Reference manual for the GiPSiNet simulation description file

Qingbo Cai
Revised 10/6/2005

Contents

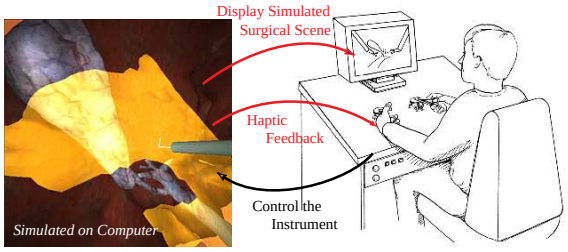

Figure 1: Surgical simulator concept. Simulation snapshot courtesy of F. Tendick [4].

1 Introduction

Computer simulations have become an important tool for medical applications, such as surgical training. Among the various simulation methods for surgical education, virtual environments are a promising new medium (Figure 1). With virtual environments, a user can perform surgery on simulated patient through simulated instruments (haptic devices). However, the current state of medical simulation field is characterized by scattered research projects using a variety of models that are neither inter-operable nor independently verifiable models.

To facilitate shared development of reusable models and to provide a framework that accommodates and interfaces heterogeneous computation models, an open source/open architecture software development framework GiPSi (General Interactive Physical Simulation Interface) is developed. GiPSi is specifically designed to be independent of the specifics of the modeling methods used and therefore facilitates seamless integration of heterogeneous models and processes. Moreover, each model has separate geometries for visualization, simulation, and interfacing, allowing the model developer to choose the most natural geometric representation for each case.

Furthermore, the accessibility to the simulation resources can be substantially extended by network communication. However, the network settings also pose several non-trivial problems for the distributed simulation. An important reason is that haptic devices usually require a high update rate (e.g., the PHANToM Omni has a typical update rate of 1kHz) in order to render stable haptic feedback, but the physical communication platform could be constrained in terms of bandwidth. Moreover, the haptic data traffic cannot expect any special handling within a best-effort network such as the Internet, and consequently can be subject to delays or packet losses due to congestion. Thus, the quality of a networked simulation critically depends on methods to enable an effective remote interaction.

To achieve a simulation with high degree of fidelity and realism over a best-effort network, an middleware module GiPSiNet, which acts as an intermediary between GiPSi and the network and takes actions to remediate for the lack of network *QoS* (Quality of Service), is designed and implemented. GiPSiNet extends GiPSi to the network environment and enhances the user-perceived quality of a networked simulation. Moreover, GiPSiNet inherits from GiPSi its flexibility and potential for extensibility to a variety of sur-

gical simulations.

This document give details for the GiPSiNet simulation description file, where the parameters to set up a networked simulation are included. It is organized as follows: Section 2 introduces how a simulation description file can be generated and used; the parameters in the simulation description file will be described in Section 3; the relation of the parameters with the modules in the simulation system, *i.e.*, how the parameters are dispatched to the modules needing them, will be specified in Section 4; and finally, the data types used by the parameters are defined in Section 5.

2 The use of a simulation description file

Before a simulation begins, a user interface client is required to enter the simulation parameters in a client-side GUI (or HTML forms). The application collects the user interface client's input data, saves them in a simulation description file (in an XML format) on the client-side hard disk so that a client can retrieve the data from his hard disk and reuse them for the setup of simulations later on. Before a simulation begins, this simulation description file is sent to the simulation server. Upon receiving this file, the server application parses this file, and dispatches the parameters to the corresponding modules (see Section 4) to set up the configuration. During the course of a simulation, some parameters (e.g., the visualization parameters) can be reconfigured and transmitted between the simulation server and client. However, this reconfiguration will not affect the simulation description file on the stable storage.

3 Parameter profiles

Depending on the purpose of the parameters, we categorize the parameters into five categories as follows:

1. general parameters;
2. simulation object parameters;
3. connector parameters;
4. visualization parameters;
5. collision detection and response parameters.

Each parameter category will be described in a subsection. Some parameters use the data types defined in Section 5.

