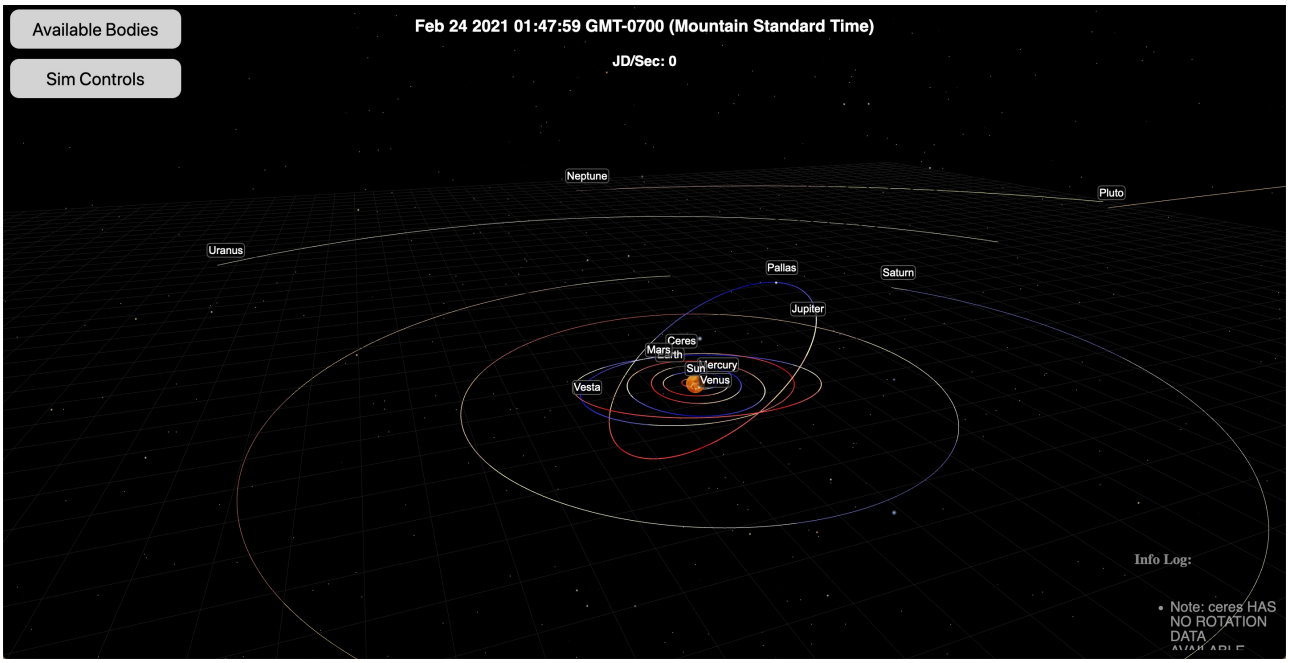
Aether-Capstone-19-20

This repository is for the 2019-2020 CU Boulder Computer Science Capstone project sponsored by NASA/JPL.



About Aether

Aether is an interactive 3D rendition of the solar system which runs inside your web browser. Aether is built using the SPICE system from JPL/NAIF, allowing it to visualize object trajectories over time with high scientific accuracy. In addition, it also visualizes the rotations and radii of major celestial bodies.

The motivation behind our project is to give NASA mission designers an easy way to visualize and
compare the trajectories of different celestial bodies and spacecraft. Therefore, Aether's core design is centered around flexibility and extensibility. Users may create custom simulations with
any origin (e.g Solar System Barycenter, Pluto Barycenter, Mars, etc.) In addition, users may upload
new binary SPK kernels to Aether, allowing for custom trajectories to be visualized.

Aether has two main components, the backend and frontend. The backend consists of a REST server which provides data (positions for trajectories, mass/radii, rotation equations, etc. from SPICE kernels) to the frontend for visualization. The frontend consists of a web-based 3D visualization
written in Javascript using the Spacekit.js and THREE.js frameworks.

Recommended System Requirements

- At least 4 GB free space on the filesystem (default SPICE kernel set is ~3 GB)
- 8 GB RAM (recommended)

- Dedicated GPU (highly recommended for smooth frame-rate, but not required)

Installation

To install Aether on your computer, first clone the repository or obtain an archive of the source code.

Backend Dependencies

Installation of the backend requires the docker command-line interface. Installing docker depends on
your OS.

Note that docker requires virtualization is enabled within the BIOS.

