JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

SEW

Solarsystem

Softwareentwicklung

René Hollander
Paul Kalauner

Version 1.0
Begonnen am 03. November 2015
Beendet am 29. November 2015

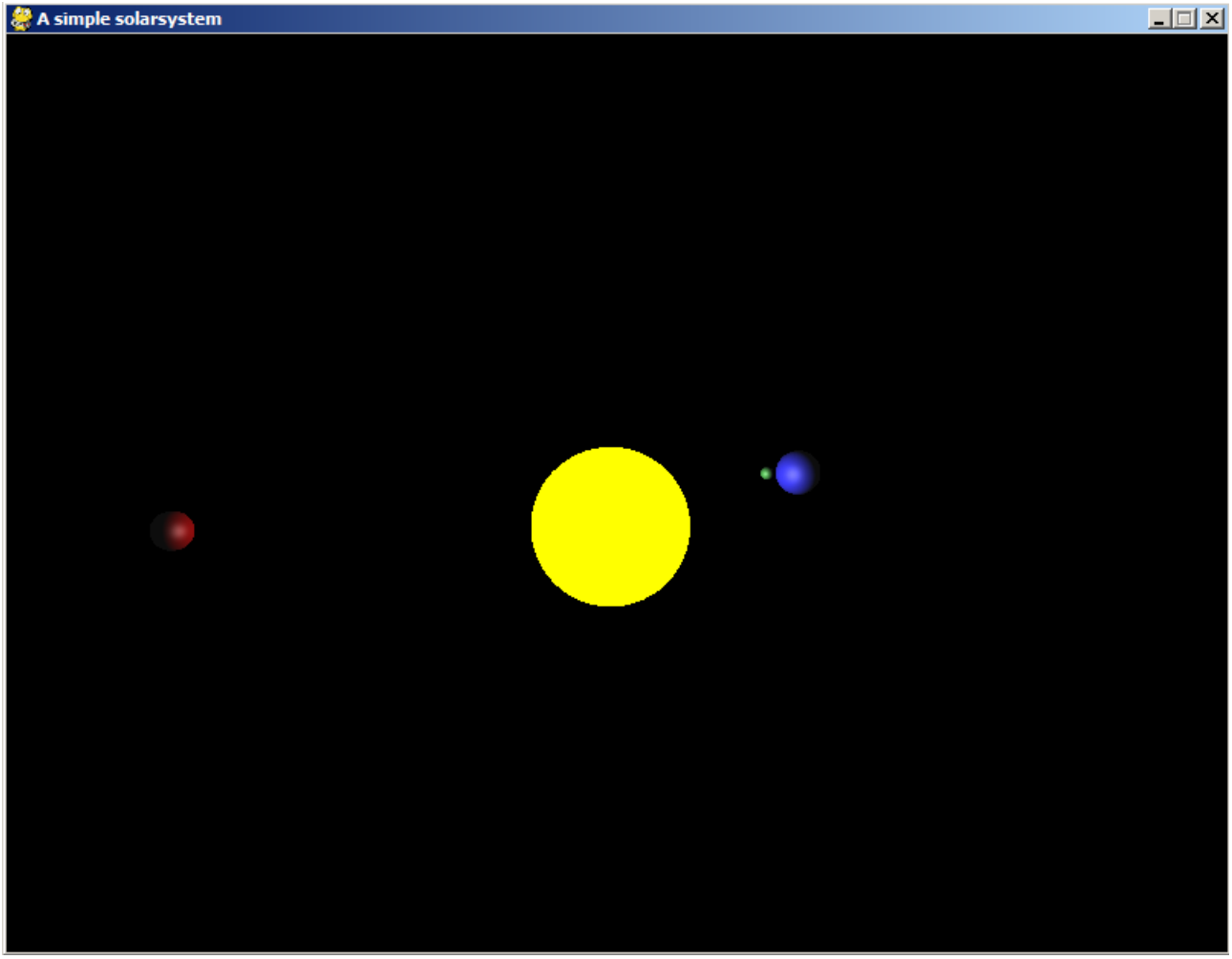
Inhaltsverzeichnis

1 Einführung

1.1 Aufgabenstellung

Wir wollen unser Wissen aus SEW nutzen, um eine kreative Applikation zu erstellen. Die Aufgabenstellung:

Erstelle eine einfache Animation unseres Sonnensystems!



