
0

TABLE OF CONTENTS

0	Table of Contents	1
1	Introduction	2
1.1	Purpose	2
1.2	Document Conventions	2
1.3	Scope	2
1.4	Document Outline	2
1.5	References	2
2	Description	3
2.1	Product Perspective	3
2.2	Product Functions	3
2.3	User Characteristics	3
2.4	Operating Environment	3
2.5	Implementation Constraints	3
3	System Requirements	4
3.1	External Requirements	4
3.1.1	Software Requirements	4
3.1.2	Hardware Requirements	4
3.2	System Functions	4
3.2.1	Threading Support	4
3.2.2	State Abstraction	4
3.2.3	Resource Management	5
3.2.4	Software Rendering Pipeline	5
3.2.5	Resource Virtualization	5
3.3	Operational Attributes	6
3.3.1	Scalability	6
3.3.2	Maintainability	6
3.3.3	Portability	6
3.3.4	Performance	6
3.3.5	Security	7
4	High-Level Design	8
4.1	Design Overview	8
4.1.1	Initialisation	8
4.1.2	Rendering	8
4.2	Namespace Organisation	8
5	Schedule	10
6	Appendix	11
6.1	Appendix A: Glossary	11
6.2	Appendix B: Issues	11
6.2.1	Implementing Resources Bound to a Rendering Context	11
6.2.2	Format of Data to be Submitted to the Rendering Pipeline	12
6.2.3	Decoupling Vertex Coordinates, Texture Coordinates, Colours and Normals	12
6.2.4	Disposal of Rendering Context Resources	12
6.3	Appendix C: Diagrams	14
6.3.1	Interop Layers	14
6.3.2	Namespace Interdependencies	14
6.3.3	Pipeline Stages and Interactions	15
6.3.4	Preliminary Schedule	15

1

INTRODUCTION

1.1

PURPOSE

The following functional specification is designed to give a broad overview of the proposed 3D Graphics Rendering Framework. This includes a listing and descriptions of the core functions that the 3D Graphics Rendering Framework will need to implement to be useful in practical situations. Where possible, high-level descriptions of how these functions are expected to be implemented will be provided.

1.2

DOCUMENT CONVENTIONS

From here on, the 3D Graphics Rendering Framework will simply be referred to as the Framework. Unless otherwise stated, it can be taken that this will apply to the whole document.

The intended implementation will run atop of Microsoft® Window® and the .NET Framework 2.0. All references to implementation details will assume this. If implementation details for other platforms are to be discussed, it will be explicitly stated prior to the relevant statements.

The Framework is not a standalone component. Its services must be called by external code. From here on, this external code will be referred to as the *host application*. This term as does not imply what form the external code will take. Specifically, despite the use of the word "application", the host application may be not be a standalone application itself, but another reusable component.

Any reference to the object model entities such as, but not limited to, classes, interfaces and type members, will have their name in *Courier New*.

Terms that are defined in the appendix at the end of the document are shown in *italics*.

1.3

SCOPE

The product to be discussed is the 3D Graphics Rendering Framework. It will provide services to enable the rapid development of three dimensional rendering systems. This will include managing operating system level interactions and providing a high-level framework that reduces the amount of work required to cater for different hardware configurations and constraints.

The Framework will not provide the same level of abstraction that the majority of domain specific engines claim to support, as these are tailored to specific market segments. This Framework is intended to be as general as possible. No support is provided for releasing the Framework directly to the end user. It can however be used as a base to build such an engine.

1.4

DOCUMENT OUTLINE

Section 2, provides an overview of the purpose the Framework and a non-technical description of the functions which it should be capable of once complete.

Section 3, System Requirements, gives a broad description of the various requirements that the system should support and their priority during the design process. This section is split up into three subsections. The External Requirements section, 3.1, defines the list of mandatory software and hardware required for the Framework's base level services to operate. Section 3.2, System Functions, lists and describes the intended functions of the Framework when completed. The descriptions provided are of a relatively high-level nature, and outline how it is envisioned that those functions will be implemented. Operational Attributes, section 3.3, outlines general operational characteristics of the Framework, along with their priority and importance in the design goals of the Framework.

Section 4, High Level Design gives an overview of design decision in the Framework. Section 4.1 outlines the expected usage of the Framework components, both physical and logical. The organization of these components, their interactions, and dependencies on other components is outlined in section 4.2 and its associated diagrams.

The preliminary development schedule is outlined in section 5.

The Glossary in section 6 provides a listing of definitions of terms used throughout the document, a listing of issues that have come up in the design process, and all diagrams referenced elsewhere in the document.

1.5

REFERENCES

Chris Brumme (CLR architect) on finalization:
<http://blogs.msdn.com/cbrumme/archive/2004/02/20/77460.aspx>

OpenGL Specification:
<http://www.opengl.org>

2

DESCRIPTION

2.1

PRODUCT PERSPECTIVE

Since consumer level PCs became capable of rendering three dimensional graphics, it has been the preserve of C and C++ programmers. Only recently however, with the advent of relatively higher-level languages such as Java and C# has 3D rendering become available to those who don't have an intimate knowledge of pointers and low-level constructs. This is apparent in the number of wrapper libraries and engines that have recently become available for such managed languages. However these two classes of systems seem to leave some room for improvement.

