Eartharium User Guide

Table of Contents

[Introduction 1](#_Toc132185348)

[The Basic Eartharium Concepts 1](#_Toc132185349)

[Simple Example (Hello Earth) 1](#_Toc132185350)

[Astronomy object 3](#_Toc132185351)

[Time in Astronomy 3](#_Toc132185352)

[Coordinates in Astronomy 5](#_Toc132185353)

[Angles in Astronomy 7](#_Toc132185354)

[Distances in Astronomy 7](#_Toc132185355)

[Astronomy() constructor 7](#_Toc132185356)

[Astronomy Stellar Object Database 7](#_Toc132185357)

[Astronomy::stellarobjects 8](#_Toc132185358)

[Astronomy::stellarobject\_xrefs 9](#_Toc132185359)

[EDateTime object 10](#_Toc132185360)

[Static Functions 10](#_Toc132185361)

[EDateTime::normalizeDateTime() 10](#_Toc132185362)

[EDateTime::myDivQuotient() 10](#_Toc132185363)

[EDateTime::myDivRemainder() 11](#_Toc132185364)

[EDateTime::getDateTime2JD\_UTC() 11](#_Toc132185365)

[EDateTime::getDateTime2JD\_TT() 11](#_Toc132185366)

[EDateTime::getUnixTime2JD\_UTC() 11](#_Toc132185367)

[EDateTime::getUnixTime2JD\_TT() 11](#_Toc132185368)

[EDateTime:: getDateTime2UnixTime() 11](#_Toc132185369)

[Appendix A – History of Time and Date 12](#_Toc132185370)

# Introduction

I started developing Eartharium to demonstrate various shortfalls of Flat Earth claims. It has evolved to a highly accurate astronomical modelling system which can equally well be used for general astronomy education.

Eartharium is written in C++ (currently it requires version 2017) using OpenGL to render most simulations in real time. The animation capabilities are not well developed, but supports writing individual frames to PNG files, which can be loaded into video editing software such as DaVinci Resolve for further editing.

# The Basic Eartharium Concepts v2

The main goal of Eartharium is to render high quality animations of a wide range of astronomical and geographic scenarios. To accomplish this, Eartharium has several sub-components that all work together to provide an API that is as capable and flexible as possible (that is, as possible as my programming skills and imagination allows; I am sure there is plenty of room for improvement.)

## Rendering System

Eartharium contains a simple compositing rendering system based on different types of RenderLayer. The main one is RenderLayer3D, which is responsible for rendering a given 3D Scene into one frame of a larger animation. By default, a RenderLayer will take up the full frame, but it is possible to specify a smaller rectangle within the full-sized frame.

For interactive demonstrations, it can be practical to build a small GUI with sliders and buttons etc, a RenderLayerGUI provides functionality for that via the open-source library ImGUI. Typically, the RenderLayerGUI will take up the whole frame and overlay any lower layers, but it is possible to confine it to a smaller rectangle of the full frame.

There is a modest data plotting RenderLayer available called RenderLayerPlot, which is a thin wrapper for ImPlot (which depends on ImGUI).

It can also be helpful to overlay simple text objects in 2D, for example to indicate the current time and date, or to track some other parameter. For that Eartharium offers a RenderLayerText (which internally also uses ImGUI but is separate from the RenderLayerGUI layer.) RenderLayers are drawn to screen from the bottom up, with upper layers overwriting lower layers if they overlap. It is possible to have more than one of each type of RenderLayer.

Currently Eartharium will go through all RenderLayer3D layers and render them in the order they were defined, then the same for RenderLayerPlot layers, then RenderLayerText layers, and finally the RenderLayerGUI layers.

The render() function in the Application class handles the overall rendering by calling the specific render() functions of the various layers. After each frame has been rendered, Application will swap the on-screen frame buffers to display the newly rendered frame, and optionally write it to a PNG file. When writing to file, it will number the frames for assembly into video files by some other program (I recommend DaVinci Resolve which, while not open source, is free software.)

## Graphical Objects

All graphical objects are inserted into a Scene, which can then be rendered using a RenderLayer3D. There are a few complex objects available which will do most of the work for you. For example, there is an Earth object which can change geographical geometry as needed, and show things like the current direction to the Sun, the GP of the Moon etc.

For things that are specific to a particular observer location, one or more Location objects can be associated with the Earth object. Each Location can then be configured to show directions to the planets, Sun and Moon, the daily sky path of any of those, etc. It is also possible to attach a SkySphere to either Earth itself (to illustrate the celestial sphere) or a Location (to show the local sky at that location.)

When working with many Locations (e.g., when loading community survey data from several different surveys) it is possible to create LocGroups where the properties of several Locations can be configured simultaneously. This functionality is currently somewhat limited.

Another useful object is SolarSystem, which displays each planet as a small dot, and can show both heliocentric and geocentric planetary orbits. This is also work in progress in terms of fancy features, but it does provide full high precision orbital paths and positions from VSOP87.

A separate object called SubPointSolver is available for celestial navigation demonstrations. It currently only supports stars, but Sun, Moon and planets should be relatively easy to add (SubPointSolver only takes SubStellarPoints now.)

A lot of the features of each of the above objects are basic dots, arrows, lines, and paths. Those are all implemented as graphics primitives, which can also be used individually. Since all objects must be part of a Scene for RenderLayer3D to render them, I have chosen to make Scene a factory for new objects. Thus, when you want to have a new Arrow, simply ask Scene to allocate one.

All Scene objects will be inheriting from SceneObject, but the implementation of this has only just begun. Parenting and hierarchical inheritance is possible but not yet in use. I am considering having multiple inheritance structures, so that an Arrow can inherit geometric details from Earth, but also astronomical details from a Sun object. However, I may never get around to writing the code.

