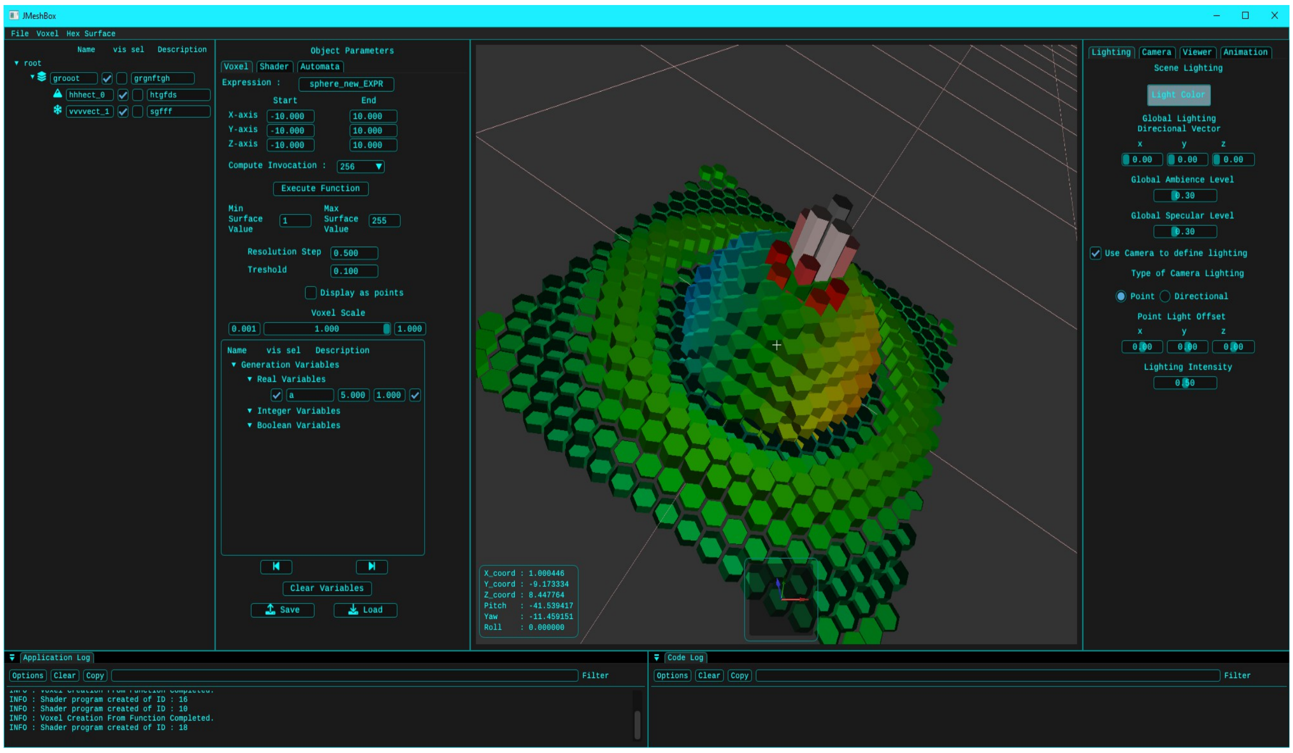
Virtual Worlds Application Documentation (draft)

User interface



```
Name      vis sel  Description
▾ root
  ▾ grooot    ✓       grgnftgh
      hhhect_0  ✓     htgfds
      vvvvect_1 ✓     sgfff
```

```
Expression :  sphere_new_EXPR
          Start      End
X-axis   -10.000    10.000
Y-axis   -10.000    10.000
Z-axis   -10.000    10.000
Compute Invocation :  256
Min Surface Value  1     Max Surface Value  255
Resolution Step  0.500
Treshold         0.100
☐ Display as points
Voxel Scale  0.001  1.000  1.000

Name    vis sel   Description
▾ Generation Variables
  ▾ Real Variables
      ✓ a  5.000  1.000 ✓
  ▾ Integer Variables
  ▾ Boolean Variables
```

```
X_coord : 1.000446
Y_coord : -9.173334
Z_coord : 8.447764
Pitch   : -41.539417
Yaw     : -11.459151
Roll    : 0.000000
```

```
Global Lighting Direcional Vector
  x 0.00   y 0.00   z 0.00
Global Ambience Level  0.30
Global Specular Level  0.30
✓ Use Camera to define lighting
Type of Camera Lighting
  [✓ Point]  Directional
Point Light Offset
  x 0.00   y 0.00   z 0.00
Lighting Intensity  0.50
```

```
INFO : Voxel Creation From Function Completed.
INFO : Shader program created of ID : 16
INFO : Shader program created of ID : 10
INFO : Voxel Creation From Function Completed.
INFO : Shader program created of ID : 18
```

Fig 1 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 using OpenGL. The procedural generation of 3D objects is done through using an OpenGL compute shader program to perform parallel processing via utilising the capabilities of the GPU.