Wrapper libraries simply expose the native functions of the underlying rendering API in a way that is friendly to the managed platform that is targeted at. This provides the most flexibility to the host application, but completely ignores the advantages that are provided by the managed runtime environment, such as the use of object-oriented features and more importantly, automatic memory management.

The path taken by systems that can generally regarded as engines take the opposite approach. They insulate the user from all details of the underlying API. While this provides the easiest entry into the world of 3D rendering, it binds the user to implementation specifics of the engine they choose. This can limit the options open to the user and possibly limit them when it comes to optimising the code path for their specific usage scenario.

2.2

PRODUCT FUNCTIONS

The primary goal of this Framework is to provide a class library that abstracts the underlying rendering API to a level somewhere in between that provided by both solutions described above. This should provide enough opportunity for domain specific customization of the final rendering path, and yet also provide an extra layer of insulation between the application and the graphics hardware.

The main elements that will be cover by the Framework are as follows:

- Multi-threading support
- Geometry reading, writing and storage
- Transformation stack extensions
- Multi-pass support for scenes that the underlying system hardware cannot render in one pass
- Flexible material and geometry rendering pipeline

2.3

USER CHARACTERISTICS

It is foreseen that the main user of the Framework will be those who require a rapid development platform, with the need to develop their own rendering path with specialised requirements that could not be attained through the use of third party engines. Absolute performance will not be the primary requirement of the expected user class.

Specific examples could include being used as a base for projects with a graphics element which is not the primary function, but merely to visualize some other aspect of the project. There is also the potential for using the Framework as a teaching tool. Without the need for a deep understanding of how the underlying rendering API works or low-level languages, students would get the opportunity to focus solely on elements specific to three dimensional rendering such as geometry definitions and transformations.

2.4

OPERATING ENVIRONMENT

The Framework will require Microsoft® Windows® 2000 as the base level operating system. The underlying rendering API that will be used is OpenGL. The execution runtime that will be used is the CLR provided with the Microsoft® .NET Framework 2.0.

2.5

IMPLEMENTATION CONSTRAINTS

To be the last word in rendering performance is not the goal of the Framework. However, given that performance is usually a consideration in all graphics related applications, this factor cannot be ignored. An effort must be made to ensure that the Framework is as efficient as possible when it come to fulfilling its requirements. This may, in some instances, place restrictions on how much abstraction is provided in the final version.

The newest revision of OpenGL directly supported by Microsoft® Windows® is version 1.1, and this comes with the included software renderer. Because of this, version 1.1 has to be the base level that the Framework must support. As this is a relatively old revision of OpenGL, its feature set cannot be considered sufficient for today's usage scenarios. To deal with this, the Framework must be designed to conditionally take advantage of any newer features and improved performance provided by any dedicated hardware in the machine. Where possible, the Framework should emulate any features that the underlying system does not support. However, since it is not technically possible for all of these features to be emulated, a query mechanism must be provided for use by the host application to determine what is supported by the underlying hardware.

3

SYSTEM REQUIREMENTS

3.1

EXTERNAL REQUIREMENTS

3.1.1

SOFTWARE REQUIREMENTS

The following is a list of the base level requirements of the Framework. The Framework may be capable of using features available on later platforms, but that will be determined on a case by case basis.

- Microsoft® Windows® 2000 operating system.
 - Microsoft® .NET Framework 2.0. Initial development will be carried out on Beta 1. If newer versions are released during the development period, the Framework will be adjusted accordingly to so that it will be compatible. This, however, will be dependant on available time.
 - OpenGL 1.1 API. Microsoft® Windows® 2000 comes with a software renderer which is compliant with this version, and this is the lowest common denominator that will be catered for.
-

3.1.2

HARDWARE REQUIREMENTS

There will be no mandatory hardware requirements for the Framework to run. However, if the system on which the Framework is running has a dedicated hardware acceleration unit for OpenGL, this will be taken advantage of where possible. The extent of this usage will be dependant on how the Framework is initialized, or more specifically, what device it is initialized on, and will vary based on driver support.

3.2

SYSTEM FUNCTIONS

3.2.1

THREADING SUPPORT

With a now fast-growing emphasis on parallelism in the world of computing, the Framework must have threading capabilities at its core. Since the Framework is built upon the OpenGL API, it is therefore subject to the same limitations as OpenGL in regards of multi-threading support.

OpenGL uses the concept of a *rendering context*. This represents the state container for a rendering session. To draw three dimensional primitives or modify the rendering state, a rendering context must be current to the executing thread that wants to utilise it. Because current rendering hardware does not allow for threading support on the GPU, a context may be current to only one thread at any one time. The Framework cannot eliminate this limitation, but it can provide high-level support for working around the limitation.

