mctracer

photon propagation in complex sceneries

mindset and how to use

by

Sebastian Achim Müller

sebmuell@phys.ethz.ch

Institute for Particle Physics

ETH Zurich

2015-2017

Contents

Chapter 1

Abstract

The mctracer is a simulation for geometrical optics. It can propagate photons in a complex 3D scenery. The mctracer simulates reflection, refraction and absorbtion. It does not cover diffraction. For the investigation of optical devices or phenomena, mctracer records the full photon's trajectory starting with the production, through all the photon's interactions until its final absorbtion. A small set of primitiv surfaces is provided in mctracer to form simple optical devices, such as lenses, imaging mirrors, light concentrators and aperture stops. Further, complex objects can be simulated using triangular meshes. To produce photons, mctracer comes with a set of light sources to illuminate your scenery. For more complex light sources, photons can be read from external files. mctracer can handle very complex sceneries with million of primitives while beeing fast and accurate on a scientific level. You can feed your scenery into mctracer using common CAD files for triangular meshes and a custom mctracer xml file which describes the primitives provided by mctracer itself. When the provided tools do not cover your demand you have the chance to implement the features yourself. The mctracer was originally created for simulations in Astro Particle Physics. Imaging Atmospheric Cherenkov Telescopes like FACT and the CTA MST made use of mctracer to investigate and improve their performance. The Atmospheric Cherenkov Plenoscope (ACP) was born in this simulation.

Chapter 2

The Atmospheric Cherenkov Plenoscope (ACP)

The mctracer can simulate ACPs. An ACP consists out of two main parts. First, an imaging system like e.g. a segmented imaging reflector as it is used for classic Imaging Atmospheric Cherenkov Telescopes (IACTs). Second, a light field sensor.

2.1 Create an ACP scenery

Lets create an example scenery of an ACP called PLERITAS, which is a plenoptic version to a VERITAS IACT.

```
| /demo$ mkdir pleritas
| /demo$ cd pleritas/
```

All the resources needed to describe PLERITAS have to be in a folder.

```
| demo/pleritas$ vi  scenery.xml
```

Create a xml file called scenery.xml and describe your scenery in there. For our PLERITAS we use a basic VERITAS like imaging reflecor (created using the segmented reflector tool),

demo/pleritas/scenery.xml

```
        <segmented_reflector>
            <set_frame name="imaging_reflector" pos="[0,0,0]" rot="[0,0,0]"/>
            <set_surface reflection_vs_wavelength="mirror_reflection"/>
            <set_segmented_reflector
                focal_length="12.0"
                max_outer_aperture_radius="5"
                min_inner_aperture_radius="0.2"
                DaviesCotton_over_parabolic_mixing_factor="0.0"
                facet_inner_hex_radius="0.3"
                gap_between_facets="0.01"/>
        </segmented_reflector>
```

a light field sensor

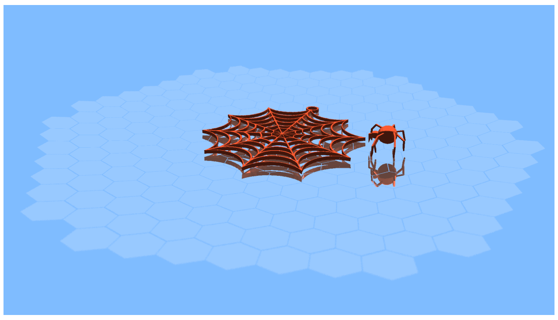
demo/pleritas/scenery.xml

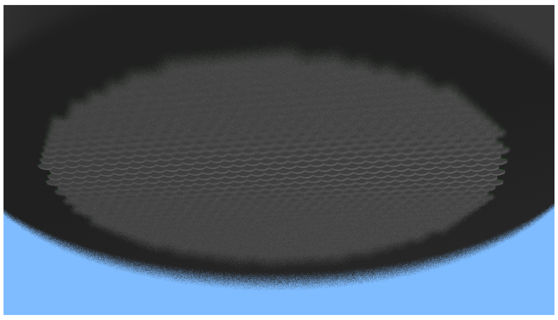
```
        <light_field_sensor>
            <set_frame name="light_field_sensor"  pos="[0,0,12.0]"  rot="[0,0,0]"/>
            <set_light_field_sensor
                expected_imaging_system_focal_length="12.0"
                expected_imaging_system_aperture_radius="5"
                max_FoV_diameter_deg="3.5"
                hex_pixel_FoV_flat2flat_deg="0.1"
                housing_overhead="1.3"
                number_of_paxel_on_pixel_diagonal="5"
                lens_refraction_vs_wavelength="lens_refraction"
                bin_reflection_vs_wavelength="mirror_reflection"
            />
        </light_field_sensor>
```

