Virtual Surface

Version 1.4
01th April 2016
Gareth Edwards

1. **Overview**

2. **Introduction**

3. **MVC**
    1. Design Pattern
    2. JW01's Overview
    3. Trygve Reenskaug's Formulation
    4. VS and MVC

4. **System**
    1. Dependency Injection
    2. Configuration
    3. Context
        1. BOOL(ean) properties
        2. CHAR(acter) properties
        3. DOUBLE properties
        4. DWORD properties
        5. FLOAT properties
    4. Framework
    5. Versioning

5. **Application**
    1. Interface
    2. Configuration
    3. Context
        6. System and MVC
        7. Configuration
        8. Data
        9. Graphic Interface
        10. Element Context and Display Lists
        11. Graphic Context
        12. Object Configuration
    4. Framework
    5. API
    6. View Callback
    7. Development
    8. Versioning

6. **Graphics**
    1. Context
    2. Framework
    3. API
    4. Display Callback
    5. Control Callback

Appendices:

A. Trygve Reenskaug – MVC Formulation
B. JW01 - MVC is all about separation of concerns
C. Hierarchical Description Language (HDL) Files
D. Application Description File
E. System Description File
F. View Description File
G. Versioning
H. C++ Conventions

Illustrations:

1. Virtual Surface Component Diagram
2. Class Diagram
3. System Class Diagram
4. Application (MVC) Class and (Empty) Diagrams
5. Application Interface Include Diagram
6. Application Process Diagram
7. Model Process Diagram
8. View Class Diagram
9. Control Process Diagram
10. Graphics Class Diagram
11. Element Context Class Diagram
12. Graphics Private Implementations and Components

Notes:

1. Please note that the all code snippets are given to illustrate process and conventions, and are not complete implementations.

1. Overview

Virtual Surface (VS) provides a developer with a Managed Application Development Environment (MADE). This is comprised of an application development framework SDK, a Managed Application API, and a Graphics API.

VS is designed to rapidly develop graphical applications, in which all of the System and Graphics, and much of an applications "housekeeping" are managed by VS. This results in the minimum of new application design, coding and documentation being required.

VS takes its name from a computer graphics virtual 2D or 3D plane or shaped mesh primitive, created either functionally or procedurally, or by using a graphics application such as a 3D editing system.

Recently a non-graphical console based version of VS, VSX, has been created. This shares much of the same structure and code base as VS. Specific reference to VSX will indicated by prefixing paragraphs with "**VSX**".

2. Introduction

Even the simplest graphic primitives, whether they are 2D or 3D, require a complex set of functions and programming steps to initialise, create and then draw them in the window of an application. For animation and interaction this must also all happen within a real-time framework that can handle a multitude of different system events such as a window being resized, minimised, maximised and closed; a mouse location being updated; and a key on a keyboard being pressed down and then released.

This complexity is substantially increased by the requirements of a graphics application to draw graphics primitives using a special hardware device; the Graphics Processing Unit (GPU). This device must, like the application, be initialised and a context created with properties, such as pixel width and height of an application, and the graphical capabilities of each PC. There are many system events, such as the window of an application being resized, when this context must be updated, during which time it's context described as being "lost", and when re-initialised it's context is described as "regained".

The real-time system framework, its management and interaction with an application, and the graphics API's can be the same for every application. There are also many parts of (once developed) applications which are the same, or which are reusable, between different applications.

VS is comprised of three functionally separate parts; System, Application and Graphics. The design pattern for each of these is based on the Model, View Controller (MVC) Design Pattern, all of which are based on a real-time framework comprised of five principle functional parts; setup, setup graphics (invoked at run time and when context is "regained"), display, cleanup graphics (invoked when context is "lost") and cleanup.

The System and Graphics parts are provided, and as the View component of the Application Implementation MVC Design Pattern (see below) can be functionally the same for every application it is encapsulated within an Application API, and so in a VS based application it is only the Control and Model components that need to be developed. As mentioned above much of these can be reused or repurposed from examples or other applications.

This document describes the MVC Design Pattern, then the three functionally separate parts of VS; System, Application and Graphics. The Appendices and Diagrams are provided in a separate document so that they can be read and viewed in conjunction with this document.

3. MVC

3.1 Design Pattern

A design pattern is a general solution to a common software problem. The term was made popular by the book Design Patterns: Elements of Reusable Object-Orientated Software.

The VS architecture uses many different design patterns and techniques, including the Pimpl idiom, Singleton Method, Monotone Method, Dependency Injection Method, and the Proxy, Adapter, Facade, Observer and as mentioned above various MVC Design Patterns.

The most important – in that it defines the whole shape of the VS architecture is the MVC Design Pattern.

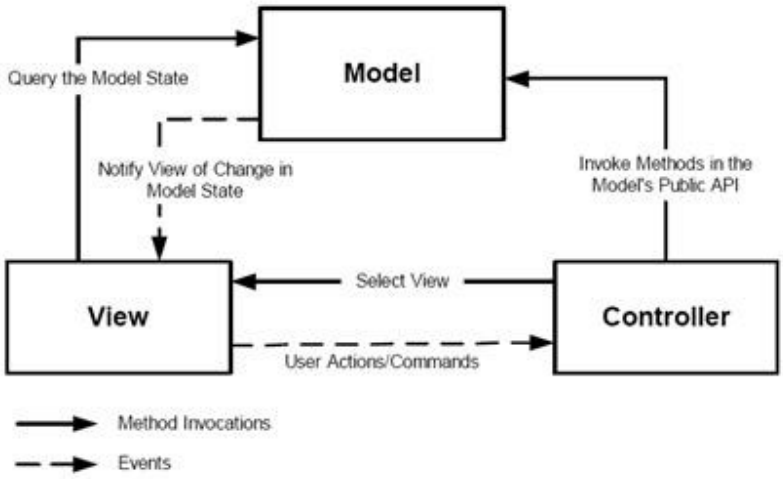


Fig x: MVC diagram.

As such, before describing it's usage in VS, this document provides two introductory texts; JW01's Overview from Programmers Stack Exchange and Trygve Mikkjel Heyerdahl Reenskaug's Formulation.

3.2 JW01's Overview

"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 box."

"The box is the model, the photographer is the controller and the camera is the view."

"Because the box does not know about the camera or the photographer, it is completely independent. This separation allows the photographer to walk around the box and point the camera at any angle to get the shot/view that he wants."

