Chapter 5

WoodFrameMesh Program Architecture

This chapter presents the architecture of the WoodFrameMesh program. The program is capable of generating complete 3-dimensional finite element models of wood frame houses. The originality of the generator relies on providing an object oriented platform for combining different mesh generation algorithms and modeling procedures. The use of object oriented design and the dynamic nature of object oriented programming facilitates enhancement of the program. A study of finite element meshing techniques shows that the problem lends itself very well to object oriented design i.e. abstraction, encapsulation and polymorphism. Geometric primitives such as triangles, tetrahedral, cubes and cylinders form natural “objects” (Cross et al., 1999). The program is written in C++ which is one of the widely used object oriented programming languages. The data structure design involves extensive use of the C++ Standard Template Library (STL) which provides efficient approaches for storing and processing data. As dynamic storage, insertion, deletion and sorting of meshing data are some of the key issues involved in mesh generation processes, STL proves to be the best choice in handling these operations.
5.1 Object Oriented Concepts

Object oriented programming is currently seen as the best way of designing any application, as it confers high flexibility and adaptability to it. In object oriented design, all entities are represented by objects. Each object constitutes a number of attributes that define the purpose and state of the object. An attribute, also called an instance variable, may be a simple integer, floating point, string, or a complex data structure. However, the definition of the object extends beyond the data structure to include also its behavior. The behavior of the object is defined by methods, which are procedures which manipulate or return the state of the object. Instance variables are private to the object and are made invisible to the outside world. This aspect is known as encapsulation or “information hiding”. Depending upon the purpose of the object, some methods may be defined as private.

Objects interact with each other by sending messages to each other. Messages along with any arguments that accompany them constitute the public interface of an object. An object reacts to the message by executing one or more methods and by returning a response. Objects that have similar instance variables and methods are grouped together into classes. A class describes the list of the attributes (instance variables) and behavior (methods) for all objects that are members of the class. Figure 5.1 shows a C++ interface for setting the attributes and methods of a sample class xyz.

```cpp
class xyz
{
public:
    xyz();
    xyz(int a, int b, string c);
    ~xyz();
    /*Methods*/
    void SetA(int &a);
    void SetB(int &b);

private:
    /*Attributes*/
    int a;
    int b;
    string c;
};
```

Figure 5.1: Class xyz’s attributes and methods (C++ interface)
A popular feature in which one object inherits the members of its base class is called inheritance. Inheritance refers to creating a new class as an extension (specialization) of an existing class. This feature saves additional programming effort and hence avoids duplication. The new sub-class, shown in Figure 5.2, automatically possesses all the methods and attributes of its super-class, in addition to the new attributes and methods defined. Also, the new sub-class can be treated as an object of the base class. In short, inheritance is a formal and controllable way of developing classes stage by stage, while still being able to use any of the intermediate versions (Lafore, 2001).

```cpp
class uvw: public xyz
{
public:
    uvw();
    uvw(int a, int b, string c, int d, int e);
    ~uvw();
    /*Methods*/
    void SetD(int &d);
    void SetE(int &e);
    .....  
private:
    /*Attributes*/
    int d;
    int e;
    ..... 
};
```

Figure 5.2: Inherited class uvw’s attributes and methods. All public and protected attributes and methods of class xyz also belong to class uvw (C++ interface)

The method for any class is identified by three things: (1) its name, (2) the class it belongs to, and (3) the data passed to it. Thus, it is possible to have methods with the same name, attached to different classes, or inside the same class with different type of data passed to it. This feature is called polymorphism. This method can be applied to operators too, and in such a case it is known as operator overloading. Figure 5.3 presents an interface showing operator overloading.
class abc
{
public:
    abc();
    abc(int a, int b, double c);
    ~abc();
    /*Methods*/
    void operator * (const int &im);
    void operator * (const double &dm);
    .......
    .......
private:
    /*Attributes*/
    int a;
    int b;
    .......
    .......
};

Figure 5.3: Example of polymorphism (operator overloading, C++ interface)

Writing an object-oriented program requires detailed planning and a thorough understanding of the problem to ensure the right selection of classes, messages, responses, inheritances, and so on. Maintenance, reusability and extensibility are very important norms to judge the performance of the software, and a well designed object oriented program provides a sound platform for it.

