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 initialy 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.

But the documentation 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 proto-type 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 disiplayed. Something That I did not want to deal with.

However, QT QML was eventually found to be insufficient as well. The View3D QML could not handle geometry shaders, which was esssencial, 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 inerface. 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 multicolored 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 inyfiledialogues formed the basis of the framework of this application.

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 aplication 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 manager in their own right.

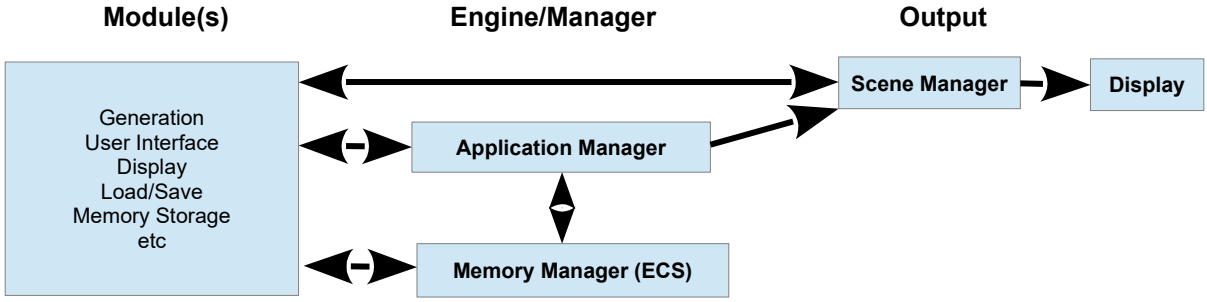

Fig 01 Basic Virtual Worlds design concept

The file structure of the application is basically divided into three main directories that also make up the basic design of the application. A directory called Universal contains all the include files that are common and used or needed to be accessed by the modules and/or managers. Within the Source Directory is the Virtual Worlds main application manager directories that store the various manager source files, as well as the source code foe each module within a directory called Modules.

(add more information here about file structure and what each sub-directory contains plus a directory diagram)

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 central viewer panel to view 3D objects, and an outliner panel to create, select and edit objects. There is a panel of tabs that has selections to modify the display parameters of the camera, lighting, and method of viewing the object through the camera as well as animating the objects within the scene. This is all very basic and primitive compared to many 3D applications.

There is an empty panel that is where all the parameters of each object type is to be displayed and changes depending on which object is selected via the outliner. Depending upon which object is selected, creation, loading and saving of data for a particular object is done within this panel.

At the bottom is a logging panel to display messages such as failed/suceeded tasks or debugging information.

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

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 that is defined for each module that is incorporated in the design as depicted in fig 01.

For the purposes of simplicity and to avoid too much complexity of memory management, the voxel and hex surface entity objects are not deconstructed into smaller entity components from their original C++ object design, but allowed to remain as a single entity component or C++ object. Thus there is no true Entity Component system (ECS) deployed, but the memory manager acts as a kind of ECS manager in that it handles the memory management of each different module (ie entity) data type as if they are a separate component of a larger entity.

In essence, the memory manager determines what entity data type is to be retrieved, or stored, changed etc, and then accesses the array or data to search for an entry of a given entity id, or to add to the end of the existing data within that array. The arrays actually store a pointer to the data that is to be operated on.

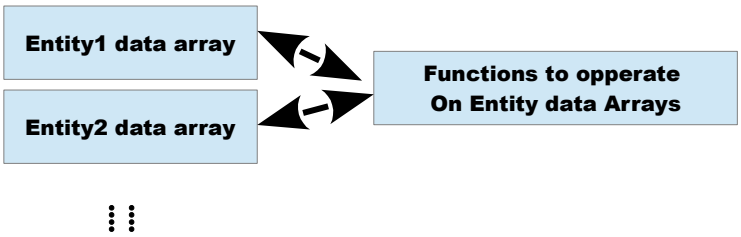

Fig03 : Basic design of Memory management system

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.

Scene Manager

The scene manager main task is to to perform scene functionality such as creating, deleting or clearing the scene of all objects and to handle all aspects required to display data to the screen. It does not perform any functions on processing the data of entity objects, that is done through the user interface classes that the user interacts with. The scene manager does store and handle data such as shader vertex buffer data and shader uniform variable data for each enity object that exists within the scene that is required to display data to the computer screen. The scene manager can be also considered as a kind of graphics engine as it performs many of the tasks of a graphics engine including the management of a viewport or window to display graphics to.

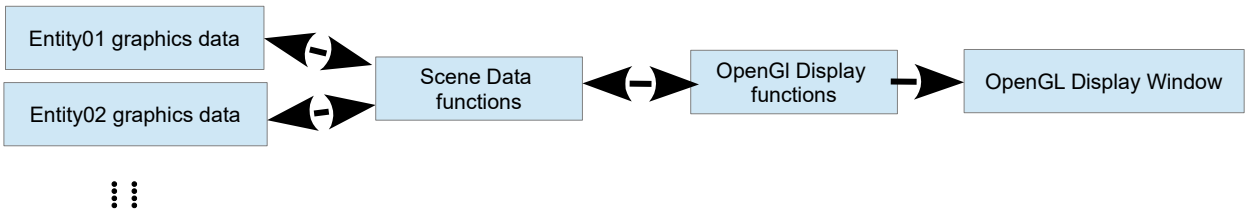

Fig04 : Basic design of Scene management system

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 computer screen 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 the desired 3D object in computer memory and other aspects to render it to the computer screen.