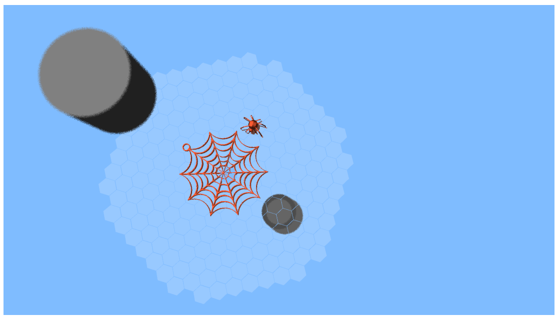
and a spider web which causes shadowing and is described in a CAD file named spider.stl which is also in the scenery folder.

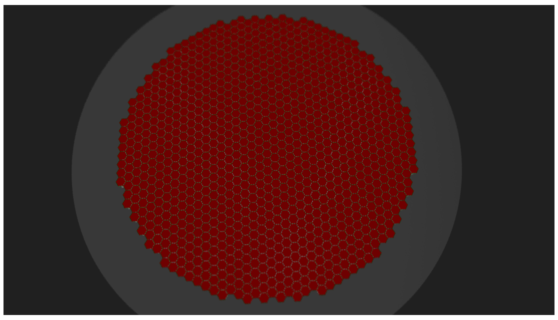
demo/pleritas/scenery.xml

```
        <stl>
            <set_frame name="spider" pos="[−1.5,␣−1.75,␣0.4]" rot="[0,0,0]"/>
            <set_stl  file="spider.stl"    scale="0.12"/>
            <set_surface  reflection_vs_wavelength="zero"   color="[255,91,49]"/>
        </stl>
```









All resources like the CAD file of the spider web must be placed in the scenery folder.

```
| /demo/pleritas$ ls
| scenery.xml  spider.stl
```

Explore your scenery using mctShow.

```
| /demo/pleritas$ mctShow −s scenery.xml
```

2.2 Run the light field calibration

```
| /demo$ mctPlenoscopeCalibration −s scenery −o pleritas_calibration   −n 3
| Plenoscope Calibrator:  propagating 3M photons
| 1 of 3
| 2 of 3
| 3 of 3
| /demo$
```

Inside the calibration output folder we find a copy of the scenery folder in the input folder and the results in 3 binary blocks.

```
| /demo$ cd pleritas_calibration/
| /demo/pleritas_calibration$  ls
| input
| light_field_sensor_geometry.header.bin
| lixel_positions.bin
| lixel_statistics.bin
```

Make summary plots of the calibration result.

```
| /demo$ mctPlenoscopePlotLixelStatistics  −i pleritas_calibration
```

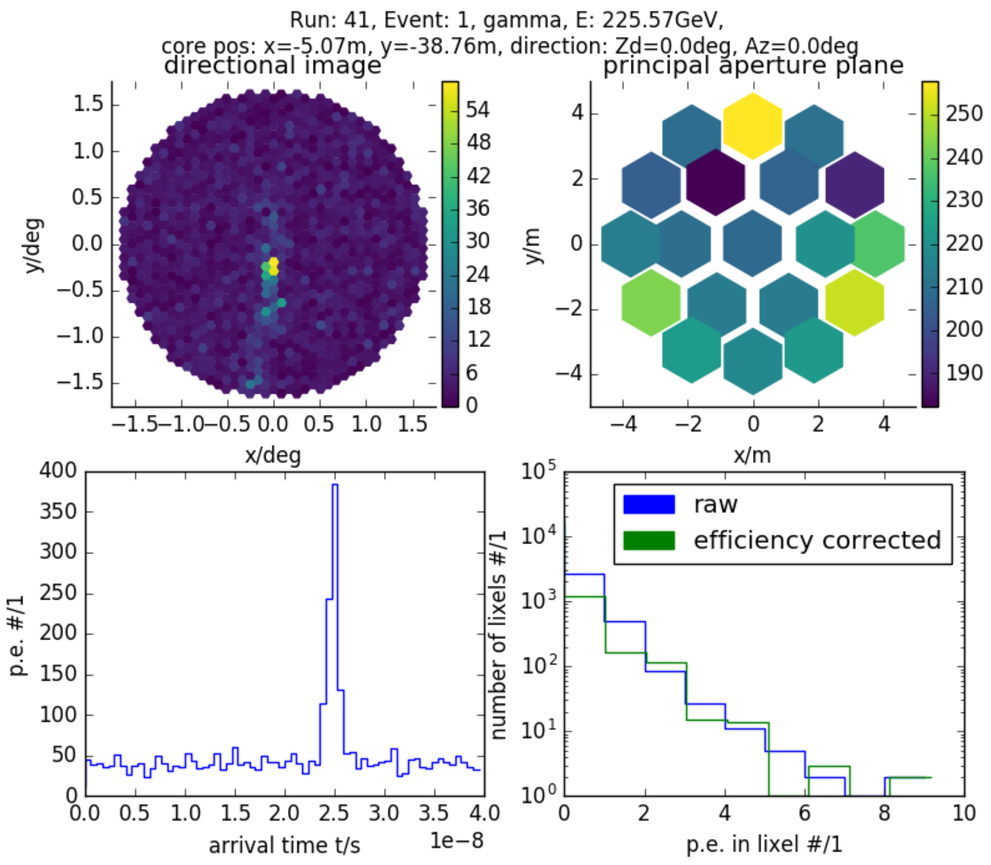
no output path is defined, a folder with plots is placed in the input folder.

