Virtual Worlds Application Documentation (draft)

User interface

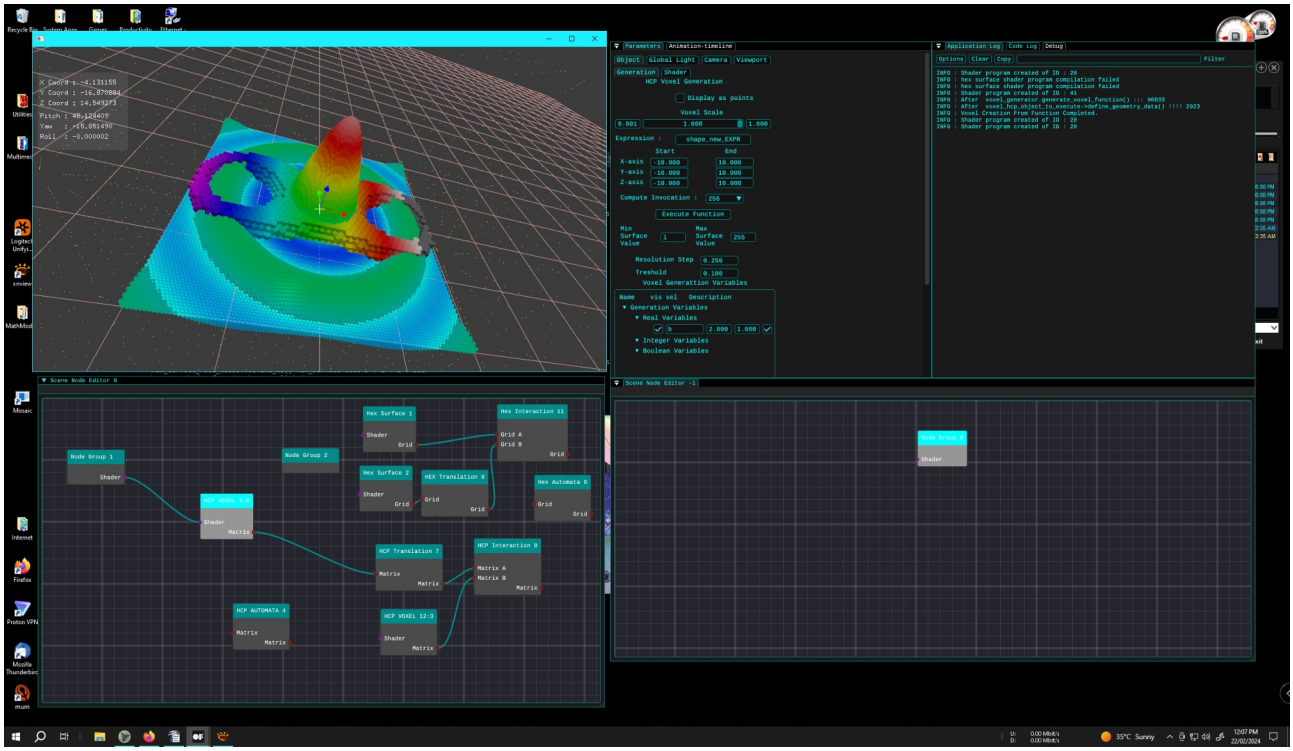

Fig 1 Virtual Worlds interface

The basic design of the Virtual Worlds application in its latest iteration is to generate cloud point data objects which make up 3D objects and perform modification or operations on the generated cloud point data that are then displayed in a 3D view port. Generated point cloud data is performed through a method of the user using glsl code within a compute shader program and displaying the generated and optionally modified data using OpenGL. The procedural generation of 3D objects done through using an OpenGL compute shader program utilises the parallel processing powers and capabilities of the GPU.

The basic interface of the Virtual Worlds thus looks on first impressions like most other 3D applications with a central viewer port to view 3D objects, and an array of panels to create, select and edit 3D objects, the view port camera and environmental settings. An additional timeline panel is introduced in this iteration to allow and perform animation tasks. This is all very basic and primitive compared to many comorcial 3D applications, but has the scope to be improved and polished up.

This iteration has no outliner panel that displays all the objects in the scene to be selected for editing or other tasks, however in a future version this may be implemented. All creation and editing of objects is performed via a node editor. The node editor has an inbuilt menu system where the user selects what object is to be generated, and through links, what modifications or operations to be performed as in most node editors. By selecting a node, the parameters and inputs to perform tasks are displayed in the parameter panel as well as a button to execute any task that the node is designed to perform. The nodes themselves do not have widgets to enter or modify parameter inputs as this would mean very large nodes that look more like floating windows than nodes.

Most if not all nodes in the node panel can be listed in the timeline widget to perform the same tasks in a parallel fashion. The timeline panel is not fully functional in this iteration as only a single timeline interval (ie interval when to perform a nodes task) is implemented. A future iteration is planned to have multiple time intervals in a single track along with multiple key frame capability.

The parameter panel changes depending upon which node is selected. The creation, loading and saving of parameter data, and processing or generation of data for the particular module data object is done within this panel.

A logging panel to display messages such as failed/succeeded tasks or debugging information is provided.

The application utilises and integrates third party tools for the GUI and graphics. IMGUI for the GUI and Openframe works for the graphics with some others as well in a more minor role.

All this is basic and not optimal, or indeed even production ready. But this does fit the purpose as this application stands, which is a demonstration, proof of concept and basis to build upon for the method of creating and displaying the procedural generation of 3D point could objects.

Basic Design Concepts adopted

This application is written in the C++ language and utilises as its main design concept, object orientate basis class types utilising standardised defined universal virtual functions to perform the same task such as displaying a node, but for each derived class from this basis class, may have different code to perform that task with increased diversity and functionality.

