Virtual Surface Light

Application Development Framework

VSL Version 9.06

Gareth Edwards

25 July 2023

1 Introduction 6
1.1.1 Preface 6
1.1.2 Abbreviations 6
1.1.3 Acronyms 6
1.1.4 Contents .... **Error! Bookmark not defined.**
1.1.5 What is VSL? 7
1.1.6 Why use VSL? 7
1.1.7 Framework 7
1.1.8 Components 8
1.1.9 Graphic Elements 8
1.1.10 Graphics Classes 9
1.1.11 Framesets 9
1.1.12 Origins & Legacy 9
1.1.13 VS Lite 9
1.1.14 VSL 7 & 8 9
1.1.15 VSL 9 9
1.2 Application Components 11
1.3 Application Real-Time 14
1.4 Application Layers 15
1.5 Development .... Error! Bookmark not defined.
1.5.1 VSL Codebase 16
1.5.2 Windows Component 16
1.5.3 Interface Component 17
1.5.4 Application Component 17
1.5.5 Selecting an Application Component 17
1.5.6 Versioning and Setup 17
1.5.7 Installation 19
1.5.8 Run-time 20
1.5.9 Application Direct3D Device Reset 20
1.6 PImpl Design Pattern 21
1.7 MVC Design Pattern 22
1.7.1 Photographer MVC Design Pattern 22
1.7.2 The MVC Design Pattern 23
1.7.3 VSL Framework usage of the MVC Design Patterns 24
1.7.4 Previous VSL versions usage of MVC 25
1.8 Direct3D 26
2 Windows Classes 27
2.1 Win_Create 27
2.2 Win_Engine 29
3 Graphics Classes 31
3.1 Command 31
3.2 D3dx 32
3.3 Element 33
3.4 Element Engine 35
3.5 Element Configure 37
3.6 Element Coordinate 38
3.7 Element Component 39
3.8 Frameset 40
3.9 Log 41
4 Application Classes 42
4.1 Overview 42
4.2 Private Implementation (PImpl) 43

4.2.1 Definition ... 44
4.2.2 Deleter ... 45
4.2.3 Get ... 46
4.3 Base Class Definition ... 47
4.4 Constructor & Setup ... 49
4.5 Framework Methods ... 50
4.5.1 Setup ... 50
4.5.2 Display ... 52
4.5.3 Cleanup ... 53
4.5.4 Get ... 54
4.5.5 Set ... 55
4.6 Virtual Methods ... 56
4.6.1 Initialise ... 56
4.6.1.1 Frameset ... 56
4.6.1.2 Project ... 57
4.6.1.3 Configurations ... 58
4.6.1.4 Coordinates ... 58
4.6.1.5 Components ... 59
4.6.2 Display ... 60
4.6.2.1 In Client Adjusted Viewport ... 60

5 Application Framework ... 61

5.1 Framework Components ... 61
5.2 Diagram: Framework Components ... 63
5.3 Windows Component Functions ... 64
5.3.1 Win_Main ... 64
5.3.2 Win_Procedure ... 65
5.3.3 Win_Engine ... 66
5.4 Application Interface Functions ... 70
5.5 Application Component Classes ... 71
5.5.1 Stand-Alone Class Definition ... 72
5.5.2 Base Class Definition ... 73
5.5.3 Example Derived Class Definition ... 75
5.5.4 Selecting an Application Class ... 76
5.6 Diagram: Application Component Class ... 78
5.7 Diagram: Application Component Class Display ... 79

6 Base Class ... 80

6.1 Overview ... 80
6.2 Private Implementation (PImpl) ... 81
6.2.1 Definition ... 82
6.2.2 Deleter ... 83
6.3 Constructor & Setup ... 84
6.4 Framework Methods ... 85
6.4.1 Setup ... 85
6.4.2 Display ... 87
6.4.3 Cleanup ... 88
6.4.4 Get ... 89
6.4.5 Set ... 90
6.5 Virtual Methods ... 91
6.5.1 Initialise ... 91
6.5.1.1 Frameset ... 91
6.5.1.2 Project ... 92
6.5.1.3 Configurations ... 93
6.5.1.4 Coordinates ... 93
6.5.1.5 Components ... 94
6.5.2 Display ... 95

6.5.2.1 In Client Adjusted Viewport ... 95
6.5.2.2 In A Frameset ... 96

7 Graphic Frameset ... 97

8 Graphic Elements ... 99

8.1 Gfx_Element Class ... 99
8.2 Private Implementation ... 99
8.3 Configuration ... 100
8.4 Coordinate ... 100
8.5 Component ... 101
8.5.1 Gfx_Element ... 101
8.5.2 Private Implementation ... 101
8.5.3 Kandinsky Interface ... 101
8.5.4 Kandinsky Bitmasks ... 101
8.5.5 Kandinsky Life-Cycle ... 102
8.5.5.1 Setup ... 102
8.5.5.2 Display ... 103
8.6 Kandinsky ... 104
8.6.2 Decouple ... 104
8.6.3 Source Files ... 104
8.6.4 Shape Files ... 105
8.6.5 Shape File Naming ... 105
8.6.6 Adding a Kandinsky Shape ... 106
8.6.6.1 Create [name] shape file. ... 106
8.6.6.2 Add [name] shape file to shape file list. ... 106
8.6.6.3 Add declarations for [name] shape methods. Directory: "vsl_library/header/". ... 106
8.6.6.4 Enumerate Gfx_Kandinsky_Component [name] shape. ... 106
8.6.6.5 Add Gfx_Kandinsky_Interface [name] shape methods. ... 107
8.6.6.6 Add Gfx_Kandinsky_Component [name] to interface. ... 107
8.6.6.7 Add [name] shape Config & Create method declarations. ... 108
8.6.6.8 Add GetCallbacks for Parameters & Component ... 108
8.6.6.9 Summary ... 109
8.7 Gfx_Kandinsky ... 110
8.8 Gfx_Element_Engine ... 110
8.8.1 Setup ... 110
8.8.1.1 Validate Element Component ID ... 110
8.8.1.2 Append Element Configuration & Coordinate Parameters ... 110
8.8.1.3 Add Element Component Parameters ... 111
8.8.1.4 Summary: ... 112
8.8.1.5 Example: ... 113
8.9 Gfx Diagrams & Flow Charts ... 114
8.9.1 Gfx_Element_Engine ... 114
8.9.2 Gfx_Element ... 115
8.9.3 Gfx_Element_Engine::Setup ... 116
8.9.4 Gfx_Element_Engine::Element_Setup ... 117
8.9.5 Gfx_Element_Engine::Element_Display ... 118
8.9.6 Gfx_Element_Engine & PImpl ... 119
8.9.7 Gfx_Element & PImpl ... 120
8.9.8 Gfx_Element_Configure & PImpl ... 121
8.9.9 Gfx_Kandinsky_Interface & Pimpl ... 122
8.9.10 Gfx_Kandinsky & Pimpl ... 123

9 Appendices ... 124

9.1 Frameset ... Error! Bookmark not defined.
9.2 Error Handling ... 124
9.3 Folder Structure & Namespaces ... 125
9.4 VSL C++ Style Guide ... 126

1 Introduction

1.1.1 Preface

This document is Visual Surface Light (VSL) version 9 manual and technical reference.

It is comprised of 9 chapters.

These are:

- **Chapter 1:** Introduction
- **Chapters 2 to 4**: the Windows, Graphics and Application C++ classes.
- **Chapters 5 and 6**: Describe and details the structure, design, and usage of the VSL Application Framework and Application classes.
- **Chapters 7 and 8**: Describe and details the VSL Graphical system.
- **Chapter 9**: Appendices.

I decided to provide a complete listing (with commentary) of the VSL building blocks, the C++ classes, in Chapters 2 to 4, before an explanation of what VSL Applications are, in Chapters 5 and 6, as this required a complete familiarity with the former.

The above explanatory narrative did not require a detailed description of the VSL graphics system.

This is provided in Chapter 7 and 8, with a comprehensive description of the Gfx_Element, the most important graphical C++ class, its composition, and functionality, and how it is displayed using the most complex of the VSL classes, the Gfx_Engine class, in all or part of an applications desktop window.

Gareth Edwards

1.1.2 Abbreviations

WndCom	Windows Component
AppInt	Application Interface Component
AppCom	Application Components

1.1.3 Acronyms

ADF	Application Development Framework
FSF	Free-Standing Functions
GUI	Graphical User Interface
MSVC	Microsoft Visual C++
MVC	Model, Control and View (design pattern)
VSL	Virtual Surface Light

1.1.4 What is VSL?

VSL is an Application Development Framework (ADF), one that is specifically designed for use with Microsoft Visual Studio, and with Microsoft Visual C++ (MSVC), which is a compiler for the C, C++, C++/CLI and C++/CX programming languages by Microsoft. It is a C++ high code development system. Examples, and demo source code, is provided.

The name VSL is an acronym for Virtual Surface Light. Originally it was named VS, but this was often confused with (Microsoft's) Visual Studio, and much else. The additional 'L' was chosen as VS includes an Image Rendering, or Image Synthesis, system. This is the process of generating 2D and 3D scenes. To make 3D scenes look realistic required **Virtual Surface Light**ing, and that the rendering software must solve the illumination rendering equation.

1.1.5 Why use VSL?

Because it provides a substantial system of proven, considered, tried, and tested functionality.

This greatly reduces the time and energy required to develop PC applications, especially those that require real-time interactive 2D & 3D graphics, and which are fully integrated with digital image acquisition, processing, and archiving.

The VSL design is a product of over 40 years of commercial development (in Fortran, C, C++, Java, and Python) and usage, by the author of this document.

1.1.6 Framework

The VSL Framework is comprised of three principal Components (see section below on Components): Windows Component (WndCom), Application Interface Component (AppInt), and Application Component (AppCom).

The WndCom manages the lifecycle of an application via the AppInt. It provides for the creation of a window, then the setup, real-time management, and exit from an application.

The AppInt is functionally discrete, but organisationally an integral part of the WndCom and is the interface between it and an application. As such the WndCom and the AppInt together could be considered as a single component. However, referring to them as separate has proven to be conceptually clearer, and practically more useful.

All AppCom's must possess ten identical functional equivalent, C++ methods, which fall into three groups: the "Get", "Set" and "Interface" groups.

Two "Get" methods return pointers to the C++ Gfx_Create and Gfx_Engine structs: the first stores information required to create and configure a Microsoft Window; the second stores information required to configure the real-time Microsoft Windows real-time engine. The WndCom stores these pointers for later access and update. These are initialised at runtime when an instance of an AppCom is created. The structs are properties of an AppCom's private implementation and can therefore be accessed by every method of an AppCom.

Three "Set" methods are used by the WndCom for real-time interactive control of an AppCom.

Five "Interface" methods, provide:

1. One-time setup (when an application is started),
2. Setup when an application has lost the Direct3D device OR when the window is resized,
3. Display
4. Cleanup when an application has lost the Direct3D device OR when the window is resized,
5. One-time cleanup (when an application is exited).

It is this AppCom common functionality that provides for the interchangeability of AppCom's.

There are two types of AppCom: a stand-alone application class, and an application base class (and derived application classes).

The stand-alone application class is not currently provided with VSL graphics. This class requires that each stand-alone application provides for all low-level graphical and housekeeping needs, including graphical shape creation, display, and the management of graphical storage and object lifecycle e. For uncomplicated applications, this “re-inventing the wheel” is not an onerous burden, but is inefficient, time-consuming, and prone to error.

1.1.7 Components

As mentioned above, the VSL Framework is comprised of three Components.

Software architecture and design nomenclature is often contentious. The following summary has been used as a guide, with the key phrase bold highlighted.

> *“In terms of granularity the Component sits between the Module and the Object. The purpose of a component is to put together a collection of general-purpose objects to form a purpose specific unit. As the name implies, unlike the Module,* ***the Component is not “self-contained”, it is a part of a larger functional whole****.”*
>
> *“This type of Software building is known as component-based software. In this, there are different components of software and each of the components represents a modular, easy to debug and fix, and replaceable part of a system that wraps up and represents a set of interfaces.”*
>
> *Keith S, Stack Exchange (Dec 10, 2012)*

1.1.8 Graphic Elements

The base and derived class applications hierarchically combine VSL graphics elements (Gfx_Element class) to create one or more data sets.

Each graphical element has an id, a name, and a value; and can be hierarchically grouped, instanced, configured, located, and associated with instances of 2D or 3D graphical shapes.

Why shape, not “object” or “component”?

> Firstly, these two last terms can be confused with software terms.
>
> Secondly:
>
> *A* ***shape*** *or* ***figure*** *is a graphical representation of an object or its external boundary, outline, or external surface, as opposed to other properties such as colour, texture, or material type.*
>
> *https://en.wikipedia.org/wiki/Shape*

To complicate matters, many technology providers such as Microsoft, and many Authors, refer to 2D or 3D data that is stored on a GPU as being an “object”. Where appropriate this document will refer to such GPU data as such.

To provide for real-time, efficient, interactive, or parameterised change, each graphical element can be “bookmarked” to provide for efficient access and update.

When either a new stand-alone application, or a new derived class application, is required, an existing project or demo class that is a “best fit”, can be copied, renamed, and added to the VSL AppCom folder structure.

1.1.9 Graphics Classes

To manage and display graphics elements, VSL provides utility & housekeeping classes.

For example, when a graphics element data set is to be displayed, it is passed to the graphics element engine (Gfx_Element_Engine class), which handles (as required) graphical element configuration (Gfx_Element_Configure class), the initialisation of an element's coordinates (Gfx_Element_Coordinate class), components (Gfx_Element_Component class), associated shape creation, update, and display.

The Gfx_Element_Engine class, as detailed later in this document, is the most complex, substantial, and important of the VSL graphics classes.

1.1.10 Framesets

Graphics elements can either be displayed within the WndCom displayed portion of a window, the Client Adjusted Viewport(CAV), or within a frame (Gfx_Frame, which can be combined and controlled hierarchically) that is part of a frameset (Gfx_Frameset class) within the CAV.

Each graphics frame is comprised of three concentric parts: Margin, Border, and Padding, which wrap around the content within a Frame. This convention is derived from the CSS (Cascading Style Sheet) box model, which is essentially a box that wraps around every HTML element. The properties (e.g., size, location, visibility, colour) of graphics frames can be updated in real-time.

1.1.11 Origins & Legacy

Much of the conceptual design of the newest version of VSL has been inherited from previous versions of VSL, especially the most recent (VSL 7 and VSL 8), which have been proven over the last two decades to be functionally fit for purpose, and operationally efficient, extendable, manageable, and stable.

1.1.12 VS Lite

The inspiration for the latest version of VSL, is a C++ 3D graphics Application Development Framework (ADF) named VS Lite. This was designed specifically for use by Students and Interns: all functionality and usage designed to be self-explicable, clear in purpose, and provide satisfying achievable visual results during an assignment or project development.

From 2015 onwards, after a decade of proven use, the robust core of the VS Lite replaced the functionally equivalent VSL 7 logic. This was done to take advantage of its superior performance and implementation clarity. An additional benefit was the adoption of the VS Lite “plug-and-chug” architecture, providing for an exceptionally clear separation between the WndCom and an AppCom class, using an intermediate AppInt class.

1.1.13 VSL 7 & 8

In 2020 it became clear that VSL 7 required a substantial refresh: the code base had become bloated with old projects and outdated operational requirements, some parts of which required several days to re-acquire context, and understand implementation logic, before these could be modified or updated.

To maintain support for the then current and legacy supported projects, VSL 7 was frozen. Then a copy was created, from which all old non-active projects, unused legacy code, and libraries were removed. What remained was then refreshed, extensively tested against all existing projects, and renamed VSL 8. This is the current operationally version of VSL.

1.1.14 VSL 9

During this tidy-up and refresh it became clear that to take advantage of the latest 2017 C++, Microsoft Visual Studio, and other 3rd party utilities and libraries, required a complete re-write.

In 2022 it was decided to create a new “from the ground up” version of VSL.

VSL 8 then became a transitional resource.

The ambition was to not only provide for a fresh codebase, but to substantially improve efficiency, increase comprehensibility and implementation clarity, and reduce the size of the code base.

1.2 Application Components

A software framework is a platform that provides a foundation for developing software applications. It can be considered as a template of a working program that can be selectively modified by adding code.

The VSL framework is comprised of three main software components.

> *In computer programming, an application framework consists of a software framework used by software developers to implement the standard structure of application software.*
>
> *Application frameworks became popular with the rise of Graphical User Interfaces (GUIs) since these tended to promote a standard structure for applications. Programmers find it much simpler to create automatic GUI creation tools when using a standard framework, since this defines the underlying code structure of the application in advance. Developers usually use object-oriented programming (OOP) techniques to implement frameworks such that the unique parts of an application can simply inherit from classes extant in the framework.*
>
> https://en.wikipedia.org/wiki/Application_framework

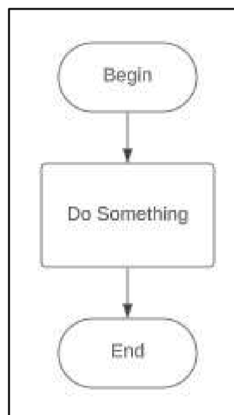
VSL Components

This section explains the VSL component design.

1. Starting with a simple program diagram,

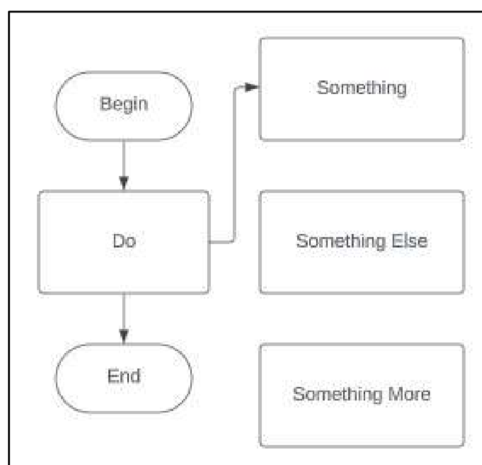
   "Do something" is a block of code that is executed between "Begin" and "End" blocks of code.

   These blocks of functionality might be blocks of code, Free Standing Functions (FSF), or Object methods.



2. The "Do" and the "Something" can be split to indicate control being passed from the "Do" to a "Something" function.

   Using the C++ "call by function pointer" method, "Something Else", or "Something More" could be invoked instead.



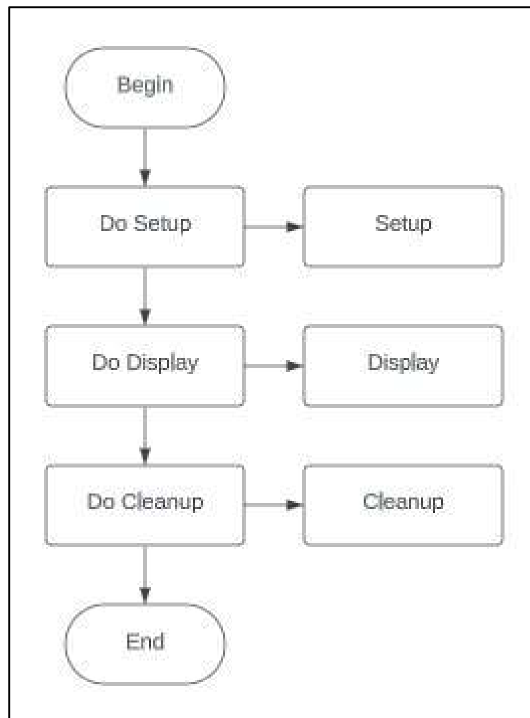
3. A sequential set of “Do’s” can pass control to a functionally matched group of functions or methods.

   With this logical separation, decoupling the left and right columns enhances clarity.

   The left column is a bare bones sketch of the WndCom.

   The right column is a bare bones sketch of the AppCom.

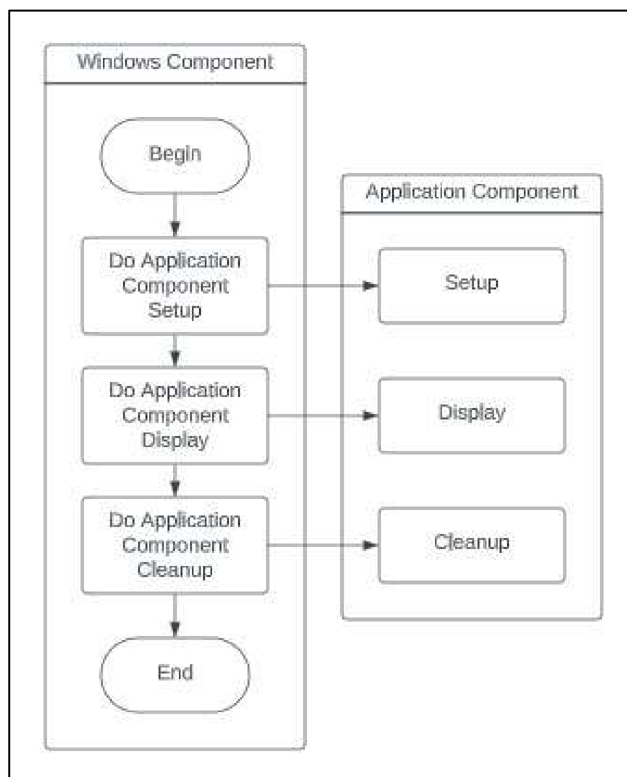
   Using the C++ “call by function pointer” method, swapping the “Setup”, “Display” and “Cleanup” functions for three other functionally compatible functions replaces one AppCom with another.



4. Logical functional groups such as the left and right columns are candidates to be described as a “Component”.

   This grouping provides for clarity of functional identity and purpose.

   It also provides for a first glimpse at how these Components might be candidates for the Controller and View parts of a MVC, Model–View–Controller Design Pattern.



5. In practice the previous diagram, the Windows Component's sequence of "Do" functions hide the need for substantial "housekeeping" code".

   Adding this additional code yet retaining the simplicity and clarity of the previous diagrams, a third intermediary Component is required; the Application Interface Component (AppInt) "fits" between the WndCom and the AppCom and provides for the WndCom to be decoupled from AppCom.

   WndCom invokes AppInt functions within a simple "try - throw" block, to test for exceptions.

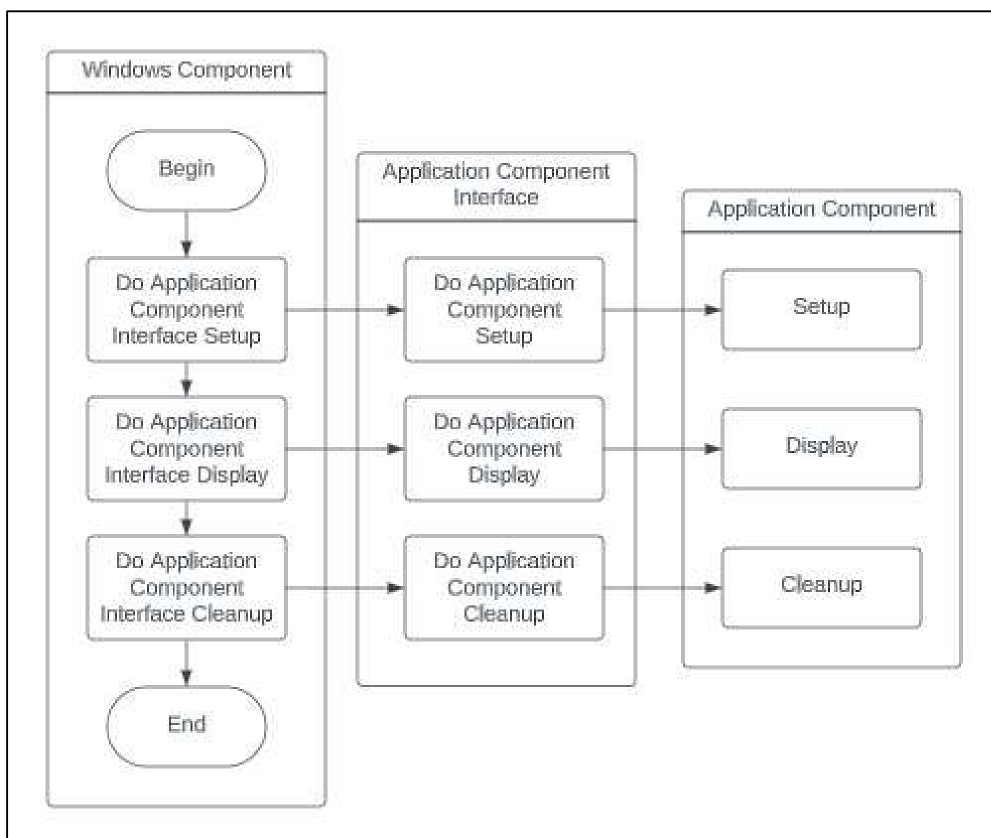
   The AppInt functions "housekeeping" provides "before" and "after" functionality for an AppCom function.

   If the "before" fails, control is passed back to WndCom with an exception value.

   AppInt invokes AppCom functions within a simple "try - throw" block, to test for exceptions. If the there is an exception control is passed back to WndCom with an exception value.

   If the "after" fails, control is passed back to WndCom with an exception value.

   Functionally the WndCom and AppInt are grouped together as the Controller in a MVC Design Pattern.



1.3 Application Frame Rate

An application is real-time, when it provides a sufficient rate of interaction and frame display update to provide visually for a perceived smooth User/Application interaction. This is known as the persistence of vision, an optical illusion where the human eye perceives the continued presence of an image after it has disappeared (also known as retinal persistence).

The persistence of vision of a normal human eye is 1/16th of a second.

Typically, an AppCom function is invoked N times per second (by the AppInt function, which in term is invoked by the WndCom function) where N is usually equal to the refresh rate of a display monitor.

The standard refresh rate for desktop monitors is 60Hz, though high-performance monitors have been developed that support 20Hz, 144Hz and even 240Hz refresh rates, which ensure ultra-smooth content viewing.

In the diagram below control is passed from the WndCom, to the AppInt, then the AppCom. If during this an event is detected that requires the application to finish (e.g., a fatal error, user selected close or quit) then control is passed to “End”, otherwise control is looped back to the WndCom.

Operational Usage

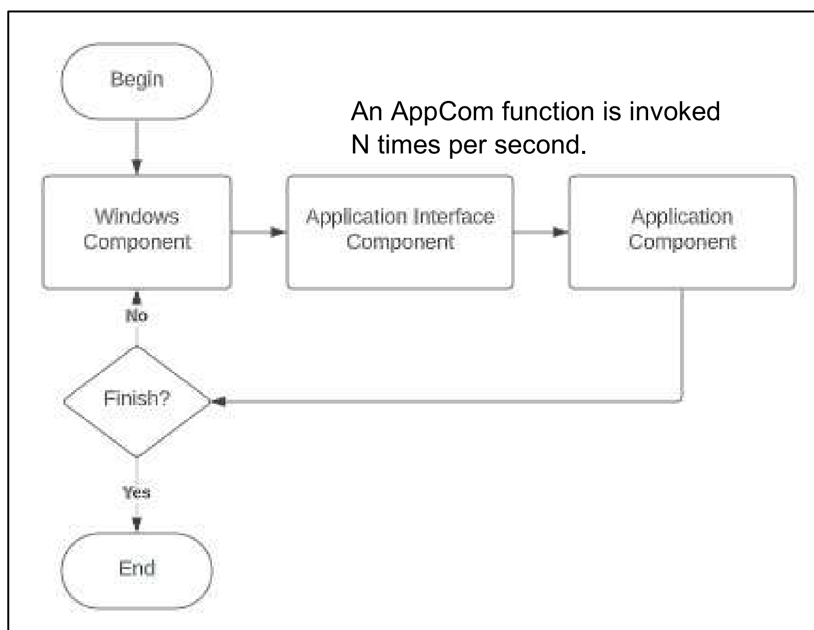
Real-time applications operate within an immediate timeframe; sensing, analysing, and acting on (streaming) data as it happens. This contrasts with a database-centric application where information is ingested and stored in a database (in the cloud or on-premises) for future analysis.

Many real-time applications rely on an Event-Driven Architecture (EDA) to allow for asynchronous processing of streaming data. This can be vital when operating within a specific time constraint.

An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website.

The VSL WndCom message handling function (WndEngine) is a hybrid EDA.

This hybrid will maintain a constant framerate at a self-correcting preset N times per second providing for a desired display update AND auto correcting when this is over or under achieved AND can achieve an appropriate event sampling framerate.



1.4 Application Layers

Each of the three VSL components and VSL libraries contain specific collections of reusable application elements, and these provide for the logical layers required to produce applications.