Note:

1. For the first cycle, the units of the parameters are in the MKS (meter, kilogram, and second) metric system, and error messages will be emitted if non-MKS units are used. However, user defined units can be handled in later cycles.
2. The parameters are required unless specified otherwise. Error messages will be emitted to the user interface client if a required parameter is missing.

3. Unless specified otherwise, there are no default values for the parameters.

3.1 General parameters

1. *verboseMode*

Type : string;

Description : It specifies where to output the log info at the server side ("Screen": to screen, and "File": to a log file);

Note : If the value of this parameter is set to "File", the server should assign an appropriate file name to the log file such that the log file will not be overwritten by later simulations;

Example value : "Screen".

2. *verboseLevel*

Type : integer between 0 and 9;

Description : It specifies the verbose level. The possible verbose levels are between 0 and 9 from the most brief to the most detailed log info;

Example value : "3".

3. *computationalHook*

Type : boolean;

Description : It specifies whether or not to turn on the computational hook (true: hook on, and false: hook off). When the computational hook is turned on, the data captured are saved to the log file at the server side or displayed to a server-side window, depending on whether the verbose mode is set to "File" or "Screen". Error messages will be emitted if a file or window cannot be opened;

Example value : true.

4. *networkHook*

Type : boolean;

Description : It specifies whether or not to turn on the network hook (true: hook on, and false: hook off). When the network hook is turned on, the data captured are saved to the log file at the server side or displayed to a server-side window, depending on whether the verbose mode is set to "File" or "Screen". Error messages will be emitted if a file or window cannot be opened;

Example value : false.

5. *simTime*

Type : float;

Description : It specifies the desired execution time for one cycle of the main simulation loop in the simulation kernel. If one cycle of the main simulation loop has the actual execution time greater than *simTime*, then, the simulation server ignores this parameter and the execution of the main loop will not be affected. This means that the simulation will not be run in real-time, *i.e.*, the simulation time runs slower than the clock time. Moreover, messages will be emitted to inform a user interface client that the simulation kernel cannot run at the desired rate. Otherwise, the simulation time equals the clock time, and the behaviors of the simulation kernel is reserved for future extensions.

3.2 Simulation object parameters

In the GiPSi framework, organs and physical processes associated with them are represented as Simulation Objects. For example, we can define the heart as an object that simulates the muscle mechanics if we were interested in muscle model of a beating heart [1]. Haptic interface objects are treated as a special type of simulation objects. Two (one haptic interface object and one non-haptic simulation object) or more simulation objects can be specified and used in a simulation.

1. *simObject*

Type : *SimObject* [ ];

Description : It specifies the simulation objects.

Constraint : The number of haptic interface objects should be equal or greater than 1, and the number of non-haptic simulation objects should also be equal or greater than 1.

3.3 Connector parameters

1. *connector*

Type : *Connector* [ ];

Description : It specifies the connectors used in a simulation.

3.4 Visualization parameters

1. *scene*

Type : *Scene* [ ];

Description : It specifies the scenes in a simulation.

2. *shaders*

Type : *VisShader* [ ];

Description : It specifies the shader pool used in a simulation.

3. *textures*

Type : *Texture* [ ];

Description : It specifies the texture pool used in a simulation.

3.5 Collision detection and response parameters

1. *enabled*

Type : boolean;

Description : It indicates whether to use collision detection and response (true) or not (false) in the simulation;

Example value : true.

2. *colDetectionMethod*

Type : string;

Description : It specifies the collision detection method used in a simulation. Currently, the only supported value is "Tri-Tri" standing for triangle/triangle intersection test method.

Example value : "Tri-Tri".

3. *colResponseMethod*

Type : string;

Description : It specifies the collision response method used in a simulation.

Note : Possible methods reserved for future consideration.

4. *colDetectionParameters*

They specify the method specific parameters used in a collision response method.
Note: The parameters of other methods are reserved for future extensions.

