Standalone Interactive Graphics (SIG) Toolkit

1 Introduction

SIG is a class toolkit for the development of interactive 3D graphics applications. SIG is designed to be small, flexible, fast, portable, and standalone. It is fully written in C++ but with moderate use of C++ most recent features. SIG does not use STL classes. While STL will most likely be used in your programs, SIG only relies on its internal classes for data structures, leading to more legible compiler error messages, consistent behavior across different compilers, significantly faster compilation, and improved execution times in some cases.

SIG includes a simple and extendible scene graph, a flexible skeleton structure supporting a variety of joint definitions, classes for loading and manipulating `.bvh` motion files and `.obj` geometry files, several utilities for developing motion planners, and an analytical Inverse Kinematics solver [1]. It also includes functionality to manage resources such as textures and fonts. SIG is completely standalone, exposing the entire process of using OpenGL, making it a great tool for research and for learning computer graphics. Previous versions of SIG existed under different names and have been used to support a variety of research projects.

Distribution terms: SIG is being released under the Apache License Version 2.0. Please read file license.txt available in the base folder of the distribution for details.

1.1 Code Structure

SIG is currently organized in 4 libraries: sig, sigogl, sigos, and sigkin. The source files in each library are divided in modules identified by the first 2 letters of each file. An overview of the modules is available in Table 1.

Lib	Module	Description	Examples
sig	gs	generic graphics and system classes	**`GsVec`**, **`GsMat`**, **`GsOuput`**, **`GsArray`**
sig	sn	scene graph nodes	**`SnModel`**, **`SnLines`**, **`SnTransform`**
sig	sa	scene graph actions	**`SaBBox`**, **`SaRenderMode`**
sig	cd	interface to external collision detectors	**`CdImplementation`**, **`CdManager`**
sigos	wsi	OS-specific window system interface	**`wsi_get_ogl_procedure()`**
sigogl	gl	OpenGL-related functions and classes	**`GlProgram`**, **`GlTexture`**
sigogl	glr	scene node renderers based on OpenGL	**`GlrMode`**, **`GlrLines`**, **`GlrText`**
sigogl	ui	OpenGL-rendered GUI classes	**`UiButton`**, **`UiStyle`**, **`UiManager`**
sigogl	ws	Window system classes	**`WsWindow`**, **`WsViewer`**
sigkin	kn	kinematics of articulated structures	**`KnJoint`**, **`KnSkeleton`**, **`KnMotion`**

Table 1: Libraries and modules in SIG.

SIG follows a simple yet consistent naming convention which is briefly described in **`gs.h`**. While it is a bit different than modern conventions, it has proven to be effective over the many years it has been used.

1.2 Compilation and Supported Systems

SIG is mostly being developed in MS Windows. Project files for Visual Studio Community 2017 are provided to compile all libraries and as well several examples.

Makefiles for Linux are included in folder **make** based on an interface to **glfw**, which should allow SIG to be used in any platform. While this integration is still incomplete, it already generates a usable test executable in Linux.

1.3 A First SIG Application

Starting a SIG application is as simple as declaring a viewer and a scene, and then checking events, for example with **ws_run()**. See example in Listing 1 and its result in Figure 1.

Listing 1: My first SIG application.