2.3 Simulate ACP responses to EAS

```
| /demo$ mctPlenoscopePropagation −c propagation_config.xml
| −l pleritas_calibration    −i /some/simulation/gamma1_RUN41.dat −o my_run
| event 1,  PRMPAR 1, E 225.569 GeV
| event 2,  PRMPAR 1, E 168.365 GeV
| event 3,  PRMPAR 1, E 46.5717 GeV
| ...
```

2.4 Explore the siumlated events

```
| /demo$ ipython
| In [1]:  import plenopy as pop
| In [2]:  run = pop.Run('my_run')
| In [3]:  event = run.event(1)
| In [4]:  event.plot()
```



2.5 Light Field calibration

Each read out channel (lixel) on the light field sensor of the ACP corresponds to a specific ray in the light field. Each of these rays has a support on the principal aperture plane at position x and y on and a direction vector descibed by c_x and c_y. Further, each of these lixel has a specific time delay t_{delay} which the light needs to travel when comig from the principal aperture plane and each lixel has its own efficency η due to its specific geometrical position in the set up. So in the light field calibration we determine η, x, y, c_x, c_y and t_{delay} for each lixel. The calibration is done by throughing photons into the ACP, where we randomly draw both the photons intersection on the principal aperture plane x, y and their incoming direction c_x, c_y. In the calibration, many photons are used and several of them will be absorbed in the lixels. For each lixel, there is list of the photon properties (x, y, c_x, c_y and t_{delay}) of the photons that reached this lixel. From this list, the lixel is assigned the averages of all these properties, as well as their standard deviations. The number of photons reaching the lixel during the calibration gives the efficiency η.

../Plenoscope/Calibration/LixelStatistics.h

```
    struct LixelStatistic {
        float efficiency, efficiency_std;
        float cx_mean, cx_std;
        float cy_mean, cy_std;
        float x_mean, x_std;
        float y_mean, y_std;
        float time_delay_mean, time_delay_std;
        LixelStatistic();
    };
```

Chapter 3

1D functions

The Function::Func1D class provides 1D mapping for floating numbers.

$$\begin{aligned} y &= f(x) & (3.1) \\ x &\in X & (3.2) \end{aligned}$$

All functions have limits which need to be respected. Any call of a function $f(x)$ with $x \notin X$ will throw an exception. We are strict about this behaviour to enforce that no propagation passes silently where e.g. your mirror's reflective index is only defined up to 600 nm but you shoot 800 nm photons onto it. Functions live in their own namespace. ../Tests/Examples/Func1DExample.cpp

3.1 Domains and their limits

First we define limits for the domains of our functions. ../Tests/Examples/Func1DExample.cpp

```
        Limits limits(0.0,    1.0);
```

The limits here include the lower bound 0.0 and exclude the upper one 1.0. A limit can assert that a given argument is in its domain. If not, it will throw an exception.

