BOOGA Scene Description Language

Reference Manual

Booga and BSDL Version 1.67

Andrey Collison

Draft 0.11

April 1997

Contents

1 Introduction **1**
1.1 What is BOOGA? 1
1.2 What is BSDL? 1
1.3 What is this Document About? 1

2 BOOGA Scene Description Language **2**
2.1 Language Concepts 2
2.2 Language Definition 3
2.3 Example 4

3 BSDL2 Language Extension **6**

4 BSDL3 Language Extension **6**
4.1 Primitive Objects 6
4.2 Aggregate Objects 8
4.3 Specifying a View 8
4.4 Transformations 10
4.5 Textures 11
4.6 User Defined Objects 11
4.7 Animated Objects 12
4.7.1 Modelling Functions 15
4.7.2 Animation Example 16

1 Introduction

1.1 What is BOOGA?

BOOGA (**B**erne's **O**bject-**O**riented **G**raphics **A**rchitecture) is an object-oriented graphics framework aimed at a wide range of application areas within the domain of computer graphics, namely *Geometric Modelling, Image Synthesis, Image Processing, Image Analysis, Scene Recognition, and Computational Geometry.* It provides data abstractions and mechanisms for 2D and 3D objects and includes a component layer to model high-level operations that generate or manipulate graphical objects of different types. The key concepts underlying the BOOGA-framework are explained in [AmStBi96]. A more detailed description (in german) is given in [Streit97].
The framework serves as a research platform for computer graphics at the University of Berne. It currently consists of more than 200'000 lines of C++ code distributed over more than 600 classes.
A wide variaty of applications have allready been built using the BOOGA-framework e.g. scene-previewers, object-browsers, 3D-editors, ray-tracer, wireframe-renderer, sirds-renderer, graphics database and more. Many of the applications might also be interesting to a technically oriented computer graphics user.

1.2 What is BSDL?

BSDL (BOOGA **S**cene **D**escription **L**anguage) is the language supported by the BOOGA framework. It is used to define 2D or 3D scenes. Most of the BOOGA-applications are capable of reading or writing BSDL files.

1.3 What is this Document About?

This document gives a short description of BSDL. It enables the technical-minded person with good knowledge in computer graphics to define his own scenes or to write generators that produce BSDL scenes.
An overview of the BSDL language concepts is given in Section 2. The 2D and 3D language extensions (BSDL2 and BSDL3) are described in Sections 3 and 4.

2 BOOGA Scene Description Language

2.1 Language Concepts

The BOOGA framework supports its own scene definition language the so called **B**OOGA **S**cene **D**escription **L**anguage (From now on the shortcut BSDL will be used in this document). This language may be used to either define 2D or 3D scenes.
BSDL's goals are to provide a simple, dynamically extensible language. In fact BSDL-kernel supports 4 keywords only, but may be easily extended at run-time e.g. to support various primitive objects, textures, transformations and view specifications. This section gives a short overview of the BDSL-kernel and the underlying concepts. BSDL extension packages for 2D and 3D computer graphics will be presented in Section 3 and Section 4.
Let's have a look at some of BSDL's features:

- *few keywords*

  In contrast to other languages in the graphics domain (as e.g. RAYSHADE) BSDL supports very few keywords, namely: `define`, `const`, `using`, and `namespace`. Therefore the language has a very simple structure. We will have closer look at the language structure later in this section.

- *dynamic configuration*

  The BSDL parser provided by the BOOGA framework may be configured dynamically i.e. at run-time. Only after the appropriate configuration the parser is able to cope with the definition of e.g. a sphere. Before configuration the corresponding keyword `sphere` is not understood by the BSDL parser.

  This concept allowes for an arbitrary extensible language (a fundamental requirement for an extensible framework).

- *mathematical expressions*

  BSDL supports simple arithmetic expressions consisting of the four basic operations +, -, *, /. However arbitrary mathematical functions may be added to the language at run-time.

This is how e.g. all of the trigonometric functions are provided by the BOOGA framework.

