Virtual Worlds Engine/Manager

Introduction

The Virtual Worlds application uses the openFrameWorks version 0.12 application framework system as a basis for its core operations and boiler point coding. For information on openFrameWorks beyond what is explained here, it is best to visit the openFrameWorks web page and documentation, even if is not all to good in explaining itself.

The openFrameWorks design and coding matches closely to that of the completed prototype of Virtual Worlds and that influenced the choice of using it to advance to the next level of creating a workable application. The documentation given here does not use many a words as it is a documentation in graphical and note form for reference.

01 : Application operation

Application base Class ofBaseApp.

The execution and running of the application is implemented and managed using the openFrameWorks application classes. Without any detail which has not been explored, openFrameWorks utilises a base class called ofBaseApp that is passed into a function called ofRunApp. OfRunApp is a member of an application header file called ofAppRunner.h which has a series of other application related functions. OfRunApp seems to execute an initialise phase and then run the ofBaseApp class in a loop until an exit signal is given to exit this loop.

The OfBaseApp class is entirely made up of virtual functions that are seemingly executed within each loop cycle of an application run cycle. Of these virtual functions the ones that the user needs to be mainly concerned with are

application initialisation

Setup() : Perform initial setting up of the application variables, structures and data before the main application run time loop starts

application runtime loop

update() : Update application data
draw () : Perform draw tasks to display GUI widgets and graphics
exit() : Perform tasks to exit the application.

There are other virtual functions, but these can be ignored for the large part as they have default application functionality that need not be modified for the application to run.

To execute an openFrameWorks version 0.12 application however, all of the necessary include files needed to create an application and run it are defined in one ofmain.h file. Without including this file in the C++ main application file the openFrameWorks framework classes and functions would not be accessible and the application would not able to be compiled or run.

In Brief
1: The C++ application header file to define the application class needs have the openFrameWorks file ofmain.h included at the top.
2: The ofBaseApp class needs to be inherited by the user defined application class.
3: This user defined application class has all the application functionality defined with the virtual functions setup, update, draw and exit functions to be executed in an application runtime.

Steps 2-3 define the openFrameworks application class to be executed, which has the form in an application header file.

```
#include  ofmain.h

class user_application_class : public  ofBaseApp{
public:
      void setup(){
      }

      void update(){
      }

      void draw(){
      }

      void exit(){
      }
}
```

but to run the application using ofRunApp, this function is called with the applications entry point, usually called main.cpp file

This main.cpp file has the form

```
include user_application_class.h// defined in steps1-3

int main( ){
    ofGLWindowSettings settings;
    settings.setGLVersion(3,2);
    settings.setSize(1400,1200);
    settings.setPosition(glm::vec2(100, 100));
    settings.windowMode = ofWindowMode::OF_WINDOW;
    // More window settings here

    user_application_class *app = new  user_application_class;

    ofCreateWindow(settings);

    // this kicks off the running of my app
    ofRunApp(app)
}
```

4: This user defined application class is passed to the function ofRunApp within the C++ application entry to run the application.

With steps 1-4 Completed, and the user as a first step having the virtual functions setup(),update(), draw(), and exit() empty compilation and running this should bring up a blank screen with a console.

What follows is the Virtual Worlds application engine and management that is implemented within this openFrameWorks application framework.

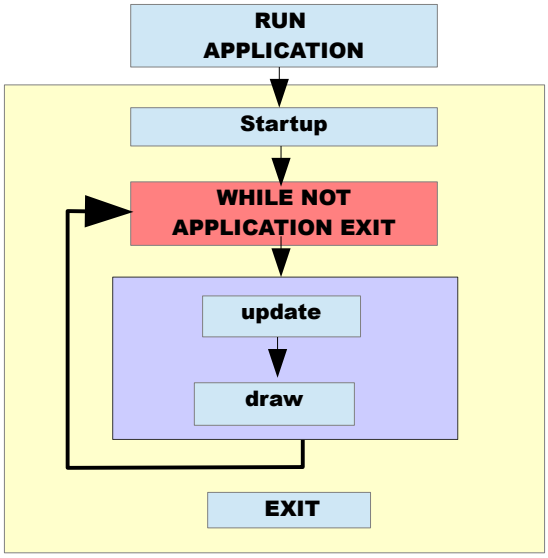

Fig 01.01 Schematic of Virtual Worlds/openFrameWorks overall application operation

02 : Virtual Worlds Loop Cycle

The Virtual Worlds application tasks to be performed per loop cycle as illustrated in **Fig 01.01** can be summarised for each loop cycle function update, draw and exit as

update()

draw()

ImGui setup
Display log panel
Display parameter panel
Display node editor panel
Display timeline panel

Render scene in view port

Draw view port overlays.
ImGui exit

exit()

table 02.01

As can be seen in **table 02.01** the Virtual Worlds application draw routine does not as of this writing directly perform any application updates or exit routines, and in essence performs the tasks of displaying application panels and rendering the scene to the view port. All the user interaction and performing of tasks is done indirectly within the parameter panels displayed.

The displaying of panels as part of the draw routine

03 : OpenFrameWork Operation

OpenFrameWorks works similar to the ImGUI framework by setting up and exiting its various 3D attributes within the draw task of the application (Fig 01).

An example is the openFrameWork easy camera class ofeasycam.

To use the camera and display what it “sees” to a view port, it has a begin() function that executes routines to set it up for use, and an end() routine to perform tasks to release it and have the application not use it..

What this means is that an openFrameWorks camera is only valid to display 3D objects and anything else that the application renders to the view port that is between this camera begin() and end() block of code.

eg. Assume that an application camera of this camera class is called Camera. Then only code that is between the Camera.begin() and Camera.end() will be seen in the view port that this camera is designated to display to. If any drawing of 3D objects is performed outside this code block, it is not rendered to the view port that the camera is set to display.

This applies to most things 3D in OpenFrameWorks. 3D objects, materials, lights objects, shadows etc. One can think of it rather like the scope of a C++ or other programming language function or code block where the scope of a block block determines the relevancy of functions, variables etc for that code blocks execution.

Fig 03_01 gives a schematic outline on how openFrameWorks operates. object01, object02,object03 will be drawn to the view port that Camera is assigned to while object04, will draw to nothing. object01,object02, will have material assigned to them in the rendering process while object03 will be what ever default material is given. Each object will have their locations set except object03 because the setlocation function is outside object03 scope ie not within the object03 begin() end() block.