It is proposed that a `RenderingContext` class be used to encapsulate the native handle for the rendering context. Any manipulation of the underlying rendering context must go through public methods exposed by this class. These methods would include support for making the context current to executing thread and subsequently releasing it, along with methods to check whether the context is current and wait for the context to become available. Objects which are owned by a rendering context or explicitly require one for their operation must implement the `IRenderingContextObject` interface. This interface will have only one member, a `Context` property that will allow the host application to identify which rendering context the object belongs to. All public members exposed by objects which implement this interface, and require a current rendering context for their operation, are expected to call the `VerifyAccess` method on the associated `RenderingContext` object. This method will throw a `RenderingContextAccessException` if the context is not available on the current thread, and will ensure that the host application will not encounter any undefined behaviour in the case of a coding error.

Rapid development of fully multi-threaded applications will be provided for through an inheritable `RenderingLoop` base class. Timing and frame-rate throttling will be catered for, along with automatic adjustment of thread priority based on the timing requirements. It is expected that the majority of multi-threaded host applications will render using this facility. It should also be possible for code running on an external thread, such as a window manager, to access the rendering context. It is expected however, that the vast majority of rendering code will run under the control of the `RenderingLoop` object, and requests for access to the rendering context will be infrequent and brief in nature. As such, access patterns will be optimised for this scenario.

3.2.2

STATE ABSTRACTION

The vast majority of this state information in OpenGL is global. While this is efficient, and discourages redundant state changes, it does not however suit the design philosophy of encapsulation. To transform this model in one encouraged by object-oriented design, properties will be attached to appropriate object types. In most cases changing these properties will not cause an immediate change to the underlying state of the OpenGL context. The new state will only be applied when the object is next used during the rendering process. This will help avoid the previously mentioned redundant state changes and will open up optimization opportunities.

Not all state however, can be logically attached to objects. This remaining state can truly be regarded as being global to the whole rendering process. Examples of such state would include the viewport bounds and the transformation stacks. Therefore, a new type, `Renderer`, will be created to maintain this information. Each `Renderer` object will

represent a rendering session and will hold a reference to a unique `RenderingContext` and to a changeable rendering target. All drawing commands issued through methods provided by the `Renderer` class will be sent to the associated rendering context and will require it to be active on the calling thread.

3.2.3

RESOURCE MANAGEMENT

The resources used by the Framework can be split into three categories. The first, managed resources, are those that are created in the context of the CLR. The management of these resources is of little concern to the Framework. It is the sole responsibility of the garbage collector to clean up these resources at the most appropriate time.

The second type of resource created by the Framework is identified by types that implement the `IResource` interface. These will represent unmanaged resources that may be automatically released by the garbage collector. The `IResource` interface will extend the `IDisposable` interface provided by the .NET base class library, adding an `IsDisposed` property. It is expected that the host application will call the `Dispose` method as soon as it is finished with the resource. This will result in the most efficient use of memory. While it is bad coding practise, there is a margin for error here. If the host application fails to call the `Dispose` method, the garbage collector will be able to use the `Finalize` method to clean up the unmanaged resources before disposing of the object.

The third resource type is that which is bound to a rendering context. This class of resource implements the `IRenderingContextResource` interface, which inherits from both the `IResource` and `IRenderingContextObject` interfaces. The creation of such resources will require a current rendering context. Once created, a resource of this type will be bound to the rendering context that was active during its creation. This means it can only be used by the `Renderer` associated with that rendering context. Because of the reliance on a rendering context, these resources must be explicitly disposed by the host application. Failure to do so will result in potentially massive resource leaks within the graphics subsystem. See the issue in section 6.2.1 for more information on what action should be taken in the event of a failure to explicitly release an `IRenderingContextResource`.

The immediate usefulness of the interfaces described above may seem somewhat questionable. After all, these they do not reduce the implementation burden, each method must still be implemented separately. It is expected that in more sophisticated host applications, a generalised resource manager may be used to load and release resources in a way that most suits the system it is running on. In this scenario, the ability to identify different resource types through the use of common interfaces could be beneficial. It would be possible for the resource manager to treat resources differently based on their classification and disposal requirements.

3.2.4

SOFTWARE RENDERING PIPELINE

The primary goal of the software rendering pipeline is to reduce the overall complexity of the rendering path within the Framework. Splitting up the task of rendering primitives into distinct stages will also be beneficial with regard to the scalability and maintainability of the Framework. The pipeline however, will not be directly accessible for modification by the host application. Its structure will be determined by global rendering state, and options chosen by the host application.

Each stage in the pipeline will be responsible for a single aspect of rendering. The stages will be chained together and will execute in order. Stages can be added, removed or have their position in the pipeline changed at any time between rendering calls. No notification of such changes will be provided to the stages. The only action expected of each stage is to call the next stage in the pipeline at least once. It will be possible for each stage to call the following stage more than once. This action will result in the pipeline below the current stage being restarted and is expected to allow for multi-pass rendering techniques.