- *polymorphic variables*

  Simple numeric values, vectors (2D and 3D), matrices (2D and 3D), and character strings may be used in expressions or as parameters to object definitions. The different data types may be freely combined. Automatic conversions are initiated by the system if appropriate.

2.2 Language Definition

Using EBNF-Notation the BSDL language may be desribed as follows:

A world consists of an aribitrary number of definitions and objects.

```
World ::= { <Definition> | <Object> }+
```

A definiton produces a new object, or namespace, or constant and assigns it to an identifier.

```
Definition ::= define IDENTIFIER <Specifier>
          |   define IDENTIFIER namespace ‘;’
          |   const  IDENTIFIER <Value>   ‘;’
```

The following rule specifies object definitions. An object definition starts with a keyword followed by an arbitrary number of parameters and an aribitrary number of object definitions (recursivly).

```
Specifier     ::= IDENTIFIER <ValueList> <OptSpecifiers>
              |   using IDENTIFIER ‘;’
OptSpecifiers ::= ‘;’
              |   ‘{’ { <Specifier> }+ ‘}’ [ ‘;’ ]
ValueList     ::= [ <Value> ]
              |   ‘(’ <Values> ‘)’
```

The `using` *directive activates a userdefined namespace (For further details have a look at the example in the following section).*
Object definition parameters or arguments of mathematical expressions are composed of numerical values, vectors, matrices or character strings (Character strings start and end at double quotes).

```
Values ::= { <Value> ‘,’ } <Value>
Value  ::= NUMBER
     |   VECTOR2D | VECTOR3D
     |   MATRIX2D | MATRIX3D
     |   STRING   | IDENTIFIER
     |   <Expression>
```

Mathematical expressions may be composed according to the following rule.

```
Expression  ::= ‘(’ <Value> ‘)’
          |   <Value> ‘+’ <Value>
          |   <Value> ‘-’ <Value>
          |   <Value> ‘*’ <Value>
          |   <Value> ’/’ <Value>              |   ‘-’ <Value>
          |   IDENTIFIER ‘(’ <Values> ‘)’
```

Of course additional functions that have been dynamically integrated into the parser may also be used.

2.3 Example

In this section we look at a simple example scene that uses all of the basic keywords provided by BSDL. The scene contains two spheres and Figure 1 shows the scene displayed by a raytracer-application.

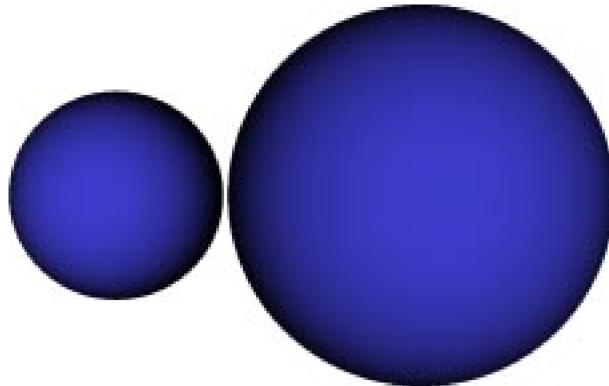

Figure 1: A simple scene visualized by a raytracer.

```
using 3D; // we want to use the 3D namespace

camera {  // a perspective camera
  perspective {
    eye    [1, 1.5, 11];  // position,
    lookat [1, 1.5, 0 ];  // position of interest (focus point)
    up     [0, 1,   0 ];  // camera orientation
  }
}

// Define an own namespace.
define MyNS namespace;

// Define a texture
define MyNS::sphereTexture phong {
  ambient [.1, .1, .3];
  diffuse [.3, .3, 1.];
}

// define the sphere's radius as a constant
const RADIUS 1.4;

// define a point light source with intensity 1 emmitting white light
pointLight (1, [1, 1, 1]) {
  position [0, 0, 100];
}

// first sphere
sphere (RADIUS, [-1.5, 1.5, 0]) {
  MyNS::sphereTexture; // use the previously defined texture.
}

// second sphere with transformaton
sphere (RADIUS*1.8, [-1.5, 1.5, 0]) {
  translate [4, 0, 0];
  MyNS::sphereTexture;
}
```

3 BSDL2 Language Extension

Sorry, this section hasn't been written yet.