"Non-MVC architectures tend to be tightly integrated together. If the box, the controller and the camera were one-and-the-same-object then, we would have to pull apart and then re-build both the box and the camera each time we wanted to get a new view. Also, taking the photo would always be like trying to take a selfie - and that's not always very easy."

JW01: Programmers Stack Exchange
Date: 18 December 2012

A copy of his notes on MVC is included in this document as Appendix A.

3.3 Trygve Reenskaug's Formulation

Trygve Mikkjel Heyerdahl Reenskaug is a Norwegian computer scientist and was professor emeritus of the University of Oslo. He formulated the model-view-controller (MVC) pattern for graphical user interface (GUI) software design in 1979 while visiting the Xerox Palo Alto Research Center (PARC).

Reenskaug wrote:

"MVC was conceived as a general solution to the problem of users controlling a large and complex data set. The hardest part was to hit upon good names for the different architectural components. Model-View-Editor was the first set. After long discussions, particularly with Adele Goldberg, we ended with the terms Model-View-Controller."

A copy of his definition of MVC is included in this document as Appendix B.

3.4 VS and MVC

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).

It could be argued that the Model component of the System MVC design pattern should be Application and not Application Interface (with overlapping MVC design patterns). Functionally the Application Interface provides a real separation between System and an application, furthermore when used to provides an interface to multiple applications the separation of the MVC patterns is clearly correct.

4. System

The System is designed to be a "black box", and as such it should not be necessary for a developer to know what it does, just that it does it well – all the time – every time.

However, an in-depth understanding of the "System" part is helpful to achieving the best usage of VS as a whole.

System is designed to manage all high and low level system functionality and resources.

This is achieved by providing:

- Internal resource management (e.g. images, icons),
- External resource management (e.g. configuration files, images, 2D and 3D data sets),
- Structured code organisation,
- Structured system, project, library and application folders,
- Compile dependencies, and

- Application selection via the application interface.

When a VS application is compiled, this process includes the icon and image resources into the executable. It also, via the application interface header file, selects the VS application.

When a VS application is “run” these resources are “read” from the executable file into the application and stored.

The System loads the required configuration files from the “data” folder, initialises and then configures the System Context, and then via the Application Interface, allocates and initialises the selected application. It then invokes this applications Setup method. Using the System Context it then creates, initialises and then opens an application Window. Using the System Context it then creates and initialises the Graphics Context, gains the graphics device context, and when successful, invokes an applications SetupGraphics method. Control is then passed to a message pump function, which after initialising various System Context temporal properties, then loops until an “exit” or “close” event, during which applications Display method is invoked at the required frame rate.

When the graphics device context is lost, the System invokes applications CleanupGraphics Method, then when context is regained, invokes an applications SetupGraphics method.

When an application is closed the System invokes an applications Cleanup method.

4.1 Dependency Injection

To provide a VS application with access to the current system context, current system input, output to a single log file and console, and the Graphics API, a single instance of the AppDependency class is used as a dependency container and injected throughout the entire VS hierarchy (passed by reference) at runtime via the Setup method of each MVC Control component.

The Application API is comprised of multiple components, and as such, it is recommended that these be allocated and constructed as required.

The example given below shows the injection by reference of an instance of the Virtual Surface AppDependency class into an applications Setup method.

A local reference (or an application class property) is initialised, then as required various accelerated local references initialised.

Example:

```
INT application::Setup(vs_system::AppDependency *app_dep)
{
	// Local
	vs_system::AppDependency *app_dependency;
	vs_system::GfxInterface *gfx_interface;
	vs_system::SysContext *sys_context;
	vs_system::SysInput *sys_input;
	vs_system::SysOutput *sys_output;

	// Inject
	app_dependency = app_dep;

	// Initialise accelerated local references
	gfx_interface = app_dependency->GetGfxInterface();
	sys_context = app_dependency->GetSysContext();
```

```
        sys_input = app_dependency->GetSysInput();
        sys_output = app_dependency->GetSysOutput();
}
```

These classes are depicted in Diagrams 2 (Class Diagram), 3 (System Class Diagram) and 4 (Application (MVC) Class and Application (Empty) Diagrams).

4.2 Configuration

Three Hierarchical Description Language (HDL) format files configuration files are required by a VS application; a system, application and view file.

Respectively these files are:

1. sys_config.hdl
2. app_config.hdl
3. view_config.hdl

The HDL file format is detailed in Appendix C.

Examples of these three files are detailed respectively in Appendices D, E and F.

When an application is deployed these are located in the “data” folder which must be collocated with the application executable.

The System and View files are handled by the VS System.

The “sys_config.hdl” file is used to initialise various properties of the System Context, and is described below.

The “app_config.hdl” file is used by an application to initialise those properties and structures, etc., not initialised or managed by the application, and its suggested usage is described below.

The “view_config.hdl” file is used to initialise the layout, properties and structures required to describe an application set of frames (the frameset) and panels, and is described below.

4.3 Context

The VS System Context is comprised of two types of properties, those that CAN be initialised at runtime by the “sys_config.hdl” configuration file, and those can’t.

Setting context is by type.

For example, with a pointer to (`vs_system::SysContext *sys_context`) to the System Context, invoking the following method sets the value of the `VS_APP_VERSION_MAJOR` property:

```
sys_context->SetDWord(
        1,
        VS_APP_VERSION_MAJOR
        );
```

There are similar methods for other types, including SetBool, SetChar, SetDouble and SetFloat.
To get the value of a System Context property, methods including GetBool, GetChar, GetDouble, GetDWord and GetFloat are provided.

An example of the “sys_config.hdl” is detailed in Appendix D.

4.3.1 BOOL(ean) properties

Value set by SetBool method, and retrieved by GetBool method.

	CAN	Notes
CONSOLE	X	
DEBUG	X	
LOG	X	
Window		
CURSOR	X	
TOUCH	X	
WINDOWED	X	
DESKTOP	X	
Window Style		
BORDER_STYLE	X	
MENU_STYLE	X	
OVERLAP_STYLE	X	
ACTIVE	-	
MAXIMISED	-	
MINIMISED	-	
MOVED	-	
MOVING	-	
SIZING	-	

4.3.2 CHAR(acter) properties

Value set by SetChar method, and retrieved by GetChar method.

