GDK Coding Conventions and Patterns

Table of Contents

[Language specification, compiler version, features, and extensions 2](#_Toc332827969)

[Standard types and libraries 2](#_Toc332827970)

[Math types 2](#_Toc332827971)

[Headers 2](#_Toc332827972)

[DLLs 2](#_Toc332827973)

[Target platforms 3](#_Toc332827974)

[Debug notifications, status reporting 3](#_Toc332827975)

[Memory management 3](#_Toc332827976)

[Error handling 3](#_Toc332827977)

[Comments 3](#_Toc332827978)

[Strings 4](#_Toc332827979)

[Parameters 4](#_Toc332827980)

[SAL and code analysis 4](#_Toc332827981)

[Using statements 4](#_Toc332827982)

[Warning level and warnings as errors 4](#_Toc332827983)

[Unit tests 5](#_Toc332827984)

[Class definitions 5](#_Toc332827985)

[Naming conventions 5](#_Toc332827986)

# Language specification, compiler version, features, and extensions

* C++ 11
* No vendor specific extensions

# Standard types and libraries

* Use stdint.h numeric types (ex: int*X*\_t) and min/max defines (ex: INT*X*\_MIN)
* Prefer STL for well-known containers and algorithms (ex: vector, sorting)

# Math types

* GDK math types should be used throughout for consistency

# Headers

* Avoid #include of dependencies in headers. Leave it up to the caller to include the right things
* Use #ifndef guard, since pragma once isn’t supported by other compilers
  + #ifndef \_MYHEADER\_H\_  
    #define \_MYHEADER\_H\_  
    …  
    #endif // \_MYHEADER\_H\_
* Platform.h should contain only platform level abstractions, it is not a dumping ground for common stuff

# DLLs

* All reusable subsystems should be separate modules (DLLs on Windows). Ex:
  + GDK.Win32.dll
  + GDK.Metro.dll
  + GDK.DxRenderer.Win32.dll
* Exported APIs across module boundaries should consist only of C functions and/or interfaces
* Exported APIs should not contain STL types in the signature (see common marshaling patterns section of this guide for how to convert)
* Exported APIs should always report status using the ErrorCode enum return value
* Exceptions should never be thrown across the module boundary
* An allocation from one module should never be freed by another module. Nano-COM based interfaces handle this for objects. See memory manager section of this guide for how to handle non-interfaces

# Target platforms

* Current target platforms are:
  + Win32 – Windows classic API, usable on Win7 and Win8 classic. (AMD64 only)
  + Metro – Windows Metro API, usable on Win8 metro only. (AMD64 and ARM)
  + Durango (AMD64)
* Under each module’s directory, the following directory structure should be used:
  + Common – common header and source files used by all versions of the module
  + <Platform1> - project, header, and source files used exclusively by Platform1
  + <Platform2> - project, header, and source files used exclusively by Platform2

# Debug notifications, status reporting

* We will have a standard debug notification interface that objects can implement to provide rich feedback to interested callers. Examples of what might be on the interface:
  + RegisterCallback(pfn, &token)
  + UnregisterCallback(token)
  + EnableNotifications(bool)
  + SetVerbosity(enum)

# Memory management

* Use RAII + smart types & pointers
* The use of new and delete should be avoided in most cases
* The common base template class can override new/delete for use with memory managers
* Class constructors/destructors should be private or protected, preventing outside new/delete
* There will be a custom allocator/memory manager pattern for use in modules that require external clients to free memory. TBD

# Error handling

* Use exceptions within modules
* GDK::Exception will be the general exception object, and will contain details about the failure
* ErrorCode enum should be used for module boundary APIs
* Common macros will be created for folding common error types into GDK::Exception. Ex:
  + CHECKHR() will map HRESULT -> exception
  + CHECKGL() will map glEnum -> exception

# Cross-platform SAL, COM & WRL

* Platform.h will ensure that IUnknown, GUIDs, WRL::ComPtr, and SAL work on all platforms
* We are using nano-COM, which uses IUnknown contract, but nothing more. No marshaling, etc…
* Use uuidof<> instead of \_\_uuidof(), since the latter is a compiler-specific intrinsic which relies on midl metadata.

