Dragonfly

Program a Game Engine from Scratch

Mark Claypool

Chapter 7 - Projects

Chapter 7

Projects

If this book is used for a course, this chapter provides 3 programming projects that can be used as course assignments for students. Project 1 requires working through the tutorial (Chapter 3) in order to learn Dragonfly from the game programmer's perspective. Project 2 provides a roadmap to build Dragonfly (Chapter 4) through 3 different milestones. Project 3 charts a path for students to build games of their own inspiration using their own Dragonfly engines. Instructors will want to tailor each to fit the particular course requirements and time constraints.

7.1 Project 1 – Catch a Dragonfly

The first project is for students to get used to Dragonfly, meant as a first exposure to a text-based, 2d game engine. Students work through a tutorial that has them make a simple, "stock" game using Dragonfly. This helps students better understand a game engine by developing a game from a game programmer's perspective, providing the foundational knowledge needed for building their own Dragonfly game engine in project 2 (Section 7.2, and designing and developing their own game from scratch with it in project 3 (Section 7.3).

In project 1, students:

1. Visit the Dragonfly Web page and briefly familiarize themselves with the contents. The Web page includes a download of the Dragonfly game engine (compiled – no source code), documentation with details on the classes and methods for the game programmer, and links to games and utilities that may be helpful for subsequent projects.

2. Download the Dragonfly game engine for the environment of choice (e.g., Linux or Windows) and setup their development environment. This means ensuring all the needed external libraries are in place (e.g., SFML), installing the Dragonfly libraries and header files in an appropriate place[1], and creating a basic `Makefile` or project and simple test program to be sure development can proceed. The same basic setup is used for the students' own game engine development in project 2.

[1]Dragonfly can be installed in user space – root/administrator permissions are not needed.

3. Complete the tutorial (see Section 3.3 on page 15). The tutorial has students build an arcade-style shooting game, called *Saucer Shoot*, where the player flies a space ship into combat against an ever-increasing number of enemy saucers. The tutorial has all sprites and audio files needed for development as well as working sample code for students to reference.

4. Extend the Saucer Shoot game in a meaningful way by adding 10% or more functionality. For example, students may add additional weapon types or enemies, health and/or multiple lives, a high score table, or something entirely of their own creation. The actual 10% extension done is up to each student, but s/he indicates what is done with brief documentation when submitting the assignment.

Students work alone for project 1. When done, students turn in a source code package with all code necessary to build their games, including header files and any needed additional sprites (depending upon their extension). In addition, each student includes a `Makefile` or project file for compiling their game, a README file explaining the platform, files, code structure, and anything else needed to understand (and grade) his/her game, and a GAME file providing a short description of the additional 10% functionality extension to the Saucer Shoot tutorial game, including indicating the code written.

7.2 Project 2 – Dragonfly

In the second project, students build their own version of Dragonfly. Project 2 is broken into three parts: A) *Dragonfly Egg* (Section 7.2.1), B) *Dragonfly Naiad* (Section 7.2.2) and C) *Dragonfly* (Section 7.2.3), that build upon each other to end with a fully functional, full-featured game engine. Since it is critical that game engine code be easily understood (from the game programmer's perspective) and, equally importantly, robust, the three projects are structured such that completing parts A and B provides for a fully functional, if somewhat limited, game engine. This level of proficiency enables students to proceed to project 3, where they make games using their engines. Completing part 2C provides for a full featured game engine, with functionality that makes it easier to create a broader range of games.

For timing and grading, the due dates should be staggered so that most of the time is allocated for part A and part B, but there is still time for completing part C for the top students in the class. In addition, points are allocated such that completing part B is sufficient for earning a "B" grade for project 2, while fully completing part C provides an opportunity for earning an "A" grade for project 2.

Students should be informed that it is much better have tested, trusted robust code that only implements part A and part B then it is to have buggy, partially working code that attempts to get into part C. Since students make use of their own engine for project 3, wise students tend to heed this advice.

Students work alone for all parts of project 2. While group work is important for many aspects of software engineering, including game development, developing the engine solo ensures students have complete and deep understanding of both the game engine and the programming skills needed to develop it – there is no way to "hide" behind a more experienced teammate. That is not to say students are alone, however – discussing the project with other students is encouraged, even for help in debugging each other's code. The line is drawn at not allowing sharing of code in that each student must write all the engine code him/herself.

All development is done in C++. Students are expected to be familiar with C++ from earlier computer science classes, but are not expected to be experts in the language. While development is done as "homework" outside of class, the requirements and design of Dragonfly are presented in class, with discussions of design rationale, implementation choices and alternatives, and more advanced features.

Individual classes, with high-level descriptions of attributes and methods, are provided.

7.2.1 Dragonfly Egg

Part A of the project is to construct the foundations of a game engine that provides the following capabilities:

- Game initialization: Start and stop gracefully.
- Logging: Write messages to a file, including the values of variables of different types (e.g., integers, strings, or floats).
- Object support: Add and remove game objects. Objects support 2d game world positions for objects.
- Game loop: Run a game loop with: 1) A fixed update rate (e.g., 30 Hz), and 2) updates sent to all objects each loop

To implement this functionality, students develop, code and test about a dozen base classes.

No visual depiction of the game is required for part A. Instead, all output is done via printing to the screen or to a log file via the logfile manager functionality built into the game engine. As suggested above, at the successful completion of part A, students do *not* have a game engine. Instead, they have a robust, foundational code base they can build upon to get a functional game engine in part B.