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

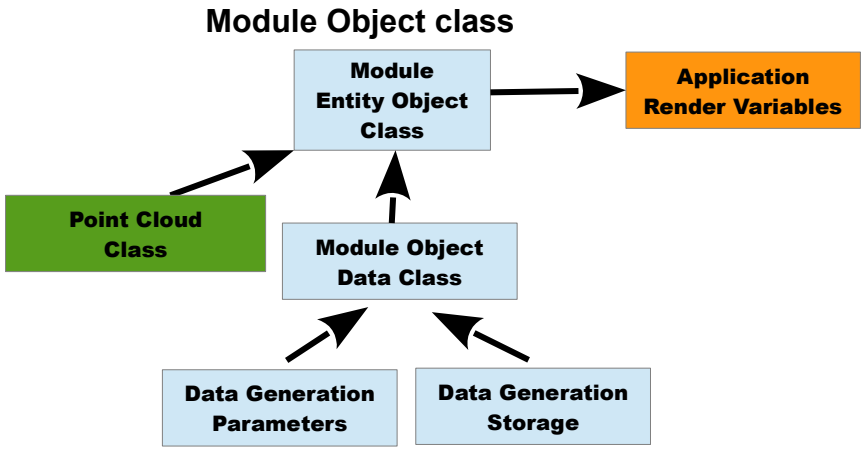

Fig 05 Schematic of a General module object class structure

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 task 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 deamed 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 fig05. The point cloud class is a universal class that all module object classes use and is located in the directory path

Universal→VW_Graphics→Scene_Graph→Point_Cloud→point_cloud.h

Module Compute Generator Class

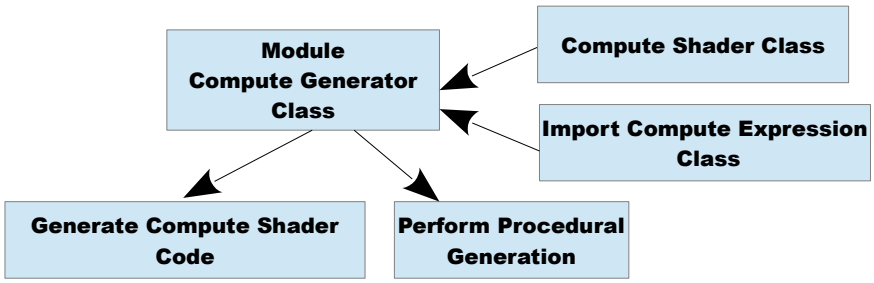

Fig 06 Schematic of the Module compute generation class

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

Universal→VW_Graphics→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

Universal→VW_Graphics→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.

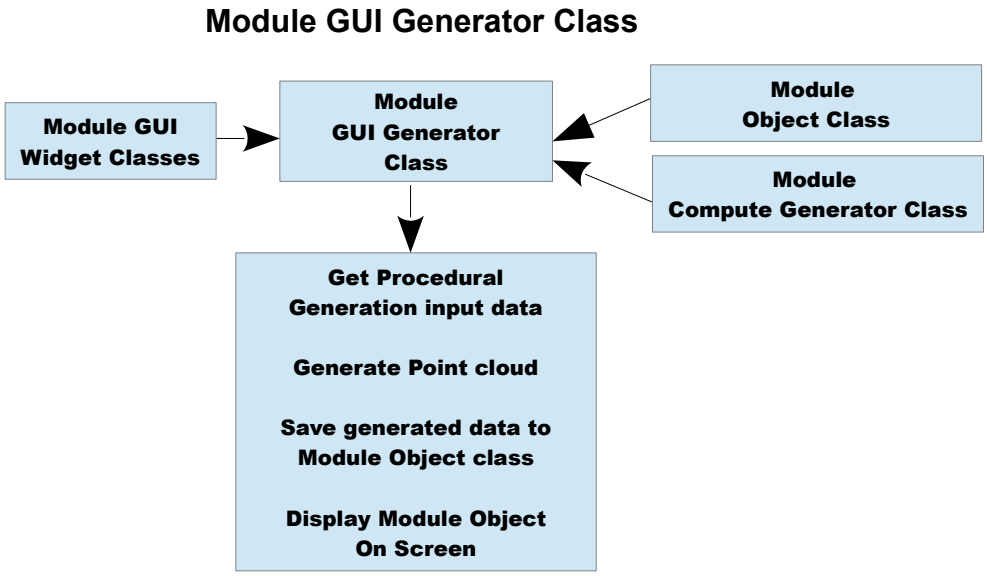

Fig 7 Schematic of the Module GUI Generator class

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 model and any variations or expansions of it. But one module GUI class that every module must have is the GUI class that defines the widgets to enter and modify parameters, and 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_Byte_Voxel→Editor_Widgets→editor_voxel_function_editor_widget.h

This class has a function called execute_voxel_function that takes all of the user inputs that define the parameters to generate the module's entity object model data from the widget entry values into a data structure, and passes this to the voxel module object data class. Then this voxel module object data class is passed to the Module compute generator class and 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 on screen.

This is the main function of the GUI Generator class that is common to all modules. Every other module that expands or modifies upon use of this module would also have GUI widgets for user interaction and parameter inputs to perform a procedural generation, and a similar or different method of the procedural generation function, but all in Virtual Worlds would have the basic design of the GUI class as in fig 10. There are many other thing not included in fig 10 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 Qt.

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, and then performs a procedural generation to create new point cloud data to replace the existing point cloud data stored in the module object data, and display onto screen. Such functions are called step_forward_generation and step_back_generation within the class.

