The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry

Robert Haimes*
Aerospace Computational Design Laboratory
Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139

John F. Dannenhoffer, III†
Aerospace Computational Methods Laboratory
Syracuse University, Syracuse, New York, 13244

Within the multi-disciplinary analysis and optimization community, there is a strong need for browser-based tools that provide users with the ability to visualize and interact with complex three-dimensional configurations. This need is particularly acute when the designs involve shape- and/or feature-based optimizations. Described herein is a family of open-sources software products that provides such a capability. At the top is a browser-based system, called the Engineering Sketch Pad (ESP), which provides the user the ability to interact with a configuration by building and/or modifying the design parameters and *feature tree* that define the configuration. ESP is built both upon the WebViewer (which is a WebGL-based visualizer for three-dimensional configurations and data) and upon OpenCSM (which is a constructive solid modeler; it in turn is built upon the EGADS and OpenCASCADE systems). Each of these open-source software components are described as well as the interactions amongst them.

I. Introduction

When analyzing (or designing/optimizing) some physical object that will ultimately be manufactured, it is common practice to create at least two different models: a geometric definition in a conceptual design tool and then later a fully realizable 3D representation in a CAD or CAD-like system. Generally great care is taken in ensuring that these representations are close enough to each other so that what is built is the same as what was designed. This care generally requires a large amount of time (and human intervention), making automation of the process extremely difficult, if not impossible, especially in a Multi-Disciplinary Analysis and Optimization (MDAO) environment.

The most common method for transferring information amongst the various models is via file standards. The first such standard commonly used was the IGES file format which contains data that is defined as disjoint and unconnected surfaces and curves; that is, it only contains geometry with no notion of topology. Topology, in this context, is the hierarchy and connectivity of the various geometric elements. Since 3D meshing software ultimately requires a closed watertight model, much effort is therefore needed to take the geometric data, trim the curves and surfaces, and then deduce the topology. STEP, a more complete

*Principal Research Engineer, Department of Aeronautics & Astronautics, AIAA Member.
†Associate Professor, Mechanical and Aerospace Engineering, AIAA Associate Fellow.

file standard, supports the transmittal of topology as well as geometry so that a Boundary Representation (BRep) can be built. This is therefore the preferable file type to hold geometric data. Surprisingly, this format is seldom used in practice, probably due to the fact that constructing a STEP reader is complex and it requires a complete solid modeling geometry kernel to deal with the data.

Even with the above, most organizations have often found it difficult to seamlessly transfer the geometry from conceptual design systems to realizable representations. To alleviate this problem (and those associated with transmitting geometry via a standard file), APIs that couple directly with the source of the geometry can be utilized. One clear advantage to this approach is that the geometry never needs to be translated and hence remains simpler and closed (to within the modeler's tolerance). Also a geometry system that can be used at a conceptual level and can continue to be leveraged through preliminary design has obvious advantages.

All the above discussion focuses on methods for dealing with a configuration once it has been defined; such a view is sufficient if one is only concerned with analysis. But for design, one must also deal with the process by which the configuration is defined. In parametric CAD systems, this configuration definition is done through a master model that consists of both a build recipe (called the *feature tree*) and a set of driving parameters. This information (the *design intent*) must be made available to the MDAO process since it defines the design space and informs how to build and optimize the configuration. Most commercial CAD systems hold this information in proprietary file formats that cannot easily be read or modified by outside programs.

Developed herein is an integrated software system that solves the above issues by providing the tools to generate various representations of a design (either multi-fidelity or multi-disciplinary, or both) from a single master model.

As described below, the top of the software system is the Engineering Sketch Pad (`ESP`), which is a fully-parametric, feature-based solid-modeling system that is web-enabled. `ESP` is built both upon the `WebViewer` (which is a WebGL-based visualizer for three-dimensional configurations and data) and upon `OpenCSM` (which is a constructive solid modeler that is in turn built upon `EGADS` and OpenCASCADE). All of this software is open-sourced and freely available without licensing restrictions.

The sections that follow describe each of these systems from the bottom up. It starts with a description of `EGADS` (the Engineering Geometry Aerospace Design System) and OpenCASCADE. It then describes `OpenCSM` (the open-sources constructive solid modeler). Then the paper transitions to the `WebViewer` with its reliance on WebGL and WebSockets. Finally all this is drawn together in a section that describes `ESP` itself.

II. Engineering Geometry Aerospace Design System

A major objective in setting up geometry for Multidisciplinary Design, Analysis and Optimization (MDAO) is the ability to define a parametric suite that is appropriate for both low- and high-fidelity analyses. In this way, the suite of parameters can be driven through a geometric modeler to realize an instance of the model in a manner commensurate with the analysis at hand. This multi-fidelity geometry perspective can then support the entire MDAO process from conceptual to detail design in a seamless manner. Another important objective for any geometry modeling subsystem is the ability to support different disciplines such as structural and aerodynamic analyses from the same geometric model.

It can be argued that Constructive Solid Geometry (CSG) is the natural foundation for attaining these goals. Two different approaches which can employ CSG at low level are considered: 1) CAD and CAD-like systems and their feature-based view of construction, and 2) bottom-up methods, which generate solid components. Although bottom-up methods do not have the turn-key features of commercial CAD systems, it is clear that their flexibility and potential open nature is an advantage in the long term, especially if geometric design-gradient information is required for optimization.

Any CSG manipulation of geometry requires a solid-modeling geometry kernel. Unfortunately, the construction of a solid modeling geometry kernel is a daunting task, but clearly a prerequisite. This problem can be mitigated by the use of OpenCASCADE,[1] a fully functional open-source solid modeling geometry

kernel, that has the following characteristics:

- Support for manifold and non-manifold geometry.
- Has the ability to perform bottom-up construction.
- Has both CSG operations and other abstract feature-like construction methods.
- Can read and write IGES, STEP and native file formats.
- Is a fully Object-Oriented C++ API with about 17,000 methods in 2+ million lines of code!

The last point is an obvious problem. Both the C++ nature of OpenCASCADE and the level of programming complexity with the huge suite of methods makes the use of OpenCASCADE rather a difficult undertaking. Its lack of documentation adds to the enormous task of understanding this large, but capable software suite.

To realize the MDAO objectives via the bottom-up (potentially mixed with a top-down) approach, a new software suite, the Engineering Geometry Aerospace Design System (`EGADS`),[2] has been developed and described below. It is a relatively simple and compact open-source object-based API built on top of the extensive OpenCASCADE solid-modeling kernel. In a real sense this is a simplification and rationalization of OpenCASCADE, where the functionality is focused on building multi-fidelity, parametric, geometric models for design.

`EGADS` is object-based but functionally procedural. The `EGADS` routines implement relatively high-level operations which insulate the user from OpenCASCADE's size and complexity, and for maximum flexibility can be driven by either C, C++, and/or FORTRAN user applications.

A. Boundary Representations

The concept of a Boundary Representation (BRep) is an object that holds both the topological entities and the geometric components that comprise the result of building a model. The BRep provides a unambiguous connected collection of geometric entities that when looked from the whole can uniquely separate regions of space from one another. This is critical for automatically supporting 3D meshing. This hierarchy can be seen in Table 1.

Table 1. `EGADS` Topological Entities.

