

Ayam Custom Objects Writers Guide

Randolf Schultz (rschultz@informatik.uni-rostock.de)

2. Sep 2001

This document describes how to write a custom object for Ayam.

Contents

1	Introduction	2
2	Preparation	2
3	Functions	3
3.1	Initialization (mandatory)	3
3.2	Create (mandatory)	5
3.3	Delete (mandatory)	6
3.4	Copy (mandatory)	7
3.5	Draw	7
3.6	Shade	8
3.7	Drawh - Draw Handles	8
3.8	SetProp - Set Properties	9
3.9	GetProp - Get Properties	10
3.10	GetPoint, Single Point Editing	11
3.11	Write/Read (mandatory)	13
3.12	Wrib - Write RIB	14
3.13	BBC - Bounding Box Calculation	15
3.14	Notification	16
3.15	Conversion	16
3.16	Provide	16
3.17	Tree-Drop	17
4	Custom Objects from the Tcl Side of Life	17
4.1	Property GUI Elements	18
4.1.1	Labels	19
4.1.2	Parameters	19
4.1.3	Menus	19

4.1.4	Checkbuttons	19
4.1.5	Color Selectors	20
4.1.6	File Selectors	20
4.1.7	String Entries	20
4.1.8	Buttons	20
4.2	The Custom Menu	21
5	Compiling, Installing, Loading a Custom Object	21

1 Introduction

What is a custom object? A custom object is a piece of shared code that gets loaded at runtime into the core of Ayam to allow modeling with new types of geometric objects (a plug-in-system for object types).

You should take a look at (or print out) the files "plugins/csphere.c" and "plugins/csphere.tcl" from the src directory of the Ayam distribution. They contain a more or less documented example custom object that does nothing exciting, but implements a simple sphere. Additionally, they may serve as starting point for your own custom object. Just copy the files and rename all occurrences of the string "csphere" to your own new name.

Take a look at the modules "extrude.c", and "icurve.c" in the objects sub directory of the sources. They implement objects too, but support some additional advanced functionality: notification (extrude), single point editing (icurve), conversion (extrude, icurve), and provide mechanism (icurve).

2 Preparation

You decided to implement a new custom object? Fine, here is what to do first:

- Pick a short, pregnant, descriptive, non colliding name for your object type. To avoid clashes, I suggest to precede the name with the initials of the author (yours) as it is tradition with RenderMan shaders.
- Copy "csphere.c" and "csphere.tcl" to "yourname.c" and "yourname.tcl" respectively.
- Rename all occurrences of csphere in those files to yourname.
- Edit the struct that holds the parameters of your object. This struct will be referred to later as the custom object.

Here is the corresponding code snippet from the csphere example:

```
#include "ayam.h"

/* csphere.c - csphere custom object */

static char *csphere_name = "CSphere";

static unsigned int csphere_id;
```

```
typedef struct csphere_object_s
{
    char closed;
    char is_simple;
    double radius;
    double zmin, zmax;
    double thetamax;
} csphere_object;
```

Now you are ready to adapt the implementation of some functions as they are described in the next sections of the manual.

3 Functions

A custom object implements a set of functions that are called from the Ayam core now and then.

Some functions are mandatory, some that would e.g. implement point editing support or notification are not.

3.1 Initialization (mandatory)

```
int
Yourname_Init(Tcl_Interp *interp)
```

Initialization of a custom object is done by a function named `$yourname_Init(Tcl_Interp *interp)`. Note, that `$yourname` is the filename of the shared object that gets loaded, and that the first character of the name has to be uppercase, regardless of the case of the file name. It is a good idea to mimic the naming strategy of the `csphere` example.

The initialization is called once when the custom objects code gets loaded into Ayam.

What does the initialization do? First, it registers the custom object at the Ayam core by calling:

```
ay_otype_register().
```

```
int ay_otype_register(char *name,
                    ay_createcb *crtcb,
                    ay_deletecb *delcb,
                    ay_copycb *copycb,
                    ay_drawcb *drawcb,
                    ay_drawcb *drawhcb,
                    ay_drawcb *shadecb,
                    ay_propcb *setpropcb,
                    ay_propcb *getpropcb,
                    ay_getpntcb *getpntcb,
                    ay_readcb *readcb,
                    ay_writecb *writecb,
                    ay_wribcb *wribcb,
```

```

    ay_bbccb      *bbccb,
    unsigned int *type_index);

```

If the registration fails, the function returns a nonzero value. You should really check this return value, and if it is nonzero call `ay_error()` with it and return `TCL_ERROR`.