Module Animation Class

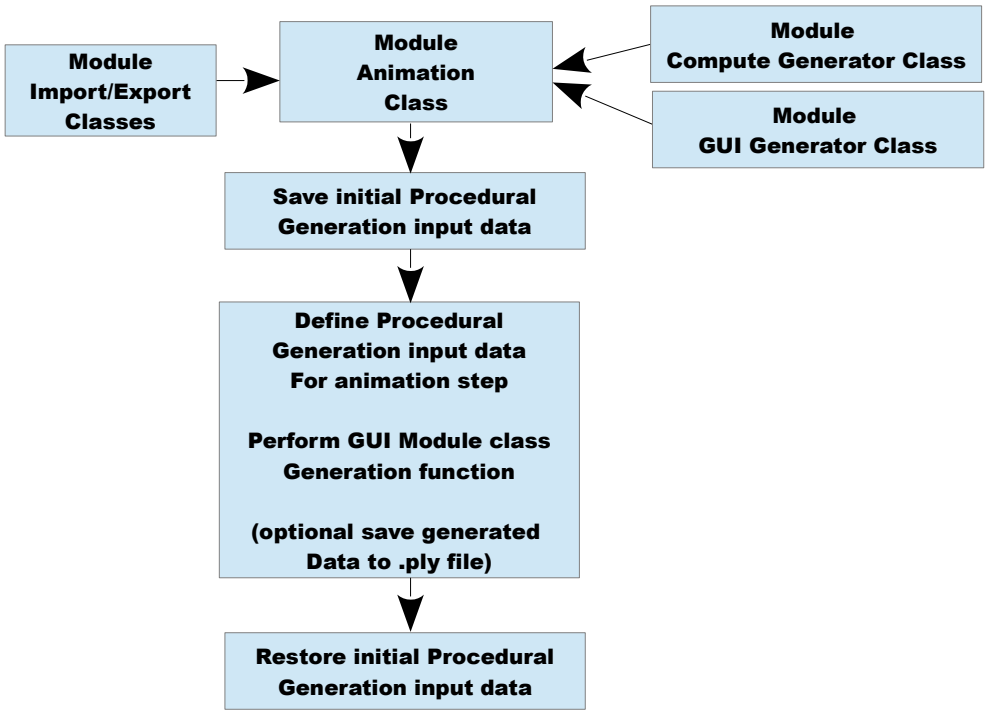

Fig 8 Schematic of the Module Animation class

Each Virtual Worlds Module that can perform frame step procedural generation of point cloud data as described in Module Generation Class section should also have an animation class. For the module voxel the animation class is called voxel_animation_functions_class stored in the C++ header file

Modules→Module_Byte_Voxel→Animation→voxel_animation_functions.h

The animation class is a set of C++ functions to be included as a class in the Virtual Worlds engine animation class and referenced to generate and display one or more module object(s) on the screen for each animation frame. To do this the module animation class gets all of the objects of the same module object type and data associated with each and stores all this in the class vector arrays.

Before any animation is performed, an initial state of each entity data object is saved so as to restore the scene to its initial state once the animation is concluded, or the user, through interaction with the application asks for it.

The animation proceeds one frame at a time for each module object by referencing the step function that exists within the GUI generator class and displaying the newly generated object model on screen. Once the last animation frame is finished, the initial state of the scene can be restored by simply retreiving the initial parameter data saved, and performing the procedural generation on this data and displaying the resultant module object data on screen.

Other procedural generation classes that expand or build upon the basic procedural generation class may also be included in the animation generation, adding some complexity and additional code. Both the Voxel and Hex surface object class has a cellular automata class that can also be animated as an example.

The animation class has some more complexity in that certain selections of module objects can be specified for animation, and by viewing the source code documentation, it can be seen how this is done.

Fig 11 gives the basic design schematic of each module animation class.

These animations may include changes to shader uniform values as well as the generation of the module object, and thus have animations of shaders. To see how this is possible and done, the source code documentation for the voxel or hex surface module should give a guide to how Virtual Worlds does this.

Each step of animation can also be saved as a .ply file for each module object. This is achieved by including functions to export the generated point cloud data in whatever form the user has specified or chosen, and using a standard file naming convention to identify each object and animation frame the .ply refers to. In the design of Virtual Worlds Modules, each module has an export geometry class for this purpose. This class has export functions that accept a file name and the module object data object with what kind of data is to be created int exported .ply file. Eg point, surface etc.

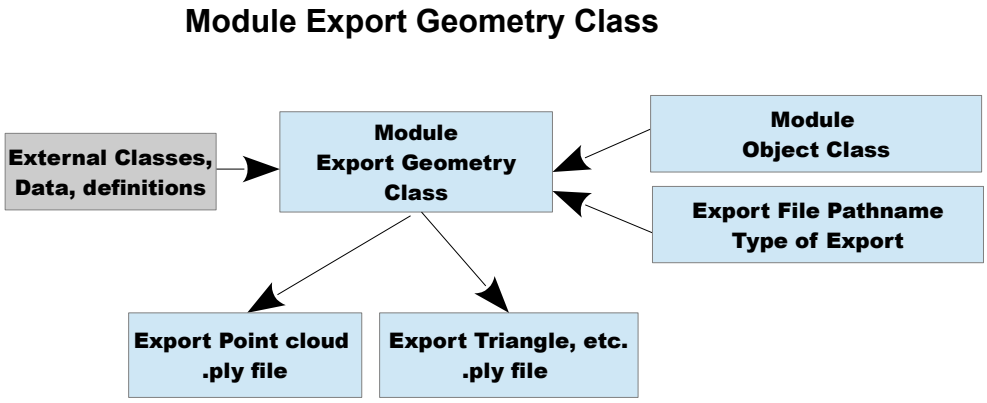

Fig 9 Schematic of the Module Export Geometry class

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 should be as in fig 12. 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.

UNIVERSAL

The Virtual Worlds universal directory contains as the word implies, a suite of standard universal C++ classes to define the classes, data structures and functions to perform the essential tasks that any Virtual Worlds class and/or module can use or need for its function.