Installing Docker

Debian/Ubuntu

On Debian-based Linux distros, docker can be easily installed via `apt`. Run the following commands
to install it...

```
sudo apt install docker.io
sudo systemctl start docker
sudo systemctl enable docker
```

This installs docker and enables the docker daemon so that it runs by default.

Windows, Mac, other Linux distributions

For Windows and Mac, Docker Desktop may be installed by following the provided instructions here (https://docs.docker.com/engine/install/).

For non-Debian-based Linux distros, docker provides package files for installation, which can be found here (https://docs.docker.com/engine/install/).

Frontend Dependencies

Installing the frontend requires both `node` and `npm` to be installed first.

Installing Node.js and Node package manager

To install Node.js and npm on both Windows and Mac, download the installer here (https://nodejs.org/en/download/)

For Linux, install Node.js and npm via snap or a package manager. Instructions can be found here (https://nodejs.org/en/download/package-manager/).

LINUX AND MAC USERS, READ THIS:

The rest of the installation has been automated. Run `install.sh` in your
terminal to automatically build the backend docker image, create the docker container and install required Node.js packages. After running this script, skip to Running the Application

below.

WINDOWS USERS

We're sorry but you'll have to do this manually. Please follow the instructions in the next two sections.

Note: docker commands may require super-user/administrator privileges

Building the Docker Image and Container

Once docker has been installed on your machine, open a terminal and navigate to the `Aether-Capstone-19-20` directory. Run the following commands to build the backend docker image...

```
cd backend/
docker build -t aether-backend:v1 .
```

Note: Building the docker image may take ~15 minutes depending on your internet speed. This is because the docker must download approximately 3 GB of SPICE kernels from JPL/NAIF.

After the image is finished being built, run the following command to create the container...

```
docker create -it --name="Aether-Backend" -p 5000:5000 aether-backend:v1
```

Installing Frontend Node Packages

Once Node.js and npm are installed on your computer, once again open a terminal and navigate to the `Aether-Capstone-19-20` directory. Run the following commands to install the frontend dependencies...

```
cd frontend/
npm install
```

Running the Application

LINUX AND MAC USERS, READ THIS:

Running the application has been automated. Run `run.sh` in your terminal to start the application, then connect to localhost:8080 from your web browser.

WINDOWS USERS:

The application must be run manually, please continue reading this section.

Start the backend

To run the backend, use the following command to star the docker container you built previously...

```
docker start Aether-Backend
```

Start the frontend

From the `Aether-Capstone-19-20` directory, run the following command to start the frontend...

```
node frontend/server.js
```

Now that both the backend and frontend are both running, connect to localhost:8080 from your web
browser.

Stopping the application

To stop the backend docker container, run the following command...

```
docker stop Aether-Backend
```

To stop the Node.js web server, simply use `Ctrl+C` from the terminal where the Node server is running.

Aside - backend debug mode

If you wish to view a log of API requests or modify files on the container, read on.
Perform actions in this section at your own risk, modifying files within the backend container may break the application. A familiarity with docker is strongly recommended for debugging.

To view the current log of API requests within the backend, run the following command from a separate terminal...