EGADS Topological Object	OpenCASCADE term	Geometric Object
Model	Compound Shape	
Body	Solid (or other Shapes)	
Shell		
Face		Surface
Loop	Wire	* *see note*
Edge		Curve
Node	Vertex	[point]

*note: Loops may be geometry-free or have associated PCurves (one for each Edge) and the surface where the PCurves reside. A PCurve is the $[u, v]$ curve on the surface.

In general, topological entities lower in Table 1 bound those entities directly above:

- **Nodes**. The simplest entity, which is the topological equivalent to a point in 3 space.
- **Edges**. Most Edges have an underlying 3D curve that is bounded by 2 Nodes. The first Node is at t_{min} and the last is at t_{max}. An Edge can be *degenerate*, which coincides with a collapse of a surface's parameterization.

- **Loops**. A Loop without a surface reference is an ordered collection of Edges with corresponding senses; in this case no Edges may be *degenerate*. A Loop with a surface reference also contains PCurves, each associated with the Edge listed. The PCurve provides the mapping from the Edge's parametric coordinate (t) to the surface's parametric coordinates, $[u, v]$. Here *degenerate* Edges mark placeholders so that the PCurve can properly bound the surface's parametric mapping. It should be noted that an Edge can be found in a Loop twice (once in the positive orientation and the other negative) when found in a SolidBody. This even occurs for closed surfaces such as cylinders. For this situation there are two different PCurves, each representing the different limits of the periodic nature of the surface. The Loop may be of the *open* or *closed* (when it ends at the Node where it started).

- **Faces**. A Face refers to a surface that is bounded by one or more *closed* Loops. There must be one *outer* Loop and all the others are *inner* Loops (or holes). The reference geometry of the Loops must match the Face's surface. Attributes on Faces are robustly tracked through all high-level (*feature*-based) construction. For example when using the solid Boolean operators, the resultant BRep(s) have the Face attributes from the source bodies. If all Faces were marked in the input BReps, then all Faces in the output will also be marked.

- **Shells**. Shells are simply a collection of orientated Faces. This represents a *closed* shell if all Edges found in the Face's Loops are accounted for twice (unless *degenerate*).

- **Bodies**. This represents a functional aggregation of entities that can be used as the group. There are four types in `EGADS`:

  - WireBody – A single Loop which can be *open* or *closed*. These can be used as input to Extrude, Revolve and Lofting operations where the result is a SheetBody.
  - FaceBody – A single Face which also can be used as input to Extrude, Revolve and Lofting operations where the result is a SolidBody.
  - SheetBody – One or more Shells that can be either *open* or *closed*. Any non-manifold topology above a FaceBody is also placed in this category.
  - SolidBody – One or more *closed* Shells. There is always a single *outer* Shell with any number of *inner* Shells that represent the removal of material from the interior of the Solid.

  All but the SolidBody refer to non-manifold BReps.

- **Models**. This is the top-level `EGADS` container which can hold any number of Bodies. This is what the system reads and writes to disk (which includes the import and export of IGES and STEP file formats). In order to deal with fully-connected non-manifold entities in the "Body" context described above, it is appropriate to have multiple Bodies within a single Model that share topological objects. For example a SolidBody that reflects the OML of a aircraft could share the wing trailing-edge Edges with a SheetBody that represents the wake surface. There is an `EGADS` function that allows for the testing of *equivalency* of objects so that the wake surface could, in a sense, be reconnected to the manifold representation of the aircraft.

B. Bottom-up Build

Performing this type of geometric construction starts by building topologic objects at the bottom of Table 1 (hence the name). Nodes are required before Edges can be constructed because the Edge definition requires the bounding Nodes. Building in this manner produces the unambiguous definition required by the BRep but requires a lot of effort; this is because unlike data from IGES files, entities are generally connected in `EGADS`.

The `EGADS` API has been designed to be used by applications that support parametric builds of simpler components and then can assemble them (like aircraft conceptual design geometry tools). `EGADS` integration with these tools can be fairly simple by taking the parametrically built components (i.e., wings, fuselages,

tails, nacelles, and etc.) and using the bottom-up functions to construct the topology to *close* the geometry at the SolidBody level. At this point the component could be written out (in a file format that supports solids) or used by the higher-level CSG operations (such as the solid Boolean operators and others discussed below) to continue the construction towards more realistic and complex configurations.

C. CSG Style Build

Parametric CAD has a foundation based on Constructive Solid Geometry (CSG) concepts. In this way parts of arbitrary complexity can be generated as collections of simply abstracted operations ("features"), and at any step the resultant geometry is a SolidBody and therefore suitable for high-fidelity analysis. Furthermore, 3D meshes can be generated in an automated manner making this kind of geometry generation perfect for design settings. These favorable characteristics stem primarily from the use of solid models.

`EGADS` provides the following CSG-like functions:

- **Simple Solid Creation**. Makes a simple SolidBody. Can be either a box, cylinder, sphere, cone, or torus.
- **Solid Boolean Operators**. Performs the Solid Boolean Operations (SBOs) on the source SolidBody. This supports intersection (common), subtraction (difference), and union (fuse).
- **Offset & Hollow**. A hollowed solid is built from an initial SolidBody and a set of Faces that initially bound the solid. These Faces are removed and the remaining Faces become the walls of the hollowed solid with the specified thickness. If no Faces are specified, the solid is offset by the thickness.
- **Revolve**. Rotates the source about the axis through the angle specified. If the object is either a Loop or WireBody the result is a SheetBody. If the source is either a Face or FaceBody then the resultant object is a SolidBody.
- **Extrude**. Extrudes the source through the input direction at the specified distance. If the object is either a Loop or WireBody the result is a SheetBody. If the source is either a Face or FaceBody then the resultant object is a SolidBody.
- **Sweep**. Sweeps the source through the curve specified by the input Edge. If the object is either a Loop or WireBody the result is a SheetBody. If the source is either a Face or FaceBody then the resultant object is a SolidBody.
- **Loft**. Lofts the input objects (either Nodes or WireBodies) to create either a SheetBody or SolidBody.

D. Other *Feature*-based Operations

The following `EGADS` functions cannot be classified as CSG but are high-level *Body*-based operations:

- **Intersection**. Intersects the source Body (that is not a WireBody) with a surface in the form of a FaceBody or a single Face. The result is returned as a series of new Edges and the Faces where the Edges reside, and also as WireBodies.
- **Imprinting**. Edge/Face pairs (possibly from *Intersection*) can be applied to the source Body to change the topology by inserting the specified Edges.
- **Fillet & Chamfer**. Fillets or chamfers the Edges specified on the source Body object (either Sheet or Solid). The resultant body is the same as the source but with the specified Edges removed by surfaces that are tangent to the Faces (at the original Edge) or by planar surfaces, in the case of a chamfer.

E. Attribution

The ability to associate metadata to geometry is a critical part of any MDAO procedure. It allows for the ability to specify material properties or boundary conditions so that preparing for analysis can be automated. It can also facilitate matching surfaces that may be in a single BRep to those in another if built from the same source (such as in fluid/structure interaction).

In `EGADS` attributes can be associated with any object, but those attached to topologic entities are persistent. Any number of attributes can be associated with an `EGADS` object, but each must have a unique name. An attribute in `EGADS` can be assigned at anytime and consists of:

- **Name**. The unique name.
- **Type**. Either Integer, Real or String.
- **Length**. The number of entities of the specified type.
- **Data**. The data with the number of entries of the specified type.

