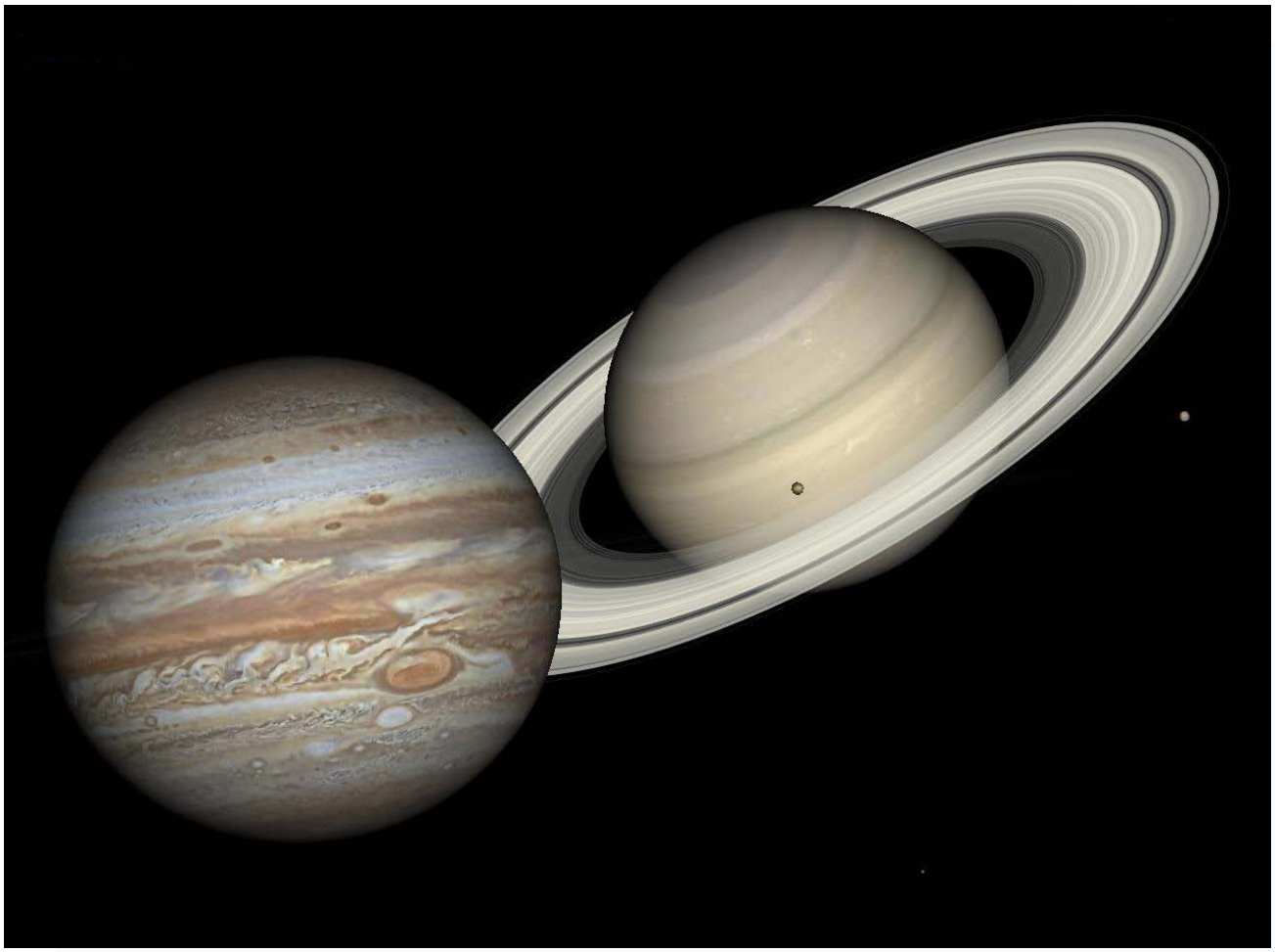

CELX Scripting for Celestia

How to migrate from CEL scripting

Version 1.2
Revision January 12, 2010

Autor: Marco Klunder

Principal authors: Selden Ball, Chris Laurel, Vincent Giangiulio
"Celestia/Celx Scripting/CELX Lua methods"
Don Goyette
"Celestia .Cel Scripting Guide v1-0g"

Publisher: Marco Klunder

Copyright notices

License notice

History section:

Document	Version	Author	Date	Modification
Celestia/Celx Scripting/CELX Lua Methods http://en.wikibooks.org/wiki/Celestia/Celx_Scripting/CELX_Lua_Methods	May 16, 2009	Selden Ball, Chris Laurel, Vincent Giangiulio	May 16, 2009	Principal CELX scripting document
Cel Script Guide v1-0g.doc http://www.donandcarla.com/Celestia/	v1-0g	Don Goyette	June 17, 2004	Principal CEL scripting document
CELX Scripting for Celestia (How to migrate from CEL scripting)	V1.0	Marco Klunder	July 4, 2009	Document containing texts phrases of the above documents in combination with new texts, examples and explenations to help using CELX and migrate from CEL to CELX.
CELX Scripting for Celestia (How to migrate from CEL scripting)	V1.1	Marco Klunder	September 23, 2009	Added Celestia 1.6.1 method references. Changed setURL equiv. Bugfix CELX script init. Minor other adjustments.
CELX Scripting for Celestia (How to migrate from CEL scripting)	V1.2	Marco Klunder	January 12, 2010	Added more Celestia 1.6.1 method references. Adjusted CELX script initialization. Additional CELX equivalents for goto, gotoloc. Adjusted CEL: time chapter. Corrected CEL: setposition chapter. Adjusted CEL: seturl chapter. Index to new available WIKI rotation methods added. Minor other adjustments. This document is now also wikified as an integrated part of already existing WIKI documentation. New releases will only become available in WIKI.

Table of Contents

Background Information

CELX scripting is based on the LUA object oriented programming Language. The principles of LUA are beyond the scope of this document, but can be found on the internet at the following address: http://www.lua.org/manual/5.1/.

Some of the available and valuable WIKI documentation "Celestia/Celx Scripting/CELX Lua Methods**"** by Selden Ball, Chris Laurel and Vincent Giangiulio, about CELX scripting in Celestia is also used within this document. It is however not the intention of this guide to repro this content, but use it for further explaination about the basics of CELX scripting. Besides, it is also not the intention of this document to explain the working of all the Celestia objects and methods. However, to be as complete as possible, this document will contain the links to all available Celestia methods and objects information in "Celestia/Celx Scripting/CELX Lua Methods**".**

Related to the migration of CEL commands to CELX objects and methods, this document will address the access to the different objects and methods in a structured way, including the corresponding explanation and the use of it with some examples. As a source for this explanation, the available Celestia .Cel Scripting Guide v1-0g by Don Goyette is used. According the sequens and explanation of CEL script commands in Don's document, the use of the corresponding LUA and CELX objects and methods will be addressed in this "CELX Scripting for Celestia (Sub-title: How to migrate from CEL scripting)" document, including many of his examples and syntaxes, to be as complete and consistent as possible to understand how the migration from CEL to CELX works.

This document describes the CELX objects and methods as they are available in Celestia v1.6.1. Because in Celestia v1.6.0 / v1.6.1 there are some new object, methods and parameters available, it is highly recommended to use this document combined with Celestia v1.6.0 or Celestia v1.6.1. Version dependent objects, methods and parameters is this guide are marked according the colors in this paragraph.

Because this document contains hyperlinks to internal and external adresses, it is adviced to use this document online. It is however possible to make a printed copy of this document and use it as an offline reference manual. In that case, the hyperlinks will logically NOT work.

This document "CELX Scripting for Celestia" v1.2 was written by Marco Klunder, January 2010. I started CEL scripting for Celestia a few years ago and really enjoyed is. Because of some new features within Celestia, I started in 2009 to program de scripts in CELX. In the beginning I really had some "cold water fear", and differences between CEL and CELX needed to be understood first. Reading the available documentation on the internet, I found out a lot, but not all! Sometimes, help on the internet was very useful, and in June 2009, I finished my CELX script about Conjunctions and Occultations successfully and released it on the Motherlode. After that, I wrote my additional experiences about CELX scripting and the migration from CEL to CELX in this document, in the hope to help more people in the world to start using CELX scripting. The document contains a lot of useful information already, but I'm also sure it can be improved in future releases, because each day I discover new things and possibilities again. If you have serious comments, remarks, additional information, etc., please feel free to contact me. I hope you will enjoy using CELX in the future, as much as I do now!
Contact: marco.klunder@hccnet.nl

Syntax

The complete description of the syntax of LUA is beyond the scope of this document. For the interested, it can be found at: http://www.lua.org/manual/5.1/.

For some basic CELX scripting, it is however handy to read the following notes and examples, to better understand some principles:

Simply write your CELX script line after line, using a line editor like “Notepad”, “Notepad++” or “Wordpad”, without text formatting.

Comment lines start with a "--" (two hyphens)

```
-- This is a comment line
```

Variables & Types

1. **Variables** in LUA en CELX don't have to be declared at the beginning of your script and they don't have a type. However the **content** of a variable has a type (e.g. Integer, String).

```
-- Examples of the same variable "a" having content of different type:
a = 1                -- a is 1, an Integer Number
a = a / 3            -- a is now 0.5, changed into a Floatingpoint Number
a = "Hello World"    -- a contains a String now
a = a / 2            -- ERROR, because a String is divided by a Number!
a = true             -- a is true, a Boolean
a = not a            -- a is now false, a Boolean
```

2. Besides Variables, **Tables** also exist in LUA en CELX, which can be used as arrays or hashes. A Table can be used like this:

```
t = {}                                  -- define Table t
t["key1"] = "value1"                    -- long form to store a string
t.key2 = "value2"                       -- short form to store a string
u = { key1="value1", key2="value2" }    -- u is now the same as t
v = { 1,2,3,4,5 }                       -- v[1] is now 1, v[2] is 2, etc.
```

3. At the beginning of a script, the **celestia object** is automatically defined, holding a reference to the core-functionality of Celestia! To operate on this **celestia object** you can call **methods** which are defined on it (see chapter Index celestia methods), by using the “**celestia:**” prefix, as shown in the following examples:

```
-- Get observer instance of the active view and store it in "obs"
obs = celestia:getobserver()

-- Find the celestial object representing Earth and store it in "earth"
earth = celestia:find("Sol/Earth")

-- You then could use "obs" and "earth" like this.
--    Start the goto command to move the observer to Earth,
--    as if you had pressed the [G] key in Celestia:
obs:goto(earth)
```

4. Besides the predefined **celestia objects**, there are also other **celestia related objects** to control Celestia. You can't create an object (i.e. userdata) yourself, you must call some method to create one. As you only have the **celestia object** (ignoring methods which are not Celestia related) available when the script is starting, you must use it to create other objects. The following **celestia related objects** can be used within a CELX script:
   - ➔ Observer object:
     An **observer object** is used to access properties specific to a view, such as viewer position, viewer orientation, frame of reference, and tracking status. To operate on an **observer objects**, you can call **observer methods** which are defined on them, as shown in chapter Index observer methods.
   - ➔ Object objects:
     An **"object" object** in CELX refers to a celestial object like a planet or a star. To operate on **"object" objects**, you can call **"object" methods** which are defined on them, as shown in chapter Index object methods.
   - ➔ Position objects:
     A **position object** contains the exact coordinates of a point in space. To operate on **position objects**, you can call **position methods** which are defined on them, as shown in chapter Index position methods.
   - ➔ Vector objects:
     A **vector object** is a geometric object that has both a length and direction [X,Y,Z] in a 3-dimensional Coordinate System.
     To operate on **vector objects**, you can call **vector methods** which are defined on them, as shown in chapter Index vector methods.
   - ➔ Rotation objects:
     A **rotation object** is internally a Quaternion, which is one possibility to mathematically describe a rotation in 3 dimensions (i.e. it can be converted to a rotation matrix). A rotation can also be used to describe the orientation of objects or the observer (i.e. where the observer is looking to, and where "up" is). To operate on **rotation objects**, you can call **rotation methods** which are defined on them, as shown in chapter Index rotation methods.
   - ➔ Frame objects:
     The **frame object** describes the celestia coordinate cystem and tells how the X, Y, and Z, axes of each 3-dimensional coordinate system are aligned. Coordinate systems are well documented in the available Celestia .Cel Scripting Guide v1-0g by Don Goyette, in the chapter: "Coordinate Systems". To operate on **frame objects**, you can call **frame methods** which are defined on them, as shown in chapter Index frame methods.
   - ➔ Phase objects:
     The timeline of an object can be subdivided into one or more phases. Each **phase object** has its own trajectory, rotation model, orbit frame, and body frame. To operate on **phase objects**, you can call **phase methods** which are defined on them, as shown in chapter index phase methods.
   - ➔ CELscript objects:
     A **CELscript object** contains a string with a valid CEL script, which can be inbedded in a CELX script by using the **"celestia:createcelscript()" method**.

For basic CELX scripting in Celestia, you mostly need to know only about these available objects, the methods defined on them and how to call them! This will be further explained in the following chapters of this "CELX Scripting for Celestia" document.

Function definition and call:

A function definition is an executable expression, compiled from a block of LUA and CELX code, whose value has type **function**.

A **function** can have arguments for input values and a **function** can also return zero, one or more values.

One or more **functions** can be defined at the beginning of your CELX script and they can be called once or several times from the main body of your CELX script, as shown in the following example:

```
-- Define function with name "add_one" and parameterlist "i" (a number).
function add_one(i)
   -- use 'local' to declare variables local to function:
   local j = i + 1
   -- return the value of "j", a number.
   return j
end

-- Define function with name "divide" and parameterlist "i, j" (two numbers).
function divide(i, j)
   -- return the value of the division "i / j", a number.
   return i / j
end

-- Start of the main body of your script
< ... other script code ...>
-- and now call the functions:
a = add_one(1)
b = divide(a, 2)
< ... other script code ...>
```

Besides you can define your own functions in a CELX script, there are also many functions already defined in LUA, such as mathematical functions or string operations.

Control structures:

Within CELX scripting you can also use typical **control-structures**, like "**for**", "**while**", "**repeat**" and "**if**".

Below you find some examples on how you can use them:

```
-- Execute the block of codelines (...) 10 times after each other,
--    where "i" is incremented with 1 during each loop.
for i = 1, 10 do
   ...
end
```

```
-- Execute the block of codelines (...)
--    as long as "i" is smaller then or equal to 11.
i = 1
while i <= 11 do
   ...
   i = i + 1
```

```
end
```

```
-- Execute the block of codelines (...)
--    until "i" is equal to "j".
i = 1
j = 22
repeat
   ...
   i = i + 1
until i == j
```

```
-- Compare "i" and "j" with each other and execute a
--    certain block of codelines (...1, ...2 or ...3),
--    depending on the result of this comparizon.
if i > j then
   ...1
elsif i < j then
   ...2
else
   ...3
end
```

To loop over the contents of a table "tbl", you can use this:

```
for key, value in pairs(tbl) do
   -- tbl[key] == value
end

-- Use this for tabels used as arrays, i.e. indexed by numbers 1..n:
for i, value in ipairs(tbl) do
   -- tbl[i] == value
end
```

How CELX works in Celestia

Celestia works roughly by repeating the next sequence:

- Check user input and change rendering settings accordingly (e.g. enable rendering of orbits, change observer position/orientation)
- Update the simulation time
- Update the observer's position (if GOTO is active)
- Render all objects using the current settings (renderflags, time, positions)

If a CELX-script has been started, it is executed just before rendering begins. Then Celestia gives control to the LUA-interpreter, which continues to execute the script where he stopped the last time.

Most actions in CELX scripting don't really change anything immediately. Instead they change a setting which is used later on during rendering, which from the point of view of the script happens while calling **“wait()”**.

So if you change the position of the observer ten times without calling wait() in between, this will have no effect - only the last position will actually be used in rendering.

Celestia objects and methods:

Within CELX scripting, it's **NOT** possible to create an object yourself. Instead, you must call some method to create one. Within this document, we ignore methods which are not Celestia related, so you only will use celestia objects to create other objects.

As mentioned before, we know the following celestia related objects (Celestia v1.6.0 / Celetia v1.6.1 objects, methods and parameters are marked according the colors in this paragraph):

- Celestia objects
- Observer objects
- Object objects
- Position objects
- Vector objects
- Rotation objects
- Frame objects
- Phase objects
- Celscript objects

Within the next sub-chapters will be explained how objects can be obtained (except for the **celestia object**, which is already pre-defined and standard available for use in a CELX script). Sometimes there are more possibilities to obtain an object.

Celestia object:

The **celestia object** is predefined and gives access to all the celestia-related functionality by **methods of Celestia**. Within a CELX script, these **celestia methods** need to be prefixed with "celestia:".

Using a celestia method

```
celestia:<celestia method>

-- OR --

objectname = celestia:<celestia method>
```

Examples:

```
-- Double the actual ambient light level.
--
-- getambient() and setambient(n) are both known Celestia methods.

ambientlightlevel1 = celestia:getambient()
ambientlightlevel2 = 2 * ambientlightlevel1
celestia:setambient(ambientlightlevel2)
```

```
-- Show the labels of planets and turn orbits on.
-- Accelerate the time 600 times.
-- Then print a message on the screen in lower left corner for 10 seconds
-- after that, hide the labels of planets and turn orbits off.
--
-- showlabel(), renderflags(), settimescale(), print() and hidelabel()
--    are all known Celestia methods.

celestia:showlabel("planets")
celestia:setrenderflags{orbits=true}
celestia:settimescale(600)
celestia:print("Time runs 600 times faster with planets labeled", 10.0, -1, -1, 1, 3)
wait(10)
celestia:hidelabel("planets")
celestia:setrenderflags{orbits=false}
```

```
-- Set the time in Celestia to January 1, 2000, 00:00:00 hours UTC.
-- (You first have to convert UTC to TDB time, as used in CELX scripts).
--
-- utctotdb() and settime() are both known Celestia methods.

dtbtime=celestia:utctotdb(2000,01,01,00,00,00)
celestia:settime(dtbtime)
```

Index celestia methods

This chapter contains a list of all available **celestia methods**, which can be used on the **celestia object**. The mentioned **celestia methods** are hyperlinks to the Celestia WIKI books about CELX scripting, where further explanation can be found about the functionality of these **celestia methods**.

GOTO Index **observer** methods

GO BACK TO Index **phase** methods

Celestia methods:

- print
- flash
- show
- hide
- getrenderflags
- setrenderflags
- showlabel
- hidelabel
- getlabelflags
- setlabelflags
- getorbitflags
- setorbitflags
- getambient
- setambient
- getfaintestvisible
- setfaintestvisible
- getminorbitsize
- setminorbitsize
- getstardistancelimit
- setstardistancelimit
- getminfeaturesize
- setminfeaturesize
- getstarstyle
- setstarstyle
- find
- getselection
- select
- mark
- unmark
- unmarkall
- gettime
- settime
- gettimescale
- settimescale
- getscripttime
- getobserver
- getobservers
- tojulianday
- fromjulianday
- getscreendimension
- newvector
- newposition
- newposition (base64)
- newrotation (axis-angle)
- newrotation
- newframe
- requestkeyboard
- requestsystemaccess
- getscriptpath
- takescreenshot
- createcelscript
- getstarcount
- getstar
- getdsocount
- getdso
- utctotdb
- tdbtoutc
- stars
- dsos
- gettextwidth
- getaltazimuthmode
- setaltazimuthmode
- getoverlayelements
- setoverlayelements
- getgalaxylightgain
- setgalaxylightgain
- log
- registereventhandler
- geteventhandler
- setlabelcolor
- setlinecolor
- getlabelcolor
- getlinecolor
- getsystemtime
- showconstellations
- hideconstellations
- setconstellationcolor
- ispaused
- gettextureresolution
- settextureresolution
- windowbordersvisible
- setwindowbordersvisible
- synchronizetime
- istimesynchronized
- seturl
- geturl
- settextcolor
- gettextcolor

Observer objects:

Observer objects are used to access properties specific to a view, such as viewer position, viewer orientation, frame of reference, and tracking status.

Within a CELX script, the **observer methods** need to be prefixed with the obtained **observer object**, separated with a semicolon.

The observer methods can be used on an "observer" object, which first need to be obtained by:

Using the celestia "getobserver()" method regarding the active view:

```
-- Get observer instance of the active view and store in obs .
-- Use observer method on observer instance obs.

obs = celestia:getobserver()
obs:<observer method>

-- OR --

celestia:getobserver():<observer method>
```

Here, "obs" contains the current active observer
(the active view can be changed by the user in case of a multiview).

Using the celestia "getobservers()" method in case of a multiview:

```
-- Get observer instances of all used observers and
--    store in table (array) obstab.
-- Use observer method on the first observer instance obstab[1].
-- Use observer method on the first observer instance obstab[2]
--    in case of a multiview with 2 observers.

obstab = celestia:getobservers()
obstab[1]:<observer method>
obstab[2]:<observer method>

-- OR --

celestia:getobservers():<observer method>
```

Here, "obstab" contains a list of all used observers
(one observer for each view in case of a multiview).

Examples:

```
-- Cut the actual FOV value in half.
--
-- getobserver() is a known Celestia method.
-- getfov() and setfov(n) are both known observer methods.

obs = celestia:getobserver()
fovvalue = obs:getfov()
fovvalue = fovvalue / 2
obs:setfov(fovvalue)
```

```
-- Set the speed of the current active observer to 100 ly/sec,
--    displayed in lower left coner of the screen.
--    (100 ly/sec = 100000000 microlightyears per second).
-- Wait 10 seconds and then set the speed of the observer to 0 again.
--
-- getobserver() is a known Celestia method.
-- setspeed() is a known observer method.
-- wait() is a predefined Celestia function.

obs=celestia:getobserver()
obs:setspeed(100000000)
wait(10)
obs:setspeed(0)
```

```
-- Split the actual view in the middle vertically.
-- Wait 10 seconds and then restore the singleview again.
--
-- getobserver() and getobservers() are both known Celestia methods.
-- splitview() and singleview() are both known observer methods.
-- wait() is a predefined Celestia function.

obs = celestia:getobserver()
obs:splitview("V", 0.5)
observers = celestia:getobservers()
wait(10.0)
obs:singleview()
```

Index observer methods:

This chapter contains a list of all available **observer methods**, which can be used on **observer objects**. The mentioned **observer methods** are hyperlinks to the Celestia WIKI books about CELX scripting, where further explanation can be found about the functionality of these **observer methods**.

GOTO Index **object** methods GO BACK TO Index **celestia** methods

Observer methods:

- goto
- goto (table)
- gotolonglat
- gotolocation
- gotodistance
- gotosurface
- center
- centerorbit
- travelling
- cancelgoto
- follow
- synchronous
- chase
- lock
- track
- setposition
- getposition
- getorientation
- setorientation
- rotation
- lookat
- gettime
- getspeed
- setspeed
- getsurface
- setsurface
- getlocationflags
- setlocationflags
- getfov
- setfov
- getframe
- setframe
- splitview
- deleteview
- singleview
- isvalid
- gettrackedobject
- makeactiveview
- orbit

Object objects:

An "**object**" object in CELX does refer to a celestial object like a planet or a star, but it can also be a spacecraft or location. Within a CELX script, the **object methods** need to be prefixed with the obtained **"object" object**, separated with a semicolon.

The object methods can be used on an "object" object, which can be obtained in different ways:

By using the Celestia "find()" method:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Use object method on objectname,
--    or use objectname in some other methods requirering object objects.

objectname = celestia:find( <string> )
objectname:<object method>

-- OR --

celestia:find( <string> ):<object method>
```

By using the Celestia "getselection()" method:

```
-- Get current selected object and store in objectname.
-- Use object method on objectname,
--    or use objectname in some other methods requirering object objects.

objectname = celestia:getselection()
objectname:<object method>

-- OR --

celestia:getselection():<object method>
```

By using the Celestia "getstar(number)" method:

```
-- Get star object with index number <number> in
--    the star database and store in objectname.
-- Use object method on objectname,
--    or use objectname in some other methods requirering object objects.

objectname = celestia:getstar( <number> )
objectname:<object method>

-- OR --

celestia:getstar( <number> ):<object method>
```

By using the Celestia "getdso(number)" method:

```
-- Get Deep Space object with index number <number> in the
--    Deep Space object database and store in objectname.
-- Use object method on objectname,
--    or use objectname in some other methods requirering object objects.

objectname = celestia:getdso( <number> )
objectname:<object method>

-- OR --

celestia:getdso( <number> ):<object method>
```

By using the observer "gettrackedobject()" method.
You have to get an observer instance from Celestia first and then call "gettrackedobject()" on it:

```
-- Get observer instance of the active view and store in obs.
-- Return currently tracked object in observer
--    instance obs and store in objectname.
--    If no object is currently tracked, an empty object,
--    is returned (this is different from nil).
-- Use object method on objectname,
--    or use objectname in some other methods requirering object objects.

obs = celestia:getobserver()
objectname = obs:gettrackedobject()
objectname:<object method>

-- OR --

objectname = celestia:getobserver():gettrackedobject()
objectname:<object method >

-- OR --

celestia:getobserver():gettrackedobject():<object method>
```

By using the frame "getrefobject()" method.
You have to get a frame instance from Celestia first and then call "getrefobject()" on it.
How to obtain a frame instance, is explained further on:

```
-- Get observer instance of the active view and store in obs.
-- Get frame instance from observer and store in frame.
-- Get the referenced object of the frame and store in objectname.
-- Use object method on objectname,
--    or use objectname in some other methods requirering object objects.

obs = celestia:getobserver()
frame = obs:getframe()
objectname = frame:getrefobject()
objectname:<object method>

-- OR --

frame = celestia:getobserver():getframe()
objectname = frame:getrefobject()
objectname:<object method>

-- OR --

objectname = celestia:getobserver():getframe():getrefobject()
objectname:<object method>

-- OR --

celestia:getobserver():getframe():getrefobject():<object method>
```

By using the frame "gettargetobject()" method.
You have to get a frame instance from Celestia first and then call "gettargetobject()" on it.
How to obtain a frame instance, is explained further on:

```
-- Get observer instance of the active view and store in obs.
-- Get frame instance from observer and store in frame.
-- Get the target object for a phaselock frame and store in objectname.
-- Use object method on objectname,
--    or use objectname in some other methods requirering object objects.

obs = celestia:getobserver()
frame = obs:getframe()
objectname = frame:gettargetobject()
objectname:<object method>

-- OR --

frame = celestia:getobserver():getframe()
objectname = frame:gettargetobject()
objectname:<object method>

-- OR --

objectname = celestia:getobserver():getframe():gettargetobject()
objectname:<object method>
-- OR --

celestia:getobserver():getframe():gettargetobject():<object method>
```

Examples:

```
-- Preload the texture of Mercury.
--
-- find() is a known Celestia method.
-- preloadtexture() is a known object method.

mercury = celestia:find("Sol/Mercury"):preloadtexture()
```

```
-- Preload the texture of the current selection.
--
-- getselection() is a known Celestia method.
-- preloadtexture() is a known object method.

currentsel = celestia:getselection()
currentsel:preloadtexture()
```

```
-- Display currently tracked object name.
--
-- getobserver() and print() are both known Celestia methods.
-- gettrackedobject()is a known observer method.
-- name() is a known object method.
-- wait() is a predefined Celestia function.

obs = celestia:getobserver()
tracked_obj = obs:gettrackedobject()
tracked_obj_name = tracked_obj:name()
celestia:print("Tracking "..tracked_obj_name)
wait(5.0)
```

```
-- Find and select Jupiter's Moon Callisto and store in object callisto.
-- Follow Callisto and goto a distance of 10 times the radius of
--    Callisto's center, at longitude 180 degrees (= math.pi radians)
--    and latitude 0 degrees (= 0 radians) in 5 seconds.
--
-- find(), select() and getobserver() are all known Celestia methods.
-- follow() and gotolonglat() are both known observer methods.
-- radius() is a known object method.
-- wait() is a predefined Celestia function.
-- math.pi is a predefined LUA object containing the value π = 3.14159265.

callisto=celestia:find("Sol/Jupiter/Callisto")
celestia:select(callisto)
celestia:getobserver():follow(callisto)
rcallisto=callisto:radius()
rcallisto10=10*rcallisto
celestia:getobserver():gotolonglat(callisto,math.pi,0,rcallisto10,5.0)
wait(10)
```

Index object methods:

This chapter contains a list of all available **object methods**, which can be used on **"object" objects**. The mentioned **object methods** are hyperlinks to the Celestia WIKI books about CELX scripting, where further explanation can be found about the functionality of these **object methods**.

GOTO Index **position** methods

GO BACK TO Index **observer** methods

Object methods:

Position objects:

A **position object** contains the exact coordinates of a point in space. A position is relative to a coordinate system and may need to be converted to or from universal coordinates before further use.

Within a CELX script, the **position methods** need to be prefixed with the obtained **position object**, separated with a semicolon.

The position methods can be used on a "position" object, which can be obtained in different ways:

By using the Celestia "newposition(x,y,z)" method:

```
-- Create new position object from numbers and store in pos.
--    The units of the components of a position
--    object are millionths of a light-year.
-- Use position method on pos,
--    or use pos in some other methods requirering position objects.

pos = celestia:newposition( <xnumber> , <ynumber> , <znumber> )
pos:<position method>

-- OR --

celestia:newposition( <xnumber> , <ynumber> , <znumber> ):<position method>
```

By using the Celestia "newposition(base64)" method:

```
-- Create new position object from URL-style Base64-encoded values
--    (the strings after x=, y=, z= in a CELURL) and store in pos.
-- Use position method on pos,
--    or use pos in some other methods requirering position objects.

pos = celestia:newposition( <xstring> , <ystring> , <zstring> )
pos:<position method>

-- OR --

celestia:newposition( <xstring> , <ystring> , <zstring> ):<position method>
```

By using the observer "getposition()" method.
You have to get an observer instance from Celestia first and then call "getposition()" on it:

```
-- Get observer instance of the active view and store in obs.
-- Return the current position of this observer
--    in Universal coordinates and store in pos.
-- Use position method on pos,
--    or use pos in some other methods requirering position objects.

obs = celestia:getobserver()
pos = obs:getposition()
pos:<position method>

-- OR --

pos = celestia:getobserver():getposition()
pos:<position method>

-- OR --

celestia:getobserver():getposition():<position method>
```

By using the object "getposition(t)" method.
You have to get an object instance from Celestia first and then call "getposition(t)" on it:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return the position of objectname in
--    universal coordinates and store in pos.
--    <time> is the time used to determine
--    the position of objectname.
-- Use position method on pos,
--    or use pos in some other methods requirering position objects.

objectname = celestia:find( <string> )
pos = objectname:getposition( <time> )
pos:<position method>

-- OR --

pos = celestia:find( <string> ):getposition( <time> )
pos:<position method>

-- OR --

celestia:find( <string> ):getposition( <time> ):<position method>
```

By using the position "addvector(v)" method (newpos = pos + vector).
You have to get a position instance from Celestia first and then call "addvector(v)" on it.
How to obtain a vector (v) is explained further on:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return the position of objectname in
--    universal coordinates and store in pos.
--    <time> is the time used to determine
--    the position of objectname.
-- Add <vector> to the position and return the result in newpos.
-- Use position method on newpos,
--    or use newpos in some other methods requirering position objects.

objectname = celestia:find( <string> )
pos = objectname:getposition( <time> )
newpos = pos:addvector( <vector> )
newpos:<position method>

-- OR --

pos = celestia:find( <string> ):getposition( <time> )
newpos = pos:addvector( <vector> )
newpos:<position method>

-- OR --

newpos = celestia:find( <string> ):getposition( <time> ):addvector( <vector> )
newpos:<position method>

-- OR --

celestia:find( <string> ):getposition( <time> ):addvector( <vector> ):<posit
ion method>
```

By using the frame "to(p)" method,
to convert Universal coordinates to frame coordinates.
You have to get a frame instance from Celestia first and then call "to(p)" on it.
How to obtain a frame instance, is explained further on:

```
-- Have a position object in Universal coordinates stored in pos1.
-- Get observer instance of the active view and store in obs.
-- Return frame of reference for this observer and store in frame.
-- Convert pos1 from Universal coordinates to frame
--   coordinates for this observer and store in pos2.
-- Use position method on pos2,
--    or use pos2 in some other methods requirering position objects.

pos1 = <position object in universal coordinates>
obs = celestia:getobserver()
frame = obs:getframe()
pos2 = frame:to(pos1)
pos2:<position method>

-- OR --

pos1 = <position object in universal coordinates>
frame = celestia:getobserver():getframe()
pos2 = frame:to(pos1)
pos2:<position method>

-- OR --

pos1 = <position object in universal coordinates>
pos2 = celestia:getobserver():getframe():to(pos1)
pos2:<position method>

-- OR --

pos1 = <position object in universal coordinates>
celestia:getobserver():getframe():to(pos1):<position method>
```

By using the frame "from(p,t)" method
to convert frame coordinates to Universal coordinates.
You have to get a frame instance from Celestia first and then call "from(p,t)" on it.
How to obtain a frame instance, is explained further on:

```
-- Have a position object in frame coordinates stored in pos1.
-- Get observer instance of the active view and store in obs.
-- Return frame of reference for this observer and store in frame.
-- Convert pos1 from frame coordinates to Universal
--    coordinates at the specified time <time>.
--    If <time> is omitted, the current simulation time is used.
-- Use position method on pos2,
--    or use pos2 in some other methods requirering position objects.

pos1 = <position object in frame coordinates>
obs = celestia:getobserver()
frame = obs:getframe()
pos2 = frame:from(pos1)
pos2:<position method>

-- OR --

pos1 = <position object in frame coordinates>
frame = celestia:getobserver():getframe()
pos2 = frame:from(pos1)
pos2:<position method>

-- OR --

pos1 = <position object in frame coordinates>
pos2 = celestia:getobserver():getframe():from(pos1)
pos2:<position method>

-- OR --

pos1 = <position object in frame coordinates>
celestia:getobserver():getframe():from(pos1):<position method>
```

By using the phase "getposition(t)" method.
You have to get a phase instance from Celestia first and then call "getposition(t)" on it.
How to obtain a phase instance, is explained further on:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Get the active timeline phase at the specified <time>
--    and store in phase. If no time is specified, the current
--    simulation time is used. This method returns nil if the
--    object is not a solar system body, or if the time lies
--    outside the range covered by the objectname's timeline.
-- Return the position in frame coordinates at the specified
--    <time> and store in pos. Times outside the span covered
--    by the phase object are automatically clamped to either
--    the beginning or ending of the span.
-- Use position method on pos,
--    or use pos in some other methods requirering position objects.

objectname = celestia:find( <string> )
phase = objectname:getphase( <time> )
pos = phase:getposition( <time> )
pos:<position method>

-- OR --

phase = celestia:find( <string> ):getphase( <time> )
pos = phase:getposition( <time> )
pos:<position method>

-- OR --

pos = celestia:find( <string> ):getphase( <time> ):getposition( <time> )
pos:<position method>

-- OR --

celestia:find( <string> ):getphase( <time> ):getposition( <time> ):<position
method>
```

It is also possible to add or subtract positions or add/subtract a position with a vector.

Note: Subtracting two positions will produce a vector and not a position.

- position + position → position
- position - position → **vector**
- position + vector → position
- position - vector → position

Examples:

```
-- Determine and print the actual distance from
--    the observer to the center of the Sun in km.
--
-- getobserver(), find(), gettime() and print() are all known Celestia methods.
-- getposition() is a known observer method.
-- getposition(t)is a known object method.
-- distanceto()is a known position method.
-- wait() is a predefined Celestia function.

obs = celestia:getobserver()
posobs = obs:getposition()
sun = celestia:find("Sol")
now = celestia:gettime()
possun = sun:getposition(now)
sundist = posobs:distanceto(possun)
celestia:print("Distance Sun: " .. sundist .. " km", 10, -1, -1, 1, 3)
wait(10.0)
```

```
-- Goto a position, 20,000 km away in the X-direction of the
--    "Universal" coordinate system, from the center of Mars and display
--    the position of Mars in the lower left corner of your screen.
--    Then center Mars in the middle of your screen and follow the planet.
-- uly_to_km is used to convert km to millionth of a lightyear.
--
-- getobserver(), newframe(), find(), select(), gettime(), newposition()
--    and print() are all known Celestia methods.
-- setframe(), goto(), center() and follow() are all known observer methods.
-- getposition(t) is a known object method.
-- getx(), gety() and getz() are all known position methods.
-- wait() is a predefined Celestia function.

uly_to_km = 9460730.4725808
obs = celestia:getobserver()
obs:setframe(celestia:newframe("universal"))
mars = celestia:find("Sol/Mars")
celestia:select(mars)
now = celestia:gettime()
posmars = mars:getposition(now)
posmars_x = posmars:getx()
posmars_y = posmars:gety()
posmars_z = posmars:getz()
celestia:print("Actual position of Mars:\nX = " .. posmars_x .. "\nY = " ..
posmars_y .. "\nZ = " .. posmars_z, 15, -1, -1, 1, 6)
pos = celestia:newposition( (posmars_x + (20000 / uly_to_km)), posmars_y,
posmars_z )
obs:goto(pos,5.0)
wait(5.0)
obs:center(mars,2.0)
obs:follow(mars)
wait(10.0)
```

Index position methods:

This chapter contains a list of all available **position methods**, which can be used on **position objects**. The mentioned **position methods** are hyperlinks to the Celestia WIKI books about CELX scripting, where further explanation can be found about the functionality of these **position methods**.

GOTO Index **vector** methods GO BACK TO Index **object** methods

Position methods:

Vector objects:

A **vector object** is a 3-tuple of double precision floating point values, which represent a geometric object that has both a length and direction [X, Y, Z] in a 3 dimensional coordinate system.

Within a CELX script, the **vector methods** need to be prefixed with the obtained **vector object**, separated with a semicolon.

The vector methods can be used on a "vector" object, which can be obtained in different ways:

By using the Celestia "newvector(x,y,z)" method:

```
-- Create a new vector and store in vec.
--    <xnumber>: The x component of the new vector.
--    <ynumber>: The y component of the new vector.
--    <znumber>: The z component of the new vector.
-- Use vector method on vec,
--    or use vec in some other methods requirering vector objects.

vec = celestia:newvector( <xnumber> , <ynumber> , <znumber> )
vec:<vector method>

-- OR --

celestia:newvector( <xnumber> , <ynumber> , <znumber> ):<vector method>
```

By subtracting position objects.

You have to get position instances from Celestia first and then subtract them:

```
-- Find object with name <string1> and store in objectname1.
--    <string1> is the name of an object, as known by Celestia.
-- Find object with name <string2> and store in objectname2.
--    <string2> is the name of an object, as known by Celestia.
-- Return the position of objectname1 in
--    Universal coordinates and store in pos1.
--    <time> is the time used to
--    determine the position of objectname1.
-- Return the position of objectname2 in
--    Universal coordinates and store in pos2.
--    <time> is the time used to
--    determine the position of objectname2.
-- Subtract pos1 from pos2 and store in vec.
-- Use vector method on vec,
--    or use vec in some other methods requirering vector objects.

objectname1 = celestia:find( <string1> )
objectname2 = celestia:find( <string2> )
pos1 = objectname1:getposition( <time> )
pos2 = objectname2:getposition( <time> )
vec = pos2 – pos1
vec:<vector method>

-- OR --

pos1 = celestia:find( <string1> ):getposition( <time> )
pos2 = celestia:find( <string2> ):getposition( <time> )
vec = pos2 – pos1
vec:<vector method>

-- OR --

vec = celestia:find( <string2> ):getposition( <time> ) - celestia:find( <str
ing1> ):getposition( <time> )
vec:<vector method>

-- OR --

(celestia:find( <string2> ):getposition( <time> ) - celestia:find( <string1>
 ):getposition( <time> )):<vector method>
```

By using the position "vectorto(p)" method.

You have to get position instances from Celestia first and then call "vectorto(v)" on it:

```
-- Find object with name <string1> and store in objectname1.
--    <string1> is the name of an object, as known by Celestia.
-- Find object with name <string2> and store in objectname2.
--    <string2> is the name of an object, as known by Celestia.
-- Return the position of objectname1 in
--    Universal coordinates and store in pos1.
--    <time> is the time used to
--    determine the position of objectname1.
-- Return the position of objectname2 in
--    Universal coordinates and store in pos2.
--    <time> is the time used to
--    determine the position of objectname2.
-- Return the vector pointing from pos1 to pos2 and store in vec.
-- Use vector method on vec,
--    or use vec in some other methods requirering vector objects.

objectname1 = celestia:find( <string1> )
objectname2 = celestia:find( <string2> )
pos1 = objectname1:getposition( <time> )
pos2 = objectname2:getposition( <time> )
vec = pos1:vectorto(pos2)
vec:<vector method>

-- OR --

pos1 = celestia:find( <string1> ):getposition( <time> )
pos2 = celestia:find( <string2> ):getposition( <time> )
vec = pos1:vectorto(pos2)
vec:<vector method>

-- OR --

vec = celestia:find( <string1> ):getposition( <time> ):vectorto(celestia:fin
d( <string2> ):getposition( <time> ))
vec:<vector method>

-- OR --

celestia:find( <string1> ):getposition( <time> ):vectorto(celestia:find( <st
ring2> ):getposition( <time> )):<vector method>
```

Note: This method gives the same result as subtracting position objects (previous way to obtain a vector)

By using the vector "normalize()" method.

You have to get a vector instance from Celestia first and then call "normalize()" on it:

```
-- Find object with name <string1> and store in objectname1.
--    <string1> is the name of an object, as known by Celestia.
-- Find object with name <string2> and store in objectname2.
--    <string2> is the name of an object, as known by Celestia.
-- Return the position of objectname1 in
--    Universal coordinates and store in pos1.
--    <time> is the time used to
--    determine the position of objectname1.
-- Return the position of objectname2 in
--    Universal coordinates and store in pos2.
--    <time> is the time used to
--    determine the position of objectname2.
-- Return the vector pointing from pos1 to pos2 and store in vec1.
-- Return a normalized vector of vec1 and store in vec2.
--    This vector has the same direction, but with length 1.
--    It does not modify the original vector. Normalizing a
--    zero-length vector causes division by zero.
--    The result will be a vector of NaNs.
-- Use vector method on vec2,
--    or use vec2 in some other methods requirering vector objects.

objectname1 = celestia:find( <string1> )
objectname2 = celestia:find( <string2> )
pos1 = objectname1:getposition( <time> )
pos2 = objectname2:getposition( <time> )
vec1 = pos1:vectorto(pos2)
vec2 = vec1:normalize()
vec2:<vector method>

-- OR --

pos1 = celestia:find( <string1> ):getposition( <time> )
pos2 = celestia:find( <string2> ):getposition( <time> )
vec1 = pos1:vectorto(pos2)
vec2 = vec1:normalize()
vec2:<vector method>

-- OR --

vec1 = celestia:find( <string1> ):getposition( <time> ):vectorto(celestia:fi
nd( <string2> ):getposition( <time> ))
vec:<vector method>
vec2 = vec1:normalize()
vec2:<vector method>

-- OR --

vec2 = celestia:find( <string1> ):getposition( <time> ):vectorto(celestia:fi
nd( <string2> ):getposition( <time> )):normalize()
vec2:<vector method>

-- OR --

celestia:find( <string1> ):getposition( <time> ):vectorto(celestia:find( <st
ring2> ):getposition( <time> )):normalize():<vector method>
```

By using the rotation "imag()" method.
You have to get a rotation object from Celestia first and then call "imag()" on it.
How to obtain a rotation object, is explained further on:

```
-- Have a rotation object stored in rot.
-- Return the x, y and z values of rotation rot as a vector and store in vec.
-- Use vector method on vec,
--    or use vec in some other methods requirering vector objects.

rot = <rotation object>
vec = rot:imag()
vec:<vector method>

-- OR --

rot = <rotation object>
rot:imag():<vector method>
```

By using the rotation "transform()" method.
You have to get another vector object from Celestia first according one of the above methods in this vector objects section.
You have to get a rotation object from Celestia first and then call "transform()" on it.
How to obtain a rotation object, is explained further on:

```
-- Have another vector object stored in vec1.
-- Have a rotation object stored in rot.
-- Transform vector vec1 by applying rotation rot to vec1 and store in vec2.
-- Use vector method on vec2,
--    or use vec2 in some other methods requirering vector objects.

vec1 = <vector object>
rot = <rotation object>
vec2=rot:transform(vec1)
vec2:<vector method>

-- OR --

vec1 = <vector object>
rot = <rotation object>
rot:transform(vec1):<vector method>
```

It is also possible to add or subtract vectors, multiply a vector with a number or to determine the dot product and cross product of vectors:

- Addition: vector + vector → vector
- Subtraction: vector - vector → vector
- Scalar multiplication: number * vector → vector
- Scalar multiplication: vector * number → vector
- Dot product: vector * vector → number
  ➔ *vector v * vector w = v.x * w.x + v.y * w.y + v.z * w.z*
- Cross product: vector ^ vector → vector
  ➔ *new vector is perpendicular (according the right-hand rule) to the plane containing the two input vectors*
  ➔ *vector v ^ vector w = - vector w ^ vector v*

Examples:

```
-- Make a vector, pointing in the Y-direction with length 1.
--
-- newvector()is a known Celestia method.

up_vector = celestia:newvector(0,1,0)
```

```
-- Get a vector giving the heliocentric position of the Earth
--    and display the X-, Y- and Z-components of that vector
--    in the lower left corner of your screen.
--
-- find(), gettime() and print are all known Celestia methods.
-- getposition(t)is a known object method.
-- vectorto()is a known position method.
-- getx(), gety() and getz() are all known vector methods.
-- wait() is a predefined Celestia function.

now = celestia:gettime()
sunpos = celestia:find("Sol"):getposition(now)
earthpos = celestia:find("Sol/Earth"):getposition(now)
heliovector = sunpos:vectorto(earthpos)
heliovector_x = heliovector:getx()
heliovector_y = heliovector:gety()
heliovector_z = heliovector:getz()
celestia:print("Heliocentric position of Earth:\nX = " .. heliovector_x ..
"\nY = " .. heliovector_y .. "\nZ = " .. heliovector_z, 10, -1, -1, 1, 6)
wait(10.0)
```

```
-- Determine and display the actual distance from the obserber to Neptune.
-- uly_to_km is used to convert millionth of a lightyear to km.
--
-- gettime(),find(), getobserver() and print() are all known Celestia methods.
-- getposition()is a known observer method.
-- getposition(t)is a known object method.
-- length is a known vector method.
-- wait() is a predefined Celestia function.

uly_to_km = 9460730.4725808
now = celestia:gettime()
neptunepos = celestia:find("Sol/Neptune"):getposition(now)
obs = celestia:getobserver()
obspos = obs:getposition()
vectortoneptune = obspos - neptunepos
distance = vectortoneptune:length() * uly_to_km
celestia:print("The center of Neptune is " .. distance .. " km away", 9, -1, -1, 1,
3)
wait(9.0)
```

Index vector methods:

This chapter contains a list of all available **vector methods**, which can be used on **vector objects**. The mentioned **vector methods** are hyperlinks to the Celestia WIKI books about CELX scripting, where further explanation can be found about the functionality of these **vector methods**.

GOTO Index **rotation** methods GO BACK TO Index **position** methods

Vector methods:

- getx
- gety
- getz
- normalize
- length

Rotation objects:

A CELX **rotation object** contains the characteristics of an orientation or rotation.

Within a CELX script, the **rotation methods** need to be prefixed with the obtained **rotation object**, separated with a semicolon.

The rotation methods can be used on a "rotation" object, which can be obtained in different ways:

By using the Celestia "newrotation(v,a)" method:

```
-- Create new rotation (i.e. a quaternion) of an <angle> about
--    the specified <axis> and store in rot.
--    <axis> is a vector, describing the axis of this rotation.
--    <angle> is a number, defined in radians (NOT degrees !!!).
-- Use rotation method on rot,
--    or use rot in some other methods requirering rotation objects.

rot = celestia:newrotation( <vector> , <angle> )
rot:<rotation method>

-- OR --

celestia:newrotation( <vector> , <angle> ):<rotation method>
```

By using the Celestia "newrotation(ow,ox,oy,oz)" method:

```
-- Create new rotation (i.e. a quaternion) from four scalar
--    values (the strings after ow=, ox=, oy=, oz= in a CELURL),
--    and store in rot.
-- Use rotation method on rot,
--    or use rot in some other methods requirering rotation objects.

rot = celestia:newrotation( <ownum> , <oxnum> , <oynum> , <oznum> )
rot:<rotation method>

-- OR --

celestia:newrotation( <ownum> , <oxnum> , <oynum> , <oznum> ):<rotation method>
```

By using the observer "getorientation()" method.

You have to get an observer instance from Celestia first and then call "getorientation()" on it:

```
-- Get observer instance of the active view and store in obs.
-- Return the current orientation of this observer and store in rot.
-- Use rotation method on rot,
--    or use rot in some other methods requirering rotation objects.

obs = celestia:getobserver()
rot = obs:getorientation()
rot:<rotation method>

-- OR --

rot = celestia:getobserver():getorientation()
rot:<rotation method>

-- OR --

celestia:getobserver():getorientation():<rotation method>
```

By using the position "orientationto(p,v)" method.

You have to get position instances from Celestia first and then call "orientationto(p,v)" on it:

```
-- Find object with name <string1> and store in objectname1.
--    <string1> is the name of an object, as known by Celestia.
-- Find object with name <string2> and store in objectname2.
--    <string2> is the name of an object, as known by Celestia.
-- Return the position of objectname1 in
--    Universal coordinates and store in pos1.
--    <time> is the time used to
--    determine the position of objectname1.
-- Return the position of objectname2 in
--    Universal coordinates and store in pos2.
--    <time> is the time used to
--    determine the position of objectname2.
-- Create a vector [x,y,z] from <xnum>, <ynum>, <znum> and store in upvec.
-- Return a rotation object which can be used to orient a viewer
--    to point from pos1 toward pos2 to focus on and store in rot.
-- Use rotation method on rot,
--    or use rot in some other methods requirering rotation objects.

objectname1 = celestia:find( <string1> )
objectname2 = celestia:find( <string2> )
pos1 = objectname1:getposition( <time> )
pos2 = objectname2:getposition( <time> )
upvec = celestia:newvector( <xnum> , <ynum> , <znum> )
rot = pos1:orientationto(pos2,upvec)
rot:<rotation method>

-- OR --

pos1 = celestia:find( <string1> ):getposition( <time> )
pos2 = celestia:find( <string2> ):getposition( <time> )
upvec = celestia:newvector( <xnum> , <ynum> , <znum> )
rot = pos1:orientationto(pos2,upvec)
rot:<rotation method>

-- OR --

rot = celestia:find( <string1> ):getposition( <time> ):orientationto(celesti
a:find( <string2> ):getposition( <time> ), upvec = celestia:newvector( <xnum
> , <ynum> , <znum> ))
rot:<rotation method>

-- OR --

celestia:find( <string1> ):getposition( <time> ):orientationto(celestia:find
( <string2> ):getposition( <time> ), upvec = celestia:newvector( <xnum> , <y
num> , <znum> )):<rotation method>
```

By using the frame "to(rotation)" method.

You have to get a frame instance from Celestia first and then call "to(rotation)" on it.
How to obtain a frame instance, is explained further on:

```
-- Have a rotation object in a Universal frame, stored in rot1.
-- Get observer instance of the active view and store in obs.
-- Return frame of reference for this observer and store in frame.
-- Convert rot1 from the Universal frame to this obtained
--    frame at the specified <time> and store in rot2.
--    If <time> is omitted, the current simulation time is used.
-- Use rotation method on rot2,
--    or use rot2 in some other methods requirering rotation objects.

rot1 = <rotation object in Universal frame>
obs = celestia:getobserver()
frame = obs:getframe()
rot2 = frame:to(rot1, <time> )
rot2:<rotation method>

-- OR --

rot1 = <rotation object in Universal frame>
frame = celestia:getobserver():getframe()
rot2 = frame:to(rot1, <time> )
rot2:<rotation method>

-- OR --

rot1 = <rotation object in Universal frame>
rot2 = celestia:getobserver():getframe():to(rot1, <time> )
rot2:<rotation method>

-- OR --

rot1 = <rotation object in Universal frame>
celestia:getobserver():getframe():to(rot1, <time> ):<rotation method>
```

By using the frame "from(rotation)" method.
You have to get a frame instance from Celestia first and then call "from(rotation)" on it.
How to obtain a frame instance, is explained further on:

```
-- Have a rotation object from a frame stored in rot1.
-- Get observer instance of the active view and store in obs.
-- Return frame of reference for this observer and store in frame.
-- Convert rot1 in this frame to the
--    Universal frame and store in rot2.
-- Use rotation method on rot2,
--    or use rot2 in some other methods requirering rotation objects.

rot1 = <rotation object in a frame>
obs = celestia:getobserver()
frame = obs:getframe()
rot2 = frame:from(rot1)
rot2:<rotation method>

-- OR --

rot1 = <rotation object in a frame>
frame = celestia:getobserver():getframe()
rot2 = frame:from(rot1)
rot2:<rotation method>

-- OR --

rot1 = <rotation object in a frame>
rot2 = celestia:getobserver():getframe():from(rot1)
rot2:<rotation method>

-- OR --

rot1 = <rotation object in Universal frame>
celestia:getobserver():getframe():from(rot1):<rotation method>
```

By using the phase "getorientation(t)" method.

You have to get a phase instance from Celestia first and then call "getorientation(t)" on it.
How to obtain a phase instance, is explained further on:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Get the active timeline phase of objectname
--    at the specified <time> and store in phase.
--    If no <time> is specified, the current simulation time is used.
--    This method returns nil if the object is not a
--    Solar system body, or if the time lies outside
--    the range covered by the object's timeline.
-- Return the orientation in frame coordinates
--    at the specified time and store in rot.
--    Times outside the span covered by the phase are automatically
--    clamped to either the beginning or ending of the span.
-- Use rotation method on rot,
--    or use rot in some other methods requirering rotation objects.

objectname = celestia:find( <string> )
phase = objectname:getphase( <time> )
rot = phase:getorientation( <time> )
rot:<rotation method>

-- OR --

phase = celestia:find( <string> ):getphase( <time> )
rot = phase:getorientation( <time> )
rot:<rotation method>

-- OR --

rot = celestia:find( <string> ):getphase( <time> ):getorientation( <time> )
rot:<rotation method>

-- OR --

celestia:find( <string> ):getphase( <time> ):getorientation( <time> ):<rotation method>
```

By using the rotation "slerp()" method.

You have to get two other rotation objects from Celestia first according one of the above methods in this rotation objects section:

```
-- Have a rotation object stored in rot1.
-- Have another rotation object stored in rot2.
-- Return an interpolation of rot1 and rot2, using a
--    <number> between 0 and 1 to indicate how much of
--    each rotation should be used, and store the result in rot3.
-- Use rotation method on rot3,
--    or use rot3 in some other methods requirering rotation objects.

rot1 = <rotation object>
rot2 = <rotation object>
rot3=rot1:slerp(rot2, <number> )
rot3:<rotation method>

-- OR --

rot1 = <rotation object>
rot2 = <rotation object>
rot1:slerp(rot2, <number> ):<rotation method>
```

Examples:

```
-- Change the current camera view by 180 degrees (like a rear-view mirror) by
--    1. Define the Up vector for the rotation (Y-axis).
--    2. Define a rotation object "lookback" with an angle of 180 degrees
--       (= math.pi radians), along the specified Y-axis in the UP vector.
--    3. Rotate the observer according the specified rotation.
--
-- newvector(), newrotation and getobserver() are all known Celestia methods.
-- rotate()is a known observer method.
-- math.pi is a predefined LUA object containing the value π = 3.14159265.

up_v=celestia:newvector(0,1,0)
lookback=celestia:newrotation(up_v,math.pi)
celestia:getobserver():rotate(lookback)
```

```
-- Rotate the current camera view by 360 degrees during about 10 seconds.
--
-- newvector(), newrotation and getobserver() are all known Celestia methods.
-- rotate()is a known observer method.
-- math.pi is a predefined LUA object containing the value π = 3.14159265.

up_v=celestia:newvector(0,1,0)
obs_rotation=celestia:newrotation(up_v,2*math.pi/500)
for i = 1, 500 do
   celestia:getobserver():rotate(i * obs_rotation)
   wait(10/500)
end
```

Index rotation methods:

This chapter contains a list of all available **rotation methods**, which can be used on **rotation objects**. The mentioned **rotation methods** does NOT contain hyperlinks to the Celestia WIKI books about CELX scripting yet, because further explanation about these **rotation methods** are NOT documented in the WIKI books yet.

GOTO Index **frame** methods GO BACK TO Index **vector** methods

Rotation methods:

Frame objects:

A reference **frame** is an origin and set of axes which define the coordinate system used for a body's trajectory and orientation. The origin is some other body defined in a catalog file. There are a number of ways to set the coordinate system axes.

Coordinate systems are well documented in the available Celestia .Cel Scripting Guide v1-0g by Don Goyette, in the chapter: “Coordinate Systems”.

The reference **frames** used for a body's trajectory and its orientation do not have to be the same. This is useful in some situations. For example, the orbit of a satellite may be given in a geocentric equatorial coordinate system, while the attitude is given in a local vertical-local horizontal system.

Within a CELX script, the **frame methods** need to be prefixed with the obtained **frame object**, separated with a semicolon.

The frame methods can be used on a “frame” object, which can be obtained in different ways:

By using the Celestia “newframe(c,or,ot)” method:

```
-- Create new reference frame and store in frame.
--    <coordsysname> is a string, describing the type
--    of frame, which can be one of the following:
--       → universal
--       → ecliptic
--       → equatorial
--       → planetographic
--       → observer (deprecated)
--       → lock
--       → chase
--       → bodyfixed
--    <objectref> [optional]: The reference object for the new frame.
--       Not needed for type "universal".
--    <objecttar> [optional]: The target object for this frame.
--       Only needed for frames of type "lock".
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

frame = celestia:newframe( <coordsysname> , <objectref> , <objecttar> )
frame:<frame method>

-- OR --

celestia:newframe( <coordsysname> , <objectref> , <objecttar> ):<frame method>
```

By using the observer "getframe()" method.
You have to get an observer instance from Celestia first and then call "getframe()" on it:

```
-- Get observer instance of the active view and store in obs.
-- Return frame of reference for this observer and store in frame.
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

obs = celestia:getobserver()
frame = obs:getframe()
frame:<frame method>

-- OR --

frame = celestia:getobserver():getframe()
frame:<frame method>

-- OR --

celestia:getobserver():getframe():<frame method>
```

By using the object "bodyfixedframe()" method.
You have to get an object instance from Celestia first and then call "bodyfixedframe()" on it:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return the body-fixed frame for objectname and store in frame.
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

objectname = celestia:find( <string> )
frame = objectname:bodyfixedframe()
frame:<frame method>

-- OR --

frame = celestia:find( <string> ):bodyfixedframe()
frame:<frame method>

-- OR --

celestia:find( <string> ):bodyfixedframe():<frame method>
```

By using the object "equatorialframe()" method.

