# Virtual Worlds Application Overview (draft)

## Purpose and motivation

A desire was to create 3d objects or worlds via a pure procedural generation method for the purpose of creating 3D worlds for first person game or simulation so as to create random landscapes or purposeful ones without the time consuming method of manually using a 3D creation tool like Blender. After some time of investigating, some applications that can do this on a limited scale, and documentation on how procedural methods are carried out, an idea sprung that instead of using cubic or Cartesian grid or matrix coordinate systems, there must be a better method of using a coordinate system that has evenly spaced center to center neighbours of 2D and 3D spacial shapes.

The hexagon was obvious for 2D, and after some investigation, the 3D equivalent was found to be the Trapezo-rhombic dodecaheron shape for 3D voxels that can be stacked to represent a hexagonal close packed spherical space, where the center of each sphere surrounding a central sphere are of an equal distance. From this a gradual progression and evolution of creating 2D surface and 3D procedural objects was developed. From this concept and further research into the display and editing of 3D objects, The original Virtual Worlds changed from the goal of creating 3D landscapes for a first person experience into one of greater potential of modelling and simulation of 3D and 2D mathematical generated objects to be used in 3D applications like Blender.

As a final step when Blender was found not to be easy to integrate such a concept into its design, the decision was made to create a fundamental and basic 3D editor environment to display and demonstrate the original concept and vision in the hope that others with greater experience, skills and know how can take this to a next higher level or contribute to help developing it.

### Choice of framework for the application

For quite some time an investigation was conducted to find a graphical framework to base the code writing on, two criteria was essential.

1 : The frame work must be based in C++ as this offered the greatest flexibility and documentation, as well as having the largest pool of resources and existing applications to refer to utilise or use as a guidance. The author was also most familiar with C++ than its derivative C# and saw no advantage in using any other programming language.

2 : The frame work must well documented or easy to understand, offer good support and be open source or at least be free to use on a personal basis.

3 : The frame work must offer a decent set of tools or resources to display 3D graphics onto the computer screen without having to know too much knowledge of openGL or Vulkan etc.

The best framework that offered these two criteria was initially Qt. Qt was not the option that was desired as it was far more than required, and it used and required too many .dll files to be included in distributions of applications to get them to run. It also had other undesirable attributes as being a commercial software product that may cease its open source availability at any time. Qt OpenGL tools and libraries also seemed to be more suited to displaying existing 3D objects rather than for creating and editing thereof. One last undesirable aspect of Qt was that it seemed that the company that created Qt were moving away from direct support of openGL tools and into creating tools just to display 3D objects into a scene for display purposes only. And upon release of version 6.x of Qt, it was found that creating and modifying projects became more complicated and prone to corruption as it used Cmake to compile code, and is not easy to use for the uninitiated. So the latest version of Qt 5 series was chosen.

Documentation for Qt is superb, user usage large and could find answers to problems without needing to go onto the forums. But since the aim was not to create a finished commercial product, but a demonstration or prototype of a concept, nothing else that was tried or investigated at the time was found to be as good.

And it was concluded the once the concept was proven, an alternative framework can then be looked for.

As of July 2021, a final proof of concept and application prototype was completed using Qt. After a brake of 3

months to decide how to proceed further, it was decided in early December 2021 that Qt was not suitable, as maintenance and modification or adding to existing code was inefficient and more difficult than desired to go to any next step. Looking for alternatives, the Qt QML framework was found to be easier to work with and existing core C++ code that was not related to the GUI or OpenGL was able to be used with very little modification. The main motivation to move to QML was the use of a View3D QML type that could display 3D graphics in a more efficient and easier manner that concentrated on providing the data to display graphics, and not having to deal with all the setting up a graphics display that OpenGL or Vulcan demands before even a single pixel is displayed.

However, QT QML was eventually found to be insufficient as well. The View3D QML could not handle geometry shaders, which was essential, and then QML was found to be getting slower as more GUI elements were added and other annoying minor issues being encountered that required workarounds, or no solution available.

In the end, the decision was to try out ImGui as the main GUI interface. After initially considering ImGui, and rejecting it because of lack of documentation and tools, It was discovered much of the Qt code could be substituted using public domain c++ string functions, c++ glm headers, and the Qt dialogues and file selector with a c++ functions library called tinyfiledialogues.

What was discovered was that the ImGui examples were sufficient to get a grasp of how to use ImGui with a few tutorials and forum posts giving some information of the pitfalls of using ImGui. What also helped was that the experience with Qt and QML coding gave a better and more complete understanding of how to use ImGui. What was discovered was that migrating the core non GUI code from QT to gml and C++ was simple. However, once again the undesired process of having to deal with OpenGl had to be contended with.

Most of the GUI interface had to be rewritten, but it was found that much of the complexity to get the GUI up and running in QT was simpler and straight forward in ImGui, even if the display of widget graphics options in ImGui is not as flexible. No multi colored pictures or glyphs are able to be used with Imgui widgets it seems. A bonus was that some desired interface functionality not able to be done in Qt/QML was easily possible with ImGui. Having to do away with Qt signals and slots increased flexibility, ease of use and reduced enabling code. Over all, for the purposes of this application, ImGui, glm and other C++ tools such as tinyfiledialogues formed the basis of the framework of this application.

In early January 2023, the final prototype of Virtual Worlds was complete using ImGui as a basis for the GUI, and a having a custom graphics engine to display and interact with the 3D graphics displayed in the applications view port. The design of the prototype was finalised and all was in a working order, but problems and issues existed that made using the application not as easy as desired.

Exhausted from coding for such a long period, a six month break was taken to think about if this project should be taken any further and it was decided that the project needed to be reworked to get into a more satisfactory condition. Another look at existing open source 3D frameworks was taken to this time integrate the existing code into one so as for easier coding and implementation. A new design of utilising base classes with virtual functions as the core was the go, and to have as much coding tasks as possible performed by the chosen framework.

Once again the candidates considered were the same ones looked at previously. Cinder and OpenFrameWorks were the major contenders, with a preference towards Cinder as its coding look top notch. But Cinder was not easy to migrate to as a previous attempt found issues and lack of documentation caused difficulties that had no work arounds that could be found. A closer look at OpenFrameWorks revealed it matched almost to the exact same design concepts of existing coding that was done for Virtual Worlds.

OpenFrameworks had examples of code that displayed how it works sufficiently enough that after some testing, it was found that the existing virtual worlds project could be implemented and improved using OpenFrameWorks. Also some of the examples were of use for the Virtuial Worlds application, with a large user base of addons that could be of use, and even it not thorough some documentation.

The only issue was that found out was that setting up an OpenFrameWorks project meant that the project resided within the OpenFrameWorks directory structure itself and could only easily be done through its project creation application. Trying to set up a project otherwise proved to be time consuming and ended in failure.

The use of OpenFrameWorks as a C++ as a framework for displaying 3D graphics has proven to be both problematic and advantageous. Advantageous not needing to code boiler point 3D code, and utilising its

inbuilt tasks and procedures to run and manage an application. Problematic in that it seems that its classes cannot be used to create derived classes from and expand without creating problems. Thus to expand or gain access to certain data variables with the OpenFrameWorks classes, the original code needs to be modified and added to. But given OpenFrameWorks is open source and this can be done in a safe manner that does not interfere with its basic functionality this has proven not to be a too great a problem.

The Virtual Worlds project utilises the ImGUI frameworks for is GUI and OpenFrameWorks for its application runtime management, and 3D graphics rendering.

# Design Concept

The basic design concept of the Virtual Worlds application is to have a central engine component that manages all the aspects of the applications interface and operations of performing all the tasks to generate, and display the Virtual Worlds voxel and other components onto the computer screen in an interactive and consistent manner.

This central application manager component acts as a main link between module component's that define and generate the data that is to be used, a computer memory memory manager to store the generated data, and a scene manager that handles the displaying of the data to a computer screen.(Fig 01)

Each module has within them the methods of displaying a user interface to interact with the module inputs and generation, display etc, which are accessed and coordinated by the main application manager driving the application as in fig 01. Each module can be considered as a kind of application plugin.

