Android Final Project Report

This document records design decisions and other notes regarding my final project from COMP7970.

Game engine

The game engine was designed to be highly versatile, which has benefits and drawbacks.

Assets class

Much of the functional design centers around the idea of the Assets class. This class, constructed specifically for the game and not to be confused with the Android AssetsManager, serves as an access provider for most of the objects needed for any given frame of the game. It has several public, static member fields by which the disparate parts of the game can access one another without going through getters and setters. The motivation for this is twofold. First, based on several method traces from DDMS at various points in development, I found that the cost of repeatedly accessing objects through method calls was not trivial, and that the access could be sped up by using direct access. The runtime savings were about 12-18%. Second, as the only person working on the game, I can guard against improper use, while still being able to take advantage of extensive runtime modification to the objects in any frame to support things like special abilities and level requirements.

This setup would be a poor choice if more than just one person were responsible for the code. In addition, there is little to guard against macro-level errors, like instantiating objects to go in certain fields of Assets in the wrong order. That being said, in a tightly-controlled environment like a video game engine, a runtime error of any sort generally derails the game, so preventing them should be happening at design time as much as possible.

In addition to the various runtime objects held in Managers, the Assets class provides access to sounds, music, and a random number generator.

Manager objects

Nearly all in-game entities are held in one of several different types of Manager objects: the EntityManager, the BulletManager, and the ParticleManager. All three Manager objects hold the types of objects you would expect them to. The main game loop is responsible for simply calling the update() and draw() methods on each of these managers each frame, and the managers chain those calls to any objects they contain that should receive them. The Manager objects are responsible for performing “garbage management” of any of their contained objects by checking for appropriate conditions in each frame and removing references to any objects that should be cleaned up.

EntityManager and BulletManager

The EntityManager holds references to the player ship, player-allied objects, all enemies, and all other assorted “neutral” entities, like pickups dropped by enemies. During the collision-

detection phase, only objects that can interact with each other are tested for collision. For example, enemy bullets do not need to check for collision with other enemies. To do so would quickly become prohibitively expensive, as this would ultimately be an $O(n^2)$ algorithm that blew out of control with any serious number of enemy and enemy bullet objects.

The BulletManager performs as you would expect, used in collision detection. It is separated from EntityManager for purely semantic reasons. Objects in this manager are cleaned up if they have either left the field of play, or have collided with an Entity in this frame.

ParticleManager and particle effects

All particle effects in the game are routed through this manager. Individual particles are tied to ParticleEmitters, which determine the type of particles shown, their movement pattern, and their decay rate. Each individual particle is responsible for updating and drawing itself, and when any particle has decayed to an alpha color value of 0, it flags itself to be deleted by its parent emitter. When any given emitter no longer has any particles it is managing, it too flags itself for deletion by the ParticleManager.

Routing all particle requests through this manager allows for two important framerate considerations. First, the number of particles spawned by any given emitter can be toned down based on user preferences. This change can be made in one place rather than in every single object that needs to spawn a particle effect. Second, individual particles can be pooled and reused. Spawning and garbage-collecting new particles each time quickly becomes an unmanageable drain on the processor. By pooling used particles, the ParticleManager can draw from an existing pool instead of instantiating new particles each time. This of courses cases the program to incur a (very slight) hit in runtime memory footprint, but with the benefit of a much more stable framerate.

Collision detection

All collision detection is implemented through a fully-static CollisionScan class. This makes it a class that can be swapped in and out quite easily if different kinds of collision detection are required, or if I come up with more efficient collision detection algorithms later.

In each frame, all objects that could feasibly collide are checked against all objects they could feasibly collide with. Again, although this is ultimately an $O(n^2)$ algorithm, I shrink the potential pool of collision targets based on the designs outlined above.

All objects available for collision detection have a BoundingShape that (either loosely or precisely, depending on the object) outlines the area that should register a collision. BoundingShapes can be either rectangles or circles. For detailed notes about the algorithms used to determine collision, please see the project proposal document or the source code.

Game loop

All of the above lead to a game object that is extremely clean. It consists first of a constructor that prepares all necessary objects. Second, an update() loop chains that call to all Manager objects, the Level object, and the Control object, as well as asks for a collision scan. Third, the draw() method again chains that call to all necessary objects.

The update() method receives as a parameter the amount of time that passed since the last call to update(), which is used to standardize the events of the game regardless of framerate.

Object “blueprints”

Because many of the objects (ships, weapons, shields, etc.) have to be accessed both by the combat engine and by the Android UI-driven preferences and loadout sections of the game, I separated many such objects into “blueprints” and then actual instantiations. The “blueprint” objects (which are named appropriately: ShipBlueprint, etc.) contain all the information that could possibly be needed to process any configuration of the object in question. This may include its power, its increases for leveling, its worth, text descriptions, names, or any other such information. However, much of this information is unnecessary in the frame-to-frame rendering of the game. Said object is boiled down to an implementation of the object with no extraneous information for use in the combat engine. For instance, the player’s Ship object is only concerned with the exact attributes with which it is instantiated, not all possible attributes for that class of Ship. Therefore, a Ship object takes a ShipBlueprint as its constructor, determines the attributes it needs to have partially in combination with that ShipBlueprint, and discards the ShipBlueprint object since the rest of it is unnecessary.

Game framework

The framework for the game is similar to the Android component of the libGDX framework, which is itself licensed under the Apache 2.0 License. However, the specific framework used in my game was constructed by hand, based on classes outlined in the Beginning Android Games book by Mario Zechner. Zechner is the lead developer of the libGDX framework, which explains the similarities.

The framework provides interfaces between the Android system itself and the classes used by the game engine. Specifically, the framework provides hooks to access the Android frameworks for sound, input, graphics, etc. Most of the framework pieces are used as-is, with the notable exceptions of the AndroidGame and AndroidGraphics classes, which were modified for use with BlockInvaders.

Multi-threaded approach

As Block Invaders is a real-time and thus processor-intensive game, I make use of a second thread in addition to the standard UI thread. This second thread, which is spawned by the AndroidFastRenderView class, handles the update() and present() methods of the game. It is also responsible for reporting the deltaTime to those methods. The game will run as fast as it can redraw itself to the screen, but utilizing the deltaTime allows the game to perform at the same update speed regardless of the graphics on screen. The downside of this is that very slow processors (VERY slow processors) will not be able to display the game frames fast enough to play. This is problem faced by most if not all modern games. The converse, called frame-syncing, which was popular in games made for arcade cabinets, means the game runs exactly as fast as it is drawn. Because arcade games were made for very exact system specifications, this was desirable and controllable. Since Android is in no such position, frame syncing is not an option. Only in the most extreme cases will the deltaTime approach make the game unplayable. The upshot is that on nearly every other system, all users will get the same game experience, framerate notwithstanding.

Fixed graphics display

To accommodate different screen sizes, the game begins by querying the actual dimensions of the screen in pixels. It uses this information to set a pair of scale values. Each touch event is then multiplied by the scale values to convert from the actual screen location that was touched to the in-game location that the framework reports. Second, all graphics are drawn to a fixed resolution of 320x480. Once all drawing has been computed to an Android Bitmap of that size, the Bitmap is scaled to the actual screen size before being displayed.