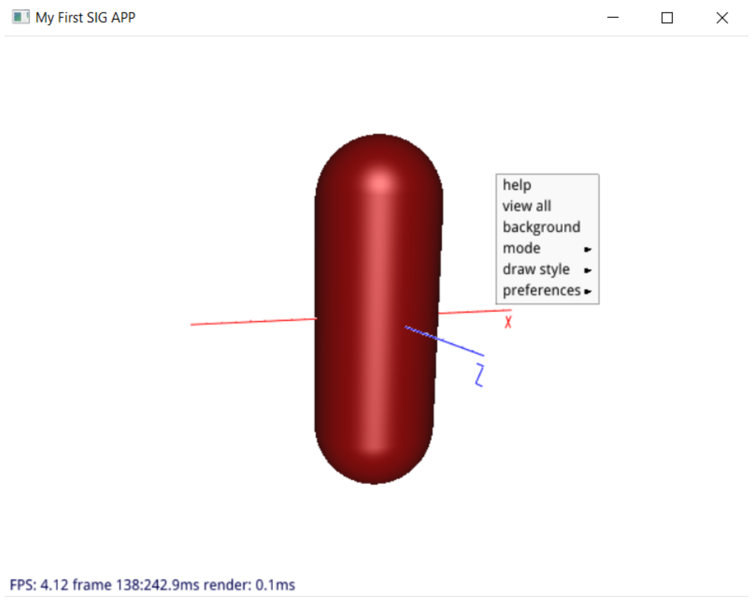
```
# include <sig/sn_primitive.h>
# include <sigogl/ws_viewer.h>
# include <sigogl/ws_run.h>

int main ( int argc, char** argv )
{
    WsViewer* v = new WsViewer ( -1, -1, 640, 480, "My␣First␣SIG␣APP" );

    SnPrimitive* p = new SnPrimitive ( GsPrimitive::Capsule, 5.0f, 5.0f, 9.0f );
    p->prim().nfaces = 100;
    p->prim().material.diffuse = GsColor::darkred;

    v->rootg()->add ( p );
    v->cmd ( WsViewer::VCmdAxis );
    v->cmd ( WsViewer::VCmdStatistics );
    v->view_all ();
    v->show ();

    ws_run ();
    return 1;
}
```



FPS: 4.12 frame 138:242.9ms render: 0.1ms

Figure 1: Result of the example in Listing 1.

In most cases however the user will derive **WsViewer** with its own viewer class in order to catch GUI events and extend additional functionality by overriding the virtual methods of **WsViewer**.

When working on your own projects you may use as starting point the project in **`sigapp.7z`**, which is included in the **`examples/`** folder of the distribution. This project demonstrates how to declare a simple scene graph and GUI, how to respond to events, and one possible way to control the main loop for a simple animation. For compilation the project should be placed in the same folder as **`sig/`**, because the libraries are linked with relative paths.

Multiple examples are also available in the SIG distribution:

- cameratest: example manipulations with the camera transformation;
- customnode: example of how to build your own scene node and node renderer;
- fontgen: example of how to test and define new fonts;
- gstests: examples and tests with the many classes in the gs module;
- modelviewer: example application to view and inspect models, and to load .obj files;
- polyeditor: example of how to use the polygon editor scene node;
- scenetest: example testing several scene graph features;
- shapes: example program where all shapes can be tested;
- skelviewer: example application to view skeletons and motions;
- uidemo: example application testing GUI elements;
- vgtest: example application testing the included visibility graph path planner.

These example applications demonstrate how the main SIG features should be used.

Some of the examples are in active development.

1.4 File Formats

A number of SIG classes have their own file formats for saving and loading data:

- **`GsModel`**: .m file format: description of geometry information.
- **`KnSkeleton`**: .s file format: main skeleton definition file.
- **`KnSkeleton`**: .sd data file: skeleton definitions automatically loaded to override or complement definitions in a .s file, when in same folder and with same name.
- **`KnPosture`**: .sp file format: describes joint values for a skeleton posture.
- **`KnMotion`**: .sm file format: motion file described by key frames.

A description of each file format is available in the Appendix of this document.

2 Overview of Main Classes

2.1 Arrays and other Data Structures

One of the most used classes is `GsArray`, which is an important class designed to efficiently manipulate memory blocks. The class works similarly to `std::vector` of the standard template library. However it has a main difference: it manipulates the internal data of the array as non-typed memory blocks, so the elements of the array are not treated as objects and the constructors and destructors of the elements in the array are never called when the array is manipulated. `GsArray` relies on low-level C functions for memory allocation, re-allocation, and deletion. Therefore `GsArray` is an efficient memory management class which outperforms `std::vector` in several operations.

