Developing the Glenn-HT GUI

Gülnur Z. Avcı
Columbia University, New York, NY, 10027

Computational fluid dynamic (CFD) simulations are a fast and accurate way of collecting data that would otherwise be too expensive to obtain experimentally and simply not feasible analytically. Glenn-HT is one of the programs developed by NASA engineers to study heat transfer in turbo machinery by solving Navier-Stokes equations at each point of a 3D mesh. The Glenn-HT code is written in Fortran, which is a language now known mostly by physicists. Additionally, the successful operation of the program requires the input of many complex files, identifying surfaces, boundary conditions, and other convoluted parameters. The ability to do this depends on a concrete understanding of how to interpret the input files correctly, which makes the software less accessible for general use. For this reason, a graphical user interface (GUI) was developed to provide an easily and efficiently be able to run Glenn-HT simulations.

I. Introduction

This final report will serve as a record of the progress that has been made during the fall 2021 internship session and guidance for the following interns in further developing the Glenn-HT GUI. The first Glenn-HT GUI was developed in Java3D, but since it was discontinued, it was rebuilt in the Unity game engine in C#. However, review of the existing GUI revealed that it had to be restructured to be able to support further development. For this reason, the entire back-end and front-end code had to be rewritten. The new GUI now has an organized data model and allows unit testing, resulting in more efficient development and better user experience.

II. Simulation Files

As mentioned earlier, Glenn-HT is a software that runs simulations on meshes of airfoils and other structures. These meshes are made of "blocks" and "faces". Faces make up blocks, and it is the interaction of these elements that make the simulations possible. The functions that the GUI must be able to do require the interpretation and manipulation of these elements. Information about the blocks and faces that make up a mesh are imported into the GUI through different types of simulation files.

A. Block and GridPro Connectivity Files

A block file (.pdc file) is a file with a set of numbers that define the blocks that make up the mesh. A connectivity file (.conn file) specifies the interactions of faces of the blocks with each other, gives identification numbers, and identifies the type of each face. A GridPro formatted connectivity file must be formatted into a Glenn-HT format connectivity file to be simulated. Figure 1 shows the super blocks portion of a GridPro format connectivity file1. The number on the top states the number of superblocks. The most relevant columns for the sake of this report are the columns "sbid" and "pty":

1) "Sbid" stands for "super block id".
2) "pty" stands for "property".

A odd super block pty indicates a fluid zone while an even pty indicates solid zone. Fluid and solid zones are what are referred to as volume zones. A collection of super blocks with the same pty, depending on whether the pty is odd or even, belong to the same fluid or solid zone. For example, if there are 30 super blocks with half of them having a pty of 1 and the other half with a pty of 2, the group of super blocks with a pty of 1 would be considered a single fluid volume zone and the other group with a pty of 2 would be considered a single solid volume zone. The blocks read from the .pdc file correspond to the superblocks in order.

Figure 2 shows the face patches section of a GridPro format connectivity file. The number on the top indicates the number of face patches specified2. Face patches define interactions of faces with each other. There are 21 columns in this section. "pid" simply means patch identification number. The interaction of two blocks and their surfaces are indicated by "sb1 sf1 sb2 sf2" (superblock 1, surface 1, superblock 2, surface 2). "ijk 1L 1H 2L 2H" indicate the coordinates for the face or faces that a face patch represents and can be read as "1-Low-i 1-Low-j 1-Low-k 1-High-i 1-High-j 1-High-k

2-Low-i 2-Low-j 2-Low-k 2-High-i 2-High-j 2-High-k". "fmap" is a three digit number that determines how these coordinates need to be manipulated:

1) If fmap contains a 3, "2-High-i" and "2-Low-i" switch places.
2) If fmap contains a 4, "2-High-j" and "2-Low-j" switch places.
3) If fmap contains a 5, "2-High-k" and "2-Low-k" switch places.

"pty" determines the property of the face patches. Each line in the facepatches section is either a connectivity or a boundary/outer patch:

1) **Connectivity** if sb2 is not 0 (meaning the current patch is attached to another superblock/surface) and the pty is 1 or 3. This is when two faces connect with each other.
2) **Outer face** if sb2 is 0 (meaning the current patch is not attached to another superblock/surface). This is when a face isn't connected to anything.

