OpenCSM: An Open-Source Constructive Solid Modeler for MDAO

John F. Dannenhoffer, III*

Syracuse University, Syracuse, NY, 13244, USA

The advent of Multi-Disciplinary Analysis and Optimization environments (MDAO) has put increased emphasis on the automatic generation of configuration boundary representations directly from a feature tree (build recipe) and a set of externally-driven parameters. Many commercial computer-aided design systems exist to satisfy this need, but all currently suffer for several limitations: 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 primitives; the build process is not explicitly "differentiated", thus making the gradients required by optimizers available only via finite differences; and, the systems are licensed in such a ways that distributed, simultaneous execution on many computers can be cost prohibitive.

Described here is an open-source constructive solid modeler, named OpenCSM, that has been developed to circumvent these limitations. OpenCSM, which is built upon the OpenCASCADE geometry kernel and the EGADS geometry generation system, is freely available on virtually any computing system. The use of OpenCSM as part of NASA's OpenMDAO optimization system and as the basis for automatic overset grid generation is demonstrated.

I. Introduction

GEOMETRY generation has played a pivotal, yet under-appreciated, part of the grid generation and CFD processes for the past several decades. Recent attempts to improve the definition of geometries has centered on direct links between Computer-Aided Design (CAD) systems and the grid generators. For example, the Pointwise grid generation system[1] has direct links to the ACIS, Catia (V4 and V5), Parasolid, Pro/ENGINEER, and SolidWorks geometry kernels and to CAD systems built upon them.

Early CAD systems approached the construction of models using a bottom-up approach: points were defined in space, which were joined by curves, which were combined into surfaces, and ultimately stitched together into volumes. While very powerful, this approach was very time consuming and often yielded volumes that contained little flaws (often due to tolerances) that caused downstream processes, such as grid generation, untold headaches. In fact, a whole industry was created to take care of the "CAD Healing" process.[2]

Fortunately, CAD systems have evolved over the past decade. Modern CAD systems have approached the process from a top-down perspective, known as constructive solid modeling. In this approach, a model consists of two types of items: a build recipe (sometimes called a feature tree) that describes the types of and order of operations that one must perform; and a set of parameters that influence the exact shape of objects created (and sometimes which part of the feature tree should be executed).

Once the model is defined, the CAD system then executes the build recipe to generate a boundary representation (BRep) that consists of nodes, edges, faces, and solids.

Unfortunately, modern CAD systems are quite large and have several shortcomings:

- 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 primitives;

*Associate Professor, Mechanical & Aerospace Engineering, AIAA Associate Fellow.

- the build process is not explicitly "differentiated", thus making the gradients required by optimizers available only via finite differences; and
- the systems are licensed in such a ways that distributed, simultaneous execution on many computers can be cost prohibitive. This is especially important for very large scale computations (such as multi-disciplinary analysis and optimization – MDAO), where it would be nearly impossible to have one, or a few, computational nodes perform all the geometric operations.

II. Design Objectives

The objective of the current work is to circumvent the above difficulties by designing and building a system that:

- is open-sources that is (will be) freely available;
- is built only upon open-source software;
- has ASCII description file that can be created by and/or modified by any external program;
- is extensible via user-defined primitives; and
- directly provides sensitivities.

In particular, this paper describes a system, named `OpenCSM`, is built upon `EGADS`,[3] which provides simple access to the `OpenCASCADE`[4] geometry generation system.

III. OpenCSM Model

As mentioned above, models in CAD systems (and `OpenCSM` in particular) are comprised of a set of (design) parameters and a feature tree.

In `OpenCSM`, the parameters have unique names and are all stored as two-dimensional arrays of floating point numbers; scalars are simply parameters with one row and one column. Parameters in `OpenCSM` can either be *external* or *internal*. External parameters are those that are exposed to the outside world and can be modified programatically in `OpenCSM` via external calls to `OpenCSM`'s API (discussed below). The values of these external parameters are always specified as numbers. Internal parameters are those used during the execution of a model. In a sense, internal parameters can be thought of as multi-valued temporary variables. As such, internal parameters can either be specified as a number or as a MATLAB-like expression based upon the current values of external and other internal parameters; these specifications can only be done by altering the feature tree. One unique feature about external parameters in `OpenCSM` is that, in addition to a value, external parameters can have a "dot" associated with them which is used in the calculation of sensitivities. The "dot" values for all external parameters are combined in order to compute the "change" in any of the internal parameters, feature-tree arguments, or the resulting BRep.

Feature trees in `OpenCSM` are defined in terms of a binary-like tree of branches. Each branch has an associated type (such as `box`, `union`, `fillet`, or `rotatex`) which describes the operation to be performed. Each branch requires zero to nine arguments, depending on its type. For example, the `box` branch requires six arguments (the locations of two opposite corners in three-dimensions) while the `rotatex` branch only requires three arguments (the rotation angle and the y and z coordinates about which the rotation will occur). Arguments are specified as expressions and are evaluated at build time based upon the current values of the external and internal parameters.

As mentioned above, branches are stored in a binary-like tree structure. Branches can have zero to two parents; primitive solids have no parents since they start the branch of a tree; transformation branches (such as `rotatex`) have one parent while Boolean operations (such as `intersect`) have two parents. For each parent link, there is a reciprocal child link. Any branch without a child link corresponds to a solid body (or BRep) that is the ultimate product of the build operation in `OpenCSM`.