Much of openFrameWorks operates like this, and hidden within the background are setting up and tasks to perform and manage the data and rendering to a view port 3D and other preform other features.

```
Camera.begin()

material.begin()
        object01.begin()
              object01.setlocation(x1,y1)
              object01.draw()
        object01.end()

        object02.begin()
              object02.setlocation(x2,y2)
              object02.draw()
        object02.end()
material.end()

object03.begin()
      object03.draw()
object03.end()
object03.setlocation(pitch,yaw,roll)

Camera.end()

object04.begin()
      object04.setlocation(x4,y4)
      object04.draw()
object04.end()
```

Fig 03.01 Schematic of openFrameWorks operation

Thus in the Virtual worlds design and implementation when using openFrameWorks as a back end, where performing these begin and end routines will be critical. Thus a designated order of operation of objects and routines such as for cameras, lights and objects etc must be taken into consideration.

04 : Virtual Worlds Scene Class

The scene class is the Virtual Worlds application central engine class that is the main gateway to store and manage the applications data, and perform the setting up and tasks associated with rendering a scene. The Virtual Worlds application thus revolves around the scene class for all its functioning and it sits at the heart of the application functionality.

Therefore, the Virtual Worlds scene class needs to perform and manage the setup and exit routines for the openFrameWorks class objects that is explained in **section 03** for rendering an entire scene.

The Scene class incorporates a data manager class to perform all the scene geometry data storage and management tasks as well as storing the global settings and attributes for performing the setup and rendering management of the scene.

By following the openFrameWorks examples and limited documentation, to render a scene using openFramWorks the following order of procedure needs to be performed to render a scene.

setup scene lights
setup scene camera

setup scene objects
render_scene_objects
exit scene objects

render_scene_overlays

exit scene cameras
exit scene lights

Through testing of code, the implementation of displaying shadows and using the openFrameWorks material class was proven to be successful by implementing it within this class. However, it was then chosen not to be included in the rendering of a scene as displaying shadows can slow the rendering of a scene to a crawl, and that the materials class would never be used for a procedural point cloud object. Also these were not part of the goal or important of the Virtual Worlds application.

Therefore the Virtual Worlds scene class is to perform the above rendering of a scene as a function and act as a gateway to the data storage manager class.

To perform the task render scene in view port of **table 02.01** a function render_scene() of this class performs this task.

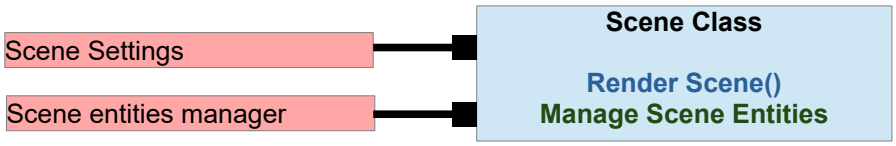

Fig 04.01 Schematic of Virtual Worlds scene class

The C++ code for this scene entity manager class is defined as the C++ class vw_scene_class located in the directory Source→VW_framework→scene within the file

vw_scene.h

05.0 : Virtual Worlds Scene Entities Manager

The Virtual Worlds entities manager is the central engine class that stores and manages all the applications scene objects data.

The scene objects that are always present in a scene are lights and cameras. Lights are needed to define how a scene is rendered with shadow and other effects to bring the scene objects to life as 3D objects, and cameras to define a view point from which to render images of the scene from. These objects can have their own class and in do in the openFrameWorks system. ofLight for lights, and ofCamera and ofEasyCam for the camera.

The ofCamera and ofEasyCamera classes were insufficient for the Virtual Worlds project. So a copy of these classes was made and modified and was able to be successfully incorporated into the Virtual Worlds application without any issues. These, thus far as of this writing are the only openFrameWork classes that have been able to do this without disturbing the original openFrameWork source code. All other attempts have created linking and application runtime crashes.This camera class is called vw_camera_class.

To manage multiple lights and cameras that may exist in a scene, a manager class for each of the lights and camera class was created to perform all the storage and management of lights and camera entities in a scene. These managers then act as a subset of the main entities manager class.

Given that the lights and cameras have there own managers, it is fitting that all other scene objects will need to have their own manager system to deal with their own individual and unique entity data type. A unique entity data types are called category, and thus there needs to be a single manager class that can perform all the management tasks on all categories.

To do this, there needs to be a single base data class type that can be used to represent a data type of any kind. Such a base data class is C++ base class of common virtual functions that perform the tasks of storing, accessing and manipulating data of any type or structure. The category manager has the task of storing, and managing the different categories in just the same manner as the lights and camera manager classes. The category manager is an expression of a more general manager of the lights and camera manager, and thus all have the same functionality and management routines. Since the light and camera objects can be considered as a category of a 3D data object, then these objects can be also used as a template to create a universal C++ base object class that can represent any 3D object.

Both the light and camera object are of the openFrameWorks origin and use. Both inherit the openFrameWorks ofNode class. As stated in the openFrameWorks documentation, the ofNode class is a basis class of all things 3D. ofNode manages aspects of location and orientation of a 3D object and creates the necessary transformation matrices for displaying 3D objects to a view port. Thus any object base class would also require to inherit this ofNode class.

The other common functions required in a scene editor to manage a 3D object through a object base class of any data type and render it into a view port are discussed in more detail in section 07.

Fig 05.01 gives an overview schematic of the scene entities manager class.

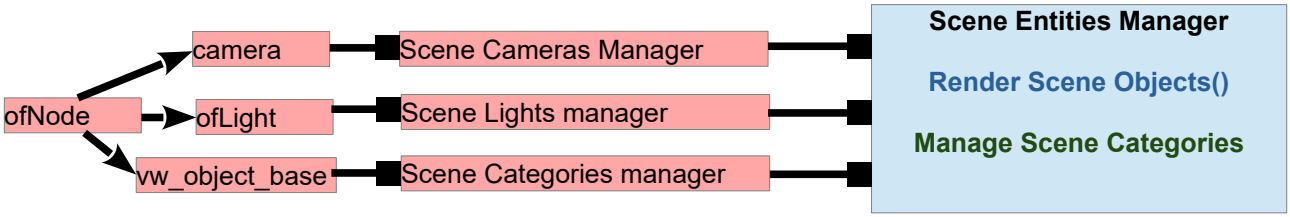

Fig 05.01 Schematic of Virtual Worlds entities manager class

The scene cameras, lights and categories managers perform all the same tasks of managing their respective base data types in creating, deleting, retrieving of data etc.The scene entities manager manages the tasks of creating, deleting and retrieving data of each category or entity data type.