In this software architecture layer diagram, there are the five layers.

1. Presentation: also called the UI layer, handles the interactions that users have with the software. It's the most visible layer and defines the application's overall look and presentation to the end-users.

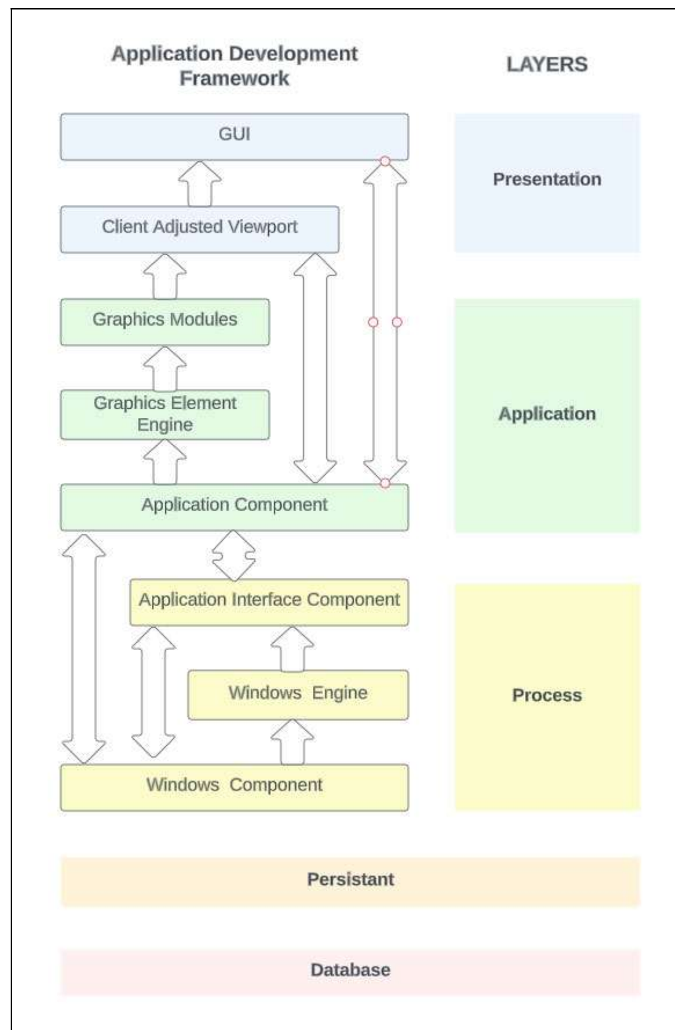
2. Application: handles the application architecture. It includes the application specific code definitions and functions.

   Developers create VSL presentation and applications layers using the VSL libraries (e.g., graphics, imaging) and 3rd party libraries.

3. Process: where the application's process logic operates. This is a collection of rules that tell the system how to handle an application.

4. Persistence: also called the data access layer, acts as a protective layer. It contains the code that's necessary to access the database layer.

5. Database: where the system stores all the data. It's the lowest tier in the software architecture.



The Presentaion and Application layers are provided with the VSL libraries (e.g., graphics, imaging) and 3rd party libraries.

The Persistence and Database layers are provided with graphics data sets and VSL libraries.

Temporary Notes:

https://www.ibm.com/topics/three-tier-architecture#:~:text=Three%2Dtier%20architecture%20is%20a,associated%20with%20the%20application%20is

https://www.indeed.com/career-advice/career-development/what-are-the-layers-in-software-architecture#:~:text=5%20layers%20in%20software%20architecture

1.5 Application Development

An Application Development Framework (ADF) is comprised of one or more integrated software libraries, that provide a complete fundamental structure to support the development of applications for a specific environment.

An ADF provides for the fast and easier development of programs by removing the need for code other than that required for a specific application. This is achieved by providing template source code for applications, graphics and Graphical User Interface (GUI) demos, and other speciality software resources.

An ADF acts as the skeletal support required to build an application.

1.5.1 VSL Codebase

For a developer a top to bottom familiarity with the VSL codebase is a must to get the best results.

Acquiring this knowledge includes becoming familiar with:

1. Background context (e.g., VS Lite and VSL 8)
2. VSL components & architecture
3. Structure and interface requirements of the:
    - Windows Component
    - Application Interface Component
    - Application Components:
        - Standalone
        - Base class
        - Derived classes
4. Model View Control (MVC) design pattern
5. Direct3D device
6. Compiling a demo stand-alone, and derived class demo
7. Design and building an application (e.g., animated Rubik's cube)

Points 1 & 2 have been dealt with in the 1.0 Introduction.

Points 3, 4 & 5 are detailed below.

Points 6 and 7 are tackled in a hands-on session with a trainer.

1.5.2 Windows Component

This is a robust, and simple to understand component. It must not be updated by a developer and is supplied so that they have acquire a fuller understanding of the framework.

If an additional features or functionality are required, then this must be requested from, and be subject to an impact assessment process, then modification, update & testing. Finally, a new version release date and documentation will be agreed.

It is comprised of Free-Standing Functions (FSF) and global variables. Much of the code is stylistically Microsoft, except for those parts which are GE logic, and as such they are clearly stylistically GE.

The Windows framework functions includes Win_Main, the application entry point, Win_Procedure (which defines the behaviour of the window), and Win_Engine (includes the top-level application loop, message handling, and timed frame handling).

Control from Win_Main and Win_Engine is passed as appropriate to the Application Interface, and then to the Application Component. Where Windows or Direct3D housekeeping is required this is implemented in the AppInt functions.

1.5.3 Interface Component

Like WndCom, AppInt is robust, and simple to understand. It must not be updated by a developer and is supplied so that they have acquire a fuller understanding of the framework.

Also, like WndCom, it is It is comprised of Free-Standing Functions (FSF) and global variables.

1.5.4 Application Component

The AppInt interface functions are a one-time application setup, Direct3D device setup, application display, Direct3D device clean-up and finally an application one-time clean up.

The AppCom interface is comprised of matching application object class methods.

Respectively the AppInt functions invoked these AppCom methods:

Application Interface	Application Component
AppInt_Setup	AppFw_Setup
AppInt_SetupDX	AppFw_SetupDX
AppInt_Display	AppFw_Display
AppInt_CleanupDX	AppFw_CleanupDX
AppInt_Cleanup	AppFw_Cleanup

The AppCom is a stand-alone class, or an application Base class, or an application class derived from that Base class.

1.5.5 Selecting an Application Component

At compile-time the Windows implementation includes a Microsoft Visual C++ (generated) resources header file, a C++ pre-processor application selection header file, and a Windows framework header functions interface and global properties file.

The C++ pre-processor application selection header file defines various application “names” token strings and sets an application token string to be the “selected name”.

This selects the application.

The application token string is then used in the included file to select a block of code that includes one or more application specific header files (the first of which must be an application class interface file), and then instantiates an instance of a global variable of type class (g_app).

This global variable is then used throughout the WndCom and AppInt to invoke the methods of this instance.

To avoid breaking future version of VSL, the methods invoked are a minimal, but complete (famous last words!), functional set.

1.5.6 Versioning and Setup

Compared to previous version of VSL, this is a radically simplified process.

Future versions of this VSL may include fully automated versioning and setup.

The following is provided **ONLY** for insight into the provisioning for each application of unique versioning, desktop icons, application icons, and texture atlas (used for GUI components, but which can also be accessed by an application.

This is done using a batch file to run a boot-strap utility application, that generates a resource script, “vs.rc”.

First an application specific batch file is run:

```
echo off
setlocal
      path=%CD%;%path%
      app_version -n c3 -i 1 -d "ImgCam
REM (application camera_05 configured from camera_03)"
      app_setup vs camera_03 c3 -c camera_05 -i c5
endlocal
```

Here, “c3” is an application project file prefix.

The “app_version” utility application updates this application semantic versioning #’s.

Then “app_setup” generates a “vs.rc” resource file, which might look like:

```
/////////////////////////////////////////////////////////////////////////////
// VSX C++ generated resource script.
//

#include "vs_resources\resource.h"

// ***** LOTS OF RESOURCE INSERTED STUFF HERE *****

/////////////////////////////////////////////////////////////////////////////
//
// Versioning

VS_VERSION_INFO VERSIONINFO
   FILEVERSION 7,1,1,2042
   PRODUCTVERSION 2,1,1,134
   FILEFLAGSMASK 0x3fL

#ifdef _DEBUG
   FILEFLAGS 0x1L
#else
   FILEFLAGS 0x0L
#endif

   FILEOS 0x40004L
   FILETYPE 0x1L
   FILESUBTYPE 0x0L
BEGIN
   BLOCK "StringFileInfo"
   BEGIN
      BLOCK "080904b0"
      BEGIN
         VALUE "CompanyName",  "Observant Innovations"
         VALUE "FileDescription",  "Camera 03"
         VALUE "ProductVersion",   "2.1.1.134"
         VALUE "FileVersion",      "7.1.1.2042"
         VALUE "InternalName",     "vs.exe"
         VALUE "OriginalFilename", "camera_03.exe"
         VALUE "LegalCopyright",   "Copyright ©2013 - 2023"
         VALUE "ProductName",      "Camera 05"
      END
   END
   BLOCK "VarFileInfo"
   BEGIN
      VALUE "Translation", 0x809, 1200
   END
END
```

```
///////////////////////////////////////////////////////////////////////////
//
// ICONS (also defined in resource.h)

IDI_HICON         ICON       "vs_resources\\icons_c5\\48.ico"
IDI_HICONSM       ICON       "vs_resources\\icons_c5\\16.ico"

///////////////////////////////////////////////////////////////////////////
//
// IMAGES for TEXTURE ATLAS (also defined in resource.h)

IDR_RCDATA00       RCDATA       "vs_resources\\atlas_c5\\00.tga"
.
.
.
IDR_RCDATA19       RCDATA       "vs_resources\\atlas_c5\\19.tga"

#endif    // English (U.K.) resources

///////////////////////////////////////////////////////////////////////////
```

1.5.7 Installation

There is also utility application that can update the NSIS installer scripts.

NSIS: Nullsoft Scriptable Install System download | SourceForge.net

From NSIS web page:

NSIS (Nullsoft Scriptable Install System) is a professional open-source system to create Windows installers. It is designed to be as small and flexible as possible and is therefore very suitable for internet distribution.

NSIS is script-based and allows you to create the logic to handle even the most complex installation tasks. Many plug-ins and scripts are already available: you can create web installers, communicate with Windows and other software components, install, or update shared components and more.

1.5.8 Run-time

At run-time the Windows application entry point function (Win_Main) invokes (after initialising, creating, registering, and then opening a window) the application framework setup function (AppFw_Setup).

Then the application interface setup functions (AppInt_Setup, AppInt_SetupDX).

Then invokes the Win_Engine function (which loops until an error condition arises that can't be handled OR the user quits/closes).

Then the application clean-up methods (AppInt_CleanupDX, AppInt_Cleanup).

1.5.9 Direct3D Device Reset

The application interface setup and clean-up methods (AppInt_Setup, AppInt_Cleanup) do not invoke the application framework methods (AppFw_Setup, AppFw_Cleanup, as these invoke one-time methods) and the application interface setup and clean-up methods are invoked when the Direct3D device requires a reset (e.g., the window is resized).

This requires that the WndCom "cleans up", re-initialises and then resets the Direct3D device, then re-setups all Direct3D device resources.

So:

- AppInt device clean-up method (AppInt_CleanupDX) releases the Direct3D device.
- AppInt and AppCom device clean-up methods (AppInt_CleanupDX and AppFw_CleanupDX) release the Direct3D graphic objects (e.g., fonts, images), then recursively releases Direct3D buffers (e.g., vertex, index).
- In the AppInt the Direct3D presentation parameters object is re-initialised (new width & height), and the Direct3D device reset.
- AppInt and AppCom device setup methods (AppInt_SetupDX and AppFw_SetupDX) recreate the Direct3D graphic objects (e.g., fonts, images), then recursively recreate Direct3D buffers (e.g., vertices, vertex, index).

See Direct3D section at the end of this chapter for an explanation of what a Direct3D device is.

1.6 PImpl Design Pattern

"Pointer to implementation" or "pImpl" is a C++ programming technique that removes implementation details of a class from its object representation by placing them in a separate class, accessed through an opaque pointer.

A shorter version of the above might be: “The pimpl idiom is a modern C++ technique to hide implementation, to minimize coupling, and to separate interfaces.”

This technique is used to construct C++ library interfaces with stable ABI (Application Binary Interface) and to reduce compile-time dependencies.

Because private data members of a class participate in its object representation, affecting size and layout, and because private member functions of a class participate in overload resolution (which takes place before member access checking), any change to those implementation details requires recompilation of all users of the class.

pImpl removes this compilation dependency; changes to the implementation do not cause recompilation. Consequently, if a library uses pImpl in its ABI, newer versions of the library may change the implementation while remaining ABI-compatible with older versions.

The alternatives to the pImpl idiom are:

- inline implementation: private members and public members are members of the same class.
- pure abstract class (OOP factory): users obtain a unique pointer to a lightweight or abstract base class; the implementation details are in the derived class that overrides its virtual member functions.

Runtime overhead:

1. Access
2. Space
3. Lifetime management

1.7 MVC Design Pattern

The Model-View-Controller (MVC) software design pattern can be found throughout VSL.

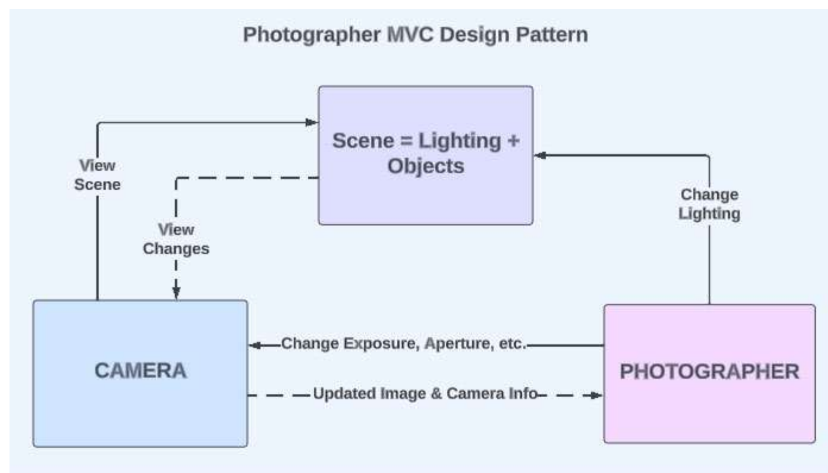
It is used to isolate implementation logic, the Model, from the View and Control.

It is also found in many different areas of organisation and usage throughout the “real” world.

1.7.1 Photographer MVC Design Pattern

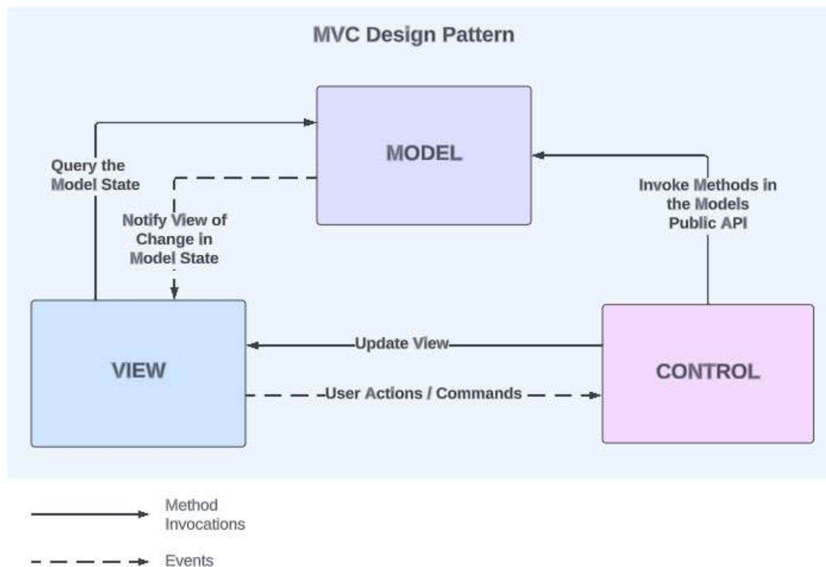
For example:

> *"MVC (Model, View, Controller) is a pattern for organising code in an application to improve maintainability."*
>
> *"Imagine a photographer with his camera in a studio. A customer asks him to take a photo of a nicely lit group of objects – the scene."*
>
> *"The scene is the model; the photographer is the controller, and the camera is the view."*
>
> *"Because the scene does not ‘know’ about the camera or the photographer, it is completely independent.”*
>
> *“This separation allows the photographer to walk around the scene and point the camera at any angle to get the shot/view that he wants."*
>
> *“If the photographer is not satisfied with the lighting, he can change/adjust the lighting; this changes the scene, a change viewed by the camera, an image of which is seen by the photographer, who can then change/refine the lighting. And so on…”*
>
> *Adapted from JW01’s MVC for Programmers, Stack Exchange (18 December 2012)*



> *"Non-MVC architectures tend to be tightly integrated together.”*
>
> *“If the scene, the controller, and the camera were one-and-the-same-object, then we would have to pull them apart and then re-build both the scene and the camera each time we wanted to get a new view. Also, taking a photo would always be like trying to take a selfie - and that's not always very easy."*
>
> *Adapted from JW01’s MVC for Programmers, Stack Exchange (18 December 2012)*

1.7.2 The MVC Design Pattern



Model

Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects...

There should be a one-to-one correspondence between the model and its parts on the one hand, and the represented world as perceived by the owner of the model on the other hand. The nodes of a model should therefore represent an identifiable part of the problem.

The nodes of a model should all be on the same problem level, it is confusing and considered bad form to mix problem-oriented nodes (e.g., calendar appointments) with implementation details (e.g., paragraphs).

View

A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter.

A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. All these questions and messages must be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents. (It may, for example, ask for the model's identifier and expect an instance of Text, it may not assume that the model is of class Text).

Control

A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on .to one or more of the views.

A controller should never supplement the views, it should for example never connect the views of nodes by drawing arrows between them.

Conversely, a view should never know about user input, such as mouse operations and keystrokes. It should always be possible to write a method in a controller that sends messages to views which exactly reproduce any sequence of user commands.

Editors

A controller is connected to all its views, they are called the parts of the controller. Some views provide a special controller, an editor, that permits the user to modify the information that is presented by the view. Such editors may be spliced into the path between the controller and its view and will act as an extension of the controller. Once the editing process is completed, the editor is removed from the path and discarded.

Note that an editor communicates with the user through the metaphors of the connected view, the editor is therefore closely associated with the view. A controller will get hold of an editor by asking the view for it - there is no other appropriate source.

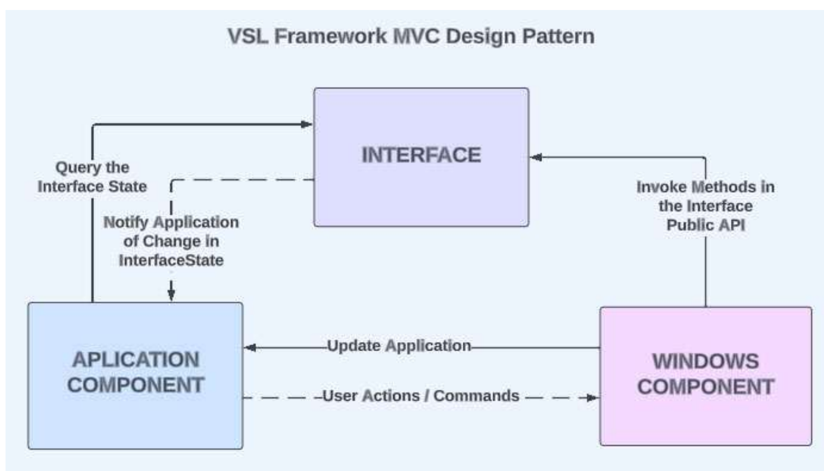
Italicised text is from Trygve Reenskaug's "MVC Formulation" (10 December 1979)

1.7.3 VSL Framework usage of the MVC Design Patterns

Model: All application implementation logic resides inside the AppCom. This provides implementation C++ classes for managing and displaying graphics.

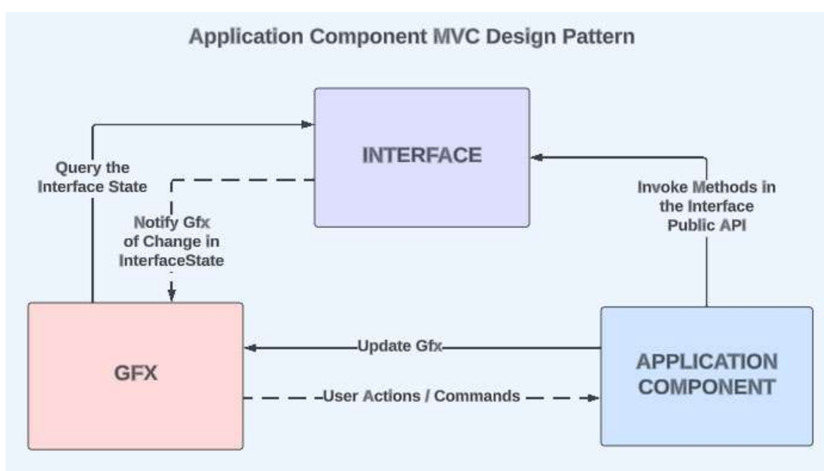
View: All View implementation logic resides inside the APPLICATION COMPONENT. This provides the application C++ class.

Control: All Control implementation logic resides inside the WINDOWS COMPONENT.



VSL Application Component usage of MVC Design Patterns

View: All View implementation logic resides inside GFX, which provides the Gfx C++ classes.

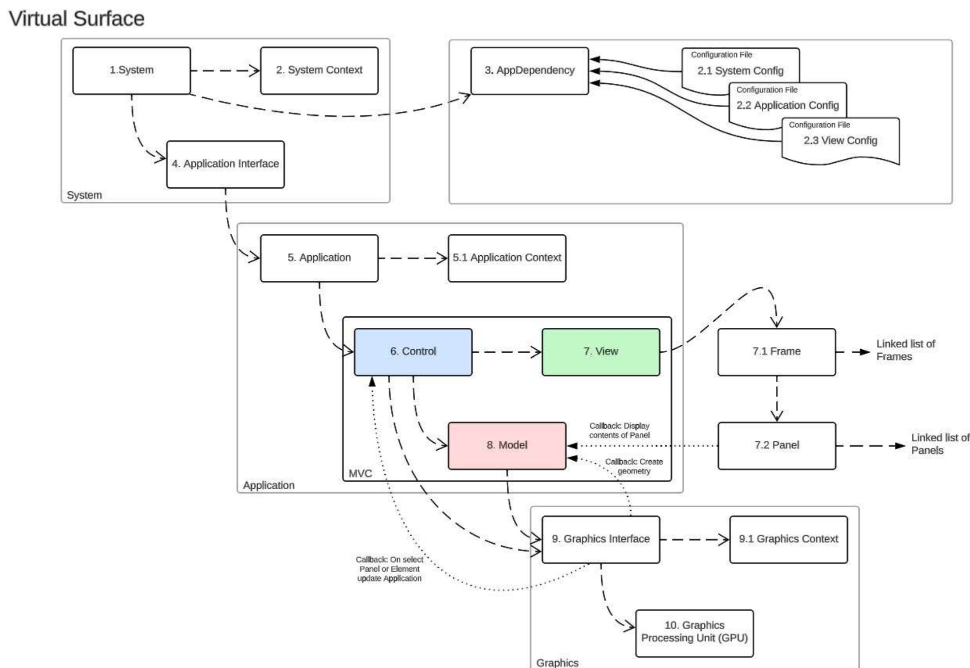


1.7.4 Previous VSL versions usage of MVC

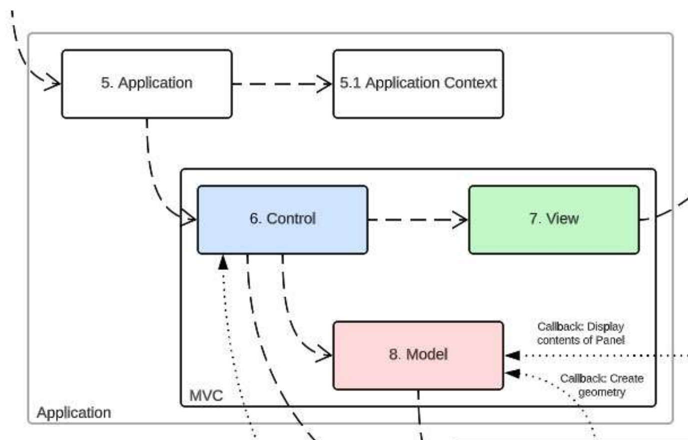
VSL versions 7 & 8 make extensive use of the MVC Design Pattern. The following has been taken from 2016 Virtual Surface 7 Documentation:

> *"As described in section 2 above, VS is comprised of three functionally separate parts; System, Application and Graphics."*
>
> *"In System the MVC design pattern is comprised of Virtual Surface (Control), System Context (View) and Application Interface (Model)."*
>
> *"In Application the MVC design is comprised of Application (Control), Application Context (View) and Application (Model). The latter is itself a nested Implementation MVC design comprised of Control, View, and Model."*
>
> *"In Graphics the MVC design is comprised of Graphics Interface (Control), Graphics Context (View) and GPU (Model)."*
>
> *"These are depicted in illustration 1 (Virtual Surface Component Diagram)."*

"Illustration 1 (Virtual Surface Component Diagram)."



A detail from *"Illustration 1 (Virtual Surface Component Diagram)."*



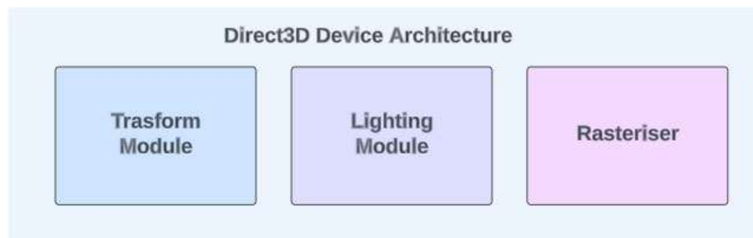
1.8 Direct3D

Two of the five AppInt functions and two of the AppCom methods principally handle the Direct3D device.

A Direct3D device is the rendering component of Direct3D. It encapsulates and stores the rendering state. In addition, a Direct3D device performs transformations and lighting operations and rasterizes an image to a surface.

A Direct3D device is the rendering component of Direct3D. It encapsulates and stores the rendering state. In addition, a Direct3D device performs transformations and lighting operations and rasterizes an image to a surface.

Architecturally, Direct3D devices contain a transformation module, a lighting module, and a rasterizing module, as the following diagram shows.



Direct3D currently supports two main types of Direct3D devices:

- A hal device with hardware-accelerated rasterization and shading with both hardware and software vertex processing.
- A reference device

You can think of these devices as two separate drivers. Software and reference devices are represented by software drivers, and the hal device is represented by a hardware driver. The most common way to take advantage of these devices is to use the hal device for shipping applications, and the reference device for feature testing. These are provided by third parties to emulate particular devices - for example, developmental hardware that has not yet been released.

The Direct3D device that an application creates must correspond to the capabilities of the hardware on which the application is running. Direct3D provides rendering capabilities, either by accessing 3D hardware that is installed in the computer or by emulating the capabilities of 3D hardware in software. Therefore, Direct3D provides devices for both hardware access and software emulation.

Hardware-accelerated devices give much better performance than software devices. The hal device type is available on all Direct3D supported graphic adapters. In most cases, applications target computers that have hardware acceleration and rely on software emulation to accommodate lower-end computers.

Except for the reference device, software devices do not always support the same features as a hardware device. Applications should always query for device capabilities to determine which features are supported.

Because the behaviour of the software and reference devices provided with Direct3D 9 is identical to that of the hal device, application code authored to work with the hal device will work with the software or reference devices without modifications. Note that while the provided software or reference device behaviour is identical to that of the hal device, the device capabilities do vary, and a particular software device may implement a much smaller set of capabilities.

2 Windows Classes

2.1 Win_Create

Provides for “setting” & “getting” the properties required to create a window. An instance of this is created and initialised by an Application Base class constructor.

