3DEQUALIZER 4

Adjustment Scripting

Science-D-Visions, 2014-12-05

Contents

1 *Introduction*

The python interface has proved to be a powerful way of extending `3DE4`'s capabilities. As of `3DE4/r4b3` we add a new type of python scripts in order to provide more flexibility in parameter adjustment. The purpose of this document is to help you understand how scripts for parameter adjustment differ from traditional scripts.

1.1 Versions

Document version	3DE4 Release	Document published	Changes
2	3DE4/r4b3	unpublished	Minor changes, renamed to "Adjustment Scripting"
1	3DE4/r4b3	2014-11-18	First release

1.2 Parameter Adjustment

By the time you read this document you most likely know 3DE4's *Parameter Adjustment Window*, since even the most basic 3DE4 project require at least an optimization of focal length or lens distortion. Nonetheless, let us recapitulate what happens in the adjustment procedure. Schematically, parameter adjustment is done as shown in fig. 1.

- All data relevant for the *core* are extracted from 3DE4's database and stored in an object we call the *blob*. Usually, this data structure is transparent for the user. But in order to understand scripts for parameter adjustment, it might be helpful to know. While the database can be very large and contains a lot of data irrelevant for the core, the *blob* is more light-weight and we can copy it at little expenses or send it through the network for multi-host processing.
- The parameter adjustment engine will now vary the parameters marked for adjustment by the user and send the *blob* to the *core* for computation (red loop).
- The results are transferred back from the *core* to the parameter adjustment engine. In an adaptive adjustment, the resulting deviation controls how the engine will vary the parameters for subsequent calculations (red loop).
- The application periodically queries parameter values and the corresponding deviation from the engine and displays them in the GUI.

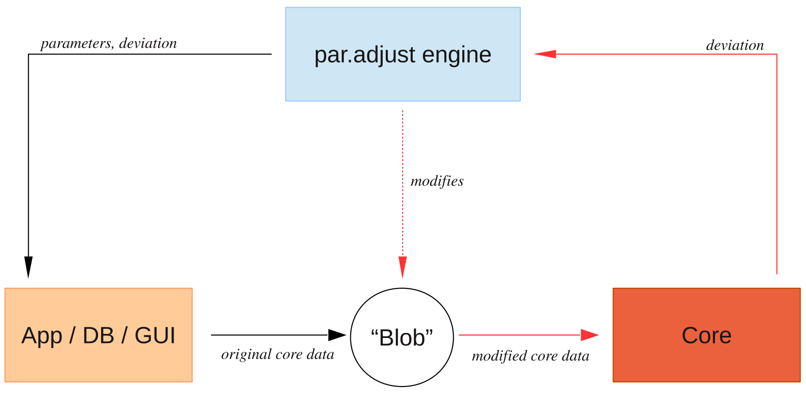


Fig. 1: *Parameter adjustment*

Without adjustment scripting it is "hard-wired" in 3DE4's GUI, which parameters can be adjusted. Each adjustable parameter has an option menu or a toggle button in order to mark it for adjustment. The problem is, that you may want to adjust parameters which do not have a setting for adjustment, like for instance scaling and offset of a given curve or function values of selected curve vertices.

1.3 Adjustment Scripting

As of version 3DE4/r4b3 we have introduced a feature named "adjustment scripting". Adjustment scripting always requires **two scripts**: the first one is called the *gui script*, the second one is called the *modifier script*. The *gui script* is executed directly as a consequence of a user action, like any other script you know. Its main purpose is to create entries in 3DE4's *Parameter Adjustment Window*. A parameter created this way does not have any meaning; it only has a name and a range, like built-in parameters as well. However, each of these parameters contains the name of a second script, the *modifier script*, and it contains a set of arguments to be passed to that script. We will see this in an example later.

2 *Related python commands*

There is a small set of python commands required for adjustment scripting. Some of them are only useful inside the *gui script*, others will only appear in the *modifier script*. We'll have a look now at the function definitions. If you would like to see an example first, please read section 2.2.3.

2.1 The *gui script*

As mentioned, the *gui script* is just a common python script. It appears in 3DE4's GUI as a button or a menu item. Typically, at some point a *gui script* contains one or more calls of the function

```
tde4.createParameterAdjustScript(<script>,<par_name>,<custom_pars>,<range_from>,<range_to>).
```

For each call a script parameter entry is created in the parameter adjustment window.

- The first argument `<script>` is the (absolute or relative) path to the *modifier script* in the file system.
- The second argument `<par_name>` is the name of the parameter.
- The third parameter `<custom_pars>` allows to pass context to the parameter. The adjustment engine needs to know which lens object the parameter is associated to, so it must be passed here. `custom_pars` is a general string. You define the format for this string by yourself. The *modifier script* will extract and interpret this custom parameter string. For instance, you are going to optimize a property for one of your lenses. Then `custom_pars` will most likely contain the ID or the name of the lens.
- Arguments four and five contain the range for this parameter.

2.2 The *modifier script*

The following python commands only make sense within a modifier script. In this section we will discuss them in detail.

```
<double> tde4.getParameterAdjustCurrentValue()
<string> tde4.getParameterAdjustCustomParameters()
<0|1>    tde4.getParameterAdjustPhaseRunningFlag()
         tde4.applyParameterAdjustDataModification(<selector_string>,<double>)
```

As mentioned in the previous section, parameter adjustment consists of two successive phases which we shall label "running: 1" and "running: 0" in the following. The script will know in which of these phases it is running by calling the function

```
<0|1>    tde4.getParameterAdjustPhaseRunningFlag()
```

Each *modifier script* will distiguish between these two phases. Let us have a look at the phases now.

2.2.1 *Running: 1*

During parameter adjustment the adjustment engine generates values for each parameter. In a brute-force adjustment, these values will strictly lie within the range of the parameter, while in adaptive adjustment it may leave the range. For each set of values to be dispatched by the *core* the engine invokes the *modifier script* of all script parameters in a well-defined order. Using the function

```
tde4.getParameterAdjustCurrentValue()
```

the current value is obtained from the engine. The task of the *modifier script* is now to place this value in the *blob* by means of a *selector* string, so that the *core* will use it for the subsequent calculation. The selector addresses the piece of data in the *blob* which is going to be modified. In order to build the selector you will need some contextual information like e.g. which lens, camera, point group, or point is going to be modified. This context was passed in the *gui script*, and we extract it here by means of the command

```
<string> tde4.getParameterAdjustCustomParameters().
```

In section 3 all selector patterns relevant for parameter adjustment are listed. Once the selector is constructed, we modify the *blob* by the command.

```
tde4.applyParameterAdjustDataModification(selector,value)
```