	CAN	Notes
APP_FOLDER_PATH	-	MAXCHARPATHLEN
PROJECT_FOLDER_PATH	-	MAXCHARPATHLEN
APP_CONFIG_FILE	-	MAXCHARPATHLEN
SYS_CONFIG_FILE	-	MAXCHARPATHLEN
VIEW_CONFIG_FILE	-	MAXCHARPATHLEN
APP_NAME	-	VS_MAXCHARLEN

DOUBLE properties

Value set by SetDouble method, and retrieved by GetDouble method.

TIME_DELTA	CAN	Notes
TIME_ELAPSED	-	
TIME_ACTUAL_INTERVAL	-	
TIME_EXPECTED_INTERVAL	-	
TIME_LAST	-	
TIME_NOW	-	
TIME_START	-	

DWORD properties:

Value set by SetDWord method, and retrieved by GetDWord method.

	CAN	Notes
MSG	-	
Graphics	-	
AAQ	X	
FPS	X	
MIN_WIDTH	X	
MIN_HEIGHT	X	
CLIENT_WIDTH	X	
CLIENT_HEIGHT	X	
MAX_WIDTH	-	
MAX_HEIGHT	-	
PREVIOUS_WIDTH	-	
PREVIOUS_HEIGHT	-	
DESKTOP_WIDTH	-	
DESKTOP_HEIGHT	-	
WINDOW_WIDTH	-	
WINDOW_HEIGHT	-	
WINDOW_STYLE	-	
X_OFFSET	-	
Y_OFFSET	-	
CX_BORDER	-	
CX_TOP	-	
WINDOW_HANDLE	-	
MSG_WPARAM	-	
MSG_LPARAM	-	
APP_VERSION_MAJOR	-	
SYS_VERSION_MAJOR	X	
SYS_VERSION_MINOR	X	
SYS_VERSION_PATCH	X	
SYS_VERSION_BUILD	X	
APP_VERSION_MAJOR	-	
APP_VERSION_MINOR	-	
APP_VERSION_PATCH	-	
APP_VERSION_BUILD	-	

FLOAT properties:

	CAN	Notes
BG_RED	X	
BG_GREEN	X	
BG_BLUE	X	

4.4 Framework

The VS MVC based architecture is based on a real-time framework comprised of five principle functional parts; setup, setup graphics, display, cleanup graphics and cleanup. Respectively these are implemented as methods named Setup, SetupGraphics, Display, CleanupGraphics and Cleanup.

Each of these is implemented as a method, each of which invokes its corresponding hierarchical functional part in the VS three part MVC.

The functionality that these methods provide includes:

1. Setup:
    a. Store injected dependency
    b. Allocate, construct and Setup implementations
    c. Initialise properties
2. SetupGraphics for implementations
3. Display:
    a. Display logic for updating and handling input
    b. Display logic for implementations
    c. Display implementations
4. CleanupGraphics for implementations
5. Cleanup implementations

These methods are respectively invoked:

1. Once at run time,
2. Once at run time and also when "lost" device context is "regained",
3. For each frame,
4. When device context is "lost" and
5. Once on exist or close.

For example:

At runtime The VS System automatically creates an instance of an application and then invokes via the application interface its Setup and SetupGraphics methods. The VS System then passes control to a message handling function that loops at a specific time increment per frame until exist or close. For each frame the VS System invokes via the application interface an applications Display method. If the graphics context is "lost", the VS System invokes via the application interface the CleanupGraphics method, and when graphics context is “regained” the SetupGraphics method. The VS System invokes via the application interface the Cleanup method on exit or close. Within an application all dependent framework methods must be hierarchically invoked in turn, e.g. an application Setup method must invoke it’s Setup methods and also those instances of allocated and initialised objects, and so on.

These methods are detailed in illustrations 6 (Application Process Diagram), 7 (Model Process Diagram), and 9 (Control Process Diagram).

4.5 Versioning

VS System versioning is based on Semantic Versioning 2.0.0.

Version numbers are typically assigned in increasing order and correspond to new developments in the software.

VS System versioning depends upon a system header file “vs/vs_system/header/vs_sys_version.h”.

An example is shown below:

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

// ---------- vs_sys_version.h ----------
/*!
\file vs_sys_version.h
\brief VS System [Major].[Minor].[Patch].[Build] version
\author Gareth Edwards
*/
```

```
// ---------- VS System [Major].[Minor].[Patch].[Build]version ----------
//
// Note: Based on Semantic Versioning 2.0.0
// http://semver.org/
//
// Summary
//
// Given a version number MAJOR.MINOR.PATCH, increment the:
//
// MAJOR version when you make incompatible API changes,
// MINOR version when you add functionality in a backwards
//       compatible manner, and
// PATCH version when you make backwards - compatible bug fixes.
//
// BUILD (metadata) SHOULD be ignored when determining version precedence
//

// Major:
// 1. Including, architecture, functionality, layout, etc., to 20/4/15
// 2. Including, architecture, functionality, layout, etc., to  1/9/15
//    Changed:
//    Major update to application control
#define VS_SYS_VERSION_MAJOR 2

// Minor:
// 5. Including all functionality to 20/4/15
// 6. Including all functionality to 1/9/15
#define VS_SYS_VERSION_MINOR 6

// Patch:
// 1. Including all functionality to 1/9/15
#define VS_SYS_VERSION_PATCH 1

// Build:
// 1. Including all functionality to 1/9/15
#define VS_SYS_VERSION_BUILD 1
```

This is included when VS is compiled and used to set the "VS_SYS_VERSION" properties of the System Context class SysContext.

Semantic Versioning 2.0.0 is detailed in Append G.

VSX: System versioning depends upon a system header file "vsx/vsx_system/header/vsx_sys_version.h".

5. Application

Application

5.1 Interface

VS is designed such that it can be used to develop and manage multiple applications by selecting a target application framework from a set of application frameworks.

The selected applications can either be

A VS utility app_setup.exe (source code and project supplied in the VS utilities folder) takes as its arguments the application project name (and also its name space) and applications code prefix (and also the folder suffix for resources such as icons and images), and generates a header file (“vs_app_include.h”) that is included when an application is compiled, by the top level System source file (“vs_sys_windows.cpp”), the application interface header file (“vs_app_interface.h”) and the application interface source file (“va_app_interface.cpp”).

Before each “#include” a “#define” specifies which part of the include file is to be used.

For example:

```
// Set application name
#define VS_APPLICATION_APPNAME
#include "../../vs_application/header/vs_app_setup.h"
```

Which when VS is compiled sets the application name:

```
// ---------- NAME ----------
//
//Included in vs_sys_windows.cpp
//
// Step 2: Initialise char *app_name = "[namespace]";
//
#ifdef VS_APPLICATION_APPNAME
char *app_name = "review_02";
#undef VS_APPLICATION_APPNAME
#endif
```

Which can then be set as the value of the System context by:

```
// Set application name
virtual_surface.sys_context->SetChar(
	virtual_surface.sys_context->APP_NAME,
	app_name
	);
```

For example, using an "app_setup.exe" utility application - which is located in the main VS folder - in a command prompt window:

- app_setup vs review_02 r2

Sets the setup utility application:

- to generate a VS setup file ("vs_app_setup.h"),

- source location as “vs/applications/ review _02/” folder, and
- namespace as “review _02”.

And requires that all:
- source and header files are prefixed by “r2” (e.g. “r2_application”, “r2_model”, “r2_control”),
- resource icons are located in “vs/resources/icons/icons_r2/” (or “icons_vs”) and
- resources images files are located in “vs/resources/icons/images_r2/” (or “images_vs”).

This sets a specific application name (and which can be referenced via the injected dependency instance of the system context), an applications version information, an applications header path, a “socket” (which is a union of class declarations, providing a mechanism for building a single application comprised of many separate applications) and set a pointer to a interface method that handles by reference the five framework functions.

This is detailed in illustration 5 (Application Interface Include Diagram).
The “app_setup” command, invokes the “app_setup.cmd” script file located in users “bin” directory. The “bin” can be added to the PATH environment variable by adding (via Advanced system settings -> Environment Variables) %USERPROFILE%\bin.

An example of this file:

```
@echo off
K:
cd "[VS_PATH]\VS\vs_01060101_150903\vs\vs_application\header"
"app_setup.exe"
```

An icon with the same naming convention, but located in a different folder can be selected by adding a further command line argument, e.g.:

- app_setup vs review_02 r2 c2

This requires that:
- resource icons are located in “vs/resources/icons/icons_c2/” (or “icons_vs”) and
- resources images files are located in “vs/resources/icons/images_c2/” (or “images_vs”).

VSX: A VSX application can be setup in the same way, by substituting "vsx" for "vs", e.g.:

- app_setup vsx camera_03 c3 c2

5.2 Configuration

There are three configuration files required by a VS application.

When an application is deployed these are stored in a “data” folder that is collocated with the application executable. Typically an IDE will manage this such that the executable is located in a “Release” or “Debug” folder.

These are Hierarchical Description Language (HDL) format files.

This is detailed in Appendix C.

These files are:
- sys_config.hdl (system)

- app_config.hdl (application)
- view_config.hdl (view)

These are detailed in respectively in Appendices D, E and F.

5.3 Context

Detailing line by line VS context is beyond the scope of this introductory document.

The following details are intended to provide in outline only, core concepts of VS application context, and how this is used to manage an application, and to its display.

Please see the supplied code examples for further information and examples of usage.

5.3.1 System and MVC

An applications context is primarily comprised of:

```
	// VS dependency injection object
	vs_system::AppDependency *app_dependency;
```

An applications VS system dependency context is injected at runtime when the system passed by reference a vs_system::AppDependency object.

And:

```
	// MVC context objects
	vs_system::AppView	*app_view;
	review_02::Control	*control;
	static review_02::Model	*model;
```

An applications MVC design pattern should also be created, and setup, at runtime, e.g.

```
INT application::Setup(vs_system::AppDependency *app_dep)
{
	// Local
	INT result = 0;

	// Store application dependency pointer
	app_dependency = app_dep;

	// Allocate and construct a VS system supplied View object
	app_view = new vs_system::AppView();

	// Allocate and construct application supplied Model and Control objects
	control = new review_02::Control();
	model = new review_02::Model();

	// ----- MVC: Setup and configure -----
	result = model->Setup(app_dependency);
	result = app_view->Setup(app_dependency);
	result = control->Setup(app_dependency, app_view, model);

	// More setup code...

}
```

When a VS system View object is allocated and then created the VS system uses the "app_config.hdl" file to configure the required application framesets (system::AppFrame class) and panels (system::AppPanels class).

An application can "switch" between different framesets via the system::AppFrame class API by "selecting" a frameset.

An application Display method invokes the system::AppView objects Display method to display the currently "selected" frameset, whose panels and content are then rendered.

To comply with the functional requirements of the MVC design pattern, it is recommended that an application Display method invokes a Model Display method that invokes the AppView Display Method.

The Model and Control objects are provided by the developer.

Considerable support for both Model and Control objects is supplied in the provided code examples, and it is strongly recommended copies of these are developed upon.

Please see the supplied code examples for further information and examples of usage.

5.3.2 Configuration

The "app_config.hdl" file, which is loaded at runtime, and supplied by reference to an applications Setup method.

Please see the supplied code examples for further information and examples of usage.

5.3.3 Data

Objects and Data managed entirely by an application.

5.3.4 Graphic Interface

To comply with the functional requirements of the MVC design pattern, it is recommended that only an application Model object uses the `vs_system::GfxInterface` API.

A reference to the `vs_system::GfxInterface` can be accessed through the application dependency object injected into an application at runtime, e.g.:

```
// Set shortcut to an injected dependency property
vs_system::GfxInterface *gfx = app_dependency->GetGfxInterface();
```

5.3.5 Element Context and Display Lists

The Graphic Interface object provides access to a viewport render engine.

A viewport "owns" one or more display lists comprised of a list of linked `vs_system::ElmContext` object's.

5.3.6 Graphic Context

Each `vs_system::ElmContext` object is comprised of graphical render context flags, graphical data buffers, and shares with all other `vs_system::ElmContext` object's in a display list a reference to `vs_system::GfxContext`, which stores graphical context data and also references to three user (or where provided, system) defined callback methods, the first of which "defines" a graphic object, and the second and third "fill" the two graphical data buffers (vertex and vertex index).

5.3.7 Object Configuration

System defined graphical objects are available via the `vs_system::ObjConfig` library object.

5.4 Framework

Five application class framework methods are required.

An applications lifecycle is managed through these methods, as is the entire logic of an application.

These are invoked by the Application interface.

An example is shown below:

```
INT application::Setup(vs_system::AppDependency *app_dep)
{
	return VS_OK;
}

INT application::SetupGraphics()
{
	return VS_OK;
}

INT application::Display()
{
	return VS_OK;
}

INT application::CleanupGraphics()
{
	return VS_OK;
}

INT application::Cleanup ()
{
	return VS_OK;
}
```

See section 6.1 Framework, for further information.

5.5 API

As can be understood from the outline detailing of application management and display usage above, the VS application API is comprised of methods belonging to numerous different VS system supplied objects.

Most of these are managed by VS.

The exception is the `vs_system::AppView` object.

This must be exposed, and partially managed by an application as it provides access to the viewport render engine, framesets, and panels.

See section 6. Graphics, for further information.

5.6 View Callback

When the `vs_system::AppView` object Display method is invoked, it is provided with a reference to a callback Model method.

This method is invoked for each viewport, thus allowing a VS system managed object to invoke a method, and thus not requiring an API for the `vs_system::AppView`.

See section 6.3 Display View Callback, for further information.

5.7 Development

VS is complicated. Very complicated, and is comprised of many interlocking components.

It is therefore strongly advised that a Developer starts from the supplied code examples. These are well documented exemplars of the MVC design patterns detailed in this document, and detail how VS application frameworks can be best used.

5.8 Versioning

VS Application versioning is based on Semantic Versioning 2.0.0.

Version numbers are typically assigned in increasing order and correspond to new developments in the software.

VS System versioning depends upon an application header file “[name]_version.h”.

An example is shown below:

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

// ---------- [project prefix]_version.h ----------
/*!
\file [project prefix]_version.h
\brief VS Application [Major].[Minor].[Patch].[Build] version
\author Gareth Edwards
\note Vanilla C++
*/

// ---------- VS Application [Major].[Minor].[Patch].[Build] version ----------
//
// Note: Based on Semantic Versioning 2.0.0
// http://semver.org/
//
// Summary
//
// Given a version number MAJOR.MINOR.PATCH, increment then update:
//
// MAJOR version when you make incompatible API changes,
// MINOR version when you add functionality in a backwards compatible manner, and
// PATCH version when you make backwards compatible bug fixes.
//
// BUILD (metadata)SHOULD be ignored when determining version precedence
```

```
//

// Major:
// 1. Including, architecture, functionality, layout, etc., to 20/4/15
// 2. Including, architecture, functionality, layout, etc., to 1/9/15
//    Changed:
//    Control structure (see VS versions)
#define VS_APP_VERSION_MAJOR 2

// Minor:
// 5. Including all functionality to 20/4/15
//    Including IDS, JPLY, PIC cameras, 555 Streaming
//    Fixed:
//    Control - Persistant input on deselecting CLICK action (25+/4/15)
//    Control - Persistant right mouse button action (25+/4/15)
//    Added:
//    ObjConbfig - Implement Cylindrical mapping
//    Application - Run-time auto_load of last (or designated) archive (25+/4/15)
//    Shared Memory - Implement V1 test-bed (OFF via compile pre-processor for
distribution) (27+/4/15)
//    Library ObjConfig - button method (1/5/15)
//    Model - Capture settings panel (2/5/15)
//    Library - CamConfig - camera configuration object (3/5/15)
//    Control - Implement interface to CamConfig (3/5/15)
// 6. Including all functionality to 1/9/15
#define VS_APP_VERSION_MINOR 6

// Patch:
// 2. Including all functionality to 20/4/15
// 3. Including all functionality to 1/9/15
#define VS_APP_VERSION_PATCH 3

// Build:
// 1. JS on 20/4/15
// 2. Singapore on 22/4/15

// 4. PK on 11/5/15 for Patrol 1001
// 5. Including all functionality to 1/9/15
#define VS_APP_VERSION_BUILD 4
```

This is included when VS is compiled and used to set the "VS_APP_VERSION" properties of the System Context class SysContext.

Semantic Versioning 2.0.0 is detailed in Append G.

6. Graphics

Many of the points detailed below have already been mentioned in section 5. above.

6.1 Framework

The instance of the Graphics API (GfxInterface) class passed by reference by dependency injection, will require its framework Setup, SetupGraphics, Display, CleanupGraphics and Cleanup methods to be invoked by the application's respective framework methods.

For example:

```
vs_system::AppDependency *app_dependency;
vs_system::GfxInterface *gfx_interface;

INT application::Setup(vs_system::AppDependency *app_dep)
{
	// Inject
	app_dependency = app_dep;
	gfx_interface = app_dependency->GetGfxInterface();
	// Note: the Graphics API has already been setup by VS system
	return VS_OK;
}

INT application::SetupGraphics()
{
	INT setup_graphics = gfx_interface->SetupGraphics();
	return setup_graphics;
}

INT application::Display()
{
	// No action
	return VS_OK;
}

INT application::CleanupGraphics()
{
	INT cleanup_graphics = gfx_interface->CleanupGraphics ();
	return cleanup_graphics;
}

INT application::Cleanup ()
{
	// Note: the Graphics API will be cleaned up by VS system
	return VS_OK;
}
```

If for logical MVC design all graphics operations are confined to an applications Model class, then:

```
vs_system::AppDependency *app_dependency;
Model *model;

INT application::Setup(vs_system::AppDependency *app_dep)
{
```

```
        // Inject
        model = new Model();
        INT setup_model = model->Setup(app_dependency);
        return setup_mode;
}

INT application::SetupGraphics()
{
        INT setup_graphics = model->SetupGraphics();
        return setup_graphics;
}

INT application::Display()
{
        // No action
        return VS_OK;
}

INT application::CleanupGraphics()
{
        INT cleanup_graphics = model->CleanupGraphics ();
        return cleanup_graphics;
}

INT application::Cleanup ()
{
        if ( model != NULL ) delete model;
        return VS_OK;
}
```

And:

```
vs_system::AppDependency *app_dependency;
vs_system::GfxInterface *gfx_interface;

INT Model::Setup(vs_system::AppDependency *app_dep)
{
        // Inject
        app_dependency = app_dep;
        return VS_OK;
}

INT Model::SetupGraphics()
{
        INT setup_graphics = gfx_interface->SetupGraphics();
        // Note: the Graphics API has already been setup by VS system
        return setup_graphics;
}

INT Model::Display()
{
        // No action
        return VS_OK;
}
```

```
INT Model::CleanupGraphics()
{
	INT cleanup_graphics = gfx_interface->CleanupGraphics ();
	return cleanup_graphics;
}

INT Model::Cleanup ()
{
	// Note: the Graphics API will be cleaned up by VS system
	return VS_OK;
}
```

6.2 API

From runtime to exit or close, the VS system will handle all system events, update system context, update system input status, and invoke as appropriate an applications constructor, destructor and framework methods.