As at the time of documentation, almost all the code and libraries contained within this directory are a copy of code provided by a third party that is essential or provides functionality required by the virtual worlds and other applications.

Graphics Engine

Compute Shader Class

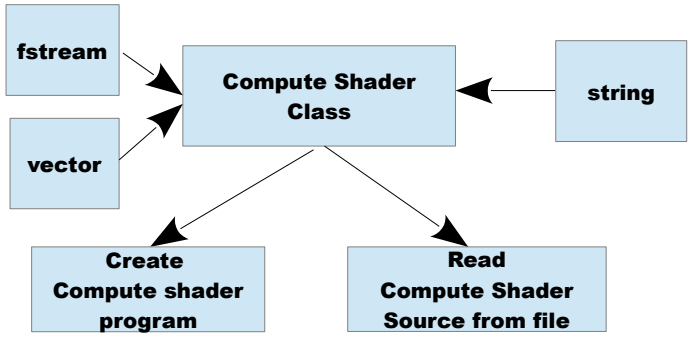

Fig 10 Schematic of the Compute Shader class

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.

Import Compute Expression Class

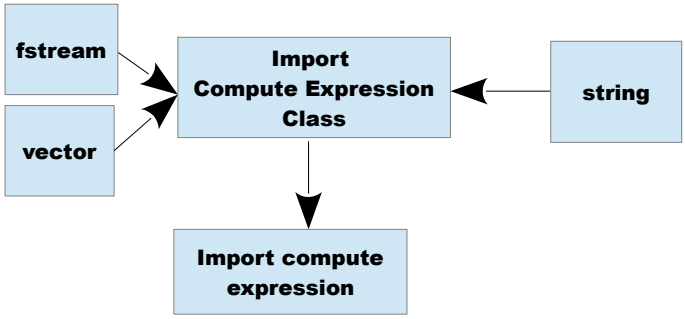

Fig 11 Schematic of the Import Compute Expression class

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

Render

The Graphics Engine render classes are a series of classes that manage the creation and storage of the OpenGL vertex buffer files that make up part of a base render class.

Render Object Class

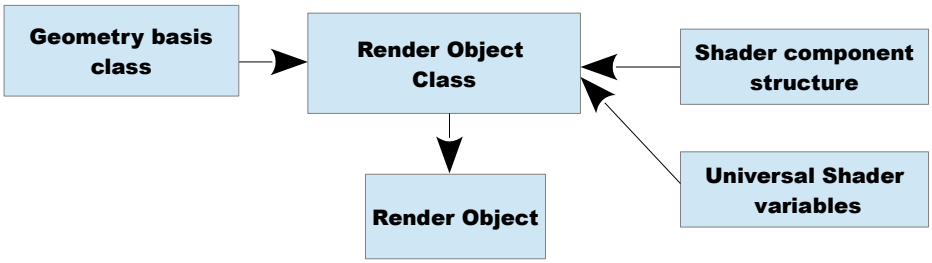

Fig 12 Schematic of the Render Object class

The render object class handles the display of each scene object onto to the computer screen. The render object class stores the geometry and shader data and functions to display any object onto the computer screen, and performs the functions to render that object when called. The universal shader variables is a structure class that stores and updates a set of universal shader uniform variables that every shader will need to function such as camera position.

This class source code exists in the file of path name

Graphics Engine→Scene→Scene_objects→render_object.h

Shader Components Structure

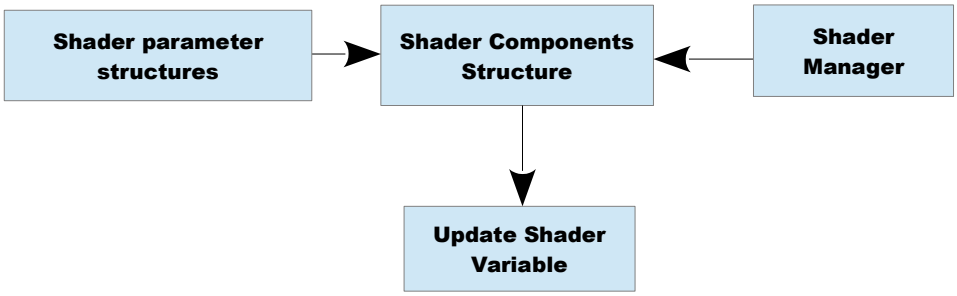

Fig 13 Schematic of the Shader Components Stricture Type

The shader components structure stores and updates the dynamicly and interactive shader variables at run time for a given shader program that is defined for this 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

Graphics_Engine→Shader→shader_components.h

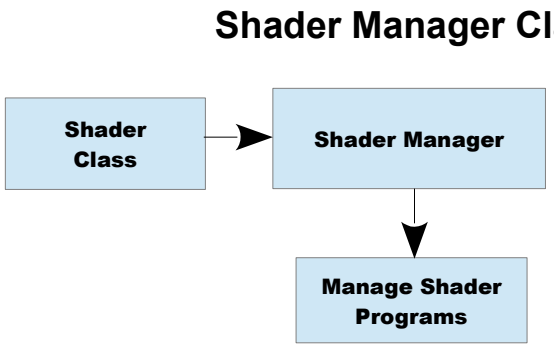

Fig 14 Schematic of the Render Shader Manager Class

The shader manager class manages the creation and storage of all shaders created in the Virtual Worlds application.