You have to get an object instance from Celestia first and then call "equatorialframe()" on it:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return the equatorial frame for objectname and store in frame.
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

objectname = celestia:find( <string> )
frame = objectname:equatorialframe()
frame:<frame method>

-- OR --

frame = celestia:find( <string> ): equatorialframe()
frame:<frame method>

-- OR --

celestia:find( <string> ): equatorialframe():<frame method>
```

By using the object "orbitframe()" method.

You have to get an object instance from Celestia first and then call "orbitframe()" on it:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return the orbit frame for objectname and store in frame.
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

objectname = celestia:find( <string> )
frame = objectname:orbitframe()
frame:<frame method>

-- OR --

frame = celestia:find( <string> ):orbitframe()
frame:<frame method>

-- OR --

celestia:find( <string> ):orbitframe():<frame method>
```

By using the object "bodyframe()" method.

You have to get an object instance from Celestia first and then call "bodyframe()" on it:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return the body frame for objectname and store in frame.
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

objectname = celestia:find( <string> )
frame = objectname:bodyframe()
frame:<frame method>

-- OR --

frame = celestia:find( <string> ):bodyframe()
frame:<frame method>

-- OR --

celestia:find( <string> ):bodyframe():<frame method>
```

By using the phase "orbitframe()" method.

You have to get a phase instance from Celestia first and then call "orbitframe()" on it.
How to obtain a phase instance, is explained further on:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Get the active timeline phase of objectname
--    at the specified <time> and store in phase.
--    If no <time> is specified, the current simulation time is used.
--    This method returns nil if the object is not a
--    Solar system body, or if the time lies outside
--    the range covered by the object's timeline.
-- Return the orbit frame for this timeline phase and store in frame.
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

objectname = celestia:find( <string> )
phase = objectname:getphase( <time> )
frame = phase:orbitframe()
frame:<frame method>

-- OR --

phase = celestia:find( <string> ):getphase( <time> )
frame = phase:orbitframe()
frame:<frame method>

-- OR --

frame = celestia:find( <string> ):getphase( <time> ):orbitframe()
frame:<frame method>

-- OR --

celestia:find( <string> ):getphase( <time> ):orbitframe():<frame method>
```

By using the phase "bodyframe()" method.

You have to get a phase instance from Celestia first and then call "bodyframe()" on it.

How to obtain a phase instance, is explained further on:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Get the active timeline phase of objectname
--    at the specified <time> and store in phase.
--    If no <time> is specified, the current simulation time is used.
--    This method returns nil if the object is not a
--    Solar system body, or if the time lies outside
--    the range covered by the object's timeline.
-- Return the body frame for this timeline phase and store in frame.
-- Use frame method on frame,
--    or use frame in the observer method setframe(frame).

objectname = celestia:find( <string> )
phase = objectname:getphase( <time> )
frame = phase:bodyframe()
frame:<frame method>

-- OR --

phase = celestia:find( <string> ):getphase( <time> )
frame = phase:bodyframe()
frame:<frame method>

-- OR --

frame = celestia:find( <string> ):getphase( <time> ):bodyframe()
frame:<frame method>

-- OR --

celestia:find( <string> ):getphase( <time> ):bodyframe():<frame method>
```

Examples:

```
-- Set the coordinate system of the frame of reference to lock,
--    with Earth as reference object and the Sun as target object.
--
-- find(), select(), newframe() and getobserver are all known Celestia methods.
-- setframe() is a known observer method.

objectname_ref = celestia:find("Sol/Earth")
celestia:select(objectname_ref)
objectname_tar = celestia:find("Sol")
frame = celestia:newframe("lock", objectname_ref, objectname_tar)
obs = celestia:getobserver()
obs:setframe(frame)
```

```
-- Obtain the actual observer frame of reference and
--    determine if there's an object followed.
-- If no followed object, use the current selection.
-- Set the orientation of the observer to look at the followed or
--    selected object and go to that object.

obs=celestia:getobserver()
obsframe=obs:getframe()
center=obsframe:getrefobject()
if not center then
   center = celestia:getselection()
end
up=celestia:newvector(0, 1, 0)
obs:lookat(center:getposition(), up)
obs:goto(center,5.0)
wait(5.0)
```

Index frame methods:

This chapter contains a list of all available **frame methods**, which can be used on **frame objects**. The mentioned **frame methods** are hyperlinks to the Celestia WIKI books about CELX scripting, where further explanation can be found about the functionality of these **frame methods**.

GOTO Index **phase** methods GO BACK TO Index **rotation** methods

Frame methods:

Phase objects:

The timeline of an object is subdivided into one or more **phases**. Each **phase** has its own trajectory, rotation model, orbit frame, and body frame.

For example, the timeline for the Huygens probe might have three **phases**:

- Attached to Cassini:
  body fixed frame of Cassini, trajectory is just a fixed position.
- Free flight after separation:
  Saturn-centered frame, trajectory from an xyz file or SPICE.
- On the surface of Titan:
  Titan-fixed frame, trajectory is a fixed position

Within a CELX script, the **phase methods** need to be prefixed with the obtained **phase object**, separated with a semicolon.

An object's phase can be obtained in one of two ways:

By using the object "getphase(t)" method.

You have to get an object instance from Celestia first and then call "bodyfixedframe()" on it:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return the active timeline phase of objectname
--    at the specified <time> and store in phase.
--    If no <time> is specified, the current simulation time is used.
--    This method returns nil if the object is not a Solar system
--    body, or if the time lies outside the range covered by the
--    object's timeline.
-- Use phase method on phase.

objectname = celestia:find( <string> )
phase = objectname:getphase()
phase:<phase method>

-- OR –

phase = celestia:find( <string> ):getphase()
phase:<phase method>

-- OR –

celestia:find( <string> ):getphase():<phase method>
```

By using the object "getphases()" method.

You have to get an object instance from Celestia first and then call "bodyfixedframe()" on it:

```
-- Find object with name <string> and store in objectname.
--    <string> is the name of an object, as known by Celestia.
-- Return an iterator over all the phases in
--    an object's timeline and store in phases.
--    Only Solar system bodies have a timeline. For all other
--    object types, this method will return an empty iterator.
--    The phases in a timeline are always sorted from earliest
--    to latest, and always cover a continuous span of time.
-- Use phase method on phases.

objectname = celestia:find( <string> )
phases = objectname:getphases()
phase[1]:<phase method>
phase[2]:<phase method>
...

-- OR –

phases = celestia:find( <string> ):getphases()
phase[1]:<phase method>
phase[2]:<phase method>
...

-- OR –

celestia:find( <string> ):getphases():<phase method>
```

Examples:

```
-- Find object Cassini in our Solar system.
-- Convert midnight January 1, 2000 UTC to TDB date/time:
--    UTC is used in the Celestia's Set Time dialog and it's also
--       the time displayed in the upper right of the screen.
--    TDB = Barycentric Dynamical Time.
--       When using scripts for Celestia, the time scale is TDB !!!
-- Get the timeline phase for Cassini at specified date/time.
--
-- find() and utctotdb() are both known Celestia methods.
-- getphase() is a known object method.

cassini = celestia:find("Sol/Cassini")
tdb = celestia:utctotdb(2000, 1, 1, 0, 0, 0)
phase = cassini:getphase(tdb)
```

```
-- Copy all of the phases of a selected object into the array timeline.
--
-- getselection() is a known Celestia method.
-- phases() is a known object method.

timeline = { }
count = 0
for phase in celestia:getselection():phases() do
    count = count + 1
    timeline[count] = phase
end
```

Index phase methods:

This chapter contains a list of all available **phase methods**, which can be used on **phase objects**. The mentioned **phase methods** are hyperlinks to the Celestia WIKI books about CELX scripting, where further explanation can be found about the functionality of these **phase methods**.

GOTO Index **celestia** methods GO BACK TO Index **frame** methods

Phase methods:

CELscript objects

Within CELX scripting it is possible to create a **CELscript object** from a <*string*> containing a valid CEL script, by using the **"celestia:createcelscript()" method**. If the <*string*> doesn't contain a valid CEL script, this **method** will cause an error.

A **CELscript object** can be used to execute a CEL script embedded in a CELX script. This is done by passing the CEL script in a <*string*> to this **method** and creating a **CELscript object**. Then the **celscript:tick() method** for the **CELscript object** is called repeatedly until it returns "false", indicating that the CEL script has terminated.

NOTE: LUA supports a syntax for long strings using double brackets which is useful here. Using this you can in nearly all cases simply copy and paste complete CEL scripts without modification into the CELX script.

NOTE: When using CEL script parts within your CELX script, the **PAUSE** function by pressing the [**Spacebar**] key will not work completely correct. Although the scene you're looking at will pause directly, the CEL script parts will continue to run in the background, resulting in a part of the script being visually skipped when you press the [**Spacebar**] key again to continue the script.

First define the following function at the beginning of your script:

```
function CEL(source)
   local script = celestia:createcelscript(source)
   while script:tick() do
      wait(0)
   end
end
```

Within your CELX script you can now use the CEL script commands as follows:

```
<... other CELX script code ...>
CEL([[{ <string> }]])
<... other CELX script code ...>
```

Examples:

```
function CEL(source)
   local script = celestia:createcelscript(source)
   while script:tick() do
      wait(0)
   end
end

CEL([[{
   cancel   { }
   select   { object "Sol/Earth" }
   goto     { time 3 distance 7 upframe "universal" }
   wait     { duration 5 }
   track    { }
   timerate { rate 1000 }
}]])
```

Differences between CEL and CELX scripting

- The extension of CEL and CELX scripts differ from each other:
  - CEL script: .CEL
  - CELX script: .CELX
- More words and more commands make CELX scripts look more complex then CEL scripts, but it also gives your scripts more possibilities, including the usage of LUA logic (like buiding loops, make decisions on what to do, calculations, etc.), and interactive keyboard input.
  It is the intention of this "CELX Scripting for Celestia (Sub-title: How to migrate from CEL scripting)" document to help you with the more complex CELX equivalents of the CEL commands, so you can use these equivalents in your own script. Just go to the Table of Contents of this document, click on the CEL: <*command*> index and you'll find the equivalent, including examples, in the specific chapter. You can copy the equivalents in your own script and modify them according your own wishes. After a while you'll become more familiar with CELX scripting and copying or typing a sequence of CELX methods and objects will become easier each time.
- Keyboard interaction is possible in CELX, but not in CEL scripting. Within CELX you can use the callback for keyboard input, which must have the name "**celestia_keyboard_callback**". After a script activates the handling keyboard-input by calling "**celestia:requestkeyboard(true)**", any keypress will result in a call to a method with this name. The further usage of keyboard interaction is explained in chapter "Handy CELX functions".
- Angles in CELX are defined in radians instead of degrees in CEL. 180 degrees (half a circle) is the same as π radians = 3.14159265 radians. You can use the Lua **math.rad()** and **math.deg()** functions to convert degrees to radians and vise versa:
  - radians = math.rad(<*number:degrees*>)
  - or you can convert yourself: radians = (<*number:degrees*> / 180 * 3.14159265)
  - or use Lua math.pi: radians = (<*number:degrees*> / 180 * math.pi)

  - degrees = math.rad(<*number:radians*>)
  - or you can convert yourself: degrees = (<*number:radians*> * 180 / 3.14159265)
  - or use Lua math.pi: degrees = (<*number:radians*> * 180 / math.pi)

- The units of the components of a position object in CELX are **millionths of a light-year**. When you have position components defined in km (CEL scripting) or miles, you first have to convert those components to millionths of a light year.
  Therefore, you can use a constant, which must be initialized first within your CELX script:
  - From km to millionths of a light year, use constant: uly_to_km = 9460730.4725808.
  - From miles to millionths of a light year, use constant: uly_to_mls = 5912956.5453630.

  Next you can convert km or miles to millionths of a light year as follows:
  - millionths_of_a_light_year = <*number:km*> / uly_to_km
  - millionths_of_a_light_year = <*number:miles*> / uly_to_mls

- Be aware that some default parameter values have changed in CELX compared with the equivalent defaults in CEL scripting. The old and new defaults are mentioned throughout the sub-chapters of chapter "CEL to CELX equivalents".
- Time in CELX is defined in TDB (Barycentric Dynamical Time) instead of UTC or Juliandate, because UTC includes "leap seconds" in order to stay aligned with the Earth's varying rotation. Leap seconds happen essentially randomly, when they are needed. Although Celestia does incorporate a table of leap second times, its use of UTC causes problems when used with ephemerides which are defined using TDB. Starting with v1.5.0, although it still displays UTC on the screen, Celestia uses the TDB time scale internally for everything else. As a result, Celestia places objects much more accurately than before.
    - To convert from Juliandate and/or UTC to TDB is explained in sub-chapter "CEL: time".
- Transparency of markers can be **true** or **false** in CELX (starting from Celestia v1.6.0). In CEL scripting there's no possibility to enable or disable the transparency of markers. The Celestia releases v1.5.1 and older always displayed the markers, but from Celestia v1.6.0 (or Celestia-ED v1.5.1), the default marker transparency is **true**, resulting in disappearing markers in case of Occultations, or partially show markers on the surface of celestial objects. To overcome this, the usage of the **false** parameter in the CELX: **"object:mark()"** method is necessary, which also makes CELX scripting necessary, instead of CEL scripting! The further usage of the CELX: **"object:mark()"** method is explained in sub-chapter "CEL: mark".
- Sometimes, timing can be more sensitive in CELX scripting then it is in CEL scripting, especially when lots of LUA commands and celestia (related) methods are used within a short period of time with a high timerate value (time has been speeded up). Solving those timing issues on your own computer may not always give the same result on other computers with different speeds. In those cases, the usage of a CEL equivalent part within your CELX script, may give a more consistent result.
- It is possible to insert CEL script parts within a CELX script (see also chapter CELscript objects), by defining the following function at the beginning of your script:

```
function CEL(source)
   local script = celestia:createcelscript(source)
   while script:tick() do
      wait(0)
   end
end

<... other CELX script code ...>
CEL([[{ <one or more lines with CEL script commands> }]])
<... other CELX script code ...>
```

- Scripts can be temporary paused, by pressing the [**Spacebar**] key. Pressing the [**Spacebar**] key again will result in the continuation of the script.
  Note: When using CEL script parts and functions within your CELX script, this **pause** function will not work completely correct. Although the scene you're looking at will pause directly, the CEL script parts and functions will continue to run in the background, resulting in a part of the script being visually skipped when you press the [**Spacebar**] key again to continue the script.

CELX Script Initialization

When you program a CEL or CELX script in Celestia, many Celestia options can be set, that differ from the the standard settings of a user. However, when a CEL script is done, there is no quick reset command to return the settings to the way they were before.

Within CELX scripting we can reset the settings to the way they were before, by using the Cleanup callback function. Whenever a LUA script terminates (because it ended, the user terminated it or an error occured), Celestia will try to execute a function with the name "**celestia_cleanup_callback**". So this function can be used to perform any cleanup actions, notably restoring settings to the values in use before the script started.

The use of this **cleanup** function is essential to program your CELX scripts **user friendly** !!!

NOTE: A callback function is executed as a result of external events. Callbacks are limited in what you can do: the callback must complete within 1 second, or it will be terminated, and you can't use wait() in a callback.

The following example code can standard be inserted at the beginning of any CELX script to restore the Celestia settings to the way they were before running the script and to initialize those setting to the way you like them within your CELX script.

New Celestia v1.6.0 options are marked light green.
New Celestia v1.6.1 options are marked dark green.

```
-- Define Cleanup function to restore the settings
-- to the way they before running the script.

function celestia_cleanup_callback()
   celestia:setrenderflags(orig_renderflags)
   celestia:setlabelflags(orig_labelflags)
   celestia:setorbitflags(orig_orbitflags)
   celestia:setambient(orig_amb)
   celestia:setfaintestvisible(orig_faintest)
   celestia:setminorbitsize(orig_minorbit)
   celestia:setstardistancelimit(orig_stardistance)
   celestia:setminfeaturesize(orig_minfeature)
   celestia:setstarstyle(orig_starstyle)
   celestia:settime(orig_time + (celestia:getscripttime() / 24 / 3600) )
   celestia:settimescale(orig_timescale)
   celestia:setgalaxylightgain(orig_galaxylight)
   celestia:getobserver():setfov(orig_fov)
   celestia:getobserver():setlocationflags(orig_locationflags)
   celestia:getobserver():singleview()
   celestia:setoverlayelements(orig_overlay)
   celestia:setaltazimuthmode(orig_altazimuthmode)
   celestia:settextureresolution(orig_textres)
   celestia:settextcolor(orig_txt_r,orig_txt_g,orig_txt_b)
end

-- immidiatily followed by the following code to obtain the
-- current user settings and save them during the script.
```

```
orig_renderflags                = celestia:getrenderflags()
orig_labelflags                 = celestia:getlabelflags()
orig_orbitflags                 = celestia:getorbitflags()
orig_amb                        = celestia:getambient()
orig_faintest                   = celestia:getfaintestvisible()
orig_minorbit                   = celestia:getminorbitsize()
orig_stardistance               = celestia:getstardistancelimit()
orig_minfeature                 = celestia:getminfeatureresize()
orig_starstyle                  = celestia:getstarstyle()
orig_time                       = celestia:gettime()
orig_timescale                  = celestia:gettimescale()
orig_galaxylight                = celestia:getgalaxylightgain()
orig_fov                        = celestia:getobserver():getfov()
orig_locationflags              = celestia:getobserver():getlocationflags()
orig_overlay                    = celestia:getoverlayelements()
orig_altazimuthmode             = celestia:getaltazimuthmode()
orig_textres                    = celestia:gettextureresolution()
orig_txt_r,orig_txt_g,orig_txt_b = celestia:gettextcolor()

-- The following initialization part can be customized
-- by the wishes of the CELX writer to the way he/she
-- wants the settings to be during the script.

celestia:setorbitflags{Planet = true,
                       Moon = false,
                       Asteroid = false,
                       Comet = false,
                       Spacecraft = false,
                       Invisible = false,
                       Unknown = false,
                       DwarfPlanet = false,
                       MinorMoon = false,
                       Star = false
                      }

celestia:setrenderflags{atmospheres = true,
                        automag = true,
                        boundaries = false,
                        cloudmaps = true,
                        cloudshadows = true,
                        comettails = true,
                        constellations = false,
                        eclipseshadows = true,
                        galaxies = true,
                        grid = false,
                        markers = false,
                        nebulae = false,
                        nightmaps = true,
                        orbits = false,
                        planets = true,
                        ringshadows = true,
                        stars = true,
                        smoothlines = true,
                        lightdelay = false,
                        partialtrajectories = false,
                        ecliptic = false,
                        equatorialgrid = false,
                        galacticgrid = false,
                        eclipticgrid = false,
                        horizontalgrid = false
                       }
```

```
celestia:setlabelflags{planets = false,
                       moons = false,
                       spacecraft = false,
                       asteroids = false,
                       comets = false,
                       stars = false,
                       galaxies = false,
                       locations = false,
                       constellations = false,
                       i18nconstellations = false,
                       openclusters = false,
                       nebulae = false,
                       dwarfplanets = true,
                       minormoons = true,
                       globulars = true
                      }

celestia:setambient(0.05)
celestia:setfaintestvisible(7)

-- Make your standard Star Style choice below by
-- commenting/uncommenting one of the following lines:
celestia:setstarstyle("point")
-- celestia:setstarstyle("fuzzy")
-- celestia:setstarstyle("disc")



-- End of initialization part of the script



-- Followed by the rest of the script
```

CEL to CELX equivalents

The following sub-chapters explain the migration of CEL commands to CELX objects and methods according the sequens and explanation of CEL script commands in the available Celestia .Cel Scripting Guide v1-0g by Don Goyette.

The sub-chapters also contain many of Don's examples and syntaxes, to be as completely and consistent as possible to understand how the migration from CEL to CELX works.

CEL: cancel

- Cancels `GOTO` and `TRACK` commands, and resets the Coordinate System to `UNIVERSAL`, which means it also cancels `FOLLOW`, `SYNCHRONOUS` and other Coordinate System related commands. This command is much like pressing the [**Esc**] key when running Celestia in its interactive mode.
- Parameter list: nil

CELX equivalent:

The CELX equivalent consist of a sequence of 3 methods:

1. Get observer instance of the active view and stop any GOTO or CENTER command currently in progress.

```
obs = celestia:getobserver()
obs:cancelgoto()
```

2. Stop tracking an object.

```
obs:track(nil)
```

3. Create new frame and reset the coordinate system to universal.

```
frame = celestia:newframe("universal")
obs:setframe(frame)
```

Sumarized:

```
obs = celestia:getobserver()
obs:cancelgoto()
obs:track(nil)
frame = celestia:newframe("universal")
obs:setframe(frame)
```

Example:

The first part of this script selects the Earth, goes to it, and then performs other commands, maybe to change it's view, speed up time, etc. The second section executes a `cancel` command, selects Mars, and then performs other commands on Mars:

CEL:

```
select { object "Sol/Earth" }
goto   { time 3 }
wait   ( duration 3 }
# ...
# <other commands doing other things>
# ...
cancel { }
select { object "Sol/Mars" }
goto   { time 3 }
wait   { duration 3 }
# ...
# <other commands doing other things>
# ...
```

CELX:

```
earth = celestia:find("Sol/Earth")
obs = celestia:getobserver()
obs:goto(earth, 3.0)
wait(3.0)
-- ...
-- <other methods doing other things>
-- ...
obs:cancelgoto()
obs:track(nil)
frame = celestia:newframe("universal")
obs:setframe(frame)
mars = celestia:find("Sol/Mars")
obs:goto(mars, 3.0)
wait(3.0)
-- ...
-- <other methods doing other things>
-- ...
```

CEL: center

- Centers the currently selected object in the display in <*duration*> seconds. you must first use the select command to select an object to be centered.
- Parameter list:
  - time <*duration*>, default = 1.0 second.
    Number of seconds to take centering the object.

CELX equivalent:

Based on the **observer:center()** method.

1. Find and select target object with name <*string*> to be centered and store in "objectname"

```
objectname = celestia:find( <string> )
```

2. Get observer instance of the active view and center objectname.
   If no <*duration*> is given, the default time = 5.0 seconds !!!

```
obs = celestia:getobserver()
obs:center(objectname, <duration> )
```

3. Wait <*duration*> seconds.

```
wait( <duration> )
```

Sumarized:

```
objectname = celestia:find( <string> )
obs = celestia:getobserver()
obs:center(objectname, <duration> )
wait( <duration> )
```

Example:

The following example selects the Earth and takes 5.5 seconds to center it on the screen. If you do not see the Earth, press the [**G**] key on your keyboard and you should see the Earth zoom into view from the center of the display.

CEL:

```
select { object "Sol/Earth" }
center { time 5.5 }
```

CELX:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:center(earth, 5.5)
wait(5.5)
```

CEL: changedistance

- Changes the camera's distance from the currently selected object in <*duration*> seconds, with the specified <*rate*>, to achieve a certain <*distance*> (not specified in CEL). You must first use the select command to select an object.
- Parameter list:
  - duration <*duration*>, default = 1.0 second.
    Number of seconds to take when changing the distance.
  - rate <*rate*>, default = 0.0.
    Speed at which to change the distance. Small values work best, from decimal values, to 5 or 6. A negative value moves the camera closer to the object, while a positive value moves the camera further away ("+" sign is not necessary).

CELX equivalent:

Based on the **observer:gotodistance()** method.

1. Find the target object with name <*string*> to go to and store in “objectname”.

```
objectname = celestia:find( <string> )
```

2. Select “objectname” to be equivalent with CEL scripting.
   ➔Actually, this step is not necessary in CELX scripting!

```
celestia:select(objectname)
```

3. Get observer instance of the active view and goto given <*distance*> of “objectname” in <*duration*> seconds.

   a. <*distance*> is the distance from the center of target where to stop in km.
      i. If no <*distance*> is given, the default distance = 20000 km.
      ii. If <*distance*> is smaller than the radius of the object,
      the goto ends within the object !!!
      iii. To obtain the radius of an object in km,
      you can use the **“object:radius()” method**.

```
earth = celestia:find("Sol/Earth")
EarthRadius = earth:radius()
```

   b. <*duration*> is the number of seconds the goto should take.
      i. If no <*duration*> is given, the default time = 5.0 seconds !!!

Note: The **“observer:gotodistance()” method** uses <*distance*> to determine to goto and has NO ability to define the <*rate*>. So you first have to find out the resulting distance using the CEL: changedistance command at the specified <*rate*> and then convert that distance into the <*distance*> parameter of the **“observer:gotodistance()” method**.

```
obs = celestia:getobserver()
obs:gotodistance(objectname, <distance>, <duration> )
```

4. Wait <*duration*> seconds.

```
wait( <duration> )
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
obs = celestia:getobserver()
obs:gotodistance(objectname, <distance>, <duration> )
wait( <duration> )
```

Example:

The following example selects the Sun, travels to it, then selects Earth, travels to it, and then spends 2.5 seconds changing the display distance to be further away from the Earth.

CEL:

```
select { object "Sol" }
goto   { time 3 }
wait   { duration 5 }
select { object "Sol/Earth" }
goto   { time 3 }
wait   { duration 5 }
changedistance { duration 2.5 rate 0.5 }
wait   { duration 2.5 }
```

CELX:

```
sun = celestia:find("Sol")
celestia:select(sun)
obs = celestia:getobserver()
obs:goto(sun, 3.0)
wait(5.0)
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs:goto(earth, 3.0)
wait(5.0)
obs:gotodistance(earth, 89048+6378, 2.5)
wait(2.5)
```

CEL: chase

- Tells Celestia to set the active Coordinate System to `CHASE`, which *chases* the currently selected object through space.
- Parameter list: nil

CELX equivalent-1:

Based on the **observer:setframe()** method.

1. Find the target object with name <*string*> to be chased and store in “objectname”.

```
objectname = celestia:find( <string> )
```

2. Create new frame of reference to chase with “objectname” as reference object and store in “frame”.

```
frame = celestia:newframe("chase", objectname)
```

3. Get observer instance of the active view and set the coordinate system of the frame of reference to “frame”.

```
obs = celestia:getobserver()
obs:setframe(frame)
```

Sumarized:

```
objectname = celestia:find( <string> )
frame = celestia:newframe("chase", objectname)
obs = celestia:getobserver()
obs:setframe(frame)
```

CELX equivalent-2:

Based on the **observer:chase()** method.

1. Find the target object with name <*string*> to be chased and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Get observer instance of the active view and activate chase-mode on "objectname". *Chase-mode is the same as setting the frame of reference to chase with objectname as reference object.*

```
obs = celestia:getobserver()
obs:chase(objectname)
```

Sumarized:

```
objectname = celestia:find( <string> )
obs = celestia:getobserver()
obs:chase(objectname)
```

Example:

The following example selects the Moon, sets the Coordinate System to CHASE, and then goes to the Moon.

CEL:

```
select { object "Sol/Earth/Moon" }
chase  { }
goto   { time 2 }
wait   { duration 2 }
```

CELX-1 with observer:setframe() method:

```
moon = celestia:find("Sol/Earth/Moon")
celestia:select(moon)
frame = celestia:newframe("chase", moon)
obs = celestia:getobserver()
obs:setframe(frame)
obs:goto(moon, 2.0)
wait(2.0)
```

CELX-2 with observer:chase() method:

```
moon = celestia:find("Sol/Earth/Moon")
celestia:select(moon)
obs = celestia:getobserver()
obs:chase(moon)
obs:goto(moon, 2.0)
wait(2.0)
```

CEL: cls

- Clears the display screen of any left-over text that was printed via the print command.
- Parameter list: nil
- There is no much need to use this command.

CELX equivalent-1:

Based on the **celestia:print()** method.

1. Clear the display screen of any left-over text, by printing a blanc line.

```
celestia:print("")
```

CELX equivalent-2:

Based on the **celestia:flash()** method.

1. Clear the display screen of any left-over text, by flashing a blanc line.

```
celestia:flash("")
```

Example:

The following example prints text for 5 seconds, but after 2 seconds the display is cleared.

CEL:

```
print { text "Text to be displayed for 5 sec." row -3 column 1 duration 5 }
wait  { duration 2 }
cls   { }
wait  { duration 3 }
```

-- OR --

```
print { text "Text to be displayed for 5 sec." row -3 column 1 duration 5 }
wait  { duration 2 }
print { text "" }
wait  { duration 3 }
```

CELX-1 with celestia:print() method:

```
celestia:print("Text to be displayed for 5 sec." , 5.0, -1, -1, 1, 3)
wait(2.0)
celestia:print("")
wait(3.0)
```

CELX-2 with celestia:flash() method:

```
celestia:print("Text to be displayed for 5 sec." , 5.0, -1, -1, 1, 3)
wait(2.0)
celestia:flash("")
wait(3.0)
```

CEL: follow

- This command tells Celestia to set the Coordinate System to `ECLIPTICAL`, and to *follow* the currently selected object.
- Parameter list: nil

CELX equivalent-1:

Based on the **observer:setframe()** method.

1. Find the target object with name <*string*> to be followed and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Create new frame of reference to ecliptic with "objectname" as reference object and store in "frame".