In einem Team (2) sind folgende Anforderungen zu erfüllen.

- Ein zentraler Stern
- Zumindest 2 Planeten, die sich um die eigene Achse und in elliptischen Bahnen um den Zentralstern drehen
- Ein Planet hat zumindest einen Mond, der sich zusätzlich um seinen Planeten bewegt
- Kreativität ist gefragt: Weitere Planeten, Asteroiden, Galaxien,...

- Zumindest ein Planet wird mit einer Textur belegt (Erde, Mars,... sind im Netz verfügbar)

Events:

- Mittels Maus kann die Kameraposition angepasst werden: Zumindest eine Überkopf-Sicht und parallel der Planentenbahnen
- Da es sich um eine Animation handelt, kann diese auch gestoppt werden. Mittels Tasten kann die Geschwindigkeit gedrosselt und beschleunigt werden.
- Mittels Mausklick kann eine Punktlichtquelle und die Textierung ein- und ausgeschaltet werden.
- Schatten: Auch Monde und Planeten werfen Schatten.

Wählt ein geeignetes 3D-Framework für Python (Liste unter https://wiki.python.org/moin/PythonGameLibraries) und implementiert die Applikation unter Verwendung dieses Frameworks.

Abgabe: Die Aufgabe wird uns die nächsten Wochen begleiten und ist wie ein (kleines) Softwareprojekt zu realisieren, weshalb auch eine entsprechende Projektdokumentation notwendig ist. Folgende Inhalte sind in jedem Fall verpflichtend:

- Projektbeschreibung (Anforderungen, Teammitglieder, Rollen, Tools, ...)
- GUI-Skizzen und Bedienkonzept (Schnittstellenentwürfe, Tastaturbelegung, Maussteuerung, ...)
- Evaluierung der Frameworks (zumindest 2) inkl. Beispielcode und Ergebnis (begründete Entscheidung)
- Technische Dokumentation: Architektur der entwickelten Software (Klassen, Design Patterns)
  - Achtung: Bitte überlegt euch eine saubere Architektur!
  - Den gesamten Source Code in 1 Klasse zu packen ist nicht ausreichend!
- Kurze Bedienungsanleitung
- Sauberes Dokument (Titelblatt, Kopf- und Fußzeile, ...)

Hinweise zu OpenGL und glut:

- Ein Objekt kann einfach mittels glutSolidSphere() erstellt werden.
- Die Planten werden mittels Modelkommandos bewegt: glRotate(), glTranslate()
- Die Kameraposition wird mittels gluLookAt() gesetzt
- Bedenken Sie bei der Perspektive, dass entfernte Objekte kleiner - nahe entsprechende größer darzustellen sind.
  Wichtig ist dabei auch eine möglichst glaubhafte Darstellung. gluPerspective(), glFrustum()
- Für das Einbetten einer Textur kann die Library Pillow verwendet werden! Die Community unterstützt Sie bei der Verwendung.

Viel Spaß und viel Erfolg!

1.2 Team

- René Hollander
- Paul Kalauner

1.3 Zeitaufzeichnung

Name	Datum	Zeit in Minuten	Beschreibung
Hollander	03.11.2015	60	Grundgerüst
Hollander	03.11.2015	120	Rotation der Planeten
Hollander	04.11.2015	120	Kamera
Kalauner	05.11.2015	60	Maus-Support für Kamera
Kalauner	10.11.2015	100	Protokollierung der erfolgten Arbeitsschritte
Hollander	20.11.2015	60	Einbindung von Texturen
Hollander, Kalauner	22.11.2015	60	Mehr Planeten hinzugefügt
Kalauner	22.11.2015	30	Steuerung des Kameratempos
Kalauner	23.11.2015	30	Sphinx-Konfiguration
Hollander	24.11.2015	60	Orbit-Plotting
Kalauner	24.11.2015	30	Visualisierung der Orbits togglen
Kalauner	24.11.2015	40	Protokoll für Zwischenabgabe aktualisieren
Hollander	24.11.2015	30	Hilfe-Menü
Kalauner	29.11.2015	40	Hilfe-Menü fertiggestellt
Kalauner	29.11.2015	20	Texturen ein/ausschalten
Hollander	29.11.2015	40	Mehr Planeten hinzugefügt
Hollander	29.11.2015	60	Skybox
Kalauner	29.11.2015	30	Sphinx Fehler beheben

1.4 Tools

Als IDE wird PyCharm verwendet. Die Frameworks werden im Abschnitt 2.1 evaluiert.