## Astronomical Calculations

Eartharium includes P Naughter’s AA+ library which is based on the book Astronomical Algorithms written by Jean Meeus. In AA+ some angles are passed or returned in degrees, some in arcminutes or arcseconds, and some in hour angles. In Eartharium I have chosen to always use degrees (or radians if desired) for ease of

## Scripting

While creating animations, it is burdensome to keep recompiling and running Eartharium. To alleviate this burden, Eartharium can be scripted via Python. The scripting interface is work in progress and is far from complete. Currently the script name is hard-coded to “hello.py”. At runtime, Eartharium will present as a python module named “eartharium” which is imported in the python script.

To gain access to the Application object, use the getApplication() function. With that, other objects can be instantiated just like when using C++. Here is the hello earth example in python:

from eartharium import \*

app = getApplication()

astro = app.newAstronomy()

scene = app.newScene()

cam = scene.w\_camera # Pick up default camera

app.currentCam = cam

layer = app.newLayer3D(vpx1 = 0.0, vpy1 = 0.0, vpx2 = 1.0, vpy2 = 1.0, scene = scene, astro = astro, cam = cam)

earth = scene.newEarth("AENS", 180, 90)

app.currentEarth = earth

earth.w\_sinsol = True

There are no current plans to offer scripting access to the AA+ library directly, however the Astronomy class provides access to many of the commonly used functions directly.

See the Python Scripting chapter for more details on using scripts and which functions are available to scripts.

# The Basic Eartharium Concepts

The main idea behind Eartharium is to construct and animate various objects, while having them align with different astronomical calculations. To achieve this Eartharium inserts all renderable objects into a Scene, which has a Camera that can view the objects. The Scene also has an Astronomy component which tracks time and supplies astronomical data such as positions and directions.

In addition to rendering 3D objects, Eartharium can also plot graphs and overlay text with dates and times or other output. This is done via render layers, which composite to the final image. For example, there is a RenderLayer3D which defines where on the screen the render of a given Scene ends up. It is possible to place this on the top ¾ of the frame, while having a RenderLayerPlot on the bottom ¼ of the frame.

It is possible to have more than one Scene and display those using more than one RenderLayer3D. That would allow displaying Earth full screen, while having a small rectangle insert with the solar system for context (like a Picture in Picture feature). Scenes can also have more than one camera, so two RenderLayer3Ds can display the same Scene from two viewpoints. The two Scenes would then share the same Astronomy instance, and astronomical calculations would only be performed once.

Eartharium has an Application object that keeps track of various interconnections between various objects and the low level system dependent code (such as keyboard and mouse input, compositing the RenderLayers, etc.) Application also takes care of allocating RenderLayer, Scene and Astronomy objects, as well as writing frames to files.

## Simple Example (Hello Earth)

Consider setting up a simple animation showing a globe Earth with day and night in real time. These are the required steps:

Application app = Application();

Astronomy\* astro = app.newAstronomy();

Scene\* scene = app.newScene();

Camera\* cam = scene->w\_camera;

app.currentCam = cam;

RenderLayer3D\* layer = app.newLayer3D(0.0f, 0.0f, 1.0f, 1.0f, scene, astro, cam);

Earth\* earth = scene->newEarth("NSAE", 180, 90);

app.currentEarth = earth;

earth->w\_sinsol = true;

The rendering chain that renders and composits each RenderLayer is called via Application::render(). Additionally, Application::update() will poll for user interactions and react to most of them automatically. One exception is when the window close button is clicked (or the user presses the Esc key). This allows clean-up before exiting. The easiest is to use a while loop to continuously update and render the Scene:

App.anim = true;

while (!glfwWindowShouldClose(app.window))

{

app.update();

if (app.anim) {

astro->setTimeNow();

}

app.render();

}

Here I specifically call app.update() in the loop so the while condition can become false as a response to user input. But app.render() actually calls update() too, so there is no real need.

Instead of calling astro->setTimeNow(), one can use astro->addTime(days, hours, minutes, seconds) to progress at a custom interval. When astro was created, it automatically set the time to the current system time (which can be done manually with astro->setTimeNow()). If a different starting time is desired, use astro->setTime(year, month, day, hour, minute, second)

While running the above, app.update() will respond to key presses from the user. Here are a few that might be fun to play with:

W, A, S, D = Orbit the camera around the Earth. The app.currentCam setting has allowed Application to know which camera to update when pressing the keys

Q, E = Zooms out and in respectively.

N, M = Will morph between a spherical and a flat Earth. The “NSAE” supplied when creating the Earth object specifies NS = Normal Sphere and AE = Azimuthal Equidistant geometries respectively. Other options include RC = Right Cylinder, ER = EquiRectangular. It is the “app.currentEarth = earth;” line that tells Application which Earth object to morph when pressing the keys.

F = toggle between full screen and windowed mode.

SPACE = Start / Stop animation. This toggles the app.anim Boolean, which is tested in the while loop.

Of course, the Earth object has many features. Try the following:

earth->addGrid();

earth->flatsunheight = 0.0f;

earth->addSubsolarPoint();

When inserted before the while loop (but after Earth has been created) these add latitude and longitude grids, and a small Sun at its GP. Notice how these objects are automatically updated as time passes, or Earth is morphed; very convenient!

# Astronomy class

The Astronomy class is responsible for providing accurate astronomy calculations according to the current time set. It is based on the AA+ astronomical library from P Naughter, which implements the algorithms discussed in the book Astronomical Algorithms written by Jean Meeus (second edition published in 1992). The book offers truncated VSOP87 calculations for the planets (including Pluto), but the AA+ library additionally offers access to the full VSOP87 tables, which offer higher precision at the expense of longer calculations.

The AA+ interface uses astronomically correct units, for example Right Ascensions are given in hour angles, whereas Declinations are in degrees. The constant conversions between radians and human readable units started to annoy me, so I have implemented several of the functions directly to simplify things. All function calls to Astronomy functions take degrees by default, and radians as an option. This is accomplished by using a Boolean flag.