The user needs however to pay attention to only use it with primitive types, or for pointers to objects with the use of `GsArrayPt`. See `gs_array.h` for details.

It is also important to observe that `GsArray` automatically doubles its internal memory when it needs more space in certain operations. So references to objects in the array will only be guaranteed to be valid while no new elements are added to the array. For example, a command sequence like `int& e=array[0]; array.push()=5;` should never be written because when 5 is pushed to the array memory re-allocation may happen, in which case reference `e` to element index 0 will become invalid.

2.2 Viewer and other Windows

To be completed.

3 SIG's Scene Graph

SIG implements a scene graph structure which allows to organize shapes and transformations in multiple ways.

All classes starting with `Sn` are *scene nodes*. Class `SnNode` is the base class for all scene nodes. An instance of it cannot be directly created because it contains a pure virtual method for processing traversal actions. Every scene node class that can be inserted in a scene graph has to derive `SnNode` and implement that virtual method. Class `SnNode` inherits `GsShared`, therefore all nodes of a scene graph can be shared as smart pointers controlled by reference counting, which means methods `ref()` and `unref()` are called to control ownership.

There are 4 scene node classes directly deriving `SnNode`, each providing a specific way of interpreting node functionality:

- `SnShape` is a base class for nodes that are responsible for drawing shapes. This class also has pure virtual methods to be customized by specific shapes, so it cannot be directly instanced and used. Instead, multiple classes deriving `SnShape` are available, each for representing specific types of shapes, for example: `SnModel` can load and represent generic triangle-based boundary representation objects, `SnLines` can represent line segments or polygonal lines, `SnLines2` is a specialized version for 2D lines, etc.

- `SnGroup` is a class that holds several scene nodes as children nodes. When a scene traversal is called for a `SnGroup`, the traversal will proceed to each children node, from the child index 0 onwards. Class `SnGroup` is the class that allows creating a hierarchy of scene nodes. A group can contain any other scene node, including additional groups, therefore allowing the creation of tree and graph hierarchies. Note that cycles in the hierarchy must not be created and there is no mechanism to prevent their creation. If cycles are created scene traversals will not terminate. A group also has an important role in organizing transformations. It can be set to isolate transformations applied to its children from the rest of the scene graph by calling method `separator(true)`, making the group behave as a *separator*, as further explained below.

- `SnTransform` is a node that stores a 4x4 3D homogeneous transformation matrix in line-major format. Every time a scene traversal is executed an identity matrix is initialized as the current transformation to be applied to all shapes encountered during the traversal. If a `SnTransform` is encountered during the traversal, its transformation is multiplied to the current one and will therefore affect all the following shapes encountered. However, transformations can affect members of a group differently: if a group is set to behave as a *separator*, then after traversing all nodes in the group, the current transformation is restored to the transformation that existed before starting to traverse the group members.

- `SnEditor` is a special type of base class for nodes which need to react to user events such as key presses and mouse movements. It has specific virtual methods to handle such events in order to allow interactivity. It serves as the base class for `SnPolyEditor` which is a class for editing 2D polygons, and `SnManipulator` which is a class that displays a bounding box around any object inserted in it with method `child()`, and by clicking on the sides of the box the user can manipulate the position of the object.

Classes starting with letters `Sa` are *scene actions*, which are classes performing a scene traversal in order to achieve some computation on the scene graph. For example, `SaBBox` computes the bounding box of the scene under a given scene node, `SaRenderMode` changes the rendering mode flag of all shapes under a given node of the hierarchy, and `GlRenderer` renders the scene from the given scene root node. Note that `GlRenderer` does not start with `Sa` because it makes calls to `OpenGL` and SIG is a standalone library. The renderer is implemented in the `OpenGL` module of SIG which is available in library `libsigogl`.

3.1 Examples

Listing 2 builds a scene graph where two primitive objects might move during the application and therefore a transformation is placed before each primitive, with both nodes under a separator group. This allows the transformations to affect only the object placed in each group. In the viewer class, member variables providing direct access to the transformation nodes are saved so that there is fast direct access to update the transformations as needed.