Example:

```
// ---- Win_Create - get & initialise window create struct
      Win_Create *win_cr8 = Get_Win_Create();
      win_cr8->SetName("Base_01");
      win_cr8->SetCentred(FALSE);
      win_cr8->SetDesktop(FALSE);
      win_cr8->SetClient(800,600);
      win_cr8->SetAaq(4);
```

Definition:

```
class Win_Create
{

public:

// ---- cdtor
      Win_Create(VOID);
      ~Win_Create();

// ---- initialised by application constructor
      VOID SetAaq(UINT aaq);
      VOID SetCentred(BOOL centred);
      VOID SetDesktop(BOOL desktop);
      VOID SetName(std::string name);
      VOID SetXY(UINT x, UINT y);
      VOID SetClient(UINT width, UINT height);
      VOID SetRect(UINT width, UINT height);
      VOID SetYCaption(UINT y);
      VOID SetYBorder(UINT y);
      VOID SetYEdge(UINT y);

      UINT GetAaq(VOID);
      BOOL GetCentred(VOID);
      BOOL GetDesktop(VOID);
      std::string GetName(VOID);
      UINT GetX(VOID); UINT GetY(VOID);
      UINT GetClientWidth(VOID); UINT GetClientHeight(VOID);
      UINT GetRectWidth(VOID); UINT GetRectHeight(VOID);
      UINT GetYCaption(VOID);
      UINT GetYBorder(VOID);
      UINT GetYEdge(VOID);

private:

      class Pimpl_Win_Create; Pimpl_Win_Create  *pimpl_win_create;
};
```

Important:

To provide for the application to have a unique name (or any of the other properties), the SetName method must be invoked in an App_Initialise_Project override method of a class derived from the Base class, e.g.:

```
win_cr8->SetName("Base_01");
```

2.2 Win_Engine

Provides for “setting” & “getting” the properties required by, and to manage, the windows message (pump) engine.

An instance of this is created and initialised by Base class constructor.

Example:

```
// ---- Get_Win_Engine - get & initialise windows engine struct
      Win_Engine *win_eng = Get_Win_Engine();
      win_eng->SetColour(92, 92, 92);
      win_eng->SetFps(60);
```

Definition:

```
class Win_Engine
{

public:

// ---- cdtor
      Win_Engine(VOID);
      ~Win_Engine();

// ---- display set: by application constructor
      VOID  SetFps(UINT fps);
      VOID  SetColour(UINT r, UINT g, UINT b);

// ---- display: get
      UINT  GetFps(VOID);
      std::string GetFpsReport(VOID);
      UINT  GetRed(VOID), GetGreen(VOID), GetBlue(VOID);

// ---- fps: get & set
      VOID  SetFpsNow(UINT fps_now);
      UINT  GetFpsNow(VOID);

// ---- framework time: get & set
      VOID  SetMsStart(FLOAT ms_start);
      VOID  SetMsDelta(FLOAT ms_delta);
      VOID  SetMsElapsed(FLOAT ms_elapsed);
      VOID  SetMsNow(FLOAT ms_delta);
      VOID  SetSecondsLast(DWORD seconds_last);
      VOID  SetSecondsNow(DWORD seconds_now);

      FLOAT GetMsStart(VOID);
      FLOAT GetMsDelta(VOID);
      FLOAT GetMsElapsed(VOID);
      FLOAT GetMsNow(VOID);
      DWORD GetSecondsLast(VOID);
      DWORD GetSecondsNow(VOID);

      VOID  SetRegulated(BOOL regulated);
      BOOL  GetRegulated(VOID);

// ---- framework device: get & set
```

```
        /*
        note: these are non vsl_library legacy methods,
        used by applications NOT based on vsl_application
        */

        VOID  SetMouseLeftButtonDownMove(
        INT mouse_move_x, INT mouse_move_y);
        VOID  SetMouseWheelClickDelta(INT wheel_click_delta);

        // not used!
        INT   GetMouseLeftButtonDownX(VOID);
        INT   GetMouseLeftButtonDownY(VOID);
        INT   GetMouseWheelClickDelta(VOID);

        /*
        note: used in windows framework Win_Procedure function
        to acquire text input
        */
        VOID  SetKeyDown(INT key_down);
        VOID  SetKeyShift(INT key_shift);
        INT   GetKeyDown(VOID);
        INT   GetKeyShift(VOID);

        /*
        note: command line interface
        */
        VOID  SetCmdLineMode(BOOL mode);
        BOOL  GetCmdLineMode(VOID);

        std::string GetCmdLine();
        VOID SetCmdLine(std::string line);

private:

        class Pimpl_Win_Engine; Pimpl_Win_Engine *pimpl_win_engine;
};
```

3 Graphics Classes

3.1 Command

```
class Gfx_Command
{

public:

	// ---- cdtor
		Gfx_Command();
		~Gfx_Command();

	// ---- get
		DWORD GetKeyJustPressed(VOID);
		DWORD GetKeyLastPressed(VOID);
		FLOAT GetMouseLeftButtonDownMoveX(VOID);
		FLOAT GetMouseLeftButtonDownMoveY(VOID);
		FLOAT GetMouseWheelClick(VOID);

	// ---- set
		VOID  SetKeyJustPressed(DWORD key_just_pressed);
		VOID  SetKeyLastPressed(DWORD key_last_pressed);
		VOID  SetMouseLeftButtonDownMove(FLOAT x, FLOAT y);
		VOID  SetMouseWheelClick(FLOAT mouse_wheel_click_z);
		VOID  SetDefaultMouseWheelClick(FLOAT wheel_click);

	// ---- toggle
		HRESULT GetToggleValue(CHAR key, FLOAT *value);
		HRESULT SetToggleValue(CHAR key, FLOAT value);
		HRESULT SetToggle(CHAR key);

	// ---- big picture
		VOID Reset(VOID);
		VOID Update(FLOAT time_ms_delta);

private:

	// ---- private implementation
		class Pimpl_Gfx_Command;
		std::unique_ptr<Pimpl_Gfx_Command> pimpl_gfx_command;

};
```

3.2 D3dx

```
// ---------- Gfx_D3dx class interface ----------
class Gfx_D3dx
{

public:

	// ---- cdtor
		Gfx_D3dx();
		~Gfx_D3dx();

	// ---- set framework wnd struct
		VOID Set_Win_Create(vsl_system::Win_Create *win_cr8);
		VOID Set_Win_Engine(vsl_system::Win_Engine *win_eng);
		VOID Set_Gfx_Command(vsl_library::Gfx_Command *gfx_com);

	// ---- framework
		HRESULT Setup(VOID);
		HRESULT SetupDX(LPDIRECT3DDEVICE9);
		HRESULT SetupViewport(LPDIRECT3DDEVICE9);
		HRESULT SetupProjection(LPDIRECT3DDEVICE9);
		HRESULT SetupLookAtLH(LPDIRECT3DDEVICE9);
		HRESULT Display(LPDIRECT3DDEVICE9);
		HRESULT CleanupDX(LPDIRECT3DDEVICE9);
		HRESULT Cleanup(VOID);

		HRESULT ClearFrame(LPDIRECT3DDEVICE9, Gfx_Frame *frame);
		HRESULT DrawFrame(LPDIRECT3DDEVICE9, Gfx_Frame *frame);
		HRESULT SetupPerspectiveFovLH(LPDIRECT3DDEVICE9,
			Gfx_Frame *frame);

	// ---- utility
		HRESULT Clear(LPDIRECT3DDEVICE9);
		HRESULT DisplayText(std::string&,
			D3DCOLOR& colour, RECT& rect);

private:

	// ---- private implementation
		class Pimpl_Gfx_D3dx;
		std::unique_ptr<Pimpl_Gfx_D3dx> pimpl_gfx_d3dx;

};
```

3.3 Element

```
class Gfx_Element
{

public:

        // ---- cdtor
                Gfx_Element(VOID);
                virtual Gfx_Element::~Gfx_Element();

        // ---- build element (& param groups)
                Gfx_Element *Append(VOID);
                Gfx_Element *Append(std::string name);
                Gfx_Element *Append(std::string name, UINT id);
                Gfx_Element *Append(std::string name, std::string value);
                Gfx_Element *Append(std::string name, std::string value,
                                          UINT id);
                Gfx_Element *InsertAfter(VOID);
                Gfx_Element *InsertAfter(std::string name, UINT id);

        // ---- utility
                VOID List(VOID);
                Gfx_Element *Find(std::string name);

        // ---- get stuff
                Gfx_Element_Configure  *GetConfigure(VOID);
                Gfx_Element_Coordinate *GetCoordinate(VOID);
                Gfx_Element_Component  *GetComponent(VOID);

        // ---- set stuff
                VOID SetComponent(Gfx_Element_Component *component);

        // ---- get properties
                UINT   GetId(VOID);
                std::string GetName(VOID);
                std::string GetValue(VOID);

        // ---- set properties
                VOID   SetId(UINT id);
                VOID   SetName(std::string name);
                VOID   SetValue(std::string value);

        // ---- hierarchical node links
                Gfx_Element *GetParent(VOID);
                Gfx_Element *GetFirst(VOID);
                Gfx_Element *GetLast(VOID);
                Gfx_Element *GetPrevious(VOID);
                        Gfx_Element *GetNext(VOID);
                VOID  SetParent(Gfx_Element *parent);
                VOID  SetFirst(Gfx_Element *first);
                VOID  SetLast(Gfx_Element *last);
                VOID  SetPrevious(Gfx_Element *previous);
                VOID  SetNext(Gfx_Element *next);

        // ---- parameter group
                Gfx_Element *AppendParamGroup(std::string name);
                Gfx_Element *FindParamGroup(std::string name);
```

```
            Gfx_Element *GetFirstParamGroup(VOID);
            Gfx_Element *GetLastParamGroup(VOID);

            HRESULT SetParameterValue(std::string group,
                                      std::string name,
                                      std::string value
                                );
            VOID SetFirstParamGroup(Gfx_Element *first_param_group);
            VOID SetLastParamGroup(Gfx_Element *last_param_group);

private:

      // ---- private implementation
            class Pimpl_Gfx_Element;
            std::unique_ptr<Pimpl_Gfx_Element> pimpl_gfx_element;

};
```

3.4 Element Engine

```
class Gfx_Element_Engine
{

public:

        // ---- cdtor
                Gfx_Element_Engine(VOID);
                virtual ~Gfx_Element_Engine();

        // ----  framework
                HRESULT Setup(VOID);
                HRESULT SetupDX(LPDIRECT3DDEVICE9 device);
                HRESULT Display();
                HRESULT CleanupDX();
                HRESULT Cleanup(VOID);

        // ---- recursion
                HRESULT Setup(Gfx_Element *element, UINT level);
                HRESULT SetupDX(Gfx_Element *element, UINT level);
                HRESULT Transform(Gfx_Element *element, UINT level);
                HRESULT Display(Gfx_Element *element, UINT level);
                HRESULT CleanupDX(Gfx_Element *element, UINT level);
                HRESULT Cleanup(Gfx_Element *element, UINT level);

        // ---- element
                HRESULT Element_SetupDX(Gfx_Element *element);
                HRESULT Element_Transform(Gfx_Element *element);
                HRESULT Element_Display(Gfx_Element *element);
                HRESULT Element_CleanupDX(Gfx_Element *element);

        // ---- bookmarks
                VOID AddBookMark(Gfx_Element *element);
                std::list <vsl_library::Gfx_Element *>GetBookMarks();

        // ---- housekeeping
                HRESULT Log(VOID);
                HRESULT Log(std::string message);
                HRESULT Log(Gfx_Element *element, UINT level);

        // ---- get
                Gfx_Element *GetEngineRoot(VOID);
                Gfx_Log     *GetGfxLog(VOID);
                Gfx_Element *GetProjectRoot(VOID);
                Gfx_Kandinsky_Interface *GetKandinskyConfig(VOID);

        // ---- set
                HRESULT SetGfxLog(Gfx_Log *log);
                HRESULT SetGfxProject(Gfx_Element *element);

        // ---- gfx device get & set
                LPDIRECT3DDEVICE9 GetDevice(VOID);
                HRESULT SetDevice(LPDIRECT3DDEVICE9 device);

private:

        // ---- private implementation
```

```
            class Pimpl_Gfx_Element_Engine;
            std::unique_ptr<Pimpl_Gfx_Element_Engine>
                   pimpl_gfx_element_engine;

};
```

3.5 Element Configure

```
class Gfx_Element_Configure
{

public:

	// ---- cdtor
		Gfx_Element_Configure(VOID);
		~Gfx_Element_Configure();

	// ---- is
		BOOL IsComponent(VOID);
		BOOL IsInstance(VOID);
		BOOL IsVisible(VOID);

	// ---- get
		BOOL GetComponent(VOID);
		std::string GetComponentName(VOID);
		BOOL GetVisible(VOID);
		D3DXMATRIX *GetMatrix(VOID);

	// ---- set
		VOID SetComponent(BOOL component);
		VOID SetComponentName(std::string component_name);
		VOID SetVisible(BOOL visible);
		VOID SetMatrix(D3DXMATRIX &matrix);

	// ---- get instance
		BOOL GetInstance(VOID);
		std::string GetInstanceName(VOID);
		Gfx_Element *GetInstanceElement(VOID);

	// ---- set instance
		VOID SetInstance(BOOL instance);
		VOID SetInstanceName(std::string instance_name);
		VOID SetInstanceElement(Gfx_Element *instance_element);

	// ---- get parameter groups
		VOID GetParameterGroups(Gfx_Element *parameter_groups);

private:

	// ---- private implementation
		class Pimpl_Gfx_Element_Configure;
		std::unique_ptr<Pimpl_Gfx_Element_Configure>
			pimpl_gfx_element_configure;

};
```

3.6 Element Coordinate

```
class Gfx_Element_Coordinate
{

public:

	// ---- cdtor
		Gfx_Element_Coordinate(VOID);
		~Gfx_Element_Coordinate();

	// ---- get
		VOID GetScale(vsl_system::Vsl_Vector3& scale);
		VOID GetRotate(vsl_system::Vsl_Vector3& rotate);
		VOID GetTranslate(vsl_system::Vsl_Vector3& translate);
		VOID GetRotationOrder(std::string& rotation_order);
		VOID GetRotationOrderIndex(UINT& rotation_order_index);

	// ---- set
		VOID SetScale(vsl_system::Vsl_Vector3& scale);
		VOID SetRotate(vsl_system::Vsl_Vector3& rotate);
		VOID SetTranslate(vsl_system::Vsl_Vector3& translate);
		VOID SetRotationOrderIndex(UINT& rotation_order_index);

	// ---- get parameter groups
		VOID GetParameterGroups(Gfx_Element *parameter_groups);

private:

	// ---- private implementation
		class Pimpl_Gfx_Element_Coordinate;
			std::unique_ptr<Pimpl_Gfx_Element_Coordinate>
			pimpl_gfx_element_coordinate;

};
```

3.7 Element Component

```
class Gfx_Element_Component
{

public:

	// ---- cdtor
		Gfx_Element_Component(VOID);
		~Gfx_Element_Component();

	// ---- get
		Gfx_Kandinsky                  *GetKandinsky(VOID);
		LPDIRECT3DVERTEXBUFFER9        *GetVertexBuffer(VOID);
		LPDIRECT3DINDEXBUFFER9         *GetIndexBuffer(VOID);
		UINT                            GetConfigBitmask(VOID);
		Gfx_Kandinsky_Interface_Callbacks*
		GetKandinskyInterfaceCallbacks(VOID);

	// ---- set
		VOID SetVertexBuffer(LPDIRECT3DVERTEXBUFFER9 vertex_buffer);
		VOID SetIndexBuffer(LPDIRECT3DINDEXBUFFER9 index_buffer);
		VOID SetConfigBitmask(UINT config_bitmask);

private:

	// ---- private implementation
		class Pimpl_Gfx_Element_Component;
		std::unique_ptr<Pimpl_Gfx_Element_Component>
			pimpl_gfx_element_component;

};
```

3.8 Frameset

```
class Gfx_Frameset
{

public:

	// ---- cdtor
		Gfx_Frameset();
		~Gfx_Frameset();

	// ---- links
		Gfx_Frame *AddFrame(std::string name);
		Gfx_Frame *GetFirst();

	// ---- dimensions
		HRESULT SetDimensions(UINT width, UINT height);
		UINT    GetWidth();
		UINT    GetHeight();

	// ---- calculate
		HRESULT Setup();

	// ---- log & report
		HRESULT SetGfxLog(Gfx_Log *gfx_log);

private:

	// ---- private implementation
		class Pimpl_Gfx_Frameset;
		std::unique_ptr<Pimpl_Gfx_Frameset> pimpl_gfx_frameset;
};
```

3.9 Log

```
class Gfx_Log
{

public:

	// ---- cdtor
		Gfx_Log();
		~Gfx_Log();

	// ---- set
		VOID SetShowDate(BOOL show_date);
		VOID SetShowTime(BOOL show_time);
		VOID SetShowLineNumber(BOOL show_line_number);
		VOID SetShowSimple(BOOL show_simple);

	// ---- write
		HRESULT Write(VOID);
		HRESULT Write(CHAR *message);
		HRESULT Write(std::string message);

	// ---- banner
		HRESULT WriteBanner(std::string banner_message);
		HRESULT WriteBannerLine();
		HRESULT WriteBannerMessage(std::string banner_message);


private:

	// ---- private implementation
		class Pimpl_Gfx_Log;
		std::unique_ptr<Pimpl_Gfx_Log> pimpl_gfx_log;

};
```

4 Application Classes

4.1 Overview

The Base class definition is given below.

The definition can be divided into seven conceptual blocks.

These are:

- Concrete:
  1. Application Component methods invoked by the Application Interface
  2. Get setup struct methods invoked by the Windows Component
  3. Set mouse & keyboard update methods invoked by the Windows Component

- Virtual methods invoked by Application Component methods:
  4. Initialisation invoked at run-time.
  5. Display invoked each frame (e.g., 60 times a second).
  6. Update invoked each frame (ditto).

- Private Implementation (PImpl)
  7. Initialised by Base class constructor.

The whole of this section details implementation.

Note: Leveraging the VSL Framework design, the supplied application Base class could be replaced with another (though ALL the non-virtual methods would need to be replicated "as is" to provide the necessary Windows and Application Interface functionality); and a new "family" of derived application classes could be constructed.

4.2 Private Implementation (PImpl)

"Pointer to implementation" or "pImpl" is a C++ programming technique that removes implementation details of a class from its object representation by placing them in a separate class, accessed through an opaque pointer.

This technique is used to construct C++ library interfaces with stable ABI (Application Binary Interface) and to reduce compile-time dependencies.

Because private data members of a class participate in its object representation, affecting size and layout, and because private member functions of a class participate in overload resolution (which takes place before member access checking), any change to those implementation details requires recompilation of all users of the class.

pImpl removes this compilation dependency; changes to the implementation do not cause recompilation. Consequently, if a library uses pImpl in its ABI, newer versions of the library may change the implementation while remaining ABI-compatible with older versions.

The alternatives to the pImpl idiom are:

- inline implementation: private members and public members are members of the same class.
- pure abstract class (OOP factory): users obtain a unique pointer to a lightweight or abstract base class; the implementation details are in the derived class that overrides its virtual member functions.

Runtime overhead:

4. Access
5. Space
6. Lifetime management

4.2.1 Definition

Comprising constructor, windows framework structs, and Gfx objects.

The framework structs are system objects (hence vs_system namespace).

The Gfx objects are library objects (hence vs_library namespace).

```
class Base_01::Pimpl_Base_01
{

public:

	// ---- cdtor
		Pimpl_Base_01() {}

	// ---- required by windows framework:
		vsl_system::Win_Create fw_win_create;
		vsl_system::Win_Engine fw_win_engine;

	// ---- required gfx:
		vsl_library::Gfx_Command  gfx_command;
		vsl_library::Gfx_D3dx     gfx_d3dx;
		vsl_library::Gfx_Frameset gfx_frameset;
		vsl_library::Gfx_Log      gfx_log;

		vsl_library::Gfx_Element_Engine gfx_element_engine;

};
```

It is declared in the Base class definition.

```
private:

// ---- private implementation
	class Pimpl_Base_01;
	std::unique_ptr<Pimpl_Base_01> pimpl_base_v01;
```

This is initialised when an instance of the Base class is instantiated, and automatically deleted when this instance of the Base class is destroyed.

See the next section “Deleter” for more information.

4.2.2 Deleter

The deleter for a shared pointer is created here:

```
Widget::Widget(): pImpl(new Impl) {}
```

Until that point, all the shared pointer has is the equivalent of a `std::funciton<VOID(Impl*)>`.

When you construct a shared_ptr with a T*, it writes a deleter and stores it in the `std::function` equivalent.

At that point the type must be complete.

So the only functions you have to define after `Impl` is fully defined are those that create a pImpl from a T* of some kind.

Note: The `unique_ptr<>` template holds a pointer to an object and deletes this object when the `unique_ptr<>` object is deleted.

So, in the example above, it does not matter if the function scope is left through the return statement, at the end of the function or even through an exception: The `unique_ptr<>` destructor is always called and therefore the object (int in the example) always deleted.

Note: deleted when it goes out of scope.

See: Herb Sutter's Exceptional C++.

The Base class:

```
	// ---- base class interface
		class Base_01
		{
		public:
			Base_01();
		private:
			class Pimpl_Base_01;
			std::unique_ptr<Pimpl_Base_01> pimpl_base_v01;
		};
```

The Base Private Implemention class:

```
	// ---- base class implementation derived from private implemetation
		class Base_01::Pimpl_Base_01
		{
			// ---- stuff
		};
```

The Base class constructor:

```
	// ---- constructor initialises private implemetation
		Base_01::Base_01(VOID) : pimpl_base_v01(new Pimpl_Base_01)
		{
			// ---- stuff
		}
```

4.2.3 Get

In the example above, the base class must implement methods such as those listed below, so that a derived class, or other (e.g., the FSF Win_Main), can "Get" the private implementation properties.

```
// ---- get Win objects
       vsl_system::Win_Create *Get_Win_Create(VOID);
       vsl_system::Win_Engine *Get_Win_Engine(VOID);

// ---- get gfx objects
       vsl_library::Gfx_Command        *GetCommand(VOID);
       vsl_library::Gfx_D3dx           *GetD3dx(VOID);
       vsl_library::Gfx_Log            *GetLog(VOID);
       vsl_library::Gfx_Frameset       *GetFrameset(VOID);
       vsl_library::Gfx_Element_Engine *GetElementEngine(VOID);
```

4.3 Base Class Definition

```
namespace vsl_application
{

	// ---------- Base_01 class ----------
	class Base_01
	{

	public:

		// ---- cdtor
		Base_01(VOID);
		virtual ~Base_01();

		// ---------- APPLICATION COMPONENT ----------

		// ---- required by component
		HRESULT AppFw_Setup(VOID);
		HRESULT AppFw_SetupDX(LPDIRECT3DDEVICE9);
		HRESULT AppFw_Display(LPDIRECT3DDEVICE9);
		HRESULT AppFw_CleanupDX(LPDIRECT3DDEVICE9);
		HRESULT AppFw_Cleanup(VOID);

		// ---- required by component: get wnd structs
		VOID AppFw_Get_Win_Create(vsl_system::Win_Create **fw_win_create);
		VOID AppFw_Get_Win_Engine(vsl_system::Win_Engine **fw_win_engine);

		// ---- required by component to set Gfx_Command parameter
		VOID AppFw_Set_MouseLeftButtonDownMove(INT x, INT y);
		VOID AppFw_Set_MouseWheelClick(INT mouse_wheel_click_z);
		VOID AppFw_Set_Keydown(WPARAM param);

		// ---------- APPLICATION ----------

		// ---- invoked by AppFw_Setup
		virtual HRESULT App_Initialise_Frameset(VOID);
		virtual HRESULT App_Initialise_Project(VOID);
		virtual HRESULT App_Initialise_Element_Configurations(VOID);
		virtual HRESULT App_Initialise_Element_Coordinates(VOID);
		virtual HRESULT App_Initialise_Element_Components(VOID);

		// ---- invoked by AppFw_Display
		virtual HRESULT App_Display_Project_In_CAV(LPDIRECT3DDEVICE9);
		virtual HRESULT App_Display_Project_In_Frameset(
					LPDIRECT3DDEVICE9 device,
					vsl_library::Gfx_Frame *frame, UINT level);

		// ---- invoked by AppInt_Display
		virtual VOID App_Update_Element_Bookmarks(VOID);
		virtual VOID App_Update_By_Command(VOID);
		virtual VOID App_Update_By_AsyncKeys(VOID);
		virtual VOID App_Update_Cav_Text(VOID);


		// ---------- PRIVATE IMPLEMENTATION ----------

		// ---- get Win objects
		vsl_system::Win_Create *Get_Win_Create(VOID);
		vsl_system::Win_Engine *Get_Win_Engine(VOID);
```

```
		// ---- get gfx objects
		vsl_library::Gfx_Command        *GetCommand(VOID);
		vsl_library::Gfx_D3dx           *GetD3dx(VOID);
		vsl_library::Gfx_Log            *GetLog(VOID);
		vsl_library::Gfx_Frameset       *GetFrameset(VOID);
		vsl_library::Gfx_Element_Engine *GetElementEngine(VOID);

	private:

		// ---- private implementation
		class Pimpl_Base_01;
		std::unique_ptr<Pimpl_Base_01> pimpl_base_v01;

	public:

		// ---- get private implementation
		//
		// usage: std::unique_ptr<Pimpl_Base_01> *pimp = GetPimp(VOID);
		//
		// std::unique_ptr<Pimpl_Base_01>
		//  *GetPimp(VOID) { return &pimpl_base_v01; }

	};

}
```

4.4 Constructor & Setup

```
Base_01::Base_01(VOID) : pimpl_base_v01(new Pimpl_Base_01)
{

 // ---- local
        HRESULT hr;

 using namespace vsl_system;

 // ---- Windows Component

 // ---- Win_Create - get & initialise window create struct
        Win_Create *win_cr8 = Get_Win_Create();
        win_cr8->SetName("Base_01");
        win_cr8->SetCentred(FALSE);
        win_cr8->SetDesktop(FALSE);
        win_cr8->SetClient(800,600);
        win_cr8->SetAaq(4);

 // ---- Get_Win_Engine - get & initialise windows engine struct
        Win_Engine *win_eng = Get_Win_Engine();
        win_eng->SetColour(92, 92, 92);
        win_eng->SetFps(60);

 using namespace vsl_library;

 // ---- Graphics (Gfx)

 // ---- Command (IO)
        Gfx_Command *gfx_command = GetCommand();
        gfx_command->SetDefaultMouseWheelClick(-50);
        hr = gfx_command->SetToggle((CHAR)'T');

 // ---- D3dx (Direct 3D)
        Gfx_D3dx *gfx_d3dx = GetD3dx();
        gfx_d3dx->Set_Win_Create(win_cr8);
        gfx_d3dx->Set_Win_Engine(win_eng);
        gfx_d3dx->Set_Gfx_Command(gfx_command);

 // ---- Gfx_Log
        Gfx_Log *gfx_log = GetLog();
        gfx_log->SetShowLineNumber(TRUE);
        gfx_log->SetShowDate(FALSE);
        gfx_log->SetShowTime(TRUE);
        gfx_log->SetShowSimple(FALSE);

 // ---- Gfx_Log - write this app name
        hr = gfx_log->WriteBanner(win_cr8->GetName());

 // ---- Gfx_Element_Engine
        Gfx_Element_Engine *gfx_gee = GetElementEngine();
        gfx_gee->SetGfxLog(gfx_log);
 }

 Base_01::~Base_01()
 { ; }
```

4.5 Framework Methods

4.5.1 Setup

```
// ---------- AppFw_Setup ----------
/*!
\brief required by application interface: setup
\author Gareth Edwards
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_Setup(VOID)
{

	// ---- local
		HRESULT hr;

	// ---- scope
	using namespace vsl_library;

	// ---- device
		hr = GetD3dx()->Setup();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- setup & config hierarchical frameset
		hr = App_Initialise_Frameset();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- setup & config hierarchical project structure
		hr = App_Initialise_Project();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- (one-time) recursively setup elements,
		element instancing & verify kandinsky component
		hr = GetElementEngine()->Setup();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- (one-time) recursively setup element coordinates
		hr = App_Initialise_Element_Coordinates();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- (one_time) recursively setup element components
		hr = App_Initialise_Element_Components();
		if (FAILED(hr)) return ERROR_FAIL;

	return SUCCESS_OK;
}


// ---------- AppFw_SetupDX ----------
/*!
\brief required by application interface: setup dx
\author Gareth Edwards
\param LPDIRECT3DDEVICE9 - pointer to an IDirect3DDevice9 structure
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_SetupDX(LPDIRECT3DDEVICE9 device)
{

	// ---- local
		HRESULT hr;
```

```
	// ---- d3d
		hr = GetD3dx()->SetupDX(device);
		if (FAILED(hr)) return hr;

	// ---- setup DX gfx element engine
		hr = GetElementEngine()->SetupDX(device);
		if (FAILED(hr)) return hr;

	// --- ignore – test or demonstration
		if (FALSE)
		{
			using namespace vs_library;

			// ---- create
			DotObjElement *object = new DotObjElement();

			// ---- read
				CHAR *filename = "object\\teapot\\teapot.obj";
				vs_library::DotObjUtilities dot_obj_utilities;
				HRESULT hr = (HRESULT)dot_obj_utilities.Read(
								object,
								filename);
				if (SUCCEEDED(hr))
				{
					dot_obj_utilities.Report(object, filename);
				}
		}

	return SUCCESS_OK;
}
```

