Virtual image generator for asteroid mission navigation

Jiri Burant

School of Electrical Engineering

Espoo

Thesis supervisor:

Prof. Esa Kallio

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

Author: Jiri Burant

Title: Virtual image generator for asteroid mission navigation

Date: Language: English Number of pages: 4+14

Department of Radio Science and Technology

Professorship:

Supervisor and advisor: Prof. Esa Kallio

This project deals with testing of possible setups of a camera mounted on spacecraft. The idea is to simulate simplified movement of this spacecraft above the surface and capture images along the way. Based on those images, it can be decided which camera is best suitable for the given mission and at which angle it should be facing. The generated images can also be used in the next stage to test image analysis tools or image reconstruction software.
To do so, a simple application has been developped in java programming language, simulating the movement of the spacecraft and generating the pictures. It consists of a big canvas with 2D image, above which the spacecraft moves and simple GUI, allowing the user to change basic parameters such as the altitude of the spacecraft, the angle of the mounted camera and do on.
The basic application has been developped and tested, giving satisfying results, but there is a lot of room for future extensions, some of them being described in this report.

Keywords: Image generator, space technology project, openGL, java

Preface

I want to thank Professor Esa Kallio for the opportunity to work on this project and for his guidance and optimism.

Helsinky, 6.11.2016

Jiri Burant

Contents

1 Introduction

The project of the virtual image generator has been developped as a helper software tool for the ASPECT project [2], which is part of the ESA's Asteroid Impact Misstion (AIM) [1].

The goal of the AIM is to demonstrate new technologies, mainly in the telecommunications domain. The AIM mission will consist of the main spacecraft carrying the Mascot-2 asteroid lander, as well as two or more CubeSats. The mission should launch in in October 2020, then travel to the Didymos asteriod pair. The spacecraft would perform high-resolution visual, thermal and radar mapping of the moon and build detailed maps of its surface and interior structure.

The ASPECT project focuses on developping a single three-unit CubeSat equipped with a visible/near-infrared spectrometer to assess the asteroid composition and effects of space weathering and metamorphic shock, as well as post-impact plume observations.

The Cubesat will be deployed from a 10 km height from the target's surface and orbit it at around 1 to 3 km from the surface. From this distance it will measure the reflectance which will give an understanding of the composition of Didymoon and may provide answers to how this is influenced by space weathering and shock effect.

The navigation of the ASPECT CubeSat is a problematic task, and one of the things that could help it are the camera images. By using appropriate camera and appropriate software, position and speed could be estimated by analyzing the captured images. In order to determine a feasible camera setup and to test the analysis software, before it is deployed to the spacecraft, The Virtual Image Generator, VGI tool has been developped.

It simulates the movement of the spacecraft above surface along specified trajectory (assumed as line), capturing images below. Those images are saved as screenshots, and can later be used for testing image reconstruction, mapping software or speed and position estimation software.

The outcome of this work is a basic software application, ready for future extensions. Some of possible future improvements or applications are discussed in section 4. of this report.

2 Software design

Basic information

The application has been developped in java programming language, using JDK 1.8[3] and NetBeansIDE[4]. Apart from that, openGL in form of lwjgl [5] library has been used, details can be found on official pages or in Appendix A.

Architecture description

The application project itself consists of two packages. Package called *utils* consists of parts of the source code from the joml java library. It contains classes and methods for manipulating vectors and matrices, which is useful when rendering the openGL objects. Package *cameraproject* comprises all the application logic. It contains classes concerning the openGL tasks, as texture loading and scene rendering, it also contains the GUI setup and operation logic.

The code itself is thoroughly documented and self-explanatory. Nevertheless, a brief description of the most important classes follows.

The main class is called *CameraProject.java* In this class, the basic GUI initialization is carried out as well as the openGL initialization. Moreover, the main rendering loop is also located there.

For each of the rendered objects, that means the target location, the surface plane and the spacecraft, a dedicated class containing the parameters for the given object, has been created. Those classes are named: *Target.java*, *Land.java* and *Carrier.java*, respectivelly. These three classes have a common parent named *SceneObject.java*, which contains the common parameters, that each scene object must have, as Texture *Texture.java* and Model *Model.java*. Class *Texture.java* contains the loaded texture for the given object and the class *Model.java* contains parameters and methods for rendering of the scene objects.

