Public Dissemination of Deep Space Missions through Immersive Visualization

Anton Arbring

Master's Thesis, Media Technology and Engineering

August 2015

Examiner:	Anders Ynnerman
Supervisors:	Alexander Bock, Carter Emmart
Department:	Media and Information Technology

Abstract

This is a thesis focusing on the additions made to the open source software OpenSpace, during a semester at the American Museum of Natural History (AMNH) in collaboration with Linköping University (LiU). The goal was to be able to visualize several space missions made by National Aeronautics and Space Administration (NASA) and European Space Agency (ESA) to be able to reach a wide range of people with the incredible stories of space exploration.

Special focus in this thesis is given to extending the software in such a way that several missions can be visualized. This includes work on estimating positions for missions where the existing data of the different missions do not overlap. The main additions to the core functionality in OpenSpace was time indicating trajectories, night side texture improvement, image projections in space and image projections to irregular bodies.

Acknowledgements

OpenSpace is an open source project and there are several previous developers. The modules mentioned in this thesis as well as several modules omitted in this thesis are continuously improved and contributed to by these developers as well as the author. The full OpenSpace team can be found on the project web page [1].

I would like to express my gratitude towards LiU and ITN for the preparation I got for this thesis and beyond. I want to thank the American Museum of Natural History for hosting me in New York City. I want to show great appreciation to my examiner Professor Anders Ynnerman and my university supervisor Alexander Bock for giving me this opportunity and guiding me during the project. I want to thank Dr. Carter Emmart for the daily inspiration and vision he continuously provided. I would also like to thank Michal Marcinkowski for being helpful from day one on site at AMNH and during most of my numerous overtime hours. Last but not least I would like to acknowledge the support from my family and friends that kept me sane throughout my engineering studies.

Contents

1 Background

NASA and ESA spends huge amounts of resources to explore the solar system and to find out more about the universe, and the results of these missions does not always reach the general public to a desirable extent. OpenSpace is a cross platform interactive tool for space visualization. It can not only be run on single desktop computers but also in immersive environments and has proven to be able to network from a master computer to domes all over the world.

Apart from visualizing space missions, OpenSpace is used on site at NASA for visualizing space weather in real time, which has not been possible to achieve before collaborating with the Media Technology department at LiU. This chapter goes into giving a context to this thesis work and purpose of it in OpenSpace.

1.1 Purpose

The aim of this thesis was to continue the work on the open source software OpenSpace. The software was already used to visualize the New Horizons mission. The focus was divided between continuing the preparation for New Horizons flyby of Pluto in July 2015 and to extend the software to be able to visualize other missions, such as Dawn and Rosetta. The goal was to visualize the scientific results of these missions, in a way that is scientifically accurate and is easy to comprehend.

In order to do this, the software had to be extended to be able to perform image projection on arbitrary objects and a few new modules had to be written and improved. The thesis work also covered improving the models and the visualizations as a whole.

1.2 New Horizons

New Horizons is a space mission launched by NASA in January 2006. With a gravity assist of Jupiter in 2007, it was able to fly by Pluto on July 14, 2015, after which it will continue out in the Kuiper belt. During the flyby it managed to take high resolution images of the surface of Pluto and its moons using its main imaging instrument LORRI, far better than the ones taken from the Hubble telescope. These images are projected onto Pluto's surface by projective texturing [2].

Even though Pluto was downgraded from a planet to a dwarf planet by the International Astronomical Union (IAU) soon after New Horizons was launched, the media coverage has been quite large and its successful data gathering is, to some extent, already publicly available [3].

1.3 Rosetta

The Rosetta mission was launched by ESA in 2004 and its main purpose is to research the comet 67P/Churyumov-Gerasimenko. It is the first spacecraft to orbit a comet using ion thrusters powered mainly by solar cells. During close orbit of 67P, a lander called Philae was sent to collect data from the surface. After 211 days of hibernation it woke up in June 2015.

OpenSpace did, at the start of this internship, assume that the bodies projected to were well approximated by a sphere, which was not the case for 67P. A big part of the internship was therefore focusing on image projection to irregular bodies. Most of the images were taken recently and ESA wanted to keep them to themselves to check for possible errors but as the project went on, more images were released. The work in this thesis focused on the Rosetta NAVCAM [4].

1.4 Dawn

Dawn is NASA's mission to the protoplanet Vesta and dwarf planet Ceres. Like Rosetta, Dawn is driven by solar cells in order to be able to orbit bodies. Other factors making Dawn a priority to include in OpenSpace was that a detailed model of Vesta is available [5] and since Dawn was there in 2011, most of the pictures taken by the Dawn Framing Camera are released. Dawn also has in common with New Horizons and Rosetta that it was relevant during the time of the internship as it was approaching Ceres [6].

2 Introduction

This section covers core concepts in OpenSpace and computer graphics. Each of the subsections consists of essential parts of the implementations discussed later in this thesis.

2.1 OpenSpace

The code base of OpenSpace is in C++ with wide use GLSL shaders and Lua files for the modules in the scene and configuration. The Lua script files returns an object with keys and values that define what the module represents and define what class instance is added to the scene.

Another key component in being able to visualize space in real time is the simulation time. Each frame the simulation time is progressed according to the simulation time speed multiplied with the difference in the computer's internal time since the previous frame. The time class is a Singleton and can be called anywhere in the code base and is used for updating the modules properly. There is also a frame count is stored as a variable to be able to do runtime calculations on the average frame rate per second and do animations based on it.

The modularity of OpenSpace made it easier to extend and keep separate uses of the software in different parts of the code. There were a few design patterns in place in OpenSpace, such as the Singleton and Factory patterns [7]. The benefit of using design patterns is not only to solve the problem at hand but also to increase the code readability and facilitate understanding for other developers [1].

2.2 Power Scaled Coordinates

A recurring problem in visualizing in such a huge coordinate space as the universe is the numerical limits of the integer and floating point types. To extend the range of what a coordinate can represent, a fourth number is added to the three dimensional coordinate which indicates what exponent e should be in Equation 1. The implementation is based on the work by Fu and Hanson [8].

$$(x, y, z) = (x', y', z') \cdot 10^e \tag{1}$$

Even though this was an improvement in OpenSpace, there are still issues with the precision of the floating point values that represent the position of objects far away from the global origin, the solar system barycenter. A workaround was implemented just before the work of this thesis where the user can manually change the global origin to be closer to the camera, hence decreasing the position values and increasing their accuracy.

2.3 Interleaved Arrays

Another central concept in computer graphics programming is how to represent the data of the vertices of a model to the Graphical Processing Unit (GPU). Interleaved arrays are used in several of the implementations discussed in this thesis. It was achieved by using the OpenGL function `glVertexAttribPointer` [9]. A code example from OpenSpace can be found in Appendix A and Figure 1 is demonstrating how the array is structured for accessing the data in the GLSL shaders, the texture coordinate and normal vectors are associated with a vertex position and located next to each other in the memory.

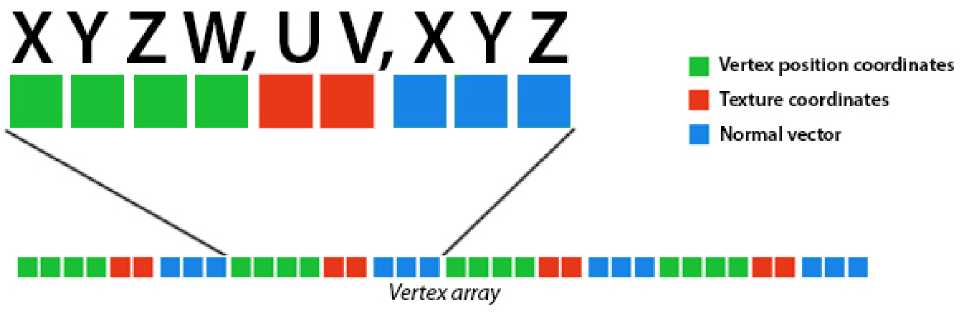


Figure 1: Interleaved vertex array

2.4 SPICE

SPICE is an information system with a toolkit available in C. OpenSpace has a singleton wrapper class for CSPICE to calculate positions, rotation matrices, converting coordinates between different coordinate systems and much more. There are many data set kernels publicly available for the solar system and the space missions handled in this thesis. Before the rendering loop is called in the initialization of OpenSpace, the kernels defined in the currently used module files are loaded to be able to retrieve the interesting data [10].