4.5.2 Display

```
// ---------- AppFw_Display ----------
/*!
\brief required by application interface: Display
\author Gareth Edwards
\param LPDIRECT3DDEVICE9 - pointer to an IDirect3DDevice9 structure
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_Display(LPDIRECT3DDEVICE9 device)
{

	// ---- local
		HRESULT hr;

	// ---- clear backbuffers
		hr = GetD3dx()->Clear(device);

	// ---- display application
		hr = App_Display(device);

	// ---- user update project
		App_Update_By_Command();
		App_Update_By_AsyncKeys();
		App_Update_Element_Bookmarks();

	// ---- display within frameset OR within CAV
		vsl_library::Gfx_Frameset *gfx_frameset = GetFrameset();
		vsl_library::Gfx_Frame *frame = gfx_frameset->GetFirst();
		if ( frame != NULL )
			hr = App_Display_Project_In_Frameset(device, frame, 0);
		else
			hr = App_Display_Project_In_CAV(device);

		App_Update_Cav_Text();

	return SUCCESS_OK;
}
```

4.5.3 Cleanup

```
// ---------- AppFw_CleanupDX ----------
/*!
\brief required by application interface: CleanupDX
\author Gareth Edwards
\param LPDIRECT3DDEVICE9 - pointer to an IDirect3DDevice9 structure
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_CleanupDX (LPDIRECT3DDEVICE9 device)
{

	// ---- local
		HRESULT hr;

	//---- d3d
		hr = GetD3dx()->CleanupDX(device);
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- cleanup gfx element engine
		hr = GetElementEngine()->CleanupDX();
		if (FAILED(hr)) return ERROR_FAIL;

	return SUCCESS_OK;
}


// ---------- AppFw_Cleanup ----------
/*!
\brief required by framework: cleanup
\author Gareth Edwards
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_Cleanup(VOID)
{

// ---- cleanup gfx element engine
	HRESULT hr = GetElementEngine()->Cleanup();
	if (FAILED(hr)) return ERROR_FAIL;

	return SUCCESS_OK;
}
```

4.5.4 Get

```
// ---------- AppFw_Get_Win_Create ----------
/*!
\brief required by application interface: get pointer to wnd create struct
\author Gareth Edwards
\param Win_Create **wc
*/
VOID Base_01::AppFw_Get_Win_Create(vsl_system::Win_Create **fw_win_create)
{
	*fw_win_create = Get_Win_Create();
}


// ---------- AppFw_Get_Win_Create ----------
/*!
\brief required by application interface: get pointer to wnd engine struct
\author Gareth Edwards
\param Win_Engine **wc
*/
VOID Base_01::AppFw_Get_Win_Create(vsl_system::Win_Engine **fw_win_engine)
{
	*fw_win_engine = Get_Win_Engine();
}
```

4.5.5 Set

```
// ---------- AppFw_Set_MouseLeftButtonDownMove ----------
/*!
\brief required by application interface: if mouse left button down then update move x
y
\author Gareth Edwards
\param FLOAT x
\param FLOAT y
*/
VOID Base_01::AppFw_Set_MouseLeftButtonDownMove(INT x, INT y)
{
	GetCommand()->SetMouseLeftButtonDownMove((FLOAT)x, (FLOAT)y);
}


// ---------- AppFw_Set_MouseWheelClick ----------
/*!
\brief required by application interface: if mouse wheel click event update move z
\author Gareth Edwards
\param intd d (+/- 1)
*/
VOID Base_01::AppFw_Set_MouseWheelClick(INT d)
{
	FLOAT mouse_wheel_click = GetCommand()->GetMouseWheelClick();
	mouse_wheel_click += (FLOAT)d;

	GetCommand()->SetMouseWheelClick(mouse_wheel_click);
}


// ---------- AppFw_Set_Keydown ----------
/*!
\brief required by application interface: if key down handle toggle, etc.
\author Gareth Edwards
\param WPARAM (parameter)
*/
VOID Base_01::AppFw_Set_Keydown(WPARAM param)
{
	GetCommand()->SetToggle((CHAR)param);
}
```

4.6 Virtual Methods

4.6.1 Initialise

Invoked by AppFw_Setup.

```
virtual HRESULT App_Initialise_Frameset(VOID);
virtual HRESULT App_Initialise_Project(VOID);
virtual HRESULT App_Initialise_Element_Configurations(VOID);
virtual HRESULT App_Initialise_Element_Coordinates(VOID);
virtual HRESULT App_Initialise_Element_Components(VOID);
```

4.6.1.1 Frameset

```
// ---------- App_Initialise_Project --------
/*!
\brief on time initialisation of frameset
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Frameset(VOID)
{

        // ---- name
        vsl_system::Win_Create *win_cr8 = Get_Win_Create();
        win_cr8->SetName("Base");

        // ---- Gfx_Frameset
        using namespace vsl_library;
        Gfx_Frameset *gfx_frameset = GetFrameset();
        gfx_frameset->SetDimensions(
                win_cr8->GetClientWidth(),
                win_cr8->GetClientHeight());

        return SUCCESS_OK;
}
```

4.6.1.2 Project

```
// ---------- App_Initialise_Project --------
/*!
\brief on time initialisation of project
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Project(VOID)
{

	// ---- scope
	using namespace vsl_library;

	// ---- stop gap for "read file"
	try
	{
		// ---- build project
		Gfx_Element *engine_root_element = NULL;
		if (engine_root_element = GetElementEngine()->GetEngineRoot())
		{
			if (Gfx_Element *project = engine_root_element->Find("Project"))
			{
				// ---- element to be instanced
				Gfx_Element *element_tr0 = project->Append("TR0", 0);

				Gfx_Element_Configure *configure = element_tr0->GetConfigure();
				configure->SetComponentName("PyRhoDo_VBO");
				configure->SetVisible(FALSE);

				// ---- group id 1 : element id 4 (or could be 3!)
				Gfx_Element *element_group_1 = project->Append("G1", 1);
				GetElementEngine()->AddBookMark(element_group_1);
				{
					Gfx_Element *element_tr1 = element_group_1->Append("TR1", 4);
					GetElementEngine()->AddBookMark(element_tr1);

					Gfx_Element_Configure *configure =
						element_tr1->GetConfigure();
					configure->SetComponentName("Cuboid_VBO");
					configure->SetVisible(TRUE);
					configure->SetInstance(FALSE);
					configure->SetInstanceName(element_tr0->GetName());
				}

				// ---- group id 2 : element id 4
				Gfx_Element *element_group_2 = project->Append("G2", 2);
				GetElementEngine()->AddBookMark(element_group_2);
				{
					Gfx_Element *element_tr2 = element_group_2->Append("TR2", 4);
					GetElementEngine()->AddBookMark(element_tr2);

					Gfx_Element_Configure *configure =
						element_tr2->GetConfigure();
					configure->SetComponentName("Cuboid_VIBO");
					configure->SetVisible(TRUE);
					configure->SetInstance(TRUE);
					configure->SetInstanceName(element_tr0->GetName());
				}
			}
			else throw("Project");
		}
		else throw("Engine");

		// ---- report
		engine_root_element->List();

	}
	catch (CHAR *element_name)
	{
		CHAR msg[128];
		sprintf_s(msg, 128, "Element %s not found!\n Select OK to Exit", element_name);
		INT msgboxID = MessageBox(NULL, msg, "VSL method: Base_01::AppFw_Setup()",
			MB_ICONWARNING);
```

```
                return FALSE;
        }

        return SUCCESS_OK;
}
```

4.6.1.3 Configurations

```
// ---------- App_Initialise_Element_Configurations --------
/*!
\brief one time initilisation of project element configurations
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Element_Configurations(VOID)
{
        return SUCCESS_OK;
}
```

4.6.1.4 Coordinates

```
// ---------- App_Initialise_Element_Coordinates --------
/*!
\brief one time initialisation of project coordinates
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Element_Coordinates(VOID)
{

        // ---- scope
        using namespace vsl_library;

        // ---------- update element component properties & parameters ----------
        Gfx_Element *engine_root_element = GetElementEngine()->GetEngineRoot();

        // ---------- update element coordinate properties ----------
        Gfx_Element *element_tr1 = engine_root_element->Find("TR1");
        if (element_tr1 != NULL)
        {
                Gfx_Element_Coordinate *coordinate = element_tr1->GetCoordinate();
                vsl_system::Vsl_Vector3 v = { 1, 2, 1 };
                coordinate->SetScale(vsl_system::Vsl_Vector3(1, 2, 1));
        }

        return SUCCESS_OK;
}
```

4.6.1.5 Components

```
// ----------  App_Initialise_Element_Components --------
/*!
\brief one time initialisation of project components
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Element_Components(VOID)
{

	// ---- scope
	using namespace vsl_library;

	// ---------- update element properties & parameters ----------
	Gfx_Element *engine_root_element = GetElementEngine()->GetEngineRoot();
	Gfx_Element *element_tr0 = engine_root_element->Find("TR0");
	if (element_tr0 != NULL)
	{
		// ---- PyRhoDo_VBO
		Gfx_Element_Component *component = element_tr0->GetComponent();
		Gfx_Kandinsky *kandinsky = component->GetKandinsky();

		// ---- build - include inside -> update element component properties
		kandinsky->Set(Gfx_Kandinsky_Param::INSIDE, 1);

		// ---- move -> update both component & corresponding kandinsky parameters

		// ---- get group
		Gfx_Element *element_param_group = element_tr0->FindParamGroup("Component");
		if (element_param_group)
		{
			// ---- set parameter
			std::string move = "0.5";
			HRESULT hr =
				element_param_group->SetParameterValue("Dimension", "Move", move);

			// ---- if OK? - set kandinsky
			if (SUCCEEDED(hr))
			{
				kandinsky->SetParameterValue("Dimension", "Move", move);
			}
		}

	}
	return SUCCESS_OK;
}
```

4.6.2 Display

Invoked by `AppFw_Display`.

4.6.2.1 In Client Adjusted Viewport

```
// ---------- App_Display_In_CAV --------
/*!
\brief display elements in Client Adjusted Viewport
\author Gareth Edwards
\note invoked by AppFw_Display
*/

HRESULT Base_01::App_Display_Project_In_CAV(LPDIRECT3DDEVICE9 device)
{

	// ---- local
		HRESULT hr;

	// ---- scope
		using namespace vsl_library;

	// ---- d3d (no test for returned value)
		Gfx_D3dx *gfx_d3dx = GetD3dx();
		hr = gfx_d3dx->SetupClientViewport(device);
		hr = gfx_d3dx->SetupProjection(device);
		hr = gfx_d3dx->SetupLookAtLH(device);

	// ---- set default render states and lighting
		hr = gfx_d3dx->SetupRenderDefaults(device);
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- recursively display gfx elements
		Gfx_Element_Engine *gfx_ee = GetElementEngine();
		hr = gfx_ee->Display();
		if (FAILED(hr)) return ERROR_FAIL;

	return SUCCESS_OK;
}
```

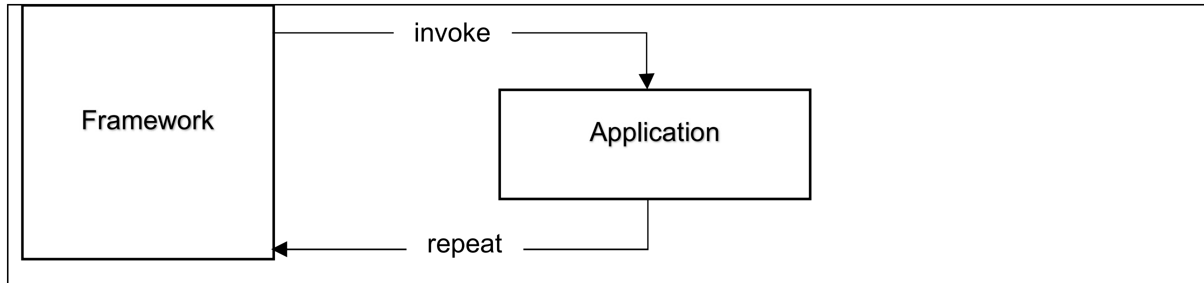
5 Application Framework

5.1 Framework Components

The VSL application framework is comprised of three principal component.

These are:

1. Windows Component
2. Application Interface Component
3. Application Component.



The **Windows Component** is comprised of three principal functions.

These are:

1. Win_Main - the Windows application entry point,
2. Win_Procedure - the Windows function defines most of the behaviour of a window, and
3. Win_Engine – continuous loop handling which principally handles:
   - Windows message pump
   - Invoking application interface methods.

The functions definitions are:

```
INT WINAPI WinMain(
                HINSTANCE hInstance,
                HINSTANCE hPrevInstance,
                LPSTR     lpCmdLine,
                INT       nCmdShow
        );

LRESULT CALLBAppComK Win_Procedure(
                HWND   hWnd,
                UINT   msg,
                WPARAM wParam,
                LPARAM lParam
        );

HRESULT CALLBAppComK Win_Engine();
```

The **Application Interface Component** is comprised of five functions.

These are:

1. AppInt_Setup - application one-time setup.
2. AppInt_SetupDX - application one-time setup DX (DirectX).
3. AppInt_Display – application display.
4. AppInt_CleanupDX - application one-time clean up DX (DirectX).
5. AppInt_Cleanup - application one-time clean up.

These function definitions are:

```
HRESULT AppInt_Setup(VOID);
HRESULT AppInt_SetupDX(VOID);
HRESULT AppInt_Display(VOID);
HRESULT AppInt_CleanupDX(VOID);
HRESULT AppInt_Cleanup(VOID);
```

The **Application Component** is comprised of five application Base class methods that functionally match the Application Interface Component.

These functions are:

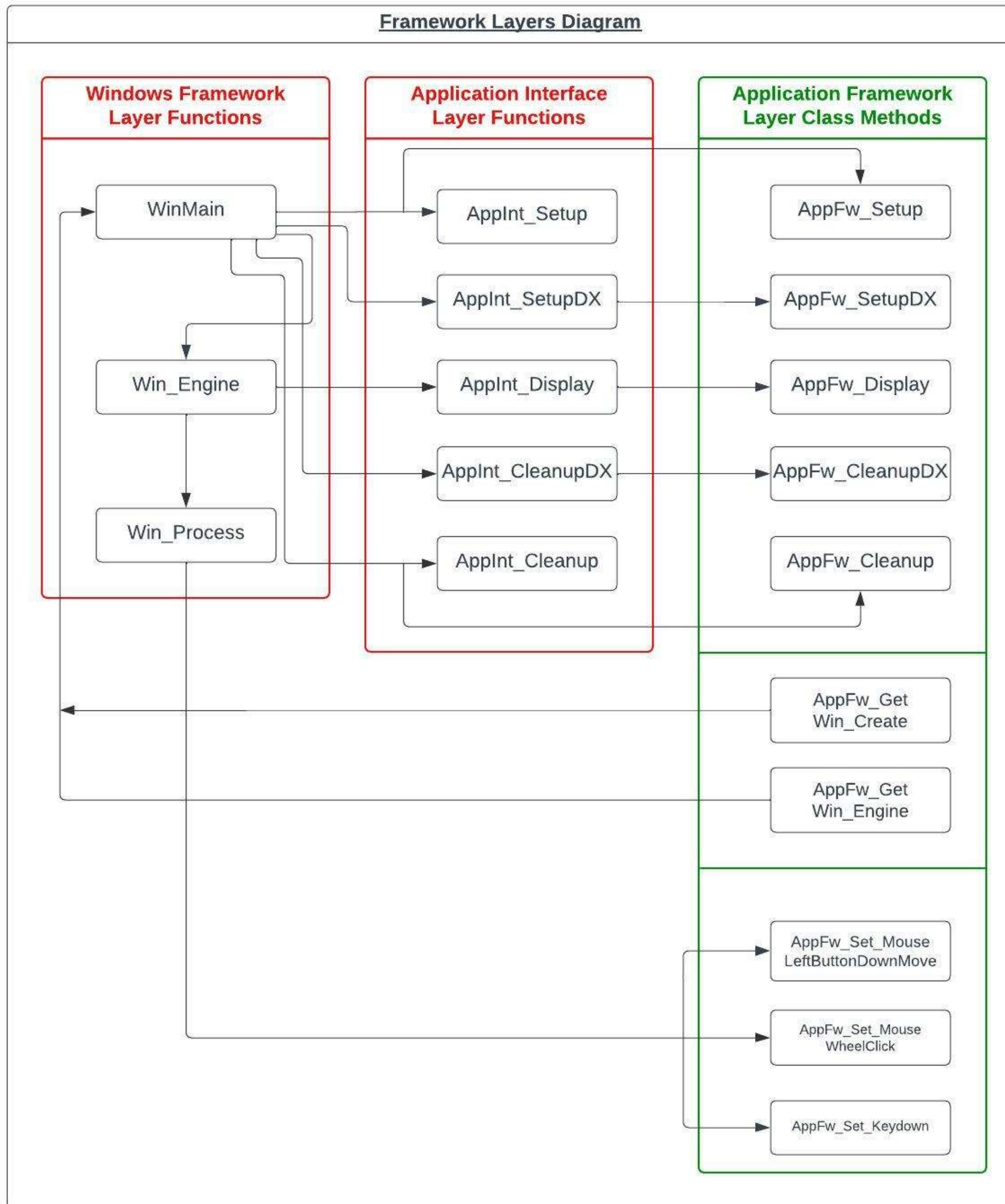
```
HRESULT AppFw_Setup();
HRESULT AppFw_SetupDX(LPDIRECT3DDEVICE9);
HRESULT AppFw_Display(LPDIRECT3DDEVICE9);
HRESULT AppFw_CleanupDX(LPDIRECT3DDEVICE9);
HRESULT AppFw_Cleanup();
```

There are also several “Get” & “Set” methods.

These include:

```
// ---- system struct methods
VOID AppFw_Get_Win_Create(vsl_system::Win_Create **wc);
VOID AppFw_Get_Win_Engine(vsl_system::Win_Engine **wc);

// ---- system notify event methods
VOID AppFw_Set_MouseLeftButtonDownMove(INT, INT);
VOID AppFw_Set_MouseWheelClick(INT);
VOID AppFw_Set_Keydown(WPARAM param);
```

5.2 Diagram: Framework Components



5.3 Windows Component Functions

5.3.1 Win_Main

This function:

- Initialises an instance of a Windows WNDCLASSEX object, and then register this using the Windows RegisterClassEx function. If this fails, then the application exits.

  Note: Win_Procedure function is passed as a member of the WNDCLASSEX object.

- Using the global variable pointer to an instance of an application object "g_app" (see How is a VSL applications selected?), an instance of vsl_system::Win_Create, and vsl_system::Win_Engine are instantiated (for details, see next two sections).

  Note: Declared in in the "vsl_win_framework.h" file.
  ```
        vsl_system::Win_Create *g_win_create;
        vsl_system::Win_Engine *g_win_engine;
  ```

  ```
  g_app->AppFw_Get_Win_Create(&g_win_create);
  g_app->AppFw_Get_Win_Engine(&g_win_engine);
  ```

  Note: Win_Create GetDesktop method used to get desktop status.

- Get & store system metrics (& store in Win_Create structure).

  Note: Win_Create GetClientWidth and GetClientHeight methods used to get client window dimensions (already initialised in application object creator method).

  Note: Win_Create GetCentred method used to get window centred status (already initialised in application object creator method).

- Set window style, location & dimensions (& store in Win_Create structure).

- Using Windows CreateWindowEx function, create, and then register window.

  Note: Win_Create GetName, GetX, GetY, GetRectWidth, and GetRectHeight methods used to get client window parameters (already initialised in application object creator method).

- Do windows housekeeping (including ShowWindow and UpdateWindow)

- Setup the framework, loop message handling (using Win_Engine function) until application exits, and then clean up the framework using following code:

  ```
  // ---- non-device one time application setup
        hr = g_app->AppFw_Setup();
        if (FAILED(hr)) return ERROR_FAIL;

  // ---- non-device one time setup
        hr = AppInt_Setup();
        if (FAILED(hr)) return ERROR_FAIL;

  // ---- device dependent setup
        hr = AppInt_SetupDX();
        if (FAILED(hr)) return ERROR_FAIL;

  // ---- loop, handling messages and application
  ```

```
//      until WM_CLOSE, WM_DESTROY or WM_ESCAPE
        hr = Win_Engine();
        if (FAILED(hr)) return ERROR_FAIL;

// ---- device dependent cleanup
        hr = AppInt_CleanupDX();
        if (FAILED(hr)) return ERROR_FAIL;

// ---- non-device dependant one time cleanup
        hr = AppInt_Cleanup();
        if (FAILED(hr)) return ERROR_FAIL;

// ---- non-device one time application cleanup
        hr = g_app->AppFw_Cleanup();
        if (FAILED(hr)) return ERROR_FAIL;

// ---- unregister windows class
```

5.3.2 Win_Procedure

Windows handles updates to an application by invoking this function (passed as a member of the WNDCLASSEX object.):

```
LRESULT CALLBAppComK Win_Procedure(
        HWND   hWnd,    // a handle to this applications window
        UINT   msg,     // an unsigned integer message token
        WPARAM wParam, // (UINT_PTR) used for passing and returning
                           polymorphic values (typedef UINT_PTR WPARAM)
        LPARAM lParam, // long int (typedef LONG_PTR LPARAM).
)
{ /*- function body -*/ }
```

Using the message token, a switch statement selects an action from the following:

```
// ---- get global vsl_system::Win_Engine *g_win_engine;
        vsl_system::Win_Engine *win_eng;
        g_app->AppFw_Get_Win_Engine(&win_eng);


// ---- switch using message token
        switch( msg )
        {
                case WM_KEYUP: ... break;
                case WM_KEYDOWN: ... break;
                case WM_MOUSEWHEEL: ... break;
                case WM_LBUTTONDOWN: ... break;
                case WM_LBUTTONUP: ... break;
                case WM_MOUSEMOVE: ... break;
                case WM_EXITSIZEMOVE: ... break;
                case WM_SIZE: ... break;
                case WM_CLOSE: ... break;
                case WM_DESTROY: ... break;
        }
```

The following is a synopsis of functionality associated with each of these message tokens.

WM_KEYUP: Set key up flag TRUE.

WM_KEYDOWN: handle VK_ESCAPE (Post Quit message), VK_SHIFT, VK_CLEAR, VK_RETURN and all other keys (pen up / pen down to build a text command line.

WM_MOUSEWHEEL: Set mouse delta +/- 1 and set mouse wheel click flag TRUE.

WM_LBUTTONDOWN: Store, then set new mouse XY position.

WM_LBUTTONUP: Set mouse(ing) false.

WM_MOUSEMOVE: Store, set, and update application framework XY move.

WM_EXITSIZEMOVE: Set update size TRUE (handled by WM_SIZE below)

WM_SIZE: Handle sizing, maximise & minimise.

WM_CLOSE: Post Quit Message.

WM_DESTROY: Post Quit Message.

5.3.3 Win_Engine

Windows passes control to this function from Win_Main until either a WM_CLOSE or WM_DESTROY message token is detected in the Win_Procedure function, which results in a WM_QUIT message token being posted.

If a user selects the ESC(ape) key then a WM_ESCAPE message token is (optionally) handled by the application framework via a pop-up dialogue, which if confirmed results in a WM_QUIT message token being posted.

If the WM_QUIT message token is detected, then control is passed back to the Win_Main function, and the application, after cleanup, is terminated.

This function begins by initialising time variables:

```
// ---- FPS (Frames Per Second) variables:
      BOOL regulated = TRUE;
      UINT last = 0, now = 0;
      UINT fps = g_win_engine.GetFps()

// ---- Millisecond FLOAT variables:
      FLOAT ms_start = (FLOAT)timeGetTime();
      FLOAT ms_now = 0, ms_last = 0 & ms_delta = 0

// ---- Millisecond DOUBLE variables:
      DOUBLE d_ms_interval = (1000/fps);
      DOUBLE d_ms_elapsed_target = interval;
      DOUBLE d_ms_elapsed = 0;

// ---- Seconds DWORD (unsigned long) variables
      DWORD seconds_elapsed_last = 0;
      DWORD seconds_elapsed = 0;
```

Then the main render loop below continues until a WM_QUIT message token is posted (see Win_Procedure function).

```
// ---- message to be processed (see Win_Procedure function above)
      MSG msg;
```

```
        ::ZeroMemory(&msg, sizeof(MSG));

    // ---- loop until completed
        while( msg.message != WM_QUIT )
        {

            // ---- IF there is a message to process THEN translate and
            //      dispatch to Win_Procedure function
            if ( ::PeekMessage( &msg, 0, 0, 0, PM_REMOVE) )
            {
                ::TranslateMessage(&msg);
                ::DispatchMessage(&msg);
            }

            // ---- ELSE
            else
            {
                // ---- NEXT FRAME TEST
                {
                    // ---- DISPLAY NEXT FRAME
                }
            }
        }

    return (HRESULT)msg.wParam;
```

This inner **NEXT FRAME TEST** and **DISPLAY NEXT FRAME** logic (see below) is a complicated set of simple calculations.

The objective is to invoke the application framework display method at discrete intervals which are (almost) exactly (hence use of DOUBLE and tolerances) equal to the frame milliseconds interval, and thus achive a regular Frames Per Second.

Unfortunately this happy state can fail due to computational or graphical load, window resizing, etc.

When this occurs the various various elapsed variables and counters have to be recalculated and/or re-aligned.

The **NEXT FRAME TEST** is:

```
    // ---- NEXT FRAME TEST until current time exceeds next time
    DOUBLE d_ms_get_time = (DOUBLE)timeGetTime();
    d_ms_elapsed = d_ms_get_time - ms_start;
    if (d_ms_elapsed >d_ms_elapsed_target)
    {
        // ---- DISPLAY FRAME
    }
```

If **NEXT FRAME TEST** succeeds, then **DISPLAY NEXT FRAME**.

First – update (recalculate) millisecond & seconds:

```
    // ---- update Win_Engine millisecond elapsed & now
        g_win_engine->SetMsElapsed((FLOAT)d_ms_elapsed);
        g_win_engine->SetMsNow((FLOAT)d_ms_get_time);

    // ---- update milliseconds local last & next, then calculate local delta
        ms_last = ms_now;
```

```
        ms_now = (FLOAT)d_ms_elapsed;
        ms_delta = ms_now - ms_last;

    // ---- update Win_Engine millisecond delta
        g_win_engine->SetMsDelta(ms_delta);

    // ---- update seconds elapsed
        seconds_elapsed = (DWORD)(d_ms_elapsed / 1000);
        g_win_engine->SetSecondsLast(seconds_elapsed_last);
        g_win_engine->SetSecondsNow(seconds_elapsed);

    // ---- DISPLAY NEXT FRAME
        HRESULT hr = Vsl_Display();
        if (FAILED(hr)) throw (1);
```

Second – if next second then regulate, and seconds elapsed last:

```
    // ---- if NEXT second(!) - update seconds & reset fps (frames per second)
        if (seconds_elapsed != seconds_elapsed_last)
        {
            // calc fps regulate status and set Win_Engine fps-regulated
            fps_regulated = fps_now + 1 < fps ? FALSE : TRUE;
            g_win_engine->SetRegulated(fps_regulated);

            // set Win_Engine frames per second now!
            g_win_engine->SetFpsNow(fps_now);

            // update seconds & reset frames per second
            seconds_elapsed_last = seconds_elapsed;
            fps_last = fps_now;
            fps_now = 0;

        }
```

Third – increment fps counter:

```
    // ---- increment fps "this second" count
        fps_now++;
```

Fourth – handle both regulated and unregulated fps:

```
    // *** interval & elapsed MUST be DOUBLE to "exactly"
    //     calculate the in-second millisecond
        if (fps_regulated)
        {
            // calculate the # of intervals to "now"
            DOUBLE d_ms_elapsed_right_now = (DOUBLE)timeGetTime() -
        ms_start;
                DWORD total_ms_intervals_to_now =
                (DWORD)(d_ms_elapsed_right_now / d_ms_interval);

            // add an interval to get (ceiling) target value...
            d_ms_elapsed_target =  (total_ms_intervals_to_now + 1) *
                                                          d_ms_interval;
```

```
}
else
{
	d_ms_elapsed_target = d_ms_elapsed;
}
```

THAT'S it -

5.4 Application Interface Functions

This is comprised of five functions.