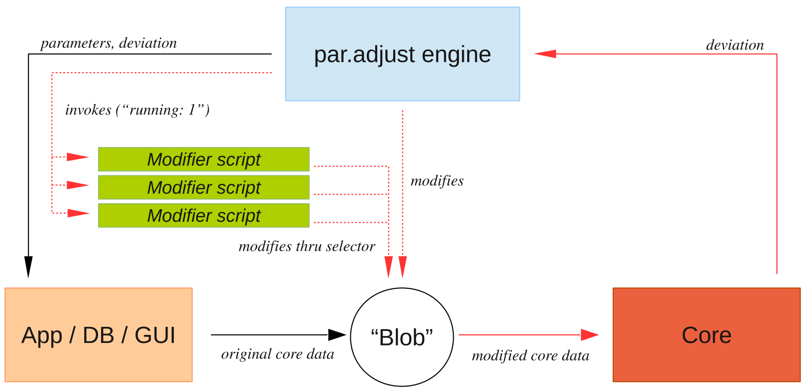


Fig. 2: *Phase "running: 1" of parameter adjustment*

2.2.2 *Running: 0*

In this phase the optimal value of each parameter is transferred to 3DE4's database. The parameter adjustment engine and the core are inactive. The function

```
tde4.getParameterAdjustCurrentValue()
```

will now deliver the optimal value determined in phase "running: 1". The *modifier script* transfers this value to 3DE4's database. When this is done for each parameter, the adjustment procedure is complete.

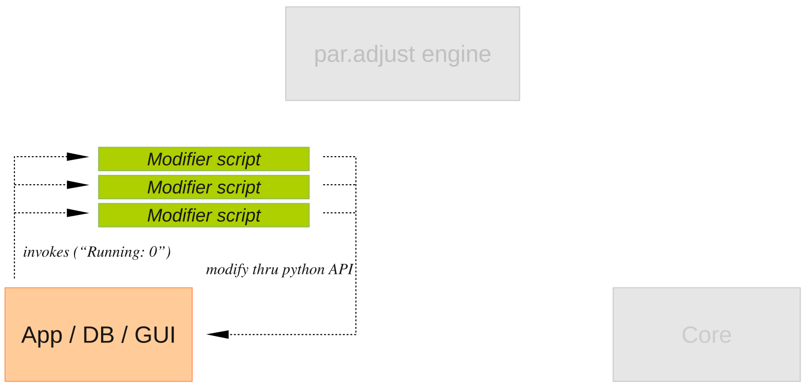


Fig. 3: *Phase "running: 0" of parameter adjustment*

2.2.3 *Example: Adjusting distortion vertices*

The following example demonstrates, how adjustment scripting can be used for adjusting vertices of lens distortion curves for focus driven lens distortion. As mentioned, we need two scripts, the *gui script* and the *modifier script*. They are called

`adjust_distortion_vertices_gui.py`

and

`adjust_distortion_vertices.py`

As a convention we suggest to choose the same base name for both scripts and add "_gui" for the *gui script*. Let us have a look at the *gui script* first:

```
# 3DE4.script.name: Adjust Distortion Vertices...
# 3DE4.script.version:  v0.5 (demo)
# 3DE4.script.gui:  Main Window::Adjustment
#
```

```
# 3DE4.script.comment:  Add all selected vertices to parameter adjustment window
# 3DE4.script.comment:  in order to adjust their value. All vertices must be set to "LINEAR"

import os
import string

# exceptions
class key_is_not_linear(BaseException):
    pass
```

So far, nothing unusual has happend. A *gui script* is a common script, which is supposed to appear somewhere in 3DE4's GUI, in this case it's the menu *Adjustment* in the *Main Window*. We have defined a simple exception class, because the script only works properly if all selected vertices are set to *LINEAR*.

```
# main function
def adj_dist_vert_main():
# Current camera and lens
    id_cam = tde4.getCurrentCamera()
    id_lens = tde4.getCameraLens(id_cam)

# Get lens distortion model of current lens
    model = tde4.getLensLDModel(id_lens)
    num_par = tde4.getLDModelNoParameters(model)
```

We are interested in the distortion model of the current lens. A more sophisticated version of this script would also allow adjusting vertices of curves from more than one lens, but for our purposes this is sufficient. Next, we pick the filename of the *modifier script*. In this example both scripts are located in my home directory:

```
# Location of adjustment script. In this case it's in my home directory.
    script = os.getenv("HOME") + "/.3dequalizer/py_scripts/adjust_distortion_vertices.py"
```

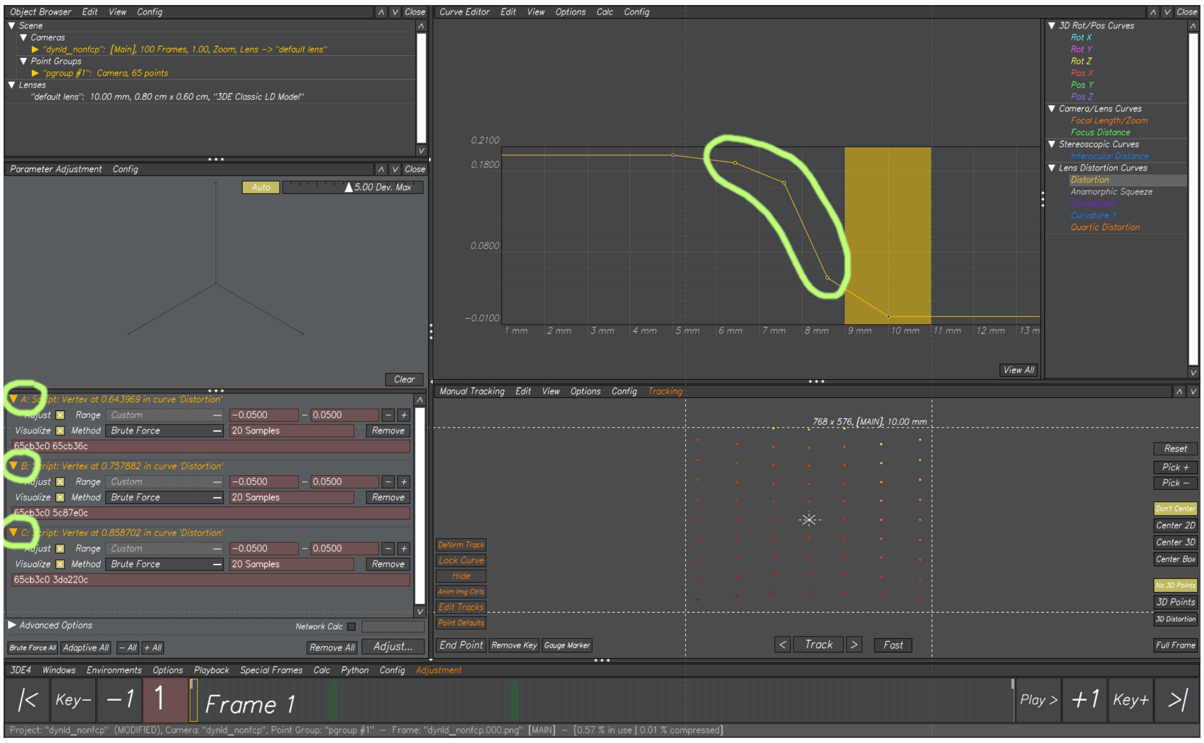
The script is supposed to create parameters in the *Parameter Adjustment Window* for each selected vertex of each lens distortion parameter. So, this is what we do next.

```
# Run through parameters and get the curve.
    ids_key = []
    for i in range(num_par):
        name_par = tde4.getLDModelParameterName(model,i)
        id_curve = tde4.getLensLDAdjustableParameterCurve(id_lens,name_par)
# Run through selected vertices for this curve
        for id_key in tde4.getCurveKeyList(id_curve,True):
            mode    = tde4.getCurveKeyMode(id_curve,id_key)
            if mode != "LINEAR":
                raise key_is_not_linear()
            value = tde4.getCurveKeyPosition(id_curve,id_key)
# In r4b3 parameter names must be unique. We encode the x-position
# and the curve identifier in the parameter name. As of r4b4 this won't
# be required any more, and a simple name will do. But for now:
            par_name = "Vertex at %f in curve '%s'" % (value[0],name_par)
```

