# Technical Feasibility

**10/25/18**

**Paired Planet Technologies**
and
**Lowell Observatory**

Mentor:
**Isaac Shaffer**

Team:
**Zach Kramer, Brian Donnelly, Matt Rittenback**

*The purpose of this technical feasibility document is to layout our approach for modeling binary systems with light curves. This document lays out the challenges that we face, and evaluates possible solutions. We plan to have the accuracy determined by the user by either selecting a preset or manually inputting tolerances. The program will use OpenMP to parallelize all of the CPU code, and the Eigen math libraries for handling vector mathematics. The rendering and ray tracing will be done using Vulkan. We have a simple, robust solution that should be efficient and well automated.*

*Table of Contents*

# 1. Introduction

We are Paired Planet Technologies, a team of three under the mentorship of Isaac Shaffer. Together, we have strong experience in build and release systems, C and C++, and even CUDA programming. Our sponsor, Lowell Observatory, needs a better way to do light curve modelling of binary systems. Specifically, they are interested in modelling the light curve of two asteroids orbiting each other, which introduces new challenges compared to singular systems, such as taking into account the shadows that are cast onto each other.

## 1.1 Purpose

Understanding the properties of binary systems is vital to space exploration, as there are many binary systems in the Kuiper belt that we want to explore, but in order to do that we first need to understand the properties from the ground. Currently, Lowell thinks that the best way to infer these properties is by observing the light curves of the system, since often times the input data is extremely limited due to the distance of the system we are modelling.

## 1.2 Problem

Dr. Will Grundy and Dr. Audrey Thirouin at Lowell Observatory have tasked us with improving their current solution for doing this. Their *current* solution is not a *complete* solution that can be run from point A to point B. Instead, much of the work is done manually by the user and there exists only code to translate some inputs into more useful data. It is written in IDL, which is an outdated interpreted language that results in subpar performance, which is critical when the modelling can take days at a time.

## 1.3 Solution

Instead of expanding upon the IDL code, we have decided to build a solution from the ground-up. We are creating a C++ API that contains a forward model, which takes a variety of input parameters and outputs a matrix of brightness values and rendered images, utilizing ray tracing to do the modelling of the light curves and to render the images.

## 1.4 Document Layout

We will encounter technical challenges along the way, such as which libraries fit our requirements and how we will integrate all of these IDL functions into a C++ API. We begin by outlining the key technical challenges in Section 2, then analyze how to overcome them in Section 3. Since the challenges are mostly independent, we will also explore how we will glue all of our micro-solutions together into something that makes sense, which can be found in Section 4. The idea of this document is that afterwards we will have our bases covered and can launch into solution design.

# 2. Technical Challenges

The project has four major hurdles that need to be overcome for a successful final program. The concrete hurdles are rendering an image and performing ray tracing. The challenges of efficiency and accuracy are dependent on inputs and can vary depending on the inputs. These will be harder to measure, but there are certain metrics that can be used for comparison.

## 2.1 Efficient Solution

The code needs to execute relatively quickly compared to the current IDL scripts. The solution that the clients are using now works, but is so slow that it is not a viable option for solving larger problems. One of the main reason the clients asked for this new software is to have a solution that is fast and optimized. Regardless of how we choose to optimize, with parallelization on the CPU or CUDA or some other method, the final program needs to run significantly faster than their current implementation.

## 2.2 Render an Image

As part of the results set, we need to render an image, or a set of images, that accurately represents the data generated. This image can be black and white, but needs to clearly show the light curve results. The set of images can either be made as the program runs and then passed to the results, or the image can be rendered after everything has finished.

## 2.3 Accuracy

A major component of the software is math based. The largest section of computation time can be contributed to series of inner-loop mathematics. It is possible to use probabilistic computing, like Monte Carlo algorithms to accelerate this part of the code, but we need to make sure that any approximations we use are less than their error in observations. All of our approximations should be insignificant compared to other errors with data collection.