These are:

1. `AppInt_Setup`

   Invoke selected application Setup method.

   Application framework setup resources and settings.

   Only called once during the application's initialization phase.

   Therefore, it can't contain any resources that need to be restored every time the Direct3D device is lost, or the window is resized.

2. `AppInt_SetupDX`

   Invoke selected application SetupDX method.

   This function is called when device dependant resources or settings are to be initialised, or restored after losing the Direct3D device OR when the window is resized.

3. `AppInt_Display`

   Application framework display.

   There are four steps:

   1. D3D - Begin Scene
   2. Invoke selected application Display method.
   3. D3D – End Scene
   4. Present (render)

4. `AppInt_CleanupDX`

   Invoke selected application CleanupDX method.

   Called when device D3D dependant resources are to be released.

5. `AppInt_Cleanup`

   Invoke selected application Cleanup method.

   Called once when an application is quit.

5.5 Application Component Classes

There are three application Class variants.

These are:

1. **Stand-alone** (and fully functional) classes.

   As the name suggests, these classes are neither child classes (i.e., they are not derived out of any classes) nor are they parent classes (i.e., they do not act as a base class).

2. **Base** (and fully functional) class.

   A non-abstract & non-concrete class (see notes below; it can be used as a stand-alone), the Base class provides the required Windows & Application Interface functionality, and virtual initialisation, display, and update methods (see Base class interface details).

3. D**erived** from the **Base** class.

   Inherits the Windows & Application Interface functionality: can override the initialisation, display, and update methods; and provide additional functionality.

Notes:

Concrete Base Classes and Concrete Derived Classes

A concrete class has well defined member functions. Their functions are not virtual or pure virtual.

A concrete base class, as the name suggests, is a class which has well defined data members and functions and acts as a base for another class to derive from. A concrete derived class is a concrete class which is derived from the existing base class. It inherits the properties of the base class.

Abstract base class

A class containing the pure virtual function cannot be used to declare the objects of its own .Such classes are known as **abstract base classes.**

The main objective of the abstract base class is to provide the features to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Abstract derived class

A class which is derived from an abstract base class is called as an abstract derived class. The main objective of the abstract derived class is to inherit the features of the abstract base class. The remaining functions are all virtual functions. It is still an abstract class and hence objects of the class cannot be defined.

5.5.1 Stand-Alone Class Definition

These applications require the Application Component methods, and several “Get” & “Set” methods. Each class must be defined in its entirety.

This is the definition for an `Example` class.

```
namespace vsl_application
{

	// --------- Example Class ----------
	class Example
	{

	public:

		// ---- cdtor
		Example ();
		virtual ~ Example ();

		// ---------- FRAMEWORK ----------

		// ---- application component methods
		HRESULT AppFw_Setup();
		HRESULT AppFw_SetupDX(LPDIRECT3DDEVICE9 device);
		HRESULT AppFw_Display(LPDIRECT3DDEVICE9 device);
		HRESULT AppFw_CleanupDX(LPDIRECT3DDEVICE9 device);
		HRESULT AppFw_Cleanup();

		// ---- get methods
		VOID AppFw_Get_Win_Create(vsl_system::Win_Create **fw_win_create);
		VOID AppFw_Get_Win_Engine(vsl_system::Win_Engine **fw_win_engine);

		// ---- set methods
		VOID AppFw_Set_MouseLeftButtonDownMove(INT x, INT y);
		VOID AppFw_Set_MouseWheelClick(INT mouse_wheel_click_z);
		VOID AppFw_Set_Keydown(WPARAM param);

		// ---------- APPLICATION methods----------

	private:

		// ---- system structs
		vsl_system::Win_Create fw_win_create;
		vsl_system::Win_Engine fw_win_engine;

	public:

		// ---- More application methods, e.g.:
		VOID TeapotsSetup(LPDIRECT3DDEVICE9);
		VOID TeapotsDisplay(LPDIRECT3DDEVICE9, D3DXMATRIX, FLOAT, INT);
			...
	};

}
```

5.5.2 Base Class Definition

```
namespace vsl_application
{

	// ---------- Base_01 class ----------
	class Base_01
	{

	public:

		// ---- cdtor
		Base_01(VOID);
		virtual ~Base_01();

		// ---------- APPLICATION COMPONENT ----------

		// ---- required by component
		HRESULT AppFw_Setup(VOID);
		HRESULT AppFw_SetupDX(LPDIRECT3DDEVICE9);
		HRESULT AppFw_Display(LPDIRECT3DDEVICE9);
		HRESULT AppFw_CleanupDX(LPDIRECT3DDEVICE9);
		HRESULT AppFw_Cleanup(VOID);

		// ---- required by component: get wnd structs
		VOID AppFw_Get_Win_Create(vsl_system::Win_Create **fw_win_create);
		VOID AppFw_Get_Win_Engine(vsl_system::Win_Engine **fw_win_engine);

		// ---- required by component to set Gfx_Command parameter
		VOID AppFw_Set_MouseLeftButtonDownMove(INT x, INT y);
		VOID AppFw_Set_MouseWheelClick(INT mouse_wheel_click_z);
		VOID AppFw_Set_Keydown(WPARAM param);

		// ---------- APPLICATION ----------

		// ---- invoked by AppFw_Setup
		virtual HRESULT App_Initialise_Frameset(VOID);
		virtual HRESULT App_Initialise_Project(VOID);
		virtual HRESULT App_Initialise_Element_Configurations(VOID);
		virtual HRESULT App_Initialise_Element_Coordinates(VOID);
		virtual HRESULT App_Initialise_Element_Components(VOID);

		// ---- invoked by AppFw_Display
		virtual HRESULT App_Display_Project_In_CAV(LPDIRECT3DDEVICE9);
		virtual HRESULT App_Display_Project_In_Frameset(
				LPDIRECT3DDEVICE9 device,
				vsl_library::Gfx_Frame *frame, UINT level);

		// ---- invoked by AppInt_Display
		virtual VOID App_Update_Element_Bookmarks(VOID);
		virtual VOID App_Update_By_Command(VOID);
		virtual VOID App_Update_By_AsyncKeys(VOID);
		virtual VOID App_Update_Cav_Text(VOID);
```

```
		// ---------- PRIVATE IMPLEMENTATION ----------

		// ---- get Win objects
		vsl_system::Win_Create *Get_Win_Create(VOID);
		vsl_system::Win_Engine *Get_Win_Engine(VOID);

		// ---- get gfx objects
		vsl_library::Gfx_Command        *GetCommand(VOID);
		vsl_library::Gfx_D3dx           *GetD3dx(VOID);
		vsl_library::Gfx_Log            *GetLog(VOID);
		vsl_library::Gfx_Frameset       *GetFrameset(VOID);
		vsl_library::Gfx_Element_Engine *GetElementEngine(VOID);

	private:

		// ---- private implementation
		class Pimpl_Base_01;
		std::unique_ptr<Pimpl_Base_01> pimpl_base_v01;

	public:

		// ---- get private implementation
		//
		// usage: std::unique_ptr<Pimpl_Base_01> *pimp = GetPimp(VOID);
		//
		// std::unique_ptr<Pimpl_Base_01>
		//  *GetPimp(VOID) { return &pimpl_base_v01; }

	};

}
```

Note 1: Virtual methods - are declared within the base class and are (or can be) re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Note 2: Private Implementation - This technique is used to construct C++ library interfaces with stable ABI and to reduce compile-time dependencies. "Pointer to implementation" or "pimpl" is a C++ programming technique that removes implementation details of a class from its object representation by placing them in a separate class, accessed through an opaque pointer:

Note 3: shared_ptr - type is a smart pointer in the C++ standard library that is designed for scenarios in which more than one owner might have to manage the lifetime of the object in memory. After you initialize a shared_ptr you can copy it, pass it by value in function arguments, and assign it to other shared_ptr instances.

All the instances point to the same object and share access to one "control block" that increments and decrements the reference count whenever a new shared_ptr is added, goes out of scope, or is reset. When the reference count reaches zero, the control block deletes the memory resource and itself.

Note 4: Classes derived from the `Base_01` class use a combination of shared pointer AND private implementation to hide AND manage the lifecycle of the private implementation.

5.5.3 Example Derived Class Definition

All Application Component classes (other than `Base_01`!) must be derived from `Base_01`.

All Application Component classes MUST be derived from this class.

A derived class is a class that is constructed from a base class or an existing class. It tends to acquire all the methods and properties of a base class. It is also known as a subclass or child class.

An example of a derived application class.

```
namespace vsl_application
{

	// ---- derived class
	class Darkland_01 : public Base_01
	{

		public:

		// ---------- cdtor ----------
		Darkland_01::Darkland_01(VOID);
		Darkland_01::~Darkland_01(VOID);

		// ---------- APPLICATION ----------

		// ---- invoked by AppFw_Setup – setup
		HRESULT App_Initialise_Frameset(VOID) override;
		HRESULT App_Initialise_Project(VOID) override;
		HRESULT App_Initialise_Element_Configurations(VOID) override;
		HRESULT App_Initialise_Element_Coordinates(VOID) override;
		HRESULT App_Initialise_Element_Components(VOID) override;

		// ---- invoked by App_Display
		VOID App_Update_Element_Bookmarks(VOID) override;
		VOID App_Update_By_AsyncKeys(VOID) override;
		VOID App_Update_Cav_Text(VOID) override;

		// ---- properties
		BOOL text_mode = FALSE;

	};

}
```

5.5.4 Selecting an Application Class

When a VSL program is compiled the C/C++ Pre-Processor (CPP) is used to pre-select a specific application. The CPP is not a part of the compiler but is a separate first step in the compilation process. In simple terms, CPP is a text substitution tool.

This substitution, swapping one application for another, is only possible because all application class definitions are required to possess a minimum common set of identically named, and functionally equivalent, methods.

The selection is defined in the "vsl_application\shared\header\vsl_select.h".

The selection process consists of three steps:

1. First define all application class identifier replacements.
2. Set selected application class identifier.
3. Create an instance of that class and initialise a WndCom global application pointer "g_app".

For example:

```
// ---- define application identifier & replacement
...
#define VSL_Surface_01  6 // pyramidal rhombic dodecahedron
#define VSL_Darkland_01 7 // test project

// ---- set application identifier
#define VSL_APP VSL_Darkland_01

// ---- set global application pointer "g_app"
#if VSL_APP == [...]
...
#elif VSL_APP == VSL_Surface_01
#include "../../vsl_application/framework/header/vsl_base_01.h"
#include "../../vsl_application/framework/header/vsl_surface_01.h"
vsl_application::Surface_01 *g_app = new vsl_application::Surface_01();
#elif VSL_APP == VSL_Darkland_01
#include "../../vsl_application/framework/header/vsl_base_01.h"
#include "../../vsl_application/framework/header/vsl_darkland_01.h"
vsl_application::Darkland_01 *g_app = new vsl_application::Darkland_01();
#endif
```

Note, in step 3 example above:

> The definition of the base class `Base_01`, from which other application classes are derived, is included, then the definition of the derived class `Darkland_01` is included.

5.5.5 Adding a new derived Application Class

1. Choose an application derived class that is a "best" fit.

2. Choose a "good" appropriate name for the application class:

   e.g., The “Virtual Conveyor Belt” class might be VCBelt (easier to type).

3. Copy “vsl_application/framework/header/["best" fit]_[#].h” to “vsl_application/framework/header/[new name]_[#]”.

   e.g., “copy vsl_application/framework/header/vsl_darkland_01.h” to “vsl_application/framework/header/vsl_vcbelt_01.h”.

4. Copy “vsl_application/framework/header/["best" fit]_[#].cpp” to “vsl_application/framework/header/[new name]_[#].cpp”.

   e.g., copy “vsl_application/framework/header/vsl_darkland_01.cpp” to “vsl_application/framework/header/vsl_vcbelt_01.cpp”

5. Add these to the MS VS filters “vsl_application/framework/header” & “vsl_application/framework/source”.

6. Edit the “[name].h” file and replace old class name with new.

7. Edit the “name].cpp” file and replace old class name with new.

8. In the “[name].cpp” file, rename app, e.g.:

```
HRESULT VCBelt_01::App_Initialise_Frameset(VOID)
{ . . .
        // ---- rename
        vsl_system::Win_Create *win_cr8 = Get_Win_Create();

        FROM: win_cr8->SetName("Darkland 01");

        TO: win_cr8->SetName("VCBelt 01");
. . . }
```

9. In the “vsl_application/framework/shared/vsl_select.h”, add a new token string

```
#define VSL_VCBelt_01   8 // virtual conveyor belt project
```

10. In the same file add the CPP switch

```
        #elif VSL_APP == VSL_VCBelt_01
```

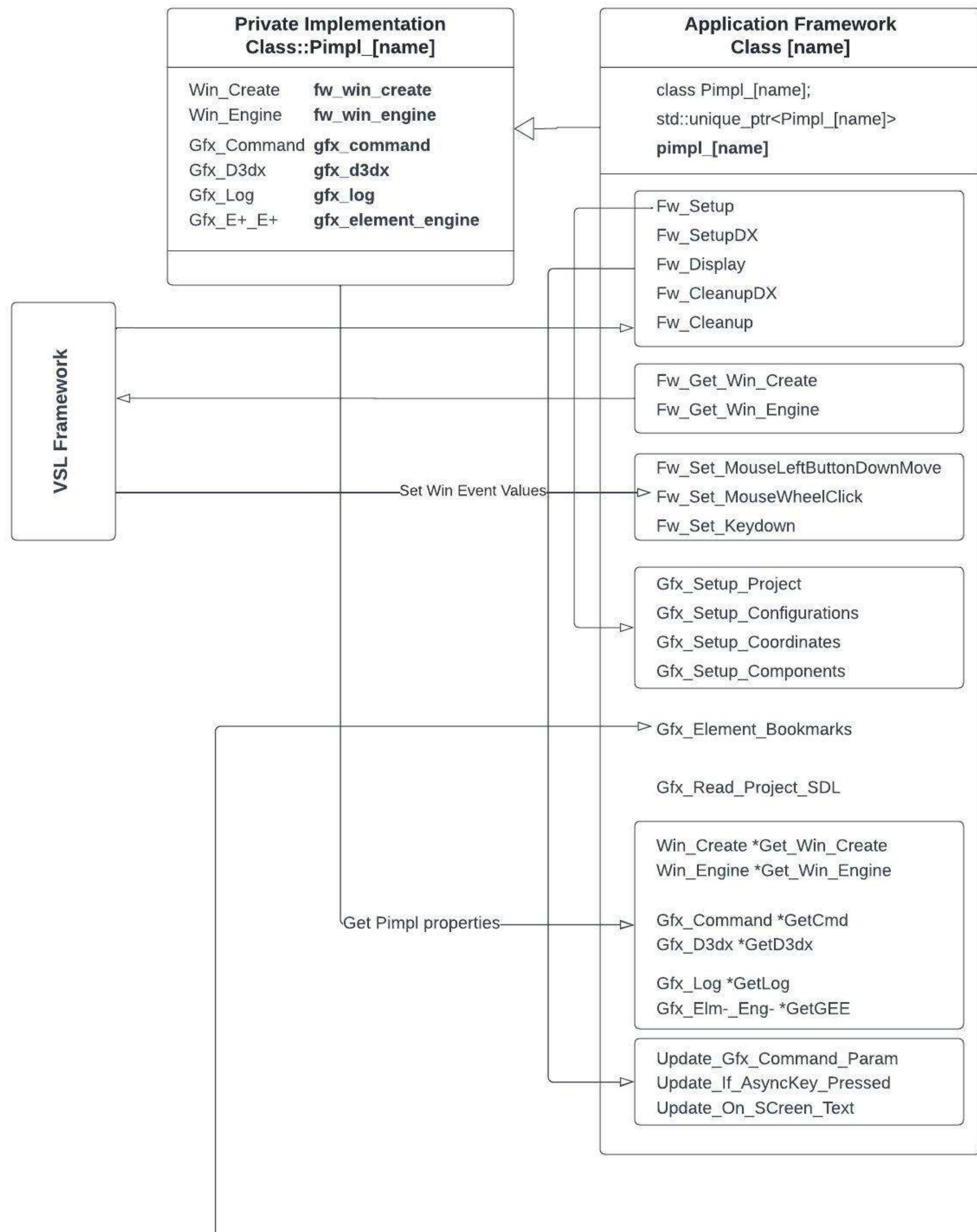
11. Then the code to include the base class header file and the derived class header file.

```
                #include "../../vsl_application/framework/header/vsl_base_01.h"
                #include "../../vsl_application/framework/header/vsl_vcbelt_01.h”
```

12. Then the code to instantiate a new instance of this class, and set global pointer.

```
                vsl_application::VCBelt_01 *g_app = new vsl_application::VCBelt_01();
```

5.6 Diagram: Application Component Class

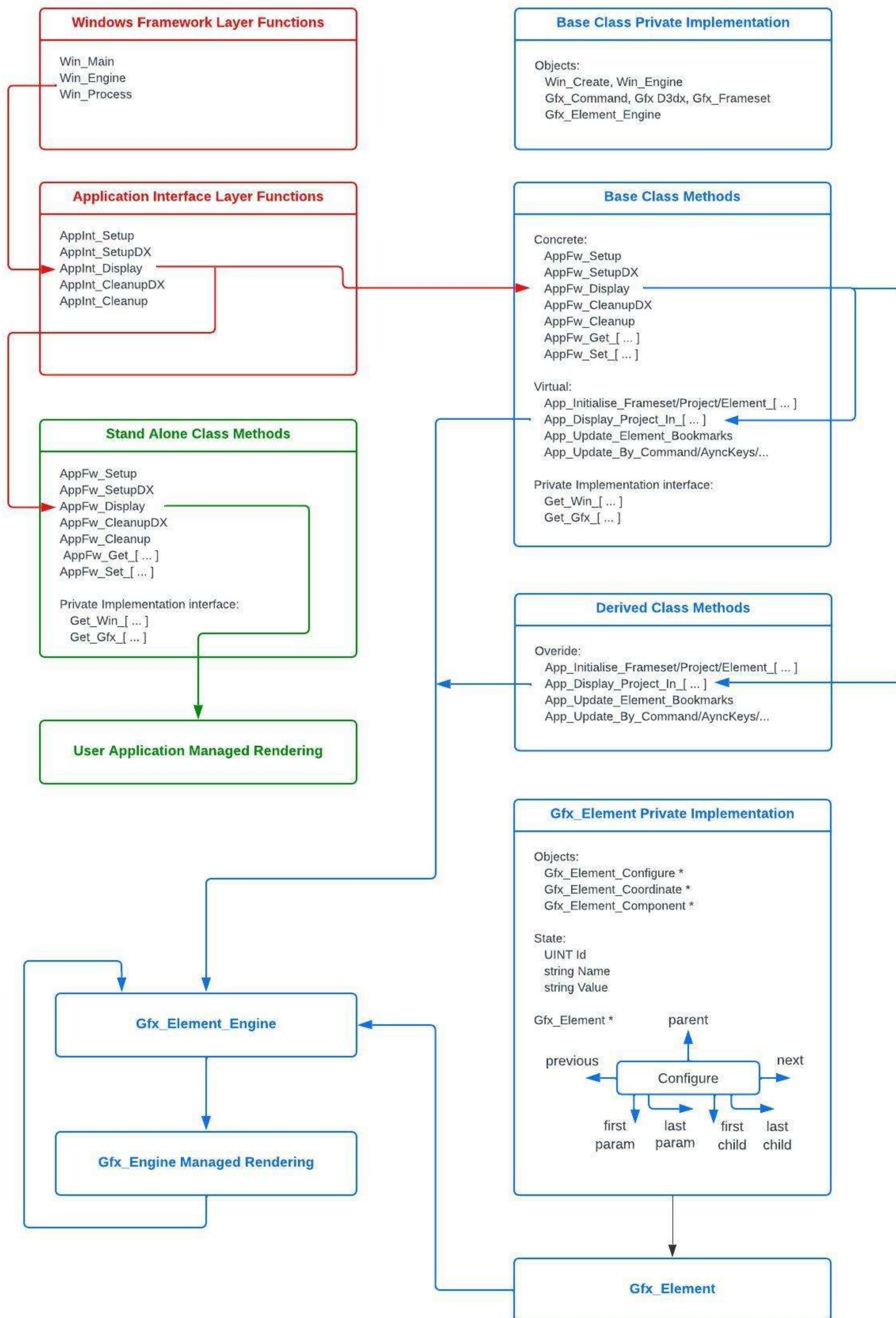


Note: Gfx_Element_Bookmarks handle sgfx elements that have been bookmarked.

Note: The pimpl_[name] unique_ptr<> template holds a pointer to an object and deletes this object when the unique_ptr<> object is deleted. The unique_ptr<> destructor is always called and therefore the object (int in the example) always deleted.

See: Herb Sutter's Exceptional C++.

5.7 Diagram: Application Component Class Display



6 Application Base Class

6.1 Overview

The Base class definition is given above.

The definition can be divided into seven conceptual blocks.

These are:

- Concrete:
  1. Application Component methods invoked by the Application Interface
  2. Get setup struct methods invoked by the Windows Component
  3. Set mouse & keyboard update methods invoked by the Windows Component

- Virtual methods invoked by Application Component methods:
  4. Initialisation invoked at run-time.
  5. Display invoked each frame (e.g., 60 times a second).
  6. Update invoked each frame (ditto).

- Private Implementation (PImpl)
  7. Initialised by Base class constructor.

The whole of this section details implementation.

Note: Leveraging the VSL Framework design, the supplied application Base class could be replaced with another (though ALL the non-virtual methods would need to be replicated “as is” to provide the necessary Windows and Application Interface functionality); and a new “family” of derived application classes could be constructed.

6.2 Private Implementation (PImpl)

"Pointer to implementation" or "pImpl" is a C++ programming technique that removes implementation details of a class from its object representation by placing them in a separate class, accessed through an opaque pointer.

This technique is used to construct C++ library interfaces with stable ABI (Application Binary Interface) and to reduce compile-time dependencies.

Because private data members of a class participate in its object representation, affecting size and layout, and because private member functions of a class participate in overload resolution (which takes place before member access checking), any change to those implementation details requires recompilation of all users of the class.

pImpl removes this compilation dependency; changes to the implementation do not cause recompilation. Consequently, if a library uses pImpl in its ABI, newer versions of the library may change the implementation while remaining ABI-compatible with older versions.

The alternatives to the pImpl idiom are:

- inline implementation: private members and public members are members of the same class.
- pure abstract class (OOP factory): users obtain a unique pointer to a lightweight or abstract base class; the implementation details are in the derived class that overrides its virtual member functions.

Runtime overhead:

7. Access
8. Space
9. Lifetime management

6.2.1 Definition

Comprising constructor, windows framework structs, and Gfx objects.

The framework structs are system objects (hence vs_system namespace).

The Gfx objects are library objects (hence vs_library namespace).

```
class Base_01::Pimpl_Base_01
{

public:

	// ---- cdtor
		Pimpl_Base_01() {}

	// ---- required by windows framework:
		vsl_system::Win_Create fw_win_create;
		vsl_system::Win_Engine fw_win_engine;

	// ---- required gfx:
		vsl_library::Gfx_Command  gfx_command;
		vsl_library::Gfx_D3dx     gfx_d3dx;
		vsl_library::Gfx_Frameset gfx_frameset;
		vsl_library::Gfx_Log      gfx_log;

		vsl_library::Gfx_Element_Engine gfx_element_engine;

};
```

It is declared in the Base class definition.

```
private:

// ---- private implementation
	class Pimpl_Base_01;
	std::unique_ptr<Pimpl_Base_01> pimpl_base_v01;
```

This is initialised when an instance of the Base class is instantiated, and automatically deleted when this instance of the Base class is destroyed.

See the next section “Deleter” for more information.

6.2.2 Deleter

The deleter for a shared pointer is created here:

```
Widget::Widget(): pImpl(new Impl) {}
```

Until that point, all the shared pointer has is the equivalent of a `std::funciton<VOID(Impl*)>`.

When you construct a shared_ptr with a T*, it writes a deleter and stores it in the `std::function` equivalent.

At that point the type must be complete.

So the only functions you have to define after `Impl` is fully defined are those that create a pImpl from a T* of some kind.

Note: The `unique_ptr<>` template holds a pointer to an object and deletes this object when the `unique_ptr<>` object is deleted.

So, in the example above, it does not matter if the function scope is left through the return statement, at the end of the function or even through an exception: The `unique_ptr<>` destructor is always called and therefore the object (int in the example) always deleted.

Note: deleted when it goes out of scope.

See: Herb Sutter's Exceptional C++.