2 Durchführung

2.1 Evaluierung der Frameworks

PyGame

- Für Spieleentwicklung entworfen
- Kümmert sich um Fensteraufbau, Kollisionen und Events

[1]

Beim Example von PyGame war für uns ersichtlich, dass aufgrund der umständlichen Einbindung von Texturen, evtl. besser geeignete Frameworks für diese Aufgabe existieren. Weiters verwenden die Example-Codes verschiedene (alte) Python-Versionen. Unter Windows war die Installation nicht auf die Schnelle möglich.

PyOpenGL

- Python binding to OpenGL
- Arbeitet gut mit PyGame zusammen

[2]

Pyglet

- Objekt-orientiertes Programmierinterface für 3D-Anwendungen
- Example funktionierte sofort

[3]

2.2 Fazit

Da **Pyglet** Out of the box funktionierte, haben wir uns für dieses Framework entschieden.

2.3 Aufsetzen der Entwicklungsumgebung

Als ersten Schritt haben wir ein Projekt in *PyCharm* erstellt. Mit einem passender *.gitignore* haben wir dieses auch mittels *git* für beide Personen zugreifbar gemacht.

Als nächsten Schritt haben wir die benötigten Frameworks installiert. Diese sind:

- Pyglet
- Pyglet-Gui
- Euclid: Für mathematische Berechnungen

Pyglet

Pyglet ließ sich einfach mittels `pip install pyglet` installieren. Die Installation von und Euclid war etwas aufwändiger:

Euclid

Euclid kann mittels

```
pip install git+https://github.com/brad/pyeuclid.git
```

installiert werden. Allerdings wird das package *numpy* benötigt. Unter Windows war es in unserem Fall erforderlich, dieses separat herunterzuladen. Um numpy unter Windows zu installieren, musste ein vorkompiliertes Package von http://www.lfd.uci.edu/~gohlke/pythonlibs/ heruntergeladen werden.

Wichtig war es, das Paket für die richtige Python-Version zu wählen. Der Dateiname der richtigen Datei für die aktuelle Python-Version (3.5) ist folgender:

```
numpy-1.9.3+vanilla-cp35-none-win32.whl
```

Pyglet-Gui

Pyglet-Gui kann mittels

```
pip install git+https://github.com/jorgecarleitao/pyglet-gui.git
```

installiert werden.

Virtualenv mit Setup-Script (Linux)

Alternativ kann virtualenv und unser (Bash) Script verwendet werden, welches alle Packages automatisch installiert. Dazu muss mittels

`pip install virtualenv` virtualenv installiert werden. Danach können mit

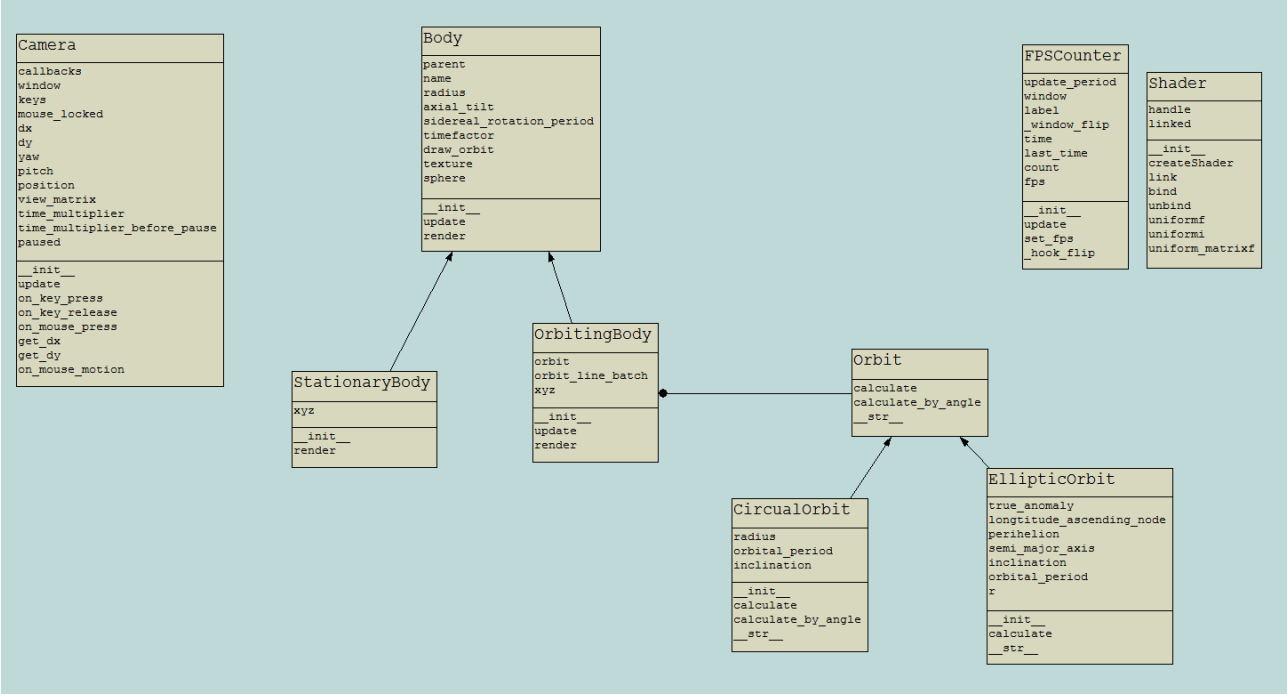
`./dev_setup.sh` die Requirements installiert werden.

