Technological Feasibility Report
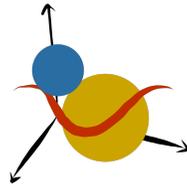
Team Triaxis

Sponsors:

Dr. Will Grundy, Dr. Audrey Thirouin

Faculty Mentor:

Sambashiva Kethireddy

Team Members:

Eleanor Carlos

Reyna Orendain

Andres Sepulveda

Brandon Visoky

Overview

This document outlines and details Team Triaxis' anticipated challenges, technologies, and integration for the continuation of the Licht software for Lowell Observatory.

Table of Contents

1. Introduction

With the universe being an ever-expanding, mysterious place, scientists and astronomers around the world have dedicated their lives to studying as much of it as they can. Over the last several hundred years, humanity's technological advancements have been able to help with this pursuit of knowledge, and with the advent of ever increasing computational power, we are able to visualize the universe in ever improving ways.

We are Team Triaxis and we are working with our clients, Dr. Will Grundy and Dr. Audrey Thirouin, on the project "Complex Asteroid Shapes in Modeling of Binary Asteroid Systems." We are responsible for delivering the goals put forth by our clients and Lowell Observatory for this project. This is the third year and third iteration of this project given to us by Lowell Observatory. Team Triaxis looks forward to continuing the development of this project as left to us by our former peers, Paired Planet Technologies and Team Andromeda.

Our iteration of this project includes implementing the code to render models of the asteroids observed by Lowell Observatory, our primary objective, to render them as triaxial ellipsoids. We hope that the modules we are implementing have a positive effect on the study of our universe, and will help directly help Lowell make more important discoveries about our universe.

Purpose

Knowledge of our surrounding universe and how it functions is crucial not only to scientists, but to humanity as a whole. Visualizing and studying asteroids and other objects in space is fundamental to understanding our universe and how it began.

Currently, faraway binary asteroid systems are studied by measuring the amount of light reflected off the asteroids with a telescope. The amount of light seen by a telescope over a period of time is represented as a lightcurve; since the amount of light decreases as one asteroid passes over another and casts a shadow, there is a "dip" in the middle of a lightcurve. However, the amount of data and information astronomers can gather by looking at the observed light curve alone can be limited.

Astronomers like our clients at Lowell use lightcurve graphs of the observed light measured with their telescopes and compare them with predicted light curve data. Our clients are able to tweak parameters in their predicted lightcurve so that it can better match the observed lightcurve data; if the lightcurves are similar in shape, then they can guess that the parameters they have match the actual characteristics of the binary asteroid system.

Problem

To address this problem of modeling binary asteroid systems, our clients, Dr. Will Grundy and Dr. Audrey Thirouin, have worked with students on two previous iterations of this project to build a program that allows users to input lightcurve data to render and model binary asteroid systems. The software that we'll be referring to throughout this document is called "Licht," which we will describe in further detail in our explanation of understanding previous years' implementations.

With any large project, certain goals were not met and further implementation goals have been requested. The requirements for this iteration of the project are as follows:

- **Fix Earlier Implementations:** In the last iteration of this project, the implementation of the module for modeling and ray tracing triaxial ellipsoids was introduced, but was unsuccessful and incomplete. Our goal would be to study the earlier partial implementation of triaxial ellipsoids and determine what needs to be fixed so that a full implementation of a triaxial ellipsoid module may be added.
- **More Complex Shapes:** Previous iterations of this project were able to implement modules for modeling basic shapes like spheres and faceted objects. Our clients would like the ability to render more complex objects, such as bodies in fluid equilibrium and possibly building in the framework for our clients to render any object given an arbitrary equation.
- **Improve Efficiency:** The more complex shapes this program is given, the more processing power and time it takes to render the data provided. We are to find a way to improve the efficiency of the rendering done by this program, should it be given more complex shapes.

Solution

Given these problems to solve by our clients, we have come to a generalized feasible solution for how to approach each problem.

- **Fix triaxial ellipsoid implementation:** We will be reviewing previous years' implementation of these modules to discover what issues there were with the triaxial ellipsoid implementation, and to fix them. As noted by our clients, this may require an entire rewrite of the modules for modeling a triaxial ellipsoid, but may be as simple as debugging a few lines of code.
- **GPU implementation:** To increase the efficiency of modeling these more complex shapes, particularly an implementation involving triangular facets, a GPU implementation is the most obvious and immediate solution. If this solution works, then it may be possible to expand upon it and rewrite past modules to use the same solution.

2. Technological Challenges

With any large project, it is necessary to take a look at the challenges that will be faced in its course so we are readily prepared issues that may arise. The challenges in this project include implementing new modules to the existing program, increasing the efficiency of the program, and ensuring that the product will not be difficult for users to utilize. This section will serve to address each of our initial concerns and potential challenges in this project.

