along the curve was the *midpoint method*, a modified version of *forward Euler* that has higher precision. This approximation method has a growing error that increases with the step size. To reduce the numerical error each frame was split into n iterations to get a step size $h = \frac{dt}{n}$. A too large step size might lead to visible errors, especially close to visible objects in the scene. While a large step size does not matter in the emptiness of space, where the visual changes are minimal, taking too large steps close to an object might lead to an erratic motion. For example, when approaching a planet the step size should be small enough to not result in discontinuities in the perceived speed of the camera. If the step taken is too big, the visible size of the object on screen might change a lot between individual frames. Using a small step size everywhere would be ideal, but might introduce a large computational overhead for a long path. For increased efficiency, a larger n could be used where there is a high risk of visible errors compared to where the risk is low.

Computing the Arc Length

The path traversal requires computing the arc length at different points on the spline. Considering a particle traveling along a parametric spline curve $f(t)$, the distance traveled is equal to the integral of the speed of the particle over the time for the motion, that is:

$$L = \int_{t_{min}}^{t_{max}} |f'(t)| \, \mathrm{d}t, \tag{6.1}$$

where L is the traveled distance and t_{min} and t_{max} are the values for the integration parameter t at the start and end of the motion. The integral in Equation 6.1 has no closed-form solution, but can be approximated using numerical integration methods. For this project, 5-point *Gaussian quadrature* with Legendre points was used to compute the arc length. This method was chosen due to its efficiency and high accuracy. An n-point Gaussian quadrature is guaranteed to yield exact results for polynomial functions up to order $2n - 1$, which is true for all splines used in this implementation. Other numerical integration methods, like the midpoint or Simpson's rule, often require a very large resolution for the error to be sufficiently small, leading to a lot of iterations. This is especially the case in this application where the distances are extremely large and relatively small errors lead to large offsets.

Finding the Curve Parameter

The relation between the interpolation parameter of the spline curve and the distance traveled along the spline is rarely linear. A mapping between the curve parameter $u \in [0, 1]$ and the arc length $s \in [0, L]$ must be determined to be able to regulate the traversal speed along the curve and use the traveled distance as an interpolation parameter. L is the total length of the spline curve. Notice that the curve parameter is in Equation 6.2 denoted as u rather than t, to not be confused with the time parameter for the speed function in Equation 6.5.

Let $\boldsymbol{Y}(u)$ be a parametric spline defined over $u \in [u_{min}, u_{max}]$. The curve parameter u matching a given arc length s can then be found by inverting the integral:

$$s = \int_{u_{min}}^{u} \left| \frac{d\boldsymbol{Y}(\tau)}{du} \right| \mathrm{d}\tau, \tag{6.2}$$

where $\frac{d\boldsymbol{Y}(\tau)}{du}$ is the rate of change of the spline curve in $u = \tau$. The integral in Equation 6.2 must be inverted using numerical methods. A version of Newton's method for root finding proposed by Eberly was used for this. The version suggested by Eberly guarantees convergence even for non-convex functions, by using bisection when the candidate computed from

Newton's method is outside the current root-bounding interval [37]. This makes it reliable for real-time situations.

The algorithm for computing u from an arc length $s \in [0, L]$ along a composite spline, using Newton's method and bisection, can be summarized as follows:

1. Determine the current segment matching the arc length s, by comparing s against precomputed values of the arc length of each segment.
2. Compute the arc length traveled along that segment, s_{seg}, that corresponds to s.
3. Make an initial guess for the candidate:

$$u = u_{min} + \frac{s_{seg}}{L_{seg}}(u_{max} - u_{min}), \tag{6.3}$$

   where L_{seg} is the length of the current spline segment and u_{min} and u_{max} is the minimum and maximum value for the curve parameter within that segment.
4. Initialize the root-bounding interval $[a, b]$ to $[u_{min}, u_{max}]$.
5. Until a root is found, or a maximum number of iterations have been performed, do:
   - Compute the arc length along the current segment matching the range $[u_{min}, u]$.
   - Let $F = arcLength(u_{min}, u) - s_{seg}$. If $|F|$ is less than a specified tolerance, report the current candidate u as the root.
   - Otherwise, generate a new candidate and update the root-bounding interval according to:

$$[a, b] = \begin{cases} [a, u], & \text{if } F > 0 \\ [u, b], & \text{otherwise} \end{cases}$$

   - If the new candidate is outside the interval $[a, b]$, use the bisection instead: $u = \frac{upper+lower}{2}$.

If no root is found after the maximum number of iterations the last computed candidate is used as a result for the curve parameter. The bisection guarantees that the interval length strictly decreases over time and that the candidate gets closer to the real solution for every iteration.

The curve parameter is computed from the traveled distance during run-time. The method described above requires several iterations and evaluations of the arc length every frame. An alternative would be to approximate the curve parameter by precomputing sample pairs of s and u values that are fitted together using some sort of polynomial regression, as proposed by Eberly [37]. This would speed up the curve parameter computation a fair amount, but the result would be an approximation. Due to the huge distances traveled, a lot of samples would be needed to achieve a sufficient resolution that does not introduce any visual artifacts due to insufficient precision. Implementing the fit of the samples would also be somewhat time consuming. Since the efficiency of the chosen arc length evaluation method is high enough to not introduce any significant degradation in frame rate if done at run-time, enhancing the efficiency of this step was left as a later possible improvement.

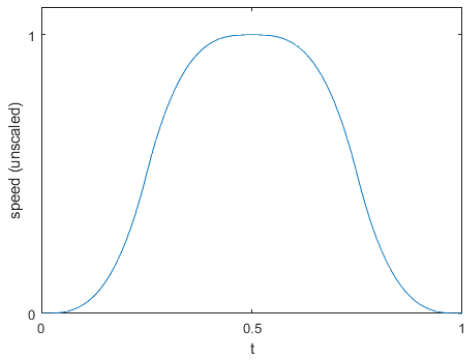

Figure 6.3: A transfer function used for controlling speed, defined piece-wise by two cubic easing functions.

Controlling the Speed

An important aspect of the motion is controlling the speed to generate a smooth and comfortable movement. To avoid long uninteresting shots without anything interesting in view the speed should be as high as possible when far away from any visible objects in the scene. When approaching a planet, the camera should decelerate smoothly and eventually come to a stop. To avoid disorientation and the risk of motion sickness, continuity in both speed and acceleration should be considered.

The speed is represented with a C^2-continuous transfer function using the portion of the total duration as parameter $t \in [0, 1]$. The shape of the transfer function can be designed to match the scene and desired speed behavior. One function was created by merging pair of easing functions to create a dampened motion at the start and end of a segment. An example of such a transfer function using cubic easing is shown in Figure 6.3.

To compute the traveled distance, the speed function must be scaled such that the distance traveled by a particle along a path matches the integral of the speed along that path. This is achieved by ensuring that the speed function $\sigma(t)$ matches the constraint:

$$L = duration \cdot \int_0^1 \sigma(t)\,\mathrm{d}t, \tag{6.4}$$

where L is the total length of the path and *duration* specifies the desired duration of the motion. Based on the constraint the scaled speed function $\sigma(t)$ for $t = t_s$ can be computed as

$$\sigma(t_s) = \frac{L \cdot \bar{\sigma}(t_s)}{duration \cdot \int_0^1 \bar{\sigma}(t)\,\mathrm{d}t}, \tag{6.5}$$

where $\bar{\sigma}(t)$ is the unscaled speed function. The integral in the denominator of Equation (6.5) can be pre-calculated. If no closed-form solution exists it can be approximated by a sum using numerical integration.

Thanks to the speed function scaling proposed by Eberly, this approach takes the camera to the target at a specified duration. The numerical integrations introduce an error that can lead to the camera reaching the end too early or not at all. The latter case happens when the speed function becomes zero before the path is finished. To make sure that the curve always reaches the end the speed was set to a constant minimal non-zero value. This leads to a discontinuous acceleration, but was more preferable than not reaching the target at all.

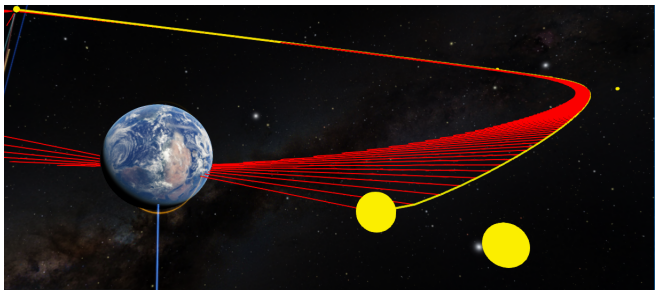

Figure 6.4: Example of a rendered path (yellow) with control points shown in yellow and the camera view direction in red. The camera is oriented to look at the Earth during the beginning of the motion.