7.2.2 Dragonfly Naiad

Part B is to continue construction of the game engine, each student using their own code base from part A, adding the following additional capabilities:

- Output: Support 2d, text characters with color. Provide a clean refresh each game loop.
- Input: Accept non-blocking keyboard and mouse input. Send input to all game objects.
- Velocity: Automatically move objects with an (x,y) velocity. Support speeds both greater and less than one space per step.

- Collisions: Provide a "solid" attribute for game objects. Detect collisions between solid objects. Send an event to both objects involved in a collision.

- Misc: Provide deferred, batch removal of game objects. Provide support for an "altitude" attribute for game objects to support layered drawing. Notify game objects that move out of game world with "outofbounds" event.

All of the above capabilities must be thoroughly tested, bug-free and ready for a game programmer to make a game (the students themselves, in project 3).

7.2.3 Dragonfly

Part C is to continue construction of the game engine, each student using their own code base from part A and part B, adding the following additional capabilities:

- Sprites: Provide multi-character frames. Associate one or more frames with a game object. Play frames in sequence to achieve animation. Support "slowdown" of animation to less than one frame per game loop.

- Resource Management: Read sprite data from files. Provide bounding boxes for game objects. Allow game objects to be larger than a single character (for movement and collisions). Associate bounding boxes with sprites.

- Sound: Support sound effects and music. With resource management, load/associate audio for sound and music.

- Camera Control: Allow the game world to be larger than the screen, providing a "viewport". Enable free viewport movement around the game world, including the ability to follow one object (e.g., the player's avatar).

- View Objects: provide an alternative (to game objects) object that supports "heads-up display" functionality.

As for project 2A and 2B, all of the above capabilities must be thoroughly tested, bug-free and ready for a game programmer to make a game (the students themselves, in project 3).

For each part, students turn in a package with all code necessary to build their game engine, including header files and a `Makefile` or project file for building their engine. Game programmer code (i.e., code someone would write using their engine) is required to demonstrate the full functionality of what has been built (so far). This can be more than one program, if needed. Documentation is required to explain the platform, files, code structure, how to compile the engine and game code, and anything else needed to understand (and grade) a student's game engine.

7.3 Project 3 – Dragonfly Spawn

In project 3, students use the Dragonfly game engines they built in project 2 to make their own, original games from scratch. The end result is expected to be a robust (bug-free), playable, and balanced game (it may even be fun).

Like a typical, large game development effort, the project is broken into several milestones: plan, alpha and final. Each milestone is submitted and graded separately, while all apply towards the total project 3 grade. The intent of the milestones is to provide production guidance to yield a fully-functional, complete, playable game built with their own game engine.

Students work in teams of two for project 3. Students are free to partition the work among the team as they see fit, but all team members are encouraged to help (say, with design and debugging) and be knowledgeable (in terms of how the game code executes) for all parts of the game.

Development must be in C++ using their game engine from project 2. Under exceptional circumstances (e.g., both partners not completing project 2b), students are allowed to use the pre-made Dragonfly engine from project 1. No engine source code is provided, however, only the pre-compiled engine.

7.3.1 Plan

Student teams provide a game plan document within the first two weeks of the project. The plan document provides a detailed description of the game they plan to build, including the technical challenges it entails, a bit about any significant artistic aspects of the game, and the timeline to successfully complete development in the time provided. In planning, students are asked to draw upon experiences from other classes (e.g., other programming or game development courses), to inform the creation of the plan document. While the actual length of the plan is not a requirement, as a guideline the plan is expected to be approximately 2-3 pages – much less and students have probably have not supplied enough details.

For the plan submission, students turn in a written document.

7.3.2 Alpha

At alpha stage, the student games have all of the required features implemented, but not necessarily working completely correctly. Game code must be tested thoroughly enough to eliminate any critical gameplay flaws, but minor bugs or glitches may be present.

Games must compile cleanly and be runnable, even if all aspects of gameplay are not available from one program. Separate features of the game may be demonstrable from separate game code programs (e.g., separate game programs illustrating a kind of weapon or a specific opponent).

Games are likely not yet be finally balanced nor the levels designed for all experiences (beginning to advanced) of game player.

Games may contain some placeholder art assets. For example, in the alpha release, a simple, non-animated square may be used for an opponent with the intent of creating a figure and frames of animation for the final version.

For the alpha submission, students hand in a package with all the source code necessary to build their game engines and their games. All header files must be included, as well as `Makefile`s or project files for building the games.

7.3.3 Final

The final game versions have all game content complete – design, code and art. Games must be tested thoroughly for bugs, both major and minor, removing all visual and gameplay glitches. Game code must compile cleanly and be easily runnable. Upon startup, instructions for the player on how to play must be readily available, and with clear indications on how to begin play. Gameplay must be balanced, providing appropriate difficulty for beginners and/or early gameplay, with increased difficulty as the game progresses. Games must have a clear ending condition (i.e., winning or losing) and the player must be able to exit the game easily and cleanly.

For the final submission, students submit their engine and game, with necessary support files and `Makefile`s or project files. The typical READMEs are required, as well as DESIGN documents providing all the details in the plan, but updated to reflect the games as actually built. For example, the functionality, milestones and work responsibilities need to be updated from the plan to reflect the development. Major deviations from the original plan must be noted.