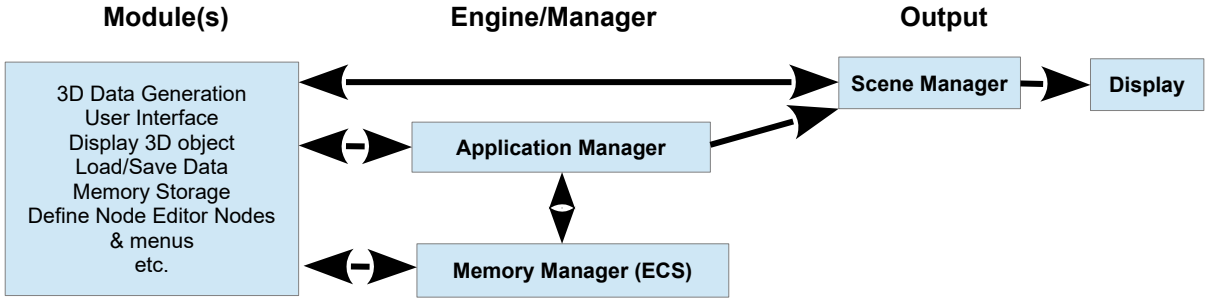


**Fig 01** Basic Virtual Worlds design concept

The application source code file structure is basically divided into two main directories. A directory called FrameWork contains all the source code files that are common and used or needed to be accessed by the modules and/or application engine. Another directory VW is the Virtual Worlds main application manager directory that store the various application engine source files, as well as the source code foe each module within a sub directory called Modules.

# 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.

## Code conventions

The C++ code conventions currently adopted is to have all the C++ code, unless it cannot be avoided to exist in .h header files only. This is to make it easier and quicker to edit and maintain a single file rather than having to set up separate header files and then having to create an associated .c source file. This allows it to be easier to read and track such things as variables, and virtual functions. Often when reading and tracking code written with separate header and source files, it was found that one needs two files open to determine such things like a class variable that is used within a class or even whether a certain function is inherited or not.

To make the code as readable as possible, abbreviations or condensing of class, variable,and function names are not performed widely, though is sometimes used. The naming convention is such that there is no upper case variables used and each word of a name is separated by an underscore to make reading easier and more natural. User defined macro variable names follow this rule but have all characters in upper case so as to distinguish them from a normal variable or class.

To make it as plain and as descriptive as possible for the user reading the code to understand the purpose or functionality of the code, the naming of all variables, data structure types , classes, functions etc is such as to describe as precise as possible what that variable, data structure, class, function represents or functionality is. By doing so then allows the user reading the code to better understand and interpret correctly the code written.

So as to make it easier to define when reading the C++ code what user created data type a certain variable name refers to, a convention is adopted as follows

All class types have the word class at the end.
All data structure types have the words struct_type at the end.
All enumeration types has the word enum at the end

# Design Operation

**User interface**

Fig 2 Virtual Worlds interface

The basic design of the Virtual Worlds application is to be similar in appearance and operation to any normal 3D application like Blender except in the aspect that the 3D objects created are done through a pure procedural method of the user using code to generate the 3D objects and displaying them. Because of the only real available method that can be applied within the authors skill set and abilities, and due to the poor resources offered on the internet in creating and displaying 3D objects on the computer screen, OpenGL was chosen to perform this task as it is widely used and has enough examples and documentation to get by for an interactive application that utilises parallel processing.

In addition to OpenGL, the author discovered that the generation of 3D objects can be performed utilising parallel processing of an additional OpenGL shader called a compute shader to perform all the generation of 3D procedural generated objects.

The basic interface of the Virtual Worlds thus looks on first impressions like most other 3D applications with a view port panel to view 3D objects. This version has no outliner to select and edit objects. Instead, there is a main node editor where the user selects, through a floating menu, nodes to be displayed in the node editor. By selecting a node, in a parameters panel, widgets related to that node is displayed for the user to interact with to modify or create data or what is displayed in the view port. The node editor operates similar all node editors in that nodes can have ports or pins that can link nodes together to perform complex tasks. A special node in the virtual worlds node editor is the group node. A group node is in essence to the user a kind of sub-directory where nodes can be defined to be displayed and even linked to upper level node groups.

The node editor is the main form of user interaction with creating, selecting, modifying and displaying of 3D data.

There is a panel of tabs that has selections to modify the global display parameters of the camera, lighting, and method of viewing the object through the camera. A tab selection is available to define overlays that are to be displayed in the view port and apply modifications to them.

A logging panel to display messages such as failed/suceeded tasks or debugging information.

All panels use the ImGui window docking system and can be arranged in any manner the user wishes.

**Memory Management system**

The basic design of the internal memory management system for the application to store, retrieve and use is a simple list of data in an array format for each type of data structure or category is depicted in fig 01.

The scene consists of object groups or categories of which the various procedural object types are to be associated or stored within. Each object is considered as an entity that has constituent elementary components that define the entity object. It was originally intended to have the memory management of object entities constructed as an entity component system (ECS) model. However, for the purposes of simplicity and to avoid too much complexity of memory management, the initial voxel and hex surface entity objects were not deconstructed into smaller entity components from their original C++ class object design, but allowed to remain as a single entity component or C++ object.

To cater for this basic design, a higher level C++ entity manger class is created to manage all of the creation, deletion and retrieval of entity categories and other management tasks.

The scene manager also manages the data that is to be rendered on screen via delegating the tasks to create, delete, access and render the graphical data used to render the generated graphics to screen to a the scene graph manager.

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.

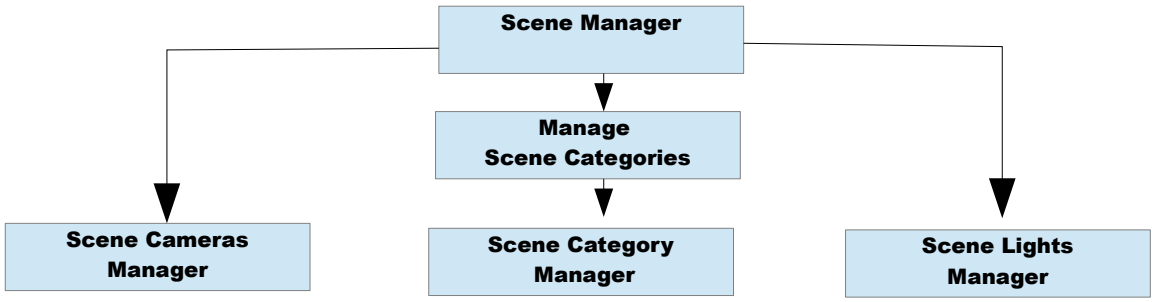


**Fig03 :** Schematic of Virtual Worlds memory system model

The scene cameras, lights and category managers can all be considered equivalent in their functionality, which is to manage the creation, deletion, retrieval and other management tasks of their respective data types, or categories.

### Entity module concept and design

The concept of the design for Virtual Worlds is that each entity module is a self contained system of code that contains all the C++ classes to create, generate and display onto screen the procedural generated data of that entity type. Virtual worlds does not facilitate a plugin system (though ultimately that would be desirable) so there is still need for code in the main editor of the application manager to be added to to enable the module to be added to be functional.

What each module must have are

1: classes to display the user interface of widgets to interact and modify the entity parameters for, and to generate the procedural data.

2: classes to display the resultant procedural data

3: classes to save/load data and the resultant generated procedural graphical data to import into third party applications.

The function of the main Virtual World application manager is to access the functions of these modules to operate on.

The existing design requires for the generation and display of procedural entity generation to be performed using opengl shaders, of which the user modifies a file of a subset to be added to an existing default shader code. This is presently done through a text editor that the user must provide. It is envisioned that in future this will be done within the application itself through an inbuilt text editor.

# Memory Management

## Scene

At the top level memory management is the C++ scene_entities_manager_class class (fig 3) object that manages all aspects of the virtual worlds data systems for a given virtual worlds scene.

The scene_entities_manager_class class object main task to to perform scene functionality such as creating, deleting or clearing the scene of all objects, and allocating the selection of objects to perform tasks on etc. It does not perform any functions on generating the data of entity objects, that is done through the user interface classes that the user interacts with and is discussed below.