Full Implementation of Triaxial Ellipsoids

The main goal of this project is implementing a working triaxial ellipsoid shape, namely making sure that the program properly renders shadows for triaxial ellipsoids. In previous projects, sphere and faceted objects were used to model the asteroids. Even though they are computationally simple, they are not structurally accurate; that is to say, asteroids do not look like perfect spheres.

Triaxial ellipsoids are closer to how an asteroid is shaped. Of course, not all asteroids look alike, but adding a module for modeling triaxial ellipsoids can help our clients study binary asteroids more effectively. We will begin by reviewing last team's code to see where they went wrong when trying to implement a triaxial ellipsoid module to model asteroids.

Implementation of Fluid Equilibrium

Although triaxial ellipsoids are closer to what asteroids look like than spheres, we may also want to see if it is feasible to implement more complex shapes or even more complex behaviors. For example, in our discussions with our clients at the Lowell Observatory, they mentioned that to get closer to how asteroids actually behave, we may want to expand the simulation to account for fluid bodies. This is when bodies in space "reach" for each other as they pass each other in orbit. Although this may be a goal to pursue after we are able to fully implement triaxial ellipsoids, it is still an important implementation to consider since we want our clients to be able to simulate asteroids as realistically as possible.

GPU Implementation

For any software implementation, code can be further optimized for better runtimes and performance. The previous iterations of this project have managed to parallelize a large amount of the code for the shape modules that are currently implemented in the Licht program. Our clients have discussed with us their wishes to model more advanced objects than spheres and triaxial ellipsoids, such as collections of triangular facets. These objects require a decent amount of processing power to render, and while this is possible to provide a CPU, completing these computations on the CPU would take a long amount of time since they would be competing with

other processes on the users' computers. Our users would prefer to spend less time waiting for computations to complete and more time completing tests with the program. A GPU implementation of this module or a GPU reimplementation of the rest of the code is a solution to this.

GUI Implementation

After consulting with our clients, we determined that the possibility of a GUI (Graphic User Interface) implementation for this program would be extremely useful to the users of the Licht program. Currently, to tweak the parameters in the program, our clients change the values in a text file every time they want to run the program.

Originally, we had thought that the main benefit of having a GUI implementation would be to provide an easy entry point for non-technical end users. However, our clients have pointed out that the additional value of having a GUI implementation would provide an "ease of testing" for our users. Being able to easily iterate through and modify the parameters associated with a binary asteroid system would be extremely valuable to the client. The GUI would ease the traversal of the problem space, as it would allow for less time editing text documents and more time analyzing data.

Understanding Previous Years Implementation

An inherent, yet equally important, challenge of being the third team to work on this capstone project is the "start-up" time associated with understanding the existing project. The past two teams have developed a C++ based architecture called "Licht." Licht is an API for modelling binary asteroid systems. It has a ray tracer that can render different 3D shapes like spheres and faceted objects. It also has a nonlinear minimizer (NLM) that our clients can use to find parameters that will best match the observed lightcurve data. Our clients then have the option to analyze the results of the NLM as data plots or rendered images.

Many members on this team are not yet familiar enough with the C++ language to begin with. We will need some time to learn the language so that we can understand the existing code put forth by the past two capstone teams and be able to develop further modules that are compatible with the current program structure.

Furthermore, we must understand the project itself to be able to effectively develop additional modules. Fixing the triaxial ellipsoid module, for example, requires us to have more than a cursory understanding of our development environment. Although this is not a technological challenge, for Team Triaxis to be able to deliver a quality product for the client, this is the first and foremost challenge for us to overcome.

3. Technological Analysis

This section serves as a more in-depth look into each of the technological challenges and the solutions we have deemed feasible to pursue for each of them. We will break down this analysis by identifying our desired characteristics for a solution for a particular challenge, descriptions and analysis of possible approaches that we have considered, and how we chose the solution that we have decided to pursue for our iteration of the project. We will also provide descriptions on how we will test the solutions that we have chosen.

a. Triaxial Ellipsoids

Introduction

The equation for rendering a triaxial ellipsoid is similar to the calculation for rendering a sphere; the difference between them is that triaxial ellipsoids have three different radii while a sphere only has one. Our clients are asking for a functional module for modeling triaxial ellipsoids. The previous team that worked on this project tried implementing the triaxial ellipsoid module already. Although they were able to model the shape of a triaxial ellipsoid, the module could not successfully render shadows. The decision that we face is between trying to debug the existing module for triaxial ellipsoids in the Licht program or building the module from scratch.

Desired Characteristics

An ideal solution for rendering triaxial ellipsoid shapes would allow our users to model asteroids as triaxial ellipsoids. The most important characteristics in have in our chosen solutions are as follows:

- **Development Effort:** We want to implement a solution that is not too extensive so that we can produce a function implementation by the end of the time we have allotted for this project.
- **Ease of Integration:** The existing modules of the Licht program are written in C++, which means we should continue to write in C++. We also want to make sure that any changes or additions we make to the module are able to be integrated with the rest of the existing modules.