../Tests/Examples/Func1DExample.cpp

```
        EXPECT_THROW( limits.assert_contains(−0.1), Limits::OutOfRange );
        EXPECT_NO_THROW( limits.assert_contains(0.0) );
        EXPECT_NO_THROW( limits.assert_contains(0.5) );
        EXPECT_THROW( limits.assert_contains(1.0), Limits::OutOfRange );
```

All functions have a domain within their limits. The limits are given to the funcions during construction. ../Tests/Examples/Func1DExample.cpp

```
        Constant con(1.337, limits);
```

Functions assert, the argument to be inside their domain. ../Tests/Examples/Func1DExample.cpp

```
        EXPECT_THROW( con(−0.1), Limits::OutOfRang
        EXPECT_NO_THROW( con(0.0) );
        EXPECT_NO_THROW( con(0.5) );
        EXPECT_THROW( con(1.0), Limits::OutOfRange
```

3.2 Constant

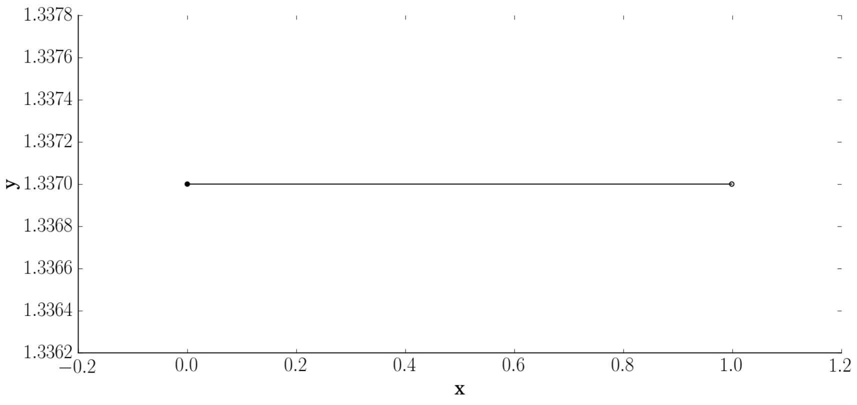
Sometimes it needs a constant function which will return the same value for any argument inside their domain limits.

$$y = f(x) = c \tag{3.3}$$

A constant function is created given its single constant value e.g. 1.337 and its domain limits. ../Tests/Examples/Func1DExample.cpp

```
Constant c(1.337, Limits(0.0, 1.0));
```

When called, within the limits, it will always return its constant value. ../Tests/Examples/set_up_scener



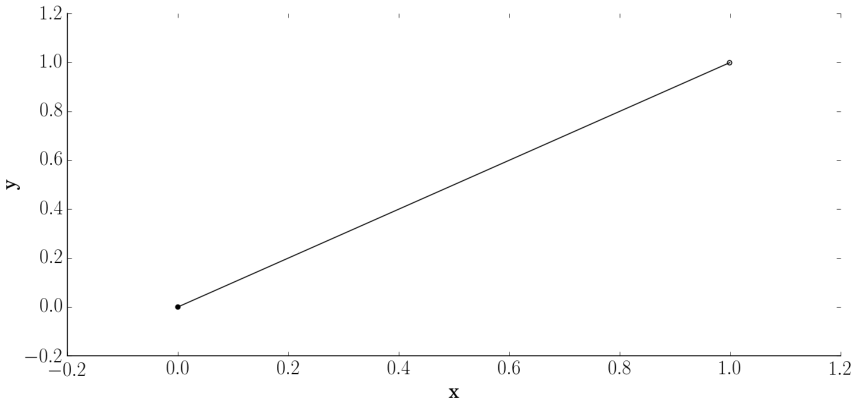
3.3 Polynom3

The versatile polynom to the power of 3 is defined by its four parameters a, b, c and d.

$$y = f(x) = ax^3 + bx^2 + cx^1 + dx^0 \tag{3.4}$$

We initialize the Polynom3 using a, b, c, d and the limits. By setting the higer orders to zero, we create e.g. a linear mapping. ../Tests/Examples/Func1DExample.cpp

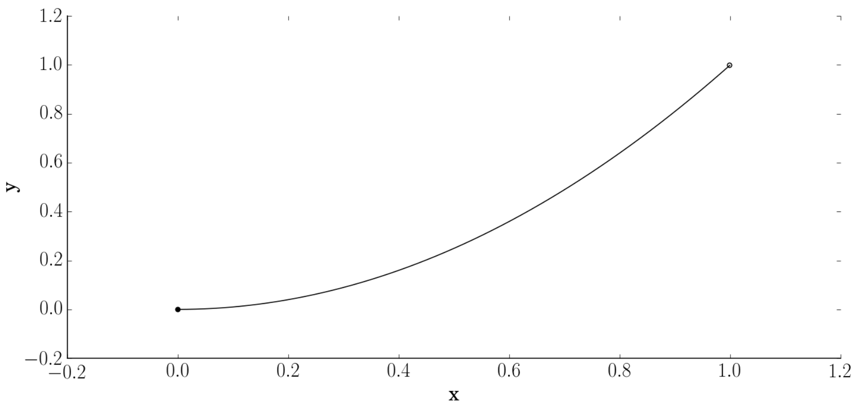
```
Polynom3 p3(0.0, 0.0, 1.0, 0.0, Limits(0.0, 1.0));
```



We can do a quadratic mapping.

