# Program a Game Engine from Scratch

Mark Claypool

## Chapter 1 - Introduction

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 9.0). Information online at: `http://dragonfly.wpi.edu/book/`

# Chapter 1

# Introduction

## 1.1 Goals

In working through the entire book, an aspiring game programmer will gain an in-depth understanding of a game engine. Such a programmer will not only know how a game engine is implemented, but also *why* it is implemented the way it is, understanding choices required to achieve general purpose functionality to support a variety of games. Moreover, an aspiring game programmer will understand game programming from the game programmer's point of view, being able to differentiate functionality in game code versus functionality provided by the game engine. This understanding will be reinforced by making a game, albeit a simple one, using a full-featured, fully functional game engine. Lastly, and for some perhaps most importantly, an aspiring game programmer will have created a substantial body of code potentially suitable for a portfolio, a showable record of what s/he can *do*. The built-from-scratch engine itself can be shown, with one or more games demonstrating its functionality, with the potential for an in-depth conversation (say, with a potential employer) about how, exactly, the engine and game(s) are implemented. All of this achievable by *programming a game engine from scratch*.

More specifically, the goals of this book are to provide an understanding of:

1. a game engine from the game programmer's perspective;
2. the structure and design of a game engine;
3. the trade-offs between complexity, fidelity, and interactivity in game engines; and
4. software engineering techniques that can be applied to creating parts of a game engine.

In order to accomplish these goals, this book:

1. Gives detailed instructions on how to implement a game using Dragonfly.
2. Provides an overview of the Dragonfly architecture.
3. Provides the design of Dragonfly, in the form of header files with design rationale, presented incrementally in order of implementation.
4. Details step-by-step how to fully implement Dragonfly following the design.

## 1.2 Game Engine Overview

There are many experienced programmers that have played hours (and hours) of computer games. These same programmers have used many game engines as players and may have even heard the names of some game engines. Despite this, most game-playing programmers probably *still* have questions as to what, exactly is a game engine and how it works. In order to start describing what a game engine is, it is useful to first describe game engine functionality from a user's perspective. In particular, the user of a game engine – the *game programmer* – is the programmer that is building the game – in contrast to the computer that actually runs the code that makes the game happen.

First, consider a game from the game programmer's perspective. The game programmer is making a game with a goal (or a set of goals) for the player to achieve. For example, the game goal may be to save the princess (with a sub-goal of finding the enchanted sword). Rules govern the gameplay that moves the player closer (or further) from the goal. For example, rules govern when a player can eat a ghost in a Pac-Man-type game, or how a ball collides with blocks in a pong-type game. For the player, a game includes visual and audible content, such as pretty graphics and catchy sound effects. The game programmer needs to provide control techniques, such as the mapping of buttons and mouse clicks to game actions, in order for the player to reach the goal.

Now, consider a game from the computer's perspective (also, the perspective of the *game engine programmer*). From the perspective of the computer, a game is set of resources managed to support an entertainment application. The resources that need managing are: 1) the display graphics, such as the models and animations that need to be drawn to the screen; 2) the sound device, such as playing an sound track to the speakers; 3) the user interface, such as the keystrokes and mouse movements that need to be captured; 4) scripts that need to be invoked at certain times; 5) events that need to be processed, such as for collisions and timers; and 6) file input/output for reading and recording game data. Additional resources that need to be managed by many games are networking, artificial intelligence, and physics.

The line between game code and game engine code is often blurry. For example, for one game engine, the engine may know how to "draw an ogre". For this engine, the game programmer would issue a command to the game engine having it draw an ogre at a certain location and the game engine would handle the ogre drawing automatically. However, for another game engine, the engine may only provide features for rendering and shading. The game programmer would have to code "ogre-ness" drawing aspects entirely in the game code. It may be tempting to assume that the ogre-drawing engine is better than the one that does not draw ogres – after all, it does more of the game programmer's work – but the ogre-drawing engine may end up being less flexible than the non-ogre-drawing engine, only able to make ogre-type games.

In general, there is no definitive separation of game code and game engine code since many "built-in" components of a game engine could be done in game code. For example, animating a sprite could be done in game code, with the game programmer providing images for the game engine to render at the appropriate times. Or, animating a sprite could be done by the game engine, where the game programmer specifies the images ahead of time and the game engine renders them automatically. As another example, the game programmer

could compute whether or not two objects are at the same location, handling a collision if they do so. Or, the game engine could do the same computation, with the collision between the objects being pre-defined, say, by having solid objects bounce off of each other. In other words, if an engine does not provide needed functionality (because, say, the engine didn't envision supporting that functionality), the game programmer can often implement equivalent functionality in game code. There is no single, right way for the engine to behave nor single, correct set of services an engine should provide.

