Software Design

Version 1

2/8/19

Sponsor
Lowell Observatory

Clients
Audrey Thirouin, Will Grundy

Team
Paired Planet Technologies
Zach Kramer, Brian Donnelly, Matt Rittenback

Mentor
Isaac Shaffer

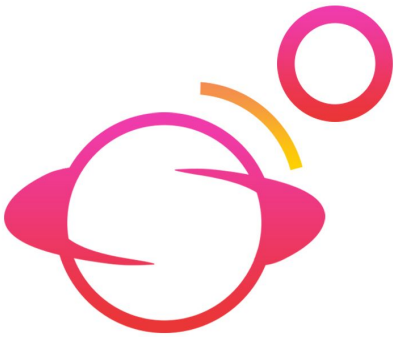

Table of Contents

1. Introduction

Space exploration has always enthralled humanity. Every year, billions of dollars are spent trying to understand our Solar system. Humanity sends probes to other planets and to small bodies. Humans have not stopped looking for answers in space since the time of Galileo. This search has driven technological development in all areas. Velcro, computers and GPS are just some of the technological byproducts of space exploration. Every day, observatories like Lowell are gathering information to expand the understanding of the space humanity lives in.

The clients, Dr. Audrey Thirouin and Dr. Will Grundy, work at Lowell Observatory to analyze astronomical data. They focus on analyzing data about small bodies farther from Earth in space which are harder to observe directly. Right now, they are working on modeling binary systems in the Kuiper Belt. To do this, they need to use special techniques which make the most use of the data available. For most objects that far away in the Solar system from Earth, only a single point source can be observed. That point source can be used to determine the object brightness at a given point in time. These luminosity recordings can be combined together to form a graph called a light curve, which displays brightness values over time.

Light curves can be used to infer properties about the objects that generate them. For example, an asteroid that is non-spherical will reflect more light when a larger amount of surface area is reflecting light from the Sun to the observer. Since the object is reflecting more light at certain points in its rotation, the brightness will be different depending on when it is observed in its rotation. Those brightness values can then be graphed. In most cases this will make the light curve sinusoidal. The rotational speed can then be found based on the period of the curve. The amplitude of the curve can be used to roughly guess at the proportions of the object. A large number of other characteristics can be inferred using light curves.

The clients want to make use of light curves to better understand binary systems, since they are extremely prevalent in the Kuiper Belt. Binary systems are composed of two objects that orbit each other about a point in space called a barycenter. The gravity from both objects affect each other, introducing new challenges in creating a model.

The clients want software that can model these binary systems and calculate light curves. They plan on using this project to generate potential models that fit the observed data of binary asteroid systems. Currently, the clients are modeling with fragmented code that is slow and lacks some functionality. This project's solution will allow the modeling of binary systems quickly, conveniently, and accurately. The design of such a solution is detailed in this document.

2. Implementation Overview

The implementation of a lightcurve modeler of binary systems is no trivial task. The main request of the client is to have an **API with consistent design, naming principles, and an otherwise integrated code base**.

This can be accomplished by refactoring the legacy code base or by creating a new one. The latter option is more time consuming and wouldn't be considered if it was just a matter of creating an *integrated IDL code base*, but there are other requirements. Specifically, the request for high-performance code led Paired Planet Technologies to choose the C++ language.

A lot of functionality is needed to generate a lightcurve, ranging from generating 3D shapes, to simulating their orbits, to tracing the rays from the Sun. Because of this, the code base is designed in modules. **Each source file is intended to be an independent sub-module that, by itself or combined with other sub-modules, forms a larger, conceptual module.** For instance, ray tracing is a conceptual module that involves both tracing a ray (Ray submodule) and calculating the reflected luminosity (Hapke submodule).

These modules are utilized by a main function called the forward model. The forward model is exposed as a C function, which allows IDL to call it. From a user perspective, this is the only function of the API. The input consists of many variables and file paths and the output returns a lightcurve and a 3D rendering.

Due to the nature of this project, where the functionality is composed almost purely of custom calculations, **there are no significant frameworks being used**. Some infrastructure-related frameworks are in place:

- To better support the project's dependencies and to be compatible with multiple technologies and platforms, **the build-system uses CMake**. CMake is a variation of *make*, but allows for deeper configuration of a make system.

- Given the amount of numerical calculations in this domain, and that many iterations of optimizations are expected, unit tests are also a must. Unit tests allow for validation that each module of the software performs as designed. **Google Test is the de facto C++ testing framework and is being utilized in this project.**

- Finally, this domain also performs many calculations on vectors, which C++ poorly supports. **Eigen is a C++ linear algebra library that provides convenient and efficient data structures** that support the needed calculations, such as coefficient-wise operations, so it is being utilized in this project.

3. Architectural Overview

The system is comprised of six conceptual modules. The main driver module is the forward model. The forward model makes use of all of the other conceptual models to generate a light curve. The conceptual modules are designed to provide a specific functionality of the code. A module's or submodule's functionality can be easily utilized by other modules. This leads to a robust and easily maintained solution.

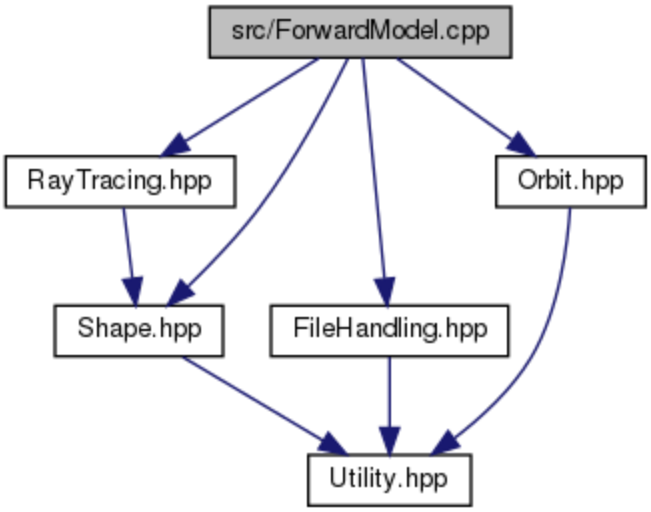

Figure 1: How the conceptual modules interact

Forward Model

The forward model drives the functionality of the software. This module makes either direct or indirect use of every other module. The forward model uses the orbit module for finding the properties of the non-primary object. The module uses the ray tracing module for finding creating the light curve and rendering images. The forward model uses the shape module for storing and interacting with binary objects. The forward model also contains its own functionality which calculates the vertices location of the prime object for every time step.

Orbit

The orbit module finds the locations for the non-primary objects. The orbit module uses a large set of parameters to find locations relative to the primary object. Once everything is found relative to the primary, it then translates the locations to the overall coordinate system.

File Handling
The file handling module reads in all of the settings. The file handling module takes in ephemeris data and translates it to the program's coordinate system.

Ray Tracing
The ray tracing takes in the shapes and runtime parameters from the forward model and returns a luminosity value. The ray tracing module also has the ability to render an image and save it to file. The ray tracing creates rays to find the luminosity of every pixel which are summed for a brightness value.

Shape
The shape module holds all of the information for the objects being modeled. This module has submodules for dealing with facets and function based shapes. The shapes will either be custom and composed of facets, or premade and defined by a function. The premade shapes will be faster in the ray tracing module. The custom shapes will be stored as a set of vertices that define facets. Any custom shapes are uploaded into the shape module and then passed to the forward model.

Utility
The utility module holds most of the helper functions that the other modules use. This includes the Eigen math library and functions for translating between units and coordinate systems.

All of these modules come together to provide a comprehensive software solution. Each module is made up of a set of smaller modules that are detailed in the next section.