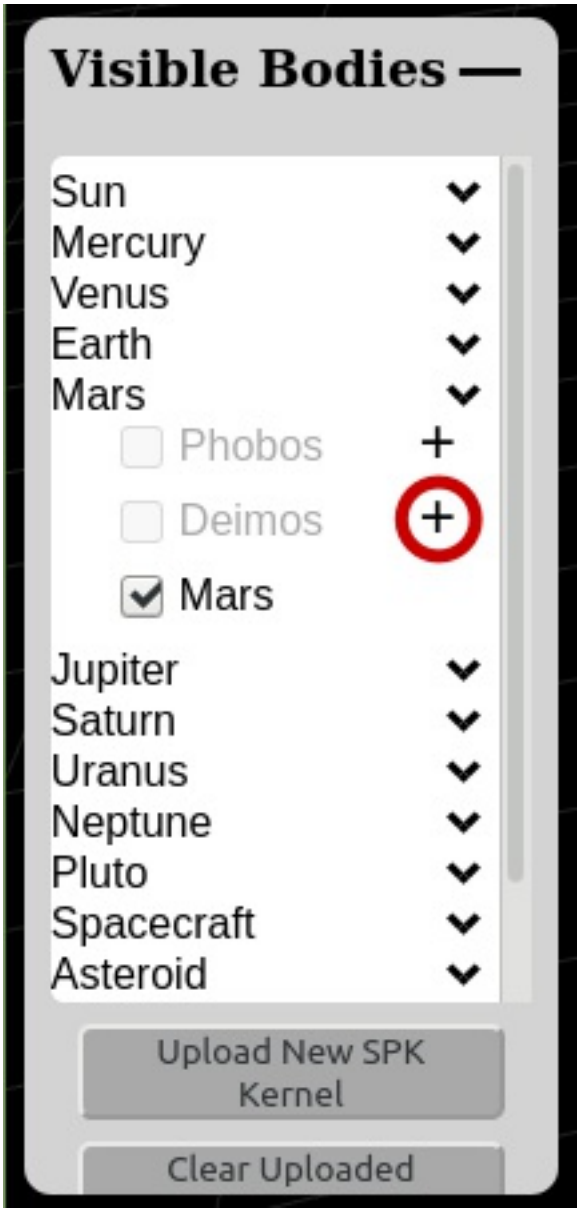
```
docker attach Aether-Backend
```

To modify files on the backend, you must rebuild the image and container after modifying the files
locally. See installation instructions above for the corresponding commands.

Tutorials

Add new objects to the current simulation

After the simulation initially loads, you will see a box on the left titled "Visible Bodies." Within this menu, you can toggle object visibility, zoom to objects, and load new objects into the simulation.
Click on a category (e.g Sun, Mercury, Spacecraft) to expand it. In this drop-down, any name in black
text is already loaded in the simulation. Any name that is grayed out is available from the backend,
but is not currently loaded into the simulation. Click the '+' on the object you would like to add. The name should change to black text once the data has been retrieved. At this point you can view and
interact with the object.



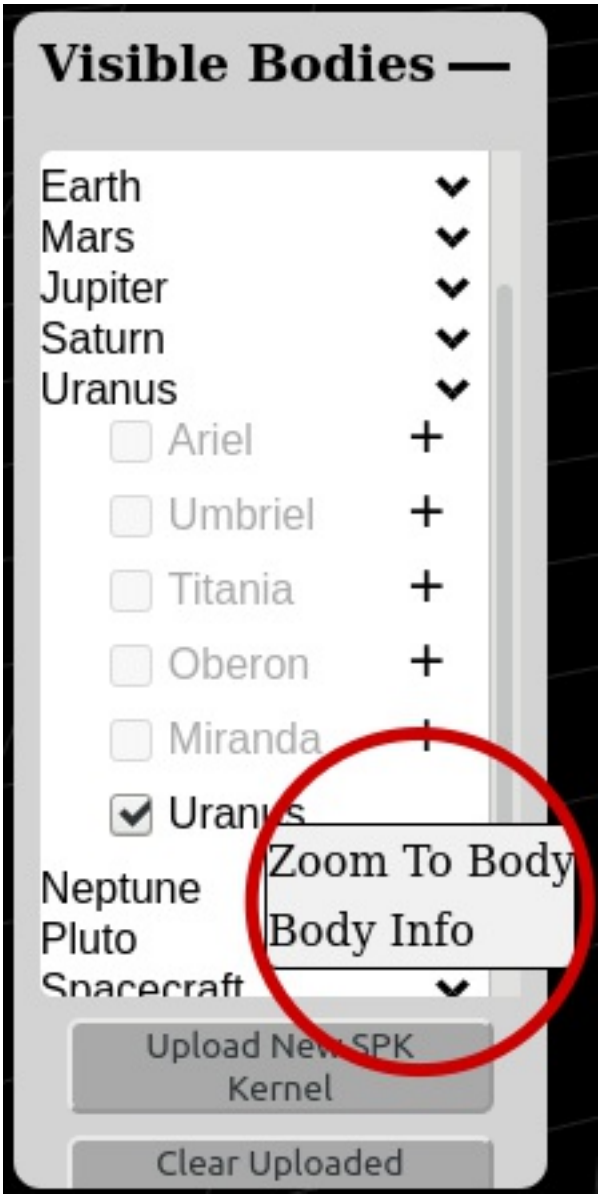
Changing the rate of time and tail length

The simulation's rate of time is displayed at the top-center of the screen underneath the current date.
The rate of time is 0 when the simulation is paused. Within the "Time Controls" menu on the left, either
adjust the slider to change the rate of time, or enter a value in JD/sec manually. This will not resume
the simulation if it is paused.

Similarly, to change the tail length (length of trajectory line for each object, represented in JD prior
to current date). Use the corresponding slider/input form on the "Time Controls" menu.

Zooming to objects and viewing object info

Within the "Visible Bodies" menu, you may right click on any object that is loaded in the current simulation (i.e. if the name's text is black, see the tutorial on adding new objects above). Here you should see options to zoom to the object, or view its info.