4 BSDL3 Language Extension

This section covers the 3D language extension of BSDL (BSDL3). The BSDL3 extension is defined in the 3D namespace. This namespace is supported by all of BOOGA's 3D applications.
Before we can use the 3D extension we have to make the 3D namespace visible by placing the `using 3D;` directive e.g. at the beginning of our BSDL3 file.

```
using 3D; // from now on we want to use BOOGA's 3D extension
//
// A 3D scene definition ...
//
```

Note that in this section of the document a vector like $\vec{v}$ will stand for a vector in three dimensional space. It will have to be replaced by something like `[1, 0, 1.2]` in order to produce correct BSDL3 syntax. The brackets [] will enclose optional language elements. Words given in *itallic* writing stand for a class of language elements and have to be replaced by concrete members of the specific class.

4.1 Primitive Objects

`box` $(\vec{c_1}, \vec{c_2})$ { *[textures] [transforms]* }
: Creates an axis-aligned box which has $\vec{c_1}$ and $\vec{c_2}$ as opposite corners.

`cone` $(r_1, \vec{c_1}, r_2, \vec{c_2})$ { `[open | closed];` *[textures] [transforms]* }
: Creates a cone starting at $\vec{c_1}$ ending at $\vec{c_2}$ with radius r_1 at $\vec{c_1}$ and r_2 at $\vec{c_2}$. The cone may either be `open` or `closed` according to the keyword given in the attribute section.

`cylinder` $(r, \vec{c_1}, \vec{c_2})$ { `[open | closed];` *[textures] [transforms]* }
: Creates a cylinder starting from $\vec{c_1}$ to $\vec{c_2}$ with radius r The cylinder may be either `open` or `closed` according to the keyword given in the attribute section.

`disc` $(r, \vec{c}, \vec{n})$ { *[textures] [transforms]* }
Creates a disc with radius r centered at $\vec{c}$ lying in a plane perpendicular to $\vec{n}$.

`line` $(\vec{s}, \vec{e})$ { *[textures] [transforms]* }
Creates a line from $\vec{s}$ to $\vec{e}$. Please note that lines will only be visible in wireframe renderers. The reason for this is that they are infinitly thin and therefore ignored by most of the renderers.

`polygon` $(\vec{v_1}, \vec{v_2}, ...\vec{v_i})$ { `[hole` $(\vec{h_1}, \vec{h_2}, ...\vec{h_j})$;] *[textures] [transforms]* }
Creates a closed polygon with vertices $\vec{v_1}$ to $\vec{v_i}$. At least three non-collinear vertices have to be provided. It is also possible to define a hole in the polygon (i.e. a smaller polygon that will be cut out of the bigger polygon). This may be done by adding the keyword `hole` to the attribute section, followed by the vertices $\vec{v_1}$ to $\vec{v_i}$ that define the cut-out polygon.

`polyextrude` $(d, \vec{v_1}, \vec{v_2}, ...\vec{v_i})$ { `hole` $(\vec{h_1}, \vec{h_2}, ...\vec{h_j})$; *[textures] [transforms]* }
Creates an extruded polygon with vertices $\vec{v_1}$ to $\vec{v_i}$ and height d. Analog to `polygon` extruded polygons may also have an optional hole defined by the vertices $\vec{h_1}$ to $\vec{h_j}$.

`sphere` $(r, \vec{c})$ { *[textures] [transforms]* }
Creates a sphere with radius r and center $\vec{c}$.

`text` *(fontsize, strokeThickness, text)* { `font` *fonttype*; *[textures] [transforms]* }
Creates *text* of size *fontsize* using *strokeThickness*. The font is speciefied in the attribute section. Currently `font "ROMAN"` is the only fonttype supported.

```
text (.3,.05, "Don't run away we are friends") {
  font "ROMAN";
}
```

`torus` $(R, r, \vec{c}, \vec{n})$ { *[textures] [transforms]* }
Creates a torus with 'big radius' R and 'smaller radius' r. The torus is centered at $\vec{c}$ and lies perpendicular to $\vec{n}$.

