Chapter 58
Modular Game Engine Design

Aaron Boudreaux
University of Louisiana at Lafayette, USA

Brandon Primeaux
University of Louisiana at Lafayette, USA

ABSTRACT

The usage of software engineering principles in designing a game engine is discussed in this chapter using a simple tower defense game implemented using C# and the XNA Framework to illustrate usage of the engine. Essential functions, such as collision detection, input/output, graphics, object management, state management, and sound, will be implemented as independent units called managers. Because each manager is independent from the rest, essential development tasks such as implementing each manager and isolating bugs are much simpler.

INTRODUCTION: WHAT IS A GAME ENGINE?

A game engine is a system designed to facilitate rapid development of video games. A game engine generally acts as a layer of abstraction between hardware or low level code and the engine interface to reduce the amount of time spent designing or coding a game. Additionally, game engines may utilize third party libraries to provide desired functionality, such as physics, sound, and artificial intelligence, rather than having to "reinvent the wheel". There exist many libraries on each of these topics which can both quickly add new features to a game engine and reduce the overall time required for engine development.

"Game Engine" is a generic term and does not define any specific features which must be included other than aiding in the game development process. However, many game engines include, but are not limited to, any number of the following standard features:

- 2D or 3D rendering
- Audio
- Physics
- Input Detection
- Scripting

DOI: 10.4018/978-1-4666-4301-7.ch058

- Entity Management
- Networking
- Modeling and Animation
- Graphical User Interface Management
- Particle Systems

In addition to these core features, it is also desirable to develop the engine to provide these features using a component based architecture. Ideally, each of the previously listed features is implemented as an individual, independent component. Realistically, some inter-component dependencies may be required in order for some libraries to function correctly, or to prevent duplication of work. For example, the GUI system requires the input system in order to detect GUI interactions. While a GUI library may implement its own basic input detection, it would be completely unnecessary to use both the basic GUI library input detection, as well as a more fully-featured input library which may support many other input libraries.

Component based systems tend to be more modular. This allows engine developers to either swap out one implementation of a module or component for another, or provide multiple component implementations, allowing a game developer to have a choice of implementation. Additionally, a game engine may allow a game developer to either add new components or replace the default components with their own custom components. A great example would be a component based 3D rendering system which allows the developer to choose a target platform such as the Microsoft Xbox 360, Sony Playstation 3, or Nintendo Wii. It is even possible to compile the game using each of these systems to allow game developers to focus on multiple platforms.

This chapter illustrates the concept of modular design through actual design and implementation of a simple game engine. It features many of the components listed above, including 2D rendering, audio, input detection, and graphical user interface management.

BACKGROUND

This section will cover the basics of the XNA framework and C# before delving into our engine.

C# Keywords and Other Functionality

The game engine uses several keywords and other features of C# that may be unfamiliar to those without experience in the language. A brief overview will be provided here. For more information, please refer to the C# books listed in the additional reading section at the end of the chapter.

- **Override:** Used to extend or change an abstract or virtual member of a base class. The override keyword must be used in the declaration of the derived class or member (Override, 2011).
- **Sealed:** The sealed keyword can be applied to class members, methods, fields, properties, or events that override a virtual member of the bass class. Other classes or members cannot override a sealed object (Abstract and Sealed Classes and Class Members, 2011).
- **Internal:** A method or class member that is declared to be internal is only accessible to other classes within the same namespace (internal, 2011).
- **Delegate:** The delegate keyword functions similarly to a function pointer in C or C++. It is used to specify the signature of the events which will be thrown by events specified using this delegate. Methods registered for the event must also match the signature of this delegate (i.e., a void method with a single parameter of type string) (Delegates Tutorial, 2011).
- **Event:** The event keyword is used to specify an object which can be used to store a list of specific methods to be called. An event must be declared using a delegate type.