All `EGADS` CSG and other *feature*-based operations track the attributes through to the result and reassign the data to the matching (or fragments of matching) Faces. For example, if all of the Faces were attributed in both SolidBodies that are unioned, then the resultant SolidBody will have all Faces attributed. The attributes are those found in the body that provided the source of the Face or Face fragment. In an operation that produces new Faces (such as *Filleting*), these newly created objects will have no attributes. This fact (assuming all other Faces have been attributed) can be used to determine the source operation which creates that specific geometry.

When SBOs are used in constructing multidisciplinary geometry, the association due to attribution is critical. One can determine shared geometry by querying the attributes even when Face objects differ and the Faces are trimmed in differing manners.

F. Tessellations

BReps have a tolerance that determines the meaning of "closure" for connected entities. This means that the Nodes that bound an Edge are probably not on the underlying curve. Also, Edges that bound a Face (through the Loops) do not necessarily sit exactly on the supporting surface. However, for a valid closed solid all that is required is that the bounding objects (Nodes/Edges) be within a specified tolerance of the higher dimensioned entity (Edges/Faces). Therefore, for any precision higher than the tolerance, gaps and overlaps may exist in the geometry definition. Note that the default tolerance is generally much larger than those associated with double precision floating-point arithmetic.

To deal with gaps and overlaps without an error being raised (that then requires intervention) most BRep-based applications must "fix" the geometry. This usually entails translating the geometric definition to another simpler representation where the bounding entities fall closer to the higher-dimensional object. This type of translation has a variety of side effects, including:

- **Inconsistency**. Since "fixing" the geometry changes it, the representation is different from that in the source system.
- **Errors**. By changing the geometry, unquantified errors are introduced into the process.
- **Complexity**. At times, additional Faces are required to close the model. There is no way to predict how many of these "sliver faces" may need to be introduced; moreover, slivers can cause significant problems for grid generators.
- **Not automatic**. There are always situations that cannot be healed in a *hands-off* manner. The requirement of user intervention is problematic for any fully automated process such as design optimization.

The perspective taken in EGADS is that the geometry is *truth* (in some regards) and should not be modified. Therefore fixing (or "healing") the model should not be required as part of the analysis procedure.

An API that only gives the programmer access to the BRep is a fairly difficult starting point for 3D meshing. The burden of deciphering the data and attempting to generate a discrete representation of the surfaces required for mesh generation is quite high. Fortunately, many grid generation systems used in CFD and other disciplines can use Stereo Lithography (STL) files as input. Combining a discretized view of the BRep as well as its geometry and topology can provide a complete, and easier to use, access point. A tessellation of the object that contains not only the mesh coordinates and supporting triangle indices but other data, such as the underlying surface parameters for each point, as well as the connectivity of the triangles, assists in traversing through and dissecting a complex part. This perspective was developed for CAPRI.[3, 4]

The discrete representation of a Body in EGADS is closed, even though the analytic definition (through the BRep) is open at machine precision. A Tessellation object can be generated for any Body with adjustable parameters. The resultant discretization can be examined a Face at a time. The triangulation of trimmed surfaces includes a single discretization of the Edges (for both Faces that are trimmed). Therefore for SolidBodies, when the Face triangulations are put together they form a completely closed manifold triangulation.

To support analyses that may need a quadrilateral discretization of the Body (for example: panel codes), EGADS provides non-automatic techniques to place patches on the BRep. A template scheme is used that patches a single Looped Face where 3 or 4 sides can be identified. The point counts on opposing sides are used (from the Body tessellation) to select the template and then filled with as many as 17 larger unstructured quadrilateral patches. These are then subdivided in a regular manner based on the point counts of the exposed sides. This is not automatic because there are situations that can not be templated due to an odd number of points on the larger Loop. EGADS provides functions that allow for the movement, addition and removal of Edge discretization points so that the quad-patching can provide surface meshing commensurate with the task at hand. This quadrilateral meshing scheme is a modification of the method used in CAPRI to generate anisotropic triangular meshes.[5]

There is also the ability to generate tessellations for geometry alone. This untrimmed discretization is useful for viewing the data in EGADS so that orientation and extent can be examined in order to support BRep building.

III. The Open Source Constructive Solid Modeler

When building a configuration in a CAD system, one needs both a method for performing the necessary geometric calculations (such as intersections) and for describing the operations that need to be performed. The EGADS system (described above) provides the former; OpenCSM[6] provides the latter.

All modern CAD systems, such a CatiaV5, SolidWorks, UniGraphics, and Pro/ENGINEER, are based upon construction of geometry using a constructive solid modeling (CSG) approach. In these approaches, a model consists of two types of items:

1. A build recipe (sometimes called a *feature tree*) that describes the types of and order of operations that one must perform.

2. A set of parameters that influence the exact shape of objects created (and sometimes which part of the *feature tree* should be executed).

These systems then "execute" the build recipe to generate a boundary representation. Unfortunately, modern CAD systems are quite large and have several shortcomings that include:

- The build recipe is encoded in files with proprietary formats that are hard to generate and/or modify externally.

- The primitives are pre-defined and it is very difficult to create fully-parametric user-defined 3D shapes.

- The build process is not explicitly "differentiated", thus computing the gradients required by some optimizers is available only via finite differences.
- The systems are licensed in such a way that distributed, simultaneous execution on many computers can be cost prohibitive.

`OpenCSM` circumvents these difficulties by designing and building an open-source constructive solid modeler that gives the user access to the master model (design parameters and *feature tree*). It is built upon `EGADS`,[2] thereby providing simple access to the OpenCASCADE[1] geometry generation system.

A. Design Objectives

The design objectives for `OpenCSM` include:

- Provide an open-source alternative to CAD systems that can be used throughout the traditional phases of design (conceptual, preliminary and detailed).
- All software dependencies are open-source.
- Has a simple ASCII description file that can be created and/or modified by an individual (with a text editor) or any external program.
- Can be extensible via user-defined primitives to define custom parametric 3D shapes.
- Can directly provide sensitivities.

B. `OpenCSM` .csm File Description

The first major component of `OpenCSM` is the description language. One of the design objectives for `OpenCSM` was that the build recipe be described in an ASCII file that could easily be created and/or modified by an external method or program. (The next section gives an example of the .csm file for a simple configuration.)

When building a configuration, `OpenCSM` executes the build recipe using a stack-based process, where each operation consumes 0, 1, or 2 "Bodies" from the stack and produces 0 or 1 new "Body". Each of the *feature tree* branches is fully parametrized, so that configurations can be easily modified in a design setting.

The simple primitive Bodies that can be generated are the box, cylinder, cone, sphere, and torus. Other simple Bodies can be created by starting with a sketch (which is comprised of linear segments, circular arcs, and splines), and then extruding, lofting, or revolving into a SolidBody. `OpenCSM` also supports the application of fillets and chamfers to an existing Body.

Once Bodies are defined (and are held on the stack), they can be combined using the Boolean union (fusion), difference (subtract), and intersection (common) operators. Similarly the Body on the top of the stack can be transformed by applying a translation, rotation, or scaling operator.

When `OpenCSM` finishes its build process, all Bodies that remain on the stack are returned to the user.