A concept diagram as to how the pipeline is expected to be structured is provided in section 6.3.3. The first stage in the chain (stage 0) is shown outside the pipeline. While not specifically part of the pipeline it is convenient to define the same semantics for its operation. It will allow for the data to be segmented intelligently, if required, by the provider before submission to the pipeline. Stage 1, lighting, would set up the supply the vertex normals and lighting positions. If lighting is disabled, this stage would simply pass control to the next stage. The material stage is itself represented as pipeline. This will allow for easier extension and modification of the material capabilities for the Framework. While all previous stages in the pipeline set up rendering state, the final stage will be responsible for the submission the geometric data to the renderer.

While the addition of the software pipeline is expected to reduce the complexity in the longer term, there are some issues. The first, and most obvious, is performance. Some loss in performance is expected, however, if the rendering calls are batched correctly it is hoped this overhead will be relatively small. The second relates to how multiple distinct data sources can be bound together and submitted to the pipeline at once. This is discussed in section 6.2.3.

3.2.5

RESOURCE VIRTUALIZATION

There are two main areas where the Framework will be able to extend the resource limits imposed by the underlying implementation. The first encompasses the OpenGL *transformation stacks*. The three primary transformations stacks are the world view, model view and texture stacks. As all modifications to these stacks can be intercepted by the Framework, a backing store that mirrors the stack contents can be built. When necessary the Framework can clear the OpenGL stacks and later repopulated without the host having any knowledge of the process. Of course, this will incur a performance penalty whenever the stacks need to be switched in or out, but it seems like a preferable course of action to outright failure. The host application should be given the facility to query the native capacity of the stacks, as it may be able to avoid this swapping.

The second area involves limitations that can be overcome with the use of multi-pass rendering techniques. It is hoped that the Framework will support two initial multi-pass algorithms. *Multi-texturing* is not supported by the minimum required version of OpenGL, but the Framework will support it through the use of its pipelined architecture. When the underlying implementation exposes more than one texture unit, the texturing stage will be able to take advantage of these to reduce the number of passes required. Performance penalties are, again, to be expected in both cases, but it will be possible for the host application to determine the number of texture units that are natively supported by the implementation.

3.3

OPERATIONAL ATTRIBUTES

3.3.1

SCALABILITY

Given the range of capabilities of graphics hardware on the market and available in already purchased PCs, scalability would seem to be the operational attribute of most concern. It is important that the Framework makes as much use of the available features and resources that are available to it at runtime. It should be designed so that it scales to take advantage of the feature set provided by the underlying OpenGL implementation. This can be done in two ways. The first will make features openly available to the host application to take advantage of. The provision of feature support queries will allow the host application to scale with the feature set exposed by the Framework. The second use of available features will be implicit. This will encompass features which can be used by the Framework to improve the performance of standard operations.

The modular design of the rendering pipeline should allow for future improvement in the number of rendering techniques supported, primarily in the use of programmable hardware units. It is conceivable that two new stages could be added to allow for the use of vertex and fragment processing programs.

3.3.2

MAINTAINABILITY

Related to scalability, a high level of maintainability will make it easier to modify and add features to the Framework at a future point. A strong element of maintainability should come with good design of both the internal and external object-model models. Separation of functions into classes and namespaces, will allow the system to be broken into distinct segments. In instances where implementation is expected to vary over time, interfaces and abstract classes will be used to access required functions, instead of forcing the use of concrete class.

3.3.3

PORTABILITY

It is not an explicit design goal for the Framework to be directly portable to other platforms. Necessary calls to the Win32 API will mean that the Framework will be restricted to running on the Microsoft® Windows® operating system, without further modification. The diagram in section 6.3.1 shows how the Framework will interact with the underlying platform. These function calls will be marshalled across to unmanaged code by the *platform invoke* layer, which is provided as an integral part of the Microsoft® .NET Framework. These functions can be generalised into two categories: *Win32* and OpenGL functions.

Given the platform independent nature of OpenGL, it is expected that most classes which rely solely on these functions will run unmodified on an alternative platform, such as Mono, which provides a layer functionally equivalent to platform invoke. Certain calls required in setting up and managing OpenGL are specific to the Windows® platform. These functions, referred to as Win32 extensions to OpenGL, will not be included in the OpenGL interop layer, despite their function, and their presence in the OpenGL library. This mainly includes context management functions, and if ported to another platform it can be expected that similar entry points would be available.

Those classes which rely on Win32 function calls would require significant modification to operate in the same manner on another platform. Relative to the number of classes in the entire Framework however, the number that relies on the Win32 API is expected to be small.

In both cases modifications to the Frameworks interop layer are expected to be necessary.

3.3.4

PERFORMANCE

The primary goal of the Framework is to provide a friendly object-model for host applications to use when graphics are required, and is not specifically designed for performance. As the Framework will be running under a managed runtime, a certain level of unavoidable overhead is to be expected. However, there are certain measures that can be taken to minimise the more apparent performance penalties.