Solutions

The shadows that binary asteroids project on each other cause variations in overall brightness over time, which we call a "lightcurve". This data carries information about the sizes, shapes, and mutual orbit of the pair. A solution to better render complex binary asteroids as triaxial ellipsoid shapes is to compare computed lightcurves to actual observed lightcurves. We could accomplish this by either debugging the existing module or starting a new one from scratch.

Debug Current Triaxial Ellipsoid Module

As mentioned, the team that worked on the previous iteration of our project has a partial implementation of a triaxial ellipsoid module. In their tests, the shape of the ellipsoid could be rendered successfully. However, in the ray tracing process, where light is projected onto objects, the program could not successfully render the shadows that would appear when an object was blocking another object from the light source. So, as posited by our clients, it would be beneficial for us to look into debugging the current module since it is already partially built.

Create a New Triaxial Ellipsoid Module from Scratch

On the other hand, our clients have also suggested that we start from scratch if we find debugging the current implementation of the ellipsoid module unfeasible. This solution would be beneficial if we found the structure of the current implementation unusable or if we weren't able to understand the code from last year's team successfully.

Analysis

Development Effort

- **Debug current module:** The existing module for the triaxial ellipsoid is already partially built and ready for us to debug. Plus, the previous team's code is very readable in that it is organized clearly and has extensive comments. Furthermore, we are in contact with the previous team and it would not be difficult for us to ask them questions about the current implementation of this module. As a result, the development effort for this solution would be low because most of the module is already built for us and it would not be difficult to debug since we have many resources on how to understand the existing code.
- **Start from scratch:** Adding a new triaxial ellipsoid module from scratch would take more work and time to develop since starting from scratch inherently takes a significant amount of development effort. This would require going through the design process for this module all over again; we would have to lay out the structure for the new module and test/debug it again. This would also include learning how to ray trace an object from scratch. Although learning how to ray trace is not impossible, it would take a significant amount of development effort for all of use to learn how to ray trace. However, the development effort for this solution will be high.

Ease of Integration

- **Debug current module:** Integrating this module into the rest of the program wouldn't be an issue, since the module is already written in C++ like the rest of the Licht program. Plus, since most of the module is already built and integrated with the rest of the program, we would only have to make sure that we stay within the bounds of what has already been built

as we debug the file. The ease of integration of this solution with the rest of the program is high.

● **Start from scratch:** Since most modules for this project are written in C++, we would have to write a new module in C++ as well. We would also be cautious about adding new code to implement the new module since it has to be compatible with the rest. We would have to make sure that the new module is integrated with the rest of the program in the same way as the original module. So, the ease of integration for this solution is moderate.

Chosen Approach

Between deciding to debug the current triaxial ellipsoid module or start from scratch, it is clear based on the analysis which one we lean more towards. In this case, continuing the work on the current module seems the most feasible.

Analysis of Possible Triaxial Ellipsoid Implementation Methods			
Possible solution	Development Effort	Compatibility	Ranking
Debug Current Implementation	Low	High	1
Start From Scratch	High	Moderate	2

The development effort and compatibility were the factors we considered when choosing this approach. Compared to starting from scratch, debugging the current implementation would be far more feasible since the existing code is readable and the current module is already highly compatible with the rest of the program.

Proving Feasibility

The implementation of the triaxial ellipsoid module for this project is highly feasible, it is only a matter of researching further into the triaxial ellipsoid problem, and seeing where the previous capstone team failed. To prove feasibility we would simply need to identify the problems and deliver a solution, then compare our model to the expected models to ensure it is correct. This will be our team's primary goal throughout this project.

b. Fluid Equilibrium

Introduction

Implementing fluid equilibrium would be a beneficial addition to the Licht program because our clients will be able to more accurately predict the behaviors and characteristics of the asteroids that they're studying. According to our discussions with our clients, fluid bodies are when the asteroids are reaching for each other. The question for us is how we would actually implement fluid bodies. Would we make another shape like triaxial ellipsoids, or would we have to implement another module for modeling three-dimensional shapes altogether? This section will go into detail with our analysis of the possible solutions, as well as our conclusion on the best way to implement them.

Desired Characteristics

An ideal solution would be to add another option to the user input text file that is currently being used by our clients to tweak different parameters. Currently, only faceted objects and spheres are supported, with plans to implement triaxial ellipsoids in this iteration of the project. If we implemented another object for fluid bodies, then we would add another set of parameters to be tweaked inside of the text file, similar to how spheres or ellipsoids can be tweaked.