Listing 2: Scene with two primitives.

```
class MyViewer : public WsViewer
{  private :
    SnTransform* _t1;
    SnTransform* _t2;
    ...
   public :
    ...
    void build_scene ();
    ...
};

void MyViewer::build_scene ()
{
    // Create object 1:
    GsModel* obj1 = new GsModel; // Object 1 (this could be your own node)
    obj1->make_cylinder ( GsPnt(0,0,0), GsPnt(0,1,0), 0.2f, 0.2f, 25, true );

    // Put object 1 in scene graph:
    SnGroup *g1 = new SnGroup;
    g1->separator ( true );
    g1->add ( _t1=new SnTransform ); // _t1 is our class member variable
    g1->add ( new SnModel(obj1) );
    g1->top<SnModel>()->color ( GsColor::red );
    _t1->get().translation ( 0.5f, 0, 0 ); // set initial translation

    // Create object 2:
    GsModel* obj2 = new GsModel;
    obj2->make_capsule ( GsPnt(0,0,0), GsPnt(0,1,0), 0.2f, 0.2f, 25, true );

    // Put object 2 in scene graph:
    SnGroup *g2 = new SnGroup;
    g2->separator ( true );
    g2->add ( _t2=new SnTransform ); // _t2 is our other class member variable
    g2->add ( new SnModel(obj2) );
    g2->top<SnModel>()->color ( GsColor::blue );
    _t2->get().translation ( -0.5f, 0, 0 ); // set initial translation

    // Finally add nodes to the scene root:
    rootg()->add(g1);
    rootg()->add(g2);
}
```

Listing 3 builds a scene graph for a particle system displaying many spheres and in this case it makes sense to share the shape node describing the sphere. However we need to define one specific transformation and color per sphere. As in the example of listing 2 we can use transformations under separator groups, but in addition we will use node `SnMaterial` to assign a different color to the same shared sphere.

Listing 3: Scene displying multiple spheres with different colors but sharing a single sphere shape node.

```
void MyViewer::build_scene ()
{
    // Create the sphere to be shared:
    SnModel* sphere = new SnModel;
    sphere->model()->make_sphere ( GsPnt::null, _sphereradius, nfaces, true );

    // Nodes to be used:
    SnGroup* g;     // one group per particle will be created
    SnTransform* t; // first group node will be a translation
    SnMaterial* m;  // then we set the desired color (the shared sphere will come next)

    // Build scene graph:
    // (assumes positions stored in position_array of size psize)
    for ( int i=0; i<psize; i++ )
    {   rootg()->add ( g=new SnGroup );
        g->separator(true);           // set transformation to only affect nodes in group g
        g->add ( t=new SnTransform ); // sphere position will be set here
        g->add ( m=new SnMaterial );  // color will be set here
        g->add ( sphere );            // now add shared sphere geometry
        t->get().translation ( position_array[i] ); // set position
        m->material().diffuse = GsColor::random();  // set color
    }
}
```

4 Skeletons and Motions

The Kinematics module of SIG includes several tools for working with skeletons and motions. Classes starting with `Kn` compose the Kinematics module, which generates an independent library `libsigkin` to be linked with your application.

The main classes in the Kinematics module are listed below::

- `KnSkeleton` is the main class for working with hierarchical objects. It maintains an array of joints, each joint being represented with `KnJoint` which handles different joint parameterizations with range limits and automatic conversions between parameterizations. Skeletons can be loaded from the popular `.bvh` format, or from SIG's `.s` skeleton format. Joints can be efficiently retrieved by name using a built-in hash table, can be listed using the array of joints, or traversed recursively using the parent-child relationship in the hierarchy tree. The header file `kn_skeleton.h` provides details of additional supported functionality such as connecting to collision detectors, storing pre-defined postures to be applied to the skeleton, attaching user data, and handling rigid body parts or skin deformations.