4. Module and Interface Descriptions

The current version (version 1) contains the alpha prototype design for the modules. Since the overall design is an API, there is a very high level of modularity. Each major module is semi-independent and only loosely interfaces with the others. Because of the minimal cohesion, the way the modules are linked is loosely defined. Figure 2 shows the overall design of the modules and how they depend on each other.

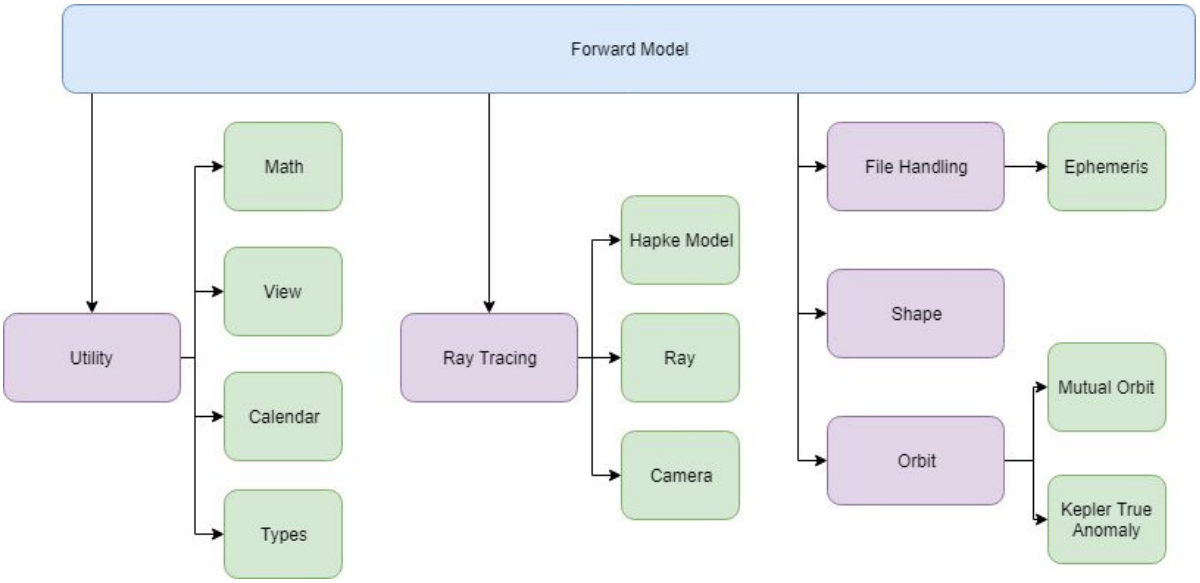

Figure 2: How the code base is split into conceptual modules

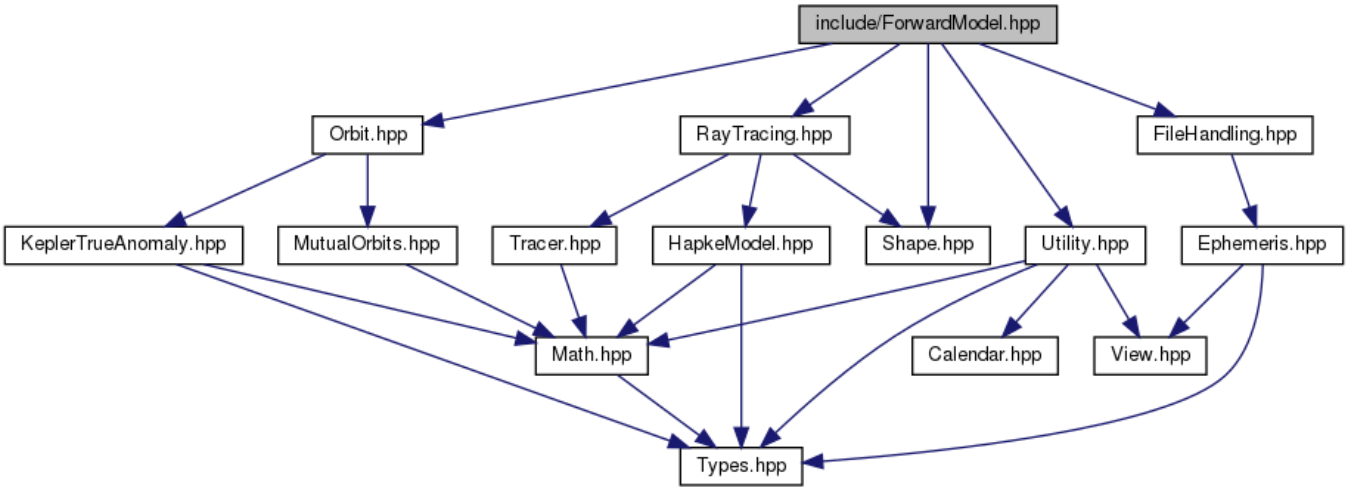

Figure 3: How the submodules interact -- reference diagram for all modules

Figure 3 displays the conceptual submodule interactions. In this section the submodules are detailed. The foundational module, Utility, is used by the all of the other modules.

4.1 Utility

The Utility module provides tools necessary for the program to manipulate, contain, and display data as needed. It holds 4 submodules: Calendar, Types, View and Math.

4.1.1 Calendar

The calendar submodule converts Gregorian to Julian (J2000) date and time.

Dependencies

- None

Use cases

- Ephemeris data uses a Gregorian date system and the astronomical calculations use a J2000 date system. The calendar function converts the dates for the forward model.

Design

getJulianDay()
This is the primary function of the submodule and calculates the Julian Day Number for a given month, day, year, hour, minute, and second.

Input:
- Year value
- Month value
- Day value
- Hour value
- Minute value
- Second value

Output:
- Double representing the Julian Day Number of the given calendar day and time.

4.1.2 Types

The use of custom data types allows for other modules of the software to reference more complicated data types with quicker and simpler names and provides control of these data types in one central file.

Dependencies

- None

Use cases

- In Figure 3, the types submodule is used by most submodules. This is due to the convenient and useful data types that it provides. As more common types are needed in the code base, they get added here.
- Specifically it provides the ArrayN1 type, which allows us to create arrays with access to coefficient-wise operations.

Design

The submodule is mostly composed of type aliases and some helper functions that print or otherwise manipulate the type aliases.

ArrayN1

An ArrayN1 type is an array that has a dynamic number of columns (N) and one row. ArrayN1 is a Eigen::Array type which provides access to coeff-wise operations.

printArray()

This function prints out the contents of a ArrayN1 variable.

4.1.3 View

A View is a vector of 3D vectors. Each column represents an observation time that contains a vector from the viewer to the center of the primary.

Dependencies

- None

Use cases

- Data read from the Ephemeris tables gets stored in a View.
- The View is then used to calculate the location of the two bodies in the system at the given times.

Design

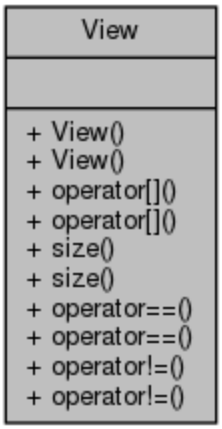

Figure 4: UML diagram of View class

View is a class that acts like a data structure. The API of the class is simply a series of overloaded operators, so that it acts like a normal data type.

It can be constructed with 3 parameters or just with the number of observations and then populated later.

Data
When first constructing a View, the object needs to store the right ascension, declination, and delta values inside an ArrayN1. Before storing these values, the values need to be converted into radians and kilometers.

Constructors

- Taking in right ascension, declination, and delta
  This constructor takes in 3 parameters -- right ascension (hours), declination (degrees) and delta (AU). It creates a 1D array containing 3 element vectors which hold the corresponding input parameters. This constructor also converts the input parameters into desired units for the 1D array, with the right ascension and declination converted to radians, and delta converted to kilometers.

- Taking in number of observations
  This constructor creates an empty View object instantiated with 0, and determines the number of given columns for the View based on the number of observations.