The Base class:

```
// ---- base class interface
       class Base_01
       {
       public:
              Base_01();
       private:
              class Pimpl_Base_01;
              std::unique_ptr<Pimpl_Base_01> pimpl_base_v01;
       };
```

The Base Private Implemention class:

```
// ---- base class implementation derived from private implemetation
       class Base_01::Pimpl_Base_01
       {
              // ---- stuff
       };
```

The Base class constructor:

```
// ---- constructor initialises private implemetation
       Base_01::Base_01(VOID) : pimpl_base_v01(new Pimpl_Base_01)
       {
              // ---- stuff
       }
```

6.3 Constructor & Setup

```
Base_01::Base_01(VOID) : pimpl_base_v01(new Pimpl_Base_01)
{

 // ---- local
        HRESULT hr;

 using namespace vsl_system;

 // ---- Windows Component

 // ---- Win_Create - get & initialise window create struct
        Win_Create *win_cr8 = Get_Win_Create();
        win_cr8->SetName("Base_01");
        win_cr8->SetCentred(FALSE);
        win_cr8->SetDesktop(FALSE);
        win_cr8->SetClient(800,600);
        win_cr8->SetAaq(4);

 // ---- Get_Win_Engine - get & initialise windows engine struct
        Win_Engine *win_eng = Get_Win_Engine();
        win_eng->SetColour(92, 92, 92);
        win_eng->SetFps(60);

 using namespace vsl_library;

 // ---- Graphics (Gfx)

 // ---- Command (IO)
        Gfx_Command *gfx_command = GetCommand();
        gfx_command->SetDefaultMouseWheelClick(-50);
        hr = gfx_command->SetToggle((CHAR)'T');

 // ---- D3dx (Direct 3D)
        Gfx_D3dx *gfx_d3dx = GetD3dx();
        gfx_d3dx->Set_Win_Create(win_cr8);
        gfx_d3dx->Set_Win_Engine(win_eng);
        gfx_d3dx->Set_Gfx_Command(gfx_command);

 // ---- Gfx_Log
        Gfx_Log *gfx_log = GetLog();
        gfx_log->SetShowLineNumber(TRUE);
        gfx_log->SetShowDate(FALSE);
        gfx_log->SetShowTime(TRUE);
        gfx_log->SetShowSimple(FALSE);

 // ---- Gfx_Log - write this app name
        hr = gfx_log->WriteBanner(win_cr8->GetName());

 // ---- Gfx_Element_Engine
        Gfx_Element_Engine *gfx_gee = GetElementEngine();
        gfx_gee->SetGfxLog(gfx_log);
 }

 Base_01::~Base_01()
 { ; }
```

6.4 Framework Methods

6.4.1 Setup

```
// ---------- AppFw_Setup ----------
/*!
\brief required by application interface: setup
\author Gareth Edwards
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_Setup(VOID)
{

	// ---- local
		HRESULT hr;

	// ---- scope
	using namespace vsl_library;

	// ---- device
		hr = GetD3dx()->Setup();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- setup & config hierarchical frameset
		hr = App_Initialise_Frameset();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- setup & config hierarchical project structure
		hr = App_Initialise_Project();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- (one-time) recursively setup elements,
		element instancing & verify kandinsky component
		hr = GetElementEngine()->Setup();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- (one-time) recursively setup element coordinates
		hr = App_Initialise_Element_Coordinates();
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- (one_time) recursively setup element components
		hr = App_Initialise_Element_Components();
		if (FAILED(hr)) return ERROR_FAIL;

	return SUCCESS_OK;
}


// ---------- AppFw_SetupDX ----------
/*!
\brief required by application interface: setup dx
\author Gareth Edwards
\param LPDIRECT3DDEVICE9 - pointer to an IDirect3DDevice9 structure
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_SetupDX(LPDIRECT3DDEVICE9 device)
{

	// ---- local
		HRESULT hr;
```

```
	// ---- d3d
		hr = GetD3dx()->SetupDX(device);
		if (FAILED(hr)) return hr;

	// ---- setup DX gfx element engine
		hr = GetElementEngine()->SetupDX(device);
		if (FAILED(hr)) return hr;

	// --- ignore – test or demonstration
		if (FALSE)
		{
			using namespace vs_library;

			// ---- create
			DotObjElement *object = new DotObjElement();

			// ---- read
				CHAR *filename = "object\\teapot\\teapot.obj";
				vs_library::DotObjUtilities dot_obj_utilities;
				HRESULT hr = (HRESULT)dot_obj_utilities.Read(
									object,
									filename);
				if (SUCCEEDED(hr))
				{
					dot_obj_utilities.Report(object, filename);
				}
		}

	return SUCCESS_OK;
}
```

6.4.2 Display

```
// ---------- AppFw_Display ----------
/*!
\brief required by application interface: Display
\author Gareth Edwards
\param LPDIRECT3DDEVICE9 - pointer to an IDirect3DDevice9 structure
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_Display(LPDIRECT3DDEVICE9 device)
{

	// ---- local
		HRESULT hr;

	// ---- clear backbuffers
		hr = GetD3dx()->Clear(device);

	// ---- display application
		hr = App_Display(device);

	// ---- user update project
		App_Update_By_Command();
		App_Update_By_AsyncKeys();
		App_Update_Element_Bookmarks();

	// ---- display within frameset OR within CAV
		vsl_library::Gfx_Frameset *gfx_frameset = GetFrameset();
		vsl_library::Gfx_Frame *frame = gfx_frameset->GetFirst();
		if ( frame != NULL )
			hr = App_Display_Project_In_Frameset(device, frame, 0);
		else
			hr = App_Display_Project_In_CAV(device);

		App_Update_Cav_Text();

	return SUCCESS_OK;
}
```

6.4.3 Cleanup

```
// ---------- AppFw_CleanupDX ----------
/*!
\brief required by application interface: CleanupDX
\author Gareth Edwards
\param LPDIRECT3DDEVICE9 - pointer to an IDirect3DDevice9 structure
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_CleanupDX (LPDIRECT3DDEVICE9 device)
{

	// ---- local
		HRESULT hr;

	//---- d3d
		hr = GetD3dx()->CleanupDX(device);
		if (FAILED(hr)) return ERROR_FAIL;

	// ---- cleanup gfx element engine
		hr = GetElementEngine()->CleanupDX();
		if (FAILED(hr)) return ERROR_FAIL;

	return SUCCESS_OK;
}


// ---------- AppFw_Cleanup ----------
/*!
\brief required by framework: cleanup
\author Gareth Edwards
\return HRESULT (SUCCESS_OK if ok)
*/
HRESULT Base_01::AppFw_Cleanup(VOID)
{

// ---- cleanup gfx element engine
	HRESULT hr = GetElementEngine()->Cleanup();
	if (FAILED(hr)) return ERROR_FAIL;

	return SUCCESS_OK;
}
```

6.4.4 Get

```
// ---------- AppFw_Get_Win_Create ----------
/*!
\brief required by application interface: get pointer to wnd create struct
\author Gareth Edwards
\param Win_Create **wc
*/
VOID Base_01::AppFw_Get_Win_Create(vsl_system::Win_Create **fw_win_create)
{
	*fw_win_create = Get_Win_Create();
}


// ---------- AppFw_Get_Win_Create ----------
/*!
\brief required by application interface: get pointer to wnd engine struct
\author Gareth Edwards
\param Win_Engine **wc
*/
VOID Base_01::AppFw_Get_Win_Create(vsl_system::Win_Engine **fw_win_engine)
{
	*fw_win_engine = Get_Win_Engine();
}
```

6.4.5 Set

```
// ---------- AppFw_Set_MouseLeftButtonDownMove ----------
/*!
\brief required by application interface: if mouse left button down then update move x
y
\author Gareth Edwards
\param FLOAT x
\param FLOAT y
*/
VOID Base_01::AppFw_Set_MouseLeftButtonDownMove(INT x, INT y)
{
	GetCommand()->SetMouseLeftButtonDownMove((FLOAT)x, (FLOAT)y);
}


// ---------- AppFw_Set_MouseWheelClick ----------
/*!
\brief required by application interface: if mouse wheel click event update move z
\author Gareth Edwards
\param intd d (+/- 1)
*/
VOID Base_01::AppFw_Set_MouseWheelClick(INT d)
{
	FLOAT mouse_wheel_click = GetCommand()->GetMouseWheelClick();
	mouse_wheel_click += (FLOAT)d;

	GetCommand()->SetMouseWheelClick(mouse_wheel_click);
}


// ---------- AppFw_Set_Keydown ----------
/*!
\brief required by application interface: if key down handle toggle, etc.
\author Gareth Edwards
\param WPARAM (parameter)
*/
VOID Base_01::AppFw_Set_Keydown(WPARAM param)
{
	GetCommand()->SetToggle((CHAR)param);
}
```

6.5 Virtual Methods

6.5.1 Initialise

Invoked by AppFw_Setup.

```
virtual HRESULT App_Initialise_Frameset(VOID);
virtual HRESULT App_Initialise_Project(VOID);
virtual HRESULT App_Initialise_Element_Configurations(VOID);
virtual HRESULT App_Initialise_Element_Coordinates(VOID);
virtual HRESULT App_Initialise_Element_Components(VOID);
```

6.5.1.1 Frameset

```
// ---------- App_Initialise_Project --------
/*!
\brief on time initialisation of frameset
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Frameset(VOID)
{

        // ---- name
        vsl_system::Win_Create *win_cr8 = Get_Win_Create();
        win_cr8->SetName("Base");

        // ---- Gfx_Frameset
        using namespace vsl_library;
        Gfx_Frameset *gfx_frameset = GetFrameset();
        gfx_frameset->SetDimensions(
                win_cr8->GetClientWidth(),
                win_cr8->GetClientHeight());

        return SUCCESS_OK;
}
```

6.5.1.2 Project

```
// ---------- App_Initialise_Project --------
/*!
\brief on time initialisation of project
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Project(VOID)
{

	// ---- scope
	using namespace vsl_library;

	// ---- stop gap for "read file"
	try
	{
		// ---- build project
		Gfx_Element *engine_root_element = NULL;
		if (engine_root_element = GetElementEngine()->GetEngineRoot())
		{
			if (Gfx_Element *project = engine_root_element->Find("Project"))
			{
				// ---- element to be instanced
				Gfx_Element *element_tr0 = project->Append("TR0", 0);

				Gfx_Element_Configure *configure = element_tr0->GetConfigure();
				configure->SetComponentName("PyRhoDo_VBO");
				configure->SetVisible(FALSE);

				// ---- group id 1 : element id 4 (or could be 3!)
				Gfx_Element *element_group_1 = project->Append("G1", 1);
				GetElementEngine()->AddBookMark(element_group_1);
				{
					Gfx_Element *element_tr1 = element_group_1->Append("TR1", 4);
					GetElementEngine()->AddBookMark(element_tr1);

					Gfx_Element_Configure *configure =
						element_tr1->GetConfigure();
					configure->SetComponentName("Cuboid_VBO");
					configure->SetVisible(TRUE);
					configure->SetInstance(FALSE);
					configure->SetInstanceName(element_tr0->GetName());
				}

				// ---- group id 2 : element id 4
				Gfx_Element *element_group_2 = project->Append("G2", 2);
				GetElementEngine()->AddBookMark(element_group_2);
				{
					Gfx_Element *element_tr2 = element_group_2->Append("TR2", 4);
					GetElementEngine()->AddBookMark(element_tr2);

					Gfx_Element_Configure *configure =
						element_tr2->GetConfigure();
					configure->SetComponentName("Cuboid_VIBO");
					configure->SetVisible(TRUE);
					configure->SetInstance(TRUE);
					configure->SetInstanceName(element_tr0->GetName());
				}
			}
			else throw("Project");
		}
		else throw("Engine");

		// ---- report
		engine_root_element->List();

	}
	catch (CHAR *element_name)
	{
		CHAR msg[128];
		sprintf_s(msg, 128, "Element %s not found!\n Select OK to Exit", element_name);
		INT msgboxID = MessageBox(NULL, msg, "VSL method: Base_01::AppFw_Setup()",
			MB_ICONWARNING);
```

```
                return FALSE;
        }

        return SUCCESS_OK;
}
```

6.5.1.3 Configurations

```
// ---------- App_Initialise_Element_Configurations --------
/*!
\brief one time initilisation of project element configurations
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Element_Configurations(VOID)
{
        return SUCCESS_OK;
}
```

6.5.1.4 Coordinates

```
// ---------- App_Initialise_Element_Coordinates --------
/*!
\brief one time initialisation of project coordinates
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Element_Coordinates(VOID)
{

        // ---- scope
        using namespace vsl_library;

        // ---------- update element component properties & parameters ----------
        Gfx_Element *engine_root_element = GetElementEngine()->GetEngineRoot();

        // ---------- update element coordinate properties ----------
        Gfx_Element *element_tr1 = engine_root_element->Find("TR1");
        if (element_tr1 != NULL)
        {
                Gfx_Element_Coordinate *coordinate = element_tr1->GetCoordinate();
                vsl_system::Vsl_Vector3 v = { 1, 2, 1 };
                coordinate->SetScale(vsl_system::Vsl_Vector3(1, 2, 1));
        }

        return SUCCESS_OK;
}
```

6.5.1.5 Components

```
// ----------  App_Initialise_Element_Components --------
/*!
\brief one time initialisation of project components
\author Gareth Edwards
*/
HRESULT Base_01::App_Initialise_Element_Components(VOID)
{

	// ---- scope
	using namespace vsl_library;

	// ---------- update element properties & parameters ----------
	Gfx_Element *engine_root_element = GetElementEngine()->GetEngineRoot();
	Gfx_Element *element_tr0 = engine_root_element->Find("TR0");
	if (element_tr0 != NULL)
	{
		// ---- PyRhoDo_VBO
		Gfx_Element_Component *component = element_tr0->GetComponent();
		Gfx_Kandinsky *kandinsky = component->GetKandinsky();

		// ---- build - include inside -> update element component properties
		kandinsky->Set(Gfx_Kandinsky_Param::INSIDE, 1);

		// ---- move -> update both component & corresponding kandinsky parameters

		// ---- get group
		Gfx_Element *element_param_group = element_tr0->FindParamGroup("Component");
		if (element_param_group)
		{
			// ---- set parameter
			std::string move = "0.5";
			HRESULT hr =
				element_param_group->SetParameterValue("Dimension", "Move", move);

			// ---- if OK? - set kandinsky
			if (SUCCEEDED(hr))
			{
				kandinsky->SetParameterValue("Dimension", "Move", move);
			}
		}

	}
	return SUCCESS_OK;
}
```

6.5.2 Display

Invoked by `AppFw_Display`.

6.5.2.1 In Client Adjusted Viewport

```
// ---------- App_Display_In_CAV --------
/*!
\brief display elements in Client Adjusted Viewport
\author Gareth Edwards
\note invoked by AppFw_Display
*/

HRESULT Base_01::App_Display_Project_In_CAV(LPDIRECT3DDEVICE9 device)
{

        // ---- local
                HRESULT hr;

        // ---- scope
                using namespace vsl_library;

        // ---- d3d (no test for returned value)
                Gfx_D3dx *gfx_d3dx = GetD3dx();
                hr = gfx_d3dx->SetupClientViewport(device);
                hr = gfx_d3dx->SetupProjection(device);
                hr = gfx_d3dx->SetupLookAtLH(device);

        // ---- set default render states and lighting
                hr = gfx_d3dx->SetupRenderDefaults(device);
                if (FAILED(hr)) return ERROR_FAIL;

        // ---- recursively display gfx elements
                Gfx_Element_Engine *gfx_ee = GetElementEngine();
                hr = gfx_ee->Display();
                if (FAILED(hr)) return ERROR_FAIL;

        return SUCCESS_OK;
}
```

6.5.2.2 In A Frameset

```
// ---------- App_Display_Project_In_Frameset --------
/*!
\brief display elements in frameset
\author Gareth Edwards
\note invoked by AppFw_Display
*/

HRESULT Base_01::App_Display_Project_In_Frameset(
                LPDIRECT3DDEVICE9 device,
                vsl_library::Gfx_Frame *frame,
                UINT level
        )
{

        using namespace vsl_library;

        // ---- local
        HRESULT hr;
        Gfx_Log *gfx_log = GetLog();
        Gfx_D3dx *gfx_d3dx = GetD3dx();
        Gfx_Element_Engine *gfx_ee = GetElementEngine();

        while (frame != NULL)
        {

                if (frame->GetActive())
                {

                        if (frame->GetVisible())
                        {

                                // ---- render frame chrome & clear frame
                                hr = gfx_d3dx->DrawFrame(device, frame);

                                // ---- setup display
                                hr = gfx_d3dx->SetupPerspectiveFovLH(device, frame);
                                hr = gfx_d3dx->SetupLookAtLH(device);

                                // ---- setup default render states and lighting
                                hr = gfx_d3dx->SetupRenderDefaults(device);
                                if (FAILED(hr)) return ERROR_FAIL;

                                // ---- recursively display gfx elements
                                hr = gfx_ee->Display();
                                if (FAILED(hr)) return ERROR_FAIL;

                                }

                                if (Gfx_Frame *child = frame->GetFirst())
                                {
                                        App_Display_Project_In_Frameset(device, child, level + 1);
                                }
                        }

                        // ---- next
                        frame = frame->GetNext();

                }

                // ---- restore client adjusted viewport....
                Gfx_D3dx *d3dx = GetD3dx();
                hr = d3dx->SetupClientViewport(device);

        return SUCCESS_OK;
}
```

7 Graphic Frameset

The Frameset is comprised of Frames, each of which is comprised of three concentric parts: Margin, Border, and Padding, which wrap around the content within a Frame.

This convention is derived from the CSS box model, which is essentially a box that wraps around every HTML element.

For example:

The padding property in html gives space around the innermost element's content of the box-like structure. The margin property in html gives space around the outermost element's content of the box-like structure. The space around padding and margin is called a border.
The difference between the padding, margin, and border you can observe below:

```
// ---------- App_Setup_Project --------
/*!
\brief setup project
\author Gareth Edwards
\note invoked by Fw_Setup
*/
HRESULT Darkland_01::App_Initialise_Framest(VOID)
{

        // ---- rename
        vsl_system::Win_Create *win_cr8 = Get_Win_Create();
        win_cr8->SetName("Darkland 01");

        // ---- scope
        using namespace vsl_library;

        // ---- Gfx_Frameset
        Gfx_Frame::Rectangle outside  = { 0.05f, 0.05f, 0.95f, 0.95f };
        Gfx_Frame::Rectangle outside1 = { 0.05f, 0.05f, 0.45f, 0.45f };
        Gfx_Frame::Rectangle outside2 = { 0.55f, 0.05f, 0.95f, 0.45f };
        Gfx_Frame::Rectangle outside3 = { 0.05f, 0.55f, 0.95f, 0.95f };

        Gfx_Frame::Rectangle margin  = {  2,   2,   2,   2 };
        Gfx_Frame::Rectangle border  = { 10,  10,  10,  10 };
        Gfx_Frame::Rectangle padding = {  4,   4,   4,   4 };
        Gfx_Frame::Rectangle inside;

        Gfx_Frameset *gfx_frameset = GetFrameset();

        gfx_frameset->SetDimensions(
                win_cr8->GetClientWidth(),
                win_cr8->GetClientHeight());

        Gfx_Frame *frame = gfx_frameset->AddFrame("Frame");
        frame->SetActive(TRUE);
        frame->SetVisible(TRUE);
        frame->SetFrameRect(Gfx_Frame::TYPE::OUTSIDE, &outside);
        frame->SetFrameRect(Gfx_Frame::TYPE::MARGIN,  &margin);
        frame->SetFrameRect(Gfx_Frame::TYPE::BORDER,  &border);
        frame->SetFrameRect(Gfx_Frame::TYPE::PADDING, &padding);

        Gfx_Frame *cf1 = frame->AddChild("CF1");
        cf1->SetActive(TRUE);
        cf1->SetFrameRect(Gfx_Frame::TYPE::OUTSIDE, &outside1);
        cf1->SetFrameRect(Gfx_Frame::TYPE::MARGIN,  &margin);
        cf1->SetFrameRect(Gfx_Frame::TYPE::BORDER,  &border);

        Gfx_Frame *cf2 = frame->AddChild("CF2");
        cf2->SetActive(TRUE);
        cf2->SetFrameRect(Gfx_Frame::TYPE::OUTSIDE, &outside2);
        cf2->SetFrameRect(Gfx_Frame::TYPE::MARGIN,  &margin);
        cf2->SetFrameRect(Gfx_Frame::TYPE::BORDER,  &border);

        Gfx_Frame *cf3 = frame->AddChild("CF3");
        cf3->SetActive(TRUE);
        cf3->SetFrameRect(Gfx_Frame::TYPE::OUTSIDE, &outside3);
        cf3->SetFrameRect(Gfx_Frame::TYPE::MARGIN,  &margin);
        cf3->SetFrameRect(Gfx_Frame::TYPE::BORDER,  &border);
        cf3->SetFrameRect(Gfx_Frame::TYPE::PADDING, &padding);

        Gfx_Log *gfx_log = GetLog();
        gfx_frameset->SetGfxLog(gfx_log);

        gfx_frameset->Setup();

        return SUCCESS_OK;

}
```

8 Graphic Elements

8.1 Gfx_Element Class

A 3D scene contains three things: geometry (one or more 3D objects), lighting (without which a 3D scene will be black!), and a camera, to define the point of view from which the scene will be rendered.

This 3D is rendered either directly into a CAV (Client Adjusted Viewport, an entire application window minus the borders, etc.), or a frame within a frameset (see previous section).

The CAV or frame might also include a GUI (Graphical User Interface), a 2D scene, which is comprised of one or more 2D shapes, which when a user interacts with it (click, touch, drag, etc.) generate a "message" which is processed by an application effecting change.

As previously mentioned, all GUIs conform to the MVC design pattern.

The Gfx_Element class is the building block with which all 2D shapes & 3D objects are managed. As mentioned previously VSL nomenclature uses the term "shape" for 3D objects. This is not only correct but avoids confusion with the software terms "object" and "component".

Graphic elements (Gfx_Element) are appended to a default "Project" element, and to each other to create an element tree structure. Typically, their properties are initialised during this process.

To provide for on-the-fly swapping of one project element tree for another, the "Project" element is the child of an "Engine" root element.

Example:

```
Gfx_Element *engine_root_element = NULL;
if (engine_root_element = GetElementEngine()->GetEngineRoot())
{
	if (Gfx_Element *project = engine_root_element->Find("Project"))
	{
		Gfx_Element *element_tr0 = project->Append("Cube", 0);
		...
	}
}
```

Each `Gfx_Element` is a comprised of a single private implementation object `Pimpl_Gfx_Element`, which is instantiated when an element is instantiated (using the initialisation list).

8.2 Private Implementation

Each `Pimpl_Gfx_Element` is a comprised of pointers to three objects: `Gfx_Element_Configure`, `Gfx_Element_Coordinate`, and `Gfx_Element_Component`. These are instantiated when a `Pimpl_Gfx_Element` is instantiated.

The `Pimpl_Gfx_Element` has the following attributes:

```
// ---- gfx
Gfx_Element_Configure  *configure  = NULL;
Gfx_Element_Coordinate *coordinate = NULL;
Gfx_Element_Component  *component  = NULL;

// ---- state
UINT  id = 0;
std::string name;
```

```
std::string value;

// ---- hierarchical links
Gfx_Element *parent   = NULL;
Gfx_Element *first    = NULL;
Gfx_Element *last     = NULL;
Gfx_Element *previous = NULL;
Gfx_Element *next     = NULL;

// ---- parameter group links
Gfx_Element *first_param_group = NULL;
Gfx_Element *last_param_group = NULL;
```

All of these are accessed via `Gfx_Element` methods.

8.3 Configuration

All `Gfx_Element` have various attributes and states.

These are stored in, and accessed, using the `Gfx_Element_Configuration` class.

Example showing `Gfx_Element` appending, instancing, and configuring (in the `App_Initialise_Project` method).

```
// ---- element to be instanced
Gfx_Element *element_tr0 = project->Append("TR0", 0);

Gfx_Element_Configure *configure = element_tr0->GetConfigure();
configure->SetComponentName("PyRhoDo_VBO");
configure->SetVisible(FALSE);

// ---- group id 1 : element id 4 (or could be 3!)
Gfx_Element *element_group_1 = project->Append("G1", 1);
GetElementEngine()->AddBookMark(element_group_1);
{
	Gfx_Element *element_tr1 = element_group_1->Append("TR1", 4);
	GetElementEngine()->AddBookMark(element_tr1);

	Gfx_Element_Configure *configure = element_tr1->GetConfigure();
	configure->SetComponentName("Cuboid_VBO");
	configure->SetVisible(TRUE);
	configure->SetInstance(FALSE);
	configure->SetInstanceName(element_tr0->GetName());
}
```

8.4 Coordinate

All `Gfx_Element` have a list of coordinate parameters which are used when it is displayed to set the rotation, scaling, and translation (and the order of these).

These are stored in, and accessed, using the `Gfx_Element_Coordinate` class.

Example showing `Gfx_Element` updating `Gfx_Element_Coordinate` (in the `App_Initialise_Element_Coordinates` method):

```
// ---------- update element coordinate properties
Gfx_Element *element_tr1 = engine_root_element->Find("TR1");
if (element_tr1 != NULL)
{
	Gfx_Element_Coordinate *coordinate = element_tr1->GetCoordinate();
```

```
        vsl_system::Vsl_Vector3 v = { 1, 2, 1 };
        coordinate->SetScale(vsl_system::Vsl_Vector3(1, 2, 1));
}
```

8.5 Component

The Gfx_Element_Component class is the interface to the Gfx_Kandinsky class, the graphics shape creator.

See the next section below for Gfx_Kandinsky class.

A Gfx_Element_Component is primarily comprised of “get” and “set” methods:

```
// ---- get
Gfx_Kandinsky                    *GetKandinsky(VOID);
LPDIRECT3DVERTEXBUFFER9          *GetVertexBuffer(VOID);
LPDIRECT3DINDEXBUFFER9           *GetIndexBuffer(VOID);
UINT                              GetConfigBitmask(VOID);
Gfx_Kandinsky_Interface_Callbacks *GetKandinskyInterfaceCallbacks(VOID);

// ---- set
VOID SetVertexBuffer(LPDIRECT3DVERTEXBUFFER9 vertex_buffer);
VOID SetIndexBuffer(LPDIRECT3DINDEXBUFFER9 index_buffer);
VOID SetConfigBitmask(UINT config_bitmask);
```

8.5.1 Gfx_Element

When a Gfx_Element is created, an instance of the Gfx_Element_Component is also created.

8.5.2 Private Implementation

When a Gfx_Element_Component is created, and instance of the private implementation object Pimpl_Gfx_Element_Component is also created, which contains Gfx_Kandinsky Gfx_Kandinsky_Interface_Callback (see below).

```
// ---- kandinsky interface callbacks
struct Gfx_Kandinsky_Interface_Callbacks
{
        HRESULT(*kandinsky_interface_append_parameters) (Gfx_Element *) = NULL;
        HRESULT(*kandinsky_interface_config_and_create)
                                        (Gfx_Element_Component *) = NULL;
};
```

8.5.3 Kandinsky Interface

The Gfx_Element_Engine uses Gfx_Kandinsky_Interface_Callbacks to get a function pointer to a Kandinsky shape method that is used to append Kandinsky shape parameters.

This is invoked one-time only, when the Gfx_Element_Engine first processes a Gfx_Element.

The Gfx_Element_Engine also uses Gfx_Kandinsky_Interface_Callbacks to get a function pointer to a Kandinsky shape method that is used to configure and create a Kandinsky shape.

8.5.4 Kandinsky Bitmasks

The Gfx_Element_Engine uses the **enumerated** Gfx_Component_Buffer_Bitmasks values to test the Kandinsky shape buffers states.

```
// ---- enumerate component buffer bitmasks
typedef enum Gfx_Component_Buffer_Bitmasks
```

```
{
        NONE            = 0,
        KANDINSKY       = 1,
        VERTEX_BUFFER   = 2,
        INDEX_BUFFER    = 4
} Gfx_Component_Buffer_Bitmasks;
```

The following code shows usage in the Gfx_Element_Engine::Element_SetupDX method.

```
// ---- lambda
auto IsBitSet = [](byte b, int pos) -> bool
{
        return (b & (1 << pos)) != 0;
};


// ---- initialise kandinsky buffers
if (!IsBitSet(config_bitmask,Gfx_Component_Buffer_Bitmasks::KANDINSKY))
{
        // ---- initialise both kandinsky vertex & index buffers
}


// ---- create vertex_buffer & then copy kandinsky_vertex_buffer
//      into vertex_buffer ( and then later for the index_buffer)
if (!IsBitSet(config_bitmask,Gfx_Component_Buffer_Bitmasks::VERTEX_BUFFER))
{
        // - release vertex_buffer
        // - get kandinsky vertex buffer info
        // - create vertex buffer
        // - copy kandinsky_vertex_buffer into vertex_buffer

}
```

8.5.5 Kandinsky Life-Cycle

An instance of a Gfx_Element contains a Gfx_Element_Component object, which is an interface to the Direct3D device vertex and index buffers.

It also contains - and is an interface to - a Gfx_Kandinsky shape.

A Gfx_Kandinsky object stores information about a graphic shape, and its vertex & index buffers that store 2D or 3D geometry data in the same format as required by the Direct3D vertex & index buffers.

8.5.5.1 Setup

When the Gfx_Element_Engine::SetupDX method processes a Gfx_Element it tests the status of the Gfx_Element_Component Direct3D device buffers.

This then:

- Gets a pointer to the Gfx_Element_Component, Gfx_Kandinsky object.
- Gets a pointer to the Gfx_Element_Component, Gfx_Kandinsky_Interface_Callbacks.
    - There are two callback methods; **create** and **configure**.
- Gets the Gfx_Element_Component, Gfx_Component_Buffer_Bitmasks bitmask.

- If a Gfx_Kandinsky shape has not been created, then **force creation** by setting appropriate `Gfx_Component_Buffer_Bitmasks::KANDINSKY` value.

- If `Gfx_Component_Buffer_Bitmasks:KANDINSKY` value denotes `Gfx_Kandinsky` shape NOT configured, then invoke `Gfx_Kandinsky` **configure** method. This initialises the Kandinsky vertex and/or Index buffers.

- if `Gfx_Element_Component` buffers not initialised as per `Gfx_Kandinsky` copy `Gfx_Kandinsky` vertex & index buffers to the Direct3D device vertex & index buffers.

8.5.5.2 Display

When the `Gfx_Element_Engine::Display` method processes a `Gfx_Element` it tests the status of the `Gfx_Element_Component` buffers.

- If `Gfx_Element_Component` buffers not initialised as per `Gfx_Kandinsky`
    - Copy Kandinsky vertex & index buffers to the Direct3D device vertex & index buffers
    - Invoke Direct3D display primitive method.

Note: See Section on the Gfx_Element_Engine for more details.

8.6 Kandinsky

A Gfx_Kandinsky class object stores vertex & index information about a graphic 2D or 3D shape, and the parameters required to configure, then create this shape.

This information is referred to as a Kandinsky shape.

This Kandinsky shape geometry data is stored in vertex & index buffers in the same format as required by the Direct3D vertex & index buffers.

A Kandinsky shape is decoupled from both VSL and Direct3D.

A Kandinsky shape is generated and retained (or "stored ") and provided on demand (e.g., runtime, setup, Direct3D device reset) to VSL, which then "loads" them into Direct3D buffers.

This retained Kandinsky shape can be provided with minimal delay due to a shape not requiring to be recreated, even if the Direct3D device needs to be reset, etc.

For dynamic shapes, those being regularly updated, judicious update (e.g., only changing those parts of a Kandinsky shape that require update) can significantly impact time required for recreation.

This strategy also enables Kandinsky shape generation methods to be threaded.

8.6.1 Direct3D Vertex & Index Buffers

The following succinct summary is from the Direct3D 9 documentation (dating from 2010, and now superseded by Direct3D 12).