C. `OpenCSM` API

The API of `OpenCSM` currently consists of 31 functions (which are callable from C). These functions can be broken into the following groups:

Loading and Building the Master Model:

- ocsmLoad — read a .csm file and create a model (parameters and *feature tree* branches)
- ocsmCopy — create a copy of a model
- ocsmSave — save a model to a .csm file
- ocsmBuild — execute the *feature tree* and create a group of Bodies

Interrogating and editing the parameters in a Model:

- ocsmGetPmtr — get the size of (rows and columns) and type of (external or internal) of a parameter
- ocsmNewPmtr — create a new external parameter
- ocsmGetValu, ocsmSetValu, ocsmSetDot — get/set the definition, value, and sensitivity of a parameter

Interrogating and editing the branches in a Model:

- ocsmGetBrch, ocsmSetBrch — get/set information associated with a *feature tree* branch
- ocsmGetArg, ocsmSetArg — get/set an argument associated with a particular branch
- ocsmNewBrch — add a branch to the end of the *feature tree*
- ocsmDelBrch — delete the last branch from the *feature tree*
- ocsmSuprBrch, ocsmResmBrch — suppress or resume (unsuppress) a *feature tree* branch
- ocsmGetAttr, ocsmRetAttr, ocsmSetAttr — get/return/set an attribute associated with a particular branch
- ocsmGetName, ocsmSetName — get/set the name of a particular branch

Interrogating the BRep:

- ocsmGetBody — to use the underlying `EGADS` evaluators

Utility Functions:

- ocsmVersion — identify the version of `OpenCSM` currently running
- ocsmSetOutLevel — define the level of user feedback that `OpenCSM` produces
- ocsmInfo — return number of branches, parameters, and Bodies in a model
- ocsmPrintPmtrs — create "printable" listing of parameters
- ocsmPrintBrchs — create "printable" listing of branches
- ocsmPrintBodys — create "printable" listing of Bodies
- ocsmFree — free up all storage associated with a model

D. `OpenCSM` Example

To demonstrate the above, the "bottle" configuration from the OpenCASCADE tutorial has been defined in a csm file, producing the configuration shown in Figure 1.

The csm file that produced the configuration is:

```
# design parameters
desPmtr     width       10.00 cm
desPmtr     depth        4.00 cm
desPmtr     height      15.00 cm
desPmtr     neckDiam     2.50 cm
desPmtr     neckHeight   3.00 cm
desPmtr     wall         0.20 cm
desPmtr     filRad1      0.25 cm
desPmtr     filRad2      0.10 cm

# basic bottle shape (filleted)
```

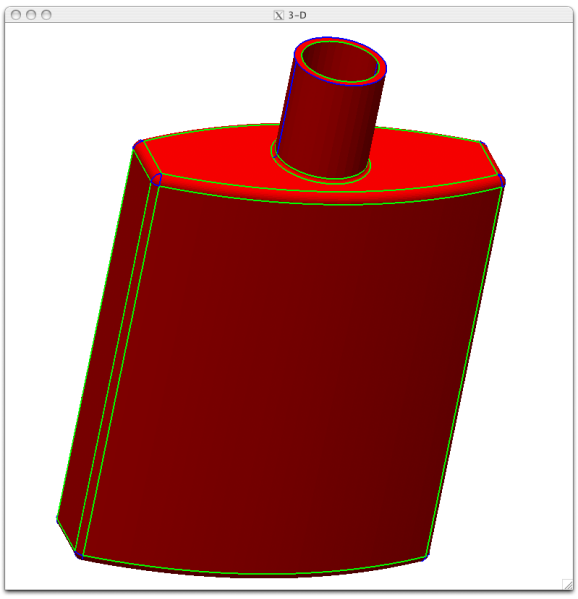

Figure 1. The OpenCASCADE tutorial "bottle" represented in `OpenCSM`.

```
set         baseHt   height-neckHeight
skbeg       -width/2 -depth/4 0
   cirarc   0        -depth/2 0  +width/2 -depth/4  0
   linseg   +width/2 +depth/4 0
   cirarc   0        +depth/2 0  -width/2 +depth/4  0
skend
extrude     0        0        baseHt
fillet      filRad1  0        0

# neck with a hole
set         holeBot height-neckHeight/2
cylinder    0   0   baseHt  0 0 height      neckDiam/2
cylinder    0   0   holeBot 0 0 height+wall neckDiam/2-wall
subtract

# join the neck to the bottle and apply a fillet at the union
union
fillet      filRad2  0        0
```

Note that the design parameters and their initial values are listed at the top of the file, followed by the build recipe. The `set` command specifies a *driven* variable. `skbeg` begins a *sketch* and sets the initial point, where `cirarc` specifies a circular arc starting at the previous point and going through the two specified locations. `linseg` creates a line segment from the previous point through the one specified as part of the command. `skend` ends the sketch and if open, closes the sketch by inserting a line segment. The *sketch* now contains the cross section for the bottle. The `OpenCSM` command `extrude` takes the *sketch* and extrudes it through the distance `baseHt` in a perpendicular direction to the *sketch*. The command `fillet` takes the newly exposed Edges and performs the fillet operation using the specified radius. The bottleneck is created by performing an SBO subtraction from two cylinders. And finally, the neck is fused (`union`) to the body and the newly created trimming curves are filleted using `filRad2` as the radius.

IV. The WebViewer

During the assessment of the building blocks used to construct engineering analysis software, there are two critical decisions that strongly effect the end-user experience. These are (1) the best use of 3D graphics to effectively enable both the pre- and post-processing phases and (2) the construction of the User Interface (UI). A poor decision for either can keep most users away from even the best-in-class solvers. Most current traditional analysis suites have selected some UI toolkit and usually have a 3D viewer component that renders imagery in OpenGL. The selection of the UI toolkit is the most important; an extraordinary amount of effort goes to constructing the UI (hence the selection of a toolkit – to reduce that effort and provide a consistent "look and feel"). But this is also a problem: it becomes difficult, if not impossible, to then change the "look and feel" and/or the underlying toolkit.

When looking at the entire suite of applications that may be run within a Multidisciplinary Design/Analysis and Optimization (MDAO) process, 3D viewing can be used in:

- **Geometry Construction**. Being able to examine the geometry being built is crucial during the parametric CAD construction phase. This sets up the global design space (by properly defining the parameters that drive the model). Specifying and examining the resulting instances at the bounds that the parameters can take ensures that the model can be regenerated throughout much of the design space.
- **Meshing**. Viewing the mesh for those *a priori* grid generation schemes is important in order to get a sense if the mesh is appropriate for the solver. Also some grid generators require a great deal of user interaction with the data to construct the mesh which is usually done within a UI with either 2D or 3D graphics.
- **Scientific Visualization**. The ability to effectively deal with the kinds of data that are output from the solver as they pass through the visualization pipeline is critical in understanding the data from a single analysis. This entails viewing data through the filters of cutting planes and surfaces, iso-surfaces, streamlines, and other visualization tools.
- **Multidimensional Design Space Exploration**. Having a way to interpret, and therefore understand, the decisions being made by an optimizer is important in getting a realistic optima. This can be done by decomposing the design space into something that can be viewed in 2 or 3 dimensions and then applying some of the scientific visualizations tools.

Current tools each have their own UI and their own viewing paradigms. That puts the burden on the user of each phase of a larger problem to learn and know how to traverse through a complex and differing suite of applications' "looks and feels".