6.3 Searching the Scene Graph

When generating the path it can be useful to have knowledge of the positions of other objects in the scene. Unfortunately, OpenSpace has no implemented method for searching the scene graph based on node positions. Moreover, the scene graph is not sorted in a way that matches the topology of renderable objects. For instance, a planet and its moons are siblings in the tree [5]. Iterating through all nodes in the entire scene graph multiple times during run-time is not feasible, especially since not all nodes correspond to a rendered object in the scene. However, the scene graph nodes do have identifying strings, so called "tags", specifying the type of the node or a group it belongs to. Some examples are tags for moons, orbit lines, planets or spacecrafts.

The above mentioned tag system was utilized to generate a reduced list of relevant nodes at start-up to reduce expensive on-line searches of the entire scene graph. This list can then be iterated over during run-time without significantly affecting the application frame rate. A scene graph node is only added to the list if it matches any of the tags considered relevant for collision checks and if it corresponds to a rendered object with a bounding sphere larger than zero. For this project the planets and moons in the solar system were considered relevant, but additional tags can be added by the user.

6.4 Rendering the Path

A generated path can be rendered by sampling n subsequent positions along the curve and then render line segments between the sampled points. Thus, the result is an approximation of the curve with increasing accuracy for larger n. The user can also choose to include the camera view direction at the sampled points of the rendered line. The view direction is then rendered as a line from the point on the path in the direction the camera is facing. The appropriate length of these lines varies between cases, due to the sparsity and scale differences. The curve control points can also be rendered as spheres with a changeable radius. These can provide valuable insights during the curve design process. Figure 6.4 shows an example of a rendered curve, control points and lines for the camera view direction.

7 Path Generation

A path is represented as a list of path segments. Each segment is independent of the other segments in the path and stores information about:

- at least two waypoints (start and end) that the camera shall pass through
- duration for the motion along the segment
- the speed along the segment
- interpolation of camera position (a spline curve)
- interpolation of camera orientation

Each instruction in the path specification is used to create one path segment. All types of instructions provided by the user are translated to a start and end waypoint for a segment. The start waypoint of the first segment in the path is created from the current state of the camera, unless a specific start state was specified by the user. As mentioned in the previous chapter, the user also has the option to specify the duration. If not specified, the system computes a suitable default duration based on the length of the path.

The methods for controlling the speed, camera position and camera orientation are implemented as procedural strategies in the system. A combination of three strategies defines a new *curve type* that the user can choose from when creating a path.

The system is designed to support any evaluation type for interpolation of position and orientation as long as it matches the start and end waypoints of the segment. Notice that the combination of position and orientation should create a motion that a human observer will perceive as natural. One can not be designed without considering the other. The position has a dependency to collision handling and orientation interpolation should aim to both reduce the angular velocity and keep objects of interest in view. The system can be extended with new curve types and different methods for spline evaluation, orientation interpolation or speed control. This made it easier to explore and compare different methods. It also allows other developers to create their own paths in the future.

When a path is created the shape of the curve is the first thing being generated. Then follows the method for speed and orientation. A reason for this order is that the algorithms for controlling speed and rotation might require evaluation of positions along the path.

7.1 Generating a Spline Curve

Given the list of intermediate way points a spline curve can be generated to represent the position for the camera. The path design process includes making a choice of what spline to

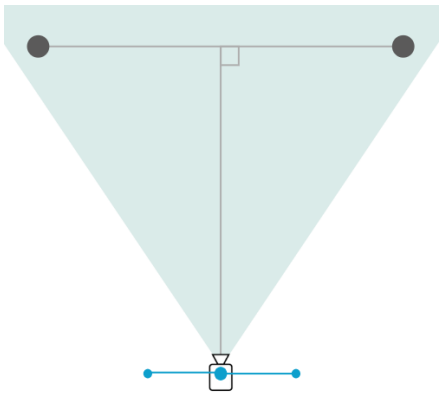

Figure 7.1: Extra control point generation to show an overview of two targets. The control point as well as its related tangents used for the Bézier spline is shown in blue.

use and how to generate its control points and possible tangents using the existing information in the scene. It is also possible to use the information stored in each waypoint, such as the camera orientation in that point as well as information about the associated scene graph node, for the control point creation.

When designing a path it is important to consider the choice of spline and what effect it has on the resulting curve. The start and end of the curve has to look natural in relation to an object, which can be easier to design using a curve with local control. However, spline curves with global control in general yield a smoother and more natural path. Furthermore, the curve should preferably have a few number control points, to avoid introducing unnecessary turns that might be a source of confusion or motion sickness. Keeping the path as simple as possible, while still fulfilling its purpose and yielding pleasant and interesting motions, has been the motto of the curve design for this project.

The simplest case of a spline is a linear interpolation between the position, which was implemented as a useful test case. This approach does not create a particularly interesting motion and does not consider possible collisions. Two main types of spline curves were implemented to address different scenarios. The first is mainly focused on creating interesting motions between two targets in the scene, using Bézier splines. The second curve was more focused on avoiding collisions with nodes in the scene.

Bézier Spline for Controlling Curve Shape

Bézier splines are often used for animating motion due to their intuitive representation. When changing one control point the effect on the shape of the curve can easily be anticipated. This property made them suitable for a first attempt of creating camera paths and studying desirable shapes for the curve. When designing these curves the use case focused on was going between two visible targets. This was considered the most probable use case for the system and an appropriate starting point.

The idea was to create control points and design the tangents of the Bézier spline based on the properties of the targets, such as its position and size. The position of the camera in relation to the target node in combination with its direction in the start and end state can be used to create tangents that generates an outward and inward motion for leaving and approaching the targets respectively. By setting the tangents based on the position and size of the start and end node it is also possible to reduce the risk of collision with these nodes. Additionally, extra knots can be added to further control the shape of the curve. For one of the generated paths a control point was added in the middle of the curve. The distance was set so that both

targets are visible for a short time, as long as the orientation of the camera is set to look in the direction of the targets (see Figure 7.1). The aim of this motion was to provide a sense of the current frame of reference and the distance between the targets, similarly to the zooming out often executed by OpenSpace pilots when manually navigating between targets.

When extra knots are inserted to create a composite Bézier spline, the tangents at these knots must be manually manipulated to guarantee C^1 continuity. This is done by ensuring that the two tangents connected at each knot are parallel to one another. Furthermore, the shape of the Bézier curve largely depends on the length of its tangents and it is important to consider the scale differences when setting these values. In other applications the tangent lengths are often set through dependence to the other knots in a composite spline. The scale difference might result in dependencies between segments with a scale difference of one to a million, thus giving tangents with extremely different lengths. If a tangent is too short in relation to the rest of the curve it might result in a very sharp turn or a negligible effect on the curve shape. An extremely long tangent might instead make the curve bend in undesired ways.

Avoiding Collision

The above mentioned curve approach works well for travel between different planets in the solar system. However, it does not take into account any other nodes for the path creation apart from the start and end node. Even though space is sparse, objects in the scene are often positioned in clusters due to the effect of gravitational forces. Common examples are moons or spacecraft orbiting a planet. Within these clusters there is a risk of collision that should not be neglected. Especially problematic cases are smaller targets objects located close to the surface of a celestial body, such as the International Space Station (ISS) or the rover Curiosity on the surface of Mars.

To reduce the risk of collision, a path curve using a naive but simple approach for collision avoidance was implemented. The idea was to approximate a final Catmull-Rom spline using straight lines and recursively create extra lines to avoid collision. Given a list of at least two points, the algorithm will recursively add intermediate control points on the side of the object the line would otherwise intersect (see Figure 7.2). To avoid passing too close to a node, a buffer is added to the bounding spheres when testing collision, as long as none of the end points of the lines are located inside the buffer sphere. This approach for collision avoidance was chosen for its simplicity and does not completely guarantee a collision free result. For example, no check is done to ensure that the generated control points are not created inside another node. Due to the sparsity of the environment the risk that this would happen is however relatively low, even for nodes in close proximity to one another. The largest risk is for paths to or from a small node that is located close to or on the surface of a much larger node.