This is used widely and as much as possible as it simplifies the code, makes editing and modification of newly added code easier and efficient, as well as maintenance and understanding of the engine that this application runs under. By use of one basis class, multiple derived classes can be referenced by the applications engine through this basis class of virtual functions without needing worry about the derived class itself, or the code that may be different from one derived class to the next. This is an extremely powerful functionality of C++ that is utilised as much as possible in the Virtual Worlds application coding.

By utilising this basis class of virtual functions concept, modularity of the application is at the heart of its design. As much as possible, the application has modules of code that can be added to the core engine code in a manner that allows adding or retraction of code easier and more maintainable as possible. Thus the design as much as possible has a core application engine that has “bolted” to it modules of code that adds and expands the application's functionality.

As part of this core functionality are the third party tools of IMGUI and OpenFrameWorks. IMGUI is used to perform all the GUI input and execution tasks unaltered at this stage, but with some personal additional functions to simplify the IMGUI code into more simple code.

OpenFrameWorks is used as a frame work to display the 3D graphics, however has proven to be not entirely workable thus far on its own. To perform some of the desired functionality of Virtual Worlds , additions and modification to some of the OpenFrameWorks classes needed to be performed and one glaring bug resolved. The reason is that it seems that OpenFrameWorks is designed to be used primarily as a kind of tool similar to shadertoy rather than as it namesake, a framework for applications to use. However with out going into detail, as knowledge on how it works is gained, and with modification and adding to its code, this can be overcome and further development of using it. This is because whenever a derived class is created from some of its classes, it was found that the derived class did not work. Most of the problems seem with using OpenFrameWorks was of its use of shared pointers that created massive problems.

Even so, OpenFrameWorks will at this stage be continued to be used as it fits so closely with the Virtual Worlds initial code and design philosophies.

One final basic design concept is to utilise and use any third party code that is suitable and available as open source for this application, and if permitted to be used or modified. The node editor for example is a modified version of ImNodes for example. To make it suitable for this application, ImNode was modified into a c++ class rather than is original form of static functions. This made the node editor in this application much easier to implement and expanded its functionality while not having to reinvent the wheel and waste time. The final design concept, don't do what some one else has made available if it is good enough or better than what can be done by oneself.

Application Directory Structure

The application directory structures is governed at this time by being included as an application that exists within the application directory of the OpenFrameWorks directory structure. The Virtual Worlds application was created by using the OpenFrameWorks project creation application to set up the visual studio project because not doing so would create too many uneccessay problems.

OpenFrameWorks

The OpenFrameWorks directory structure is as in **Listing 01**. A full description of this can be obtained from the OpenFameWorks web site. Directories are in bold and are highlighted in grey.

addons
apps
docs
examples
libs
projectGenerator
scripts
.gitignore
CHANGELOG.md
CODE_OF_CONDUCT.md
INSTALL.md
INSTALL_FROM_GITHUB.md
LICENSE.md
of_inbuilt_uniforms.txt
README.md
SECURITY.md
THANKS.md

Listing 01 : OpenFrameWorks root directory structure

Virtual Worlds application directory

The Virtual Worlds application directory structures is located in the OpenFameWorks apps directory.

apps->myApps->Virtual_Worlds.

And is as in **Listing 02**.

bin
Build
Documentation
Libs
obj
source
addons.make
config.make
icon.rc
Virtual_Worlds.sln
Virtual_Worlds.vcxproj
Virtual_Worlds.vcxproj.filters
Virtual_Worlds.vcxproj.user

Listing 02 : Virtual Worlds root directory structure

The bin directory contains the OpenFrameWorks binary files to build and link the application
The Build directory contains all the Virtual Worlds execution and other files required to run and test the application.
The Documentation directory contains the application documentation files like this one.
The Libs directory contains the Virtual Worlds application required third party library files that are used to compile and link the application.
The obj directory is a directory created and used by visual studio to create the executable.

Source directory

The Source directory is where the source code is of the Virtual Worlds project. And is described in **Listing 03**

```
Application
FrameWork
VW
main.cpp
```

Listing 03 : Vitual Worlds Source directory structure

As of this writing, The Application Directory is a directory generated by OpneFrameWorks and is not in use

The FrameWork directory contains the source code and files that are to be used as a framework for the Virtual Worlds and possibly other applications that does not involve the OpenFrameWorks source code.

The VW Directory contains the source code of the Virtual Worlds project that is not applicable or of use to any other application.

main.cpp is the application's entry point file. This location may change.

FrameWork directory

Listing 04 gives the directory structure of the FrameWork directory.

```
Kernels
Universal_FW
VW_framework
```

Listing 04 : Virtual Worlds Source FrameWork directory structure

The Kernals directory is where the universal kernels source code that are essential to the functioning of any application but specific to none are located.

The Universal_FW directory is similar to the Kernals directory in purpose and designation. It is envisaged that the Kernels and Universal_FW directories will ne merged into one at some future point to avoid confusion as of this writing. As of this moment this directory has third party, modified third party and original created code that can lead to self contained functional code that can be used in any application.

The VW_framework directory contains framework source code specific to the Virtual Worlds application that can be used in other similar applications or is specific for the Virtual Worlds modules to use. That is source code that is a basis and framework for modules to use so as to be able to to be added to to the Virtual Worlds application as a class, structure or other functionality. Many of the classes and structures of files in this directory will have a basis class of virtual functions and universal constants and variables that are to be inherited by derived classes for universal usage.

VW directory

Listing 05 gives the directory structure of the core application source code that defines the Virtual World engine and utilises kernel and framework code that has basis classes or structures of virtual functions to allow universal generation and execution in a standard manner of point cloud or other types of data and to display in a standardised manner a representation of this data in a viewport.

```
Editor
Modules
```

Listing 05 : Virtual Worlds Source VW directory structure

Editor is the directory of where the code of the Virtual Worlds editor engine and GUI interface resides. The code in this directory handles all of the core engine functionality to manage and display data, widgets and application settings as well as importing and exporting of application specific data of modules that are linked to the engine for it to use.