5.2 Object Oriented C++ Programming for FE Mesh Generation

The application of object oriented C++ design to FE mesh generation has several advantages. The primary advantage is that it encourages the developer to abstract out the essential immutable qualities of the components of the FE meshing. These abstractions form the objects which develop the building block of the program. The use of object data structures such as node, line segment, triangle, etc., accelerates the algorithm implementation. Even different mesh generation algorithms are treated as objects and they act upon different geometries to produce the mesh. The secondary advantage which follows is the flexibility in design. Flexibility in design is defined as an ease of adding or modifying an object data structure without affecting the other parts of the program. The other advantages include:
1) Easy expansion of the architecture to the distributed one.

2) Increase in computational efficiency. Several mesh generation algorithms require insertion, deletion, replacement and sorting completely dynamically. That is why object oriented design and C++ implementation is a best fit for such applications. Also, as one does not know the number of triangles, nodes, and other objects which would be generated from a given geometry, a dynamic storage scheme is required to have a more memory efficient system. Object oriented programs such as C++ provide a standard template library for such situations.

3) Easy integration with object oriented computer aided design (CAD) modeler and solver. Object oriented programming opens up prospects for much more “integrated packages” (Cross et al., 1999).

4) The object oriented approach is more modular and testing can be better handled.

5) The approach provides close integration of design, analysis and development phases. This enables a fast turnaround time in the software development.

6) Numerical computing becomes much more compact and easier to follow.

7) The objects ability to save all the pertinent data is of great use for writing files for use in finite element programs.

Several FE mesh generation algorithms have been developed to date and a huge amount of literature is available on it, but the literature pertaining to object oriented mesh generation implementation using C++ has not very large. The object oriented C++ implementation made by Bastion and Li (2003) and Karamete et al., (1997) is the only formal discussion found in literature. These papers present a broader and efficient approach towards mesh generation using object oriented programming, but they lack the use of powerful in built data structures provided by C++. Instead, the use of user designed doubly linked list, structures, containers and binary trees are presented. The C++ language has evolved in recent years and one of the most powerful features it accompanies lies in its Standard Template Library (STL). The STL can be used very efficiently for generic programming techniques. The user can define various data types
and use STL containers (sequential or associative) for storage. These storages provide
dynamic insertion and deletion of data. STL also provide different sets of iterators
(counters), for fast access of information and algorithms for conditional sorting,
searching, etc., over STL containers. Various other features which are not mentioned
here are described in books on C++, C++ data structures, or C++ STL (Lafore, 2001;
Drozdek, 2001).

5.3 Program Architecture and Modeling Classes

The objective is to develop software which generates a complete finite element model
of wooden structures. The architecture of the program is conceived to accommodate
triangular, quadrilateral and other element generation. Various object oriented techniques
are applied throughout the implementation, which include encapsulation, class hierarchy,
inheritance and polymorphism. Instead of a procedural data flow, the program is made of
a set of objects. The execution of the program is accomplished by the interaction among
objects. The platform developed is robust, geometry independent, and requires minimum
user interaction. Under the current meshing capability, the program can generate
triangular and quadrilateral shell elements. The triangular elements can be generated
over any surface domain, whereas the quadrilateral elements are currently limited to only
quadrilateral domains. In addition, the program can also assign materials, sections, frame
elements, non-linear link elements\(^1\), and nodal loads at the desired locations. The
information required to generate the complete finite element model is provided inside a
.txt file. The steps in the creation of finite element model are as follows:

1.) Read in the input data
2.) Build the model to be meshed
3.) Mesh the model
4.) Write the results to a file

The basic architecture of the program is shown in Figure 5.4. The area inside the
dotted boundary is the portion implemented inside the WoodFrameMesh program. The
detailed description of the layout is discussed in the following sub-sections. The
interfaces for the classes are presented as pseudo C++ code. The pseudo C++ code

\(^1\) These elements are not really non-linear in this version and act as linear link elements similar to springs
presented is similar to the one implemented inside the program, the only difference being that private and protected entities are not shown.