(a) *Tri-TriParameters*

The parameters in the triangle/triangle intersection collision detection method are defined as follows. They are ignored unless the value of *colDetectionMethod* is set to "Tri-Tri".

i. *colThreshold*

Type : float

Description : It specifies the distance threshold such that a collision is detected if the distance of two objects is less than this threshold.

5. *colResponseParameters*

Note : They define the method specific parameters used in a collision response method. The parameters are reserved for future extensions.

4 Parameter dispatching

Upon receiving the simulation description file at the beginning or during the course of a simulation, the simulaiton server dispatches the parameters to appropriate modules as follows.

1. The visualization parameters, some simulation object parameters (*i.e.*, *geometries*, *visualizationParameters* and *transformation*) are dispatched to the visualization engine to render images.
2. The connector parameters, the collision detection and response parameters, and the rest of simulation object parameters are dealt by the simulation kernel, which is responsible for creating and maintaining the simulation objects, and for coordinating the object interactions (connector objects).
3. For the parameters to be handled by the haptic server, the simulation kernel creates the haptic interface objects, and feeds the haptic interface object parameters except the parameters handled by the visualization engine (*i.e.*, *geometries*, *visualizationParameters*, and *transformation*), to the haptic server which manages the haptic interfaces and corresponding haptic interface objects.

5 Data types

Besides the simple data types (integer, float, string, boolean) which are defined in the XML Schema specification, complex data types (*Vector, Camera, Color*, etc.) are also used in the simulation description file for the parameter values.

5.1 Vector

The *Vector* data type defines a 3-dimensional vector. It can be used to denote a point or vector in a 3D space. It has one data member:

1. *pos*

Type : float[3];

Description : It specifies the 3 elements of the 3-dimensional vector;

5.2 Geometry

The *Geometry* data type defines a geometry associated with a simulation object. It has the following data members:

1. *geometryFile*

Type : *File*;

Description : It specifies the file where the geometry is defined.

2. *textureNames*

It specifies the names of the textures applied to a simulation object.

(a) *textureName*

Type : string;

Description : It specifies the name of a texture.

5.3 Scene

The *Scene* data type defines a scene. Since generating a view of an object in 3D is similar to photographing the object and, given the spatial position, orientation, and field of view of the "camera", views can be generated [2], parameters related to the "camera" are included in visualization parameters. Moreover, realistic displays of a scene involves applying natural lighting effects to visible surfaces of objects. Hence, the parameters regarding the light source are also included in scene parameters.

For the first cycle, only one camera is used and error messages will be emitted if more than one camera is found. However, multiple cameras should be handled in later cycles.

The camera settings are starting parameters and are changeable during the simulation through mouse, keyboard, and haptic interfaces. If the camera settings are reconfigured during the course of a simulation, instead of altering the existing simulation description file, the application captures and saves the updated camera configurations to a temporary file at the client side, and then sends it to the server, *i.e.*, the client sends an XML subtree (visualization parameters) to the server. Upon receiving this temporary file, the server dispatches the parameters to the visualization engine which will apply the changes to the camera settings.

The *Scene* data type has the following data members:

1. *simulationObjectNames*

Type : string [ ];

Description : It specifies the names of simulation objects in a scene. The value "ALL" means that a scene contains all simulation objects;

2. *cameras*

Type : *Camera* [ ];

Description : It specifies the cameras used in a simulation.

3. *light*

Type : *Light* [ ];

Description : It specifies the static light source used in a simulation.

5.4 Camera

A camera is defined by two positions (the camera position which specifies the view reference point, and the look-at position which specifies the focus point), a direction (view-up vector), and an angle which determines the camera field of view (the larger the field of view, the smaller objects appear in the scene).

The *Camera* type has the following members:

1. *name*

Type : string;

Description : It specifies the identifier of a camera.

2. *type*

Type : string;

Description : It specifies the type of a camera. Currently, the only supported value are "ATTACHED"/"FREE" indicating whether the camera is attached to some haptic interface or not, respectively.

3. *attachedHapticInterfaceID*

Type : integer;

Description : It specifies the identifier of the haptic interface that the camera is attached to.