3 Method

This section of the thesis aims to explain ways to achieve the goals of the thesis and further building on the theory used in section 4. It elaborates on data collection and handling, the approach for creating a plane in space showing an image and the basics in projecting the images to target bodies.

3.1 Gathering the Data

To visualize the space missions in OpenSpace, the following data has to be acquired:

- A 3D spacecraft model.
- SPICE mission specific data kernels for the instruments, the position and the rotation of the spacecraft.
- Images taken by the spacecraft with information about the exact time they were taken and the target of the image.
- A SPICE data kernel about the target's rotational frames, positions and radii.
- If the target is not accurately described as a triaxial ellipsoid, an accurate 3D model of the target.

The data was gathered from different sources, images and kernels usually from official File Transfer Protocol (FTP) servers. The implementation also required detailed knowledge and information about the spacecraft's and payload on board.

In order to be able to get the best possible result from projecting the taken images to 3D models, these models have to be as detailed and accurate as possible. The two primary targets used for model projection was Vesta and 67P. In the case of Vesta, NASA released a highly detailed model. One of the very best models available was from a Swedish enthusiast named Mattias Malmer [11], who released updated models as better pictures of the actual comet was released.

3.2 Missing Data

A common issue in most data visualization is how to handle missing data. SPICE generates errors when missing data is requested [12]. The kernels containing data about the space missions can have gaps in time for when a position or rotational matrix can be retrieved. During the startup of OpenSpace, while the data kernels are loading, a map of positional and rotational data coverage is filled. This map facilitates runtime determination whether the current simulation time of OpenSpace is covered by the data kernels loaded. For OpenSpace not to crash when trying to look at spacecrafts outside of the kernel defined time range, position and rotation approximation algorithms were implemented and explaned in 4.1.

3.3 Image Plane

In order to be able to show images that were taken by the spacecraft but only partially or not at all on the surface of a target; an image plane is introduced. The idea of the plane is to be placed in the view frustum of the camera when the image is taken and show the image as a texture on the plane.

This plane made it possible to see some exciting edge images such as the Europa rise in Figure 8. To reduce cluttering and avoid planes sticking out of bodies, only the most recent image for an instrument is shown, and can be toggled off at runtime. Since all bodies in the solar system are in constant motion and rotation with respect to the Sun, the plane was *reparented*. This means that the plane is expressed in the target local coordinate system and, for example, stays on the edge of its target by adding the targets global position and rotation each frame.

3.4 Texture Projection

Projective texture mapping [2] is used for image projecting to bodies and requires a number of matrices and matrix multiplications. To get the projected texture coordinate, the position of each vertex is multiplied with the model transformation and then the projector matrix as can be seen in the vertex shader in Appendix B. The model transform for the body in the current simulation time is returned by the SPICE function `pxform_c`.

The projector matrix is a result of a view matrix, a perspective projection matrix and a bias matrix for normalization. The view matrix is shown in Equation 2 contains the following normalized vectors:

- Spacecraft camera boresight, retrieved using SPICE function `getfov_c` (v_1).
- The cross product vector of the upwards direction and the boresight of the camera (v_2).
- The cross product vector of v_1 and v_2 (v_3).
- The dot products between the position of the scene camera and vectors v_2, v_3 and v_1 (v_4).

$$Vm = \begin{pmatrix} v_2.x & v_3.x & v_1.x & 0 \\ v_2.y & v_3.y & v_1.y & 0 \\ v_2.z & v_3.z & v_1.z & 0 \\ v_4.x & v_4.y & v_4.z & 1 \end{pmatrix} \tag{2}$$

The perspective projection matrix is found using the OpenGL Mathematics (GLM) function `glm::perspective` providing the angular separation, aspect ratio, near and far plane for the spacecraft camera. The bias matrix used for normalization is shown in Equation 3. The full C++ implementation of the projector matrix calculation can be found in Appendix C.

$$Bm = \begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0.5 & 0.5 & 0.5 & 1 \end{pmatrix} \tag{3}$$

The projected texture coordinates are within the field of view if they are within the range of 0 to 1. The full coordinate calculation is expressed in Equation 4.

$$\begin{aligned} ProjectedCoordinate = Bm \cdot PerspectiveProjectionMatrix \\ \cdot Vm \cdot ModelMatrix \cdot VertexPosition \end{aligned} \tag{4}$$

3.5 Irregular Body Projection

In order to be able to project images to bodies that are not well approximated by a sphere, a new projection algorithm is implemented and discussed further in Section 4.5 and Appendix B.

Because drawing to a texture on the Central Processing Unit (CPU) would take a really long time and the fact that there is no support for reading and writing to a texture in a single shader pass, a projection pass was introduced before the actual drawing, creating a feedback loop in the graphics pipeline.

There are, however, still problems to be solved with the projection. The criteria for whether a fragment should be projected to is to be within the field of view of the instrument taking the image at the time that it is taken and that the normal direction should point towards the spacecraft. This can cause some false projections for irregular bodies when fragments meet these criteria, as illustrated in Figure 2. A possible solution to the problem is a Z buffer implementation in the shaders.

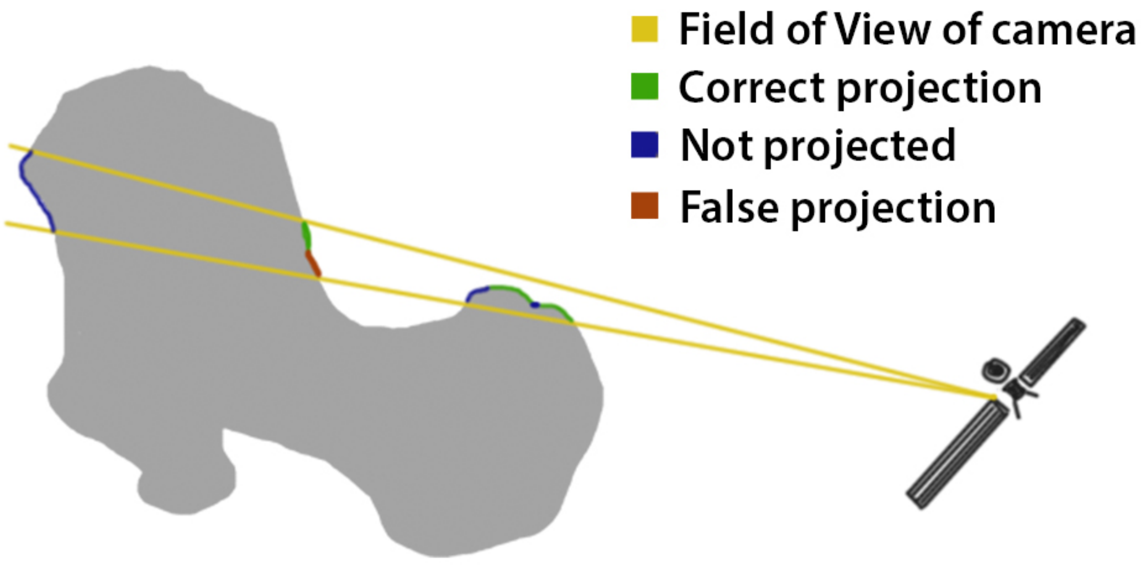


Figure 2: False projections illustrated on the shape of 67P. The fragments covered in red should not be projected to as it is not visible for the spacecraft to the right.