Given that there is no clear functionality as to what must (or must not) be in a game engine, the question may arise as to what, exactly a game engine is for. A major goal of a game engine could be that it be reusable, allowing game programmers to make many different games from one engine, even if they are of a similar variety (e.g., different variants of first person shooter games). Such generality can be supported through the ability to allow game programmers to modify the base game content, such as by providing new models and textures in addition to levels and gameplay rules, giving rise to "mods".

However, many game engines are created with the sole intent of making exactly one game – the current game the developers (and publishers) have in mind. In this case, the functionality of the game engine is designed to be efficient, tuned to optimize the performance of the most common operations (e.g., rendering a textured, 3d model on the screen) quickly. This may be exactly the case for the ogre-drawing engine mentioned above. In addition to efficiency, game engines support many general-purpose functions, such as allowing game programmers to be able to check the state of an object easily.

In nearly every case, a game engine is designed with a certain game *genre* in mind. Even if the game engine is somewhat general purpose, it will best support games created in the intended genre. For example, the target game for an engine may be a side-scrolling platformer. As such, the engine may support keyboard input (say, arrow keys to move and jump) and isomorphic, 2d sprite animations. Using the same engine for a first person shooter, where the mouse is used extensively for aiming and shooting, and the graphics need a linear perspective and 3-dimensions, may be prohibitive. As an exercise one might consider the differences in features for a game engine designed for one of each common genre: Arcade (e.g., *Tetris*), Side-scroller (e.g., *Super Mario*), 3d isometric (e.g., *Diablo*), first person (e.g., *Call of Duty*), Massively-multiplayer role playing (e.g., *World of Warcraft*), Turn-based (e.g., *Civilizations*), and Story-based (e.g., *Heavy Rain*).

While game engines vary in the depth and breadth of game services they provide, there are some components that are common to most game engines that can be identified.

Even game engines that are built from scratch are almost never coded entirely by hand. Writing code directly to the hardware used to be possible and even desirable in early computer games in order to get the maximum performance out of the underlying computer system. However, in today's computer systems, writing directly to the hardware means re-inventing the functionality provided by the many software system layers (e.g., device drivers and memory management). Moreover, given the optimizations modern software may already have in place, doing such replication may often not result in the intended performance speedup after all, while certainly adding the risk of introducing unintended bugs into the system.

Instead, game engines make use of a rich software substrate on most computer platforms. In particular, desktop computers have rich operating system services that can help manage

hardware functionality (e.g., file I/O and networking). Even console systems have software layers that make game programming much easier than writing to the hardware. Robust 3rd-party software libraries exist for graphics, providing a programmer a common interface for simple graphics primitives, as in *OpenGL* or Microsoft *DirectX*. User-level libraries, such as the *Standard Template Libraries*, often provide cross-platform, powerful, efficient utilities for common data manipulations, such as lists and iterators, and math functions. However, for some select features, some game engines make use of core systems that have been developed in-house by the game development team. These core systems may provide low-level services, such as memory management, game engine configuration, parsing (for configuration files), debugging and performance testing (unit testing, profiling and error logging), and system startup (initialization) and shutdown (final state).

Building upon the software substrate, core aspects of a game engine include:

**Representation of the world.** All games have a game world, whether this is a fixed grid with limited topology (e.g., a tic-tac-toe board) or a rich, 3-dimensional, textured, alien landscape (e.g., an massive, outdoor setting). Fundamental to most game world representations are game objects and their locations in the world, given as position attributes, such as (x,y) coordinates, along with their orientations. Positions and orientations can be absolute to the game world, or they can also be relative to other objects, such as a tank turret being at an (x,y) location relative to the tank treads.

**Timing support.** Most computer games operate in real-time, meaning events that happen in the game immediately make a difference to the gameplay, whether an object exploding, a door closing or a player moving an avatar. As such, support for precise timing, often fine grained (e.g., at the millisecond level) is a core game engine fundamental. While some events just need to be relative to other game events (e.g., a grenade explosion that happens 3 seconds after the pin is pulled), other events need to be absolute (e.g., a boss spawning at 9:57pm, 1 minute after the player pulls the switch), and still others need to be synchronized across computers (e.g., a monster dropping treasure at the same time on two different players' computers).