> ***Direct3D Vertex buffers****, represented by the IDirect3DVertexBuffer9 interface, are memory buffers that contain vertex data. Vertex buffers can contain any vertex type - transformed or untransformed, lit or unlit - that can be rendered through the use of the rendering methods in the IDirect3DDevice9 interface.*
>
> ***Direct3D Index buffers****, represented by the IDirect3DIndexBuffer9 interface, are memory buffers that contain index data. Index data, or indices, are integer offsets into vertex buffers and are used to render primitives using the IDirect3DDevice9::DrawIndexedPrimitive method.*
>
> *A vertex buffer contains vertices; therefore, you can draw a vertex buffer either with or without indexed primitives. However, because an index buffer contains indices, you cannot use an index buffer without a corresponding vertex buffer.*

8.6.2 Decouple

To reduce time required to (re)acquire contextual understanding, Kandinsky is decoupled from VSL, though for convenience it does share two VSL header files.

These are "vsl_system/header/vsl_include.h" and "vsl_system/header/vsl_sem.h".

The first includes the required C & C++ header files.

The second includes required **S**tructs, **E**numerations and **M**acros header file.

8.6.3 Source Files

These four files must be updated to add a new Kandinsky shape.

Headers : `"vsl_library/header/"`

```
        -> "vsl_gfx_kandinsky.h"
        -> "vsl_gfx_kandinsky_interface.h"
```

C++: `"vsl_library/source/"`

```
-> "vsl_gfx_kandinsky.cpp"
-> "vsl_gfx_kandinsky_interface.cpp"
```

Note: An updater application will be provided soon that will provide for the addition, and removal, of Kandinsky shapes from these four files.

8.6.4 Shape Files

For convenience the substantive code for configuring & creating Kandinsky shapes are in separate files, which are included explicitly within the Kandinsky class definition.

Included substantive code is kept in files with a ".hpp" suffix.

They are in the same directory path as the VSL headers & C++ files.

Included substantive C++ code: "vsl_library/hpp_obj/"

Details are given below.

8.6.5 Shape File Naming

"vsl_gfx_kandinsky_[name]_[type, e.g., VBO or VIBO].hpp"

e.g., "vsl_gfx_kandinsky_cuboid_vbo.hpp", "vsl_gfx_kandinsky_cuboid_vibo.hpp"

8.6.6 Adding a Kandinsky Shape

8.6.6.1 Create [name] shape file.

Directory: `"vsl_library/hpp_obj/"`.

File: `"vsl_gfx_kandinsky_[name].hpp"`.

It is good practice to copy an existing Kandinsky shape to provide a working shape, as this allows the other parts of this process to be progressed knowing that the Create & Config are proven.

A Kandinsksy shape requires Config & Create methods.

The Config method "sets" the parameters required to Create a shape.

The Create method generates vertex & index buffers.

See below for “How to Create a Kandinsky Shape”.

8.6.6.2 Add [name] shape file to shape file list.

This is a list of included substantive C++ code shape found at the bottom of this file.

Directory: `"vsl_library/source/"`.

File: `"vsl_gfx_kandinsky.cpp"`.

e.g.,

```
#include "../hpp_obj/vsl_gfx_kandinsky_[name].hpp"
```

8.6.6.3 Add declarations for [name] shape methods.

Directory: `"vsl_library/header/"`.

File: `"vsl_gfx_kandinsky.h"`.

e.g.,

```
// ---- with ONLY a vertex buffer
       HRESULT [name]_VBO_Config(VOID);
       HRESULT [name]_VBO_Create(VOID);

// ---- with a vertex AND index buffer
       HRESULT [name]_VIBO_Config(VOID);
       HRESULT [name]_VIBO_Create(VOID);
```

8.6.6.4 Enumerate Gfx_Kandinsky_Component [name] shape.

Directory: `"vsl_library/header/"`.

File: `"vsl_gfx_kandinsky.h"`.

e.g.,

```
// ----enum
typedef enum Gfx_Kandinsky_Component
{
```

```
        COMPONENT_UNKNOWN,
        CUBOID_VBO,
        Cuboid_VIBO,
        PYRHODO_VBO,
        [name]_[type]
} Gfx_Kandinsky_Component;
```

Hint: Compile to verify that the above has been successful.

8.6.6.5 Add Gfx_Kandinsky_Interface [name] shape methods.

Directory: "vsl_library/source/".

File: "vsl_gfx_kandinsky_interface.cpp".

Methods:

- [name]_[type]_Config_Kandinsky_Parameters
- [name]_[type]_Config_Kandinsky_Component

e.g.,

```
// ---------- [name] - vertex buffer vesion ----------
HRESULT Gfx_Kandinsky_Interface::[name]_[type]_Config_Kandinsky_Parameters(
              Gfx_Element *component_param_group
              )
{
        AppendColourParameters(component_param_group);
        AppendTransformParameters(component_param_group);
        return SUCCESS_OK;
}

HRESULT Gfx_Kandinsky_Interface::[name]_[type]_Config_Kandinsky_Component(
              Gfx_Element_Component *gfx_element_component
        )
{

        Gfx_Kandinsky *kandinsky = gfx_element_component->GetKandinsky();
        kandinsky->[name]_[type]_Config();
        kandinsky->[name]_[type]_Create();

        return SUCCESS_OK;
}
```

8.6.6.6 Add Gfx_Kandinsky_Component [name] to interface.

Directory: "vsl_library/source/".

File: "vsl_gfx_kandinsky_interface.cpp".

Add shape name to interface.

e.g.,

```
// ---- code example:
HRESULT Gfx_Kandinsky_Interface::GetComponentTypeId(const std::string& name)
{

        if (name == "[name]")
        {
```

```
                return Gfx_Kandinsky_Component::[enum name]_VBO;
        }
        else if (name == "[some other name]")
        {
                return Gfx_Kandinsky_Component::[some other enum name]_VIBO;
        }
        else
        {
                return ERROR_FAIL;
        }

        return SUCCESS_OK;
}
```

8.6.6.7 Add [name] shape Config & Create method declarations.

Directory: `"vsl_library/source/"`.

File: `"vsl_gfx_kandinsky_interface.cpp".`

e.g.,

```
// ---- code example:

static HRESULT [name]_[type]_Config_Kandinsky_Parameters(
        Gfx_Element *component_param_group);

static HRESULT [name]_[type]_Config_Kandinsky_Component(
        Gfx_Element_Component *gfx_element_component);
```

8.6.6.8 Add GetCallbacks for Parameters & Component

Directory: `"vsl_library/source/".`

File: `"vsl_gfx_kandinsky_interface.cpp".`

The GetCallbacks method takes two arguments: a pointer to a `Gfx_Element_Component`, and pointer to a `Kandinsky_Interface_Callbacks` struct.

Using the component type this method initialises a `Gfx_Kandinsky_Interface_Callbacks` struct.

```
// ---- code example:

struct Gfx_Kandinsky_Interface_Callbacks
{
        HRESULT(*kandinsky_interface_append_parameters)
                        (Gfx_Element *) = NULL;
        HRESULT(*kandinsky_interface_config_and_create)
                        (Gfx_Element_Component *) = NULL;
        };
```

The append method hierarchically appends groups of parameters that are unique to a specific Kandinsky shape.

The second invokes the configure & create methods.

This method returns pointer to these methods.

```
// ---- code example:

HRESULT Gfx_Kandinsky_Interface::GetCallbacks(
        Gfx_Element_Component *gfx_element_component,
        Gfx_Kandinsky_Interface_Callbacks *gfx_kandinsky_interface_callbacks
        )
{

        Gfx_Kandinsky *kandinsky = gfx_element_component->GetKandinsky();
        UINT component_type = 0;

        HRESULT hr = kandinsky->Get(
                Gfx_Kandinsky_Param::COMPONENT_TYPE,
                &component_type);

        switch (component_type)
        {
                case Gfx_Kandinsky_Component::CUBOID_VBO:
                {
                        gfx_kandinsky_interface_callbacks
                                ->kandinsky_interface_append_parameters =
                                [name]_[type]_Config_Kandinsky_Parameters;
                        gfx_kandinsky_interface_callbacks
                                ->kandinsky_interface_config_and_create =
                                [name]_[type]_Config_Kandinsky_Component;
                }
                break;
                default:
                {
                        gfx_kandinsky_interface_callbacks
                                ->kandinsky_interface_append_parameters = NULL;
                        gfx_kandinsky_interface_callbacks
                                ->kandinsky_interface_config_and_create = NULL;
                }
                return SUCCESS_FAULT;
        }

        return SUCCESS_OK;
}
```

8.6.6.9 Summary

To add a new Kandinsky shape:

1) Create [name] shape file.
2) Add [name] shape file to shape file list.
3) Add declarations for [name] shape methods.
4) Enumerate `Gfx_Kandinsky_Component` [name] shape.
5) Add `Gfx_Kandinsky_Interface` [name] shape methods.
6) Add `Gfx_Kandinsky_Component` [name] shape to interface.
7) Add [name] shape Config & Create method declarations.
8) Add GetCallbacks for Parameters & Component

The first four steps update the `"vsl_gfx_kandinsky.cpp"` & `"vsl_gfx_kandinsky.h"` files.

The first four steps update the `"vsl_gfx_kandinsky_interface.cpp"` and `"vsl_gfx_kandinsky_interface.h"` files.

8.7 Gfx_Kandinsky

8.8 Gfx_Element_Engine

The Gfx_Kandinsky class is a comprised of a single private implementation object Pimpl_Gfx_Kandinsky, which is instantiated when a Gfx_Kandinsky is instantiated (using the initialisation list), and a sequence of paired configuration and creation functions.

An elements component name identifies the Kandinsky geometry creator methods that are to be used. As per the previous configuration example:

```
configure->SetComponentName("PyRhoDo_VBO");
```

8.8.1 Setup

In the Element Engine Setup method there are three main steps.

8.8.1.1 Validate Element Component ID

If a component name is a valid Gfx_Kandinsky creator name (e.g., "PyRhoDo_VBO")(using the Gfx_Kandinsky_Interface) then the Gfx_Element_Component, Gfx_Kandinsky component type id is set AND the configure component status set TRUE.

```
// ---- if element has a valid component id, and
//      is therefore a valid component

// ---- get component id using component configure name
Gfx_Kandinsky_Interface gfx_kandinsky_interface;
HRESULT component_type_id =
       gfx_kandinsky_interface.GetComponentTypeId(
              element->GetConfigure()->GetComponentName());

// ---- valid?
if (SUCCEEDED(component_type_id))
{
       // ---- get element component object
       Gfx_Element_Component *gfx_element_component = element->GetComponent();

       // ---- get component’s Kandinsky object
       Gfx_Kandinsky *gfx_component_kandinsky =
              gfx_element_component->GetKandinsky();

       // ---- set component’s Kandinsky id
       HRESULT hr =
              gfx_component_kandinsky->SetComponentType(component_type_id);

       // ---- set element configure component status as TRUE
       element->GetConfigure()->SetComponent(TRUE);
}
```

8.8.1.2 Append Element Configuration & Coordinate Parameters

If not valid, then an error is logged, and the element will not have graphical data.

1. The element then has two parameter groups appended. These are the “Configuration” and the “Coordinate” groups.

```
// ---- So, every element has:

// ---- "Configuration" child parameter groups
Gfx_Element *config_param_group = element->AppendParamGroup("Configuration");
Gfx_Element_Configure *configure = element->GetConfigure();
configure->GetParameterGroups(config_param_group);

// ---- "Coordinate" child parameter groups
Gfx_Element *coord_param_group = element->AppendParamGroup("Coordinate");
Gfx_Element_Coordinate *coordinate = element->GetCoordinate();
coordinate->GetParameterGroups(coord_param_group);
```

8.8.1.3 Add Element Component Parameters

```
// ---- if valid component then add "Component" child parameter groups and
//      an initialised Kandinsky object with component parameters

HRESULT hr;
Gfx_Element_Component *gfx_element_component = element->GetComponent();
if (gfx_element_component != NULL) try
{


       // ---- 1st: get pointer to kandinsky config callbacks
       //     Note, these callbacks are:
       //     Gfx_Element * kandinsky_interface_append_parameters (used here!)
       //     Gfx_Element_Component * kandinsky_interface_config_and_create
       //                                       (used in Element_SetupDX)
       Gfx_Kandinsky_Interface_Callbacks *gfx_kandinsky_interface_callbacks =
       gfx_element_component->GetKandinskyInterfaceCallbacks();

       // ---- 2nd: use first interface callback to get
       //      kandinsky component callbacks
       Gfx_Kandinsky_Interface gfx_kandinsky_interface;
       gfx_kandinsky_interface.GetCallbacks(gfx_element_component,gfx_kandinsky
       _interface_callbacks);

       // ---- 3rd: append element "Component" parameter group
       Gfx_Element *component_param_group =
              element->AppendParamGroup("Component");

       // ---- 4th: every component element has a unique
       //      kandinsky parameter group(s) appended using config callback
       if (gfx_kandinsky_interface_callbacks->
              kandinsky_interface_append_parameters != NULL)
       {
              hr = gfx_kandinsky_interface_callbacks->
              kandinsky_interface_append_parameters(component_param_group);
              if (FAILED(hr))
                     throw("Failed: To append Component Kandinsky Parameters");
       }

       // ---- required to append parameters
       Gfx_Kandinsky *gfx_component_kandinsky =
              gfx_element_component->GetKandinsky();

       // ---- 5th: create component parameters
       //
       //   note: to decouple Kandinsky from the Gfx System these
       //   parameters are not comprised of Gfx_Elements, but are
       //   Kandinsky [group->group->...]->param->param->... lists
```

```
		//
		//  note: also speeds up the retrieval of parameter values...
		//
		Gfx_Element *group = component_param_group->GetFirst();
		while (group)
		{
			std::string group_name = group->GetName();
			Gfx_Element *param = group->GetFirst();
			while (param)
			{

				hr = gfx_component_kandinsky->AppendParameter(
					group_name,
					param->GetName(),
					param->GetValue()
				);
				if (FAILED(hr))
				throw("Failed: To append Kandinsky Parameter");

				param = param->GetNext();
			}
			group = group->GetNext();
		}

	}
	catch (CHAR *error)
	{
	...
	}
```

8.8.1.4 Summary:

Add parameter “Configuration” & “Coordinate” groups.

Then, if valid component name:

Get component Kandinsky interface callback functions.

Create Kandinsky interface.

Get Kandinsky

8.8.1.5 Example:

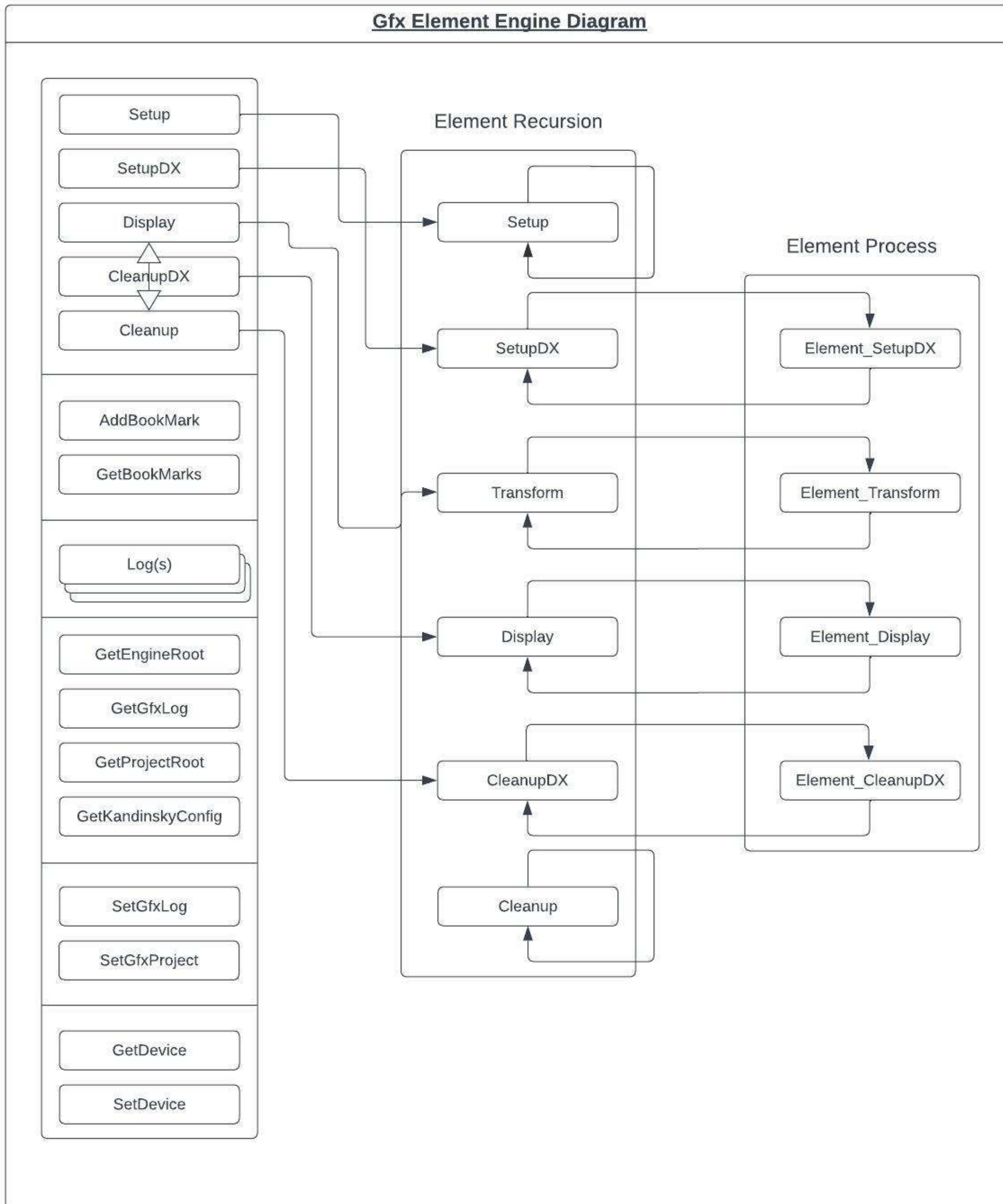
Here a Gfx_Element named “TR0”, is updating a Gfx_Element_Component.

Typically found in the App_Initialise_Element_Components method.

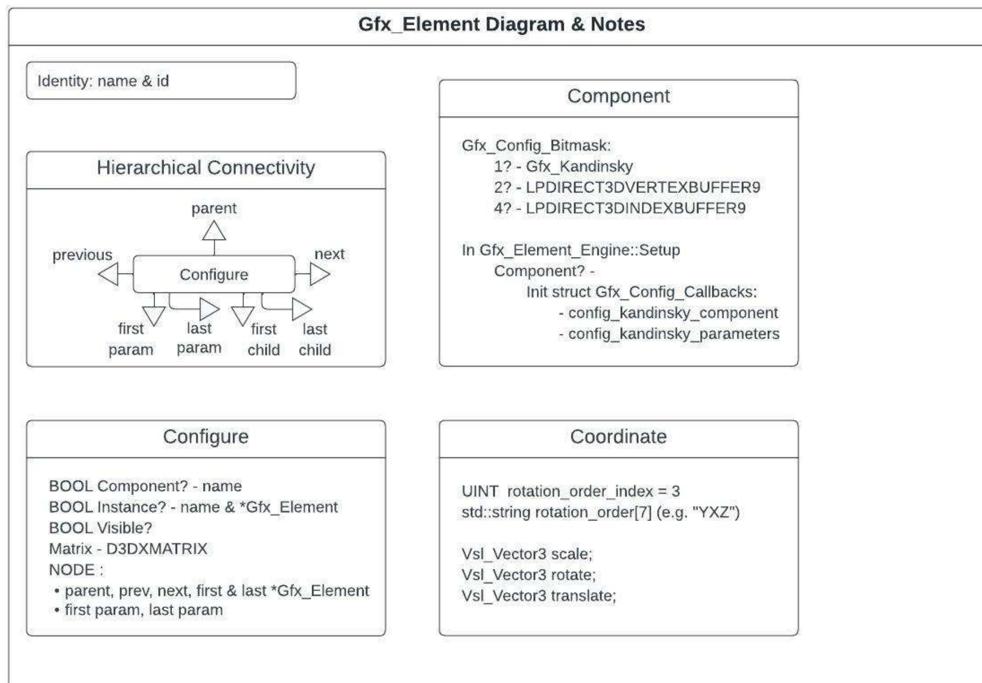
```
// ---- code example:

Gfx_Element *element_tr0 = engine_root_element->Find("TR0");
if (element_tr0 != NULL)
{
	// ---- PyRhoDo_VBO
	Gfx_Element_Component *component = element_tr0->GetComponent();
	Gfx_Kandinsky *kandinsky = component->GetKandinsky();

	// ---- build - include inside -> update element component properties
	kandinsky->Set(Gfx_Kandinsky_Param::INSIDE, 1);

	// ---- move -> update both component &
	//      corresponding kandinsky parameters

	// ---- get group
	Gfx_Element *element_param_group =
		element_tr0->FindParamGroup("Component");
	if (element_param_group)
	{
		// ---- set parameter
		std::string move = "0.5";
		HRESULT hr = element_param_group->
			SetParameterValue("Dimension", "Move", move);

		// ---- if OK? - set kandinsky
		if (SUCCEEDED(hr))
		{
			kandinsky->SetParameterValue("Dimension", "Move", move);
		}
	}

}
```

8.9 Gfx Diagrams & Flow Charts

8.9.1 Gfx_Element_Engine



8.9.2 Gfx_Element



Note: Typical hierarchical Gfx_Element listing from a vsl application SetupDX method:

```
Gfx Element "Project"
{
    Gfx Element "TR0"
    {
        Gfx Element "G1"
        Gfx Element "TR1"
        {
            Gfx Element "G2"
        }
        Gfx Element "TR2"
        {
            Gfx Element "G3"
        }
    }
}
```

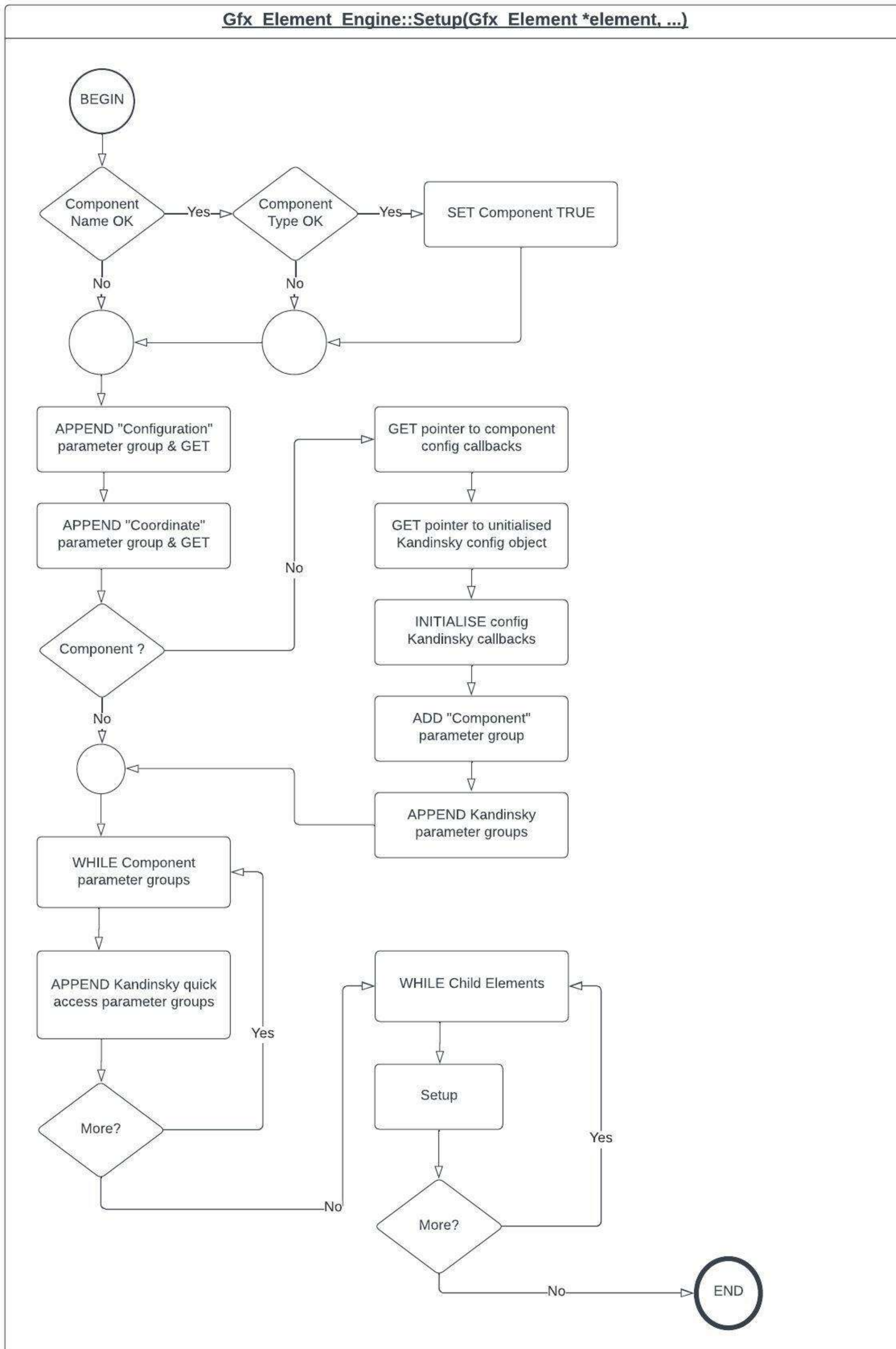
Note: The root node - not listed above - is named "Engine", so a complete hierarchical listing would include:

```
Gfx Element "Engine"
{
    Gfx Element "Project"
    {
        Gfx Element "TR0"
        [etc.,...]
    }
}
```

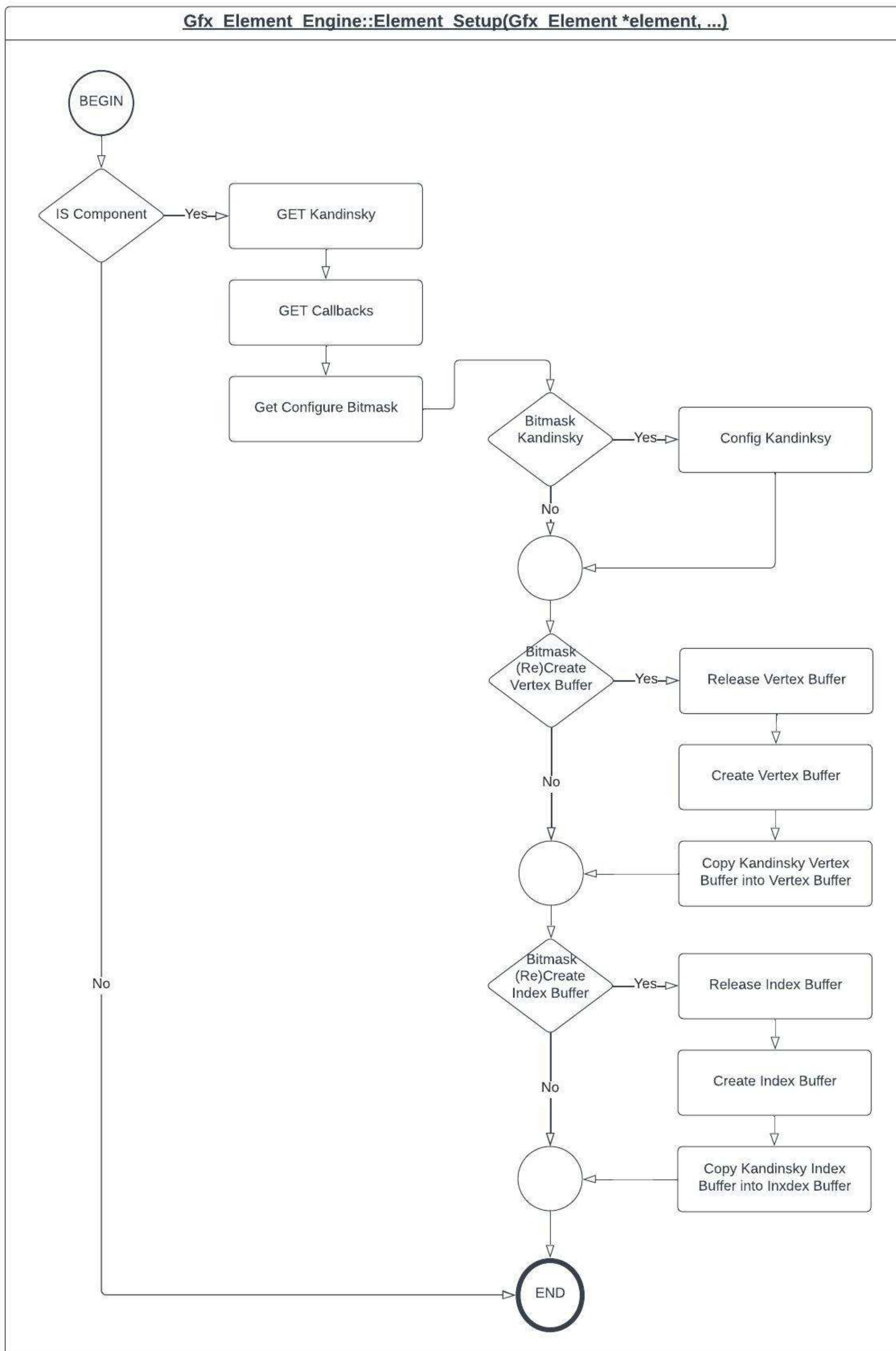
Note: Part of a typical hierarchical parameter listing from the Gfx_Element_Engine SetupDX method:

```
Project
{
    Configuration{
        Basic{ Component:0; Visible:1; }
        Instance{ Instance:0; Name{ } }
    }
    Coordinate{
        Translate{ X:0.000000; Y:0.000000; Z:0.000000; }
        Scale{ X:1.000000; Y:1.000000; Z:1.000000; }
        Rotate{ X:0.000000; Y:0.000000; Z:0.000000; Order:3; }
    }

    TR0{
        Configuration{
            Basic{ Component:1; Visible:0; }
            Instance{ Instance:0;   Name{ } }
        }
        Coordinate{
            Translate{ X:0.000000; Y:0.000000; Z:0.000000; }
            Scale{ X:1.000000; Y:1.000000; Z:1.000000; }
            Rotate{ X:0.000000; Y:0.000000; Z:0.000000; Order:3; }
        }
        Component{
            Colour{ Red:1; Green:1; Blue:1; Alpha:1; }
            Dimension{ Size:4; Move:0; }
            FX{ Explode:0.0; }
            Shift{ X:0; Y:0; Z:0; }
        }
    }
    [etc.,...]
}
```

