**Files:**

**File Extensions**

All files meant to run C++ code should either be a .h or .cpp file, since there may be other files in the project folder with various file extensions and using two specific file extensions helps quickly verify which files are C++ and which are not.

**Header Files**

All files should have an associated .h file that is used to quickly check usage and functionality. The only exceptions to this are the launcher file (which holds the main function) and any test files that are solely intended to test various functions / classes. Basically, as long as it does more than run code and facilitate, it should have a .h file.

However, not all .h files need an associated .cpp file, in the case that is simply a set of defines or is a struct definition.

All header files (and all .cpp files with a header, by connection) should have #define guards.

**Classes:**

This project has three primary use-cases for classes. Each class should state in a comment which category it falls under at the beginning of the .h file. Type classes and container classes should have comments with any complex or potentially confusing code, but for the most part they should be written in such a way that most of the code could feasibly be understood without comments by anyone who knows enough C++.

**Type Classes**

Type classes are primarily designed to be accessed as if they are an advanced variable “type”. They have a few required design strategies associated with this idea:

* Operator overloads for basic math operations that could potentially be utilized
* A default constructor
* Public fields, for easier access

These classes may also have any required additional math/utility functions built in if necessary, as well as additional constructors if it would make the type class easier to use.

**Container Classes**

Container classes are intended to act as a “container” or “wrapper” for type classes to assist in readability and ease of use if required. These classes also have a couple of required design strategies:

* Public fields, again for easier access
* A default constructor that “zeroes” everything

These classes will also likely have utility functions associated with them to make the container class actually useful, although the use and design of these is based on their intended usage.

**Control Classes**

Control classes are designed to be used by the main class in order to control everything. These are by far the largest classes in terms of complexity, so the less of them the better. However, they are still useful for containing complex code for reuse later, so with that comes some design strategies:

* These should be by far the most commented code. While code in type classes and container classes should be fairly self-documenting, code in control classes are by definition likely to be too complex for that to be feasible.
* Protected fields, with public getter and setter functions for more secure data control
* Utility functions should be separated visually from the core control functions

Getter and setter functions likely don’t need to be commented, but utility functions and control functions should have a short comment for each in the .h file explaining what they do, if not more within the .cpp file to understand usage

**General Code Practices:**

**Variable Names**

Variable names should be readable first and foremost, but also should be relatively short in order to not take up too much space. If variables are more than a loop index or briefly used temporary variable, they should be readable enough to be relatively self-explanatory.

Avoid variable names such as “temp”, “a”, or “b”, except for very specific circumstances where the variable is used for a couple lines at most.

Loop iterators or indices should ideally be named something related to their purpose, such as “idx” for index, “row” and “col”, “x” and “y”, etc. If all else fails, the variable names “i”, “j”, “k”, and “l”, specifically lowercase, are reserved for loop iterators. By connection, any other variables should be named something other than those four lowercase letters, if at the very least making them capitalized to distinguish them.

Variable names should be camel case, with a starting lowercase (camelCase).

**Function Names**

Function names (except for constructors and overloads of course) should be named in such a way that it could feasibly be determined generally what they do if you were given no other information other than the name of the function and the name of the class.

Function names should be camel case, with a starting lowercase (camelCase).

**Precision**

Due to the nature of the project, data used to hold information about the mesh or objects used to render the mesh should be doubles or higher precision. Otherwise, variables should use the most common variable type that could be used, for simplicity. For example, an integer should be used over a short or a long, unless a long is required, and a bool should be used for Boolean logic instead of an integer, since an integer may be confused for something that is intended to hold a numeric value instead of true or false.

**Other Names**

Class names should be camel case, with a starting capital (CamelCase).

Defines should be all caps (#define DEFINE), named very descriptively of what they do, and used sparingly.