**Low-level utilities.** Much of the time spent executing in a game engine is doing basic, but fundamental, services in support of the game. These included handling resources in/out of files (e.g., reading in a sprite), logging progress and error messages, managing memory, encrypting data for transmission over a network, and more.

Components that are fundamental to a game engine include:

**Graphics system.** The graphics (or rendering) system manages how to display a game scene to the player. This may include resolving issues in lighting, occlusion, and textures. Many game engines support variable camera positions and views over the whole game world. Special effects features (e.g., particles) may also be part of a rendering system. For example, a graphics system may be in charge of rendering a game room filled with 3d objects, complete with textures and lights, from a top-down camera.

**Input management.** Games require input from the user, whether through a keyboard and mouse, a game controller, a touchscreen, or a motion-recognition device. Game engine input management maps specific, often hardware-dependent actions, to game-specific commands. For example, on one platform, pressing the left mouse button may be mapped to a command to move an avatar left, while on a different platform, moving the thumb stick on a gamepad left may be mapped to the same command. Input management must "understand" the differences between hardware types.

**Resource management.** Many game-related assets can be large and have asset-specific formats. As such, one component in a game engine typically manages loading in all assets needed for the game, recognizing their formats and getting them in a form usable by the game. Resource management also manages the lifetime of an asset, keeping assets around when they are in use (e.g., a typical texture for the level) but discards them when no longer needed (e.g., the player progresses to the next level). For example, a game engine may have a particular texture (e.g., a brick wall) that is used when the player is inside a building, but discard the wall texture when the player goes outside. Or, a game may have different background music that needs to be played for the home screen versus during the game.

**Gameplay foundations.** At the heart of a game engine are the foundations that allow a game to be played. Often, this is in the form of game objects that can either be static (not changing over the course of the game) or dynamic (modifying themselves or being modifiable in response to game events). Events that happen in the game (e.g., a player pressing a button) trigger actions in the game, often enacted by messages being sent from one object to another. For example, a player may press the "A" button, causing an "A-button-press" event to be generated by the input manager. This event, in turn, gets delivered to the player's avatar object where it "translates" the A-button to a jump action, moving the object.

**Physics system.** Newtonian physics governs how objects move and interact in many games. To calculate these physical interactions, game engine objects usually have states, providing for location, velocity, orientation and more. Basic interactions need to determine if there is a collision between two moving objects and, if so, what the appropriate reaction is. For example, a typical platformer game may provide "gravity" which gives constant acceleration towards the bottom of the screen. If a movable object (e.g., the player's avatar) encounters a non-movable object underneath it (e.g., a platform), the moving object may stop falling or may even bounce up to a height proportional to the downward velocity.

The above components are all part of the Dragonfly* game engine and provide enough support to develop rich, full-featured games. However, many game engines have additional components.

* **Did you know (#1)?** Dragonflies were among the first winged insects 300 million years ago. Modern dragonflies typically have wingspans of two to five inches, but fossilized dragonflies have been found with wingspans of up to two feet. – "14 Fun Facts About Dragonflies", *Smithsonian.com*, October 5, 2011.

**Sound system.** While a game programmer's emphasis may often be on visuals, sound should not be overlooked – sound can be thought as one-third of the player's experience! Nearly all games have sound. A game engine sound system handles playing music along with dialog and sound effects, often needing to combine sounds as appropriate. Formats along with timing and resource management are often part of a sound system, too.

**Online support.** Computers are increasingly networked, as are the applications that run on them. Games are no exception. Even games that are not multiplayer across multiple computers often have online components, where games connect to a server to obtain player profile information or download new content. Real-time, multiplayer games have specific services that may be provided, facilitating connections to servers or other players over a variety of network devices.

**Artificial intelligence system.** As games get closer and closer to photo-realistic graphics and full-featured sound effects, what many consider the next great frontier for games is in the realm of artificial intelligence (AI). In short, AI is a way of making "smart" objects, such as an opponent that employs a particular strategy or an NPC with which the player can converse. But AI also includes more low-level, but important, behaviors such as pathfinding[1] which can be part of an AI system.

**Tip 1! AI programming for games.** The interested student, and most game programmers, will want to learn more about AI for games. There are many good resources on this topic, but a couple of good places to start are *Artificial Intelligence for Games* by Millington and Funge [7] and *Programming Game AI by Example* by Buckland [1]. The latter is programming-centric, akin to the flavor of this book.