The entity manager class manages all the entity functions that perform all the functionality of the storage and maintenance of the entity data pertaining to the entity objects that make up the scene to be processed and displayed on the computer screen.

The scene graph manager class manages the rendering of objects to the screen.

The scene_entities_manager_class is a virtual worlds frameworks class and the C++ source files for this class are located in the directory Source→VW_framework→scene as

scene_entities_manager.h

## Scene Category 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. However, virtual worlds uses data of different types that need to be stored and thus the scene category manager needs to be able to handle different types of data. This can be achieved by using a C++ base class that has common virtual functions and variables that all derived classes of different core data types use. Thus all scene category managers have as a storage data object, the same base object data class, that base class references a different scene data type and functionality. 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 04 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.

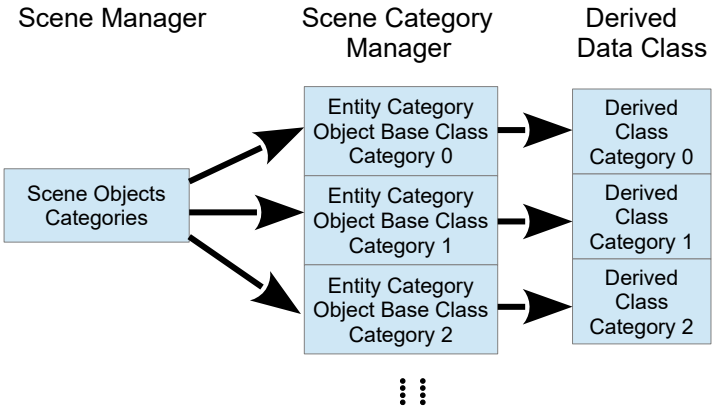


**Fig 04 Schematic of Virtual Worlds Scene categories data management**

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

# 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 custom 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 05c 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 which 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.

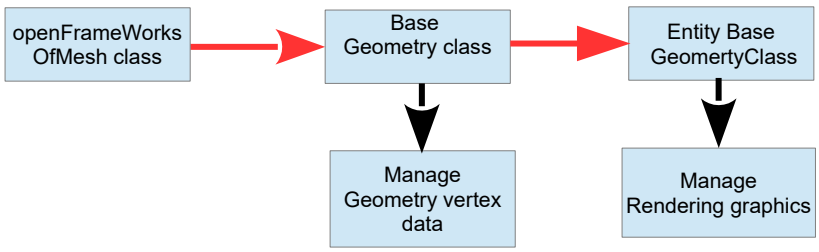


**Fig 05 Render Objects data model.Inherited classes in red**

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

# Object Base Class

In the section on the scene category manager, 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 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 06**

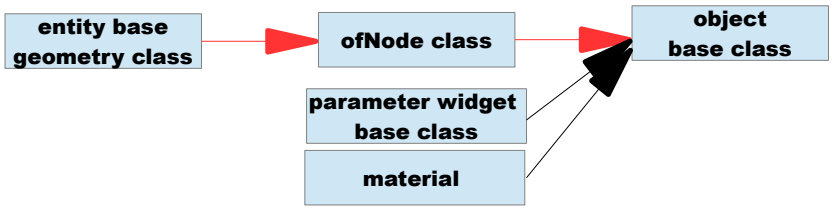


**Fig 06 Schematic of an object base class. Inherited classes in red**

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

# 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.

Thus 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 a 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 06** gives a schematic of the relationship of a parameter widget class to a 3D object base class.

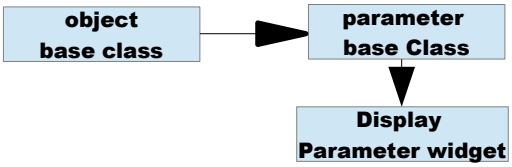


**Fig 07 Schematic of a parameter widget base class**

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

# Animation Object Base Class

In the same design and implementation concepts for the virtual worlds application wishing to perform time step or frame step animations for different object types efficiently and easily in an application engine would require a common animation base class with all the same functions and variables that any 3D object class would need to perform an animation of its geometry data.

Thus an animation object base class that can be inherited into derived animation class is created. This base class needs to set up animation parameters to define and be used in the animations process and be updated. The animation process itself is to perform one time frame step actions on the geometry data the animation object base class is associated with. This is usually performing a function defined within the C++ object class that the animation base object is associated with. A pointer to a 3D object class for example.

Fig 08 illustrates how an animation object base class is implemented. The animation base class is inherited by a 3D object animation class. The 3D object animation class points to a 3d Object class. This 3D object class has a function to generate, modify or perform some other function or feature on the geometry data on the 3D object. The Animation object data class has the function perform animation for frame which is a virtual function in the base class. When called by the application, this function performs its animation task by calling and performing a function defined within the 3D object class.

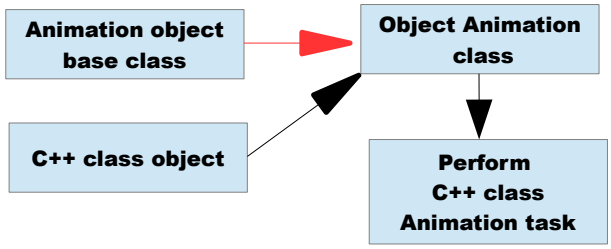


**Fig 08 Schematic of a parameter widget base class. Inherited classes in red**

By use of this animation object base class, the virtual worlds engine only needs to reference this common base animation class to perform animations for any C++ object class without needing to have knowledge of the C++ class object or the data associated with it.

The animation object base class widget base class is a universal framework class and exists in the directory

Universal_FW→Animation

with file name

animation.h

# Application Module

The application module was developed and conceived as a branch of the Visual Worlds application system that is mostly self contained to perform all the tasks of generating and displaying one specific object type to the view port, or as an output file to be imported into a third party 3D application like Blender.

At the heart of the module around which forms the basis of the code is a C++ data object class that models and manages a specific 3D object in computer memory and other aspects to render it to the computer screen. This C++ data object class inherits a object base class that contains all the storage and functionality to manage and display 3D data to screen. The derived application module class has all the additional structures, variables, and functions to store and manipulate raw 3D data that is in a different format or form to that in the the object base class, or has unique functions not present.

Thus an application module data object class has part of its function to act as an intermediary between translating or transforming 3D data from one form of computer memory to create and manipulate 3D data in computational form to 3D data in terms of vertices and faces etc.

The other functionality is to whatever is required to perform the tasks required.

Within the application module is code for the main application engine or manager to access to display widgets in the GUI for the user to input parameters and generate the data for the application manager to display and render on screen. The list of tasks for an application module commonly performs are

3D Data Generation
User Interface
Display 3D object
Load/Save Data
Memory Storage
Define Node Editor Nodes & menus

and are illustrated in **Fig 1** above gives a basic illustration of what functionality a Virtual Worlds module must contain to work with the main application engine.

Below are the module functions performed in Virtual Worlds for the voxel and hex surface modules that may be present in future additional application modules.

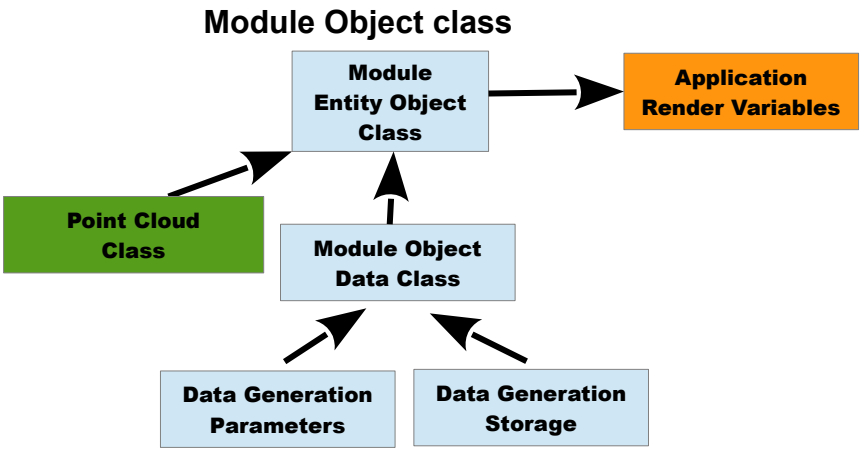


**Fig 09 Schematic of a General module object class structure**

The module object class is the heart of the module around which most other classes and code needs to refer to for their functionality and tasks to perform. The module object class stores the data that defines the object type and which is accessed for generation or processing within the module or in the main engine.