The parameters of the function `ay_otype_register()` are as follows:

- `char *name`: pointer to a global string, containing the custom object type name
- some pointers to functions; the functions will be explained in the next sections
- `unsigned int *type_index`: pointer to a global variable that holds an identifier for registered types (will be filled by `ay_otype_register()`), this identifier may be used for type checking and for registration of special functionality.

After the call to `ay_otype_register()`, callbacks for special functionality (notification, conversion, or provide mechanism) may be registered.

Furthermore, you may specify with a call to `ay_matt_nomaterial()`; that your new object type shall not be associable with material objects, e.g. if it is just a geometric helper object that does not get exported to RIB.

Some new Tcl commands for additional functionality not covered by the provided standard and special callbacks may be registered now by calling `Tcl_CreateCommand()`.

Finally, it is time to load some Tcl code from Tcl files into the interpreter. This code implements the property GUIs of the object type specific properties (do not worry, this is easy).

Csphere example:

```

/* note: this function _must_ be capitalized exactly this way
 * regardless of filename (see: man n load)!
 */
int
Csphere_Init(Tcl_Interp *interp)
{
    int ay_status = AY_OK;
    char fname[] = "csphere_init";

    ay_status = ay_otype_register(csphere_name,
                                csphere_createcb,
                                csphere_deletecb,
                                csphere_copycb,
                                csphere_drawcb,
                                NULL, /* no points to edit */
                                csphere_shadecb,
                                csphere_setpropcb,
                                csphere_getpropcb,
                                NULL, /* no picking */
                                csphere_readcb,
                                csphere_writecb,
                                csphere_wribcb,

```

```

        csphere_bcccb,
        &csphere_id);

if(ay_status)
{
    ay_error(AY_ERROR, fname, "Error registering custom object!");
    return TCL_ERROR;
}

/* source csphere.tcl, it contains Tcl-code to build
   the CSphere-Attributes Property GUI */
if((Tcl_EvalFile(interp, "csphere.tcl")) != TCL_OK)
{
    ay_error(AY_ERROR, fname,
            "Error while sourcing \\\"csphere.tcl\\\"!");
    return TCL_OK;
}

ay_error(AY_EOUTPUT, fname,
        "CustomObject \\\"CSphere\\\" successfully loaded.");

return TCL_OK;
} /* Csphere_Init */

```

The following sections describe the functions that are parameters of `ay_otype_register()`.

3.2 Create (mandatory)

```
int yourname_createcb(int argc, char *argv[], ay_object *o);
```

In this callback you should allocate memory for your new object, and initialize it properly. `Argc` and `argv` are the command line of the `crtOb yourname` Tcl-command (invoked e.g. by the main menu entry `Create/Custom Object/yourname`, by Tcl scripts, or directly from the console). The user may deliver additional arguments to this command, which may be evaluated by this callback.

Furthermore, in this callback you can adjust the attribute property of the newly created object, e.g. hiding it or its children initially. Just set the appropriate flags in the `ay_object` pointed to by `o`. Look up the definition of `ay_object`, in `ayam.h` to see, what may be adapted.

If you want your object be able to have child objects you should set the `o->parent` attribute to `AY_TRUE`. You may create first child objects in your create callback. But note, that each level in the scene hierarchy needs to be terminated properly with a so called `EndLevel` object. Such an object might be created easily using `ay_object_crtendlevel()`:

```

ay_object *my_child = NULL;

/* create my_child here */
...

```

```

/* link my_child */
o->down = my_child;
/* terminate level */
ay_object_crtendlevel(&(my_child->next));

```

If you do not create child objects immediately, but set `o->parent` to true, Ayam will create the EndLevel object automatically for you.