The basic interface of the Virtual Worlds thus looks on first impressions like most other 3D applications with a central viewer 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 where all the parameters of each object type is to be displayed and changes made. The type of parameters and display of widgets in this panel depends on which module data object type is selected via the outliner . The creation, loading and saving of parameter data, and processing or generation of data for the particular module data object is done within this panel.

At the bottom is a logging panel to display messages such as failed/succeeded 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.

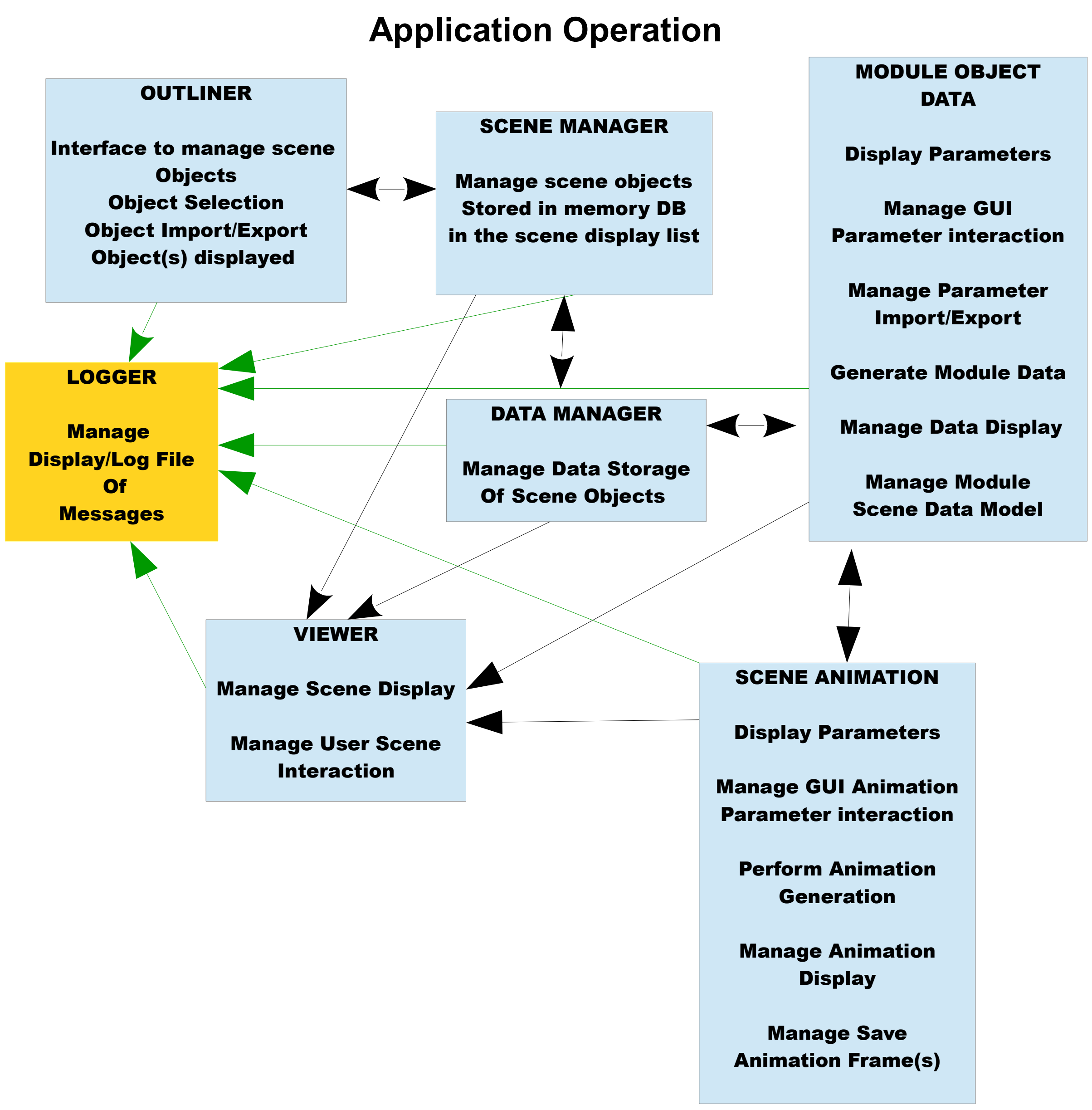

Application Operation
OUTLINER
Interface to manage scene Objects
Object Selection
Object Import/Export
Object(s) displayed
SCENE MANAGER
Manage scene objects Stored in memory DB in the scene display list
MODULE OBJECT DATA
Display Parameters
Manage GUI Parameter interaction
Manage Parameter Import/Export
Generate Module Data
Manage Data Display
Manage Module Scene Data Model
LOGGER
Manage Display/Log File Of Messages
DATA MANAGER
Manage Data Storage Of Scene Objects
VIEWER
Manage Scene Display
Manage User Scene Interaction
SCENE ANIMATION
Display Parameters
Manage GUI Animation Parameter interaction
Perform Animation Generation
Manage Animation Display
Manage Save Animation Frame(s)

Fig 02 Schematic of Virtual Worlds overall flow of operation

The Virtual worlds application operation to create and display a 3D scene is done like most 3D applications. As of writing this, the application operation is of 5 basic sections

1: Through the Outliner, the user creates or selects an existing module data object type.
2: In the Module Object parameter panel is a selection of parameter class widgets associated with the selected or created module data object type that the user has highlighted in the outliner in 1:. The user interacts with this panel to define parameters that create/modify the 3D data to be displayed in the Virtual World scene.
3: Generate the 3D data or change the appearance of the module data object.
4: In the scene animation parameters panel, the user can define parameters to perform animation of module data objects that have been set up to perform animations in the scene.