The chordal version of Catmull-Rom was chosen to create a rounder shape for the curve, which yields a more interesting motion. As shown in Figure 3.4, the chordal version can deviate a lot from the control polygon, especially for longer segments. To avoid this, extra control points were added in the direction of the node normal at the start and end respectively if the position was close to the associated node. This results in shorter segments and also creates a motion that moves in a straight line when approaching or leaving a target, which increases the chance of keeping the node in view for a longer time when interpolating the orientations.

7.2 Camera Orientation Interpolation

The camera orientation during path traversal should be controlled in a way that shows the observer something interesting for as long as possible, without introducing rapid rotations

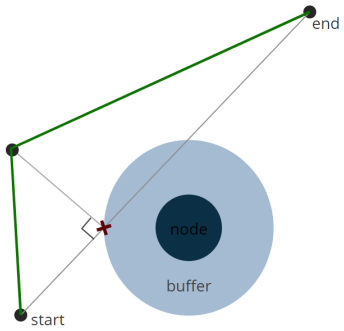

Figure 7.2: Illustration of how collision free line segments can be computed by inserting new points at the occurrence of a collision. The red 'x' marks the point of collision with the buffer sphere of the line between the start and end point.

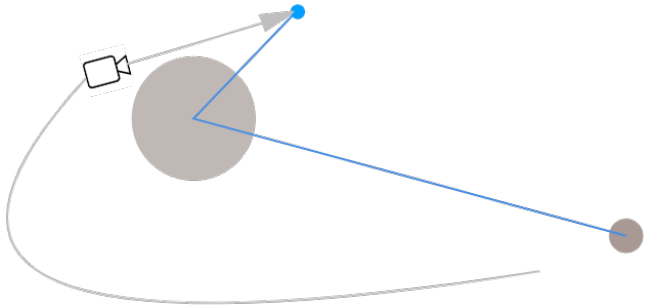

Figure 7.3: In the *look-at orientation interpolation*, an aim (blue dot) for the camera to look at is interpolated along the blue line to decide the orientation of the camera along its path.

and rolling motions. It should also avoid gazing out in emptiness for a large portion of the motion. Transitioning between two different targets as pleasantly as possible turned out to be one of the most important features to consider. Other algorithms for camera planning are often focused on the same target through the whole path, or simply looking in the direction of movement.

Due to the sparsity of the environment it can be assumed that many paths will have a part where the observer has no clear frame of reference to another object. Many targets are too far away to be visible, which can make it difficult for an observer to anticipate and understand the rotation that aims the camera towards a target. As this could induce motion sickness it might be preferable to apply a slow and steady rotation to the camera. For this a simple spherical linear interpolation (SLERP) of the start and end orientation quaternions was implemented. To increase the time the camera is looking at a target, the orientation is kept static for a while in the beginning and end of the path. This approach is suitable for paths where the camera approaches the target position from the direction opposite of the target view direction. This means that the node will be in view while the camera is approaching it, if the end state is set to look at the target node.

Another approach to set the orientation is to control what the camera is looking at using a look-at computation (see section 3.1). This might be a more suitable approach for an unknown curve shape. The look-at approach was used to create a transition of the camera view direction by aiming the camera towards an invisible aim, which position is interpolated between a sequence of positions. The aim path is here designed with a piece-wise linear spline and can be naturally be extended to include multiple targets. During a path traversal both the camera and the aim are moved along their respective path using the same interpolation

parameter. By combining the interpolations this approach lets the camera leave or approach a planet while keeping it centered in view and turn towards a next target in a smooth turn. An example is shown in Figure 7.3 where the camera will follow the gray path while the aim position is first interpolated towards the center point of the start target from a point far away in the view direction. This allows for a tilted start or end camera state, where a possible target is not centered in view. After this both the camera and aim move towards the end target.

Optional Roll-rotation

While there is no upside down in space by definition, the human observers have an illusion of a gravitational direction. The roll rotation defines the up-direction of the camera and is handled separately to match paths with different scenic characteristics better. Keeping the roll of the start state will minimize the angular rotation and might reduce motion sickness. However, when using a navigation state as input, this can make the scene appear upside down or sideways on some displays and possibly create confusion. Roll removal was added as an option that is enabled per default, which gives the pilot the option to choose.

The roll is removed from the resulting orientation by enforcing a constant up-vector for the camera. Given the new camera orientation computed for a frame, a position in front of the camera was computed using the resulting view direction. This position was then used in a look-at operation to compute a new, roll free, quaternion by using the old up-vector of the camera as up direction.

8 Results

The final result is a system that can take in a path specification, create a path from that specification and move the camera along the path. The path consists of multiple continuous segments, often between two targets in the scene. In addition, the user can choose to pause the path in between segments and either freely navigate in the scene or apply an orbiting motion around the current target node.

The system is implemented as a module in OpenSpace and is designed to be extendable. Two example curve types were implemented with individual main focus and different approaches for rotation: the *Zoom Out Overview* curve and the *Avoid Collision* curve. Additionally, the part of the system interpreting the path specification can be extended to support new types of targets and instructions. Finally, additional behaviors for pauses, or motions in relation to a target, may also be implemented.

8.1 Providing Input

A path can be created by providing input to the system through a path specification, which is essentially a table that includes the following fields:

- *Instructions*: A list of instructions about destinations and targets,
- *StopAtTargets*: (optional) A boolean specifying whether the camera should make a stop at intermediate destinations.
- *StartState*: (optional) A navigation state that determines the start state for the camera path.

The system supports two different types of instructions for a destination: either a target node in the scene or a navigation state. In the first case it is also possible to specify a height above the bounding sphere of the targeted node, or a position in relation to a node. For each instruction it is also possible to specify whether the path should make a stop at that destination, and what behavior that should be applied during that stop, if any. As mentioned in section 6.1, two types of pause behaviors were implemented. It is also possible to specify a duration both for the motion and for the stop. If no duration is specified for a stop the path can be continued by pressing a button on the keyboard.

In addition to the path specification the system allows the user to set the desired default values through the property system of OpenSpace, where the properties are accessible through the GUI of the application. Through this system the user can specify what curve type to be used, default behavior when stopping at targets, whether roll rotation shall be included, what tags to be used for finding relevant nodes in the scene, as well as some parameters for controlling the speed.

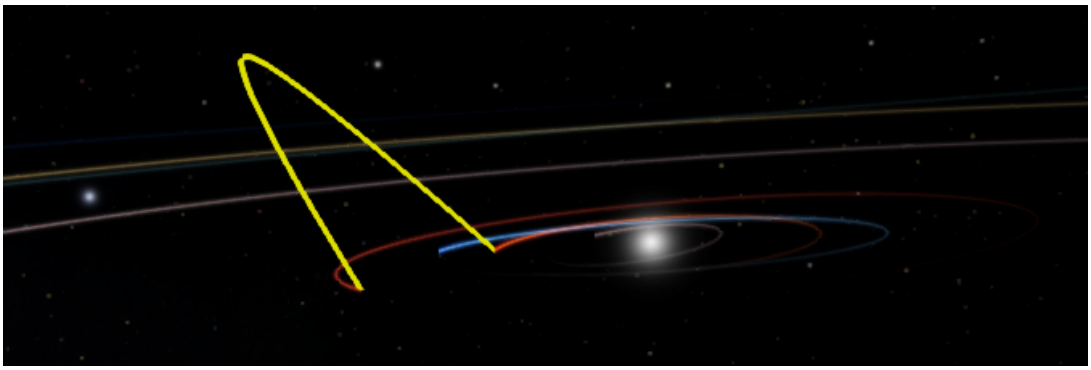

Figure 8.1: An overview with both targets in the camera frustum is given at $t = 0.5$ in the *Zoom Out Overview* curve type. The curve type has the same general shape between any targets and gives the observer a sense of the targets relation to each other.