Furthermore, the pty value of outer faces identifies what type of boundary condition it is:

1) **Inlet** if pty equals 5.
2) **Outlet** if pty equals 6.
3) **Symmetry/Slip** if pty equals 4.
4) **Wall** if pty equals 2.
5) **GIF** if pty is between 12 and 21 or equal to 1000.

Each face patch group of lines with the same sf1 and pty constitute a boundary condition of whichever one that pty represents (Ex: 10 face patch lines that all have a sf1 of 17 and pty of 6 would be an outlet boundary condition). We convert the group of face patches to faces and assign this array of faces to the boundary condition.

```
 30  super_blocks
#SB sbid I J K ebid eb2sbMap pty lbid
SB   1   5 17  9 0 000 1 -1
SB   2   5 37  9 0 000 1 -1
SB   3  37 13  9 0 000 1 -1
```

Fig. 1 Connectivity File: Superblocks

```
 130  face_patchs
#P pid sb1 sf1 sb2 sf2 fmap ijk1L 1H  2L 2H pty lbid
P    1   1  18   0   0 012   0 0 0 4 16 0  0 0 0 0 0 0   4 -1
P    2   1   7   0   0 012   4 0 0 4 16 8  0 0 0 0 0 0   4 -1
P    3   1  19   0   0 012   0 0 8 4 16 8  0 0 0 0 0 0   4 -1
P    4   2   7   0   0 012   0 0 0 0 36 8  0 0 0 0 0 0   4 -1
```

Fig. 2 Connectivity File: Facepatches

B. GlennHT Connectivity Files

The formatted GlennHT connectivity file is split into three sections. The connectivity section begins with a single number that indicates the number of connections. For example, 50 connections mean there are 100 lines for this section, since each line represents one face - this means that line 1 has connectivity with line 2, line 3 has connectivity with line 4, etc. The first column to the left represents the block index for that face, and the rest of the columns represent the diagonal/coordinates for that face 4.

```
      50
2             1   37    1             5   37    9
1             5   17    1             1   17    9
3             1    1    1            37    1    9
2             5   37    1             5    1    9
4             1    1    1             1   13    9
3            37    1    1            37   13    9
5             1   37    1             5   37    9
1             1    1    1             5    1    9
```

Fig. 3 Glenn-HT Connectivity: Connectivity

The outer patch section of the Glenn-HT connectivity file looks very similar to the connectivity function, starting with the number on the top that represents the number of outer patches. In this case, if it says there are 80 outer patches, this means that there are only 80 lines since these faces aren't connected to anything. Similarly, the left most column represents the block index, the next six columns represents the face diagonal, and the last column on the right represents the surface number for that face. It should be noted that multiple faces can make up a surface, since as it was mentioned earlier, faces retrieved from face patches are grouped together to make boundary conditions based on what surface they belong to.

```
      80
1           1   1   1           5   17   1          18
1           5   1   1           5   17   9           7
1           1   1   9           5   17   9          19
2           1   1   1           1   37   9           7
2           1   1   1           5   37   1          18
2           1   1   9           5   37   9          19
3           1   1   1          37   13   1          18
3           1   1   9          37   13   9          19
4           1   1   1           5   13   1          18
4           5   1   1           5   13   9          16
4           1   1   9           5   13   9          19
5           1   1   1           5   37   1          18
5           5   1   1           5   37   9           7
```

Fig. 4 Glenn-HT Connectivity: Outer patches

The bottom portion of the Glenn-HT connectivity file provide some information that is not explicitly expressed in the Gridpro format. Firstly, the number on line 1 represents the number of GIFs present in the mesh. GIFs are interactions between two surfaces that don't fit together perfectly (so as to be continuous) but they have an important interaction with each other which must be defined. The Gridpro format of the connectivity file defines which surfaces are designated GIF surfaces, but there is not information in the file itself that matches the two surfaces. These surfaces must be matched together by the user in the GUI. After this is done, the last portion of the Glenn-HT connectivity file is able to be written. Lines 2-6 each represent a GIFs connection between two surfaces:

1) **First number** represents surface 1 id.
2) **Second number** represents surface 2 id.
3) **Third number** represents the GIF type. RSI=0, Mixing Plane=1, Normal=2, Conjugate=3.
4) **Third number sign** represents the GIF Coordinate, where it is positive if its Cartesian and negative if its Polar.
5) **Fourth number** represents the GIF order. Zero Order = 0, Linear Order = 1, Cubic Order = 2.