In analyzing possible solutions, we had to consider some key characteristics:

- **Ease of Integration:** Our solution will need to be implemented into existing code structure so that we do not have to build the entire program from scratch. Part of our overall challenge is to build off of the work that past teams have already done, so keeping it compatible with existing code is beneficial.
- **Ease of Use:** We must ensure that users are able to use the final product with ease. In particular, our current user base consists of those who are familiar with past implementations of our product. We want to ensure that our clients are able to use the module similarly to how they have used other modules in past iterations of the project or make sure if they have to learn how to use new software, they are able to do so with relative ease.
- **Development Effort:** We want to implement a solution that is not too extensive so that we can produce a function implementation by the end of the time we have allotted for this project.

Solutions

Based on expectations for an ideal solution and our research, we have determined to analyze the following possible solutions:

Adding Another Shape Module for Fluid Bodies

One possible solution would be to add another shape module like the sphere and ellipsoid classes. These modules were developed by the past two capstone teams. Each class contains information about how to model each shape, including where they are in virtual space, how big they are, and how to determine where light from the sun is hitting it. The code found in those files are used in NLM and the parameters are tweaked in the user input file that our clients use. When the program is executed, a series of renders are generated based on the information input by the user. Currently, faceted objects and spheres are supported by the program, with plans for implementing triaxial ellipsoids underway. Based on the existing class files, we could implement fluid bodies in the same way.

Adding a Module for a Generic Shape

Another possible solution would be to build a new program where users could insert an equation for any three-dimensional shape and model that shape. We are considering this solution because it was brought up by our clients. Similar to how our clients have been tweaking different parameters in a user input file to change what the modelled asteroid looks like, users could adjust parameters in an arbitrary equation for modeling a generic three-dimensional shape in this new program. According to a paper from 1995 [1], "a *generic* shape model being able to represent a class of objects is often more useful in many practical applications where the objects of the same class are not identical in shape." Not every asteroid is identical in shape or size, so a generic shape model may be useful. In the same 1995 paper, generic shape modeling is used for "recovering 3-D shapes of objects from 2-D images."

Analysis

Ease of Integration

- **Adding a module for fluid bodies:** This solution would have a very high ease of integration because it would be based on existing code. Since it would be based on the existing Licht program code, integrating this module to the existing code would not be difficult.
- **Adding a module for generic shape:** This solution would also have a high ease of integration because it would have the same structure of another shape module. However, its content would be different in that the module would need to be able to take in any equation for a generic shape. Then, it would have to determine how to appropriately process it for ray tracing.

Ease of Use

- **Adding a module for fluid bodies:** Again, since this solution would be based on the existing modules in the Licht program, this solution would have a high ease of use for our users. Our users are already familiar with the current version of Licht, so the only challenge for them would be figuring out how to tweak the new parameters in the fluid bodies module.

- **Adding a module for generic shape:** This solution would also have a high ease of use. It would be designed similarly to the pseudo-GUI platform that our users are familiar with. Similar to the module for fluid bodies, the only challenge would be for our users to learn how to use the new module.

Development Effort

- **Adding a module for fluid bodies:** In order to develop this module, it would take a moderate amount of development effort. Fluid bodies have parameters like acceleration and components of attraction according to a paper [2] that one of our clients provided. Plus, since fluid bodies reach for each other when they pass each other, the shape of the model would change. This is different from previously implemented shape modules because the shapes themselves were always static and didn't change. However, now the shape would have to change during the simulation.
- **Adding a module for generic shape:** The implementation of a generic shape module would take a high amount of development because we would need to figure out a function that could render a shape with an arbitrary equation. Shapes are rendered with ray tracing, where vectors are drawn from the virtual camera, through a view plane, and then either intersect with an object or don't intersect with any objects in the virtual world. There are specific intersection functions for each of the shapes that have been implemented. It would be a significant challenge to develop an intersection function that can be applied to any generic shape.

Chosen Approach

Both approaches that we have considered for implementing fluid bodies have high compatibility with the existing structure of the Licht program and would be fairly easy for our users to utilize. However, the ultimate factor that led us to choose the approach to pursue was development effort; implementing generic shapes would be a much larger challenge than simply implementing another shape file for fluid bodies.

Analysis of Possible Complex Shapes Implementations				
Possible solution	Ease of Integration	Ease of Use	Development Effort	Ranking
Only Fluid Equilibrium	Very High	High	Moderate	1
Generic Shapes	High	High	High	2

If more time was available for this iteration of the project, we may have decided to pursue this solution. However, we have a limited amount of time and it is more feasible to commit to adding a

module for fluid equilibrium. Should we implement a module for fluid bodies successfully, we might be able to tackle a preliminary generic shape implementation.

Proving Feasibility

We would pursue this solution after we are able to successfully implement triaxial ellipsoids. We would test our choice by creating a new module for fluid bodies and then have our clients test it by tweaking the parameters in the existing user input file. This feedback would clarify whether or not our module executed what they were looking for in fluid bodies.

c. GPU Implementation

Introduction

Certain kinds of problems require many different kinds of solutions. For standard processing and computational algorithms, CPU parallelization may be all that is needed for a high level of efficiency within the codebase. This is not true for many kinds of graphical implementations, including the ray tracing algorithms and systems that are being implemented inside of this project. The GPU offers a much larger amount of cores for parallelization that these ray tracing algorithms can work with, meaning that they are done much faster than is possible on the CPU. The issues and challenges that may arise while implementing parallelized code on the GPU are multifaceted, though, and the largest issue we may come into contact with will purely be our inexperience with parallel processing.