Since the Framework will rely heavily on calls into unmanaged code, specifically, frequent calls into the OpenGL library, the overhead introduced by these must be minimised. Most types marshalled between managed and unmanaged code are *blittable* types and therefore cannot be optimised any further. Because calls into unmanaged code cannot be verified, the caller must have the `UnmanagedCode` permission. Every time platform invoke initiates an unmanaged function call, a full stack walk ensures that caller has this permission. This stack walk is very costly in terms of performance. Therefore, all OpenGL interop methods will be tagged with `SuppressUnmanagedCodeSecurityAttribute` to prevent these security checks.

Where possible, internal collections should use the generics support provided by the Microsoft® .NET 2.0 runtime. This will reduce the number of typecasts required.

Given that the Framework will only be dealing with rendering graphics, security is not seen as an area for major concern. However, because of its reliance on calls into unmanaged code, it will require a relative high level of trust to execute this unverifiable code. To partially alleviate this, the framework itself will require the host application to have been granted `GraphicsPermission`, a built in permission provided by the Framework.

As mentioned in the previous section, calls to OpenGL functions will be marked with `SuppressUnmanagedCodeSecurityAttribute`. This does not eliminate all security checks on calls into unmanaged code. A link time check will be performed by the compiler to ensure that the immediate caller has the correct permissions. This rule will not extend to functions defined by the Win32 extensions to OpenGL. These will be treated on a case by case basis. Rendering context creation and destruction functions will still retain full security checks as they involve the creation and destruction of system resources. Frequently called query functions may be allowed to bypass the stack walks.

All Win32 function calls will retain full security checks. This should not prove to be a barrier to performance, as these functions will be called infrequently and will not be part of time critical code.

4

HIGH-LEVEL DESIGN

4.1

DESIGN OVERVIEW

4.1.1

INITIALISATION

To begin a rendering session two main components must exist. The first is a *Surface*. This represents the drawing surface which is the target of all rendering commands. Because of the drawing model used by the Windows® GDI, and consequently required by the Windows® OpenGL implementation, these drawing surfaces must be backed by a *device context*. Device contexts can be acquired from any element that the Windows® GDI can draw to, but the Framework will only directly support windows and in-memory bitmaps. It should be possible for the host application or another piece of code to extend this support.

Once a surface has been acquired, it will be necessary for the host application to set the pixel format to be used when rendering to the surface. The pixel format will define the format of the data stored in the frame buffer, including the number of colour planes and their format, the presence and type of depth and stencil buffers, and double buffering information. The available pixel formats will vary depending on the implementation and drivers being used. It is expected that the host application will always show preference for the hardware accelerated formats, and information as such will be made available through the Framework types. It should be noted that once the host application has set the pixel format for a surface, it cannot be changed. This is a limitation imposed by Windows®. To work around this, the host application must be able to create a new surface of the desired format and set it as the rendering target. To this end, the changing of rendering targets must be supported.

4.1.2

RENDERING

When a surface has been acquired and its pixel format has been set appropriately, the host application will be able to create a *Renderer*. This class will provide all the rendering services to the host application and will own a *RenderingContext*. Any access to the *Renderer* object will require its associated *RenderingContext* to be active on the current thread. Failure to satisfy this condition will result in an exception being thrown, but no change to the state of the *Renderer* will occur.

OpenGL provides what is essentially an immediate mode rendering architecture. It is an explicit design goal of the Framework to make as much of the flexibility provided by OpenGL available to the host application. However, the retained mode paradigm brings with a less severe learning curve that would seem to fit in well with the concept of a high level framework. As such, the Framework will be written primarily as an immediate mode system, but will contain elements which will provide some of the convenience of retained mode. These elements will be completely optional for the host application and the extent to which retained mode rendering will be employed will be minimal.

The Framework will provide support for reading and writing geometry to and from streams. In-memory storage will be provided by buffers of different types. The Framework will support classes and interfaces for the rendering of three primitive types: points, lines and triangles. All other geometry must be composed of these three primitive types. When the host application wants to render geometry, it must first be loaded into one or more of these classes. Once prepared, they can be passed to the *Renderer* from where the geometry will be sent through the rendering pipeline.

Issues relating to the partitioning of geometric data are discussed in sections 6.6.2 and 6.2.3.

4.2

NAMESPACE ORGANISATION

To organise the types within the Framework, they will be grouped by function into relevant namespaces. The diagram in section 6.3.2 provides an overview of how each of the namespaces relates to others in the Framework. Below is a list of all the namespaces currently planned and a short description of the contents of each.

- *Graphics* – This is the root namespace for all interaction with the graphics subsystem. It will contain types relating to surfaces, rendering sessions and other core aspects of the rendering system.
- *Graphics.Geometry* – Types used for the storage, manipulation, and rendering of geometry will be contained within this namespace.
- *Graphics.Interop* – Internal access only. This namespace will be subdivided in two sections, *Graphics.Interop.Win32* and *Graphics.Interop.OpenGL*, containing code (external function definitions, enumerations and structures) for interop with Windows® and OpenGL respectively. Helper types with no direct affinity to either category will be located in the root *Graphics.Interop* namespace. Types used for the loading and management of OpenGL extensions will be located in the *Graphics.Interop.OpenGL.Extensions* namespace.
- *Graphics.Materials* – Will contain classes used for the application of materials to rendered geometry.

