

J. Luke Scott

*Mobile Computing
MiCS Program
University of Luxembourg*

Mobile App Project
Monday 4th June, 2012

BONK! – Final Summary

1 Introduction

Our game, BONK!, is a simple multi-player target shooting game, reminiscent of classic 2D arcade games but with a newer, quick-game-play mobile app feel. It was designed to be a quick diversion among friends with a few minutes to kill (or to BONK!).

During implementation, our original designs for this game proved to be very effective. Only a few details or our plans showed themselves to be untenable. And for my implementation, I chose the name BONK! (instead of Slam!). But because of the challenges of writing code while learning new libraries, and because of the odd behavior that always happens with young, cutting-edge platforms, some buggy behaviors were very hard to pin down and development was slow. Not all features were implemented, but the proof of concept that has been created does show a good level of functionality. The remaining work to do is fairly straightforward.

2 Graphical User Interface

The Graphical User Interface (GUI) uses the Android platform's standard View objects; other animation strategies exist, for instance using a "Drawable" object to insert custom-generated bitmaps. But for the relatively small number of animated objects on the screen, and because of the

2.1 Animation

The bullets are animated with "Animator" and "AnimatorSet" objects. Because bullet motion is defined by its trajectory and its rebounds off of the walls, using AnimatorSet objects in the Android environment is suited perfectly.

An object trajectory can be split into its travel segments; one segment represents the path traveled

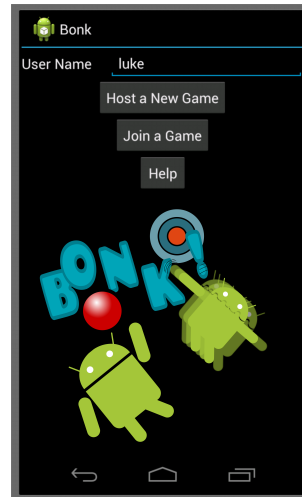


Figure 1: Main entry screen

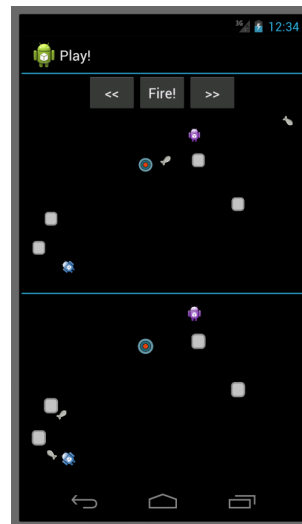


Figure 2: Players firing bullets in direct view (upper half) and network view (lower half)

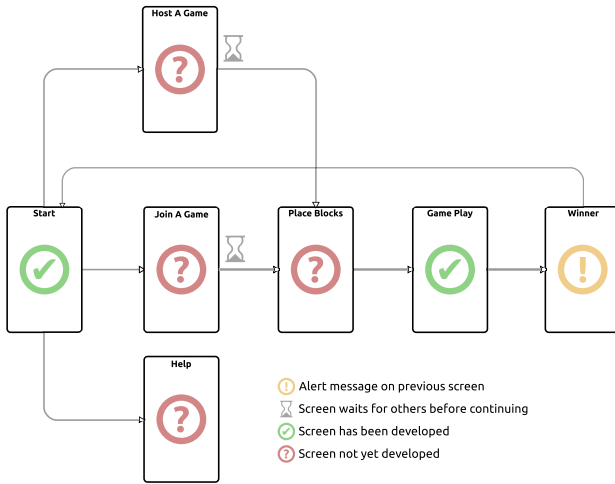


Figure 3: Storyboard of BONK! User Interface

between two walls. In this way, a bullet that is fired, bounces four times, and then hits a target (or burns out) could be said to have five travel segments. But I chose to represent x -axis travel separately from y -axis travel. Each rebound off of a wall hits either a vertical wall (reversing the x -axis direction) or a horizontal wall (reversing the y -axis direction), so by separating the axes I could consider travel segments to be the path between rebounds on a single dimension. By doing this, calculations are much simpler, and I discovered an added benefit that the path of travel is a bit goofy and unpredictable – adding to the silly feel of playing the game.

