Shader Constructor Emulator
SCON

Introduction

In developing the Virtual World application to display 3D graphics to the computer screen, the OpenGL API was required to be used. As a design feature of the virtual worlds application, there was the necessity to have a major part of the 3D graphics rendering operations to be performed utilising the OpenGL geometry shader to process point cloud data and use that data to render 3D surfaces by creating uniform triangle geometry data.

Other tasks such interactive colour and graphical manipulation was to be performed utilising the graphics card parallel processing hardware through use of coding and compiling OpenGL shader programs. Unlike coding and compiling software on a CPU, the debugging of OpenGL code beyond the syntax compile errors is not straightforward or easy to do with interacting applications as no inspection of variables or data was possible. Many hours was wasted to find out the source of various rendering problems and correct them. Even use of third party OpenGL debugging software such as renderdoc or Nsight Graphics was limited. The limitations to only observe the state of a single frame of rendering and not displaying the shader code variables values proved to be in many cases, not helpful at all.

So, after some time of deliberation and investigation it was desired to seek out if an openGL shader emulator software existed to assist in creating openGL shader code and debug if similar to that of normal CPU centred software. Non could be found. So it was decided to develop my own shader emulation software coding package and call it Shader Constructor, or SCON.

SCON is an emulator of all aspects of openGL, designed specifically emulate the OpenGL pipeline in CPU code from the generation of the graphics data on the CPU and sending it to the GPU, and the pipeline of shader code utilitising vertex, geometry and pixel or fragment shaders. Because this OpenGL pipeline emulator is of CPU coding, the ability to enable inspection of all the data and variables of the OpenGL shader pipeline would be easy to inspect and debug.

This is the purpose of SCON. To construct emulated GLSL code that has been tested and debugged, and then translate this emulated GLSL code into true GLSL code and incorporated into a GLSL shader program.

The OpenGL GLSL Shader Emulation Constructor SCON

SCON can in no way be considered as an application. This is because It does not stand as an application that a user can just open and start to use to import data, process that data and get a result. SCON is a C++ programming tool that requires the user to have coding skills and knowledge to understand openGL and GLSL coding concepts so as to be able to compose and debug code in a C++ IDE. SCON was developed in and uses the community version of Visual Studio to compile and run code.

SCON can be considered as a series of C++ coding tools that emulates the openGL GLSL graphics pipeline to display graphics, and its intended purpose is to be used to develop and debug GLSL programs using standard C++ debugging methods. However, the openGL GLSL emulation can be used in principle to be incorporated into an application to display graphics, but will be limited by the CPU processing power.

The workflow method and steps needed to follow to use SCON is in general simplified terms to be

1: Generate the graphical data that is to be migrated to the GPU and a GLSL shader program to use to display graphics.

2: Translate the data generated in step 1 into a format to emulate the openGL buffer data that is to be migrated to the GPU and used by a GLSL shader program.

3: Define the emulation of openGL uniform and attribute GLSL data variables that are to be used in a GLSL shader program.

4: Create the GLSL emulation code of the GLSL shader program that is to be coded and compiled.

5: Compile and run the C++ emulation code to display the graphical output of the GLSL program.

6: Insert debugging code where needed and inspect the variables, emulated uniform and graphical data to debug and construct the GLSL shader program.

7: Repeat steps 4-6 until the final desired result is achieved

8: Once a final emulated GLSL program code is achieved, translate the emulated GLSL to a true GLSL code within a GLSL shader program that is to be compiled and run in an openGL application.

This workflow should be sufficient for most emulation to true GLSL shader programming.

Application Directory Structure

The directory structure of the Virtual Worlds project as currently defined in August 2025 consists as follows.

```
Bin             (compilation binary and run time application data files)
Documentation   (Application documentation )
FrameWork       (Application Framework code that is designed to be shared between  applications)
libs            (Required third party dependency library files)
Source          (Application  Source code)
thirdparty      (Source code of dependent third party support applications)

ShaderCON.sln
ShaderCON.vcxproj
ShaderCON.vcxproj.filters
ShaderCON.vcxproj.user
```

Listing 01 : Virtual Worlds root directory structure

Sub directory structure

```
Bin → Intermediate      (compiler intermediate code)
      x64 →Debug        (Application debug executable and data files)
           Release      (Application release executable and data files)

FrameWork → AFW         (Application framework code that can be used in any application)
            NodeFW      (Framework code that can be used to create a general node editor) (not used)
            OGLFW       (Framework code that can be used for general OpenGL graphics applications)
            VWFW        (Virtual Worlds Framework used only in the Virtual Worlds environment)

libs → Debug            (Dependency third party libraries used for debug compilation mode)
       Release          (Dependency third party libraries used for release compilation mode)

Source →SCON                    (Shader constructor application source files)
         shaderCON_main.cpp     (C++ Shader constructor application application entry files)

thirdparty → assimp
             glad
             glew
             glfw
             glm
             ImGui
             spdlog
             stb_image
             tiny_obj_loader
             tinyFileDialog
             tinyply
             tinyxml
             utf8
             vma
```

Listing 02 : Virtual Worlds sub directory structure

Dependant third party version of software is given in the third party source documentation.
For all intents and purposes, the third party directory can be considered as part of the applications framework and may be moved at some later stage to the FrameWork directory.

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 ..cpp 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.

Visual Studio Code Compilation

The visual studio C++ code was compiled as ISO C++20 Standard using the Visual studio 2022(v143) toolset and Windows SDK 10.0.22621.0

Advised not to use earlier then C++20 Standard to maintain compatability.

Code Documentation

Documentation and explanation of source code exists within the source code itself as much as possible and naming of variables and functions, classes etc is kept to a standard of readability as much as possible. However, this is often not enough to explain the many nuances and work runarounds that have been needed to be implemented to get things to work as intended and to comply with compiler demands.

Thus over time as code is finalised, documentation of code may be performed in a more formal manner of being documented in a pdf file to explain and visualise what the function of the code is for and does. This documentation is not just for any member of the public, but is also for to remind the author of what was coded and the relationships and links between all the classes, functions, structures etc. that may have been forgotten over a long period of time.

Coding Design Principles

Base classes

The SCON opanGL emulation application adopts widely within its code the principle of object orientated programming as much as necessary to obtain a consistent standard of operation and maintenance of code such that once it is understood and is operational in one part of the application, the same method can be applied wherever possible in other sections of the application.

The major C++ adoption of an object orientated programming approach is to widely use a base classes that are inherited by higher ranking classes. These Base classes have common features that are used in all inherited class entities of differing types of data or coding methods within a common function used to achieve a the same end result.

These functions are defined in the base class as a virtual function with a common name and purpose of process, but have the core code different for each class that inherits this base class.

One such function for a 3D geometric entity may be to render to screen a 3D geometric entity. One 3D entity may be a point cloud as used widely within the Virtual Worlds application to draw a point cloud, and another

a 3D mesh that renders a solid 3D surface defined as triangles. The code to render such 3D geometry is different, however the function name Draw to call to perform the rendering operation is the same.

Thus a C++ base class with a virtual function called Draw can be defined that is then inherited by a geometric class for each 3D geometry entity type, and the code defined to perform the render can be unique to each 3D geometry entity class.. The great power of C++ is that to render any 3D geometric object, a single reference is requested of the base class from which each 3D geometry is derived from, not the many 3D geometry classes that may exist from which this base class was derived from. **Fig CDP01** gives a simplified illustration of this approach.

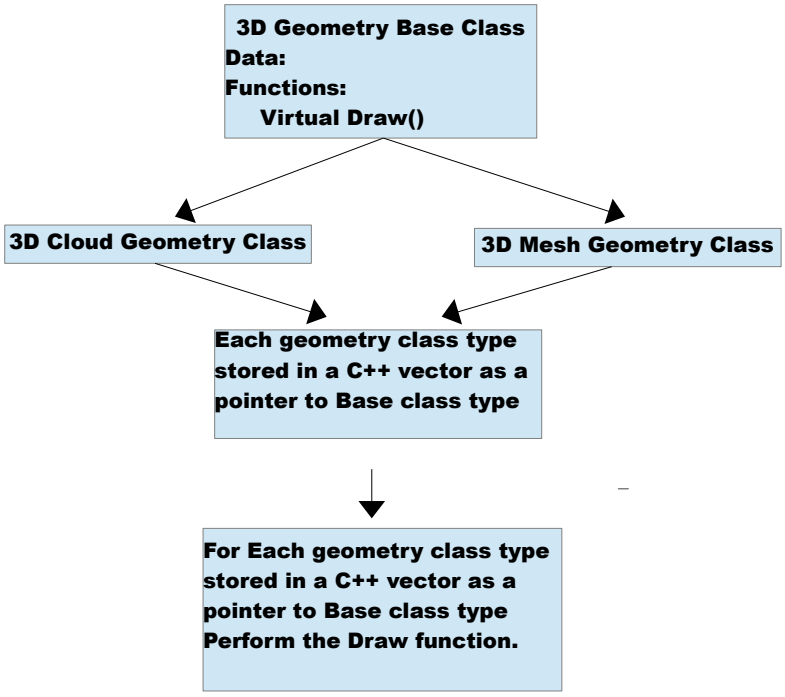


Fig CDP01

The C++ pseudo code would look something like for

Base Class

```
class 3D_geometry_base_class{
public:
        virtual void Draw();
}
```

Derived Classes

```
class point_cloud_class : public 3D_geometry_base_class{
public:
        virtual void Draw() override{
           /* c++ point cloud render code */
        }
}

class mesh_class : public 3D_geometry_base_class{
public:
        virtual void Draw() override{
           /* c++ mesh render code */
        }
}
```

3D objects storage

```
point_cloud_class *point_cloud_object;
mesh_class        *mesh_object;

std::vector< 3D_geometry_base_class*> 3D_objects;

3D_objects.push_back(point_cloud_object);
3D_objects.push_back(mesh_object);
```

Render 3D objects

```
for each  3D_objects {
        3D_objects→draw();
}
```

This singular powerful c++ feature is exploited through out the virtual worlds application not only as mentioned for 3D objects, but also for the nodes for use in the node editor and the various user interface panels that the user interacts with.

Global Variables (Singleton Classes)

For the Virtual Worlds application to function as desired, a necessity to have a form of transferring data that could not be performed by normal function parameter definitions was required. A form of a set of global variables that was common and needed to be accessed or referenced across the entire application was required. The solution found was to create a singleton class that is visible and accessible through out the entire application. This singleton class stores global variables and the functions to set and get these global variables.

Two such singleton classes exist and must be defined as an included file within the main application .cpp file before all other include file declarations that may use them otherwise the global variable data that exists within them are not set correctly, are cannot be accessed.

These singleton classes called afw_globalc (application framework) and oglfw_globalc (opengl framework) are very widely used to store and read such global data as the current selected 3D object to display or edit that is not possible to convey through the application by other means.

For more information on how singleton class are defined and how they work, an online search will give adequate information on this topic better than can be described here.

01 : Application operation

Application base Class afw_app_base_class

The execution and running of the application is implemented and managed using an abstract application base class afw_app_base_class.

afw_app_base_class is designed to define and handle the execution and function of any user application that incorporates as part of it function a user input and interaction of use of a keyboard and mouse. Within this class is a virtual function called setup is called to define the application setup routines and code and then a another virtual function run_loop is called that defines the application run time execution loop. Once the execution run loop is exited due to a user interaction, a final application virtual close function is called, from which any clean up operations are to be performed.

afw_app_base_class could also act as a base class from which multiple applications could be defined and run from a single application entry point.

The afw_app_base_class is inherited by the main application class from which the developer specifies the application code for each of the virtual functions to handle mouse, keyboard interactions and the application startup, run loop and close functions.

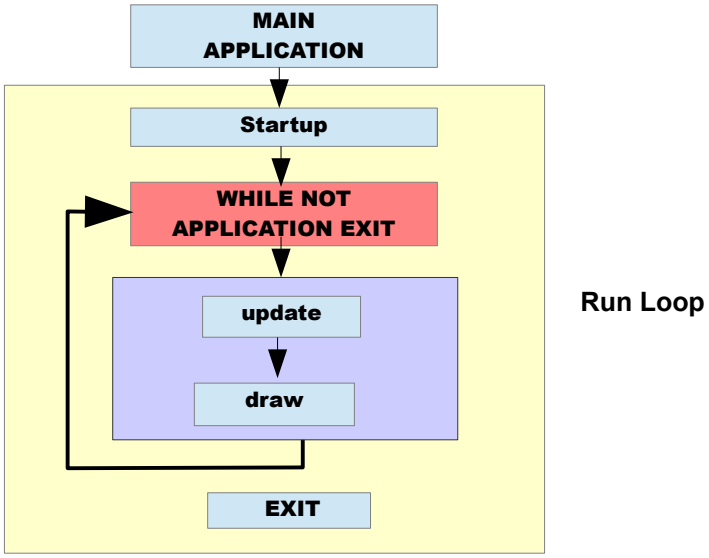


Fig 01.01 Schematic of Application base Class afw_app_base_class application run operation

The application main() function entry point can be as simple as

```
int main() {
    app_test_class* app = new app_test_class;
    app->run();
    exit(0);
}
```

02 : Loop Cycle

The SCON toolkit tasks to be performed per loop cycle as illustrated in **Fig 01.01** can be summarised for each loop cycle function update, draw and exit in table 02.01 as

update()

draw()

ImGui setup
Display log panel
Display parameter panel(s)
Display openGL GLSL pipeline emulation data panels

Render scene in view port
Draw view port overlays.
ImGui exit

exit()

table 02.01

The SCON application draw routine displays the emulation data panels as part of the draw routine as well as rendering the graphics to the screen.

03 : Camera Class

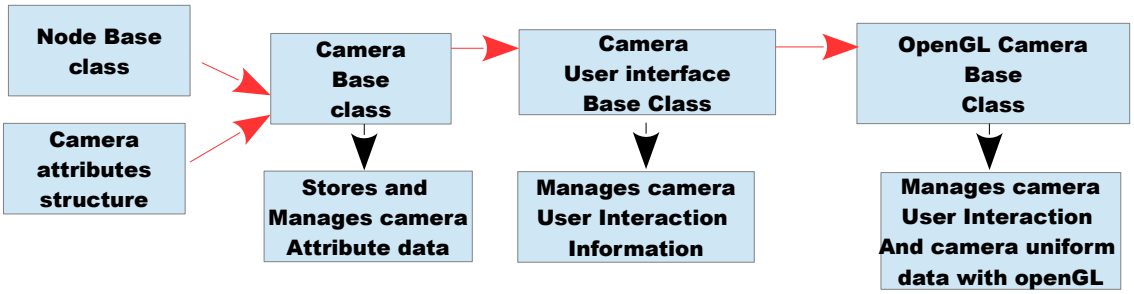


Fig 03.01 Schematic of the Camera Class design. Inherited classes have red arrows

To view a 3D rendered scene and the objects within it, as per usual in 3D applications, a virtual camera has to be defined and implemented. SCON has such a virtual camera defined not as a single C++ class, but is a derived class built up by inheriting the base elements from base classes that are required for any application to display 3D graphics.

The principle of utilising base classes is explained in section coding design principles base classes above.

The base class afw_camera_base_class defines the basic data and functions all cameras require to function with any graphics application.

A second base class afw_camera_ui_base_class inherits the afw_camera_base_class and has additional functions and data variables to enable a camera for user interaction with mouse and keyboard.

A third base class oglfw_camera_base_class inherits afw_camera_ui_base_class and has additional code to set up and process user interactions with a virtual camera that is to display graphics in an OpenGL graphics environment. oglfw_camera_base_class Thus has functionality to convey a standard set of variables to an OpenGL shader program of a camera shader variables.