A. In the Web Browser

The emerging technologies of WebGL[7] and WebSockets[8] have the potential to provide a platform for general Computer Aided Engineering (CAE) within the Web Browser. In addition, JavaScript (the programming language of Web Browsers) enables an interesting environment for building graphical UIs. Providing a Web-based "portal" to the entire MDAO process or any individual CAE component has the following advantages:

- **Computer Platform Independence**. Any workstation, laptop, or other computer that has a browser can interact with properly constructed HTML and JavaScript files to provide the interface into the application(s). There is no concern for operating system, revision, or native pointer size. All one needs to do is to be careful with using the suite of features found in all supported browsers. This is usually not a problem for Google Chrome, the Mozilla Browsers (FireFox and SeaMonkey), Apple Safari, but can sometimes be an issue for Internet Explorer.

- **Moves Traditional CAE to SOA**. This paradigm forces the separation of the analysis from its viewing and control and general interaction (through the UI). MVC (Model-View-Controller) architectures have been the appropriate software-design model for interactive applications for some time. Using the browser enforces this model in that the view and parts of the control have been separated and run in different address spaces. This also moves traditional applications to a client/server setting (which brings the programming model closer to Services Orientated Architecture – SOA).

- **Tablet Enabled**. Android powered tablets and iPads have browsers that are compatible with their computer counterparts. This means that these devices can be used in very much the same manner to view and control the suite of applications.

- **Cloud Ready**. Once the UI has be separated from the computer(s) that run the application suite, where the execution is performed becomes of little concern (except that we must be able to see the server on the net). Using some form of SOA, which includes the "cloud", is a natural extension of severing the MVC architecture and the ability to view and control remotely.

JavaScript is the interpretive language of Web Browsers used in order to create enhanced user interfaces and dynamic websites. It is not well named, in that it is not derivative of Java, but more like interpretive C with a number of interesting object-like extensions (which include automatic hashing of names). It is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. JavaScript is not explicitly multithreaded but the programmer can set up functional loops (which can be timed) that provide what appears to be multiple threads of execution. In this way rending can be performed, in an asynchronous manner, simultaneously with the data being accumulated (from a server) to adjust the next scene. This JavaScript feature (once understood) is quite flexible and can provide a high-performance platform that can effectively use the computing resources available.

Any sophisticated website probably uses JavaScript extensively in order to provide user interaction. An interesting part of the web is the ability to view the source of any page that the browser is displaying. That means that any UI that is found to be interesting, advanced, or just the correct way to do some task is readily available (truly open source). One can develop an "organic" UI that is not constrained by a traditional toolkit and can simply be enhanced by routinely *borrowing* from interesting websites.

B. WebGL

WebGL[7] is a cross-platform, royalty-free web standard for a low-level 3D-graphics JavaScript API that is based on OpenGL ES 2.0 (the subset of OpenGL available on mobile devices). WebGL is for rendering interactive 3D graphics and 2D graphics within any compatible web browser, without the use of plug-ins. WebGL is completely integrated into the web standards of the browser allowing GPU accelerated usage as part of the web page canvas. WebGL renderings can be mixed with other HTML elements and composited with other parts of the page or page background. Developers familiar with OpenGL ES 2.0 will recognize WebGL as a shader-based API using GLSL (the OpenGL Shading Language – C-like language used to write the shaders).

WebGL is integrated with Google Chrome, the Mozilla Browsers (FireFox and SeaMonkey), Apple Safari (at Rev 6.0 and higher) but is not included with Internet Explorer. Microsoft has exclaimed that WebGL is a security concern because it talks to the graphics hardware directly and will probably not be including this technology with Internet Explorer in the near future.

C. WebSockets

WebSockets[8] is a web technology providing full-duplex communications channels over a single TCP/IP connection. The WebSockets API has become part of the new HTML5 standard. JavaScript *typed* arrays are supported, which is important for WebGL because it requires strong typing in dealing with the Shaders.

The HTML5 WebSockets specification defines a JavaScript API that enables web pages to use the WebSockets protocol for two-way communication with a remote host. It defines a full-duplex communication

channel that operates through a single socket over the Web. HTML5 WebSockets provides an enormous reduction in unnecessary network traffic and latency compared to the unscalable polling and long-polling solutions that were used to simulate a full-duplex connection by maintaining two connections.

WebSockets has been implemented in web browsers and web servers, but it can be used by any client or server application. The WebSocket Protocol is an independent TCP/IP-based protocol and may be further enhanced by using multiple internal "protocols" (either binary or text) over the same port.

D. wv: The WebViewer Client

The goal of this software module is to generate a JavaScript component that can be used for the types of 3D viewing required by any part of the MDAO process (as described above). It uses WebGL for the rendering and WebSockets to move data back and forth between the browser and (the application acting as) the server. Two WebSocket internal 'protocols' are used:

- **Binary data**. Data of this type goes only from server to web browser and contains the data to be rendered in a specific form as described below.
- **UI text data**. This bi-directional communication is for passing messages between the browser and server having to do with the UI. The server must be written in a manner that properly responds to this communication, usually with messages back to the browser that provide the requested information.

In order to make this software reusable it is necessary to separate the 3D viewer from parts of the code that will drive the UI. This allows for the UI to be easily customized without having to modify the rendering aspects of this WebViewer. It is imagined that the CAD-like component will have a different UI than the scientific visualization component, but can share the 3D viewer and have a consistent "look and feel". Access is made available at specific times during the rendering sequence through JavaScript call-backs. These can simply be added to the viewer program by specifying the JavaScript files to load in the initiating HTML file which contain these functions:

- `wvInitUI()`. This call-back is invoked once, at program initialization, and should be used to initialize the variables used of the UI and their state.
- `wvUpdateUI()`. Called every iteration through the rendering loop so that the state of the UI can be adjusted based on the events that may have occurred.
- `wvUpdateView()`. Allows for the adjustment of the view matrix (again potentially from UI events that may have occurred) before the scene is rendered again.
- `wvUpdateCanvas(gl)`. Facilitates the customization of what is rendered in the 3D view by additional WebGL calls not found in the data sent from the server.
- `wvServerMessage(text)`. Called when an ASCII text message has been received from the server (in the UI text protocol).

An additional function that is part of **wv** should be used to communicate UI text data back to the server:

- `g.socketUt.send(text)`. This should obviously be the routine invoked to communicate UI information to the server. Note that this function can be used from within any **wv** call-back.

1. Vertex Buffer Objects

Because JavaScript is interpretive, traditional vertex-based immediate-mode OpenGL-like rendering is not implemented in WebGL. This was a good design decision because looping over every vertex for every triangle in the scene and specifying position, color, and optionally a normal would not provide good graphics performance. An early extension to OpenGL was the concept of Vertex Buffer Objects (VBOs). These are

arrays of data which refer to a component of the vertices to render. For example: a VBO floating point array may contain the 3D coordinates of the points that can be assembled into a series of triangles.

Using VBOs allows WebGL to get good performance for a series of graphics primitives by avoiding looping over the vertices. WebGL has functions that take *typed* arrays in order to build the VBOs in the graphic card's memory. This has the added advantage that once no longer referenced, the *typed* arrays are cleaned up during JavaScript garbage collection and therefore do not end up bloating the browser's memory. The VBO components supported by **wv** are:

- **Vertices**. These are the series of 3D coordinates (3 floating point numbers per vertex). They must be packed into a JavaScript `Float32Array`.
- **Indices**. This optional component specifies an indexed view into the *Vertices* component. Without this component, the coordinates are grouped together based on the primitive (that is, if triangles every 3 vertices make up a triangle); with Indices every 3 indices make a triangle where each is an index into the Vertices component. This is a JavaScript `Uint16Array` (unsigned short) in **wv**. Note that this limits the size of Vertices to 65536 when used. Therefore when using this component for long VBOs, the data must be striped into appropriate sized subcomponents.
- **Colors**. This optional component specifies the colors (RGB) to be used to render data at the vertices. Linear color shading is applied between vertices based on the primitive. When specified, this must be a JavaScript `Uint8Array` (unsigned char) which has the length of 3 times the number of Vertices (for red, green, and blue bytes). If not included the graphics primitive is rendered in the default color.
- **Normals**. This optional component is only required for triangles and assigns the normal to each vertex for the lighting calculations. Like Vertices, this is a series of 3 floating point numbers and must be packed into a JavaScript `Float32Array` with the same length as Vertices. If this component is not specified, then the default normal is used (this is efficient for rendering planar information).