4 Implementation

This chapter covers the implementation details of the main additions to OpenSpace. All visible objects in the scene of OpenSpace are subclasses of a class called Renderable and implement the functions `render` and `update`. Each frame, a so called engine traverses through a scene graph, updates and renders the renderable objects. A big part of the work covered in this thesis was to modify and add subclasses to Renderable, even though some modifications were also made in the render engine and scene graph classes, mainly to facilitate the new classes. Every Renderable subclass is linked to vertex and fragment shaders, where the actual position in the scene and color of each fragment is calculated.

4.1 Missing Data

As discussed in section 3, approximated positions and rotations are calculated instead of generating errors. The following cases are considered:

- The current simulation time is *before* the times covered in the loaded kernels: The spacecraft is placed in the first available position.
- The current simulation time is *after* the times covered in the loaded kernels: The spacecraft is placed in the last available position retrieved from the data kernels.
- The current simulation time is *between* times of kernel coverage: The position is interpolated according to the interpolation in Algorithm 4.1.

```
difference = later - earlier;
quote = (time - earlier) / difference;
pos[0] = (pos_earlier[0] * (1 - quote) + pos_later[0] * quote);
pos[1] = (pos_earlier[1] * (1 - quote) + pos_later[1] * quote);
pos[2] = (pos_earlier[2] * (1 - quote) + pos_later[2] * quote);
```

Algorithm 4.1: Linear interpolation, for `pos[0]`, `pos[1]` and `pos[2]` positions.

A similar algorithm is put in place for the rotational matrices and can be found in Appendix D.

The estimated position can give a false sense of accuracy. It is therefore essential to show the user that the spacecraft position might not be accurately placed in the simulation. In order to show the possible inaccuracy, a uniform transparency level varying with time was given to the spacecraft model in all of these three cases. Using the simulation frame count and a sine wave function, the alpha component of all the fragments in the model varies in uniform between 0.5 and 1, as demonstrated in Algorithm 4.2.

```
int frame = _frameCount % 180;
float fadingFactor = static_cast<float>(sin((frame * M_PI) / 180));
_alpha = 0.5f + fadingFactor * 0.5f;
```

Algorithm 4.2: Partial transparency depending on the frame count implemented in C++.

4.2 Night Side

For the general impression of the visualization and to give the planets (with Earth as primary objective) a more realistic dark side, the possibility to add a night side texture to planets is added. This texture is then mixed with the day light texture on the GPU for fast calculations, using the GLSL function `mix` [9] in combination with the dot product of the opposite direction of the sun and the normal direction of the fragment. The code is shown in Algorithm 4.3.

```
mix(day, night, (1 + dot(fragmentNormal, -(sunDirection)) / 2);
```

Algorithm 4.3: Mixing texture in the fragment shader with the dot product between the fragment normal and opposite sun direction as the quote, implemented in GLSL.

4.3 Image Plane

In order to get the coordinates for the image plane, expressed in the targets local coordinate system, a number of SPICE functions are utilized. `getfov_c` returns the bounds of the Field of View (FoV) for a specified instrument. `spkpos_c` provides the vector between the spacecraft and the target, and `pxform_c` convert these vectors from spacecraft frame to the galactic frame to be able to express the plane in different coordinate systems. The FoV bounds are projected to a vector orthogonal to the vector from the spacecraft to the target using `vproj_c`. To finally get the corner coordinates in the local coordinate system, the target vector is subtracted and converted into a *power scaled coordinate*.

These corner coordinates are then used for drawing a plane and the image taken by the spacecraft is used as texture for the plane, with texture and position coordinates in an *interleaved array*. An algorithm traversing through all possible targets is used for cases with multiple targets, where the closest target becomes the parent of the plane. See Appendix E for the code used for converting the corner positions and creating the plane vertex data.

4.4 Time Indicating Trajectories

The trajectories were initially drawn using only `GL_LINE_STRIP` between points along the trajectory. This does show where the spacecraft is traveling but not how fast it is moving or when it is in a specific position.

By carefully choosing in what time interval these points were placed and combining the line with `GL_POINTS`, the speed of the spacecraft can be derived from the path as well. Figure 4 demonstrates not only how New Horizons flew past Pluto, but also gives a sense of its speed. In this case the markers is placed in 15 minute intervals. To determine whether a marker is an hourly marker, `gl_VertexID` and the GLSL function `mod` is used. The color interpolation is done by default when the color is set in the vertex shader and then using the color for each fragment in the fragment shader. A maximum distance from the target body is also used to only display the points during the closest approach. The GLSL shader code to achieve this is found in Appendix F.

4.5 Irregular Body Projection

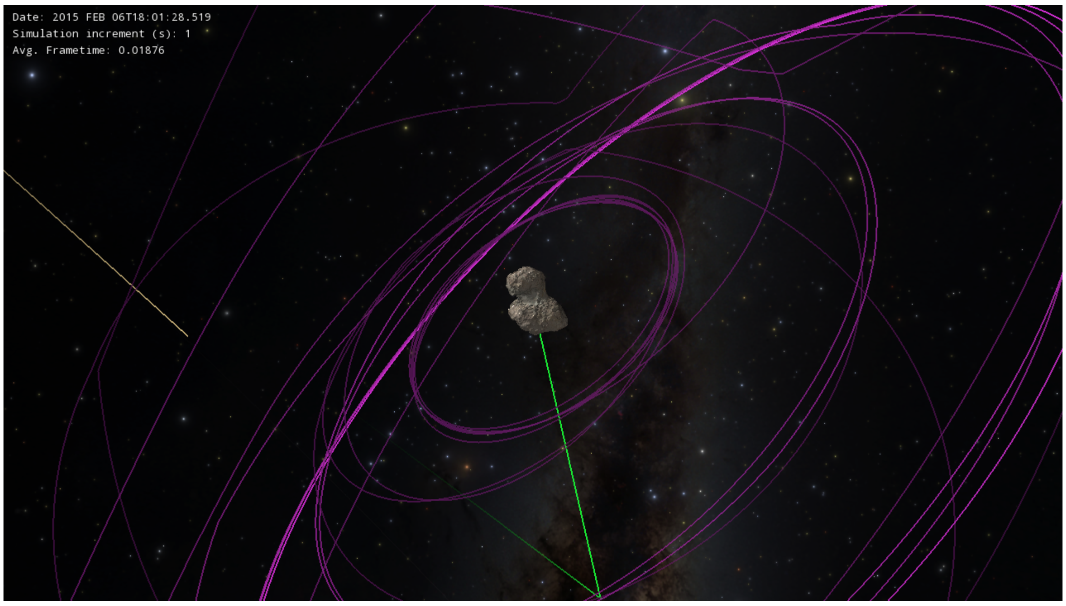
In the projection pass of the image projection, a texture is bound as the Frame Buffer Object (FBO) and the full model data with location, texture and normal are sent as an *interleaved array*. The texture coordinate was then matched to the clip space coordinates, outputting a new texture with the image projected onto it. The vertex and fragment shaders of the projection pass can be found in Appendix B.

5 Results

This section is showing screen captures from OpenSpace of the modules discussed in Sections 3 and 4. Figure 3 shows the main parts of Rosetta's observations of 67P. The same trajectory drawing class is used for drawing Dawn's orbit around Vesta in Figure 5. For the trajectory of New Horizons flyby of Pluto, the time indicating trajectory is seen in Figure 4. The section is also showing the results of the night side texturing, image plane and model projection.