A point cloud class is part of the object class to store vertex data as a point cloud for rendering and any other purpose that requires vertex data. A separate object data class forms part of the object class to store parameter data to generate the object. If the data is not stored directly as vertex location as with the HCP voxel and Hex surface data objects, the object data class stores the objects generated data in whatever method is deemed appropriate by the programmer. In both cases of the HCP voxel and Hex surface data objects, this is a one dimensional array suitable for parallel GPU processing.

These are the common structural requirements that all application module object classes must have for the engine to use and is illustrated in fig07. The point cloud class is a universal class that all module object classes use and is located in the directory path

FW_framework→Geometry→vw_point_cloud.h

and each module object class should be present in the following directory for each module

Module name->Object

## Module Compute Generator Class

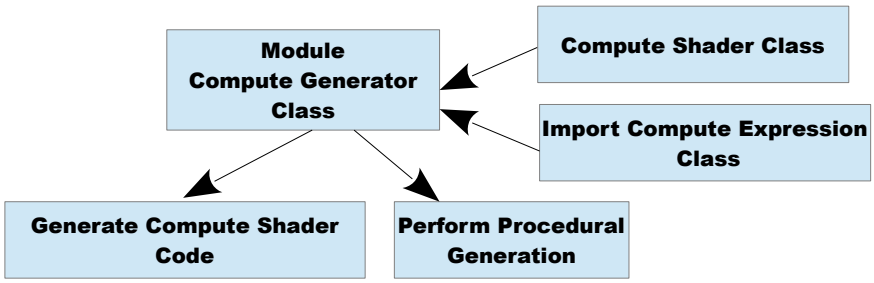


**Fig 10 Schematic of the Module compute generation class**

The module compute generator class is the C++ class that performs the procedural mathematical calculations and operations that create the point cloud data that models the module object. This can be done through use of a compute shader program to utilise the parallel capabilities of speed of the GPU instead of the CPU, and for the voxel and hex surface data objects which have their data stored as a one dimensional array this is the best method to use and is explained here.

The module compute generator class has as its basis an inherited universal compute shader class

VW_framework→Compute→compute_shader.h

The inherited compute shader class facilitates the creation of a compute shader program from a source file of text that makes up the compute shader code. The compute shader code is structured similar to any OpenGL vertex shader program in that is has certain sections of code that defines the shader program.

These sections are

1 : OpenGL version to use definition
2: Work group invocations definition of how the GPU is to process the computation
3: Reserved uniforms that all compute shader programs must have
4: User defined uniform variables to use
5: Inbuilt functions that all compute shader programs must have
6: User defined functions to use
7: Code of main function that all compute shaders must have
8: User code within main function to perform procedural generation of data
9: Code of output and end section that all compute shaders must have

The unhighlighted sections 3, 5, 7 and 9 suggests that these sections of code can be set as constant and unchanging, and thus exist in every compute shader for the module. If the OpenGL version in section 1 is set as being constant, then this can also be set as unchanging code. Section 2, as it was found in developing this class is best to allow the user to chose the group invocations so as to give the opportunity for the user to find or choose the best setting to use for the GPU to maximise its efficiency. The highlighted sections 4, 6, and 8 thus are the only sections of the compute shader that require the user to write code for.

With this in mind, a method to create the compute shader code was conceived for the sections 1, 3, 5 and 7 to be hardwired or predefined for the module compute shader program, while a method is devised for the user to create the shader code for sections 4,6 and 8 and then have all the separate sections combined into one compute shader program and compiled.

This is what the module compute shader class effectively does. Sections 1,2, 3,5,7, and 9 can differ between each module data object type,so these sections are defined within this class. A universal class import_compute_expression_class in the C++ header file

VW_framework→Compute→import_compute_expression_code.h

handles the reading of a user created text file with the user defined sections 4,6, and 8 into a separate text string variables that then are used to combine these sections and merged into one source code text variable for compilation.

The compilation is performed by a define_compute_program() in the in inherited compute_shader_class and

any errors found reported as a pop up window.

With a successful compute shader compilation, the procedural generation of the module data is performed and results saved in the module object data class ready for further processing of for rendering the data model to screen etc.

If the GPU cannot perform this task due to the limitations of the GPU processing of certain data types and/or memory structures, then this class would have the code written for that purpose and the above description of this class would be invalid, but the purpose all the same.

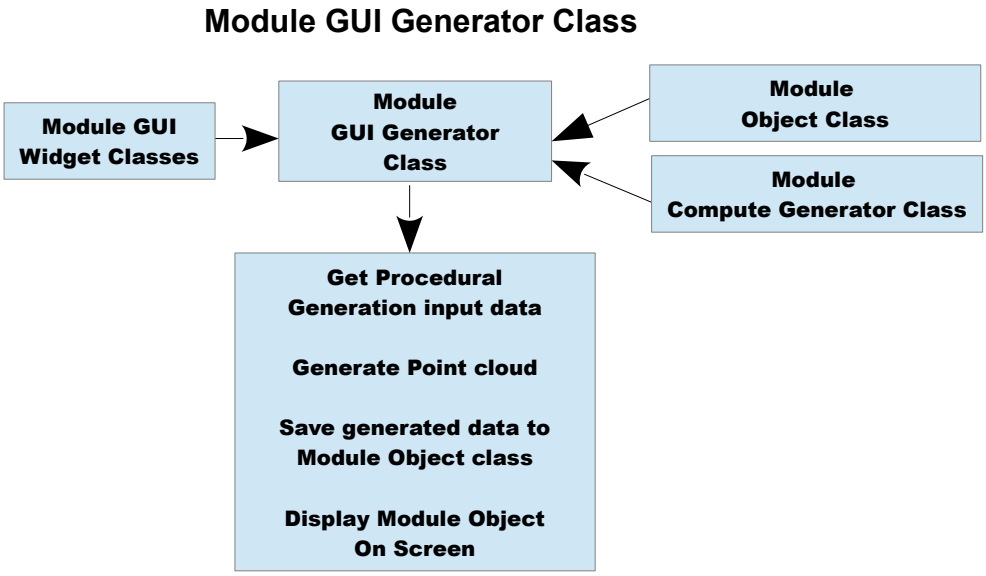


**Fig 11 Schematic of the Module GUI Generator class**

Each Virtual Worlds module has its own unique set of GUI classes to display widgets for the user to enter and modify parameters to generate and display the module entity object 3D data.

One module GUI class that every module must have is a GUI class that defines the widgets to enter and modify parameters that are used to generate the core entity object model. In the case of the HCP voxel module, this class is called voxel_function_editor_widget_class that exists in the C++ file

Modules→Module_HCP→Editor_Widgets→hcp_voxel_generator_widget.h

This generator class has a function called execute_voxel_function that takes all of the user inputs that are defined in the parameters to generate the module's entity object model data into a data structure that is passed to the voxel module object data class. This voxel module object data class is passed to the Module compute generator class which then performs the processing to generate the point cloud data and store that generated data in the voxel module object data class. Once this has been successfully completed, the voxel module object data class draw function is called to display the graphical representation of this point cloud model to the view port.

This is the main function of the GUI Generator class is common to all modules. Every other module would also have GUI widgets for user interaction and parameter inputs to perform a procedural 3D data generation, and a similar or different method of the procedural generation function. All modules in Virtual Worlds would have the basic design of the GUI class as in **Fig 07**. There are many other thing not included in **Fig 07** that must be present, but that is left to the source documentation or in a separate section dealing with what is needed to get the functionality to work with openFrameWorks.

Another function of the GUI Generator that may be optional is to have step or frame animation functionality where a step forward and step back function increments or decrements various parameter variables.A procedural generation is preformed to create new point cloud data that replaces the existing point cloud data stored in the module object data, and then display this onto screen. Such functions are called step_forward_generation and step_back_generation within the class.