The oglfw_camera_base_class is sufficient to be used as a stand alone class for all user openGL application needs and is the camera class used in the Virtual Worlds application.

The purpose of this design is so as to be able to create applications that can use a virtual camera to display graphics that does not need any user interaction (camera base class)

The camera base class source code exists in the directory

FrameWork→VWFW→Objects

of file names

afw_camera_base.h
afw_camera_ui_base.h

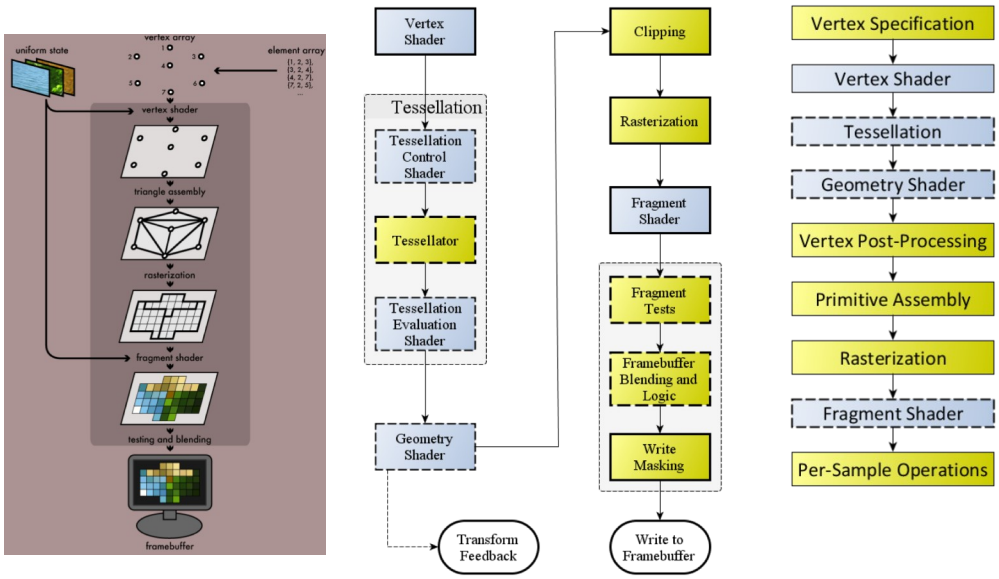
The openGL base class source code exists in the directory

FrameWork→OGLFW→Objects

of file names

oglfw_camera_base.h

04 : SCON OpenGL Pipeline Emulation



(a) (b) (c)

Fig 04.01

Performing a general search on the internet, one can gain an overall and specific understanding of the standard OpenGL pipeline as set out by the Kronos Group that all applications must follow. Such a pipeline is illustrated and summarised in **Fig 04.01(a),(b) (c)**.

The OpenGL pipeline displayed in Fig 04.01 does not include the initial steps required to generate and prepare the vertex data and additional OpenGL pipeline uniform variables to be migrated into this OpenGL pipeline. These additional steps are illustrated and summarised in **Fig 04.02**.

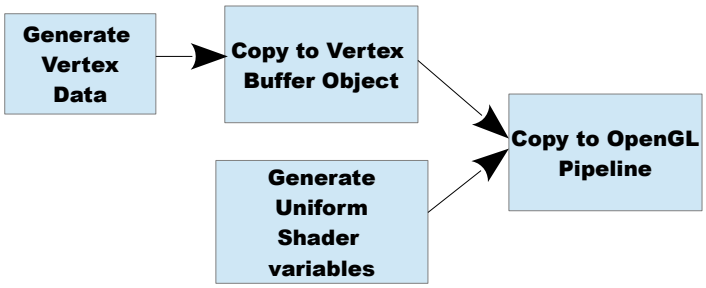


Fig 04.02

The steps illustrated in **Fig 04.02** are performed by the CPU, whereas the OpenGL pipeline illustrated in **Fig 04.01** is performed by the GPU. Thus the preparation and copying of vertex and uniform related data is defined by coding and compiling for the CPU, while the OpenGL pipeline is coded and compiled in the GLSL language to be performed by the GPU.

SCON must perform all of these steps to emulate the OpenGL pipeline. This emulation code by its very nature will not utilise the normal OpenGL code that is compiled and executed. As will be revealed, all of the coding for the OpenGL emulation is in C++ up until the last step of the emulation process where to display the end result, the actual true OpenGL pipeline has to be used.

SCON as of the time of this documentation only performs emulation of the vertex and geometry shaders. Tessellation is not performed as this is not in the interests or need of the Virtual Worlds application that the SCON toolkit is designed to cater for. The fragment shader is not emulated as this shader deals with the editing of the pixels of resultant raterised image before rendering the image to screen.

Thus SCON is not a full 100% emulation of the entire OpenGL pipeline. It is an emulation of the vertex and geometry shader functionality of the the OpenGL pipeline.

04:01 SCON openGL pipeline emulation loop

If one looks at some simple GLSL code and the OpenGL pipeline given by **Fig 04.01**, it is noticed that the GLSL code for any of the OpenGL pipeline shaders do not have any code that resembles a looping operation to process the vertex data one at a time as it would need to do for a normal CPU set of instructions. This is because of the parallel processing that GPU hardware performs. This parallel processing is not performed in

the SCON toolkit, but is emulated as a normal programming loop to where the vertex is processed one at a time. This is the main difference between the SCON CPU driven emulation of the OpenGL pipeline, and if the emulation requires vertex data that is of a large enough size, the efficiency and speed of the emulation will suffer and experience stutter.

Thus a first step of the OpenGL C++ emulation a loop to emulate this parallel processing given by a simple for loop. In pseudo code this can be expressed as

```
for(i=0; i < vertex_data_size;i++){
    Process ith vertex data element;        E-04.01.01
}
```

Every shader program be it vertex, geometry or fragment etc has a main function in its code structure from which is defined the instructions to be performed to process the functionality of that shader program. Process ith vertex data element of **Psuedo code 04.01** is in effect the main function of the shader program being emulated**.**

04:02 SCON openGL pipeline emulation : GLSL internal vertex data structure and storage

Looking at the OpenGL GLSL shading language defined by the Kronos Group, all shader programs have the vertex data that is processed tied to an internal predetermined variable structure called gl_in for input vertex data, and gl_out for output data.

The structure of gl_in and gl_out variable is of the form

```
gl_in {
    vec4 gl_position;
    float  gl_pointsize;
    float gl_ClipDistance[];
}

gl_out {
    vec4 gl_position;
    float  gl_pointsize;
    float gl_ClipDistance[];
}
```

gl_in and gl_out are identical in structure and can have a same C++ structure defined to emulate these OpenGL GLSL vertex structure types. This C++ structure is given the same name as defined in the Kronos OpenGL documentation as

```
gl_PerVertex {
    vec4 gl_position;
    float  gl_pointsize;                    E-04.02.01
    float gl_ClipDistance[];
}
```

Because gl_ClipDistance[] is never used in the emulation, it is commented out in SCON.

To emulate the one dimensional arrays that OpenGL shader programs utilises as input and output, the standard C++ vector feature is used as a basis for storage of vertex data. Thus the data storage for vertex data is if the form

```
std::vector<gl_PerVertex>                   E-04.02.02
```

E-04.02.02 is the basis data type that represents in the SCON C++ GLSL shader emulation the GLSL internal data structure gl_in and gl_out. That is, the C++ representation of GLSL gl_in and gl_out for all shaders is

```
std::vector<gl_PerVertex> gl_in  = {};      E-04.02.03
std::vector<gl_PerVertex> gl_out = {};      E-04.02.04
```

the gl_PerVertex structure and other GLSL emulation definitions is define in the SCON framework in the directory

FrameWork→SCONFW→Shader

within the file

sfw_shader_definitions.h

04:03 SCON openGL pipeline emulation : GLSL shader code emulation using C++ glm library

C++ does not have the native standard data types that GLSL has such as GLSL vectors vec4, vec3, and

some GLSL functions such as normalize and clamp etc. However there exists a C++ glm library that claims to emulate and perform the same functionality of the standard GLSL code. This C++ glm library forms the core of the SCON emulation toolkit, and as such the vec4 datatype expressed in gl_PerVertex in **E-04.02.01** becomes glm::vec4 by including the C++ glm library giving the C++ data structure of gl_PerVertex as

```
struct  gl_PerVertex {
   glm::vec4 gl_Position;
   float gl_PointSize;// to use this must enable  GL_PROGRAM_POINT_SIZE via      E-04.03.01
                      //  glEnable (GL_PROGRAM_POINT_SIZE);
    //float gl_ClipDistance[]; //uncomment if ever going to use
};
```

Therefore gl_PerVertex of **E-04.03.01** is the data structure that is used in the data structures of **E-04.02.03** and **E-04.02.04** of the SCON C++ shader emulation toolkit.

04:04 SCON openGL pipeline emulation : Vertex Shader layout structure data types

The importation of vertex data in GLSL code is defined by the definition of a vertex data layout format statement in the form of

layout (location = <loc_num>) in <input data type> <input data name>; **E-04.04.01**

eg

layout (location = 0) in vec4 position;

Similarly, the output of vertex data from a GLSL vertex shader to a geometry shader in the OpenGL pipeline in GLSL code is defined by the definition of a vertex data layout format statement in the form of

out <output data type> <output data name>; **E-04.04.02**

eg

out vec4 vertex_data_out;

Because the GLSL input and output statements **E-04.04.01** and **E-04.04.02** represents an entire one dimensional list or array of vertex attribute data, this data in a C++ emulation would be stored as C++ std::vector data type.

Eg, for layout (location = 0) in vec4 position, the C++ storage of layout data would be

std::vector<glm::vec4> position;

What this indicates is that to emulate the GLSL input and output statements **E-04.04.01** and **E-04.04.02**, information of the layout type, location, data type and data name need to be defined. This would suggest that the emulation of the GLSL layout statement in C++ would be of the form of a structure. A C++ base structure is indeed implemented in SCON to emulate the GLSL layout and is implemented as vlayout_base_struct defined as

```
struct vlayout_base_struct {
     vlayout_base_struct(int loc, shader_stream_type_enum st, int dt, std::string name) {
          initialise(loc, st, dt, name);
     }

     vlayout_base_struct(shader_stream_type_enum st, int dt, std::string name) {
          initialise(-1, st, dt, name);
     }

     vlayout_base_struct() {}

     virtual void initialise(int loc, shader_stream_type_enum st, int dt, std::string name) {
          location = loc;
          stream_type = st;
          layout_name = name;
          data_type = dt;
    //set_data_type(); // Cannot have a virtual function called within a constructor or function called by a constructor
     }

     int           location = -1;
    shader_stream_type_enum stream_type = shader_stream_type_enum::undefined;
    int           data_type = GLSL_DT_UNDEFINED;
    std::string layout_name = "";

    virtual size_t get_data_size() = 0;
    virtual void* get_data_element(size_t i) { return nullptr; }
    virtual void* get_data_vector() { return nullptr; }
```

```
};
```

E-04.04.03

The layout location number, data type and name are variables concurrent with the same meaning as for a GLSL layout.

The vertex attribute data that is for each vertex data point can be of any data type to represent any kind of data. Eg color or gradient. and even non 3D vertex data point. Thus the data storage data type for this layout base structure is unknown in advance. The data cannot be specified as a C++ type void * because to do so in this base class would make coding messy and difficult. This is because in any GLSL shader, multiple layouts can be specified for both input and output of vertex attribute data. Multiple input and output layouts suggests then in a C++ emulation of a shader can be defined as a list of C++ input and output layout structures that can be of the std::vector data type.

That is to have within a C++ emulation of a shader, a list of GLSL layouts would be in the form of

```
std::vector<vlayout_base_struct> input_layouts;
std::vector<vlayout_base_struct> output_layouts;
```

Utilising the magic of C++ to be able to process a base structure or class with a virtual function that can return a common void * data type, the solution was to enable such virtual functions within the vlayout_base_struct. These functions are

```
virtual void* get_data_vector() { return nullptr; }
```

to return the entire array of vertex attribute data, and

```
virtual void* get_data_element(size_t i) { return nullptr; }
```

to return one data element of array index i from the vertex attribute data.

One problem was that a data type void * does not have the ability to read the data and determine the number of vertex attribute data points is present, so a virtual function to get this data size needs to be present.

The inspiration for utilising this solution comes from OpenGL itself where such a method is used on the C++ side of OpenGL to define object vertex buffers to transfer vertex attribute data to the GPU hardware and compiled shader code.

The stream type variable indicates if the C++ emulation layout structure is for input into or output from the C++ emulated shader.

Two constructor functions are defined to initialise such a GLSL C++ emulated layout structure.

The C++ emulation of a GLSL layout of a specific data type is to define a C++ data structure of a name to reflect that data type that inherits the **E-04.04.03** vlayout_base_struct, and defines within it the storage of the data type as a C++ vector, and a data element variable of the same data type;

Such an example of a GLSL layout statement

```
layout (location = 0) in vec4 position;
```

emulated as a C++ structure is

```
struct vlayout_vec4 : public vlayout_base_struct {
	using vlayout_base_struct::vlayout_base_struct;// Bring to derived class the vlayout contructor

	glm::vec4 data_element; // Must be present to add data element to layout for output

	std::vector<glm::vec4> data = {};

	size_t get_data_size() override { return data.size(); }
	void* get_data_element(size_t i) override { return &data[i]; }
	void* get_data_vector() override { return &data; }

	void add_data_element() { // Must be present to add data element to layout for output
		data.push_back(data_element);
	}
};
```

E-04.04.04

By design, all C++ layout data type structure types have the same variable name for the vertex attribute data of data, and data_element for the data element of that data type. Following this this convention, when

creating a layout data structure of any data type, the above code for the structure vlayout_vec4 can be copied and pasted in full, then only the name of vlayout_vec4 needs to be changed and the data type of the variables data_element and data.

In a C++ shader emulation such a layout structure is defined for

layout (location = 0) in vec4 position;

is

vlayout_vec4 position = vlayout_vec4(0,shader_stream_type_enum::in,GLSL_DT_VEC4, "position");**E-04.04.04**

to create and initialise this layout variable ready to have vertex position data assigned to it;

The C++ emulation code for this and more layout data type structures exists in the SCON framework in the directory

FrameWork→SCONFW→Shader

within the file

sfw_shader_layout_data_types.h

04:05 SCON openGL pipeline emulation : C++ definitions of OpenGL primitive data types

OpenGL defines a list of vertices in a one dimensional array as a sequence of data primitives that represents the geometry of an entity to be rendered to screen.

The simplest such primitive is a point cloud of unrelated vertex data points. Others being in the form of a sequence of vertex points representing lines, triangles, line strips, triangle strips and others. A summary of these primitives taken from the Kronos documentation in a table is

TABLE 04:05:01

GS input	OpenGL primitives	TES parameter	vertex count
points	GL_POINTS	point_mode	1
lines	GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP	isolines	2
lines_adjacency	GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY	N/A	4
triangles	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN	triangles, quads	3
triangles_adjacency	GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY	N/A	6

Vertex data being input or output from a GLSL shader must be one of these primitive forms to be rendered and processed in the OpenGL GLSL pipeline.

SCON as time of this documentation only uses the OpenGL primitives of the input of points, lines, and triangles.

04:05 SCON openGL pipeline emulation : Geometry Shader layout and primitive data types

A geometry shader is, as stated in the Kronos documentation, designed to to accept a specific primitive data type as input, and to output a specific primitive data type. The accepted input primitive data type is defined in the GLSL shader code by the GLSL input layout statement

layout(<input primitive>) in; (**E-04.05.01**)

Where <input primitive> is one of the primitive data types defined by **TABLE 04:05:01**

and the general GLSL output layout statement is given by

layout(<output primitive>, max_vertices = X) out; (**E-04.05.02**)

Where <output primitive> is one of the primitive data types defined by **TABLE 04:05:01**. max_vertices gives the maximum number of vertices that is output from the geometry shader per primitive data type. Eg if the output primitive type is triangle_strip and max_vertices = 8, then only a maximum of 8 vertex data points can be defined per primitive triangle strip created. Any more defined than 8 will be ignored.