Desired Characteristics

An ideal solution for our clients would be to figure out how to get these ray tracing algorithms to work on the GPU, and to be able to create a functional module that our clients could use for rendering asteroids. To reach the stretch goal that our clients have asked for in the original requirements, we would simply need a basic implementation of rendering on the GPU and to do some initial trials to gain better understanding of the computational power it would take for ray tracing these triangular facets.

Since our baseline goal is fairly low and a stretch goal, the desired characteristics of a GPU parallelization framework are fairly base level and include:

- **Compatibility:** A framework that is usable to develop and run on the users machine, namely ours and our clients machines.

- **Ease of Integration:** A framework that can be added to our existing codebase.

- **Easy Learning Curve:** Knowing that a GPU solution is something that none of us have ever worked with, the framework that allows us to begin working on a solution the quickest will be the best.

Solutions

There are several frameworks that we could use as a solution for this problem, of which we have narrowed it down to two: CUDA and OpenCL. These two frameworks seem to be the two more popular ones that are considered when working with similar applications, and so they are the two we will be looking at.

- **CUDA:** CUDA is a parallel computing platform that has been developed by NVIDIA, and as such is only available on NVIDIA hardware. It is a well established framework, being around for over a decade, and is still very well supported. It is widespread in its adoption, and we have several resources to be able to approach coding in this environment.

- **OpenCL:** OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs) and graphics processing units (GPUs). This is to say, systems using more than a single kind of processor. It has been around for less time than CUDA, but still over a decade. The benefits of OpenCL over CUDA is that it is not vendor specific, so we will not be limited to who can run and develop using this framework.

Analysis

The analysis between these two frameworks are rather straightforward and simple when judged by the factors listed above.

Compatibility

- **CUDA:** It is only operable on NVIDIA hardware. Several members in our group have access to this hardware and our clients have access to it as well.

- **OpenCL:** It is operable on any hardware, including NVIDIA hardware. This is a good option because if our clients were unable to acquire an NVIDIA GPU, they could still use the application.

Ease of Integration

In order to analyze the differences between these two frameworks, we had to look at how these two frameworks are integrated. Both offer a set of extensions for the C/C++ languages, which is perfect as that ties in exactly to the language of our codebase. This means that both frameworks satisfy this key characteristic.

Learning Curve

The easiest way to determine how difficult it will be to learn a new system is first to compare it to how similar it is to things we are already familiar with. Since we are all familiar with C programming languages in this group, that is a nice starting point. Thankfully, one of the members on our team has basic knowledge of the CUDA framework, which means there is already a leg up on implementing that solution.

Chosen Approach

Both CUDA and OpenCL would allow us to tie in their frameworks to our already existing codebase and would work on our clients machines. The biggest benefit CUDA has over OpenCl is that the resources we have to learn about and implement CUDA is far greater than that of OpenCl.

Analysis of Possible GPU Implementations				
Possible solution	Compatibility	Ease of Integration	Ease of Learning Curve	Ranking
CUDA	Pass	Pass	Moderate/Difficult	1
OpenCL	Pass	Pass	Difficult	2

For this table, we decided to list our best estimates of each of the key attributes we wanted our solution to have out of 5. From the research we have done, neither CUDA or OpenCL would be any more or less difficult to tie into our current codebase. Accessibility is where OpenCL has a much bigger advantage, since it is able to be run on any kind of system instead of just NVIDIA hardware. But the Ease of Learning Curve is where CUDA comes out ahead. We simply have access to so many more resources and points of contacts to work with CUDA than we do for OpenCL.

Proving Feasibility

Moving forward with our decision to use CUDA as our GPU parallelization framework, we will need to prove that we can implement and use this framework in any functioning capacity. To do this we will need to create a demo that uses this framework in some way, though maybe not necessarily a demo that integrates any part of the ray tracing algorithms used in the project. This will give a much better idea of how much work will go into a GPU solution for our clients, and how plausible it is as a stretch goal.

d. GUI Implementation

Introduction

Although a GUI would certainly be a beneficial addition to the project, making a quality GUI that will satisfy the client is a unique challenge. As posited by our client Dr. Grundy, well-designed GUIs

teach the user how to utilize the program effectively and allow discovery to occur as the user begins playing with the interface. The GUI must also be more feasible to use than a CLI (Command Line Interface) or the current NLM (Non-Linear Minimization) solution implemented. Otherwise, users will default to those options as they have in the past.