4.1.4 Math

The math submodule contains useful, independent arithmetic functions and constants for use by other modules.

Dependencies

- Types

Use cases

- Nearly every submodule uses Math for its useful constants, such as converting from degrees to radians.

Design

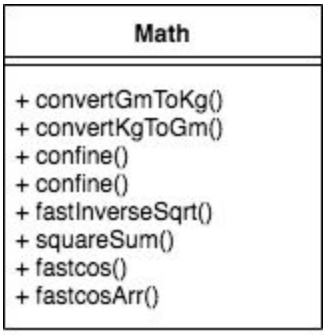

Figure 5: UML diagram of Math class

Data
This submodule provides some useful constants for the sake of consistent calculations inside other modules such as Pi and its variants and degree/radians converters. There are also look up tables for cosine to provide a faster computation at the cost of accuracy.

convertGmToKg() / convertKgToGm()
These two functions convert mass numbers between GM units and kilograms.

confine() *(double / vector)*
This function can be overloaded with a double or a Eigen VectorXd parameter. It confines a value between two limits.

fastInverseSqrt()
This function calculates the inverse square root using a well-known fast approach, as multiplying by an inverse square root is faster than dividing by a normal square root and provides a small amount of optimization in our calculations.

squareSum()
This function adds the elements of an array together after each element has been squared.

fastcos()
This function uses lookup tables to calculate the cosine of x for the purposes of speed. This provides a faster result at the slight cost of accuracy. The input x can either be an array or double.

4.2 Orbit

The orbit module holds, organizes, and otherwise manipulates orbital parameters. It is broken up into two submodules: holding mutual orbit parameters, such as spin state and location, and solving Kepler's equation for an elliptical orbit.

4.2.1 Mutual Orbits (Zach)

The first submodule is called Mutual Orbits and hence handles Keplerian mutual orbits. It has two major components: a dynamic data structure that holds all of the needed data and some functions that manipulate the data.

Dependencies

- Math
- Types

Use cases

- After the forward model creates a primary and secondary object, the mutual orbit submodule calculates the orbit of the secondary object.

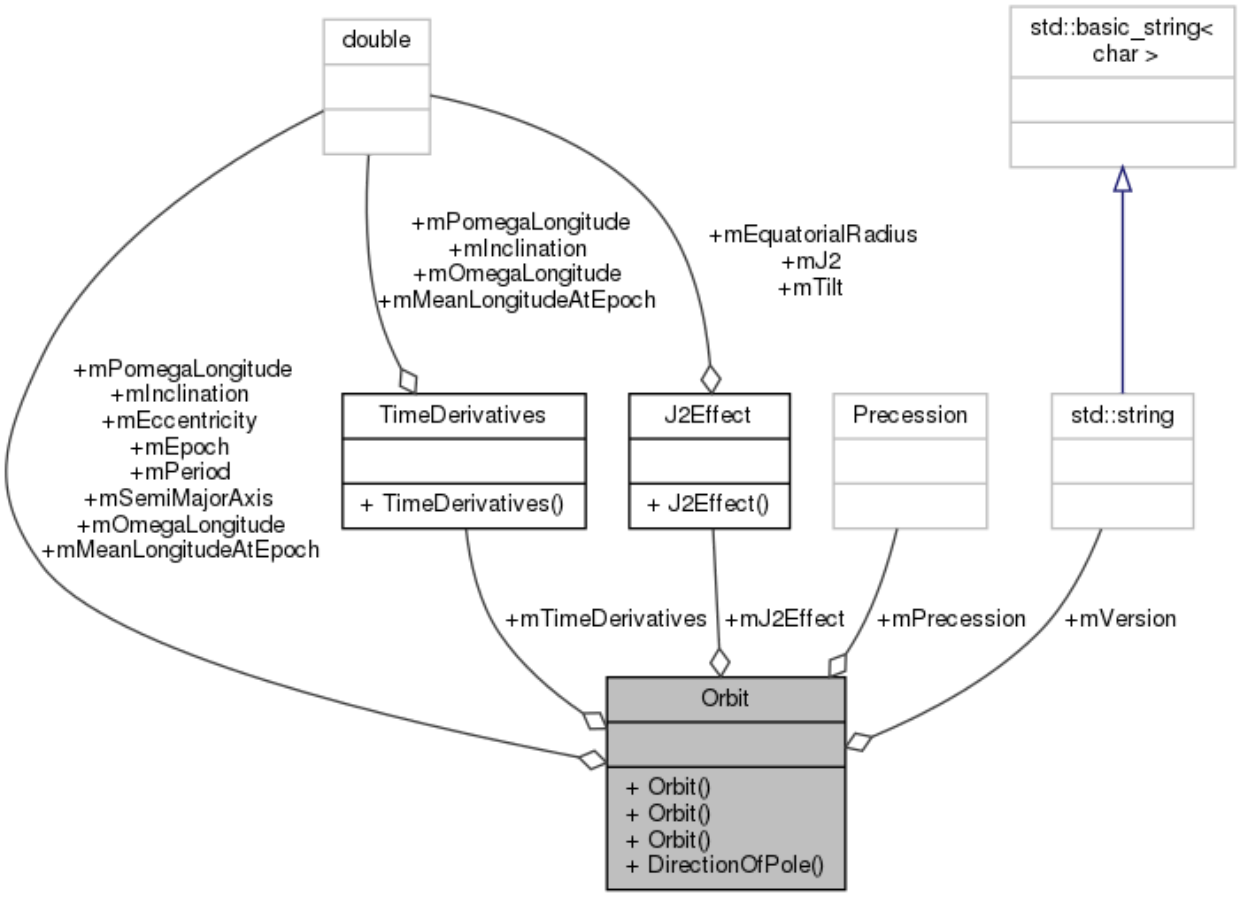

Figure 6: UML Diagram of Orbit class

The data structure is a class called Orbit. It's dynamic because it has multiple overloaded constructors that take in certain parameters and use them to calculate further parameters. Specifically, there is a default constructor taking in eight parameters, a constructor that takes in time derivatives and calculates J2 effects, and a third constructor that does the opposite.

As seen in the grey box in Figure 6, the public API of Orbit is quite simple. There are three constructors and a function that returns the direction of the pole. There are two data structures that account for precession: time derivatives and the J2 effect. Currently there is no use for precession, because the assumption is that the bodies are in a fixed orbit, though the functionality exists just to be future-proof.

Data

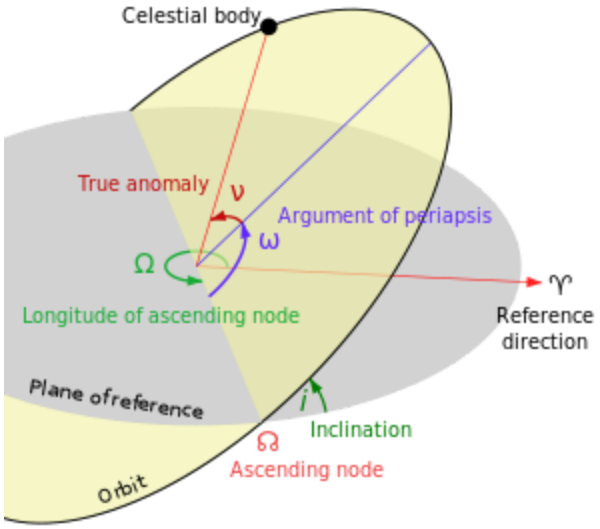

Figure 7: Keplerian orbital elements [1]

The Orbit class contains the Keplerian orbital elements, some of which can be seen in Figure 7:

1. The period of time (days)
2. The semi-major axis (km)
3. The eccentricity (dimensionless)
4. The inclination relative to J2000 equatorial plane (degrees)
5. The mean longitude at epoch (radians)
6. The longitude of ascending node (radians)
7. The longitude of periapsis (radians)
8. The epoch, or reference Julian dates (days)
9. An enumeration value representing what type of precession is being used
10. The J2, or the quadrupole moment of primary mass (dimensionless)
11. The tilt, or the angle between primary spin and orbit pole (radians)
12. The equatorial radius of primary (km)
13. The change of mean longitude at epoch (radians/day)
14. The change of Omega longitude (radians/day)
15. The change of Pomega longitude (radians/day)
16. The change of inclination (radians/day)
17. An optional descriptive string

[1] Source: https://en.wikipedia.org/wiki/Kepler_orbit

Everything past the epoch, element 9 onwards, deals with precession. The true anomaly, seen in Figure 7, is generated in the Kepler True Anomaly submodule. The Orbit class also contains some helper functionality, such as a print overload that allows the data members to be formatted and printed.

