Coding Style Guide

C++ Version

All code should be submitted using C++ 17, the latest version available.

Where possible, only standard libraries should be used, however due to the nature of this assignment graphical frameworks and test libraries from outside of the standard library will need to be used.

Header Files

Every .cpp file should have its own .h file, the .h file should contain declarations for classes and functions used in the associated .cpp file.

This improves the usability and modularity of the code.

Header files should compile on their own and therefore be self-contained. They should also end in .h for ease of understanding.

Order of Includes

Header files should be included in the following order: Related headers, C system headers, C++ standard library headers, other libraries' header, this project's header.

Separate each included header file type with a line. For example:

```
#include "relatedheader1.h"


#include <CSystemHeader1.h>
#include <CSystemHeader2.h>

#include <StdLibHeader1.h>
#include <StdLibHeader2.h>

#include "this/ProjectsHeader.h"
```

Scoping

Namespaces

Namespaces should be used to separate distinct scopes and will therefore prevent conflicts in the global scope, such as global variables with the same name. Namespaces should have unique, descriptive name of what the scope achieves.

Non-member, Static Member, and Global Functions

Always place non-member functions in a relevant namespace, avoid using a class to group static member functions, and avoid using entirely global functions.

Variables

Use variables as locally as possible. Prefer declaring variables in a function, then a class, then a namespace, and avoid using global variables overall.

Classes

Try to use classes as frequently as possible where appropriate.

Implicit Conversions

Avoid using implicit conversions, instead use the explicit keyword for conversions.

Structs or Classes

In general, prefer to use a class over a struct. As classes and structs behave very similarly in C++, it is best to avoid confusion and opt to use one consistently. As classes are more conventionally used, use a class rather than a struct.

Access Control

Make attributes of a class private in general. As an exception, attributes can be public if they are constants.

Order of Declaration

Declare public class members first, then protected, then private. Each different level of access should be grouped together, with all public together, all protected, and all private. Generally, related attributes should be declared together to minimise confusion.

Functions

Generally, return values should be used rather than output parameters for functions.

Short Functions

Functions should be kept as short as reasonably possible. This improved code readability and generally keeps the code tidy. Functions should not exceed 40 lines, though this is not a strict rule. If a function exceeds 40 lines, the programmer should consider where the function can be broken up into smaller function. For instance, if adding values to an array and then sorting the array in the same function, perhaps sorting the array can be separated into its own function. This should be left to the judgment of the developer.

Naming

Naming should be consistent for each entity. This is so that the naming style can be used to correctly identify what the entity is, for instance, it should be immediately obvious whether the developer is using a constant value or a variable simply at a glance from the naming convention used by each.

General Naming Rules

Names should be as readable as possible, regardless of what the entity being named is. It should be as obvious as possible to other people reading the code what the purpose of the named item is based on a glance at the name. With that being said, names shouldn't be overly long and clunky, as this can clutter the code and decrease the overall quality and therefore it is important for the developer to find a balance between descriptive naming and concise naming. Where possible, magic numbers should be avoided and instead have a descriptive but concise variable name assigned to them. Widely accepted, common names (for example, the use of "i" for an iterator in a for loop) are acceptable when used in the correct context.

File Names

File names should be all lowercase using underscores to separate words. For example, a good file name would be: "trees_image.png" whereas a bad file name would be: "Treesimage.png".

C++ files should end in .cpp and header files should end in .h, and all files should adhere to the general naming rules with reference to being descriptive and concise.

Class and Struct Names

Classes and structs should be named with a capital letter for each new word, without underscores. For example: "GoodClassName" or "GoodStructName". This can therefore differentiate them from variables names, which use a different naming convention.

Variable Names

Variable names should be named using camel case, and therefore have the first word starting with a lower-case letter, and every word following starting with an upper-case letter. An

example of this would be: "goodVariableName". This helps to differentiate from other named entities.

Data members included in types such as classes or structs should follow the same naming convention as regular variables.

Constant Names

Constant should be named in the same way as variables; however, the first 'word' will always be a lower-case letter 'k', with every following word beginning with an upper-case letter. An example of this: "kMonthsInAYear".

Function Names

Function names should follow the same naming convention as classes and structs, and therefore all contain an upper-case letter at the start of each word, for instance: "GoodFunctionName". Parameters should also be given sensible names, using variable formatting.

Namespace Names

Namespaces should be in all lower case with no separating characters, for instance: "givennamespace". This helps to differentiate namespaces from classes, functions, and variables.

Comments

Comments should not be used overly frequently. Good code should not require the use of excessive comments but should rather be readable on its own as much as possible. However, there will be times where comments are necessary, but prior to including a comment the developer should ask themselves if the code has descriptive variable names and is formatted well.

Both // and /* */ are acceptable styles of commenting.

Function Comments

Comments should be included before the definition of each function. The comments should describe what the function does and how to use it. Comments may be used inside the function to describe a particularly complex block of code (for example a long mathematical function) which would make it easy to understand but adhere to general comment guidelines as described above.

An example of a well commented function:

```
// This function takes in an array and sorts it using an insertion sort.
// Example:
//       double [] sortedArray = InsertionSort(nonSortedArray);

double InsertionSort(givenArray) {

 …
};
```

Variable Comments

The name of a variable, if using the naming guidelines described above, should be enough in explaining what the variable is for. On rare occasions, comments may be required such as in the case of global variables. When a global variable is declared, they should have a comment alongside them explaining what the global variable does and why it must be global rather than local.

This also help the developer to think about whether the global variable needs to be global or not.

Spelling, Punctuation, and Grammar

All comments should be given with correct spelling, punctuation, and grammar which should be easily read and descriptive. Comments should be written in prose and use as simple language as possible to help the next developer to understand what the code is doing. If the comment does not make sense, the comment is likely not very useful.

Formatting

Line Length

Lines of code (excluding comments) should be no longer than 100 characters including spaces, this helps to keep the code as readable as possible.

Spaces or Tabs

Tabs should be used rather than spaces to format block of code such as if statements, functions, and loops.

Function Declarations and Definitions

Function definitions should be done all on the same line, including the return type and the parameters, but excluding the curly braces. An example of this:

```
ReturnType ClassName::FunctionName(Type parOne, Type parTwo)
{

        …

}
```

Curly braces should begin on the next line, in line with the start of the function while closing in line with the first curly brace, as shown above.

Calling Functions

Functions should where possible be called on one line only, rather than wrapping onto another line. If functions must be called across multiple line, separate the parameters rather than the rest of the definition. For example:

```
Type result = FunctionName(aRidiculouslyLongArgument, parTwo,
                                        parThree, parFour);
```

Conditionals, Loops, and Switch Statements

For conditionals, for example an if/else statement, the formatting will be much the same as for functions but with the addition of the next section of the statement, for instance an else for a conditional statement. An example:

```
if (condition)
{
        …
}
else
{

        …

}
```

Variable Initialisation

A variable can be declared as per the developer's preference, using either =, ( ), or { } as all are valid within C++. Valid examples are as follows:

```
int x = y;
int x(y);
int x{y};
```

All of the above are valid ways of assigning a variable.

Namespaces

Namespaces will not have an indentation level, nor will anything within the namespace on the first level. This is because it is unnecessary to add a lot of white space and will generally make the code feel too sparse, rather than keeping it all neat and aligned.

Operators

Operators such as +, -, /, *, etc should always have spaces around them. This help to keep the code readable and easy to understand. Brackets do not require spaces. Examples include:

```
x + y = z
a / b = c
f * (t - s) = c
```