This is not the first time a GUI has been attempted by a capstone Team on this project. There was an attempt by Team Andromeda to utilize the GUI library called Qt. However, there were some issues that arose last year:

- **Incompatibility**: Qt has its own distinct build environment and many other idiosyncrasies that were difficult to mesh with the existing Licht C++ architecture.
- **Ineffective GUI Design:** Too much time was spent getting it working and not enough time spent to develop a quality GUI design. GUIs should teach the user; intuitive inputs and help popups to guide a new user to effectively utilize the program were points of feedback with the last team.

Because of these issues, the clients and team settled on utilizing the NLM package as a pseudo-GUI by turning off the fitting function.

Desired Characteristics

An ideal solution (i.e. framework or library) to the GUI problem would have the following key characteristics:

- **Easy to merge with Licht C++ Architecture:** An architecture-agnostic solution would most likely be best. The GUI is only useful to the client if it merges well with the existing product. This is the highest priority characteristic in determining the proper GUI solution.
- **Relatively Quick Startup:** Since a decent portion of this project involves getting up to speed with the language and the pre-existing modules, it would be most beneficial to develop a working prototype GUI as soon as possible.
- **Extensive Framework:** The number of input options in the GUI will depend on what exactly the client needs to input. The more potential options we can present to the client, the finer tuned the GUI will be to the end user.
- **Open Source Licensing:** When possible, we would like to work with an open source license on our GUI framework solution. The more freedom we have in utilizing the solution, the less time we have to worry about potential costs or roadblocks associated with licensed products.

A GUI, as noted by our client, should also have these characteristics to be considered excellent. Although not necessarily framework specific, these aspects of GUI development are still worth noting for future reference during development:

- **Descriptive and Intuitive Options w/ examples:** A quality GUI should not need a full user guide in order to be used properly (although documentation is almost certainly necessary). Inputs should be properly labeled with "example input" to lead the user towards proper utilization of the program.
- **Incremental Display of Inputs:** If a user does not need every single input in order to create a sphere, for example, they should not be presented with every input parameter. Conditional statements will be utilized in order to incrementally display necessary parameters for certain shapes and aspects of modeling binary asteroid systems.
- **Help Pop-ups for inputs:** A simple link to either a help pop-up and/or the documentation for that input parameter would help users understand how to properly use the GUI. For each parameter, the extent of the help pop-up's content would need to be determined in order to maximise the quality of the user experience.
- **Descriptive Error Messages:** When a user makes an error when utilizing the GUI, the program should tell the user what their error was. It would also be preferable to know how to fix it. Especially with inputs within a certain range, typos and mistakes happen. The documentation should account for possible errors, and the GUI should produce error messages that can be intuitively fixed, or easily found in documentation in order to fix.

Solutions

The following approaches are being considered, based on our research [3] and expectations for an effective solution:

Qt

Qt is a framework available for Linux, macOS, and Windows that allows us to create a GUI for our program. It is at the top of our potential solutions due to the fact that it is the baseline standard for GUIs developed by previous capstone teams on this project.

Although many C++ programmers may tell us to lean towards Qt, the previous iteration of this project was not able to utilize Qt successfully. From our understanding, Qt's architecture is not very useful if it hasn't already been included from the very beginning. User testimonies online only compound this; according to these testimonies, the framework is slow to build and takes up a lot of space/memory to be used properly.

Qt has both commercial and GPL3 / LGPL3 license options which allow us freedom to develop this application fully so long as we abide by the GPL3/LGPL3 guidelines.

wxWidgets

Whenever Qt is mentioned in our research, wxWidgets is touted as an excellent replacement. wxWidgets is a C++ library that lets developers create applications for Windows, macOS, Linux and other platforms with a single code base. It has popular language bindings for Python, Perl, Ruby

and many other languages. Additionally, unlike other cross-platform toolkits, wxWidgets gives applications a native look and feel because it uses the platform's native API rather than emulating the GUI.

It's also extensive, free, open-source and mature. There are plenty of resources and tutorials available on setting up and optimizing a GUI using wxWidgets, so the starting investment of learning wxWidgets is less overall than other options. The wxWindows Library License is similar to the LGPL license, so utilizing the library will also be easy to manage along with any open source requirements of development.

The only downside after research that users noted was a "convoluted API," presumably referring to the structure of how calls in the library are handled. This downside will hopefully be offset by the easy startup and documentation available, however it is a concern to take into account moving forward.

GTK+

GTK, or the GIMP Toolkit, is a multi-platform toolkit for creating graphical user interfaces. Offering a complete set of widgets, GTK is suitable for projects ranging from small one-off tools to complete application suites.

Although this was a recommended alternative to Qt, many users note how convoluted building and linking a UI is utilizing this toolkit. This may partially be due to appearances or structure of the API calls, but results seem inconclusive and subjective. GNOME would be required to be able to utilize this toolkit, which may not be a possibility depending on the hardware at Lowell Observatory. The license for GTK is also a version of LGPL (2.1 to be specific) so development for us can start quickly and with relatively little roadblocks.