Additionally, most functions support a custom dynamic time in JD, and default to current time if omitted. For example, the signature for the function returning the true obliquity of the ecliptic is as follows: double TrueObliquityOfEcliptic(double jd\_tt = NO\_DOUBLE, rad = false). If it is called without arguments jd\_tt is taken from current time and the result is returned in degrees. To get the result for current time in radians, use TrueObliquityOfEcliptic(NO\_DOUBLE, true). The NO\_DOUBLE macro is simply set to the highest value that can be represented by a double float, far outside the reasonable range. This mechanism is used by many other functions too.

Internally Astronomy uses my own EDateTime class to keep current time, rather than the CAADate class supplied by AA+. This is because AA+ doesn’t fully support proleptic Gregorian dates (e.g. dates before the Papal Reform that changes calendaring from Julian calendar to Gregorian.) The following section will discuss different aspects of time keeping in further detail.

## Time in Astronomy

Following common practice in astronomy, the Astronomy object internally measures time in “Julian Date” format. This is a contiguous count of days since noon on 24th of November 4714 BCE. I don’t know why this starting date was selected. In early astronomy meridian transits were often considered, which means astronomers traditionally measured angles from South rather than from North like navigators do. The Sun transits the meridian at noon, this is probably why the time scale starts at noon rather than midnight.

The CAADate module in AA+ contains functions for conversion between different calendar and time systems, but it does not support proleptic Gregorian dates. Proleptic means that the calendar is extended back in time to way before it was invented. So I wrote the EDateTime class to fully support proleptic Gregorian dates (which is simpler than using the Julian calendar for dates before 1582 like CAADate does.) If conversions between different calendars are needed, the CAADate library is of course still available.

Internally Astronomy uses EDateTime to represent moments in time, as far back and forward as the algorithms are valid. How far that is varies by algorithm and, when known, is noted in the documentation for the specific algorithm. A general rule of thumb is accuracy to an arc second for 2000 years either side of year 2000. For more information on EDateTime, see the relevant chapter.

A Julian Date (abbreviated JD in the following) is not closely related to the Julian Calendar. It is simply a fractional count of days since the above-mentioned point in time. However, JDs come in two (more, but two are important) flavours in Astronomy / EDateTime; JD\_TT and JD\_UTC.

In the UTC time scale a second is defined by atomic clocks, and we consider a day (say, from midnight to midnight) to be exactly 86400 seconds. There is another time scale called UT1, which precisely measures out a day as the period it takes for distant star to return to the exact same position in the sky, which is one rotation of Earth. Since there are tiny variations in the rotation time of Earth (due to unpredictable phenomena like earthquakes and large weather systems etc), UT1 and UTC are not exactly in sync.

In UT1 a second is 1/86400 of an Earth rotation on a given day. Since UTC is a civil time scale, it is desirable to stay as close to UT1 as feasible while still maintaining a fixed length second (via atomic clocks). The UTC standard specifies that UTC must stay within 0.9 seconds of UT1, by the use of leap seconds. Currently (April 2023) there have been a total of 37 leap seconds, all positive. There is an organization called IERS (International Earth Rotation Service) which tracks the UT1 to UTC discrepancy very accurately and makes recommendations for when to insert a leap second. In the past they have done so when the difference was greater than 0.6 seconds and trending towards increasing difference.

Because it is impossible to predict the occurrence of these leap seconds, UTC is not useful when calculating astronomical details for future dates. Likewise, UT1 with its unpredictable variations is also not a suitable time scale for astronomical predictions. For this reason, astronomers use a time scale referred to as Terrestrial Time (TT), or Dynamic time. For historical reasons TT was offset by 32.184 seconds at the time when the UTC standard came into effect. Additionally, it ticks forward with complete disregard for leap seconds. Thus, currently it is offset from UTC by 32.184 + 37 = 69.184 seconds, a value that will change if further leap seconds are added or subtracted in UTC.

TT is the time scale used for JDs when calculating positions of planets and stars on the celestial sphere, so almost all functions in Astronomy which use accept a point in time will require JD\_TT. When translating celestial coordinates to local observer coordinates Astronomy makes use of Greenwich Sidereal (GSID) time to account for the rotation of Earth relative to the celestial coordinates. As UTC attempts to say close to the rotation of Earth, the Astronomy algorithms using GSID require JD\_UTC for accuracy.

In other words, since an observer will see UTC date times on their clocks, Astronomy should use JD\_UTC when converting to the local coordinate system of the observer, so that the positions returned will match the sky at that time of the day. EDateTime (and therefore Astronomy) calculates both JD\_TT and JD\_UTC automatically, and Astronomy will use the correct one whenever a JD parameter is omitted. The function signatures will indicate either jd\_tt or jd\_utc in case you want/need to be explicit about the moment in time.

For dates in the past EDateTime assumes that proleptic Gregorian dates are in UTC, so it will compensate for the 32.184 seconds offset, and the zero leap seconds that were (or weren’t) applied before UTC was adopted. CAADynamicalTime from the AA+ library uses a different approach, where times are converted to UT1 when they are outside the range of UTC. This involves tracking the actual long-term variations in Earth’s rotation by using DeltaT tables. If the added accuracy is required, please use the AA+ library for conversions and manually pass the values to Astronomy. The variation is on the order of a few milliseconds per year, so when calculating the sky a few thousand years ago, this amounts to only a few minutes offset in observation time [FACT CHECK THIS ESTIMATE].

For more on the history of time keeping and the different time scales in use, see appendix A.

## Coordinates in Astronomy

Because the numerical accuracy of computer calculations is limited, each astronomical calculation is performed in the coordinate frame that is most convenient for that type of calculation. This ensures that the calculations are as short as possible, and thus allow for the minimum number of rounding errors.

It is straight forward to observe that stars remain fixed relative to each other while traversing the sky in both diurnal (daily) and annual (yearly) motions. The Sun, Moon, and planets (solar system objects) however appear to be moving relative to this background of stars. It turns out that it is very convenient to model the movements of solar system objects relative to the background of stars. The background of stars then becomes a coordinate reference frame which we call the Celestial Sphere, which appears to rotate uniformly around us once per day.