Line 7 represents the number of volume zones, in this case there is only one. Line 8 lists the type of each volume zone, 1 if it's fluid and 2 if it's solid. Wherever the volume zone lines end (in this case starting line 9), starts the lines that list the type of each block (1 for fluid, 2 for solid).

```
5
18 17 2 1
19 20 2 1
21 33 2 1
35 34 2 1
36 37 2 1
1
1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
```

Fig. 5 Glenn-HT Connectivity: Other

C. Boundary Condition and Job File

While the connectivity and block file identify the elements that make up a mesh, the boundary condition (.bcs file) and job file (.ght file) specify the properties that belong to those elements. The boundary condition file specifies properties for inlets, outlets, symmetry/slips, walls, GIFs, and volume zones.

```
 &INLET_BC
inlet_subtype=1, inlet_ref_Mach_Nr=.5,
have_inlet_prof=.FALSE.,
filen_inlet_profile='no_file',
T0_const=1, P0_const=1, Tu_const=.01,
Ts_const=.01, ang1_const=45, bet1_const=0,
annular_inlet=.TRUE., deltah=0, deltat=0,
direction=1, surfID_inlet=20
 &END

 &SLIP_BC
slip_subtype=0, surfID_slip=23
 &END
```

Fig. 6 Boundary condition file

```
 &JobFiles
 DcmpFILE="ddcmp_ncpu7.dat",
 ConnFILE="../mesh/finalmesh-split.ght_conn",
 BCSpecFILE="boundary_conditions.bcs",
 GridFile="../mesh/finalmesh-ASCII-split.xyz",
GridFileFormat="formatted", Plot3DFileFormat="unformatted",
 SolnInFile="../solve_1/Out.soln",
SolnInFileFormat="unformatted",
 SolnOutFile="Out.soln", SolnOutFileFormat="unformatted",

           residFILE="his.subs",

           residFILE2="his.nosubs"
!,fvProbeDefinitionFile="fvProbes.input"
 &end


 &JobControl
 mRunLevel=0, LUNout=6, RestartSoln=.False., SaveSoln=.TRUE.
,SaveTransientSoln=.FALSE., VerboseScreenOutput=.TRUE.
 &end

 &TurbModelInput
 TbModelType=1
 &end
```

Fig. 7 Job File

III. Back-end

Some issues encountered in the old GUI were:

1) No data persistence. Information wasn't stored in classes, making it hard to keep track of the data.
2) Repetitive. Lack of a data model necessitated repetitive code that made the code hard to navigate.
3) Time-consuming to debug. The length and complexity of the code due to repetitiveness and lack of data model made it hard to debug.

A. Data Model

The term "data model" refers to the way the back-end code is structured and organized into classes. Discussion of how a mesh is represented in the simulation files gives clues as to how it should be stored in the data model. Table 1 lists the folders and scripts that store information. 1

The first folder **GridPro** has two important scripts called SuperBlock.cs and FacePatch.cs. In C# classes can be treated as objects that store information. So for example, each line of the superblock section in a GridPro connectivity file 1 can be stored in a unique SuperBlock object. For the sake of explanation, a SuperBlock class has separate integer variables called "sbid", "I", "J", "K", "ebid", "eb2sbMap", and "pty". By converting what was just a text file into separate objects, it becomes exponentially easier to be able to search through a list of SuperBlocks. Each line in the facepatch section of the GridPro connectivity file 2 is stored in FacePatch objects in a similar way.

The second group of classes under **Other** are Block.cs, Connectivity.cs, and Face.cs. These three type of objects are

retrieved from the information collected in SuperBlock and FacePatch objects. Block objects are retrieved from the .pdc file with a function (functions to be addressed in another section) and the type of the blocks are specified by the pty information from the SuperBlock objects. Both Connectivity and Face objects are retrieved from information from the FacePatch objects based on conditions discussed earlier.