This class source code exists in the file of path name

Graphics_Engine→Shader→shader_manager.h

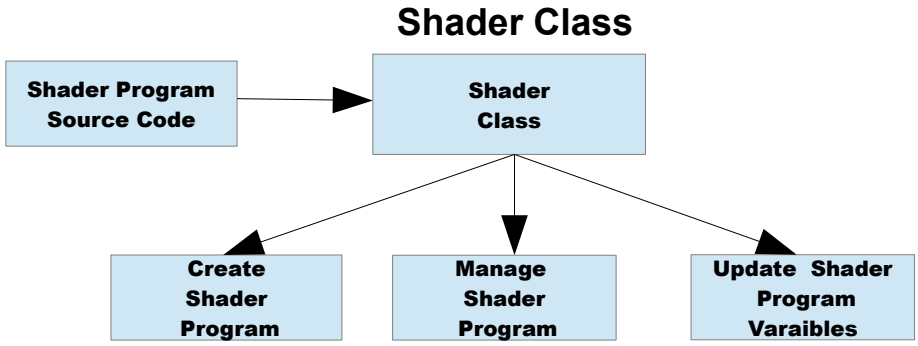

Fig 15 Schematic of the Shader Class

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.

Graphics_Engine→Shader→shader.h

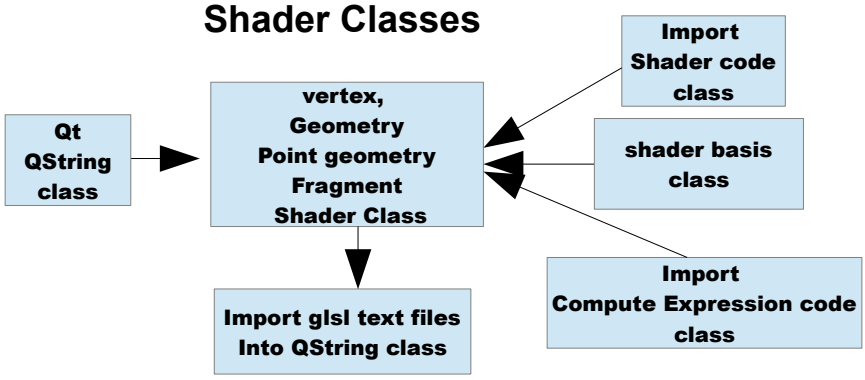

Fig 16 Schematic of the Render Object Shader Class

The virtual world shader classes are the basic OpenGl shader types of vertex, geometry and fragment shaders as a C++ class form. These shader classes have a common structure and functionality and if one is to look at the C++ code for each, one would find the code is very similar. Because the the basic glsl code that each shader type has as a basis for its functionality is different, a separate C++ class for each shader type is created. Eg a vertex buffer is the first shader of the graphics pipeline to import vertex and other data to be rendered on screen, while the fragment shader is the last shader of the pipeline and is used to output the final color of the pixels to be displayed. Thus a fragment shader would not need to import vertex data as part of its function.

What each shader type C++ class does is to import a mandatory reserved code for that shader type, user created glsl code, and user defined variables into a single Qt QString class of glsl code to be compiled. The path names to these files are to be specified from an outside source.

The resultant glsl code for each shader is then available as a Qt QString class variable to be accessed for compilation into an OpenGL glsl shader program.

Fig 16 illustrates the design and functionality of the shader classes which are all the same. As can be seen, a common shader base class is inherited by all shader classes.

The point geometry shader is a geomerty shader class to manipulate and display the object data class point cloud vertex data as points in whatever configuration the user pleases. Eg the Hex surface data object is displayed as hexagonal columns.

The geometry shader is a geomerty shader class to manipulate and display the object data class point cloud vertex data in whatever configuration the user pleases. Eg for hcp voxels, the geometry shader displays each point as a Trapezo-rhombic dodecahedral voxel.

The class source code exists in the file of path name

Graphics_Engine→Shader→vertex_shader.h
Graphics_Engine→Shader→geometry_shader.h
Graphics_Engine→Shader→point_geometry_shader.h
Graphics_Engine→Shader→fragment_shader.h

Shader Basis Class

This basis class contains the string class variable to hold a shaders glsl code and a function to specify which OpenGL version to include in the shader glsl code. This class also has a function to define a header in the glsl code for which OpenGL version is being used.

This class source code exists in the file of path name

Graphics_Engine→Shader→vw_shader_basis.h

Import Shader code Class

Thie import shader code class is a small C++ class that all shaders use with the one function of reading a text file of any kind of a given file path name into a string class variable, and returning that string variable to the calling function.This class can thus be used for any purpose of reading any text file into a string variable and returning the string variable and is why this is not part of the shader basis class. Renaming this class to something like import text file class is to be considered.

This class source code exists in the file of path name

Graphics_Engine→Shader→import_shader_code.h

Import Compute Expression Class

The functionality of this class is explained in the section Import Compute Expression Class above. The user defined OpenGL shader glsl code is the same as for a compute shader, and thus this class is used in

creating a user defined glsl in the same manner as that for a compute shader.

This class source code exists in the file of path name

Graphics_Engine→Compute→import_compute_expression_code.h

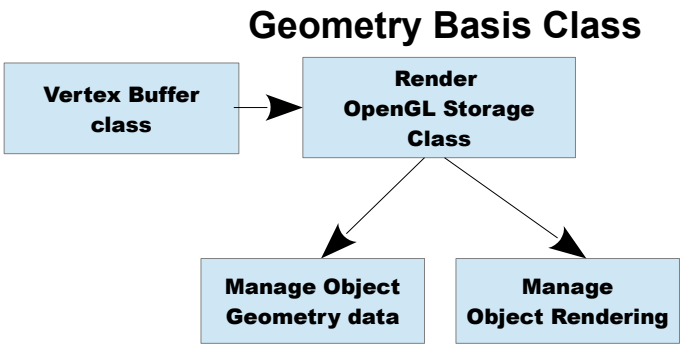

Fig 17 Schematic of the Geometry Basis Class

The geometry basis class is the base class that all objects that are to be rendered onto the computer screen must inherit into a derived geometry class. This basis class is what is referenced by the Graphics engine render class to draw whatever vertex or other data that is defined onto the computer screen, depending upon the type of geometry data that is stored in the associated vertex buffer class.