Csphere example:

```

int
csphere_createcb(int argc, char *argv[], ay_object *o)
{
    csphere_object *csphere = NULL;
    char fname[] = "crtcsphere";

    if(!o)
        return AY_ENULL;

    if(!(csphere = calloc(1, sizeof(csphere_object))))
    {
        ay_error(AY_EOMEM, fname, NULL);
        return AY_ERROR;
    }

    csphere->closed = AY_TRUE;
    csphere->is_simple = AY_TRUE;
    csphere->radius = 1.0;
    csphere->zmin = -1.0;
    csphere->zmax = 1.0;
    csphere->thetamax = 360.0;

    o->refine = csphere;

    return AY_OK;
} /* csphere_createcb */

```

3.3 Delete (mandatory)

```
int yourname_deletecb(void *c);
```

In this callback you should free all memory allocated for the custom object, the argument `c` points to one of your custom objects. No type check necessary.

Csphere example:

```
int
csphere_deletcb(void *c)
{
    csphere_object *csphere = NULL;

    if(!c)
        return AY_ENULL;

    csphere = (csphere_object *) (c);

    free(csphere);

    return AY_OK;
} /* csphere_deletcb */
```

3.4 Copy (mandatory)

```
int yourname_copycb(void *src, void **dst);
```

The copy callback is mandatory too, it is vital for clipboard and undo functionality. You should allocate a new object and copy the custom object pointed to by source to the new allocated memory, and finally return a pointer to the new memory in dst. The argument src points to one of your custom objects. No type check necessary.

Csphere example:

```
int
csphere_copycb(void *src, void **dst)
{
    csphere_object *csphere = NULL;

    if(!src || !dst)
        return AY_ENULL;

    if(!(csphere = calloc(1, sizeof(csphere_object))))
        return AY_EOMEM;

    memcpy(csphere, src, sizeof(csphere_object));

    *dst = (void *)csphere;

    return AY_OK;
} /* csphere_copycb */
```

3.5 Draw

```
int yourname_drawcb(struct Togl *togl, ay_object *o);
```

In this callback you should draw your custom object. You do not get a pointer to your custom object as parameter, but a pointer to a Ayam object, which is a step higher in the object hierarchy!

This is, because you may freely decide whether to use the standard attributes stored with every Ayam object. These are for instance affine transformations.

See the example source for information on how to finally get to your custom object.

Csphere example (extract):

```
int
csphere_drawcb(struct Togl *togl, ay_object *o)
{
    csphere_object *csphere = NULL;

    ...

    if(!o)
        return AY_ENULL;

    csphere = (csphere_object *)o->refine;

    if(!csphere)
        return AY_ENULL;

    radius = csphere->radius;

    ...

    return AY_OK;
} /* csphere_drawcb */
```

3.6 Shade

```
int yourname_shadecb(struct Togl *togl, ay_object *o);
```

This callback is basically the same as the draw callback, but the user expects to get a shaded representation of the object.

3.7 Drawh - Draw Handles

```
int yourname_drawhcb(struct Togl *togl, ay_object *o);
```

This callback is not mandatory, and needs just to be implemented if your object supports single point editing. If you want to do this, you should draw with `glPoint()` just the points of your object that may be modified by a single point editing action in this callback.

3.8 SetProp - Set Properties

```
int yourname_setpropcb(Tcl_Interp *interp, int argc, char *argv[],
ay_object *o);
```

Using this callback you copy data of your object type specific properties from the Tcl to the C context. Note the use of `Tcl_IncrRefCount()` and `Tcl_DecrRefCount()` to avoid memory leaks.

Also note, that if your object is used as parameter object for a tool object you should inform the tool object (your parent) about changes now using:

```
ay_status = ay_notify_parent(void);
```