Note : This parameter is optional, and it appears only if the camera is attached to some haptic interface.

4. *position*

Type : Vector;

Description : It specifies the position where a camera is located in a simulation space.

5. *lookAt*

Type : Vector;

Description : It specifies focus position (target point) of a camera in a simulation space.

6. *upVector*

Type : Vector;

Description : The viewing direction is determined by the *cameraPosition* and the *lookAt* point. However, a camera can rotate around the viewing direction, and this rotation can be described by the upward direction (up vector) on the image plane. This parameter specifies the up vector of a camera.

7. *fieldOfView*

Type : float;

Constraints : The *fieldOfView* parameter is in degrees within the range (0, 180);

Description : It specifies the camera field of view in the horizontal direction (relative to the camera). The vertical field of view angle is computed automatically based on this parameter and the image resolution. Changing this value affects the size of graphics objects displayed in the axes, the greater the angle, the larger the field of view, the smaller the objects appear in the scene.

5.5 Light

A light source is defined by a position, a direction, a light color, and a light type.

The *Light* type has the following members:

1. *type*

Type : string;

Description : It specifies the type of a light source. Currently, the only supported value are "ATTACHED"/"FREE" indicating whether the light is attached to some camera or not, respectively.

2. *position*

Type : *Vector*;

Description : It specifies the position where a light is located in a simulation space.

3. *direction*

Type : *Vector*;

Description : It specifies the light ray direction.

4. *color*

Type : *Color*;

Description : It specifies the color of a light.

5. *attachedCameraName*

Type : string;

Description : It specifies the name of the camera where the light is attached.

Note : This parameter is optional, and it appears only if the light is attached to some camera.

6. *cameraNames*

Type : string [ ];

Description : It specifies the names of the cameras to which the light is visible. The value 'ALL' means that the light is visible to all cameras in the scene.

5.6 Color

The *Color* data type defines a color in the RGB and opacity form. It has the following data members:

1. *red*

Type : float between 0 and 1;

Description : It specifies the intensity of the red color.

2. *green*

Type : float between 0 and 1;

Description : It specifies the intensity of the green color.

3. *blue*

Type : float between 0 and 1;

Description : It specifies the intensity of the blue color.

4. *opacity*

Type : float between 0 and 1;

Description : It specifies the opacity value. This parameter is optional and its default value is 0.5.

5.7 Shader

The *Shader* data type defines the computer simulation or calculation for how the faces of a simulation object will look like when illuminated by a virtual light source. It has the following data members:

1. *name*

Type : string;

Description : It specifies the name of the shader.

2. *params*

Type : *ShaderParam* [ ];

Description : It specifies the parameters associated with a shader.

5.8 ShaderParam

The *ShaderParam* data type defines a parameter associated with a shader. It has the following data members:

1. *name*

Type : string;

Description : It specifies the name of a shader parameter.

2. *value*

Type : string;

Description : It specifies the value of a shader parameter.

5.9 VisShader

The *VisShader* data type defines a shader used in a simulation. It has the following data members:

1. *name*

Type : string;

Description : It specifies the name of a shader.

2. *path*

Type : string;

Description : It specifies the path where the files of a shader are stored.

3. *passes*

Type : integer;

Description : It specifies the number of passes incorporated in a shader.

5.10 Texture

The *Texture* data type defines a texture used in a simulation. It has the following data members:

1. *name*

Type : string;

Description : It specifies the name of a texture.

2. *type*

Type : string;

Description : It specifies the type of a texture. Currently, four texture types are supported:
i) "GIPSI_2D_DYNAMIC_SERVER";
ii) "GIPSI_2D_STATIC_SERVER";
iii) "GIPSI_2D_STATIC_CLIENT";
iv) "GIPSI_3D_STATIC_CLIENT".

3. *file*

Type : *File*;

Description : It specifies the file where a texture is defined.

5.11 Rotation

The *Rotation* data type defines the rotation applied to a 3D object. Three options (*i.e.*, rotation matrix, rotation axis and angle, rotation angles around *x, y, z* axes) are provided to specify a rotation.