An additional capability associated with branches is that they can be suppressed (and resumed) by a user before a build operation. Branches that are suppressed are treated as if they are not present in the feature tree, making it an easy task to add or remove fillets and other features. In an MDAO setting, such a capability is essential if one wants to use the same model to create different representations (as described below).

Lastly, one of the unique features of `OpenCSM` is that each branch in a feature tree can have zero or more "attributes". These attributes, which are composed by a name:value pair are carried throughout the build process are are automatically applied to the various faces of the BReps that `OpenCSM` produces. In this way, one can easily determine which branch in the feature tree is associated with a particular face in a BRep. This facilitates multi-disciplinary coupling in a natural way.

IV. OpenCSM API

There are two major software components of `OpenCSM`: an Application Programming Interface (API) that is described in this section and a description language (described in the following section).

The API of `OpenCSM` currently consists of 31 functions that are callable from C. While not strictly object-oriented, the API has been designed in an object-based manner, making it directly interfacable with object-oriented systems written in Python, Tcl/Tk, C++, or Java.

Within the `OpenCSM` API, the functions can be broken into several groups. The first group includes functions for loading and building the master model. Specific functions include:

ocsmLoad — read a .csm file (described below) and create a model in memory that consists of parameters and the feature tree.

ocsmCopy — create a copy of a model

ocsmSave — save a model to a .csm file

ocsmBuild — execute the feature tree (with the current parameters) that is in memory and create a group of bodies (returned as a list of BReps)

ocsmFree — free up all storage associated with a model

As a minimum, every application that uses `OpenCSM` will likely call `ocsmLoad`, `ocsmBuild`, and `ocsmFree`.

Once a model is created in memory, users likely will want to interrogate and/or edit the parameters in the model. To this end, `OpenCSM` contains five functions:

ocsmGetPmtr — get the size of (rows and columns) and type of (external or internal) of a parameter

ocsmNewPmtr — create a new external parameter (with a specified number of rows and columns). Note that internal parameters are created by a "set"-type branch being added to the feature tree.

ocsmGetValu, ocsmSetValu — get/set the definition of one of the elements (specified in terms of its row and its column) of a parameter. Values can only be set for external parameters.

ocsmSetDot — set the "change" in the parameter for which the sensitivity of the other parameters or BRep will be evaluated

These routines will typically be called after a model has been set up in memory (via `ocsmLoad` or `ocsmCopy`) but before it is (re-)built (via `ocsmBuild`). In particular in an MDAO environment, it is likely that these functions will get called before every `ocsmBuild` operation in order to generate the various instances of the configuration that the MDAO system will evaluate on its journey toward an optimal solution.

In a like manner, the feature tree can be interrogated and/or edited once it has been loaded into memory. Functions that facilitate these operations include:

ocsmGetBrch, ocsmSetBrch — get/set information associated with a feature tree branch. At this time, the only information that can be set on a branch is its activity (that is, whether the branch is suppressed or active). If the type or arguments of a branch are to be changed, one must first delete the old branch and then create a new one.

ocsmGetArg, ocsmSetArg — get/set an argument associated with a particular branch.

ocsmGetName, ocsmSetName — get/set the name of a branch. The default name of a branch is `Brch_xxxxxx`, where `xxxxxx` is a unique sequence number.

ocsmNewBrch — add a branch to the end of the feature tree

ocsmDelBrch — delete the last branch from the feature tree

ocsmGetAttr, ocsmRetAttr, ocsmSetAttr — get/return/set an attribute associated with a particular branch.

Once a model has been executed, it creates one or more bodies that are expressed a BReps. Within `OpenCSM`, these interrogations are typically performed by direct calls to the underlying geometry kernel (either `EGADS` or via `CAPRI`.[5] The function:

ocsmGetBody — returns identifying information that is needed in the `EGADS` and/or `CAPRI` library calls.

Finally, `OpenCSM` provides a set of utility functions that perform a number of house-keeping duties. Included in this category are:

ocsmVersion — identify the version of `OpenCSM` currently running

ocsmSetOutLevel — define the level of user feedback that `OpenCSM` produces. The levels can range from "0" which produces only error and warning messages up to "3" which provides full debug information.

ocsmInfo — return number of branches, parameters, and bodies in a model. Since branches, parameters, and bodies are always reference by their index (in the routines described above), a typical application needs to call this function to determine the appropriate indices in any model.

ocsmCheck — determine if the branches in a model are properly ordered. This quick check is particularly useful to ensure that a model being created will likely be built properly when `ocsmBuild` is called.

ocsmPrintPmtrs, ocsmPrintBrchs, ocsmPrintBodys — create "printable" listing of parameters, branches or bodies (which is useful both for documentation and for debugging)

ocsmGetText — returns a textual representation of any internal `OpenCSM` code.

ocsmGetCode — returns the internal `OpenCSM` code given its textual representation.

V. OpenCSM .csm File Description

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 primitive bodies that can be generated are the box, cylinder, cone, sphere, and torus as shown in Fig. 1. Each of these has no parent branch (since they represent the "beginning" of a path within the feature tree).

