Hangar 13 Coding Test Document

Name: Liang LUO
Email: bobluoluo@hotmail.com
Tel: 201-320-2618

Requirement:

It is a 2D shooting game with some colliders on the screen bouncing around.

Hours:

25 - 35 hours

Environment:

Visual Studio 2013

Overview:

The game is based on the good-old Model-view-Controller-ish architecture. “Model” is the engine part that is for abstracting the general purpose of game programming. “View” is the interface from inputs, rendering and feedbacks in a game system. “Controller” is the game logic. It hooks game together.

Specifications:

I. Memory management:

a. Allocator

Use “new” to allocate memory and “delete” to deallocate memory. There is no need for a custom allocator or overloading “new” keyword because this game won’t have memory constraints because the golden rule of computer engineering is “no great feature comes free.” And “free” means performance penalty.

b. GC

C++ also does not have default Garbage Collection functionality but it doesn’t mean I should have one(c++11 has smart_pointer though), because GC alleviate the working burden of memory leaks in bigger project. Usually, I would like have a StackGuard<T> for my projects because we don’t like naked pointer, but I can not see why I need one this in current practice.

C. Factory pattern: src/game/

Actor and sprite factories are responsible for allocating memory/objects. With the help of the factories, I released the pressure of memory allocations/deallocations from other parts of the system.

II. Containers

Use “STL” library. Sometimes we do prefer custom container for out projects simple because it hack-able and handy and, more importantly, it is a good way to handle future performance bottleneck due to the fact that container classes are memory-adjacent concerns. Hence,

general-purpose “STL” will be just fine for this project.

II. Math

Math library does do the heavy lifting in game engine since I can not find a single case that we don’t need math in game programming especially in graphics development. For simplicity, 2d vector class would be enough for computing 2d graphics and collision.

III. Game Loop

a: timing:

The framework adopts a method of using fix timing by providing a fixed cap of frame timing. Sleep(..) helps a program doesn’t run too fast while it does not solve if my game runs too slow. Bigger commercial games will not satisfy with this way of dealing with game time, but it seems to be good enough for this game with only simple geometries running on it.(To be honest, I really don’t like this pattern of timing, but the requirement asks “no touching the framework”. So...)

b. C_Application:

However, game loop only containing one method: ticks() is not good enough to me. It should at least have 4 methods:

Naming:

header: C_Application.h
cpp: C_Application.cpp
class: C_Application
folder: src/
namespace: hctg (Hangar Coding Test Game)

Implementations:

Init() : void // do game initialization e.g. Parse “config.init” file
handleInput(key : InputKey) void // handle input
tick(): void // update game states. E.g. Collision detection
draw() : void // draw all drawables, separates rendering from game logic.

III. Input src/engine/input

A singleton class that stores key states in order to track key pressing.
Since I am not allowed to touch the framework of the coding test, I will skip wrapping the input interface.

III. Event(Messaging) System: src/engine/events/

Event system, I think, is the one of the greatest system in the software engineering. I used event bus to abstract user commands from user input. I call them “USER_CMD events. If an event system can implement correctly, it is also good for serialization that benefits online communication.

IV. Data-Driven : src/engine/data

Rule #1, no hard-code data. Data-driven is great because it allows hotting loading and user modifications without recompiling the code. And, it prevents the potential of contaminating the code. It helps game designer to tweak the game without knowing the code.

In normal condition, I should build a parser for JSON, INI, etc files to abstract data from the whole system. But the example you provided doesn't seems to support some kind of scripting system. So, for simplicity, I load all data in resource.h/resource.cpp - a game designer can tweak thee game there.

V. Scene Manager: src/engine/scene

The core dispatcher and data-structure for the game.

VI. Interfaces

The global view of a game is actually a pipeline system. A game engineer has to generalize some concept of objects and process them in queue. It is still valid idea even for a mufti-threading system. Therefore, I need some interfaces. With interfaces I can push them to the pipeline whenever it is necessary.

My Interface are:
IColliders: can be pushed to collision detection system, checking with a scene-graph a spatial partitioning structure using spatial hashing.
IDrawables: can be pushed into to render and be rendered on the screen.
ITickables: can be updated in the game loop.

VII. Collision Detection: src/engine/scene/scene_graph

My options are quardtree and fixed grid. Quardtree is not a very ideal structure for updating huge amount of dynamic objects. The fixed grid with spatial hashing can handle movement-intensive computing very good, which is applicable for current project.

And thank you for such a fun coding test.