As the name suggests, positions on the celestial sphere are given in spherical coordinates. The spherical coordinate system is aligned with the Earth north and south poles and equator. Rather than latitude and longitude as used on Earth, the celestial coordinates are called Declination and Right Ascension. Like latitudes, declinations (Dec in the following) are given in -90.0 to +90.0 degrees. Right ascension (RA in the following) however, are given in 0.0 to 24.0 hours, reckoned from a point called the vernal equinox (or historically the first point of Aries.) I will define this point later, after first defining the ecliptic coordinate system.

The RA values are given in hours because Earth’s daily rotation (by definition) aligns with the RA axis, so using hours makes it easy to work out how the celestial sphere is oriented relative to Earth (or rather how Earth is aligned with the celestial sphere.) This is done by calculating how many hours ago that the vernal equinox point was last above the Greenwich observatory. That value is called the Greenwich Sidereal hour angle (GSID in the following.)

For observers that are not on the Greenwich longitude (i.e., not on the prime meridian), the longitude can be translated to an hour angle with a simple multiplication by 15 (because the sky rotates 360 degrees in 24 hours, and 360/24 = 15.) The result is called the local hour angle (LHA.) Adding up the RA, the GSID and the LHA gives the position of the star relative to Earth at the instant in time.

The Astronomy object uses radians for all internal calculations. For consistency and ease of use, it will communicate with the user in degrees unless radians are requested specifically. In the case of hour angles, Astronomy breaks with the usual convention by returning degrees rather than hour angles. That is to avoid some pitfalls when atypical units are required by some algorithms. This only really matters when screen or console output of hour angle values are required. Functions are supplied to format such output appropriately as needed, see Angles in Astronomy for details.

Star catalogues provide the Dec and RA of the star. While the stars appear fixed on the celestial sphere over human lifespans when making naked eye observations, using powerful telescopes or comparing observations over long time periods (hundreds of years) reveal that stars do move ever so slightly when compared to neighbouring stars (neighbouring in terms of Dec and RA, due to different distances they may not be physically close to each other.) This individual motion of each star is called Proper Motion and is typically specified in milliarcseconds per year (an arc second is 1/3600 of one degree) or equivalently arcseconds per millennium.

Astronomy has a built-in catalogue of all stars visible to the naked eye under optimal conditions. The catalogue contains the celestial coordinates of each star, along with their proper motion, and functions are provided to obtain apply the proper motion according to the current time (or a specified time if desired.)

In addition to proper motion, there are three other effects that influence the apparent Dec RA of a given celestial object; precession, nutation, and aberration. Astronomy provides functions to calculate each of these accurately, or more conveniently functions which return Dec RA values where these corrections have been applied already.

Precession is caused by the Earth’s rotational pole slowly rotating and thus pointing towards different positions on the celestial sphere. The path is roughly circular and has a period of slightly less than 26000 years. As a result, rather than Polaris, Thuban was the apparent pole star about 5000 years ago, and there are long periods where no star is very near the celestial pole.

Nutation is a much smaller effect, but also has a much shorter period. The main terms are due to the gravitational influence of the Moon on Earth’s rotation axis and has a period of about 18.6 years. The effect is in the order of 10 to 20 arcseconds.

Aberration is due to the finite speed of light, and imparts a small shift in observed angle depending on the instantaneous direction of Earth’s movement around the Sun. The effect is similar to how vertically falling rain appear to be falling at an angle towards a person moving briskly through the rain. Since the speed of Earth around the Sun doesn’t vary much the effect also doesn’t vary much. It is around 20 arcseconds. Of course, the direction does vary, as does the direction of Earth’s movement around the Sun.

These three effects are not observable with the naked eye over brief periods of time. But especially the effect of precession builds up over centuries. Like the aberration due to the Earth’s annual movement, there is a smaller aberration due to Earth’s diurnal rotation. That too is accounted for in the functions supplied in Astronomy.

An even smaller effect is due to parallax when viewing a star from opposite sides of Earth’s orbit around the Sun. This effect is so small that Astronomy disregards it. Parallax of the Moon is large enough, even when viewed from opposite sides of the Earth itself, so that is taken into account in the relevant functions. Parallax of the Sun and planets are handled when converting from VSOP87 reference frame to celestial coordinates [VERIFY THIS CLAIM].

This brings us to the reference frame used when calculating the position of solar system objects (apart from our Moon.)

Earth orbits the Sun in a flat plane called the Ecliptic, or equivalently the Sun annually travels across the sky background star map in a plane called the Ecliptic. All the major solar system objects (the Sun, planets, asteroids etc) travel in a narrow band around the ecliptic, so it makes sense to model their positions using ecliptic coordinates. The orbits are near perfect ellipses with the Sun at one focal point, so it is convenient to centre the ecliptic coordinate system on the Sun (or the barycentre of the solar system which is very near the Sun). Thus heliocentric (or barycentric) ecliptic coordinates are used in the VSOP87 calculations for the planet positions.

From Earth we observe that every astronomical object rises in the east (not due east, simply the eastern part of the sky), and sets in the west. This is a daily motion that is well modelled by Earth rotating once per day. If this diurnal (daily) rotation is subtracted, we notice that

## Angles in Astronomy

Angles are always represented as radians internally. However, when passing values into functions and receiving values back from functions, the default is degrees. That can be changed to radians if so desired, using the optional Boolean flag “rad”.

Although AA+ uses a mix of angles in hours, degrees, minutes of degree and seconds of degree, Astronomy internally represents all angles as radians. All functions that are meant to be used directly will return degrees only, never hour angles or minutes/seconds. Using the “rad” Boolean flag, users can optionally receive the radian values directly. When using the “rad” flag to receive radians, the inputs must also be supplied in radians.