DirectionOfPole()
The sole function outside of the constructors is DirectionOfPole(). It reports the direction of the orbit pole in equatorial and ecliptic coordinates. The equatorial coordinates are generated through simple trigonometry and the ecliptic coordinates are generated through a custom, private Euler function.

Input:
- Right ascension angles
- Declination angles
- Whether or not to print out the results

Output:
- A pair of doubles representing the ecliptic coordinates
- (Equatorial coordinates are printed if desired)

4.2.2 Kepler True Anomaly (Zach)

The Kepler True Anomaly submodule solves Kepler's equation for an elliptical orbit, specifically to give the angular location along an elliptical orbit as a function of time. For consistency with archaic nomenclature, this angle is called the true anomaly.

Dependencies

- Math
- Types

Use cases

- The true anomaly is part of a Kepler orbit, but is generated in the forward model as opposed to the Orbit constructor.
- It is solely used by the forward model. The true anomaly is used as part of some equations to generate orbital offsets that would be observed for a set of dates.

Design

This submodule consists of a single function called keplerTrueAnomaly()

keplerTrueAnomaly()
Solves Kepler's equation for the true anomaly.

Input:

- Semi-major axis of the orbit
- Eccentricity of the orbit
- Vector of times in units of fractional period since periapsis passage
- An optional threshold for Newton's method

Output:

- The true anomaly in radians, which is the angle away from periapsis as seen from the barycenter
- Optionally, the distance from the barycenter

4.3 Shape

The Orbit module would be useless without something to attach it to. A pair of shapes and orbits is needed to simulate a binary asteroid system. The Shape module is responsible for reading in and generating these 3D shapes.

Dependencies

- None, though heavy use of the Standard Template Library

Use cases

- Used by the forward model to create a binary asteroid system (using a primary shape and secondary shape with mutual orbits)
- Manipulated in the ray-tracing module (need an object to ray-trace)

Design

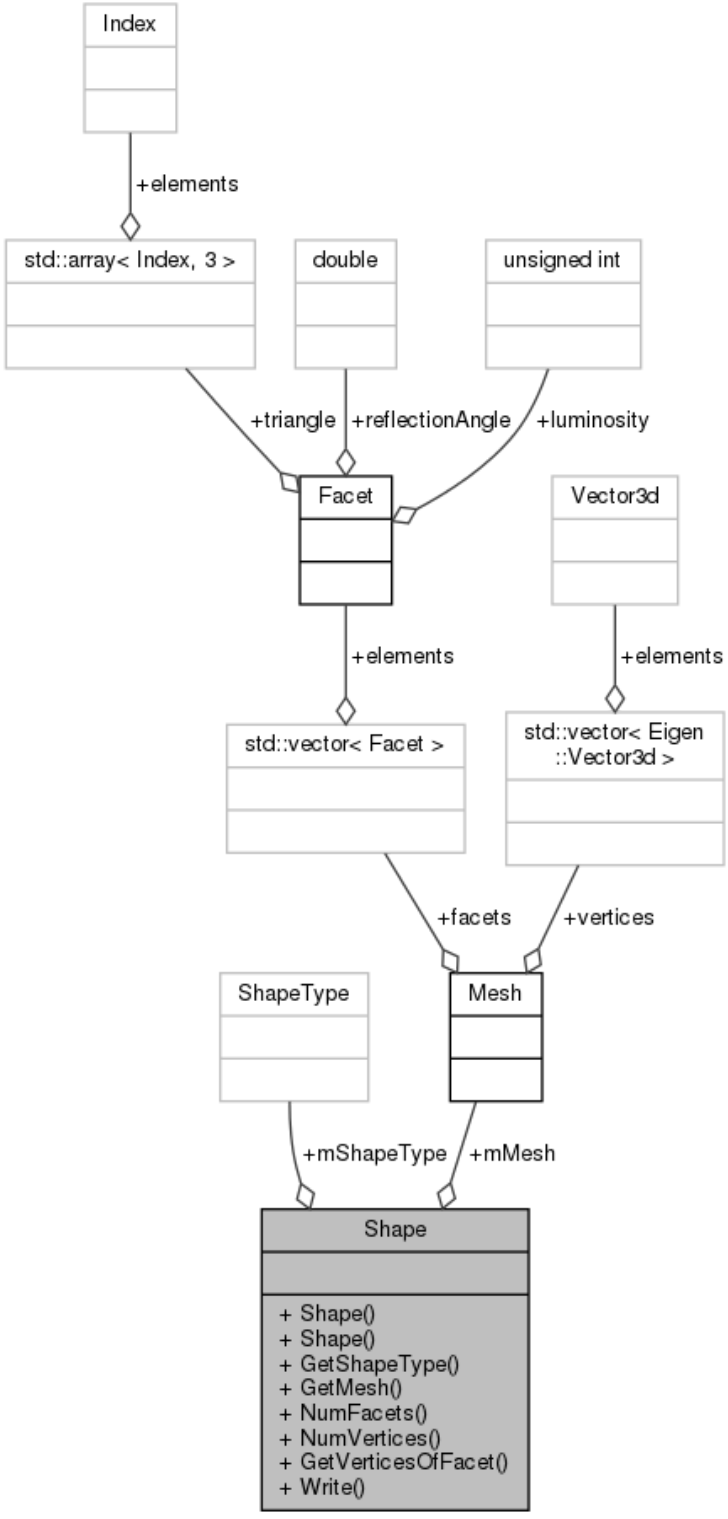

Figure 8: UML Diagram of Shape class

As seen in the grey box in Figure 8, the public API is verbose but the functionality itself is simple.

To create a shape, one can pass in a path to a Wavefront (.obj) file or provide the type of shape to generate along with some settings. There are also functions that provide convenient access to member variables. The beta prototype will include the ability to rotate the shape around an axis.

Overall, a shape is really just a mesh, and a mesh is really just a list of facets and vertices.

Data

A shape is a mesh, but how the data relate to one another is more detailed than that:

- Facets
    - A facet consists of a triangle and a reflection angle.
        - The triangle represents the indices of the three vertices and the reflection angle represents the angle at which a beam has reflected off of it.
    - Initially a facet will only contain the indices of the vertices.
    - Facet also extends a special class, hitable, in the ray-tracing module.
- Vertices
    - Vertices are much simpler in that they are just a vector of 3D vectors, which are vectors containing three values.
        - In this case, the values are doubles and represent the x, y, and z coordinates.
    - The vertices are created at shape generation time and can be rotated.
    - Note that it is important to separate facets and vertices for many reasons, but one of which is that when rotating vertices the facets should remain the same.
- Mesh
    - A mesh, or an indexed mesh, simply holds a list of facets and a corresponding list of vertices.
        - The *indexed* part simply means that the vertices have corresponding indices, which are the facets.

- Shape Type
    - A shape type defines the valid types of shapes that can be generated.
    - It is an enumeration, meaning each shape type is represented by an auto-incremented number, as opposed to some custom string.
        - Two types of shapes can be generated: an icosahedron and a triaxial ellipsoid.
        - If the shape is generated from a file, the shape type is “Custom”. This is mostly used in constructing the shape, but is kept as metadata.