```
frame = celestia:newframe("ecliptic", objectname)
```

3. Get observer instance of the active view and set the coordinate system of the frame of reference to "frame".

```
obs = celestia:getobserver()
obs:setframe(frame)
```

Sumarized:

```
objectname = celestia:find( <string> )
frame = celestia:newframe("ecliptic", objectname)
obs = celestia:getobserver()
obs:setframe(frame)
```

CELX equivalent-2:

Based on the **observer:follow()** method.

1. Find the target object with name <*string*> to be followed and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Get observer instance of the active view and activate follow-mode on "objectname". *Follow-mode is the same as setting the frame of reference to ecliptic with "objectname" as reference object.*

```
obs = celestia:getobserver()
obs:follow(objectname)
```

Sumarized:

```
objectname = celestia:find( <string> )
obs = celestia:getobserver()
obs:follow(objectname)
```

Example:

This example selects Mars, sets the Coordinate System to ecliptical (`follow`), and then travels to Mars.

CEL:

```
select { object "Sol/Mars" }
follow { }
goto   { time 2 }
wait   { duration 2 }
```

CELX-1 with observer:setframe() method:

```
mars = celestia:find("Sol/Mars")
celestia:select(mars)
frame = celestia:newframe("ecliptic", mars)
obs = celestia:getobserver()
obs:setframe(frame)
obs:goto(mars, 2.0)
wait(2.0)
```

CELX-2 with observer:follow() method:

```
mars = celestia:find("Sol/Mars")
celestia:select(mars)
obs = celestia:getobserver()
obs:follow(mars)
obs:goto(mars, 2.0)
wait(2.0)
```

CEL: goto

- Moves the camera to the currently selected object, taking <*duration*> seconds, stopping <*radiusdistance*> from the object (in units of object's radius + 1), using the <*upframestring*> Coordinate System, and defining the <*upvector*> axis that points up.
- Parameter list:
  - time <*duration*>, default = 1.0 second.
    The number of seconds to take going to the object.
  - distance <*radiusdistance*>, default = 5.0.
    Describes how far away from the object you want to be positioned, in units of the object's radius, plus 1. Special <*radiusdistance*> values are:
    - ➔ 0 (zero) = Center of the object. In version 1.3.1, using this value causes the program to incorrectly recognize further positioning values, so do not use zero.
    - ➔ 1 = Surface of the object. Traveling to the exact surface level of an object may cause some graphics cards to display random polygons on the display screen, so it is best to use a value slightly above the surface.
  - upframe <*upframestring*>, default = "observer".
    Sets the specified Coordinate System, which **must** be one of the following values (for more information, refer to the Coordinate System descriptions earlier in this guide):
    - ➔ **`chase`**
    - ➔ **`ecliptical`**
    - ➔ **`equatorial`**
    - ➔ **`geographic`**
    - ➔ **`lock`**
    - ➔ **`observer`**
    - ➔ **`universal`**
  - up <*upvector*>, default = [0 1 0].
    Defines which axis points up, X [1 0 0], Y [0 1 0] or Z [0 0 1].

CELX equivalent-1:

Based on the **observer:gotodistance()** method.

1. Find and select the target object with name <*string*> to go to and store in "objectname".

```
objectname = celestia:find( <string> )
celestia:select(objectname)
```

2. Get observer instance of the active view instance and set the coordinate system of the frame of reference to <*upframestring*>.

```
obs = celestia:getobserver()
frame = celestia:newframe( <upframestring>, objectname)
obs:setframe(frame)
```

3. Determine radius of "objectname" and store in "radius".

```
radius = objectname:radius()
```

4. Determine <*distance*> as: <*radiusdistance*> * "radius" of "objectname".

```
distance = <radiusdistance> * radius
```

5. Goto determined <*distance*> of "objectname" in <*duration*> seconds.

   a. <*distance*> is the distance from the center of target where to stop in km.
      i. If no <*distance*> is given, the default distance = 20000 km.
      ii. If <*distance*> is smaller than the radius of the object, the goto will end within the object !!!
      iii. To obtain the radius of an object in km, you can use the **"object:radius()" method**.
   b. <*duration*> is the number of seconds the goto should take.
      i. If no <duration> is given, the default time = 5.0 seconds !!!

   Note: The **"observer:gotodistance()"** method has NO ability to define the axis that points up.

```
obs:gotodistance(objectname, distance, <duration> )
```

6. Wait <*duration*> seconds.

```
wait( <duration> )
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
obs = celestia:getobserver()
frame = celestia:newframe( <upframestring>, objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = <radiusdistance> * radius
obs:gotodistance(objectname, distance, <duration> )
wait( <duration> )
```

CELX equivalent-2:

Based on the **observer:goto(table)** method.

Using this method, many different types of gotos can be performed. The strenth of this method is that the observer orientation can be programmed, so the axis that points up after the goto can also be determined. This method uses positions to goto, instead of a distance from the object, so that needs some additional calculations. The parameters for the goto must be given in the table:

- parameters.duration: duration (number)
- parameters.from: source position
- parameters.to: target position
- parameters.initialOrientation: source orientation
- parameters.finalOrientation: target orientation
- parameters.startInterpolation:
- parameters.endInterpolation:
- parameters.accelTime:

1. Find and select the target object with name <*string*> to go to and store in “objectname”.

```
objectname = celestia:find( <string> )
celestia:select(objectname)
```

2. Get observer instance of the active view instance and set the coordinate system of the frame of reference to <*upframestring*>.

```
obs = celestia:getobserver()
frame = celestia:newframe( <upframestring>, objectname)
obs:setframe(frame)
```

3. Determine radius of “objectname” and store in “radius”.

```
radius = objectname:radius()
```

4. Determine distance in km to goto as: <*radiusdistance*> * “radius” and store in “distance”.

```
distance = <radiusdistance> * radius
```

5. Convert the determined “distance” to “objectname” into positions as follows:

```
uly_to_km = 9460730.4725808
-- Determine and store the current observer position
frompos = obs:getposition()
-- Determine and store the current position of objectname
objectpos = objectname:getposition()
-- Determine the vector between the current observer
-- position and the current position of objectname
distancevec = frompos:vectorto(objectpos)
-- Determine the length of this vector in millionths of a light-year.
distancelength = distancevec:length()
-- Normalize this length to 1
normalvec = distancevec:normalize()
-- Distance to travel = (distancelength - distance /uly_to_km)
-- Direction to travel = normalvec
travelvec = (distancelength - distance /uly_to_km)*normalvec
-- Determine and store the position to goto
topos = frompos + travelvec
```

6. The target observer orientation must point from the target position “topos” towards “objectname” with <*upvector*> as up vector:

```
upvec = celestia:newvector( <upvector> )
torot = topos:orientationto(objectpos, upvec)
```

7. Define and initialize the parameter table as follows:

```
parameters={}
```

   a. Determine the number of seconds the goto should take.

```
parameters.duration = <duration>
```

   b. Obtain the current position of the observer.

```
parameters.from = obs:getposition()
```

   c. New position object = topos (step 5)

```
parameters.to = topos
```

   d. Obtain the current orientation of the observer.

```
parameters.initialOrientation = obs:getorientation()
```

   e. New observer orientation = torot (step 6)

```
parameters.finalOrientation = torot
```

   f. Start adjusting the observer orientation after 20 percent of the time to goto.

```
parameters.startInterpolation = 0.2
```

   g. End adjusting the observer orientation after 80 percent of the time to goto.

```
parameters.endInterpolation = 0.8
```

   h. Spend 10% of the time for accelerating and decelerating

```
parameters. accelTime = 0.1
```

8. Goto target position with target orientation in <*duration*> seconds.

```
obs:goto(parameters)
```

9. Wait <*duration*> seconds.

```
wait(parameters.duration)
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
obs = celestia:getobserver()
frame = celestia:newframe( <upframestring>, objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = <radiusdistance> * radius
uly_to_km = 9460730.4725808
-- Determine and store the current observer position
frompos = obs:getposition()
-- Determine and store the current position of objectname
objectpos = objectname:getposition()
-- Determine the vector between the current observer
-- position and the current position of objectname
distancevec = frompos:vectorto(objectpos)
-- Determine the length of this vector in millionths of a light-year.
distancelength = distancevec:length()
-- Normalize this length to 1
normalvec = distancevec:normalize()
-- Distance to travel = (distancelength - distance /uly_to_km)
-- Direction to travel = normalvec
travelvec = (distancelength - distance /uly_to_km)*normalvec
-- Determine and store the position to goto
topos = frompos + travelvec
upvec = celestia:newvector( <upvector> )
torot = topos:orientationto(objectpos, upvec)
parameters={}
parameters.duration = <duration>
parameters.from = obs:getposition()
parameters.to = topos
parameters.initialOrientation = obs:getorientation()
parameters.finalOrientation = torot
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters. accelTime = 0.1
obs:goto(parameters)
wait(parameters.duration)
```

Example:

The following example selects Mars and takes five seconds to travel 10 times the radius of mars above the surface and Y-axis pointing up.

CEL:

```
select { object "Mars" }
goto   { time 5.0 distance 11.0 upframe "ecliptic" up [0 1 0] }
print  { text "We're on our way to Mars." row -3 column 1 duration 5 }
wait   { duration 5 }
```

CELX-1 with observer:gotodistance() method:

```
mars = celestia:find("Sol/Mars")
celestia:select(mars)
frame = celestia:newframe("ecliptic", mars)
obs = celestia:getobserver()
obs:setframe(frame)
marsradius = mars:radius()
marsdistance = 11 * marsradius
obs:gotodistance(mars, marsdistance, 5.0)
celestia:print("We're on our way to Mars." , 5.0, -1, -1, 1, 3)
wait(5.0)
```

CELX-2 with observer:goto(table) method:

```
mars = celestia:find("Sol/Mars")
celestia:select(mars)
frame = celestia:newframe("ecliptic", mars)
obs = celestia:getobserver()
obs:setframe(frame)
marsradius = mars:radius()
distance = 11 * marsradius
uly_to_km = 9460730.4725808
frompos = obs:getposition()
objectpos = mars:getposition()
distancevec = frompos:vectorto(objectpos)
distancelength = distancevec:length()
normalvec = distancevec:normalize()
travelvec = (distancelength - distance /uly_to_km)*normalvec
topos = frompos + travelvec
upvec = celestia:newvector(0,1,0)
torot = topos:orientationto(objectpos, upvec)
parameters={}
parameters.duration = 5.0
parameters.from = obs:getposition()
parameters.to = topos
parameters.initialOrientation = obs:getorientation()
parameters.finalOrientation = torot
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters. accelTime = 0.1
obs:goto(parameters)
celestia:print("We're on our way to Mars." , 5.0, -1, -1, 1, 3)
wait(parameters.duration)
```

CEL: gotoloc

- Move the camera to the currently selected object, taking <*duration*> seconds, traveling to the specified position (or the Base64 values x, y, and z from a Cel://URL), using the orientation specified via xrot, yrot, and zrot (or ow, ox, oy, and oz).
- Parameter list-1:
  - time <*duration*>, default = 1.0 second.
    The number of seconds to take going to the object.
  - position <*vector*>, default = [0 1 0].
    Defines a point in the current Coordinate System specified in units of kilometers. For all but the `UNIVERSAL` Coordinate System, a `POSITION` of [0 0 0] is the center of the object. The kilometer value you specify must include the radius of the object.
    - ➔ The 1st value <*xnumber*> represents the camera location, in kilometers, along the X axis.
    - ➔ The 2nd value <*ynumber*> represents the camera location, in kilometers, along the Y axis.
    - ➔ The 3rd value <*znumber*> represents the camera location, in kilometers, along the Z axis.
  - xrot <*xrot*>, default = 0.
  - yrot <*yrot*>, default = 0.
  - zrot <*zrot*>}, default = 0.
    - ➔ <*xrot*> , <*yrot*> and <*zrot*> are the "Eular Angle" or "Angle-Axis" representation of the camera's orientation in degrees. Think of them as Pitch, Yaw, and Roll in aviation.
- Parameter list-2:
  - time <*duration*>, no default.
    The number of seconds to take going to the object.
  - x <*xbase64*>, no default.
  - y <*ybase64*>, no default.
  - z <*zbase64*>, no default.
    - ➔ The x, y, and z values represent POSITION as stored by a Cel://URL.
      These values are obtained from the following Cel://URL values: ?x= &y= &z=
  - ow <*ownumber*>, no default.
  - ox <*oxnumber*>, no default.
  - oy <*oynumber*>, no default.
  - oz <*oznumber*>, no default.
    - ➔ The ow, ox, oy and oz values represent ORIENTATION as stored by a Cel://URL.
      These values are obtained from the following Cel://URL values:
      &ow= &ox= &oy= &oz=

CELX equivalent-1:

Based on Parameter list-1 and the **observer:setorientation()** and **observer:gotolocation()** methods.

The disadvantage of this combination of methods is the sequential execution of these methods and the new observer orientation is set at once instead of smoothly during the goto.

1. Create new position object from <*xnumber*>, <*ynumber*> and <*znumber*> and store in "pos". The units of the components of a position object are millionths of a light-year, so you have to convert km to millionths of a light year, using the "uly_to_km" constant.

```
uly_to_km = 9460730.4725808
pos = celestia:newposition( <xnumber>/uly_to_km, <ynumber>/uly_to_km, <znumber>/uly_to_km )
```

2. Specify and create the axis of this rotation [<*xrot*> , <*yrot*> , <*zrot*>] and store it in "vec". <*xrot*> , <*yrot*> and <*zrot*> are the "Eular Angle" or "Angle-Axis" representation of the camera's orientation in degrees. Think of them as Pitch, Yaw, and Roll in aviation.

```
vec = celestia:newvector( <xrot> , <yrot> , <zrot> )
```

3. Create new rotation (i.e. a quaternion) about the specified axis and store in "rot". "angle" = math.pi / 180 (= 3.14159265 / 180), to obtain radians out of degrees.

```
angle = math.pi/180
rot = celestia:newrotation(vec, angle)
```

4. Get observer instance of the active view instance and rotate the observer according the created new rotation "rot".

```
obs = celestia:getobserver()
obs:setorientation(rot)
```

5. Go to target position "pos" in <*duration*> seconds.

```
obs:gotolocation(pos, <duration> )
```

6. Wait <*duration*> seconds.

```
wait( <duration> )
```

Sumarized:

```
uly_to_km = 9460730.4725808
pos = celestia:newposition( <xnumber>/uly_to_km, <ynumber>/uly_to_km, <znumber>/uly_to_km )
vec = celestia:newvector( <xrot> , <yrot> , <zrot> )
angle = math.pi/180
rot = celestia:newrotation(vec, angle)
obs = celestia:getobserver()
obs:setorientation(rot)
obs:gotolocation(pos, <duration> )
wait( <duration> )
```

CELX equivalent-2:

Based on Parameter list-2 and the **observer:setorientation()** and **observer:gotolocation()** methods.

The disadvantage of this combination of methods is the sequential execution of these methods and the new observer orientation is set at once instead of smoothly during the goto.

1. Create new position object from URL-style Base64-encoded values and store in "pos".
   <*xbase64*>: The X-component of the new vector, as a string-value taken from a cel-style URL ?x=.
   <*ybase64*>: The Y-component of the new vector, as a string-value taken from a cel-style URL &y=.
   <*zbase64*>: The Z-component of the new vector, as a string-value taken from a cel-style URL &z=.

```
pos = celestia:newposition( <xbase64>, <ybase64>, <zbase64> )
```

2. Create new rotation (i.e. a quaternion) from four scalar values (the values used in a CEL://URL), and store in "rot".
   <*ownumber*>: The OW-component of the new rotation, as a number-value taken from a cel-style URL &ow=.
   <*oxnumber*>: The OX-component of the new rotation, as a number-value taken from a cel-style URL &ox=.
   <*oynumber*>: The OY-component of the new rotation, as a number-value taken from a cel-style URL &oy=.
   <*oznumber*>: The OZ-component of the new rotation, as a number-value taken from a cel-style URL &oz=.

```
rot = celestia:newrotation( <ownumber>, <oxnumber>, <oynumber>, <oznumber> )
```

3. Get observer instance of the active view and rotate the observer according the created new rotation "rot".

```
obs = celestia:getobserver()
obs:setorientation(rot)
```

4. Go to target position "pos" in <*duration*> seconds.

```
obs:gotolocation(pos, <duration> )
```

5. Wait <*duration*> seconds.

```
wait( <duration> )
```

Sumarized:

```
pos = celestia:newposition( <xbase64>, <ybase64>, <zbase64> )
rot = celestia:newrotation( <ownumber>, <oxnumber>, <oynumber>, <oznumber> )
obs = celestia:getobserver()
obs:setorientation(rot)
obs:gotolocation(pos, <duration> )
wait( <duration> )
```

CELX equivalent-3:

Based on the **observer:goto(table)** method.

Using this method, many different types of gotos can be performed. The strength of this method is that the observer orientation can be programmed and interpolated, so the orientation of the observer can be changed during the goto. This method uses position objects to goto and rotation objects to set the observer orientation. Both positions and rotations can be extracted from Parameter list-1 and Parameter list-2 arguments.

The parameters for the goto must be given in the table:

- parameters.duration: duration (number)
- parameters.from: source position
- parameters.to: target position
- parameters.initialOrientation: source orientation
- parameters.finalOrientation: target orientation
- parameters.startInterpolation:
- parameters.endInterpolation:
- parameters.accelTime:

1. Get observer instance of the active view and store in “obs”.

```
obs = celestia:getobserver()
```

2. Define and initialize the parameter table as follows:

```
parameters={}
```

   a. Determine the number of seconds the goto should take.

```
parameters.duration = <duration>
```

   b. Obtain the current position of the observer.

```
parameters.from = obs:getposition()
```

c. Create new position object.
There are 2 possibilities regarding the originally used CEL parameterlist:

i. from <*xnumber*>, <*ynumber*> and <*znumber*> and store in "parameters.to". The units of the components of a position object are millionths of a light-year, so you have to convert km to millionths of a light year, using "uly_to_km".

```
uly_to_km = 9460730.4725808
parameters.to = celestia:newposition( <xnumber>/uly_to_km, <ynumber>/uly_to_km
, <znumber>/uly_to_km )
```

Note: The given positions are expected to be relative to the universal frame-of-reference, while the non table-based goto uses positions relative to the current frame-of-reference. So depending on the used frame of reference, you have to convert the position first to "Universal".

```
-- Example: frame of reference is "ECLIPTIC" (follow),
-- with <objectname> as reference object.
obs = celestia:getobserver()
objectname = celestia:find( <string> )
obs:follow(objectname)
uly_to_km = 9460730.4725808
vector = celestia:newvector( <xnumber>/uly_to_km, <ynumber>/uly_to_km,
 <znumber>/ uly_to_km )
objectpos = objectname:getposition()
parameters.to = objectpos + vector
```

ii. from URL-style Base64-encoded values and store in "parameters.to":
<*xbase64*>: The X-component of the new vector, as a string-values taken from a cel-style URL ?x=.
<*ybase64*>: The Y-component of the new vector, as a string-values taken from a cel-style URL &y=.
<*zbase64*>: The Z-component of the new vector, as a string-values taken from a cel-style URL &z=.

```
parameters.to = celestia:newposition( <xbase64>, <ybase64>, <zbase64> )
```

d. Obtain the current orientation of the observer.

```
parameters.initialOrientation = obs:getorientation()
```

e. Create new rotation object.
There are 2 possibilities regarding the originally used CEL parameterlist:

i. Specify and create the axis of the observer rotation [<*xrot*> , <*yrot*> , <*zrot*>] and store it in "vec". <*xrot*> , <*yrot*> and <*zrot*> are the "Eular Angle" or "Angle-Axis" representation of the observer orientation in degrees. Think of them as Pitch, Yaw, and Roll in aviation. Then create new rotation (i.e. a quaternion) about the specified axis and use "angle" = math.pi/180 (= 3.14159265 / 180), to obtain radians out of degrees. Store this new rotation in "parameters.finalOrientation".

```
vec = celestia:newvector( <xrot> , <yrot> , <zrot> )
angle = math.pi/180
parameters.finalOrientation = celestia:newrotation(vec, angle)
```

ii. from four scalar values (the values used in a CEL://URL), and store in "parameters.finalOrientation".
<*ownumber*>: The OW-component of the new rotation, as a number-value taken from a cel-style URL &ow=.
<*oxnumber*>: The OX-component of the new rotation, as a number-value taken from a cel-style URL &ox=.
<*oynumber*>: The OY-component of the new rotation, as a number-value taken from a cel-style URL &oy=.
<*oznumber*>: The OZ-component of the new rotation, as a number-value taken from a cel-style URL &oz=.

```
parameters.finalOrientation =
celestia:newrotation( <ownumber>, <oxnumber>, <oynumber>, <oznumber> )
```

f. Start adjusting the observer orientation after 20 percent of the time to goto.

```
parameters.startInterpolation = 0.2
```

g. End adjusting the observer orientation after 80 percent of the time to goto.

```
parameters.endInterpolation = 0.8
```

h. Spend 10% of the time for accelerating and decelerating

```
parameters. accelTime = 0.1
```

3. Goto target position with target orientation in <*duration*> seconds.

```
obs:goto(parameters)
```

4. Wait <*duration*> seconds.

```
wait(parameters.duration)
```

Sumarized:

```
-- Use the following block of code for target positions
-- relative to the universal frame-of-reference:
obs = celestia:getobserver()
parameters={}
parameters.duration = <duration>
parameters.from = obs:getposition()
uly_to_km = 9460730.4725808
parameters.to = celestia:newposition( <xnumber>/uly_to_km, <ynumber>/uly_to_
km, <znumber>/uly_to_km )
parameters.initialOrientation = obs:getorientation()
vec = celestia:newvector( <xrot> , <yrot> , <zrot> )
angle = math.pi/180
parameters.finalOrientation = celestia:newrotation(vec, angle)
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters.accelTime = 0.1
obs:goto(parameters)
wait(parameters.duration)
```

-- OR --

```
-- Use the following block of code for target positions
-- relative to the “ECLIPTIC” (follow) frame of reference,
-- with “objectname” as reference object.
obs = celestia:getobserver()
objectname = celestia:find( <string> )
obs:follow(objectname)
parameters={}
parameters.duration = <duration>
parameters.from = obs:getposition()
uly_to_km = 9460730.4725808
vector = celestia:newvector( <xnumber>/uly_to_km, <ynumber>/uly_to_km, <znum
ber>/ uly_to_km )
objectpos = objectname:getposition()
parameters.to = objectpos + vector
parameters.initialOrientation = obs:getorientation()
vec = celestia:newvector( <xrot> , <yrot> , <zrot> )
angle = math.pi/180
parameters.finalOrientation = celestia:newrotation(vec, angle)
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters.accelTime = 0.1
obs:goto(parameters)
wait(parameters.duration)
```

-- OR --

```
obs = celestia:getobserver()
parameters={}
parameters.duration = <duration>
parameters.from = obs:getposition()
parameters.to = celestia:newposition( <xbase64>, <ybase64>, <zbase64> )
parameters.initialOrientation = obs:getorientation()
parameters.finalOrientation = celestia:newrotation( <ownumber>, <oxnumber>,
 <oynumber>, <oznumber> )
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters.accelTime = 0.1
obs:goto(parameters)
wait(parameters.duration)
```

Example:

The next example demonstrates both GOTOLOC methods. It travels to the Earth, setting a position so you can see the night lights over the USA, view the Sun in the upper left hand corner of the display, and the Moon in the upper right hand corner.

Note: The lines of code starting with "#" (CEL) and "--" (CELX) are comment lines for further explanation:

Note: The CelURL values in this example are compatible with Celestia version 1.6.0.

CEL:

```
# Pause time and set the date and time to 2003 Aug 23 04:37:24 UTC
  timerate  { rate 0 }
  time      { jd 2452874.692638889 }
# Activate markers, clear all existing markers and
#    place a blue square marker around the Moon...
  renderflags { set "markers" }
  unmarkall { }
  mark      { object "Sol/Earth/Moon" size 20.0 color [0 0 1] symbol "square" }
# Set the Field Of View value to 47.0 degrees...
  set       { name "FOV" value 47.0 }
# Select, follow and center Earth (so the coordinate system
#    of the frame of reference is set to ecliptic).
  select    { object "Sol/Earth" }
  follow    { }
  center    { }
  wait      { duration 1 }
# Goto location: X = +7000km, Y = +9000km, Z = +11000km.
#    Pitch Up +13 degrees (down = -), Yaw Left -32 degrees (right = +),
#    Roll 0 degrees (left = - , right = +)
  gotoloc   { time 5 position [7000 9000 11000] xrot 13 yrot -32 zrot 0 }
  wait      { duration 5 }
  print     { text "Blue square is the Moon ---" duration 4 row -24 column 32 }
  wait      { duration 6 }
  print { text "First result done." duration 3 row -3 column 2 }
  wait { duration 3 }
  cancel { }
# Goto location using URL-style Base64-encoded values.
  gotoloc { time 5 x "AAAAbtdBM3XLDA" y "aG/b0Q34Pw" z "AABiyVOKuxUI"
            ow 0.703773 ox 0.068516 oy -0.703786 oz 0.068514 }
  wait { duration 5 }
  print { text "Second result done." duration 4  row -3 column 2 }
  wait { duration 4 }
```

CELX-1 with observer:gotolocation() method:

```
-- Pause time and set the date and time to 2003 Aug 23 04:37:24 UTC.
-- First convert UTC to TDB date/time:
--    UTC is used in the Celestia's Set Time dialog and it's also
--       the time displayed in the upper right of the screen.
--    TDB = Barycentric Dynamical Time.
--       When using scripts for Celestia, the time scale is TDB !!!
celestia:settimescale(0)
dt=celestia:utctotdb(2003, 08, 23, 04, 37, 24)
celestia:settime(dt)
-- Activate markers, clear all existing markers and
--    place a blue square marker around the Moon...
celestia:setrenderflags{markers=true}
celestia:unmarkall()
moon = celestia:find("Sol/Earth/Moon")
moon:mark("blue", "square", 20.0)
-- Set the Field Of View value to 47.0 degrees.
-- In CELX, angles are stored in radians, instead of degrees, so you have to convert
-- degrees to radians by multiplying degrees with math.pi (= 3.14159265) and divide
-- by 180. The LUA "math.rad()" function can also be used for this.
obs = celestia:getobserver()
obs:setfov(47*math.pi/180)
-- Select, follow and center Earth (so the coordinate system
--    of the frame of reference is set to ecliptic).
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs:follow(earth)
obs:center(earth, 1.0)
wait(1.0)
-- Goto location: X = +7000km, Y = +9000km, Z = +11000km.
--    Pitch Up +13 degrees (down = -),
--    Yaw Left -32 degrees (right = +),
--    Roll 0 degrees (left = - , right = +)
obs = celestia:getobserver()
-- In CELX, angles are stored in radians, instead of degrees, so you have to convert
-- degrees to radians by multiplying degrees with math.pi (= 3.14159265) and divide
-- by 180. The LUA "math.rad()" function can also be used for this.
vec = celestia:newvector(13, -32, 0)
angle = math.pi/180
-- Set the observer orientation:
rot = celestia:newrotation(vec, angle)
obs:setorientation(rot)
-- In CELX, positions are stored in millionths of a light year,
-- so you have to convert km to millionths of a light year, using "uly_to_km".
uly_to_km = 9460730.4725808
pos = celestia:newposition(7000/uly_to_km, 9000/uly_to_km, 11000/uly_to_km)
-- Goto target position.
obs:gotolocation(pos, 5.0 )
wait(5.0)
celestia:print("Blue square is the Moon ---" , 4.0, -1, -1, 32, 24)
wait(6.0)
celestia:print("First result done.", 3.0, -1, -1, 2, 3)
wait(3.0)
-- Cancel GOTO and TRACK commands, and reset the Coordinate System to universal.
obs:cancelgoto()
obs:track(nil)
obs:setframe(celestia:newframe("universal"))
-- Goto target position, using URL-style Base64-encoded values.
obs = celestia:getobserver()
rot = celestia:newrotation(0.703773, 0.068516, -0.703786, 0.068514)
obs:setorientation(rot)
pos = celestia:newposition("AAAAbtdBM3XLDA", "aG/b0Q34Pw", "AABiyVOKuxUI")
obs:gotolocation(pos, 5.0 )
wait(5.0)
celestia:print("Second result done.", 4.0, -1, -1, 2, 3)
wait(4.0)
```

CELX-2 with observer:goto(table) method:

```
-- Pause time and set the date and time to 2003 Aug 23 04:37:24 UTC.
-- First convert UTC to TDB date/time:
--    UTC is used in the Celestia's Set Time dialog and it's also
--       the time displayed in the upper right of the screen.
--    TDB = Barycentric Dynamical Time.
--       When using scripts for Celestia, the time scale is TDB !!!
celestia:settimescale(0)
dt=celestia:utctotdb(2003, 08, 23, 04, 37, 24)
celestia:settime(dt)
-- Activate markers, clear all existing markers and
--    place a blue square marker around the Moon...
celestia:setrenderflags{markers=true}
celestia:unmarkall()
moon = celestia:find("Sol/Earth/Moon")
moon:mark("blue", "square", 20.0)
-- Set the Field Of View value to 47.0 degrees.
-- In CELX, angles are stored in radians, instead of degrees, so you have to convert
-- degrees to radians by multiplying degrees with math.pi (= 3.14159265) and divide
-- by 180. The LUA "math.rad()" function can also be used for this.
obs = celestia:getobserver()
obs:setfov(47*math.pi/180)
-- Select, follow and center Earth (so the coordinate system
--    of the frame of reference is set to ecliptic).
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs:follow(earth)
obs:center(earth, 1.0)
wait(1.0)
-- Goto location: X = +7000km, Y = +9000km, Z = +11000km, relative to Earth !!!!
--    Pitch Up +13 degrees (down = -),
--    Yaw Left -32 degrees (right = +),
--    Roll 0 degrees (left = - , right = +)
obs = celestia:getobserver()
parameters={}
parameters.duration = 5.0
parameters.from = obs:getposition()
-- In CELX, positions are stored in millionths of a light year,
-- so you have to convert km to millionths of a light year, using "uly_to_km".
uly_to_km = 9460730.4725808
vec = celestia:newvector(7000/uly_to_km, 9000/uly_to_km, 11000/uly_to_km)
earthpos = earth:getposition()
parameters.to = earthpos + vec
parameters.initialOrientation = obs:getorientation()
-- In CELX, angles are stored in radians, instead of degrees, so you have to convert
-- degrees to radians by multiplying degrees with math.pi (= 3.14159265) and divide
-- by 180. The LUA "math.rad()" function can also be used for this.
vec = celestia:newvector(13, -32, 0)
angle = math.pi/180
-- Set the observer orientation:
parameters.finalOrientation = celestia:newrotation(vec, angle)
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters.accelTime = 0.1
-- Goto target position.
obs:goto(parameters)
wait(parameters.duration)
celestia:print("Blue square is the Moon ---" , 4.0, -1, -1, 32, 24)
wait(6.0)
celestia:print("First result done.", 3.0, -1, -1, 2, 3)
wait(3.0)
-- Cancel GOTO and TRACK commands, and reset the Coordinate System to universal.
obs:cancelgoto()
obs:track(nil)
obs:setframe(celestia:newframe("universal"))
```

```
-- Goto target position, using URL-style Base64-encoded values.
parameters={}
parameters.duration = 5.0
parameters.from = obs:getposition()
parameters.to = celestia:newposition("AAAAbtdBM3XLDA", "aG/b0Q34Pw", "AABiyVOKuxUI")
parameters.initialOrientation = obs:getorientation()
parameters.finalOrientation = celestia:newrotation(0.703773, 0.068516, -0.703786,
0.068514)
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters.accelTime = 0.1
-- Goto target position.
obs:goto(parameters)
wait(parameters.duration)
celestia:print("Second result done.", 4.0, -1, -1, 2, 3)
wait(4.0)
```

CEL: gotolonglat

- Go to the currently selected object, taking <*duration*> seconds, stopping <*radiusdistance*> from the object, using <*upvector*> orientation, positioning yourself above the specified longitude <*longnumber*> and latitude <*latnumber*> surface coordinates.
- **Note:** GOTOLONGLAT does **not** reset the coordinate system.
  It retains the previously set coordinate system.
- Parameter list:
  - time <*duration*>, default = 1.0 second.
    The number of seconds to take going to the object.
  - distance <*radiusdistance*>, default = 5.0
    Describes how far away from the object you want to be positioned, in units of the object's radius, plus 1. Special <distance> values are:
    - ➔ 0 (zero) = Center of the object.**Note:** In version 1.3.1, using this value causes the program to incorrectly recognize further positioning values, so do not use zero.
    - ➔ 1 = Surface of the object. **Note:** Traveling to the exact surface level of an object may cause some graphics cards to display random polygons on the display screen, so it is best to use a value slightly above the surface.
  - up <*upvector*>, default = [0 1 0].
    Defines which axis points up, X [1 0 0], Y [0 1 0] or Z [0 0 1].
  - longitude <*longnumber*>, no default.
    This value describes the longitude surface coordinate you want to be positioned above, and should be specified as a decimal value in degrees.
    - ➔ Longitude is specified as a negative number for the Western hemisphere and as a positive number for the Eastern hemisphere (“+” sign is not necessary).
  - latitude <*latnumber*>, no default.
    This value describes the latitude surface coordinate you want to be positioned above, and should be specified as a decimal value in degrees.
    - ➔ Latitude is specified as a negative number for the Southern hemisphere, and as a positive number for the Northern hemisphere (“+” sign is not necessary).

CELX equivalent-1:

Based on the **observer:gotolonglat()** method.

Note: The **"observer:gotolonglat()" method** has NO ability to define the axis that points up.

1. Find the target object with name <*string*> to go to and store in "objectname".
   ➔In CELX scripting it is not necessary to select "objectname".

```
objectname = celestia:find( <string> )
```

2. Determine the radius of "objectname" and store in "radius".

```
radius = objectname:radius()
```

3. Determine distance in km to goto as: <*radiusdistance*> * "radius" and store in "distance".

```
distance = <radiusdistance> * radius
```

4. Convert <*longnumber*> and <*latnumber*> from degrees to radians by multiplying degrees with math.pi (= 3.14159265) and divide by 180. The LUA "**math.rad()**" function can also be used for this.

```
longitude = math.rad( <longnumber> )
latitude = math.rad( <latnumber> )
```

5. Get observer instance of the active view and go to the "longitude" and "latitude" surface coordinates at the determined "distance" from the surface of "objectname" in <*duration*> seconds.
   If no <*duration*> is given, the default time = 5.0 seconds !!!
   If no <*distance*> is given, the default is 5 times the objects radius.
   If no <*longitude*> is given, the default = 0.
   If no <*latitude*> is given, the default = 0.

```
obs = celestia:getobserver()
obs:gotolonglat(objectname, longitude, latitude, distance, <duration> )
```

6. Wait <*duration*> seconds.

```
wait( <duration> )
```

Sumarized:

```
objectname = celestia:find( <string> )
radius = objectname:radius()
distance = <radiusdistance> * radius
longitude = math.rad( <longnumber> )
latitude = math.rad( <latnumber> )
obs = celestia:getobserver()
obs:gotolonglat(objectname, longitude, latitude, distance, <duration> )
wait( <duration> )
```

CELX equivalent-2:

Based on the **observer:goto(table)** method.

Using this method, many different types of gotos can be performed. The strenth of this method is that the observer orientation can be programmed, so the axis that points up after the goto can also be determined. This method uses positions to goto, instead of longitude, latitude and distance from the object parameters, so that needs some additional calculations. The parameters for the goto must be given in the table:

- parameters.duration: duration (number)
- parameters.from: source position
- parameters.to: target position
- parameters.initialOrientation: source orientation
- parameters.finalOrientation: target orientation
- parameters.startInterpolation:
- parameters.endInterpolation:
- parameters.accelTime:

1. Find and select the target object with name <*string*> to go to and store in “objectname”.

```
objectname = celestia:find( <string> )
celestia:select(objectname)
```

2. Determine radius of “objectname” and store in “radius”.

```
radius = objectname:radius()
```

3. Determine distance in km to goto as: <*radiusdistance*> * “radius” and store in “distance”.

```
distance = <radiusdistance> * radius
```

4. Convert the determined “distance” to “objectname” and <*longnumber*> and <*latnumber*> into positions as follows:

```
uly_to_km = 9460730.4725808
-- Determine and store the current observer position
obs = celestia:getobserver()
frompos = obs:getposition()
-- Determine and store the current position of objectname
objectpos = objectname:getposition()
-- Determine the vector of longitude and latitude:
vec_x = math.cos(math.rad( <longnumber> ))
vec_y = math.sin(math.rad( <longnumber> ))
vec_z = math.sin(math.rad( <latnumber> ))
vec = celestia:newvector(vec_x, vec_y, vec_z)
-- Normalize the length of this vectr to 1
normalvec = vec:normalize()
-- Determine vector from object position to target position
distancevec = distance / uly_to_km * normalvec
-- Determine and store the position to goto
topos = objectpos + distancevec
```

5. The target observer orientation must point from the target position “topos” towards “objectname” with <*upvector*> as up vector:

```
upvec = celestia:newvector( <upvector> )
torot = topos:orientationto(objectpos, upvec)
```

6. Define and initialize the parameter table as follows:

```
parameters={}
```

   a. Determine the number of seconds the goto should take.

```
parameters.duration = <duration>
```

   b. Obtain the current position of the observer.

```
parameters.from = obs:getposition()
```

   c. New position object = topos (step 5)

```
parameters.to = topos
```

   d. Obtain the current orientation of the observer.

```
parameters.initialOrientation = obs:getorientation()
```

   e. New observer orientation = torot (step 6)

```
parameters.finalOrientation = torot
```

   f. Start adjusting the observer orientation after 20 percent of the time to goto.

```
parameters.startInterpolation = 0.2
```

   g. End adjusting the observer orientation after 80 percent of the time to goto.

```
parameters.endInterpolation = 0.8
```

   h. Spend 10% of the time for accelerating and decelerating

```
parameters. accelTime = 0.1
```

7. Goto target position with target orientation in <*duration*> seconds.

```
obs:goto(parameters)
```

8. Wait <*duration*> seconds.

```
wait(parameters.duration)
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
radius = objectname:radius()
distance = <radiusdistance> * radius
uly_to_km = 9460730.4725808
-- Determine and store the current observer position
obs = celestia:getobserver()
frompos = obs:getposition()
-- Determine and store the current position of objectname
objectpos = objectname:getposition()
-- Determine the vector of longitude and latitude:
vec_x = math.cos(math.rad( <longnumber> ))
vec_y = math.sin(math.rad( <longnumber> ))
vec_z = math.sin(math.rad( <latnumber> ))
vec = celestia:newvector(vec_x, vec_y, vec_z)
-- Normalize the length of this vectr to 1
normalvec = vec:normalize()
-- Determine vector from object position to target position
distancevec = distance / uly_to_km * normalvec
-- Determine and store the position to goto
topos = objectpos + distancevec
upvec = celestia:newvector( <upvector> )
torot = topos:orientationto(objectpos, upvec)
parameters={}
parameters.duration = <duration>
parameters.from = obs:getposition()
parameters.to = topos
parameters.initialOrientation = obs:getorientation()
parameters.finalOrientation = torot
parameters.startInterpolation = 0.2
parameters.endInterpolation = 0.8
parameters. accelTime = 0.1
obs:goto(parameters)
wait(parameters.duration)
```

Example:

This example selects the Earth and positions the camera over Seattle, Washington, USA.

CEL:

```
select      { object "Sol/Earth" }
synchronous { }
gotolonglat { time 5 distance 3 up [0 1 0] longitude -122 latitude 47 }
print       { text "Traveling to Seattle, Washington, USA."
              row -3 column 1 duration 5 }
wait        { duration 5 }
print       { text "Hovering over Seattle, Washington, USA."
              row -3 column 1 duration 5 }
wait        { duration 5 }
```

CELX-1 with observer:gotolonglat() method:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:synchronous(earth)
earthradius = earth:radius()
earthdistance = 3 * earthradius
longitude = -122 * math.pi/180
latitude = 47 * math.pi/180
obs:gotolonglat(earth, longitude, latitude, earthdistance, 5.0)
celestia:print("Traveling to Seattle, Washington, USA.", 5.0, -1, -1, 1, 3)
wait(5.0)
celestia:print("Hovering over Seattle, Washington, USA.", 5.0, -1, -1, 1, 3)
wait(5.0)
```

CELX-2 with observer:goto(table) method:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:synchronous(earth)
...
```

CEL: labels

- Set (turn on) or clear (turn off) labeling of one or more items.
- Parameter list:
  - { set <*labelflagsstring*> }, no default.
  - { clear <*labelflagsstring*> }, no default.
- The set and clear string values can be any combination of the following items. Multiple values are specified within a single set of quotes (") by separating them with a vertical bar "|" (i.e. "moons|stars"):
  - ➔ **asteroids**
  - ➔ **comets**
  - ➔ **constellations**
  - ➔ **galaxies**
  - ➔ **moons**
  - ➔ **planets**
  - ➔ **spacecraft**
  - ➔ **stars**

CELX equivalent-1:

Based on the **celestia:hidelabel()** and **celestia:showlabel()** methods.

1. Enable labeling with celestia:showlabel(<*labelflagstring*>) method. <*labelflagstring*> describes the type of labels to enable. Multiple label types can be enabled at once by giving multiple arguments to this method, separated with a comma ",". Must be one of:
   a) **planets, moons, spacecraft, asteroids, comets, stars, galaxies, locations, constellations, constellations, openclusters, nebulae, dwarfplanets, minormoons, globulars.**

```
celestia:showlabel( <labelflagstring> )
```

2. Disable labeling with celestia:hidelabel(<*labelflagstring*>) method. <*labelflagstring*> describes the type of labels to disable. Multiple label types can be disabled at once by giving multiple arguments to this method, separated with a comma ",". Must be one of:
   b) **planets, moons, spacecraft, asteroids, comets, stars, galaxies, locations, constellations, constellations, openclusters, nebulae, dwarfplanets, minormoons, globulars.**

   Multiple features can be enabled at once by giving multiple arguments to this method, separated with a comma ",".

```
celestia:hidelabel( <labelflagstring> )
```

CELX equivalent-2:

Based on the **celestia:setlabelflags()** method.

1. Use the celestia:setlabelflags { <*labelflags*> = boolean } method to enable or disable the rendering of labels.
   Note the curly braces.
   <*labelflags*> is a table which contains the labelflags as keys and booleans as values for each key.
   The labelflag keys must be one of:
   a) **planets, moons, spacecraft, asteroids, comets, stars, galaxies, locations, constellations, constellations, openclusters, nebulae, dwarfplanets, minormoons, globulars.**
   Multiple features can be enabled at once by giving multiple arguments to this method, separated with a comma ",".

```
celestia:setlabelflags{<labelflagstring1> = false, <labelflagstring2> = true }
```

Example:

The following examples demonstrate how to `clear` and `set` labels.

CEL:

```
labels { clear "comets|constellations|galaxies|stars" }
labels {   set "asteroids|moons|planets|spacecraft" }
```

CELX with celestia:hidelabel() and celestia:showlabel() methods:

```
-- Disable labeling with celestia:hidelabel() method.
celestia:hidelabel("comets", "constellations", galaxies", "stars")

-- Enable labeling with celestia:showlabel() method.
celestia:showlabel("asteroids", "moons", "planets", "spacecraft")
```

CELX with celestia:setrenderflags() method:

```
-- Enable and disable the rendering of specific labels.
-- Note the curly braces

celestia:setlabelflags{ comets=false, constellations=false, galaxies=false,
stars=false, asteroids=true, moons=true, planets=true, spacecraft=true }
```

CEL: lock

- This command *locks* two objects together. The first object selected is called the **reference** object, and the second object selected is called the **target** object. When they are **lock**ed together as a pair, as you rotate the scene, the target object stays locked to the reference object, and the distance displayed is the distance *between the two objects*.
- In Lock mode, your position remains fixed relative to a line between the reference object and the target object. As the target object moves around the reference object (relatively speaking) so do you. Thus if the target object is Sol, the illuminated fraction of the reference object (its "phase") remains the same because you're moving around the object in sync with Sol.
- Parameter list: nil

CELX equivalent-1:

Based on the **observer:setframe()** method.

1. Find and select the reference object with name <*refstring*> and store in “objectname_ref”.

```
objectname_ref = celestia:find( <refstring> )
celestia:select(objectname_ref)
```

2. Find the target object with name <*tarstring*> which must be locked with “objectname_ref” and store in “objectname_tar”.

```
objectname_tar = celestia:find( <tarstring> )
```

3. Set the coordinate system of the frame of reference to lock with “objectname_tar” as target-object and “objectname_ref” as reference and store in “frame”.

```
frame = celestia:newframe("lock", objectname_ref, objectname_tar)
```

4. Get observer instance of the active view instance and set the coordinate system of the frame of reference to “frame”.

```
obs = celestia:getobserver()
obs:setframe(frame)
```

Sumarized:

```
objectname_ref = celestia:find( <refstring> )
celestia:select(objectname_ref)
objectname_tar = celestia:find( <tarstring> )
frame = celestia:newframe("lock", objectname_ref, objectname_tar)
obs = celestia:getobserver()
obs:setframe(frame)
```

CELX equivalent-2:

Based on the **observer:lock()** method.

1. Find and select the reference object with name <*refstring*> and store in “objectname_ref”.

```
objectname_ref = celestia:find( <refstring> )
celestia:select(objectname_ref)
```

2. Find the target object with name <*tarstring*> which must be locked with “objectname_ref” and store in “objectname_tar”.

```
objectname_tar = celestia:find( <tarstring> )
```

3. Get observer instance of the active view and activate lock-mode on the axis from “objectname_tar” to “objectname_ref” selected object.
   Lock-mode is the same as setting the frame of reference to lock with “objectname_tar” as target-object and “objectname_ref” as reference.

```
obs = celestia:getobserver()
obs:lock(objectname_tar)
```

Sumarized:

```
objectname_ref = celestia:find( <refstring> )
celestia:select(objectname_ref)
objectname_tar = celestia:find( <tarstring> )
obs = celestia:getobserver()
obs:lock(objectname_tar)
```

Example:

The example below will maintain your position with respect to the center of the Earth, and keep both the Sun (Sol) and the Earth at a fixed location in the camera view. It's the same as if you typed the following key sequence on your keyboard: [**3**] key → [**F**] key → [**0**] key (zero) → [**shift** + **:**] keys.

CEL:

```
select { object "Sol/Earth" }
follow { }
select { object "Sol" }
lock   { }
```

CELX with the observer:lock() method:

```
-- Activate Lock-mode on target object.

objectname_ref = celestia:find("Sol/Earth")
celestia:select(objectname_ref)
objectname_tar = celestia:find("Sol")
obs = celestia:getobserver()
obs:lock(objectname_tar)
```

CELX with the observer:setframe() method:

```
-- Set the coordinate system of the frame of reference to lock.

objectname_ref = celestia:find("Sol/Earth")
celestia:select(objectname_ref)
objectname_tar = celestia:find("Sol")
frame = celestia:newframe("lock", objectname_ref, objectname_tar)
obs = celestia:getobserver()
obs:setframe(frame)
```

CEL: lookback

- Change the current camera view by 180 degrees (like a rear-view mirror).
- Parameter list: nil

CELX equivalent:

Based on the **observer:rotate()** method.

1. Define the axis up_vector for rotation and store in “up_vec”.

```
up_v = celestia:newvector(0,1,0)
```

2. Create new rotation around “up_vec” axis by 180 degrees = math.pi = 3.14159265 radians and store in “lookback”.

```
lookback = celestia:newrotation(up_v, math.pi)
```

3. Get observer instance of the active view and rotate observer according the new created rotation “lookback”.

```
obs = celestia:getobserver()
obs:rotate(lookback)
```

Sumarized:

```
up_v = celestia:newvector(0,1,0)
lookback = celestia:newrotation(up_v, math.pi)
obs = celestia:getobserver()
obs:rotate(lookback)
```

Example:

See the above CELX equivalent also as the CELX example for CEL: lookback { }.

CEL:

```
lookback { }
```

CELX with the observer:rotate() method:

```
up_v = celestia:newvector(0,1,0)
lookback = celestia:newrotation(up_v, math.pi)
obs = celestia:getobserver()
obs:rotate(lookback)
```

CEL: mark

- This command marks the <*objectstring*> with the specified <*symbolstring*> of the specified <*sizenumber*> and <*colorvector*>.
- Parameter list:
  - object <*objectstring*>, no default.
    Name of the object to be marked.
  - size <*sizenumber*>, default = 10.0.
    Diameter, in pixels, of the symbol.
  - color <*colorvector*>, default = [1 0 0] – Red.
    Defines the color of the symbol. The three number values define the strength of Red, Green and Blue (RGB) respectively. Allowable values are from 0 to 1, with decimals being specified with a leading 0, such as 0.5. Any value over 1 is handled the same as if it were specified as 1.
  - symbol <*symbolstring*>, default = "diamond".
    Defines what shape of symbol to mark the object with, which must be one of the following string values:
    - ➔ **`diamond`**
    - ➔ **`plus`**
    - ➔ **`square`**
    - ➔ **`triangle`**
    - ➔ **`x`**

CELX equivalent-1:

Based on the **celestia:mark()** method.

Note: It is not possible with this method to define the size, color and symbol of the marker. The **object:mark()** method should be used instead (see CELX equivalent-2).

1. Find an object with name <*objectstring*> which must be marked and store in "objectname".

```
objectname = celestia:find( <objectstring> )
```

2. Mark the specified object.

```
celestia:mark( objectname )
```

Sumarized:

```
objectname = celestia:find( <objectstring> )
celestia:mark( objectname )
```

CELX equivalent-2:

Based on the **object:mark()** method.

1. Find an object with name <*objectstring*> which must be marked and store in "objectname".

```
objectname = celestia:find( <objectstring> )
```

2. Place a marker on "objectname", according the specified parameters:
   <*colorstring*>: The color to be used for the marker. Default is lime.
   a) Either use HTML-style hex color strings such as "#00ff00" (lime), or some predefined color names like: "**red**", "**green**", "**yellow**", "**blue**", "**white**", "**black**", etc.. A list of supported color names can be found on the HTML Color Names page.
   <*symbolstring*>: Name of symbol to be used for the marker. Default is "diamond".
   b) The available marker symbols are:
   "**diamond**", "**triangle**", "**square**", "**plus**", "**x**", "**filledsquare**",
   "**leftarrow**", "**rightarrow**", "**uparrow**", "**downarrow**", "**circle**", "**disk**".
   <*sizenumber*>: Size of the marker in pixels. Default is 10.
   <*opacitynumber*>: A value from 0 to 1 that gives the opacity of marker.
   The Default is 1.0, completely opaque.
   <*labelstring*>: A text label which can be added to the marker.
   The default is the empty string, indicating no marker label.
   <*boolean_occludable*>: A boolean flag indicating whether the marker will be hidden by objects in front of it. The default is true !!!
   Note: In previous Celestia releases (version 1.5.1 and older), the marker was NOT hidden by objects in front of it, which is NOT the default in Celestia version 1.6.0 and later !!!
   Note: Using <*boolean_occludable*> in Celestia version 1.5.1 and older, will result in a Celestia error !!!

```
objectname:mark( <colorstring>, <symbolstring>, <sizenumber>, <opacity
number>, <labelstring>, <boolean_occludable> )
```

Sumarized:

```
objectname = celestia:find( <string> )
objectname:mark( <colorstring>, <symbolstring>, <sizenumber>, <opacitynumber
>, <labelstring>, <boolean_occludable> )
```

Example:

The following example marks the Earth with a green "x".

CEL:

```
mark   { object "Sol/Earth" size 15 color [0 1 0] symbol "x" }
```

CELX with the celestia:mark() method not completely compatible:

```
-- This example will NOT meet the required size, color and symbol !!!
objectname = celestia:find("Sol/Earth")
celestia:mark(objectname)
```

CELX with the object:mark() method:

```
-- In previous Celestia releases (v151 and older).
objectname = celestia:find("Sol/Earth")
objectname:mark("green", "x", 15.0, 1, "")

-- In newer Celestia releases (v160 and newer).
objectname = celestia:find("Sol/Earth")
objectname:mark("green", "x", 15.0, 1, "", false)
```

CEL: move

- Move the camera at the specified <*vector*> velocity in km/sec for each of the X, Y and Z axes, for the specified <*duration*> in seconds.
  A **wait** command is not necessary after a **move** command.
- Parameter list:
  - duration <*duration*>, no default.
    Number of seconds to take during the move.
  - Speed as defined for the currently active coordinate system, in km/second for each of the X, Y and Z axes in <*vector*> = [< *xspeed* > < *yspeed* > < *zspeed* >], no default.
    Positive and negative <*vector*> values are used to indicate forward and reverse respectively ("+" sign is not necessary for positive values).

CELX equivalent-1:

Based on the **observer:goto(table)** method.

Note: The given positions in the **observer:goto**(**table**) method are expected to be relative to the universal frame-of-reference, while the non table-based goto uses positions relative to the current frame-of-reference.

1. In CELX, positions are stored in millionths of a light year, so you have to convert km to millionths of a light year, using the “uly_to_km” constant.

```
uly_to_km = 9460730.4725808
```

2. Obtain observer instance of the active view and store in “obs”.

```
obs = celestia:getobserver()
```

3. Determine the actual observer position and store in “frompos”:

```
frompos = obs:getposition()
```

4. Determine the position to goto as follows:
   a. Convert vector [<*xspeed*> <*yspeed*> <*zspeed*>] to millionths of a light year and store in “velocityvector”.
   b. Multiply “velocityvector” with <*duration*> seconds and add the result to the actual server position. The result must be stored in “topos”.

```
velocityvector = celestia:newvector( <xspeed>/uly_to_km, <yspeed>/uly_to_km, <zspeed
>/uly_to_km)
topos = frompos + <duration> * velocityvector
```

5. Define and initialize the parameter table as follows:

```
parameters={}
```

   a. Determine the number of seconds the goto should take.

```
parameters.duration = <duration>
```

b. Obtain the current position of the observer = frompos (step 3).

```
parameters.from = frompos
```

c. New position object = topos (step 4)

```
parameters.to = topos
```

d. Obtain the current orientation of the observer.

```
parameters.initialOrientation = obs:getorientation()
```

e. Because CEL: move does not change the observer orientation, the final orientation is the same as the initial orientation.

```
orientation.parameters.finalOrientation = parameters.initialOrientation
```

f. Start adjusting the observer orientation after 0 percent of the time to goto (it's the same orientation)

```
parameters.startInterpolation = 0
```

g. End adjusting the observer orientation after 0 percent of the time to goto (it's the same orientation).

```
parameters.endInterpolation = 0
```

h. Spend 10% of the time for accelerating and decelerating

```
parameters. accelTime = 0.1
```

6. Move the observer <*duration*> seconds towards target position.

```
obs:goto(parameters)
```

7. Wait <*duration*> seconds.

```
wait(parameters.duration)
```

Sumarized:

```
uly_to_km = 9460730.4725808
obs = celestia:getobserver()
frompos = obs:getposition()
velocityvector = celestia:newvector( <xspeed>/uly_to_km, <yspeed>/uly_to_km, <zspeed>/uly_to_km)
topos = frompos + <duration> * velocityvector
parameters = { }
parameters.duration = <duration>
parameters.from = frompos
parameters.to = topos
parameters.initialOrientation = obs:getorientation()
parameters.finalOrientation = parameters.initialOrientation
parameters.startInterpolation = 0
parameters.endInterpolation = 0
parameters.accelTime = 0.1
obs:goto(parameters)
wait(parameters.duration)
```

CELX equivalent-2:

Based on the **observer:goto(table)** method within a predefined function.

1. When you want to use the CELX equivalent for CEL: move more often in your script, it can be handy to predefine a CELX "move_obs" function at the beginning of your script, containing the CELX equivalent-1 code with local function parameters.

```
function move_obs(time, x, y, z)
   local uly_to_km = 9460730.4725808
   local obs = celestia:getobserver()
   local parameters = { }
   parameters.duration = time
   parameters.from = obs:getposition()
   local vector = celestia:newvector(x/uly_to_km, y/uly_to_km, z/uly_to_km)
   parameters.to = parameters.from + parameters.duration * vector
   parameters.initialOrientation = obs:getorientation()
   parameters.finalOrientation = parameters.initialOrientation
   parameters.startInterpolation = 0
   parameters.endInterpolation = 0
   parameters.accelTime = 0.1
   obs:goto(parameters)
   wait(time)
end
```

2. Within the main script, this function can be called many times with just each time ONE simple line of code:

```
<... other CELX script code ...>

move_obs( <duration>, <xspeed>, <yspeed>, <zspeed> )

<... other CELX script code ...>
```

Example:

This example selects the Earth and positions the camera over Seattle, Washington, USA. Then moves the camera for 10 seconds as follows:

- along the X axis at a speed of 1000 km/second.
- along the Y axis at a speed of 2000 km/second.
- along the Z axis at a speed of 1500 km/second.

CEL:

```
select      { object "Sol/Earth" }
synchronous { }
gotolonglat { time 5 distance 3 up [0 1 0] longitude -122 latitude 47 }
print       { text "Traveling to Seattle, Washington, USA."
              row -3 column 1 duration 5 }
wait        { duration 5 }
move        { duration 10  velocity [ 500 1000 750 ] }
```

CELX with the observer:goto(table) method:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:synchronous(earth)
earthradius = earth:radius()
earthdistance = 3 * earthradius
longitude = -122 * math.pi/180
latitude = 47 * math.pi/180
obs:gotolonglat(earth, longitude, latitude, earthdistance, 5.0)
celestia:print("Traveling to Seattle, Washington, USA.", 5.0, -1, -1, 1, 3)
wait(5.0)
uly_to_km = 9460730.4725808
obs = celestia:getobserver()
frompos = obs:getposition()
velocityvector = celestia:newvector( 500/uly_to_km, 1000/uly_to_km, 750/uly_to_km)
topos = frompos + 10 * velocityvector
parameters = { }
parameters.duration = 10
parameters.from = frompos
parameters.to = topos
parameters.initialOrientation = obs:getorientation()
parameters.finalOrientation = parameters.initialOrientation
parameters.startInterpolation = 0
parameters.endInterpolation = 0
parameters.accelTime = 0.1
obs:goto(parameters)
wait(parameters.duration)
```

CELX with the observer:goto(table) method within a function:

```
function move_obs(time, x, y, z)
  local obs = celestia:getobserver()
  local uly_to_km = 9460730.4725808
  local parameters = { }
  parameters.duration = time
  parameters.from = obs:getposition()
  local vector = celestia:newvector(x/uly_to_km, y/uly_to_km, z/uly_to_km)
  parameters.to = parameters.from + parameters.duration * vector
  parameters.initialOrientation = obs:getorientation()
  parameters.finalOrientation = parameters.initialOrientation
  parameters.startInterpolation = 0
  parameters.endInterpolation = 0
  parameters.accelTime = 0.1
  obs:goto(parameters)
  wait(time)
end

--
-- Main script body
--

earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:synchronous(earth)
earthradius = earth:radius()
earthdistance = 3 * earthradius
longitude = -122 * math.pi/180
latitude = 47 * math.pi/180
obs:gotolonglat(earth, longitude, latitude, earthdistance, 5.0)
celestia:print("Traveling to Seattle, Washington, USA.", 5.0, -1, -1, 1, 3)
wait(5.0)
move_obs(10, 500, 1000, 750)
```

CEL: orbit

- Orbit the currently selected object during <*duration*> seconds at <*rate*> speed (in units of degrees per second), using the currently defined Coordinate System, around the specified <*axisvector*>. You must first use the `SELECT` command to select an object, and optionally use the `SETFRAME` command to define a Coordinate System, if one is not currently defined.
- Parameter list:
  - duration <*duration*>, default = 1.0.
    Number of seconds to orbit the object.
  - rate <*rate*>, default = 1.0
    Speed at which to orbit the object, in units of degrees per second. Positive and negative values are used to indicate the direction of orbit ("+" sign is not necessary).
  - axis <*axisvector*>, no default.
    Define which axis to orbit around [x y z]. Set the `X`, `Y`, or `Z` value to 1 for **yes**, 0 for **no**. You may also specify multiple axes.

