GAME ENGINE TO SIMPLIFY PYGAME DEVELOPMENT ON A CONSTRAINED PLATFORM

Lewis, Justin. jtl1728@rit.edu. Computer Science, GCCIS, RIT.
Hockey, Kevin. kdh7733@rit.edu. Computer Science, GCCIS, RIT.

Mentor/Supervisors:
Jacobs, Stephen. Associate Professor Interactive Games and Media, RIT. sxjics@rit.edu
DeCausemaker, Remy. Lab for Technological Literacy/Center for Student Innovation, remyd@civx.us

Introduction

It's good to give users control, but at the same time with great power comes great responsibility. If a programmer wanted to quickly write a game for the XO he or she would need a framework in place to automatically deal with resource management. Writing a game without any planning can lead to complicated, hard-to-follow and ugly code. We found this to be true when building our Mathematical Adventure: Fortune Hunter game. Our original approach was very monolithic and we soon found that there was a need for a small game engine: to manage resources and organize our code by creating smaller objects which register their events and drawing with the engine.

Background

One Laptop Per Child (OLPC) is a project who's goal is "to create educational opportunities for the world's poorest children by providing each child with a rugged, low-cost, low-power, connected laptop with content and software designed for collaborative, joyful, self-empowered learning."[3] The OLPC project is "...not a laptop project. It's an education project."[4] They have developed a laptop that meets these requirements called the XO.

The XO Laptop runs a python driven environment called Sugar, which is a platform built on top of Fedora, with the aim of making the computer more easily navigable for children. Sugar is designed to be used by children in an intuitive and collaborative environment. Instead of applications, there are activities; self contained python code that is designed to be easy to install and use. Activities are usually designed to allow networked play and learning. With education in mind, it is also designed to allow easy access to the source code of any activity on the XO so the user may learn about the program if they desired. Because activities are primarily written in python, games written for the XO generally uses Pygame as their event and drawing handler as Pygame provides extra functionality that complements the standard drawing library and helps developers write multimedia games.

Motivation

The games we are developing for the XO utilize the Pygame library. The Pygame library is a cross-platform portable framework built on top of the Standard Drawing Library. The framework is designed to be modular and simple. Because of this, Pygame's developers take pride in the fact that "Pygame is used in the OLPC project and has been taught in essay courses to young kids, and college students."[1] It is developed with the goal to leave the developer in charge of the code, their documentation even states that "You control your main loop. You call pygame functions, they don't call your functions."[1]

Giving the developer full control can be great when they take the time to design their program. When rapidly developing programs for a class, one doesn't have a lot of time to design and redesign the program as it becomes more sophisticated. Because of this, the code will slowly get convoluted and complex. Generally games are run by tight event loops: written to be as efficient as possible. Rapid development without planning leads to large monolithic event loops that grow out of control and eventually become very chaotic.

Fortune Hunter was a game that was plagued with a huge event loop. It began with a simple loop to run the game, but inexperienced developers were learning how to use Pygame as they wrote the code and eventually the loop grew out of control. Every time a new feature was added to the code, a new if-statement was added to the main loop to direct program flow to this new block of code. As things grew out of control, these blocks of code became separated which led to redundancies and illogical event flow.

By building a simple game engine that hides the main event loop of Pygame the developer is encouraged to implement more modular code.

Main Engine Strategies

Every object that registers with the game engine passes the engine its own event handler function; the game engine implements a simple loop that calls functions of registered objects when any event takes place. The engine also provides a few features that help the programmer to debug their code: a game console and a basic time profile system show which registered functions are taking the most game time. The engine is responsible for setting up the window with Pygame, holding objects within the engine that are globally accessible, handling Pygame events, simplifying timers, and determining when the window should be redrawn.

Evolution of the Event Loop

Throughout the evolution of the engine, three types of event loops were tested. The first implementation used a basic event loop. The disadvantage of this type of loop is that an event must be fired to wake up the drawing method. When the event is fired, it would be passed to and ignored by all of the registered objects. Having components ignoring the event was less than ideal as function calls are fairly expensive (time-consuming) in python. The one advantage

to this type of loop is that there was no unnecessary drawing.

The problem was that we needed a way to support animation, and a timer was also required to fire events into the queue; this was impractical as the number of registered event callbacks increased. The solution was threaded event and draw loops. This would allow animations to run independently from the main event loop, which frees up the event loop so it only has to deal with the logic of the game. One of the problems with this solution was that the drawing system was always drawing, even when nothing was changing. It also required the use of a thread lock that prevented the event and drawing loops from updating during a draw cycle. Python Threading is already slow due to the Global Interpreter Lock (GIL) that prevents multiple threads from utilizing a multi core system. By adding our own lock on top of the Python GIL, the performance hit was noticeable. The XO laptops only have one core anyway.