The C++ code for this scene entity manager class is defined as the C++ class scene_entities_manager_class located in the directory Source→VW_framework→scene within the file

scene_entities_manager.h

05.1 : Scene Categories Manager

The scene category manager performs all the management tasks that is expected of a data manager. Create, delete, retrieve stored data of the particular type of data stored. The scene category manager has as a storage data object, a base object data class, that can reference a scene data type of any kind and functionality (see section 05.0 and 05.2). Each scene category has it own unique ID and name and description to identify and reference a data type by, but has common function names to reference and perform tasks, even if those functions perform tasks on different types of data using different code.

Fig 05.1.1 is a Schematic of how the scene objects are managed. The Scene entities manager manages one or more scene category data types, and each category data type references the same object data base type, of which is inherited by a class that has at it core a data type and code for each data category.

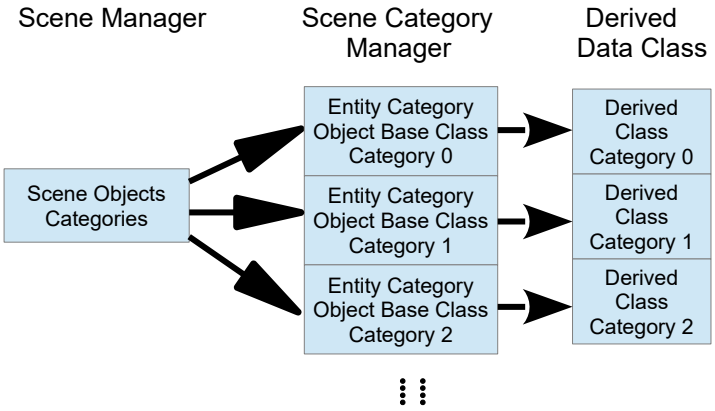

Fig 05.1.1 Schematic of Virtual Worlds Scene categories data management

By use of an object base class, the scene entity category data management is simplified by only needing a pointer to an object base class and not the derived class. Thus all the management tasks of the scene category manager only need to be concerned with a base class object and not any derived class.

The C++ code for this scene entity manager class is defined as the C++ class vw_scene_objects_category_class located in the directory Source→VW_framework→scene within the file

scene_object_categories.h

05.2 : Object Base Class

In the section 05.0 and 05.1, it was mentioned that the scene category manager manages data objects that are a derived 3D class object of the C++ base object class. This C++ object base class consists of virtual functions and variables that all 3D scene objects of a designated type use to store 3D data, manage that data, perform tasks and even render the 3D object to a view port. To perform perform all these functions, the openFrameWorks framework system is utilised.

Within openFrameWorks, there is a ofNode class which is, as stated in the openFrameWorks documentation, the basis class of all things 3D. OfNode manages aspects of location and orientation of a 3D object and creates the necessary transformation matrices for displaying 3D objects to a view port. However, this class was considered incomplete and had additional functions added to it to suite the needs of the virtual worlds project.

Additions made to the ofNode class was an entity base geometry class to store and display 3D data, and functions to modify and manage that 3D data. It was desired have a derived class of ofNodes to have these additional functionalities added and used, but it was discovered, as with many things to do with using openFrameWorks, this created problems that could not be resolved when trying to render 3D geometry or perform tasks on geometry data. To avoid external dependency of the geometry class being outside of the openFrameWorks project, this geometry class and anything needed by it was added to the openFrameWorks directory structure. An additional directory called geometry was added with the C++ geometry class code files.

This modified ofNode class was thus a base class that would be used as part of any 3D objects base class, but it was not a complete Virtual Worlds object base class as it did not have the C++ virtual functions and variables specific to the Virtual Worlds application required to be able to manage and display 3D objects. The object base class would inherit this modified ofNode class and the virtual functions of the object base class would reference the geometry data, variables and functions of the modified ofNode class to manage and display 3D objects.

Another function of the object base class is to store a pointer to a parameter widget base class such that if the 3D object that this object base class is selected by the user, a parameter widget displaying all the parameter data and ability to perform functions to modify or create the geometry data and its displaying in the view port is available.

The 3D objects material is also stored and available for a parameter widget to access and have functions to modify for rendering purposes.

A schematic of the object base class is given in **Fig 05.2.1**

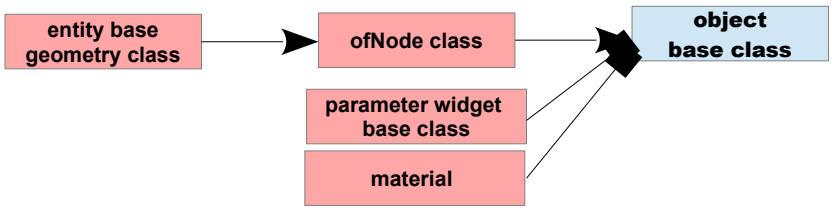

Fig 05.2.1 Schematic of an object base class.

The design for this class as for the entire application, is to have a base class of common virtual functions of a base class can have totally different code for each of the derived classes to perform the same task with only needing to reference the base class.

The C++ code for the base object class to manage and store a 3D objects and render that 3D object to a view port is defined as the C++ class vw_object_base_class and is located in the directory VW_framework/object in the file

vw_object_base.h

05.3 : Geometry Base Class

Render System

The rendering of entity objects is managed and performed through the OpenFrameworks openGL framework system. OpenFrameWorks is used to create and handle all the buffers and other C++ openGL functionality needed to copy vertex and other data to the GPU.

The Virtual Worlds engine performs the creation of glsl programs and management of glsl uniforms and graphical data to be utilised in those glsl programs.

Rendering Objects

The rendering of objects in Virtual worlds is done by adding a custom geometry base class to the OpenFrameworks framework system. This geometry base utilises much of the functionality that openFrameWorks uses to render objects in a scene. Thus much of what is in this base class is a copy or modification of openFrameWorks code with some additional custom code that is needed and required to render objects using OpenFrameworks.

The geometry base class can thus have a derived class created to display and render any kind of data and/or be added to any data object as a means to reference the base class virtual function to render data to a view port. This same geometry base class stores and manages vertex data and the shader program ID to use to render to the view port. Thus to render vertex data to a view port, one base class geometry function common to all scene objects need to be referenced.

Fig 05.3.1 gives an overview schematic of how the vertex geometry data is managed and rendered. The lowest base class is the openFrameWorks ofMesh class that is inherited by a custom geometry base class. The ofMesh class according to the openFramWorks documentation represents a set of vertices in 3D spaces, and normals at those points, colors at those points, and texture coordinates at those points. Each of these different properties is stored in a vector.

The geometry base class performs the management of vertex data that defines any 3D object. The base geometry class is inherited by an entity base geometry class which handles and manages the rendering of the vertex data of the base geometry class. The reason for having these two base classes is to be able to define geometry data for an object separately from any rendering. This is important for such actions that do not require openGL such as generation or editing of vertex data.