## 2.4 Ray Tracing

One of the major hurdles with the program is using ray tracing to determine shadows that the binary objects cast on eachother. To do this we will need the location of the object, the light source (the sun), and observer. We will need to convert all of the coordinates from keplerian to cartesian. Once we have the coordinates, we need to trace the light and generate a light curve model that would result based on different properties of the binary objects. The ray tracing can be excessively time consuming, so this part will need to be heavily optimized to be viable.

# 3. Technology Analysis

## 3.1 Performing Efficient Vector Math

### 3.1.1 Overview of Problem

Light curve modelling requires a plethora of mathematics, mostly coeff-wise operations on large vectors, along with a variety of simpler math functions like cosine. There are many C++ math libraries available for any need that we might have, the challenge is choosing the best one. Because our input data is large, it is imperative that we choose one of the more efficient solutions that could possibly be parallelized.

The current solution does not use any external math libraries. The only non-primitive math is cos, sin, etc. There are derivatives, but they are generated with simple, primitive math. This makes it seem like we do not need any external libraries either, but almost all of the math is performed on vectors -- IDL provides built-in vector-math, whereas the C family does not. It would be easiest, and most efficient, for us to choose an external library that provides this functionality.

### 3.1.2 Alternatives

There are a few common options for vector and matrix manipulation: ATLAS, Eigen, INTEL_MKL, GOTO, gmm, and uBLAS. From our perspective, the number one aspect we are concerned about is performance. Ease of use is nice, but not a requirement, and nearly all the libraries offer identical functionality. The choice should be based on benchmarks. Here are three excerpts that compare these libraries:

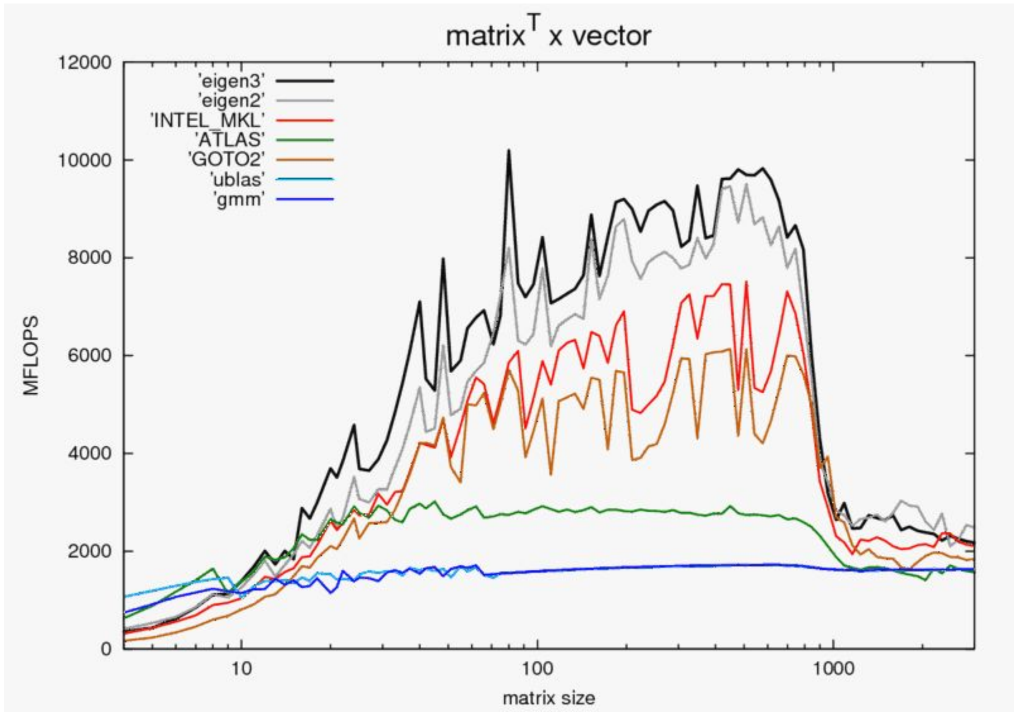

*Figure 3a: Matrix transpose multiplied by a vector*

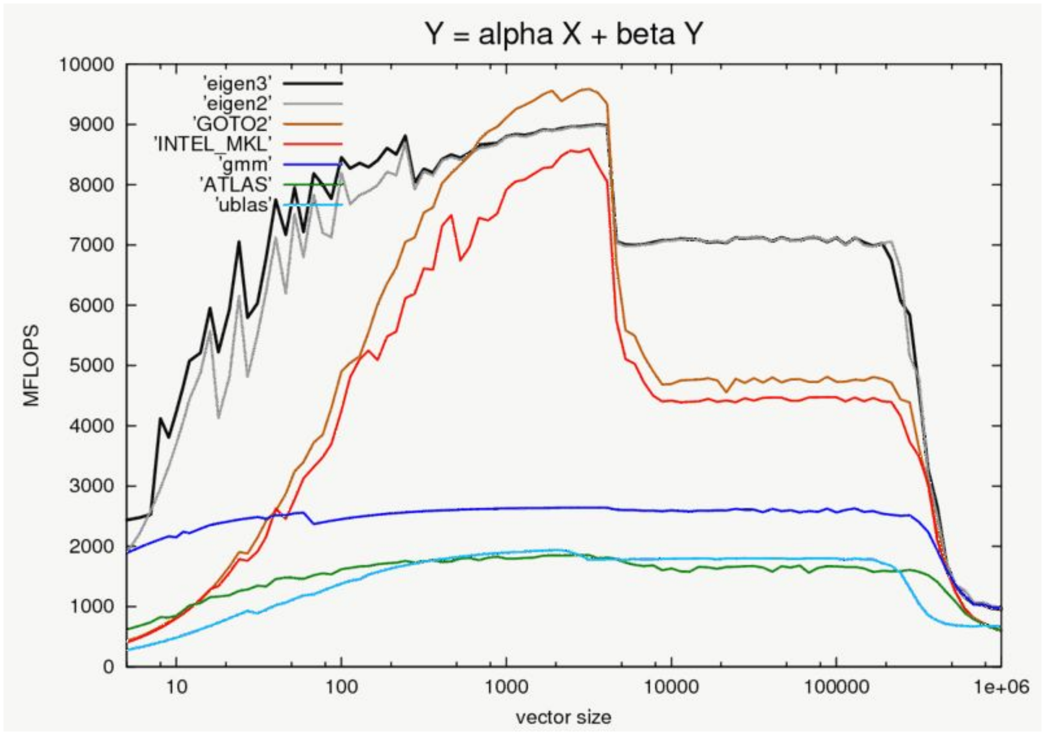

*Figure 3b: Y = alpha X + beta Y*

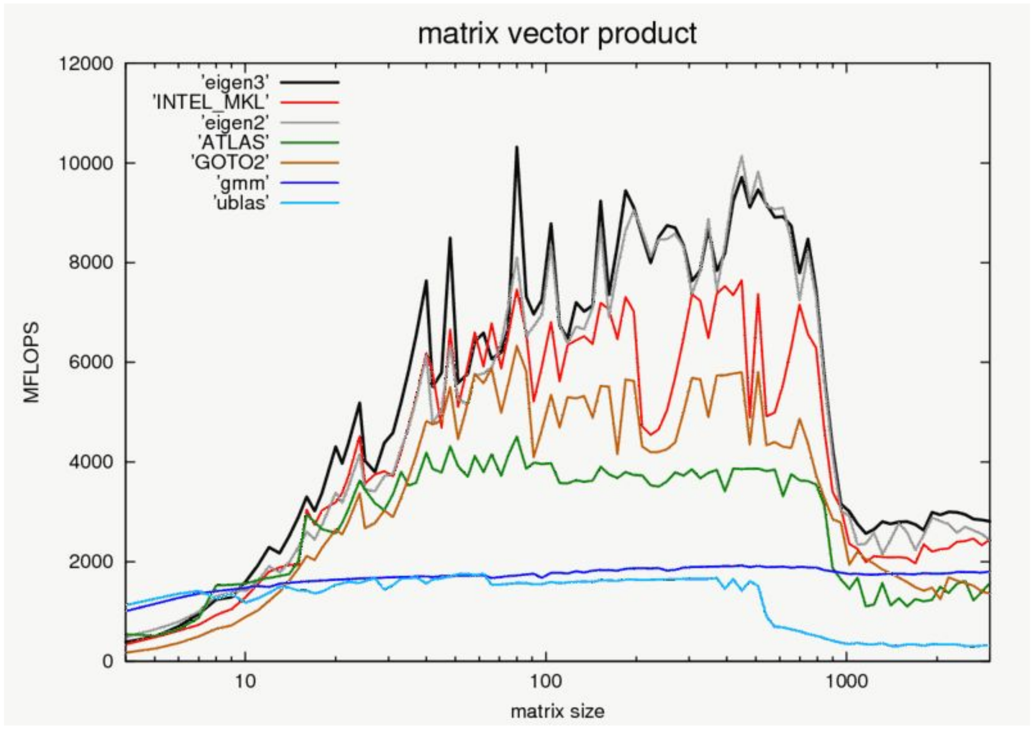

*Figure 3c: Matrix vector product*

As seen in the above figures[1], Eigen is the clear winner in most matrix and vector operations. Although uBLAS is shown as a weak competitor here, the CUDA variation, cuBLAS, is highly respected. BLAS by itself is inferior in performance to Eigen, since when handling complex expressions Eigen can optimize a whole operation globally, whereas BLAS requires that the user splits up complex operations into small steps, which introduces many temporaries. cuBLAS is so heavily parallelized that this downside is overlooked, but it does require an NVIDIA GPU. By design, Eigen is suited for the CPU and cuBLAS is suited for the GPU.

### 3.1.3 Chosen Approach

In conclusion, since we need some basic matrix and vector operations, we are choosing Eigen as the main library. If there are any large calculations that can optimized on the GPU, we will also utilize cuBLAS. This decision is solely based on their performance and community support, meaning they are some of the most common libraries and have sufficient online resources for troubleshooting.

We also need some common math functions, but there is really only one solution to this, which is the built-in solution called cmath, or math.h, which contains simple built-in math concepts, such as sqrt, cos, and pi. These simple functions are streamlined and almost every other math library utilizes cmath (in fact, Eigen includes it, so we technically do not have to).

[1] Math Library Benchmarking http://eigen.tuxfamily.org/index.php?title=Benchmark