A unique feature of `OpenCSM` is that users can define their own primitives. For example, one could write a "wing" or "fuselage" primitive (as in the Vehicle Sketch Pad[6]) that produces a fully-parameterized solid. This capability in `OpenCSM` is not one that is typically found in commercial CAD packages. Within `OpenCSM`, these primitives are called "user defined primitives", or UDPs, and are implemented in user-supplied code using the "bottom-up" building approach that is supported by `EGADS`. Currently, `OpenCSM` ships with UDPs for ellipses, NACA airfoils, waffles, and freeform solids. UDPs have no parents.

Other simple solids can be created by starting with a sketch (which is comprised of linear segments, circular arcs, and splines). A typical sketch is shown in Fig. 2. These sketches can then be extruded, lofted (using 2 or more sketches), or revolved about an axis that is aligned with one of the coordinate axes, producing the solids shown in Fig. 3. The branches associates with these solids have no parent branches.

Once one of the solid bodies have been produced, there are a variety of operations that can be performed on it. The simplest of these is a coordinate transformation, in which the body can be translated, scaled, or rotated about any coordinate-aligned axis. (More complex rotations can be performed by performing a series

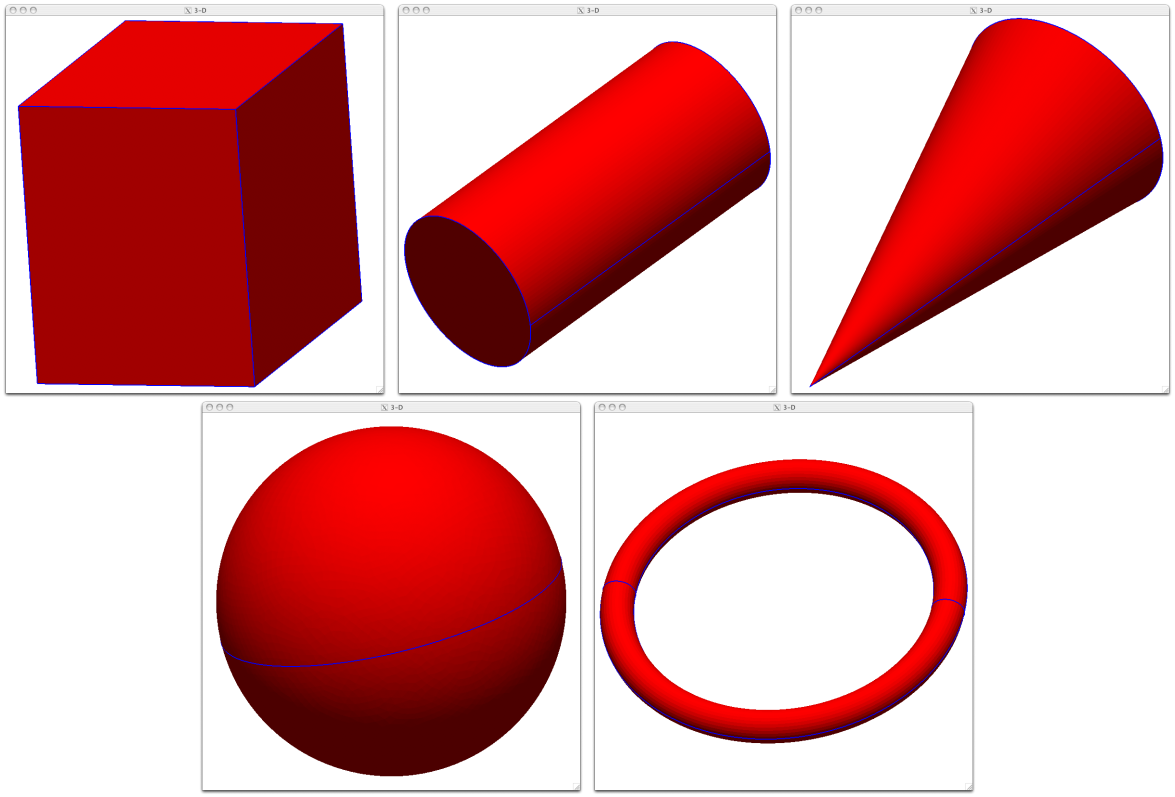

Figure 1. Primitive solids created by `OpenCSM`. (box, cylinder, cone, sphere, and torus)

of simpler rotations.) In each of these cases, the feature-tree branch associated with the transformation has one parent branch (that is, the body that is to be transformed).

In the spirit of constructive solid modeling, a key capability of `OpenCSM` is its ability to perform Boolean operations on a pair of bodies. The supported operations are the union (known in `OpenCASCADE` as fusion), difference (subtract), and intersection operators (common). The results of these operations performed on a box and cylinder are shown in Fig. 4.

`OpenCSM` also supports the application of fillets and chamfers to an existing solid, as shown in Fig. 5. Recently added to `OpenCSM` is the ability to "hollow out" a solid body (producing a thinned-wall body) and the ability to create a body that is "offset from" a previous body. All of these operation have one parent branch in the feature tree.

When `OpenCSM` finishes its build process, all solids that remain on the stack are returned to the user. These will be the feature-tree branches that do not have any child branches (that is, they are not "used" in any other `OpenCSM` operation).