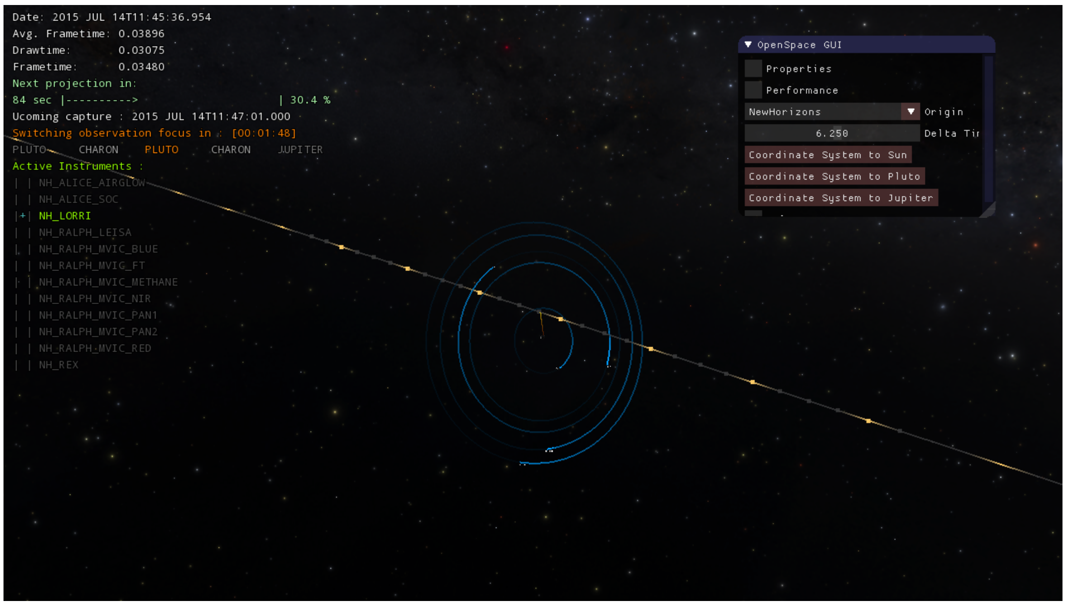
5.1 Trajectories

All of the spacecraft trajectories in these images are drawn with respect to the target body. When drawing the trajectories with regards to the Sun, they are almost parallel to the target movement which is less intuitive.



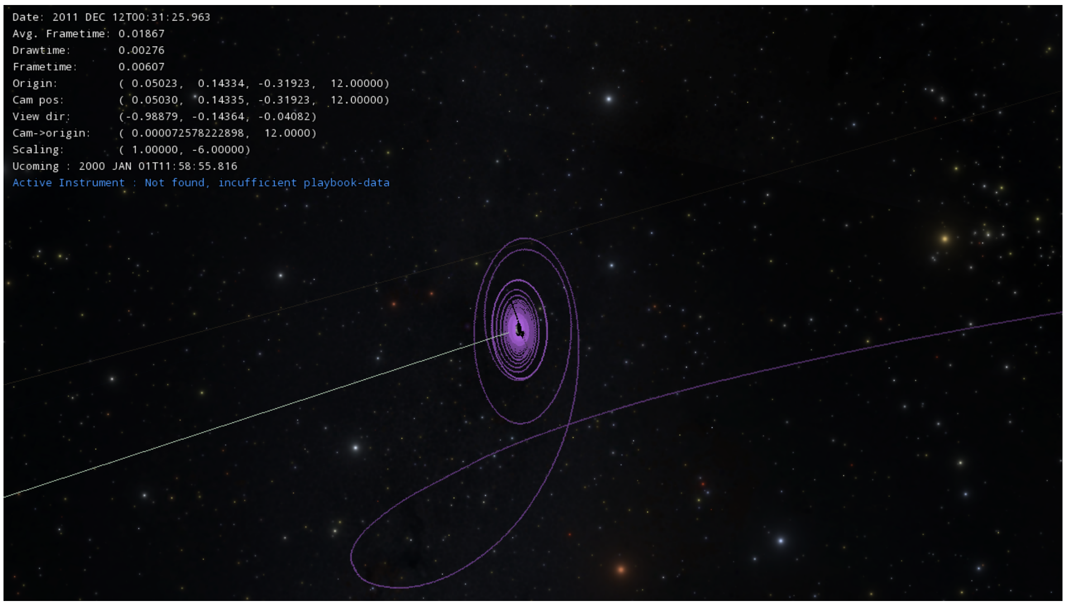
```
Date: 2015 FEB 06T18:01:28.519
Simulation increment (s): 1
Avg. Frametime: 0.01876
```

Figure 3: Trajectory without time stamps, used for Rosetta's path around 67P in 2014 and 2015. It is showing both low and high altitude orbits and running the maneuvers required to land Philae.



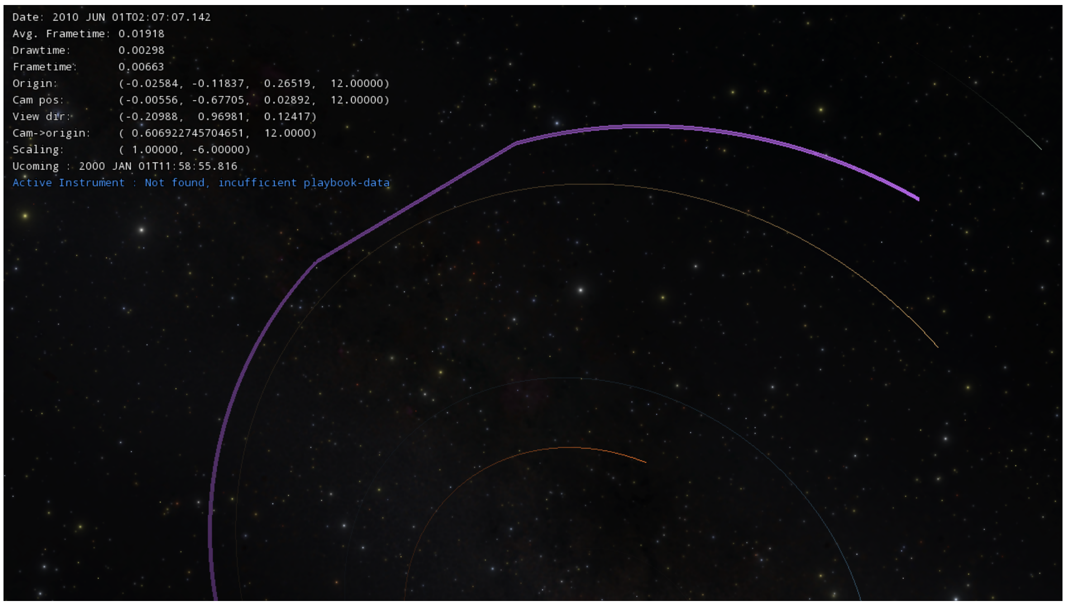
```
Date: 2015 JUL 14T11:45:36.954
Avg. Frametime: 0.03896
Drawtime:       0.03075
Frametime:      0.03480
Next projection in:
84 sec |---------->                      | 30.4 %
Ucoming capture : 2015 JUL 14T11:47:01.000
Switching observation focus in : [00:01:48]
PLUTO     CHARON     PLUTO     CHARON     JUPITER
Active Instruments :
| | NH_ALICE_AIRGLOW
| | NH_ALICE_SOC
|+| NH_LORRI
| | NH_RALPH_LEISA
| | NH_RALPH_MVIC_BLUE
| | NH_RALPH_MVIC_FT
| | NH_RALPH_MVIC_METHANE
| | NH_RALPH_MVIC_NIR
| | NH_RALPH_MVIC_PAN1
| | NH_RALPH_MVIC_PAN2
| | NH_RALPH_MVIC_RED
| | NH_REX
```

Figure 4: For New Horizons' closest approach to Pluto, the trajectory with time stamps is used to indicate the speed and distance to Pluto. The yellow dots indicate hourly marks and the gray dots indicate 15 minute intervals for the current UTC time.



```
Date: 2011 DEC 12T00:31:25.963
Avg. Frametime: 0.01867
Drawtime:       0.00276
Frametime:      0.00607
Origin:         ( 0.05023,  0.14334, -0.31923,  12.00000)
Cam pos:        ( 0.05030,  0.14335, -0.31923,  12.00000)
View dir:       (-0.98879, -0.14364, -0.04082)
Cam->origin:    ( 0.000072578222898,  12.0000)
Scaling:        ( 1.00000, -6.00000)
Ucoming : 2000 JAN 01T11:58:55.816
Active Instrument : Not found, incufficient playbook-data
```

Figure 5: Dawn's g-shaped orbit around Vesta in 2011. While Rosetta was altering its altitude, Dawn kept approaching until it departed for Ceres.