### 3.1.4 Proving Feasibility

Feasibility is already known due to previous experience with Eigen and looking at the operations we need to perform. Nevertheless, we have implemented a simple working example of Eigen that tests a variety of operations. We have also implemented a significant section of the existing IDL code base using Eigen to perform the array operations. To prove cuBLAS, we can write similar tests. Currently there are no large matrix operations occurring, so it's not realistic to implement it in the code base yet, though we have used cuBLAS in the past on matrices so we know of its benefits and limitations.

## 3.2 Accuracy Selection

### 3.2.1 Overview of Problem

The time it takes for a model to be generated is considerable; one way of offsetting this is by reducing the accuracy. The user will need to be able to select the accuracy in some meaningful way. This can either be as a tolerance on the results or as an arbitrary label that we translate into a tolerance.

Some of the major difficulties in handling accuracy will be to check the accumulation of approximations made and the error from those methods. We will need to pass the error amount from one section to another so we can keep track of it. The final result should also include an error value so that the user can see how accurate the results are.

There are a few opportunities to optimize the code at the sake of accuracy, the largest of which is in ray tracing, where algorithms like Monte Carlo can be utilized. The accuracy on these can be low and we might not be able to use them if the required tolerance is very tight. We will most likely need different workflows based on tolerances. An exact answer will most likely not be possible, but a very low tolerance may need to use slower but more accurate approximation methods. There are also many spots in the modeller that rely on tolerances and set iterations -- we can also adjust these for accuracy. Finally, there are internal optimizations that can be done, such as using fast math and lookup tables.