CELX equivalent-1 for Celestia v1.6.1 and later:

Based on the 1.6.1 observer:orbit() method.

This equivalent orbits the reference object during **about** <*duration*> seconds over **exactly** <*duration*> * <*rate*> degrees.

1. Find and select the object with name <*string*> that must be orbited and store in "objectname".

```
objectname = celestia:find( <string> )
celestia:select(objectname)
```

2. Get observer instance of the active view and store in "obs".

```
obs=celestia:getobserver()
```

3. Make "objectname" the reference object in the "ecliptic" (follow) frame of reference.

```
obs:follow(objectame)
```

4. "duration" = <*duration*> = the duration of the orbit in seconds.

```
duration = <duration>
```

5. <*rate*> is the orbit velocity in degrees per second, which must be transformed in radians per second by multiplying the <*rate*> with math.pi (= 3.14159265) and divide by 180 and stored in "radiansrate". The Lua **math.rad**(<rate>) function can also be used for this.

```
radiansrate = math.rad( <rate> )
```

6. Create a vector <axisvector> for the axis to orbit around.

```
axis_vector = celestia:newvector( <axisvector> )
```

7. Determine the orbit angle in radians by multiplying "duration" with "radiansrate".

```
orbitangle = duration * radiansrate
```

8. Split up the orbit in 30 steps per second and determine the orbit steptime for each single step. The factor 0.75 is an estimate and may depend on the speed of your computer.

```
orbitsteps = 30 * duration
orbitsteptime = 0.75*duration/orbitsteps
```

9. Create new rotation object of split up orbit angle around the specified axis.

```
rot = celestia:newrotation(axis_vector, orbitangle/orbitsteps)
```

10. Actually execute the orbit.

```
for i = 1, orbitsteps do
   obs:orbit(rot)
   wait(orbitsteptime)
end
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
obs=celestia:getobserver()
obs:follow(objectame)
duration = <duration>
radiansrate = math.rad( <rate> )
axis_vector = celestia:newvector( <axisvector> )
orbitangle = duration * radiansrate
orbitsteps = 30 * duration
orbitsteptime = 0.75*duration/orbitsteps
rot = celestia:newrotation(axis_vector, orbitangle/orbitsteps)
for i = 1, orbitsteps do
   obs:orbit(rot)
   wait(orbitsteptime)
end
```

Sumarized as a function:

```
function orbit_object_angle(period, orbitrate, axis)
   local orbitangle = period * orbitrate
   local orbitsteps = 30 * period
   local orbitsteptime = 0.75*period/orbitsteps
   local rot = celestia:newrotation(axis, orbitangle/orbitsteps)
   local obs = celestia:getobserver()
   for i = 1, orbitsteps do
      obs:orbit(rot)
      wait(orbitsteptime)
   end
end

objectname = celestia:find( <string> )
celestia:select(objectname)
obs = celestia:getobserver()
obs:follow(objectame)
radiansrate = math.rad( <rate> )
axis_vector = celestia:newvector( <axisvector> )
orbit_object_angle( <duration> , radiansrate, axis_vector)
```

CELX equivalent-2 for Celestia v1.6.1 and later:

Based on the 1.6.1 observer:orbit() method.

This equivalent orbits the reference object during **exacly** <*duration*> seconds over **about** <*duration*> * <*rate*> degrees.

1. Find and select the object with name <string> that must be orbited and store in "objectname".

   ```
   objectname = celestia:find( <string> )
   celestia:select(objectname)
   ```

2. Get observer instance of the active view and store in "obs".

   ```
   obs=celestia:getobserver()
   ```

3. Make "objectname" the reference object in the "ecliptic" (follow) frame of reference.

   ```
   obs:follow(objectame)
   ```

4. "duration" = <*duration*> = the duration of the orbit in seconds.

   ```
   duration = <duration>
   ```

5. <*rate*> is the orbit velocity in degrees per second, which must be transformed in radians per second by multiplying the <*rate*> with math.pi (= 3.14159265) and divide by 180 and stored in "radiansrate". The Lua **math.rad**(<*rate*>) function can also be used for this.

   ```
   radiansrate = math.rad( <rate> )
   ```

6. Create a vector <*axisvector*> for the axis to orbit around.

```
axis_vector = celestia:newvector( <axisvector> )
```

7. Determine the orbit angle in radians by multiplying "duration" with "radiansrate".

```
orbitangle = duration * radiansrate
```

8. Split up the orbit in 30 steps per second and determine the orbit steptime for each single step. The factor 0.75 is an estimate and may depend on the speed of your computer.

```
orbitsteps = 30 * duration
orbitsteptime = 0.75*duration/orbitsteps
```

9. Create new rotation object of split up orbit angle around the specified axis.

```
rot = celestia:newrotation(axis_vector, orbitangle/orbitsteps)
```

10. Get the elapsed time in seconds since the script has been started and store in "t0".

```
t0 = celestia:getscripttime()
```

11. Actually execute the orbit.

```
while celestia:getscripttime() <= t0 + duration do
   obs:orbit(rot)
   wait(orbitsteptime)
end
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
obs=celestia:getobserver()
obs:follow(objectame)
duration = <duration>
radiansrate = math.rad( <rate> )
axis_vector = celestia:newvector( <axisvector> )
orbitangle = duration * radiansrate
orbitsteps = 30 * duration
orbitsteptime = 0.75*duration/orbitsteps
rot = celestia:newrotation(axis_vector, orbitangle/orbitsteps)
t0 = celestia:getscripttime()
while celestia:getscripttime() <= t0 + duration do
   obs:orbit(rot)
   wait(orbitsteptime)
end
```

Sumarized as a function:

```
function orbit_object_time(period, orbitrate, axis)
   local orbitangle = period * orbitrate
   local orbitsteps = 30 * period
   local orbitsteptime = 0.75*period/orbitsteps
   local rot = celestia:newrotation(axis, orbitangle/orbitsteps)
   local obs = celestia:getobserver()
   local t0 = celestia:getscripttime()
   while celestia:getscripttime() <= t0 + period do
      obs:orbit(rot)
      wait(orbitsteptime)
   end
end

objectname = celestia:find( <string> )
celestia:select(objectname)
obs = celestia:getobserver()
obs:follow(objectame)
radiansrate = math.rad( <rate> )
axis_vector = celestia:newvector( <axisvector> )
orbit_object_time( <duration> , radiansrate, axis_vector)
```

CELX equivalent-3:

Based on an arithmetic sequence of methods.

Note: NO <*axisvector*> can be defined in this CELX equivalent.

1. Find and select the object with name <*string*> that must be orbited and store in "objectname".

   ```
   objectname = celestia:find( <string> )
   celestia:select(objectname)
   ```

2. "duration" = <*duration*> = the duration of the orbit in seconds.

   ```
   duration = <duration>
   ```

3. <*rate*> is the orbit velocity in degrees per second, which must be transformed in radians per second by multiplying the <*rate*> with math.pi (= 3.14159265) and divide by 180 and stored in "radiansrate". The Lua **math.rad**(<*rate*>) function can also be used for this.

   ```
   radiansrate = math.rad( <rate> )
   ```

4. Get observer instance of the active view and store in "obs".

   ```
   obs=celestia:getobserver()
   ```

5. "v1" is the vector from the viewer to "objectname", normalized to length 1.
   "up" is the camera up direction (standard Y-direction).
   "v2" is normal to both "v1" and "up", normalized to length 1.

```
now = celestia:gettime()
v1 = obs:getposition() - objectname:getposition(now)
distance = v1:length()
v1 = v1:normalize()
up = obs:getorientation():transform(celestia:newvector(0, 1, 0))
v2 = v1 ^ up
v2 = v2:normalize()
```

6. Determine the orbit angle in radians by multiplying <*duration*> with "radiansrate".

```
orbitangle = duration * radiansrate
```

7. Orbit in the plane containing "v1" and normal to the up direction in <*duration*> seconds

```
start = celestia:getscripttime()
t = (celestia:getscripttime() - start) / duration
while t < 1 do
   t = (celestia:getscripttime() - start) / duration
   theta = orbitangle * t
   v = math.cos(theta) * v1 + math.sin(theta) * v2
   obs:setposition(center:getposition() + v * distance)
   obs:lookat(center:getposition(), up)
   wait(0)
end
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
duration = <duration>
radiansrate = math.rad( <rate> )
obs=celestia:getobserver()
now = celestia:gettime()
v1 = obs:getposition() - objectname:getposition(now)
distance = v1:length()
v1 = v1:normalize()
up = obs:getorientation():transform(celestia:newvector(0, 1, 0))
v2 = v1 ^ up
v2 = v2:normalize()
orbitangle = duration * radiansrate
start = celestia:getscripttime()
t = (celestia:getscripttime() - start) / duration
while t < 1 do
   t = (celestia:getscripttime() - start) / duration
   theta = orbitangle * t
   v = math.cos(theta) * v1 + math.sin(theta) * v2
   obs:setposition(center:getposition() + v * distance)
   obs:lookat(center:getposition(), up)
   wait(0)
end
```

Sumarized as a function:

```
function orbit_object(period, orbitrate)
   -- Period is the duration of the orbit in seconds
   -- Orbitrate is the orbit velocity in radians per second
   local obs = celestia:getobserver()
   local obsframe = obs:getframe()
   local center = obsframe:getrefobject()
   -- If there's no followed object, use the current selection
   if not center then
      center = celestia:getselection()
   end
   if not center then return end
   -- v1 is the vector from the viewer to the center
   -- up is the camera up direction
   -- v2 is normal to both v1 and up
   local now = celestia:gettime()
   local v1 = obs:getposition() - center:getposition(now)
   local distance = v1:length()
   v1 = v1:normalize()
   local up = obs:getorientation():transform(celestia:newvector(0, 1, 0))
   local v2 = v1 ^ up
   v2 = v2:normalize()
   -- Determine orbit angle in radians
   local orbitangle = period * orbitrate
   -- Orbit in the plane containing v1 and normal to the up direction
   local start = celestia:getscripttime()
   local t = (celestia:getscripttime() - start) / period
   while t < 1 do
      t = (celestia:getscripttime() - start) / period
      local theta = orbitangle * t
      local v = math.cos(theta) * v1 + math.sin(theta) * v2
      obs:setposition(center:getposition() + v * distance)
      obs:lookat(center:getposition(), up)
      wait(0)
   end
end

objectname = celestia:find( <string> )
celestia:select(objectname)
radiansrate = math.rad( <rate> )
orbit_object( <duration>, radiansrate )
```

Example:

The following example orbits Saturn for 12 seconds.

CEL:

```
select { object "Sol/Saturn" }
center { }
goto   { time 3 distance 8 up [ 0 1 0 ] upframe "equatorial" }
wait   { duration 3 }
orbit  { axis [ 0 1 0 ] rate 30 duration 12 }
```

CELX with the Celestia version 1.6.1 observer:orbit() method:

This equivalent orbits the reference object during **about** 12 seconds over **exactly** 12 * 30 = 360°.

```
objectname = celestia:find("Sol/Saturn" )
celestia:select(objectname)
obs = celestia:getobserver()
obs:center(objectname, 1.0)
wait(1.0)
frame = celestia:newframe("equatorial", objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = ( 8 + 1 ) * radius
obs:gotodistance(objectname, distance, 3.0)
wait(3.0)
duration = 12.0
radiansrate = math.rad( 30.0 )
axis_vector = celestia:newvector( 0, 1, 0 )
orbitangle = duration * radiansrate
orbitsteps = 30 * duration
orbitsteptime = 0.75*duration/orbitsteps
rot = celestia:newrotation(axis_vector, orbitangle/orbitsteps)
for i = 1, orbitsteps do
   obs:orbit(rot)
   wait(orbitsteptime)
end
```

CELX with the Celestia version 1.6.1 observer:orbit() method in a function:

This equivalent orbits the reference object during **about** 12 seconds over **exactly** 12 * 30 = 360°.

```
function orbit_object_angle(period, orbitrate, axis)
   local orbitangle = period * orbitrate
   local orbitsteps = 30 * period
   local orbitsteptime = 0.75*period/orbitsteps
   local rot = celestia:newrotation(axis, orbitangle/orbitsteps)
   local obs = celestia:getobserver()
   for i = 1, orbitsteps do
      obs:orbit(rot)
      wait(orbitsteptime)
   end
end

objectname = celestia:find("Sol/Saturn" )
celestia:select(objectname)
obs = celestia:getobserver()
obs:center(objectname, 1.0)
wait(1.0)
frame = celestia:newframe("equatorial", objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = ( 8 + 1 ) * radius
obs:gotodistance(objectname, distance, 3.0)
wait(3.0)
radiansrate = math.rad( 30.0 )
axis_vector = celestia:newvector( 0, 1, 0 )
orbit_object_angle(12.0, radiansrate, axis_vector)
```

CELX with the Celestia version 1.6.1 observer:orbit() method:

This equivalent orbits the reference object during **exacly** 12 seconds over **about** 12 * 30 = 360 °.

```
objectname = celestia:find("Sol/Saturn" )
celestia:select(objectname)
obs = celestia:getobserver()
obs:center(objectname, 1.0)
wait(1.0)
frame = celestia:newframe("equatorial", objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = ( 8 + 1 ) * radius
obs:gotodistance(objectname, distance, 3.0)
wait(3.0)
duration = 12.0
radiansrate = math.rad( 30.0 )
axis_vector = celestia:newvector( 0, 1, 0 )
orbitangle = duration * radiansrate
orbitsteps = 30 * duration
orbitsteptime = 0.75*duration/orbitsteps
rot = celestia:newrotation(axis_vector, orbitangle/orbitsteps)
t0 = celestia:getscripttime()
while celestia:getscripttime() <= t0 + duration do
   obs:orbit(rot)
   wait(orbitsteptime)
end
```

CELX with the Celestia version 1.6.1 observer:orbit() method in a function:

This equivalent orbits the reference object during **exacly** 12 seconds over **about** 12 * 30 = 360 °.

```
function orbit_object_time(period, orbitrate, axis)
   local orbitangle = period * orbitrate
   local orbitsteps = 30 * period
   local orbitsteptime = 0.75*period/orbitsteps
   local rot = celestia:newrotation(axis, orbitangle/orbitsteps)
   local obs = celestia:getobserver()
   local t0 = celestia:getscripttime()
   while celestia:getscripttime() <= t0 + period do
      obs:orbit(rot)
      wait(orbitsteptime)
   end
end

objectname = celestia:find("Sol/Saturn" )
celestia:select(objectname)
obs = celestia:getobserver()
obs:center(objectname, 1.0)
wait(1.0)
frame = celestia:newframe("equatorial", objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = ( 8 + 1 ) * radius
obs:gotodistance(objectname, distance, 3.0)
wait(3.0)
radiansrate = math.rad(30.0)
axis_vector = celestia:newvector( 0, 1, 0 )
orbit_object_time(12.0, radiansrate, axis_vector)
```

CELX with an arithmetic sequence of methods:

```
objectname = celestia:find("Sol/Saturn" )
celestia:select(objectname)
obs = celestia:getobserver()
obs:center(objectname, 1.0)
wait(1.0)
frame = celestia:newframe("equatorial", objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = ( 8 + 1 ) * radius
obs:gotodistance(objectname, distance, 3.0)
wait(3.0)
duration = 12.0
radiansrate = math.rad(30.0)
obs=celestia:getobserver()
now = celestia:gettime()
v1 = obs:getposition() - objectname:getposition(now)
distance = v1:length()
v1 = v1:normalize()
up = obs:getorientation():transform(celestia:newvector(0, 1, 0))
v2 = v1 ^ up
v2 = v2:normalize()
orbitangle = duration * radiansrate
start = celestia:getscripttime()
t = (celestia:getscripttime() - start) / duration
while t < 1 do
   t = (celestia:getscripttime() - start) / duration
   theta = orbitangle * t
   v = math.cos(theta) * v1 + math.sin(theta) * v2
   obs:setposition(objectname:getposition() + v * distance)
   obs:lookat(objectname:getposition(), up)
   wait(0)
end
```

CELX with an arithmetic sequence of methods in a function:

```
function orbit_object(period, orbitrate)
   -- Period is the duration of the orbit in seconds
   -- Orbitrate is the orbit velocity in radians per second
   local obs = celestia:getobserver()
   local obsframe = obs:getframe()
   local center = obsframe:getrefobject()
   -- If there's no followed object, use the current selection
   if not center then
      center = celestia:getselection()
   end
   if not center then return end
   -- v1 is the vector from the viewer to the center
   -- up is the camera up direction
   -- v2 is normal to both v1 and up
   -- Orbit in the plane containing v1 and normal to the up direction
   local now = celestia:gettime()
   local v1 = obs:getposition() - center:getposition(now)
   local distance = v1:length()
   v1 = v1:normalize()
   local up = obs:getorientation():transform(celestia:newvector(0, 1, 0))
   local v2 = v1 ^ up
   v2 = v2:normalize()
   local orbitangle = period * orbitrate
   local start = celestia:getscripttime()
   local t = (celestia:getscripttime() - start) / period
   while t < 1 do
      t = (celestia:getscripttime() - start) / period
      local theta = orbitangle * t
      local v = math.cos(theta) * v1 + math.sin(theta) * v2
      obs:setposition(center:getposition() + v * distance)
      obs:lookat(center:getposition(), up)
      wait(0)
   end
end

objectname = celestia:find("Sol/Saturn" )
celestia:select(objectname)
obs = celestia:getobserver()
obs:center(objectname, 1.0)
wait(1.0)
frame = celestia:newframe("equatorial", objectname)
obs:setframe(frame)
radius = objectname:radius()
distance = ( 8 + 1 ) * radius
obs:gotodistance(objectname, distance, 3.0)
wait(3.0)
radiansrate = math.rad(30.0)
orbit_object(12.0, radiansrate)
```

CEL: preloadtex

- Pre-load the specified texture file from disk into memory.
  Object <*string*> is the name of the object who's texture is to be pre-loaded into graphics card memory.
- Depending on the file size of the texture being loaded, you may want to follow this command with a `WAIT` command. If you are preloading multiple textures, or large textures, you should definitely use a `WAIT` command. The `WAIT` duration will depend on the size of the texture file(s) and may require some testing to get it just right.
- Parameter list:
  - object <*string*>, no default.
    The name of the object who's texture is to be pre-loaded into graphics card memory.

CELX equivalent:

Based on the **object:preloadtexture()** method.

1. Find an object with name <*string*> which must be preloaded and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Preload the texture of "objectname" from disk into memory.

```
objectname:preloadtexture()
```

Sumarized:

```
objectname = celestia:find( <string> )
objectname:preloadtexture()
```

Example:

This example pre-loads the texture file for Mars:

CEL:

```
preloadtex { object "Sol/Mars" }
```

CELX with the object:preloadtexture() method:

```
mars = celestia:find("Sol/Mars")
mars:preloadtexture()
```

CEL: print

- The `PRINT` command allows you to display text in Celestia's display window, defining where the text is positioned and for how long it should be displayed.
- You must follow the print command with a `WAIT` command of a duration equal to or greater than the specified <*duration*> is required.
- Beware: If the text you want printed is longer than the width of the user's window, it will run off the right edge of the window.
- Parameter list:
  - text <*textstring*>, no default.
    The text you want displayed, surrounded by quote marks. You may also use "\n" to generate a CR/LF at any position within the text string.
  - origin <*originstring*>, default = "**`bottomleft`**".
    Starting position of the first character of your text.
    Must be one of the following values:
    - ➔ **`bottom`**
    - ➔ **`bottomleft`**
    - ➔ **`bottomright`**
    - ➔ **`center`**
    - ➔ **`left`**
    - ➔ **`right`**
    - ➔ **`top`**
    - ➔ **`topleft`**
    - ➔ **`topright`**
  - duration <*duration*>, default = 1.0.
    Number of seconds to display the text message.
  - row <*rownumber*>, default = 0.
    Defines the starting display line (vertical position), ***offset*** from the <*originstring*>, on which the text should be displayed. Positive values move the starting row **down**, while negative values move the starting row **up** ("+" sign is not necessary).
  - column <*columnnumber*>, default = 0.
    The column number (horizontal position) where the first character of the text should be displayed. Column 0 (zero) is the left edge of the display screen.

CELX equivalent-1:

Based on the **celestia:flash()** method.

Note: This method has no ability to position the text on a specific position on the screen.

1. Print text on screen, similar to print.
   <*textstring*> A string containing the message to be printed. The string can contain newlines "\n" to break lines, or many special characters encoded in UTF-8.
   <*duration*> is the number of seconds the text is shown. Default is 5.0 sec.

```
celestia:flash( <textstring>, <duration> )
```

CELX equivalent-2:

Based on the **celestia:print()** method.

1. Print text on screen. Celestia supports UTF-8 encoded text strings for showing non-ASCII characters.
   <*textstring*> is a string containing the message to be printed. The string can contain newlines "\n" to break lines, or many special characters encoded in UTF-8.
   <*duration*> is the number of seconds the text is shown. Default is 5.0 sec.
   <*horig*> is the horizontal origin of text: -1 is left, 0 center, 1 right.
   <*vorig*> is the vertical origin of text, -1 is bottom, 0 center, 1 top.
   <*hoffset*> is the horizontal offset relative to the horizontal origin.
   <*voffset*> is the vertical offset relative to the vertical origin.

```
celestia:print( <textstring>, <duration>, <horig>, <vorig>, <hoffset>, <voffset> )
```

Example:

This example prints the message "Hello Universe!" three rows up from the bottom of the display screen, starting two columns from the left edge:

CEL:

```
print { text "Hello Universe!" row -3 column 2 }
wait  { duration 5 }
```

CELX with the celestia:flash() method:

```
celestia:flash("Hello Universe!", 5.0)
wait(5.0)
```

CELX with the celestia:print() method:

```
celestia:print( "Hello Universe!", 5.0, -1, -1, 3, 2)
wait(5.0)
```

Example:

This example prints the word "Hello" five rows up from the bottom of the display screen, starting two columns from the left edge, and on the next line, prints the word "Universe!".

CEL:

```
print { text  "Hello\nUniverse!" origin "left" duration 5 row -5 column 2 }
wait  { duration 5 }
```

CELX with the celestia:flash() method:

```
celestia:flash("Hello\nUniverse!", 5.0)
wait(5.0)
```

CELX with the celestia:print() method:

```
celestia:print( "Hello\nUniverse!", 5.0, -1, -1, 5, 2)
wait(5.0)
```

CEL: renderflags

- SET (turn **on**) or CLEAR (turn **off**) one or more of the items below to be displayed on the screen.
- Parameter list:
  - set < *renderflagstring* >, no default.
  - clear < *renderflagstring* >, no default.

  The SET and CLEAR values can be any combination of the following values.
  - ➔ **`atmospheres ......`** **Atmospheres**
  - ➔ **`automag ..........`** **Auto Magnitude**
  - ➔ **`boundaries .......`** **Constellation boundaries**
  - ➔ **`cloudmaps ........`** **Cloud textures**
  - ➔ **`comettails .......`** **Comet tails**
  - ➔ **`constellations ...`** **Constellation labels**
  - ➔ **`eclipseshadows ...`** **Eclipse shadows**
  - ➔ **`galaxies .........`** **Galaxy rendering**
  - ➔ **`grid ............`** **Earth-based equatorial coordinate grid**
  - ➔ **`markers ..........`** **Markers**
  - ➔ **`nightmaps ........`** **Night-side planet maps**
  - ➔ **`orbits ...........`** **Object orbits**
  - ➔ **`planets ..........`** **Planet labels**
  - ➔ **`pointstars .......`** **[no longer used -- see SET command]**
  - ➔ **`ringshadows ......`** **Ring Shadows**
  - ➔ **`stars ............`** **Stars**

  Multiple values are specified within a single set of quotes (") by separating them with a vertical bar "|" (i.e. `"orbits|stars"`).

CELX equivalent-1:

Based on the **celestia:show()** and **celestia:hide()** methods.

1. Enable one or many rendering features. This method exists for backward compatibility with older scripts; **celestia:setrenderflags()** should be used instead.
   <*renderflagstring*> is a string, describing the render-feature to be enabled. Must be one of:
   a) "**orbits**", "**cloudmaps**", "**constellations**", "**galaxies**", "**planets**", "**stars**", "**nightmaps**", "**eclipseshadows**", "**ringshadows**", "**comettails**", "**boundaries**", "**markers**", "**automag**", "**atmospheres**", "**grid**", "**smoothlines**", "**lightdelay**", "**partialtrajectories**", "**cloudshadows**", "**ecliptic**", "**equatorialgrid**", "**galacticgrid**", "**eclipticgrid**", "**horizontalgrid**".
   Multiple features can be enabled at once by giving multiple arguments to this method.

```
celestia:show( <renderflagstring> )
```

2. Disable one or many rendering features. This method exists for backward compatibility with older scripts; **celestia:setrenderflags()** should be used instead.
   <*renderflagstring*> is a string, describing the render-feature to be enabled.
   Must be one of:
   a) "**orbits**", "**cloudmaps**", "**constellations**", "**galaxies**", "**planets**", "**stars**", "**nightmaps**", "**eclipseshadows**", "**ringshadows**", "**comettails**", "**boundaries**", "**markers**", "**automag**", "**atmospheres**", "**grid**", "**smoothlines**", "**lightdelay**", "**partialtrajectories**", "**cloudshadows**", "**ecliptic**", "**equatorialgrid**", "**galacticgrid**", "**eclipticgrid**", "**horizontalgrid**".
   Multiple features can be enabled at once by giving multiple arguments to this method.

```
celestia:hide( <renderflagstring> )
```

CELX equivalent-2:

Based on the **celestia:setrenderflags()** method.

1. You can use the **celestia:show()** and **celestia:hide()** or the **celestia:setrenderflags()** methods, all have equivalent functionality. The main reason for the existence of this method is as a counterpart of **celestia:getrenderflags()**, e.g. to reset all renderflags to values saved at the beginning of a script (for cleanup).
   "renderflagstable" is a table which contains the <*renderflagstring*> as keys, and booleans as values for each key.
   <*renderflagstring*> must be one of:
   a) "**orbits**", "**cloudmaps**", "**constellations**", "**galaxies**", "**planets**", "**stars**", "**nightmaps**", "**eclipseshadows**", "**ringshadows**", "**comettails**", "**boundaries**", "**markers**", "**automag**", "**atmospheres**", "**grid**", "**smoothlines**", "**lightdelay**", "**partialtrajectories**", "**cloudshadows**", "**ecliptic**", "**equatorialgrid**", "**galacticgrid**", "**eclipticgrid**", "**horizontalgrid**".
   Multiple features can be enabled or disabled at once by giving multiple table keys.

```
-- Define and initialize renderflagstable first, before setting renderflags:
renderflagstable = { }
renderflagstable.<renderflagstring1> = true
renderflagstable.<renderflagstring2> = false
-- more renderflag keys may be initialized.
celestia:setrenderflags(renderflagstable)
```

-- OR --

```
-- Shorter notation, but note the curly braces.
celestia:setrenderflags{ <renderflagstring1> = true, <renderflagstring2> = false }
```

Example:

CEL:

```
renderflags { set "automag|atmospheres|nightmaps" }
renderflags { clear "boundaries|galaxies|markers" }
```

CELX with the celestia:show() and celestia:hide() methods:

```
celestia:show("automag", "atmospheres", "nightmaps")
celestia:hide("boundaries", "galaxies", "markers")
```

CELX with the celestia:setrenderflags() method:

```
renderflagstable = { }
renderflagstable.automag = true
renderflagstable.atmospheres =true
renderflagstable.nightmaps = true
renderflagstable.boundaries = false
renderflagstable.galaxies = false
renderflagstable.markers = false
celestia:setrenderflags(renderflagstable)
```

-- OR --

```
-- Shorter notation, but note the curly braces.
celestia:setrenderflags{automag=true, atmospheres=true, nightmaps=true,
                        boundaries=false, galaxies=false, markers=false}
```

CEL: rotate

- Rotates the camera using the currently defined Coordinate System during <*duration*> seconds. You must first use the `SELECT` command to select an object, and optionally use the `SETFRAME` command to set Coordinate System, if it is not currently defined.
- The `ROTATE` command must be followed by a `WAIT` command with a <*duration*> equal to or greater than that of the `ROTATE` command.
- Parameter list:
  - duration <*duration*>, default = 1.0 second.
    The length of time, in seconds, to execute the `rotate` command.
  - rate <*rate*>, default = 0.0.
    Speed at which to rotate the camera. Positive and negative values are used to indicate the direction of rotation.
  - axis <*axisvector*> = [<*x*> <*y*> <*z*>], no default.
    Defines which axis to rotate around. Set the <*x*> , <*y*> or <*z*> value to 1 for YES, 0 for NO. You may also specify multiple axes.