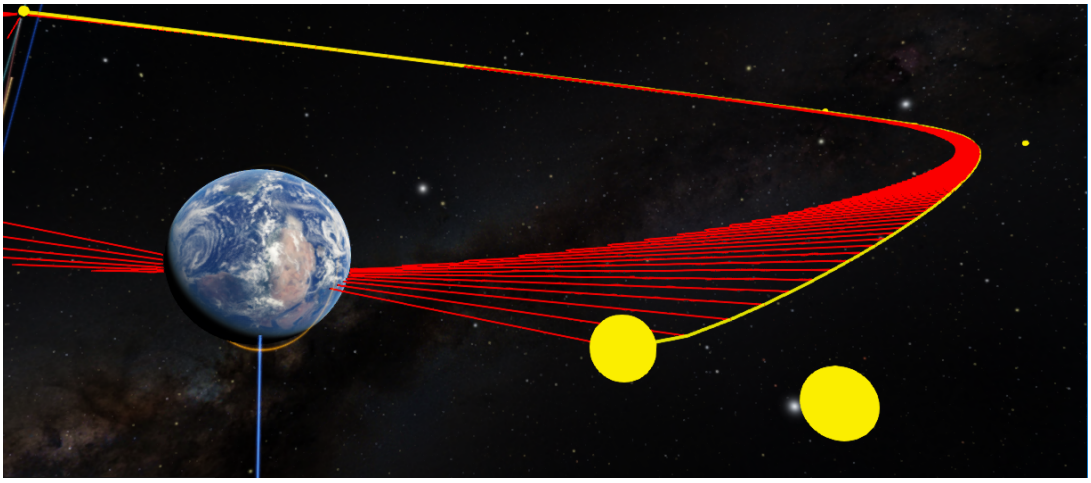

Figure 8.2: The orientations (in red) are aimed using the look-at approach along the camera path (in yellow together with its cubic Beziér control points as yellow dots). The targets are centered in view when close and the outwards shape reduces the angular velocity when changing target.

8.2 Curve Type - Zoom Out Overview

Instead of minimizing the distance, the *Zoom Out Overview curve* creates an outward motion to a camera state where both targets are included inside the frustum using a cubic Beziér spline. The curve is designed to match the look-at orientation interpolation method described in section 7.2 and removes the risk for an extreme angular velocity in the middle of the path. In Figure 8.1 a path between Venus and Mars can be seen. Even if the targets are too small to be visible from the furthest point their position is understood by the orbit lines and the zooming out motion also presents an overview of the solar system, creating something to look at through the whole path. For the spline curve a middle knot is translated 1.5 times the distance between the targets from a point between the targets. The tangents of the middle knot are parallel to the line between the targets, with lengths 0.2 times the length of that line.

The start and end tangents are set so that the curve avoids collision with the targets. An example of when the camera leaves Earth can be seen in Figure 8.2. In the figure the start view direction centers the planet but both tilted start and end navigation states are supported. The look-at aim is interpolated from a point in the the start view direction to the start node center

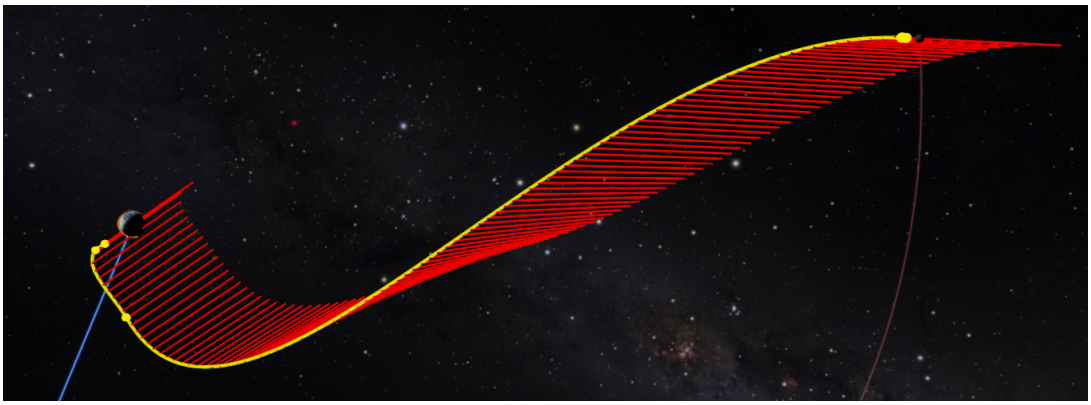

Figure 8.3: An *Avoid Collision* path from Earth to Moon, rendered using 100 sample points with lines representing the orientation (red). The third control point (yellow) from the left is generated as a result of the collision avoidance algorithm, trying to avoid a possible collision with Earth.

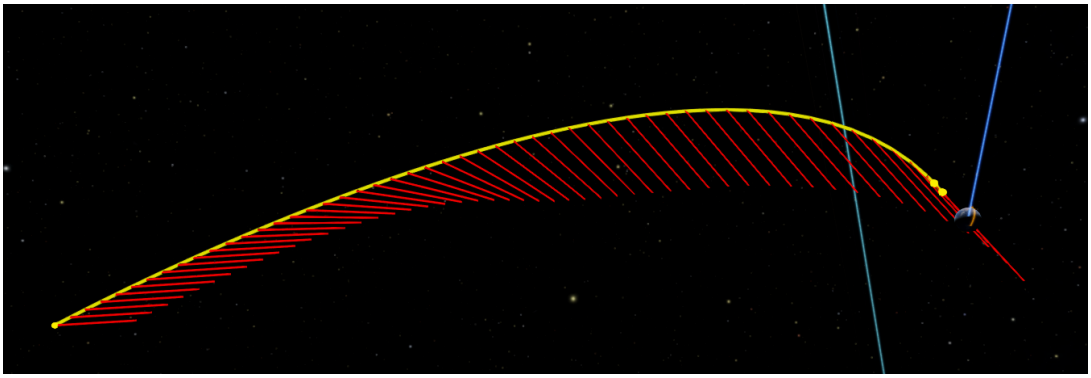

Figure 8.4: An *Avoid Collision* path to Earth from a further distance. This is an example of a case that is not well handled by the Bézier based curve. For the generation of that curve it is assumed that the path goes between two different targets and the path would start with a zooming out motion.

in $t = [0, 0.2]$, then towards end target in $t = [0.2, 0.8]$ and lastly to a point in the end view direction from the end camera position.

The curve type is designed mainly for a case taking the camera between overviews of two different planets. Exceptions creating strange shapes for this curve type are more likely to occur in less considered situation such as target objects located close to or on the surface of a planet, going to the opposite side of the same planet or going from a very distant target. These could to some extent be accounted for by setting start and end tangents with more consideration.

8.3 Curve Type - Avoid Collision

The second curve type, called the *Avoid Collision* curve, is applicable in more general scenarios and not just between two different targets. It also reduces the risk of collision at the start and end of the path and avoids passing too close to objects in the scene. Chordal Catmull-Rom is used to interpolate between a set of generated control points, yielding a bending shape over long segments, and the orientation is controlled using the SLERP approach. If the start or end position is within a certain distance to the node, an outward or inward motion is created

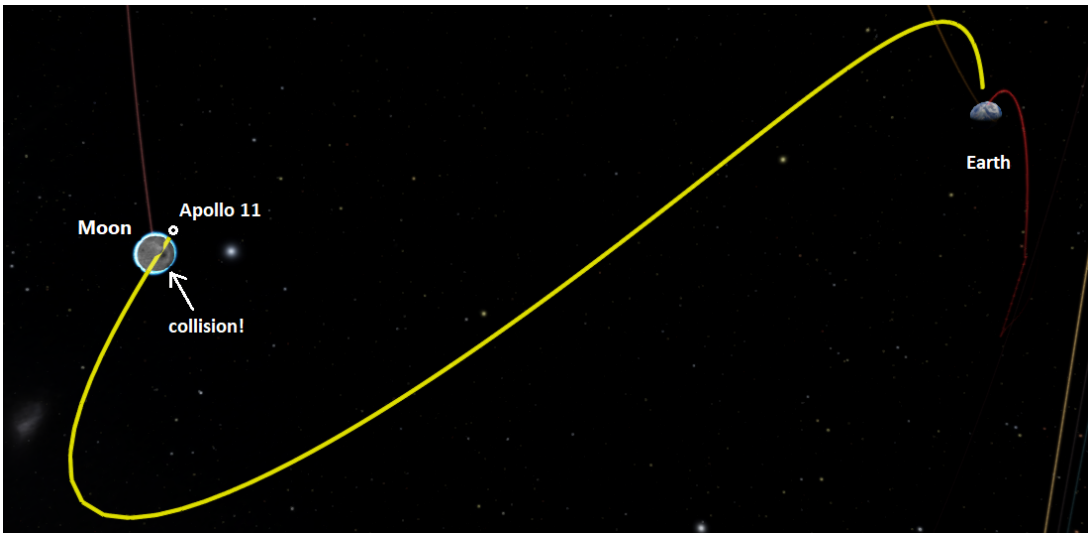

Figure 8.5: An example of a problematic case where the collision avoidance still might fail. The system is asked to go to Earth from looking at Apollo 11 spaceflight, with the moon in the background. One first control point is added to make the camera back away from Apollo 11, but due to the large distance to the next control point (close to Earth) the chordal Catmull-Rom evaluation overshoots in a way that shoots the camera through the Moon. The resulting collision is a bit difficult to see in the rendered path, but is marked in the image.

by adding an extra control point. Also, additional control points are added recursively until no part of the control polygon for the spline is passing too close to any relevant node in the scene. This further reduces the risk of collision.