Constructors

There are currently two constructors. One of them takes in a path to a Wavefront file and the other takes in a shape type and some settings. The settings are currently the number of subdivisions to perform and the initial radius or radii of the shape.

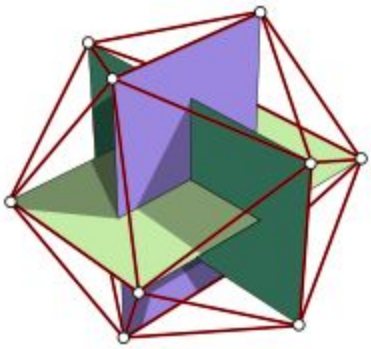

Figure 9: The starting state of an icosahedron [2]

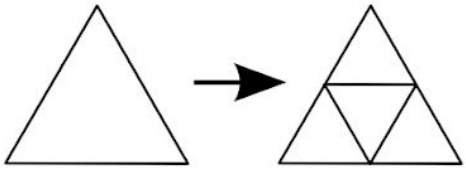

Figure 10: Subdividing a triangular facet [3]

A generated shape begins with hard-coded vertices and facets, such as the icosahedron in Figure 9. Once the constructor is called, the vertices are multiplied by the radius or radii and the facets are then divided into four new facets the number of times that the subdivisions input declares, as shown in Figure 10.

This is all that is needed to generate a shape.

[2] Source: https://en.wikipedia.org/wiki/Icosahedron
[3] Source: http://blog.andreaskahler.com/2009/06/creating-icosphere-mesh-in-code.html

GetShapeType()
This returns the string form of the enumeration value. For instance, if a shape was enumerated with the value *ShapeType::Icosahedron* then this function would return the string “Icosahedron”. This is simply potentially useful metadata and has various miscellaneous uses, such as creating a name for a Wavefront file.

GetMesh()
This simply returns a reference to the mesh data member. It is generally good practice to expose member variables through functions as opposed to directly accessing them, as it creates a more intuitive API. The mesh needs to be accessible to iterate through the vertices and facets in ray-tracing.

NumFacets()
This returns the number of facets by calculating the size of the GetMesh().facets variable. Is useful for iterating through facets in ray-tracing.

NumVertices()
This returns the number of vertices by calculating the size of the GetMesh().vertices variable. Is useful for iterating through vertices in ray-tracing.

GetVerticesOfFacet()
This takes in a constant reference to a facet and returns the three vertice values of it (this is of the type *TriangleVertices*). A facet is just three indices, so to get the actual values of the vertices one has to access the GetMesh().vertices variable with the GetMesh().facets.triangle variable. This function provides that convenience, since the logic is a bit verbose.

Write()
This is a debugging function that is publicly accessible. It saves the current shape to a Wavefront file so one can inspect the vertices and facets in a viewer like Blender, as seen in Figure 11. This has already been used to verify that icosahedrons are being generated properly, though it can also be used to get a visual sense of how high resolution a shape is.

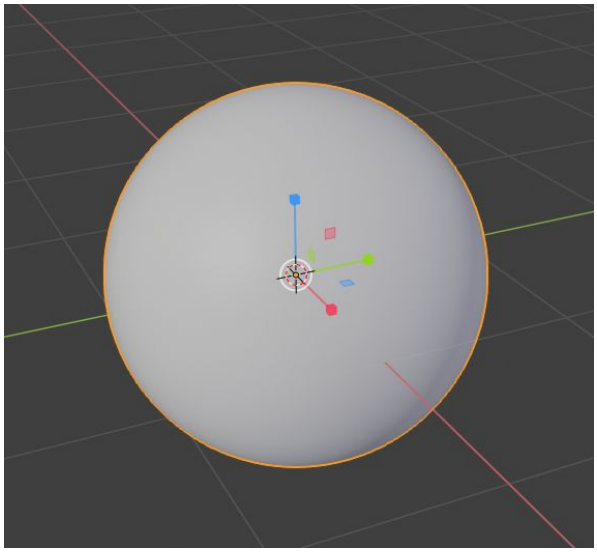

Figure 11: Viewing an icosahedron generated via the Shape module in Blender. 8 subdivisions, ~1.3 million facets.

4.4 File Input

This module handles and reads in necessary parameters from the provided files, then returns the parsed results. Some of these parameters are converted inside the submodule before being returned.

Currently this module is only composed of one submodule, Ephemeris. Other submodules to handle other file types can be created and added.

4.4.1 Ephemeris (Matt)

This submodule contains a single function that takes in a '.txt' file representing an Ephemeris table.

Dependencies

- Types
- View

Use cases

- The Ephemeris function is used by the forward model to find the location of the primary object at the given observation times.

Design

The Ephemeris submodule is just a single function that takes in a path to a text file and returns a View object.

Input:

The Ephemeris file will provide a table with the following columns with "$$SOE" indicating the start of the table and "$$EOE" indicating the end of the table.

- Date__(UT)__HR:MN
    - Contains the date in Universal Time (UT)
    - Example: [`2019-Jan-28 00:00`]
    - Above the data is formatted with the year followed by the abbreviated month, day and time
    - Units: [Year-Month-Day Time]

- R.A._(ICRF/J2000.0)_DEC
    - Contains right ascension and declination of target object
    - Example: [`08 22 05.45 +27 01 47.9`]
    - The numbers before the '+' belong to the right ascension and the numbers after the "+" belong to the declination.
    - Units:
        - Right Ascension in hours-minutes-seconds of time (HH MM SS.ff)
        - Declination in angular degrees-minutes-seconds (DD MM SS.f)
- r (distance from the sun in AU)
    - Contains the heliocentric range of the target object
- Delta (in AU)
    - Contains delta of target object
- S-T-O (in degrees)
    - Contains the interior vertex angle of the Sun-Target-Observer vector.

Output:

The output of this module is a View object containing the following parameters for each time value in the table provided in the Ephemeris file.

- Right Ascension (in hours)
- Declination (in degrees)
- Delta (in AU)

4.5 Ray Tracer

The purpose of the ray tracer is to calculate the amount of reflected light which can be used to create a light curve. The ray tracer can also generate a rendered set of images corresponding to the light curve. The images are saved to file as a .ppm. The methodology for ray tracing implemented is image centric. The classes and methods are standard for ray tracing. For more information and an overview about this ray tracing method see the website scratchapixel.com[4].