## Distances in Astronomy

All distances are given in kilometres. AA+ often uses AU, but I wrap and convert those. You are welcome to use the AA+ library directly, it resides in the “aaplus” folder in the project. Using AUs is tradition because determining angles and ratios is relatively easy, whereas determining accurate distances is difficult. Famously, Aristarchus was able to determine the sizes and distances of the Sun and Moon, but only relative to the size of the Earth. It was Eratosthenes that worked out the size of Earth, and thus enabled the calculation of the actual distances and sizes of the Sun and Moon.

Additionally, a kilometre is quite small in solar system scales, so when performing manual calculations using AUs allows the use of smaller numbers and is thus more convenient. Finally giving a value in kilometres offers a false sense of accuracy. I disregard all these to provide output that is more directly accessible. Most people have a sense of how long a kilometre is, not as many have an intuition for the length of one AU (about 150,000,000 kilometres.)

## Astronomy() constructor

You can construct an Astronomy object via a call to Application::newAstronomy(). Currently there is no real reason for this, as Application doesn’t store any reference to the Astronomy object provided.

So in short, you may construct the Astronomy object directly yourself, and pass it to the other objects that need to be aware of it (RenderLayers for example). The constructor takes no arguments. I may add additional constructors in the future that initialize the current time while constructing the Astronomy object, rather than having to call one of the setXXX() functions after construction.

The constructor does initialize the Astronomy object to the current system time converted to UTC.

The signature is as follows (i.e. a default constructor):

Astronomy()

In code, use it as follows:

Astronomy\* astro = new Astronomy();

The Astronomy constructor also loads the stellar object database, see below.

## Astronomy Stellar Object Database

The Astronomy class comes with access to a static database of stellar objects. Those are stars, nebulae, and galaxes; all outside our solar system. The database can be used even without instantiating an Astronomy object, but with limited functionality. The database is basically a modified export from SIMBAD [REFERENCE], with the following datapoints for each object:

* RA and Dec of the object in ICRS J2000.0 which is nearly identical to FK5 J2000.0.
* Proper Motion in RA and Dec. The RA value is already multiplied by cos(Dec).
* The magnitude of the object in the visible spectrum.
* The colour of the object in RGB space, which matches OpenGL’s colour space.
* The SIMBAD canonical NAME identifier string, e.g. “\* tet Per” for theta Perseid.

These values are kept in a struct named Astronomy::stellarobject. Additionally, a list of 82 stars (including the 57 standard stars used in celestial navigation) have been officially named by the IAU. Those names are in a cross reference vector Astronomy::stellarobject\_xrefs with the common name and the identifier string used in the stellarobjects database. Querying by name always attempts to resolve the name in this cross reference first, before searching the identifier strings in stellarobjects.

The two database tables (std::vector) are defined as static along with the following functions and variables. This means they can be queried without having instantiated an Astronomy object. However, the databases are not loaded into memory until you explicitly ask for it, or until at least one Astronomy object is instantiated.

To check if the static stellar objects database has been loaded, query the Astronomy::stellarobjects\_loaded Boolean variable. To manually load the database without instantiating an Astronomy object, use the Astronomy::loadStellarObjects() function.

Once loaded, the database can be queried as follows:

If (!Astronomy::stellarobjects\_loaded) Astronomy::loadStellarObjects();

glm::vec4 sirius\_color = Astronomy::getColorbyName(“Sirius”);

std::cout << “Sirius has a visual light magnitude of “ << sirius\_color.a << ‘\n’;

Alternatively, if you need several datapoints, you can obtain a stellarobject directly:

…

stellarobject sirius = getSObyName(“Sirius”);

std::cout << “Sirius has a visual light magnitude of “ << sirius.vmag << ‘\n’;

std::cout << “Sirius has a proper motion (RA,Dec) of “ << sirius.pm\_ra << ‘,’ << Sirius.pm\_dec << ‘\n’;

The following query functions are available without instantiating an Astronomy object:

static stellarobject& getSObyName(const std::string starname);

static LLH getDecRAbyName(const std::string starname, bool rad = false);

static LLH getDecRAwithPMbyName(const std::string starname, double jd, bool rad = false);

static glm::vec4 getColorbyName(const std::string starname);

Feel free to access the vectors directly. There is also a member (double stellarobject\_epoch;) which contains the epoch for the database entries. This is currently hard coded to J2000.0 and is unlikely to change. Note: The proper motion values in the raw Hipparcos catalogue itself are at epoch 1991.5, but they have been recalculated to 2000.0 in SIMBAD.

### Astronomy::stellarobjects

Signature:

Std:vector<stellarobject> stellarobjects;

This is a vector of stellarobject structures. The structure used to keep and return a full database entry in the stellarobjects database:

struct stellarobject {

double ra = 0.0;

double dec = 0.0;

double pm\_ra = 0.0;

double pm\_dec = 0.0;

double vmag = 0.0;

double red = 0.0;

double green = 0.0;

double blue = 0.0;

std::string identifier;

};

The ra and dec members are the position at J2000.0 in ICRS, both given in decimal degrees. The pm\_ra and pm\_dec members are the angular proper motion components, both given in milliarcseconds per year (mas/yr), and pm\_ra is already multiplied with cos(pm\_dec).

The vmag member is the usual magnitude in the visual spectrum. The database contains almost all SIMBAD objects with a vmag <= 6.5, a few nebulas have been removed since Eartharium is unable to render them in a meaningful way (they appear a relatively large smears, even to the naked eye.)

The limit of 6.5 was chosen for 2 reasons:

1. That is the claimed limit for naked eye observation for someone with optimal eyesight under optimal observing conditions.
2. This cut-off results in a database with nearly 10.000 objects, which is the limit for obtaining near real-time performance on average computer hardware produced around 2020.

The vmag field can be used as a filter when attaching stellar objects to e.g. the SkySphere object. This is a handy way to generate a SkyShpere local to one observer, based on the observing conditions at that location.