The third folder **Zones** contain VolumeZone.cs, FluidZone.cs, and SolidZone.cs. The VolumeZone class is an example of what is called an abstract class. An abstract class can be thought of as a type of class that other subclasses can inherit from, but itself can't be instantiated as a separate object. Therefore, VolumeZone abstract class defines variables that are common to both the FluidZone and SolidZone class from which they can inherit from.

The fourth folder **Boundary Conditions** contain BCGroup.cs, BoundaryCondition.cs, InletBc.cs, OutletBC.cs, SymmetricSlipBC.cs, WallBC.cs, and GIF.cs. To start off, BoundaryCondition is an abstract class from which all the other classes within this folder with the exception of BCGroup inherit from. Except for BCGroup, the other classes include boundary condition specific parameters that must be specified. BCGroup is a class that holds lists of all the other boundary conditions to make it easier to be able to parse through them during the processing stage.

The fifth folder **Job** contains JobFiles.cs, JobControl.cs, TurbModelInput.cs, Plot3DParameters.cs, InitialCond.cs, TimeStpControl.cs, SPDSchemeControl.cs, RKSchemeControl.cs, MGSchemeControl.cs, GasPropertiesInput.cs, and ReferenceCond.cs. These are classes represent all the properties that need to be listed in the job file and include within the relevant parameters for each property. These classes are linked together through an interface called IJobProperties. An interface is a tool in C# that specifies what functions a class should include, but doesn't go further then simply listing these functions - the implementation must be done within the class that inherits from the interface. All the job property objects inheriting from IJobProperties makes it possible to group them together into a list and have a general reference to them that is not class specific.

B. Controllers

The **Controllers** folder contains scripts that perform certain actions on the information stored in the data model. The **GridPro.cs** script contains three functions that retrieve information from input GridPro files. *ReadGridProToBlocks()* reads a .pdc file to generate a list of Block objects. *ReadGridProConnectivity()* reads a GridPro connectivity files and returns an array of SuperBlock and an array of FacePatch objects. *ProcessGridProConnectivity()* first takes in the Block, SuperBlock, and FacePatch arrays returned from *ReadGridProToBlocks()* and *ReadGridProConnectivity()* functions. The tasks this function performs are listed below:

1) Performs fmap manipulation on the FacePatch objects. 2
2) Retrieves information from the manipulated FacePatch objects and creates lists of objects for connectivities (Connectivity), outer patches (Face), inlet faces (Face), outlet faces (Face), symmSlipFaces (Face), wall faces (Face), and GIF faces (Face).
3) Calls the function *CreateBoundaryConditions()* from **GlennHT.cs** (to be discussed later).
4) Creates SolidZone and FluidZone objects.
5) Updates the BlockType variable inside Block object to either fluid or solid (based on the information in the SuperBlock objects).
6) Returns an array of Connectivity, Face (outer faces), Face (GIF faces), VolumeZone (volume zones), and an object BCGroup (contains all the boundary conditions except GIFs since they need to be handled separately).

The **GlennHT.cs** script contains functions that help generate a Glenn-HT format connectivity file. Firstly, *CreateBoundaryConditions()* is called in **GridPro.cs** to generate boundary conditions from the lists of faces that were sorted out from the FacePatch objects. All faces that share the same surface number belong to a single surface. As such, unique boundary conditions can be generated from the list of faces for each type of boundary condition by finding all the distinct surface numbers that the faces have. New boundary condition objects of each type are created based on the surface numbers, and all the other parameters that must be specified for that boundary condition object are done within the GUI by the user. *CreateGIF()* is a function that creates GIF objects after the user specifies which two faces go together and other parameters that need to be defined. After the GIFs are matched, *GridExportToConnectivityFile()* is called to write the Glenn-HT formatted connectivity file. Furthermore, *ExportToBoundaryCondition()* and *ExportToJobFile()* functions can be called to write the .bcs and .ght files after the user specifies all the necessary parameters in the GUI. These two functions are able to be short and concise because they utilize the function *ExportToNameList()*, which takes any given objects and properly formats every public variable in a class to an appropriately formatted output string. This removes the need for repetitive code when having to parse through all the objects defined in the data model while writing an output file. *MergeConnFile* and the class *MergedFile* are used to merge connectivity file when the user wants to