Note, how we construct the custom parameter string. As mentioned, we are free to choose any format we like for this string. Concatenating strings with white space will do the job here. The range values we pass here are interpreted by the *modifier script* as relative values. We will get back to this in a minute when we discuss the *modifier script*.

```
# Tell the modifier script what to modify
            custom_pars = id_curve + " " + id_key
# Range. This is a demo script. In practice it would be nice to enter them in a GUI.
            range_from,range_to = -0.05,+0.05
# Now create the parameter entry:
            tde4.createParameterAdjustScript(script,par_name,custom_pars,range_from,range_to)
try:
    adj_dist_vert_main()
except(key_is_not_linear):
    print "For this script, all selected vertices must be set to mode 'LINEAR'."
    print "See Curve Editor->Edit->Set CVs to->Linear"
```

If we invoke this script, the following happens. Assume, we have three selected vertices in the distortion curve of parameter "Distortion". Then the script will create three entries in the *Parameter Adjustment Window*, as shown in the figure below.



Now let us have a look at the *modifier script*. In contrast to the *gui script*, this is not supposed to appear anywhere in 3DE4's GUI. For this reason, the script starts as follows:

```
# 3DE4.script.hide: true
```

Now we determine the phase, i.e. is the parameter adjust engine still running (1) or do we have to move the result to 3DE4's database. Also, we evaluate the custom parameter string passed to the parameter entry in the *gui script*.

```
running = tde4.getParameterAdjustPhaseRunningFlag()
paras   = tde4.getParameterAdjustCustomParameters()
id_curve,id_key = paras.split()
```

Remember, that this script is invoked over and over again during parameter adjustment. Each time it is invoked the parameter this script belongs to may have a different value. The task is now to modify the *blob* and make sure the *core* does its computations with this value. As mentioned the range parameters are interpreted relative to the current value in database. The variable `value` ranges from `-0.05` to `+0.05` in our demo script, but we add it to the current database value before writing it to the *blob*. The advantage of this practice is that once you have a result you can start another parameter adjustment without losing the previous result.

```
# During adjustment, this is the value currently used by core.
# After adjustment, it's the best value found during adjustment.
value   = tde4.getParameterAdjustCurrentValue()

# adjustment is running
if running == 1:
# We build the selector. An ID is marked by a leading '@'.
    mod = "@%s.py" % id_key
    px,py = tde4.getCurveKeyPosition(id_curve,id_key)
    tde4.applyParameterAdjustDataModification(mod,py + value)
```

After optimization, i.e. in phase "running: 0", the best value is transferred back to the database:

```
# adjustment is done, now copy best results back into 3DE's database...
if running == 0:
# We won't change the x-coordinate of the vertex, only the y-value.
```

```
    px,py = tde4.getCurveKeyPosition(id_curve,id_key)
    tde4.setCurveKeyPosition(id_curve,id_key,[px,py + value])
```

and that's all. The three parameters should behave similar to non-script parameters. There is, however, the following problem: As you have seen we pass database IDs to the script parameter. When you save the project and reopen it, these ids are meaningless. That means the parameter entries have to be deleted, and the *gui script* must be invoked again. This is clearly a drawback of using database IDs. For other objects like lenses, cameras, point groups and points, this can be avoided by adressing them by their name instead. For curve vertices, which do not have a name, this is not possible.

2.3 Error messages

Syntax errors or generally errors which occur when the python interpreter is parsing the script appear in the *Python Console Window*. For traditional scripts and also for *gui scripts* the python console pops up automatically, while for *modifier scripts* it does not (currently). So it might be a good idea to keep it open while you are writing and testing your scripts.

3 *Selectors*

In this section we will see how the selector string for the function

```
tde4.applyParameterAdjustDataModification(selector,value)
```

is constructed. As mentioned before, this function appears in the *modifier script*. Its purpose is to modify the *blob*. The selectors reflect the tree-like structure of the *blob*.

3.1 Syntax

The selector syntax itself is not very complicated. The main problem in writing a *modifier script* is finding the appropriate selector. As a representation of the *blob* the selectors strictly correspond to the *core*-way of viewing `3DE4`'s database. This is partially different from the GUI-view. Let us start with some definitions.

- A *selector* is a sequence of one or more segments separated by dots.
- A *segment* is one of the following:
  - a literal like `cameras`, `lenses`, `frames`, ...
  - a placeholder like <id> or <index>.
- The placeholder <id> is one of the following:
  - The character "@" followed by an ID from `3DE4`'s database, like `@1234abcd` where `1234abcd` could be for instance the return value of `tde4.getFirstLens()`. You most likely know these database IDs from other scripts.
  - The name of a database object, enclosed in double quotes, like `"mylens"`.
  - An integer number i which addresses the i-th item of some list in `3DE4`'s database.
- The placeholder <index> is an integer number. It adresses an element in an array or list.
- A selector may address a vector. In this case appending "[i]" with an integer argument i addresses component i of this vector.
- A selector may address a matrix. In this case appending "[i][j]" with integer arguments i and j addresses component (i, j) of this matrix.

