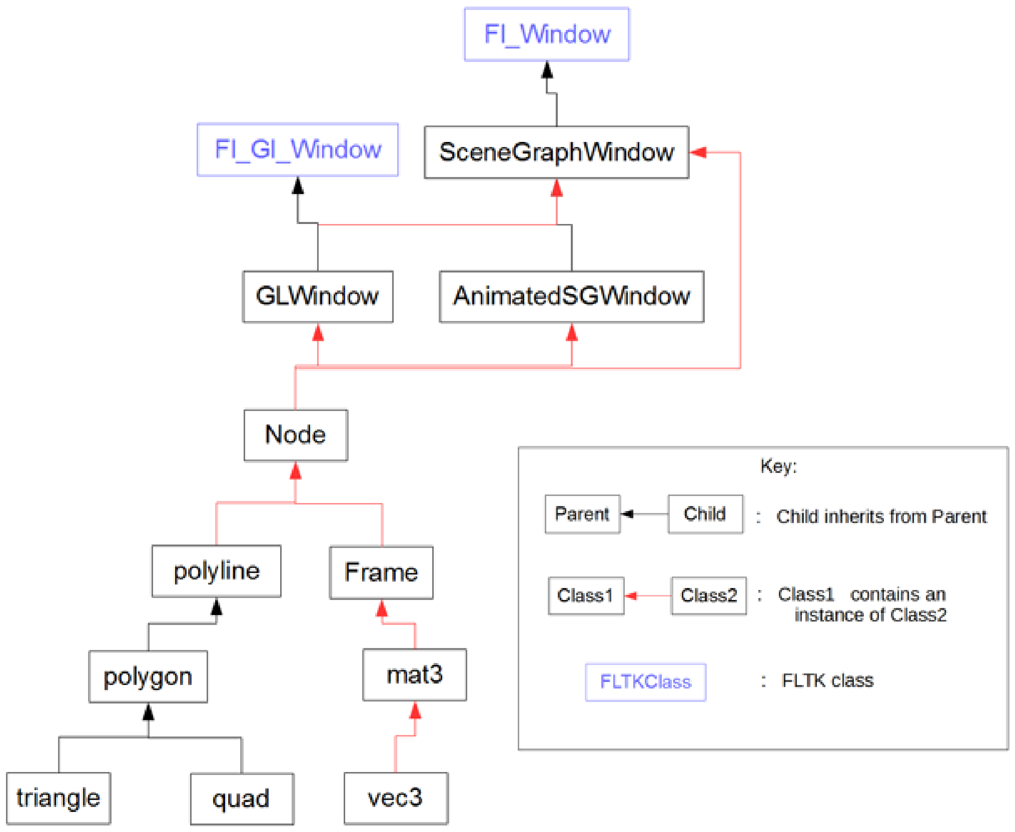
Zachary Ferguson
Professor Jan Allbeck
CS351-HW05
23 March 2015

Scene Graph Editor and Animator Design Document

Class Structure:



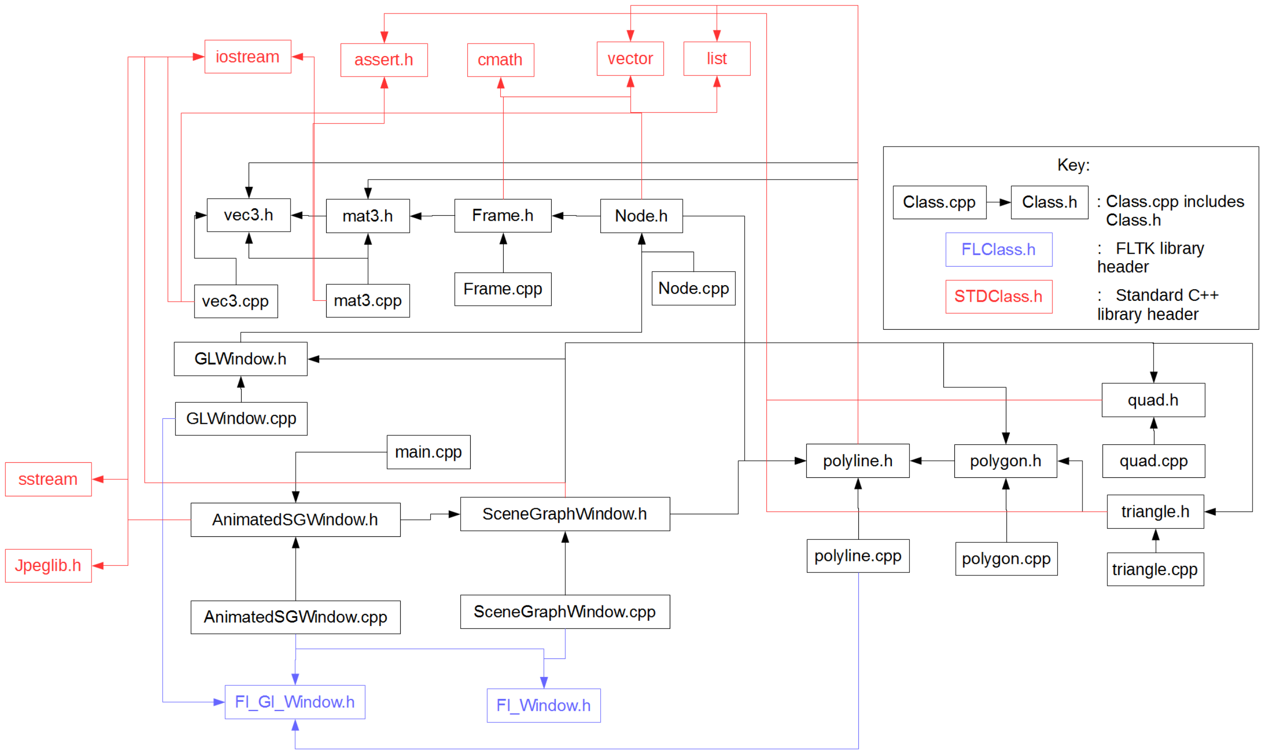
The main instance of inheritance is seen in the SceneGraphWindow class and its child class, AnimatedSGWindow. However, the geometries also utilize inheritance to share functionality. The AnimatedSGWindow was chosen to inherit from the SceneGraphWindow because it is a type of SceneGraphWindow that incorporates animation and a time-line widget. The common functionality of both classes allows for the AnimatedSGWindow to expand and override the SceneGraphWindow.

Following proper object oriented design, the geometry classes were created to inherit from one another because of their shared properties and functionality. A polyline is the base geometry with polygon inheriting from it. Because of its added property of a fill color and a closed curve, polygons can be viewed as a special type of polyline. The choice to make the triangle and quad classes children of the polygon class was natural because they are just specific cases of polygons, three and four-sided respectively.

The class structure could be improved, however. Namely the Frames could be more directly related to the AnimatedSGWindow because they are inherently designed for animation. Additionally, the design choice to change the Node class to use Frames could have be improved by simply sub-classing the Node class with an Animated Node class. This would allow the code to be more versatile, simple, and easier to read.

Currently, the major problem with the code as it stands is the length of each class. This makes it hard to read and modify the code if needed. Adding more sub-classes and abstractions in addition to removing of unused code would greatly relieve this problem.

Source Code:



The main design of the code is to make the headers include all the necessary libraries so other headers could include the ones created and get all the necessary libraries. For the most part, this worked out except for standard C++ library headers. The standard headers were over included in many cases, and while this does not hurt performance, it does make the code harder to read. Overall, most of the includes are efficient and only include what cannot be accessed otherwise.

The length of some files are somewhat unmanageable, so it might be beneficial to split classes up into additional source files making it easier to read and modify them. This could be done by splinting the largest classes, SceneGraphWindow and AnimatedSGWindow, into multiple files. One for the non-static methods, and another for the static callback functions.

Encapsulation:

Each class is designed to work independently of each other, incorporating the idea of encapsulation to package data members in the private section and allow public controlled access to them. Originally this was broken by my inclusion of the scene graph traversal in the GLWindow class, but this has been changed allowing the GLWindow to work independently of the Node class. The traveseSceneGraph method still uses public accessor methods, however, allowing for a controlled and safe traversal and insuring the scene graph is not modified.