Csphere example:

```
/* Tcl -> C! */
int
csphere_setpropcb(Tcl_Interp *interp, int argc, char *argv[], ay_object *o)
{
    char *n1 = "CSphereAttrData";
    Tcl_Obj *to = NULL, *toa = NULL, *ton = NULL;
    csphere_object *csphere = NULL;
    int itemp = 0;

    if(!o)
        return AY_ENULL;

    csphere = (csphere_object *)o->refine;

    toa = Tcl_NewStringObj(n1,-1);
    ton = Tcl_NewStringObj(n1,-1);

    Tcl_SetStringObj(ton,"Closed",-1);
    to = Tcl_ObjGetVar2(interp,toa,ton,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);
    Tcl_GetIntFromObj(interp,to, &itemp);
    csphere->closed = (char)itemp;

    Tcl_SetStringObj(ton,"Radius",-1);
    to = Tcl_ObjGetVar2(interp,toa,ton,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);
    Tcl_GetDoubleFromObj(interp,to, &csphere->radius);

    Tcl_SetStringObj(ton,"ZMin",-1);
    to = Tcl_ObjGetVar2(interp,toa,ton,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);
    Tcl_GetDoubleFromObj(interp,to, &csphere->zmin);

    Tcl_SetStringObj(ton,"ZMax",-1);
    to = Tcl_ObjGetVar2(interp,toa,ton,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);
```

```

Tcl_GetDoubleFromObj(interp,to, &csphere->zmax);

Tcl_SetStringObj(ton,"ThetaMax",-1);
to = Tcl_ObjGetVar2(interp,toa,ton,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);
Tcl_GetDoubleFromObj(interp,to, &csphere->thetamax);

if((fabs(csphere->zmin) == csphere->radius) &&
    (fabs(csphere->zmax) == csphere->radius) &&
    (fabs(csphere->thetamax) == 360.0))
    {
        csphere->is_simple = AY_TRUE;
    }
else
    {
        csphere->is_simple = AY_FALSE;
    }

Tcl_IncrRefCount(toa);Tcl_DecrRefCount(toa);
Tcl_IncrRefCount(ton);Tcl_DecrRefCount(ton);

return AY_OK;
} /* csphere_setpropcb */

```

3.9 GetProp - Get Properties

```

int yourname_getpropcb(Tcl_Interp *interp, int argc, char *argv[],
ay_object *o);

```

Using this callback you copy data of your object type specific properties from the C to the Tcl context.

Csphere example:

```

/* C -> Tcl! */
int
csphere_getpropcb(Tcl_Interp *interp, int argc, char *argv[], ay_object *o)
{
    char *n1="CSphereAttrData";
    Tcl_Obj *to = NULL, *toa = NULL, *ton = NULL;
    csphere_object *csphere = NULL;

    if(!o)
        return AY_ENULL;

    csphere = (csphere_object *) (o->refine);

    toa = Tcl_NewStringObj(n1,-1);

```

```

ton = Tcl_NewStringObj(n1,-1);

Tcl_SetStringObj(ton,"Closed",-1);
to = Tcl_NewIntObj(csphere->closed);
Tcl_ObjSetVar2(interp,toa,ton,to,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);

Tcl_SetStringObj(ton,"Radius",-1);
to = Tcl_NewDoubleObj(csphere->radius);
Tcl_ObjSetVar2(interp,toa,ton,to,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);

Tcl_SetStringObj(ton,"ZMin",-1);
to = Tcl_NewDoubleObj(csphere->zmin);
Tcl_ObjSetVar2(interp,toa,ton,to,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);

Tcl_SetStringObj(ton,"ZMax",-1);
to = Tcl_NewDoubleObj(csphere->zmax);
Tcl_ObjSetVar2(interp,toa,ton,to,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);

Tcl_SetStringObj(ton,"ThetaMax",-1);
to = Tcl_NewDoubleObj(csphere->thetamax);
Tcl_ObjSetVar2(interp,toa,ton,to,TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY);

Tcl_IncrRefCount(toa);Tcl_DecrRefCount(toa);
Tcl_IncrRefCount(ton);Tcl_DecrRefCount(ton);

return AY_OK;
} /* csphere_getpropcb */

```

3.10 GetPoint, Single Point Editing

```
int yourname_getpntcb(ay_object *o, double *p);
```

This callback enables all the single point editing facilities (including the selection mechanism for single points) for your object.