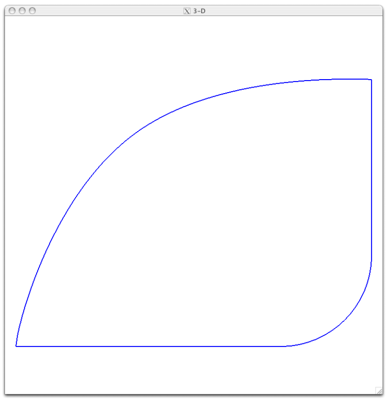

Figure 2. Sketch comprised of line segments and circular arcs.

.

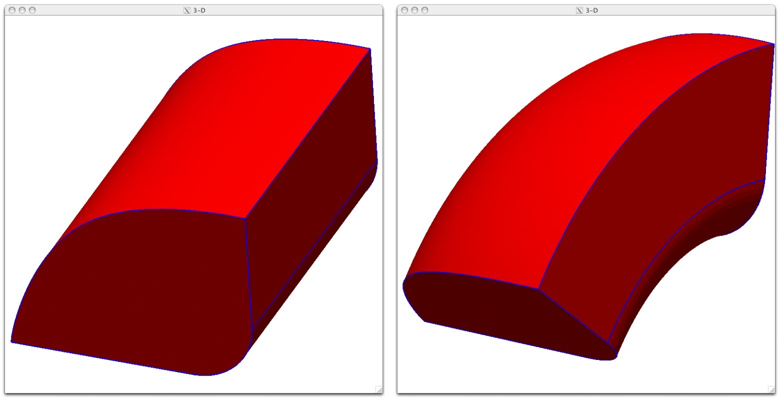

Figure 3. Solid bodies generated by extruding or revolving a sketch. (extrude and revolve)

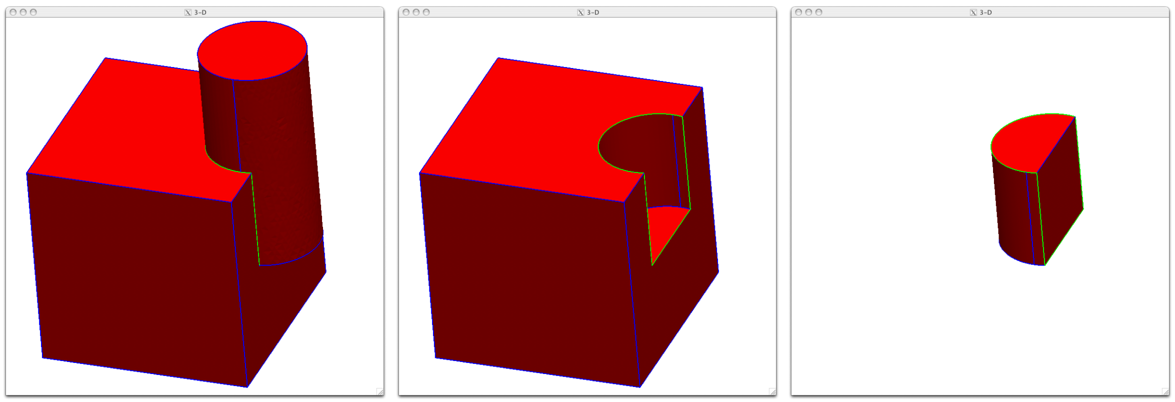

Figure 4. Solid bodies performed by the Boolean combination of a box and a cylinder. (union, subtract, and intersect)

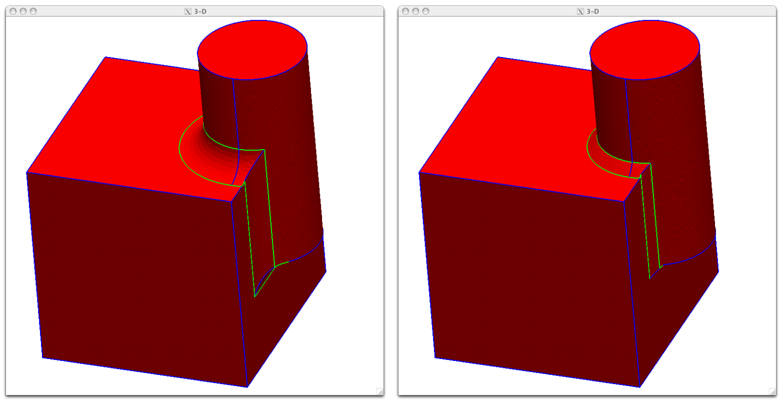

Figure 5. Solid bodies by apply fillets and chamfers to a solid.

VI. OpenCSM Example

To demonstrate the above, the "bottle" configuration from the OpenCASCADE tutorial[7] has been defined in a .csm file, producing the configuration shown in Fig. 6.