3.1.1 *Shorthand notation*

When you address an object in the *blob* by its database ID, like e.g. the function value of a curve vertex, it is not necessary to write down the fully qualified selector string, because the database ID identifies the target object uniquely. That means, instead of addressing Vertex `4321dcba` in curve of parameter 5 of lens `abcd1234` like this

```
project.lenses.@abcd1234.lens_distortion_model.pars.5.curve.vertices.@4321dcba.py
```

you may write

```
@4321dcba.py
```

in order to keep things simple. This shorthand notation only works for database IDs, not for names or indices. Names are not unique across object categories and indices require context (the *i*-th element of what?).

3.1.2 *Recommended usage*

As mentioned, the placeholder <id> stands for a database ID, a name or an index. Please consider the following:

- Avoid using indices for lenses, cameras, point group and points. You cannot be sure that an object of one of these categories has the same index in the *blob* as it has in `3DE4`'s database. Objects may be disabled in the database or considered irrelevant for the *core* and therefore not present in the *blob* which obfuscates the correspondence between index and object.
- Consider using names for addressing lenses, cameras, point groups and points if you want to make sure the *modifier script* will be valid after save and reload of a project. If this is not important for you feel free to use IDs instead, since they allow extremly compact selectors.

3.1.3 *Units*

Lengths are always given in centimeters (*cm*), times are usually in seconds (*sec*), if not denoted otherwise (e.g. frames). Angles are alway given in *degree*.

3.2 Reference

In this reference we restrict ourselves to selectors which could be relevant for parameter adjustment, i.e. all float-valued, vector-valued and matrix-valued data elements in the *blob*. The reference tables in the following sections are constructed as in the example below.

`common.head.of.selectors`		Comments
`.example.tail.of.selector`	*type*	Comments
		`tde4.exampleRelatedPythonCommand(...)`
`.another.tail.of.selector`	*type*	Comments
		`tde4.anotherRelatedPythonCommand(...)`

The fully qualified selectors, which you will need in the *modifier script*, are:

```
common.head.of.selectors.example.tail.of.selector
common.head.of.selectors.another.tail.of.selector
```

3.2.1 *Cameras*

`project.cameras.<id>.constraints`		Selectors related to camera constraints, e.g. line constraints, plane constraints, locked channel constraints
`.frame.<index>.position_3d`	*vec3d*	Camera position for locked channel constraints. <index> starts at 1 (convenience selector).
		`tde4.setPGroupPosition3D(pgroup_id,camera_id,frame,vec3d)`
`.position.enforce_y_value`	*num*	*y*-value for plane constraints
		`No command available`
`.rotation.z_roll`	*num*	Constant *z*-roll angle for angular constraints
		`No command available`
`project.cameras.<id>.focal`		Selectors related to camera-based focal length
`.focal.curve.vertices.<id>.px`	*num*	The curve maps frames into focal length values. See section A.1.2

		`tde4.getCameraZoomCurve(camera_id)`
		`tde4.setCurveKeyPosition(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent1(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent2(curve_id,key_id,vec2d)`

`.focal.curve.vertices.<id>.py`	*num*	Frame dependent focal length in *cm*
`.focal.curve.vertices.<id>.t1x`	*num*	Left tangent vector *x*
`.focal.curve.vertices.<id>.t1y`	*num*	Left tangent vector *y*
`.focal.curve.vertices.<id>.t2x`	*num*	Right tangent vector *x*
`.focal.curve.vertices.<id>.t2y`	*num*	Right tangent vector *y*
`.focal.value_cm`	*num*	Static value, associated to camera, see *Attribute Editor → Camera → Lens → Focal Length* `tde4.setCameraFocalLength(camera_id,frame,value)`
`project.cameras.<id>.focus`		Selectors related to camera-based focus distance

`.focus.curve.vertices.<id>.px`	*num*	The curve maps frames into focus distance values. See section A.1.2
		`tde4.getCameraFocusCurve(camera_id)`
		`tde4.setCurveKeyPosition(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent1(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent2(curve_id,key_id,vec2d)`

`.focus.curve.vertices.<id>.py`	*num*	Focus Distance in *cm*
`.focus.curve.vertices.<id>.t1x`	*num*	Left tangent vector *x*
`.focus.curve.vertices.<id>.t1y`	*num*	Left tangent vector *y*
`.focus.curve.vertices.<id>.t2x`	*num*	Right tangent vector *x*
`.focus.curve.vertices.<id>.t2y`	*num*	Right tangent vector *y*
`.focus.value_cm`	*num*	Static value, associated to camera, see *Attribute Editor → Camera → Lens → Focus Distance* `tde4.setCameraFocus(camera_id,frame,value)`
`project.cameras.<id>`		Various camera selectors
`.footage.frame_rate`	*num*	Frame rate in frames per second `tde4.setCameraFPS(camera_id,value)`
`.fov.xa`	*num*	Field of View, left edge, in *unit* coordinates with respect to the image. `tde4setCameraFOV(camera_id,xa,xb,ya,yb)`
`.fov.xb`	*num*	Field of View, right edge
`.fov.ya`	*num*	Field of View, bottom edge
`.fov.yb`	*num*	Field of View, top edge
`.rolling_shutter.timeshift`	*num*	Rolling shutter timeshift in *sec*, see *Attribute Editor → Camera → Rolling Shutter Compensation* `tde4.setCameraRollingShutterTimeShift(camera_id,value)`
`.sync.timeshift`	*num*	Timeshift in frames for synchronized cameras, see *Attribute Editor → Camera → Synchronization → Timeshift* `tde4.setCameraSyncTimeshift(camera_id,value)`
`.weight`	*num*	Weight factor, impact of the camera on point groups, see *Attribute Editor → Camera → Camera → Weight* `tde4.setCameraWeight(camera_id,value)`
`project.cameras.<id>.stereo`		The following selectors address data elements in the *blob* which are related to stereo. Please note, that in the *blob* stereo related data are strictly associated to the **secondary** camera (while in 3DE4's GUI

		they are associated to the primary camera). In the *blob* the primary camera determines a coordinate system and the secondary camera is displaced by a 3d-vector relative to the primary camera. The relationship between the two cameras is described in our document about stereoscopic matchmoving.