- `KnMotion` is the main class for working with motions defined as series of frames. It maintains an array of postures representing frames, each frame being a shared `KnPosture` which handles joint values to be sent to attached joints. Motions can be loaded from the popular `.bvh` format, or from SIG's `.m` motion definition format. After a motion is created or loaded, motions should be first connected to a skeleton or posture before applying frames to the connected object. Frames can be applied by frame index (using `apply_frame()`), or by keyframe interpolation (using `apply()`) where the desired keytime is given and used to interpolate the adjacent frames to the given keytime. Motions can therefore be defined for keyframe interpolation or for constant-rate motion capture representation. Motion connection is made by joint name, leading to a very flexible mechanism to map motions to different skeletons and postures for interpolation and blending operations.

- `KnPosture` is the class that handles values to be sent to skeleton joints or other postures. It needs to be first connected to another posture or to a skeleton, and then its values can be applied to the connected object. Postures also include a list of *channels* which specify joint names and degrees of freedom (DOFs) of each stored value. Channels provide the needed information for connecting to any given skeletons, and only matched channels are actually connected. Once connected a posture remains connected with direct pointers to the connect object for fast transfer of posture values. The header file `kn_motion.h` provides details of additional supported functionality such as maintaining key positions for a posture distance metric computation, generation of random values, and posture interpolation.

- `KnIk` implements a fast analytical IK solver for 7-DOF linkages. It requires specific parameterizations in the target linkage in order to support handling range limits and collision avoidance techniques, as described in [1]. The different classes starting with `KnIk` provide different ways for applying the IK solver to hierarchies and to connect it with scene graph manipulators.

- `KnCtScheduler` provides functionality to declare multiple motions, in the form of generic controllers, in a timeline including multiple activation times and smooth transition blending. It is a good example code of how several operations with motions and skeletons can be implemented.

4.1 Examples

Listing 4 presents an example of a .s file describing a simple arm structure.

Listing 4: Scene with two primitives.

```
skeleton
```

```
root joint1
{ visgeo "link.m"
  colgeo "link.m"
  modelrot 0 0 1 -90 \# to orient the link in x direction
  channel ZRot 0 lim -180 180

  joint joint2
   { offset 8 0 0
     visgeo "link.m"
     colgeo "link.m"
     modelrot 0 0 1 -90 \# to orient the link in x direction
     channel ZRot 0 lim -180 180

     joint wrist
      { offset 8 0 0
        channel ZRot 0 lim -180 180

        joint hand
         { visgeo "hand.m"
           colgeo "hand.m"
           modelrot 0 0 1 -90 \# to orient the hand in x direction
           channel ZRot 0 lim 0 0
         }
      }
   }
}
```

Listing 5 demonstrates some operations with joints transformations.

Listing 5: Scene with two primitives.

```
// Retrieve joints from names:
KnJoint* j1 = sk->joint("joint1");
KnJoint* j2 = sk->joint("joint2");
KnJoint* w = sk->joint("wrist");
KnJoint* h = sk->joint("hand");

// Apply local rotations around Z axis:
const int Z=2;
j1->euler()->value(Z,GS_TORAD(30));
j2->euler()->value(Z,GS_TORAD(30));

// Get local transformations in joint frame:
GsMat L1 = j1->lmat();
GsMat L2 = j2->lmat();
GsMat W = w->lmat();
GsMat H = h->lmat();

      // Display matrices for inspection:
gsout<<"Link␣1␣Local␣Transformation:\n"<<L1<<gsnl;
gsout<<"Link␣2␣Local␣Transformation:\n"<<L2<<gsnl;
gsout<<"Wrist␣Local␣Transformation:\n"<<W<<gsnl;
gsout<<"Hand␣Local␣Transformation:\n"<<H<<gsnl;

// Compute global matrix of hand joint:
GsMat Hfk = L1 * L2 * W * H;
gsout<<"Hand␣Global␣Frame␣by␣FK:␣\n"<<Hfk<<gsnl;

// We can also ask the global matrix to the skeleton:
sk->update_global_matrices();
GsMat Hg = h->gmat();
gsout<<"Hand␣Global␣Frame:␣\n"<<Hg<<gsnl;
gsout<<"Hand␣Center␣in␣Global␣Coordinates:␣\n"<< GsVec(Hg[3],Hg[7],Hg[11]) << gsnl;
```

5 Frequently Asked Questions

SIG was developed over several years according to the needs of a number of projects it has supported in different platforms. It provides an integrated framework that is efficient and very flexible.