- `Graphics.Pipelining` – Internal access only. All types related to the creation and management of the rendering pipeline will be located here. Types relating to each pipeline stage will also be contained here, possibly in their own sub-namespaces.
- `Graphics.Security.Permissions` – This namespace will contain the types that are used for implementing the permissions that will be required by host applications intending to use the Framework.
- `Graphics.Threading` – All types relating to multi-threaded rendering will be in this namespace. This includes context management, the `RenderingLoop` base class and associated types, along with exceptions specific to threading errors within the graphics subsystem.

Extra namespaces may be provided for features that are added during the development cycle.

5

SCHEDULE

A preliminary diagram for the outlay of the development schedule is provided in section 6.3.4. Being a preliminary schedule, it is expected that this will change as the development process proceeds, to take account of unforeseen issues. Two weeks are provided at the end for writing the product documentation and demonstration code. Some of this time may be reallocated towards development if any serious issues are discovered.

- **Code Review** – As some components were written before the development of this document, this process will involve tracing through the code to ensure that it operates to the guidelines laid out in the above sections.
- **Threading Support** – This stage of the development involves building all the base support for OpenGL contexts and managing them across threads. Support for a contained rendering loop will also be added later in this phase as it will need to interact with the completed context management code.
- **Global Rendering State** – This phase encompasses the development of a `Renderer` class to manage the global OpenGL state.
- **Pipelining Infrastructure** – Development of the initial support infrastructure begins in this phase. It is expected that during the overlap with the Geometry, and Materials and Lighting phases, adjustments will be made in parallel to cater for unforeseen issues.
- **Geometry** – The coding of the geometry phase class is carried out in this stage. Once these are complete, work on the geometry submission phase of the pipeline can begin and should be completed around the time the Materials and Lighting phase begins. This will allow for the initial interactions between the two pipeline stages to be tested.
- **Materials and Lighting** – The material base classes will be developed at this point, followed by the Material pipeline stage. Lighting will be added later, although the amount of support in the final code base will be time dependant. This phase of development will be the most experimental and hence the long timeframe allowed. The final development of these pipeline stages may overlap with the Documentation phase.
- **Documentation and Demonstration** – In this, the final phase, all technical documentation, including the user manual and technical specification will be written. Demonstration code will also be written.

6

APPENDIX

6.1

GLOSSARY

Blittable	Refers to types which can be used across the interop layer without any modification. Their representations in managed and unmanaged code are identical.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime. This is the execution environment of all managed code. It is responsible for compilation, security checks and execution of the IL code that makes up a managed assembly/program.
Device Context	State container for drawing functions in the Windows® GDI.
GDI	Graphical Device Interface. The drawing component of the Win32 architecture.
IL	Intermediate Language. This is the platform agnostic code that all .NET languages compile to. At runtime it is compiled to the native machine code of the underlying system.
Interop	General term used to refer to the communication between managed code and another form of code such as native code.
Rendering Context	State container for an OpenGL rendering session.
Win32	Term used to refer to the user mode portions of the Microsoft® Windows® API that programs call to execute system provided services.

6.2

ISSUES

6.2.1

DISPOSAL OF RENDERING CONTEXT RESOURCES

Issues

Because such a resource is tied to a specific rendering context, that context must be current on the executing thread when the resource is manipulated. This includes releasing the resource. The CLR uses a separate thread for the execution of object finalizers. Therefore, to have a finalization method dispose of a rendering context resource, it must be possible to make the context current on the finalization thread. This however cannot be guaranteed with any certainty.

Since there is no way to ensure the context has been freed from all application threads prior to the beginning of the collection, attempting to make it current may result in deadlocking the finalization thread and, consequently, the application. The current Microsoft® implementation of the CLR uses a single thread for garbage collection. There however no restriction in the CLR specification from having multiple threads perform the collection in future releases. That situation would then result in contention issues between two or more finalization threads that try to dispose of a rendering context resource at the same time.

The garbage collector disposes of objects in a non-deterministic order. Therefore there is no guarantee that the `RenderingContext` object hasn't already been disposed and therefore the context that it encapsulates. This is not such a problem, because, in all likelihood, the driver would have freed the memory at the same point that the context was destroyed. It does however mean that you need to access an object other than the one being disposed which violates the finalization rules.

Possible Solutions

1. Take a neutral approach and place the burden of explicit disposal on the user of the Framework. This however seems to go against the ideology of a framework such as this. The goal is to aid the developer, not place more burdens in their way.
2. Access the `RenderingContext` object, check whether it is still valid, and if so try to make it current to the finalization thread. If successful, release the resource. What happens if making the context current is not successful? This resolution could also raise contention issues in a multi-threaded finalization environment. This would seem to leave a lot of uncertainty in its wake. This is not needed in a system which already has to deal with complex interactions.
3. If a resource which is bound to a rendering context makes it through to the finalization phase without having previously being disposed, throw an exception. This would be a clear indication to the developer

that they have a potential resource leak. How clear this would be is another issue. The specification for the Microsoft® CLR implementation states that any exceptions thrown in a finalizer will be treated as a method return. It is not clear whether the exception will ever be visible. Initial testing however, on version 1.1 of the .NET Framework, shows that exceptions in finalizers are caught when running under a debugger. Further research regarding behaviour in version 2.0 is required.