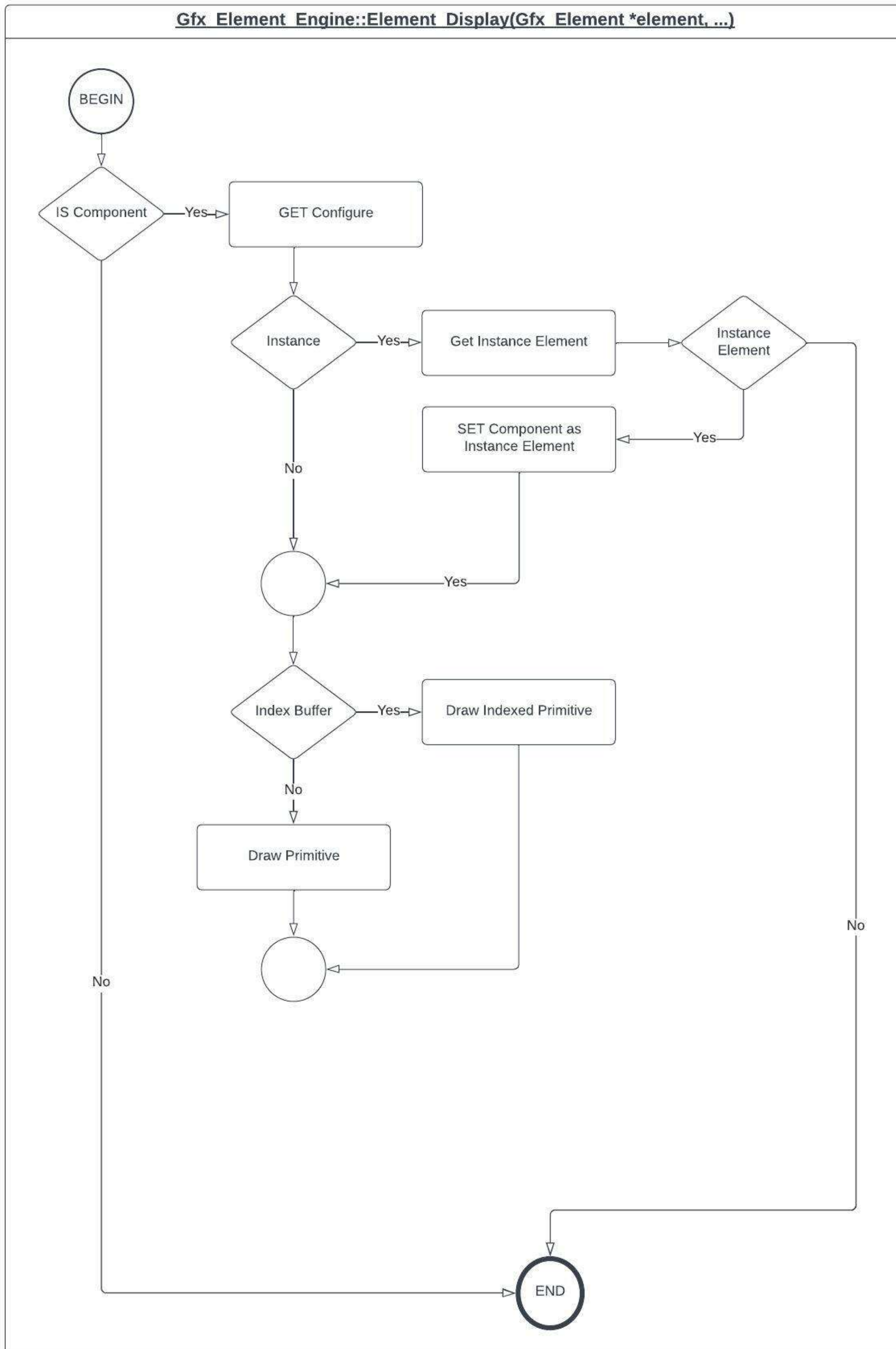
8.9.3 Gfx_Element_Engine::Setup



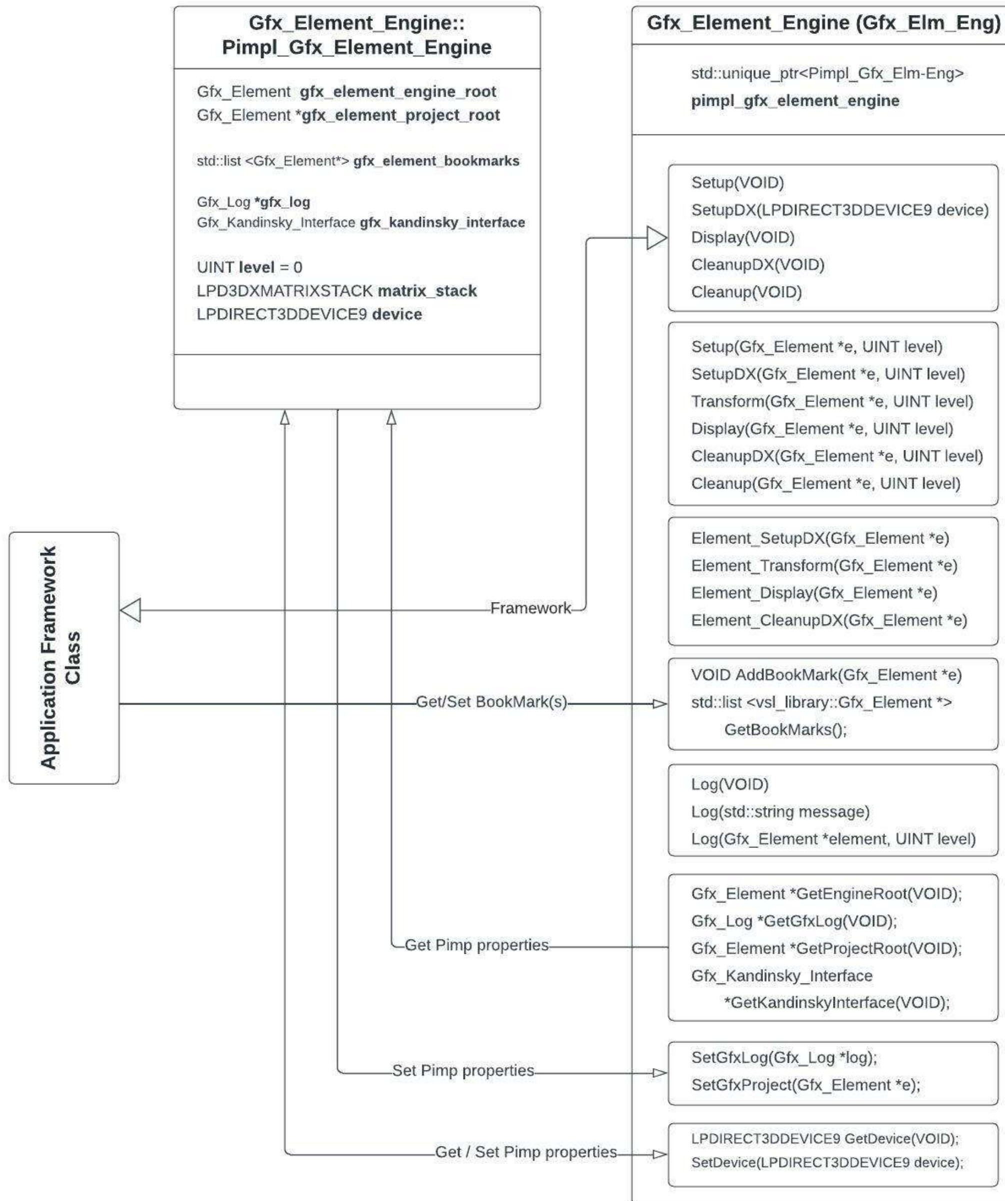
8.9.4 Gfx_Element_Engine::Element_Setup



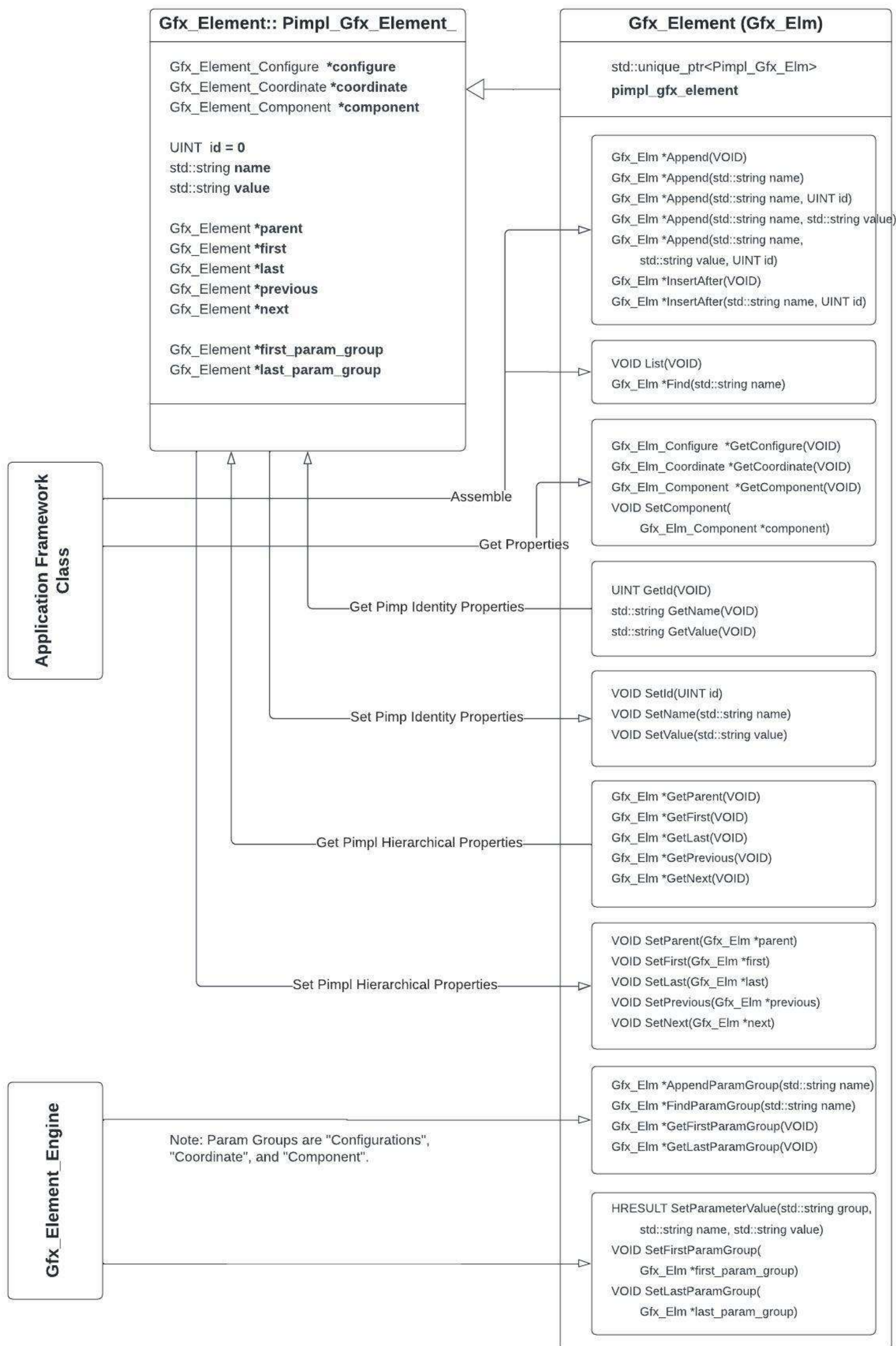
8.9.5 Gfx_Element_Engine::Element_Display



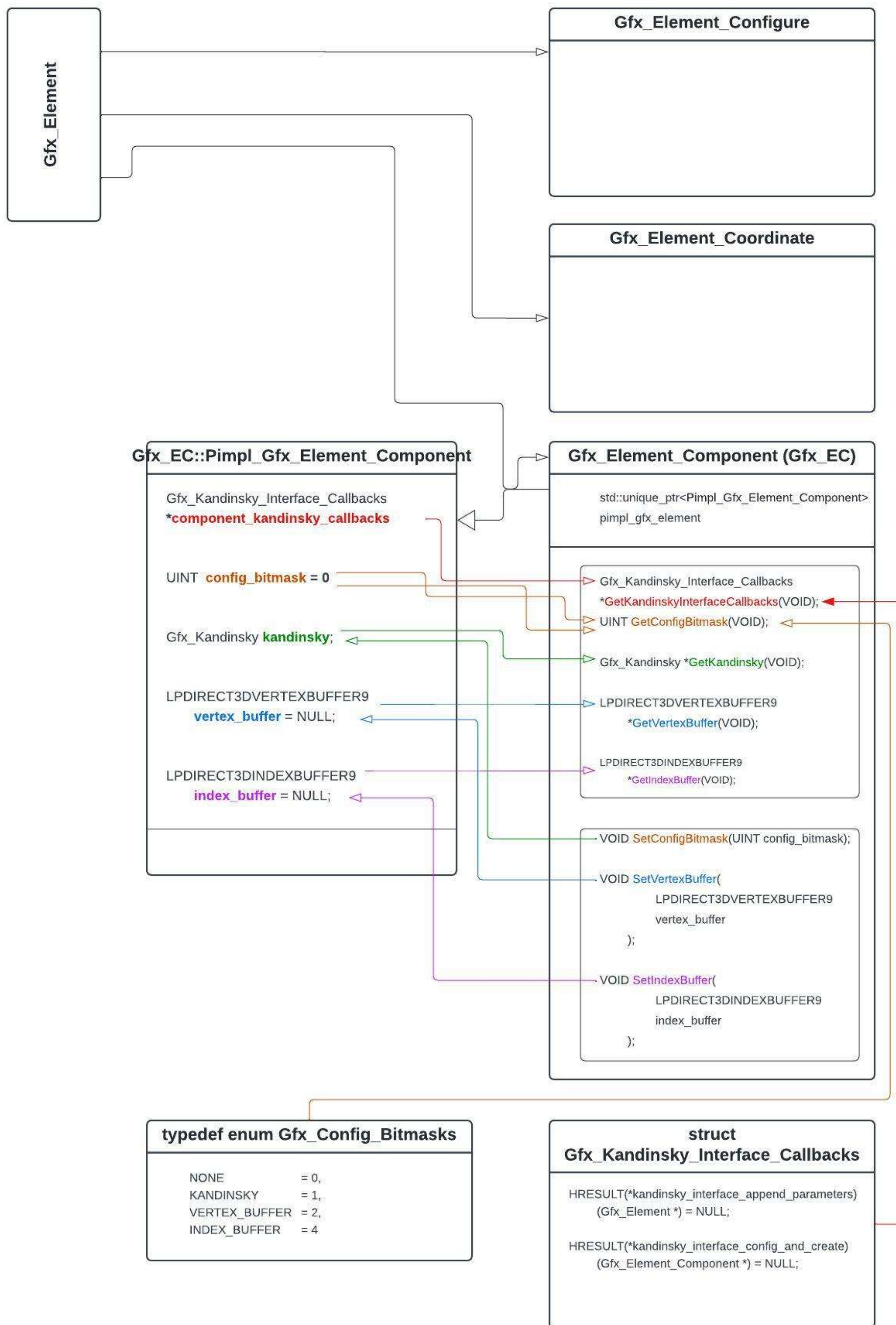
8.9.6 Gfx_Element_Engine & PImpl



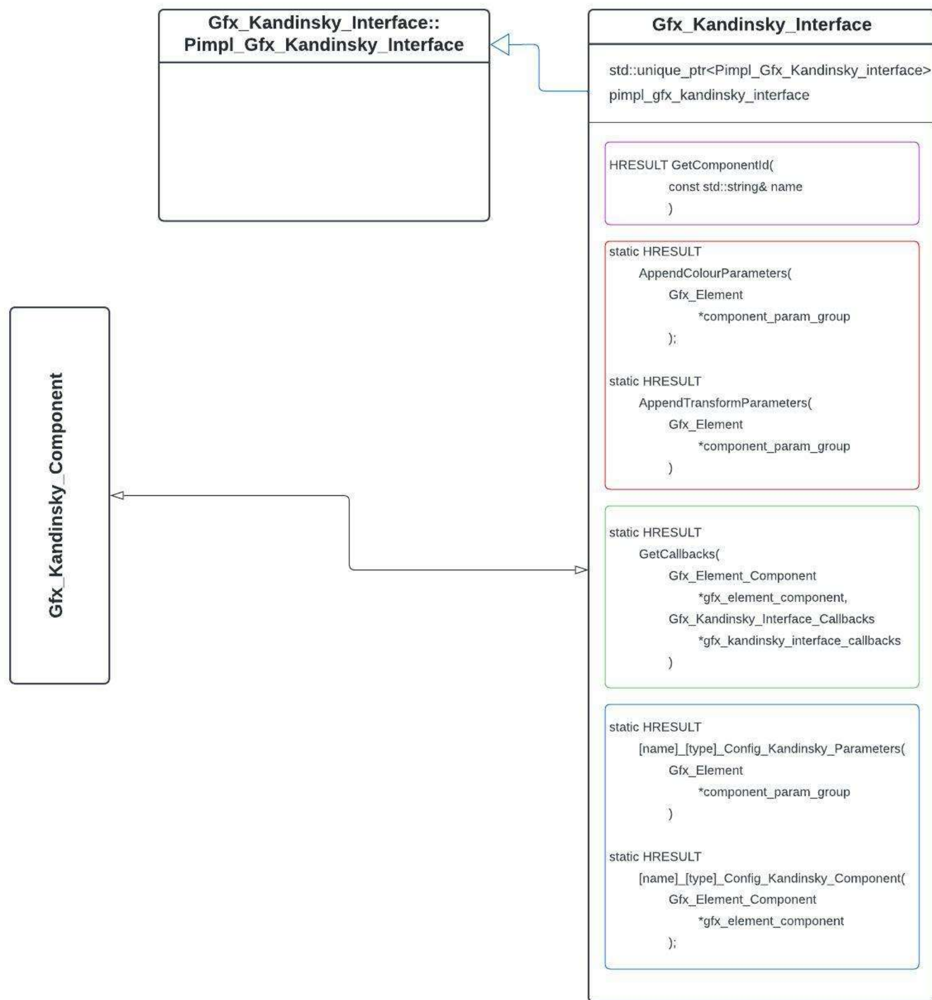
8.9.7 Gfx_Element & PImpl



8.9.8 Gfx_Element_Configure & PImpl



8.9.9 Gfx_Kandinsky_Interface & Pimpl



8.9.10 Gfx_Kandinsky & Pimpl

Gfx_Kandinsky::
Pimpl_Gfx_Kandinsky

```
struct Parameter {
    Parameter *parent
    Parameter *first
    Parameter *last
    Parameter *previous
    Parameter *next
    std::string name
    std::string value
}

Parameter * root

UINT    *index_buffer

FLOAT *vertex_buffer

INT     *header_buffer

INT HeaderBufferInitialise(UINT buffer_size);
INT IndexBufferInitialise(UINT buffer_size);
INT VertexBufferInitialise(UINT buffer_size);
```

Gfx_Kandinsky

```
std::unique_ptr<Pimpl_Gfx_Kandinsky>
pimpl_gfx_kandinsky

AppendParameter(
    const std::string& group_name,
    const std::string& param_name,
    const std::string& param_value
    )

GetParameterValue(
    const std::string& group_name,
    const std::string& param_name,
    FLOAT& value
    )

SetParameterValue(
    const std::string& group_name,
    const std::string& param_name,
    const std::string& param_value
    )

UINT GetIndexBuffer(UINT **index_buffer)
UINT GetIndexBufferSize(VOID)
SetIndexBufferSize(UINT index_buffer_size)

UINT GetVertexBuffer(FLOAT **vertex_buffer)
UINT GetVertexBufferSize(VOID)
UINT GetVertexFormat(VOID)
UINT GetVertexFormatSize(VOID)
SetVertexBufferSize(UINT vertex_buffer_size)

UINT GetPrimitiveCount(VOID)
UINT GetPrimitiveType(VOID)

BOOL Gfx_Kandinsky::IsComponent()
Gfx_Kandinsky_Component
    Gfx_Kandinsky::GetComponentType()

SetComponentType(const std::string& name)

Get(Gfx_Kandinsky_Param token,
    UINT *token_value)
UINT  Get(Gfx_Kandinsky_Param token)
Set(Gfx_Kandinsky_Param token,
    UINT token_value)
```

Gfx_Kandinsky_Interface

9 Appendices

9.1 Error Handling

Always use result handle: HRESULT hr == S_OK.

Every condition that begins with "E_" is a failure, i.e., E_FAIL: Something is wrong. Do not query this key. Can easily check using FAILED(hr). Using this to query a non-defined value is an error.

COM uses HRESULT values to indicate the success or failure of a method or function call.

- Various SDK headers define various HRESULT constants.
- A common set of system-wide codes is defined in "WinError.h".

The following table shows some of those system-wide return codes.

```
E_AppComCESSDENIED 0x80070005      Access denied.
E_FAIL          0x80004005  Unspecified error.
E_INVALIDARG    0x80070057   Invalid parameter value.
E_OUTOFMEMORY   0x8007000E  Out of memory.
E_POINTER       0x80004003   NULL was passed incorrectly for a pointer value.
E_UNEXPECTED    0x8000FFFF  Unexpected condition.
S_OK            0x0         Success.
S_FALSE         0x1         Success (but not a success!).
```

Correct way to test:

```
// ---- Wrong
       HRESULT hr = SomeFunction();
       if (hr != S_OK)
       {
              printf("Error!\n"); // Bad. hr might be another success code.
       }

// ---- Right
       HRESULT hr = SomeFunction();
       if (FAILED(hr))
       {
              printf("Error!\n");
       }
```

The success code S_FALSE deserves mention. Some methods use S_FALSE to mean, roughly, a negative condition that is not a failure. It can also indicate a "no-op"—the method succeeded but had no effect.

```
       if (hr == S_FALSE)
       {
              // Handle special case.
       }
       else if (SUCCEEDED(hr))
       {
              // Handle general success case.
       }
       else
       {
              // Handle errors.
              printf("Error!\n");
       }
```

9.2 Folder Structure & Namespaces

VSL Namespaces are based upon the VSL folder structure.

This is organised thus (where g denotes a folder):

- → vsl_application
  - → framework
    - → header
    - → hpp
    - → source
  - → shared
    - → header
  - → non framework application 1
    - → header
    - → source
  - → non framework application 2
    - → header
    - → source
- → vsl_library
  - → header
  - → hpp_obj
  - → source
- → vsl_notes
- → vsl_resource
- → vsl_system
  - → header
  - → source

The namespaces are:

- vsl_application
- vsl_library
- vsl_system

And are used thus:

```
namespace vsl_application
{

	class Darkland_01
	{

		public:
			...
		private:
			...

	};

}
```

9.3 VSL C++ Style Guide

The C++ conventions used in VSL, and VSL applications, are loosely based on the Google C++ Style Guide.

See https://google.github.io/styleguide/cppguide.html

```
////////////////////////////////////////////////////////////////////////////////
```

Blocks of Logic

No block of logic, function, or method, be greater in length than an A4 page.

Blocks of logic are separated by 80 x '/' (as per above).

Blocks of logic are title and commented , e.g.:

```
// ---------- framework : get window structs ----------
```

```
////////////////////////////////////////////////////////////////////////////////
```

Doxygen Format Documentation

Files:

```
// ---------- vsl_base_01.cpp ----------
/*!
\file vsl_base_01.cpp
\brief implementation of PyRhoDo_01 class derived from Base_01 class
\author Gareth Edwards
*/
```

Functions & Methods:

```
// ---------- AppFw_Display ----------
/*!
\brief required by framework: display
\author Gareth Edwards
\param LPDIRECT3DDEVICE9 - pointer to an IDirect3DDevice9 structure
\return HRESULT (SUCCESS_OK if ok)
\note
*/
```

```
////////////////////////////////////////////////////////////////////////////////
```

Header & Include Files

In general, every .cc file should have an associated .h file.

Header files should be self-contained (compile on their own) and end in ".h".

Non-header files that contain substantive code and which are for inclusion end in ".cpp".

Non-header files that do not contain substantive code for inclusion end in ".inc".

```
////////////////////////////////////////////////////////////////////////////////
```

Preventing Multiple Inclusions

Compatibility is not provided for.

Use the #pragma pre-processor directive.

N.B. Used to use the define guard:

```
#ifndef VSL_BASE_01
#define VSL_BASE_01

...

#endif
```

//

Forward Declarations

A forward declaration is a declaration of an identifier for which the programmer has not yet given a complete definition, or a "forward declaration" is a declaration of an entity without an associated definition.

The use of the word “forward” is hotly debated. It is argued that these are simply “declarations”, as “forward” is a meaningless term in coding.

Try to avoid declarations by using “#include”.

//

Inline Functions

- Define functions inline only when they are small, say, 10 lines or fewer.
- Beware use requiring destructors!
- Avoid loops or switch statements!

//

Name and order of Includes.

1. C system headers (more precisely: headers in angle brackets with the `.h` extension), e.g., `<unistd.h>`, `<stdlib.h>`.
2. A blank line
3. C++ standard library headers (without file extension), e.g., `<algorithm>`, `<cstddef>`.
4. A blank line
5. Other libraries' `.h` files.
6. A blank line
7. Your project's `.h` files.

//

Namespaces

With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Do not use *using-directives* (e.g., `using namespace foo`). Do not use inline namespaces.

For unnamed namespaces, (internal linkage):

> When definitions in a `.cc` file do not need to be referenced outside that file, give them internal linkage by placing them in an unnamed namespace or declaring them `static`. Do not use either of these constructs in `.h` files.

Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.

//

Nonmember, Static Member, and Global Functions

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Do not use a class simply to group static members. Static methods of a class should generally be closely related to instances of the class or the class's static data.

Nonmember and static member functions can be useful in some situations. Putting nonmember functions in a namespace avoids polluting the global namespace.

Nonmember and static member functions may make more sense as members of a new class, especially if they access external resources or have significant dependencies.

Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables and should nearly always exist in a namespace. Do not create classes only to group static members; this is no different than just giving the names a common prefix, and such grouping is usually unnecessary anyway.

If you define a nonmember function and it is only needed in its `.cc` file, use internal linkage to limit its scope.

//

Local Variables

With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Do not use *using-directives* (e.g., `using namespace foo`).