Anschließend muss noch in der Run-Configuration der venv-Python-Interpreter ausgewählt werden.

2.4 Grundsätzliche Überlegungen

Wir wollen die Bewegung, Geschwindigkeit, Inklination, usw. der Planeten so original getreu wie möglich gestalten. Dazu werden verschiedene Werte wie Radius oder Achsneigung benötigt.

Diese Informationen werden in der `__init__()` Funktion von der abstrakten Klasse `Body` bzw. in den konkreten Klassen für die einzelnen Himmelskörper übergeben. Hier ein UML-Diagramm unseres Designs:



Die `Orbit`-Klasse nimmt weitere, für Berechnungen wichtige Werte, entgegen.

Dabei wird zwischen `StationaryBody` (z.B. für die Sonne) und einem `OrbitingBody` (ein Himmelskörper der sich auf einem Orbit bewegt) unterschieden. Jeder `OrbitingBody` besitzt eine Referenz auf Orbit, um die für die Positionsberechnung wichtigen Daten zu besitzen.

Dies setzt außerdem ein Strategy-Pattern um, da die Art des Orbits jederzeit ausgetauscht werden kann. (*CircularOrbit* oder *EllipticOrbit*)

Es wird ebenfalls eine Art Decorator-Pattern umgesetzt. Dieses wird verwendet um einem Planeten ein *parent* zu übergeben. Beispielsweise ist das *parent* der Erde die Sonne, da sich die Erde um die Sonne bewegt

2.5 Wichtige Codeteile

Mittels JSON werden die Werte für die Planeten definiert. Ein solches File sieht für Himmelskörper mit kreisförmigen Orbits folgendermaßen aus:

```
{
  "name": "Moon",
  "parent": "earth",
  "texture": "moon.jpg",
  "basecolor": {
    "r": 118,
    "g": 118,
    "b": 118
  },
  "radius": 1.737,
  "orbit": {
    "type": "circular",
    "radius": 15.36,
    "orbital_period": 29.530589,
    "inclination": 0.08979719
  },
  "axial_tilt": 0.116710167,
  "sidereal_rotation_period": 27.321582,
  "mass": 7.349e+25
}
```

Elliptische Orbits sind wie folgt definiert:

```
{
  "name": "Earth",
  "parent": "sun",
  "texture": "earth.jpg",
  "basecolor": {
    "r": 29,
    "g": 60,
    "b": 109
  },
  "radius": 6.371,
  "orbit": {
    "type": "elliptic",
    "apoapsis": 152100000000,
    "periapsis": 147095000000,
    "longtitude_ascending_node": -0.196535244,
    "argument_of_periapsis": 1.993302665,
    "inclination": 0.000000873,
    "initial_mean_anomaly": 6.259047404,
    "multiplier": 0.000000001
  },
  "axial_tilt": 0.409092629,
  "sidereal_rotation_period": 0.99726968,
  "mass": 5.97237e+24
}
```

Durch das Strategy-Pattern ist es möglich, kreisförmige **und** elliptische Orbits zu verwenden. Es muss nur beachtet werden, dass elliptische Orbits zusätzliche Werte benötigen.