Figure 5.4: Basic architecture of the WoodFrameMesh program

5.3.1 **Input Model**

The input model originates from the CAD model\(^2\), consisting of points, lines and polygons which in turn are outlined as a computational domain inside the input file. The remaining part which defines the material properties, section properties, restraints, springs, frames, non-linear links and nodal loads are defined by the user inside the input file, as per the requirement. The input data is stored in a .txt input file. The format for the input data is discussed in Appendix A. The *input model* is a generalized object and it can be presented in different format inside the input file. The program currently provides a .txt file reader which is a child class of the *domain builder* class. This .txt file reader is

\(^2\) Currently the input model is generated manually from the CAD model by the sponsor.
5.3.2 Domain Builder and Txt File Reader

The \textit{KMDomainBldr} class is defined as an abstract base class (ABC). An abstract base class is one which provides only methods and their implementations are provided in their child classes. Objects of such a class cannot be instantiated\(^3\) at run time of the program. \textit{KMDomainBldr} defines a virtual method called BuildDomain which reads the input file and populates the \textit{KMDomain} class container. The \textit{KMDomain} class object is passed as an input to the \textit{KMDomainBldr} class object. The \textit{KMDomainBldr} class interface is shown in Figure 5.5.

\begin{verbatim}
class KMDomainBldr
{
  public:
    KMDomainBldr(KMDomain *Domain);
    virtual ~KMDomainBldr();
    virtual bool BuildDomain(void) = 0;
};
\end{verbatim}

Figure 5.5: Interface of KMDomainBldr class

The \textit{KMTxtFileReader} class is a child class of \textit{KMDomainBldr}. The BuildDomain method is the only public method defined inside this class. The other methods defined are private and are responsible for reading different blocks of input from the input file and simultaneously they populate the \textit{KMDomain} class containers. All these private methods are called inside the BuildDomain method. If the file format is as discussed in Appendix A, the method returns true, else it returns false. In the former case the program continues to run and in the latter case the program flags an error message and asks the

\footnote{Generating a particular object from its class}
user to correct the file format. The class interface for KMTxtFileReader is shown in Figure 5.6.

```cpp
class KMTxtFileReader: public KMDomainBldr
{
public:
    KMTxtFileReader();
    KMTxtFileReader(const char *InputFileName, KMDomain *Domain);
    ~KMTxtFileReader();
    bool BuildDomain(void);
};
```

Figure 5.6: Interface of KMTxtFileReader class

### 5.3.3 Domain and Echo Filter

The KMDomain class acts as a repository for input data. The methods to populate the KMDomain containers are defined as public so that they can be called inside any of the child classes of KMDomainBldr class. The other methods include “accessor” methods (get methods) to access the populated data during FE model generation. The class interface for KMDomain is shown in Figure 5.7. As can be seen, the data is stored in associative containers which provide dynamic linking of objects with the help of their keys. For example, materials are associated to sections which in turn are associated to elements; nodes are associated to geometric objects. All the associated objects can directly be accessed by their keys without iterating over the containers and hence provide fast access.

Another important task which needs to be performed before any generation is done is to ensure the validity of input data. These checks are made inside the above “mutator” methods (set methods) at each step while reading the data from the input file. This provides fast checking as the data is read, checked (errors stored if found any) and echoed with a zero time lag in between. A KMEchoFilter class is written for this purpose. The input data and errors encountered are passed to the KMEchoFilter object which acts as a repository for any errors if encountered. The input data read is printed into a .eko file using the methods defined inside KMEchoFilter class. Once all the data is read, the
program stops if it encounters any errors. If any errors are found, the program prints all
the encountered errors at the end of the .eko file.

class KMDomain
{
    public:
        KMDomain();
        KMDomain(KMEchoFilter *Echo);
        virtual ~KMDomain();
    void GetNodes(std::map<int, KMNode*>&);
    void GetMaterials(std::map<std::string, KMMaterial*>&);
    void GetSections(std::map<std::string, KMSection*>&);
    void GetGeometricObjects(std::multimap<std::string, KMGeometricObject*>&);
    void GetUnits(std::string&, std::string&);
    void GetProblemName(std::string&);
    void SetUnits(const std::string&, const std::string&);
    void SetNode(const int&, const double&,
                 const double&, const double&);
    void SetRestraintData(const int&, const deque<int>&,
                          bool*, const double&);
    void SetSpringData(const int&, const deque<int>&, double*, const double&);
    void SetElasticMaterial(const std::string&, const double&,
                            const double&, const double&,
                            const double&, const double&,
                            const double&, const double&);
    void SetFrameSection(const std::string&, const std::string&,
                          const double&, const double&,
                          const double&, const double&,
                          const double&, const double&,
                          const double&);
    void SetShellSection(const std::string&, const std::string&,
                         const std::string&, const std::string&,
                         const double&, const double&,
                         const double&);
    void SetNonLinearLinkSection(const std::string&,
                                  const std::string&, double*, double*,
                                  const double&, const double&);
    void SetFrameElement(const int&, const deque<int>&,
                         const std::string&, const int&,
                         const double&, const double&);
    void SetShellData(const int&, const std::string&, const int&,
                       const std::string&,
                       const std::deque<int>&);
    void SetNonLinearLinkElement(const int&, const std::deque<int>&,
                                 const std::string&, const double&,
                                 const double&);
    void SetNodalLoad(const int&, const std::string&,
                      const std::string&, const std::deque<int>&,
                      const double&, double*, const double&);
};