No work is required by a developer to manage an application's life cycle.

This is not the case for the instance of the Graphics API (GfxInterface) class passed by reference by dependency injection to an application.

The reasons for this are straight forward: the Graphics API is the palette with which a developer "paints" a picture, and no two pictures – no two applications – are ever the same; an application will have to interact with Graphics API to create, and update graphic primitives; an application will also have to provide a number of callback functions which the Graphics API will invoke when required, and which are implemented in the Application Implementation Control and Model classes; etc.

However, the VS Graphics API does manage everything concerned with the GPU.

This includes managing:

- An atlas of run-time loaded resource images which can be referenced as textures by graphic primitives,
- Temporary application images,
- Text fonts,
- The lifecycle of graphic primitives,
- The lifecycle of interactive elements such as buttons,
- Display lists of graphic primitives,
- Rendering setup,
- Rendering modes,
- Rendering 2D orthogonal and 3D perspective viewports,
- Rendering 2D viewrects,
- Matrix stack transformations, and
- Callbacks from interactive elements.

The Graphics API class (GfxInterface), and it's MVC Design Pattern are depicted in illustrations 10 (Graphics Class Diagram), 11 (ElmContext Class Diagram) and 12 (Graphics Private Implementations and Components).

6.3 Display View Callback

For logical MVC design all graphics operations are usually confined to an applications Model class.

Given the framework requirements there must be a Model class Display method.

For example:

```
vs_system::AppDependency *app_dependency;
Model *model;
vs_system::AppView *app_view;

INT application::Setup(vs_system::AppDependency *app_dep)
{
	// Inject
	app_dependency = app_dep;

	// Model: Allocate and initialise, then setup
	model = new Model();
	INT setup_model = model->Setup(app_dependency);

	// View: Allocate and initialise, then setup
	app_view = new vs_system::AppView();
	// This method is shown below
	INT setup_app_view = app_view->Setup(app_dependency);
	return setup_app_view;
}

INT application::SetupGraphics()
{
	INT setup_graphics = model->SetupGraphics();
	return setup_graphics;
}

INT application::Display()
{
	// Recursively display all frames (AppFrame class)
	// and the panels (AppPanel class), invoking the
	// callbackDisplay method shown below for each
	// panel.
	app_view->Display((INT)this, &callbackDisplay);
	return VS_OK;
}

INT application::CleanupGraphics()
{
	INT cleanup_graphics = model->CleanupGraphics ();
	return cleanup_graphics;
}

INT application::Cleanup ()
{
	if ( model != NULL ) delete model;
	if ( app_view != NULL ) delete model;
	return VS_OK;
}

INT application::callbackDisplay(
		INT app_cast_to_int,
		vs_system::AppPanel *p
```

```
        )
{
        application* app = (application *)app_cast_to_int;
        // Note: implementation of this method not shown
        // in this example.
        app->model->Display(p->GetGfxContext(), NULL);
        return VS_OK;
}
```

And the AppView Setup method:

```
INT AppView::Setup(AppDependency *app_dep)
{
        // Inject
        pi_app_view->app_dependency = app_dep;

        // Configure
        VNode *v = pi_app_view->app_dependency->GetViewConfigData();
        Configure(v);

        // Selected frame initialised as first frame
        pi_app_view->selected_frame = pi_app_view->first_frame;

        // Setup
        AppFrame* frame = pi_app_view->first_frame;
        while (frame)
        {
                frame->Setup(pi_app_view->app_dependency);
                // Select "home" frame - update below
                if ( frame->GetHome() ) pi_app_view->selected_frame = frame;
                frame = frame->GetNext();
        }

        // Done
        return VS_OK;
}
```

6.4 Control Callback

As indicated above, for logical MVC design all graphics operations are usually confined to an applications Model class.

However, following the framework requirements there must be a Model class Display method.

Appendix A

JW01 - MVC is all about separation of concerns

Source: Programmers StackExchange
URL: http://programmers.stackexchange.com/users/6720/jw01?tab=profile
Date: 18 December 2012

The Model is responsible for managing the program's data (both private and client data). The View/Controller is responsible for providing the outside world with the means to interact with the program's client data.

The Model provides an internal interface (API) to enable other parts of the program to interact with it. The View/Controller provides an external interface (GUI/CLI/web form/high-level IPC/etc.) to enable everything outwith the program to communicate with it.

The Model is responsible for maintaining the integrity of the program's data, because if that gets corrupted then it's game over for everyone. The View/Controller is responsible for maintaining the integrity of the UI, making sure all text views are displaying up-to-date values, disabling menu items that don't apply to the current focus, etc.

The Model contains no View/Controller code; no GUI widget classes, no code for laying out dialog boxes or receiving user input. The View/Controller contains no Model code; no code for validating URLs or performing SQL queries, and no original state either: any data held by widgets is for display purposes only, and merely a reflection of the true data stored in the Model.

Now, here's the test of a true MVC design: the program should in essence be fully functional even without a View/Controller attached. OK, the outside world will have trouble interacting with it in that form, but as long as one knows the appropriate Model API incantations, the program will hold and manipulate data as normal.

Why is this possible? Well, the simple answer is that it's all thanks to the low coupling between the Model and View/Controller layers. However, this isn't the full story. What's key to the whole MVC pattern is the direction in which those connection goes: ALL instructions flow from the View/Controller to the Model. The Model NEVER tells the View/Controller what to do.

Why? Because in MVC, while the View/Controller is permitted to know a little about the Model (specifically, the Model's API), but the Model is not allowed to know anything whatsoever about the View/Controller.

Why? Because MVC is about creating a clear separation of concerns.

Why? To help prevent program complexity spiralling out of control and burying you, the developer, under it. The bigger the program, the greater the number of components in that program. And the more connections exist between those components, the harder it is for developers to maintain/extend/replace individual components, or even just follow how the whole system works. Ask yourself this: when looking at a diagram of the program's structure, would you rather see a tree or a cat's cradle? The MVC pattern avoids the latter by disallowing circular connections: B can connect to A, but A cannot connect to B. In this case, A is the Model and B is the View/Controller.

BTW, if you're sharp, you'll notice a problem with the 'one-way' restriction just described: how can the Model inform the View/Controller of changes in the Model's user data when the Model isn't even allowed to know that the View/Controller, never mind send messages to it? But don't worry: there is a solution to this, and it's rather neat even if it does seem a bit roundabout at first. We'll get back to that in a moment.