With this callback you get an object and a point in the local space of that object. You are asked to search through your internal structures for points of yours that match the coordinates given in "p". If there are such points you should build an array of pointers to your points as the following example does:

```

yourname_getpntcb(ay_object *o, double *p)
{
    double *control = NULL, min_distance = ay_prefs.pick_epsilon,
           distance = 0.0;

    /* first, we clear the old array */
    if(ay_point_edit_coords) free(ay_point_edit_coords);
    ay_point_edit_coords = NULL;

```

```

/* now we scan our points for the given coordinates*/
control = array_of_your_points;
for(i = 0; i < max_points; i++)
{
    distance = AY_VLEN((objX - control[j]),
                      (objY - control[j+1]),
                      (objZ - control[j+2]));
    if(distance < min_distance)
    {
        pecoords = &(control[j]);
        min_distance = distance;
    }

    j += 3;
}

if(!pecoords)
    return AY_OK;

/* are the points homogenous? */
ay_point_edit_coords_homogenous = AY_FALSE;

/* now we create the new array */
if(!(ay_point_edit_coords = calloc(1,sizeof(double*))))
    return AY_OUTOFMEM;
/* and fill it */
ay_point_edit_coords_number = 1;
ay_point_edit_coords[0] = pecoords;

return AY_OK;
} /* */

```

The code above does just handle the selection of a single point, it is possible, however, to put an arbitrary number of points in the array at once! This is necessary for the special case of `(p[0] == DBL_MIN) && (p[1] == DBL_MIN) && (p[2] == DBL_MIN)`. If all elements of "p" are "DBL_MIN" you should put all editable points of the object into the array (the user wants to select all points).

The following global variables are in use: `ay_point_edit_coords` (the adress of the array), `ay_point_edit_coords_number` (an integer that tells Ayam, how many pointers are in `ay_point_edit_coords`), `ay_point_edit_coords_homogenous` (a flag that tells Ayam, wether the points are homogenous or not; note, that there can only be points of one type in the array).

Also note, that the Ayam core will poke into the memory you pointed it to later. The core expects the points itself to be arrays of doubles:

```

double a_non_homogenous_point[3];
double a_homogenous_point[4];

```

3.11 Write/Read (mandatory)

```
int yourname_readcb(FILE *fileptr, ay_object *o);
```

These callbacks are for writing and reading Ayam scene files. As you can see in the example source, simple `fprintf()`, `fscanf()` calls are currently in use. Note, that you do not have to worry about your child objects, if there are any, they will be saved automagically.

Csphere example:

```
int
csphere_readcb(FILE *fileptr, ay_object *o)
{
    csphere_object *csphere = NULL;
    int itemp = 0;
    if(!o)
        return AY_ENULL;

    if(!(csphere = calloc(1, sizeof(csphere_object))))
        { return AY_EOMEM; }

    fscanf(fileptr, "%d\n", &itemp);
    csphere->closed = (char)itemp;
    fscanf(fileptr, "%lg\n", &csphere->radius);
    fscanf(fileptr, "%lg\n", &csphere->zmin);
    fscanf(fileptr, "%lg\n", &csphere->zmax);
    fscanf(fileptr, "%lg\n", &csphere->thetamax);

    if((fabs(csphere->zmin) == csphere->radius) &&
        (fabs(csphere->zmax) == csphere->radius) &&
        (fabs(csphere->thetamax) == 360.0))
        {
            csphere->is_simple = AY_TRUE;
        }
    else
        {
            csphere->is_simple = AY_FALSE;
        }

    o->refine = csphere;

    return AY_OK;
} /* csphere_readcb */

int
csphere_writecb(FILE *fileptr, ay_object *o)
{
    csphere_object *csphere = NULL;
```

```

if(!o)
    return AY_ENULL;

csphere = (csphere_object *) (o->refine);

fprintf(fileptr, "%d\n", csphere->closed);
fprintf(fileptr, "%g\n", csphere->radius);
fprintf(fileptr, "%g\n", csphere->zmin);
fprintf(fileptr, "%g\n", csphere->zmax);
fprintf(fileptr, "%g\n", csphere->thetamax);

return AY_OK;
} /* csphere_writecb */

```

3.12 Wrib - Write RIB

```
int yourname_wribcb(char *file, ay_object *o);
```

This callback is for exporting your object to a RIB. Just use the appropriate Ri-calls. Just like the drawing callbacks you get a pointer to a Ayam object and not to your custom object. Csphere example:

```

int
csphere_wribcb(char *file, ay_object *o)
{
    csphere_object *csphere = NULL;

    ...

    if(!o)
        return AY_ENULL;

    csphere = (csphere_object*)o->refine;

    if(!csphere->closed)
    {
        RiSphere((RtFloat)csphere->radius,
                (RtFloat)csphere->zmin,
                (RtFloat)csphere->zmax,
                (RtFloat)csphere->thetamax,
                NULL);
    }

    ...