## Module Animation Class

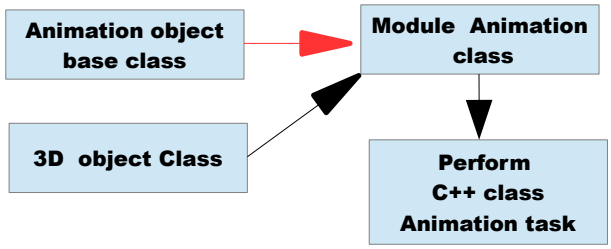


**Fig 12 Schematic of a parameter widget base class. Inherited classes in red**

Each Virtual Worlds Module has a series of animation classes that can be used to perform animation tasks. Fig 12 is a modified version of fig 08 where a 3D object class (hcp voxel or hex surface) is to have generation and/or shader variables animated. 3D generation and shaders have variables that have time frame step functions that can be animated. There also exist a cellular automata class that also has time frame step functionality and can be animated as well.

See the section animation object base class for a deeper explanation on how the animation class works.

## Module Export Geometry Class

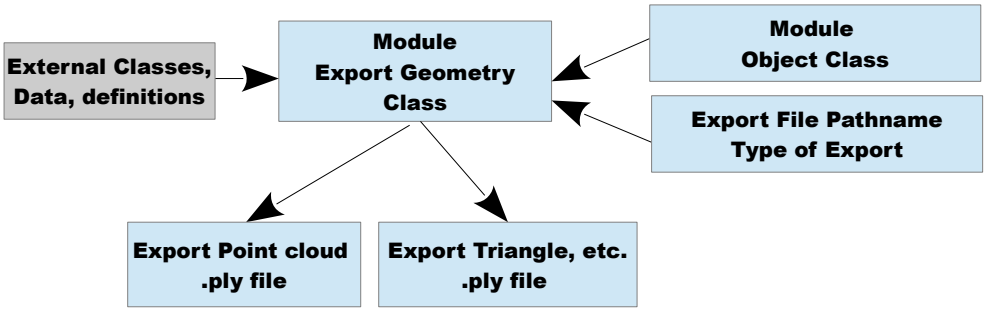


**Fig 13 Schematic of the Module Export Geometry class**

Each Module has an Export Geometry Class which is a class that handles the export of geometry data to a .ply for the purpose of importing into a third party 3D application like Blender. The type of geometry data to export is dependent upon the Module object data type, such as the generated cloud point data, or a function to transform the point cloud data into vertex data representing a surface or some other form.

In the case of the module for HCP voxel, which the generated data represents a volume, the data can be exported as a point cloud where each point is the center of the voxel, or as a surface where the volume surface is exported as a series of vertices making up a triangle.

Every Module would have the export functions specific to the module data type, and thus methods to generate export data, but the basic design is as in **fig 13**. The external Classes, Data, definitions are specific to the Module Object data type and export requirements and as such is an option and not be required.

# FRAMEWORK

Existing in the Virtual Worlds project directory structure exists a directory called Framework. This a kind of universal directory which contains as the word implies, a suite of standard universal C++ classes to define common classes, data structures and functions to perform the essential tasks that any application can use , or are specific to the Virtual Worlds application and engine but are referenced widely.

**Listing 01** gives the directory structure of the FrameWork directory.

```
Kernels
Universal_FW
VW_framework
```

**Listing 01 : 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. As of time of writing, this directory has the ImGUI docking source code and some custom ImGUI functions.

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 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.

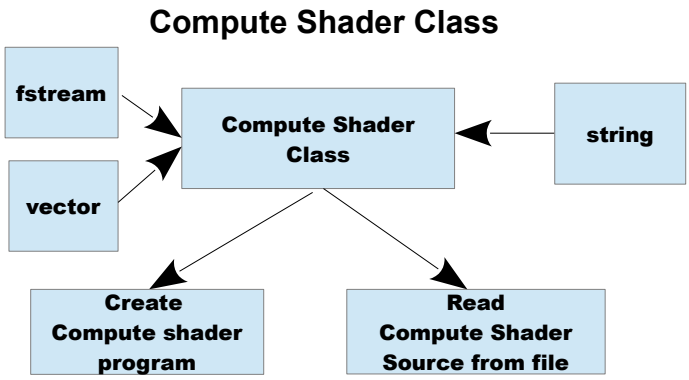


**Fig 10 Schematic of the Compute Shader class**

The universal C++ compute shader class (compute_shader_class) is the basis class for any C++ class that is created to generate or use a compute shader program to perform parallel processing by utilising the GPU of a graphics card.

Graphics_Engine→Compute→compute_shader.h

This class has the common attributes and functions that all derived compute shader classes need to have to be able to function.

The purpose of any class derived from this class or uses it, is to supply the compute shader code in the form of a string variable to create the compute shader program.

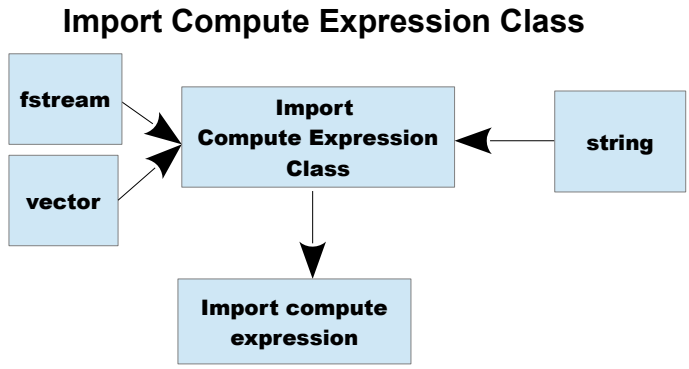


**Fig 11 Schematic of the Import Compute Expression class**

The universal C++ import compute expression class (compute_shader_class) is the basis class for any C++ class that creates a shader program of any form from the design method expressed in the Module Compute Generator Class section. This class reads a user defined shader code from a text file where in that text file the shader code is in the format

```
begin_function
        user created GLSL functions
end_function

begin_expression

        user created GLSL main function code to give an output value

        output_result = output value ;

end_expression
```

Between begin_function and end_function are the user defined functions that are to be used in the main glsl source code that the user enters between begin_expression and end_expression to create a value to output from the shader. The output_result must be present to do this and is a reserved word.

What this class does is extract the code between these two flag blocks as a separate text strings to be later merged together as described in the Module Compute Generator Class section.

In the Visual Worlds application, all shaders, including the compute shader uses this method to create the shader programs.

This class source code exists in the file of path name

Graphics_Engine→Compute→inport_compute_expression_code.h

# Shader

The Virtual Worlds engine to display 3D objects to the view port utilising openGL requires glsl shader programs to be coded and compiled, and changes or updates of the user interaction to dynamically change those the display with in the view port via shader uniform variables. To be able to do this, a shader class is required to create and manage shader programs and the associated uniform variables.

## Shader Material Base Structure

When displaying 3D objects in a view port, the appearance of a 3D object on screen is determined by how a glsl shader program is coded, and the parameters that are passed to that program that define how the object is rendered. Such parameters is defined as a material and if a standard data structure is defined to store this parameter data and standard functions to manage them, a material base structure object can be created and implemented in the same way as any C++ base class and for the same design reasons and implementation. Such a material structure type is illustrated in the schematic of fig 13

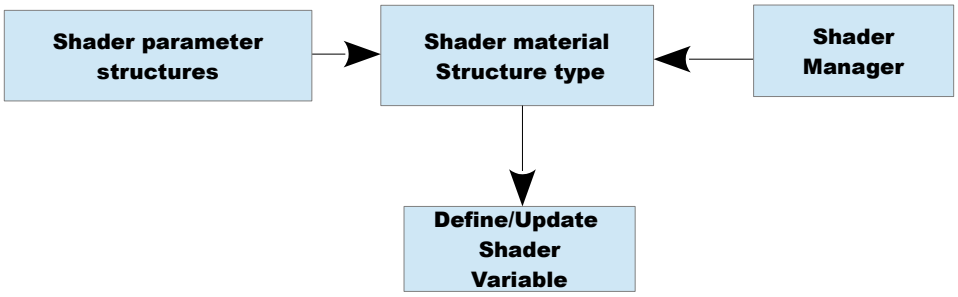


**Fig 13 Schematic of the Shader material Structure Type**

The shader material structure stores and updates the dynamically and interactive shader variables at run time for a given shader program that is defined for a class object. This allows shader code to be written with shader variables defined through the user interface and not needing to conform to hard coded variable names and types, or to recompile the Virtual Worlds application to facilitate shader variables.