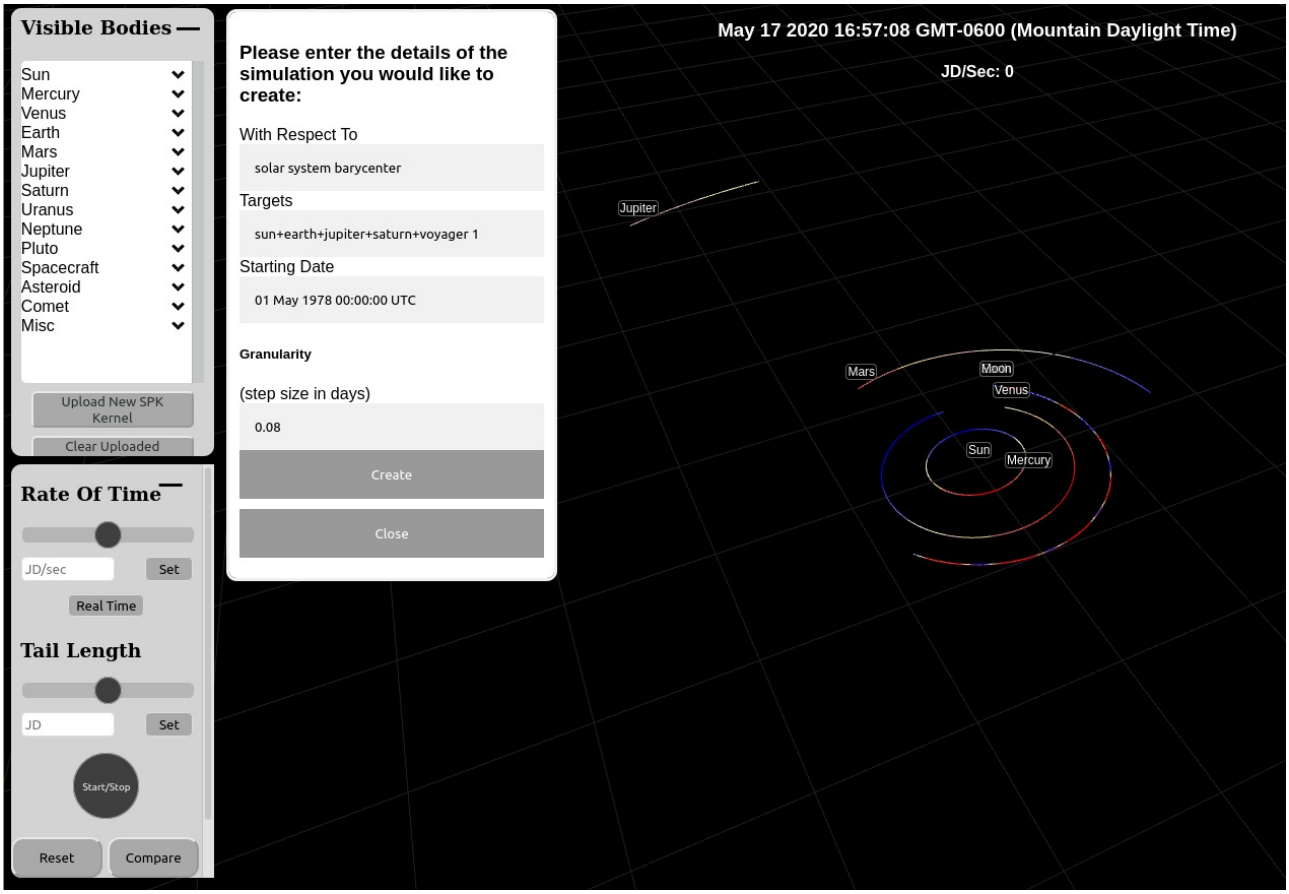
Uploading new binary SPK SPICE kernels

Within the "Visible Bodies" menu on the left, there is a button titled "Upload New SPK Kernel." Clicking this button will open a prompt. Selecting "Browse" will open a file explorer where you can
select a file for uploading. Only binary SPK SPICE kernels (.bsp) may be uploaded, otherwise a warning will appear. Once the file has been uploaded, the new bodies contained in that kernel will
be automatically added to the "Visible Bodies" drop-down in the appropriate category. For example,
if the kernel contains a spacecraft, that object will appear in the Spacecraft drop-down. If an object
is a moon of Jupiter, it will appear in the Jupiter drop-down.

A large repository of publicly-available SPK kernels can be found on the
NAIF website (https://naif.jpl.nasa.gov/naif/data.html).