Dear ImGui

Dear ImGui is a bloat-free graphical user interface library for C++. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline enabled application. It is fast, portable, renderer agnostic and self-contained (no external dependencies). This "no external dependencies" that was mentioned throughout the research stood out to us because of how lightweight this potential GUI could be.

Additionally, the MIT license allows us to be more hands-on with certain aspects of the library and potentially mold the framework in our favor. Furthermore, the proof of concept possible with Dear ImGui would be significantly easier than the rest of the options presented above, requiring only four files, no build process, and Not many people have remarked on forums on their thoughts regarding Dear ImGui, however having an "under the radar" open source solution helps us to understand the full scope of our potential choices.

Analysis

In the analysis of these solutions, our priority was to ensure that at least one of the desired characteristics were present in the option. When talking about what is considered "easy," we made sure that we had several different sources for opinions from varying communities. This worked much better than taking a cursory glance and assuming the nature of the framework, since many users had prior experience with the solutions in small to mid-scale applications.

Easy to Merge with Licht

When it came to easy merging with the previous Licht architecture, a considerably lightweight and system agnostic approach would prevail by this criterion.

- **Qt:** Although Qt is powerful, the capstone team from last year was not able to amicably implement it alongside the C++ already written.
- **GTK+:** GTK+ may not be as easy to merge, alongside the fact that there aren't much documentation or tutorials on how to develop it.
- **Dear ImGui:** We'd most certainly be able to implement a solution with this framework quickly, however being a lesser known framework would present problems down the line if systematically it does not deliver all the functionality we need.
- **wxWidgets:** In comparison to our other possible solutions, wxWidgets surpasses all of these options. WxWidgets has been touted as extremely easy to implement by user testimony regardless, so long as the language wrapper exists for the particular language. Since a C++ wrapper exists, it stands to say, based on previous user experience, that we'll be able to implement a quality GUI for this program.

Quick Startup

A framework that would be best in the 'Quick Startup' category would have a relatively easy to implement proof of concept (i.e. a "Hello Word" level prototype), and robust documentation in order to support the development of the GUI.

- **Qt:** Qt has plenty of documentation available, so at the very least there is reference material to support prototype development. However, based on the "Getting Started" guide for Qt, there are a lot of steps to get a proof of concept going.
- **GTK+:** GTK+ has a relatively quick proof of concept tutorial but it's not the quickest of our options.
- **wxWidgets:** wxWidgets is certainly faster in implementation as well.
- **Dear ImGui:** Dear ImGui is by far the quickest and least painful proof of concept on our list. There is no build process involved with Dear ImGui, just a few imports of .cpp files and we can drag and drop five lines of code anywhere in the main.cpp file to run a prototype.

Extensive Framework

An extensive framework is one that has several pre-built functions/widgets for commonly used GUI components, such as text inputs, buttons, and additional "pop-up" windows. This category is where the characteristics regarding a quality GUI come into focus, since we need to be able to provide the user with an elegant and user friendly GUI.

- **wxWidgets, Qt, and GTK:** These solutions have much of the same functionality with minor component changes that are more up to developer preference, and the documentation/wiki supporting the utilization of said components is quite robust for all three solutions.
- **Dear ImGui:** The only option that we find to fall short is Dear ImGui due to the fact that the documentation provided via Github Wikis is not nearly as in-depth as other solutions on this list.

Open Source Licensing

Although not much of a contest, the licensing of a framework is vital to ensuring that we're spending more time developing and less time paying costs or performing legal gymnastics.

All of the solutions presented have either the L-GPL/GPL, MIT License, or a derivative license of similar nature. Each of these licenses have unique quirks, however the main theme is that all of them allow the free use of the software for academic and small-scale purposes. This characteristic being noted is to simply ensure that we are not cut short with a potential solution due to restrictions we were not previously aware of.

Chosen Approach

In the end, we decided that wxWidgets would be the best solution. It is not only system agnostic, but there is plenty of community and documentation support for development of both the proof of concept and final product. wxWidgets has stood out to us as the best solution to start out with, with Dear ImGui coming in as a powerful contender for an alternative. Qt and GTK+ unfortunately require too many dependencies and have API's that are difficult to use; if we had chosen to use either one, they may have proven to be too burdensome to work with as the project continued.