The red, green, blue values are floats in the range 0.0 to 1.0, representing the colour of the stellar object as it appears to the human eye. This has been calculated by taking the spectral type from SIMBAD, converting it into a colour temperature using well established methods, and then further converting into an RGB value.

The identifier is a text string which serves as the canonical NAME field in SIMBAD. When looking up individual stellar objects, this is the only search parameter currently available. The search is an exact match, so you need to know the precise spelling of the object you are interested in. These identifiers are not entirely intuitive. For example, the star theta Perseid is called “\* tet Per”, note that ‘tet’ is not necessarily the abbreviation you might expect for “theta”. Likewise “alpha” abbreviates to ‘alf’.

There is a cross-reference available via stellarobject\_xrefs, which can be used to map the 82 common names defined by IAU to the identifiers.

[TODO]Future work will allow you to provide your own cross-references. In the near future a HIP identifier will be added, designating the Hipparcos catalogue number. That should make object lookup more straight-forward.

### Astronomy::stellarobject\_xrefs

This is a vector (std::vector<stellarobject\_xref> stellarobject\_xrefs;) of stellarobject\_xref structures:

struct stellarobject\_xref {

std::string popular\_name;

std::string identifier;

};

The popular\_name is the official IAU name of the star, implementing the current official list of 82 named stars (e.g., “Polaris” or “Sirius”). This is a super-set of the stars usually used for celestial navigation (and thus listed in the nautical almanacs) and should make modelling of celestial navigation topics relatively straight-forward. See for example the SubPointSolver class.

# EDateTime object

The EDateTime object used by Astronomy to track the current time provides functions for setting the current time in several different ways. If no time point is provided when constructing an EDateTime, it will by default take the current local system time and convert it to UTC, then calculate the JD\_TT and JD\_UTC Julian Dates used for astronomical calculations.

It is also possible to progress the current time with a specified number of days, minutes and/or seconds. For this reason, it is possible to arrive at impossible dates such as the 32nd of February, or times such as 19:87:14.5. EDateTime has a function to normalize these into valid dates and times. That function (normalize()) is automatically applied when setting the date/time directly, and when using the addTime(double days, double, hours, double minutes, double seconds) function.

This serves two purposes: 1) to be able to display dates and times correctly, and 2) to prevent incorrect results in the functions that calculate JD\_TT and JD\_UTC.

The Astronomy class wraps most of the functions provided in EDateTime, so you may never need to use EDateTime directly, except perhaps for the static conversion functions.

## Static Functions

### EDateTime::normalizeDateTime()

Signature:

static void normalizeDateTime(long& yr, long& mo, double& da, double& hr, double& mi, double& se)

This function will normalize a date time in place, using the individual components.

If an EDateTime object is already available, it will be normalized whenever it is assigned a new time point, or the existing time point is adjusted, so there is no need for a version that takes an EDateTime object directly.

### EDateTime::myDivQuotient()

Signature:

static int myDivQuotient(int a, int b)

This, along with EDateTime::myDivRemainder(), is used when performing JD calculations, based on the algorithms developed in <https://www.aa.quae.nl/en/reken/juliaansedag.html>. These algorithms are superior to those given in the AA+ library, since their validity extends to millions of years. This does not mean that the actual astronomical calculations have this extended range.

### EDateTime::myDivRemainder()

Signature:

static int myDivRemainder(const int a, const int b)

This, along with EDateTime::myDivQuotient(), is used when performing JD calculations, based on the algorithms developed in <https://www.aa.quae.nl/en/reken/juliaansedag.html>. These algorithms are superior to those given in the AA+ library, since their validity extends to millions of years. This does not mean that the actual astronomical calculations have this extended range.

### EDateTime::getDateTime2JD\_UTC()

Signature:

static double getDateTime2JD\_UTC(const long year, const long month, const double day, const double hour, const double minute, const double second)

Returns the JD\_UTC of the given UTC date and time. The JD\_UTC can be used to calculate Greenwich sidereal time in the Astronomy class. For all the positional computations, JD\_TT is required, see getDateTime2JD\_TT() and getJDUTC2TT().

### EDateTime::getDateTime2JD\_TT()

Signature:

static double getDateTime2JD\_TT(const long year, const long month, const double day, const double hour, const double minute, const double second)

Returns the JD\_TT of the given UTC date and time. This is the JD used with all the functions in the Astronomy class, except the ones calculating Greenwich sidereal time.

### EDateTime::getUnixTime2JD\_UTC()

Signature:

static double getUnixTime2JD\_UTC(const long unixtime)

### EDateTime::getUnixTime2JD\_TT()

Signature:

static double getUnixTime2JD\_TT(const long unixtime)

### EDateTime::getDateTime2UnixTime()

Signature:

static long getDateTime2UnixTime(const long year, const long month, const double day, const double hour, const double minute, const double second)

### EDateTime::getJD2MJD()

Signature:

static double getJD2MJD(const double jd)

MJD is Modified Julian Date, which is used in some astronomical calculations. It has the same time scale as JD, but is offset to start at November 17th 1858 rather than far in the past. This may seem like an odd date to pick, but the resulting calculation is MJD = JD - 2,400,000.5 (the .5 shifts the counting of days from midday to midnight, the 2.4 million shifts the range to a much more manageable range. Dates ranging from the one provided above and about 3 centuries ahead can thus be written with 5 digits before the decimal place rather than 7 digits.)

If a JD\_UTC is passed to the function, a UTC based MJD is returned. Likewise, passing a JD\_TT results in an MLD in TT. For both, keep in mind that the days are then counted from midnight rather than midday.

### EDateTime::getMJD2JD()

Signature:

static double getMJD2JD(const double mjd)

See getJD2MJD() above for details. JD = MJD + 2,400,000.5.