To support these, and other, game components, game engines often provide some basic data structures used by many parts of the game engine and by game programmers. Examples include: array lists for fast indexing, fast insertion/deletion at the end; linked lists for slower indexing, fast insertion/deletion in the middle; and maps (or hash tables) for fast searching and insertion. Sometimes these data structures are custom-built for the specific engine, while other times they may be provided by standard or third-party libraries (e.g., *C++ Standard Template Libraries*, or *Boost C++ Libraries*).

As mentioned earlier, at the heart of a game engine is an object management system. A key functionality provided by the system is run-time type information, which allows the same engine code to handle a variety of objects. For example, a game engine would want the same code to move both a falling mouse and a falling elephant. In C++, this means polymorphism at run-time. Consider a game engine that wants to execute code for a gun to make it shoot. A general-purpose game engine does not want to have special code for shooting a shotgun versus shooting a pistol – such an implementation becomes too tied to the game at hand and is "brittle" if changes need to be made. Instead, the game engine

[1] Computation of the shortest (or best) route between two points.

just knows how to invoke "shoot" and the gun object (created by the game programmer), for example, knows how to perform the appropriate shoot action. In C++, this is done with inheritance and then run-time polymorphism. Consider the code in Listing 1.1.[2]

Listing 1.1: Run-time polymorphism in C++

```
0  class gun {
1    virtual void shoot();
2  };
3
4  class shotgun : public gun {
5    virtual void shoot();
6  };
7
8  gun *p_gun = new shotgun();
9  p_gun -> shoot();  // invokes shotgun::shoot()
```

The last line in Listing 1.1 is the code that triggers the shooting of the gun.* In this case, when the pointer `p_gun` gets dereferenced, since the object type is a shotgun, the `shoot()` method for the shotgun gets executed. If at a later time, `p_gun` changes and points to, say, a pistol, then the same invocation of `p_gun->shoot()` would execute the `shoot()` method for the pistol.

Note, languages such as Java and C++ do run-time typing of objects automatically. A language that does not support run-time typing (such as C) can still do so, but support must be handled by the game programmer.

The focus of this book is on the programming required to create a game engine. This includes:

- How to build core engine components
- How to support player interaction
- How to set rules of play and control
- How to use a game engine to make a custom game

This chapter has provided an overview of what a game engine does. What it has *not* done is provided information about the design and implementation of an engine. For example, take some of the components listed above. How should software be designed to support them? How can game-independent components be separated from game-dependent components? How should the components be defined and organized? Assuming an object-oriented approach (as in this book), what class structures should be used for the various game engine elements?

In addition, there are lots of aspects of game development that are not covered, such as nearly all art (e.g., modeling and animation), most audio (e.g., sound effects and music) as

[2]Note, this is the first code "Listing". There will be many more! All such listings are available for download as a `.zip` file from the Dragonfly book web page (`http://dragonfly.wpi.edu/book`).

* **Did you know (#2)?** The dragonfly has one of the highest success rate of any predator, catching 95% of the prey (mosquitoes!) it stalks. – "Dragonflies: Nature's Most Successful Predator", *Sarah Busby*, 2022.

well as game design. However, there are many other books on those topics which can be used in conjunction with the material in this book.

Similarly, there are many aspects of C++ that are relevant, as well as many, many books that cover them. However, in this book and in the implementation of Dragonfly, some useful C++ mechanisms are illustrated with code, including (but not limited to):

- Conditional compilation (Section 4.3.4).
- Casting (Listing 4.53).
- Operator overloading (Listing 4.38).
- Dynamically-sized arrays (Section 4.5.2.3).

as well as several different software design patterns, such as singleton, iterator, observer, flyweight and chain of responsibility.

In short, keep reading to hear more details on all that, and more.

**Tip 2! C++ Programming.** C++ is still the language of choice for developing game engines and, in many cases, programming games to work with these same engines. Aspiring game programmers would be well-served to be proficient at C++ and own a good book (or two) on C++ programming. Perhaps the "go to" book for answering questions about C++ is *C++ How to Program*, by Dietel and Dietel [2], a book dense with code examples, but also with clear explanations for most anything C++. Another good book to have at hand is *Head First Design Patterns* by Freeman et al [3]. Much of game development involves identification and use of software patterns and Freeman's book shows how to analyze, design, and write serious object-oriented software.