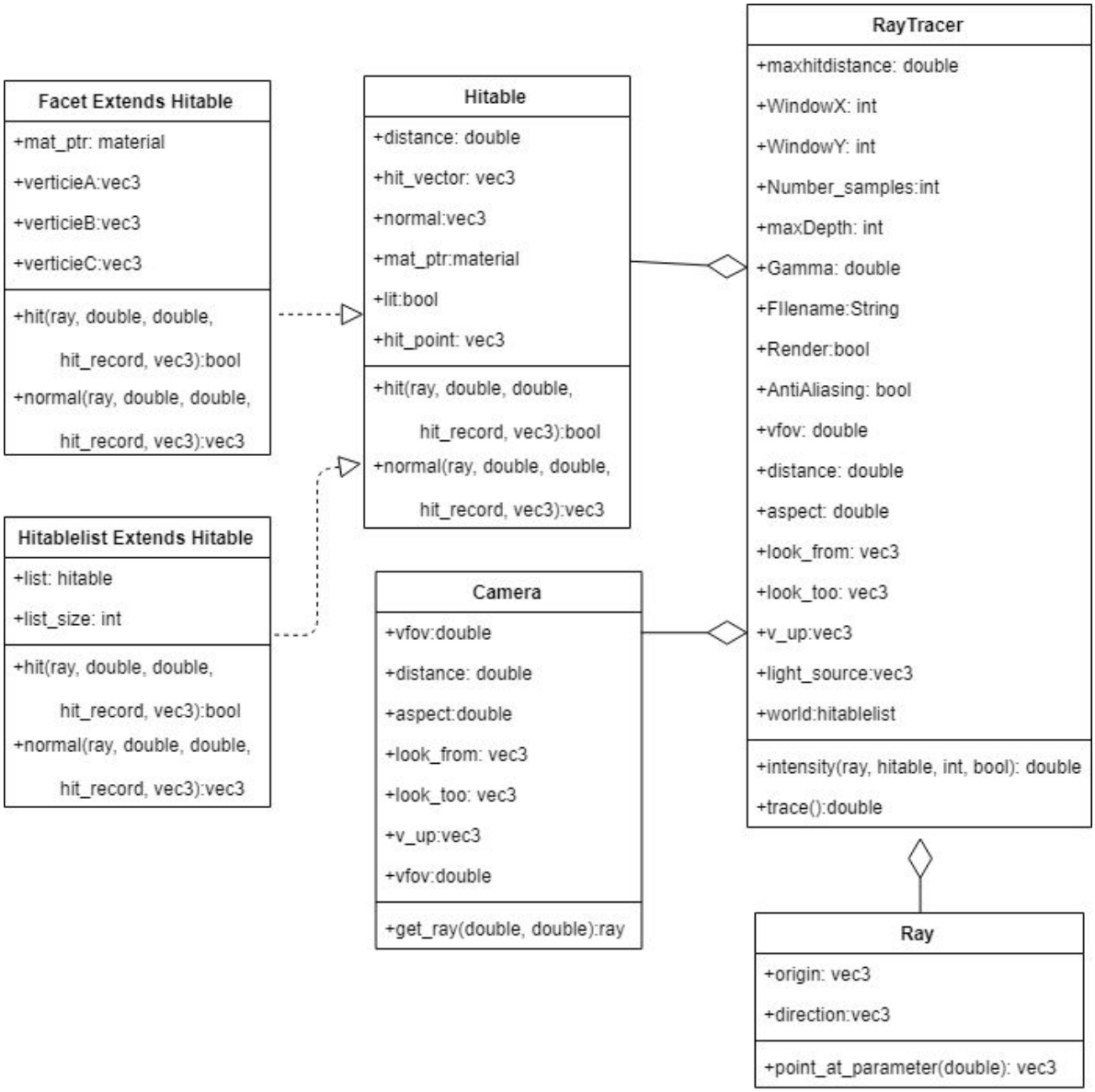

Figure 12: UML Diagram of Ray Tracing module and sub modules.

[4] http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview

4.5.1 Tracer

The tracer holds most of the logic for rendering and ray tracing. It uses other classes to set up the problem then iterates through all of the pixels to generate an image and calculate the luminosity of the scene. The logic of the tracer is:

1. Generate and set up a file (if rendering is on)
2. Get the list of hitable objects to be rendered
3. Create a camera
4. Iterate through each pixel for the following:
    a. Get the pixel location
    b. Create a ray from the camera to the pixel
    c. Get the color/luminosity of the pixel
    d. Add the luminosity to a total
    e. Translate the rays to rgb
    f. Correct the intensity for gamma levels
    g. If anti-aliasing is on, repeat step 4 for multi sampling
5. If rendering then right to file otherwise just return the brightness

While most of the logic is contained in the tracer, the light intensity function is also important. The light intensity function determines the wavelength and strength of a pixel. The light intensity function takes in a ray, all the hitable objects and a max depth of recursion. The function has the following steps:

1. Create a hit record
2. If the ray hits an object in the scene then:
    a. Create a new ray from the hit location
    b. Check if it is possible for that ray to reach the light source without hitting anything
    c. If the ray reaches the light source then return the brightness of the hit
    d. Cast out rays according to hapke diffusion parameters
    e. Check if those rays can reach the light source using the light intensity function
    f. Repeat steps d and e up to the maximum recursion depth assigned
    g. Sum up all the intensity strengths (less than one total)
    h. Return the total brightness of the pixel
3. With no hit, return a zero brightness value

The tracer function and light intensity function are used in the main ray tracing file to drive the entire ray tracing process. The ray tracing main file relies on other submodules and classes.

Dependencies

- Ray
- Shape
- Hitable
- Hit Record
- Hapke
- Camera

Use cases

- Ray tracing takes in shapes and locations and returns a single luminosity value
- Ray tracing renders the scene and returns luminosity

Design

The ray tracing module consists of a number of files and classes. Below is an overview of what those classes are and how they are used.

Hit record Struct

Contains

- Distance the ray traveled
- The normal of the hit object
- The material of the object (normally hapke)
- The location of the hit
- The vector from the ray start to the hit point

Description

The hit record is used to record and pass back information when a ray intersects a hitable object. All hit functions generate a hit record. If the object is not hit then the hit record is null. The distance the ray traveled is important for the distance squared reduction in brightness that light suffers. The material of the object is used to find diffusion rays and to calculate the amount of light returned to the observer. The location of the hit is used to help calculate the normal and the start of any other rays that the hit generates. The vector from the ray start to the hit point is kept so that its strength can be modified for luminosity values.

Ray Class

Contains

- Origin vector
- Direction Vector
- Strength value
- Boolean for light value
- Wavelength value

Description

The ray class represents the physical rays that are being traced. Rays are constructed with an origin point and a direction. The origin vector is comprised of x,y, and z coordinates. The direction vector is also a x,y,z coordinate vector, but it is also a unit vector just representing the direction of the ray. When a ray intersects an object, a new ray is generated. Rays are either lit or unlit. A lit ray is one that has either a direct path to the light source or an indirect path that is within the recursive depth limit of reflections. The strength of a ray is a normalized value. A strength of one would be the strength of light at the light source (given a non dimmed light source). The wavelength of the light is used for representing the color of the light. The wavelength will be modified by hapke parameters and the original starting wavelength of the light.

Camera Class

Contains

- Location of the observer
- Direction of view
- Field of view
- Aspect ratio
- Pixel resolution
- Distance to center of scene

Description

The camera class is used to define the observer. The camera class hold the location of the observer in a x,y, and z vector. The direction of view is a unit vector. THe field of view is an angle in degrees. The angle is the measurement from the camera to the top of the scene and to the bottom of the scene. For our uses this will be an extremely small angle to account for the distances involved. The aspect ratio is the ratio of the width of the scene divided by the height of the scene. This aspect ratio is also defined by the number of pixels chosen. The distance of the scene is the distance from the camera to the center of the scene frame. The camera will always be at the center of the scene along two axis. This will lead to a centered picture every time. The pixel resolution is selected in the

camera class. The resolution directly affects the performance of the software. An original ray is traced from the camera origin to each pixel in the scene. An increase in the number of pixels linearly increases the run time.

Hitable abstract class

Contains

- Hit function
- Normal function

Description

A hitable class is one that has a hit function and a normal function. The facet class is a hitable class, as well as anything else that will be rendered by ray tracing. The hit function takes in a ray and an object and checks if the ray hits that object. If the ray does hit the object, then the normal function can be used to figure out the angles involved in the collison. The angles are then used to determine the strength of the reflection using the hapke model functions. The hit and normal functions are dependent on the shape being checked. For example, a facet will have different hit and normal functions than a sphere. While the facet class is a hitable class, the shape class, which is comprised of facets will not be hitable directly(unless we add additional logic to reduce the number of searches for facets based on the shape orientation etc.).

Hitablelist class

Contains

- An array of hitable objects
- A hit function

Description

The hitable list class implements the hitable abstract class. When calling hit on a hitable list, the hit function iterates through each object in the list and checks to see which is the closest object that is hit. It then returns the closest object and the distance as part of a hit record.