`.interocular.curve.vertices.<id>.px`	*num*	The interocular curve maps frames into interocular distances. See section A.1.2
		`tde4.getCameraStereoInterocularCurve(camera_id)`
		`tde4.setCurveKeyPosition(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent1(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent2(curve_id,key_id,vec2d)`

`.interocular.curve.vertices.<id>.py`	*num*	Interocular distance in *cm*
`.interocular.curve.vertices.<id>.t1x`	*num*	Left tangent vector *x*
`.interocular.curve.vertices.<id>.t1y`	*num*	Left tangent vector *y*
`.interocular.curve.vertices.<id>.t2x`	*num*	Right tangent vector *x*
`.interocular.curve.vertices.<id>.t2y`	*num*	Right tangent vector *y*
`.interocular.value_cm`	*num*	Static interocular in *cm* `tde4.setCameraStereoInterocular(camera_id,value)`
`.depth_shift.value_cm`	*num*	Static depth shift in *cm* `tde4.setCameraStereoStaticDepthShift(camera_id,value)`
`.vertical_shift.value_cm`	*num*	Static vertical shift in *cm* `tde4.setCameraStereoStaticVerticalShift(camera_id,value)`

3.2.2 *Lenses*

`project.lenses.<id>`		The following selectors address lens properties. They can be used in *modifier scripts* but it's tricky. Do not try this unless you fully understand the relationship between these parameters. In 3DE4's *Attribute Editor* you see that all parameters are already equipped with adjust buttons, so in most situations 3DE4's capabilities for these parameters should be sufficient. See *Attribute Editor → Lens*.
`.film_aspect`	*num*	Film aspect is filmback width divided by filmback height. `tde4.setLensFilmAspect(lens_id,value)`
`.film_back_width_cm`	*num*	Filmback width in *cm*. The filmback is the area of the camera projection plane which corresponds to the area in the footage marked by the field of view lines in *Overview Controls* (**F1**) and *Manual Tracking Controls* (**F2**) `tde4.setLensFBackWidth(lens_id,value)`
`.film_back_height_cm`	*num*	Filmback height in *cm* `tde4.setLensFBackWidth(lens_id,value)`
`.pixel_aspect`	*num*	Pixel aspect ratio. `tde4.setLensPixelAspect(lens_id,value)`
`project.lenses.<id>`		The following selectors are not quite as tricky as the previous ones. Feel free to use them in your scripts.
`.focal_length_cm`	*num*	Static lens-based focal length in *cm*. `tde4.setLensFocalLength(lens_id,value)`
`.focus_distance_cm`	*num*	Static lens-based focus distance in *cm*. `tde4.setLensFocus(lens_id,value)`

`.lco_x_cm`	*num*	Lens center offset *x* in *cm* `tde4.setLensLensCenterX(lens_id,value)`
`.lco_y_cm`	*num*	Lens center offset *y* in *cm* `tde4.setLensLensCenterY(lens_id,value)`
`project.lenses.<id>.lens_distortion_model`		The following selectors control static and dynamic lens distortion. The adjustable parameters of each lens distortion model are numbered by an index starting at 0. We are using this index notation here as opposed to addressing parameters directly by name since otherwise the selector syntax would explicitly depend on the lens distortion model, which is not possible given the fact that lens distortion models can be plugins. 3DE4's python interface has a function to determine the index from the parameter name.

`.pars.<index>.curve.vertices.<id>.px`	*num*	The curve maps from either focal length or focus distance onto the parameter value, see *Attribute Editor → Lens → Lens Distortion → Dynamic Lens Distortion*. In both cases the *x*-value is given in *cm*. See section A.1.2
		`tde4.getLensLDAdjustableParameterCurve(lens_id,para_name)`
		`tde4.setCurveKeyPosition(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent1(curve_id,key_id,vec2d)`
		`tde4.setCurveKeyTangent2(curve_id,key_id,vec2d)`

`.pars.<index>.curve.vertices.<id>.py`	*num*	The dynamic parameter value.
`.pars.<index>.curve.vertices.<id>.t1x`	*num*	Left tangent vector *x*
`.pars.<index>.curve.vertices.<id>.t1y`	*num*	Left tangent vector *y*
`.pars.<index>.curve.vertices.<id>.t2x`	*num*	Right tangent vector *x*
`.pars.<index>.curve.vertices.<id>.t2y`	*num*	Right tangent vector *y*
`project.lenses.<id>.lens_distortion_model`		As of 3DE4/r4b3 lens parameters may be driven by both focal length and focus distance, see *Attribute Editor → Lens → Lens Distortion → Dynamic Lens Distortion*. As a consequence, the surface functions which map focal length and focus distance to lens distortion for each parameter have to be transferred from 3DE4's database to the *core*. The selectors here allow access to this function.

`.pars.<index>.surface.vertices.<index>.f`	*num*	The function value mapped to from focal length and focus distance.
		`tde4.getLensNo2DLUTSamples(lens_id,para_name)`
		`tde4.setLens2DLUTSample`
		`(lens_id,para_name,index,focal,focus,value)`

`.pars.<index>.surface.vertices.<index>.x`	*num*	Focal length in *cm*
`.pars.<index>.surface.vertices.<index>.y`	*num*	Focus distance in *cm*
`project.lenses.<id>.lens_distortion_model`		If lens distortion is not dynamic at all, the following selector is relevant.

`.pars.<index>.value.d`	*num*	Static distortion value
		`tde4.setLensLDAdjustableParameter`
		`(lens_id,para_name,focal,focus,value)`

3.2.3 *Point Groups*

`project.pgroups.<id>.constraints`	3DE4/r4b3 supports a number of point constraints, namely distance constraints and position constraints. Distance constraints appear as

		objects in *Object Browser → Scene → Point Groups → Constraints*. Position constraints appear in *Attribute Editor → Point → Position XYZ* as toggle buttons beside the point coordinate text fields.
`.distance_constraints.<id>.distance`	*num*	As of 3DE4/r4b3 distance constraints are objects in the database, which have an id like cameras and lenses. This selector addresses the distance constraint specified by <id>. `tde4.setDConstrDistance(pgroup_id,dconstr_id,value)`
`.position_constraints.<id>.radius`	*num*	See appendix A.2. For points with survey type *Approximately Surveyed* this selector addresses the bounding sphere radius for the point around the survey position. `tde4.setPointApproxSurveyRange(pgroup_id,point_id,value)`
`project.pgroups.<id>.points.<id>.blending`		Blending parameters control the impact of the tracking data on the triangulation of points.
`.position.x`	*num*	Size of positional blending zone in *x*-direction in unit coordinates. The point weight increases from 0 at the left or right image edge to 1 at the vertical lines defined by this value. `tde4.setPointPositionWeightBlending (pgroup_id,point_id,horizontal,vertical)`
`.position.y`	*num*	Size of positional blending zone in *y*-direction in unit coordinates. The point weight increases from 0 at the bottom or top image edge to 1 at the horizontal lines defined by this value. `tde4.setPointPositionWeightBlending (pgroup_id,point_id,horizontal,vertical)`
`project.pgroups.<id>.points.<id>.cameras.<id>`		The following selector is a "convenience selectors". It encapsulates a more complicated selector which reflects the structure of tracking data as e.g. in 3DE4's *Timeline Editor*. This structure would make it absurdly complicated to address tracking data in the *blob*, therefore we have this simplified version. The <index> starts at 1 for the first frame, as in 3DE4's GUI. Tracking positions are given in normalized image coordinates, from (0,0) in the lower left corner of the lower left pixel to (1,1) in the upper right corner of the upper right pixel.
`.frame.<index>.position_2d`	*vec2d*	Tracking position in *unit* coordinates `tde4.setPointPosition2D (pgroup_id,point_id,camera_id,frame,vec2d)`
`project.pgroups.<id>.points.<id>.mocap`		
`.filter.position_cutoff`	*num*	Mocap points have a time-dependent position which is subject to filtering. This parameters controls the filter strength. `setPointMocapZDepthFilter(pgroup_id,point_id,value`
`.frame.<index>.position_3d`	*vec3d*	In motion capturing point groups the positions can be pre-defined, like survey data. <index> starts at 1 (convenience selector). `tde4.setPointMoCapSurveyPosition3D (pgroup_id,point_id,camera_id,frame,vec3d)`
`project.pgroups.<id>.points.<id>`		
`.position_3d`	*vec3d*	If the point is exactly survey, this is the position. `tde4.setPointSurveyPosition3D(pgroup_id,point_id,vec3d)`
`.weight`	*num*	The weight factor specifies the impact of the point on the cameras. `tde4.setPointWeight(pgroup_id,point_id,value)`