All functions defined here are virtual, and it is expected that the appropriate functions will have the user create the code for them to operate correctly, and additional functions will be included in the derived classes where necessary. There most important is an update-geometry function that must be defined for each derived class to replace the vertex buffer with new data to allow dynamic changes to the geometry being displayed on the computer screen.

Create_buffers and delete buffers are different for each object data type and needs to be coded in the derived class, and used by the update_geometry function.

This class source code exists in the file of path name

Graphics_Engine→Geometry→geometry_basis.h

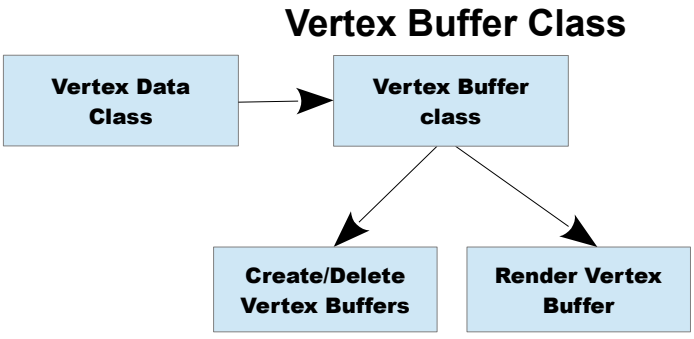

Fig 18 Schematic of the Vertex Buffer Class

The vertex buffer class is were the different vertex buffer type creations, deletions and rendering functions are defined. This one file can be separated into different classes if required for each vertex data type, and then inherited into this one class if required. It is envisaged that as new or other vertex buffer types are required, this is the class where these new buffer types are defined.

This class source code exists in the file of path name

Graphics_Engine→Render→Buffers→vertex_buffer.h

Vertex Data Class

The vertex data class is in fact a series of several different vertex data type classes that are defined to be used as a data type by the vertex buffer or other class to store a specific form of vertex data. These classes are all similar and very simple in structure. They could be considered as a structure rather than a class as they only contain the data to be copied into a vertex buffer object for the purposes of being used by a glsl shader program to display the data onto screen, or to be used in a compute shader program.

This class source code exists in the file of path name

Graphics_Engine→Geometry→vertex_data.h

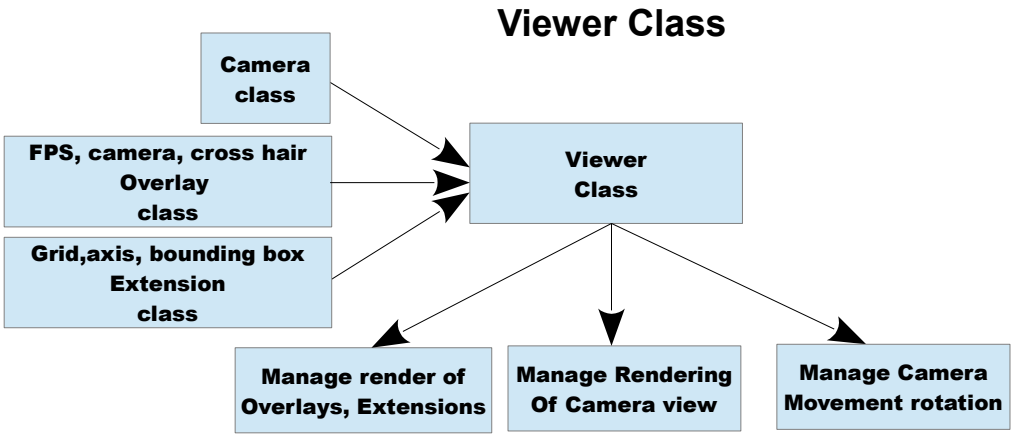

Fig 19 Schematic of the Viewer Class design

The viewer class defines the Virtual Worlds 3D graphical view port depicting a scene containing the object data types that the user has placed into the scene.

The viewer class manages the user interaction in the computer generated 3D world via the manipulation of a computer generated camera defined by a C++ camera class object. The Viewer class also manages the rendering of additional graphics to aid the user in interacting in the generated 3D computer world such as overlays of informative text (eg the cameras orientation and position) and graphics (eg a grid of the zero plane).

As can be seen in Fig 19, the viewer class is a class that incorporates and inherits other classes into its structure to perform its functional tasks and purpose.

There are three steps to render a camera view. A pre draw, draw and post draw step, of which there are also functions in the viewer class of these names. As the names of these steps and associated functions imply

predraw :– prepare varaibles and/or other requirements that are needed before rendering the virtual world scene.

draw :– Render virtual world scene using OpenGl with all the defined OpenGL objects and graphics.

postdraw :– Render additional graphics to the view after the completion of the draw step function.

These step functions are part of every frame of rendering to the computer screen when an update call is made in the application.