Table 1 Data Model Folders and Scripts

Folders	Scripts
GridPro	FacePatch.cs SuperBlock.cs
Other	Block.cs Connectivity.cs Face.cs
Zones	VolumeZone.cs FluidZone.cs SolidZone.cs
Boundary Conditions	BCGroup.cs BoundaryCondition.cs InletBC.cs OutletBC.cs SymmetricSlipBC.cs WallBC.cs GIF.cs
Job	JobFiles.cs JobControl.cs TurbModelInput.cs Plot3DParameters.cs InitialCond.cs TimeStpControl.cs SPDSchemeControl.cs RKSchemeControl.cs MGSchemeControl.cs GasPropertiesInput.cs ReferenceCond.cs

Table 2 Back-end Controller Scripts and Functions

Scripts	Functions
GridPro.cs	ReadGridProToBlocks(), ReadGridProConnectivity(), ProcessGridProConnectivity()
GlennHT.cs	*Connectivity:* GridExportToConnectivityFile() *Boundary Condition:* ExportToBoundaryCondition(), CreateBoundaryConditions(), CreateGIF() *Job:* ExportToJobFile() *Merging:* MergeConnFiles(), class MergedFile *Other:* ExportToNamelist()
Plot3D.cs	WriteBlockBinary(), WriteBlockASCII(), ExportBlocksToPlot3D()
DataModelController.cs	Miscellaneous helper functions

combine different meshes together. The functions works as follows:

1) Takes in a list of block file paths and a list of GridPro connectivity file paths.
2) Reads each block file and creates a list of all the Block objects.
3) What needs to happen when connectivity files are merged is that the sbid (super block id) and pid (face patch id) for each additional file need to start from the last highest sbid and pid of the previous file. To keep track of this, a MergedFile class is created for each merged connectivity file. The MergedFile object stores the connectivity file path and dictionaries for all the boundary condition surfaces (including GIFs). Dictionaries in C# have a key and value for each entry. The local (what the id was in the original file) and global (what the id becomes in the final merged file) ids for each surface are kept track of in these dictionaries. Since each key has to be unique in a dictionary, the global id is added as the key (since the global id of a surface won't repeat itself) and the local id is added as the value. Since the 3D visualization of the GUI hasn't been developed yet, keeping track of the global and local ids of each surface saves the trouble for the user when they need to identify which surface is in question when entering the necessary parameters in the GUI.
4) Iterates through each GridPro connectivity file path. The user is aware that each added file is a single separate zone, meaning that for the super blocks section each pty should be equal to each other.
5) *ReadGridProConnectivity()* is called for each merged file. A MergedFile object is created for each iteration.
6) Returns a merged list of SuperBlock, FacePatch, Block, and MergedFile objects that can be used to generate a final Glenn-HT format connectivity file with *GridProExportToConnectivityFile()*.

C. Unit Testing

Unit testing is code itself written by the developer to test specific portions of a program to confirm their functionality. This makes saves a significant amount of time in debugging the code and allows us to identify problems sooner rather than later. All relevant test functions needed for debugging are included within "GlennHTGUI.Tests" project.

D. Dynamic-Link Library

Initially, both the back-end and front-end code were integrated together and therefore dependent on the Unity API. However, these two aspects of the GUI must be separated in order to make the back-end more versatile, so that whatever needs to be built on top of it doesn't become dependent on how the front-end is set up. A better way to import functions that process and store data is through a Dynamic-Link Library (DLL) which allow programs to call functions only when necessary. Therefore, the back-end can be written separately and imported into Unity by simply dragging the exported DLL file into the "Assets" folder in Unity. The steps to do this is as follows:

1) In Visual Studio Community project, Go to "Build" -> "Build Solution" in the upper window tab.
2) If the solution builds successfully, a file with the extension .dll should appear in GitHub -> glennht_gui -> backend -> GlennHTGUI -> GlennHTGUI.Library -> bin -> Debug -> net48.
3) Drag the .dll file into GitHub -> glennht_gui -> Glenn-HT GUI -> Assets.
4) All the classes are now accessible from the front-end scripts written in Unity as long as the correct namespace is included at the top of each script.

Separating different aspects of the GUI allows for faster debugging and development.