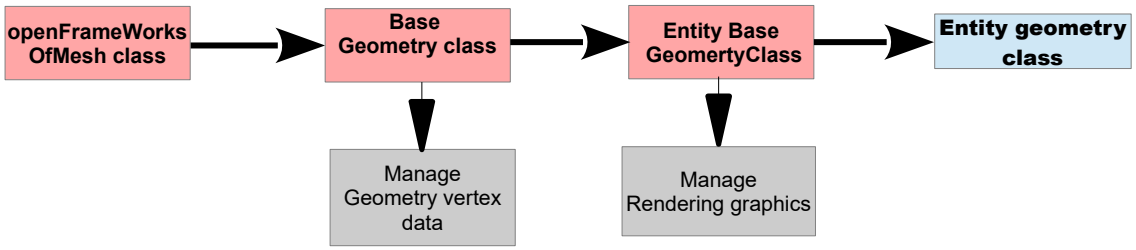

Fig 05.3.1 Entity Base Geometry class

The C++ code for the base geometry class to manage and store a 3D object and render that 3D objects to a view port is defined as the C++ class base_geometry_class and entity_base_geometry_class located in the directory openframeworks→geometry within the files

vw_geometry.h
and
vw_entity_object_geomerty.h

05.4 : Parameter Widget Base Class

As mentioned in section 5.2, and through out using the virtual worlds application, the user needs to view and interact with a GUI that displays parameter data and GUI widgets to enter, modify and perform application tasks. These are referred to as parameter widgets. Through out the application's usage, these parameter widgets are all different, and are all classes of different types, thus to simplify the display and management of these parameter widget classes, a common parameter widget base class is required that can be inherited to perform the task of displaying any parameter widget.

To display the parameter widget for a particular 3D object type, a pointer to that object type is required. And it would be better if the object type is an object base class that all 3D objects inherit. Thus a pointer to an object base class is required and virtual functions defined within the parameter widget base class be enable the parameter data for a particular 3D object to be displayed and tasks on the object 3D performed.

Fig 05.4.1 gives a schematic of the relationship of a parameter widget class.

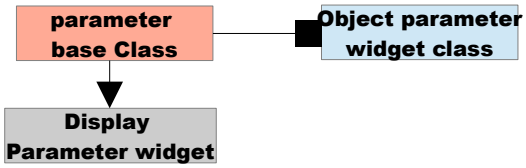

Fig 05.4.1 Schematic of a parameter widget base class

The parameter widget base class is a virtual worlds framework class and exists in the directory

FW_framework→Widgets

with file name

parameter_widget_base.h

5.5 : Virtual Worlds Scene Object

A virtual worlds scene object is the data object class that is a derived object base class that was described in section 05.1. This scene object consists of all the base classes in sections 07 to 09 that all have common virtual functions, variables and structures that are designed to store, access, manipulate, render and manage a 3D data object of any kind.

For the user to create a virtual worlds 3D scene object, the user needs to create the following classes first.

i. Create a scene object class that inherits the vw_object_base_class class (**section 05.1**) utilising the object base class virtual functions and variables etc..

ii. If not already created, create an entity category class for which to store the created scene object class in i. To do this the user needs to define a unique category name (the ID to identify the category) and an optional description. No two categories can have the same name.

iii. Add the scene object class created in i to the scene category class created in ii

Then the user needs to

iv. Create the parameter widget panel(s) class (**section 05.4 : Fig 05.4.1**) that will be used to display the scene object parameters and perform tasks for such things as the creation or modification of the geometry data or its display utilising the parameter base class virtual functions and variables, etc.

v. Create and define a pointer to a newly created parameter widget created in iv, and assign it to the scene object class parameter_widget pointer variable created in i.

vi. If not present, create an entity geometry class that inherits the entity base geometry class (**section 05.3** : **Fig 05.3.1**) to store and manage the geometry data and the rendering of that geometry data to a view port for the desired scene object data type utilising the entity base geometry class virtual functions and variables, etc.

vii. Create and define a pointer to a newly created geometry base class created in vi, and assign it to the ofNode class geometry pointer of the scene object class object created in i.

Steps i-vii above need not occur in the order given, but it is much easier to to so as testing of the various elements as they are created .can be performed.

Following the above steps, a scene object of any type can be referenced, modified and displayed on a view port by accessing the standardised base class virtual functions and variables that defines the scene object.

To see the code that performs these tasks can be examined by following the directory path name and file name given for each section above.

6.0 : Virtual Worlds Node Editor

As much as the scene class is the center of the virtual worlds data storage, management of scene data, and rendering of that data within a scene, the node editor is the central user interface to create and access the scene objects and the data that exists within the scene class as one or more nodes on a computer screen.

The Virtual Worlds node editor appears and functions like most node editors having a box or other shape representing some kind of data, or function. These nodes can be linked together to perform some function or task represented as lines between nodes, and have this functionality performed by changing GUI widget settings.

The question is, how can this be done.

Since this application uses the ImGUI system for its GUI, it is fitting to use an existing node editor based upon ImGUI since it was better not to reinvent the wheel. Two were found. ImGUI-node-editor by Michał Cichoń and ImNodes by Johann Muszynski
.
After some testing and consideration of what was needed for virtual worlds. By following the examples given for ImNodes, and through extensive testing to understand how to code for executing a node editor using ImNodes, it was selected to form the basis of the node editor as it seemed the simplest and easiest to use and customise.

The virtual worlds node editor had one design requirement. It needed to be able to spawn multiple node editors so that a design concept of a node group is to be represented as child node editor of a parent node editor. ImNodes in its raw form could not do this as it used the ImGUI method and ethos of design to be used as part of, and integrated into the ImGui system. It used a confusing, inefficient and what appeared to be clumsy method of multiple node editors.

However, by modifying ImNodes to be a node editor base class object with some additions and minor modifications, this problem was solved. All the functionality and code of ImNodes was still present and the implementation and use of ImNodes retained as it was coded to do. The only difference being that ImNodes is a C++ base class that is to be inherited into a node editor class and have all its functionality performed as a class object rather than being used within a node editor class as an ImGUI extension.

This node editor base class is called node_editor_basis_class and the source for it exists in the directory

Source→FrameWork→Universal_FW→ImNodes→Kernal

with files that contibute to the node editor

node_editor.h
node_editor.parameters.h
node_functions.h

This base class on its own is not enough for a functioning node editor. ImNodes is only a node editor engine to create the GUI node editor interface, display the graphical elements of the node editor, and determine if a node, or any node element is selected, has a mouse pointer hovering above an editor element, or is interacted with such as creating a node editor link.