Class named *Shader.java* is purely openGL based, and creates the shaders for rendering. The shaders specify, how should each pixel be rendered (color, texture, attenuation and so on.)

Class *Camera.java* contains the parameter of the camera, such as field of view, angles of the camera and so on. From those parameters, a transformation matrix is computed, which serves for rendering of the "observed" objects.

The remaining classes *TextFieldListener.java*, *FloatWrapper.java* and *InputData.java* are helper classes, which facilitate the transport of data from the GUI text fields into the program.

Graphical User Interface

The GUI has been developped using the java's native Swing library. The GUI is simple and comprises of several labels and text fields for entering the input data and two buttons, one for starting the simulation, one for loading the texture. The layout has been chosen as box layout, because it is straightforward and easy to modify.

In the fig.1, there is an example of how the running applTication looks like. On the left side, there is a surface (land) covered with texture of an island. Over it, a blue and a smaller red rectangles are drawn. They symbolize the spacecraft (blue one) and the target (red one). On the right side, the GUI panel is shown. First field is number of images to be taken during the flight. The second field specifies the output image type, for the time being there are just two options, *.jpg* and *.png*.Underneath, there is the initial position of the spacecraft, the three coordinates specify the x,y,z position in the coordinate system. The middle beginning of the coordinate system [0,0,0] is in the middle of the screen. The coordinates have no real measurement unit dimension.

The trajectory's end position specifies the target location. Last fields are camera parameters, namely two angles of the camera (alpha and omega), measured as deviation from the x and y axis respectively, and specified in degrees. The last field is the field of view of the camera, also specified in degrees. Finally, there are buttons for starting the simulation and for loading of the surface texture.

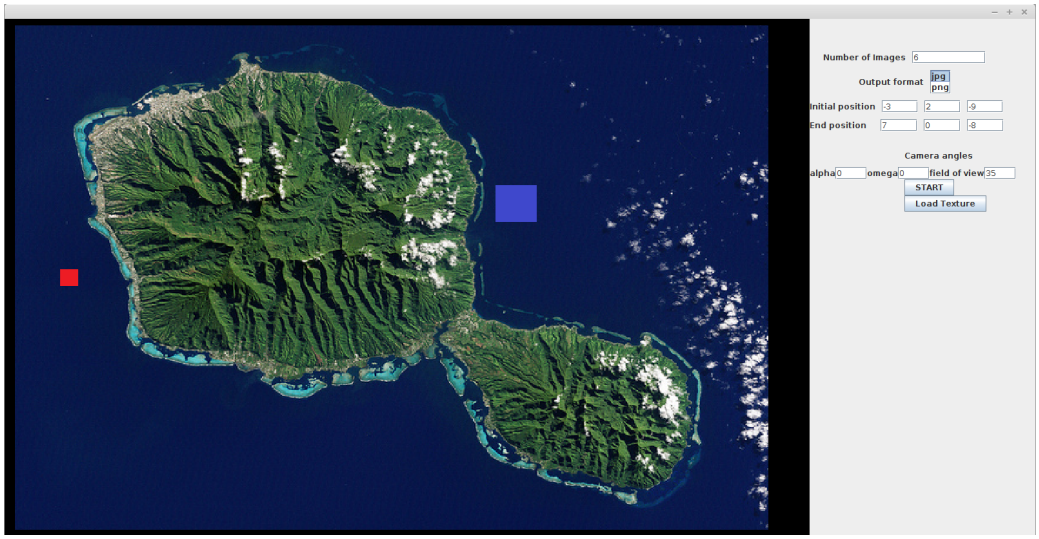

Figure 1: Example run of the application

The coordinate system of the rendered left part of the window is classical 3D Cartesian system, which spans from -9 to 9 for the x and -7 to 7 for the y coordinates. As for the z coordinates, it spans from -10 to 0, while -10 being the coordinate of the surface plane. The point z=0 is at the overview, top camera location and -10 is on the surface.

The figure 2 explains the relations of the angles and axes.