`triangle` $(\vec{t_1}, \vec{t_2}, \vec{t_3})$ { *[textures] [transforms]* }
Creates a triangle with edges $\vec{t_1}, \vec{t_2}$, and $\vec{t_3}$.

4.2 Aggregate Objects

Aggregate objects provide the possibility to compose objects out of subobjects.

`list` { *objectList [textures] [transforms]* }
Creates an aggregate object which is composed by the objects given in *objectList*.

`grid` (x, y, z) { *objectList [textures] [transforms]* }
Creates an aggregate object which is composed by the objects given in *objectList*. The space occupied by the aggregate object is divided into a $x \times y \times z$ grid of voxels (volume elements). At run time the application will assign each voxel a list of *objects* which share space with the voxel. Reasonable use of this structure will result in a significant speed up of some rendering algorithms (typically ray tracing). Thumb rules for grid usage are:

- Only use grids when an object is composed out of many (e.g. more than 30) subobjects.
- Make sure the number of voxels $x \times y \times z$ does not exceed the number of objects.

4.3 Specifying a View

When designing a BSDL3 scene, there are two primary issues that must be confronted. The first is the selection of the objects to be rendered and the appearences they should be assigned. The second issue is the choice of viewing and lighting parameters. This section deals with the latter problem. The current release of BOOGA supports two different camera models: `perspective` and `orthographic` projection. Both camera models may be parametrized by a range of *viewingAttributes*.

`camera` **{** `perspective | orthographic` { *viewingAttributes* } `[background` $\vec{b_c}$`;]` `[on`

Creates a camera using `perspective` or `orthographic` projection. The *viewingAttributes* will be explained below. A background color $\vec{b_c}$ may be defined optionally.

The following list briefly explains the *viewingAttributes*:

`eye` $\vec{e}$
: Defines the location $\vec{e}$ of the eye or camera observing the scene.

`lookat` $\vec{l}$
: Defines the location of interest l (where the eye or camera is looking at).

`up` $\vec{u}$
: Defines the up-vector $\vec{u}$ of the camera.

`resolution` (x, y)
: Specifies the $x \times y$ resolution of the image. This is an advisary value and may be overriden by the application used.

`eyesep` s
: For stereo images this defines the distance sepperating both eyes.

`fov` ω
: For perspective views this defines the field of view of the camera. Use small values for the angle ω to simulate a zoomed camera and large values (> 60) for wide angle cameras.

When designing a scene one should not forget to add some light sources.

`pointLight` $(i, \vec{e_c})$ `{ [position` $\vec{p}$`;] [on; | off;]`
: Creates a point light source emitting color $\vec{e_c}$ at an intensity i. One may specify the position of the light source at $\vec{p}$. Default value is (0,0,0). It is also possible to turn on or off the light source.

An example of a viewing and lighting specification:

```
camera {
  perspective {
    eye [0,0,6];                // Eye position
    lookat [0,0,0];             // Looking to this point
    up [0,0,1];                 // What direction is up? (z)
    fov 60;                     // Field of view
    resolution (600, 600);      // Screen resolution
    eyesep 1;                   // For stereo images ...
  }
  background [.5,.5,.5];        // Color of Background
}

// We need a light
pointLight (1, [1,1,1]) {    // Point light with intensity 1
                             // and color [1,1,1] (white)
  position [0,0,100];
}
```

4.4 Transformations

Transformations are needed to place objects in space or to deform objects. Additionally transformations may also be used to alter texture space thus modifying the appearance (not the geometry) of an object (see also Section 4.5). BSDL supports the following transformations.

`rotateX` ω;
: Rotate ω degrees around X-axis.

`rotateY` ω;
: Rotate ω degrees around Y-axis.

`rotateZ` ω;
: Rotate ω degrees around Z-axis.

`scale` $\vec{s}$;
: Scale object in x,y,z-direction by the component values supplied in $\vec{s}$.

`translate` $\vec{t}$;
: Translate object by $\vec{t}$.

