SIG Toolkit Documentation

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. It is truly standalone, it contains its own functions to create windows, load OpenGL functions and GLSL shaders, and it also contains a basic graphical user interface (GUI) which operates directly on OpenGL. SIG does not not use STL classes. While today STL is mature and can be used in your projects, all SIG data structures are defined in SIG, making it highly portable, guaranteeing the same behavior in any platform, significantly faster to compile, and often faster in execution time.

Previous versions of SIG have had different names and have been used for over a decade to support a number of different research projects from supporting the development of motion planning algorithms [1, 3] to simulating autonomous virtual characters and controlling NASA's Robonaut.

The main features are a small, simple and extendible scene graph, a flexible skeleton structure supporting a variety of joint definitions, classes for loading bvh motion capture files and including a number of blending operations between motions, postures, and skeletons, utilities for motion planners, and an analytical inverse kinematics solver [2]. It includes a viewer for 2D or 3D scenes, including a basic but useful set of GUI elements. It also has basic functionality to manage resources such as textures and fonts. It is completely standalone.

SIG has several limitations. It is not a game engine and its rendering capabilities are still basic. It can however be used as the starting point of your own game engine. It was designed over the years to support the development of several research projects, it has been mostly designed to be flexible for a number of possible different uses.

See the readme.txt file available in the base folder of the distribution for a description of the package and compilation notes.

1.1 Code Structure

SIG is divided in 3 libraries: sig, sigogl, and sigkin. Libraries are divided in modules and in each library, header files, source files and class names start with 2 letters indicating in which module the filename or class belongs to. An overview of the modules is available below:

- sig module "gs": generic graphics and system classes, for example: `GsVec`, GsMat, GsOuput, GsArray
- sig module "sn": scene graph nodes, for example: SnModel, SnLines, SnTransform
- sig module "sa": scene actions, for example: SaBBox, SaRenderMode
- sig module "cd": collision detection interface to external collision detectors, available as external distributions,
- sigogl module "gl": classes for interfacing with OpenGL 4, for example: GlProgram, GlTexture
- sigogl module "glr": OpenGL renderers for the shape nodes of the SIG scene graph
- sigogl module "ui": GUI classes, for example: UiButton, UiStyle, UiManager

- sigogl module “ws”: window system classes, for example: WsWindow, WsViewer
- sigkin module “kn”: kinematics module, currently the only module of sigkin, for example: KnSkeleton, KnJoint, KnPosture, KnMotion

1.2 A First SIG Application

Starting a SIG application is as simple as declaring a viewer and a scene, and then running the event manager with ws_run(). See example in Listing 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;
}
```

The output of the program above is shown in Figure X.

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 virtual methods. Multiple example applications are provided in the SIG package.

Multiple examples are available in the main distribution. For example:

- modelviewer: example application to view and inspect models, supporting .obj files.
- skelviewer: example application to view skeletons and motions, supporting .bvh motion files.
- etc.

The best way to start is to study these applications and see how the classes are used to perform the multiple tasks.

The project sigapp is available as a typical empty SIG application structure for starting new projects. [Put here instructions on how to copy a sigapp project for customization and reuse]

1.3 File Formats

A number of SIG classes have their own file formats for saving and loading data. A listing of the SIG data files is provided below.

- GsModel file format .m
- KnSkeleton file format .s

- KnSkeleton data file format .sd
- KnPosture file format .sp
- KnMotion file format .sm

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

2 Overview of Main Classes

`Gs` classes provide several classes which support the basic functionality of SIG applications.

One of the most used classes is `GsArray`.

`GsArray` is an important class built to manipulate arrays very efficiently. The class works similarly to `std::vector` class 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 for primitive types, or for pointers to objects with the use of `GsArrayPt`. See `gs_array.h` for details.

`GsArray` automatically double 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 mew 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 into the array memory re-allocation may happen, invalidating the reference to element 0.

3 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. While its design can be seen to not follow some common practices in modern C++ development, once you get used to it you will see how everything makes sense and you will enjoy how simple and efficient the library is.

3.1 Why not use std classes such as std::ostream and std::vector instead of GsInput, GsOutput and GsArray?

The main reasons are: 1) Too much code bloat 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. 2) Difficulty 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 example, in one of our tests SIG compiled in 19s while this time significantly increased when including std files as follows: iosfwd: 23s, vector: 32s, and iostream: 41s.

3.2 I really want to use std::vector, which is now a standard class in most projects, why not use it instead of GsArray?

In addition to avoiding code bloat and its flexibility, GsArray is also very efficient because it is designed favoring speed of operations. GsArray is often faster than std::vector because it is designed to handle low-level primitive objects and it does not call object constructors and destructors during array manipulation.

In most cases we are dealing with arrays of primitive objects and during dynamic reallocation there is no need to spend time calling unnecessary member initializations. This efficiency however requires care when using GsArray with non-primitive classes, study gs_array.h in order to correctly use the GsArray classes.

Your application can of course use std::vector as its main array class and use GsArray only to interface with SIG classes as needed. GsArray provides a variety of constructors from data, and methods adopt() and abandon() which provide significant flexibility to interface with other classes.

4 Tutorials

References

[1] M. Kallmann. Scalable solutions for interactive virtual humans that can manipulate objects. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment (AIIDE'05)*, pages 69–74, Marina del Rey, CA, June 1-3 2005.

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

[3] M. Kallmann, R. Bargmann, and M. J. Matarić. Planning the sequencing of movement primitives. In *Proceedings of the International Conference on Simulation of Adaptive Behavior (SAB)*, pages 193–200, Santa Monica, CA, July 2004.

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 description is enough to describe the used data files, which are specified 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
...
```