Figure 5.7: Interface of the KMDomain class
5.3.4 FE Model Builder

Instances of KMFEModelBldr class are used to instantiate and execute different mesh generation class objects and methods, respectively, depending upon the desired meshing type (i.e. triangular or quadrilateral). The inputs to this class are an instance of KMDomain class and an instance of the KMFEModel class. The KMDomain class contains all the information required to generate a complete finite element model and the KMFEModel class is a container for discretized elements generated by the mesh generator objects. The class interface for KMFEModelBldr is shown in Figure 5.8.

```cpp
class KMFEModelBldr
{
public:
    KMFEModelBldr();
    KMFEModelBldr(KMDomain *Domain, KMFEModel *FEModel);
    virtual ~KMFEModelBldr();
    virtual void BuildFEModel(void);
};
```

Figure 5.8: Interface for the KMFEModelBldr class

5.3.5 FE Mesh Generator

This class is a base class for different meshing child classes. It defines a virtual method called BuildFEMesh which generates mesh for a given geometric object. The geometric object could be a line object, surface object, or volume object. Line objects are “special objects” which are meshed to generate restraints, springs, frames, non-linear link and nodal loads depending upon their instances. This class also populates the KMFEModel container objects through their “mutator” methods. The other methods include performing activation of the time clock to get the time taken during mesh generation. The Figure 5.9 shows the interface for the KMFEMeshGenerator class.
class KMFEMeshGenerator
{
public:
    KMFEMeshGenerator();
    KMFEMeshGenerator(KMGeometricObject *GeomObj, KMFEModel *FEModel);
    virtual ~KMFEMeshGenerator();
    virtual bool BuildFEMesh(void) = 0;
    virtual void SetFEModel(KMFEModel *FEModel);
    virtual void SetGeometricObject(KMGeometricObject *GeomObj);
    virtual void StartClock(void);
    virtual void StopClock(void);
    virtual double GetTimeTaken(void);
};

Figure 5.9: Interface for KMFEMeshGenerator class

5.3.6 Advancing Front Triangulation

This class defined as KMAFT is a child class of KMFEMeshGenerator class. An instance of this class is responsible for meshing a surface geometric object using the advancing front method. The constructor requires KMSurfaceObject along with KMFEModel objects as inputs. The method BuildFEMesh is the only public method and is responsible for calling other methods to generate triangular elements over the KMSurfaceObject object. The nodes and elements generated are passed to the container, which is an instance of KMFEModel class. The Figure 5.10 shows the interface for the KMAFT class. The public class interface for KMQuad, which is written for generating quadrilateral elements, is the same as the one shown for KMAFT. The differences lie in the private methods used for generating the mesh.


```cpp
class KMAFT: public KMFEModelGenerator
{
public:
    KMAFT();
    KMAFT(KMSurfaceObject *SurfObj, KMFEModel *FEM);
    virtual ~KMAFT();
    bool BuildFEMesh(void);
};
```

Figure 5.10: Interface for the KMAFT class

### 5.3.7 FE Model and .S2K Writer

The `KMFEModel` class acts as a repository for generated finite elements and boundary conditions. It also serves as a base class for different file writer child classes. Different file writers may be required to produce input for different kinds of solvers. A virtual method called `PrintModelToFile` is defined inside the `KMFEModel` class which needs to be implemented inside each child class for generation of the final output model. The other public methods include “mutator” functions for setting the FE model data generated by different mesh generators. The program currently provides a .S2K file writer child class called `KMS2KFileWriter`. The .S2K format is recognized by SAP 2000 and WoodFrameSolver. The Figures 5.11 and 5.12 shows the interfaces of `KMFEModel` and `KMS2KFileWriter` classes, respectively.
class KMFEModel
{
    KMFEModel();
    virtual ~KMFEModel();
    virtual bool PrintModelToFile(void) = 0;
    void AddNode(KMNode *Node);
    void AddSpring(KMSpring *Spring);
    void AddRestraint(KMRestraint *Restraint);
    void AddFrame(KMFrame *Frame);
    void AddShell(KMShell *Shell);
    void AddNonLinearLink(KMNonLinearLink *NonLinLink);
    void AddNodalLoad(KMNodalLoad *NodalLoad);
};