This class source code exists in the file of path name

Graphics_Engine→Viewer→scene_viewer.h

Camera Class

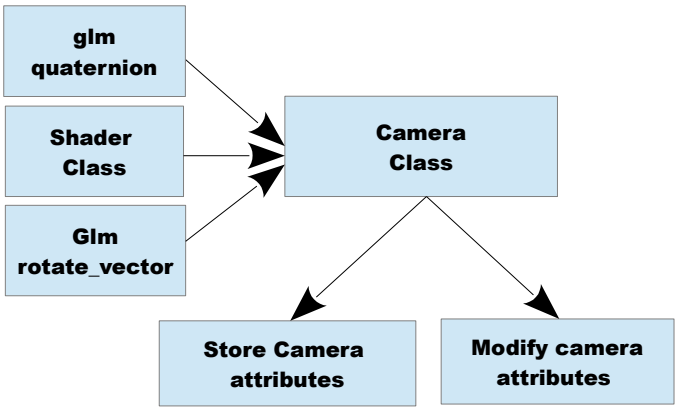

Fig 20 Schematic of the Camers Class design

The Camera class is the C++ class that defines the camera or users eye view into the computer generated scene of graphical objects. The camera class has functions that define, modifies and stores the camera view (such as perspective) and the cameras location and orientation in the computer generated 3D virtual world.

This camera class may look very similar to the camera classes of many other 3D applications as it has many of the same features and functions.

This class source code exists in the file of path name

Graphics_Engine_Scene→Scene_objects→camera_object.h

Virtual Worlds Engine/Manager

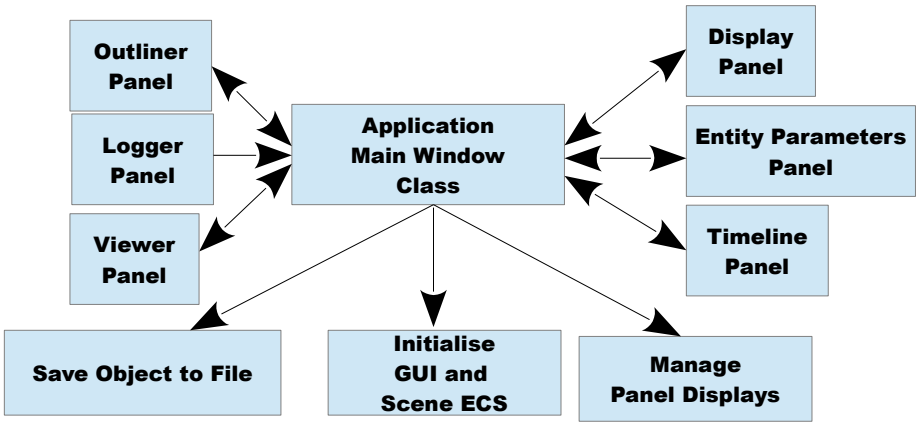

Fig 21 Schematic of the Virtual Worlds main window design

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 arranger 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