Die JSON-Files werden im *loader.py* eingelesen.

Dies *load_bodies* Funktion liefert eine Liste zurück:

```
def load_bodies(directory):
    """
    Loads all bodies that are defined in the JSON files from the given directory

    :param directory: directory to load the bodies from
    :type directory: str
    :return: list of the loaded bodies
    :rtype: list
    """

    files = glob.glob(os.path.join(directory, "*.json"))
    bodies = {}
    for file in files:
        print("Loading body " + file)
        with open(file) as data_file:
            internal_name = splitext(basename(file))[0]
            bodies[internal_name] = load_body(json.load(data_file))
    for key in bodies:
        body = bodies[key]
        if body.parent_internal_name is not None:
            body.parent = bodies[body.parent_internal_name]
            del body.parent_internal_name

    for body in bodies.values():
        print("Executing post_init for " + body.name)
        body.post_init()

    return bodies.values()
```

Der eigentliche Parsvorgang findet in der Funktion *load_body* statt:

```
def load_body(data):
    """
    Load the body from the specified JSON data. Parent is not set here!

    :param data: JSON data to load the body from
    :return: Body from the supplied data
    :rtype: :class:`solarsystem.body.Body`
    """

    name = data["name"]
    parent = None
    if "parent" in data:
        parent = data["parent"]
    texture = data["texture"]
    basecolor = data["basecolor"]
    radius = data["radius"]
    axial_tilt = data["axial_tilt"]
```

```
    sidereal_rotation_period = data["sidereal_rotation_period"] * dts
    mass = data["mass"]
    has_orbit = False
    orbit = None
    has_ring = False
    ring_texture = None
    ring_inner_radius = None
    ring_outer_radius = None

    if "orbit" in data:
        has_orbit = True
        orbit = load_orbit(data["orbit"])
    if "ring" in data:
        ring_data = data["ring"]
        has_ring = True
        ring_texture = ring_data["texture"]
        ring_inner_radius = ring_data["radius"]["inner"]
        ring_outer_radius = ring_data["radius"]["outer"]

    body = None

    if has_orbit:
        body = OrbitingBody(None, name, texture, basecolor, radius, orbit,
axial_tilt, sidereal_rotation_period, mass)
        if has_ring:
            body.renderer = OrbitingBodyWithRingRenderer()
            body = setup_ring_renderer(ring_texture, ring_inner_radius, ring_ou-
ter_radius, body)
    else:
        body = StationaryBody(None, name, texture, basecolor, radius,
axial_tilt, sidereal_rotation_period, mass)

    body.parent_internal_name = parent
    return body
```

Hier werden die Elemente aus dem übergebenen Dictionary Variablen zugewiesen. Das selbe geschieht in der Funktion *load_orbit,* nur werden dort die Attribute der Orbits eingelesen.

Mit den eingelesenen Werten werden dann die Objekte initialisiert.

In *game.py* wird die Liste einer Variable zugewiesen:

```
# load the bodies from the json files
planets = load_bodies("bodies")
```

Die planets-Liste ist notwendig, um alle Planeten der Zeit entsprechend zu aktualisieren:

```
def update(dt):
    global time
    global mvp

    timestep = 60 * 60 * 24 * 7 * camera.time_multiplier
    time += dt * timestep
    label_timestep.text = "1 second = " + str(floor(timestep / 60 / 60)) + "hours"

    camera.update(dt)
    mvp = proj_matrix * camera.view_matrix * model_matrix

    for planet in planets:
        planet.update(time)
```

Die Klasse *Camera* kümmert sich um das richtige Handling der Kamera. Außerdem nimmt sie verschiedene Key-Events entgegen. Einerseits zum Steuern der Kamera, andererseits werden beispielsweise die Events der Tasten + und – zum Anpassen der Geschwindigkeit verarbeitet.

Hier ein Ausschnitt des Kamerahandlings:

```
def update(self, delta):
    movementspeed = 30 * delta
    mousesensitivity = 0.005

    dx = self.get_dx()
    dy = self.get_dy()

    self.yaw -= dx * mousesensitivity
    self.pitch += dy * mousesensitivity

    if self.pitch > halfpi:
        self.pitch = halfpi
    if self.pitch < -halfpi:
        self.pitch = -halfpi

    if self.mouse_locked:
        if self.keys[key.LSHIFT]:
            # ...
```

Das Anpassen der Geschwindigkeit erfolgt folgendermaßen:

Die momentane Geschwindigkeit wird mit einem *time_multiplier* multipliziert. Dieser beträgt anfangs 1, das daraus resultierende Verhältnis ist 1 Sekunde zu 1 Woche in Originalzeit.