Modules is the directory where application modules reside.

Application modules are a kind of plugin of external code that links into the core editor engine such that within each module is code that defines all of the functionality that is required to display GUI widgets for the purpose of being used in a node editor, display parameter inputs and settings, generate data and display that data in the application view port.

APPLICATION OPERATION

The basic application operation is described graphically as in **Fig 02**. This operation of application may be familiar and common to all 3D or other applications.

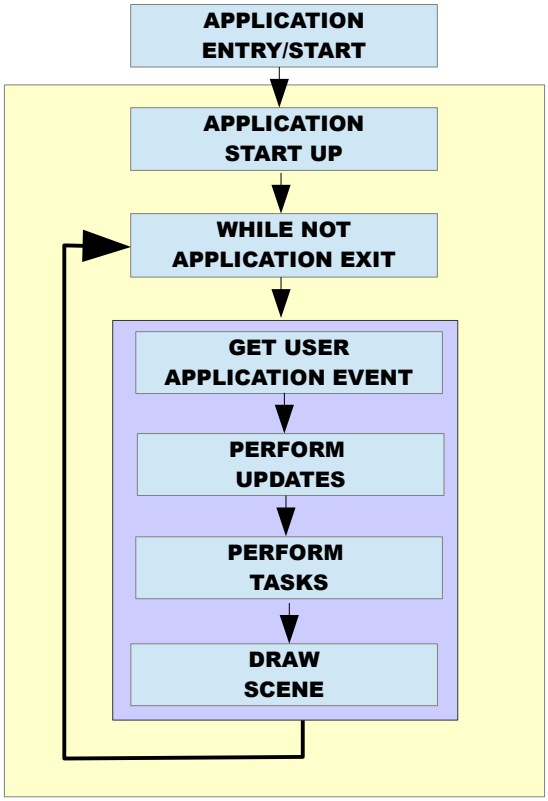

Fig 02 Schematic of Virtual Worlds overall flow of operation

OpenFrameWorks is used as a basis to run and manage the Virtual World application operations as depicted in Fig02. OpenFrameWorks has application classes that manages and performs all the necessary procedures to run an application as depicted in Fig02. This design method was utilised in the previous iteration of Virtual Worlds and thus migration into an OpenFrameWorks project was not difficult and more efficient to use.

Application start is the entry point of the application. For this application it is the file main.cpp within the source directory source/VW. Within the main.cpp file the main window and application runtime parameters are defined before the application is started.

The yellow rectangle depicts the one C++ class from which the application engine runs and is managed from. This C++ class is a derived class of the OpenFrameWorks ofBaseApp class that utilises virtual functions to perform all the application execution and management tasks such as get user application event and testing for exiting the application, thus alleviating the need to write such boiler point code. OpenFrameWorks thus hides from the user the need to worry about managing the application execution and exit and only on what is needed for the application's main functionality. This class is called the main_window_class and the code exists in *source->VW->Editor->Main_Window->main_window.h*

start up is the one off execution of routines, defining of variables, definitions and other factors required to be initiated before the main application loop commences.

While not application exit and get user application event are handled by theOpenFrameWorks application management system as are the execution of of Perform Updates, and Draw Scene virtual functions, Perform tasks is added for clarity that the user would need to include functions as part of the general operation of the application.

Perform Updates (virtual function update) defines any variable or data updates needs to be performed.

Draw scene (virtual function draw) defines all the drawing routines required for drawing to the view port.