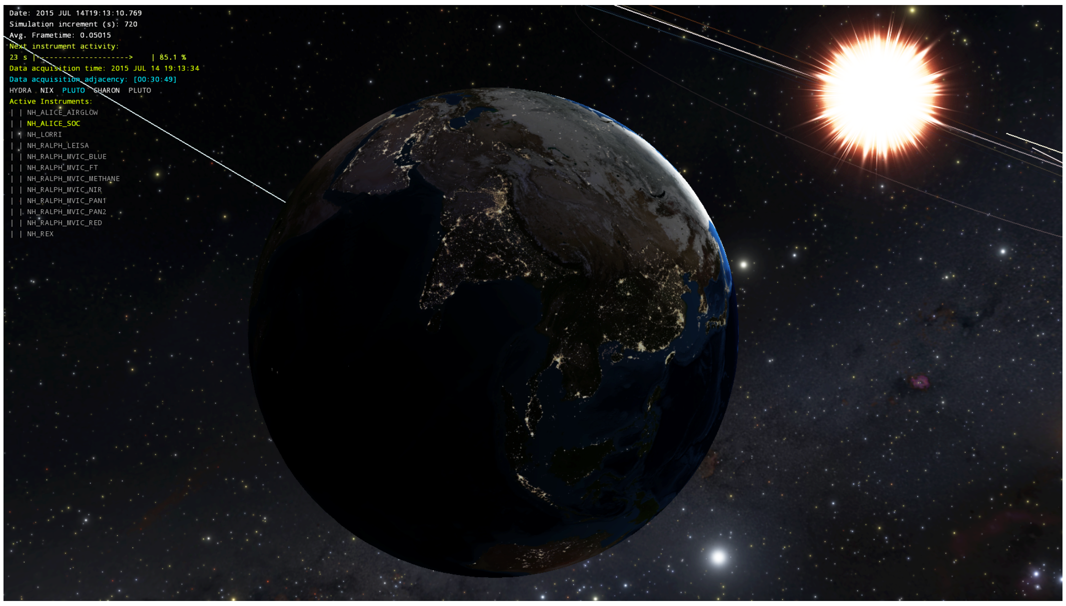


```
Date: 2010 JUN 01T02:07:07.142
Avg. Frametime: 0.01918
Drawtime:       0.00298
Frametime:      0.00663
Origin:         (-0.02584, -0.11837,  0.26519,  12.00000)
Cam pos:        (-0.00556, -0.67705,  0.02892,  12.00000)
View dir:       (-0.20988,  0.96981,  0.12417)
Cam->origin:    ( 0.606922745704651,  12.0000)
Scaling:        ( 1.00000, -6.00000)
Ucoming : 2000 JAN 01T11:58:55.816
Active Instrument : Not found, incufficient playbook-data
```

Figure 6: Interpolated trajectory of Dawn in late 2009. For demonstration purpose, the data kernel for positions in November 2009 was removed.

5.2 Night Side

Instead of being completely black or just a darker version of the day texture, Figure 7 shows how a city light map is mixed in.

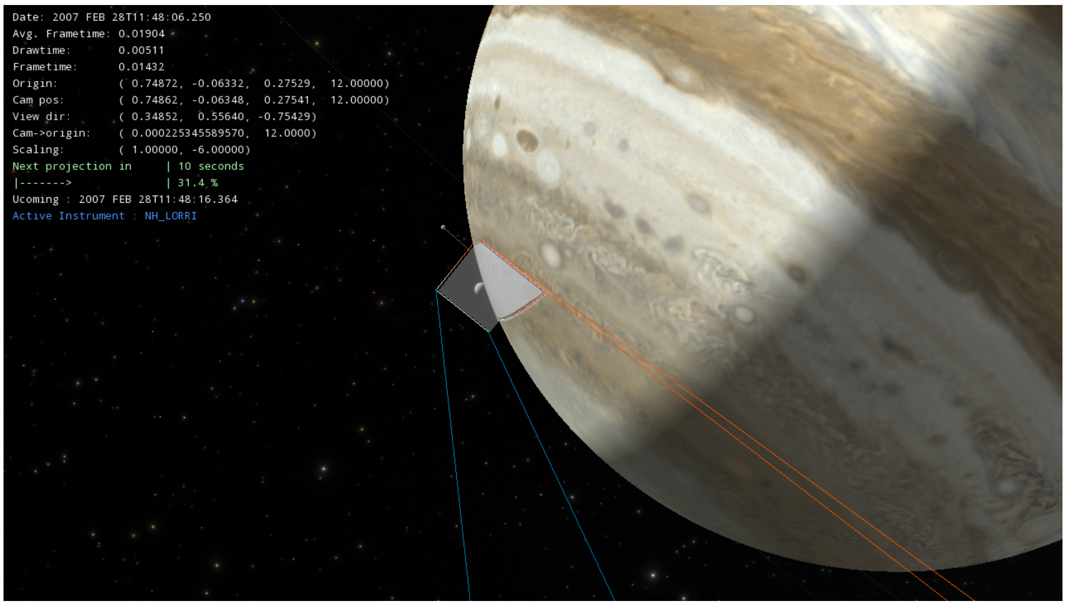


```
Date: 2015 JUL 14T19:13:10.769
Simulation increment (s): 720
Avg. Frametime: 0.05015
Next instrument activity:
23 s |---------------------->    | 85.1 %
Data acquisition time: 2015 JUL 14 19:13:34
Data acquisition adjacency: [00:30:49]
HYDRA  NIX  PLUTO  CHARON  PLUTO
Active Instruments:
| | NH_ALICE_AIRGLOW
| | NH_ALICE_SOC
| | NH_LORRI
| | NH_RALPH_LEISA
| | NH_RALPH_MVIC_BLUE
| | NH_RALPH_MVIC_FT
| | NH_RALPH_MVIC_METHANE
| | NH_RALPH_MVIC_NIR
| | NH_RALPH_MVIC_PAN1
| | NH_RALPH_MVIC_PAN2
| | NH_RALPH_MVIC_RED
| | NH_REX
```

Figure 7: The night side of Earth, mixing the day texture with a city light texture map for a more realistic look.