5: The viewer display panel is a series of GUI tab widgets that define global lighting options for the rendered scene and camera movement and orientation in the scene

In 4: the animation class, there is data that tasks that the animation must get and use in each module data object type that is animated, and some module data object type parameters that are modified while the animation process is being executed. Thus the double arrow in Fig 02 to indicate this interaction.

In 2:, 3: and 4: above, the application viewer receives the data and performs the actions and task to render the scene.

In 5: the viewer render and camera parameters options are updated interactively in real time and no generation of data is performed.

In 1: to 4: all application operations, the user can at any time import/export parameter data and generated data   of any of the module data objects in the scene.

Outliner Operation

Create Outliner Menu System →

OUTLINER

Manage Scene Objects

Scene Object Selection

Manage Scene Import/Export

Manage Scene Objects

Create/Delete Scene Groups

Create/Delete Module Objects

Create Scene Group

1: Get unique ID number for Group
2: Create new group in scene data model
3: Create Outliner tree item for a new group
Add to existing outliner tree and display on Screen.

Create Module Object

1: If currently selected Outliner tree item is a group then do
2: Get unique ID number for new module object
3: Create new module object in scene data model
4: Assign new module object to current selected Scene group
5: Create Outliner tree item for a new module object
Add to existing outliner tree and display on Screen.

Delete Scene Group

1: If currently selected Outliner tree item is a group then do
2: Get unique ID number for Group
3: Get list of all Scene data objects in the group
4: Delete all Scene data objects from Scene data model identified in step 3
5: Delete group from Scene data model
6: Delete Outliner tree items
7: Update Viewer Display.

Delete Scene Group Module Object

1: If currently selected Outliner tree item is a Module Object item do
2: Get unique ID number for Module Object
3: Get unique ID number for Module Object parent group
4: Delete selected Scene data object from Scene data model identified in step 2
5: Delete selected Scene data object from group Identified in step 3.
6: Delete Outliner tree items
7: Update Viewer Display.