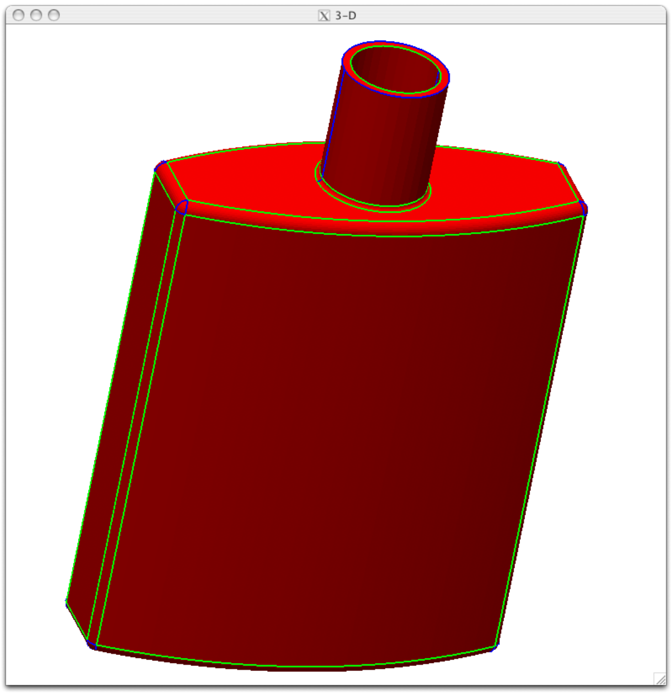

Figure 6. OpenCASCADE bottle built in OpenCSM

The csm file that produced the configuration is:

```
# bottle (from OpenCASCADE tutorial)

# default 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  wall thickness (in neck)
despmtr   filRad1            0.25 cm  fillet radius on body of bottle
despmtr   filRad2            0.10 cm  fillet radius between bottle and neck

# basic bottle shape (filletted)

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
   linseg -width/2  -depth/4  0
skend
extrude   0         0         baseHt
fillet    filRad1
```

```
# 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

end
```

Note that the design parameters and their initial values are listed at the top of the file, followed by the build recipe. The sketch (which represents the top view of the body of the bottle) is bracketed by the `skbeg` and `skend` statements and can be seen to consist of two line segments and two circular arcs. Once the sketch is complete, it is `extrude`d and `fillet`s are added. The neck of the bottle is two concentric cylinders that are `union`ed with the body of the body and a `fillet` is placed at the set of edges that are common to the bottle's body and its neck. This 14 line description is much more compact than the 7+ page tutorial provided by `OpenCASCADE`.

VII. Sample Configurations

Thus far, over 200 configurations have been modeled by the `OpenCSM` system. To demonstrate `OpenCSM`'s versatility, three configurations are shown here.

The first is a flapped-wing/pylon/store configuration (which was originally produced to study the automatic generation of overset grids). The feature tree (build recipe) for this configuration, which is 336 lines long (including 127 lines that define the "spline curves" and lots of comments), produces a boundary representation containing 163 volumes, 715 nodes, 1229 edges, and 542 faces, as shown in Fig.7.

The image on the left focuses on the vicinity of the flap. The small gap between the flap and the main wing is modelled as is the attachment hardware (including fillets between the hinge plates and the wing surface). The image on the right focuses on the pylon and its three stores. Although not readily apparent from the pictures, the stores are actually "offset" a bit from the pylon (with pins holding the stores in place); all of this is fully modeled. Similarly, there is a small gap between the fins on the stores and the bodies of the stores; the fins are connected to the stores via little pins, which facilitates the rotation of the fins during maneuvering. Also, if one looks closely the "access door" on the body of each fin can be seen.

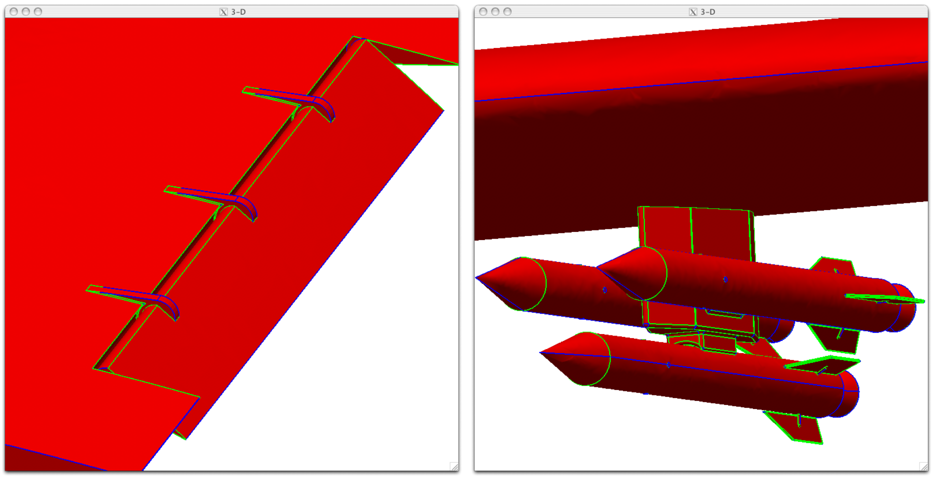

Figure 7. Flapped-wing/pyon/store configuration.

A second configuration is the JMR3, whose `.csm` file is 255 lines long (including comments), consists of 378 branches, producing a boundary representation containing 115 volumes, 296 nodes, 462 edges, and 194 faces. As shown in Fig. 8, essential features include: the stiffening rings and attachment hardware; a vent line that runs nearly the length of the rocket and which is slightly offset from the main body; various boxes, pipes, and spherical tanks near the base of the rocket; and four "thruster pods" near the nose of the body that each contain two conical thruster nozzles.