### EDateTime::getJDUTC2TT()

Signature:

static double getJDUTC2TT(const double jd\_utc)

UTC is subject to leap seconds to keep the difference between UT1 and UTC below 0.9 seconds in either direction. Including those leap seconds into astronomical formulas is difficult and unnecessary. Terrestrial Time (TT), which ticks on without adding any leap seconds is used instead. The difference between UTC and TT is 32.184 seconds (for historical reasons), plus the number of leap seconds accumulated in UTC at a given year. Currently (April 2023), this amounts to 32.184 + 37 = 69.184 seconds.

This function calculates (via a lookup table for leap seconds) the JD\_TT corresponding to the provided JD\_UTC. The inverse function is getJDTT2JDUTC().

If an EDateTime object is instantiated, the JD\_TT is pre-calculated when setting or changing the time and/or date of the object and can be obtained directly with the .jd\_tt() member function.

### EDateTime::getJDTT2UTC()

Signature:

static double getJDTT2UTC(const double jd\_tt)

UTC is subject to leap seconds to keep the difference between UT1 and UTC below 0.9 seconds in either direction. Including those leap seconds into astronomical formulas is difficult and unnecessary. Terrestrial Time (TT), which ticks on without adding any leap seconds is used instead. The difference between UTC and TT is 32.184 seconds (for historical reasons), plus the number of leap seconds accumulated in UTC at a given year. Currently (April 2023), this amounts to 32.184 + 37 = 69.184 seconds.

This function calculates (via a lookup table for leap seconds) the JD\_TT corresponding to the provided JD\_UTC. The inverse function is getJDUTC2JDTT().

This is useful, for example when using Astronomy::getGSid(), which takes JD\_UTC rather than the JD\_TT value used by most other functions in the Astronomy class.

If an EDateTime object is instantiated, the JD\_UTC is pre-calculated, and can be obtained directly with the .jd\_utc() member function.

### EDateTime::calcUnixTimeYearDay()

Signature:

static long calcUnixTimeYearDay(const long year, const long month, const double day)

### EDateTime::isLeapYear()

Signature:

static bool isLeapYear(const long year)

This function returns true or false depending on whether the passed in year is a leap year or not.

If an EDateTime object has been instantiated, the member function .isLeap() provides the same functionality without needing to pass the year.

## Member Functions

Member functions are available during or after instantiation of EDateTime objects. For the static class functions, see above section.

### EDateTime() – Constructors

When EDateTime objects are instantiated, they are always initialized to sane normalized time points. All the member variables are consistent after construction, in particular the JD\_TT and JD\_UTC values are already calculated.

Signatures:

EDateTime()

EDateTime(long year, long month, double day, double hour, double minute, double second)

EDateTime(double jd)

[TODO]EDateTime(long unixtimestamp) – This is not yet implemented

The default constructor without any arguments will initialize the EDateTime object to the current system time (converted to UTC) if available.

The constructor taking (year,month,day,hour,minute,second) arguments will normalize the input and construct the timepoint indicated. It will also calculate the JD\_TT and JD\_UTC values before returning the object.

The constructor taking a JD as argument expects the JD to be in UTC. All member variables, including JD\_TT are calculated before returning the object.

If you happen to have a JD\_TT rather than a JD\_UTC, simply convert it using the static function EDateTime::getJDTT2JDUTC() before calling the constructor. Alternatively, you can use the default constructor and then call the .setJD\_TT() member function, but this has additional overhead as EDateTime then calls the .normalize() function more than once.

### ~EDateTime() – Destructor

EDateTime uses the automatically generated default destructor, since it only uses primitive member variables.

EDateTime getters

I have grouped these to make the documentation easier to read. The following getter member functions simply return the stored values of the corresponding (private) member variables:

long .year() – Returns the current year.

long .month() – Returns the current month (of year).

double .day() – Returns the current day (of month).

double .hour() – Returns the current hour in time of day.

double .minute() – Returns the current minute in time of day.

double .second() – Returns the current (fractional) second in time of day.

double .Jd\_tt() – Returns the current timepoint as JD in TT.  
double .Jd\_utc() – Returns the current timepoint as JD in UTC.

bool .isLeap() – Returns true if .year() is a leap year, false otherwise.

Long .weekday() – Returns the week day number: 0 = Sunday, 1 = Monday etc.

[TODO} std::string .string() – Once implemented, will return the current time as UTC string.

### .setTime()

Signature:

void setTime(long year, long month, double day, double hour, double minute, double second)

Sets current time to the provided, and normalizes the date/time, then calculates the JDs.

### .setTimeNow()

Signature:

void .setTimeNow()

Sets the current time to the current system time. Note that the system time is obtained with a granularity of 1 second, due to the limited std functions available in C++ 17. JDs are calculated before returning.

If greater than 1 second of accuracy is required in a real-time while() loop, you will currently have to implement this on your own and use .setTime() with fractional seconds. This is difficult though, as Eartharium currently doesn’t offer easy access to frame render times, making it harder to predict what actual time to apply to the next frame.

### .setJD\_\*()

Signatures:

void setJD\_UTC(double jd\_utc)

void setJD\_TT(double jd\_tt)

These set the current JD in UTC or TT respectively. The corresponding date and time is then calculated and normalized. When setting JD in UTC, JD in TT is also set correctly, and vice versa.

For a discussion of JDs in UTC vs TT, see the section “JD in UTC or TT” [TODO: write that section and refactor other references to refer to that too]

### .setUnixTime()

Signature:

void setUnixTime(long unixtime)

You may have Unix Timestamps readily available, or you might be using Eartharium functionality that utilize Unix Timestamps, such as RenderLayerPlot (which is based on ImPlot, where timepoints are always supplied in Unix Timestamps).

void addTime(long year, long month, double day, double hour, double minute, double second)

void normalize()

void calcJDs()

CountryBorders

TimeZones

Scene

* SceneTree
* SceneObject
* Camera

Earth

Location

SkySphere

Lerper

# Appendix A – History of Time and Date

Time is complicated when it comes to astronomical calculations. Human time keeping is full of discrepancies. For example, adding one month to the 15th of March results in a time 31 days later, whereas adding a month to the 15th of February results in a time 28 days later, except for leap years where it is 29 days later.