Scene Object Selection

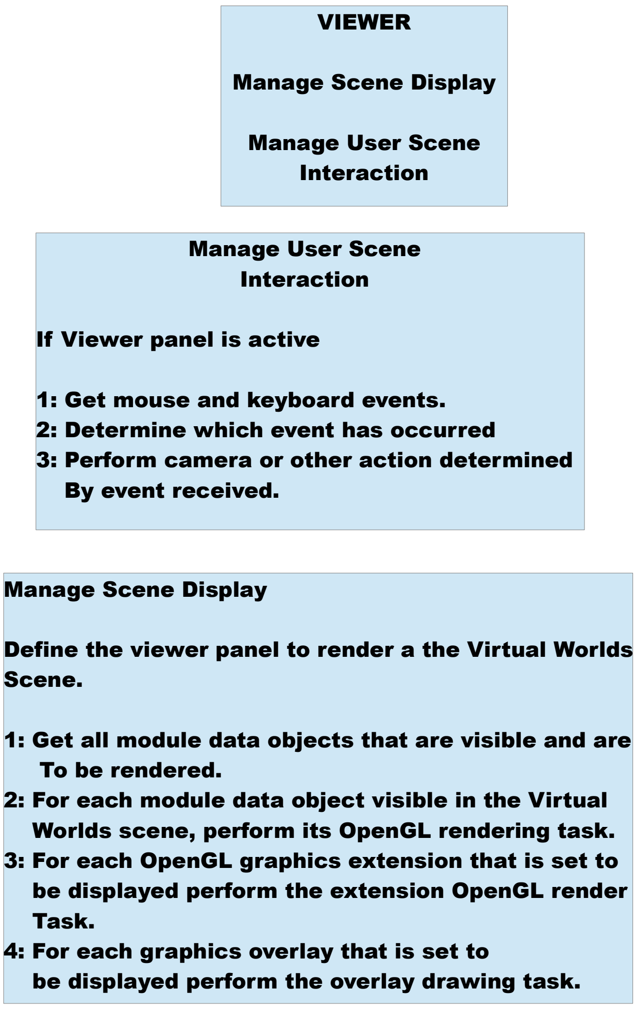
1: Get Outliner tree item selected and designate as currently selected
2: Determine what type of Module data type is selected
3: Find Module data parameters in the Scene data model and define as currently selected module data object
4: Display currently selected module data Object data parameters in the parameters panel

```
Manage Scene Export

1: Get currently selected outliner item

If currently selected outliner item is a
      module data item then

A1: get the selected module data from
      the scene data model.
A2: Perform a data parameter export for the
      Module data type of the currently selected
      Module data object.


If currently selected outliner item is a
      group item then

B1: get the group data from the scene data
      model.
B2: Perform a group export for the currently
      Selected scene group.

If menu selection is to export the Virtual
Worlds Scene

C1 : Perform a Virtual Worlds scene export.
```

```
Manage Scene Import

1: Get currently selected outliner item

If currently selected outliner item is a scene
Group item then

A1: From the menu, select what type of Module
      data item is to be imported.
A2: Select the file with parameter data to import.
A3: Create a new Module data object for the
       Currently selected scene group.
A4: Update the Scene data model with the
      Prameter data being imported.
A5: Update the outliner item name and description
      With that given in the imported file.


If currently selected outliner item is a
      group item then

B1: Select the file with parameter data to import.
B2: Create a new scene group
B3: Update the outliner group item name and
      Description with that given in the imported file.
B4: For each module data type in the imported file
      Perform steps A3-A5.

If menu selection is to Import a Virtual
Worlds Scene

C1: Select the file with scene parameter data
       to import.
B4: For each group in the imported file perform
       steps B2-B4.
```

Fig 03 Schematic outliner operation tasks

Viewer Operation



VIEWER

Manage Scene Display

Manage User Scene Interaction

Manage User Scene Interaction

If Viewer panel is active

1: Get mouse and keyboard events.
2: Determine which event has occurred
3: Perform camera or other action determined By event received.

Manage Scene Display

Define the viewer panel to render a the Virtual Worlds Scene.

1: Get all module data objects that are visible and are To be rendered.
2: For each module data object visible in the Virtual Worlds scene, perform its OpenGL rendering task.
3: For each OpenGL graphics extension that is set to be displayed perform the extension OpenGL render Task.
4: For each graphics overlay that is set to be displayed perform the overlay drawing task.

Fig 03 Schematic viewer operation tasks

Viewer Display Operation

VIEWER DISPLAY

Display Parameters

**Manage GUI Display
Parameter interaction**

**Apply Display Changes
And Display
Extensions Overlays**

Display Parameters

1: Define the parameter display widgets for each Virtual Worlds OpenGL display parameter category.

2: Create a parent tab widget and assign each OpenGL display parameter category to be displayed as tabs in This tab widget.

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

**Manage GUI Display
Parameter interaction**

1: Get the GUI QWidget signal when the user makes a change To any of the viewer display widget inputs and execute the Function that the QWidget signal is assigned to.

2: The Function defined in 1: updates the widget GUI and The uniform values of the OpenGL glsl program(s) that are Used by the application viewer

**Apply Display Changes
And Display
Extensions Overlays**

1: When the user makes a change to any of the viewer display widget inputs update the uniform values of the OpenGL glsl program(s) that are used by the application viewer.