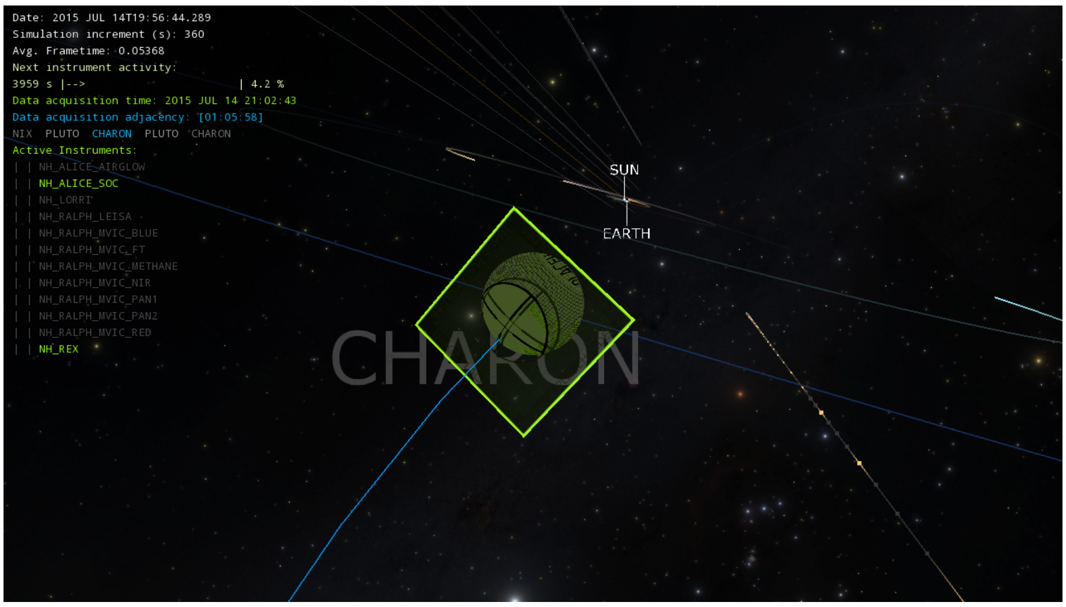
5.3 Image Plane

The image plane was introduced to be able to see the images taken on the edges of targets. This can show outgassing of comets, other objects caught in the same frame such as Europa in Figure 8 or to get a feeling of the size of the field of view from which the resolution can be calculated. In Figure 9 the image taken is framing Charon, demonstrating the surface to background ratio.



```
Date: 2007 FEB 28T11:48:06.250
Avg. Frametime: 0.01904
Drawtime:       0.00511
Frametime:      0.01432
Origin:         ( 0.74872, -0.06332,  0.27529,  12.00000)
Cam pos:        ( 0.74862, -0.06348,  0.27541,  12.00000)
View dir:       ( 0.34852,  0.55640, -0.75429)
Cam->origin:    ( 0.000225345589570,  12.0000)
Scaling:        ( 1.00000, -6.00000)
Next projection in     | 10 seconds
|------->              | 31.4 %
Ucoming : 2007 FEB 28T11:48:16.364
Active Instrument : NH_LORRI
```

Figure 8: The LORRI image plane on the edge of Jupiter on February 28, 2007, just as the moon Europa rises over the horizon from perspective of New Horizons.

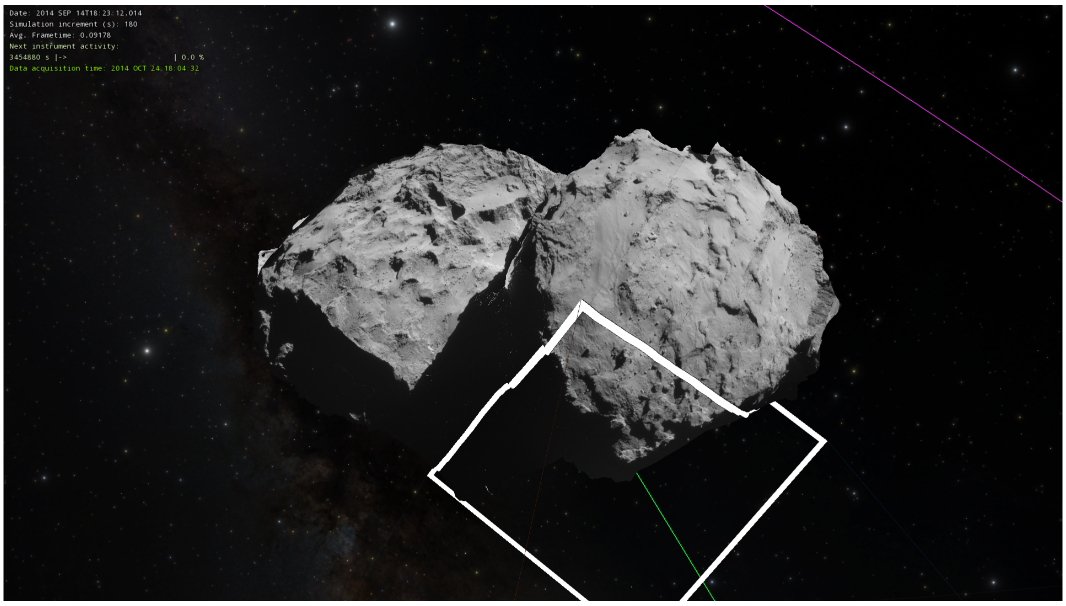


```
Date: 2015 JUL 14T19:56:44.289
Simulation increment (s): 360
Avg. Frametime: 0.05368
Next instrument activity:
3959 s |-->                        | 4.2 %
Data acquisition time: 2015 JUL 14 21:02:43
Data acquisition adjacency: [01:05:58]
NIX  PLUTO  CHARON  PLUTO  CHARON
Active Instruments:
| | NH_ALICE_AIRGLOW
| | NH_ALICE_SOC
| | NH_LORRI
| | NH_RALPH_LEISA
| | NH_RALPH_MVIC_BLUE
| | NH_RALPH_MVIC_FT
| | NH_RALPH_MVIC_METHANE
| | NH_RALPH_MVIC_NIR
| | NH_RALPH_MVIC_PAN1
| | NH_RALPH_MVIC_PAN2
| | NH_RALPH_MVIC_RED
| | NH_REX
```

Figure 9: The LORRI image plane at Pluto's largest moon Charon, just after closest approach covering most of the back side. The image used here was an image placeholder as it was produced before it actually existed. It was placed according to a predicted event schedule provided by the New Horizons mission team scientists.

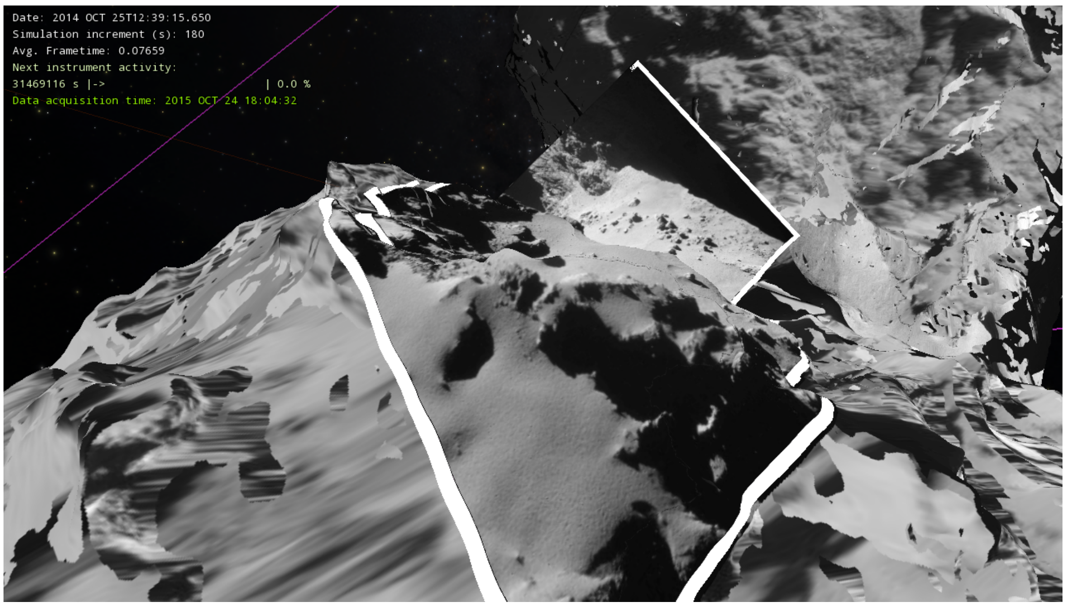
5.4 3D Model Projection

The image projection addition in this thesis was focused on irregular bodies. Figure 10 and Figure 11 are both examples from Rosetta studying 67P and how the images are pasted into the texture of the comet. This provides details that a shape model can not illustrate without high resolution textures and an advanced global illumination technique.



```
Date: 2014 SEP 14T18:23:12.014
Simulation increment (s): 180
Avg. Frametime: 0.09178
Next instrument activity:
3454880 s |->                       | 0.0 %
Data acquisition time: 2014 OCT 24 18:04:32
```

Figure 10: Image projection after a 2 by 2 mosaic of 67P taken by the Rosetta NAVCAM during approach in September 2014.



```
Date: 2014 OCT 25T12:39:15.650
Simulation increment (s): 180
Avg. Frametime: 0.07659
Next instrument activity:
31469116 s |->                          | 0.0 %
Data acquisition time: 2015 OCT 24 18:04:32
```

Figure 11: As the spacecraft comes closer, the pictures naturally get more detail of the surface. This image projected was manually picked from the official Rosetta blog [13] and was taken by the NAVCAM in late October 2015. The distortion of the image to the left is because another 2D image taken from the left is projected to the 3D surface.