4. Version 2.0 of the .NET Framework will implement Managed Debugging Assistants. These can be used to notify the debugger of failures in areas such as finalization where exceptions are not appropriate. This would be an ideal solution, however, it does not appear that it will be usable by external components until a future release.

Status

Unresolved. Since it appears that solution 4 will not be possible under the current release, solution 3 would seem preferable if the desired behaviour exists in version 2.0 of the .NET Framework.

6.2.2 DECOUPLING VERTEX COORDINATES, TEXTURE COORDINATES, COLOURS AND NORMALS

Issues

In OpenGL a vertex can be assigned a location, colour, normal and texture coordinates. However, not all of these may be required in every scene that is to be rendered. For example, if a scene is to be drawn without lighting, then there is no need to specify the normal data.

What is in question here is the level to which these should be separated in the Framework's object model. In the object model, vertex coordinates are to be stored as geometry, while colours and textures are to be abstracted to be represented by different forms of materials. Using this model, how can separate colours be specified for each vertex without storing the colours in parallel with the vertex coordinates? A similar issue occurs when it comes to storing texture coordinates that are to be used when rendering geometry using a texture based material.

Possible Solutions

1. Use a single *Vertex* structure with variables to cater for all possible values. This solution does not attempt to decouple the possible data.
2. Separate. No in-memory structure contains more than one type of data. The different forms of data are recombined within the rendering pipeline (see issue 6.2.3). This would not take advantage of the interleaved arrays that OpenGL supports.

Other possible solutions may exist, but have not been considered yet.

Status

Unresolved.

6.2.3 FORMAT OF DATA TO BE SUBMITTED TO THE RENDERING PIPELINE

Issues

Issue 6.2.2 dealt with how to decouple the different forms of data that can be attached to geometry for storage. This issue deals with the inverse: how to allow that data come together within the rendering pipeline so the geometry can be rendered.

Possible Solutions

1. Provider interfaces. When data is submitted to the pipeline, it will be in the form of a type which implements at least one provider. Each stage in the pipeline would cast the object to the type of the provider they require. These two could then communicate in some undisclosed way to retrieve the necessary data. What happens if a required provider is not implemented? What happens if two providers supply different amounts of data? e.g. the vertex provider supplies 6 vertices, but the texture coordinate provider only supplies 5 texture coordinates.

Other possible solutions may exist, but have not been considered yet.

Status

Unresolved. It is hoped that during the development phase, a satisfactory solution will become evident.

6.2.4 IMPLEMENTING RESOURCES BOUND TO A RENDERING CONTEXT

Issue

Should resources that are bound to a rendering context implement a common interface or inherit from an abstract base class.

Possible Solutions

1. No special implementation. Each resource type simply implements both *IResource* and *IRenderingContextObject*. Code for managing resource disposal is replicated across classes.

2. Implement a common interface. This would allow the maximum flexibility in how resources could deal with the situation, but there is some question as to how much flexibility is actually needed. The usefulness of an interface which inherits from two other interfaces but does not add any members of its own is questionable.
3. Inherit from an abstract base class. This route would allow a single set of base code to be used by all resources to determine whether a resource was disposed of correctly by the host application. Because the *CLI* only allows single inheritance, this could restrict inheritance hierarchies. e.g. Two related classes, only one of which needs to be regarded as a resource.