5.1 Why not update SIG to use std templates?

Besides the restructuring that would be needed, too much code bloat would be included. For example, when just the header files of ostream, istream, and vector are included in gs.h the compilation time of SIG more than doubles. Also, it is difficult to customize these classes to a variety of needs. For example arrays adopting memory from other arrays, input with built-in parsing utilities, redirection to/from generic functions, etc. For these reasons, SIG will remain independent of std templates.

5.2 My project is not compiling, what could be wrong?

First of all, make sure the same Visual Studio configurations are being compiled for the libraries and for your program. For example, select configuration “Release” everywhere, or configuration “Debug” everywhere, or configuration “ReleaseDLL” everywhere. If you are starting from `sigapp`, also make sure that SIG and your project are under the same folder.

5.3 Will the included projects work if I use a different version of Visual Studio?

In most cases yes. It may help to edit the project files and change version numbers to match your version of Visual Studio, and many times you will need to retarget the projects to the Microsoft SDK version that you are trying to use.

5.4 Mouse events are not matching with the graphics output, what could be wrong?

Before anything else, make sure the drivers of your graphics card are up to date. SIG needs OpenGL 4 support and drivers fully up to date. Shaders have been set to require version 3.3.

References

[1] M. Kallmann. Analytical inverse kinematics with body posture control. *Computer Animation and Virtual Worlds*, 19(2):79–91, 2008.

A Description of SIG Data Files

The following simple notation is used to specify the format of text data files used in SIG.

- Single letters, such as *i* or *x* denote that a number is expected. The chosen letters should help identify the meaning of the numbers, for example, *i* denotes integers while *x* a real number coordinate. Indices may be used in the following way: *i1*, *i2*, and double letters such as *nm* meaning “number of materials” may also be used.
- Keywords are any names that are not single or double-letter strings.
- Parameters are any names appearing between `< >`, indicating that a name or number is expected as a parameter.
- Optional commands will appear inside brackets, such that the entire section inside the brackets is optional. For example, command `[name <name>]` indicates that specifying a name is optional.
- Delimiter “`|`” separates a list of keywords or numbers where only one element of the list is to be chosen as a parameter.

This simple notation is enough to specify the data files described in the next sections.

A.1 GsModel .m Model Definition File

(Status: needs revision with respect to the new GsModel internal format including grouped information.)

```
GsModel           # signature to identify the file

[name <name> ]    # if not given, name becomes an empty string

[culling <0|1>]   # if not defined, back-face culling is on by default

vertices <nv>     # list of vertices is mandatory, nv is the number of vertices
<x> <y> <z>
...

faces <nf>        # list of triangular faces is mandatory
<a> <b> <c>       # a triangle is defined with indices to the vertex list, starting from 0
...

[normals <nm>     # optional list of normals per vertex
<x> <y> <z>
...]

[fnormals <nf>    # optional list of normals per face
<a> <b> <c>
...]

[materials <nm>   # list of materials, each material is read by GsMaterial input operator
amb <r> <g> <b> <a> dif <r> <g> <b> <a> spe <r> <g> <b> <a> emi <r> <g> <b> <a> shi <v> [tid <i>]
...]

[fmaterials <mf>  # indices of materials to be assigned per face
<i1>
<i2>
...]

[mtlnames         # optional names for each material index
i name1
i name2
...]

[textcoords <nt>  # texture coordinates defined per vertex
<u> <v>
...]

[textures <nt>
<image.png>       # image file for each texture to be used
...]

[ftextcoords <nf> # texture coordinates as indices per face
<a> <b> <c>
...
<a> <b> <c>]

[primitive        # if the model represents a primitive, the primitive parameters go here
 <box|sphere|cylinder|capsule> <ra> [rb] [rc] <nfaces> #(nfaces needed even for a box)
 [center <x y z>]
 [orientation axis <x y z> ang <deg>]
 [material amb <r> <g> <b> <a> dif <r> <g> <b> <a> spe <r> <g> <b> <a> emi <r> <g> <b> <a> shi <v>]
 [color <r> <g> <b> <a>] # color will set the diffuse color of the default material
 [smooth|flat]
 ;
]
```