4.5.2 Hapke Model (Zach)

The Hapke Model is a namespace that helps compute the luminosity of the pixel or facet that a beam hit. It does this by computing Hapke's bidirectional reflectance model. It takes in the angle that the beam hit the pixel or facet, along with a variety of metadata about the single asteroid, and returns the effects of macroscopic roughness of said pixel or facet. This is a complicated series of equations that takes into account many aspects of the asteroid, such as its surface roughness.

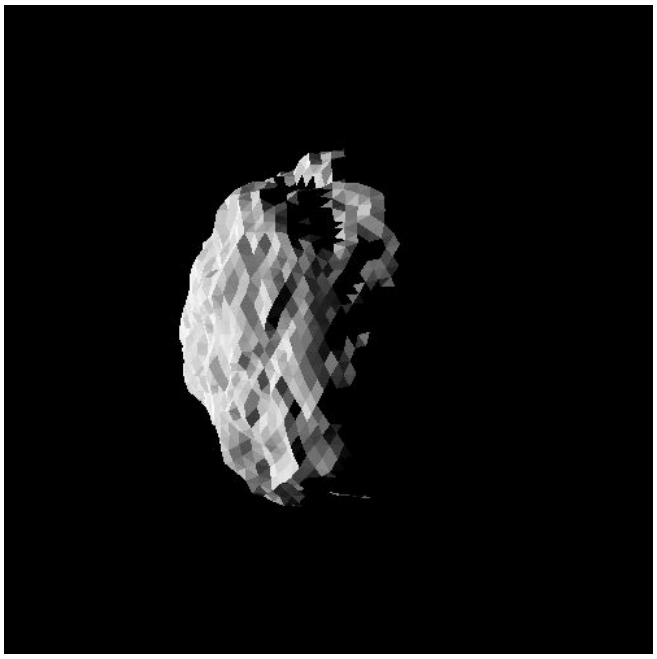

Figure 13: Rendering of illuminated facets using Hapke Model

After this submodule is used in ray-tracing, the end-result is a luminosity value for where each beam hit, as seen in Figure 13. In this case, ray-tracing was done by facet, so each facet has a luminosity value.

Dependencies

- Math
- Types

Use cases

- The ray-tracing module uses it after a beam hits a pixel to receive a modifier of the luminosity value, when is then used to compute the actual luminosity

Design

The Hapke submodule itself has quite a few functions, though most of them just help the main function: HapkeModel::bidirectional()

bidirectional()
This is the main function of the Hapke submodule and is the only one intended to be directly called by other modules.

It is equation 12.55 on page 346 in Hapke's book, "Theory of Reflectance and Emittance Spectroscopy" (1993 edition). It is also on page 323 of the 2012 edition. It calculates the

effects of macroscopic roughness on light scattered by a surface having an arbitrary diffuse-reflectance function, and then approximates the analytic bidirectional-reflectance.

Input:

- Single scattering albedo
- Cosine of the emission angle
- Cosine of the incidence angle
- Phase angle (radians)
- Compaction parameter
- Parameters of the single particle phase function
- Amplitude of opposition surge
- Surface roughness value (radians)
- A flag indicating which H-function to use
- The single particle phase function to use (defaults to a constant)
- A pedantic flag which, if set, chokes on single scattering albedo > 1, prints various additional warnings, etc.

Output:

- The bidirectional reflectance of each index in the input values

This is perhaps one of the most complicated functions of the code base and has many options, such as custom single particle phase functions and the option of using the 1993 H-Function or the 1981 version. The custom phase functions are implemented directly in the namespace. There are currently three Henyey-Greenstein functions taking in one, two, and three parameters respectively.

4.6 Forward Model

The forward model contains the entirety of the functionality of the software. It uses all of the modules to produce a light curve model and to render a s et of images. In Figure 14 the process flow can be seen, as well as the light curve generation loop.

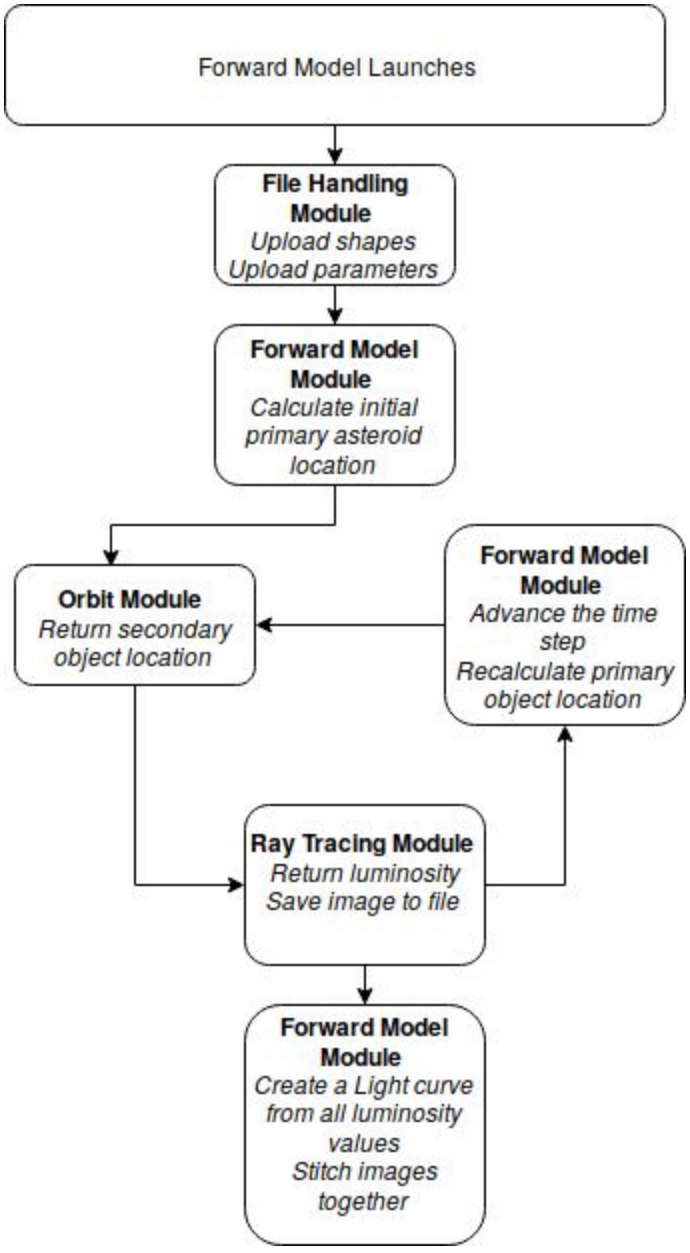

Figure 14: Forward Model Flow Chart

Dependencies

- All other modules

Use cases

- Takes in initial parameters and shapes and creates a light curve model of the system
- Renders a set of images of a system of objects over time and their illumination

Design

The forward model drives the functionality of the entire program. It ties together all of the modules and includes functionality to allow it to interface with IDL. The forward model has the following steps:

1. Read in information using the file handling module
    a. Use the information to select a use case
    b. Based on the use case, either:
        i. Upload a shape using file handling
        ii. Generate a standard shape using the shape module
2. Use the inputs to determine the orbital and positional properties of the primary object
3. Use the orbit module to determine the orbital properties of non-primary objects relative to the primary object
4. Begin the light curve generation loop
    a. Calculate the position of all objects for the current timestamp
    b. Pass the objects to the ray tracing module
    c. Save the rendered image and luminosity value from ray tracing
    d. Advance the time (time evolve the vertices)
    e. Repeat step 3 and 4
5. Create a light curve using the luminosity values and time stamps
6. If rendering was chosen, either
    a. Choose a single image
    b. Create a gif from the set of images
7. Return the matrix of brightness values that create the light curve to the calling IDL function

This design of 6 conceptual modules allows pieces of functionality to be individually developed and then relatively easily integrated. The modules, although conceptually simple, contain many details that require time and coordination to develop. For this reason, Paired Planet has devised an implementation plan.