Status

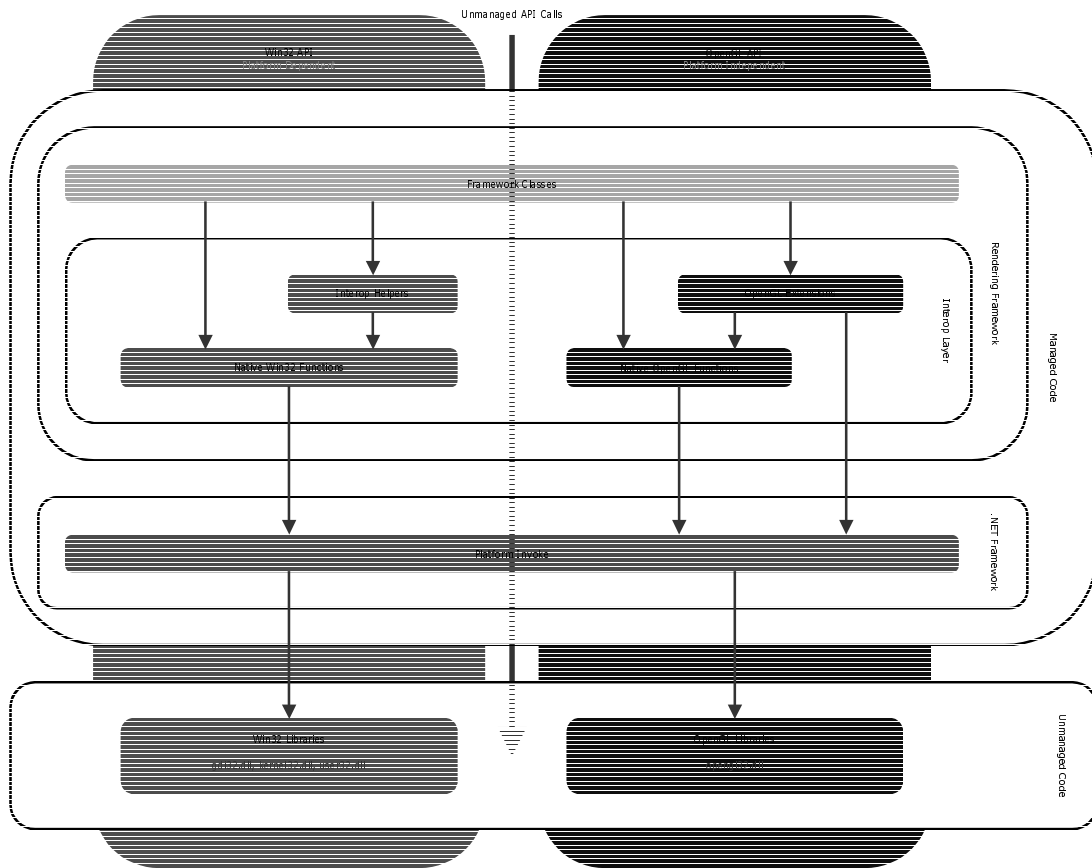
Unresolved. Currently, solution 3 is preferred. This allows the host application developer to see a rich object-model and leave the Framework to deal with resource issues (the way it should be). However, as the code base is developed, this decision will be re-evaluated based on experience.

6.3

DIAGRAMS

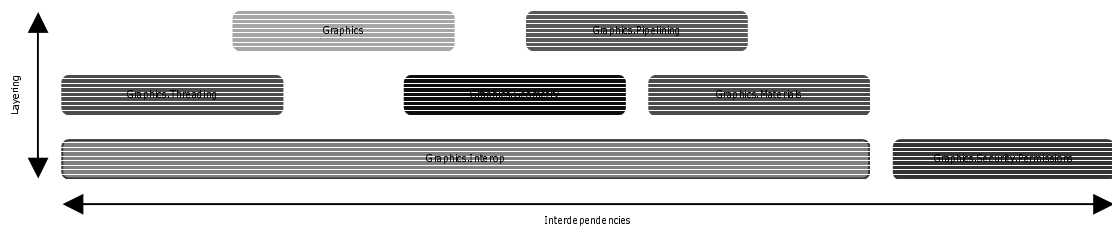
6.3.1

INTEROP LAYERS



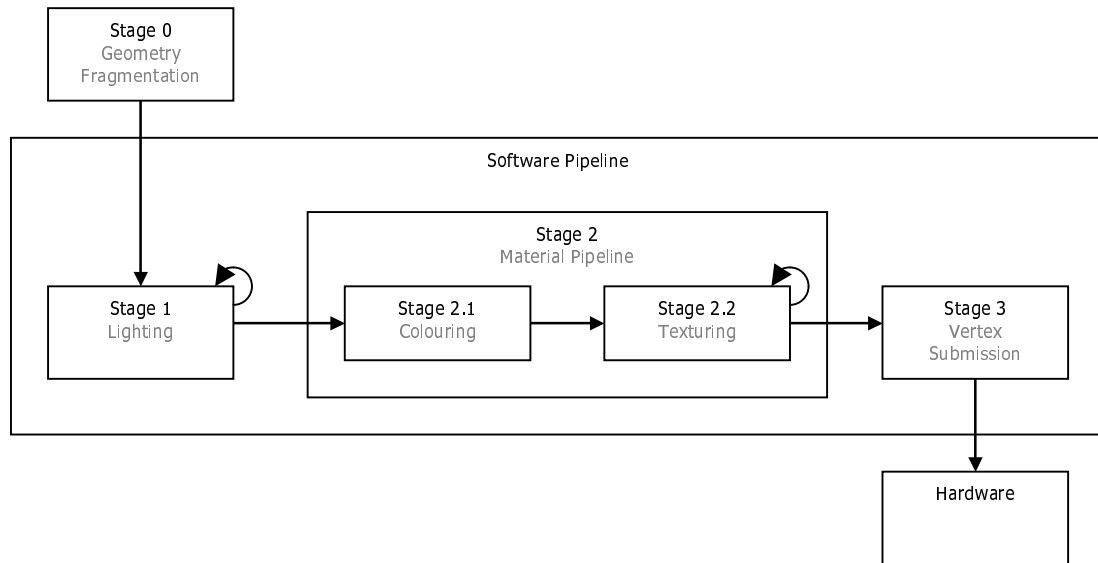
6.3.2

NAMESPACE INTERDEPENDENCIES



6.3.3

PIPELINE STAGES AND INTERACTIONS



6.3.4

PRELIMINARY SCHEDULE