`transform` $(\vec{r_1}, \vec{r_2}, \vec{r_3}, \vec{t})$`;`

Transform object.

4.5 Textures

`matte` { `[diffuse` $\vec{d_c}$`;]` }

Creates a simple matte texture.

`phong` { `[ambient` $\vec{a_c}$`;]` `[diffuse` $\vec{d_c}$`;]` `[specular` $\vec{s_c}$`;]` `[specpow` n`;]` }

Creates a texture that is shaded by the phong illumination model. The diffuse and ambient color are controlled by $\vec{d_c}$ and $\vec{a_c}$. The color of the 'hot spots' is controlled by $\vec{s_c}$ whereas the size of the 'hot spots' is controlled by n. Bigger values for n will result in smaller 'hot spots' which will give the objects surface a more reflective appearence.

```
phong {
  ambient [.2,.1,.2];
  diffuse [.9,.2,.9];
  specular [.4,.2,.4];
  specpow 20;
}
```

`whitted` {

`[ambient` $\vec{a_c}$`;]` `[diffuse` $\vec{d_c}$`;]` `[reflectivity` r`;]` `[transparency` t`;]` `[refractionIndex` n`;]` }

Creates a texture that is shaded using the whitted illumination model.

`checker` { `even` { *texture* }; `odd` { *texture* } }

Creates a checkered texture when different textures are given for even and odd checker fields.

4.6 User Defined Objects

In BSDL it is possible to create userdefined objects and textures and assign them a name. After having defined such an object an arbitrary number of instances may be created.

Please note that a BOOGA application stores only one definition of a user defined object in memory even if more than one instance of the object has been created. Making use of user defined objects in complex scenes may dramatically reduce memory consumption of BOOGA applications.

`define` *name definition*

Creates an object or texture according to *definition*. From now on the object or texture may be addressed by *name*.

```
// a blueish texture
define blueTexture matte { diffuse [.2,.2,.7]; }

// define an object
define weights list {
  cylinder (.05, [-.5,0,0], [.5,0,0]) { blueTexture; }
  sphere (.2, [-0.5, 0, 0]) { blueTexture; }
  sphere (.2, [0.5, 0, 0]) { blueTexture; }
}

weights; // add first instance of 'weights' to the scene
weights { translate [0, 1, 0]; } // add a second instance of 'weights'
```

4.7 Animated Objects

BSDL3 supports animated objects of different types. Animation objects have to be defined according to the following generic rule:

animationType { *actionAttributes* `[on; | off;]` *anObject* }

Creates an animation object that will animate *anObject* by a specific *animationType*. Supported animation types are: `grow, move, shear, tumble, turn`. The body of an animation object consists of a list of *actionAttributes*. One may activate or deactivate the animation object with the `on` or `off` keywords.

`action` $(f_{start}, f_{end}, [n], [w])$ { *animationAttributes* }

Action attributes start with the `action` keyword followed by animation control parameters. The action (i.e. the animation) begins at frame

f_{start} and ends at frame f_{end}. The animation may be repeated n times. A value of $n = 0$ results in infinit repetitions of the sequence. A pause of w frames between sequences may be specified optionally. Please note that real numbers may be given for the values f_{start}, f_{end}, and w.

animationAttributes

Animation attributes depend on the animation type choosen and are described in the corresponding animation type descriptions.

The following list gives an overview of the different animation types and their corresponding animation attributes.

`cycle` { `action(...)` { ... } *Objects* }

`cycling` (*"myFunction", start, end, step*)`;`

`center` ($\vec{c}$)`;`

Cycles between the objects given in the cycle block. Thus changing the appearance of the 'cycle' object. Note that this is the only animation object with multiple object definitions in the animation block.