A.2 KnSkeleton .s Skeleton Definition File

```
KnSkeleton                    # Signature. It should be the first keyword of the file.

[ path|add_path <path> ]      # Loaded geometry files will be searched in the
                              # same directory of the .s file, or also searched
                              # in the directories specified with the path command.
                              # Several paths can be defined to be searched.

[ name|set_name <skelname> ] # Specifies the name of the skeleton


[ scale <scale factor> ]      # Command scale is used to scale the length of the
                              # skeleton links (offsets), for example for converting
                              # units. Translational limits are also scaled.

[ globalgeo <true|false> ]    # If true, the geometry is considered to be loaded in
                              # global coordinates and therefore all geometries are
                              # converted to local coordinates after loaded.
                              # By default globalgeo is considered false.

<SKELETON_DEFINITION>         # Usually the skeleton definition comes here. The
                              # definition syntax is specified later on in this file.

[ posture <name val1 val2 ... valN> ] # Command posture specifies a posture which
                              # values must match the active channels of the
                              # skeleton. All loaded postures will share the
                              # same channels. This command must come after the
                              # skeleton definition.

[ dist_func_joints <joint1 joint2 ... jointN;>] # This command specifies which
                              # joints are used in the distance function between
                              # postures. This command must come after all posture
                              # definitions.

[ collision_free_pairs  <jointname1 jointname2 ...>; ] # List of joint pairs to
                              # to be deactivated for collision detection.

[ userdata <var1=val; var2=val1 val2; ... varN=val;> ] # This command allows the
                              # specification of any kind of user-related data.
                              # The data is loaded as a GsVars object that is
                              # maintained by the skeleton

[ ik <LeftArm|RightArm|LeftLeg|RightLeg> <jointname> ] # initialize IK with given
                                                       # joint as end effector

[ end ]                       # Optional keywork that forces end of parsing


#### SKELETON_DEFINITION ####
# A skeleton can be defined in two ways: hierarchical or flat.
# The syntax of a hierarchical skeleton definition is:

skeleton                      # tells this is a hierarchical definition
root <name>                   # specifies the root joint and its name
{ <JOINTDEFS>                 # joint definitions go here
  joint <name>                # specifies child joint and name
   { <JOINTDEFS>              # etc
     joint <name>
      { ...
      }
   }
  ...
}
```

```
# The syntax of a flat non-hierarchical skeleton definition is:

root <name>                  # first specify the root joint
{ <JOINTDEFS>
}
joint <name> : <parent name> # then specify each joint with its name and its
{ <JOINTDEFS>                # parent's name, which must be previously defined
}
...                          # etc

# It is possible to modify the settings of a previously defined joint with
# the following syntax (the syntax of a .sd file):
joint <name>
{ (JOINTDEFS)
}

#### JOINTDEFS ####
# The possible joint definitions are listed below.
# Rotations specified in "axis <x y z> ang <a>" can also be written as "x y z a",
# where a is an angle in degrees.

  [offset|center <x y z> ]

  [euler XYZ|YXZ|YZX|ZY]

  [channel XPos|YPos|ZPos|XRot|YRot|ZRot <val> [free | <min><max> | lim <min><max>] ]

  [channel Quat [axis <x> <y> <z> ang <degrees>] [frozen] ]

  [channel Swing [axis <x> <y> ang <degrees>] [lim <xradius> <yradius>] ]

  [channel Twist <val> [free | <min><max> | lim <min><max>] ]

  [modelmat <4x4matrix as 16 floats>]          # apply to the joint model

  [modelrot <axis <x> <y> <z> ang <degrees>>]  # apply to the joint model

  [prerot <axis <x> <y> <z> ang <degrees>>]    # joint pre rotation

  [postrot <axis <x> <y> <z> ang <degrees>>]   # joint post rotation

  [align <pre|post|prepost|preinv> <x y z>]    # set pre/post for aligning given vector

  [visgeo <model filename>] # models are "added" if more than one visgeo is declared

  [colgeo <model filename>] # models are "added" if more than one colgeo is declared

  [visgeo|colgeo primitive <defs>] # see Note 2 below

  [visgeo|colgeo shared]    # reuse the other col/visgeo previously defined in the joint


#### Advanced Notes ####
- Note1: Quaternion rotations can now be also loaded with format: xzy <x> <y> <z>
- Note2: Geometries can now also be created with [visgeo|colgeo primitive <defs>;],
         where <defs> are the commands in the primitive description of the .m format
```