CELX equivalent-1:

Based on the **observer:rotate()** method.

This equivalent rotates the observer during **about** <*duration*> seconds over **exactly** <*duration*> * <*rate*> degrees.

1. Find and select an object with name <*string*> to select and store in “objectname”.

```
objectname = celestia:find( <string> )
celestia:select(objectname)
```

2. Initialize “duration” of the rotation in seconds.

```
duration = <duration>
```

3. Calculate “rotationangle” as the product of “duration” * <*rate*> in units of degrees. <*rate*> is the rotation velocity in degrees per second. The “rotationangle” must be converted to radians by multiplying with math.pi (= 3.14159265) and divide by 180. The Lua **math.rad**(rotationangle) function can also be used for this.

```
rotationangle = math.rad(duration * <rate> )
```

4. Create the axis vector [<*x*> <*y*> <*z*>] for the rotation and store in “axis_vector”.

```
axis_vector = celestia:newvector( <x> , <y> , <z> )
```

5. Get observer instance of the active view and store in “obs”.

```
obs = celestia:getobserver()
```

6. Split up the rotation in 50 steps per second. The factor 0.75 is an estimate and may depend on the speed of your computer.

```
rotsteps = 50 * duration
rotsteptime = 0.75*duration/rotsteps
```

7. Create new rotation object of split up rotation angle over the specified axis.

```
rot = celestia:newrotation(axis_vector, rotationangle/rotsteps)
```

8. Actually execute the rotation.

```
for i = 1, rotsteps do
   obs:rotate(rot)
   wait(rotsteptime)
end
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
duration = <duration>
rotationangle = math.rad(duration * <rate> )
axis_vector = celestia:newvector( <x> , <y> , <z> )
obs = celestia:getobserver()
rotsteps = 50 * duration
rotsteptime = 0.75*duration/rotsteps
rot = celestia:newrotation(axis_vector, rotationangle/rotsteps)
for i = 1, rotsteps do
   obs:rotate(rot)
   wait(rotsteptime)
end
```

Sumarized as a function:

```
function rotate_obs_angle(rottime, rotangle, rotaxis)
   -- Rottime is the duration of the rotation in seconds
   -- Rotangle is the angle to rotate the observer view in radians
   -- Rotaxis is the axis vector (x,y,z) for the rotation
   local obs = celestia:getobserver()
   local rotsteps = 50 * rottime
   local rotsteptime = 0.75*rottime/rotsteps
   local rot = celestia:newrotation(rotaxis, rotangle/rotsteps)
   for i = 1, rotsteps do
      obs:rotate(rot)
      wait(rotsteptime)
   end
end
```

```
objectname = celestia:find( <string> )
celestia:select(objectname)
duration = <duration>
rotationangle = math.rad(duration * <rate> )
axis_vector = celestia:newvector( <x> , <y> , <z> )
rotate_obs_angle(duration, rotationangle, axis_vector)
```

CELX equivalent-2:

Based on the **observer:rotate()** method.

This equivalent rotates the observer during **exactly** <*duration*> seconds over **about** <*duration*> * <*rate*> degrees.

1. Find and select an object with name <*string*> to select and store in "objectname".

```
objectname = celestia:find( <string> )
celestia:select(objectname)
```

2. Initialize "duration" of the rotation in seconds.

```
duration = <duration>
```

3. Calculate "rotationangle" as the product of "duration" * <*rate*> in units of degrees. <*rate*> is the rotation velocity in degrees per second. The "rotationangle" must be converted to radians by multiplying with math.pi (= 3.14159265) and divide by 180. The Lua **math.rad**(rotationangle) function can also be used for this.

```
rotationangle = math.rad(duration * <rate> )
```

4. Create the axis vector [<*x*> <*y*> <*z*>] for the rotation and store in "axis_vector".

```
axis_vector = celestia:newvector( <x> , <y> , <z> )
```

5. Get observer instance of the active view and store in "obs".

```
obs = celestia:getobserver()
```

6. Split up the rotation in 50 steps per second. The factor 0.75 is an estimate and may depend on the speed of your computer.

```
rotsteps = 50 * duration
rotsteptime = 0.75*duration/rotsteps
```

7. Create new rotation object of split up rotation angle over the specified axis.

```
rot = celestia:newrotation(axis_vector, rotationangle/rotsteps)
```

8. Get the elapsed time in seconds since the script has been started and store in "t0".

```
t0= celestia:getscripttime()
```

9. Actually execute the rotation.

```
while celestia:getscripttime() <= t0 + duration do
   obs:rotate(rot)
   wait(rotsteptime)
end
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
duration = <duration>
rotationangle = math.rad(duration * <rate> )
axis_vector = celestia:newvector( <x> , <y> , <z> )
obs = celestia:getobserver()
rotsteps = 50 * duration
rotsteptime = 0.75*duration/rotsteps
rot = celestia:newrotation(axis_vector, rotationangle/rotsteps)
t0= celestia:getscripttime()
while celestia:getscripttime() <= t0 + duration do
   obs:rotate(rot)
   wait(rotsteptime)
end
```

Sumarized as a function:

```
function rotate_obs_time(rottime, rotangle, rotaxis)
   -- Rottime is the duration of the rotation in seconds
   -- Rotangle is the angle to rotate the observer view in radians
   -- Rotaxis is the axis vector (x,y,z) for the rotation
   local obs = celestia:getobserver()
   local rotsteps = 50 * rottime
   local rotsteptime = 0.75*rottime/rotsteps
   local rot = celestia:newrotation(rotaxis, rotangle/rotsteps)
   local t0= celestia:getscripttime()
   while celestia:getscripttime() <= t0 + duration do
      obs:rotate(rot)
      wait(rotsteptime)
   end
end

objectname = celestia:find( <string> )
celestia:select(objectname)
duration = <duration>
rotationangle = math.rad(duration * <rate> )
axis_vector = celestia:newvector( <x> , <y> , <z> )
rotate_obs_time(duration, rotationangle, axis_vector)
```

Example:

The following example selects, follows and centers Earth (so the coordinate system of the frame of reference is set to ecliptic), then rotates the camera to the right (positive **rate** value) along the Z axis of the currently selected object for 5 seconds.

CEL:

```
select { object "Sol/Earth" }
follow { }
center { }
wait   { duration 1 }
rotate { duration 5 rate 10 axis [0 0 1] }
wait   { duration 5 }
```

CELX with the observer:rotate() method, exact angle:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:follow(earth)
obs:center(earth, 1.0)
wait(1.0)
duration = 5.0
rotation_angle = math.rad(duration * 10)
axis_vector = celestia:newvector(0,0,1)
rotsteps = 50 * duration
rotsteptime = 0.75*duration/rotsteps
rot = celestia:newrotation(axis_vector, rotation_angle/rotsteps)
for i = 1, rotsteps do
   obs:rotate(rot)
   wait(rotsteptime)
end
```

CELX with the observer:rotate() method in a function, exact angle:

```
function rotate_obs_angle(rottime, rotangle, rotaxis)
   -- Rottime is the duration of the rotation in seconds
   -- Rotangle is the angle to rotate the observer view in radians
   -- Rotaxis is the axis vector (x,y,z) for the rotation
   local obs = celestia:getobserver()
   local rotsteps = 50 * rottime
   local rotsteptime = 0.75*rottime/rotsteps
   local rot = celestia:newrotation(rotaxis, rotangle/rotsteps)
   for i = 1, rotsteps do
      obs:rotate(rot)
      wait(rotsteptime)
   end
end
```

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:follow(earth)
obs:center(earth, 1.0)
wait(1.0)
duration = 5.0
rotation_angle = math.rad(duration * 10)
axis_vector = celestia:newvector(0,0,1)
rotate_obs_angle(duration, rotation_angle, axis_vector)
```

CELX with the observer:rotate() method, exact time:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:follow(earth)
obs:center(earth, 1.0)
wait(1.0)
duration = 5.0
rotation_angle = math.rad(duration * 10)
axis_vector = celestia:newvector(0,0,1)
rotsteps = 50 * duration
rot = celestia:newrotation(axis_vector, rotation_angle/rotsteps)
rotsteptime = duration/rotsteps
t0= celestia:getscripttime()
while celestia:getscripttime() <= t0 + duration do
   obs:rotate(rot)
   wait(rotsteptime)
end
```

CELX with the observer:rotate() method in a function, exact time:

```
function rotate_obs_time(rottime, rotangle, rotaxis)
   -- Rottime is the duration of the rotation in seconds
   -- Rotangle is the angle to rotate the observer view in radians
   -- Rotaxis is the axis vector (x,y,z) for the rotation
   local obs = celestia:getobserver()
   local rotsteps = 50 * rottime
   local rotsteptime = 0.75*rottime/rotsteps
   local rot = celestia:newrotation(rotaxis, rotangle/rotsteps)
   local t0= celestia:getscripttime()
   while celestia:getscripttime() <= t0 + duration do
      obs:rotate(rot)
      wait(rotsteptime)
   end
end

earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:follow(earth)
obs:center(earth, 1.0)
wait(1.0)
duration = 5.0
rotation_angle = math.rad(duration * 10)
axis_vector = celestia:newvector(0,0,1)
rotate_obs_time(duration, rotation_angle, axis_vector)
```

CEL: select

- Selects the specified `OBJECT` <*string*> in order to perform other commands on it.
- Parameter list:
  - object <*string*>, no default.
    The name of a planet, moon, asteroid, comet, galaxy, spacecraft, etc.. If you are currently positioned **outside** of our Solar System, and want to select an object **inside** of our Solar System, you must include "`Sol/`" (our sun's name) in the <*string*>.

CELX equivalent:

Based on the **celestia:select()** method;

1. Find an object with name <*string*> which must be selected and store in “objectname”.

```
objectname = celestia:find( <string> )
```

2. Use **celestia:select()** method to select “objectname”.

```
celestia:select(objectname)
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:select(objectname)
```

Example-1:

Select the star Aldebaran.

CEL:

```
select { object "Aldebaran" }
```

CELX with the celestia:select() method:

```
aldebaran = celestia:find("Aldebaran")
celestia:select(aldebaran)
```

Example-2:

Select planet Mars when **in** our Solar System.

CEL:

```
select { object "Mars" }
```

CELX with the celestia:select() method:

```
mars = celestia:find("Mars")
celestia:select(mars)
```

Example-3:

Select planet Mars when **in** or **outside** our Solar System.

CEL:

```
select { object "Sol/Mars" }
```

CELX with the celestia:select() method:

```
mars = celestia:find("Sol/Mars")
celestia:select(mars)
```

CEL: set

- Set one of the items listed below to the value specified in the value argument.
- Parameter list:
  - name <*namestring*>, no default.
    Must be one of the following values:
    - ➔ **`MinOrbitSize`**
      This value is used when Celestia's *render orbits* option is turned on. In general, when an object is very distant its orbit path will not be shown. MinOrbitSize is the minimum radius (in pixels) which the orbit path must cover before Celestia will draw it.
    - ➔ **`AmbientLightLevel`**
      Brightness level of Ambient Light, 0.0 to 1.0 is the min to max range. For realism, this should be set to 0.0. Setting it to 1.0 will cause the side of a planet facing away from the Sun to appear as bright as the lit side.
    - ➔ **`FOV`**
      Field of View. Celestia computes this value relative to the size of the display window, in pixels.
    - ➔ **`StarDistanceLimit`**
      The furthest distance at which Celestia will display stars. The default value is 1,000,000, however, as of version 1.3.1, Celestia only uses a distance out to 16,000 light years.
    - ➔ **`StarStyle`**
      This allows you to set how Celestia displays stars on the screen. The StarStyle value must be one of the following:
      - `Fuzzypoints`
      - `Points`
      - `scaleddiscs`
  - value <*valuenumber*> or value <*valuestring*>, default = 0.0 or "".
    The value you want to assign to the name <*string*> parameter.

CELX equivalents:

Based on five different celestia methods.

1. Set the minimum size of an orbit to be rendered.
   The given <*valuenumber*> is the minimum size of an orbit in pixels.

```
celestia:setminorbitsize( <valuenumber> )
```

2. Set the level of ambient light.
   The given <*valuenumber*> is the new level of ambient light and must be between 0 and 1.

```
celestia:setambient( <valuenumber> )
```

3. Set the FOV (Field Of View) for this observer.
   Convert <*valuenumber*> from degrees in radians by multiplying <*valuenumber*> with math.pi (= 3.14159265) and divide by 180 and store in "fov_radians".
   The Lua **math.rad**(<*valuenumber*>) function can also be used for this.
   Then get observer instance of the active view and set the FOV.

```
fov_radians = math.rad( <valuenumber> )
obs = celestia:getobserver()
obs:setfov(fov_radians)
```

4. Set the maximum distance of stars to be rendered.
   The given <*valuenumber*> is the maximum distance.

```
celestia:setstardistancelimit( <valuenumber> )
```

5. Set the rendering style for stars.
   The given <*valuestring*> must be one of "**fuzzy**", "**point**", "**disc**".

```
celestia:setstarstyle( <valuestring> )
```

Example:

CEL:

```
set { name "MinOrbitSize" value 3 }
set { name "AmbientLightLevel" value 0.15 }
set { name "FOV" value 35.5 }
set { name "StarDistanceLimit" value 2000000 }
set { name "StarStyle" value "points" }
```

CELX with 5 different celestia methods:

```
celestia:setminorbitsize(3)
celestia:setambient(0.15)
celestia:getobserver():setfov(math.rad(35.5))
celestia:setstardistancelimit(2000000)
celestia:setstarstyle("points")
```

CEL: setfaintestautomag45deg

- Set the lowest Magnitude of stars to be displayed when Auto-Magnitude is *on*.
- Parameter list:
  - magnitude <*faintestnumber*>, default = 8.5.
    The magnitude at which stars will become visible, when Auto-Magnitude is on.
    The Celestia user interface allows a range from about 1.0 to 15.0.

CELX equivalent:

Based on the **celestia:setfaintestvisible()** and **celestia:setfaintestvisible()** methods.

1. Enable "AutoMag" and set the faintest magnitude of stars at 45°.
   <*faintestnumber*> is the new faintest magnitude of stars.

```
celestia:setrenderflags{automag = true}
celestia:setfaintestvisible( <faintestnumber> )
```

Example:

CEL:

```
setfaintestautomag45deg { magnitude 9.0 }
```

CELX:

```
celestia:setrenderflags{automag = true}
celestia:setfaintestvisible(9.0)
```

CEL: setframe

- Set the currently active Coordinate System (please refer to the Coordinate Systems section earlier in this guide).
- Parameter list:
  - ref <*refstring*>, no default.
    Defines the reference (first) object.
  - target <*tarstring*>, no default.
    Is optional and defines the target (second) object,
    for 2-object coordinate systems, such as `LOCK`.
  - coordsys <*coordstring*>, default "`universal`".
    Must be one of the following values:
    - ➔ **`chase`**
    - ➔ **`ecliptical`**
    - ➔ **`equatorial`**
    - ➔ **`geographic`**
    - ➔ **`lock`**
    - ➔ **`observer`**
    - ➔ **`universal`**
- Coordinate systems are well documented in the available Celestia .Cel Scripting Guide v1-0g by Don Goyette, in the chapter: "Coordinate Systems".

CELX equivalent:

Based on the **celestia:newframe()** and **observer:setframe()** methods.

1. Find and select the reference object with name <*refstring*> and store in "objectname_ref".
   Not needed for frame type "universal".

```
objectname_ref = celestia:find( <refstring> )
celestia:select(objectname_ref)
```

2. Find the target object with name <*tarstring*> which must be locked with "objectname_ref" and store on "objectname_tar".
   Only needed for frames of type "lock".

```
objectname_tar = celestia:find( <tarstring> )
```

3. Create new reference frame and store in "frame".
   <*coordstring*> describs the type of frame and can be one of the following:
   a) "**universal**", "**ecliptic**", "**equatorial**", "**planetographic**", "**observer**", "**lock**", "**chase**", "**bodyfixed**".
   b) In Celestia v1.6.0, the name "**bodyfixed**" will replace "**planetographic**", although for compatibility reasons, the name "**planetographic**" will continue to work.

```
frame = celestia:newframe( <coordstring>, objectname_ref, objectname_tar)
```

4. Get observer instance of the active view and set the coordinate system of the frame of reference to “frame”.

```
obs = celestia:getobserver()
obs:setframe(frame)
```

Sumarized:

```
objectname_ref = celestia:find( <refstring> )
celestia:select(objectname_ref)
objectname_tar = celestia:find( <tarstring> )
frame = celestia:newframe( <coordstring>, objectname_ref, objectname_tar)
obs = celestia:getobserver()
obs:setframe(frame)
```

Example:

The following example sets the Coordinate System to LOCK, which locks the Earth and Moon together on the display.

CEL:

```
setframe { ref "Sol/Earth" target "Sol/Earth/Moon" coordsys "lock" }
```

CELX with the celestia:newframe() and observer:setframe() methods:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
moon = celestia:find("Sol/Earth/Moon")
frame = celestia:newframe("lock", earth, moon)
obs = celestia:getobserver()
obs:setframe(frame)
```

CEL: setorientation

- Sets the camera orientation. If you are attempting to duplicate a position based on a Bookmark or Cel://URL, you will also need to set the proper Coordinate System, position, and other parameters.
- Parameter list-1:
  - angle <*anglenumber*> in degrees, no default.
    This value may be obtained from a Celestia Bookmark, which is stored in the **favorites.cel** located in the Celestia directory.
  - axis <*axisvector*> is the rotation vector [<*xrot*> <*yrot*> <*zrot*>], no default.
    This value may be obtained from a Celestia Bookmark, which is stored in the **favorites.cel** file located in the Celestia directory.
    ➔ <*xrot*>, <*yrot*> and <*zrot*> are the Eular Angle or Angle-Axis representation of the camera's orientation. Think of them as Pitch, Yaw, and Roll in aviation.
- Parameter list-2:
  - ow <*ownumber*>, ox <*oxnumber*>, oy <*oynumber*>, oz <*oznumber*>, represent the `ANGLE` and `AXIS` as stored by a Cel://URL. They are obtained from the following Cel://URL values: `&ow= &ox= &oy=` and `&oz=`. No defaults.

CELX equivalent-1:

Based on Parameter list-1 and the **celestia:newvector()**, **celestia:newrotation()** and **observer:setorientation()** methods.

1. Create new vector, containing the axis of this rotation [<*xrot*> , <*yrot*> , <*zrot*>], and store in “vec”.

```
vec = celestia:newvector( <xrot>, <yrot>, <zrot> )
```

2. Convert the <*anglenumber*> from degrees in radians and store in “angle”: “angle” = math.pi / 180 * <*anglenumber*> (= 3.14159265 / 180 * <*anglenumber*>).
   The LUA "math.rad(<*anglenumber*>)" function can also be used for this.

```
angle = math.rad( <anglenumber> )
```

3. Create new rotation (i.e. a quaternion) of an “angle” about the specified axis of this rotation in “vec”, and store in “rot”.

```
rot = celestia:newrotation(vec, angle)
```

4. Get observer instance of the active view and rotate the observer according the created new rotation in “rot”.

```
obs = celestia:getobserver()
obs:setorientation(rot)
```

Sumarized:

```
vec = celestia:newvector( <xrot>, <yrot>, <zrot> )
angle = math.rad( <anglenumber> )
rot = celestia:newrotation(vec, angle)
obs = celestia:getobserver()
obs:setorientation(rot)
```

CELX equivalent-2:

Based on Parameter list-2 and the **celestia:newrotation()** and **observer:setorientation()** methods.

1. Create new rotation (i.e. a quaternion) from four scalar values and store in rot.
   <*ownumber*>: The OW-component of the new rotation, as a number-values taken from a cel-style URL &ow=.
   <*oxnumber*>: The OX-component of the new rotation, as a number-values taken from a cel-style URL &ox=.
   <*oynumber*>: The OY-component of the new rotation, as a number-values taken from a cel-style URL &oy=.
   <*oznumber*>: The OZ-component of the new rotation, as a number-values taken from a cel-style URL &oz=.

   ```
   rot = celestia:newrotation( <ownumber>, <oxnumber>, <oynumber>, <oznumber> )
   ```

2. Get observer instance of the active view and rotate the observer according the created new rotation in “rot”.

   ```
   obs = celestia:getobserver()
   obs:setorientation(rot)
   ```

Sumarized:

```
rot = celestia:newrotation( <ownumber>, <oxnumber>, <oynumber>, <oznumber> )
obs = celestia:getobserver()
obs:setorientation(rot)
```

Example:

Set the camera orientation according parameterlist-1:

CEL:

```
setorientation { angle 0.945208 axis [ 0.81466 -0.570975 -0.101573 ] }
```

CELX based on parameter list-1:

```
vec = celestia:newvector(0.81466, -0.570975, -0.101573)
angle = math.rad(0.945208)
rot = celestia:newrotation(vec, angle)
obs = celestia:getobserver()
obs:setorientation(rot)
```

Example:

Set the camera orientation according parameterlist-2:

CEL:

```
setorientation { ow 0.090610 ox -0.494683 oy 0.860207 oz -0.084397 }
```

CELX based on parameter list-2:

```
rot = celestia:newrotation(0.090610, -0.494683, 0.860207, -0.084397)
obs = celestia:getobserver()
obs:setorientation(rot)
```

CEL: setposition

- Moves the camera to a specific position in the 3-dimensional universe. If you are attempting to duplicate a position based on a Bookmark or Cel://URL, you will also need to set the proper Coordinate System, orientation, and other parameters. (Also see **seturl**).
- Parameter list-1:
  - base <*basevector*> = [<*xnumber*> <*ynumber*> <*znumber*>], no default. This value may be obtained from a Celestia Bookmark, which is stored in the **favorites.cel** file located in the Celestia directory.
    - ➔ The 1st value <*xnumber*> represents the camera location, along the X axis in light-years.
    - ➔ The 2nd value <*ynumber*> represents the camera location, along the Y axis in light-years.
    - ➔ The 3rd value <*znumber*> represents the camera location, along the Z axis in light-years.
  - offset <*offsetvector*>, no default.
    This value may be obtained from a Celestia Bookmark, which is stored in the **favorites.cel** file located in the Celestia directory.
    - ➔ The 1st value <*xoffsetnum*> represents the offset of the camera location, with respect to the Base <*basevector*> along the X axis in light-years.
    - ➔ The 2nd value <*yoffsetnum*> represents the offset of the camera location, with respect to the Base <*basevector*> along the Y axis in light-years.
    - ➔ The 3rd value <*zoffsetnum*> represents the offset of the camera location, with respect to the Base <*basevector*> along the Z axis in light-years.
- Parameter list-2:
  - x <*xbase64*>, y <*ybase64*>, z <*zbase64*>, represent the **base** and **offset** values as stored by a Cel://URL. No defaults.
    They are obtained from the following Cel://URL values: `?x=` `&y=` and `&z=`.

CELX equivalent-1:

Based on parameter list-1 and the **celestia:newposition(), celestia:newvector()**, **position:addvector()** and **observer:gotolocation()** methods.

1. Create new position object from <*xnumber*>, <*ynumber*> and <*znumber*> and store in "pos". The units of the components of a position object are millionths of a light-year, so you have to convert light-year in a Celestia Bookmark to millionths of a light-year in a CELX position object.

```
pos = celestia:newposition( <xnumber> * 1e6, <ynumber> * 1e6 , <znumber> * 1e6 )
```

2. Create new vector object from <*xoffsetnum*>, <*yoffsetnum*> and <*zoffsetnum*> and store in "vec". The units of the components of a vector object are millionths of a light-year, so you have to convert light-year in a Celestia Bookmark to millionths of a light-year in a CELX vector object.

```
vec = celestia:newvector( <xoffsetnum> * 1e6, <yoffsetnum> * 1e6, <zoffsetnum> * 1e6 )
```

3. Add new position object and new vector object and store in “tarpos”.

```
tarpos = pos:addvector(vec)
```

4. Get observer instance of the active view and goto “tarpos” in zero seconds.

```
obs = celestia:getobserver()
obs:gotolocation(tarpos, 0.0 )
```

Note: In this equivalent, the observer goes to a position releative to the current frame of reference. If the current frame of reference is “universal”, the **observer:setposition()** method may also be used instead of the **observer:gotolocation()** method.

Sumarized:

```
pos = celestia:newposition( <xnumber> * 1e6, <ynumber> * 1e6, <znumber> * 1e6 )
vec = celestia:newvector( <xoffsetnum> * 1e6, <yoffsetnum> * 1e6 ,<zoffsetnum> * 1e6 )
tarpos = pos:addvector(vec)
obs = celestia:getobserver()
obs:gotolocation(tarpos, 0.0 )
wait(0.0)
```

CELX equivalent-2:

Based on parameter list-2 and the **celestia:newposition(base64)** and **observer:gotolocation()** methods.

1. Create new position object from URL-style Base64-encoded values and store in “pos”.
   <*xbase64*>: The X-component of the new vector, as a string-values taken from a cel-style URL ?x=.
   <*ybase64*>: The Y-component of the new vector, as a string-values taken from a cel-style URL &y=.
   <*zbase64*>: The Z-component of the new vector, as a string-values taken from a cel-style URL &z=.

```
pos = celestia:newposition( <xbase64>, <ybase64>, <zbase64> )
```

2. Get observer instance of the active view and goto “pos” in zero seconds.

```
obs:gotolocation(pos, 0.0 )
```

Note: In this equivalent, the observer goes to a position releative to the current frame of reference. If the current frame of reference is “universal”, the **observer:setposition()** method may also be used instead of the **observer:gotolocation()** method.

Sumarized:

```
pos = celestia:newposition( <xbase64>, <ybase64>, <zbase64> )
obs = celestia:getobserver()
obs:gotolocation(pos, 0.0 )
wait(0.0)
```

Example:

Position the camera to a position in space outside of the Milky Way, according parameterlis-1.

CEL:

```
setposition { base [-64132.43  47355.11  196091.57] offset [ 0  0  -1.52e-005 ] }
```

CELX with the celestia:newposition(), celestia:newvector(), position:addvector() and observer:gotolocation() methods, based on parameter list -1:

```
pos = celestia:newposition( -64132.43 * 1e6, 47355.11 * 1e6, 196091.57 * 1e6 )
vec = celestia:newvector( 0, 0, -1.52e-005 * 1e6 )
tarpos = pos:addvector(vec)
obs = celestia:getobserver()
obs:gotolocation(tarpos, 0.0)
wait(0.0)
```

-- OR – (if current frame of reference is “universal”)

```
pos = celestia:newposition( -64132.43 * 1e6, 47355.11 * 1e6, 196091.57 * 1e6 )
vec = celestia:newvector( 0, 0, -1.52e-005 * 1e6 )
tarpos = pos:addvector(vec)
obs = celestia:getobserver()
obs:setposition(tarpos)
wait(0.0)
```

Example:

Position the camera to the same position in space outside of the Milky Way, according parameterlis-2.

Note: The CelURL values in this example are compatible with Celestia version 1.6.0.

CEL:

```
setposition { x "AMDCXoJK/3+IyGgR8f///w" y "BvP2LRdAAAD/n5UGCw" z "VUrGoeQJ+/8DyvenLQ" }
```

CELX with the celestia:newposition(base64) and observer:gotolocation() methods, based on parameter list-2:

```
pos = celestia:newposition("AMDCXoJK/3+IyGgR8f///w","BvP2LRdAAAD/n5UGCw","VUrGoeQJ+/8DyvenLQ" )
obs = celestia:getobserver()
obs:gotolocation(pos, 0.0)
wait(0.0)
```

-- OR – (if current frame of reference is “universal”)

```
pos = celestia:newposition("AMDCXoJK/3+IyGgR8f///w","BvP2LRdAAAD/n5UGCw","VUrGoeQJ+/8DyvenLQ" )
obs = celestia:getobserver()
obs:setposition(pos)
wait(0.0)
```

CEL: setsurface

- Allows you to define an alternate surface texture for the currently selected object. An alternate texture must first be defined in the **`solarsys.ssc`** file for the object in question.
    - For example, if you want to create an alternate surface texture for Earth, called "Earth-2", and it's related texture filename is **earth2.jpg**, you would add the following entry to the **`solarsys.ssc`** file, after the closing brace for the ' "Earth" "Sol" ' entry:

```
AltSurface "Earth-2" "Sol/Earth"
{Texture "earth2.jpg"}
```

- Parameter list:
    - name <*string*>, no default.
      Defines the name of the alternate texture defined in the AltSurface entry of the **`solarsys.ssc`** file, not the name of the texture file itself.