Three basic graphic primitives are supported in **wv**, each which specify how the VBO data is interpreted for drawing. These include:

- **0D − Points**. Each vertex generates a point using the optional specified color. The point size (in pixels) can be set for the collection of points.
- **1D − Lines**. Every other vertex specifies a disjoint straight line segment. Colors are interpolated from the optional color component. Constant coloring can be specified per segment by not indexing and setting each pair of colors to the same RGB. Constant coloring for the entire primitive can be accomplished by not using the color component and defaulting to the single color. The collection of lines can be rendered with a specific line width. Points can be associated with the suite of lines in order to depict the position of the data.
- **2D − Triangles**. Every 3 Vertices or every 3 Indices refer to a triangle. The orientation should be consistently right-handed (in particular if back-face coloring is used). If the entire collections of triangles does not refer to a planar representation of data, the Normals must be specified in order to get the lighting to appear correct. Lines and points can also be displayed and are useful for showing how the surface(s) were discretized.

Note that there is only a single representation for each basic type. If the data is from a series of quadrilaterals (instead of triangles) the quads must be split into triangles and the appropriate VBO components filled for triangles. When specifying the lines associated with the triangles do not include the diagonal lines so that when the grid is shown it refers to the original data.

2. The Scene Graph

The binary data, which in general, comes from the server controls what can be seen in the 3D view. This scene graph consists of any number of graphic primitives. Each graphic Primitive has a unique name, dimensional type, series of possibly-striped VBOs, and other metadata. The server can control the data by sending either new or updated VBO components at any time to the active browser session. For every iteration through the rendering loop, the scene graph is updated from the server before the rendering is accomplished. This allows for such options as being able to perform scientific visualization on transient data. It can also promote the efficient use of the network. For example, the server can update just the Color VBO component when changing the color scale or colormap.

There are also messages from the server that allows for the deletion of graphics primitives if some part of the scene no longer exists.

3. Stateless Viewer

As describe above, the viewer is a slave to the data management specified by the server. In fact, except for what is maintained by the customized UI, the viewer is *stateless* but for the following two items:

1. **View Matrix**. The server never needs to know the current view matrix because rendering is completely local in the browser (though the custom UI code could transmit this data to the server using the UI text protocol). The view matrix is modified in the browser by the UI and set at the top of the rendering loop.
2. **Graphic Primitive Plotting Attributes**. Each member of the scene graph has an attribute bit-flag that specifies the options used to render the graphics primitive. This is initially set by the server, but is maintained in the browser based on modifications by the UI code driven by user interaction. These options include:
   - **Rendering on**. The graphics primitive is rendered, if the bit is turned off, then the primitive is overlooked during scene graph traversal.
   - **Transparent**. When rendered, if this bit is set, the primitive is made transparent.
   - **Shading**. When the primitive has an associated Color VBO component, this bit flags whether the drawing of the object uses the default color or is shaded via linear interpolation (in color-space).
   - **Orientation**. This bit has no effect on Points, but for Lines it flags drawing line "decorations" (usually arrow heads). For Triangles this specifies that the back face of the triangle should be drawn in the constant back-face color.
   - **Points**. This attribute bit indicates that the optional point Indices should be used to draw points associated with the Line or Triangle primitive.
   - **Lines**. This attribute bit flags plotting the optional line Indices in order to draw lines associated with Triangles. This is primarily useful in displaying the mesh on the surface.

   With the UI modifying the attribute bit-flag, the user can simply adjust the scene rendering locally.

4. Shaders

The designers of WebGL did a remarkable job in developing a concise JavaScript API for 3D drawing. In making key decisions they clearly went with functionality over mimicking the OpenGL API (as can be seen by not implementing the vertex level OpenGL functions and going for VBOs). It was decided that custom shaders (through a subset of GLSL) would be supported. If this is the case, then providing all of the OpenGL functions that perform lighting and allow for the specification of material properties (colors and shininess) are no longer necessary. This functionality can be written in the shader code itself. Unfortunately, this ends up putting quite a burden on the WebGL programmer. The programmer must understand the rendering pipeline and be able to program in this C-like language (with no real ability to debug).

Two shaders are required for any WebGL application:

1. Vertex Shader. This GPU program is called for every vertex in the VBO in order to specify its position in screen coordinates, its color, and optionally its normal (used for lighting). It is this shader's responsibility to setup the vertex state so that when the individual pixels get *lit* they have the correct color through linear interpolation.

2. Fragment Shader. This GPU program is executed for each pixel generated from rasterizing the linear fragment of the primitive being rendered. It is the responsibility of this code to set the color at the pixel.

The shaders provided with **wv** support:

- Two-sided lighting. This lighting model ignores the sign of the dot-product between the lighting vector and the normal. This provides imagery where lighting gives clues as to the shapes of objects but does not provide dark regions (in the backs of facets away from the lighting source).
- Ambient & Diffuse. The lighting model is a simple sum of ambient and diffuse sources.
- Back-face coloring. An option exists that can show the back-face of a fragment in a different color than the front (right-handed).
- Coloring Options. Constant and/or linearly-interpolated color-space mapping can be applied to the primitives being rendered.
- Transparency. A simple, single level of transparency can be applied to the rendering of primitives.
- Effective Line Drawing. When drawing lines, the Z-screen value can be bumped toward the viewer's eye, so that lines appear well formed in the final image.

Other methods for rendering can be added to **wv** such as more complicated lighting, shading and texture mapping, but what is currently written supports most of the application areas envisioned for this software.

5. Picking & Locating

Being able to pick (select a member of the scene graph) or locate (return the 3D coordinates over which the cursor resides) are both important functions that a user may need to perform in order to interact with their data. These have been built into **wv** and can easily be accessed via the custom UI code. In both cases the rendering subsystem is put into a special mode and the scene graph is traversed. This mode tells the shaders not to do a normal rendering of color but to perform either picking or locating. When picking, the data encoded in the color bits during the execution of the Fragment Shader is an index into the current graphics primitive. When in locate mode, the data encoded into the color is the actual value of screen-Z.

When the rendering is complete the entire image contains either the data associated with the pixels primitive index or the screen-Z value closest to the viewer. The colormap is then read at the current cursor position and depending on the mode, either the index is decoded and reported or the floating point value for screen-Z is reconstructed. For locating, the cursor position relates to screen-X and screen-Y so that the 3D screen coordinates are now known. These coordinates are multiplied by the inverse of the view matrix to reconstruct the original XYZ coordinates.

The image is not rendered to the browser's canvas, but it is then cleared and rerendered using the normal color mode before the front and back buffers are swapped. The user may only notice slightly worse graphics performance when either picking and/or locating are on. Note that for this to function correctly, anti-aliasing must be turned off so that data at the fringe of the primitive's extent maintain their correct values.