3: Update the OpenGL viewer to display the changes made in 2:

Fig 04 Schematic viewer display operation tasks

Module Object Data Operation

MODULE OBJECT DATA

Display Parameters

Manage GUI Parameter interaction

Manage Parameter Import/Export

Generate Module Data

Manage Data Display

Manage Module Scene Data Model

Display Parameters

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

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

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

Manage GUI Display Parameter interaction

1: Get the GUI QWidget signal when the user makes a change To any of the module parameter widget inputs and execute the Function that the QWidget signal is assigned to.

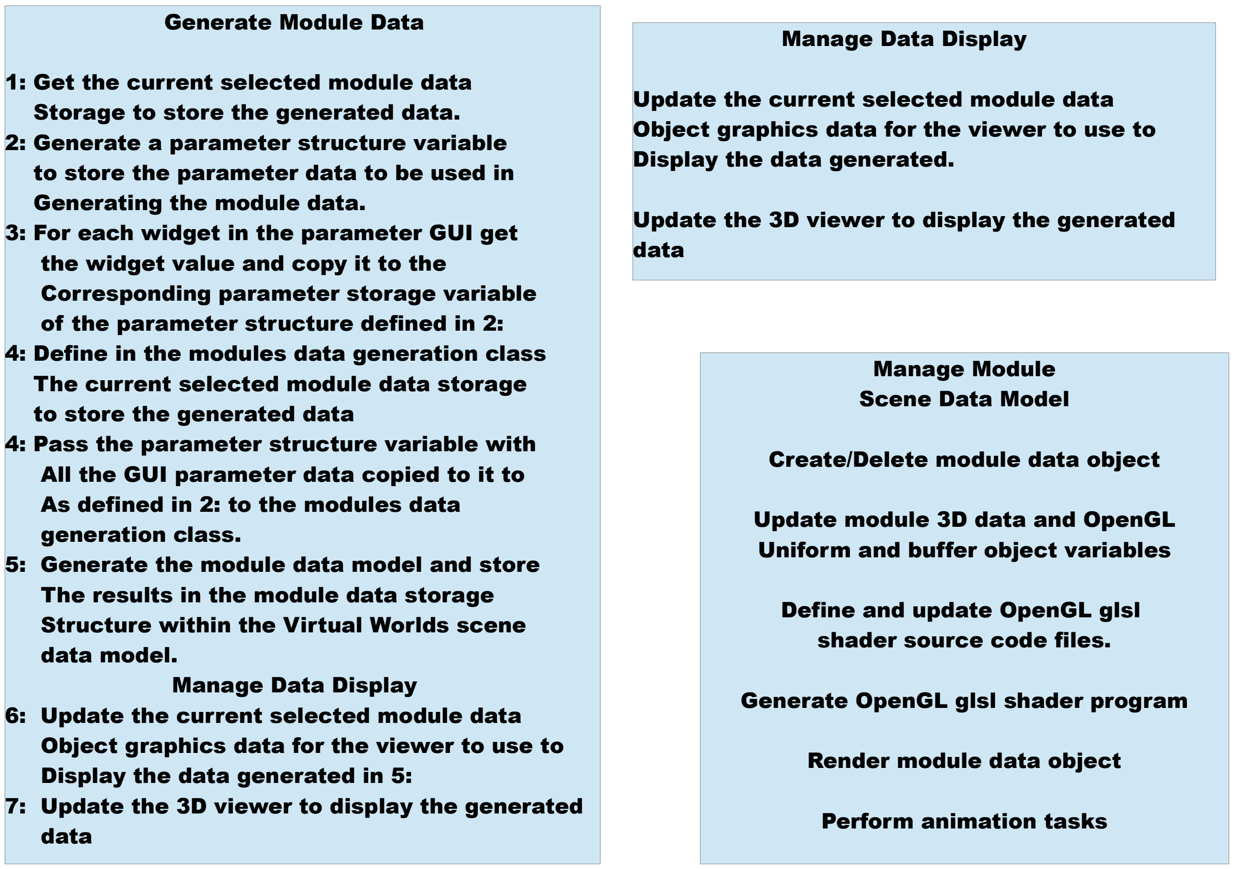
2 : Perform Manage Data Display tasks

Manage Parameter Import

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

Manage Parameter Export

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



Generate Module Data

1: Get the current selected module data Storage to store the generated data.
2: Generate a parameter structure variable to store the parameter data to be used in Generating the module data.
3: For each widget in the parameter GUI get the widget value and copy it to the Corresponding parameter storage variable of the parameter structure defined in 2:
4: Define in the modules data generation class The current selected module data storage to store the generated data
4: Pass the parameter structure variable with All the GUI parameter data copied to it to As defined in 2: to the modules data generation class.
5: Generate the module data model and store The results in the module data storage Structure within the Virtual Worlds scene data model.