06.1 Node Editor Management

The creation and functionality of a node editor in all its form is determined by the node editor class that inherits the node_editor_basis_class. This derived class in turn depends on a node editor data management system and classes to create, display and manage nodes, the links between them, and the pins or ports that the links connect to.

,A design was developed that followed the examples given to implement an ImNodes based rudimentary node editor, and expand on them to suit the needs of the virtual worlds application.

A base node editor using ImNodes requires

i. A node class to define and perform the tasks to display a node, and its input and output port or pin along with the pin ID data that are used to define the linking of input and output of data to and from the node.

ii. A pin structure to define the parameters to display the class pin, store the node that the pin belongs to and any link ID that are designated to be associated with that pin.

iii. A link structure to define the parameters to display the class link, store the to and from pin IDs that

are designated to be associated with that link.

iv. A nodes class to store and manage all the node data in a node editor

v. A pins class to store and manage all the pin data in a node editor

vi. A links class to store and manage all the links data in a node editor

vii. A graph class that acts like a central data manager and organiser of the nodes, pins and links classes to perform all the management tasks and coordinate the data for a node editor.

For the graph class to be able to perform the tasks set out in iv, it needs to have the nodes, pins and links classes to be part of it. Fig 6.1.1

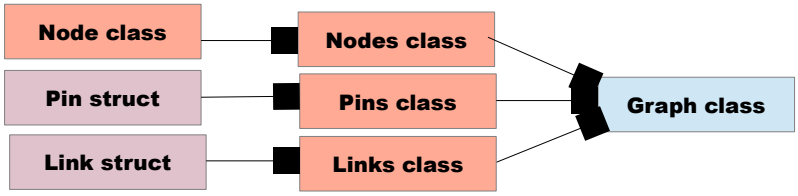

Fig 06.1.1 Schematic of a node editor graph class

The pins and links of a node editor do not change in their functionality or purpose, and thus are constant and unchanging structures rather than classes as they hold data and display settings more than performing any function. However, a variety or diversity of nodes can be expected to be be present and implemented. Thus there are more than one type of node that will be present in a node editor, but for the node editor to handle only one type of node that can have a diversity of appearance and functionality, and to be created and managed that, node needs to be a node base class of common virtual functions and variables that are to be used and managed by the nodes and graph class. Thus for the Node class of Fig 6.1.1. is replaced with a node base class. Fig 06.1.2

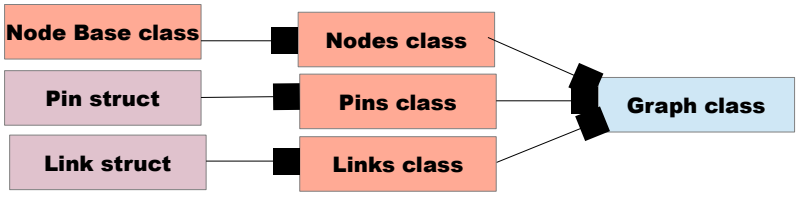

Fig 06.1.2 Schematic of a node editor graph class

What this means is that when a node of any kind is created, it must have the node base class inherited, and then have the same named virtual functions of the node base class defined and coded to perform the tasks and functionality desired for that node. This derived node class is then assigned as a node to be used in the nodes editor.

This node editor element classes exists in the directory

Source→FrameWork→Universal_FW→ImNodes→Kernal

with file names

node.h
node.cpp
pins.h
links.h
graph.h
graph.cpp

and class names node_basis_class, nodes_class, pins_class, links.class

06.2 Group Node

One derived node of the node base class described in section 06.0 that is to be used, but not to be modified or used as a basis to be inherited in any way is the node group node. The node group node is a node that links a group of nodes defined to exist within a child node editor of the node editor that the node group node exits within. It is best to think of group nodes acting similar to folder or directory icon on the windows GUI, where if one double clicks on it, it opens up a window displaying the contents of that sub directory. But in the case of a group node, double clicking on it with a mouse right button displays a child node editor of the current node editor with all the nodes that is assigned to that child node editor.

The main purpose is to organise the nodes of a scene into a more simple hierarchy of node management so

the user can organise the potentially high number of nodes into a more workable environment.

The group node is designed to be able to carry links into and out of the child node editor, and act as a means to create node functionality of a group of nodes in the child node editor as if they are a single node.

The group node on its own does not achieve this functionality, it is merely a means to display a group node that the node editor will perform the functions of calling the node editor it is assigned to, to be displayed on the screen, or to manage in some manner.

This node group node class is called node_group_class and the source code for it exists in the directory

Source→FrameWork→Universal_FW→ImNodes→Kernal

with file name

node_group.h
node_group.cpp

NOTE : It is found that when compiling virtual worlds, the node group needs to be compiled separately before compiling the entire project, otherwise linking errors will appear. If such linking errors do appear referencing the group node functions, then compile the group node .cpp file before compiling and linking the project again.

06.3 Node Editor Class Design

Using examples given for creating a rudimentary node editior using ImNodes, a design for creating a scene node editor class was developed. The Scene Node editor is the first and major form of user interaction with the application and has a floating menu system for the user to select nodes and perform tasks. The functionality of the node editor in principle is illustrated in **Fig06.3.1**.