A.3 KnSkeleton .sd Skeleton Data Definition File

```
# The file extension adopted for this file is .sd
# Character '#' is used for comments

KnSkeleton     # Signature. It should be the first keyword of the file

skeldata       # This keyword must come right after the signature
               # for defining that this is not a new skeleton but
               # modifications to the current one.

               # At this point, any skeleton command understood by
               # a .s file can be given here and will overwrite
               # previous definitions.

joint <name>   # For joint (re)definitions, first specify the joint name,
{ (JOINTDEFS)  # then any joint definition commands can be used.
}

joint <name>   # Any number of joints can be modified
{ (JOINTDEFS)
}

...

[ end ]        # Optional end keywork forces end of parsing
```

A.4 KnPosture .sp Posture Definition File

```
# The file extension adopted for this file is .sp
# Character '#' is used for comments
# A Posture file has no signature

[name <name>]      # Give the Posture a name. Even if the keyword name is
                   # omitted the parser will load a string given here as the name.

channels [<N>]     # Specification of channels start. Parameter N is the number of channels,
                   # then each channel is defined with a joint name and channel type

<ch1jname> <XPos|YPos|ZPos|XRot|YRot|ZRot|Quat|Swing|Twist>
<ch2jname> <XPos|YPos|ZPos|XRot|YRot|ZRot|Quat|Swing|Twist>
...
<chNjname> <XPos|YPos|ZPos|XRot|YRot|ZRot|Quat|Swing|Twist>

[;]                # The number of channels N may be omitted, and only in that case,
                   # a ';' must be written at the end of the channel list definition

val1 val2 ... valK # the joint values of the Posture must match the channel description:
                   # - channels of types Xpos, YPos, ZPos, XRot, YRot, ZRot require one value each,
                   # - channel Quat is described by 3 values, which are the axis-angle description
                   #    of the quaternion rotation,
                   # - channel Swing requires 2 values, the 2D axis-angle of the swing rotation,
                   # - channel Twist requires 1 angle value.
                   # - All angles are to be specified in degrees.
```

A.5 KnMotion .sm Motion Definition File

```
#
# .sm skeleton motion file Description
#

KnMotion                       # signature

[ name <namestring> ]          # specifies a name for the motion (optional but recommended)

channels <N>                   # channels are defined in the same way as for posture definitions
<ch1jname> <ch1type>           # see .sp file documentation for a detailed description
<ch2jname> <ch2type>
...
<chNjname> <chNtype>

[ startkt|start_kt <keytime> ] # optional command to adjust keytimes to start with given value

[ frames <numframes> ]         # optionally gives how many frames are in this motion
                               # if not specified, will parse until the end of the file

kt <keytime> fr <POSTVALUES>   # frame data, where POSTVALUES is written according to the .sp format,
kt <keytime> fr <POSTVALUES>   # which follows the channels description
...
```