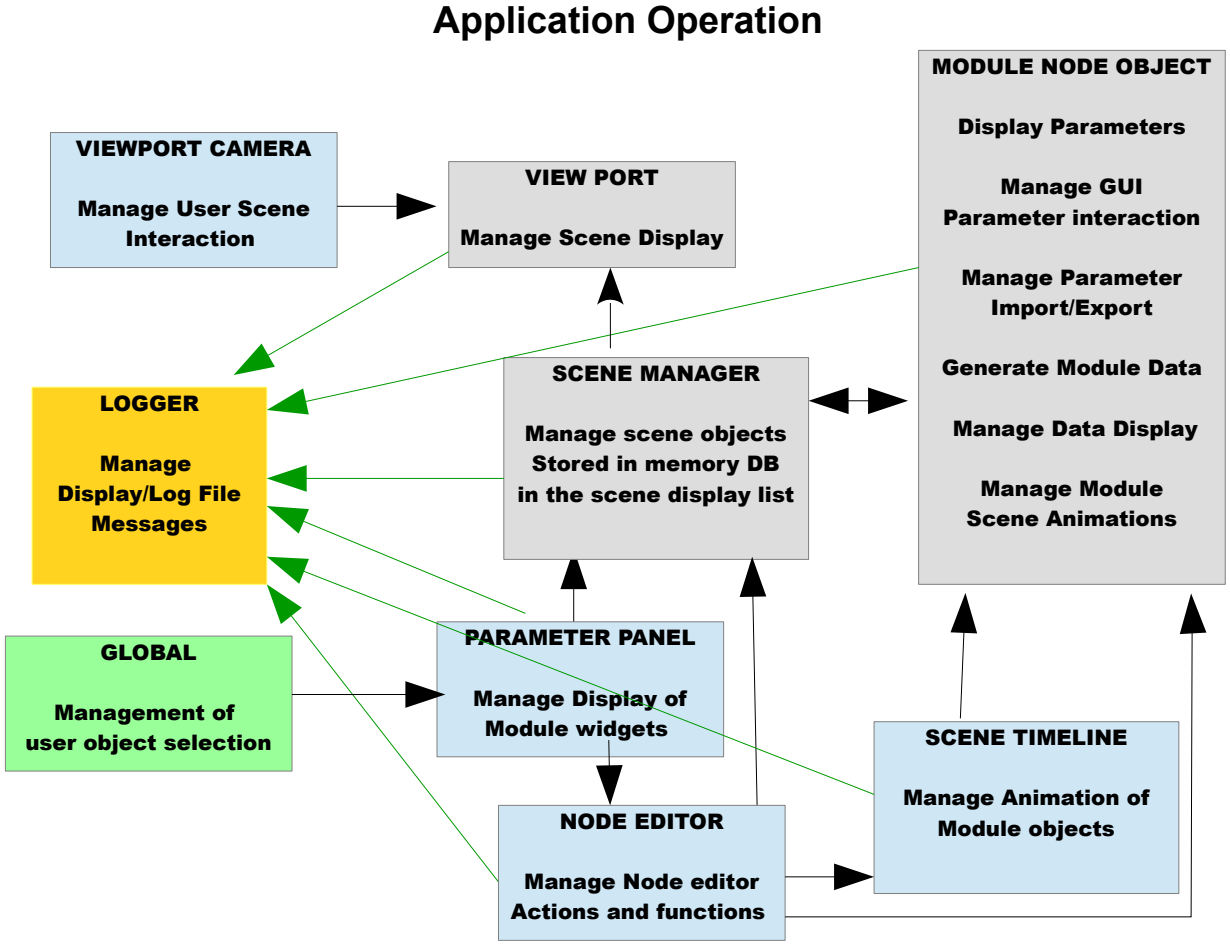

Fig 03 Schematic of Virtual Worlds overall flow of operation

Fig 03 Is a schematic of the overall functionality of the Virtual worlds application flow of data and its design. Each box represents an application module or section of functionality that may be one or more classes. There are basically four broad types of operation.
1: The user interaction with the application represented in blue.
2: A global tracking system to keep tabs on selections that the user has made represented in green.
2: The application functions that perform all the tasks required to manage and display graphical data on the screen represented in grey
3: Logging to display information and errors represented in yellow.

A set of global variables and functions (depicted in green) keeps track of the user selection of node editor or data object(s), and determines what is displayed in the application's parameter panel. An application logger to display messages is implemented and is available to have messages sent to it .

As it exists at the time or writing, the Virtual worlds application operation to create and display a 3D scene is performed primarily through a node editor.

1: In the node editor, the user creates or selects a node that defines an object or task to be performed on one or more objects. Via a floating menu, the user creates a node and then by selecting it or an existing node, the parameter panel will display a set of widgets by which the user can interact with to generate, modify or perform operations on cloud point data and to display that data to the view port.

2: Nodes that are selected will have a floating menu of selections that can be selected to perform certain tasks of data management such as saving of generated data to a file. Another menu option is to open up a timeline track related to the selected node and the data parameters that the node represents and displays in the parameter panel. Ie by creating a timeline track for a particular node, the parameters that can be changed by a frame step function in the parameter panel can be controlled through the timeline widget.

3: The scene timeline controls the execution of functions represented in the parameter panel of one or more

nodes that are linked to it. The timeline widget gives the ability to execute frame steps of procedural functions that are linked to it in parallel and thus create animations.

5: The view port camera is how the user interacts with viewing the rendered scene by moving and orientating a virtual camera it in the scene. The user controls the camera via the mouse and keyboard. There are also a series of permanent parameter panels that the user can access to change the camera view and global settings of the display port.

Node Editor Operation

The 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 **Fig04**.