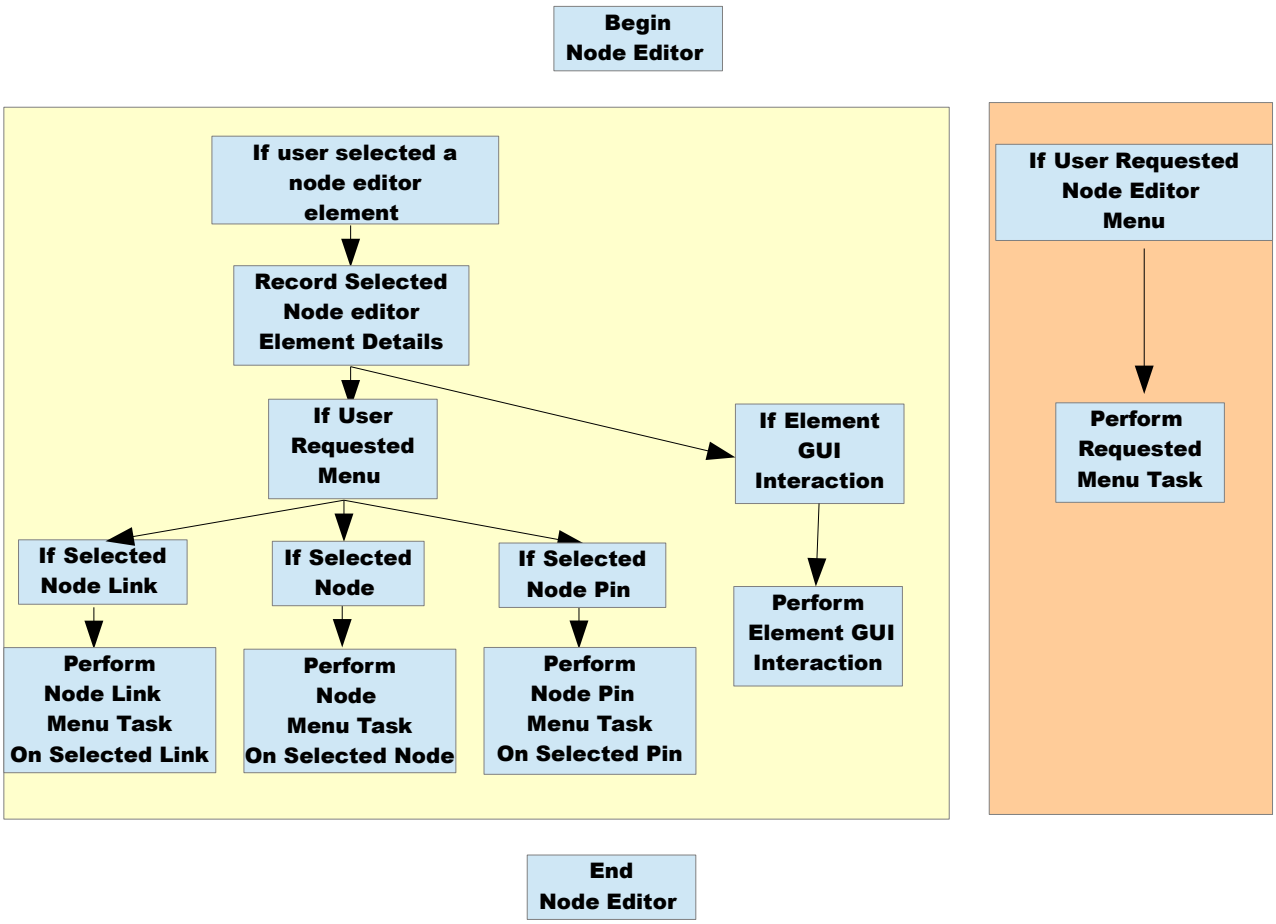

Fig 06.3.1 Schematic of Virtual Worlds Node Editor operation

The Node editor essentially consists of two main sections as illustrated in **Fig06.3.1** . A section handling the selection, and the management and editing of node elements and any application data associated with those node editor data elements as illustrated within the yellow box on the left. A floating menu system to create node editor nodes and other menu tasks when no node editor element is selected as illustrated within the orange box on the right.

The Begin and End node Editor are functions that are performed to set up or start and close the node editor as part of an application loop cycle indicative of the used ImGUI interactive GUI that the application uses. Even though **Fig 06.3.1** may appear to operate in parallel this is not the case in implementation, it is in

sequence.

06.4 Node Editor Node Base Class

The Node editor in section 6.3 has at its core the function to create, interact with and delete the nodes that are displayed within the node editor window. Each node represents either one of a 3D scene data entity, a function or operator to modify or perform a function on one or more 3D entities, or some other aspect required by the virtual worlds application. So when a user selects a node in the node editor, information on what node is selected is given to the application, and information pertaining to the selected node is displayed in a parameter panel (to be discussed in section XXXX) The user can then interact with this parameter panel to modify settings and perform or modify functions related to that selected node.

Since there can be a variety of different nodes that can be displayed of infinite possibilities, as discussed in section 06.1, the node editor needs to be designed to interact with one common base node class type otherwise it becomes unmanageable.

First and primary are the fundamental requirement needs for the node editor are things like

Parameter data
node id
node label
list of input pin id
list of output pin id
node display settings

Functions
drawing node
link connecting/disconnecting
etc.

which are mandatory for all nodes and thus require no virtual function.

Without going into detail, as the virtual worlds application evolved, this initial node base class expanded to need the requirements of virtual functions and related variables or data structures to perform such tasks as

Parameter data
scene entity data id
scene entity data category
pointer to the graph class the node exists within.

Functions
defining the node's data to be stored in the application scene (section 5)
The GUI parameter widget for the user to interact with to modify settings and perform tasks
saving/importing data that construct a new node into the node editor
define a floating menu for the node editor to display and perform tasks regarding the node.
On Link connection/disconnecting functions
procedures on deleting the node such as deleting data stored in the application scene
getting data from a linked node
etc

This is not a complete list, but gives an idea that a node of any functionality can be created to be used within the node editor and managed easily with minimal coding for any purpose by inheriting this node base class.

To use a derived node class within the node editor, that derived node class header file needs to be included as part of the node editor class, and a menu entry is added to the editor floating menu to create the derived node in the editor.(Right box of Fig 06.3.1) Virtual functions in the derived node class will create and perform all the necessary tasks to define any data to be stored in the scene data base, display a widget in the parameter panel representing the node, and drawing the node in the node editor window.

It must be noted that only nodes that are derived from the node base class can be displayed and used in the node editor. Otherwise the application will crash on attempting to create a node.

Selecting and operating on a node in the node editor uses the defined virtual functions that are coded in the derived node class to perform application tasks. This give great power and the flexibility to create any node of any type in the node editor and have the editor treat is a node of a single type.

Fig 06.4.1 Schematic of Virtual Worlds Node Editor Node Base Class Relationship

06.5 Creating a node to use in the node editor

Before creating a node class to be used in the node editor, what must be critical is to know what the purpose and function of the node. Once this is determined, and what code is required, then one can proceed.

The general steps for a node to create a scene geometry entity is given as an example as it is the most complex and is the most involved.

1 Create a file and code to define a node class object of a given name.
2 Include the node.h file and have the class in i inherit the node_base class
3 Write code to define the node for the virtual function define_node.
  3.1 For a scene entity geometry object, this will involve including code to define and create a scene entity data object, store it in the scene entities manager (section 05.0) and have a pointer defined to point to this entity data object.
  3.2 For all nodes, this is where the node appearance parameters are defined and the location of where in the node editor the node will be displayed.
4 Write code to perform tasks necessary when the node is deleted in the node editor in the virtual function delete_node_entity()
  4.1 For a scene entity geometry object, this will involve including code to delete the scene entity data object that this node represents, from the scene entities manager (section 05.0) and have the pointer defined to point to this entity data object set to null.
5 Create a ImGUI menu to be displayed in the node editor when this node type is selected and a signal in the node editor is initiated to display the menu for this node type. The virtual function editor_menu_options() is where this is node menu code is located
  5.1 If these menu options involve functions or virtual functions that do not exist then, create the functions and code for the menu to reference.
6 Within the virtual function define_ui, define the global setting values so that the application will retrieve from the scene entity manager data to display in the parameter data widget, for the scene entity object data that a node of this node type represents.
7 Define within the virtual function display_ui define the ImGUI widget to display the data that the node is associated with and any functions to be performed.
8 Define the code to export this node and relevant associated parameter data to a data stream. The virtual function export_node(std::fstream &stream) does this task.
9 Define the code to import the same node data and parameter data from a file. The virtual function import_node(std::vector<std::string> lines, int& line_number) does this task.