Once the simulation starts, the camera switches to view of the camera mounted on the spacecraft, with the specified parameters. The images captured by the spacecraft are rendered and consequently saved into a file. A few examples of such images are presented in the section 3.

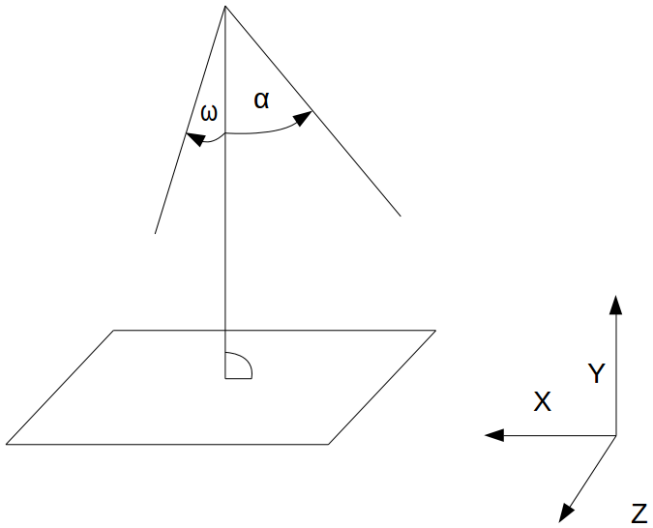

Figure 2: Angles and axes of the layout

3 Results

The system has been tested on a laptop with following specifications. Operating system: LinuxMint 17, Processor: Intel Pentium B970 2.3GHz, RAM: 4GB, integrated graphics card.

The application was tested for several different setups of camera parameters, different positions of the spacecraft and number of steps.

Following figures depict two example cases, their setup and the resulting screenshots, for both tests 4 steps were chosen.

In the first example case the spacecraft flew from coordinates [-3,-3,-9] to coordinates [3,3,-9]

In the second example case the spacecraft flew from coordinates [-3,3,-7] to coordinates [2,-1,-9]. The omega angle has been chosen as 10 degrees, therefore the image is slightly tilted.

The results of those tests were quite convincing, and they suggest, that the image generation works correctly.

Figure 3: The setup of the first test

Figure 4: First step of the first test

Figure 5: Second step of the first test

Figure 6: Third step of the first test

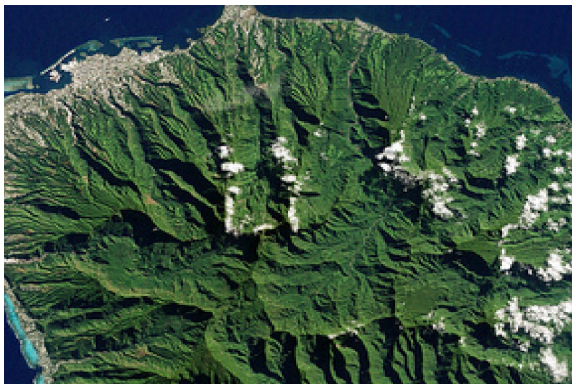

Figure 7: Fourth step of the first test

Figure 8: The setup of the second test

Figure 9: First step of the second test

Figure 10: Second step of the second test

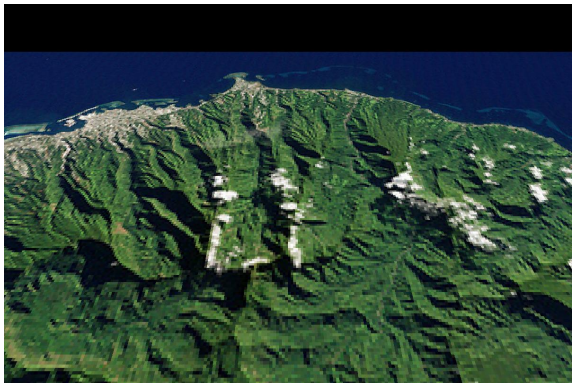

Figure 11: Third step of the second test

Figure 12: Fourth step of the second test

4 Conclusions and future works

The aim of this work was to develop an application capable of generating simulated images captured by a camera mounted on a spacecraft during a flight over 2D surface. It can be stated that such application, for simplified setup, was successfully developped and the results are convincing.