Manage Data Display

6: Update the current selected module data Object graphics data for the viewer to use to Display the data generated in 5:
7: Update the 3D viewer to display the generated data

Manage Data Display

Update the current selected module data Object graphics data for the viewer to use to Display the data generated.

Update the 3D viewer to display the generated data

Manage Module Scene Data Model

Create/Delete module data object

Update module 3D data and OpenGL Uniform and buffer object variables

Define and update OpenGL glsl shader source code files.

Generate OpenGL glsl shader program

Render module data object

Perform animation tasks

Fig 03 Schematic module data type operation tasks

The

Scene Animation Operation

SCENE ANIMATION

Display Parameters

Manage GUI Animation Parameter interaction

Perform Animation Generation

Manage Animation Display

Manage Save Animation Frame(s)

Display Parameters

1: Define the parameter display widgets for animation GUI class.

Manage GUI Animation Parameter interaction

1: Get the GUI QWidget signal when the user makes a change To any of the animation parameter widget inputs and execute the Function that the QWidget signal is assigned to.

Perform Animation Generation

1: Define objects to execute for each entity data type
2: Get directory to save animation frame files to
3: Define_animation_parameters();
4: Define_initial_frame_paramters();
5: Save initial animation frame mesh for each object that is to be animated in the scene
6: For each animation frame generate point cloud data for each object data type that is to be animated in the scene save the animation frame mesh for each object that is to be animated in the scene to disk ie Manage Save Animation Frame(s)
7: Manage Animation Display
8: Restore the scene to its initial state

Manage Animation Display

For each animation frame
Perform the Manage Sene Display task

Manage Save Animation Frame(s)

1: Get the module data object(s) that has its animation Frame data generated and is to be exported to a .ply file For current animation frame.
2: Get the directory and file animation prefix to save each Generated module data object to.
3: Get the type of data to export eg point or surface etc.
4 : Pass the generated data in 1, the parameter data in 2 and The type of data to export in 3: to the appropriate module export data class and perform the module data Objects export to file function.

Fig 03 Schematic animation operation tasks

Each

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

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

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

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

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

B: Export parameter data to formatted text based file.

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

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

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

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

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

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

```
            PARAMETER_DATA_BLOCK_BEGIN
               PARAMETER_SET1_DATA_BLOCK_BEGIN
                      PARAMETER_SET1_SUBSET1_BEGIN
                                more embedding
                      PARAMETER_SET1_SUBSET1_END
                      PARAMETER_SET1_SUBSET2_BEGIN
                                more embedding
                      PARAMETER_SET1_SUBSET2_END
               PARAMETER_SET1_DATA_BLOCK_END     :
            PARAMETER_SET2_DATA_BLOCK_BEGIN
                      more embedding
            PARAMETER_SET2_DATA_BLOCK_END
            PARAMETER_DATA_BLOCK_END
```
.
Which gives a flexibility of design that each embedded data block can be blocks of data of different kinds of related data such that data can be exported in a hierarchy of data structures such that data can be grouped together in a form such as

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

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

```
 MODULE_DATA_TYPE1 data export
 MODULE_DATA_TYPE2 data export

GROUP_DATA_TYPE1 export{
    MODULE_DATA_TYPE1 data export
    MODULE_DATA_TYPE2 data export
}

GROUP_DATA_TYPE2 export{
          <GROUP_DATA_TYPE2 export functions>
}

SCENE_DATA_BLOCK export{
     GROUP_DATA_TYPE1 export
     GROUP_DATA_TYPE2 export
}
```

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

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

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

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

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

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

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

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

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

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

```
 MODULE_DATA_TYPE1 data import
 MODULE_DATA_TYPE2 data import

GROUP_DATA_TYPE1 import{
    MODULE_DATA_TYPE1 data import
    MODULE_DATA_TYPE2 data import
}

GROUP_DATA_TYPE2 import{
            <GROUP_DATA_TYPE2 import functions>
}

SCENE_DATA_BLOCK import{
      GROUP_DATA_TYPE1 import
      GROUP_DATA_TYPE2 import
}
```