Figure 5.11: Interface for the KMFEModel class

class KMS2KFileWriter:public KMFEModel
{
    public:
    KMS2KFileWriter(KMDomain *Domain);
    virtual ~KMS2KFileWriter();
    virtual bool PrintModelToFile(void);
};

Figure 5.12: Interface for the KMS2KFileWriter class

5.4 Other Classes

This section presents classes whose objects are used in representing the basic input data from the CAD model and the output data for generating the finite element model. The similar pseudo code interface as shown in previous sections is presented for each class and a discussion is made.

5.4.1 Geometric Object

A geometric object is a generalized term used to denote a line object, a surface object or a volume object. A class KMGeometricObject is defined which contains all common attributes and methods for any geometric object. The other specialized methods are retained inside the child classes. The class interface for KMGeometricObject is shown in Figure 5.13.
class KMGeometricObject
{
public:
    KMGeometricObject();
    KMGeometricObject(int, std::string, std::deque<KMNode*>,
                     std::string, int, double);
    virtual ~KMGeometricObject();
    virtual void SetObjectID(const int &);
    virtual void SetObjectName(const std::string &);
    virtual void SetBoundaryNodes(const std::deque<KMNode*>&);
    virtual void SetMeshingType(const std::string&);
    virtual void SetObjectAsso(const int&);
    virtual void SetMeshSize(const double&);
    virtual int GetObjectID(void) const;
    virtual std::string GetObjectName(void) const;
    virtual void GetBoundaryNodes(std::deque<KMNode*>&);
    virtual int GetObjectAsso(void) const;
    virtual std::string GetMeshingType(void) const;
    virtual double GetMeshSize(void) const;
};

Figure 5.13: Interface for the KMGeometricObject class

5.4.2 Surface Object

The KMSurfaceObject class is a child class of the KMGeometricObject class and represents an object bounded by straight lines. Even curvilinear surface objects approximated by straight lines can be represented as KMSurfaceObject. The object can contain multiple openings and interior constraint lines. The boundary information for openings and interior constraint lines are provided through AddOpeningNodes and AddConstraintLineNodes methods. The SetSurfaceProperties method is defined to form an equation of a plane and to find the positive normal to the plane. The interface for KMSurfaceObject is shown is Figure 5.14.
class KMSurfaceObject
{
public:
    KMSurfaceObject();
    KMSurfaceObject(int, std::string, std::deque<KMNode*>, std::string, int, std::string, double);
    ~KMSurfaceObject();
    void SetObjectType(const std::string&);
    std::string GetObjectType(void) const;
    void AddOpeningNodes(const std::deque<KMNode*>&);
    void AddConstraintLineNodes(const std::deque<KMNode*>&);
    void SetSurfaceProperties(void);
    KMSpaceVector GetPositiveNormal(void) const;
    void GetInitialGenerationFront(std::deque<KMLineSegment*>&);
};

Figure 5.14: Interface for the KMSurfaceObject class

5.4.3 Line Object

The line object classes KMRestraintLineObject, KMSpringLineObject, KMFrameLineObject, KMNNonLinearLinkLineObject and KMLoadLineObject are all child classes of KMGeometricObject. An instance of these objects is used to define a set of nodes which represents a line object in space, over which a linear meshing is done and the attributes are assigned to the elements generated. The interfaces for these classes are shown in Figures 5.15, 5.16, 5.17, 5.18 and 5.19.

class KMRestraintLineObject: public KMGeometricObject
{
public:
    KMRestraintLineObject();
    KMRestraintLineObject(int, std::string, std::deque<KMNode*>,
                          double, bool*);
    ~KMRestraintLineObject();
    void SetRestraints(bool*);
    bool* GetRestraints(void);
};