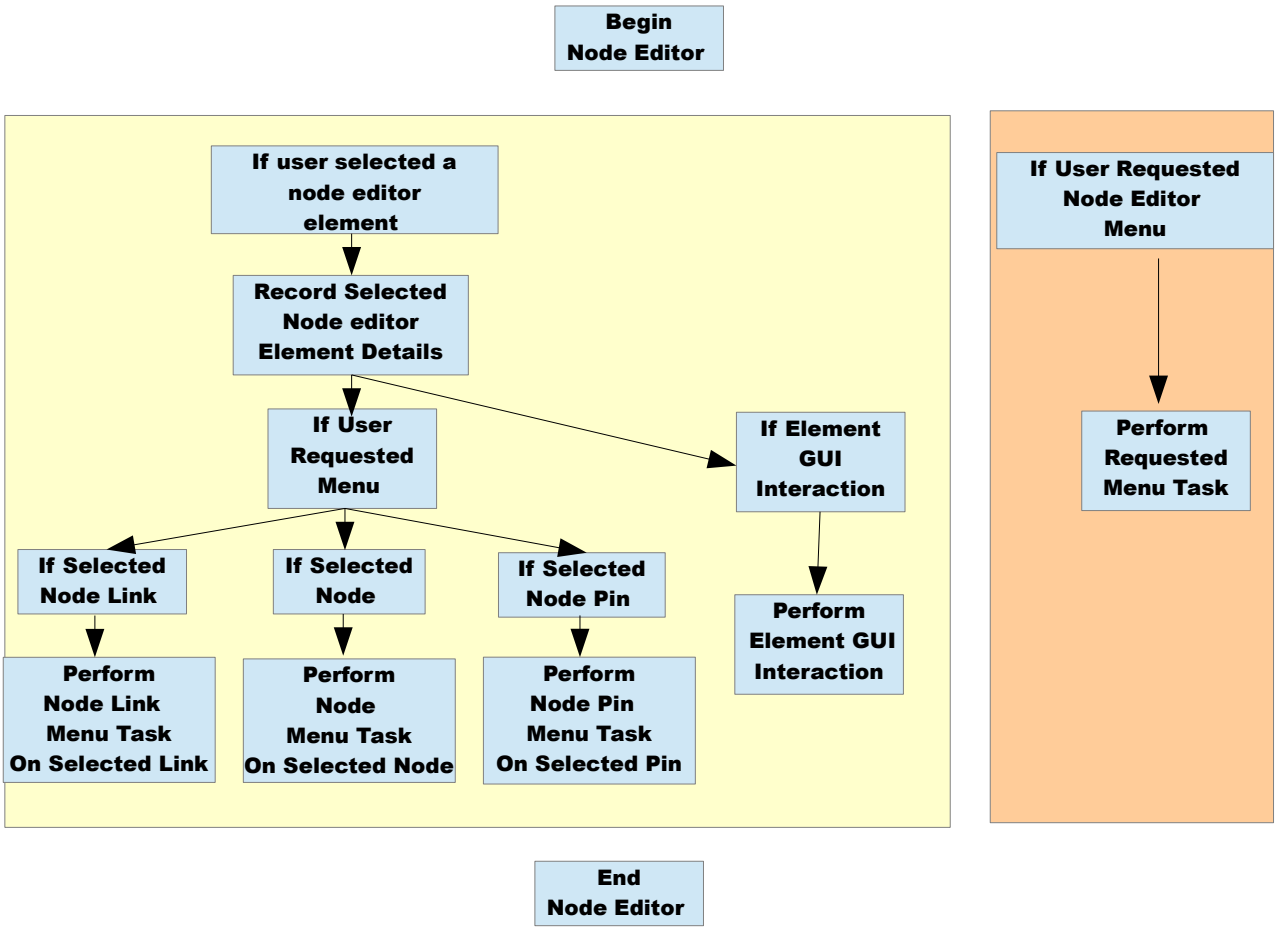

Fig 04 Schematic of Virtual Worlds Node Editor operation

The Node editor essentially consists of two main sections as illustrated in **Fig04**. 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 04** may appear to operate in parallel this is not the case in implementation it is in sequence.

The node editor used in the Virtual Worlds application is an edited derivation of the ImNodes node editor authored by Johann Muszynski on the Git Hub platform. It has been transformed into a c++ class object that makes it easier to use as a multiple node editor system as by referencing a node editor as a class removes many of the problems it initially had using a ImGUI like system.

Parameter Panel Operation

The parameter panel is the widgets GUI interface where the user interacts with the parameters and settings of a node that a user has selected in the node editor, and to perform operations or tasks based upon the inputs that the user has mode. The basic operation of the parameter panel is illustrated in **Fig 05**.

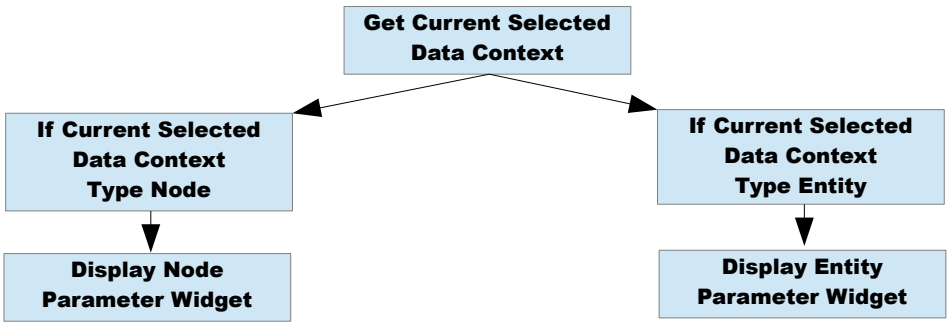

Fig 05 Schematic of Virtual Worlds Parameter Panel operation

To perform the parameter panel operation, a global variable of the current user selected data context and an id to that data type of that data context is stored and queried. If within the application's data management system such data is found, then the data is retrieved to be displayed. To display the parameter data retrieved, the selected node or entity has part of its class a defined ImGUI widget function to display the parameters and other functions to perform tasks. The defined ImGUI widget function is a virtual function of a C++ base class that every node or entity class is derived from. Using such a base class simplifies and reduces the coding greatly for this section of the application greatly.

View port Camera Operation

The view port camera is a modified copy of the camera and easycam class of OpenFrameWorks. The open frameworks camera and eaycam class is, as of writing, the only class that could be copied and modified without effecting or corrupting the OpenFrameWorks system and causing the Virtual Worlds application to crash. The reason for not utilising a derived class was that a derived class would cause a few problems with the OpenFrameWorks system. The reason for making a copy was not to edit the original camera classes of OpenFrameWorks and allow extensive customisation that was present in the previous incarnation of Virtual Worlds not present in the OpenFrameWorks coding to be integrated.

Much of the camera operation is to check to test if the computer mouse cursor position is within the view por window, and it so check to see if there is a user input signal such as a mouse button press, movement or keyboard key(s) signal and then change the computer camera location or orientation and thus the scene that is being rendered in the view port.

The basic operation of the view port camera is illustrated in **Fig 06**.

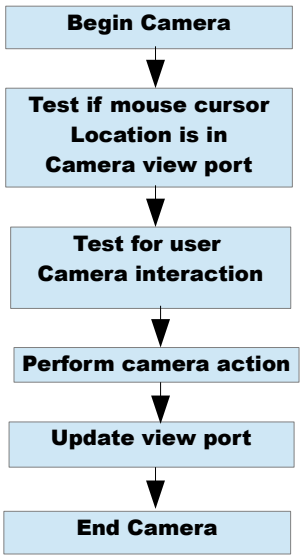

Fig 06 Schematic view port camera

The begin and end Camera routines given in Fig 06 is an OpenFrameWorks requirement that sets up the environment for the camera to accept inputs from.

Time Line Operation

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 07**.