IV. Front-end

The front-end code was also rewritten to accommodate the new data model. While the back-end code is compatible with any other software the GUI could later be built with, the front-end code is Unity specific since it uses the Unity API to carry on its functions. Due to time restrictions, only the general functions that the Glenn-HT GUI must have were able to be built. Despite this, effort was put into making sure that advanced functions and capabilities could be implemented into the GUI.

A. Import Files Window and Merged Grids

On loading the GUI, currently the first view the user will see is the **Import Files** window with the option to either "Start New Case", "Merge Grids", or "Load Case". The concept of a "case" was implemented into the GUI to overcome an identified pain point which was that the user kept having to re-enter information because the data didn't persist between windows. In the old GUI, several case files were being generated to overcome this but it made using the GUI difficult since the user had to keep saving and loading several case files. The new GUI has the ability to start a new case

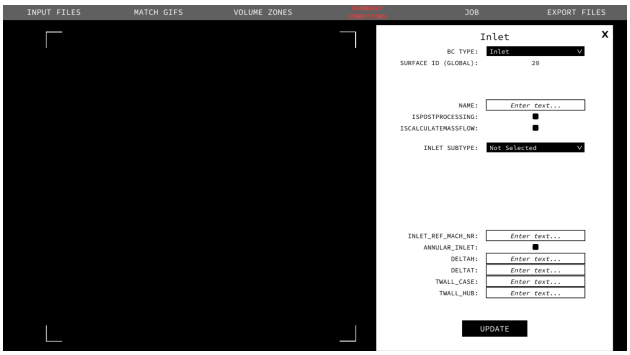

Fig. 8 Example view from GUI

and at whichever point the user decides they want to stop they can save a single case file which stores all the information that they've entered so far. When the user comes back to the case, they can load the case file through the "Load Case" option to repopulate the fields. This function is in the process of being implemented and all the tools are written to be able to accomplish it. However due to time constraints, the implementation is not completed. The case file would've been a JSON file (.json) which a file that stores simple data structures and objects. **Case.cs** includes functions called *SerializeToJson()* and *DeSerializeCaseJSON* which would be able to populate GUI Fields when called. What needs to be done by the incoming developer is to call these two functions from the "Load Case" button and the "Export Files" button and make sure that all the GUI elements are populated correctly. Writing a *LoadCase()* function might assist in this, since there are "get" and "set" functions already written for every element in the GUI (which will be discussed later). Additionally, it might be helpful to include a toggle in the **Export Files** window that asks the user if they want are ready to export files (as in writing all the necessary output files) or if they would just like to save a case file. If the user is exporting files, a case folder should be generated which includes all the necessary output files:

1) Connectivity file (.conn)
2) Boundary condition file (.bcs)
3) Job file (.ght)
4) Copy of all specified profiles and other files into the case file.

B. Match GIFS Window

Once the user presses "Process Files" in the **Import Files** window, all the other window tabs appear on the upper tab area. Similar to Figure 8, an area is designated for the development to the left of the parameter areas. The **Match GIFs** window allows the user to match all of the GIF surfaces and specifies their parameters to create GIF objects. The parameters of the added GIFs can be viewed and the GIF can be deleted and readded if necessary. Due to the current lack of a 3D view, the local and global surface id are shown in the window if the processed file were merged files.

C. Volume Zones, Boundary Conditions, Job, and Export Windows

The **Volume Zone** window lists all the fluid and solid zone and allows the user to specify any necessary parameters. The **Boundary Conditions** window has four buttons that bring up a list of each type of boundary condition. The user can click on each case and specify any necessary parameters. both the **Volume Zone** and **Boundary Condition** windows have a designated 3D view area to the left of the parameters area. The **Job** window has a tab system that allows the user to click on each job property and define any necessary parameters. The **Export Files** window has browser to define a location for the case file destination and an "Export Files" button.

D. Controllers and Other Scripts

A controller script was created for each window of the GUI. The **GeneralController.cs** does all the processing that involves the use of the controller functions from the back-end code. It also includes helper functions that are reused within other controller scripts to avoid repetitive code.