Still, there is a lot of room for future development and more improvements. Several suggestions for such improvements are listed below.

Improved GUI and rendering

The GUI, as stated earlier, is very simple. One of the possible improvements could be switching to another GUI library, for example TWL[11] or nifty[12]. That would allow more convenient control, for example adding of sliders for setting the position. Also, the rectangles symbolising the carrier and target are very primitive, therefore a better depiction could be introduced.

Besides those cosmetic changes, also coordinates system could be adjusted. Currently, the span of the x and y coordinates is fixly set from -4.5 to 4.5 for the top view, and the altitude is defined as distance from camera in the range from 0 to 10. For the real world applications, it would be a good idea to add the possibility to specify for example the real size of the 2D plane, and based on this values, the coordinates would be recalculated.

Curved surface

The primary use of this software is presented as part of the asteroids mission program, but in case it finds its use in another projects, for example concerning Earth imaging, then a nice feature could become a slight curvature of the 2D plane, imitating the curvature of the Earth's surface.

In order to do so, the 2D plane would have to be changed to 3D sphere. This can be done by using the sphere object from the *lwjgl.util.glu* package [13]. Of course, such sphere would have quite big diameter, compared to the altitude of the spacecraft.

The problem with rendering of the sphere is the texture mapping. In this work, no sphere textures were used so far, but if it is required in the future, it could be useful to express the coordinates in the spherical coordinate system, or to use the UV mapping technique [10]. Although, given the error introduced by the curvature is small, it might be possible to just map the texture as if it was 2D plane and accept this error.

Custom 3D surface

Another extension, similar to the previous one, but more complicated could be loading of arbitrary surface of the 'land'. Such feature could be useful specifically in a project such as the asteroid impact mission, because the surface of the asteriod is

very different from 2D plane, and the spacecraft would be close enough to recognize those irregularities.

The first thing such approach needs is a 3D model. The model should be presented in some easily parsable format, such as .obj, because a custom parser has to be created. The loading and parsing of a .obj model is nothing hard and there are tutorials on the internet [14]. After the model has been created and loaded into the application, the texture has to be loaded to cover it. This is the hard part, again UV mapping[10] could be helpful in this case. Unfortunately, 3D texturing is way out of the scope of this work.

Light intensity and Attenuation

When simulating images of a surface, it is certainly worth considering the parameters of the light source, it's intensity, position and so on. Also, the parameters of the surface, such as reflexivity could be taken into account.

The light intensity for every pixel should be calculated in the shader, therefore this is the main class to be modified. There are several ways how to approach this problem, in computer graphics the Lambertian Illumination Model is perhaps the most popular one. The important parameters are: intensity and direction of the incoming light, the distance from the light source, the normal vector, the basic color of the surface and the ambient light intensity.

A nice tutorial about this model and its implementation in java can be found in [16].

This modification would also require change of the GUI, to set the light source parameters.

Curved trajectory

In this work, only line trajectory of the spacecraft has been assumed. It is sufficient for most of the cases, but a possible improvement could be a curved trajectory, especially in connection with the curved or custom 3D surface. In that case, several other input parameters should be added. Assuming that the shape of the trajectory would be circular, which is the most probable case, such curvature could be specified by adding the coordinates of the center of this hypothetic circle. Using this point and the initial and target point, the trajectory can be calculated, an example is portraited on the figure 13.

Configuration file and console input

In order to automatize the image generation, the configuration file can be created. Then, the parameters can be automatically loaded to the application instead of manually entering them in the GUI.

The configuration file can be a simple text file. An example of such file is in the figure 14.