This is achieved by use of opengl functions that locate and set glsl uniform variables within the shader manager class.

This class source code exists in the file of path name

VW_Framwork→Shader→shader_components.h

## Shader Class

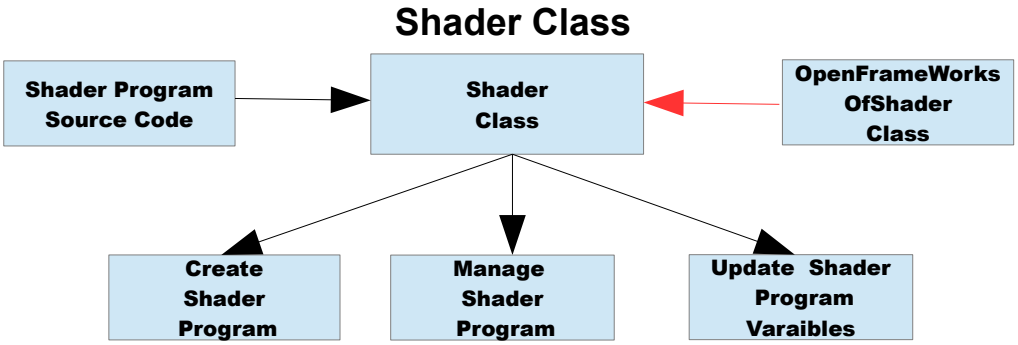


**Fig 14 Schematic of the Shader Class. Inherited class shown with red arrow**

OpenFrameWorks has it own shader class called ofShader. The ofShader class has most of the functionality needed for the purposes of the Virtual Worlds application, but needs some additions and refining to suite the needs of the Virtual Worlds application. So a C++ class, shader_class was created inheriting the ofShader class. The shader class creates and manages the OpenGL shader programs for a Virtual World data object to be rendered to the computer screen.

The vertex, geometry,and fragment shader code is passed to the function to create the shader program and other functions to use or release from use the shader program.

Functions for a shader program of a given ID value can have its uniform variable set or changed are defined in this class.

This class source code exists in the file of path name

VW_Framwork→Shader→shader.h

## Shader Format Class

The shader format class is the class that stores and manages the shader file path names, definitions of glsl variables, and glsl code that are to be compiled and used by any given entity object.

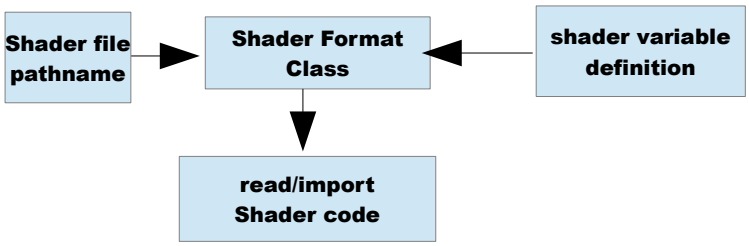


**Fig 15 Schematic of the Shader Format Class**

This class source code exists in the file of path name

VW_Framwork→Shader→shader_format.h

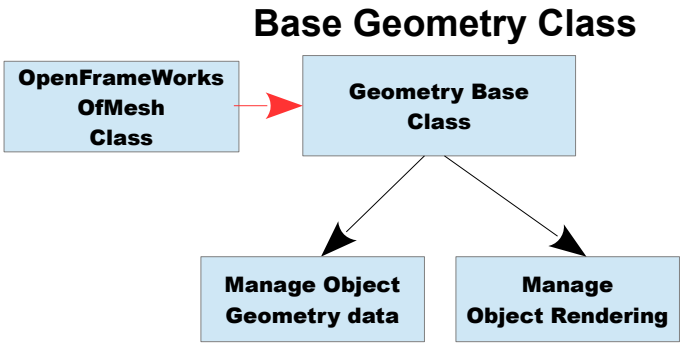


**Fig 16 Schematic of the Geometry Basis Class**

The geometry basis class is the base class that stores and manages the geometry data that is used to render 3D geometry objects to a viewport. The openFrameWorks class object ofMesh has all of what is required to do this, but lacked a few desired functions and variables. Thus a base geometry class was created inheriting ofMesh and adding a few extra functions and variables to be able to be used more effectively.

This class source code exists within the openFrameWorks directory file structure within a created directory called Geometry and has path name

libs→openframeworks→Geometry→vw_geometry.h

The reason this derived base class could not exists outside the openFrameWorks project directories was because of compilation and linking errors, as well as the application crashing due to openFrameWorks use of shared pointers.

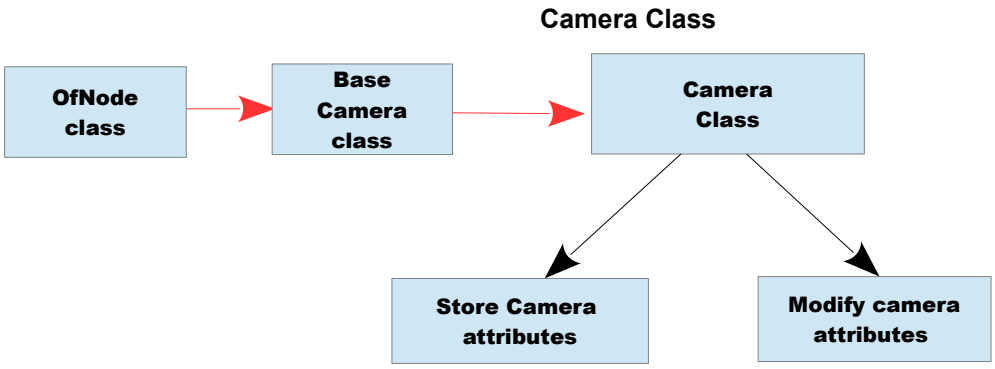


**Fig 17 Schematic of the Camers Class design. Inherited classes have red arrows**

The Virtual Worlds Camera class is a copy of the openFrameWorks ofCamera and ofEasyCam classes with added functions and variables. The ofEasyCam class utilises openFrameWorks events system that captures user mouse and keyboard inputs, ie events and does it in a manner that easily can be migrated and refactored form the previous incarnation of Virtual Worlds, reducing both complexity and code and moving the code for user interaction of the camera in the view port, as was previously away from the view port to the camera class.

This is the only ofClass that has been able to be successfully copied and separated from the ofFrameWork system and be used without issues. It performs all tasks one would expect from any camera class

This class source code exists in the file of path name

Source→FrameWork→VW_framework→3D→vw_camera_base.h

and

Source→FrameWork→VW_framework→3D→vw_camera.h

# Virtual Worlds Engine/Manager

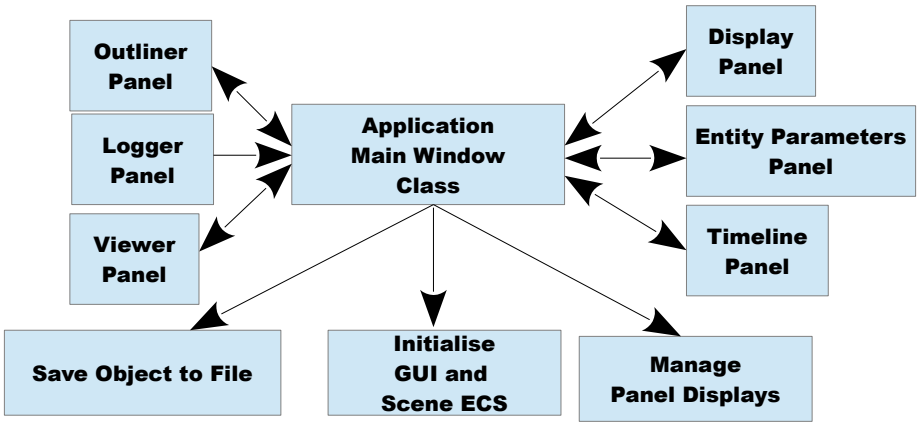


**Fig 21 Schematic of the Virtual Worlds main window design**

The Virtual Worlds engine or manager is the top most entry point to the Virtual Worlds application that brings together all the GUI and application classes or modules to manage them in a consistent and coherent manner so the user can create mathematical procedural graphical objects and display them on the computer screen. **Fig 1** illustrates the basic design of the Virtual Worlds application with the inputs being the classes and data structures described in the documentation elsewhere.