The most important aspect of this issue is to allow the user to run inaccurate simulations in a short amount of time. Then once they have some idea of what exactly they want, run a longer simulation that produces a much more refined result.

### 3.2.2 Alternatives

We have two major options for how the user can choose an accuracy. First, the user can select tolerance values. These tolerances can be used in the approximation methods so that each approximate result is within these tolerances. This could be useful for the user because if they know the tolerances of the observations made then they can select either the same tolerances, or ones just slightly higher. This way, the program doesn't need to waste time creating a result that is more accurate than the data, which would be a waste of time.

The second option is to have a presets. This could just be a choice between three options, low, medium and high. From the value they select, we can then generate tolerance values. There could also be thresholds, so that higher than a certain level, we would use different methods. For example, if the accuracy was high, we might not use a Monte Carlo ray tracing method, but rather a more accurate one that takes significantly longer. With this approach to accuracy selection, we could hard code in some workflows, and the user might not have to know exact values, but rather just choose an option that they think matchs how accurate they want their results to be.

### 3.2.3 Chosen Approach

The chosen approach for this option is to combine the selection option and the direct tolerance input options. This will allow the user to default to a preset option, which we can then interpret to our own tolerances, or to select an advanced mode. The advanced mode will just let them directly control the tolerances they want, and the methods used, for each approximation. This gives the user an easy option when they don't care about specifics, but also lets them dial in the code if they think that it is necessary.

### 3.2.4 Proving Feasibility

To prove feasibility, the best approach is to first code a working model of the software and be able to generate some output with it. From this output, we can create a simple benchmark that estimates the "accuracy" of our results. This, in itself, is a challenge that may not even be possible, due to lack of reference data. Even without it, we can manually test each major tolerance and iteration count and come up with sets of them that correspond to an estimated "quality" value. This is a task that happens towards the end of coding since we cannot make something that we don't have more accurate.

Proving this is unneeded though, since we already know it is feasible because: (1) it is used all throughout the industry across many fields and implementations and (2) we have many cancelling criteria and parameters that can be adjusted. The challenge is just implementing the *best* solution for our software.

## 3.3 Creating an Image of the Result

### 3.3.1 Overview of problem

The final result set should have the option of being accompanied by a 3D rendering of the data. This 3D model should be animated and show the binary system that was generated. The final result here should clearly show the shapes of the binary objects and how they orbit each other.

Our rendering options are somewhat limited because they will need to work cross-platform. The best cross-platform options available that work with C++ are Vulkan and OpenGL. These are both low level APIs that we can use.

### 3.3.2 Alternatives

The first option to create the final rendering is with the Vulkan API. The benefits of this are that it is fast and cross-platform. The tutorials for rendering in Vulkan are extensive, and it seems like the code will be easy to call within our C++ files.

Another option is to use OpenGL. OpenGL is an older API that can also be used within C++, and is cross-platform. OpenGL is only supposed to be slightly slower than the newer Vulkan, but in this application, should have almost identical runtimes and quality.

### 3.3.3 Chosen Approach

The best option here for us is Vulkan. Vulkan is a better option, not because its faster or better than OpenGL, but because we can also do our ray tracing in Vulkan. Since we are already using Vulkan for something, it just makes more sense to continue using it, and integrate the ray tracing and the rendering.

### 3.3.4 Proving Feasibility

A simple way to demo Vulkan is to create an image of an object. To make sure that Vulkan can do everything we need it to, the image should also be in motion. We will make a sphere that rotates and orbits a point in space. If we can do that much in Vulkan, then all we need to change for the final product is have the sphere be shaped, and add another object that syncs up with the rotation and orbit of our demo object. A demo of a spinning sphere that orbits a point should be sufficient to prove that Vulkan is capable of rendering any image that we need it to.