In practical terms, then, a View/Controller object may, via the Model's API, 1. tell the Model to do things (execute commands), and 2. tell the Model to give it things (return data). The View/Controller layer pushes instructions to the Model layer and pulls information from the Model layer.

And that's where your first MyCoolListControl example goes wrong, because the API for that class requires that information be pushed into it, so you're back to having a two-way coupling between layers, violating the MVC rules and dumping you right back into the cat's cradle architecture that you were [presumably] trying to avoid in the first place.

Instead, the MyCoolListControl class should go with the flow, pulling the data it needs from the layer below, when it needs it. In the case of a list widget, that generally means asking how many values there are and then asking for each of those items in turn, because that's about the simplest and loosest way to do it and therefore keeps what coupling there is to a minimum. And if the widget wants, say, to present those values to the user in nice alphabetical order then that's its perogative; and its responsibility, of course.

Now, one last conundrum, as I hinted at earlier: how do you keep the UI's display synchronised with the Model's state in an MVC-based system?

Here's the problem: many View objects are stateful, e.g. a checkbox may be ticked or unticked, a text field may contain some editable text. However, MVC dictates that all user data be stored in the Model layer, so any data held by other layers for display purposes (the checkbox's state, the text field's current text) must therefore be a subsidiary copy of that primary Model data. But if the Model's state changes, the View's copy of that state will no longer be accurate and needs to be refreshed.

But how? The MVC pattern prevents the Model pushing a fresh copy of that information into the View layer. Heck, it doesn't even allow the Model to send the View a message to say its state has changed.

Well, almost. Okay, the Model layer isn't allowed to talk directly to other layers, since to do so would require it knows something about those layers, and MVC rules prevent that. However, if a tree falls in a forest and nobody's around to hear it, does it make a sound?

The answer, you see, is to set up a notifications system, providing the Model layer with a place it can announce to no-one in particular that it has just done something interesting. Other layers can then post listeners with that notification system to listen for those announcements that they're actually interested in. The Model layer doesn't need to know anything about who's listening (or even if anyone is listening at all!); it just posts an announcement and then forgets about it. And if anyone hears that announcement and feels like doing something afterwards - like asking the Model for some new data so it can update its on-screen display - then great. The Model just lists what notifications it sends as part of its API definition; and what anyone else does with that knowledge is up to them.

MVC is preserved, and everyone is happy. Your application framework may well provide a built-in notifications system, or you can write your own if not (see the 'observer pattern').

Appendix B

Trygve Reenskaug - MVC Formulation

Date: 10 December 1979

MODELS - VIEWS - CONTROLLERS

MODELS

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).

VIEWS

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 have to 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.)

CONTROLLERS

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.

Appendix C

Hierarchical Description Language (HDL) Files

Example:

```
#DOCTYPE HDL4

define{
   margin_width:1;
   border_width:1;
}
```

Appendix D

System Description File

Example:

```
#DOCTYPE HDL4
main{
   Title:Camera;
   Console:1;
   Debug:1;
   Log:1;
   Log-Files:5;
   App-Data:1;
   Graphics{
      AAQ:2;
      FPS:60;
     Border-Width:4;
      Background-Colour{
         Red:0.0;
         Green:0.0;
         Blue:0.0;
      }
   }
   Window{
      Cursor:1;
      Touch:0;
      Desktop:0;
      Windowed:1;
      Dimension{
         Min-Height:256;
         Min-Width:256;
         Width:1000;
         Height:700;
      }
      Style{
         Border:0;
         Menu:0;
         Overlap:1;
      }
   }
}
```

Appendix E

Application Description File

Example:

```
#DOCTYPE HDL4
main{

   Camera{
      Config File:0;
      Config Lighting:1;
   }

   Archive{
      Mode:1;
      Compression:85;
      Max Stack:12;
   }
}
```

Appendix F

View Description File

Example:

```
#DOCTYPE HDL4

define{
   margin_width:1;
   border_width:1;
   spherical_FOV:64.36;
   spherical_tilt:-12.18;
   spherical_ur_FOV:48.0;
   spherical_ur_tilt:-19.0;
}

library{

   FRAMES{
      Panel{
         Name:Title;
         Display:Title;
         Background:0;
         Margin{ Width:1;}
         Border{ Width:1;}
      }
      Panel{
         Name:Frames;
         Display:Frames;
         Background:0;
         Margin{ Width:1;}
         Border{ Width:1;}
      }
   }

   GUI{
      Panel{
         Name:Title;
         Display:Title;
         Background:0;
         Margin{ Width:$margin_width;}
         Border{ Width:$border_width;}
      }
      Panel{
         Name:Frames;
         Display:Frames;
         Background:0;
         Margin{ Width:$margin_width;}
         Border{ Width:$border_width;}
      }
   }

   COLOR{
      Background-Colour{
```

```
         Red:0;
         Green:0.05;
         Blue:0.15;
         Red-Over:0;
         Green-Over:0.1;
         Blue-Over:0.3;
      }
      Texture-Border-Colour{
         Red:0;
         Green:0.05;
         Blue:0.15;
         Red-Over:0;
         Green-Over:0.1;
         Blue-Over:0.3;
      }
   }
}

main{

   Frameset{

      # ----- Frame 1 -----
      Frame{
         $lib:GUI;
         $lib:COLOR;
         Home:1;
         Panel{
            Name:3D Full;
            Display:Spherical;
            Group:3;
            Background:1;
            Margin{
               Width:$margin_width;
            }
            Border{
               Width:$border_width;
            }
            Projection{
               Type:Perspective;
            }
            View{ Type:XYZ; }
         }
      }
      Panel{
         Name:2D Solo;
         Display:Panorama;
         Background:1;
         Margin{ Width:$margin_width; }
         Border{ Width:$border_width; }
      }
   }
}
```

Appendix G

Semantic Versioning 2.0.0

The Semantic Versioning specification is authored by Tom Preston-Werner, inventor of Gravatars and cofounder of GitHub.

License: Creative Commons - CC BY 3.0

Note: Only the introduction is included. The rest of the specification can be found at: http://semver.org

Summary

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Introduction

In the world of software management there exists a dread place called "dependency hell." The bigger your system grows and the more packages you integrate into your software, the more likely you are to find yourself, one day, in this pit of despair.