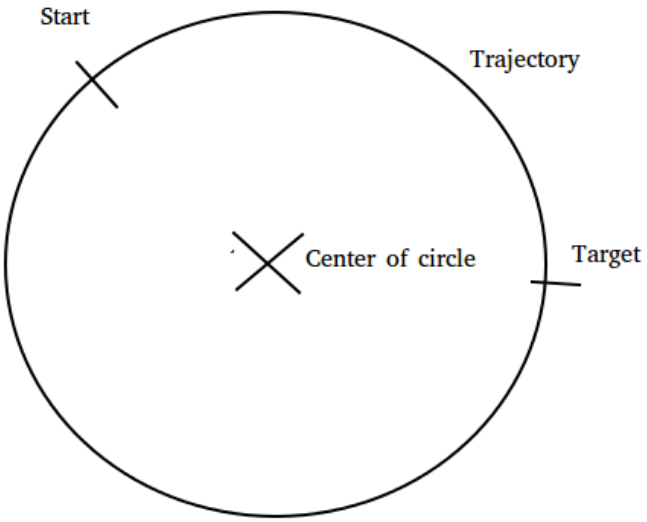

Figure 13: Curved trajectory scheme

exampleConfig.txt ×

```
images 6
init -3 0 9
end 2 1 9
alpha 0
omega 0
fov 35
```

Figure 14: Example of a configuration file

Such file can be read using the java file parser, reading the file line by line, breaking it into separate strings and then using them as key-value pairs to set up the parameters.

To automatize the process even further, the option of calling the application with parameters from console could be added. The application can be called from console by entering: java -jar CameraProject.jar, parameters can be added at the end of the call and consequently read by the java application, in the *main* method as *args* field. In such manner, the configuration file can be loaded as a parameter.

References

[1] Asteroid impact mission presentation:

http : //www.esa.int/Our_Activities/Space_Engineering_Technology/Asteroid_Impact 5th November, 2016.

[2] ASPECT project presentation:

http : //www.esa.int/Our_Activities/Space_Engineering_Technology/Asteroid_Impact 5th November, 2016.

[3] Java JDK 1.8 homepage:

http : //www.oracle.com/technetwork/java/javase/downloads/jdk8 − downloads − 2133151.html, 5th November, 2016.

[4] NetBeans homepage:

https : //netbeans.org/, 5th November, 2016.

[5] Lwjgl 2 library page:

http : //legacy.lwjgl.org/, 5th November, 2016.

[6] Lwjgl util library page:

https : //github.com/LWJGL/lwjgl/tree/master/src/java/org/lwjgl/util, 5th November, 2016.

[7] Jinput library page:

https : //github.com/jinput/jinput, 5th November, 2016.

[8] The gitHub page of this project:

https : //github.com/JBurant/Space − Camera, 5th November, 2016.

[9] Eclipse example of setting up lwjgl:

http : //wiki.lwjgl.org/wiki/Setting_Up_LWJGL_with_Eclipse.html, 5th November, 2016.

[10] UV mapping page on wikipedia:

https : //en.wikipedia.org/wiki/UV_mapping, 5th November, 2016.

[11] Themable Widget Library (TWL) for java:

http : //twl.l33tlabs.org/, 5th November, 2016.

[12] Nifty GUI library for java:

https : //github.com/nifty − gui/nifty − gui/wiki, 5th November, 2016.

[13] Lwjgl sphere javadoc:

http : //legacy.lwjgl.org/javadoc/org/lwjgl/util/glu/Sphere.html, 5th November, 2016.

[14] Video tutorial for loading a 3D model using lwjgl:

*https : //www.youtube.com/watch?v = izKAvSV3qk*0, 5th November, 2016.

[15] JarSplice utility homepage:

http : //ninjacave.com/jarsplice, 5th November, 2016.

[16] Tutorial on light intensity and attenuation:

*https : //github.com/mattdesl/lwjgl − basics/wiki/ShaderLesson*6, 5th November, 2016.

A Installation and usage

The software project, as well as the source codes and corresponding libraries can be found on Github [8], from there it can be forked or copied by anyone. As stated in section 2, the application has been developped in java programming language (JDK 1.8), using NetBeans IDE. Therefore, in order to run it, installed java is required. Apart from basic java libraries, the libraries: *jinput* [7], *lwjgl* [5] and *lwjgl util* [6] have been used. It can be run on any standard operation system (e.g. Windows, LinuxMint, Ubuntu etc.) Although, the native file is specific, therefore, it needs to be build separately for different operating systems, with different value of the native path.

The hardware requirements are not strict, any standard computer with installed java and decent graphics card is suitable. During testing, there was an issue with performance, using a notebook with just integrated graphics card, therefore a PC with standard graphics card is recommended.