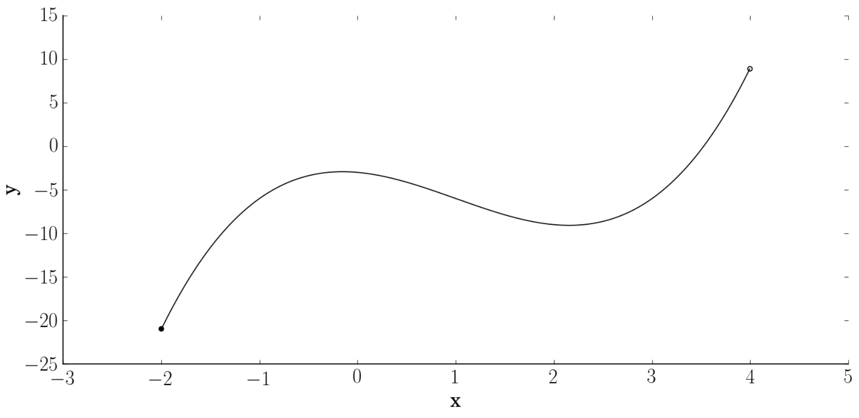
../Tests/Examples/Func1DExample.cpp
```
        Polynom3 p3(0.0, 1.0, 0.0, 0.0, Limits(0.0, 1.0));

        EXPECT_NEAR( 0.0, p3(0.0) ,1e−9);
        EXPECT_NEAR( 0.25, p3(0.5) ,1e−9);
        EXPECT_NEAR( 0.04, p3(0.2) ,1e−9);
```



The full polynom to the power of 3.

../Tests/Examples/Func1DExample.cpp
```
        Polynom3 p3(1.0, −3.0, −1.0, −3.0, Limits(−2.0, 4.0));
```

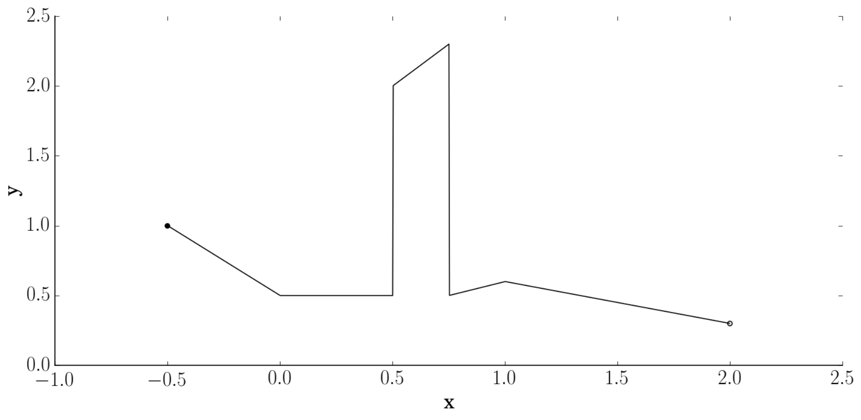


3.4 Linear interpolation look up table

In some cases, it can be tough to model an analytic $1D$ function. In these cases one can still use the a look up table with linear interpolation. The input table also defines the domain limits, so no limits have to be given during construction.

../Tests/Examples/Func1DExample.cpp
```
        std::vector<std::vector<double>> table = {
            {−0.5, 1.0},
            { 0.0, 0.5},
            { 0.5, 0.5},
            { 0.50001, 2.0},
            { 0.75, 2.3},
            { 0.75001, 0.5},
            { 1.0, 0.6},
            { 2.0, 0.3}
        };

        LinInterpol ip(table);
```