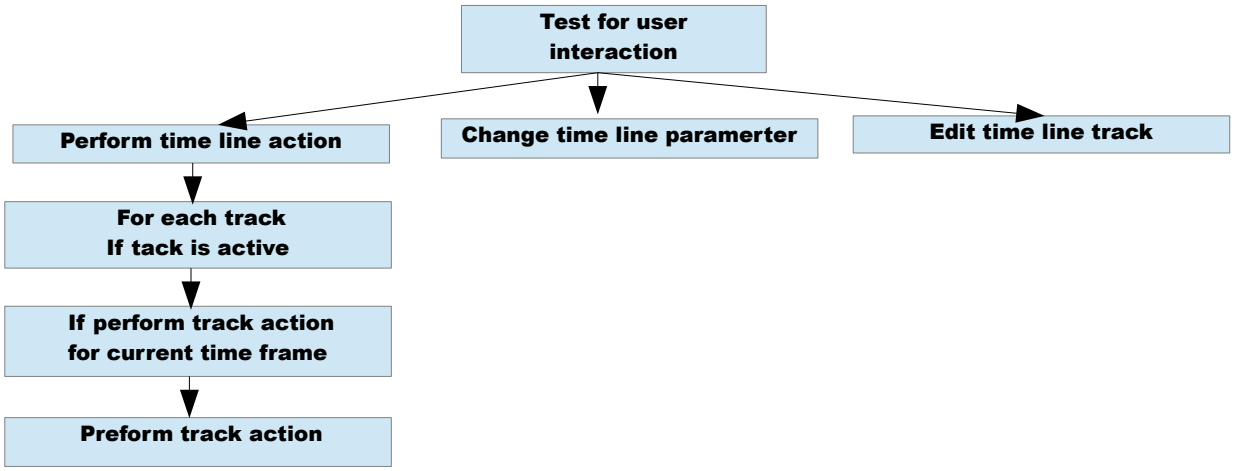

Fig 07 Schematic of Time Line operation

Scene Manager Operation

The application scene manager is the heart of the virtual worlds engine. It is where the data is stored and the management of the project is performed. Project data is stored in data sets according to a category type and thus the scene manager consists of at least two layers of data management. One being that of the categories of data, and another within each category, the category data itself. Thus to make it easier to manage and implement, the use of C++ base classes with virtual functions can be utilised.

All scenes have lights and cameras, so it was decided that these two types of scene objects were to be scene objects in their own right and thus to be their own scene objects. Everything else was to be divided into scene category objects, and each category to be a kind of scene data base. A schematic of the scene manager operation is given in **Fig 08**.

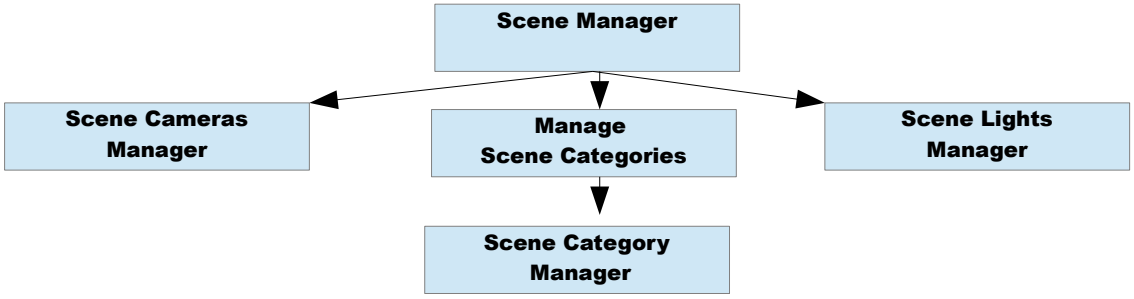

Fig 08 Schematic of Scene Manager operation

Within each of the scene data managers in Fig 08 are the same type of functions to add, delete, retrieve, find, and display data etc. Each data manager utilises a unique base data class with virtual functions that are then used to standardise and simplify coding in the project.

Module Node Object Operation

The Module node object is in effect a kind of plugin module that is linked to the virtual worlds engine. The Module node object is where the code resides to specify all aspects of the functioning of the virtual worlds application in regards to a specific scene data object. The user interaction interface to specify the creation and display of nodes for the node editor, parameter panel, time line widget and view port. Code to generate and display, save and import data.

Thus the Module Node Object can be thought as the implementation of a scene object, while everything else can be thought of as data and scene management engine system.

MODULE OBJECT DATA

Display Parameters

Manage GUI Parameter interaction

Manage Parameter Import/Export

Generate Module Data

Manage Data Display

Manage Module Scene Data Model

Display Parameters

1: Define the parameter display widgets for each module GUI parameter class.

2: Create a parent tab widget and assign each module GUI parameter class to be displayed as tabs in This tab widget.

3: Display the resultant tab widget defined in 2: within a Module parameters panel.

Manage Parameter Export

1: Generate a parameter structure variable to store the parameter data to be exported.
2 : For each widget in the parameter GUI get the widget value and copy it to the Corresponding parameter storage variable of the parameter structure defined in 1:
3 : Pass the parameter structure variable with All the GUI parameter data copied to it to The appropriate module export function to Be saved as a formatted text file.