Figure 8.3 shows an example of this curve type when going from a view of Earth to the Moon and creates an extra control point to avoid passing too close to Earth. Notice how the curvature of the longest segment is higher than for the shorter ones; a property of the Chordal Catmull-Rom splines. Furthermore, Figure 8.3 shows how the view direction is kept static for about a tenth of the path in the beginning and end.

This curve type is useful in the cases where there is no clear starting target. An example is shown in Figure 8.4, where the camera motion is created to move from a state far away from Earth to the planet. Notice how no extra control point is created at the start of the path in this case, so that the camera moves straight toward the target without any initial backward motion.

As shown in Figure 8.5, this curve does not guarantee completely collision free paths. The most problematic case is when the camera is located between two scene graph nodes that are in close proximity.

8.4 Further on the Look-at Orientation Interpolation

The look-at orientation interpolation approach has been tested in combination with other curve shapes as well. A case where the approach gave a strange result is when the camera traveled quite directly towards a new target and the aim position caught up, which creates a 180 degree turn in just a few frames, as in Figure 8.6. A less pronounced version of this is also created when the camera is going in the direction of the next target and keeps the aim on the first target slightly too long. As the aim is not dependent on the camera position, this mismatch can occur.

With a well matched timing a nice looking result can be achieved even for a less curved path. One of these cases can be seen in Figure 8.7. For this shape, which is more similar to the motion of an airplane, it looks more natural if the view direction is not too different from the direction the camera is headed. A note is that the motion will still look nicer if the path is slightly curved when matched with the look-at, as the scene easily becomes very static when moving along a straight line.

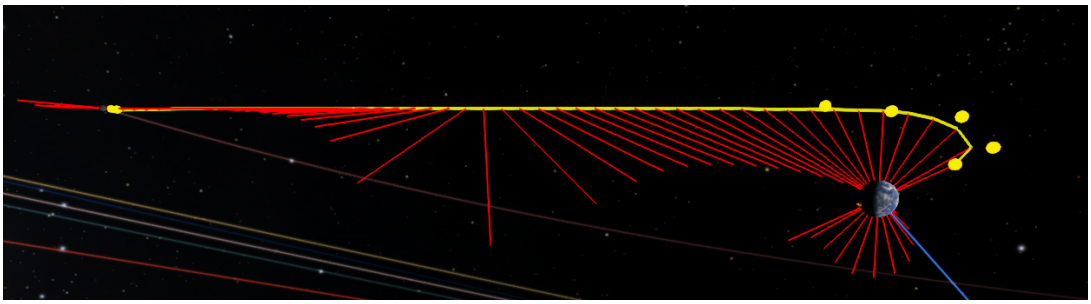

Figure 8.6: The current look-at orientation is risking to create a quick turn if used without well-defined timing. Here the aim of the camera is kept on Earth for too long on the way to the Moon (left), creating a quick rotation towards the Moon in the middle as the aim position is interpolated between the two targets.

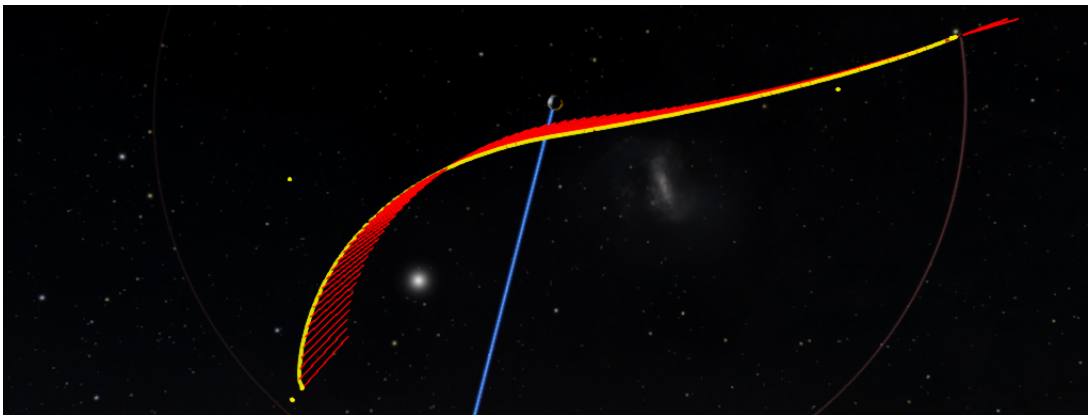

Figure 8.7: The look-at interpolation approach can also be used look at targets while flying past them with a timed interpolation of the aim, as in this case.

9 Discussion

There is always room for improvement, as well as things that could have been done differently. In this chapter, the result and methodology of the project is discussed and evaluated.

9.1 Results

The implemented system lets users specify the details for their desired paths and successfully creates a path matching the specified constraints. It provides the fundamental functionality for automatically moving the camera and also allows other developers to experiment with creating suitable paths and extend the system. The usability of the system in its current state is however limited by the fact that none of the implemented path types yield a result that works well enough in all cases.

The system was chosen to be designed in a way that lets the users choose a curve type, which is a well matched interpolation of the speed, orientation and position. As combining different interpolation approaches is unlikely to yield a good result in all cases, making this limitation more helpful than limiting for a user. Both curve types in the end result are designed to match the case considered to be most common: going from an overview of one object to an overview of a second object. There is currently no curve focused on other common situations, such as going to or from the surface to an overview, travel along the surface or going to or from a point very far away from any target. This is not necessarily a disadvantage. One curve type covering all cases proved to be difficult to develop as handling exceptions and special cases often lead to digging one hole to fill another. It is presumably easier to extend the solution with curve types designed for more specified intended use cases. The user, or even the system, could then be let to choose between them depending on the current scenario. Aspects that are interesting to consider for curve types in general includes the distance to the nodes, light direction and the disc formation of solar systems and galaxies. All of these have an impact on the visibility of either the targets or of the surroundings.

As shown in the result, there is still some risk of collisions and extreme motions for the generated paths. One possible way to increase the usability of the system would be to analyze the created path and inform the pilot about whether the generation of the path was estimated as successful or not. The pilot could then be given the choice to accept the path or discard it and create a new one. In some cases, sightly moving the camera and call for a new automatic path provides a much better result. The estimate of success could also include more precise collision checks, curvature thresholds, measured angular velocity or even the rate of change in cheaply rendered images. It is of course also interesting to keep developing the curve types to achieve a better result.

The final system includes no path to support the use case where the camera is steered through multiple waypoints in one continuous motion. This kind of motion should be implemented to provide further creative freedom for a user when for example pre-defining paths to be

used for creating a movie. These pre-defined curves makes it possible for the pilot to ensure a high quality of the path, which is a benefit as the pilot is likely to have a better judgment than the computer. The possibility of pre-defining and saving paths for later use also encourages collaboration by supporting sharing of camera motions between OpenSpace-users. An important improvement related to this would be to add the level of detail needed for instructions to enable reusing the path as intended at a later time. As of now, the simulation time is not kept track of in any way in the specification or representation of the path. This can lead to unpredictable results when creating a path from the same instruction multiple times due to the possibility of changes in the scene. It is for example simple to forget that a scene might not be in daylight at a later time, that constellations change or that the camera start state might be different.

Curve Types

Currently, the zoom out overview curve type is not setting the shape in regard to anything but the two target nodes and will by chance look in a direction showing more or less interesting views of the surrounding system. Making some assumptions on secondary points of interest could greatly improve the intermediate views of the simple path shape. For examples, pointing the view direction towards the solar system gives the observer something visual to look at during the middle part of the motion, compared to the emptiness of space. In the case of the solar system it might also be beneficial to rely on the up direction of the scene graph node, as this is likely to provide knowledge about the level of visibility of the disc formation. The view direction can be set to look towards something interesting by changing the aim or camera spline curve, or the timings of the interpolation along them. The implemented look-at approach is adapting to the given path and a better result could be acquired by changing the path shape rather than adding the extra interest point to the aim spline curve.