5. Implementation Plan (Matt)

For a successful implementation of our solution, active scheduling and task assignment between the 3 group members is critical and necessary. A well documented implementation plan provides both the group members and the clients with dates and percentages relevant to tasks being completed for the project.

As the 2019 semester started, Paired Planet Technologies immediately started discussing the development plan for the upcoming weeks and planning out the major deadlines of the semester. The Gantt chart in Figure 15, found in the appendix, represents the planning that has been completed so far. The documented is updated weekly with an updated progress bar, updated percentages, and updated dates as tasks are changed and completed.

The Gantt chart utilizes colored bars to indicate which tasks have been assigned to which team member. Blue is assigned to Matt, green is assigned to Brian, purple is assigned to Zach, yellow is assigned to the whole team and gray means it has not been assigned yet.

To begin the development process, the team assigned the remaining unimplemented features to each team member based on their strengths as a developer. Zach is assigned the Shape feature as Zach has the most experience with the code base and understands how Shape must interact with the rest of the codebase. Brian is assigned the Ray Tracing and Rendering feature as Brian is chosen to be the team's ray tracing expert and has the most experience on the team with ray tracing. Matt is assigned the File Handling feature as Matt expressed interested in the feature and wanted more file handling experience. As mentioned previously, each task is being assigned to a team member based on whom was best qualified for the task.

For testing in the project, unit tests are being written to verify the functionality of submodules. There is a dedicated time slot, after all of the core features have been implemented, to enhance the initial unit tests to better prepare for upcoming optimizations. During this time slot the team will also develop the C API structure to allow IDL to utilize the functionality.

As seen in the Gantt chart, the major phase titled "Alpha Demo Prototype" is a large upcoming programming task and once all of the major features are implemented, this large task is to be broken down and assigned to team members.

After the Alpha Prototype Demo has been completed, the primary focus is optimization and implementing potential stretch goals. Optimization involves low-level changes and most importantly parallelization. Stretch goals for the project involve addressing more use cases and adding quality of life features.

With a well developed and updated implementation plan, both team members and clients can determine what any given team member is working on for the week and what tasks have been completed. This implementation plan provides the team with a constant reminder of upcoming deadlines and upcoming tasks. The team feels confident that the current implementation plan serves as a good representation of the team's path to success in the entire project.

6. Conclusion

Space exploration has always enthralled humanity. Every year, billions of dollars are spent trying to understand what is in the Solar system. Binary asteroid systems have always been challenging to understand due to their small size. The clients, Dr. Audrey Thirouin and Dr. Will Grundy, work at Lowell Observatory on understanding binary systems in the Kuiper Belt. They accomplish this by using software that models binary systems.

One challenge is that these systems are so far away that only a single point source can be observed. The required calculations to account for this are relatively complex and the clients have only had time to develop a partial solution for the modelling. This partial solution is slow and fragmented, meaning the functionality is spread apart and scripts have to be written to utilize the code. The clients would like an API that is significantly faster and integrated, meaning they can receive the model in a single function call in a code base that follows consistent coding and naming practices. The best solution is a modularly designed, high-performance C++ code base with a single exposed function that returns the desired results.

Extensive software design has been done for this document, detailing the functionality of six modules. There is one main module, the forward model, which makes use of the five other loosely coupled modules that provide core functionality. This design provides greater support for upcoming features and creates a more approachable code base for future developers. This important for this project since more functionality is planned in the future. The explanations of the modules in this document also set up an important foundation for the upcoming user-guide.

The formalization of a conceptual design instills confidence in both the developers and the clients. It provides a blueprint for how to implement a project and hence makes meeting the requirements more straightforward. It also mitigates the risk of running into issues during development. With the extensive planning there are no foreseeable roadblocks and the independently testable modules ensure a robust solution. Paired Planet Technologies is excited to implement this design and create an efficient, intuitive solution that exceeds the client's needs.

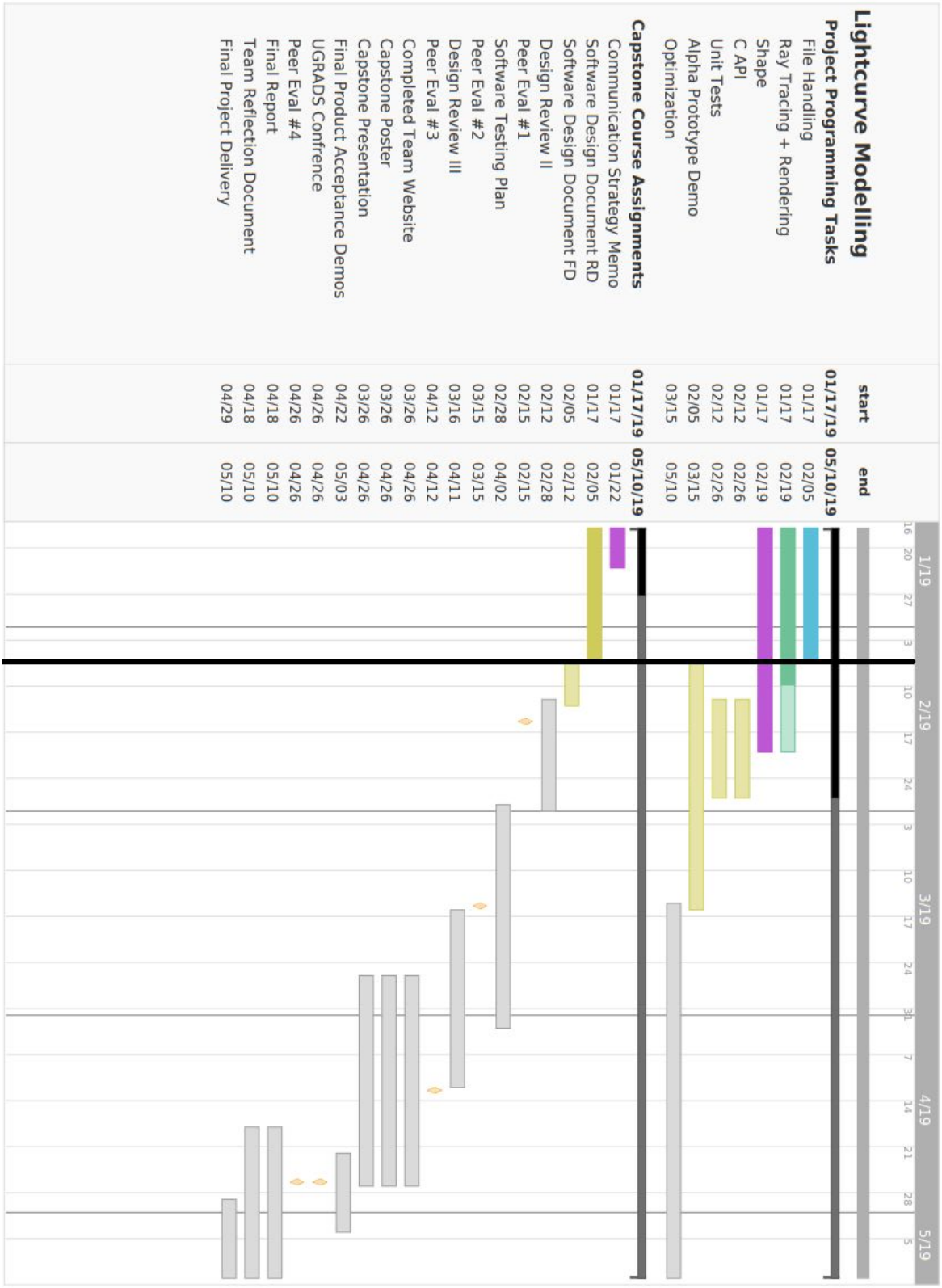

Figure 15: Gantt Chart as of 2/5/2019