3.3 Example - communication between *modifier scripts*

In the example in section 2.2.3 we see a clear 1:1 correspondence between parameter entries and objects to be modified: each parameter is connected to exactly one function value of a vertex. Often, this relationship is not so unique. In the following example we will see, how parameters collaborate in order to modify the *blob*. Abstractly speaking, let us assume we have n parameters $p_1 ... p_n$. From these parameters we would like to construct a transformation $t(p_1 ... p_n)$ which is then applied to the *blob*. As an example, imagine you would like to apply an offset d and a scale s to some curve c. In the database we label the curve c_{db}, in the *blob* we call it c_{blob}. The transformation is (that's what we are doing in phase "running: 1"):

$$c_{db} \times s + d \rightarrow c_{blob}$$

If we split this into two *modifier scripts* we get the following:

$$\text{script for } s\text{: } c_{db} \times s \rightarrow c_{blob}$$

$$\text{script for } d\text{: } c_{blob} + d \rightarrow c_{blob}$$

And here's the problem: the second script requires to **read** data from the *blob*. There is no python command for doing so, and we prefer not to implement this for several reasons. Instead we suggest a technique to collect the parameters and to apply them at once. In the scripts shipped with `3DE4/r4b3` and later this technique is called "master/slave", but the sake for clarity we will call it *collect/apply* in the following. The important point in *collect/apply* is that the *modifier scripts* are always executed in the same order as they appear in the *Parameter Adjustment Window*. They are all executed within the same python interpreter, which allows them to communicate with each other through global variables. In case of our n parameters above the *modifier script* should behave as follows in phase "running: 1":

- script for p_1 - Collect: move value to some global variable $p_{1,glob}$
- script for p_2 - Collect: move value to some global variable $p_{2,glob}$

  ...
- script for p_n - Collect: move value to some global variable $p_{n,glob}$

  - Apply: Build transformation $t(p_{1,glob} ... p_{n,glob})$ and apply to *blob*!

In phase "running: 0", i.e. when we collect all optimal values and store them back into the database, the script should do the following:

- script for p_1 - Collect: move value to some global variable $p_{1,glob}$
- script for p_2 - Collect: move value to some global variable $p_{2,glob}$

  ...
- script for p_n - Collect: move value to some global variable $p_{n,glob}$

  - Apply: move values $p_{1,glob} ... p_{n,glob}$ to database!

This still looks a little abstract, so let us have a look at some real-world example. The following script pair allows to apply an offset and a scale value to the interocular curve of a pair of stereo cameras.

3.3.1 *The gui script*

First of all, we do some administrative stuff and determine if it's a proper stereo project. If everything is okay we get a pair of stereo cameras.

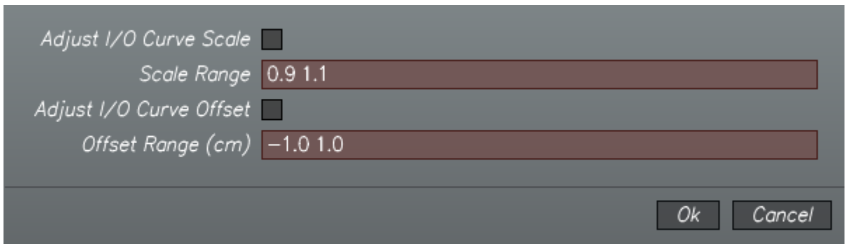
```
# 3DE4.script.name: Adjust I/O Curve Scale & Offset (primary right)...
# 3DE4.script.version:  v1.1
# 3DE4.script.gui:  Main Window::Adjustment
# 3DE4.script.comment:  Add scale and offset parameters to 3DE's parameter
# 3DE4.script.comment:  adjustment window for optimizing
# 3DE4.script.comment:  interocular distance curve of a stereo project.

import os

# find primary & secondary stereo camera...
id_cam_prim = None
id_cam_sec  = None
ids_cam = tde4.getCameraList()
for id_cam in ids_cam:
    mode    = tde4.getCameraStereoMode(id_cam)
    if mode == "STEREO_PRIMARY": id_cam_prim = id_cam
    if mode == "STEREO_SECONDARY": id_cam_sec = id_cam
```

In case of a proper stereo project we build a GUI

```
if id_cam_prim == None or id_cam_sec == None:
# Not a stereo project
    tde4.postQuestionRequester("Adjust I/O Curve Scale & Offset...","Primary stereo camera not found.","Ok")
else:
# This is a stereo project. Go ahead and build a GUI.
    req = tde4.createCustomRequester()
    tde4.addToggleWidget(req,"scale_toggle","Adjust I/O Curve Scale",0)
    tde4.addTextFieldWidget(req,"scale_field","Scale Range","0.9 1.1")
    tde4.addToggleWidget(req,"offset_toggle","Adjust I/O Curve Offset",0)
    tde4.addTextFieldWidget(req,"offset_field","Offset Range (cm)","-1.0 1.0")
# Check for Button 1, "Ok"
    if tde4.postCustomRequester(req,"Adjust I/O Curve Scale & Offset...",600,0,"Ok","Cancel") == 1:
# I'm testing this in my home directory
        path    = os.getenv("HOME")
        script  = path + "/.3dequalizer/py_scripts/adjust_io_scale_offset.py"
# Extract range values from GUI
        sr0,sr1 = tde4.getWidgetValue(req,"scale_field").split()
        or0,or1 = tde4.getWidgetValue(req,"offset_field").split()
# Extract adjust flags from GUI
        scale   = tde4.getWidgetValue(req,"scale_toggle")
        offset  = tde4.getWidgetValue(req,"offset_toggle")
```