Durch Drücken von + und – kann die Geschwindigkeit erhöht bzw. vermindert werden:

```
# Key code 43: Plus key
if symbol == key.NUM_ADD or symbol == 43:
    if self.paused:
        self.time_multiplier = self.time_multiplier_before_pause
        self.paused = False
    else:
        self.time_multiplier += 0.1
```

Weiters ist eine Pause-Funktion realisiert. Durch das Klicken von der P-Taste wird der *time_multiplier* vorübergehend auf 0 gesetzt und somit die Simulation gestoppt.

Der Inhalt des Hilfe-Menüs wird über die Datei *text/help.txt* geladen.

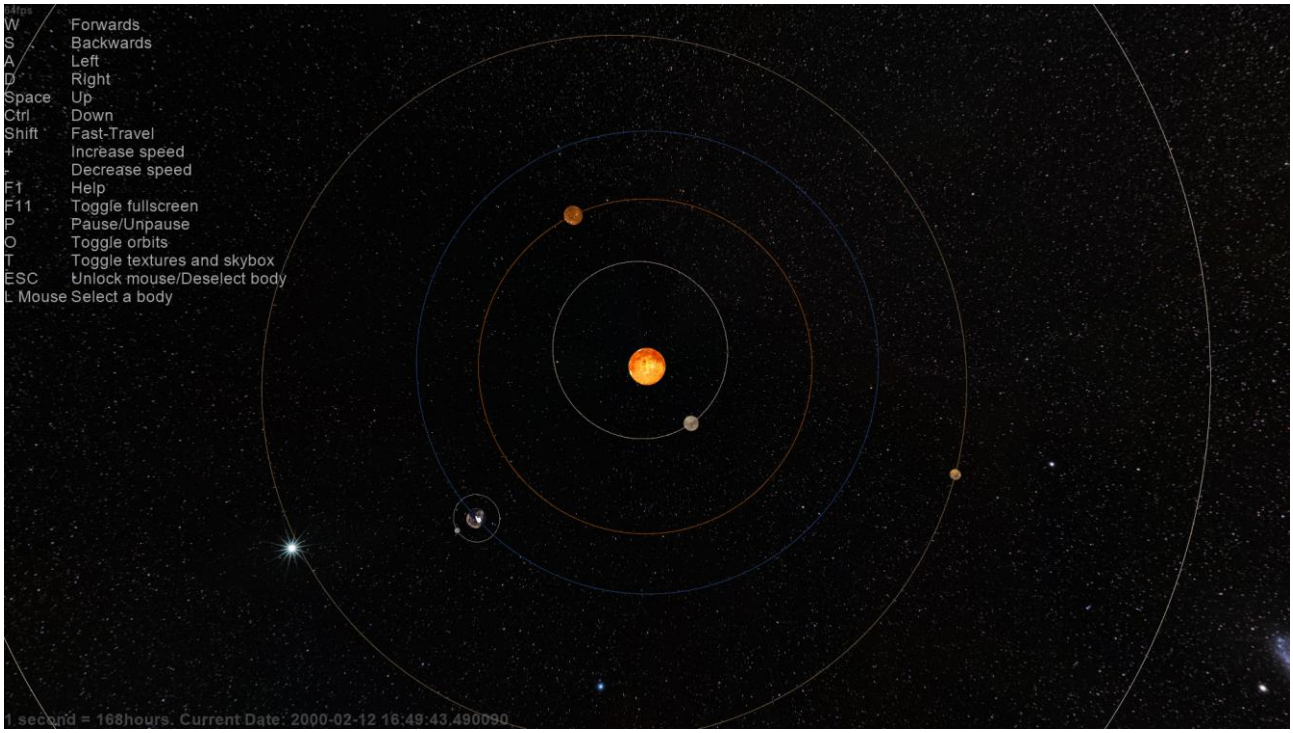
2.6 Dokumentation

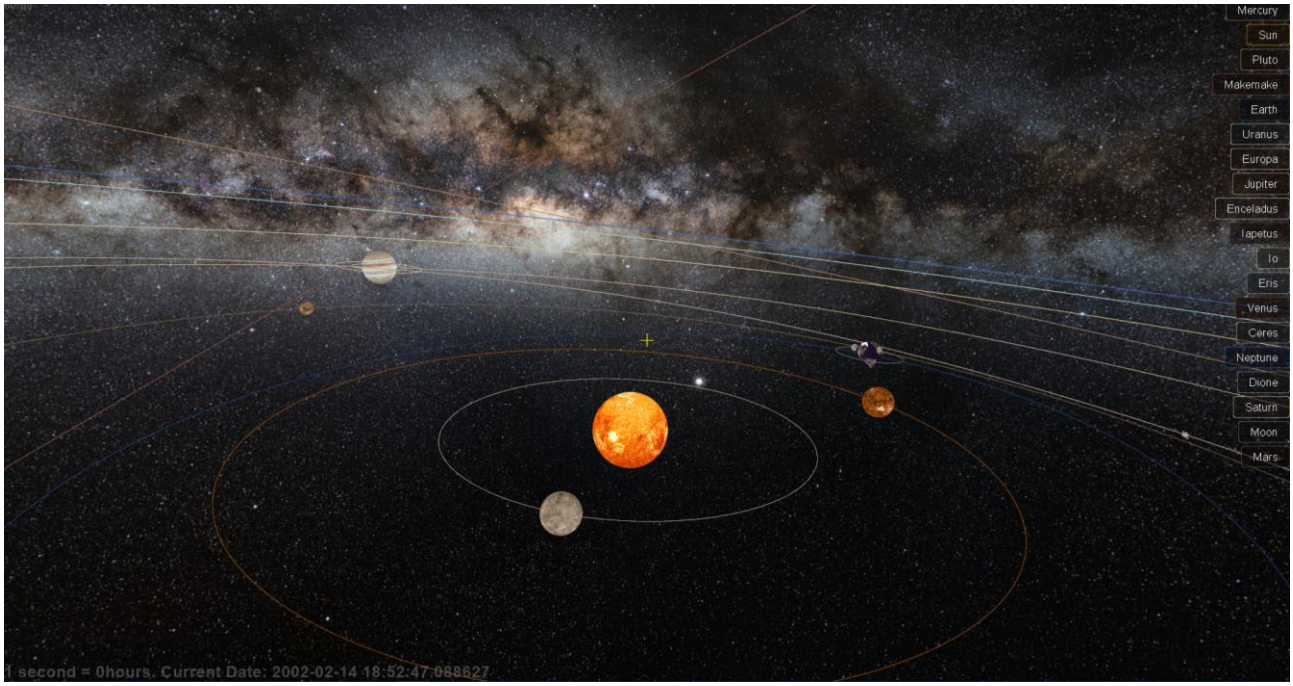
Die mittels Sphinx generierte Dokumentation befindet sich in *doc/build* und sollte durch Öffnen der Datei *index.html* betrachtet werden.

2.7 Steuerung

TASTE	AKTION
W	Kamerabewegung nach vor
S	Kamerabewegung zurück
A	Kamerabewegung nach links