To perform the OpenGL pipeline process for the geometry shader, the vertex primitive data that is being imported or exported needs to be one of these primitive forms, and the emulation code needs to know which form the data is in so as to read and process the vertex data correctly. An enumeration data type can be assigned to do this.

Such an enumeration class definition is created as

```
enum class geom_layout_data_in_enum { undefined, points,lines, triangles};
```
E-04.05.03

where the geom in this enumeration class is a naming convention to indicate that this enumeration class definition is for a geometry shader.

Because the SCON emulation toolkit is designed to emulate geometry shaders extensively, the primitive out data (**E-04.05.02**) will in most cases be of a different number of vertices than the initial primitive in data(**E-04.05.01**). In many cases, the primitive out data will not be of the same form or primitive type as that of the primitive in data.

According to documentation provided by the Kronos Group concerning geometry shaders, the only form of data that can be input into a geometry shader is that defined for geom_layout_data_in_enum of points, lines and triangles, and that the output vertex data can only be one of points, lines,line strip, line loop, triangles, triangle strip, and triangle fan.

These output primitives are in essence, a list or array of a group of vertices that define these lines,line strip, line loop, triangles, triangle strip, and triangle fan types. These primitive data types that are a group of vertices is in effect a list or array of vertices.

In the SCON emulation, this data storage of OpenGL vertex data for a single primitive data type is defined as a C++ datatype by the C++ type definition statement

```
typedef std::vector<gl_PerVertex> g_vertex;
```
E-04.05.04

where g_vertex is chosen so as to indicate that this is a GLSL emulated representation of the storage of GLSL vertex data that is used by a C++ emulated geometry shader.

A data type to represent a list or array of any GLSL primitive data type can be defined as of being of the from

```
std::vector<g_vertex>;
```
E-04.05.05

In the SCON emulation, this data storage is defined as a C++ datatype by the C++ type definition statement

```
typedef std::vector<g_vertex> g_primitive;
```
E-04.05.06

where g_primitive is chosen so as to indicate that this is a GLSL emulated representation of the storage of GLSL primitive data that is defined to be used by an emulated geometry shader.

To perform the OpenGL pipeline process for a geometry shader, the vertex data that is being exported needs to be one of these primitive forms, and the emulation code needs to know which form the data it is in so as to process the vertex data correctly. An enumeration datatype can be assigned to do this.

Such an enumeration class definition is created as

```
enum class geom_layout_data_out_enum { undefined, points,line_strip, triangle_strip};
```
E-04.05.07

As line loop and triangle fan are not used by SCON as of this time, and are excluded from this enumeration class until needed.

To conclude, emulation of the GLSL statement for the importation of vertex data into a GLSL shader given by **E-04.04.01** can be defined within a C++ data structure to define what kind of vertex data is stored, and the storage of that data itself. Such a C++ data structure is defined in SCON as

```
struct glayout_in {
        glayout_in(geom_layout_data_in_enum gdt) { data_type = gdt; }

        shader_stream_type_enum  stream_type = shader_stream_type_enum::in;
        geom_layout_data_in_enum data_type   = geom_layout_data_in_enum::undefined;

        std::vector<g_vertex> data = {};
}
```
E-04.05.08

The name glayout_in is to indicate that this structure refers to it being connected to the C++ emulation of a geometry shader.

A constructor is included to define the primitive data type upon creation of a variable of this structure type. For a vertex shader, this same data structure **E-04.04.06** is also valid for emulating the GLSL vertex data out statement defined by **E-04.04.03**. Therefore an enumeration data type shader_stream_type_enum is part of this data structure type to indicate if the emulated layout data structure variable of this type is for input or output.

The C++ enumeration data type for shader_stream_type_enum is

```
enum class shader_stream_type_enum { undefined, in, out };
```
E-04.05.09

Similar to that of glayout_in, the emulation of the GLSL statement for the output of primitive vertex data from a GLSL geometry shader given by **E-04.05.02** can be defined within a C++ data structure to define what kind of vertex data is stored, and the storage of that data itself. Such a C++ data structure is defined in SCON as

```
struct glayout_out {
	glayout_out(geom_layout_data_out_enum gdt, unsigned int ndv) {
		data_type = gdt;
		max_data_values = ndv;
	}

	shader_stream_type_enum stream_type = shader_stream_type_enum::out;
	geom_layout_data_out_enum   data_type   = geom_layout_data_out_enum::undefined;
	unsigned int max_data_values = 0;

	std::vector<g_primitive> data = {};
};
```
E-04.04.10

A constructor is included to define the primitive data type upon creation of a variable of this structure type, and the maximum number of permissible vertices per primitive data type created. The data storage of the primitive data type to output from the geometry shader as defined by **E-04.04.05** is given by the variable data.

The data storage of the primitive data type to output from the geometry shader as defined by **E-04.05.10** is given by the variable data.

The source code of these geometry shader data types and structures exist in the SCON framework as

FrameWork→SCONFW→Shader

within the file

sfw_geometry_shader_base.h

04:06 SCON openGL pipeline emulation : C++ emulation of GLSL Uniform variables

OpenGL GLSL uniforms are defined in the Kronos documentation as “a global shader variable declared with the "uniform" storage qualifier. These act as parameters that the user of a shader program can pass to that program.”

Uniform variables must be defined in GLSL at global scope.

Uniforms can be of any type, or any aggregation of types. The following are all legal GLSL code:

```
struct TheStruct
{
  vec3 first;
  vec4 second;
  mat4x3 third;
};

uniform vec3 oneUniform;
uniform TheStruct aUniformOfArrayType;
uniform mat4 matrixArrayUniform[25];
uniform TheStruct uniformArrayOfStructs[10];
```

GLSL uniforms can have a layout form to specify a uniform location number in the form of

layout(location = <location number>) uniform <data type> <uniform name>;

This is similar to the GLSL vertex attribute layout statement **E-04.04.01** without the in or out qualifier. This suggests that the C++ emulation of such GLSL uniforms is like that of GLSL vertex attribute layouts, which is a C++ structure definition. This is how SCON defines a GLSL uniform, the same principles is applied as that of a vertex layout data structure. A C++ base structure defines the elements that are common to all uniforms which is

```
struct glsl_base_uniform_struct {
	glsl_base_uniform_struct(int loc, int dt, std::string name, bool edit = false) {
		initialise(loc, dt, name, edit);
	}
```

```
	glsl_base_uniform_struct(int dt, std::string uname, bool edit = false) {
		initialise(-1, dt, uname, edit);
	}

	virtual void initialise(int loc, int dt, std::string uname, bool edit) {
		editable = edit;
		location = loc;
		uniform_name = uname;
		data_type = dt;
	}

	bool editable = false;
	int  location = -1;
	int  data_type = GLSL_DT_UNDEFINED;
	std::string uniform_name = "";

	virtual void* get_uniform_data() = 0;
};
```

E-04.06.01

The uniform location number, data type and name are variables concurrent with the same meaning as for a GLSL uniform.

The uniform data can be of any data type to represent any kind of data. Thus the data storage data type for this uniform base structure is unknown in advance. The data cannot be specified as a C++ type void * because to do so in this base class would make coding messy and difficult. This is because in any GLSL shader, multiple uniforms can be specified. Multiple uniforms suggests then in a C++ emulation of a shader can be defined as a list of C++ uniform base structures that can be of the std::vector data type.

That is to have within a C++ emulation of a shader, a list of GLSL uniforms would be in the form of

```
std::vector<glsl_base_uniform_struct> uniforms;
```

Utilising the magic of C++ to be able to process a base structure or class with a virtual function that can return a common void * data type, the solution was to enable such virtual functions within the glsl_base_uniform_struct. This function is

```
virtual void* get_uniform_data() =0;
```

The inspiration for utilising this solution comes from OpenGL itself where such a method is used on the C++ side of OpenGL to define object vertex buffers to transfer vertex attribute data to the GPU hardware and compiled shader code.

Two constructor functions are defined to initialise such a GLSL C++ emulated layout structure.

The C++ emulation of a GLSL unifom of a specific data type is to define a C++ data structure of a name to reflect that data type that inherits the **E-04.06.01** glsl_base_uniform_struct, and defines within it the storage of the uniform data type as a C++ data type, and a data element variable of the same data type;

Such an example of a GLSL uniform statement

```
uniform vec4 camera_position;
```

emulated as a C++ structure is

```
struct uniform_vec4 : public glsl_base_uniform_struct {
	using glsl_base_uniform_struct::glsl_base_uniform_struct;// Bring to derived class the
                                                             glsl_base_uniform_struct contructor

	glm::vec4 value;

	void* get_uniform_data() override {
		return &value;
	}
};
```

E-04.06.02

By design, all C++ uniform data type structure types have the same variable name for the uniform data value of value. Following this this convention, when creating a uniform data structure of any data type, the above code for the structure uniform_vec4 can be copied and pasted in full, then only the name of vlayout_vec4 for the unform data type needs to be changed, and the data type of the variable value.

In a C++ shader emulation such a uniform structure is defined for

luniform vec4 camera_position;

is

uniform_vec4 camera_position = uniform_vec4(GLSL_DT_VEC4, "camera_position");**E-04.06.03**

to create and initialise this uniform ready to have camera position data assigned to it;

The C++ emulation code for this and more uniform data type structures exists in the SCON framework in the directory

FrameWork→SCONFW→Shader

within the file

sfw_shader_uniform_data_types.h

04:06 SCON openGL pipeline emulation : C++ emulation of a GLSL shader

With the C++ emulation of the OpenGL GLSL layout and uniform data types defined as in sections **04.04 to 04.06**, the vertex and geometry shader emulation can be defined.

The Kronos documentation stipulates that all GLSL shaders have defined built in variables that exist for input and output.

There is no GLSL built in variable for the input of vertex shader data. The input data variable for a vertex shader is defined by the in layout definition that the user creates. (section **04.04**) Hoever there are some inbuilt variables that the vertex shader does have defined.

```
in int gl_VertexID;
in int gl_InstanceID;
in int gl_DrawID; // Requires GLSL 4.60 or ARB_shader_draw_parameters
in int gl_BaseVertex; // Requires GLSL 4.60 or ARB_shader_draw_parameters
in int gl_BaseInstance; // Requires GLSL 4.60 or ARB_shader_draw_parameters
```

SCON does not use these and thus are no C++ emulation of these are defined as yet

However looking at the Kronos documentation, all other shaders have a built in variable structure for every vertex data point that is of an input nature is of the form

```
in gl_PerVertex
{
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];
```

This would suggest that every C++ emulated shader would have a C++ vector of data structure type gl_pervertex as defined in **E-04.03.01** with a name gl_in which is accessed to process OpenGL vertex data from a previous shader that is in the OpenGL pipeline as outlines in **Fig 04.01**.

As well, the Kronos documentation indicates that all shaders have a built in variable structure for every vertex data point that is of an output nature is of the form

```
out gl_PerVertex
{
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_out[];
```

This would suggest that every C++ emulated shader would have a C++ vector of data structure type gl_pervertex as defined in **E-04.03.01** with a name gl_out so as to output or export vertex data to the next shader that is in the OpenGL pipeline for processing as outlines in **Fig 04.01**.

That is every shader in the C++ emulation of the OpenGL pipeline has a data storage vector for the input and output of vertex data of form

```
std::vector<gl_PerVertex> gl_in  = {};
std::vector<gl_PerVertex> gl_out = {};
```

04:06.03
04:06.04

What this suggests is that a C++ base GLSL shader class needs to be defined that is then inherited into

vertex and geometry emulation shader classes.
This is strengthened when it is considered that every shader can have within it a number of input and output vertex data attribute data types, and uniforms. Such a C++ emulation base class is defined in SCON as

```
class sconfw_shader_base {
public:
	sconfw_shader_base() {}
	~sconfw_shader_base() {}

	void define_glsl_version (unsigned int v) { glsl_version = v;}		Define a glsl version for this emulation

	void enable_point_size() {						OpenGL command to enable point
		glEnable(GL_PROGRAM_POINT_SIZE); GCE			size redering
	}

	virtual void scon_main() {}				C++ main function loop to process all vertex data

	virtual void scon_main_code(size_t i) {}		C++ emulation of glsl shader main function per vertex

	virtual void scon_main_code(size_t i,size_t j) {}	C++ emulation of glsl shader main function per vertex

	std::vector<gl_PerVertex> gl_in  = {};				Storage of shader vertex input data
	std::vector<gl_PerVertex> gl_out = {};				Storage of shader vertex output data

	std::vector<glm::vec4> color_in  = {};			Storage of shader vertex input color data
	std::vector<glm::vec4> color_out = {};			Storage of shader vertex output color data

	std::vector<glsl_base_uniform_struct *> uniforms;		Storage of shader uniforms

	void add_uniform(glsl_base_uniform_struct *uniform) {	Add a uniform to the shader uniforms storage
		uniforms.push_back(uniform);
	}

	std::vector<vlayout_base_struct*> input_layouts;		Storage of shader input vertex attribute data
	std::vector<vlayout_base_struct*> output_layouts;		Storage of shader output vertex attribute data

	void add_input_layout(vlayout_base_struct* input_layout) { Add a layout to the input layout storage
		input_layouts.push_back(input_layout);
	}

	void add_output_layout(vlayout_base_struct* output_layout) { Add a layout to the output layout storage
		output_layouts.push_back(output_layout);
	}

protected:
	unsigned int glsl_version    = 0;
	unsigned int draw_mode       = 0;
	unsigned int number_vertices = 0;

	glm::vec4 gl_Position  = {0.0f,0.0f,0.0f,1.0f};
	float     gl_PointSize = 1.0f;
	glm::vec4 gl_color     = {1.0f,1.0f,1.0f,1.0f};

	void clear_buffer_in_data() {				Clear all current input vertex data
		gl_in.clear();
		color_in.clear();
	}

	void clear_buffer_out_data() {				Clear all current output vertex data
		gl_out.clear();
		color_out.clear();
	}
}
```

04:06.05

A description of what each line of this C++ shader emulation base class does or is for is given in on the right in the colour purple.

The color_in and color_out are not strictly needed, but are part of this base class as vertex colour is such a common vertex attribute that is used in most, if not all shader programs, it is deemed to be more efficient to have it defined in the shader base class rather than having to add it as a uniform using the add_uniform function. The functions, clear_buffer_in and clear_buffer_out functions are required as the input and output of a shader will change per loop render cycle, and not clearing these emulated buffer storages of gl_in and gl_out will mean that

either the data is not updated, or that the vector storage will overflow and crash the application, if not the computer.

The scon_main function is the function that manages and defines

1 : The initialisation of input vertex gl_in, vertex attribute and color data.
2 : The running of the loop to process the data of step 1 for each vertex as a C++ shader emulation and save the C++ emulation resultant vertex data into an output storage data type for the next step of the OpenGL GLSL pipeline.
3: Save to the vertex gl_out and out layout data types to be used in next step of the OpenGL GLSL pipeline.

The scon_main_code functions are where the C++ emulation of the OpenGL GLSL main function is performed for each vertex data point primitive, and is called with the loop of the scon_main function for each vertex data point primitive.