6 Discussion

This section aims to discuss several aspects of the thesis. The difficulties for scientists who wants to contribute but are not allowed to, inaccuracies and possible flaws in the implementations as well as describing the attempts to reach a wide range of people.

6.1 Scientists Involvement

The responses from scientists at NASA and ESA were positive. The International Traffic in Arms Regulations (ITAR) [14] does however complicate the data acquisition. It makes some of the material that NASA produces unavailable, and is also a reason for the mission team to hold on to the data for a while after they get it.

The naming conventions of the kernel files are not always very straight forward which made finding the right kernels somewhat challenging. There are usually huge amounts of data associated with a mission and not everything is necessary for the visualization. Because of these factors, the help in navigating amongst the data given from scientists at JPL and ESA was much appreciated.

6.2 Interpolation

The linear interpolation for positions and rotations could have been replaced by a more advanced interpolation algorithm to more accurately approximate the positions. Spline interpolation for example would require more coverage but is a possible extension. One could however argue that it is more obvious to a user that the position is approximated when the trajectory follows a straight line as in Figure 6.

6.3 PlutoPalooza

When trying to reach a lot of people, it is important to demonstrate the ongoing progress. PlutoPalooza is a series of events initiated by NASA, one of which took place at AMNH on May 14th, 2015. This was a great chance to show OpenSpace. A few hundred teachers were invited to see mission scientists talk about the Pluto mission using OpenSpace as a visualization tool. This meant that we had to not only produce a stable release version but also had several run-throughs of what to show and when. This event together with many others, mainly 'Breakfast at Pluto' hosted at multiple planetariums across the globe, has led to a lot of media coverage for OpenSpace [1].

Even though all the last minute fixes and preparations required for the demos take a lot of valuable development time, they are essential for the core purpose of the software and its recognition.

6.4 Sources of Error

Even though everyone involved were happy with the results, some slight misalignment appeared on some of the images. There are numerous possible reasons for this, and the most likely truth is that it is some kind of combination of the issues discussed in this section.

The SPICE data kernels can vary significantly in size and time range. A predicted path or rotational matrix is usually less frequently sampled which does not only lead to smaller files but also to a lower accuracy since these samples are used to estimate the position. The internal position estimation system in the SPICE C toolkit is more advanced than the one demonstrated in Figure 6, but has no guarantee of returning the exact position at all times. Another reason for this is that the positions for the bodies are sometimes unknown.

The FoV angular separation of the camera and the image time stamps are other possible precision error sources. For the images of 67P collected from the Rosetta blog [13] the time stamps were given as whole seconds. There is no reason why the images would not be taken at such an even time but in the worst case it could have been rounded from 0.5 seconds away which could lead to a significant difference in where the camera is aimed.

The models used for image projection has to be not only accurate in shape but also in rotation and scale. The Vesta model [5] had to be carefully stitched together and scaled according to an approximated scaling factor given on their website. The result after trial and error was satisfying but might still be a source of error.

Light has finite speed. This means that there will be a difference between where a target actually is and where it is perceived to be, the latter being where it was when the photons left the target. Stellar aberration is another astronomical phenomenon that influences the apparent position of bodies depending on their relative speed. When using the SPICE function `spkpos_c`, there is an aberration correction flag argument. When using these flags, the returned value is supposed to compensate for these factors but there is a risk that these calculations are sources of inaccuracy.

7 Conclusion

The thesis was considered successful in preparing for New Horizons flyby and extending the software for multiple mission support. Projection to non-spherical objects was a very well appreciated addition even though it is not yet perfected. The image projection on 67P looked good in the dome in Hayden Planetarium where one really gets a sense of immersion.

There is definitely a certain level difficulty in visualizing such advanced engineering to both satisfy the mission scientists and keep a level of scientific accuracy while producing something comprehensible for the general public. There are still many more intuitive ways to show data to be discovered, but there is definitely progress in the right direction as we explore and learn more about our universe.

"To be able to see our context in this larger sense, at all scales, helps us all I think, in understanding where we are and who we are in the universe." - Carter Emmart, 2010 [15]

References

[1] OpenSpace project site, http://openspace.itn.liu.se - Accessed 2015-08-09

[2] Projective Texture Mapping, official NVIDIA site, http://www.nvidia.com/object/Projective_Texture_Mapping.html - Accessed 2015-08-19.

[3] Official NASA New Horizons site, https://www.nasa.gov/mission_pages/newhorizons/main/index.html - Accessed 2015-08-07.

[4] Official ESA Rosetta site, http://www.esa.int/Our_Activities/Space_Science/Rosetta - Accessed 2015-08-07.

[5] NASA's 3D Model of Vesta, http://nasa3d.arc.nasa.gov/detail/vesta - Accessed 2015-08-07.

[6] Official Nasa Dawn site, http://dawn.jpl.nasa.gov - Accessed 2015-08-07.

[7] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley. 1995.

[8] Andrew J. Hanson, Chi-Wing Fu, and Eric A. Wernert, *Very Large Scale Visualization Methods for Astrophysical Data*, http://www.cs.indiana.edu/~hansona/papers/vissym00.pdf - Accessed 2015-08-20.

[9] OpenGL Online Documentation, https://www.opengl.org/sdk/docs/man - Accessed 2015-08-06.

[10] NAIF, JPL, *About SPICE*, http://naif.jpl.nasa.gov/naif/aboutspice.html - Accessed 2015-08-06.

[11] Mattias Malmer, Adventures in Image Processing, http://mattias.malmer.nu - Accessed 2015-05-10.

[12] NAIF, JPL. *Missing SPICE Data*, http://naif.jpl.nasa.gov/naif/missing_data.html - Accessed 2015-08-06.

[13] ESA, Rosetta Blog, http://blogs.esa.int/rosetta

[14] US Department of State, *The International Traffic in Arms Regulations (ITAR)*, https://www.pmddtc.state.gov/regulations_laws/itar.html. Accessed 2015-08-09.

[15] Carter Emmart, *A 3D atlas of the universe*, http://www.ted.com/talks/carter_emmart_demos_a_3d_atlas_of_the_universe. Accessed 2015-08-10.

A Interleaved Array

The following C++ code is a practical example from OpenSpace of how to achieve an interleaved array using OpenGL functions.