`grow` { `action(...)` { ... } *anObject* }

`scalefactor` ($\vec{s}$, *"myFunction", start, end, step*)`;`

`center` ($\vec{c}$)`;`

Creates growing or shrinking objects. The object will be scaled by factor $\vec{s}$ using the modelling function specified by *"myFunction"*.

```
grow {
  action (2, 7) {
    scalefactor ([1, 2, 3], "id", 0.1, 0.8, 1/4);
    center ([13, 19, 17]);
  }
  ball;
}
```

`move` { `action(...)` { ... } *anObject* }

`direction` ($\vec{d}$, *"myFunction", start, end, step*);

Creates an object moving along a straight line. $\vec{d}$ specifies direction and distance of the movement.

```
move {
  action (5, 9.1) {
    direction ([1, 2, 3], "step", 0, 0.8, 1/7);
  }
  action (2, 7) {
    direction ([1, 1, -37], "saw", 0.3, 1, 1/7);
  }
  ball;
}
```

`shear` { `action(...)` { ... } *anObject* }

`shearfactor` ($\vec{s_2}$, *"myFunction", start, end, step*);

`axis` ($\vec{a}$);

`center` ($\vec{c}$);

Creates an object shearing perpendicular to $\vec{a}$ by the amount given by $\vec{s_2}$ (2D vector) and the distance to $\vec{c}$.

```
shear {
  action (2, 7) {
    shearfactor ([1, 3], "step", 0, 1, 0);
    center ([-13, 19, -31]);
    axis([0, 0, 1]);
  }
  ball;
}
```

`tumble` { `action(...)` { ... } *anObject* }

`tumblepath` (*"myFunction", start , end, step*) { *nurbsCurve* }

`tumblecenter` { *nurbsCurve* }

`tumbledirection` { *nurbsCurve* }

Creates an object moving along a nurbs curve. The orientation of the object may be controlled by other nurbs curves. An example of a tumbling object:

```
tumble {
  action (2, 7) {
    tumblepath ([1, 2, 3], "id", 0, 1, 0) {...}
    tumblecenter {...}
    tumbledirection {...}
  }
  ball;
}
```

`turn` { `action(...)` { ... } *anObject* }

`alpha` (α, ,*"myFunction", start, end, step*);

`axis` ($\vec{a}$);

`center` ($\vec{c}$);

Creates an object rotating α degrees around the axxis running through $\vec{c}$ with direction $\vec{d}$.

Example of a rotating object.

```
turn {
  action (2, 7) {
    scalefactor (180, "smoothstep", 0, 1, 0);
    axis ([0, 1, 1]);
  }
  ball;
}
```

4.7.1 Modelling Functions

Some of the *animationAttributes* described above take a modelling function as a parameter. There are ten predefined modelling functions to choose from.

The modelling functions are defined in Table 1. The modelling function choosen has to be written as a string variable (i.e. between double quotes) and is followed by three numbers. The first two numbers specify the range of the function to be used for the animation. The third number specifies the stepwidth of the function. A value of zero will result in continuous values. Please note that the *sin* function used here is scaled to have 10 periods in the intervall $[0, 1]$.

4.7.2 Animation Example

The following example listing defines a swing object with a seat swinging back and forth.

Figure 2: A swing.

```
define ground list {
  // Define the ground
  polygon ([-45,-45,0],[45,-45,0],[45,45,0],[-45,45,0]) { green; }
}
```

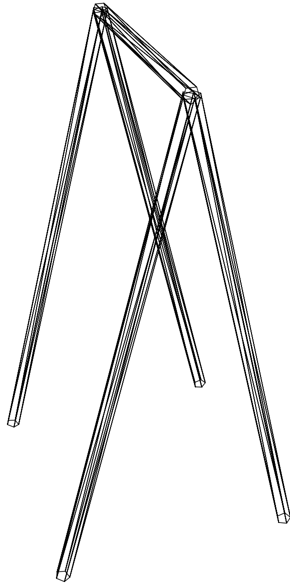

Figure 3: Stage of swing.

```
define stage list {
  cylinder (.3, [5,5,0],[5,0,20])   {grey;}  // Defines
  cylinder (.3, [5,-5,0],[5,0,20])  {grey;}  // the stage
  cylinder (.3, [-5,5,0],[-5,0,20]) {grey;}
  cylinder (.3, [-5,-5,0],[-5,0,20]){grey;}
  cylinder (.3, [-5,0,20],[5,0,20]) {grey;}
}

define seat list {
```

```
  cylinder (.1, [3,0,20],[3,0,5])   { grey;}  // Defines
  cylinder (.1, [-3,0,20],[-3,0,5]) { grey;}  // the seat
  box ([-3.5,-1,4.8],[3.5,1,5])     { brown;}
}
```