And thus has a basic form of pseudo code for any C++ shader emulation OpenGL pipeline step is

```
scon_main{
    initialise input vertex gl_in data
    initialise input vertex attribute data

    for each vertex gl_in {
            scon_main_code(vertex index);
            save processed vertex output data to a storage data type;
    }

    create gl_out, out layout vertex attribute data;
}
```

04:06.06

For all C++ GLSL emulations of a GLSL shader, a C++ class object that emulates a GLSL shader will inherit the sconfw_shader_base class as the core of its emulation. It will have the virtual functions scon_main and scon_main_code C++ code written to suit the emulation of that shader function and the GLSL code to perform the tasks that that shader performs to achieve the desired result in the emulated OpenGL pipeline.

The definition of this class and the associated C++ code of it exists in the SCON framework in the directory

FrameWork→SCONFW→Shader

within the file

sfw_shader_base.h

04:07 SCON openGL pipeline emulation : C++ emulation of a GLSL vertex shader

By examining the documentation, examples and through experience creating GLSL vertex shaders to be used in the OpenGL pipeline, on all accounts, the sconfw_shader_base class defined in section **04:07** is sufficient to be used on its own as a minimum and inherited into a C++ vertex base class without any additional functions, data structures or variables.

The definition of the vertex base class exists in the SCON framework in the directory

FrameWork→SCONFW→Shader

within the file

sfw_vertex_shader_base.h

04:08 SCON openGL pipeline emulation : C++ emulation of a GLSL geometry shader

By examining the documentation, examples and through experience creating GLSL geometry shaders, the emulation of a geometry shader not only needs to inherit the sconfw_shader_base class, but also needs some extra functions added to correctly import and export the vertex position and attribute data that a geometry shaders are designed to perform.

As indicated in section **04:05**, the layout of vertex data receives as input from the vertex shader that precedes it in the OpenGL pipeline is defined to be of a particular primitive data type as given by **TABLE 04:05:01**, and that the layout of this data is defined in GLSL by the statement

layout(<input primitive>) in; (**E-04.05.01**)

Where <input primitive> is one of the primitive data types defined by **TABLE 04:05:01**

and the general GLSL output layout statement is given by

layout(<output primitive>, max_vertices = X) out; (**E-04.05.02**)

The C++ structures to emulate the storage of these layouts was given as the C++ statements **E-04.04.07** to **E-04.04.10** in section **04:05**. Therefore a C++ class emulating of a GLSL geometry shader has as a part of

its functionality the C++ data storage structures defined by statements **E-04.04.07** to **E-04.04.10** in section **04:05**.

04:08:01 Input vertex primitive data

The input vertex primitive data that the geometry shader receives is the output primitive vertex data that the vertex shader that precedes the geometry shader in the OpenGL pipeline. This data is in the format of a one dimensional array of vertex data that represents the type of GLSL primitive specified by statment **E-04.05.01**.

The input primitives that a GLSL geometry shader can receive are the output primitives from a vertex shader as given in **TABLE 04:05:01**. However, for the current C++ emulation of the GLSL vertex and geometry shaders, the accepted vertex data primitives that are exported from the emulated vertex shader and input into the C++ emulated geometry shader is restricted to points, lines and triangles.

If the input primitive is of a type point, then each element of this input is a vertex. If If the input primitive is of a type line, then every sequence of two elements of this input array are the vertices that make up a line. If the input primitive is of a type triangle, then every sequence of three elements of this input array are the vertices that make up a triangle.(see number vertices column in **TABLE 04:05:01**.)

This is important, because in GLSL geometry shader code, gl_in[index] represents the vertex of index of the input primitive data type specifiied by the layout statement **E-04.05.01**. Thus a C++ GLSL emulation of this data storage for gl_in needs to emulate this. This storage is defined by C++ structure glayout_in as defined by **E-04.05.08**. This structure defines data primitive type that is the input to the geometry shader and the storage of the primitive data type as an array of an array of vertices that constitutes the makeup of that data primitive data type.

In the GLSL shader code, gl_in represents a single individual vertex data primitive that is being processed by the GPU. What this means is that the C++ emulation of a geometry shader has the gl_in of the geometry shader of an individual vertex data primitive being an element the vector array designated as data of the glayout_in structure. Ie std::vector<g_vertex> data.

So to emulating a GLSL gl_in statement in the C++ geometry emulation, a reference is made to data value of the glayout_in structure.

04:08:02 Output vertex primitive data

The output vertex primitive data that the geometry shader generates and is defined by the GLSL statement **E-04.05.02** is passed on to the next step of the GLSL pipeline as illustrated in **Fig 04.01**. As far as the OpenGL GLSL programmer is concerned, this is the GLSL fragment shader.

The C++ SCON OpenGL GLSL geometry shader emulation of the specified output primitive is that it has to be in a data structure that represents that output primitive data that is generated for each instant of input primitive data. Such an output data storage structure is that given by the glayout_out as defined by **E-04.05.10**. The permitted output primitive data types of the C++ SCON geometry shader emulation are points, line_strip, triangle_strip that is defined by C++ emulation class geom_layout_data_out_enum as defined by C++ statement **E-04.05.07**. Others can be added as required.

The primitive data storage given by they varible data within the glayout_out structure is a vector of the primitive data type that is to be output.

In the GLSL shader code, gl_out represents a single individual vertex data primitive that is being ouput to the next OpenGL pipeline step. What this means is that the C++ emulation of a geometry shader has the gl_out of the geometry shader of an individual vertex data primitive being an element the vector array designated as data of the glayout_out structure. Ie std::vector<g_primitive> data.

So to emulating a GLSL gl_in statement in the C++ geometry emulation, a reference is made to data value of the glayout_out structure.

04:08:02 EmitVertex and EndPrimitive Emulation

In a GLSL geometry shader, the process of creating the data for output to a primitive data type from the current input primitive data type is given aas being of the form

```
gl_position = <statement1>
EmitVertex();
gl_position = <statement2>                    E-04.08.01
EmitVertex();
        :
EndPrimitive();
```

gl_position is the gl_position variable of the GLSL gl_PerVertex structure given in **E-04.02.01**. This can be easily emulated in C++ as a variable within the C++ geometry shader base class as

```
glm::vec4 gl_Position;
```
E-04.08.02

Emulation of the EmitVertex() statement can be implied as simply being adding the current value of gl_Position to a vector array of gl_PerVertex data. Such an emulation function that is used and added to the C++ geometry shader base class is

```
void EmitVertex() {
        gl_PerVertex vertex;
        vertex.gl_Position  = gl_Position;
        vertex.gl_PointSize = gl_PointSize;
        vertices_out.push_back(vertex);
}
```
E-04.08.03

vertices_out is a variable of the data type g_vertex given by **E-04.05.04** which is an array of the data type gl_PerVertex. Ie

```
g_vertex    vertices_out;
```
E-04.08.04

Emulation of the EndPrimitive() statement can be implied as simply being adding the current primitive array data to a vector array of primitive data. Such an emulation function that is used and added to the C++ geometry shader base class is

```
void EndPrimitive() {
        primitive_out.push_back(vertices_out);
        vertices_out.clear();
}
```
E-04.08.05

primitive_out is a variable of the data type g_primitive given by **E-04.05.06** which is an array of the data type gl_vertex. Ie

```
g_primitive    primitive_out;
```
E-04.08.06

These factors of emulation of the major inputs and process of creating the output for a C++ geometry base emulation are added to the C++ geometry base emulation class that has its layout data types decribed in section **04.05**.

The source code of these geometry shader data types and structures exist in the SCON framework as

FrameWork→SCONFW→Shader

within the file

sfw_geometry_shader_base.h

05 : The SCON OpenGL Pipeline Emulation process

In section 04, the groundwork for creating an OpenGL pipeline emulation involving vertex and geometry base shaders was established. What is needed next is to implement a C++ emulation of the pipeline connecting the vertex and geometry shaders together, and display the results to the computer screen.

A C++ OpenGL pipeline emulation would need to follow the same sequence as the OpenGL pipeline depicted in **Fig 04.02** and **Fig 04.01**. The steps of this emulation is summarised and illustrated in **Fig 05.01**.

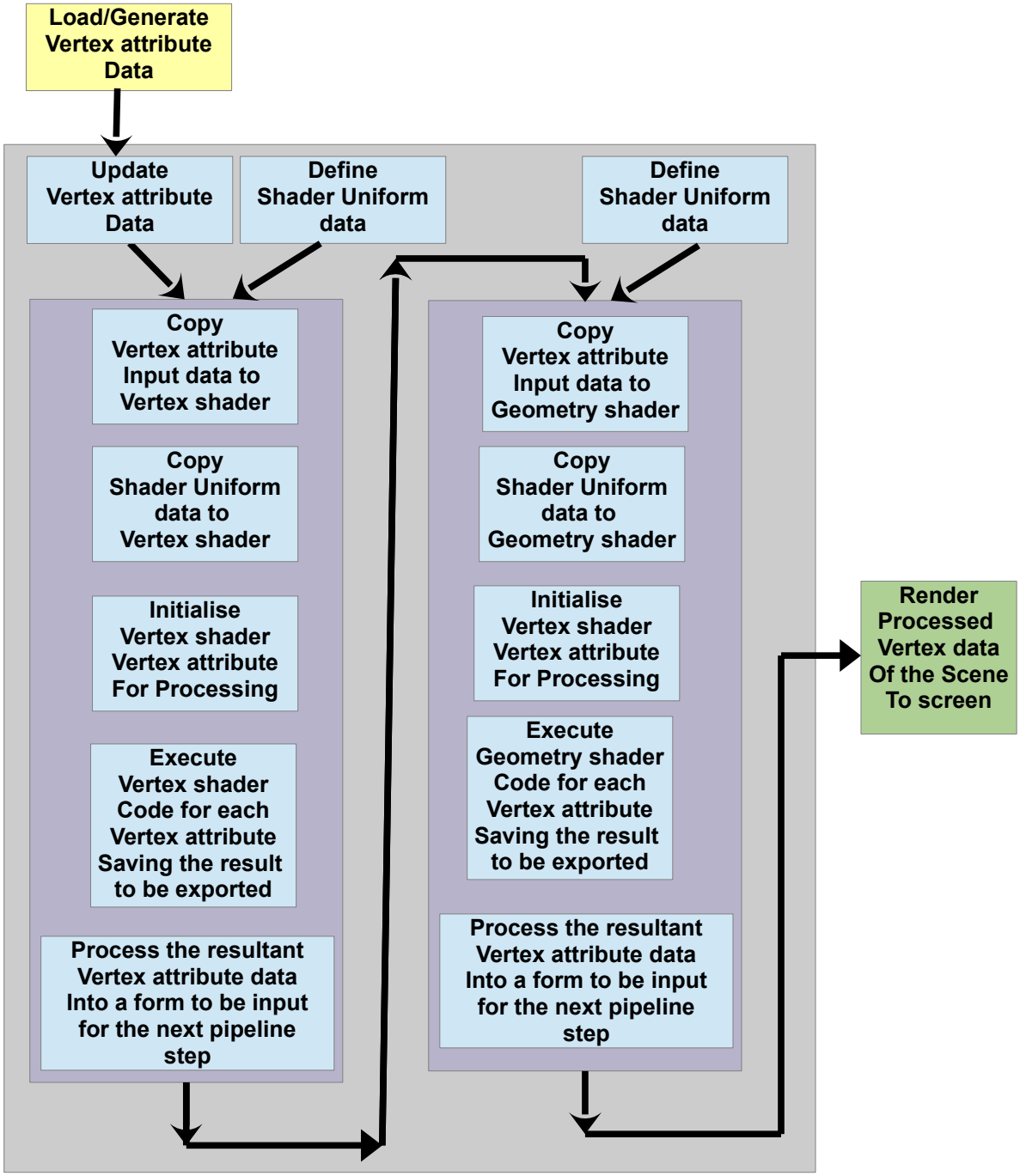


Fig 05.01

The internals of the emulation of the vertex shader and geometry shader has been discussed in section 04. To implement the emulation of the OpenGL GLSL pipeline is to perform the tasks of

S05-i : Initialisation :

Generating the initial vertex attribute data and initialise an emulation of a OpenGL vertex buffer object that is to be used by the emulated vertex shader. (Yellow box contents of **Fig 05.01**)

S05-ii : Performing pipeline emulation procedure:

Update the emulated Uniform and vertex attribute data to be used in the OpenGL pipeline emulation. Execute the vertex, and if used, the geometry shader emulation to create and save the vertex geometry data in a format to be used for rendering. (Grey box contents of **Fig 05.01**)

S05-iii : Render Geometry:

Render the output geometry generated in step ii.(Green box contents of **Fig 05.01**)

Steps i to iii above is the SCON emulation of the OpenGL pipeline process at the topmost level. Step i is performed once outside of an execution loop of user interaction, while steps ii and iii are within an execution loop of user interaction. What this means is that the C++ OpenGL emulation pipeline can be expressed as a C++ base class that has two functions to perform the pipeline procedure of step ii and rendering of geometry that is step iii. Because the pipeline and geometry rendering will be different based upon the primitive input and primitive output data type, these functions will be virtual functions. This pipeline base class is defined as in **E-05.01**

```
class sconfw_pipeline_base_class
{
public:
    sconfw_pipeline_base_class() {}
    ~sconfw_pipeline_base_class() {}

    void define_sconfw_camera(oglfw_camera_base_class *selected_camera){
        sconfw_camwera.selected_camera = selected_camera;
    }

protected:
    sconfw_camera_class sconfw_camwera;

    sconfw_buffer_data_input_base_class buffer_data_input;// defined custom user data buffer class
    sconfw_vertex_shader_base           vertex_shader;    // defined custom user vertex emulation shader class
    sconfw_geometry_shader_base         geometry_shader;// defined custom user geometry emulation shader
                                                        // class. Comment out if if not used.

    void virtual perform_pipeline_procedure(Model& mesh_object, size_t mesh_id = 0) {}

    void virtual render_geometry(){}

private:
};
```

E-05.01

The sconfw_camera_class is the camera class that is used in step iii to render the scene. The public function define_sconfw_camera is to define a pointer to the camera class that has been created and defined externally in the main application to the sconfw_pipeline_base_class.

The sconfw_buffer_data_input_base_class is the C++ emulation of an OpenGL vertex buffer object that is used to store and migrate vertex attribute data to and from the C++ GLSL shader emulations.

The source code of the sconfw_pipeline_base_class exist in the SCON framework as

FrameWork→SCONFW→Process

within the file

sfw_pipeline.h

and that of the sconfw_buffer_data_input_base_class within the file

sfw_buffer_data_input.h

06 : Implementing the C++ SCON OpenGL Pipeline Emulation

The three steps outlined in section 05 are performed to execute the emulation of an OpenGL pipeline. Because every application of the C++ OpenGL pipeline will be different depending upon the input data and goals and that the user wants to achieve, the detail of the coding of the three steps of implementation outlined in seection 05 to achieve those goals will be different. What this means is that the sconfw_pipeline_base_class is inherited into another pipeline class that has additional functions and data to perform steps **S05-i** to **S05-iii**.

For the purposes of explanation, this derived class is called sconfw_pipeline_template_base_class. This derived class thus can be used as a template class to create other template classes by a direct copy and modification, or be where applicable, inherited.

Step **S05-i** is to generate or read vertex data into the C++ emulated buffer of class

sconfw_buffer_data_input_base_class

The sconfw_buffer_data_input_base_class has the most common vertex attribute data defined in the class for vertex position, point size, indices, color and normals.

```
std::vector<glm::vec3> positions   = {};
std::vector<float>         point_sizes = {};
//std::vector<std::vector<float>>  clip_distance = {}; //uncomment if ever going to use

//Data vertex indices for index draw
std::vector<unsigned int> indices = {};

// default attributes
std::vector<glm::vec4>    colors  = {};
std::vector<glm::vec3>    normals = {};
```

If there are other vertex data types that are needed, a C++ class that inherits this base class can be created with a suitable name and std::vector of the data type of this attribute. Looking at the contents of the sconfw_buffer_data_input_base_class base class, additional clear, shrink_to_fit and initialisation functions will be needed to be added. What this also means is that this derived buffer data class will replace the sconfw_buffer_data_input_base_class in the derived sconfw_pipeline_template_base_class.