The library lwjgl (lightweight java openGL library) enables to use the openGL in the java application, providing interface to the lower level C++ openGL methods. The usage of this library is straightforward, and is very similar to using normal openGL in C++. There are also many tutorials on the internet, providing guidance for the development. An important thing to notice is that the application is using lwjgl2. There is already newer version, lwjgl3, whose interface is slightly different and should not be mixed with version 2. The older version has been chosen, because the newer one didn't provide support for any GUI libraries at the time of development of this application.

The Github bundle contains also prepared NetBeans project, which has those libraries already imported. Only thing that must be changed in the project specification is the java path to the native libraries. To do that, right click on libraries in the project tab and select properties, in the newly opened window, in the categories select Run and specify VM Options as :-Djava.library.path="Path-to-lwjgl-library/lwjgl-2.9.3/native/Desired-OS" (A1).

In case, that future development will be carried on using another IDE, the libraries must be added, and the path to native libraries must be set. An example of how to do that in eclipse can be found in [9].

After the initial imports and after setting the path, no more setup should be needed.

The application can be compiled using the IDE and consequently run. It can also be run as already compiled .jar file.

In order to compile and deploy a new .jar file, a so called fat jar must be created. This can be done for example using the JarSplice tool [15], which packs the project libraries as well as corresponding native libraries into one executable jar folder.

Step-by-step creation of such "fat jar" is shown in the figures below.

In this case only java needs to be installed. Then it can be run for example from the console by typing: "java -jar CameraProjectFat.jar" As for the output of the program, the images are automatically stored to a folder with relative path "./img/screenshots", therefore such folder must exist beforehand.