D	Kamerabewegung nach rechts
SPACE	Kamerabewegung nach oben
CTRL	Kamerabewegung nach unten
MAUS	Kamera bewegen
SHIFT (GEDRÜCKT)	Kamerabewegungen beschleunigen
+	Geschwindigkeit erhöhen
-	Geschwindigkeit reduzieren
SHIFT (GEDRÜCKT)	Geschwindigkeit schneller anpassen
F11	Fullscreen ein/aus
P	Animation pausieren/fortführen
O	Visualisierung der Orbits ein-/ausschalten
T	Texturen ein-/ausschalten
F1	Hilfe-Menü ein-/ausblenden
L MAUS	Himmelskörper auswählen um diesem zu folgen
ESC	Mouse entlocken/Himmelskörper deselektieren

2.8 Screenshots





3 Referenzen

[1] PyGame: About. Abrufbar unter: http://www.pygame.org/wiki/about [zuletzt abgerufen: 03.11.2015]

[2] Python-Wiki: PyOpenGL. Abrufbar unter: https://wiki.python.org/moin/PyOpenGl [zuletzt abgerufen: 03.11.2015]

[3] Pyglet: Overview. Abrufbar unter: https://bitbucket.org/pyglet/pyglet/overview [zuletzt abgerufen: 17.11.2015]