Positive rotation directions about the coordinate axes are counterclockwise, when looking toward the origin from a positive coordinate position on each axis [2].

1. *rotationMatrix*

   (a) *R*

   Type : float [3] [3];

Description : It specifies a rotation matrix. Assume a point at position P in 3D space. After a rotation with a rotation matrix R, the new position P' of the point in the same coordinate can be represented as:

$$P' = RP\ ;$$

Constraints : There are three constraints for this parameter:

i. The dimension of R should be 3 by 3;

ii. $RR^T = I$;

iii. $det(R) = 1$.

Error messages will be emitted if any constraint is violated.

2. *axisRotation*

   (a) *axis*

Type : *Vector*;

Description : It specifies the axis around which the rotation is applied.

Constraint : This parameter should be normalized. Otherwise, error messages will be emitted.

   (b) *angle*

Type : float;

Description : It specifies the rotation angle in degrees around the axis.

3. *CoordinateRotation*

   (a) *angles*

Type : *Vector*;

Description : It specifies the rotation angles in degrees around *x, y, z* coordinates respectively.

Note : The order of rotation is fixed as follows: x-axis rotation $\rightarrow$ y-axis rotation $\rightarrow$ z-axis rotation.

5.12 Transformation

In computer graphics, the shape of individual objects in a scene can be constructed in separate coordinate reference frames called modeling coordinates, or local coordinates. Once individual object shapes have been specified, we can place the objects into appropriate positions within the scene using a reference frame called world coordinates [2]. Accordingly, the geometry of a simulation object is defined in a geometry file in the local coordinates. Moreover, the simulation object can be customized (*i.e.*, scaled and rotated) in its local coordinates. To put individual simulation objects

together in the 3D simulation space in world coordinates, coordinates translation is necessary.

The *Transformation* data type defines the transformations applied to an object in a 3D space. The order of the transformation is as follows: scaling → rotation → translation, *i.e.*, a simulation object is first scaled and then rotated in its local coordinate frame, and at last is translated to the coordinates of the simulation space. The *Transformation* type has the following data members:

1. *scaling*

Type : *Vector*;

Description : It specifies the scaling factor in *x, y, z* coordinates. This parameter is optional and the default value is [1, 1, 1].

2. *rotation*

Type : *Rotation*;

Description : It specifies the rotation of an object. This parameter is optional and the default value is null, *i.e.*, no rotation is applied.

3. *translation*

Type : *Vector*;

Description : It specifies the shift vector used in a translation. This parameter is optional and the default value is [0, 0, 0].

5.13 File

The *File* data type defines a file (path and file name). It has the following data members:

1. *path*

Type : string;

Description : It specifies the path of a file.

2. *fileName*

Type : string;

Description : It specifies the file name of a file.

5.14 Connector

The *Connector* data type defines a connector which connects two simulation objects.

1. *name*

Type : string;

Description : It uniquely identifies a connector;

Example value : "connector1".

2. *type*

Type : string;

Description : It specifies type of the connector. Currently, possible types of a connector are:

(a) "FEM/CBE" connector which connects two simulation objects modeled by FEM and cardiac bioelectricity method respectively;

(b) "FEM/LF" connector which connects two simulation objects modeled by FEM and lumped fluid method respectively;

Example value : "FEM/CBE".

3. *objectName1/objectName2*

Type : string;

Description : These two parameters specify the names of simulation objects connected by a connector. Error message will be emitted in any of the following three cases:

(a) either or both of these two parameters are missing;

(b) either of both of these two parameters do not correspond to existing objects;

(c) the types of the connected objects do not match the type of the connector (for example, a FEM object and a LF object are connected by an "FEM/CBE" connector);

Example value : "simObject1"/"simObject2".

4. *modelParameters*

The parameters of a certain model (e.g., *FEM_CBEParameters*) are ignored unless the value of *type* is set to the corresponding model (e.g., *FEM_CBE*).