## 3.4 Ray Tracing for Binary Objects

### 3.4.1 Overview of Problem

Ray tracing needs to be done to solve for how the two binary objects will cast shadows on each other, and how that will affect the light curve. Rays of light need to be traced from the sun to the binary objects, and then back to the observers on Earth. Depending on the location of the two binary objects, they might be casting shadows on one another that affects how bright the pair appear to be. There will need to be enough rays simulated so that if the object is non-spherical, then the shadows should have the appropriate shapes and cause the correct amount of dimming in the observation.

There are a few different methods that we can use to do ray tracing for this part and some of them depend on the hardware available. It might be necessary to implement multiple methods that detect what hardware is available and then select the appropriate workflow.

### 3.4.2 Alternatives

The first option is to do the ray tracing on a GPU. If it is an NVIDIA GPU, then we can use OptiX, which is meant for their GPUs and also makes use of their new Turing architecture which has RTX cores. This would be the fastest implementation bar none, but it requires that the hardware be compatible. If we wanted an approach that worked on any GPU type, we could use Vulkan, which works on AMD cards also, and can make use of the new RTX architecture. Although Vulkan is less optimized than OptiX, it would work on any modern GPU. NVIDIA has also recently released an API specifically for ray tracing in Vulkan. Ideally we would just use their new API to do all of our ray tracing because it would be the fastest code avalible. Since the NVIDIA API is so recent though, we may run into problems and have to write ray tracing code ourselves. There are a number of examples of ray tracing in Vulcan, so that should still be doable, just harder.

Another option for ray tracing is to use OpenGL which runs on GPUs, CPUs, and hybrids like the Phi. This would make it so that we don’t have to have multiple workflows, and the OpenGL would handle which device it was running on. This might be slightly slower than other options, but it has the most documentation and community support.

A third major option is to create our own ray tracing methods that use approximation algorithms, like Monte Carlo, to get faster results, either in CUDA or C++. We would be able to control the accuracy manually this way, and keep track of everything, though it might be less efficient. If the ray tracing does not need to be accurate, then this might be the best method available to us.

### 3.4.3 Chosen Approach

The best starting approach for ray tracing is Vulkan. Vulkan is cross-platform, which means we will not run into hardware requirements like we would with OptiX. With the new NVIDIA API for Vulkan, we can also make use of the new Turing architecture that is designed for ray tracing. Vulkan is also the only option that we evaluated that has an API specifically for ray tracing that is cross-platform.

### 3.4.4 Proving Feasibility

The best tech demo here would be to take the object that we created for the rendering demo and add light. The light should cast a shadow and interact with the surface of the object. This is similar to what we would need to do in our final program, and give us a good idea of how much time it would take to program.

## 3.5 Reducing Runtime Through Parallelization

### 3.5.1 Overview of Problem

Generating the forward model can take a long time, but generating a backward model can take significantly longer. It is important for the clients that the code runs quickly, either on a weak machine like a laptop, or on the campus computer Monsoon for larger data sets. To increase the performance of the modeling, there are some raw optimizations that we can make. Without changing the accuracy we should still be able to get a significant speed up by doing a combination of a few things, but most importantly, parallelizing it.

### 3.5.2 Alternatives

The fastest way to increase the speed of the program is to parallelize it, but there are several methods that we can use in C++ to accomplish this. The first being POSIX threads. In C++, pthreads are fast but they are a code intensive way of creating parallel algorithms. We would need to lock out threads from memory access manually and make sure that there were no race conditions by hand.[2]

Another option is to use OpenMP threads. This would be easier to code in and we could scale the number of threads generated based on input from the user. This would make it easier to scale up and the parallelization would take less time to code in.

The final option is OpenMPI, which is the best at scaling. MPI would allow us to use multiple nodes on a computing infrastructure if they are available. MPI is also like OpenMP, in that it is easy to implement and can run on a variety of machines easily. Race conditions are also handled by the API so we don't need to manually account for them.