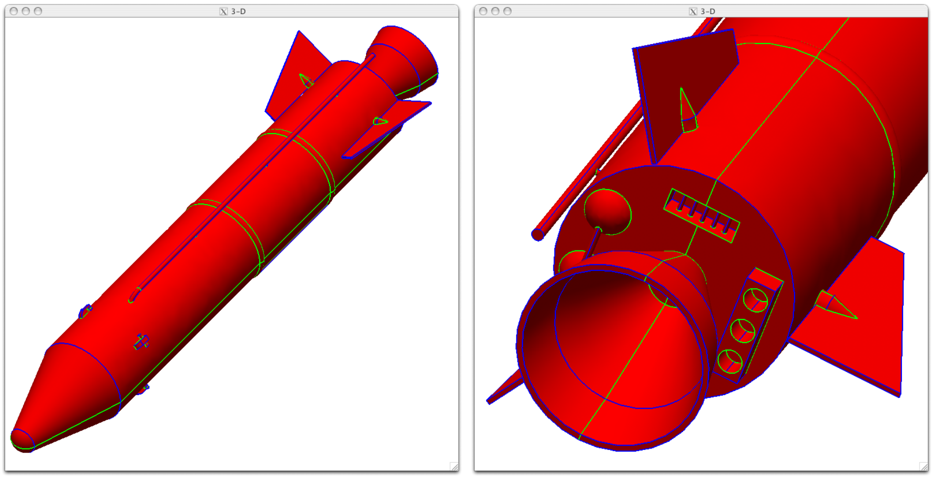

Figure 8. Baseline JMR3 configuration.

Since `OpenCSM` is fully parametric, design changes are easy to accomplish. For the JMR3, two simple modifications were made: the thrusters were moved afterward and the fins were rotated, as shown in Fig. 9. Each of these modifications were performed by appropriate calls to `ocsmSetValu` between the time the model was loaded into memory (via a call to `ocsmLoad`) and the time it was rebuilt (via a call to `ocsmBuild`).

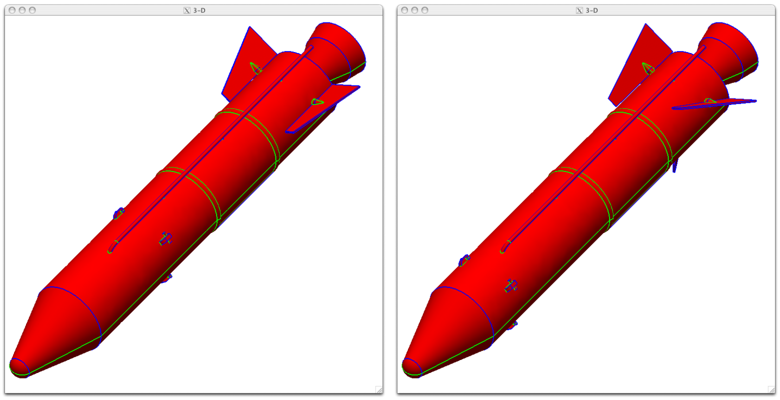

Figure 9. Examples of new configurations after changing thruster position and fin angle-of-attack.

To further demonstrate the flexibility of `OpenCSM`, the JMR3 model was "de-featured" by suppressing certain branches in the feature tree. This produced a variety of boundary representations, as shown in Fig, 10.

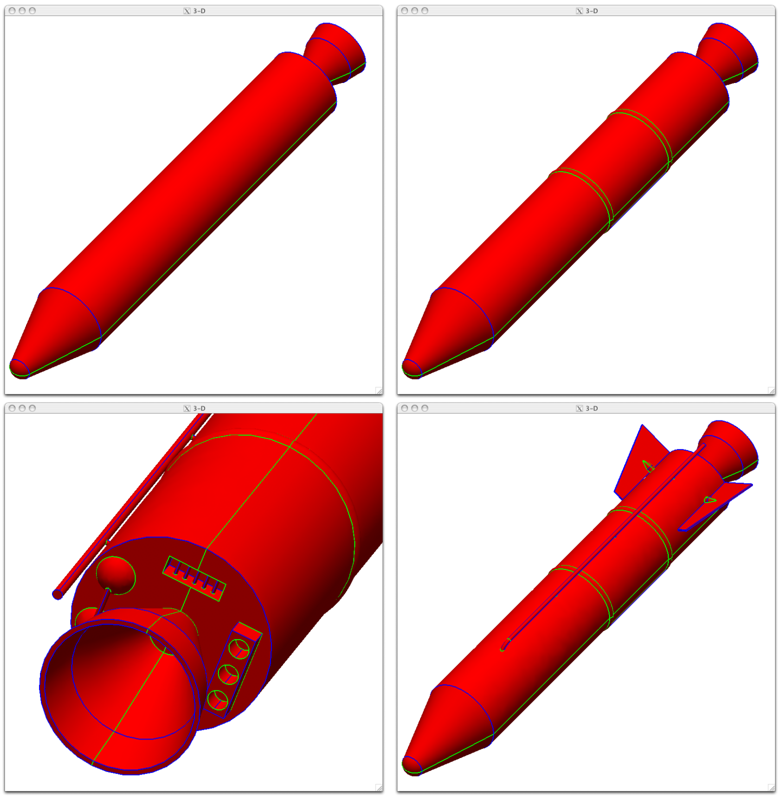

Figure 10. Examples of "de-featuring" the JMR3 by suppressing selected branches in the feature tree.