To calculate the travel segments, BONK! first takes the direction, speed and remaining lifetime of the bullet, and by doing some simple geometry, it calculates the remaining x - and y -travel distances. Each axis of travel is considered separately. Starting with the current position of the bullet, the distance and time until the next rebound on an axis is calculated. An animation segment for just this axis segment is generated, and the distance and time is subtracted from the remaining values.

Before repeating the process, the “remaining distance” value is multiplied by -1 so that the next calculation will cause the bullet to travel in the reverse direction for that axis. Also, the direction of the projectile is set to flip over the appropriate axis by adding an `AnimationListener` which is triggered at the end of each animation segment

and calls `setRotation()` on the bullet. Then the generator function calls itself recursively and the process repeats until no more distance is remaining to travel.

Then the other axis is calculated in the same way, and the two axes are added together in a parallel series using an `AnimatorSet` object. By playing this single `AnimatorSet` on the bullet, the Android system does all the work of animating the bullet using its standard `View` objects.

As the device gets updates over the network of current bullet positions, it removes the existing `AnimatorSets` from the bullets being updated and re-generates them based on the received time, position and direction data.

3 Protocol

Only insignificant changes were made to our protocol. One bug was found and fixed (the playground initiation JSON structure did not have a `messageType` field), and the name of the application was changed from Slam! to BONK! – forcing the “application” field of the invitation and acceptance messages to be changed.

Also, for simplicity, in the demo application, only UDP communication is used. For the full implementation, we still planned on using direct TCP communication because of its reliability guarantee. TCP is a slightly more heavyweight protocol than UDP – sending an acknowledgment packet with each received communication packet – but our application’s message protocol is already extremely lightweight and is not expected to saturate network capacity. Because the overhead incurred in TCP communication happens *after* the message is delivered, liveness is not expected to depend on this decision at all.

3.1 Lightweight Communication

Our application uses an extremely efficient, fine-tuned, low level protocol for sending game actions between devices. Bit-level values are con-

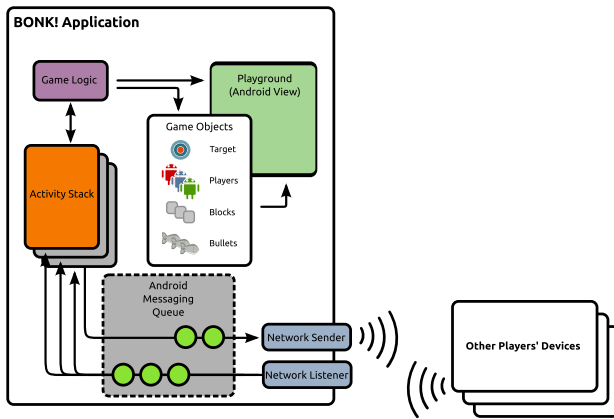


Figure 4: Application Components

sidered; three bits represent the “owners” of objects (either the players or the game itself), and 13 bits are used to represent the other objects owned by them. Positions are represented in a $[0,255]$ range, so a two-byte combination can be used for a 2D location. Similarly, the direction of a player or a player’s projectile is communicated using a single byte. These efficiencies allow the entire game state (up to over 100 objects) to be guaranteed communicable in a single packet.

4 Architecture

The application is implemented using a common Activity parent class for all BONK! activities. This common Activity contains some view helper functions, start-up code to process its parameters (from the Android Intent), and to initialize network communication. (See Figure 4.)

All network communication is done using a `NetworkCommunicator` object hierarchy. `NetworkCommunicator` is a subclass of Java’s `Thread` class, so they each run in their own thread of execution and communicate via Android’s `Handler` messaging mechanism. The child classes, `NetworkSender` and `NetworkListener`, handle all low level communication functionality, exposing only a few sending/receiving methods. `listener.getHandler().sendMessage()` is the method used to send a message in BONK!; at the time of writing, minor development

was planned to convert it to a static API similar to that used in `NetworkListener`. The static version of this call would simply be `NetworkSender.sendUdpByteString()`, and would hide the threaded/message-based nature of this component.

The technique for listening for messages is to use `NetworkListener.registerHandler()` to register a callback on the `BonkActivity` object. This is done during start-up in the base `BonkActivity` class, and the callback event is wrapped into a more convenient hook mechanism. The various subclasses of `BonkActivity` only need to implement one of a few hooks used by this callback to relay game invitation, acceptance, setup and game play messages.