Because there will be differences in what and how vertex attribute data is generated and stored in every C++ OpenGL pipeline, no common function or virtual function can be defined in the sfw_pipeline_base_class. Thus the sconfw_pipeline_template_base_class will have different functions and code to define the vertex attribute data to be used in the C++ OpenGL pipeline emulation.

In most cases, the vertex_shader and geometry_shader base classes defined in the sfw_pipeline_base_class that was given in section 05 will be a derived class of the sconfw_vertex_shader_base and confw_geometry_shader_base classes. Thus sconfw_pipeline_template_base_class will in effect in most cases only use the reference to the current camera used to render the scene, and the virtual functions to perform the C++ OpenGL pipeline and rendering tasks. The purpose of sfw_pipeline_base_class to exist is to enable the C++ feature of defining multiple derived classes of sfw_pipeline_base_class to be stored in a vector array of sfw_pipeline_base_class, so as to be able to implement more than one C++ OpenGL emulations.

These derived class of the sconfw_vertex_shader_base and confw_geometry_shader_base classes are therefore used to perform step **S05-ii**. The vertex attribute data defined and saved in the data structure sconfw_buffer_data_input_base_class or a derivative thereof is referenced by the vertex_shader class for processing. This processing is performed as illustrated in Fig 05.01.

The resultant vertex attribute data that is processed by the C++ emulation of the vertex shader is saved into a data structure of data type of

```
std::vector<gl_PerVertex> gl_out = {};
```

E-04.02.04

as explained in section 04.02 for vertex position and point size data.

Other output vertex attribute data types need to be also saved into the appropriate std::vector of their data type. Eg for color std::vector<glm::vec4> color_out;

gl_out of the vertex_shader class is the input vertex position and point size data for the C++ geometry shader emulation. That is gl_in as defined in section **04.02**.The other output vertex attribute data from the vertex shader is the input vertex attribute data of the same data type in the geometry shader. Eg color_out of the C++ vertex shader emulation becomes color_in in the C++ geometry shader emulation.

The C++ geometry shader emulation class then takes this gl_in and the other vertex attribute data and transforms it into the structure data type glayout_in (**E-04.05.08**) for as described in section **04.05** to be ready to be processed. The transformation of the gl_in data will be different depending upon what OpenGL

primitive data type that the gl_in represents.

The C++ geometry shader emulation class then updates its emulated uniform to be used in the processing of vertex attribute data.

The C++ geometry shader emulation class then processes the OpenGL primitive and any other vertex attribute data. The output primitive vertex data is stored in the data type glayout_out according to the output primitive data that is define as in section **04.05**, and **04.08**.

The C++ geometry shader emulation class then transforms this output primitive data of glayout_out data type the into the form of the gl_out data type. Any other processed vertex attribute data such as color are also tramsformed into their output data types.

The output data from the C++ geometry shader emulation class is used to perform step **S05-iii** to render the vertex primitive data and associated attribute data to screen.

Fig 06.01 gives a different perspective of the C++ OpenGL pipeline emulation in **Fig 05.01** in terms of flow of data as described above.

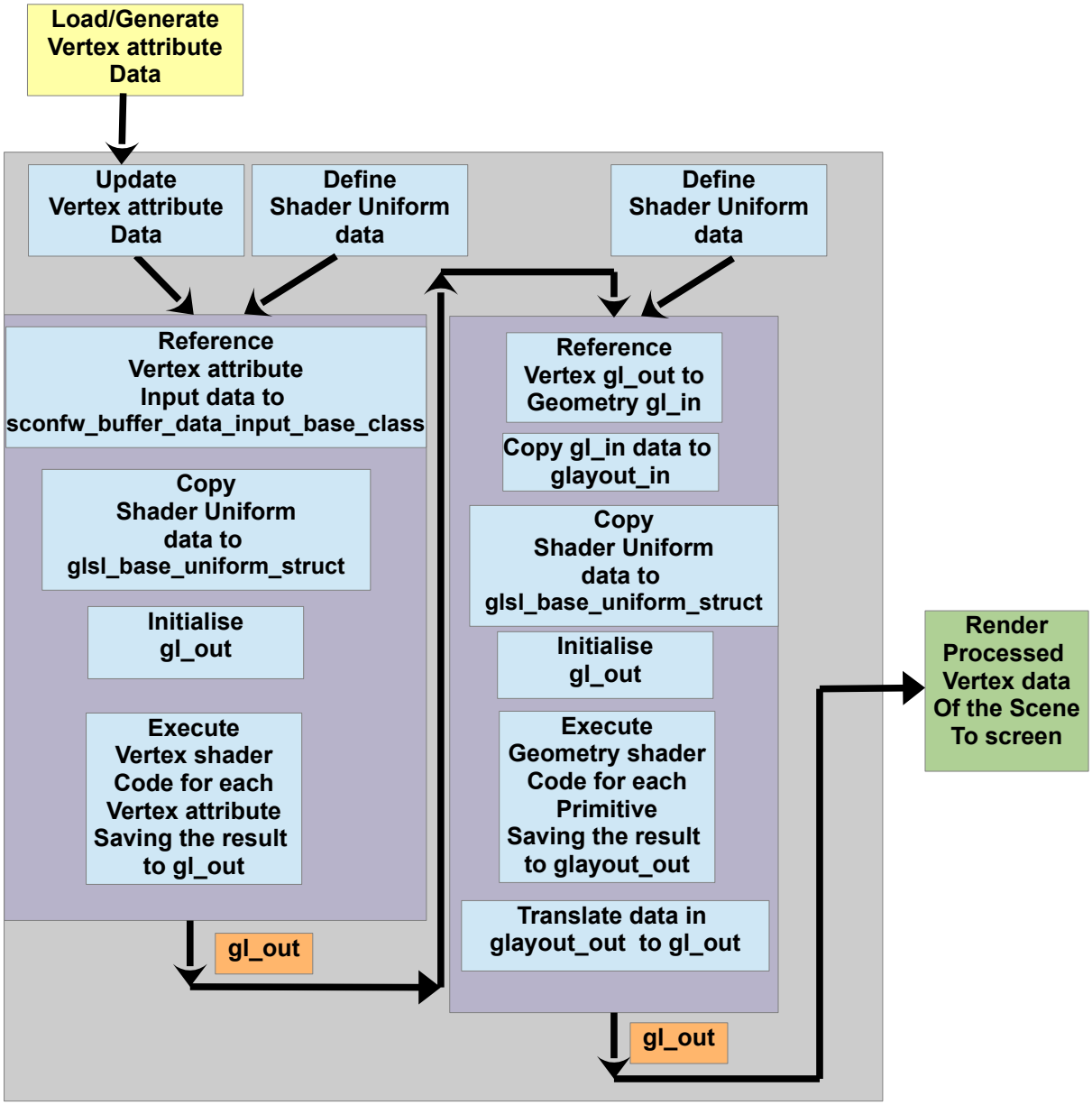


Fig 06.01

Examples of derived sconfw_pipeline_template_base_class can be found in the SCON framework in the directory

FrameWork→SCONFW→Templates

within the files

sfw_pipeline_template.h
sfw_grid_pipeline.h
sfw_triangles_pipeline.h

sfw_pipeline_template.h is an example of a pass through of point cloud vertex data without modification into an emulated vertex and geometry shader.

sfw_grid_pipeline.h is an example of point cloud vertex data passing into an emulated vertex shader and passed into a geometry shader where the geometry shader generates lines that forms a reference grid representing the planes of the axes of a 3D coordinate system.

sfw_triangles_pipeline.h is an example of point cloud vertex data of a hexagonal surface passing into an emulated vertex shader and passed into a geometry shader. The geometry shader generates from these points columns of heights representing the z value of the points of the hexagonal surface.

All pipeline classes require to have the emulated vertex and geometry shader classes incorporated into them as a class object. **Fig 06.02** gives a schematic of the inclusion of the required vertex and geometry shader emulation classes into all C++ OpenGL GLSL emulated pipeline classes.

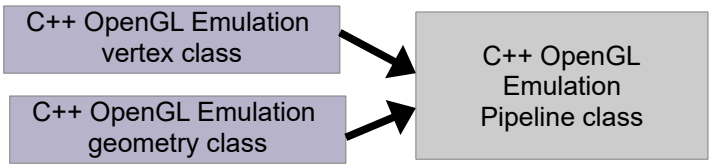


Fig 06.02

07 : Implementing the C++ SCON OpenGL Pipeline Emulation Rendering

The final step of the OpenGL pipeline emulation is the rendering of the output of the vertex attribute data generated by the C++ OpenGL GLSL shader emulation as illustrated in **Fig 05.01** and **Fig 06.01**. This rendering step is, in actuality, the use of the OpenGL GLSL pipeline as illustrated in **Fig 04.01**.

When preparing vertex attribute data for migration or transfer to the GPU and complied GLSL shader program, within any OpenGL C++ application the following steps are often, if not always required.

S07-i : Copy vertex attribute into a vertex buffer object (VBO) and define define buffer data layouts
S07-ii : Compile the code of a GLSL shader program to use and render the vertex attribute data of **S07-i**.
S07-iii : Specify the compiled OpenGL shader program of **S07-i** to use to render the data within a VBO.
S07-iv : Bind the VBO in **S07-i** to specify that this VBO is to be used for rendering
S07-v : Execute the OpenGL draw command specifying the OpenGL primitive data type that is stored within the VBO to be rendered.
S07-vi : Unbind the VBO in **S07-i** to specify that this VBO is no longer to be used for rendering.
S07-vii : Free the compiled OpenGL shader program of **S07-iii** to no longer be used.

Step **S07-v** incorporates the OpenGL statement

glDrawArrays(GLenum mode,GLint first,GLsizei count);

where by reference to the Kronos documentation
mode : Specifies what kind of primitives to render. Symbolic constants GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY and GL_PATCHES are accepted.

first : Specifies the starting index in the enabled arrays.
count : Specifies the number of indices to be rendered.

In section **Fig 06.01** indicates that the final output of the C++ emulated vertex or geometry shader is in the data structure format of gl_out that is defined by **04:06.04** (which is also gl_in **04:06.03**) as in section **04:06**. This output vertex attribute data, as specified by the standard of the Kronos group for the OpenGL GLSL program code must also be one of the OpenGL primitive data types that the glDrawArrays accepts as its mode of drawing. Thus any output vertex attribute data that the SCON C++ OpenGL GLSL emulation produces can be rendered, without modification, by a true OpenGL GLSL compiled program.

What this means is that to render the output of the C++ vertex or geometry emulation, already having the gl_out data in one of these modes, is to for the glDrawArrays statement,

S07-01-i : **E**xtract the vertex attribute data from gl_out and other associated vertex attribute data into an OpenGL VBO object.
S07-01-ii : Specify the starting index to draw from as zero.
S07-01-iii : Specify the the number of indices to be rendered as the number of vertices or size of gl_out.

Once this is done, then the GLSL shader program that is to receive and render the data can be no more than a simple GLSL to render the vertex attribute data without modification of the form for the vertex shader

```
layout (location = 0) in vec4  v_position;
layout (location = 1) in float   v_point_size;
layout (location = 2) in vec4  v_color;

out vec4 o_color;     // Output color to Fragment shader
```

E07-01

```
void main() {
   gl_Position  = v_position;
   gl_PointSize = v_point_size;
   o_color      = v_color;
}
```

and fragment or pixel shader

```
in vec4 o_color;

layout(location = 0) out vec4 frag_color;// out put color
```

E07-02

```
void main() {
   frag_color = o_color;
```

```
}
```

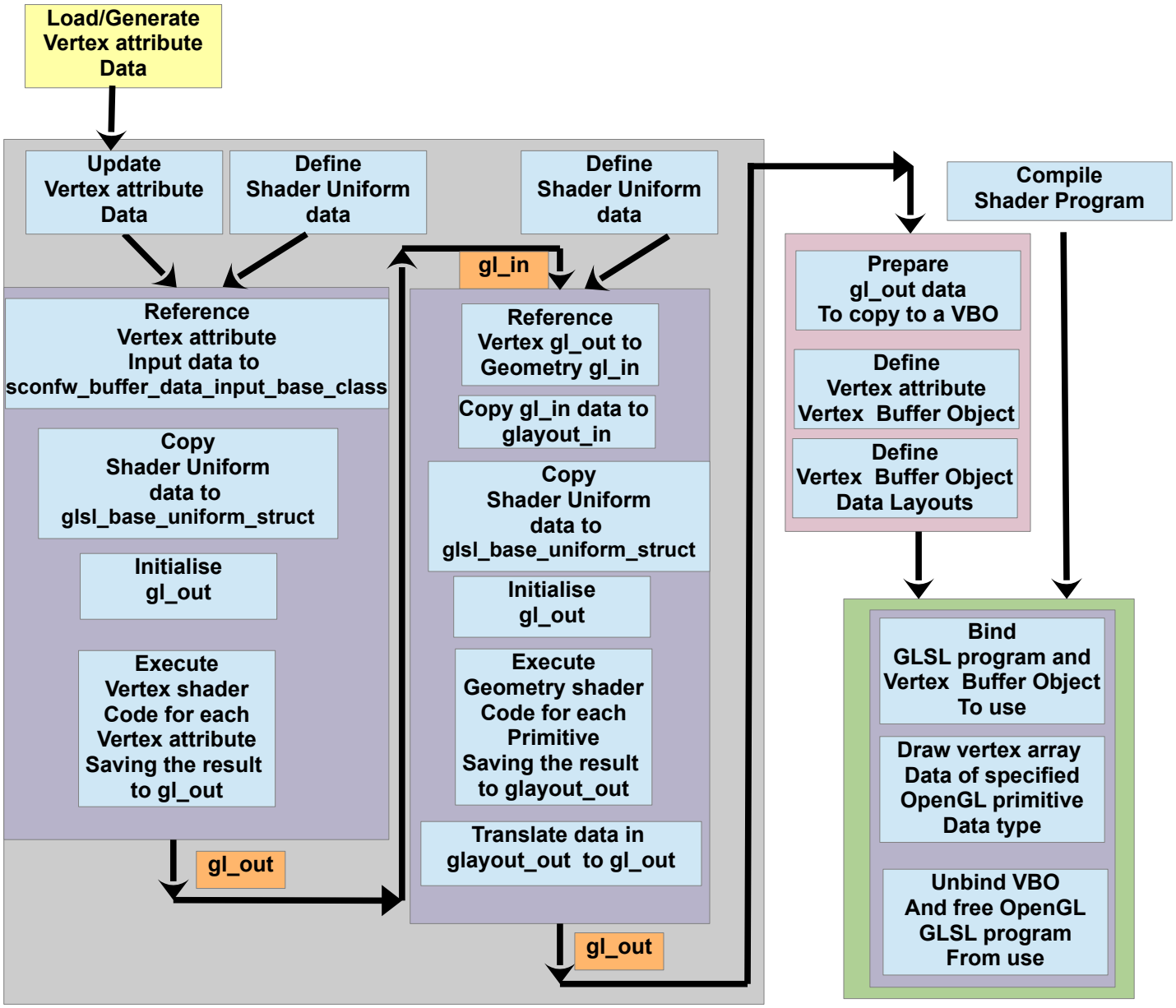
The vertex and fragment shader program code given in **E07-01** and **E07-02** should be all that is needed to render most of the C++ OpenGL GLSL emulation gl_out data that are generated. This is because the SCON C++ emulation is not designed to handle the more advanced OpenGL graphical tasks of texture mapping.