**Evaluation of Different Parallelization Methods**

| | Scalability | Speed | Coding Effort | Race Conditions |
|---|---|---|---|---|
| OpenMP | High | Medium Overhead, But fast runtime | Easy for single nodes, more work when scaling | Small Issue |
| MPI | Highest | Large Overhead, and medium runtime speed | Work upfront, but doesn't need anything to scale | Non-Issue |
| POSIX threads | Lowest | Usually Fastest for a given thread count | Ridiculously Excessive | Prevalent |

[2] See Pthreads and OpenMP by Henrick Swann, http://www.diva-portal.org/smash/get/diva2:944063/FULLTEXT02

### 3.5.3 Chosen Approach

The best option for parallelization is OpenMP. OpenMP is the easiest to write in C++, and has a default scaling ability that detects the optimal number of threads. This makes the code more adaptable for different hardware. MPI is just too much overhead if the code will be running on a laptop, while POSIX threads are too much work to code, and will have too many bugs for the amount of time we have to develop.

### 3.5.4 Proving Feasibility

We have already used OpenMP to parallelize some code that we translated from the clients IDL scripts. OpenMP showed notable performance increases over the sequential implementation, and was only a single additional line of code. It showed that OpenMP was easy to implement, and gave sufficient improvements in performance.

# 4. Technology Integration

## 4.1 Overview

We hope to use a variety of methods and libraries to make the program run quickly and maximize hardware availability. This can easily cause integration problems, because with the differences in hardware that the program could run on, we may have to adapt our solution. Right now, all of our options are cross-platform and should work on any CPU or GPU brand. In the future, it may be a good option to add specific options to benefit from things like CUDA parallelization for the calculations.

## 4.2 Integration Challenges

Our main integration issue is to go from inputs to a final rendered image. This means that we will have to take photometrics and convert them to objects that we can then use for ray tracing. The objects in the ray tracing can then be rendered. The best way to integrate the ray tracing and rendering is to use the same API and objects. To do that, we chose Vulkan, which simplifies everything nicely.

## 4.3 Future Integration Issues

With our current plan, integration is really a side issue. We need to be aware of it but there are no major hurdles for us to overcome. In the future though, we may run into some significant issue here if we try to slightly expand the scope of our solution. Right now, our solution is entirely cross-platform, and should run on most linux computers without any special configurations. However, a stretch goal for us is to make use of the computational ability of graphics cards directly with OptiX and CUDA. This would require us to check the availability of hardware at runtime and choose methods that maximize the available equipment. This may lead to integration issues.

If the program is running on different types of hardware, we will have to check compute capability of devices, driver versions, and a number of other parameters that may cause problems. Because of this difficulty, we have decided to not make this type of hardware optimization part of our initial solution. Near the end of the project, if we find that we have time, and the clients are interested, we may attempt to implement some of these types of optimizations. Up to that point, we should run into few if any integration problems.

## 5. Conclusion

Our project is progressing nicely, and we are confident in our ability to efficiently form light curve models by the end of the project. In the table below we outline the challenges, our solutions and our confidence level that the solution will work as we expect it to.

| Challenges and Solutions | | | |
|---|---|---|---|
| Tech Challenge | Solution | Confidence Level | Back Up Options |
| Performance | Parallelization/Eigen math Libraries | High | Boost/BLAS Libraries |
| Rendering | Vulkan | Medium | OpenGL |
| Accuracy | A selection option, and an advanced mode | High | Just an Advanced mode |
| Ray Tracing | Vulkan | Medium | OpenGL |

Our solution for this project is simple and robust. We are relying on technologies that are commonly used. Vulkan is the newest and least well documented technology that we will rely on, and even that has become something of an industry standard for graphics. The largest hurdle we will face is learning to use the APIs and libraries that we chose. Everything should integrate well in C++ and we are confident in our ability to create a program that is faster, has more options, and with a higher level of automation than the current implementation that is being used.