When using computer algorithms, it is desirable to have a time that increases monotonically with out conditions and special cases. For this reason, astronomical calculations are based on what is referred to as Dynamical Time, and user provided times and dates are first converted to Dynamical Time before use. There are some subtleties that are worth your attention, so in the following I will offer a lot of detail on time keeping throughout history.

In ancient times, different peoples used different calendars. While those are all very interesting, and often based on astronomical events. The Roman empire standardized on what was to be known as the Julian calendar. It had 365 days, with a leap day every 4 years. This turned out to be just a bit too many leap days, and over time it came out of sync with the seasons.

Eventually the Catholic church decided to implement a calendar that would keep better synchronization with the seasons. That is the calendar we all use now, the Gregorian calendar. It was officially put into service during what we call the Papal Reform on 15th of October 1582, but many countries were hesitant to adopt it, the last was the Soviet Union in 1973 [CHECK DATES]. For this reason, we need to be aware of the location when talking about a specific date. I have opted for a different solution which is to pretend that the Gregorian calendar has always been used as far back as the implemented algorithms are valid (thousands of years). This is called “the proleptic Gregorian calendar”.

But there is more to it than that. Historians give years as CE (current era) or AD (anno Dominicus) for positive years, and BCE (before current era) or BC (before Christ) for negative years. Officially there was never a year zero, so historians go straight from year 1 CE to year 1 BCE (when counting backwards), thus skipping year 0. That is inconvenient when doing calculations, so astronomers consider the year before 1 CE to be year 0, and the year before that to be year -1.

The Gregorian calendar inserts a leap day (29th of February) in years that are divisible by 4, except when also divisible by 100. But this is still not enough, so if divisible by 400, it IS a leap year. Astronomers thus consider year 0 to be a leap year, and the same for years -4, -8 etc, whereas historians consider the years 1 BCE, 5 BCE and 9 BCE to be the leap years.

For calculations astronomers use an integer day count called Julian Day, starting from November 24th year -4713. For times within a day, they simply add decimals, but they consider x.0 to be noon on the given day. The following midnight is then x.5, half a day later.

Time formats

Most of the functions in Astronomy take Dynamical Time, often referred to as Terrestrial Time. This is a time scale that progresses monotonically without discontinuities. Historically it was defined by astronomical processes which were more stable than the rotation of Earth. Nowadays it is based on the international atomic time (TAI), but to match up with the previously used Ephemeris Time (ET), it is offset from TAI by 32.184 seconds. I.e., TT = TAI + 32.184 seconds.

UT1 is the time frame that exactly follows the rotation of the Earth. Since this rotation varies minutely over both short and long time periods in an unpredictable manner, TAI and UT1 will deviate from each other over time. It is desirable to have wall clock time which is both relevant to the day and night cycle and is generally ticking at a regular pace. UTC is used for this, it ticks at the rate of TAI, but is required to stay within 0.9 seconds of UT1 in either direction. Thus, leap seconds are occasionally introduced to realign UTC with UT1.

Having to partition astronomical calculations into pieces to accommodate every leap second introduced is very inconvenient. Therefore, astronomical calculations use TT rather than UTC, and conversion functions are used to translate between the two systems. Tables are available detailing the leap seconds enacted since UTC was introduced. At the time of writing (July 2023) a total of 37 leap seconds have been added.

The rotation of Earth has generally been slowing down since the introduction of UTC. That causes UTC to be ahead of UT1. Perhaps counter intuitive at first, adding a leap second to UTC will cause it to realign with UT1. It all makes sense. Say UTC is 1 second ahead, it displays 23:59:59 when UT1 displays 23:59:58. By adding a leap second to UTC it will display 23:59:60 for a second, while UT1 moves to 23:59:59. On the next tick, both display 00:00:00.

The difference between UT1 and UTC is called deltaT and has been kept within 0.9 seconds since UTC was introduced in 1961. But it is interesting to perform astronomical calculations for times before 1961, so there are deltaT tables and formulas available stretching back at least to year -500. Since no leap seconds were added before 1961, UT1 and UTC will diverge the further back in time we look. For year -500 deltaT is around 7 hours.

The available tables have been generated based on historical astronomical records. Those tables have then been fitted with polynomials, which approximate deltaT to within a few seconds. Tables date back to around year 1600, so within that range one can choose to use the tables or the polynomials. Before that time, the only choice is to use the polynomials. Those have been tuned to scant historical records of eclipses and conjunctions, where available.

So that accounts for the TT/ET timeframe stretching back in time to the ancient Greeks. But they did not use the Gregorian calendar back then. There are two choices; either translate to the Julian calendar for dates before 15th October 1582, or simply pretend that the Gregorian calendar has always been in use. Most astronomy software converts to the Julian calendar. Since different countries adopted the Gregorian calendar at different times, I feel that approach hides the need to validate dates in historical documents against the conversion date relevant to that document. By only using the Gregorian calendar, I force users to do the necessary research. When used like this, it is called the proleptic Gregorian calendar (likewise the proleptic Julian calendar for those that take the former approach.)

As mentioned, most of the functions take Dynamical Time (TT). But there are a few exceptions, and apart from the relevant time conversion and set functions, they are the ones that relate directly to the rotation of Earth. These are functions that calculate Sidereal Time and Hour Angles, such as getGsid() and calculateGsid(). The function signature reveals whether a jd\_tt or jd\_utc is needed.

When converting a date within the UTC range, apply only leap seconds and TAI to TT offset (32.184 seconds). When outside of the UTC range, apply only deltaT, as it has the TAI to TT offset included. In fact, the deltaT table overlaps with the leap second table, and contains the correct values for the whole range. So one could simply apply deltaT regardless of the range. In fact, from Jan 1961 to Feb 1968, leap seconds were applied smeared, so the leap seconds table with interpolation is perhaps more accurate. Both table lookups in AA+ appear to interpolate, so maybe one day I should test them carefully.