E. The wv Server

A server is obviously required to communicate with the **wv** client running in a browser. The only requirement for the server is that it packages up messages in a manner that **wv** can understand and uses WebSockets

to perform the message passing. The data packaged up and sent over the binary protocol is essentially the VBO components. The browser then simply takes the block of memory and produces the VBO components directly without interrogating any of the data. This procedure allows for great performance.

Python has a number of packages that support WebSockets so one could easily construct a Python based server (as is being done by the OpenMDAO[9] Project), but for those who want the ability to put a simple server together that has few dependencies, a C/C++/FORTRAN API has been constructed based on the open source package *libwebsockets*. This server library is the easiest entry-point for the procedure of splitting off the CAE application's viewing and parts of the control of the MVC user interaction model. It also allows for the entire execution to appear to be initiated from a single point (as if it were a single application).

V. Engineering Sketch Pad

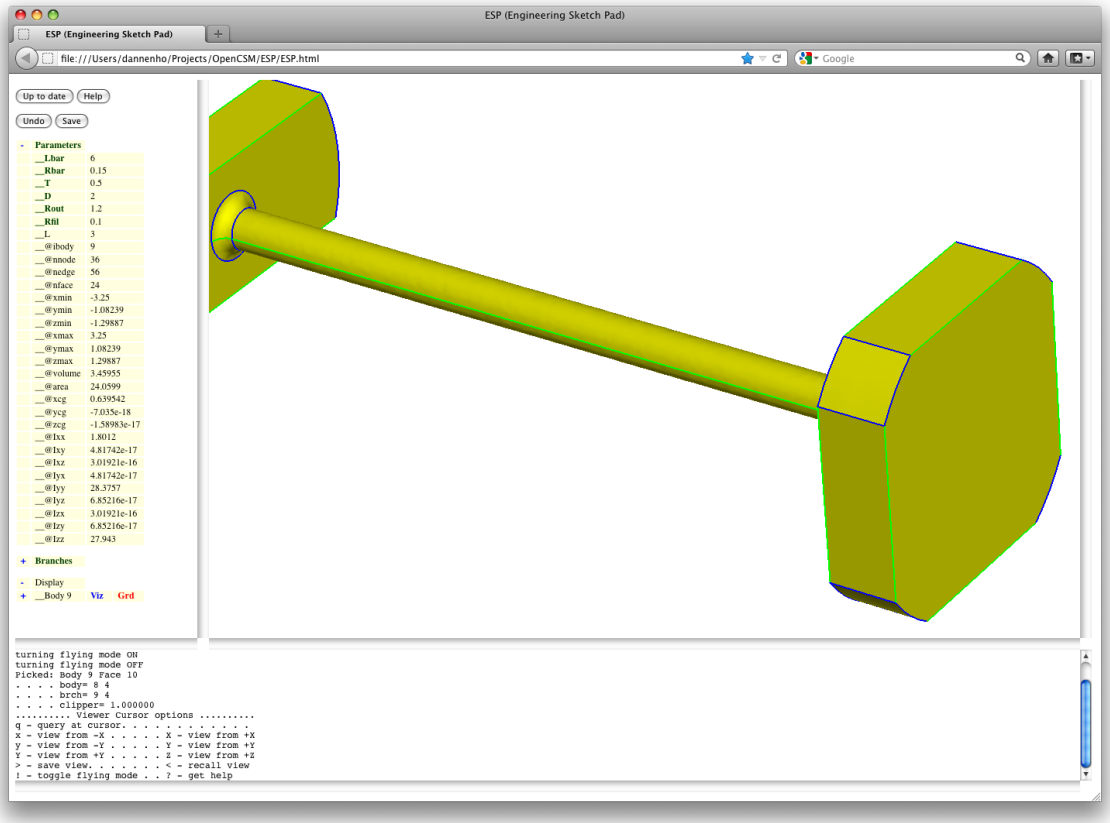

Figure 2. `ESP` main screen. The tree-window on the left shows the current parameters.

With `OpenCSM` (a "feature-based" solid-modeling geometry system) and **wv** (the library-like components to do 3D viewing and UI development), one can easily assemble these modules to perform tasks related to design. The Engineering Sketch Pad (`ESP`) is one example assemblage.

The design objectives for `ESP` are:

- Operate with any modern web browser.
- Provide the user the ability to easily modify both design parameters and suppression state of the various branches in the *feature tree*.
- Provide the user the ability to modify a *feature tree* by adding and/or deleting branches.

- Allow the user the ability to add, delete, and query attributes associated with the *feature tree* branches.
- Provide the user a graphical representation of the configuration, with the ability for querying parts of the BRep.
- Have the communication between the browser and the back-end server be as lightweight as possible.

The initial layout of `ESP` is shown in Fig. 2, it consists if a textual description of the configuration (on the left), a graphical representation (on the right), and a messages area that provides feedback to the user (at the bottom).

The textual description is contained in a tree-like widget that provides groups for the design parameters, branches (*feature tree*), and graphical display.

In order to edit a design parameter, the user clicks on the name of a parameter and a pop-up window is displayed that allow the user to modify the current value. New parameters can be added by clicking on the "Parameters" heading in the tree; the user is prompted for the new parameter name and its initial value.

Interactions with the *feature tree* are similar to the above. By contracting the "Parameters" and expanding the "Feature tree", one gets a display such as shown in Fig. 3.