Nor is it designed to emulate fragment shaders in any form. By looking at the OpenGL pipeline in **Fig 04.01**, a fragment shader is implemented after the rasterisation step, and modifies the raster image of the resultant rendering image. In other words, the fragment shader is a kind of post rendering step that to emulate in C++ would be to perform glsl code on a finished rasterised image. That is, a fragment shader emulation would involve editing an existing image, something that is not the goal of ther SCON application.

As mentioned in section **05**, the rendering step in the SCON C++ emulation is performed within the virtual function render_geometry(). Steps **S07-iii** to **S07-iii** are to be performed within this function for all emulations. Steps **S07-i** and **S07-ii** are to be performed to prepare the data and compile the vertex and fragment shader program with the code of **E07-01** and **E07-02** prior to calling the render function render_geometry().

08 : The Complete C++ SCON OpenGL Pipeline Emulation

Putting all the elements mentioned in sections **04** to **07**, a complete C++ OpenGL pipeline emulation can be summarised in **Fig 08.01**.



Emulation of Vertex Shader Emulation of Geometry Shader Emulation of Render process

Fig 08.01

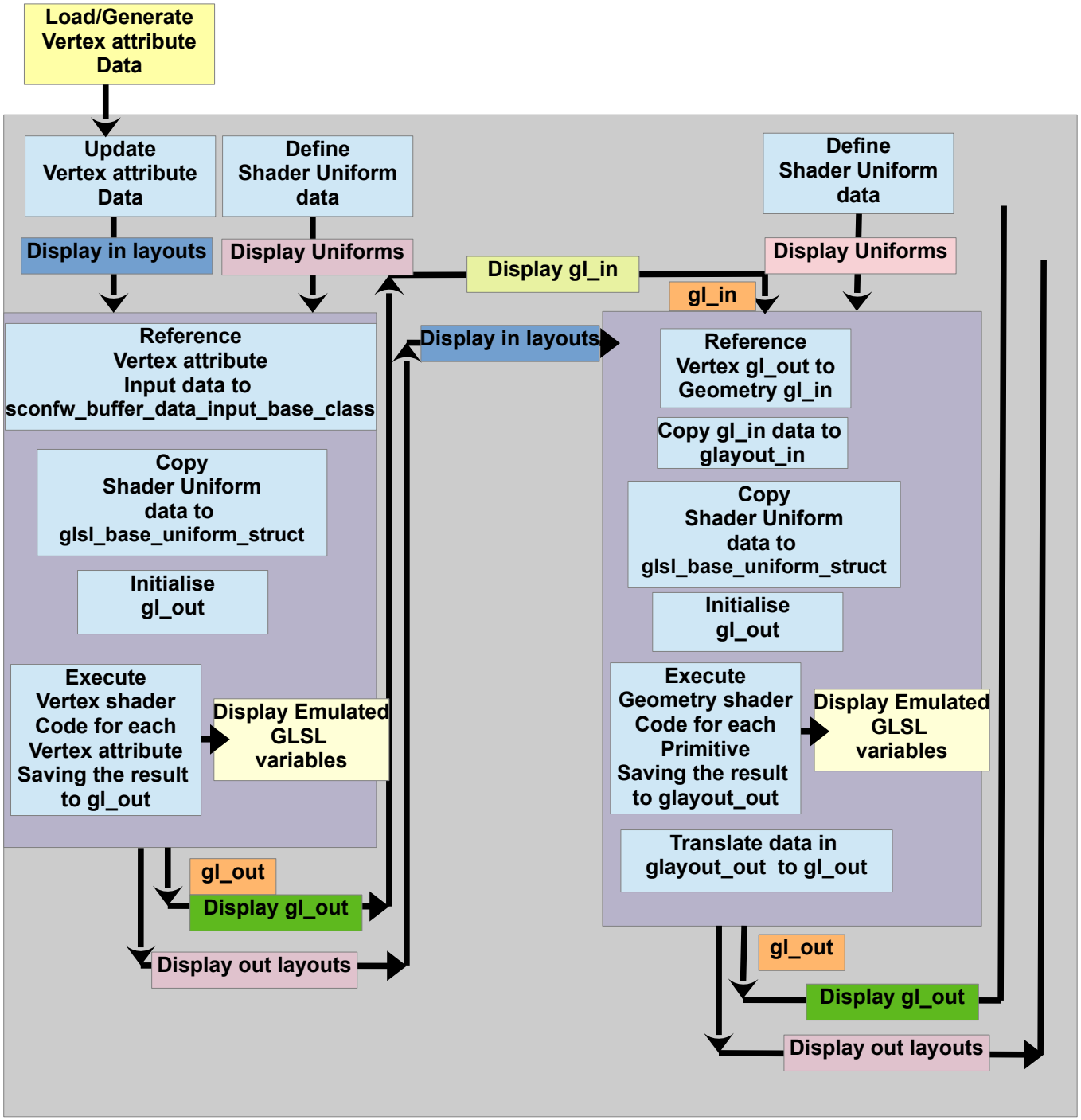
One step not mentioned in section **07** is that the resultant output of gl_out and associated vertex attribute data from the C++ OpenGL GLSL emulation needs to be prepared the to be copied into a vertex buffer object. This is added in the illustration **Fig 08.01**.

What can be seen in **Fig 08.01** is that the OpenGL emulation pipeline will still work if the geometry shader is not used as part of the emulation. The gl_out from the vertex shader emulation is in the same format as that from the emulated geometry shader. This is because the C++ OpenGL pipeline emulation design follows the Kronos specifications as close as possible as inferred by the documentation.

09 : Real Time Monitoring of, and using the SCON emulation to debug OpenGL GLSL code.

The prime purpose of the SCON toolkit is for it to be used to create, monitor, and debug emulated OpenGL vertex and geometry GLSL shader code so as to then translate that emulated code into a true OpenGL GLSL shader program to be compiled and run on a GPU.

Looking at the design of the complete SCON emulation in **Fig 08.01**, a guide into how the debugging of the OpenGL emulation can be seen.



Emulation of Vertex Shader Emulation of Geometry Shader

Fig 09.01

First is the gl_out and gl_in data streams into each of the vertex and geometry shaders. Each stream of data is an array of the OpenGL specified PerVertex data structure, regardless of what OpenGL primitive that the data represents. That is, gl_in and gl_out being the data structures defined as std::vector<gl_PerVertex> in **E-04.02.03** and **E-04.02.04** respectively can be displayed in a table similar to a spreadsheet. However, because these can be very large in number, it is best to only display a portion of this data at a time, and not all as it can use up large amounts of RAM and even potentially crash the compiled emulation code. The function to perform this enquiry of gl_in can be placed just before being processed by the geometry shader,

and gl_out can be placed just after the final step of both the vertex and geometry.(see **Fig 09.01**)

The next consideration can be applied to the input and output vertex attribute layout data. The vertex shader does not get its input vertex attribute data of position or vertex size from a gl_in data stream, but from a vertex attribute data layout. Therefore the vertex shader emulation gets its vertex position data from an emulated GLSL layout structure of the form of **E-04.04.03**. The vertex attribute data like that of gl_in and gl_out, is stored in a C++ vector structure which can be considered as a one dimensional array. Therefore, similar to the display of gl_in and gl_out data, the vertex attribute data to display and examine is in the same manner as the gl_in and gl_out data, as a spreadsheet like table. Therefore, the function to perform this enquiry of any input vertex attribute layout is placed just before being processed by a shader, and any output vertex attribute layout generated is placed just after the final step of a shader (see **Fig 09.01**)

Every shader can utilise a GLSL uniform to communicate run time variables from the C++ code to the GLSL shader program. The emulation of such uniforms is made possible in SCON by use of the C++ data structure in the form similar to **E-04.06.02** of section **04**. Uniforms are a variable of a single value that is not connected to the vertex attribute data and cannot be modified by a GLSL shader. The same applies to the SCON GLSL emulation. Thus the uniform data that is to be used by an GLSL shader emulation is best to be displayed before the emulation code is executed. (see **Fig 09.01**)

Many OpenGL GLSL shader programming debugging software like renderdoc or Nvidia Nsight require a snapshot of the OpenGL GLSL program to captured and freeze the GLSL application in a frame of execution of that GLSL program so as to examine the gl_in, gl_out, layout and uniform data. In the C++ SCON GLSL emulation toolkit, as outlined above and illustrated in **Fig 09.01**, in a correctly set up emulation this can be done in real time. This is the design, purpose and goal of the SCON toolkit, to be able to examine in real time the data that a GLSL program uses and creates in real time for the purpose of debugging or understanding the behaviour of a GLSL program.

To examine the gl_in, gl_out, vertex attribute layouts and shader uniform data, a software package or utility suitable at accepting the C++ OpenGL emulation data structure of these data types and creating a user interactive display of these data types would be appropriate to use. The SCON OpenGL emulation toolkit uses the popular C++ dear ImGui docking software package to do this, and can display the data in real time as the user interacts with the compiled SCON OpenGL emulation code.

One form of data that renderdoc or Nvidia Nsight does not display in the OpenGL snapshot that is taken by them are the GLSL shader variables that the programmer has defined and utilised in the code of the shader programs to perform the desired functionality of the shader program. These GLSL shader program variable data types are emulated in the SCON OpenGL shader emulation by use of the C++ glm library of variable data types of the same name and functionality as that of the native GLSL variable data types. (see section **04:03**.) The C++ glm library data types are used in the same manner as that of any C++ or native GLSL variable data type.

These variables can be displayed in real time within the SCON C++ GLSL shader emulation just as any other C++ variable within any C++ application. To be queried and examined for the purpose of monitoring and debugging the values and behaviour of the variables within a GLSL shader program. Therefore the placement of the C++ code to display the GLSL emulated data variable is within the GLSL emulation code as indicated in **Fig 09.01**.

This is the main goal and purpose of the SCON C++ GLSL emulation toolkit as this cannot be performed in a native GLSL shader program, and even if it could, the examination of GLSL variables could only be done as a snapshot and not in real time. The SCON C++ GLSL emulation toolkit can perform this task in real time and quickly nail down any programming data, logical or calculation or other errors. This is why the SCON C++ GLSL shader toolkit was conceived and developed.

10 : SCON C++ GLSL Emulation Data Display

10.01 : internal GLSL and vertex attribute layout data

The SCON C++ GLSL Emulation stream data flowing between the emulated vertex and geometry shader programs and to the graphics rendering function in **Fig 09.01** are displayed in spreadsheet like tables that are defined utilising the dear ImGui toolkit.

The gl_in,gl_out and input and output layout follow a common property in that they are all vertex data related, and that the display of the data of each of these data streams would be of a common method, tools and appearance. Thus a base class can be deduced to be created to display each of these data streams.

The design of the display base class needs to be that not the entire data stream contents is displayed in such a spreadsheet like table as the number of vertices could be so large as the computer memory to perform the display task is insufficient causing the application to crash. Therefore the design of the display panel needs to be such that a portion of the data is displayed at a time, and that querying or scrolling the data is simple and straightforward. Secondly, all of the gl_in,gl_out and input and output layout for each GLSL emulated shader being emulated needs to be accessible from a single panel. Such a display panel design is displayed in **Fig 10.01**.

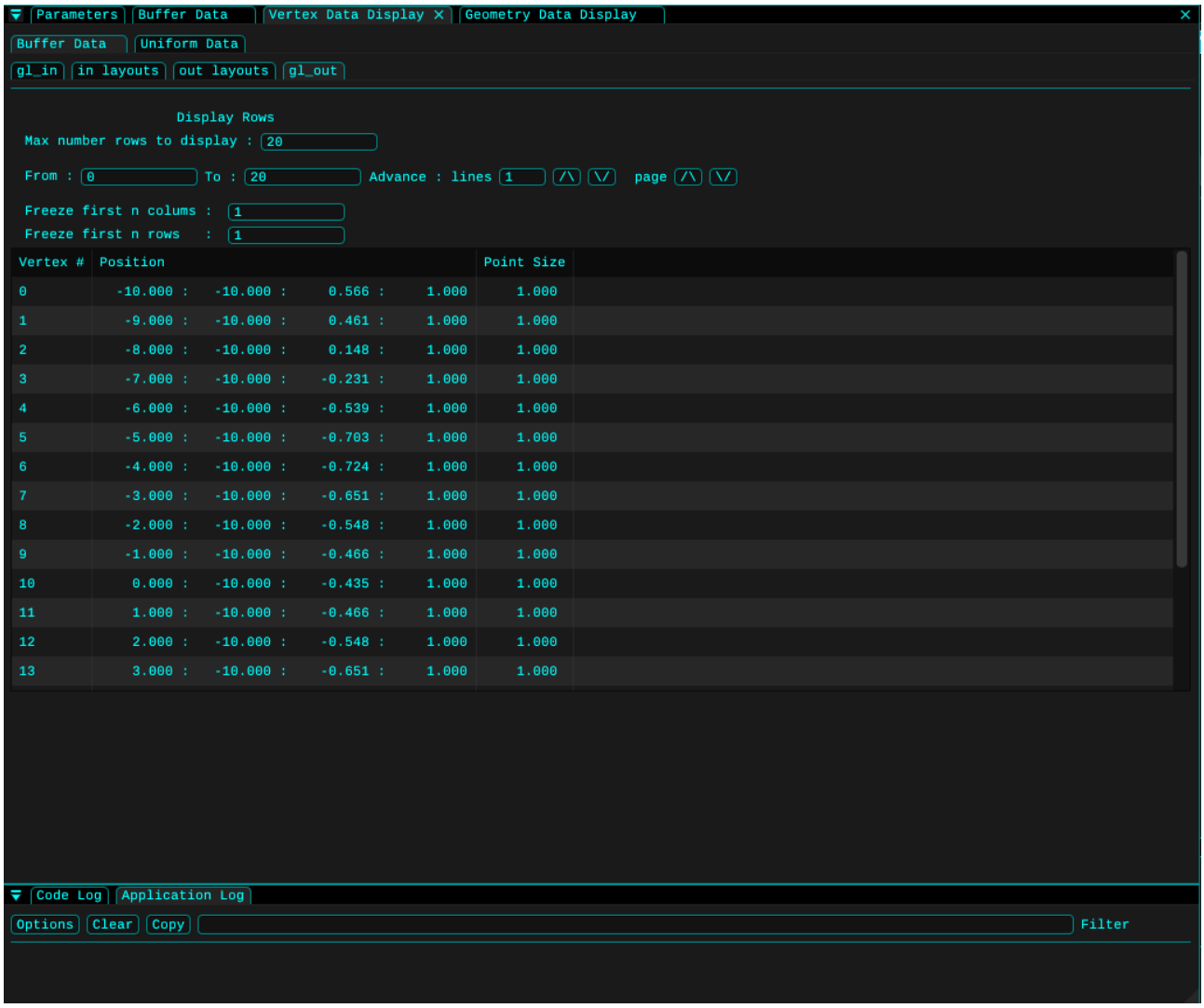


Fig 10.01

The design of the SCON C++ data display panel in **Fig 10.01** shows a list of the vertex shader gl_out data for the vertices of index 0 to 19. Various control inputs and interactions determine the number and index intervals of the vertex shader gl_out data that is to be displayed. The scroll bar on the right indicates that there is more data to be examined. A vertex index column gives index of the vertex data element that the row it sits on, and other column headers give the data type that is being displayed in each column. As expected for gl_out data, these are the position and point size for each vertex. This display is repeated for the gl_in, vertex attribute data input layouts and output layouts which are accessed by interacting with the the GUI tabs.

The SCON C++ GLSL Emulation uniform data display is displayed under the Uniform data tab in **Fig 10.01**, and if selected gives the GUI as in **Fig 10.02**. Activating the Buffer Data tag displays the vertex stream data GUI as in **Fig 10.01**.