Manage Parameter Import

1: Generate a parameter structure variable to store the parameter data to be imported.
2 : Pass the parameter structure variable to The appropriate module import function to Have its parameter variables given a value From the reading of an appropriate selected formatted text file.
3: For each widget in the parameter GUI clear its value to a zero nor NULL value
4: For each widget in the parameter GUI assign its value it to the corresponding parameter Value that is stored in the parameter structure Variable defined in 2:
5: Perform any updates of the 3D scene in the Viewer or the parameter GUI as neccesary

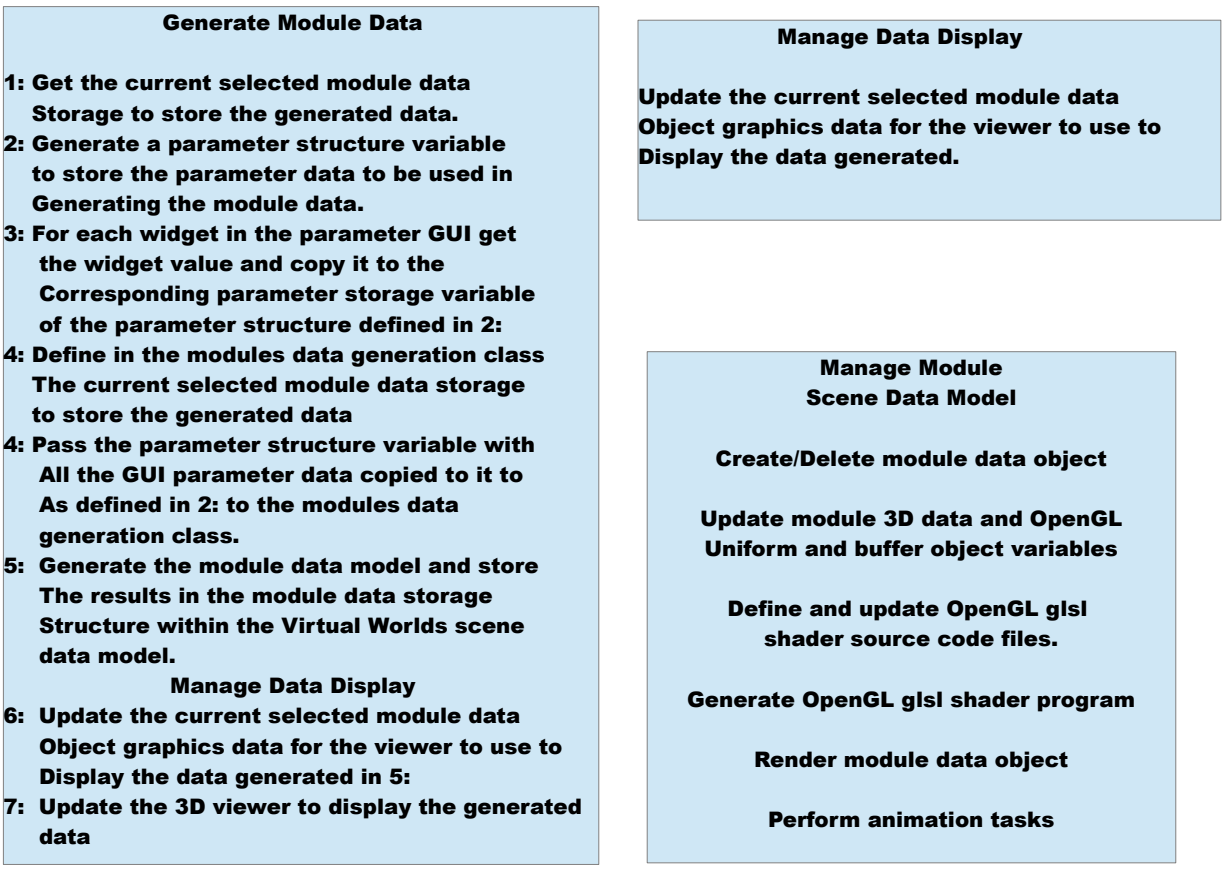

Fig 03 Schematic module data type operation tasks

View port Operation

The View port operation is the OpenFrameWorks view port. Displaying to and managing the view port is thus left for the OpenFrameWorks system. But this doe not mean that it can be ignored, as in much of the code that OpenFrameWorks uses there are references to functions to get the current view port for drawing, ie rendering tasks. I also seems that OpenFrameWorks application runtime routines that manage and implement the application execution loop keeps tabs on the open view ports and which one is currently selected. So in essence, currently virtual worlds can let OpenFrameWorks do all the view port work and no concern is required on the inner workings or management of it other than selecting it when it is i required to do so.

Virtual Worlds Application Export Parameter and Scene Data to Formatted text file

Even though the Virtual Worlds application has different parameters for each application GUI panel and module data object GUI input widgets, the export of parameters from all of these GUI widgets to a formatted text base file follows the same procedure and design method. The basic pseudo code to export every set of parameter data is

A : Prepare and pass a data structure with the parameter data to be exported to an appropriate export function.

A1: Define a parameter structure with variables of an appropriate specific data type for each parameter data type to store a particular parameter value in.
A2: Create a variable to the parameter structure type defined in 1
A3: Get the value of every GUI widget that has a parameter value defined for the data to be exported and assign that value to the appropriate variable in the parameter structure variable given in 2
A4: Pass the parameter structure variable with all the parameter data stored in it from 3 to the application or module appropriate export parameter class function.

All the appropriate export function defined in step A4 follow the same basic design method of

B: Export parameter data to formatted text based file.

B1: User defines a file path name to save the parameter data to and opens the file specified
B2: A C++ write text stream is created to write data to the file opened in step B1.
B3: Export the parameter data passed as a parameter structure variable to this export parameter function in a specific order and format.
B4: Close opened text file that is opened and text data written to.