(a) *FEM_CBEParameters*

These parameters are used by a connector which connects two simulation objects modeled by FEM and Cardiac Bioelectricity methods respectively.

The following three parameters, *i.e.*, *ExStressValueXX*, *ExStressValueYY*, and *ExStressValueZZ*, represent $\epsilon_{11}, \epsilon_{22}, \epsilon_{33}$ respectively

in the matrix

$$\begin{pmatrix} \epsilon_{11} & 0 & 0 \\ 0 & \epsilon_{22} & 0 \\ 0 & 0 & \epsilon_{33} \end{pmatrix},$$

which in turn denotes the internal contraction stress when the muscle is excited.

i. *ExStressValueXX*

Type : float

ii. *ExStressValueYY*

Type : float

iii. *ExStressValueZZ*

Type : float

(b) *FEM_LFParameters*

These parameters are used by a connector which connects two simulation objects modeled by FEM and Lumped Fluid methods respectively.

i. *corrFile*

Type : File

Description : The correspondences of the nodes on the boundaries of the FEM model and the LP model are generated by another program, and are stored in a *.corr* file residing on the simulation server. This parameter specifies the *.corr* file.

5.15 SimObject

The *SimObject* data type defines a simulation object. Haptic interface objects are treated as a special type of simulation objects.

Each simulation object may have multiple geometries. Each geometry is loaded from a file which defines the surface or volume geometry of the simulation object. This geometry is specified in a separate file other than the simulation description file, since: i) it is typically generated by another program; and ii) it conforms to some standard format, for example, the surface geometry file may conform to the Wavefront *.obj* file format for 1 group and 1 object, and the volume geometry file may conform to the NETGEN .neu file format. To provide more flexibility, a client is allowed to customize the size, shape, and location of a simulation object by specifying transformations (*i.e.*, scaling, rotation, and translation) applied to this geometry.

For each simulation object, the parameters needed to set up the simulation configuration are described as follows.

1. *name*

Type : string;

Description : It uniquely identifies a simulation object in a simulation;

Example value : "simObject1".

2. *type*

Type : string;

Description : It specifies the type of a simulation object. For a non-haptic simulation object, this parameter specifies the type of method used to model the simulation object. Currently, possible types for non-haptic objects are "FEM"/"MSD"/"CBE"/"LF"/"PHIO" standing for Finite Element Method/Mass-Spring-Damper/Cardiac Bioelectricity/Lumped Fluid method/Phantom Haptic Interface Object respectively. Error messages will be emitted if the parameter value is not a currently supported type. This parameter can be used to determine if a simulation object is a haptic interface object or a non-haptic simulation object;

Example value : "FEM";

Note : The types of haptic interface objects are reserved for future extensions.

3. *time*

Type : float;

Description : It specifies the time when the simulation object simulation begins;

4. *geometries*

Type : *Geometry* [ ];

Description : It specifies the geometries of a simulation object.

5. *visualizationParameters*

Other than the general visualization parameters (*i.e.*, cameras) defined in Section 3.4, a simulation object can have its own visualization parameters which specify the base color, texture, etc. of the object.

Faces (triangles) on the surface of an object can either be filled with a base color or a rasterized image (texture). Shading involves the computer simulating or calculating how the faces will look like when illuminated by a virtual light source. The exact calculation varies depending on the available data about the faces being shaded and the shading technique applied.

(a) *baseColor*

Type : *Color*;

Description : It specifies the base color of a simulation object.

(b) *shaders*

Type : *Shader* [ ];

Description : It specifies the shaders applied to a simulation object.

6. *transformation*

Type : *Transformation*;

Description : It specifies the transformations applied for customizing the geometry of a simulation object.

7. *objParameters*

The object type dependent parameters for non-haptic objects and haptic interface objects are *NHParameters* and *HIOParameters* respectively, and are described as follows.

6a. *NHParameters*

(a) *timeStep*

Type : float;

Description : It specifies the numerical integration time step of the individual simulation object. This time differs from the time between successive updates of a simulation object's state in that the latter may include several numerical integration time steps. The default value of this parameter is 0.01s.