All game play is marshaled by a `Game` class. Each instantiation of `Game` is its own separate environment of players, objects, and game state. Although an admittedly confusing naming decision, a `Game` object stores a hash map of many `GameObject` subclasses, which are the objects being manipulated while playing the game. The `Game` class exposes simple gameplay-related commands like `rotatePlayer()` `fireBullet()`, etc., and it triggers re-drawing objects on the screen and communicating with the other devices.

Each `Game` object contains its own `Playground`, which is a BONK!-specific Android `Layout`. A `Playground` is the parent of one Android `View` object for each `GameObject` in the `Game` object’s canonical hash map.

Game initialization is currently done with a test button because the host/join screens have not yet been finished. In the final product, the host device would generate a random arrangement of target, players, and blocks, and send it over the network to the other players in a JSON message. The JSON message is interpreted by the `PlayGameActivity`, which calls the appropriate commands on its `Game` object to set up its `Playground`.

5 Implementation Challenges

During development, two platform issues came up and one theoretical challenge. The platform issues were normal (expected) events that happen when developing in a new environment: network communication (and more generally, threaded I/O), and undocumented quirky behaviors within the platform libraries.

5.1 Animation Engine Issues

To get the animation engine design to where it is now, I went in two large circles. I actually started with this idea very soon after reading the Android animation documentation. Putting the Animator and AnimatorSet ideas together, it was obvious to me that dividing the x and y axis would simplify the code quite a bit. But because of (still not understood) issues with a `setRotation()` wrapper that I had been using, objects were being rotated out of their range of drawability during the course of the animation. They would flicker a bit and then magically appear at the end of their trajectories. The only fixes I could find were to introduce hacks that forced the animation engine to re-draw constantly – not a scalable or elegant solution at all.

Along the way, I implemented an idea to do a single animation at a time, each of which used a listener to create the next animation, effectively writing my own AnimatorSet (this only changed the problem). Another attempt was to bypass the `ObjectAnimator` and use `Animation` and `AnimationSet` objects instead; I also considered using the more video-friendly `Canvas/Drawable` system, and even `OpenGL`. I wrote some test code for each solution and stubbornly stuck with `ObjectAnimator` because it fits the requirements so well. I performed lower and lower level testing until I found the rotation quirk. Finally, I wrapped all animation code into a `BonkAnimator` object; each animated game object (currently just `Bullets`) keeps a handle to a `BonkAnimator` to do all animation tasks (and to keep animation data in one known place, so that

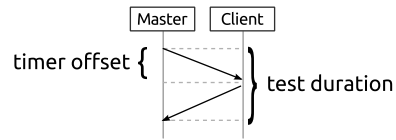


Figure 5: Timer synchronization algorithm

it can be updated later).

5.2 Network Communication

Testing any kind of real network communication was difficult. The device given to me would not accept my .apk files, and running two simulators on my laptop brought all UI responsiveness to a halt. In even simple tests, the Android emulator would not deliver communication between two simulated devices.

Instead, I chose to implement everything up to the network boundary, and then simulate two players' screens on one device. In the layout for `PlayGameActivity`, I created two separate `Playground` views – one to show the direct response of the game view and one to show what has been sent and received through the `NetworkCommunicator` objects. As Figure 2 shows, this second view is effective in illustrating the network lag between the initial event of a user pushing a button to fire a bullet and that bullet appearing on the screen of another player.

This is exactly why we had decided to use the time synchronization system explained in our design document, and pictured in Figure 5. Because network communication was never done, this synchronization could not be tested. The next planned stage of development in this area was to experiment with a “general” time offset which could be shown in the networked game play view. Instead of setting an offset based on actual measured network lag, the offset would be set to zero because each playground is using the same system clock by which to synchronize its animations.

Because the actual Java UDP libraries were being called to send and receive network communication, these tests were very close to actual behavior, in the sense that almost no code would be changed

if given a test environment in which devices could communicate with each other.