All the export of parameter data performed In step B3 is of the same design method and functionality of having parameter data of a specific type or class bounded by flags to indicate a block of parameter data of a specific type existing between these flags. In the simplest form a generalised parameter data block is defined as

```
PARAMETER_DATA_BLOCK_BEGIN
parameter data 1 value
parameter data 2 value
parameter data 3 value
        :
        :
PARAMETER_DATA_BLOCK_END
```

The reason for specifying parameter export data this way is that parameter and any exported data can be like computer source code have an embedded structure of data that can be flexible and dynamic such that any complex data structure can be exported, and then imported according to the same data structure that an application uses to store data, thus simplifying the export import process. Eg

```
PARAMETER_DATA_BLOCK_BEGIN
PARAMETER_SET1_DATA_BLOCK_BEGIN
parameter set1 data 1 value
parameter set1 data 2 value
parameter set1 data 3 value
PARAMETER_SET1_DATA_BLOCK_END       :
PARAMETER_SET2_DATA_BLOCK_BEGIN
parameter set2 data 1 value
parameter set2 data 2 value
parameter set2 data 3 value
PARAMETER_SET2_DATA_BLOCK_END
PARAMETER_DATA_BLOCK_END
```

Which can then be expanded in a visual design concept form as

```
PARAMETER_DATA_BLOCK_BEGIN
   PARAMETER_SET1_DATA_BLOCK_BEGIN
         PARAMETER_SET1_SUBSET1_BEGIN
               more embedding
         PARAMETER_SET1_SUBSET1_END
         PARAMETER_SET1_SUBSET2_BEGIN
               more embedding
         PARAMETER_SET1_SUBSET2_END
   PARAMETER_SET1_DATA_BLOCK_END    :
PARAMETER_SET2_DATA_BLOCK_BEGIN
         more embedding
PARAMETER_SET2_DATA_BLOCK_END
PARAMETER_DATA_BLOCK_END
```

.

Which gives a flexibility of design that each embedded data block can be blocks of data of different kinds of related data such that data can be exported in a hierarchy of data structures such that data can be grouped together in a form such as

```
SCENE_DATA_BLOCK_BEGIN
   GROUP_DATA_TYPE1_DATA_BLOCK_BEGIN
         MODULE_DATA_TYPE1_DATA_BLOCK_BEGIN
               MODULE_DATA_TYPE1_DATA
         MODULE_DATA_TYPE1_DATA_BLOCK_END
         MODULE_DATA_TYPE2_DATA_BLOCK_BEGIN
               MODULE_DATA_TYPE2_DATA
         MODULE_DATA_TYPE2_DATA_BLOCK_END
   GROUP_DATA_TYPE1_DATA_BLOCK_END:
GROUP_DATA_TYPE2_DATA_BLOCK_BEGIN
         more embedding
GROUP_DATA_TYPE2_DATA_BLOCK_END
SCENE_DATA_BLOCK_END
```

This simplifies the export of complex data structures down into simple functions for each data block type such that an export function would exist for the above example

```
MODULE_DATA_TYPE1 data export
MODULE_DATA_TYPE2 data export

GROUP_DATA_TYPE1 export{
   MODULE_DATA_TYPE1 data export
   MODULE_DATA_TYPE2 data export
}

GROUP_DATA_TYPE2 export{
         <GROUP_DATA_TYPE2 export functions>
}

SCENE_DATA_BLOCK export{
     GROUP_DATA_TYPE1 export
     GROUP_DATA_TYPE2 export
}
```

What this allows is if the application to export MODULE_DATA_TYPE1 parameter data, as a lone text data file in the same format that would be exported if the application was to export GROUP_DATA_TYPE1. And in turn GROUP_DATA_TYPE1 would be exported as a single formatted text file as it appears in the export file of SCENE_DATA_BLOCK. Thus great flexibility and simplification of coding for the hierarchical data structure that exists in the applications data storage model is achieved and thus is used in all data exporting functions.

The only restriction is that when actual values are exported, they are in a particular order that must be followed, as with any importation of data read in exactly the same order.

Virtual Worlds Application Import Parameter and Scene Data from Formatted text file

Even though the Virtual Worlds application has different parameters for each application GUI panel and module data object GUI input widgets, the import of parameters from a formatted text file to replace the GUI widget values follows the same procedure and design method. The basic pseudo code to import every set of parameter data is

I : Prepare a data structure with the parameter data variables to be assigned values to be copied to the appropriate GUI widget value.

I1: Define a parameter structure with variables of an appropriate specific data type for each parameter data type to store a particular parameter value in.
I2: Define a parameter structure variable of the type defined in a1 and pass a referenced variable to the appropriate import file function.
I3: Select the file to import parameter data from
I4: Read the import file data and assign the value of the parameter data read to the appropriate parameter variable of the parameter structure variable passed in step A2.
I5: For each parameter data value of the passed parameter structure variable that has been given a parameter value from the importing of parameter data, assign the value stored to the appropriate GUI widget value.
I6: perform any required updates to the GUI widget and application viewer display.

Step I4 imports the data that was exported as explained in section Virtual Worlds Application Export Parameter and Scene Data to Formatted text file. Thus the reverse procedure of the export is implemented by the import function and all importation functions follow the same concept of design and implementation.

J : Open the nominated import for reading only and read the entire file in a single QString class object.

J1: Separate the single string stored in the QString class into separate lines by use of the Qt QStringList class.
J2: Read each line one at a time and following the same order as was exported, if an appropriate start data block flag exists where is should exist, read all the following line testing for where further start/end data block flags should exist, or if where data should exist, read the data values into the appropriate passed parameter structure variable defined in step I2.
J3: Read the next line and if an appropriate end data block flag exists repeat step J2 until all data is read and a final end of data block flag is read for that importation of parameter data.

As with the exporting of parameter data explained in Virtual Worlds Application Export Parameter and Scene Data to Formatted text file, the importation of data is be of the same modular design as the export where importation of hierarchical data structures is of the same functionality of design where importing functions can be of form.

```
MODULE_DATA_TYPE1 data import
MODULE_DATA_TYPE2 data import

GROUP_DATA_TYPE1 import{
    MODULE_DATA_TYPE1 data import
    MODULE_DATA_TYPE2 data import
}

GROUP_DATA_TYPE2 import{
        <GROUP_DATA_TYPE2 import functions>
}

SCENE_DATA_BLOCK import{
     GROUP_DATA_TYPE1 import
     GROUP_DATA_TYPE2 import
}
```