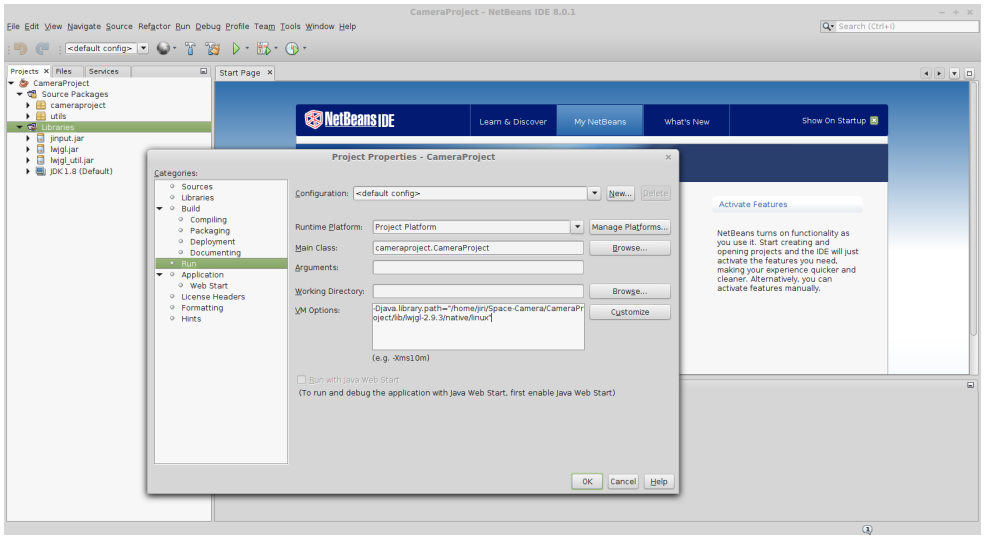

Figure A1: Setting up native path in NetBeans IDE

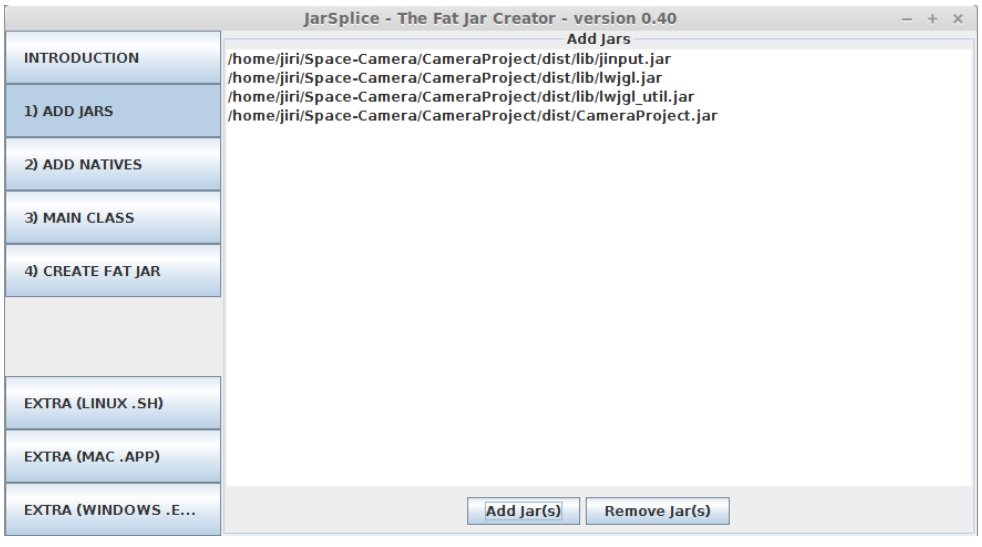

Figure A2: Building fat jar: Step1

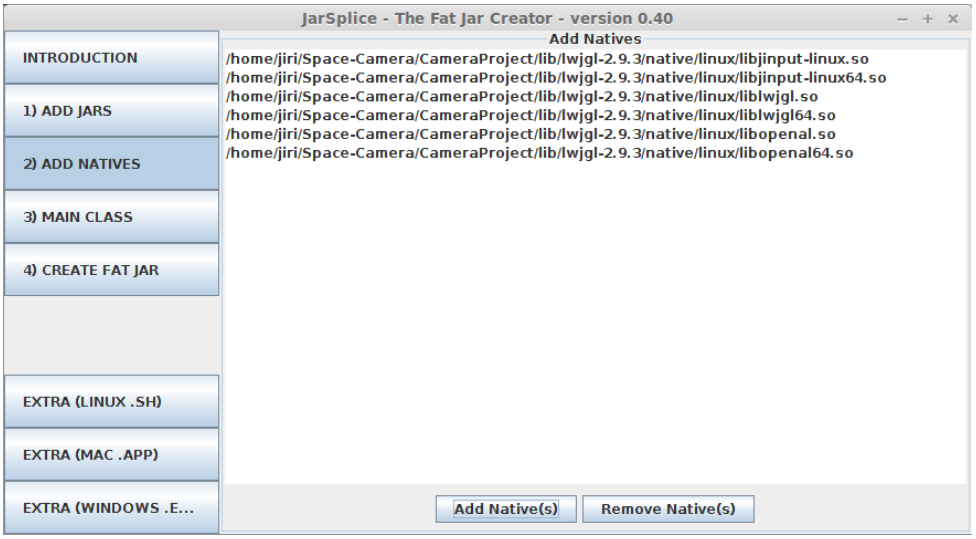

Figure A3: Building fat jar: Step2