The avoid collision curve type will often fly by next to the targets in motion not too different from how one could expect a plane to travel. Initially the curve has been designed to use SLERP as orientation method, but it may be possible to match the curve with a look-at orientation with better timing for the aim point. The appropriate timing for when the camera orientation should be changed from looking at the start node to look at the next largely depends on the shape of the curve. Consider for example the case shown in Figure 8.6. The SLERP approach works well in most cases, but has the potential risk of moving the targets out of focus for large part of the motion. The look-at approach prevents this by ensuring that the camera keeps an interesting object in view for as much as possible. The look-at orientation also result in a motion that might be considered more predictable, since it becomes more obvious what target the camera is moving in relation to. In conclusion, further experimentation with the look-at approach for this curve type could be of interest.

Result Evaluation

The evaluation of the result has been made based on own observations. A next step would be to conduct user studies with the aim of getting a better understanding of the quality of the result. In the end, the human observers' evaluation of the path is what matters when defining a path as successful. With limited knowledge of how the information on a screen is processed by the observers, asking the observers of their experience by questionnaires or sensors remains the best way to evaluate the path. In practice it might be possible to create a fair evaluation of a path by using a combination multiple factors, such as visibility of objects of interest and camera movement in relation to objects. In the best case these measures can also be extended to consider the varying sensitivity in the human visual system.

A possible user study to evaluate the quality of the resulting curve types would be to compare the automatically generated paths to the motion created through the manual navigation

by an experienced pilot. The subjects of such a study could be asked to evaluate the different motions without knowledge on which motion was created by the system or the pilot. This could also potentially give a better understanding on what navigation actions are better handled by a pilot and where the automatic system might perform better.

9.2 Method

From the beginning the system has been designed to include support for a sequence of separate paths and with varying types of inputs. The outcome of this is a widespread system with lots of functionality, but where no functionality is perfected. In hindsight it could have been better to implement one curve for one simple use case, and do it well, before continuing to design the full system. However, there is a possibility that the amount of time spent on analyzing the needs of navigators, identifying use cases and trying to define an appropriate camera motion would have been significantly reduced in that case. In the end this was a large and important part of the project. The chosen approach also lead to some other advantages. For example, defining a series of navigation states as input to the system ended up being useful to identify challenging cases for different curve types.

Speed and Motion

A challenging part of the project was controlling the camera along the path and moving with an appropriate speed. The scale of environment lead to challenges in terms of moving the camera by using numerical integration schemes without introducing errors, but also how to make sure that the camera moved in an appropriate speed in relation to visible objects.

The motion along the path was not perfected in the time frame of this project and can be improved further. While the proposed method creates a smooth motion that reaches the end in a given duration, it is difficult to get precise control of the speed where it actually matters. In addition to the numerical value representing the movement speed, the perceived speed in the sparse scene also depends on the relation between the size of any visible object and the distance to that object. An ideal speed control method should take this into account. Moreover, the speed function scaling depends on the total distance in relation to the specified duration for the motion. This often leads to too large speed values close to objects, especially when traveling between objects of different scale and at a large distance. Furthermore, the usage of numerical integration leads to a risk of errors. These emerge as discontinuities in the speed of the camera, or an error in the time it takes for the camera to traverse the path. One possible way to reduce these errors would be to use a more precise method of integration for the path traversal, such as some order of Runge Kutta. However, this would also introduce more expensive computations during run-time.

One possible option for traversing the path would be to use a force based method to move the camera, similarly to the approach by Geraerts and Overmars [15]. In that case, the path could act merely as points of attraction and the final motion would not be as sensitive to quirks or discontinuities in the path shape or from the speed function. Furthermore, repelling forces could be added to objects in the scene, making it easier to avoid collision. Another approach might be to adapt the choice of speed method based on the scale of the distance to be traveled, as suggested by Li et al. [6]. The speed might also be computed based on the distance to the closest objects in the scene. This solution was initiated but not included in the result as it still requires better smoothing for the start and end to be used. As of now it is also dependent on on-line computations done during the path traversal.

OpenSpace aims to support pre-defined paths and reduce online calculations as much as possible. One possibility would be to pre-traverse the path and sample the speed along the path and then translate the speed values to a transfer function. Pre-traversing the path without

rendering it is not expensive and can also be done using a small step size. It is possible to use a low-degree C^2-continuous spline to approximate the samples values of the relative length traversed along the path. Another benefit of translating for example a force simulation to a transfer function is that it becomes easier to ensure optimal smoothing for the start and end of the path. This could possibly be achieved by applying an easing function to the transfer function.

Rotation

The implemented approaches for rotation does not provide a satisfactory result in all scenarios. It proved difficult to control the orientation in a way that looks at something interesting for as long as possible without introducing undesired fast rotations, especially when the shape of the position spline is not known. For most of the observed cases it ended up being more important to position the targets in the view in a predictable way rather than keeping a low angular velocity. This is however not the only way to get an acceptable result.

The SLERP approach guarantees a minimal change in angle and thus angular velocity, but it has no guarantee of keeping the scene objects in view. The look-at approach guarantees that the camera is aimed towards an object of interest for most of the motion, but has no restrictions on angular velocity. It sometimes creates too fast rotations, for example if the path curve coincides with the curve for the gaze position. A possible improvement would be to further develop the look-at approach to rotate toward a visible node only when the camera is within a certain distance from the node. In the other case the camera could possibly be oriented to look at a future position along the path. This could give the observer a better sense of the direction of movement, while still aiming to look at something interesting when relevant. Another interesting approach would be using a force based model where attracting forces are applied to the aim of the camera close to objects in the scene. It would then be possible to restrict the angular velocity to a maximum value.

By extending the existing look-at solution with more clever timings for the aim interpolation it would be possible to better match the positioning of the camera. The approach would then have the potential to be a solution acceptable for use with curves of very different shapes. A useful rule of thumb is to set the aim in the direction of the target of which the camera is moving mostly in relation to. The camera should not be clearly heading towards the next target before the aim does.

As of now the roll rotation is either excluded or part of the full orientation interpolation. Another possible approach is to completely separate the roll from the aim interpolation to get more control. This way, if roll is desired, the up-vector could for example be interpolated separately in a way that minimizes the angular velocity as much as possible. This could be useful since humans seem to be more sensitive to fast changes in roll rotation. Different options for the roll rotations have not been investigated further, but might have a big impact on the quality of the path. A difference of how the roll should be handled best might also depend on if the camera motion is similar to a flight of a plane, traveling mostly in the forward direction rather than sideways. Associations for natural behavior of the camera movement could differ.

Curve Generation and Collision Avoidance

The current implementation of the system introduces some limitations on how the path can be created. An example is that the end state is determined ahead of the curve generation process. The end state of the path has a large impact on the final shape and the ideal end state might vary between the different curves. Therefore it would be interesting to allow the curve creation process to determine the end point, depending on how the shape of the curve turns out. This has the possibility of generating more well shaped paths where the curve can avoid

making unnecessary detours due to a fixed target state. A possible implementation would be to give the curve the possibility to re-set the end position during its creation, if no exact end state is requested.

The spline types chosen for the implemented path types have some drawbacks and other options should be investigated. The Catmull-Rom spline is forced to pass through all control points and can introduce undesired bumps in the path if the control points are not generated with caution. Furthermore, as shown in the result, the chordal version of Catmull-Rom is not providing a sufficiently precise approximation of the control polygon to completely guarantee collision free paths. An interesting alternative would have been to investigate the use of PH splines. This spline type is claimed to result in a smooth curve with minimized curvature [19]. This would beneficial for accurately approximating the control polygon without the issues with overshooting or sharp turns as of the different Catmull-Rom versions. PH splines are also known to be more efficient than the regular polynomial spline types and supposedly more suitable for real-time use [25].

Regarding collision avoidance, more sophisticated path finding methods could also be investigated. Such methods could benefit from a spline type that is constrained within a convex hull, such as Bézier splines or B-splines. One interesting approach to investigate further would be to create collision free corridors rather than just lines, as proposed by Geraerts and Overmars [15] and Choi et al. [17], and make sure that the convex hull of the spline is constrained by the corridors. Another option would be to improve the current approach by making sure that none of the recursively generated control points are introducing collisions. One possibility would also be to analyze the risk of collision along the curve, for example at the end points, and apply a more clever approach in scenarios where the risk is high. Another simple solution of guaranteeing that the final path is free from collision would be to keep the linear segments and achieve C^1 continuity by creating circular arcs in the intersections, as proposed by Nieuwenhuisen and Overmars [14]. This would however result in a less curved path and possibly lead to a less interesting motion.