Example value : 0.01.

(b) *numericMethod*

Type : string;

Description : It specifies the numeric method used to simulate a simulation object. Currently, possible values are "Euler"/ "MidPoint"/"RK4"/ "ImplicitEuler". Error messages will be emitted if the parameter value is not a supported numerical method;

Example value : "Euler".

(c) *modelParameters*

The parameters of a certain model (e.g., *FEMParameters*) are ignored unless the value of *type* is set to the corresponding model (e.g., *FEM*).

i. *FEMParameters*

These parameters are used when a simulation object is modeled with FEM (Finite Element Method).

A. *Rho*

Type : float;

Description : It specifies the mass density.

B. *Mu*

Type : float;

Description : It specifies the Lamee constant μ for elastic material.

C. *Lambda*

Type : float;

Description : It specifies the Lamee constant λ for elastic material.

D. *Nu*

Type : float;

Description : It specifies the first strain velocity term. It is equivalent to ψ in [3].

E. *Phi*

Type : float;

Description : It specifies the second strain velocity term [3];

ii. *MSDParameters*

These parameters are used when a simulation object is modeled with MSD (Mass-Spring-Damper) method.

A. *K*

Type : float;

Description : It specifies the stiffness multiplier coefficient.

B. *D*

Type : float;

Description : It specifies the damping multiplier coefficient.

C. *mass*

Type : float;

Description : It specifies the mass.

D. *MSDFile*

Type : *File*;

Description : It specifies the file where the MSD model of an object is defined. This file is typically generated by another program and resides on the simulation server.

iii. *CBEParameters*

These parameters are used when a simulation object is modeled with the Cardiac Bioelectricity method, which uses a simple propagating wave model.

A. *TempFreq*

Type : float;

Description : It specifies the temporal frequency of the wave in Hz.

B. *SpVel*

Type : float;

Description : It specifies the spatial propagation velocity of the wave.

C. *DutyCycle*

Type : float;

Description : It specifies the duty cycle (active percentage) of the wave.

iv. *LFParameters*

These parameters are used when a simulation object is modeled with Lumped Fluid method.

A. *Pi*

Type : float;

Description : It specifies the input pressure.

B. *Po*

Type : float;

Description : It specifies the output pressure.

C. *Pfo*

Type : float;

Description : It specifies the rest pressure inside the chamber.

D. *Kc*

Type : float;

Description : It specifies the coupling stiffness.

E. *Bc*

Type : float;

Description : It specifies the coupling damping.

F. *Ri*

Type : float;

Description : It specifies the input flow resistance.

G. *Ro*

Type : float;

Description : It specifies the output flow resistance.

6b. *HIOParameters*

These parameters are haptic interface type dependent, and most of them are reserved for future extensions.

(a) *ID*

Type : integer;

Description : It specifies the identifier of the haptic interface that is attached to the haptic interface object. The mapping between the identifier (e.g., 21) and the haptic interface (e.g., PHANToM Omni) must be defined and hard-coded. This mapping is reserved for future extensions.

References

[1] T. Goktekin and M.C. Cavusoglu. *GiPSi: A Draft Open Source/Open Architecture Software Development Framework for Surgical Simulation*, Technical Report, Case Western Reserve University, 2004.

[2] Donald Hearn and M. Pauline Baker. *Computer Graphics, C Version*, Prentice Hall, 1997.

[3] James F. O'Brien and Jessica K. Hodgins. *Graphical modeling and animation of brittle fracture*, In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 137 – 146. ACM Press/Addison-Wesley Publishing Co., 1999.

[4] F. Tendick, M. Downes, T. Goktekin, M. C. Çavuşoğlu, D. Feygin, X. Wu, R. Eyal, M. Hegarty and L. W. Way. *A Virtual Environment Testbed for Training Laparoscopic Surgical Skills*, In *Presence*, 9:3, pages 236-255, June 2000.