Adjust I/O Curve Scale ☐
Scale Range 0.9 1.1
Adjust I/O Curve Offset ☐
Offset Range (cm) −1.0 1.0

Finally, we create parameter entries in the *Parameter Adjustment Window*. The if-else-cascade makes sure that the last entry is marked as "apply" while the other of the two entries is marked as "collect". This is only a convention, you can label them the way you like. However, we must ensure that only the call of the *modifier script* from the last parameter entry will modify the *blob*.

```
# We pass the camera names to the modifier script.
        name_cam_prim   = tde4.getCameraName(id_cam_prim)
        name_cam_sec    = tde4.getCameraName(id_cam_sec)
# These are the global variables used by the parameter entries
# where we collect values from per-entry execution of modifier scripts.
# We initialize them here, so we don't ever have to worry elsewhere,
        _adjust_io_curve_scale  = 1.0
        _adjust_io_curve_offset = 0.0
# As custom parameters we pass:
# 1. The parameter to insert (scale,offset)
# 2. The mode (collect/apply)
# 3. The two camera names
        if scale:
            if offset==1:
                custom_pars = "scale collect " + name_cam_prim + " " + name_cam_sec
            else:
                custom_pars = "scale apply " + name_cam_prim + " " + name_cam_sec
            tde4.createParameterAdjustScript(script,"I/O Curve Scale",custom_pars,float(sr0),float(sr1))
        if offset:
            custom_pars = "offset apply " + name_cam_prim + " " + name_cam_sec
            tde4.createParameterAdjustScript(script,"I/O Curve Offset",custom_pars,float(or0),float(or1))
```

3.3.2 *The modifier script*

Now for the *modifier script*. First we extract the current value and the state variables which control the behaviour of the script.

```
# 3DE4.script.hide: true

running = tde4.getParameterAdjustPhaseRunningFlag()
value   = tde4.getParameterAdjustCurrentValue()
para    = tde4.getParameterAdjustCustomParameters().split()
```

Now, for each parameter entry we set the global variable to our current value. In other words we "collect" the values to be applied to the *blob*.

```
# For each parameter the script passes its value to one
# of the global variables. We do this for both, the "collect"
# and the "apply" pass.
if para[0] == "scale":
    _adjust_io_curve_scale = value
if para[0] == "offset":
    _adjust_io_curve_offset = value
```

The last entry will tell us to apply all values to the *blob*. As in every *modifier script* we have to distinguish between phases "running: 1" and "running: 0".

```
if para[1] == "apply":
# create io curve modification ("apply")...
    if running == 1:
# interocular is associated to primary camera in 3DE4's database.
        id_cam_prim = tde4.findCameraByName(para[2])
# retrieve the previously collected scale & offset parameters
        scale   = _adjust_io_curve_scale
        offset  = _adjust_io_curve_offset

        id_crv  = tde4.getCameraStereoInterocularCurve(id_cam_prim)
        ids_key = tde4.getCurveKeyList(id_crv,0)
        for id_key in ids_key:
            pos = tde4.getCurveKeyPosition(id_crv,id_key)
            t1d = tde4.getCurveKeyTangent1(id_crv,id_key)
            t2d = tde4.getCurveKeyTangent2(id_crv,id_key)
```

Each vertex in the database has a counterpart in the *blob*. We use the shorthand notation from section 3.1.1, in order to keep the selectors simple. Please note that we modify the curve as described in section A.1.2.

```
            tde4.applyParameterAdjustDataModification("@%s.py"  % id_key,pos[1] * scale + offset)
            tde4.applyParameterAdjustDataModification("@%s.t1y" % id_key,t1d[1] * scale)
            tde4.applyParameterAdjustDataModification("@%s.t2y" % id_key,t2d[1] * scale)
```

In phase "running: 0" we store the optimal values in the database.

```
# copy best results back into 3DE's database...
    if running == 0:
# interocular is associated to primary camera in 3DE4's database.
        id_cam_prim = tde4.findCameraByName(para[2])
# retrieve scale & offset parameters...
        scale   = _adjust_io_curve_scale
        offset  = _adjust_io_curve_offset

        id_crv  = tde4.getCameraStereoInterocularCurve(id_cam_prim)
        ids_key = tde4.getCurveKeyList(id_crv,0)
        for id_key in ids_key:
            pos = tde4.getCurveKeyPosition(id_crv,id_key)
            t1d = tde4.getCurveKeyTangent1(id_crv,id_key)
            t2d = tde4.getCurveKeyTangent2(id_crv,id_key)
```

The rules for transforming curves also apply when we modify the database:

```
            tde4.setCurveKeyPosition(id_crv,id_key,[pos[0],pos[1] * scale + offset])
            tde4.setCurveKeyTangent1(id_crv,id_key,[t1d[0],t1d[1] * scale])
            tde4.setCurveKeyTangent2(id_crv,id_key,[t2d[0],t2d[1] * scale])

# "apply" pass: reset global variables to default values...
    _adjust_io_curve_scale  = 1.0
    _adjust_io_curve_offset = 0.0
```

A *Appendix*

A.1 Curves

A.1.1 *Domain*

The domain of curves which map frames into some other quantity like e.g. focal length or interocular always starts at 1 and ends at number-of-frames, regardless of the value in *Attribute Editor → Live Action Footage → First Frame is Frame* or *Calc → Frame Range Calculation*. This is compatible to the python commands which modify curves in the database.

A.1.2 *Modifying curves*

In the example you have seen how single vertices in the special case of a piecewise linear curve can be adjusted. In practice, another way of modifying curves will occur as well, namely applying scale and offset to a given curve. There is a mathematical detail to consider, which we shall discuss in the following. The selectors of a curve look like:

```
project...curve.vertices.<id>.px
project...curve.vertices.<id>.py
project...curve.vertices.<id>.t1x
project...curve.vertices.<id>.t1y
project...curve.vertices.<id>.t2x
project...curve.vertices.<id>.t2y
```

Here, `.px` stands for the x-value of the curve, whatever it may be, time/frame, focal length, focus distance. `.py` represents the function value. `.t1x`/`.t1y` is the left tangent and `.t2x`/`.t2y` the right one. Let us assume you wish to apply an offset d and a scale s to the function values, so d and s are your self crafted parameters. Then function values are mapped e.g. like

```
.py -> .py * s + d
```

Please do **not** forget to apply the scale (and only the scale) s to the tangents as well:

```
.t1y -> .t1y * s
.t2y -> .t2y * s
```

Otherwise the curve will lose its shape during parameter adjustment. For surfaces, this problem does not arise, because tangents are computed automatically and cannot be edited.