Steps 1-8 above are the most important but, not complete steps needed for the node editor and application. Omissions from these steps are things like the creation and deletion of timeline links, which are mentioned in section YYYYY on the animation or timeline.

Because of the wide variety of nodes and scene data types that can be associated with a node, along with user preferences in writing code, no greater detail is given here on how to create the code for a node to be used in the node editor.

07 : Parameter Widget Base Class

Through out using the virtual worlds application, the user needs to view and interact with a GUI that displays parameter data and GUI widgets to enter, modify and perform application tasks. These are referred to as parameter widgets. Through out the application's usage, these parameter widgets are all different, and are all classes of different types, thus creating a case that to simplify the display and management of these parameter widget classes a common parameter widget base class is required that can be inherited to perform the task of displaying any parameter widget.

To display the parameter widget for a particular 3D object or node type that can use it to display settings data or have access to perform tasks , a pointer to that object type is required. And it would be better if the object type is an object base class of common virtual functions, data structures and variables so that a function can reference one single base class datatype. As a base class keeping in step with the ethos of design for the virtual worlds application this base class can be inherited by any parameter class and then referenced to display any kind of parameter data widget.

To display data for a scene object, a pointer to the object base class (section 5) is required and virtual functions defined within the parameter widget base class be enable the parameter data for a particular 3D object to be displayed, and tasks on the object 3D performed.

Fig 07.0.1 gives a schematic of the relationship of a parameter widget class to a 3D object base class.

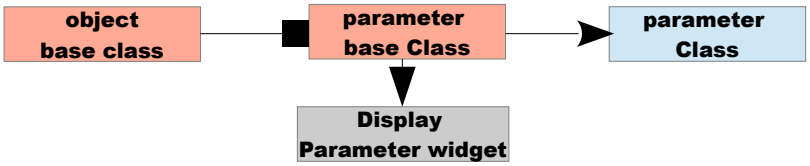

Fig 07.0.1 Schematic of a parameter widget base class inherited into a parameter class

The parameter widget base class is a virtual worlds framework class and exists in the directory

FW_framework→Widgets

with file name

parameter_widget_base.h

A parameter class can also be used by a node class (section 6) to display that nodes data settings and perform tasks. A parameter class is called and any relevant variable settings defined.

The parameter widget base class has two virtual functions.

display_parameters_widget()
set_parameter_widget_object data()

and as mentioned above a pointer to an object base class that exists within the scene entities manager.

vw_object_base_class *object_to_execute;

By a class inheriting this base class, calling the virtual function display_parameters_widget() will display a parameter widget of any form, and if defined and referenced, a 3D object data type of any kind that can be modified through user created functions in the derived parameter class.

In table 02.01 of section 2, a parameter panel is displayed as part of the application’s cycle loop. The current selected node in the node editor, if it is a 3D scene data type object stored in the scene categories manager, a parameter widget inheriting the parameter base class will be displayed in the parameters panel.

08.0 : Application Time Line

The application time line is in effect a widget class to display time line operations in a frame step mode that are represented as tracks within the widget. The user can select from within the widget, ImGUI buttons to control the frame step actions of tracks that have been selected for the time line widget function to act upon. The user specifies a time line interval in start and end frames over which the actions are to be performed. Within each track the user can refine the actions to take place depending upon the type of track being displayed and what data. A schematic of the Time line operation is given in **Fig 08.0.1**.

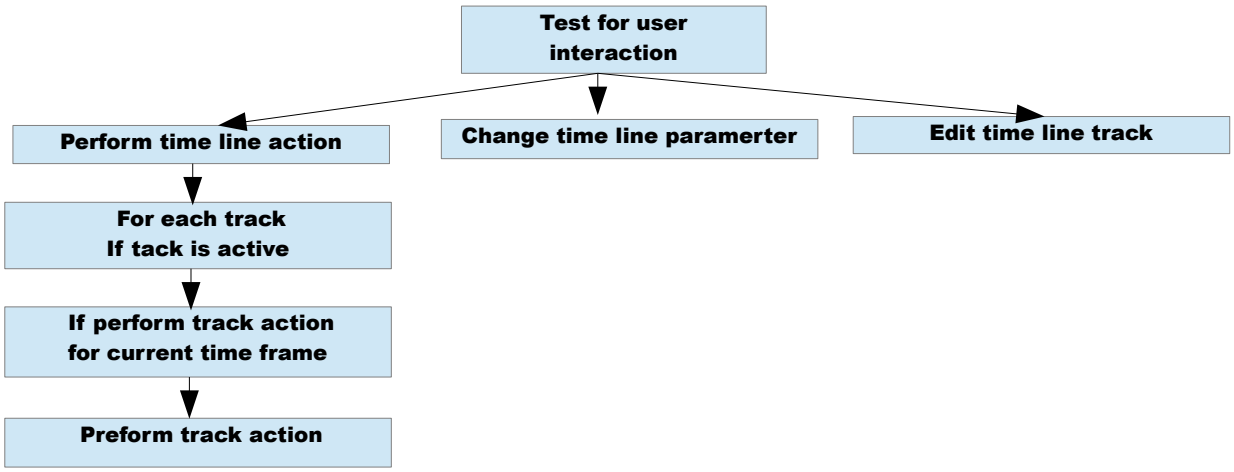

Fig 08.0.1 Schematic of Time Line operation

In table 02.01 of section 2, a timeline panel is displayed as part of the application's cycle loop, and if within one of those cycle loops the time line is selected to perform a time line action, it will perform that action as illustrated in Fig 08.01.

The tracks form the core of the timeline functionality, but need to be created, displayed and deleted dynamically. Tracks can be part of a group.

Following in the same ethos and general design of the virtual worlds application, to display tracks of any kind, the method of using a base classes was implemented. Without going into detail how the timeline widget was developed, the basic design structure of the time line widget is illustrated in Fig 08.0.2 Setting up the timeline animation and executing it is not trivial and is the most complicated of methods and procedures in the application to implement.