Creating a new simulation

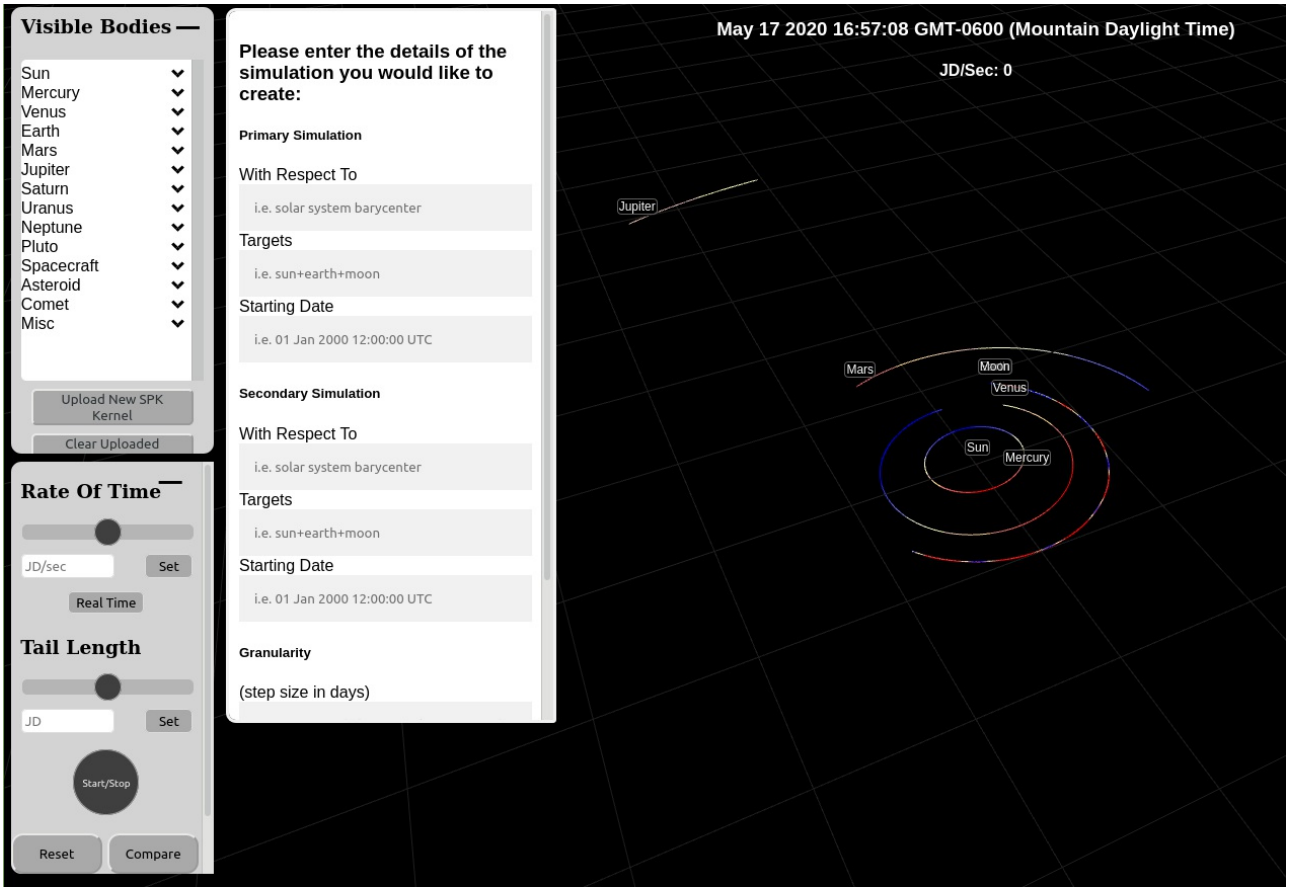
Under the simulation/time controls menu on the lower left, towards the bottom of the menu there is a
"Reset" button. This button allows you to create a new simulation at any starting date/time, with any
object as the origin. Default bodies to load must be specified, but more can always be added via the
"Visible Bodies" menu. See the screenshot below for an example.



Comparing two simulations

Comparing two simulations is quite useful for seeing the differences in object trajectories. For instance, comparing two different proposed trajectories for a new mission. In Aether, one may compare two simulations by clicking the "Compare" button towards the bottom of the simulation/time
controls menu. A form will then pop up (see screenshot below), allowing users to customize the origin,
included objects and start date of each simulation. The specified granularity will be applied to both
simulations, and changing the rate of time or tail length affects both as well.