Analysis of Possible GUI Implementations					
Possible solution	Easy to Merge with Licht	Quick Startup	Extensive Framework	Open Source Licensing	Ranking
wxWidgets	System agnostic, uses native API calls on supported systems	Plenty of tutorials and startup, robust documentation wikis and development resources	Hundreds of built-in classes, pre-existing modules for multithreading and video playback	wxWindows, similar to L-GPL.	1
Dear ImGui	Minimal dependencies, extremely lightweight, system agnostic	Lowest amount of startup work, 5 lines of code to get a proof of concept on line.	Basic functionality, may not have as robust options for GUI components	MIT	2
Qt	Previous capstone teams have had trouble with this, may not be easy	Extensive documentation and development resources	Design and development tools along with framework addons to fine tune the size of the implementation	L-GPL/GPL 3	3
GTK+	GNOME may prove difficult to work with, additional dependencies required	Subjectively complicated API, but proof of concept is relatively doable	Lots of prebuilt components with supporting documentation	L-GPL 2	4

Proving Feasibility

A technology demo that would successfully prove the technological feasibility of the GUI solution would be able to do the following:

- Compile without warnings or errors on both the team and client's Linux distribution.
- Show that inputs can be adjusted for a simple shape module.
- Properly display a rendered object to the user.

4. Technological Integration

Most of the initial challenges that we are facing with this project have been addressed in previous capstone teams. However, some issues that still need to be resolved include modelling triaxial ellipsoids and implementing a functional GUI. So, it is important for us to be forthright and honest with our expectations for the implementation of these technologies.

Current Integration Issues

The module for modeling triaxial ellipsoids was not successfully implemented in the previous iteration of this project; so, this is one of our biggest priorities this semester. This issue may result in a significant rewrite of a decent amount of their code, and may be one of the biggest starting hurdles of this project. We must make sure that any changes that we make to the existing module will not affect its compatibility with the rest of the program.

Furthermore, the previous iteration of this project does not have a successful GUI implementation. It is not entirely known why it was not implemented successfully, but we are taking into consideration the efforts of the previous team. One of our concerns with implementing a GUI is that we are not sure if it will successfully integrate with the rest of the program since we do not have a successful predecessor GUI in a previous iteration to base our expectations on. Although the platform agnostic approach of wxWidgets is beneficial, the Licht architecture is something we are not completely familiar with yet. As a result, we are not completely sure if it will integrate with the rest of the program. However, we are confident that we can build a quality prototype and that development won't be stunted by ineffective API architecture.

The final concerns that our team has in regards to a GPU implementation is that the implementation will involve the most development effort, including learning a new framework and building a new module from the ground up.

Future Integration Issues

Some possible integration issues that we have not faced yet are implementing fluid equilibrium and generic shape modules. Integrating these modules into the program will require further development than debugging existing code because we would have to build them from scratch. Plus, they will be structurally different from implementing other shape modules. With the current shape modules, we know the equations for rendering the shape and we know that their shapes won't change over time; however, with fluid bodies, the shape of the model can change during the time of the simulation and, with generic shapes, we don't know the equation of the shape until a user inputs it when running the program.

5. Conclusion

Lowell Observatory and astronomers' knowledge of the universe can only be as good as the tools they have to measure it. Our clients have tasked us with a project that two other teams in the past have worked on. Our mission is to offer a solution for modeling binary asteroid systems by adding modules to this existing project so that our clients can know more about the asteroids' characteristics. With this project come many new technologies for us, and with those come with many challenges.

Our major goals are to implement modules for rendering triaxial ellipsoids. Alongside this implementation, we also plan on implementing a GUI so that our clients can utilize the program more easily. If we accomplish these initial tasks, our next goal will be to see how we can build on our program to accept more complex shapes such as bodies in equilibrium and implementing rendering in the GPU so that the programs can run faster with quicker calculations.

In the table below, we have laid out our challenges, the solutions we have chosen for each challenge, the confidence we have in each of the solutions we have chosen, and backup solutions in case our chosen solutions do not turn out to be feasible in reality.

Technological Feasibility Summary			
Challenge	Chosen Solution	Confidence (1-10)	Backup Solution
Triaxial Ellipsoid Implementation	Debug Current Implementation	7	Start from Scratch
Implementing More Complex Shapes	Only Fluid Equilibrium	8	Fluid Equilibrium and Generic Shapes
GPU Implementation	CUDA	5	OpenCL
GUI Implementation	wxWidgets	8	Dear ImGui

Our targeted final product will be the full implementation of the modules that our clients have requested, including the implementation of a triaxial ellipsoid module at the bare minimum. An optimal final product has all of the features that we have defined as stretch goals, such as a functional GUI and GPU implementation of the triangular facets problem. These features will hopefully contribute to further Lowell Observatory's important research into binary asteroid systems.

References

1. Shen, Xinquan & Hogg, David. (1995). Generic 3-D Shape Model: Acquisitions and Applications. 970. 98-105. 10.1007/3-540-60268-2_285.

2. Chandrasekhar, S. (1967). Ellipsoidal Figures of Equilibrium - An Historical Account. Ellipsoidal Figures of Equilibrium - An Historical Account, 20.

3. Groarke, P. M. (2019, April 6). C++ UI Libraries • memdump. Philippegroarke.Com. https://philippegroarke.com/posts/2018/c++_ui_solutions/