Figure 5.15: Interface for the KMRestraintLineObject class
class KMSpringLineObject: public KMGeometricObject
{
public:
    KMSpringLineObject();
    KMSpringLineObject(int, std::string, std::deque<KMNode*>,
           double, double*);
    ~KMSpringLineObject();
    void SetStiffness(double*);
    double* GetStiffness(void);
};

Figure 5.16: Interface for the KMSpringLineObject class

class KMFrameLineObject: public KMGeometricObject
{
public:
    KMFrameLineObject();
    KMFrameLineObject(int, std::string, std::deque<KMNode*>,
                       int, double, double);
    ~KMFrameLineObject();
    void SetNumberofSegments(const int&);
    void SetAngle(const double&);
    int GetNumberofSegments(void) const;
    double GetAngle(void) const;
};

Figure 5.17: Interface of the KMFrameLineObject class

class KMNonLinearLinkLineObject: public KMGeometricObject
{
public:
    KMNonLinearLinkLineObject();
    KMNonLinearLinkLineObject(int, std::string, std::deque<KMNode*>,
                               double, double);
    ~KMNonLinearLinkLineObject();
    void SetAngle(const double &Angle);
    double GetAngle(void) const;
};

Figure 5.18: Interface for the KMNonLinearLinkLineObject class
class KMLoadLineObject: public KMGeometricObject
{
public:
    KMLoadLineObject();
    KMLoadLineObject(int,std::string,std::deque<KMNode*>,std::string,
        double,double*,double);
    ~KMLoadLineObject();
    void   SetLoadType(const std::string &LoadType);
    void   SetFactor(const double &Factor);
    void   SetForces(double *Forces);
};

Figure 5.19: Interface for the KMLoadLineObject class

5.4.4 Space Vector and Node

A space vector class called KMSpaceVector represents a physical point in space and provides methods for vector operations. These methods for vector operations are often required in solving computational geometry problems. The interface for the KMSpaceVector class is shown in Figure 5.20. The KMNode class is a child class of the KMSpaceVector class. The KMNode class object is a point in space to which elements are connected. In addition to the functionality of the KMSpaceVector class, the KMNode class defines an additional attribute called a node number. A node number is a unique attribute of the KMNode class. The KMNode class forms an important data structure as most of the physical entities generated from the CAD model or the mesh generator contain it as a basic attribute. Figure 5.21 shows the interface for the KMNode class.
class KMSpaceVector
{
    public:
    KMSpaceVector();
    KMSpaceVector(double, double, double);
    ~KMSpaceVector();
    void SetXi(const double&);
    void SetYj(const double&);
    void SetZk(const double&);
    void SetCoord(const double&, const double&, const double&);
    double GetXi(void) const;
    double GetYj(void) const;
    double GetZk(void) const;
    void GetCoord(double&, double&, double&);
    KMSpaceVector& operator = (const KMSpaceVector&);
    KMSpaceVector operator + (const KMSpaceVector&) const;
    KMSpaceVector operator - (const KMSpaceVector&) const;
    bool operator == (const KMSpaceVector&) const;
    double DotProduct(const KMSpaceVector&) const;
    KMSpaceVector CrossProduct(const KMSpaceVector&) const;
    KMSpaceVector UnitVector(void) const;
    KMSpaceVector MidPoint(const KMSpaceVector&, const KMSpaceVector&) const;
    KMSpaceVector MultiplyScalar(const double &Factor)const;
    double LengthVector(void) const;
    double FindAreaTriangle(const KMSpaceVector&,
                           const KMSpaceVector&) const;
    double ShapeQualityTriangle(const KMSpaceVector&,
                                const KMSpaceVector &B) const;
};

Figure 5.20: Interface for the KMSpaceVector class

class KMNode:public KMSpaceVector
{
    public:
    KMNode();
    KMNode(int, double, double, double);
    void SetNodeNo(const int &);
    int GetNodeNo(void) const;
    void PrintSelfS2KFormat(std::ofstream&);
};