Each class is split into two parts, the public section and the private/protected section. This allows for safe data protection as the data members of the class are stored in the private or protected sections. Most of the user interface classes, SceneGraphWindow and AnimatedSGWindow, store all their methods and callback functions in the protected section. This could pose future problems, however, because these methods are inaccessible to other classes that might try to interface with these classes. So if in the future a class was designed to use these classes to get an edited scene graph there would have to be major accessor additions to the SceneGraphWindow and AnimatedSGWindow classes.

The use of private, primarily over protected, does pose future problems as well. If in the future a class is designed to inherit from one of the classes with private sections, the use of private instead of protected would prevent the access of many data members and methods. This was encountered when designing the AnimatedSGWindow to inherit from the SceneGraphWindow because at the time the SceneGraphWindow put all of its widgets and other data members under the private heading. This prevented the access and modification of these values in the AnimatedSGWindow. Moving all the private members to the protected section fixed this problem while also preserving the protection afforded by the private heading.

Inheritance and Data Allocation/Deallocation:

The use of inheritance in the class structure is limited. Because of their shared data members and functionality, only the user interface and geometry classes were designed to make use of inheritance. However, there were instances where inheritance could have been used. Most notably, the Node class was modified to be backwards comparable while also incorporating new functionality including Frames and multiple transforms. Perhaps, however, there should have been an "AnimatedNode" class that inherited from the Node class and used Frames more efficiently leaving the Node classes original design in tact and adding more functionality through the new class.

If a classes was planned to be inherited from it included virtual methods so instances could dynamically choose which method to call. However, overall, there was little use of virtual methods and even less use of pure virtual methods. The most common use for virtual methods was in the deconstruction of classes. Knowing that a virtual destructor can prevent memory leaks all of the deconstructors were made virtual even if they did not have any content in there body. There are no pure virtual methods because all of the classes are non-abstract, but they could have been used to make an abstract geometry class that the polyline and polygon classes inherit from.

Many of the classes override the same method to allow default values and backwards compatibility. This is how the Node class was updated to be backwards compatible but include the use of Frames and multiple transforms. Each different constructor calls the main constructor with the default values in order to simplify and make the code more readable. Also, the destructors each delete all of the data members that are stored in the heap. Some call other classes' destructor or clear methods that delete all the data of a complex data type. Along those lines most data is stored as pointers to protect the contents when functions are popped and push off of the stack.

Overall, the use of inheritance could be better in order to limit the length of the code and increase readability. There are a lot of getting and setting methods that if the class was inherited could be eliminated or moved to the child class. The Node class, for example, could have the number of accessor methods halved by implementing a child class that uses Frames/multiple transformations.

Programming Conventions:

The interfaces are all well commented and include class and function description, but some of the implementations are long and include extra unused functionality. Many of the getter and setter classes are not used, but are included for future developments or modifications. In order to make the code more simple and easy to understand many of the classes could be separated into multiple files to split up the code by functionality. For example the callback functions could be put in a separate source file and this would greatly increase the quality of the code.

The naming convention is that class name are capitalized, except for primitive types (mat3, polyline, etc.), and are not too long. Variables are named to identify and make it clear what is being stored at the conceptual level. For FLTK buttons and callback functions the naming convention is the name followed by a “B” for button and that same name followed by “CB” for the corresponding callback function. The camel-back capitalization makes the names readable and the abbreviation are standard throughout. “SG” is used for scene graph, “ASG” is used for animated scene graph, and “Win” is used for different windows. An example of a descriptive but not too complicated name is the AnimatedSGWindow which describes what it is, a window for animated scene graphs, and what it inherits from, the SceneGraphWindow class.

As for function convention, the use of const and static is consistent. For all the member functions in a class, if they do not set memory they are declared const and return a constant instance as specified. The use of static functions allow for a consistent and expected functionality not dependent on the single instance or having a modifiable instance. The only instance where this is broken is the interpolate frames function in the Node class. This recurses through the tree and interpolates the frames between key frames. This function is not static but because there only needs to be one instance of this function, perhaps it should be declared as a static function.

There are no instance of global/static variable in use because there is no need to do so. If a global variable is needed it can most likely be stored as an instance data member or as a defined macro for the preprocessed to substitute into the code. It is bad practice to use global variables because they could be modified outside of the class and could lead to unwanted results.

Overall the style is consistent and that adds a lot to the readability of the code. The comments are uniform and are present for every method or block of code. The conventions are used throughout and go a long way in making the code usable as well as modifiable.