VIII. Current Use

OpenCSM has recently been released and is currently in use for a variety of applications. One notable application is as the configuration builder in NASA's OpenMDAO multi-disciplinary analysis and optimization system.[8] As part of the OpenMDAO development, an optimization study of the lean-direct injector geometry was performed. In particular, a series of 30 cases were generated to serve as the basis for a reduced-order model. Fig. 11 shows two of the configurations in the study. Both of these case was generated with the same OpenCSM model; the only difference was that the design parameters were changed. The critical design parameters in this study were the number of vanes in each injector, the swirl angle, and the diameter of each nozzle. Note that by changing the nozzle diameter, the number of nozzles automatically changed (since the configuration was defined so that "as many nozzles as possible" were to be placed in the configuration).

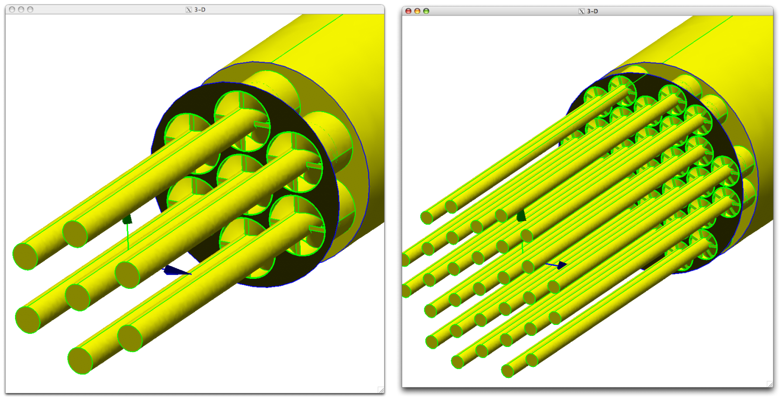

Figure 11. Cutaway views of two cases from the LDI optimization study.

A second current application is the multi-disciplinary coupling of solutions that were developed from different sub-models of a configuration. The application here is the fluid/structure interaction for a fighter-like application. Fig. 12 shows four sub-models build from the same OpenCSM model. The two models on the top represent the mid-surface aero model (which is used as part of aero-elastic computations) and the full outer-mold line model (which is appropriate for CFD modeling). The two models on the bottom represent the "built-up element model" of the structure of the fighter and a fully-3D solid model of the structures (both of which are appropriate for finite-element modeling).

Since all of these sub-models were built from the same OpenCSM model, they respond in unison to any changes in design parameters. Fig. 13 shows how the built-up element model responds to changes in the wing twist and the number of (and thickness) of the wing ribs. A more important consequence of using a single model (with sub-models) is that each of the sub-models have the same model-defined attributes. So via attributes, one can automatically identify the "surfaces that go together", which makes multi-disciplinary transfer of loads and displacement between fluid and structure a nearly-trivial process.

A third current application (and the original impetus for building OpenCSM) is for the OvrCad automatic overset grid generation system.[9] Fig. 14 shows a lander (patterned after the Morpheus lander) and a set of various grids that were automatically generated using OpenCSM's feature tree. It is only because OpenCSM's feature tree is accessible (and editable) that these overset grids can be generated automatically.

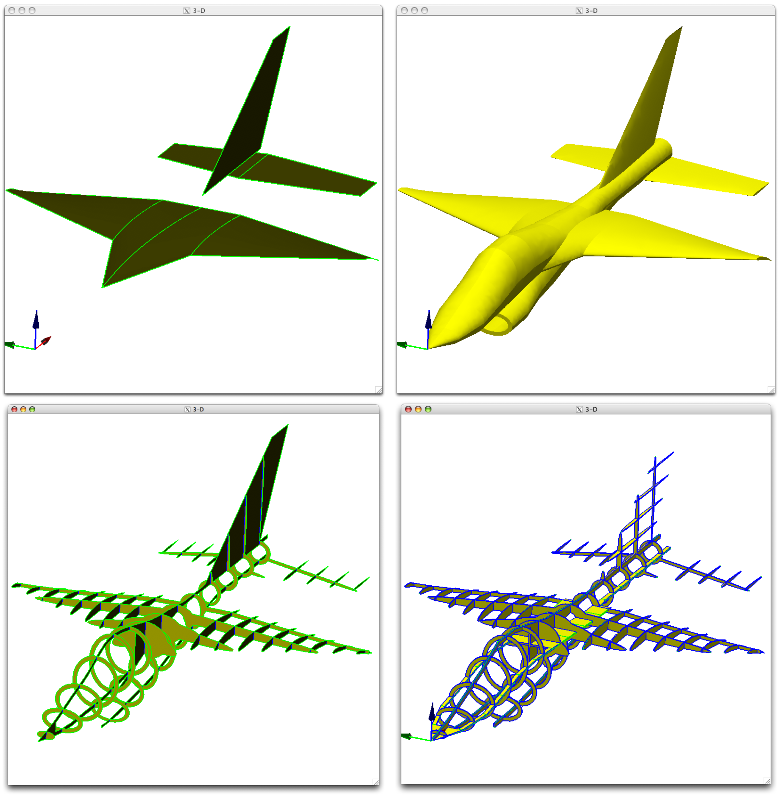
Figure 12. Four sub-models (mid-surface aero, outer-mold line, built-up element model, and solid structural model) for a fighter-like configuration.