5.3 Collision Detection

Initially we had planned on simplifying the task of detecting collisions by using a 10×10 grid (or possibly 16×16 , for a total of 256 cells) to group objects on the playground. Any time two objects occupied the same cell, they would be considered to have collided. But after digging into this a bit, I realized that the cell does simplify the numerical units involved, but does not solve the inherently hard task of detecting collisions between many objects in an environment.

I had originally planned on using an *a-priori* ordered bounding-box algorithm for detecting collisions, such that the device whose user fires a bullet does an initial projection of whether (and when and where) the bullet would hit anything. The reason for doing a-priori calculations is that, during the construction of the `AnimatorSet`, many trajectory calculations and predictions are already being made; intuitively, I think that the additional work of collision detection should be less at this time than at any other time.

Over time, I went back to the 16×16 cell idea, but with one improvement. If using just a single grid of cells, collisions could be missed if two objects were close enough to collide but the objects' centers were actually in neighboring cells. A potential improvement to this would be to use two overlapping grids of cells, with a half-cell offset in each direction. That way, if two objects' positions straddled a cell boundary in the first grid, then they should be in the same cell the second grid. But still, a situation may occur in which two objects were in opposing corners of the combined grids, as in part (a) of Figure 6. The simple solution is to add a third grid: in part (b), we see that this leaves no possibility of objects colliding without sharing at least one cell of one of the three grids. (More abstractly, we would need $d + 1$ grids, where d is the number of dimensions we are dealing with.)

Now we can see how the cells pay off in size: my collision detection algorithm would pre-emptively

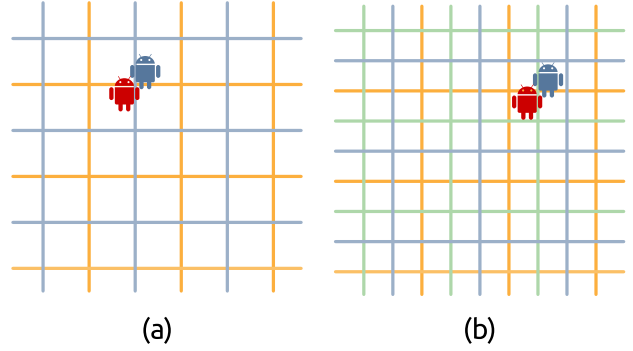


Figure 6: Collision detection using (a) two or (b) three overlaid grids

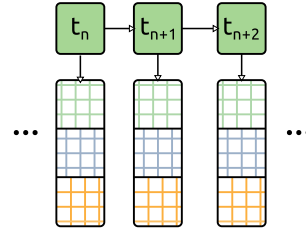


Figure 7: Collision Registry data structure

log every object at every time interval in a collision registry queue (Figure 7), with one element for every s -second time interval. Each time interval would be linked to three 16×16 grids; because each grid has 256 cells, a one-dimensional 256-item array can be used. Each item in the array is initialized at null (indicating no objects or collisions in that cell at that time), and as the objects are animated, they are registered in all appropriate cells at all appropriate times. If there is already an object in a cell being set, then that object and the one being animated are both added to a new `Collision` object, and this object replaces the original game object that had been there. The `Collision` object would also then be added to the time node for quicker access when the *collision handling* thread reads it. If a third object collides with them, it can simply be added to the `Collision` object's list of colliding objects at that time.

Once the collision registry has been populated, the time intervals are popped off the front of the queue by a separate *collision handling* thread, and if any collisions are found, it removes the colliding objects from the playground and creates a new collision or explosion animation. Also, col-

lision events are communicated to the other devices, and if the collision results in the end of the game by a bullet hitting the target, the other devices will detect this automatically.

This queue would have time-indexed root nodes pushed on and popped off as needed. It would only need to be large enough to store the longest-living moving object – probably around five seconds.

Using the default time interval of 0.1 seconds (this is the interval used by the game timer), a five-second time period would require creating $5 \times 10 = 50$ time-indexed entries for a total of $50 \times 3 \times 255 = 38,250$ individual cells. Because these are arrays the memory must be pre-allocated even if they are storing mostly null values. In a four-byte addressing system, this data structure at its largest would use a total of $38,250 \times 4 = 153,000$ bytes of object pointers.

6 Conclusion

I really enjoyed doing this project and found many challenges I did not expect. I only wish that I had had time to work on it more!