CELX equivalent:

Based on the **observer:setsurface()** method.

1. Get observer instance of the active view and set the currently used surface.
   <*string*> is the name of the surface to be used.

```
obs = celestia:getobserver()
obs:setsurface( <string> )
```

Example-1:

To use the built-in Celestia "limit of knowledge" textures, instead of the interpretive ones, use:

CEL:

```
setsurface { name "limit of knowledge" }
```

CELX with the observer:setsurface() method:

```
celestia:getobserver():setsurface("limit of knowledge")
```

Example-2:

To use the "Earth-2" texture, as mentioned above, use:

CEL:

```
setsurface { name "Earth-2" }
```

CELX with the observer:setsurface() method:

```
celestia:getobserver():setsurface("Earth-2")
```

CEL: setvisibilitylimit

- Set the Magnitude of stars to be displayed, when Auto-Magnitude is off. The Celestia user interface allows a range from about 1.0 to 15.0.
- Parameter list:
  - magnitude <*number*>, default = 6.0.
    Defines the magnitude at which stars will become visible, when Auto-Magnitude is *off*.

CELX equivalent:

Based on the **celestia:setrenderflags()** and **celestia:setfaintestvisible()** methods.

1. Disable "AutoMag" and set the faintest visible magnitude.
   <*number*> is the new faintest magnitude.

```
celestia:setrenderflags{automag = false}
celestia:setfaintestvisible( <number> )
```

Example:

CEL:

```
setvisibilitylimit { magnitude 6.5 }
```

CELX with the celestia:setrenderflags() and celestia:setfaintestvisible() methods:

```
celestia:setrenderflags{automag = false}
celestia:setfaintestvisible(6.5)
```

CEL: seturl

- Move the camera to the location of a saved "location URL" (or Cel://URL), which you capture to the clipboard using the [**Ctrl** + **C**] or [**Ctrl** + **Ins**] keys.
- Parameter list:
  - url <*urlstring*>, no default.
    Defines the Cel://URL to be used.
- **Note**: CELurls for Celestia v1.6.0 and v1.6.1 are of type "ver=3" (last parameter in <*urlstring*>), and not fully compatible with urlstrings of earlier version types. So the results of the **CEL: seturl** command and and equivalent CELX method may vary when using "ver=2" urlstring types in Celestia v1.6.1 (and v1.6.0), compared to earlier versions of Celestia.

CELX equivalent for Celestia version 1.6.1 and later:

Based on the **celestia:seturl()** method.

1. Get an observer instance and make the observer goto the a specified celURL.
   If no observer is precised, the command applies to the current active one.

```
obs=celestia:getobserver()
celestia:seturl( <urlstring>, obs)
```

CELX equivalent for Celestia version 1.6.0 and earlier:

For Celestia version 1.6.0 and earlier, there's no CELX equivalent for this CEL command available. The following CELX coding can be used to integrate the **CEL: seturl** command within a CELX script for these Celesia versions.

Based on the **celestia:createcelscript()** and **celscript:tick()** methods.

1. Define the following function at the beginning of your CELX script:

```
function CEL(source)
   local script = celestia:createcelscript(source)
   while script:tick() do
      wait(0)
   end
end
```

2. Within your CELX script you can now call the **CEL: seturl** command as follows:

```
CEL([[{seturl {url <urlstring> }}]])
```

Example:

Position yourself on top of Mount Everest and watch a very special sunrise.

CEL:

```
seturl {url "cel://SyncOrbit/Sol:Earth/2009-07-21T22:47:04.72721?
x=N1WRszkGAg&y=R5015O+GFA&z=h+mfTNDb2P//////////w&ow=0.551145&ox=0.273737&oy=-
0.643962&oz=0.454554&select=Sol&fov=29.1666&ts=50.0&ltd=0&p=0&rf=20227&lm=1243136&tsrc=0&ver=3"
}
wait { duration 180 }
```

CELX equivalent Celestia version 1.6.1 and later:

```
obs=celestia:getobserver()
celestia:seturl("cel://SyncOrbit/Sol:Earth/2009-07-21T22:47:04.72721?
x=N1WRszkGAg&y=R5015O+GFA&z=h+mfTNDb2P//////////w&ow=0.551145&ox=0.273737&oy=-
0.643962&oz=0.454554&select=Sol&fov=29.1666&ts=50.0&ltd=0&p=0&rf=20227&lm=1243136&tsrc=0&ver=3",
obs)
wait(180)
```

CELX equivalent Celestia version 1.6.0 and earlier:

```
function CEL(source)
   local script = celestia:createcelscript(source)
   while script:tick() do
      wait(0)
   end
end

--
-- Main script body
--

CEL([[{seturl {url "cel://SyncOrbit/Sol:Earth/2009-07-21T22:47:04.72721?
x=N1WRszkGAg&y=R5015O+GFA&z=h+mfTNDb2P//////////w&ow=0.551145&ox=0.273737&oy=-
0.643962&oz=0.454554&select=Sol&fov=29.1666&ts=50.0&ltd=0&p=0&rf=20227&lm=1243136&tsrc=0&ver=3"
}}]])
wait(180)
```

CEL: synchronous

- Orbit the currently selected object in synchronous orbit mode. This activates the **geographic** Coordinate System.
  The **geographic** Coordinate System allows you to remain in a stationary, or geosynchronous orbit above the selected object (not just Earth). As the object rotates below, the camera moves with it, as if it were attached to the object.
- In the **geographic** Coordinate System, the axes rotate *with* the selected object. The Y axis is the axis of rotation – counter-clockwise, so it points north for prograde rotators like Earth, and south for retrograde rotators like Venus. The X axis points from the center of the object to the intersection of its zero longitude meridian and equator. The Z axis (at a right angle to the XY plane) completes the right-handed coordinate system. An object with constant geographic coordinates will thus remain fixed with respect to a point on the surface of the object.
- The **select** command must be used first, to select an object.
- Parameter list: nil

CELX equivalent-1:

Based on the **celestia:newframe()** and **observer:setframe()** methods.

1. Find the target object with name <*string*> to be in synchronous orbit with and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Create new frame of reference to planetographic with "objectname" as reference object and store in "frame".

```
frame = celestia:newframe("planetographic", objectname)
```

3. Get observer instance of the active view and set the coordinate system of the frame of reference to "frame".

```
obs = celestia:getobserver()
obs:setframe(frame)
```

Sumarized:

```
objectname = celestia:find( <string> )
frame = celestia:newframe("planetographic", objectname)
obs = celestia:getobserver()
obs:setframe(frame)
```

CELX equivalent-2 for Celestia version 1.6.0 and later:

Based on the **celestia:newframe()** and **observer:setframe()** methods.

1. Find the target object with name <*string*> to be in synchronous orbit with and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Create new frame of reference to bodyfixed with "objectname" as reference object and store in "frame".

```
frame = celestia:newframe("bodyfixed", objectname)
```

3. Get observer instance of the active view and set the coordinate system of the frame of reference to "frame".

```
obs = celestia:getobserver()
obs:setframe(frame)
```

Sumarized:

```
objectname = celestia:find( <string> )
frame = celestia:newframe("bodyfixed ", objectname)
obs = celestia:getobserver()
obs:setframe(frame)
```

CELX equivalent-3:

Based on the **observer:synchronous()** method.

1. Find the target object with name <*string*> to be in synchronous orbit with and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Get observer instance of the active view and activate synchronous-mode on "objectname".

```
obs = celestia:getobserver()
obs:synchronous(objectname)
```

Synchronous-mode is the same as setting the frame of reference to planetographic (a.k.a. geographic or bodyfixed) with "objectname" as reference object.

Sumarized:

```
objectname = celestia:find( <string> )
obs = celestia:getobserver()
obs:synchronous(objectname)
```

Example:

This example selects Earth and sets the Coordinate System to planetographic / geographic / bodyfixed (**synchronous**).

CEL:

```
select { object "Sol/Earth" }
synchronous { }
```

CELX with the celestia:newframe() and observer:setframe() methods:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
frame = celestia:newframe("planetographic", earth)
obs = celestia:getobserver()
obs:setframe(frame)
```

CELX with the celestia:newframe() and observer:setframe() methods for Celestia version 1.6.0 and later:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
frame = celestia:newframe("bodyfixed", earth)
obs = celestia:getobserver()
obs:setframe(frame)
```

CELX with the observer:synchronous() method:

```
earth = celestia:find("Sol/Earth")
celestia:select(earth)
obs = celestia:getobserver()
obs:synchronous(earth)
```

CEL: time

- Sets the Date and Time using a Julian Date (a decimal number) or a UTC time string, in the format YYYY-MM-DDTHH:MM:SS.SSSSS.
- Parameter list-1:
  - jd <*juliandatenumber*>, Julian date, no default.
    A valid Julian date.
- Parameter list-2:
  - utc <*string*>, Universal time, no default.
    A valid date in UTC format: `YYYY-MM-DDTHH:MM:SS.SSSSS`

- **Note:** To obtain a Julian Date from a normal calendar date, try the U.S. Navy Calendar Date/Time to Julian Date/Time converter, located at: http://aa.usno.navy.mil/data/docs/JulianDate.html.
- **Note:** Within Celestia it is also possible to obtain a Julian Date from a normal calendar date. Select the "Time" option in the menu bar. Next select the option "Set Time …" after which a "Set Simulation Time" window will be displayed. This window contains both Time formats and can be used to convert Julian Date/Time to UTC Date/Time and vice versa.

CELX equivalents:

Based on the **celestia:settime()** method and some date/time conversion methods.

Starting with version 1.5.0, although Celestia still displays UTC on the screen, it uses the TDB time scale internally for everything else, so for CELX scripting !!!

1. For these newer Celestia versions, using the UTC Julian date (as a result of the celestia:tojulianday() method) in the celestia:settime() method, will cause the **WRONG** setting of the simutation time. To set the simulation time, using Julian date for these newer Celestia versions, the 1.5.0 celestia:utctotdb() method should be used instead.
2. For these newer Celestia versions, using the celestia:gettime() method to obtain a Julian date, and convert that to a Calender date/time with this celestia:fromjulianday() method, will cause to obtain the **WRONG** Calender date/time. To obtain the right Calender date/time, using the Julian date from the celestia:gettime() method, for these newer Celestia versions, the 1.5.0 celestia:tdbtoutc() method should be used instead.
3. For more information on TDB, UTC, and how time is used in Celestia, see Celestia/Time Scales.

To convert between Calender date and Julian date, UTC and TDB, the following Celestia methods can be used:

1. Convert a **UTC calendar date** to a **UTC Julian date** and set simulation date/time:
   Celestia version 1.4.1 and older.
   If not specified:
   a. <*monthnumber*> and <*daynumber*> default to one
   b. <*hournumber*>, <*minutenumber*> and <*secondnumber*> default to zero.

```
juliandate = celestia:tojulianday( <yearnumber>, <monthnumber>,
<daynumber>, <hournumber>, <minutenumber>, <secondnumber> )
celestia:settime(juliandate)
```

2. Get simulation date/time and convert a **UTC Julian date** to a **UTC calendar date**:
   Celestia version 1.4.1 and older.
   The returned table "utctable" contains the keys:
   a. utctable.year = <*yearnumber*>
   b. utctable.month = <*monthnumber*>
   c. utctable.day = <*daynumber*>
   d. utctable.hour = <*hournumber*>
   e. utctable.minute = <*minutenumber*>
   f. utctable.seconds = <*secondnumber*>

```
juliandate = celestia:gettime()
utctable = celestia:fromjulianday(juliandate)
```

3. Convert a **UTC calendar date** to a **TDB Julian date** and set simulation date/time:
   Celestia version 1.5.0 and newer.
   If not specified:
   a. <*monthnumber*> and <*daynumber*> default to one
   b. <*hournumber*>, <*minutenumber*> and <*secondnumber*> default to zero.

```
tdbdate = celestia:utctotdb( <yearnumber>, <monthnumber>, <daynumber>,
<hournumber>, <minutenumber>, <secondnumber> )
celestia:settime(tdbdate)
```

4. Get simulation date/time and convert a **TDB Julian date** to a **UTC calendar date**:
   Celestia version 1.5.0 and newer.
   The returned table "utctable" contains the keys:
   a. utctable.year = <*yearnumber*>
   b. utctable.month = <*monthnumber*>
   c. utctable.day = <*daynumber*>
   d. utctable.hour = <*hournumber*>
   e. utctable.minute = <*minutenumber*>
   f. utctable.seconds = <*secondnumber*>

```
tdbdate = celestia:gettime()
utctable = celestia:tdbtoutc(tdbdate)
```

Example:

Examples of how to set the date and time to August 11, 2003 at 9:29:24 UTC.

CEL:

```
time { jd 2452862.89542 }

...

time { utc "2003-08-11T09:29:24.0000" }
```

CELX Celestia version 1.4.1 and older:

```
celestia:settime(2452862.89542)

...

juliandate = celestia:tojulianday(2003, 08, 11, 09, 29, 24.0000)
celestia:settime(juliandate)
```

CELX Celestia version 1.5.0 and newer:

```
utctable = celestia:fromjulianday(2452862.89542)
tdbdate = celestia:utctotdb(utctable.year, utctable.month, utctable.day,
                            utctable.hour, utctable.minute,utctable.seconds)
celestia:settime(tdbdate)

...

tdbdate = celestia:utctotdb(2003, 08, 11, 09, 29, 24.0000 )
celestia:settime(tdbdate)
```

CEL: timerate

- Set the multiplier (speed) at which time changes.
- Parameter list:
  - rate <*number*>, defines the time multiplier (i.e. 100x). Default = 1.0.
    Special values:
    - ➔ 0 = Pause time
    - ➔ 1 = Reset to Real Time
    - ➔ Negative values reverse time

CELX equivalent:

Based on the **celestia:settimescale()** method.

1. Set the timescale (how many seconds of simulation time equal one second of real time). <*number*> is the new timescale.

```
celestia:settimescale( <number> )
```

Example:

This example sets the time multiplier to "1000x faster".

CEL:

```
timerate { rate 1000 }
```

CELX with the celestia:settimescale() method:

```
celestia:settimescale(1000)
```

CEL: track

- Track the currently selected object, which keeps it centered in the display. The select command must be used first, to select an object to be tracked.
- Parameter list: nil

Note: If you want the camera to remain at a constant distance from the object, add a **follow** command *after* the **track** command.
Note: If the currently selected object has **track** enabled, as of Celestia version 1.3.1 you can select a nil object "", followed by a **track** command to cancel tracking the currently selected object.

CELX equivalent:

Start tracking an object, based on the **observer:track()** method.

1. Find the target object with name <*string*> to track and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Get observer instance of the active view and set tracking on "objectname" (i.e. always keep "objectname" centered).

```
obs = celestia:getobserver()
obs:track(objectname)
```

Sumarized:

```
objectname = celestia:find( <string> )
obs = celestia:getobserver()
obs:track(objectname)
```

CELX equivalent:

Stop tracking an object, based on the **observer:track()** method.

1. Get observer instance of the active view and set tracking a nil object.

```
obs = celestia:getobserver()
obs:track(nil)
```

Example:

Release your hold on any currently selected object (CANCEL), select the Earth (SELECT), GOTO the Earth, and then TRACK it. The Earth will begin to recede from you at the speed it actually travels in space, but Celestia will TRACK the Earth by keeping it centered in the display. The code example below demonstrates this, with time sped up by 1000x.

CEL:

```
cancel   { }
select   { object "Sol/Earth" }
goto     { time 3 distance 7 upframe "universal" }
wait     { duration 5 }
track    { }
timerate { rate 1000 }
```

CELX with the observer:track() method:

```
obs = celestia:getobserver()
obs:cancelgoto()
obs:track(nil)
obs:setframe(celestia:newframe("universal"))
earth = celestia:find("Sol/Earth")
celestia:select(earth)
-- The following 2 methods are obsolete, because of methods above
-- frame = celestia:newframe("universal", earth)
-- obs:setframe(frame)
radius = earth:radius()
distance = 7 * radius
obs:gotodistance(earth, distance, 3 )
wait (5)
obs:track(earth)
celestia:settimescale(1000)
```

CEL: unmark

- If an object was previously marked, this command unmarks it.
- Parameter list:
  - object <*string*>, defines the name of the object you want to **unmark**. No default.

CELX equivalent-1:

Based on the **celestia:unmark()** method.

1. Find the object with name <*string*> to be unmarked and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Remove marker on "objectname" using Celestia method.

```
celestia:unmark(objectname)
```

Sumarized:

```
objectname = celestia:find( <string> )
celestia:unmark(objectname)
```

CELX equivalent-2:

Based on the **object:unmark()** method.

1. Find the object with name <*string*> to be unmarked and store in "objectname".

```
objectname = celestia:find( <string> )
```

2. Remove marker on "objectname" using object method.

```
objectname:unmark()
```

Sumarized:

```
objectname = celestia:find( <string> )
objectname:unmark()
```

Example:

This example unmarks Earth.

CEL:

```
unmark { object "Sol/Earth" }
```

CELX with the celestia:unmark() method:

```
earth = celestia:find("Sol/Earth")
celestia:unmark(earth)
```

CELX with the object:unmark() method:

```
earth = celestia:find("Sol/Earth")
earth:unmark()
```

CEL: unmarkall

- This command removes any previously assigned marks from all objects and disables the display of marks.
- Parameter list: nil.

CELX equivalent:

Based on the **celestia:unmarkall()** method.

1. Remove **all** markers.

```
celestia:unmarkall()
```

Example:

Removes any previously assigned marks from all objects.

CEL:

```
unmarkall { }
```

CELX with the celestia:unmarkall() method:

```
celestia:unmarkall()
```

CEL: wait

- Pauses execution of the script for the specified <*number*> in seconds.
- Parameter list:
  - duration <*number*>, number of seconds to pause time. Default 1.0.

CELX equivalent:

Based on the **wait()** function.

1. The function "wait(<*number*>)" is predefined in Celestia to wait <*number*> seconds. **It is special because it returns control to Celestia, which you have to do to avoid blocking Celestia**.

```
wait( <number> )
```

Example:

The following example pauses script execution for 15 ¼ seconds.

CEL:

```
wait { duration 15.25 }
```

CELX with the wait() function:

```
wait(15.25)
```

Handy CELX functions

This chapter contains some handy examples of CELX functions and codes, which can be used within your own CELX scripts to accomplish certain things mentioned in these examples. The examples are based on common used functions and can be modified according your own wishes.

Keyboard interaction

Keyboard interaction is possible in CELX, by using the **callback** for keyboard input, which must have the name "**celestia_keyboard_callback**".

A CELX script can activate the handling keyboard-input by calling:

- "**celestia:requestkeyboard(true)**"

Any keypress will result in a call to "**celestia_keyboard_callback**". This callback can return either TRUE or FALSE, indicating that it either has handled the keypress or that Celestia should continue normal processing for this key. Not returning any value is the same as returning TRUE.

The method should accept one parameter, a string, which holds the character of the key which has been pressed.

For instance, you can use the following example in your CELX script to handle keyboard input:

```
-- Initialize variable to have no content:
last_pressed_key = nil

-- Define functions section:
function celestia_keyboard_callback(key)
   last_pressed_key = key
   return true
end

function get_pressed_key()
   last_pressed_key = nil
   celestia:requestkeyboard(true)
   while true do
      if last_pressed_key ~= nil then
         key = last_pressed_key
         last_pressed_key = nil
         celestia:requestkeyboard(false)
         return key
      end
      wait(0.1)
   end
end

-- You can also add other functions in this section

-- Main CELX script section:

<... other CELX script code ...>
```

```
-- Section within your script, where you want to handle keyboard input:
while true do
   local key = get_pressed_key()
   if key == <keystring1> then
      valid_key = true
      <... specify your own CELX code about what to do if key matches keystring1 ...>
   elseif key == <keystring2> then
      valid_key = true
      <... specify your own CELX code about what to do if key matches keystring2 ...>
   else
      valid_key = false
   end

   if valid_key then
      <... specify your own CELX code about what to do if a valid key has been pressed ...>
   end
   wait(0)
end


<... other CELX script code ...>
```

Call an external script

When you write long scripts or scripts in different sections, at can be usefull to store them in separate script files and call those files from within your main script, for instance as the result of a specific key press in a choice menu.

```
filename = <location and filename string of your CELX script within the Celestia directory>
runscript = loadfile(filename)
runscript()
```

Stop time at a specific point in time

CELX scripting has the ability to automatically stop the time at a certain point in time, without exactly calculating the number of seconds in combination with a certain timerate, as you have to do it in CEL scripting. This can be very handy in case of periods expanding more days, weeks months or even years.

The follwing example sets the datetime at August 13, 2008 at 17:00:00 hours (source). Then the time is sped up to a timescale of 600 and we want to stop the time at August 13, 2008 at 19:00:00 hours (target).

Within CEL scripting you have to calculate the number of seconds between source and target (in this case a simple 7200 seconds) and divide the result bij the timerate (in this case 600), to know how much simulation seconds (result is 12 seconds) will elapse in your script before you can stop the time by setting the timerate to zero.

Within CELX scripting you can insert a loop which reads the current simulation datetime and compares it with the target datetime. This loop will be continued as long as the target datetime is bigger then the current simulation datetime. When the simulation datetime is equal or bigger than the target datetime, the loop will be left, the timescale can be set to zero and for precizion, the datetime is set to the target datetime. (Be aware, especially by using big timescale values, that the simulation datetime can be a bit bigger than the target datetime by leaving the loop).

```
-- set source datetime and timescale
dt=celestia:utctotdb(2008,08,13,17,00,00)
celestia:settime(dt)
celestia:settimescale(600)

< ... other CELX script code, take care about the duration of this code,
  not exceeding the elapsetime between both dates with used timescale ...>

-- set target datetime and loop until that datetime is reached
dt=celestia:utctotdb(2008,08,13,19,00,00)
now=celestia:gettime()
while now < dt do
   now=celestia:gettime()
   wait(0)
end
-- stop the time and for precizion adjust datetime to target datetime
celestia:settimescale(0)
celestia:settime(dt)
```

Zoom in/out function

When you want to zoom in or out in a CEL script, you'll have to program lots op statements after each other to adjust the FOV value each time a bit. Within CELX scripting you can overcome this, by using the following "zoom_obs" function:

```
function zoom_obs(startfov, endfov, zoomtime)
   -- Startfov is the FOV value at the beginning of the zoom in radians
   -- Endfov is the FOV value at the end of the zoom in radians
   -- Zoomtime is the duration of the zoom in seconds
   local obs = celestia:getobserver()
   local zoomsteps = 50*zoomtime
   local fovdelta=(startfov-endfov)/zoomsteps
   local zoomsteptime = zoomtime/zoomsteps
   local zoomfov=startfov
   for i = 1, zoomsteps do
      zoomfov=zoomfov-fovdelta
      obs:setfov(zoomfov)
      wait(zoomsteptime)
   end
end
```

To convert the FOV value from degrees (as shown on the screen) to radians, you can use the LUA function "math.rad()" as follows:

```
fov_target = math.rad( <numberdegrees> )
```

Within your CELX script you can also ask for the current FOV value of the observer with the following observer method:

```
fov_source = celestia:getobserver():getfov()
```

The following example shows how so set the FOV value of the observer to 25 degrees, then select and center Mars in 2 seconds and zoom in on Mars within 5 seconds, to an observer FOV value of 1 degrees:

```
fov_source = math.rad(25)
fov_source = celestia:getobserver():setfov(fov_source)
mars = celestia:find("Sol/Mars")
celestia:select(mars)
obs = celestia:getobserver()
obs:center(mars, 2.0)
wait(2.0)
fov_target = math.rad(1)
zoom_obs(fov_source, fov_target, 5.0)
```

Smoothly adjust the Ambient light level

When you want the ambient light level to be smoothly adjusted during a number of seconds in a CEL script, you'll have to program lots op statements after each other to adjust this ambient light level each time a bit. Within CELX scripting you can overcome this, by using the following "ambient_cel" function:

```
function ambient_cel(startambient, endambient, ambienttime)
   -- Startambient is the ambient light level at the beginning
   -- Endambient is the ambient light level at the end
   -- Ambienttime is the duration in seconds to change the ambient level
   local ambientsteps = 10*ambienttime
   local ambientdelta=(startambient-endambient)/ambientsteps
   local ambientsteptime = ambienttime/ambientsteps
   local ambient=startambient
   for i = 1, ambientsteps do
      ambient=ambient-ambientdelta
      celestia:setambient(ambient)
      wait(ambientsteptime)
   end
end
```

Within your CELX script you can also ask for the current Ambient light level with the following Celestia method:

```
ambient_source = celestia:getambient()
```

The following example shows a Total Moon Eclipse. Because the Moon is not completely dark during Totality (the Sunlight is refracted through the Earth's atmosphere and intersects the Earth's umbra to give the Moon a red hue), you can simulate this by adjusting the ambient light level just before and after Totality as follows:

Your script can contain the following code to show the Total Moon Eclipse, with Ambiant light level being adjusted just before and after Totality (mind to program the "seturl" command to ONE line of code, which is not possible within this document):

Using Celestia v160 or earlier:

```
-- Besides the "ambient_cel" function above, in this example you also have
to
-- define the "CEL" function when using Celestia v160 or earlier, at the
-- beginning of your script, so you can insert a CEL "seturl" command
-- within your script to start at the right position and the right time:

function CEL(source)
   local script = celestia:createcelscript(source)
   while script:tick() do
      wait(0)
   end
end


--
-- Total Moon Eclipse, having the Moon not completely dark during Totality
--

CEL([[{seturl {url "cel://PhaseLock/Sol:Earth/Sol:Earth:Moon/2007-08-
28T08:52:09.94852?x=Vl9dEAediHEO&y=GKBlV8Lvrv//////////w&z=XsW6mgf8gccG&ow=
0.527075&ox=0.000134&oy=0.849813&oz=0.003149&select=Sol:Earth:Moon&fov=1.401
795&ts=0.000000&ltd=0&p=0&rf=19971&lm=51200&ver=2" }}]])

celestia:setambient(0.0)
celestia:settimescale(600)
ambient_cel(0.0,0.15,9.0)
wait(1.75)
ambient_cel(0.15,0.0,9.0)
wait(2.0)
```

Using Celestia v161 or later:

```
--
-- Total Moon Eclipse, having the Moon not completely dark during Totality
--

obs=celestia:getobserver()
celestia:seturl("cel://PhaseLock/Sol:Earth/Sol:Earth:Moon/2007-08-
28T08:52:09.94852?x=Vl9dEAediHEO&y=GKBlV8Lvrv//////////w&z=XsW6mgf8gccG&ow=
0.527075&ox=0.000134&oy=0.849813&oz=0.003149&select=Sol:Earth:Moon&fov=1.401
795&ts=0.000000&ltd=0&p=0&rf=19971&lm=51200&ver=2", obs)
celestia:setambient(0.0)
celestia:settimescale(600)
ambient_cel(0.0,0.15,9.0)
wait(1.75)
ambient_cel(0.15,0.0,9.0)
wait(2.0)
```

Recommended CELX links

The WIKI book: "Celestia/Celx Scripting/CELX Lua methods**"** about LUA and CELX scripting by: Selden Ball, Jr., Chris Laurel and Vincent Giangiulio.

CELX scripting is also addressed on the Celestia forum. The Celestia scripting page is moderated by Selden Ball Jr. and contains many issues about writing scripts for Celestia in LUA and the CEL system.

Besides a link to the downloadable Word document about CEL scripting, Don Goyette's Celestia Scripting Resources website also contains many downloadable information examples and functions about CELX scripting.

On the Celestia motherlode many CEL and CELX scripts can be found and downloaded. Some of those scripts can be used as an example for writing your own scripts. Especially recommended is the Visual Celx Guide by Clive Pottinger, which privides syntax, explanations, sample code and a working demonstration for each of the methods available to CELX scripts. Also recommended is the LUA Superset Library by G. Anthony Davis. This LUA script library supersets most Celestia functionality, making it easy to write a script quickly for novice and advanced users.

The website "CELX in Celestia" by Harald Schmidt contains a summary that describes the support for Lua/CELX-scripting in Celestia and consists mostly of the CELX API.

CELX scripting is based on the LUA object oriented programming Language. The principles of LUA are beyond the scope of this document, but can be found on the internet at the following address: http://www.lua.org/manual/5.1/.

Dedications

This document contains many examples and explanations from other available documentation, especially:

- Celstia CELX scripting WIKI documentation about LUA and CELX scripting by:
  - Selden Ball, Chris Laurel and Vincent Giangiulio. Thank you all.
- Cel_Script_Guide_v1-0g.doc about CEL scripting by:
  - Don Goyette and the Celestia developers and forum members, with editing by Selden Ball, Jr. and other Celestia forum members. Thank you all.

Also special thanks to:

Chris Laurel and the Celestia Development team:
Without the beauty of Celestia, this “CELX Scripting for Celestia (How to migrate from CELX scripting)” document could not have been possible at all.

Vincent Giangulio, for his help on some specific CEL to CELX scripting issues.

Ulrich Dickmann a.k.a. "Adirondack", for his initial examples about script initialization and keyboard hanling techniques, which has been further enhanced in this document.

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.