The result is two side-by-side simulations which are synced with respect to camera orientation, rate
of time and trajectory tail length. See the
comparison (screenshots/clipper_eveega_vs_jup_direct.png)
of two proposed Europa-Clipper trajectories in the screenshots directory for an example.



Comparing two different trajectories with the same SPICE ID

The example linked above compares the trajectories contained in two different SPK kernels for Europa-Clipper.
The issue that arises in these situations is that both objects contain data for the same SPICE ID, which Aether
cannot differentiate between. To accomplish this task, one must first modify the SPICE ID for the conflicting
object in one of the two kernels before uploading them.

Fortunately, modifying the SPICE ID for an object contained in a binary SPK kernel is easy and can be
accomplished using a command-line utility from JPL/NAIF. Download the `bspidmod` program for your OS
from NAIF's utilities repository here (https://naif.jpl.nasa.gov/naif/utilities.html). Afterwards, make the program executable using `chmod` and follow the instructions below...

```
./bspidmod -spki <path-to-SPK>.bsp -idi <object ID to change> -ido <new object ID> -mod target
```

This will change the object ID specified by *-idi* within the given SPK kernel to the ID specified by *-ido*. By default, this will not overwrite the given SPK kernel, but instead create a new one with a suffix of "_out".

Here is an example of the same command for a Europa-Clipper kernel. Note that the spacecraft's
initial ID was -159 and it was converted to -169. This also changes the name of the object to the

specified output ID. After running the command, the file
`15F09_EVEE_L220602_A300115_V1_scpse_out.bsp`
is generated.

```
./bspidmod -spki 15F09_EVEE_L220602_A300115_V1_scpse.bsp -idi -159 -ido -169 -
mod target
```

Uninstall

LINUX AND MAC USERS, READ THIS:

Uninstalling the application has been automated. Run `uninstall.sh` in your terminal to delete the
backend docker container and image.

WINDOWS USERS:

The application must be uninstalled manually, please continue reading this section.

Delete the docker container

First, ensure the docker container exists and is stopped. To do this run `docker ps` and make sure that the Aether-Backend container does not appear as running. Then run `docker ps -a` to make sure the container does indeed exist. After verifying these two things, run the following command to delete the container...

```
docker rm Aether-Backend
```

Delete the docker image

Next, delete the docker image...

```
docker rmi aether-backend:v1
```

After both the container and image have been removed, you may delete your local copy of the repository.

Code Walkthrough

Backend

The primary component of the backend is a REST API which serves trajectory data (position coordinates and their corresponding times) and meta-data for available bodies (radii, mass, rotation
equations, valid time ranges) to the frontend. It also handles uploading new SPK kernels. Below is a
brief description of each endpoint within `aether-rest-server.py` and other files which aid in its execution.

API Endpoints

This is a brief description of each API endpoint, see the docstrings and comments within `aether-rest-server.py` for further details.

Positions

URL Format:

```
/api/positions/<string:ref_frame>/<string:targets>/<string:curVizJd>/<string:curVizJdDe
```

Methods:

```
GET
```

Description:

This endpoint serves position and time data to the frontend for visualization. It is called when the frontend
makes a GET request to the above URL using the specified params. This function can return data for either a single
body/target or multiple. In the case of multiple targets, they should be passed as either names or NAIF IDs
separated by a '+' (e.g. "sun+earth+499+jupiter+europa+pluto+904"). Case does not matter since each target is
converted to lower case by the script. See the source code for more info on what each parameter means.

Example Calls:

This call is an example of an initial positions request that the frontend performs on initialization.
Here, curVizJdDelta represents the current time. The reference frame is "solar system barycenter,"
the granularity is 0.08333 days and the tail length is 200 days.

```
http://0.0.0.0:5000/api/positions/solar system
barycenter/sun+mercury+venus+earth+mars+jupiter+saturn+uranus+neptune+pluto+moo
n/2458988.40703/0.08333333333/200/20
```

This call is an example of an update request for a single object. Here, the tail length is 0 days because only future data is needed.

```
http://0.0.0.0:5000/api/positions/solar system
barycenter/earth/2458989.40703/0.08333333333/0/5
```

Example Return:

Positions are in km from ref_frame, times are in JD.