The design of the Virtual Worlds Engine to manage the many different classes and provide user to interact with the application is essentially to place the top root GUI widgets of the various classes into what are called panels, and then arrange those panels in the main window of the application. Such panels are illustrated in fig 2 where each section of the displayed main application window is a panel that performs a specific task or function.

Fig 21 illustrates the basic design of the Virtual World application main window where the double arrows of the panels indicate that the user interacts with the panel GUI, and the user inputs then are then acted upon by the class function that the interaction is linked to. As can be seen, each panel can or has a series of tabs that the user can select to bring up a new selection of options or interactions to define attributes and/or perform tasks. These tasks may involve fetching parameters or data that exists outside the class that defines each panel, panel tab or widget, and it is the function of the main window class to link or pass this parameter and/or other data between the various classes and their associated functions.

Eg the Entity parameters panel defines the virtual worlds entity data objects to be rendered, and the viewer panel defines the viewing window in which to display that generated entity data object. What the application main window class does is to create the viewer class associated with the view panel and pass a pointer to it to the class that defines the entity parameters panel.

There are  few functions that the main window class performs independent of any of the other tasks, and the only one present as of writing this documentation is to save one or more selected object data types to a file to be imported into a third party 3D application.

The source code of the main window class exists in the file of path name

Source→Editor→Main_Window→main_window.h

and the main window panel classes exists in the file of path name

Source→Editor→Main_Window→Panels→log_panel.h
Source→Editor→Main_Window→Panels→outliner_panel.h
Source→Editor→Main_Window→Panels→parameter_panel.h
Source→Editor→Main_Window→Panels→property_panel.h

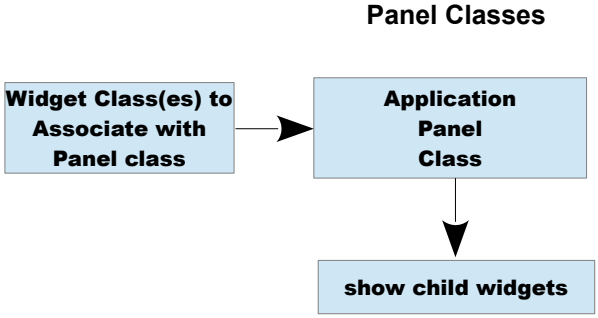


**Fig 22 Schematic of the Virtual Worlds main window Panel class design**

All the Virtual Worlds panel classes have the same basic design and structure as illustrated in fig 22. Each panel class essentially a widget parent class that has a child widget(s) class(es) that is (are) to be displayed within the defined panel class  Each Panel class is a means to construct a window that can perform scrolling, resizing and docking of the associated widget class.

# Virtual Worlds Outliner Manager Class

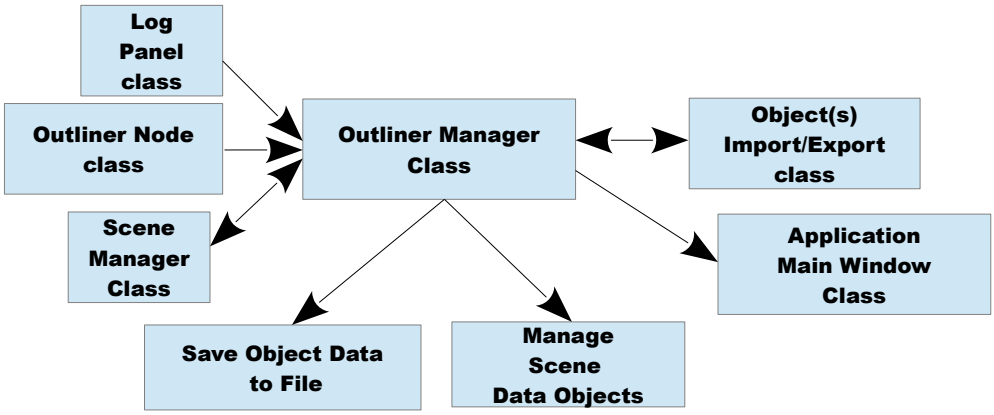


**Fig 23 Schematic of the Virtual Worlds outliner manager class design**

The Virtual Worlds scene outliner class is the applications main source of user interaction to perform tasks with the applications data management system, and gives a visualisation in table form of the structure of the Virtual Worlds scene in graphical form.

The outliner class role is to manage all aspects of the applications internal memory of entity data object for the virtual world scene that is to be generated and rendered onto screen. The basic virtual worlds outliner design is illustrated in fig 23.

Managing the scene data objects involves the creation, deletion and assigning of objects to outliner object groups or define selections to perform tasks for viewing, generating and animations through utilising the Scene manager class.

The display of the outliner tree for user interaction and processing of user action selections is largely done through using the scene manager to access the data objects that exist within the scene and displaying the associated data as an ImGui tree structure.

What this class also does is to manage the display of the currently selected virtual world object entity widget GUI in the entity parameters panel such that the entity parameter panel will display and allow user interaction with the scene object that is currently selected in the outliner panel. To allow this interaction to be possible, a variable with the selected entity id and object data type is passed to the parameter panel via the application main window class to be handled and managed to retrieve the correct data, and widget to display it on sceen to be interacted with.

The source code of the scene outliner class exists in the file of path name

Source→Editor→Kernal→outliner_manager.h

# Outliner Node Class

The Node class is a class that defines the node structure of the outliner tree that stores the information of each node of the tree that corresponds to an object within the displayed scene. This node data is used in the selection and viewing process as with displaying/editing the names and descriptions of each entity in the scene.

Source→Editor→kernal→outliner_node.h

## Entity Manager Class

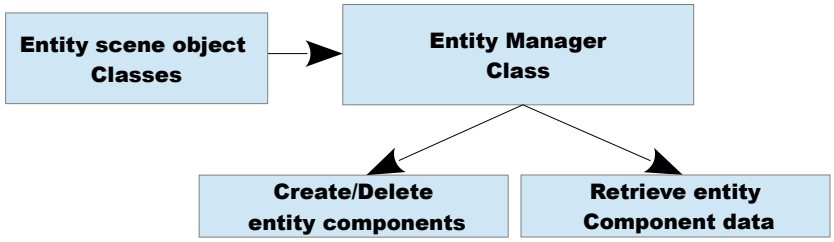


**Fig 24 Schematic of the Entity Manager Class design**

The entity manager class manages the storage of the entity object data types that are defined to be within the scene of the world. The management of creation, deletion and retrieval of entity data is currently the function of this class.

The source code of the Virtual Worlds scene class exists in the file of path name

Source→Editor→Scene→scene_entities_db_manager.h

## Entity Scene Object Class

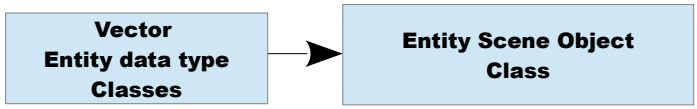


**Fig 25 Schematic of the Entity Scene Object Class design**

The entity Scene object class inherits a vector array of the entity scene object data type and has all the necessary functions to perform tasks to create, delete, retrieve and modify data to define the entity and display on screen.