```

```

return AY_OK;
} /* csphere_wribcb */

```

3.13 BBC - Bounding Box Calculation

```
int yourname_bbccb(ay_object *o, double *bbox, int *flags);
```

This callback is for the calculation of bounding boxes. "bbox" points to an array of 24 doubles (describing 8 points), the bounding box. You may put additional information into "flags" to tell the core that you:

1. have a regular bounding box (leave flags at zero)
2. have a regular bounding box but the boxes of the children should be discarded (set flags to 1)
3. have no own bbox, but children have (set flags to 2)

Csphere example:

```

int
csphere_bbccb(ay_object *o, double *bbox, int *flags)
{
    double r = 0.0, zmi = 0.0, zma = 0.0;
    csphere_object *csphere = NULL;

    if(!o || !bbox)
        return AY_ENULL;

    csphere = (csphere_object *)o->refine;

    r = csphere->radius;
    zmi = csphere->zmin;
    zma = csphere->zmax;

    /* XXXX does not take into account ThetaMax! */

    /* P1 */
    bbox[0] = -r; bbox[1] = r; bbox[2] = zma;
    /* P2 */
    bbox[3] = -r; bbox[4] = -r; bbox[5] = zma;
    /* P3 */
    bbox[6] = r; bbox[7] = -r; bbox[8] = zma;
    /* P4 */
    bbox[9] = r; bbox[10] = r; bbox[11] = zma;

    /* P5 */
    bbox[12] = -r; bbox[13] = r; bbox[14] = zmi;
    /* P6 */

```

```

bbox[15] = -r; bbox[16] = -r; bbox[17] = zmi;
/* P7 */
bbox[18] = r; bbox[19] = -r; bbox[20] = zmi;
/* P8 */
bbox[21] = r; bbox[22] = r; bbox[23] = zmi;

return AY_OK;
} /* csphere_bcccb */

```

3.14 Notification

```
int yourname_notifycb(ay_object *o);
```

The notification callback is for custom objects that rely on other objects to be children of them (e.g. Revolve).

The notification callback is to inform you, that something below your custom object has changed, and you should probably adapt the custom object to the change. The Revolve object e.g. redoes the revolution.

Notification callbacks have to be registered in the initialization callback using:

```
int ay_notify_register(ay_notifycb *notcb, unsigned int type_id);
```

3.15 Conversion

```
int yourname_convertcb(ay_object *o);
```

Conversion callbacks are in use for objects that may be converted to objects of a different type, e.g. the interpolating curve object (ICurve) may be converted to a NURBCurve object, or the Revolve object may be converted to a NURBPatch object.

In the conversion callback a new object should be created from the object pointed to by "o". Furthermore, this object needs to be linked into the scene using: "ay_object_link()". See "icurve.c/ay_icurve_convertcb()" for an example.

Conversion callbacks have to be registered in the initialization callback using:

```
int ay_convert_register(ay_convertcb *convcb, unsigned int type_id);
```

3.16 Provide

```
int yourname_providecb(ay_object *o, unsigned int type,
                      ay_object **result);
```

Provide callbacks are in use for objects that are able to provide objects from a different type. This is very much like the conversion (above) but the objects are not to be linked into the scene but used by e.g. a parent procedural object as parameter for its procedure. For instance, using the provide mechanism a Revolve object is able to revolve an interpolating curve. The argument "type" denotes the desired type of the new object and "result" should be filled with an object of the wanted type. See "icurve.c/ay_icurve_providecb()" for an example.