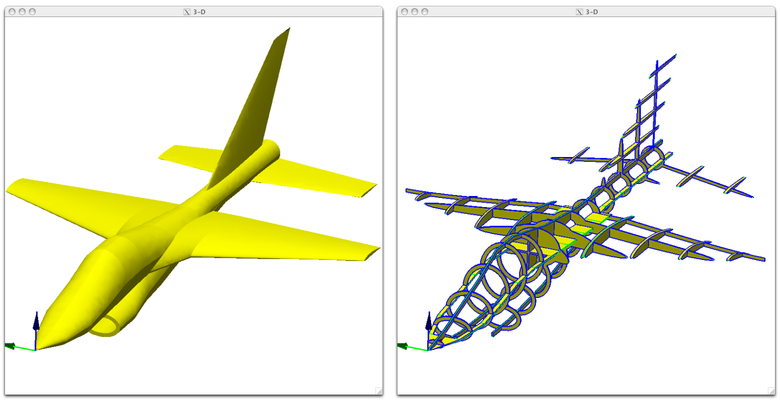
Figure 13. The effects of design changes (wing twist and wing ribs) on the fighter models.

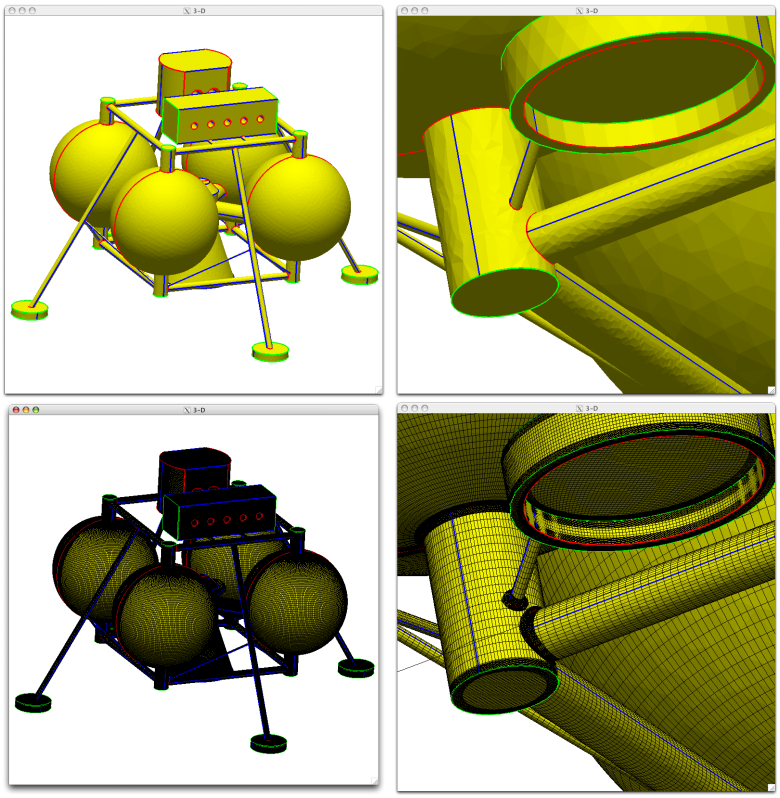

Figure 14. The solid model of a lander and the overset grids automatically generated on it (via `OvrCad`).

IX. Summary

Described herein is an open-source constructive solid modeling system named `OpenCSM`, which is built upon the open-source software (`OpenCASCADE` and `EGADS`). It uses an ASCII model file that can be created by and/or modified by any external program. A key feature of `OpenCSM` is that it is extensible via user-defined primitives; therefore adding primitives such as wings and fuselages is trivial, making `OpenCSM` suitable throughout the design process. A second key feature is the it extensively uses user-defined attributes so that multi-disciplinary coupling is possible in an automatic way. Lastly, `OpenCSM` has a built-in sensitivity calculation system, making it more useful in an MDAO environment than traditional CAD systems. Applications of `OpenCSM` for design and automatic overset grid generation are highlighted.

X. Availability

`OpenCSM` is an open-source system that is built upon only open-source components (`OpenCASCADE` and `EGADS`). It is written in C and is currently licensed under LGPL v2.1.

It can be down-loaded from `acdl.mit.edu/ESP`.

XI. Acknowledgments

This work was performed as part of NASA Cooperative Agreement NNX11AI66A; Christopher Heath (NASA-GRC) is the Technical Monitor. Also, the author wishes to thank Bob Haimes (MIT) for his many helpful suggestions and code snippets throughout all phases of this projects.

References

[1]"Pointwise CFD Meshing Software's CAD Capabilities", http://www.pointwise.com/pw/cad..shtml

[2]"CAD translation - CADfix", http://www.transcendata.com/products/cadfix/

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

[4]"OpenCASCADE Technology, 3D Modeling & numerical simulation", http://www.opencascade.org.

[5]"CAD to CAE Connectors", http://www.cadnexus.com/index.php/cae-interop.html

[6]"OpenVSP", http://www.openvsp.org

[7]"My first [OpenCASCADE] application", http://www.opencascade.org/org/gettingstarted/appli/

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

[9]Dannenhoffer, J.F., "Automatic Creation of 3-D Overset Grids Directly from Solid Models", AIAA-2011-3540, June 2011.