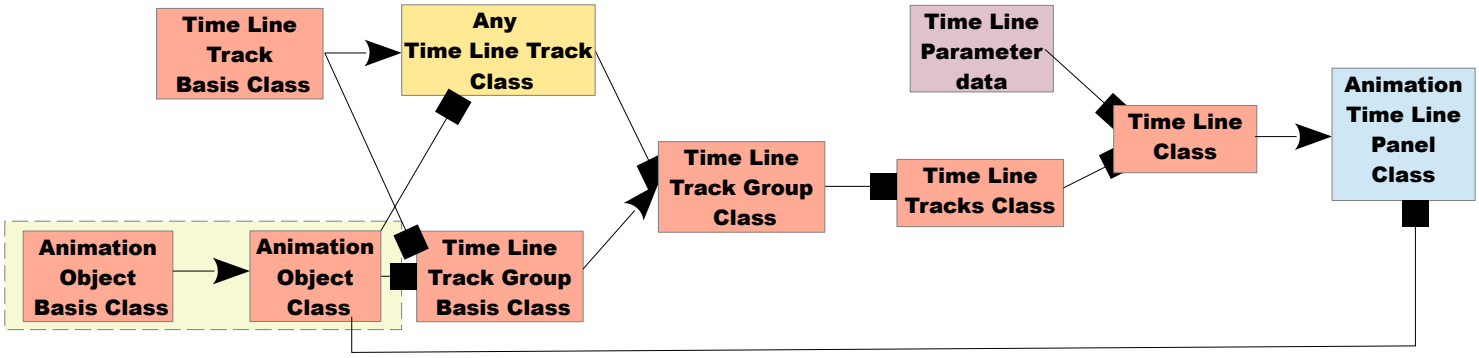

Fig 08.0.2 Schematic of Time Line Class design and Construct

The timeline panel class displays the central timeline widget and controls that executes the animation process of the scene as illustrated in Fig 08.0.1. The inherited time line class is a widget that manages the display of the time tracks as a widget that are to be in turn, referenced by the time line panel class to manage the execution of animation actions.

These animation actions are defined and manged through an animation object basis class. The animation object basis class has as of its design and purpose, common virtual functions structures and variables that are to be inherited and defined in a derived object animation class. This derived object animation class is associated with a node editor node, and

1. Changes the node parameter settings and executes function(s) that these settings influence in the parameter panel for that node.
   or
2. Executes a function

Defining an animation timeline entry as illustrated in Fig 08.0.2 involves two parts

1. Defining the timeline track(s) to display and interact with to define the animation parameters and
2. Defining the action to take that is defined by the time line track and the settings of the timeline panel controls.

Timeline Tracks

The timeline class widget, as stated previously, manages the display of one or more timeline tracks.

The timeline tracks class displays and manages groups of timeline tracks. These groups can have one or more timeline tracks per group.

The timeline track group class manages individual addition of a timeline track that has inherited the timeline track basis class into a group of tracks. It also inherits a timeline group basis class.

The timeline group basis class stores and performs all of the management tasks of timeline tracks that inherit the timeline track basis class. It also stores a pointer reference to the animation object class that has the actions to perform for any or all types of timeline track.

The timeline track basis class is a base class of common virtual functions and variables that can define and create any type of timeline track that inherits this class. This base class has virtual functions to create, delete and display a timeline track for this purpose. As indicated above, this is the class that is referred to for timeline track management and functionality.

The animation timeline panel widget class is a virtual worlds class object and exists in the directory

source→VW→Main_Window→Panels

with file name

animation_time_line_panel.h

The support timeline classes are universal class objects that exist in the directory

source→Universal_FW→Timeline

with file name

timeline.h
timeline.cpp
timeline_interval.h
timeline_interval.cpp
timeline_parameters.h
timeline_tracks.h

The animation object base class is universal class objects that exist in the directory

source→Universal_FW→Animation

with file name

animation.h

08.1 : Creating an Animation Time Line Track

Creating an animation time line track needs careful thought as creating timeline tracks are tied to nodes, and the timeline widget. Therefore a node that is to have an associated timeline track needs to have a link to it, and also the timeline track needs to have a link back to the node. Details of how to do this is not given as it is specific to the node and what action the track performs. To get an in depth example the coding required, it is best to examine the source code within application modules. Directories Animation and Node_editor.

The source code in section 8.0 need not be concerning too great as this is the code of the virtual worlds engine that is designed to add, display and manage timeline track groups added to it.

i. An animation object class that inherits the animation object basis class and uses its virtual functions defines the setup and actions to take for any or all of the timeline control buttons and settings is created and assigned as part of the node data structure.

ii. in section 6.5, Creating a node to use in the node editor, for any node that is to have an animation timeline track to be able to perform any timeline task(s) on the node's parameters or data, there needs to be a function to create and delete a link to the timeline panel widget. However this is not a node basis class virtual function (This may change) as this is menu driven and not referenced in the node editor.

iii. A pointer to the animation timeline tracks widget class is needed so as to create and display a track within a track group that is linked back to this node and the animation object class in i. This pointer is an inline variable to the animation timeline tracks class that is created and defined on the application startup, and is accessible as an application global variable. This pointer variable is called animation_timeline_tracks_widget.

iv. To add and delete the nodes animation timeline track from within the node editor, a node menu entry needs to be added to the nodes menu option (section 06.5 Creating a node to use in the node editor step 5) that calls the functions in ii.

v. A node track or group of tracks can also be deleted from the animation timeline panel for a particular node if there exists a function to delete the node's timeline. This is also critical to have so when a node is delete, so are any timeline tracks in the animation timeline widget.

09 : Global Variables

To be able to have a method of keeping track of what 3D object was selected, a class called globalc was created to set and get information the current selected entity for the purposes of 3D object data access from the application entity categories class for processing or display. The information that is set and retrieved are the 3D object entity id, entity type id and entity category id which is all that is needed for this purpose.

Globalc has the following functions to perform the function of setting and getting information the current selected 3D object.

```
static id_type get_current_selected_data_context_id()
static void    set_current_selected_data_context_id(id_type n)

static id_type get_current_selected_entity_id()
static void    set_current_selected_entity_id(id_type n)

static id_type get_current_selected_entity_type_id()
static void    set_current_selected_entity_type_id(id_type n)

static id_type get_current_selected_entity_category_type_id()
static void    set_current_selected_entity_category_type_id(id_type n)
```

To be used, the inclusion of the global.h file must be placed at the top of application include statements that is to use it in the application. Ie at the top of the user_application_class.h that was explained in section 01.

Many other classes have various variables and methods to keep track on the current selected application attributes within each class. These need not be global in nature and can have the information of user selections passed to their various linked classes or functions.