Provide callbacks have to be registered in the initialization callback using:

```
int ay_provide_register(ay_providecb *provcb, unsigned int type_id);
```

3.17 Tree-Drop

```
int yourname_dropcb(ay_object *o);
```

The tree-drop callback is for custom objects that want to get notified, when an object is dropped onto them in the tree view to invoke some special actions. This is e.g. used by the material objects that connect to all geometric objects dropped onto them, or by the view object which uses the camera settings from a camera object which is dropped onto the view object.

Tree-drop callbacks have to be registered in the initialization callback using:

```
int ay_tree_registerdrop(ay_treedropcb *cb, unsigned int type_id)
```

4 Custom Objects from the Tcl Side of Life

Yes, custom objects need some Tcl code too.

Take a look at the file "csphere.tcl".

This code, first, fills two important variables: "CSphere_props" and "CSphereAttr".

```
global ay CSphere_props CSphereAttr CSphereAttrData

set CSphere_props { Transformations Attributes Material Tags CSphereAttr }

array set CSphereAttr {
arr   CSphereAttrData
sproc ""
gproc ""
w     fCSphereAttr
}
```

"CSphere_props" holds a list of all properties of the CSphere. There are well known standard properties "Transformations Attributes Material Tags" and a new special property "CSphereAttr".

The handling of the standard properties is done entirely by the core. No need for further adjustments.

The new special property "CSphereAttr", however, must be introduced to the core properly now. This is done by filling a global array named as the property, in our case the property, and the array, are named "CSphereAttr".

The single elements denote:

1. arr: name of a global array, where this property holds its data
2. sproc: name of a procedure that transports the data from Tcl context to C context. If this is empty (sproc "") the core will use the internal mechanism and call the callback provided on registration of your object type. This is the point where you may jump into own property functionality, if you need to provide arguments to your own functions or do some other magic (call other commands etc.pp.).
3. gproc: name of a procedure that transports the data from C context to Tcl context. See discussion above.

4. `w`: name of the window of the property GUI

Now the property GUI itself needs to be created:

```
array set CSphereAttrData {
  Closed 1
  Radius 1.0
  ZMin -1.0
  ZMax 1.0
  ThetaMax 1.0
}

# create CSphereAttr-UI
set w [frame $ay(pca).$CSphereAttr(w)]

addCheck $w CSphereAttrData Closed
addParam $w CSphereAttrData Radius
addParam $w CSphereAttrData ZMin
addParam $w CSphereAttrData ZMax
addParam $w CSphereAttrData ThetaMax
```

The code is really simple and creates a static GUI consisting of a checkbox and four entries for parameters. The different available GUI elements are discussed in the next sections.

Finally, we create an entry in the main menu, for easy creation of our new object type:

```
# add menu entry to the Create/Custom Object sub-menu
mmenu_addcustom CSphere "crtOb CSphere; uS; sL; rV"

# tell the rest of Ayam (or other custom objects), that we are loaded
lappend ay(co) CSphere
```

4.1 Property GUI Elements

A property GUI is usually organized in list form. The single list elements are mostly built by calling a single Tcl command of the Ayam core. All elements are implemented in the file `"uie.tcl"` (UIE - User Interface Elements).

The template for such a command (that creates a single line of a property GUI) is as follows:

```
add{type} w array name [default]
```

`type` is the type of the entry. Predefined and heavily used by the rest of Ayam are: Text, Param, Menu, Check, Color, File, String and Command. Depending on the type, the actual parameters are a bit different but this is documented in detail later on...

`w` is the window the new entry should be created in. Just pass the name of the window of the property GUI.

`array` is the name of the (global) array where the parameters that should be edited are actually stored.

`name` is the name of the variable in the global array `array` that this entry should manipulate. In addition, this name is often used in a label to mark the entry. You should use descriptive and not too long variable names.

default is a list of default values. However, not all GUI elements support those!

You are, of course, not tied to the aforementioned entries. The seashell custom object, for instance, implements an own type, an entry consisting of a slider, that is even able to issue apply operations while dragging the slider.

The following sections document all core entry types.

4.1.1 Labels

```
addText w f text
```

adds a line containing the text `text` to the property GUI. `f` is required to generate window names, use `e1`, `e2` etc.; `f` must be unique over all entries of a property GUI.