{"earth": {"cur_time_idx": 0, "info": "Positions (x,y,z) and times (JD) of '
'Earth w.r.t. Solar system barycenter", "positions":
[[-78523771.55936542, -118193740.2804119, -51227083.643394925],
[-78343246.468999, -118295574.65418445, -51271230.094331175],
[-78162566.13038987, -118397171.82165949, -51315273.85150755],
[-77981730.92179076, -118498531.57159914, -51359214.822105244],
[-77800741.22212164, -118599653.6933881, -51403052.91353538]],
"times": [2458989.40703, 2458989.4903633, 2458989.5736967,
2458989.65703, 2458989.7403633]}}

Available Bodies

URL Format:

```
/api/available-bodies/<string:ref_frame>
```

Methods:

```
GET
```

Description:

This endpoint serves metadata about the bodies that the backend has loaded. It is called when the frontend makes
a GET request to the above URL with the specified param. This endpoint is called by the frontend whenever a new
simulation is created. It returns a list of dictionaries (one for each body). Each dictionary specifies the body
name, NAIF ID, valid time ranges, whether or not it is default or uploaded by the user and the body's approx
minimum and maximum speeds (w.r.t ref_frame). Along with that data it also specifies whether or not the body has mass,
rotation and radius data. If any of these are true for a body, the dictionary contains keys for each.

Example Calls:

This call returns body metadata for each of the bodies loaded from the backend's SPICE kernels. Min-max speeds are
calculated with respect to the Solar System barycenter.

```
http://0.0.0.0:5000/api/available-bodies/solar system barycenter
```

Example Return:

Radii are in km, mass is kg, speeds are km/sec. Note that this is just one dictionary from the list.

```
{
'body name': 'mars',
'category': 'mars',
'has mass data': True,
'has radius data': True,
'has rotation data': True,
'is uploaded': False,
'mass': 6.416908682663215e+23,
'max speed': 26.409686994581353,
'min speed': 21.974733146673085,
'radius': [3396.19, 3396.19, 3376.2],
'rotation data': {'dec': 52.8865,
'dec_delta': -0.0609,
'pm': 176.63,
'pm_delta': 350.89198226,
'ra': 317.68143,
'ra_delta': -0.1061},
'spice id': 499,
'valid times': [['1900-01-04 00:00:41.184000',
'2100-01-01 00:01:07.183000']]
}
```


SPK Upload

URL Format:

```
/api/spk-upload/<string:ref_frame>
```

Methods:

```
POST
```

Description:

This endpoint allows users to upload new SPICE SPK kernels to the backend. It is called when the frontend makes
a POST request to the above URL with the specified param. This function simply obtains the uploaded file, makes
sure it's valid, and then registers it with the SPICE subsystem and AetherBodies.

Note that there is only one URL param (ref_frame), although this endpoint also requires that the request has a
file part. The ref_frame arg is used for returning min-max speeds for the new bodies so that the frontend does
not have to call available-bodies again.

The endpoint is called similar to available-bodies above. The return format is the same as available bodies, but
the returned list only has dictionaries for the new bodies added from the uploaded SPK kernel.

SPK Clear

URL Format:

```
/api/spk-clear/
```

Methods:

```
GET
```

Description:

This endpoint clears all uploaded SPK kernels from the backend. It takes no parameters and returns a list of
strings. Each string is the name of a body which was removed (the bodies that were uploaded, and not present
in the default kernel set).

Other backend files

AetherBodies.py

This file holds the AetherBodies class which acts as a store of all the known objects in the backend of Aether.
A dictionary holds metadata of each body that trajectory data can be obtained for. A single global instance of this
class is instantiated by the REST server. It is used for checking the validity of targets passed to the REST server
from the frontend, keeping track of valid time ranges for ephemeris, and providing data to the available-bodies endpoint
which is used to populate the bodies drop-down on the frontend. Each object in the dictionary represents a valid body
stored in one or more of the SPICE kernels. These kernels are located in the `backend/SPICE/kernels` directory and data
about each body and it's valid time ranges is obtained by using the command-line utility brief. Output from
brief is parsed using the SPKParser class.

SPKParser.py

This file contains the SPKParser class. It is responsible for getting the body names, IDs and valid time ranges for
each body contained in a binary SPK kernel. It has one method, parse, which takes a path to a binary SPK and returns
a list of dictionaries. It uses the JPL/NAIF command-line utility `brief` to obtain metadata from the SPK kernel.

MetakernelWriter.py

This file contains the MetakernelWriter class which handles creation of a metakernel file. It is called by the REST
server on initialization to create the metakernel file based on all of the SPICE kernels (both default and uploaded).
It has one method, write, which simply traverses the default and user_uploaded directories of SPICE kernels and adds
each path to the metakernel file. This class is created and run once when the REST server starts.