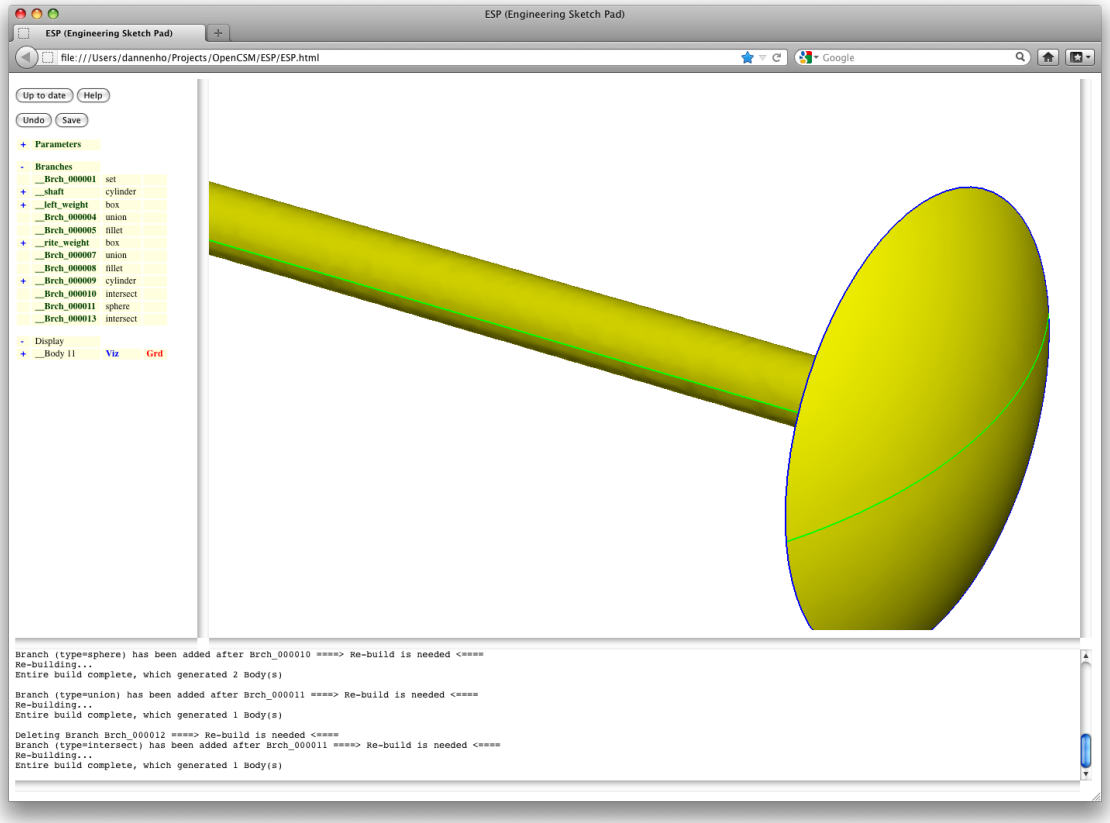

Figure 3. `ESP` main screen. The tree-window on the left shows the current branches (the *feature tree*).

By pressing the name of one of the features, one is given the options to edit the branch, add a new branch after it, delete the branch, add an attribute to the branch, or re-build the configuration up to and including the indicated branch. When adding or editing a branch, a pop-up, such as shown in Fig. 4 is presented to the user. The exact form of the pop-up window depends upon the type of branch being added or edited.

Figure 4. Pop-up that allows a user to edit a branch of the *feature tree*.

A. User Interface model

In order to maintain a high refresh rate, especially for large models, `ESP` has been designed using a traditional client-server model (as described above). In `ESP`, the client (web browser) is responsible for the View and part of the Controller of MVC:

- Performing rotations, translations, and zooming of the graphical image.
- Providing graphics-based "locating" and "querying" functions.
- Fielding the user's button presses.
- Posting pop-up menus and acquiring textual- or number-based inputs from the user.

The server is responsible for parts of the Controller and the Model portion of the MVC paradigm:

- Reading and writing all necessary files.
- Keeping track of the "current" parameter values and the *feature tree* and its suppression state.
- Building the current configuration (when requested).
- Generating the Vertex Buffer Objects that describe the configuration and sending them to the client.

Communication from the browser to the server is accomplished through small requests, such as "setPmtr; Length;1;1;3.75", which means set `Length(1,1)` to 3.75, or "build;5", which means build the configuration up to branch 5. These two requests are sent because of a button press by the user in the UI.

The browser also communicates with the server with requests such as "identify;", which the browser uses to know the types of requests that the server will be able to handle and "getPmtrs;", which the server issues every time it needs to update the current settings of the parameters for display in the UI.

The server responds to these requests with simple text-based messages, such as "identify;serveCSM;" (which indicates that the server's name is '`serveCSM`') or a JSON-based message such as "getPmtrs:<JSON>;", which actually builds the appropriate structure in the client's memory.

Through this design, the only communication between the browser and the server occurs when one is changing the model (either its design parameters or *feature tree*). This allows `ESP` to have a very low network footprint.

B. CAD: Cloud Aided Design

The entire interaction with ESP is web-based. Therefore it does not matter where the server components are executing as long as there is a network connection to the browser. A good user experience is guaranteed even with a slow network because all 3D viewing is local to the browser and the overall amount of network traffic has been minimized. As stated above, this then allows for the true use of the **Cloud** and ESP can be controlled via a traditional workstation or tablet (and even smart phones) because of the hardware support of OpenGL-ES and therefore WebGL.

VI. Conclusions

Three API components (EGADS, OpenCSM and WebViewer) have been assembled to create ESP. ESP is a feature-based solid-modeling system that is web-enabled and can be used with most modern web browsers. In many ways it mirrors the functionality of modern parametric commercial CAD systems (but clearly without the maturity).

The chief advantages of ESP over commercial CAD systems are:

- Captures the *design intent* in a simple, easy to read, nonproprietary text file that is easily modifiable by any other component in an MDAO environment.
- The ability to easily add customized features to the system. For example, it is quite difficult to generate something simple like a NACA 4−digit wing in a commercial CAD system. With the OpenCSM user-defined primitive, this only requires a small amount of bottom-up EGADS programming.
- Supports the generation of multiple models from the same parameterization. For example, one can easily generate beam, built-up element, or fully expressed solid model (or assembly) of an aircraft structural model. Simultaneously, one can use the same parameters to generate mid-surface aerodynamic models as well as full solid models for CFD analysis.
- Provides attribution at all levels, which is an essential capability when one wants to *connect* (in a multi-disciplinary way) the various parts of the various representations.
- Supports easy integration into a larger process. This includes geometry import/export using file standards (either discrete or analytic) and direct connections to a number of 3D grid generators. Also, the use of the appropriate API directly can remove the file standard limitation (i.e., the loss of critical information when the standard does not support the data).
- Easily used in distributed computing environments (such as HPC clusters) since it is open-source and is not encumbered with any licensing restrictions.
- Provides analytic parameter sensitivity (for much of the build), making it suitable for the gradient-based optimization processes that are frequently used in MDAO environments.

These API components are also being assembled into **GEM** (the **G**eometry **E**nvironment for **M**DAO). **GEM** is designed to plugin into multidisciplinary frameworks, specifically OpenMDAO.[9] **GEM**, also exposed as an API itself, has hooks which include a conservative data transfer interface based on using the geometry as the media for transfer and the ability to expose the design parameter sensitivities to the framework.

ESP along with OpenCSM, EGADS, and WebViewer are all open-source systems, which in turn are built upon only open-source components (OpenCASCADE, WebSockets, and WebGL). They are written in C, C++, JavaScript, and HTML5 and are currently licensed under LGPL v2.1. ESP can be downloaded from http://acdl.mit.edu/ESP.

Acknowledgements

This work was performed as part of NASA Cooperative Agreement NNX11AI66A "Geometry Interface for the NASA OpenMDAO Framework"; Christopher Heath (NASA-GRC) is the Technical Monitor.

References

[1] "OpenCASCADE", http://www.opencascade.org.

[2] Haimes, R, and Drela, M., "On the Construction of Aircraft Conceptual Geometry for High Fidelity Analysis and Design", AIAA-2012-0683, January 2012.

[3] Haimes, R. and Follen, G., "Computational Analysis PRogramming Interface", Proceedings of the 6^{th} International Conference on Numerical Grid Generation in Computational Field Simulations, July 1998.

[4] Haimes, R. and Aftosmis, M. J., "On Generating High Quality Water tight Triangulations Directly from CAD", Proceedings of the 8^{th} International Conference on Numerical Grid Generation in Computational Field Simulations, July 2002.

[5] Haimes, R. and Aftosmis, M. J., "Watertight Anisotropic Surface Meshing Using Quadrilateral Patches", Proceedings of the 13^{th} International Meshing Roundtable, October 2004.

[6] Dannenhoffer, J.F., "OpenCSM: An Open-Source Constructive Solid Modeler for MDAO", AIAA-2013-0701, January 2013.

[7] "WebGL", http://www.khronos.org/webgl.

[8] "WebSockets", http://www.websocket.org.

[9] Gray, J., Moore, K.T., and Naylor, B.A., "OpenMAO: An Open Source Framework for Multidisciplinary Analysis and Optimization", AIAA-2010-9101, September 2010.