In systems with many dependencies, releasing new package versions can quickly become a nightmare. If the dependency specifications are too tight, you are in danger of version lock (the inability to upgrade a package without having to release new versions of every dependent package). If dependencies are specified too loosely, you will inevitably be bitten by version promiscuity (assuming compatibility with more future versions than is reasonable). Dependency hell is where you are when version lock and/or version promiscuity prevent you from easily and safely moving your project forward.

As a solution to this problem, I propose a simple set of rules and requirements that dictate how version numbers are assigned and incremented. These rules are based on but not necessarily limited to pre-existing widespread common practices in use in both closed and open-source software. For this system to work, you first need to declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once you identify your public API, you communicate changes to it with specific increments to your version number. Consider a version format of X.Y.Z (Major.Minor.Patch). Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version.

I call this system "Semantic Versioning." Under this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next.

Appendix H

C++ Conventions

The C++ conventions used in VS and applications developed by the author are based on the Google C++

Style Guide.

Currently this Style Guide is at revision 4.45.

Note: Only the Background introduction is included. The rest of the document can be found at: https://google-styleguide.googlecode.com/svn/trunk/cppguide.html

Note: There are many vigorous online debates about the usage of the Google C++ Style Guide, and these can be found in forums such as:

- Reddit (e.g. https://www.reddit.com/r/cpp/comments/289n27)
- Linkedin Pulse (e.g. https://www.linkedin.com/pulse/20140503193653-3046051-why-google-style-guide-for-c-is-a-deal-breaker

Google C++ Style Guide Background

C++ is the main development language used by many of Google's open-source projects. As every C++ programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and maintain.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively.

Style, also known as readability, is what we call the conventions that govern our C++ code. The term Style is a bit of a misnomer, since these conventions cover far more than just source file formatting.

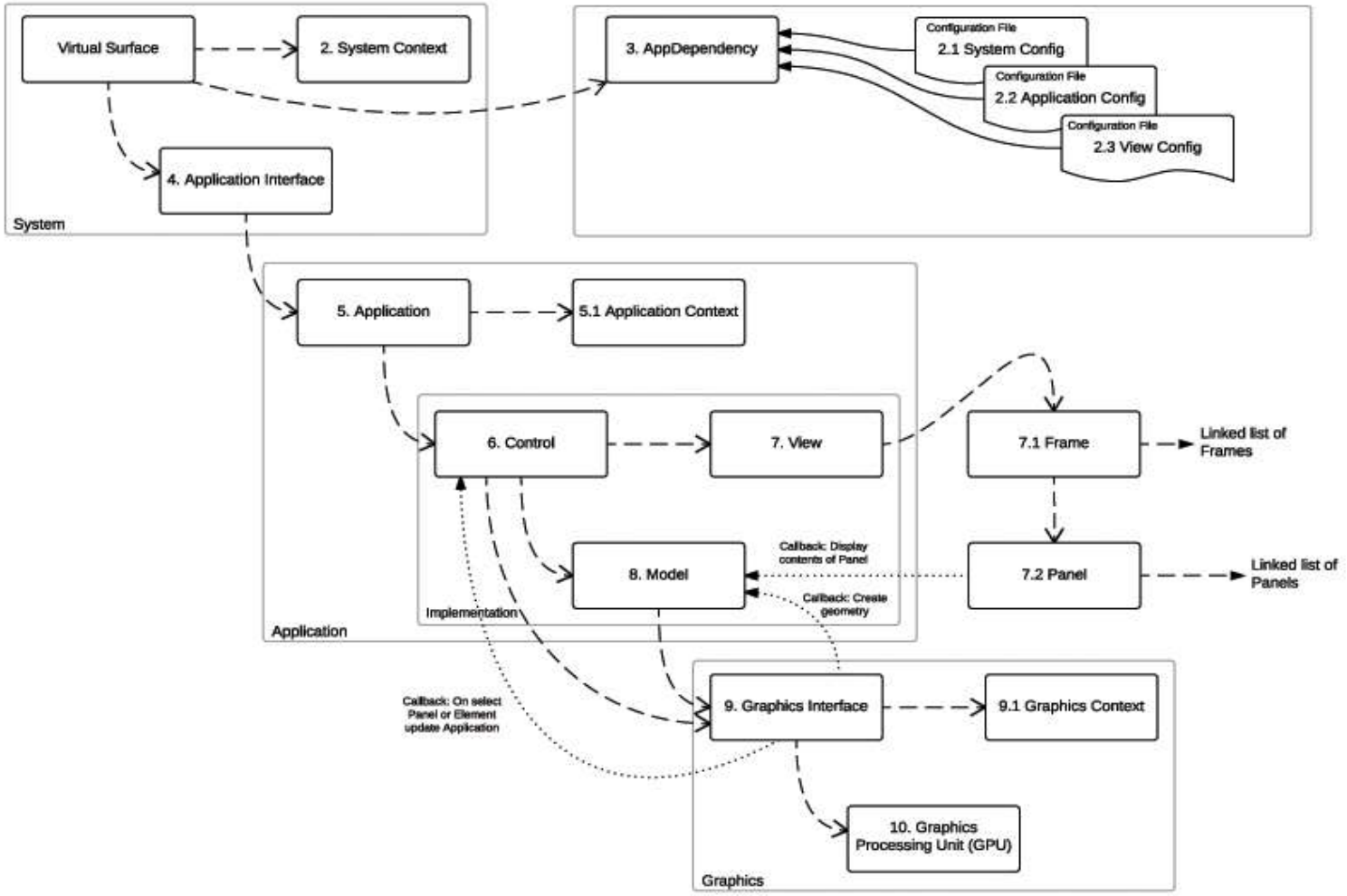
One way in which we keep the code base manageable is by enforcing *consistency*. It is very important that any programmer be able to look at another's code and quickly understand it. Maintaining a uniform style and following conventions means that we can more easily use "pattern-matching" to infer what various symbols are and what invariants are true about them. Creating common, required idioms and patterns makes code much easier to understand. In some cases there might be good arguments for changing certain style rules, but we nonetheless keep things as they are in order to preserve consistency.

Another issue this guide addresses is that of C++ feature bloat. C++ is a huge language with many advanced features. In some cases we constrain, or even ban, use of certain features. We do this to keep code simple and to avoid the various common errors and problems that these features can cause. This guide lists these features and explains why their use is restricted.

Open-source projects developed by Google conform to the requirements in this guide.

Note that this guide is not a C++ tutorial: we assume that the reader is familiar with the language.

1. Virtual Surface Component Diagram



1. Virtual Surface X Component Diagram