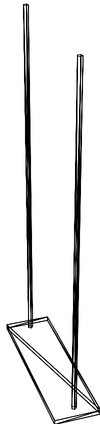

Figure 4: Seat of swing.

```
camera {                 // Defines the camera
  perspective {
    eye [30,-10,30];        // camera position
    lookat [0, 0, 10];      // point of interest
    resolution (512, 512);  // resolution
  }
  background [.3,.3,.3];
};
```

```
pointLight (2, [1,1,1]) { position [  500,   500, 500]; } // Die Lichtquelle

define green   matte { diffuse [.2,.5,.2]; }
define brown   matte { diffuse [.3,.2,.2]; }
define grey    matte { diffuse [.7,.7,.7]; }

define swing list {  // defines the swing
  stage;                    // stage
  turn {                    // start a rotating object
    action (0,12,0,12) {       // rotate from frame 0 to 12, wait 12 frames,
                                 // forever
      center ([0,0,20]);         // center of rotation (0,0,20)
                                 //
      axis ([1,0,0]);            // axis of rotation (1,0,0)
      alpha (60, "sin",0,0.05,0);// turn 60 degrees
                                 // modell animation by a sine function
                                 // on the intervall [0,0.05] (half period!)
                                 // step 0 i.e. continuous.
                                 // Back swing:
    }
    action (12,24,0,12) {      // rotate in frames 12 to 24, wait 12 frames,
                                 // forever
      center ([0,0,20]);
      axis ([1,0,0]);
      alpha (-60, "sin",0,0.05,0);
                                 // Forth swing
    }
    seat;                   // Seat (object beeing rotated)
  }
}
```

The following action block is equivalent to the two action statements in the example above.

```
    action (0,24,0) {  // rotate from frame 0 to 24, infinite repetitions
                       // only one action block for the entire swinging
      center ([0,0,20]);
      axis ([1,0,0]);
      alpha (60, "sin",0,0.1,0);
    }
```

References

[AmStBi96] S. Amann, Ch. Streit, and H. Bieri. BOOGA*: A Component-Oriented Framework for Computer Graphics.* Technical Report IAM-96-017, Institute of Computer Science and Applied Mathematics, University of Berne, 1996

[Streit97] Ch. Streit. BOOGA*: Ein Komponentenframework für Grafikanwendungen.* Inauguraldissertation der philosophisch naturwissenschaftlichen Fakultät der Universität Bern, 1997.

Definition of modelling functions		
Function	Identifier	Definition
constant function	`const`	$f(x) = start$
identity	`id`	$f(x) = x$
pulse function	`pulse`	$f(x) = \begin{cases} 0 & : \quad x < 0.25 - \epsilon \\ 0 & : \quad x \geq 0.75 - \epsilon \\ 1 & : \quad \text{sonst} \end{cases}$
square function	`quad`	$f(x) = (2(x - \frac{1}{2}))^2$
saw tooth	`saw`	$f(x) = \begin{cases} 0 & : \quad x \geq 1 - \epsilon \\ x & : \quad \text{sonst} \end{cases}$
smoothed step	`smoothstep`	$f(x) = x^2(3 - 2x)$
square root	`sqrt`	$f(x) = \sqrt{\lvert 2x - 1 \rvert}$
step function	`step`	$f(x) = \begin{cases} 0 & : \quad x < \frac{1}{2} - \epsilon \\ 1 & : \quad \text{sonst} \end{cases}$
triangle function	`triangle`	$f(x) = \begin{cases} x < \frac{1}{2} & : \quad 2x \\ x \geq \frac{1}{2} & : \quad 2 - 2x \end{cases}$
sine function	`sin`	$f(x) = sin(20\pi x)$

Table 1: Defintion of modelling functions.