The final accepted implementation of the event loop is a Non-Blocking Event Loop. The major characteristic of this loop is that when it queries Pygame for events, it will return NOEVENT if there is nothing in the queue instead of waiting for one. When the engine gets a NOEVENT signal, it then runs a draw cycle. The advantage to this system is that there are no timers needed to wake up for animations and animations only run when the event loop has nothing in its queue. This solution doesn't use threads, so Python's GIL is not an issue. Timers are no longer used, so we avoid the overhead that comes along with using them to create events to wake up the system for animation.

Animation Sub-System

In addition to handling the events and drawing functionality of all classes registered with the engine, there are also classes added to make animating easier and more efficient. The scene class keeps track of what needs to be moved and what needs to be updated. Drawable Object and Dynamic Drawable Object provide wrappers for images and animations so that they can give the scene what info it needs. Ever engine has its own scene and this system integrates with the engine with the "keep the resource management out of the programmer's hands" mentality. All objects are added to the scene via functions provided by the Game Engine Element class and then they are all removed automatically when the element is removed from the engine.

Proof of Concept

Fortune Hunter was originally written to be a project in the fall 2009 OLPC seminar class, because of this not much planning went into main loop design. When we went and started developing it as a full fledged game we foolishly built upon the shaky foundation. The result is a game that runs slowly, with code in near unreadable blocks. As a proof of concept we took this large monolith and successfully modularized it by rewriting it with our engine.

The game engine now keeps track of game data and user data; creating and removing objects as needed. This approach dramatically reduces the game's memory footprint - a serious issue for constrained platforms. Extending the Game Engine Element class makes dividing the code into modules more straightforward. Each module now contains its own drawing statement,

making it obvious where what is actually being drawn. Previously things could be drawing anywhere, as the code had draw statements randomly strewn about the main file. The framerate has jumped from 1 fps to 8 fps, meaning that games can be played on a platform as constrained as the XO. The game can now handle animations; before the game had to deal with far less taxing operations and still ran far slower.

Two other games ported over to the game engine were Lemonade Stand and Produce Puzzle. Both games are written utilizing the engine's "non-animation mode" which is useful for games that don't have animations. This mode only draws when the game tells the engine it needs to update, resulting in far less CPU usage for games that don't need the resources.

Impact

Supplying a simple game engine allows experienced and inexperienced developers to write cleaner, well organized code, that will run more efficiently on constrained platforms like the XO. This allows developers to quickly pickup the code as they are following an existing framework. This impacts the open source community by helping to more easily assimilate developers to current projects. If someone has coded a game that has used the same engine before, they will already be familiar with the design of the game. Modules are easier for people to pickup and understand because they can attack it piece by piece - as opposed to one large intimidating block of code. By encouraging developers to write modular code to connect to the main engine, projects becomes cleaner and more understandable.

The engine also provides a nice wrapper for Pygame's timer system. Pygame requires the user to start one of eight timers. When the time expires it fires an event into the event queue. It is up to the developer to keep track of what timers they have running and how to handle the event when it enters the event queue. The game engine hides this functionality from the developer. If they wish to use a timer, they just pass in the function they want called and the time between calls and the engine will take care of keeping track of the timers. The developer no longer needs to keep track of what timers they have running, instead they just need to know what functions they registered.

Debugging is also simplified by the engine. This is because of a few features the engine has provided. There is a developer console built into the draw and event loop that allows the developer to inspect the game state by examining any object registered with the engine. It even has the ability to inspect elements inside the object recursively. The console is also keeping track of time information which allows the developer to see what callbacks are taking up the most time, and where the game could use some optimization.

Due to the simplicity of the engine. It also provides ideal locations to place debugging information that would allow the developer to get a better idea of how the system is running and where problems are occurring.

Future Research

The engine still has room for optimization. It is always better to optimize the code that is being run 90 percent of the time instead of the slower code running 10 percent of the time. Therefore optimizing the core of the engine will provide the most improvement. There are a few different ways the engine could be improved. One possible improvement would be to research the use of generators and co-routines for callbacks in the main loops. Python's function calls have a measurable overhead that could be reduced or bypassed using the co-routine method described in pep-0342[5].

Another source of optimization would be to implement the engine in Cython[6]. This would allow the engine to be implemented in Python with some optimizations that the C language provides by allowing the use of ctypes and declarative types that are compiled. Cython works well with interfacing C into python.

For a fully optimized engine, implementing it entirely in C would provide the largest performance boost. The engine could be interfaced using Python's C-Api[7]. The disadvantage to this is that the engine would no longer be written in Python and would make it less appealing for most Python programmers and the Sugar Environment.

Links

References

[1] http://www.pygame.org/wiki/about
[2] http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock
[3] http://laptop.org/en/vision/mission/index.shtml
[4] http://laptop.org/en/vision/index.shtml

Further Research

[5] http://www.python.org/dev/peps/pep-0342
[6] http://www.cython.org
[7] http://docs.python.org/c-api

Related Links

http://www.pygame.org
http://www.python.org
http://www.libsdl.org

More Information

http://foss.rit.edu/projects/fortunehunter
http://wiki.sugarlabs.org/go/Fortune_Hunter
https://fedorahosted.org/fortune_hunter/wiki/FortuneEngine