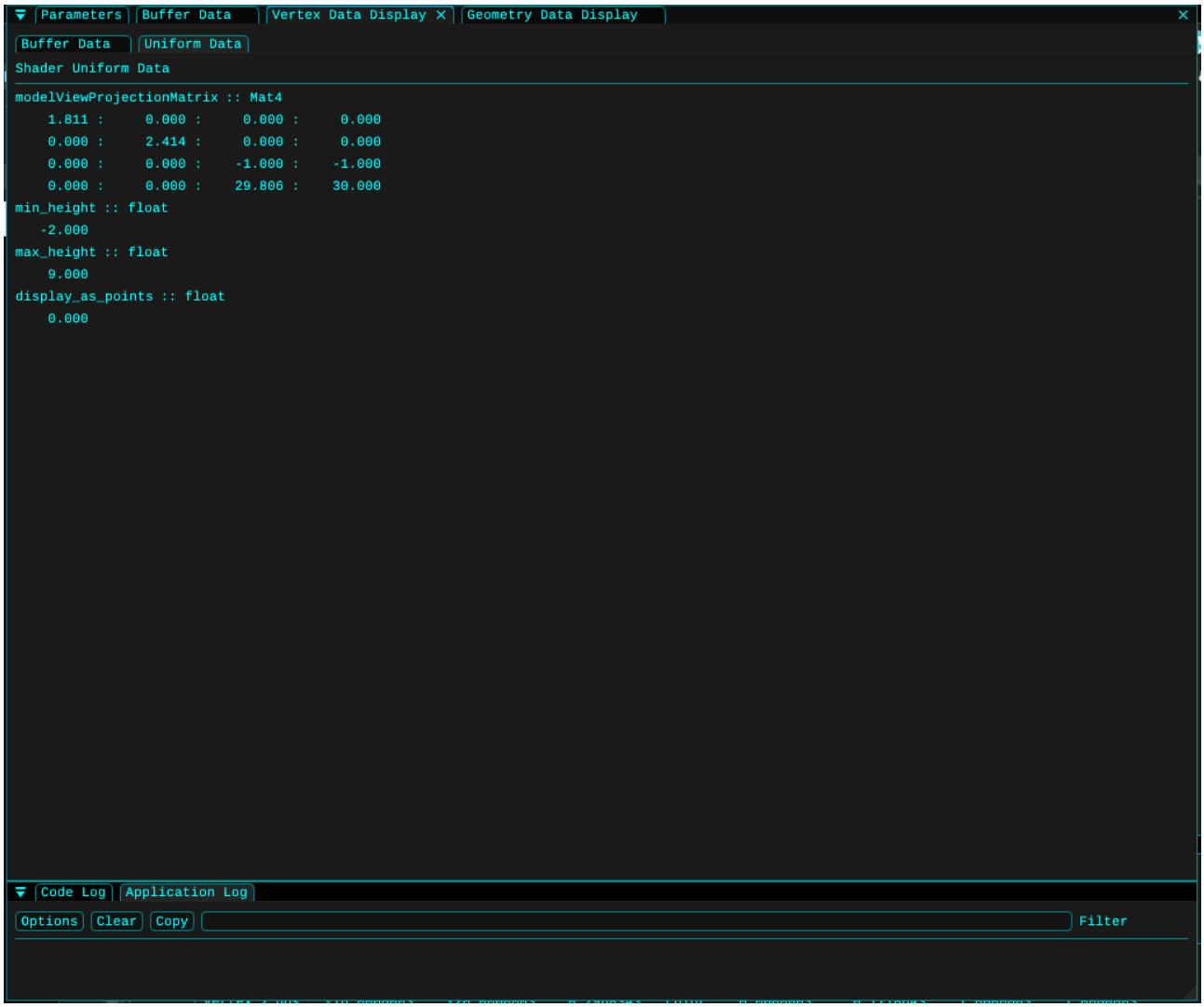


Fig 10.02

What can be seen in **Fig 10.02** are the emulated uniform data that has been defined for the emulated vertex shader to use. What are clearly displayed are the uniform names and data type, and then below this, the uniform data values.

The data displays given in **Fig 10.01** and **Fig 10.02** are repeated for the emulated geometry data which can be accessed by interacting with the Geometry Data Display tab. These data displayed by the C++ feature of inheriting a common base class that manages the storage and display of the above mentioned data into a vertex and geometry panel display class that manages the interaction and display of the tabs and widgets.

These display panel classes can then be defined as a class variable within an application prameter panel class as an ImGui tab item that calls a display function to display the emulated vertex and geometry shader data streams.

The basic concept of the C++ code design for the display of the OpenGL GLSL data streams and uniform data is illustrated in **Fig 10.03**.

gr

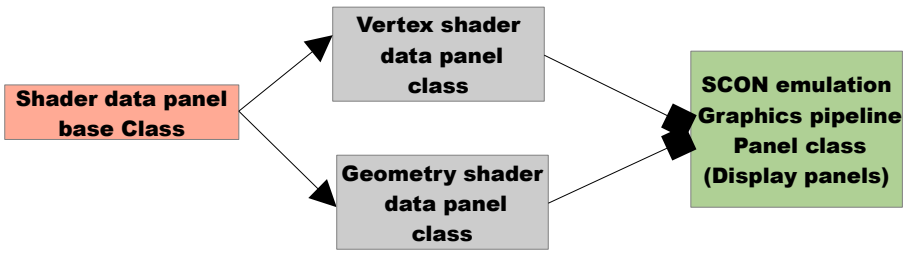


Fig 10.03 Schematic of the SCON shader data panel design

The C++ code for the shader data panel base class and vertex and geometry shader data panels exist within the directory path

FrameWork→SCONFW→Panels

with the respective file names

sfw_pipeline_panel.h
sfw_vertex_data_panel.h
sfw_geometry_data_panel.h

The code defining the SCON graphics pipeline emulation panel class exists in the directory

Source→SCON→Editor→Main_window→Panels

with the respective file name

parameter_panel.h

Note : The C++ panel classes require a pointer or reference to the gl_in, gl_out, input and output layout data structures to display them.

10.02 : Emulated GLSL variables

Displaying the emulated GLSL variables that are defined and used within the emulated shader code itself involves the user to write code within the emulated shader functions to display the emulated GLSL variables. Thus the method, technique and tools used to display emulated GLSL variables is up to the user. In the SCON toolkit, the dear ImGui library of functions is the main tool used to do this.

The GLSL variables can be split into three broad categories.

S10.02-i : Variables that do not change per vertex data point and are constant through out the entire execution of the GLSL pipeline cycle.These variables can be considered as being the same as C++ constants that do not change in the GLSL emulated program.

S10.02-ii : Variables that do not change per vertex data point, but do change for each execution of the GLSL pipeline cycle.These variables can be considered as being the same C++ variables that are used to calculate and store values to be used in further calculations, or to be used in logical comparison or other statements that effect the flow or result of the execution of the program.

S10.02-iii: Variables that do change per vertex data point for each execution of the GLSL pipeline cycle.These variables can be considered as being variables that are different for each vertex in each and every GLSL pipeline cycle.

A GLSL pipeline cycle is the cycle of execution of code of all vertex data points. This can be referred to as an execution of a frame of a graphics rendering process.

S10.02-i type variables can be displayed in the context that they do not change, and can be displayed in a manner like any data parameter within the execution of the GLSL emulation code.

S10.02-ii type variables change per cycle or frame of the GLSL emulation code. Such variables thus will change from one frame to the next of the OpenGL GLSL pipeline cycle. What is fortunate is that for most GLSL programs and conditions, if the user does not interact with the OpenGL GLSL program and change the conditions of the graphics that is being displayed, these variables do not change and can be displayed as a constant unchanging value. This is unless some outside influence such as an external uniform variable that involves a time stamp changes the variable value. Therefore, the display of **S10.02-ii** variables can be performed in a similar manner to **S10.02-i** variables.

S10.02-iii type variables change per vertex for each cycle or frame of the GLSL emulation code. What this means is that these variables need to be displayed per emulated GLSL pipeline cycle in a manner that specifies the index of the vertices that are to be displayed. Thus one method is to display such variables in a table similar to that of the gl_in,gl_out and the in and out layout data streams. This is indeed a method used in the SCON emulation toolkit template example code.

An example of emulated GLSL variable display for **S10.02-iii** type variables is illustrated in **FIG 10.04**.

Because the GLSL variables are a C++ emulation of the GLSL code, this has the added functionality open to the user of all the normal C++ debugging tools and techniques open to them to use to debug and correct programming errors. This is the main purpose of the SCON C++ OpenGL GLSL emulation toolkit. By correctly introducing the code to display and query the emulated code variable data, an easier path to develop OpenGL GLSL shader programs is made available.

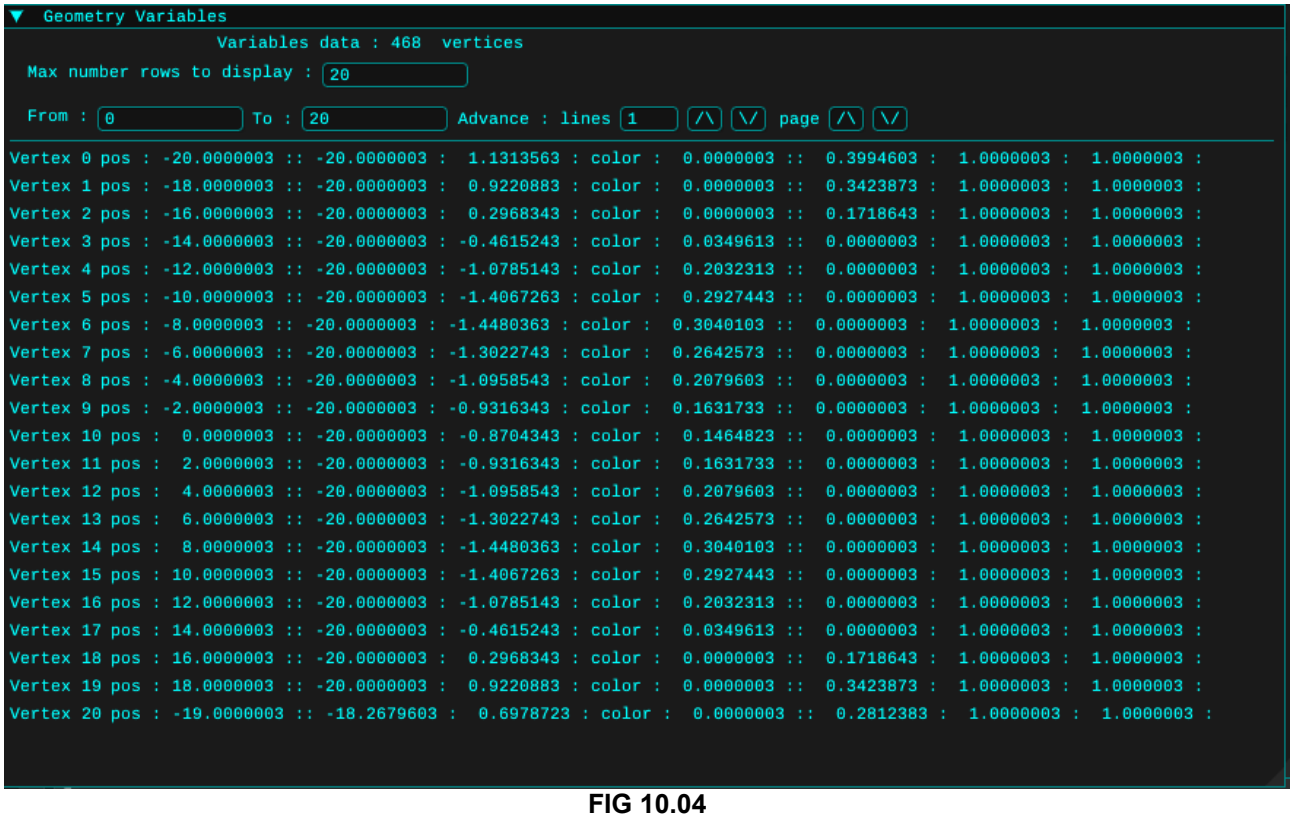

Geometry Variables
Variables data : 468 vertices
Max number rows to display : 20
From : 0 To : 20 Advance : lines 1 /\ \/ page /\ \/
Vertex 0 pos : -20.0000003 :: -20.0000003 : 1.1313563 : color : 0.0000003 :: 0.3994603 : 1.0000003 : 1.0000003 :
Vertex 1 pos : -18.0000003 :: -20.0000003 : 0.9220883 : color : 0.0000003 :: 0.3423873 : 1.0000003 : 1.0000003 :
Vertex 2 pos : -16.0000003 :: -20.0000003 : 0.2968343 : color : 0.0000003 :: 0.1718643 : 1.0000003 : 1.0000003 :
Vertex 3 pos : -14.0000003 :: -20.0000003 : -0.4615243 : color : 0.0349613 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 4 pos : -12.0000003 :: -20.0000003 : -1.0785143 : color : 0.2032313 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 5 pos : -10.0000003 :: -20.0000003 : -1.4067263 : color : 0.2927443 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 6 pos : -8.0000003 :: -20.0000003 : -1.4480363 : color : 0.3040103 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 7 pos : -6.0000003 :: -20.0000003 : -1.3022743 : color : 0.2642573 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 8 pos : -4.0000003 :: -20.0000003 : -1.0958543 : color : 0.2079603 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 9 pos : -2.0000003 :: -20.0000003 : -0.9316343 : color : 0.1631733 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 10 pos : 0.0000003 :: -20.0000003 : -0.8704343 : color : 0.1464823 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 11 pos : 2.0000003 :: -20.0000003 : -0.9316343 : color : 0.1631733 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 12 pos : 4.0000003 :: -20.0000003 : -1.0958543 : color : 0.2079603 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 13 pos : 6.0000003 :: -20.0000003 : -1.3022743 : color : 0.2642573 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 14 pos : 8.0000003 :: -20.0000003 : -1.4480363 : color : 0.3040103 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 15 pos : 10.0000003 :: -20.0000003 : -1.4067263 : color : 0.2927443 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 16 pos : 12.0000003 :: -20.0000003 : -1.0785143 : color : 0.2032313 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 17 pos : 14.0000003 :: -20.0000003 : -0.4615243 : color : 0.0349613 :: 0.0000003 : 1.0000003 : 1.0000003 :
Vertex 18 pos : 16.0000003 :: -20.0000003 : 0.2968343 : color : 0.0000003 :: 0.1718643 : 1.0000003 : 1.0000003 :
Vertex 19 pos : 18.0000003 :: -20.0000003 : 0.9220883 : color : 0.0000003 :: 0.3423873 : 1.0000003 : 1.0000003 :
Vertex 20 pos : -19.0000003 :: -18.2679603 : 0.6978723 : color : 0.0000003 :: 0.2812383 : 1.0000003 : 1.0000003 :

FIG 10.04

11 : Putting It All Together

All of the elements for the SCON C++ GLSL pipeline emulation toolkit has been described in the sections 01 to 10 above. How the SCON toolkit is all put together is illustrated in **Fig 11.01**