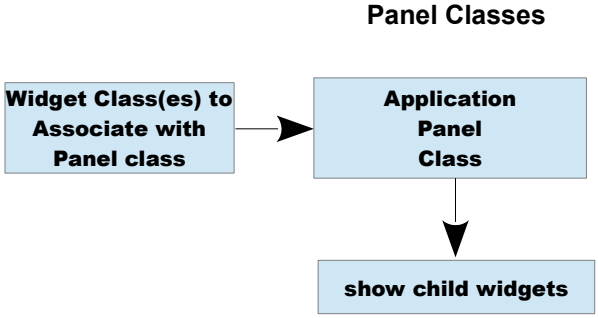

Fig 22 Schematic of the Virtual Worlds main window Panel class design

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

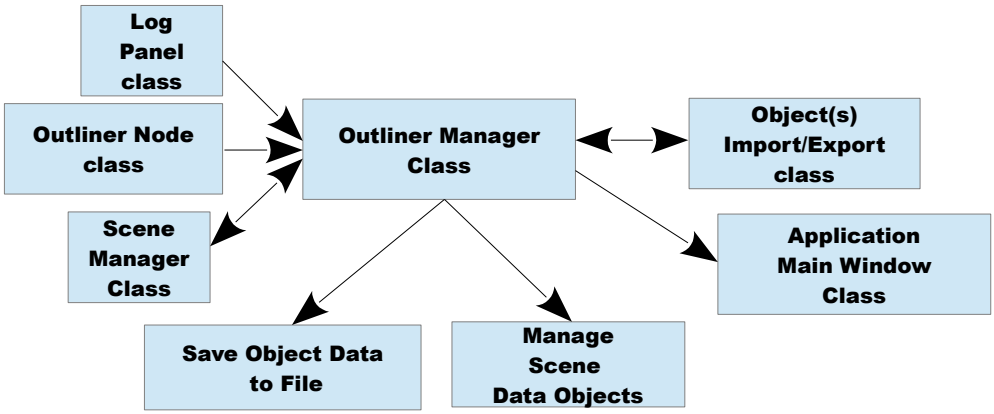

Fig 23 Schematic of the Virtual Worlds outliner manager class design

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

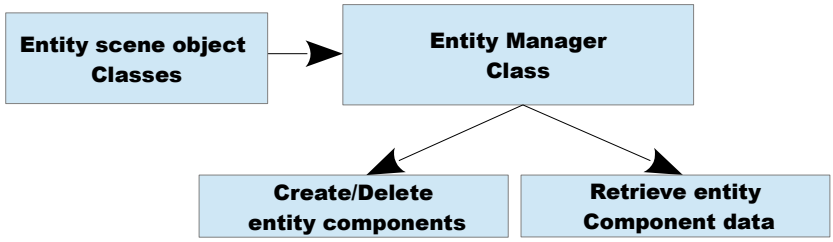

Fig 24 Schematic of the Entity Manager Class design

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

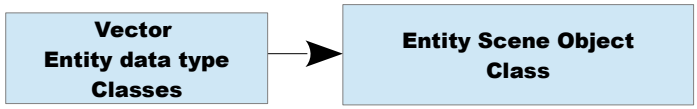

Fig 25 Schematic of the Entity Scene Object Class design

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

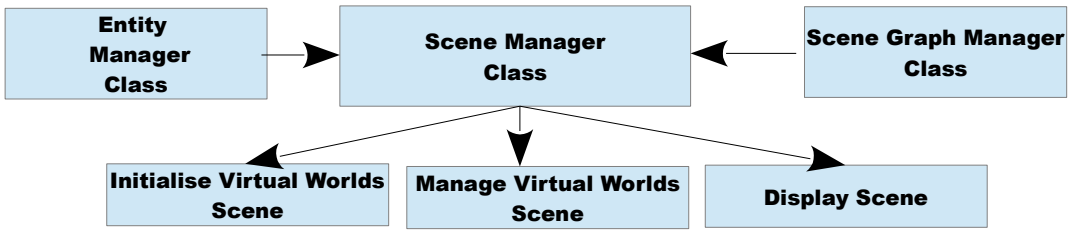

Fig 26 Schematic of the Virtual Worlds Scene Manager Class design

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

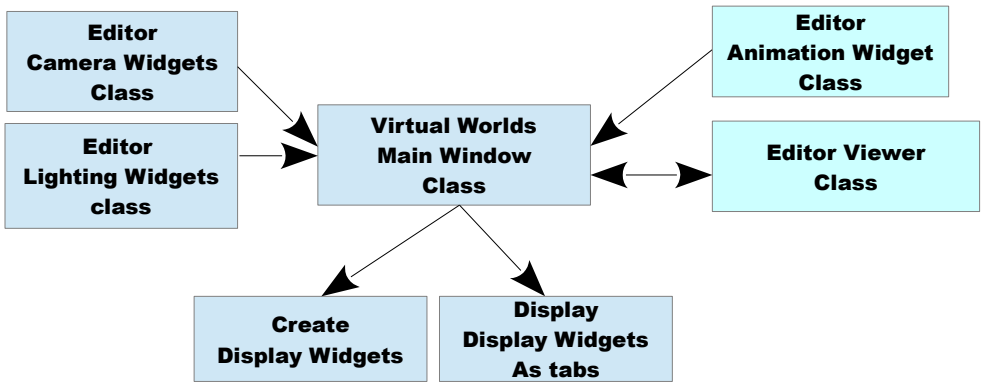

Fig 27 Schematic of the Virtual Worlds editor Main Window Class Design

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

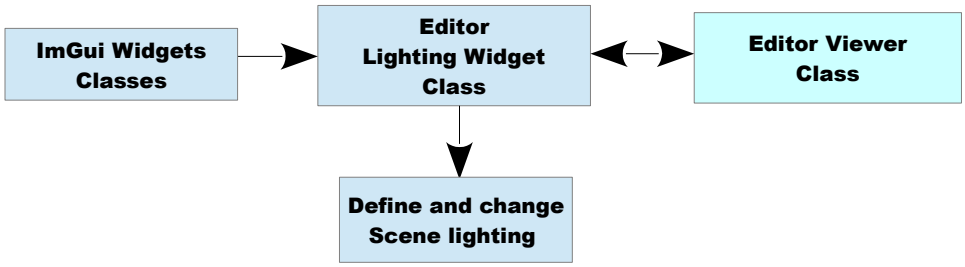

Fig 28 Schematic of the Editor Lighting Widget Class Design

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

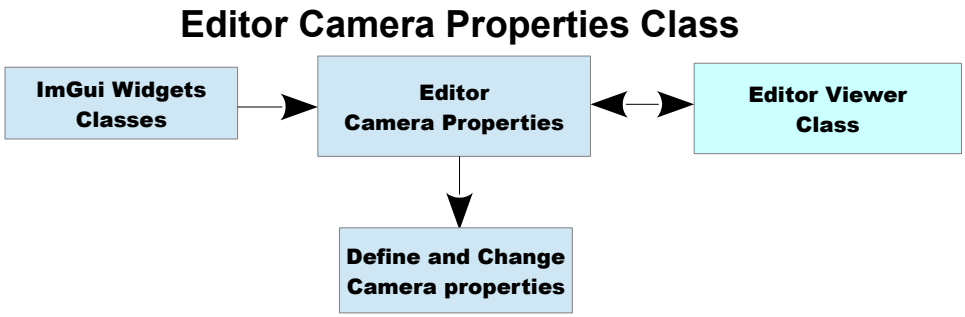

Fig 29 Schematic of the Editor Camera Properties Class Design

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

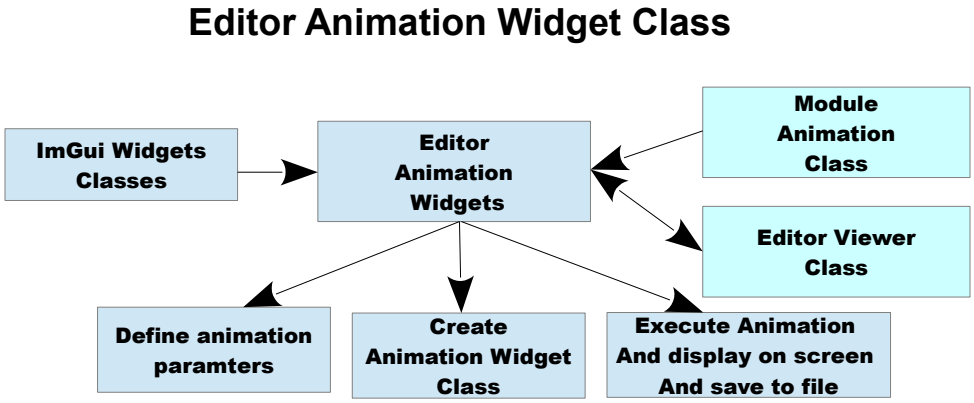

Fig 30 Schematic of the Editor Animation Widget Class Design

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