Figure 5.21: Interface for the KMNode class

5.4.5 Line Segment

The line segment class called KMLineSegment is used to define attributes and methods for line segments in space. Line segment operations become very useful in
meshing a domain using the advancing front method. An important behavior of this class is that the objects of this class can subdivide themselves to obtain a list of subdivided line segments. Other behaviors include orienting line segment with respect to a point in plane and a normal vector, finding intersection type and intersection points between two line segments, etc. The interface for the _KMLineSegment_ class is shown in Figure 5.22.

```cpp
class KMLineSegment
{
public:
    KMLineSegment();
    KMLineSegment(KMNode*, KMNode*, int);
    ~KMLineSegment();
    void SetINode(KMNode*);
    void SetJNode(KMNode*);
    void SetIJ(void);
    void SetSegNo(const int&);
    KMNode* GetINode(void) const;
    KMNode* GetJNode(void) const;
    KMSpaceVector GetIJ(void) const;
    int GetSegNo(void) const;
    double GetLength(void) const;
friend bool operator == (const KMLineSegment&, const KMLineSegment&);
friend bool operator <  (const KMLineSegment&, const KMLineSegment&);
friend bool operator >  (const KMLineSegment&, const KMLineSegment&);
KMLineSegment& operator =  (const KMLineSegment&);
KMLineSegment& OrientLineSegment(const KMSpaceVector&, const KMSpaceVector&, const char&);
KMLineSegment& SwapIJNode(void);
KMLineSegment& BisectLine(const KMSpaceVector&, const double&) const;
KMNode* PositiveArea(const KMLineSegment&, const KMSpaceVector&) const;
KMSpaceVector FindIntersectionPoint(const KMLineSegment&, const char&) const;
KMSpaceVector MidPoint(void) const;
char FindIntersectionType(const KMLineSegment&) const;
double ShortestDistance(const KMSpaceVector&) const;
int GetNumberOfSubdivisons(const double &Factor) const;
void SubdivideSelf(const int&, std::deque<KMLineSegment*>&);
};
```

Figure 5.22: Interface for the KMLineSegment class
5.4.6 Restraint, Spring, Shell, Frame, Non-Linear Link and Nodal Load

An instance of these classes defines an object which has attributes required by SAP2000 version 7.x input file. These classes provide the PrintSelfS2KFormat method to print the information as per the .S2K format. The interfaces forKMRestraint, KMSpring, KMShell, KMFrame, KMNonLinearLink and KMNodalLoad classes are shown in Figures 5.23, 5.24, 5.25, 5.26, 5.27 and 5.28, respectively.

```cpp
class KMRestraint
{
 public:
    KMRestraint();
    KMRestraint(KMNode*, bool*);
    ~KMRestraint();
    void SetRestrainedNode(KMNode*);
    void SetRestraints(bool*);
    KMNode* GetRestrainedNode(void) const;
    bool* GetRestraints(void);
    void PrintSelfS2Kformat(std::ofstream&);
};
```

Figure 5.23: Interface for the KMRestraint class

```cpp
class KMSpring
{
 public:
    KMSpring();
    KMSpring(KMNode*, double*);
    ~KMSpring();
    void SetSpringNode(KMNode*);
    void SetSpringStiffnesses(double*);
    KMNode* GetSpringNode(void) const;
    double* GetSpringStiffnesses(void);
    void PrintSelfS2Kformat(std::ofstream&);
};
```

Figure 5.24: Interface for the KMSpring class
class KMShell
{
public:
    KMShell();
    KMShell(int, std::deque<KMNode*>, std::string);
    ~KMShell();
    void SetShellNo(const int&);
    void SetNodes(const std::deque<KMNode*>&);
    void SetSectionName(const std::string&);
    int GetShellNo(void) const;
    void GetNodes(std::deque<KMNode*>&);
    std::string GetSectionName(void) const;
    void PrintSelfS2Kformat(std::ofstream&);
};

Figure 5.25: Interface for the KMShell class

class KMFrame
{
public:
    KMFrame();
    KMFrame(int,KMNode*,KMNode*,std::string,int,double);
    ~KMFrame();
    void SetElementID(const int&);
    void SetINode(KMNode*);
    void SetJNode(KMNode*);
    void SetSectionName(const std::string&);
    void SetNumberofSegments(const int&);
    void SetAngle(const double&);
    int GetElementID(void) const;
    KMNode* GetINode(void) const;
    KMNode* GetJNode(void) const;
    std::string GetSectionName(void) const;
    int GetNumberofSegments(void) const;
    double GetAngle(void) const;
    void PrintSelfS2KFormat(std::ofstream&);
};

Figure 5.26: Interface for the KMFrame class
5.4.7 Material and Sections (Shell, Non-Linear Link and Frame)