A.1.3 *A note on interocular curves*

There is a peculiarity about interocular curves, which we describe in the following. 3DE4's GUI understands interocular as a *distance* between the left and the right camera. Since each of the stereo cameras is marked as *Lefthand Camera* or *Righthand Camera* in *Camera → Stereoscopic → Orientation* this value is meant to be positive. The *core* however interprets interocular as the x-component of a vector, along with vertical shift and depth shift. As a consequence, whenever the primary camera is the righthand camera, interocular changes sign in the core. For the *modifier script* in section 3.3.2 this means that

```
scale   = _adjust_io_curve_scale
offset  = _adjust_io_curve_offset
```

should be replaced by

```
if tde4.getCameraStereoOrientation(prcam) == "STEREO_RIGHT":
        scale   = -_adjust_io_curve_scale
        offset  = -_adjust_io_curve_offset
else:
        scale   = _adjust_io_curve_scale
        offset  = _adjust_io_curve_offset
```

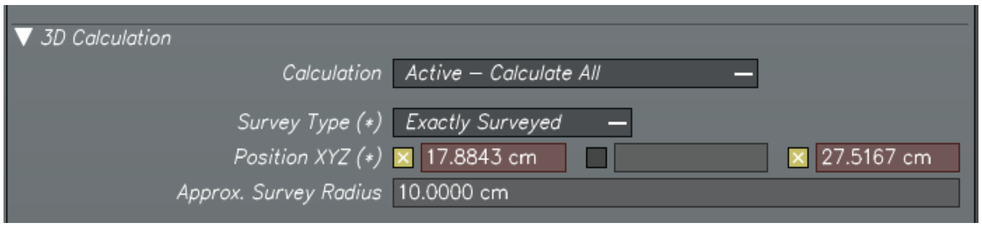
We apologize for the inconveniences. If you have trouble with your interocular script please let us know.

A.2 Position point constraints

Position point constraints look different in 3DE4's GUI and the *blob*. Since it might be interesting to build adjustment script involving these constraints we will have a look at the details.

A.2.1 *Line constraint*

In the following screenshot you see a point which is subject to a line constraint. The *Survey Type* says it's *Exactly Surveyed*, but only two of the three components are activated. This could be a point about which we know the projection onto the floor but not its altitude.



▼ 3D Calculation
Calculation: Active – Calculate All
Survey Type (*): Exactly Surveyed
Position XYZ (*): ☒ 17.8843 cm ☐ ☒ 27.5167 cm
Approx. Survey Radius: 10.0000 cm

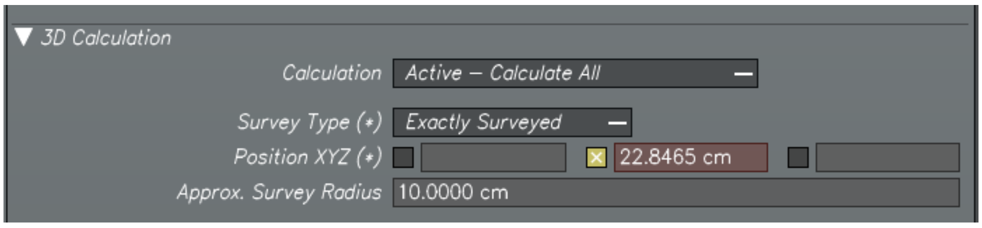
Let us assume you wish to adjust one of the known positional components because you are not quite sure about its value, say *z*. Then the selector for modifying the *blob* is

```
project.pgroups.<id>.points.<id_point>.position_3d[2]
```

where **[2]** refers to the *z*-component (*x*: [0], *y*: [1], *z*: [2]) of the vector-valued selector.

A.2.2 *Plane constraint*

In the following screenshot you see a point with well-known *y*-position but unknown *x*- and *z*-position. This represents a point lying somewhere in a plane with known altitude. Often, the known component is 0, because you define the floor level by these constraints.



▼ 3D Calculation
Calculation: Active – Calculate All
Survey Type (*): Exactly Surveyed
Position XYZ (*): ☐ ☒ 22.8465 cm ☐
Approx. Survey Radius: 10.0000 cm

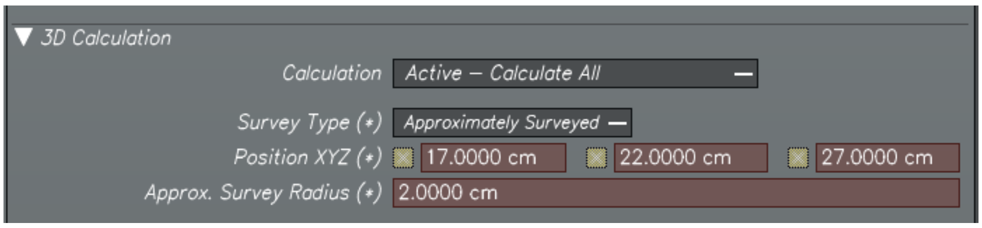
If, for some reason, you want to write a parameter adjustment script in order to optimize *y*, the selector is

```
project.pgroups.<id>.points.<id_point>.position_3d[1]
```

similar to the case of the line constraint.

A.2.3 *Ball constraint (Approximately Surveyed)*

When a point has approximate survey data in the language of the *core* this is a "ball constraint" (ball in the sense of a solid sphere). The point is allowed to float around within a ball of a given radius during core calculation. From the *core* point of view this is simply another constraint like plane or line constraint.



▼ 3D Calculation
Calculation: Active – Calculate All
Survey Type (*): Approximately Surveyed
Position XYZ (*): ▨ 17.0000 cm ▨ 22.0000 cm ▨ 27.0000 cm
Approx. Survey Radius (*): 2.0000 cm

Please note, that the survey type is now *Approximately Surveyed* and all three components are set. Additionally, the text field *Approx. Survey Radius* is now sensible. All of these four values may be changed by a *modifier script* during parameter adjustment. The selectors are:

```
project.pgroups.<id>.points.<id_point>.position_3d[0]
project.pgroups.<id>.points.<id_point>.position_3d[1]
project.pgroups.<id>.points.<id_point>.position_3d[2]
project.pgroups.<id>.constraints.position_constraints.<id_point>.radius
```

Please note that the constraint is addressed by means of the id of the point it belongs to. The reason for this is that it allows us to merge all constraints (which are all handled the same way by the *core*) at one place in the *blob* as a property `.constraints` of the point group.