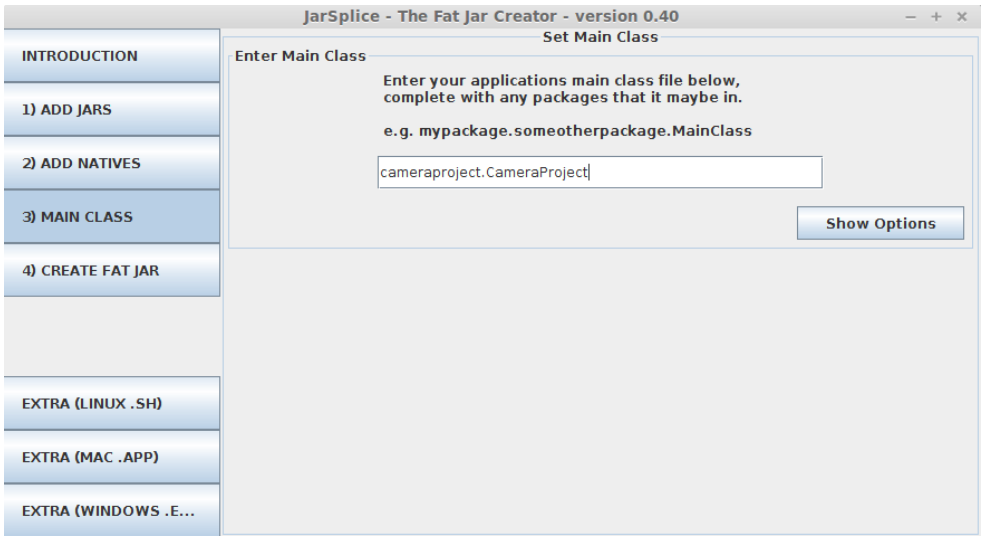

Figure A4: Building fat jar: Step3

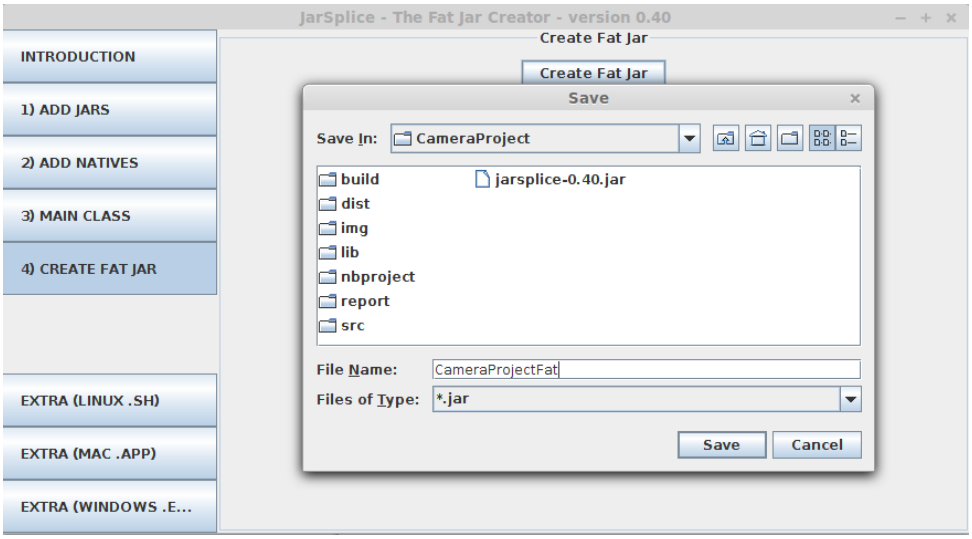

Figure A5: Building fat jar: Step4

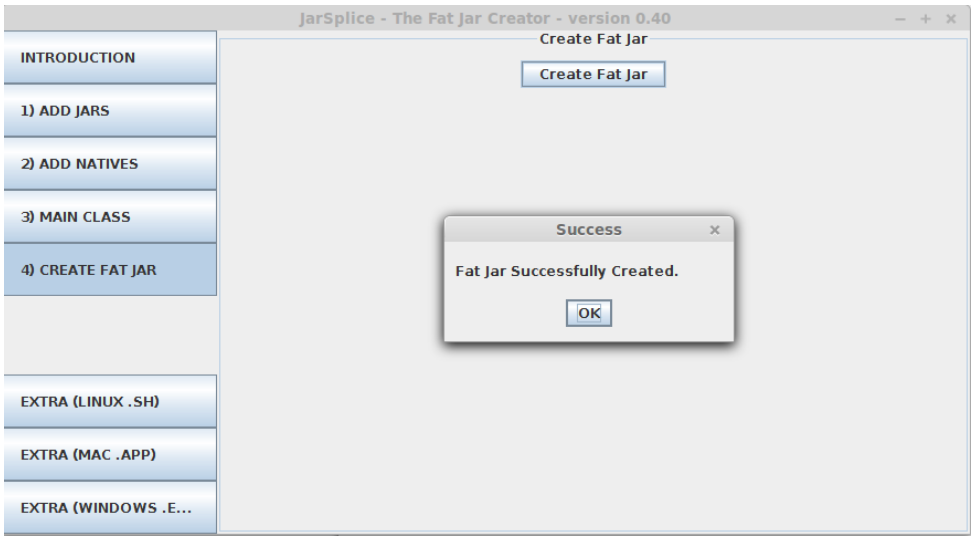

Figure A6: Building fat jar: Step5