Frontend

The frontend component of this application consists of a Node webserver and several HTML and Javascript files that are rendered by a browser. The Node server is started and handled by the `/run.sh` script, so this section will focus on the various Javascript and HTML files that facilitate the 3D simulation.

Main Frontend Files

server.js

This is the main script that starts the Node.js webserver.

public/js/solarSystem.js

This is the main JavaScript file that initiates the default simulation. It also provides function definitions that are used to create new simulations, fetch data from the backend, and add objects to the simulation. These functions are used by `eventHandlers.js`, `dropdownFunctions.js` and `AetherObject.js`. In addition, this file defines global constants and variables that are used throughout the program.

public/js/dropdownFunctions.js

This file contains definitions of functions that operate on the dynamic dropdown checklist in the "Available Bodies" menu. These operations include populating the dropdown, updating it, adding bodies dynamically, zooming the camera to bodies, and pulling up information panels. These functions are used by `eventHandlers.js`.

public/js/eventHandlers.js

This file contains event listeners that handler user input on both UI menus. These event handlers are responsible for executing the necessary functions when a use clicks a button, moves a slider, or submits a form. This file uses functions in `solarSystem.js` and `dropdownFunctions.js`.

public/js/AetherObject.js

This file contains the class definition for an *AetherObject*, which extends the Spacekit.js class *SphereObject*. These objects represent the individual bodies inside of the simulation.

public/js/AetherSimulation.js

This file contains the class definition for an *AetherSimulation*, which extends the Spacekit.js class *Simulation*. This object represents an individual simulation, and is responsible for creating the 3D scene, keeping track of the passage of time, and managing each of its *AetherObject(s)*.

public/views/welcome.html

This file contains the HTML for the splash screen that the user sees upon loading the application. It is responsible for the welcome animation and for allowing the user to launch the simulation.

public/views/index.html

This file contains the HTML for the main UI and simulation(s). It sources all of the necessary JavaScript files and ultimately kicks off the loading of the simulation(s).

public/js/three.min.js

This file contains a minimum build of the THREE.js graphics library that is used by both Spackekit.js and our application to facilitate 3D animation. This library is a JavaScript wrapper for WebGL.

spacekit/

This directory houses the Spacekit.js library that our application uses to aid in space scene and simulation creation. It uses THREE.js to create 3D space environments.

Control Flow

- Application start:
    1. The application begins by serving `welcome.html`, which displays a splash screen and welcomes the user to the application.
    2. After clicking the “Launch” button, the browser loads `index.html`, which is responsible for creating the basic user interface and sourcing all of the necessary Javascript files.
    3. `solarSystem.js` is then sourced, kicking off the creation of the default simulation. It first establishes global constants and variables that are used throughout the program, and then gives some function definitions before making API requests.
- Simulation creation:
    1. At the end of `solarSystem.js` an *AetherSimulation* object is initialized.
    2. Once initialized, the “`api/available-bodies`” endpoint is used to get details about what objects are available from the backend, what data is available for that object, and what time range that data is valid for.
    3. An `"api/positions"` request is made to the backend. The backend returns a list of `(x,y,z)` coordinates for the Sun, eight planets, Pluto, and the moon.
    4. For each body in the returned data, an *AetherObject* is created and added to the *AetherSimulation* object.
    5. Finally, an HTML object is created to display the simulation’s time and rate-of-time properties to the user.

- The user can then use the simulation however they'd like:
- The user clicks anything inside the two menus:

    1. `eventHandlers.js` handles the user's input and invokes necessary functions contained in `solarSystem.js` and `dropdownFunctions.js`.

- The *AetherObjects* update their position lists:

    1. When there is 10 real seconds of time left before the object reaches the end or beginning of its position list, it uses the `"api/positions"` endpoint.
    2. Once data is returned, the object updates its members.
    3. **WARNING**: The threshold of 10 seconds was selected to give the backend enough time to respond. If there are too many objects in the simulation, or if time in the simulation is passing too quickly, there is a chance objects can get out of sync.

- The user exits the application

Attribution

The completion of this application would have been impossible without several amazing free and open-source libraries/frameworks. Thank you to the developers and communities who create and continue to maintain these invaluable tools.

Spacekit.js - https://typpo.github.io/spacekit/

This amazing library aided us in creating our 3D space environment by abstracting much of the THREE.js functionality into a high-level and easy-to-use API.

THREE.js - https://threejs.org/

This 3D graphics library provided us finer control over our 3D environment, and allowed us to extend Spacekit.js for our specific purposes.

WebGL - https://get.webgl.org/

This framework/specification allows for 3D graphics to be rendered inside a browser, which made it possible for us to create Aether as a webapp.