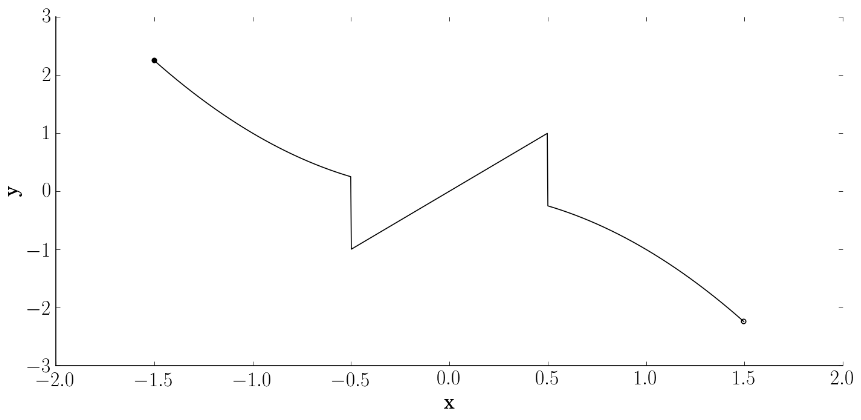
3.5 Concatenation

Functions can be concatenated when their domain limits match. The functions to be concatenated can be of any kind, even concatenated functions themselve. Since the concatenated function can deduce its domain limits from the input functions, no limit has to be given during construction.

../Tests/Examples/Func1DExample.cpp

```
    Polynom3 f1(0.0, 1.0,  0.0,  0.0,  Limits(−1.5,  −0.5));
    Polynom3 f2(0.0, 0.0,  2.0,  0.0,  Limits(−0.5,  0.5));
    Polynom3 f3(0.0, −1.0, 0.0,  0.0,  Limits(0.5,   1.5));

    std::vector<const Func1D*> funcs = {&f1, &f2, &f3};
    Concat concat(funcs);
```



3.6 Access

Access to the values of a function is done using the bracket operator.

../Tests/Examples/Func1DExample.c

```
    Polynom3 p3(1.0, −3.0, −1.0, −3.0, L

    double value1 = p3(−1.0);
    double value2 = p3(−0.0);
```

Also a function can provide a table of both argument and value. The number of samples along the domain limits of the function can be specified.

../Tests/Examples/Func1DExample.cpp

```
    std::vector<std::vector<double>> table = p3.get_sa
    EXPECT_EQ(1000u, table.size());
```

Using the ascci io, a function can be exported into a text file.

```
../Tests/Examples/Func1DExample.cpp
        AsciiIo::write_table_to_file(p3.get_samples(7),
```

The output text file is a two column matrix. First column is the argument x, second is the function value $f(x)$.

```
-2 -21
-1.142857143 -7.268221574
-0.2857142857 -2.982507289
0.5714285714 -4.364431487
1.428571429 -7.635568513
2.285714286 -9.017492711
3.142857143 -4.731778426
```

Chapter 4

How to set up a scenery in source code

We will build a little scenery of a house with a roof and chimney as well as a simple tree. Further we add a small telescope with a reflective imaging mirror. First we will define the geometry and their surfaces, second we will declare the relations between them. Third and finally we will update all frames relation w.r.t. the root frame to enable fast tracing (post initializing). First we define the main frame of our scenery. The main frame, often called world, will be the root of the scenery tree ??.

../Tests/Examples/set_up_scenery.cpp

```
        Frame world;
        world.set_name_pos_rot("World", Vec3::null,   Rot3::null);
```

Second we define frames that hold individual structures like a tree which will be composed from several objects. The tree will be placed in $x = 5\,\mathrm{m}$ w.r.t. its later mother frame, i.e. the wolrd.

../Tests/Examples/set_up_scenery.cpp

```
        Vec3 tree_pos(5.0,  0.0,  0.0);
        Frame* tree = world.append<Frame>();
        tree->set_name_pos_rot("My_Tree", tree_pos, Rot3::null);

        Color leaf_green(0,   128,  0);
        Sphere* leaf_ball   = tree->append<Sphere>();
        leaf_ball->set_name_pos_rot("leaf_ball",    Vec3(0.0,  0.0,  2.0),  Rot3::null);
        leaf_ball->set_outer_color(&leaf_green);
        leaf_ball->set_radius(0.5);

        Color wood_brown(64, 64, 0);
        Cylinder* tree_pole   = tree->append<Cylinder>();
        tree_pole->set_name_pos_rot("tree_pole",  Vec3(0.0,  0.0,  0.5),  Rot3::null);
        tree_pole->set_outer_color(&wood_brown);
        tree_pole->set_radius_and_length(0.1,   1.0);
```

Also part of the tree is the wooden pole. ../Tests/Examples/set_up_scenery.cpp

and the rest of the source... ../Tests/Examples/set_up_scenery.cpp