The source code of the Virtual Worlds scene object class exists in the file of path name

Source→Modules→<Module name>→Editor→Scene→<entity_scene_object_name>.h

## Scene Manager Class

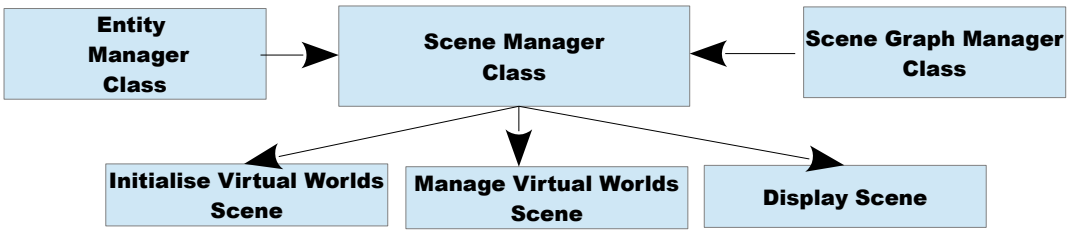


**Fig 26 Schematic of the Virtual Worlds Scene Manager Class design**

The scene manager class is the class that is the gateway to the applications Entity Component System (ECS) memory model for computer memory storage and management thereof. The entity manager classes that form part of this class enable the management of selection to display and perform tasks of only those entities that are selected for processing. The scene Manager is also the gateway to displaying the scene of objects to the computer screen.

The source code of the Virtual Worlds scene class exists in the file of path name

Source→Editor→Scene→scene_manager.h

# Editor Display Panel Classes

The editor display panel class is the applications panel that is present and static for the purpose of the user to modify or define attributes and aspects of the display of the view port that renders the Virtual Worlds scene of entity data objects onto the computer screen. As of writing this documentation, this panel has four tabs present to

1 : define a generalise lighting of the scene,
2 : the attributes of the computer camera model view,
3 : a selection of overlays and extensions to be displayed
4 : perform animations.

# Virtual Worlds Main Window Class

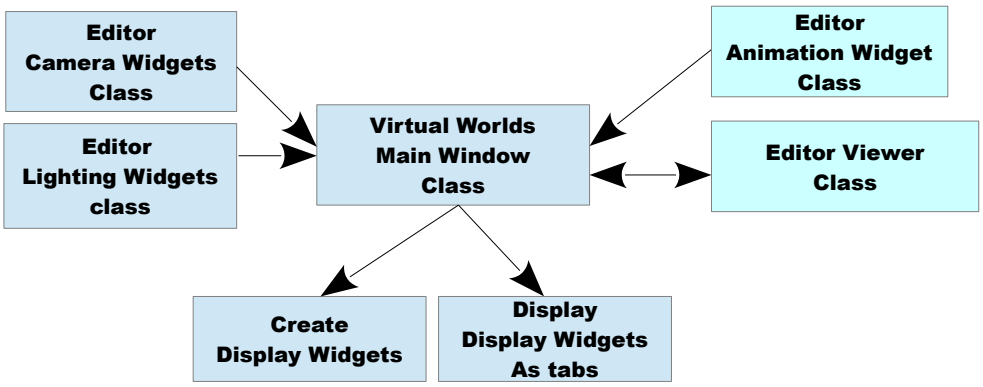


**Fig 27 Schematic of the Virtual Worlds editor Main Window Class Design**

The Virtual Worlds editor main window display class is the Virtual Worlds class that is the parent class for all child widget panel class objects that are a parent to widget classes that define the user interface to modify various aspects of the attributes that are used to display the virtual worlds scene of entity data objects in the view port. Thus the Virtual Worlds main window class acts as a bridge to all of the application panel classes that define and modify the scene objects to render in the view port.

Source→Editor→Main_Window→main_window.h

# Editor Lighting Parameter Widgets Class

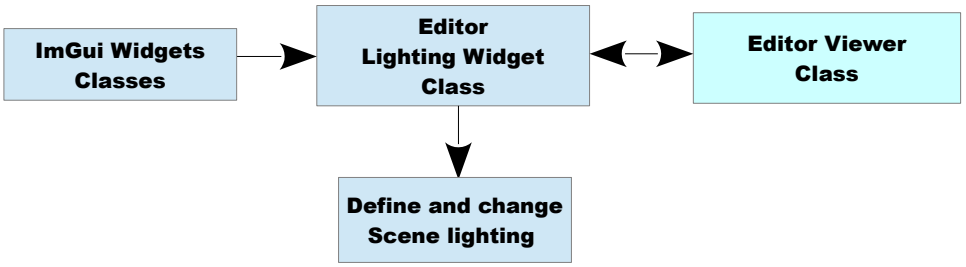


**Fig 28 Schematic of the Editor Lighting Widget Class Design**

The Editor Lighting Widget Class is basically a widget class that has functions to create a set of widget class objects that modify the world and other lighting attributes of the Virtual World scene rendered in the view port, and defined by the editor viewer class. The Editor Widgets Classes illustrated in fig 28 that depicts the design of this class is a C++ header file that contains a number of widget class objects to perform specific GUI functions.

The source code of the scene outliner group item class exists in the file of path name

Source→Editor→Main_Window→lighting_properties_widget.h

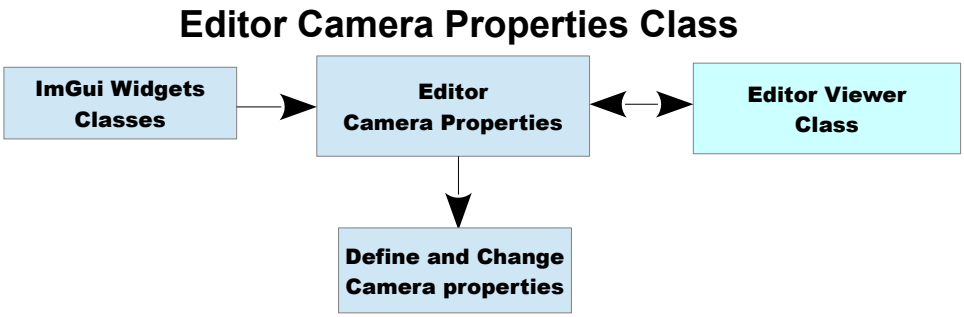


**Fig 29 Schematic of the Editor Camera Properties Class Design**

The Editor camera properties class is a collection of widget classes that has functions to create a set of widget class objects to modify the various attributes of the camera model that is defined and used in rendering the virtual worlds scene through the editor viewer class.

The source code of the editor camera properties class exists in the file of path name

Source→Editor→Main_Window→camera_properties_widget.h

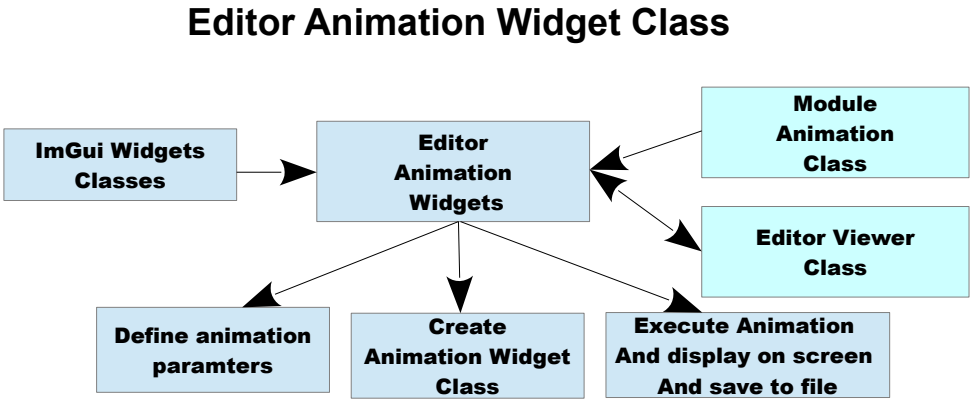


**Fig 30 Schematic of the Editor Animation Widget Class Design**

The Editor Animation Widget Class is the portal to define the Virtual Worlds application animation parameters and execute the animation functions to display in the Virtual Worlds viewer view port or to save as individual .ply files for each entity in the scene for each frame of the animation.

This Class has various controls of playing an animation and the saving of animation data, but in the end, the animation functionality is highly dependant on the entity data object type, and the animation parameters set for each and the shader programs define for each entity data object.

Because of the design of the Virtual Worlds application, this class needs to be modified and changed when a new Entity data Object type is added to, or subtracted from the application.

The source code of the scene outliner group item class exists in the file of path name

Source→Editor→Main_Window→animation_widget.h