An instance of these classes defines an object which has attributes required by the SAP2000 version 7.x input file. These classes provide PrintSelfS2KFormat method to print the information as per the .S2K format. The KMSection class is a base class for
*KMShellSection, KMNonLinearLinkSection* and *KMFrameSection*. The interfaces for these classes are shown in Figures 5.29, 5.30, 5.31, 5.32 and 5.34, respectively.

```cpp
class KMMaterial
{
public:
    KMMaterial();
    KMMaterial(std::string,double,double,double,double,double,double,double);
    ~KMMaterial();
    void SetName(const std::string &Name);
    void SetModE(const double &EModulus);
    void SetPoisson(const double &Poisson);
    void SetAlpha(const double &Alpha);
    void SetMassPerUnitVolume(const double &MassPerUnitVolume);
    void SetWeightPerUnitVolume(const double &);
    void SetEMultiplier(const double &);
    void SetSMultiplier(const double &);
    std::string GetName(void) const;
    double GetModE(void) const;
    double GetPoisson(void) const;
    double GetAlpha(void) const;
    double GetMassPerUnitVolume(void) const;
    double GetWeightPerUnitVolume(void) const;
    double GetEMultiplier(void) const;
    double GetSMultiplier(void) const;
    void PrintSelfS2KFormat(std::ofstream&);
};
```

**Figure 5.29: Interface for the KMMaterial class**

```cpp
class KMSection
{
public:
    KMSection();
    virtual ~KMSection();
    virtual void SetSectionName(const std::string&);
    virtual void SetMaterialName(const std::string&);
    virtual std::string GetSectionName(void) const;
    virtual std::string GetMaterialName(void) const;
    virtual void PrintSelfS2KFormat(std::ofstream&);
};
```

**Figure 5.30: Interface for the KMSection class**
class KMFrameSection: public KMSection
{
    public:
        KMFrameSection();
        KMFrameSection(std::string std::string, std::string double, double, double, double, double, double, double);
        ~KMFrameSection();
        void SetArea(double &A);
        void SetI2(double &I2);
        void SetI3(double &I3);
        void SetS2(double &S2);
        void SetS3(double &S3);
        void SetJ(double &J);
        void SetZ2(double &Z2);
        void SetZ3(double &Z3);
        double GetArea(void) const;
        double GetI2(void) const;
        double GetI3(void) const;
        double GetS2(void) const;
        double GetS3(void) const;
        double GetJ(void) const;
        double GetZ2(void) const;
        double GetZ3(void) const;
        void PrintSelfS2KFormat(std::ofstream&);
};

Figure 5.31: Interface for the KMFrameSection class

class KMShellSection: public KMSection
{
    public:
        KMShellSection();
        KMShellSection(std::string std::string, std::string std::string,
                       std::string double, double, double);
        ~KMShellSection();
        void SetElementType(const std::string &ElementType);
        void SetThicknessType(const std::string &ThicknessType);
        void SetMaterialAngle(const double &MaterialAngle);
        void SetThicknessMemb(const double &ThicknessMemb);
        void SetThicknessBend(const double &ThicknessBend);
        std::string GetElementType(void) const;
        std::string GetThicknessType(void) const;
        double GetMaterialAngle(void) const;
        double GetThicknessMemb(void) const;
        double GetThicknessBend(void) const;
        void PrintSelfS2KFormat(std::ofstream&);
};

Figure 5.32: Interface for the KMShellSection class
class KMNonLinearLinkSection: public KMSection
{
public:
    KMNonLinearLinkSection();
    KMNonLinearLinkSection(std::string, std::string, double*, double*, double, double);
    ~KMNonLinearLinkSection();
    void SetSectionType(const std::string &SectionType);
    void SetLinearStiff(double *Stiff);
    void SetLinearDamp(double *Damp);
    void SetDJ2(const double &DJ2);
    void SetDJ3(const double &DJ3);
    std::string GetSectionType(void) const;
    double* GetLinearStiff(void);
    double* GetLinearDamp(void);
    double GetDJ2(void) const;
    double GetDJ3(void) const;
    void PrintSelfS2KFormat(std::ofstream &);
};

Figure 5.33: Interface for the KMNonLinearLinkSection class

The input model for the program is required in .txt format as mentioned above. A detailed discussion of the required input format is made in Appendix A. Appendix A also presents a tutorial depicting the complete finite element models as viewed in SAP 2000 and some example input files. The detail for running the WoodFrameMesh program executable and its limitations are also discussed in the Appendix.