Applying more sophisticated path finding methods would require efficiently analyzing the scene. The current method for finding relevant scene graph nodes works well for scenes with a limited number of renderable nodes. However, the time complexity for searching the list of relevant nodes increases proportionally to the number of nodes in the list. If a large number of nodes are included in the scene it might be useful to investigate more efficient methods or possible representations of the scene.

The camera motion should aim for continuity in terms of both speed and acceleration. A smooth camera motion does in this case not require the spline itself to be C^2 continuous by definition, since the speed of the motion is separated from the spline representation. Combining speed functions and spline curves of higher continuity does however create a more pleasant curvature for the path as a whole. Furthermore, minimizing the curvature is preferable to reduce the risks of sharp turns and overshooting that comes with the extreme scale differences. As already mentioned, PH-curves have potential to provide improved results compared to the Catmull-Rom splines. Another option to investigate is NURBS, which have a smoothly varying curvature and also have the potential to represent circular or spiraling paths.

Rendering of the path curves, control points and orientation proved to be a helpful tool for the curve design process. It could have been developed further to become more efficient by including more parameters that can be adjusted during run time. The scale difference requires different sizes for overview and close up views for a smaller part of the path. The rendered paths might also prove to be useful to display multiple options to a pilot.

Creating a More Intelligent Path

An interesting option for creating a better adapted curve type would be to investigate solutions from the field of AI, which was delimited in this project. A more intelligent automatic path might learn what is interesting and could possibly process more information of the current scene to act on opportunities not known to the pilot, such as flying by an object that happens to be on the way. Our observation is that the biggest issue for training a model to create a more intelligent path is defining a measure translating to the quality of a path.

Information needed for a leaning algorithm to identify a definition of quality could be a combination of duration, kinematic measures of the camera movement and the level of visibility of objects of interest. Information that can be translated to visibility includes the combination of the size and shape of an object together with the relationship to its primary light source and to the camera parameters. Alternatively, a single visibility measure for each object can be calculated from a rendered image along the path and be used as a parameters defining the quality of the path. Rate of change in an image may also be used, to ensure a good balance in speed. A note is that the usability of these factors depend on how well they match what an observer actually perceives. A measure of the level of cybersickness could also in theory be used as input for learning or improving paths, but there is a high risk that the metrics to measure it are yet not representative enough to ensure a high quality. It is also worth considering the ethical concern of putting subjects trough the discomfort to gather this kind of data.

10 Conclusion

The resulting system simplifies navigation in OpenSpace by allowing a user to automatically navigate to one or more targets in a smooth motion. This fulfills the main objective of the project. Furthermore, the system can be used as a tool for interactive presentations in the software. Details about one or several paths can be specified in advance and used to trigger paths at a later time, making it easier for inexperienced users to navigate and hold presentations using OpenSpace. In addition, the project provided valuable insights regarding challenges and desirable characteristics of camera motions in OpenSpace and other large scale environments. To further conclude the work, the initial research questions will be revisited and answered.

What aspects characterizes a good path, with respect to the experience of the observer?

The main aspects of an appropriate motion are to avoid inducing motion sickness, while still creating an engaging and interesting experience by not moving too slow and avoid empty or static scenes. The observer in this case can be both the pilot and one or more external viewers, whose experience could be profoundly different. The ability to anticipate the behavior for the motion is also likely to reduce motion sickness. Hence, the motion should be created in a way that the audience perceives as natural. Smooth accelerations is typical for what humans experience as natural movements, but the more semantic aspects might be individual. User studies should be performed to evaluate the quality of the resulting motions.

Furthermore, motion sickness is less likely to occur if the scene provides a steady target for the observer to rest the eye upon. This can however be difficult in a sparse scene. The visibility of the objects in the scene should hence be considered when creating the motion. Objects such as celestial bodies and spacecraft are often only visible from a close distance and from their illuminated side, while orbit lines and stars are better seen from afar. Furthermore, the celestial objects and constellations are often orbiting within a planar disc, making the system less visible when seen from the camera located in the the same plane.

Another difficulty of the sparseness of the scene is to provide a coherent frame of reference to the observer. This is necessary to allow the viewer to get a sense of the scale of the universe. One way to achieve this is to provide an overview of the current environment when moving to a new target. For travel within the solar system it might for example be useful to zoom out to a view of the orbital plane before approaching the new destination.

The sparseness and scale of the environment also impacts the appropriate speed and rotation. The distance to visible objects in the scene in relation to their size affects the perceived travel speed. A well defined path should take this into account. The speed should be as fast as possible in the sparse areas, while not being too fast close to visible objects. For a camera motion in space it looks best when the distance to objects is naturally increasing or decreasing. Regarding rotation it is desirable to avoid roll rotation and reduce the angular velocity.

Moreover, the camera should be oriented to look at something interesting in the scene to the largest extent possible.

How can spline mathematics be used to automatically generate a camera path that provides a cinematic experience for the viewer? What challenges does the large scale of the virtual space environment entail?

To move the camera to its new state, both the position and orientation of the camera must be interpolated. The positional spline curve can be generated by procedurally creating control points based on start and target positions and knowledge about the scene. A spline type with a high level of continuity should be chosen to create a pleasing experience. It is also desirable to avoid unnecessary turns and quirks in the path by keeping a low curvature. Furthermore, the combination of speed, position and rotation has a large impact on the resulting motion and experience. It is therefore important to consider the choice of rotation and speed when designing the shape of the spline curve, or vice versa, with the aim of creating a natural motion. The movement along the curve can be done using a forward-stepping method, where one or more steps can be computed per frame.

Moving along the the spline with a natural speed requires the possibility to step in relation to the arc length rather than the interpolation parameter. For most spline types, the arc length has to be approximated using numerical integration methods that both introduce an error and a computational cost. PH-curves provide a closed form solution for the arc length and might be worth investigating further because of this. Due to the large scale differences it is vital to use a method with a small error for the approximation. The error is most probable to be visible when the camera is close to an object. Hence it is often more important to minimize step length error at the start and end of the path compared to the middle.

How can the instructions on the desired path be specified? What degree of freedom should the navigator have when specifying these instructions?

The desired level of detail in the specification varies between users. The system should provide just enough flexibility to allow the users to provide input to the system without specifying more details than required to generate the desired path. The minimal amount of information is a destination, which can be specified as the name of a node in the scene. Additionally, allowing the user to add extra information about the destination or even use exact camera states as input increases the flexibility for experienced navigators.

One important aspect of defining the path is the possibility to adjust the speed to match the used display. When not knowing the length of the path it is more intuitive to scale the default speed rather than providing an exact duration. However, when creating movies it might be more intuitive to adjust the duration instead. In this case it is also useful to set a start state for the path, which the camera is instantly moved to when the path is started, rather than using the current camera position. This makes it possible to create the exact same motion multiple subsequent times, to generate exactly the desired motion. However, it should also be possible to save information about the current simulation time in the start state. This would be useful to avoid creating a different motion if the simulation time is changed unintentionally.

The system also makes some assumptions of desired default behavior, for example when stopping at a target. It is useful that these behaviors can be changed by the user on a global scale and not only per path or specification. The possibility to define default values for speed and behaviors can reduce the need of providing redundant details in instructions to for example match different environments or displays.

10.1 Future Work

The result of this project provides a foundation for generating camera paths but the usability of the system could be increased through some future work. One thing would be to keep investigating possible methods of controlling speed to come up with a solution that works well over the large distances. As previously discussed, the size and distance to visible objects should be taken into account due to the effect on the perceived speed, but also the scale of the distance traveled. An interesting option would be to move the camera using a physical simulation approach in which forces are applied of the camera. The path would then be used as a guidance for the camera rather than its exact path. This also has the potential of reduce the effect of numerical errors currently existing in the speed integration, as well as making it easier to guarantee collision free paths.

More work should also be put into improving the paths, which should involve continued implementation of possible path types. A path that interpolates smoothly between several specified camera states should be added to cover all the use cases defined for the system. Furthermore, it might be interesting to investigate paths that are specialized to be applicable close to a planet surface. One requirement for such a path would be the ability of handling terrain features on the surface, hence taking the height of the surface into consideration. Another improvement in relation to the paths would be the ability to take simulation time into account. Traveling in space and time simultaneously would be an interesting use case. This would include solving the issue of how to create paths in relation to moving targets, without introducing collisions or overhead from expensive real-time computations.

Another useful feature would be to analyze the successfulness of a generated path and allow the user to discard it if it for example contains collisions or quick rotations. The user could then slightly move the camera to try to generate a better path candidate. This would add usability to the system, since it has been proven to be difficult to generate camera paths that provide a good result in all cases.