# Comments

* C++ style comments (//) should be preferred over C style comments (/\* \*/) due to nesting issues with C style. VS has buttons to quickly comment/uncomment using C++ style, so convenience isn’t an issue.
* Each class or struct should have a comment block above it describing the functionality. In many cases, this is sufficient and per-method comments might not be necessary.
* Comment methods as needed to explain nontrivial methods
* Examples:
  + // RuntimeObject  
    // Base class which implements default ref counted behavior.  
    template <typename I0>  
    class RuntimeObject
  + // Creates a suitable runtime object from the given data.  
    void RendererCore::CreateRuntimeTexture(\_In\_ ITextureResource\* resource);

# Strings

* Unicode (wide) strings should be used throughout
* STL string classes should be used whenever possible
* Any functionality added to this should be done via GDK helper methods that act on STL strings

# Parameters

* For input parameters, prefer const byref for anything larger than an int
* For input/output parameters, byref should be used
* For output parameters, pointers should be used to make it clear at the call site

# SAL and code analysis

* SAL 2.0 should be used throughout
* Since SAL only exists in MSVC, Platform.h will define empty definitions so that the code compiles properly on other platforms
* Code analysis will be enabled and run on every build

# Using statements

* Using statements should never be used in a header or inline (.inl) file
* Prefer only applying the using to fully qualified types and not the entire namespace. Ex:
  + using std::vector; (instead of using std;)
  + using GDK::Matrix; (instead of using GDK;)
* Namespace aliasing may also be used. Ex:
  + namespace Gdkg = GDK::Graphics;

# Warning level and warnings as errors

* Warning level 4 should always be used
* Warnings as errors should always be enabled
* Any internal code that causes a warning should always be fixed
* If there is an external file (SDK header, etc…) that causes a warning, the warning may be disabled via a pragma for that one #include only, then reenabled. This pragma should never appear in a public header, only in precompiled headers within a project.

# Unit tests

* Currently investigating C++ unit test projects in VS 2012, seems like a likely choice
* Built for target platforms just like the engine
* Single unit test project for the engine and tool chain
* Should be run (and all passing) before checking in nontrivial changes

# Class definitions

* All ref counted objects should derive from the ref counted base template
* This ensures that all of them hook into the memory managers properly
* It also helps us enforce certain patterns across the board
* Sample class pattern for a ref counted object:

class DxRendererCore : public RuntimeObject<IRendererCore>

{

public:

DxRendererCore\* Create(\_In\_ HWND hwnd);

private:

DxRendererCore();

~DxRendererCore();

Microsot::WRL::ComPtr<ID3D11Device> \_device;

};

# Naming conventions

* Hungarian notation should be avoided

|  |  |  |
| --- | --- | --- |
| Scenario | Convention | Example |
| Filenames | PascalCase | DxRendererCore.h |
| Defines and Macros | Constants and inline methods should be preferred, but otherwise ALL\_CAPS | GAME\_TITLE |
| Global scope variables (constants, globals, file statics) | PascalCase | ScreenWidth, WindowHandle |
| Function names | PascalCase | LoadShader() |
| Local scope variables (parameters, local variables) | camelCase | hitPoints |
| Namespaces | PascalCase, avoid exceeding more than 3 layers max when possible | GDK::Tools |
| Class and structure names | PascalCase | GameTime |
| Interface names | I-prefixed PascalCase | IRenderer |
| Private member fields | \_-prefixed camelCase | \_mainWindow |
| Public/protected member fields | Should be avoided in classes, but okay for plain-old-data (POD) structs. camelCase | textureHeader.surfaceFormat |
| Template parameters | PascalCase | class vector<typename T, int Capacity> |
|  |  |  |
|  |  |  |
|  |  |  |