Example:

```
set w [frame $ay(pca).$CSphereAttr(w)]
addText $w CSphereAttrData e1 "CSphere Attributes"
```

4.1.2 Parameters

```
addParam w array name [default]
```

creates the standard parameter manipulation entry consisting of a label, two buttons for quick parameter manipulation by doubling, dividing the value, and finally an entry for direct manipulation.

Example:

```
set w [frame $ay(pca).$CSphereAttr(w)]
addParam $w CSphereAttrData Radius
```

4.1.3 Menus

```
addMenu w array name list
```

adds a menu button, that toggles between the elements of the list `list`. The variable `name` always contains the index of the selected entry. Note, that the variable name has to exist before the call to `addMenu`!

Example:

```
set CSphereAttrData(Type) 0
set w [frame $ay(pca).$CSphereAttr(w)]
addMenu $w CSphereAttrData Type [list Simple Enhanced]
```

4.1.4 Checkbuttons

```
addCheck w array name
```

adds a single check button to the GUI. The variable name will be set to either 0 or 1 according to the state of the check button.

```
set w [frame $ay(pca).$CSphereAttr(w)]
addCheck $w CSphereAttrData Closed
```

4.1.5 Color Selectors

```
addColor w array name [default]
```

adds a color selection facility. The color values will be written to variables named {name}_R, {name}_G, and {name}_B.

Example:

```
set w [frame $ay(pca).$CSphereAttr(w)]
addColor $w CSphereAttrData SphereCol
```

4.1.6 File Selectors

```
addFile w array name
```

adds a string entry and a small button, that starts the standard file requester, handy for strings that contain file names.

Example:

```
set w [frame $ay(pca).$CSphereAttr(w)]
addFile $w CSphereAttrData FName
```

4.1.7 String Entries

```
addString w array name [default]
```

adds a simple string entry.

Example:

```
set w [frame $ay(pca).$CSphereAttr(w)]
addString $w CSphereAttrData Name
```

4.1.8 Buttons

```
addCommand w f text command
```

adds a big button labelled with `text` that starts the command `command`. `f` is (similar to text entries) a name for a window, I suggest to name the windows `c1`, `c2` etc.; `f` must be unique over all entries of a property GUI.

Example:

```
set w [frame $ay(pca).$CSphereAttr(w)]
addCommand $w c1 "PressMe!" "puts Hi"
```

4.2 The Custom Menu

You may link additional functionality of your custom object to entries in the custom menu.

The following example code snippet shows how to do that:

```
# link proc fooproc to Custom menu
# we need access to global array "ay"
global ay
# always create a cascaded sub-menu
$ay(cm) add cascade -menu $ay(cm).foo -label "Foo"
# create menu
set m [menu $ay(cm).foo]
# create menu entry
$m add command -label "Foo(l)" -command fooproc
```

Note, that you should always create a new sub-menu using a cascade entry instead of creating entries directly in the custom menu.

5 Compiling, Installing, Loading a Custom Object

The source code of an Ayam custom object needs to include `ayam.h`, `ayam.h` in turn includes `Togl` (OpenGL, Tcl/Tk, X11) and `RI` (RenderMan Interface) headers. When compiling your custom object, all you need to do is to make sure the compiler finds all those includes. See the Makefile of Ayam on how to collect all necessary information to build `-I` directives. The target `csphere.so`: should provide enough information on how to compile a custom object.

Compile your source with the `-c` switch. Then use the right switch for your compiler (`-shared` ?) to make a shared object (`.so`) from the `.o(s)` and you should be ready to test (`>cc -shared -o foo.so foo.o`).

Install the shared object along with the Tcl file containing the property GUI procedure and other stuff.

To load a custom object use the appropriate menu entry in the File menu or the `io_lc` (load custom) command. Both methods will automatically change the working directory to the location of the shared object to allow it to find the accompanying Tcl script more easily. Do not use the `load` command of the Tcl core directly!

If you want your custom object to be loaded automatically on startup of Ayam just create a small Tcl-script and load that on startup using the preference setting `Prefs/Main/Scripts`. Example:

```
# this script loads the mfio-plugin into Ayam
io_lc /home/randi/ays/plugins/mfio.so
return
```