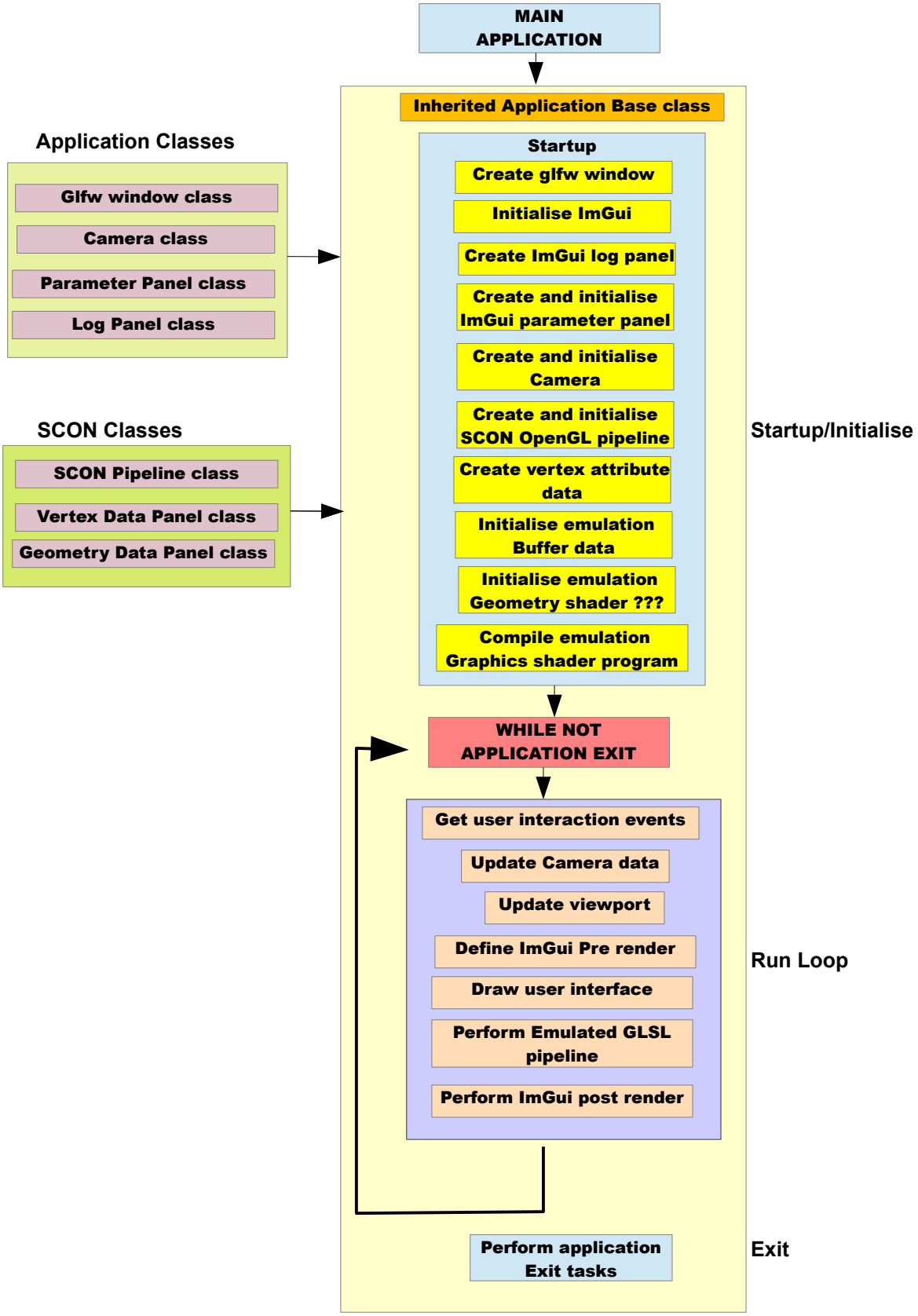


Fig 11.01 Schematic of Application main function

The schematic in **Fig 11.01** is a representation of the execution of the C++ code in the entry point of the SCON application toolkit.

In the startup/initialisation:

A common set of tasks to initialise the application window, ImGui, parameter panel and camera are performed.

A defined scon pipeline class is initialised in Create and initialise SCON OpenGL pipeline. A pointer to the already defined application camera is assigned to this pipeline class.

The Create vertex attribute data is flexible and dependent upon how the user wishes to create and define the vertex attribute data to be used. Such data can be loaded from a file, generated by a function, or defined as a structure or array that the user has hard code within the application.

In the run loop that is executed until the user exists the application :

Draw user interface is the function that displays the GUI panel that is illustrated in **Fig 10.01** and **Fig 10.02**. The GLSL emulation data streams for the vertex and geomerty shader used that is to be displayed in the GUI panel needs to have a pointer to the emulated pipeline gl_in, gl_out, input and output layout data structures defined.

A camera GUI panel also is displayed where the camera parameters such as field of view etc can be changed. This panel requires a pointer to the camera class to modify is parameters.

Perform Emulated GLSL pipeline is the execution of the C++ OpenGL GLSL emulation code that is discussed in sections **04** to **10**.

Perform application exit tasks :
When the user exits the application run loop, the application close function preforms all the tasks of closing and cleaning up the application such as freeing up memory used, ending ImGui, closing the glfw window and terminating glfw etc.

All the tasks that is featured within the main yellow box of **Fig 11.01** exists within a derived C++ main application class called shaderCON_class.

The main entry point C++ file to define and call this shaderCON_class can be found in the directory

Source→SCON→Templates

within the main function of the file of name

shaderCON_main.cpp

12 : SCON Templates

In the directory path

FrameWork→SCONFW→Templates

of the SCON project exists a series of example C++ template files to create and display 3D graphics of various types of OpenGL primitive data types for both veterx shader only and vertex + geometry shader C++ emulation.

These templates are named to reflect what kind of OpenGL primitive data type they produce as an end product from an input of vertex point cloud data.

A name of the format

sfw_vertex_shader_primitive_datatype_template.h

is a template of an emulated vertex shader that involves an OpenGL of a primitive datatype.

sfw_geometry_shader_primitive_datatype_template.h

is a template of emulated vertex and geometry shaders that involves an OpenGL of a primitive datatype that is output from the emulated geometry shader.

There is a general OpenGL pipeline emulation template named

sfw_pipeline_template.h

from which any OpenGL pipeline emulation can be formed or created from.

These all can be copied and modified to suit whatever GLSL program one needs to develop and debug before translation of emulation code to a true OpenGL GLSL program code for an application.

The files named as sfw_grid_pipeline.h and sfw_triangles_pipeline.h are such copies and modification of the sfw_pipeline_template.h file as time of writing of this document that can be used as examples and to get a better grip on how to create a SCON C++ OpenGL GLSL emulation is constructed. The user can examine the coding of these templates and pipeline example code in conjunction with the documentation provided in the sections above so as to create their own OpenGL GLSL emulation code to compile and execute.

The copied and modified vertex and geometry shader emulation classes need to have their C++ header files included in the copied and modified C++ pipeline file that they are to be used in. (See section **6 Fig 6.02**) The pipeline code needs to be modified to accommodate use these shader emulation classes.

The copied and modified C++ pipeline emulation class file then needs to be included in the main SCON C++ application file shaderCON.cpp, or an equivalent such as a copy and modification of it. (see section 1**1 Fig 11.01**) The code needs to have the appropriate modifications made to accommodate this pipeline class.

With all this done, the SCON C++ OpenGL GLSL emulation code should be able to be compiled and executed to display the GLSL graphics and debug GLSL logical and computational errors.

13: Basic Shader Constructor Coding

The description given here should be used in conjunction with the templates described in section 12.

The basic structure of the code within any SCON C++ OpenGL GLSL shader emulation pipeline class should follow the outline as illustrated in **Fig 13.01**.

```
update camera view port projection uniform

perform vertex shader emulation

If geometry shader is used
{
    define emulated geometry shader input as the vertex shader output
    define  emulated geometry shader input layout data as the vertex shader layout output data

    perform geometry shader emulation
    define output data to render as the output data of the geometry shader emulation
}
else
{
     define output data to render as the output data of the vertex shader emulation
}

define OpenGL buffers to render shader emulation output  data

render geometry of shader emulation output  data

delete OpenGL buffers to render shader emulation output  data
```

Fig 13.01

The basic structure of the code within any GLSL shader is as illustrated in **Fig 13.02**.

```
define glsl version to use

input vertex attribute layouts
output vertex attribute layouts

uniforms

global variables

support functions

main function
```

Fig 13.02

The structure C++ GLSL emulation code within the emulated GLSL shader class follows follows the same structure as that of a true GLSL shader as outlined in **Fig 13.02**.

In all GLSL shaders, the entry point of execution of the shader code is a function of the name main.

In a C++ SCON GLSL shader emulation class, the above structure is followed with the exception that the entry function to all the emulated shader is a virtual function called scon_main. This is because unlike a true GLSL shader program that accepts and processes data in parrallel, the C++ emulation does not and needs to specificly a call the emulation of the emulated main function for each vertex attribute data one at a time. This emulated main function to process vertex attribute data is a virtual function of name scon_main_code.

The basic general structure of the scon_main function is illustrated in **Fig 13.03**.

```
void scon_main() override {
get number of vertices

clear input data buffer structures of data
clear output data buffer structures of data
```

```
for each input data element {
    perform scon_main_code
    add result(s) from scon_main_code execution to output buffer;
}

create_output data for next shader or for rendering

}
```

Fig 13.03

The exact code that is used depends upon the input and output data primitive type that the shader emulation uses. This function is called from the C++ SCON pipeline emulation class to execute the emulated shader program in its pipeline sequence.

The GLSL input vertex data layout has a general form of

layout (location = <location number >) in <data type> <layout name>;

and output vertex data layout of a general form of

layout out <data type> <layout name>;

The C++ emulation of an input vertex attribute datalayout is of the form as given in section **04:04** as

<vlayout data type name> <layout name> = <vlayout data type name>(<location number >,<stream type>,<data type>,"<layout name>"); (**E-13:01**)

where <stream type> is one of shader_stream_type_enum::in or shader_stream_type_enum::out.

A geometry shader has an input layout defined for the data input gl_in as being the gl_out from a previous shader output in the GLSL pipeline in the form of

layout(<primitive data type>) in

The C++ SCON GLSL emulation of this layout declaration is given, as explained in section **04:04** in the form

glayout_in <layout name> = glayout_in(<primitive data type>)(**E-13:02**)

where <primitive data type> is one of the primitive data types that exist within the enumerator class geo_layout_data_in_enum.

A geometry shader has an output layout defined for the data output gl_out in the form of

layout(<primitive data type>, max_vertices = <max number of pimitive data vertices>) out

The C++ SCON GLSL emulation of this layout declaration is given, as explained in section **04:04** in the form

glayout_out <layout name> = glayout_in(<primitive data type>,<max number of pimitive data vertices>) (**E-13:03**)

where <primitive data type> is one of the primitive data types that exist within the enumerator class geo_layout_data_out_enum.

When composing an emulation of a GLSL layout for the vertex or geometry shader, then the appropriate emulation code as specified above in (**E-13:01-E-13:03**) is defined within the shader emulation class.

The GLSL shader uniform data has a general form of

uniform <data type> name

The equivalent C++ SCON GLSL emulation of any uniform declaration is given, as explained in section **04:06** in the form of

<Uniform data type> <name> = <Uniform data type>(<data type flag>,"<name>"); **E-13:04**

When composing an emulation of a GLSL uniform for the vertex or geometry shader, then the appropriate emulation code as specified in **E-13:04** is defined within the shader emulation class.

When composing an emulation of a GLSL global, or other type of variable top be used in any emulated shader program, the appropriate C++ or C++ glm library data type that is equivalent to the GLSL data type can be defined and used within the C++ emulated shader class.

Similarly, function declarations and calls are more or less identical in the C++ SCON GLSL emulation to that of any GLSL shader function.

By the correct reference to the layout and gl_in and gl_out layout data specified by **E-13:01**, **E-13:02**,

E-13:03, and **E-13:04**, the emulation statements of reference, comparison, and calculation etc are very close to being identical. Geometry shader declarations of EmitVertex and EndPrimitive perform the same function in the emulation as in a true GLSL geometry shader program which the user need not to be concerned about the background code of these functions. Just they do what they are supposed to do.

Because the C++ SCON GLSL emulation is C++ based, additional non GLSL code to perform the debugging and monitoring of variables, uniforms, and vertex attribute layout data or other aspects of the emulated shader code can be performed as the user wishes. Within the C++ OpenGL pipeline and shader class templates are examples and techniques of how this is done.

Therefore by looking at the template code as an example and basis in conjunction with this documentation, the user should have a basis from which to create their own C++ OpenGL GLSL shader pipeline emulation.

14: known emulation issues

As of time of of writing this documentation, two major issues are known to exists with the SCON C++ OpenGL GLSL emulation toolkit.

1 : Quaternian gimbal lock :

The present C++ OpenGL GLSL emulation code using the C++ glm library quaternion functions to perform rotations of the camera about a central point in 3D space experiences what looks to be a gimbal lock when undergoing a pitch rotation. When the camera is rotated at a ± 90º angle, the central looks like it is performing wild random rotations in an uncontrolled manner. In fact, it is the camera location in 3D space that is going wild and is very sensitive to the user interaction with the application.

This gimbal lock does not occur in the equivalent GLSL code that the templates described in section 12 are translated into. So this looks to be a coding error within the core C++ application code, the incorrect use of the glm library quaternion functions in the existing GLSL emulation, or an unlikely bug in the glm library function.

Whatever the cause, as of the writing of this document, this problem exists and has not been solved. A future version of the SCON GLSL emulation toolkit should resolve this issue.

2 : Inconsistent float variable behaviour between C++ and OpenGL GLSL code :

The development of a glsl program to display a 3D coordinate reference grid of the x,y,z axis within a 3D scene proved to be problematic when the emulation code was translated into GLSL form. The problem was that not all coordinate reference grid lines was being displayed.

After about two hours spent trying to debug the glsl code in a glsl geometry shader program, the reason was discovered was not to do with the glsl code, but to do with how a float number in the glsl program was stored when dealing with a float numbers being stored as an integer vlaue to or cast to an integer value.

It was through some debugging methods to narrow the range of the float numbers being stored that in some instances the float representation of an integer value was just above that integer value, and that for other instances it was just below the integer value. What this did was throw off any glsl floor and ceiling functions to be calculated incorrectly and to correspond to the C++ emulation code. In C++ a float variable representing an integer value to perform evaluations with other float variables or values is consistently slightly larger than a true integer value. Thus to get a true integer value from a float to perform evaluations, a floor function is performed.

In GLSL, a representation of an integer value as a float value being slighly larger than the true integer value is not guaranteed. Thus an adjustment in the GLSL code was needed to be made to add a value to the float variable before any floor or other such function was applied to get a true integer value to use.

Not sure if this is a GPU hardware memory issue, or an OpenG: GLSL driver implementation issue. What this means is that the user needs to be aware that in certain circumstances, the C++ OpenGL GLSL emulation software solution may not be a 100% correlation with it functioning in a true OpenGL GLSL program.

3 : Inconsistent float number precision between C++ and OpenGL GLSL

While performing the debugging of GLSL code as described in issue 2 above, a disturbing discovery was detected in the inconsistency of floating point number precision between a C++ float type variable number and that of an OpenGL GLSL float type variable number.

From the deduction of debugging the mentioned GLSL code, it was found that the precision of the representation of an integer value as a float varied and was inconsistent. In C++ a float has a consistent precision value of about 15 decimal places, but in GLSL this was not so and only a precision of up to 5 decimal places could be considered as reliable enough to use for the GLSL 3D coordinate reference grid function being used. Compensation for this low precision was needed to be incorporated into the GLSL program to have it function as desired.

This can cause immense problems in OpenGL GLSL programs that require accurate high precission float numbers to perform evaluations and calculations. What this also means is that there is a potential for the SCON C++ OpenGL GLSL emulation to be be different to that of a compiled OpenGL GLSL program, and for the user to need to be aware of.

Again, it is not sure if this is a GPU hardware memory issue, or an OpenG: GLSL driver implementation issue, or a Kronos standardisation issue.