Game Development Kit (GDK)

Architecture, designs, and patterns

Contents

[High level architecture overview 1](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783216)

[Component architecture overview 1](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783217)

[Build configuration 2](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783218)

[Code design 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783219)

[Types 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783220)

[Libraries 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783221)

[Class design 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783222)

[Object design 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783223)

[Allocation and lifetime 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783224)

[Extensibility 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783225)

[Misc. style guide and patterns 3](file:///C:\GameDevelopmentKit\Documents\Game%20Development%20Kit.docx#_Toc340783226)

# High level architecture overview

## Component architecture overview

The top level components in the GDK are the engine runtime, the game-specific logic, the host, and the editor (Content Studio). For the initial release of the GDK, Content Studio will play the role of the Windows host. On OSX, there will be a standalone host executable to run the game.

The dependencies between layers must adhere to a strict set of rules:

1. The engine runtime is self-sufficient. It should not have any dependencies outside of its own codebase.
2. The game-specific layer may consume anything publicly available in the engine runtime, but may not take dependencies on the host or tool layers.
3. The host/tool layer may consume anything publicly available in the engine runtime. For the initial release, the host/tool may also consume exports from the game-specific layer. This will change in future releases as we make improvements to the design around the game-specific layer.

Within the engine runtime, we have the following high level conceptual component breakdown:

Content – The source for retrieving all content at runtime. We should evaluate if this is also the place to store content (screen shots, user-created maps, save games, etc…) or if we should introduce a StorageDevice for that.

Devices – Each external system we need to communicate with (graphics, input, audio, network, storage, etc…) is modeled as a device. This device is a platform and implementation-independent object that provides access to the game to work with that system.

Game – The game object is the instance of the game itself. We can have multiple instances of the game running if we wanted to, each one a unique version. The game is provided with the devices it should use, and can be used to drive the simulation.

World – A game world is a collection of objects which may interact with each other, combined with a camera to view the world with. There are no restrictions on the number of worlds that a game instance can have.

GameObject – game objects are the individual objects that populate worlds. These provide the base functionality to be an interactive entity in the world, but games extend and customize these objects to provide unique experiences.

## Build configuration

For the initial release of the GDK, Content Studio will play the role of the Windows host. To that end, we’re just going to statically link in the Quake 2 game-specific layer, and the engine runtime layers. On OSX, the Quake 2 game layer and the engine layer will get statically linked into the OSX host. The folder and project layout is quite simple (examples below, other components follow the same pattern):

**Engine\**

**GDK.sln**

**Public\** All publicly (outside of the engine runtime) headers go in this folder.

**Internal\** All shared (within the engine) headers go here.

**Graphics\** Common graphics code can go here

**DirectX11\** DirectX specific code here

**OpenGL\** OpenGL specific code here, for all platforms (differ at file level)

**Games\**

**Quake2\** Quake2 specific content and code lives under here

**Hosts\**

**OSX\** For v1.0, we only have an OSX host

**Tools\**

**Content Studio\**

**PakExtract\**

**Quake2Pak\**

# Code design

## Types

In the core Platform.h header, there should be absolutely no platform specific headers included (no Windows.h, no mac headers, etc… Maybe we should rename this file?

The purpose of this file is to establish the common set of types and constructs that nearly all files and components require. This includes including standard int types (int32\_t, uint16\_t, etc…), our own typedefs (byte\_t, etc…), common STL headers, common macros (UNREFERENCED\_PARAMETER), and include core base classes and debug facilities (exceptions, RuntimeObject, etc…).

File processing code should NEVER use architecture-dependent types such as size\_t. Files should be well defined to exactly how many bytes each piece of data requires, and types should reflect that.

File processing structures used in read/write operations should ALWAYS be created with 1 byte packing (pragma pack 1).

Enums should ALWAYS use the new C++11 enum class declaration.

Constants should always be defined as actual constants, and not as #defines.

Macros should be avoided when possible, using inline functions where appropriate. Macros are allowed however, if they are really the best way to solve a particular problem.

## Libraries

The engine code base should be as dependency light as possible. We should evaluate each and every dependency that we are considering adding with a lot of scrutiny, and ensure we’re all onboard. The obvious dependencies (and they should be scoped to only the components that require them) are DirectX, OpenGL, and OpenAL. However, this should be limited to only those implementations that use them, and those components should be written in a way that can be conditionally compiled out (or stripped by the linker if not referenced). This means includes of library headers, etc… should all be contained to very specific places.

We should strive to use the C++11 language features, CRT, and STL library as much as possible to avoid having to pull in external code.

## Class design

All heap based, engine components and classes should derive from RuntimeObject. This ensures a common set of functionality and traits are available on all engine types. Value types such as math classes and other POD structures are exempt from this.

When designing and adding a class, we must be very explicit about whether the object is designed to be derived from. If not, make the constructor private (see allocation and lifetime section below on how you’d obtain an instance). If the object can/should be derived from, mark the constructor as protected, and make the destructor virtual. NEVER make any constructors public! We need to ensure we can control allocation in a uniform way.

Be very careful about whether or not a class needs to be exposed publicly. If it does, put the header for the base type in the Public folder. If not, but it needs to be shared within the engine, put it in the Internal folder. If it’s local to only a component, then keep the header in the component folder. Be very careful about what dependencies leak through the header. If there are a lot of members, particularly ones that require bringing in more headers, consider making the class an abstract base, and then deriving a more private implementation internally that has all the real members. The original, public class is like an interface++ since it would contain more than just the virtual methods. It could contain static factory methods and some simple members.

## Object design

## Allocation and lifetime

To ensure we can control and track memory properly in the system, we need strict control over the allocation of objects. We achieve this in a couple of ways:

No object should ever contain a public constructor. This prevents anyone from new’ing up the object directly.

// REVIEW: We need to examine this closer and make a call

Option A: All objects should expose a proper factory method which returns a std::shared\_ptr<> to the object. Protected class-level new/delete operators will be defined on the base RuntimeObject class, which the factory methods would use to create the objects. These operators will allow us to hook into whatever memory tracking or management that we wish to implement.

Option B: Certain objects will still require factory methods, to disambiguate implementation choice (example, GraphicsDevice::Create(DeviceType type) to create Dx vs OpenGL). However, for the rest of the objects, we expose a common template function Make<>, or similar, that has private access to the objects and can ensure the correct allocators are used. This saves on the hassle of having to create 1 liner factories for every little class.

## Extensibility

# Misc. style guide and patterns