The system is designed with the intention to enable design of a future GUI where navigation instructions can be listed and modified. The usability of the system depends on the types of instructions that may be provided. A type of instruction that might be interesting for a more advanced path could include both primary and secondary targets of interest, which could help the pilot to better express which objects are important in their story. The GUI should also include playback functionality similar to that of a video player, such as starting, stopping or pausing a path. Furthermore, it would also be beneficial to implement the possibility of saving and loading created paths, to avoid having to recompute the path when the same specification is provided. As of now, it is not even guaranteed that the same path is generated from the same specification the second time it is used. This is because the environment changes depending on the simulation time. One possible way to achieve consistent paths could be to add support for including the simulation time in the specification. However, this might not always be desirable. Also, it would require the system to change the simulation time on playback, which could lead to visual changes in the scene that might be confusing for a user.

Lastly, with more time it would have been desirable to perform user tests to measure the quality of the generated paths. One suggestion would be to let the system and an experienced pilot perform the same navigation task and record the result. The test subjects could then be asked to evaluate the quality of the two results, without being given knowledge about which motion was generated by the system. It would also be of interest to let users try out the system to make sure that it is sufficiently intuitive, especially for non-experienced OpenSpace users. However, a requirement for that would be to have at least a prototype of a graphical user interface.

Bibliography

[1] A. Bock, E. Axelsson, J. Costa, G. Payne, M. Acinapura, V. Trakinski, C. Emmart, C. Silva, C. Hansen, and A. Ynnerman, "OpenSpace: A system for astrographics", *IEEE Transactions on Visualization and Computer Graphics*, 2019.

[2] A. Bock, C. Hansen, and A. Ynnerman, "OpenSpace: Bringing NASA missions to the public", *IEEE Computer Graphics and Applications*, vol. 38, pp. 112–118, 2018.

[3] J. Bosson, "Multi-touch interfaces for public exploration and navigation in astronomical visualizations", Department of Science and Technology, Linköping University, 2017.

[4] H. Johansson and S. Khullar, "Grafiska användargränssnitt för multifunktionsdisplayer som stöder publik utforskning av astronomiska visualiseringar", Department of Science and Technology, Linköping University, 2018.

[5] E. Axelsson, J. Costa, C. T. Silva, C. Emmart, A. Bock, and A. Ynnerman, "Dynamic scene graph: Enabling scaling, positioning, and navigation in the universe", *Computer Graphics Forum*, 2017.

[6] Y. Li, C.-W. Fu, and A. Hanson, "Scalable wim: Effective exploration in large-scale astrophysical environments", *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 1005–11, 2006.

[7] S. Klashed, P. Hemingsson, C. Emmart, M. Cooper, and A. Ynnerman, "Uniview - Visualizing the Universe", in *Eurographics 2010 - Areas Papers*, The Eurographics Association, 2010.

[8] A. Sagristà, S. Jordan, T. Müller, and F. Sadlo, "Gaia sky: Navigating the gaia catalog", *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 1070–1079, 2019.

[9] J. Blinn, "Where am i? what am i looking at?", *IEEE Computer Graphics and Applications*, pp. 76–81, Jul. 1988.

[10] D. B. Christianson, S. E. Anderson, L. He, D. H. Salesin, D. S. Weld, and M. F. Choen, "Declarative camera control for automatic cinematography", *Proceedings of the American Association for Artificial Intelligence*, pp. 148–155, 1996.

[11] S. M. Drucker, T. A. Galyean, and D. Zeltzer, "Cinema: A system for procedural camera movements", *Proceedings of the symposium on Interactive 3D graphics*, pp. 67–70, 1992.

[12] S. M. Drucker and D. Zeltzer, "Camdroid: A system for implementing intelligent camera control", *Proceedings of the symposium on Interactive 3D graphics*, pp. 139–144, 1995.

[13] S. M. Drucker and D. Zeltzer, "Intelligent camera control in a virtual environment", *Proceedings of Graphics Interface '94*, pp. 190–199, 1994.

[14] D. Nieuwenhuisen and M. H. Overmars, "Motion planning for camera movements in virtual environments", Institute of Information and Computing Sciences, Utrecht University, 2003.

[15] R. Geraerts and M. Overmars, "The corridor map method: Real-time high-quality path planning", *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1023–1028, 2007.

[16] M. Christie, P. Olivier, and J.-M. Normand, "Camera control in computer graphics", *Computer Graphics Forum*, vol. 27, no. 8, pp. 2197–2218, 2008.

[17] J. Choi, R. Curry, and G. Elkaim, "Path planning based on bézier curve for autonomous ground vehicles", *WCECS '08*, pp. 158–166, 2008.

[18] V. Deshmukh, K. Deshmukh, and S. Patil, "Dynamic trajectory planning for mobile robot intercepting a moving target using bezier curve", *IEEE Students' Technology Symposium*, pp. 322–327, 2016.

[19] A. Amamra, Y. Amara, R. Benaissa, and B. Merabti, "Optimal camera path planning for 3d visualisation", *Proceedings of SAI Computing Conference*, pp. 388–393, 2016.

[20] M. Haigh-Hutchingson, *Real-time cameras, A guide for game designers and developers*. Morgan Kaufmann, 2009.

[21] A. J. Hanson, *Visualizing quaternions, Series in interactive 3D technology*. Morgan Kaufmann, 2006.

[22] K. Shoemake, "Animating rotation with quaternion curves", *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, vol. 19, no. 3, pp. 245–254, 1985.

[23] D. Eberly, "Quaternion algebra and calculus", Geometric Tools, Redmond WA, USA, Tech. Rep., 1999.

[24] C. Yuksel, S. Schaefer, and J. Keyser, "On the parameterization of catmull-rom curves", *Proceedings - SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pp. 47–53, 2009.

[25] L. Gajny, R. Béarée, E. Nyiri, and O. Gibaru, "Path planning with PH G2 splines in R2", *ICSCS 2012 - 2012 1st International Conference on Systems and Computer Science*, 2012.

[26] J. E. Bos, W. Bles, and E. L. Groen, "A theory on visually induced motion sickness", *Displays*, vol. 29, no. 2, pp. 47–57, 2008, Health and Safety Aspects of Visual Displays.

[27] L. Rebenitsch and S. Owen, "Review on cybersickness in applications and visual displays", *Virtual Reality*, vol. 20, no. 2, pp. 101–125, 2016.

[28] S. Clems and P. Howarth, "The menstrual cycle and susceptibility of virtual simulation sickness", *J Biol Rythms*, vol. 20, no. 1, pp. 71–82, 2005.

[29] J. Golding, K. Doolan, A. Acharya, M. Tribak, and M. Gresty, "Cognitive cues and visually induced motion sickness", *Aviat Space Environ Med*, vol. 83, no. 5, pp. 477–482, 2012.

[30] C.-L. Liu and S.-T. Uang, "Effects of presence on causing cybersickness in elderly in a 3d virtual store", *Human computer interaction international: users and applications, Orlando, USA*, 2011.

[31] Y. Ling, W.-P. Brinkman, H. Nefs, C. Qu, and I. Heynderickx, "Cybersickness and anxiety in virtual environments", *Joint virtual reality conference, Nottingham, UK*, 2011.

[32] S. Weech, S. Kenny, and M. Barnett-Cowan, "Presence and cybersickness in virtual reality are negatively related: A review", *Frontiers in Psychology*, vol. 10, p. 158, 2019.

[33] J. Mackinlay, S. Card, and G. Robertson, "Rapid controlled movement through a virtual 3d workspace", *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 171–176, 1990.

[34] F. Bonato, A. Bubka, and S. Palmisano, "Combined pitch and roll and cybersickness in a virtual environment", *Aviat Space Environ Med*, vol. 80, no. 11, pp. 941–945, 2009.

[35] B. Keshavartz and H. Hecht, "Axis rotation and visually induced motion sickness: The role of a combined roll, pitch and yaw motion", *Aviat Space Environ Med*, vol. 82, no. 11, pp. 1023–1029, 2011.

[36] C. Diels, K. Ukai, and P. Howarth, “Visually induced motion sickness with radial displays: Effects of gaze angle and fixation”, *Aviat Space Environ Med*, vol. 78, no. 7, pp. 659–665, 2007.

[37] D. Eberly, “Moving along a curve with specified speed”, Geometric Tools, Redmond WA, USA, Tech. Rep., 2007.