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);

glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, vertexSize,
reinterpret_cast<const GLvoid*>(offsetof(ModelGeometry::Vertex, location)));
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, vertexSize,
reinterpret_cast<const GLvoid*>(offsetof(ModelGeometry::Vertex, tex)));
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, vertexSize,
reinterpret_cast<const GLvoid*>(offsetof(ModelGeometry::Vertex, normal)));
```

B Irregular Body Projection

Following is the code from the vertex and fragment shaders constituting the projection pass. The challenge is to match the texture coordinate to the clipping space of the FBO which is the target body texture.

B.1 Vertex Shader

```
uniform mat4 ProjectorMatrix;
uniform mat4 ModelTransform;
uniform vec2 _scaling;
layout(location = 0) in vec4 in_position;
layout(location = 1) in vec2 in_st;
layout(location = 2) in vec3 in_normal;
...
out vec4 ProjTexCoord;
out vec2 vs_uv;
void main() {
    vs_position  = in_position;
    vec4 tmp    = in_position;
    vec4 position = pscTransform(tmp, ModelTransform);
    vs_position = tmp;
    vec4 raw_pos = psc_to_meter(in_position, _scaling);
    ProjTexCoord = ProjectorMatrix * ModelTransform * raw_pos;
    vs_normal = normalize(ModelTransform * vec4(in_normal,0));
    //match clipping plane
    vs_uv = (in_st * 2) - 1;
    gl_Position = vec4(texco, 0.0, 1.0);
}
```

B.2 Fragment Shader

```
...
in vec4 ProjTexCoord;
in vec2 vs_uv;
..
bool inRange(float x, float a, float b) {
return (x >= a && x <= b);
}
void main() {
    vec2 uv = vec2(0.5,0.5) * vs_uv + vec2(0.5,0.5);
    vec4 projected = ProjTexCoord;
    //normalize and invert coordinates
    projected.x /= projected.w;
    projected.y /= projected.w;
    projected.x = 1 - projected.x;
    projected.y = 1 - projected.y;
    if((inRange(projected.x, 0, 1) && inRange(projected.y, 0, 1))
        && (dot(n, boresight) < 0)) {
        color = texture(projectTexture, projected.xy);
    } else {
        color = texture(currentTexture, uv);
    }
}
```

C Projector Matrix Computation

This is the C++ function used for computing the projector matrix in the body projection classes of OpenSpace.

```
glm::mat4 RenderableModelProjection::computeProjectorMatrix(
    const glm::vec3 loc, glm::dvec3 aim, const glm::vec3 up) {
    //rotate boresight into correct alignment
    _boresight = _instrumentMatrix*aim;
    glm::vec3 uptmp(_instrumentMatrix*glm::dvec3(up));
    // create view matrix
    glm::vec3 e3 = glm::normalize(_boresight);
    glm::vec3 e1 = glm::normalize(glm::cross(uptmp, e3));
    glm::vec3 e2 = glm::normalize(glm::cross(e3, e1));
    glm::mat4 projViewMatrix = glm::mat4(e1.x, e2.x, e3.x, 0.f,
        e1.y, e2.y, e3.y, 0.f,
        e1.z, e2.z, e3.z, 0.f,
        -glm::dot(e1, loc), -glm::dot(e2, loc), -glm::dot(e3, loc), 1.f);

    // create perspective projection matrix
    glm::mat4 projProjectionMatrix = glm::perspective(
        _fovy, _aspectRatio, _nearPlane, _farPlane);
    // bias matrix
    glm::mat4 projNormalizationMatrix = glm::mat4(0.5f, 0, 0, 0,
    0, 0.5f, 0, 0,
    0, 0, 0.5f, 0,
    0.5f, 0.5f, 0.5f, 1);
    return projNormalizationMatrix*projProjectionMatrix*projViewMatrix;
}
```

D Transformation Matrix Estimation and Linear Interpolation

The is the algorithm in OpenSpace of how the rotational matrices are estimated in the three cases where the time requested is either before, after or between covered times. If it is in between, the transform matrix is interpolated using linear interpolation, implemented in C++.

```
if (coveredTimes.lower_bound(time) == first) {
    // coverage later, fetch first transform
    pxform_c(fromFrame.c_str(), toFrame.c_str(),
    *first, (double(*)[3])glm::value_ptr(positionMatrix));
}
else if (coveredTimes.upper_bound(time) == last) {
    // coverage earlier, fetch last transform
     pxform_c(fromFrame.c_str(), toFrame.c_str(),
    *(coveredTimes.rbegin()), (double(*)[3])glm::value_ptr(positionMatrix));
}
else {
    // coverage both earlier and later, interpolate these transformations
    earlier = *std::prev((coveredTimes.lower_bound(time)));
    later = *(coveredTimes.upper_bound(time));
    glm::dmat3 earlierTransform, laterTransform;
    pxform_c(fromFrame.c_str(), toFrame.c_str(),
        earlier, (double(*)[3])glm::value_ptr(earlierTransform));
    pxform_c(fromFrame.c_str(), toFrame.c_str(),
        later, (double(*)[3])glm::value_ptr(laterTransform));

difference = later - earlier;
quote = (time - earlier) / difference;

for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        positionMatrix[i][j] = (earlierTransform[i][j] * (1 - quote)
            + laterTransform[i][j] * quote);
    }
}
```

E Image Plane

These are the core C++ code lines for transforming the FoV of a spacecraft instrument into a plane with texture coordinates in an interleaved array.

```
for (int j = 0; j < bounds.size(); ++j) {
    frameConversion(bounds[j], frame, GalacticFrame, currentTime);
    cornerPosition = orthogonalProjection(vecToTarget, bounds[j]);
    cornerPosition -= vecToTarget;
    frameConversion(cornerPosition, GalacticFrame, _target.frame, currentTime);
    projection[j] = PowerScaledCoordinate::CreatePowerScaledCoordinate(
    cornerPosition[0], cornerPosition[1], cornerPosition[2]);
    projection[j][3] += 3; // SPICE returns distances in KM
}
...
const GLfloat vertex_data[] = { // square of two triangles
    //   x      y     z     w     s     t
    projection[1][0], projection[1][1], projection[1][2],
     projection[1][3], 0, 1, // Lower left
    projection[3][0], projection[3][1], projection[3][2],
     projection[3][3], 1, 0, // Upper right
    projection[2][0], projection[2][1], projection[2][2],
     projection[2][3], 0, 0, // Upper left
    projection[1][0], projection[1][1], projection[1][2],
     projection[1][3], 0, 1, // Lower left
    projection[0][0], projection[0][1], projection[0][2],
     projection[0][3], 1, 1, // Lower right
    projection[3][0], projection[3][1], projection[3][2],
     projection[3][3], 1, 0, // Upper left
};
glBindVertexArray(_quad);
glBindBuffer(GL_ARRAY_BUFFER, _vertexPositionBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_data), vertex_data, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE,
sizeof(GLfloat) * 6, reinterpret_cast<void*>(0));
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE,
sizeof(GLfloat) * 6, reinterpret_cast<void*>(sizeof(GLfloat) * 4));
```

F Time Indicating Trajectories

To be able to differentiate missions, the base color (used as hourly dots for New Horizons flyby of Pluto) is set as a uniform variable. Following is the main functionality in the vertex and fragment shaders for the time indicating trajectories.

F.1 Vertex Shader

```
in vec4 in_point_position;
uniform vec3 color;
out vec4 vs_point_position;
out vec4 vs_point_color;
...
void main() {
    vec4 gray = vec4(0.6f, 0.6f, 0.6f, 0.8f);
    float bigPoint = 5.f;
    float smallPoint = 2.f;
    bool isHour = (0.1f < mod(gl_VertexID, 4);

    vec4 tmp = in_point_position;
    vec4 position = pscTransform(tmp, ModelTransform);
    vs_point_position = tmp;
    ...
    if(isHour) {
        vs_point_color.xyz = color;
        gl_PointSize = bigPoint;
    }
    else {
        vs_point_color = gray;
        gl_PointSize = smallPoint;
    }
    if (distance > maxDistance) {
            gl_PointSize = 0;
    }
}
```

F.2 Fragment Shader

```
in vec4 vs_point_position;
in vec4 vs_point_color;
...
void main() {
    vec4 position = vs_point_position;
    vec4 diffuse = vs_point_color;
    ...
}
```