InputFilesController.cs controls the **Input Files** and **Export Files** window, **GIFController.cs** controls the **Match GIFs** window, **VolumeZoneController.cs** controls the **Volume Zones** window, **BoundaryConditionController.cs**

Table 3 Front-end Controller Scripts and Functions

Scripts	Functions
GeneralController.cs	*Processing functions:* ProcessNewCaseFiles(), ProcessMergeGridFiles(), ProcessLoadCaseFile(), ProcessInputFiles(), ExportFiles() *Helper functions:* GenerateButton(), FindLocalGIFSurface() and etc..., AllRequiredFieldsEntered(), SetFieldsInteractable(), SetFieldsActive()
InputFilesController.cs	*Start and Load Case:* StartNewCaseButton(), LoadCaseFileButton() *Merge Grids:* MergeGridsButton(), MergeGridsSetUp(), AddGrid(), DeleteGrid(), OnGridButtonClick(), EditGridNumbers(), CheckGridsToAdd(), CheckGridAddingFinished()
GIFController.cs	SetGIFS(), GetGIFs(), AddGIF(), DeleteGIF(), SetMergedFileGIFFields(), OnGIFButtonClick(), CheckGIFParameters(), CheckAddedGIFs(), class GIFCase
VolumeZoneController.cs	SetVolumeZones(), GetVolumeZones(), OnVolumeZoneButtonClick(), GetFields(), SetFields(), class VolumeZoneCase
BoundaryConditionController.cs	*General:* SetBoundaryConditions(), GetBoundaryConditions(), OnValueChangedDropdownBCType(), UpdateBCType() *Inlets:* CreateInletCase(), OnInletButtonClick(), GetInletFields(), SetInletFields(), InletListeners(), DropdownInletSubtypeChange(), ToggleAnnular_inlet(), AllRequiredInletFieldsEntered(), class InletCase *Outlets:* CreateOutletCase(), OnOutletButtonClick(), GetOutletFields(), SetOutletFields(), OutletListeners(), ToggleHave_Pback_prof(), AllRequiredOutletFieldsEntered(), class OutletCase *Symm/Slip:* CreateSymmSlipCase(), OnSymmSlipButtonClick(), GetSymmSlipFields(), SetSymmSlipFields(), SymmSlipListeners(), AllRequiredSymmSlipFieldsEntered(), class SymmSlipCase *Walls:* CreateWallCase(), OnWallButtonClick(), GetWallFields(), SetWallFields(), WallListeners(), DropdownWallSubtypeChange(), ToggleHave_Qwall_prof(), ToggleHave_Twall_prof(), AllRequiredWallFieldsEntered(), class WallCase
JobController.cs	*General:* SetJobProperties(), GetJobProperties() *Properties:* SetJobFiles(), GetJobFiles(), etc... for each property

controls the **Boundary Conditions** window, and **JobController.cs** controls the **Job** window. Certain trends can be noticed between these scripts. There is a general Set() and Get() for each controller:

1) *Set()* functions are called from **GeneralController.cs** once the files are processed in the **Import Files** window and pre-populate all the necessary windows for its respective controller.
2) *Get()* functions are called from **GeneralController.cs** in the *ExportFiles()* function to get all the necessary information from the windows to export files.

For elements that have instances - such as GIFs, volume zones, and boundary conditions - buttons are generated to organize each case into separate windows. There are *OnButtonClick()* functions that dictate what happens once the respective button is pressed. Often, *OnButtonClick()* functions call *Set()* functions for the element in question to populate the GUI window with the necessary information. Once the user decides they are done, the *Get()* function for that element is called to set the edited values in the data model. Other functions exist. *AllFieldsRequired()* functions exist to check if required fields are entered and use general helper function from **GeneralController.cs** to do so. The required field conditionals haven't been fully tested or set for the entire GUI, which is something the upcoming developer should consider. An *Extensions.cs* script exists that include various Unity specific helper functions that get rid of the need for repetitive code for common functions and make the code much easier to read by humans.

V. Conclusion

The main goal that was completed during this internship was the creation of a new GUI with versatile and extendable back-end and front-end code. The GUI is ready for the next developer to implement 3D view and functionality.

Acknowledgments

I would like to express special thanks to my mentor, Paht Juangphanich, along with Ali Ameri, David Rigby, and Kenji Miki for guiding me through my journey and making this project possible.