Ulf Benjaminsson, 2017-06-02 (project completed 170411, available from Github:
https://github.com/ulfben/PlatformerMK3)

Introduction:

Platformer MK3[1] is a demo for a tile-based 2D platforming game engine. It's not a complete game (there are no win- or lose conditions), but rather a demonstration to help other developers start making native games on Android.

Specifically, the demo demonstrates;

1. How to get real full-screen mode (immersive mode) with as much backwards compatibility as possible. (MainActivity.java - hideSystemUI())[2]
2. How to support multiple- and *simultaneous* input methods. In the demo I include
    a. virtual joystick (on-screen) controls
    b. physical gamepad controls
    c. accelerometer controls (toggle on/off in the pause menu)
3. How to deal with density- and resolution by
    a. mapping in-game coordinates to pixels (Viewport.java)
    b. scaling bitmap assets to match arbitrary densities (BitmapUtils.java)[3]
    c. using the hardware scaler to fill the screen when rendering to a fixed size framebuffer[4]
4. Using fragments as a crude state machine (main menu fragment, game fragment)
5. Using common drawables to create a very basic Dialog "system" (ExitDialog, PauseDialog)
6. Using the SoundPool to play sound effects, including support of the deprecated interface.
7. Using MediaPlayer to play background music .

Safety and ethical considerations:
This app does make use of persistent storage for a few configuration values; music on/off, sound effects on/off and accelerometer on/off - but no personal data is ever stored or asked for. It makes no use of networks, camera or other sensors. The manifest dutifully declares all the features that the app might use so the user can opt-in to each of them. Notably, the only required feature is the touchscreen.

I am targeting minsdk 16 because that's the API where Google added support for checking for changes in the game controllers (connecting and disconnecting). You can do manual polling to emulate similar features but I'm pretty happy with supporting 16 - it excludes less than 1.8% of the Android install base[5].

---

[1] MK3 stands for Mark 3 - this hand-in is the third iteration of my "engine". **MK1** was a very basic space shooter to get me acquainted with simple rendering and sound on Android. **MK2** was the bare-bones proof of concept of the current "game" - a platformer loading tile based level layouts, using virtual coordinates, basic physics etc. **MK3 takes MK2 but makes it a proper app fit for Android**; supporting the life cycle, using fragments for application state, using many more APIs to implement features (gamepad, mediaplayer, sensors, bitmapfactory, persistent preferences etc), provides a GUI etc.
[2] https://developer.android.com/training/system-ui/immersive.html
[3] https://developer.android.com/topic/performance/graphics/load-bitmap.html
[4] recommended to increase performance and lower energy use:
https://android-developers.googleblog.com/2013/09/using-hardware-scaler-for-performance.html
[5] https://developer.android.com/about/dashboards/index.html

Utveckling av Mobila Applikationer VT2016
Inlämningsuppgift 3 - Valfri Applikation
Ulf Benjaminsson, 2017-06-02 (project completed 170411, available from Github:
https://github.com/ulfben/PlatformerMK3)

Target grade:
I am targeting a pass with distinction (VG) but I am aware that I am over a year late with the assignment and I don't want to waste anyones time - so whatever you feel is fair. The project was completed in April but unfortunately I couldn't make the time to write the report until the spring semester was over. Thanks for your patience!

Usage:

When launching the app you'll be greeted by a start menu, offering you to launch the game or to exit the app. No processing is going on here so the app launches instantaneously.

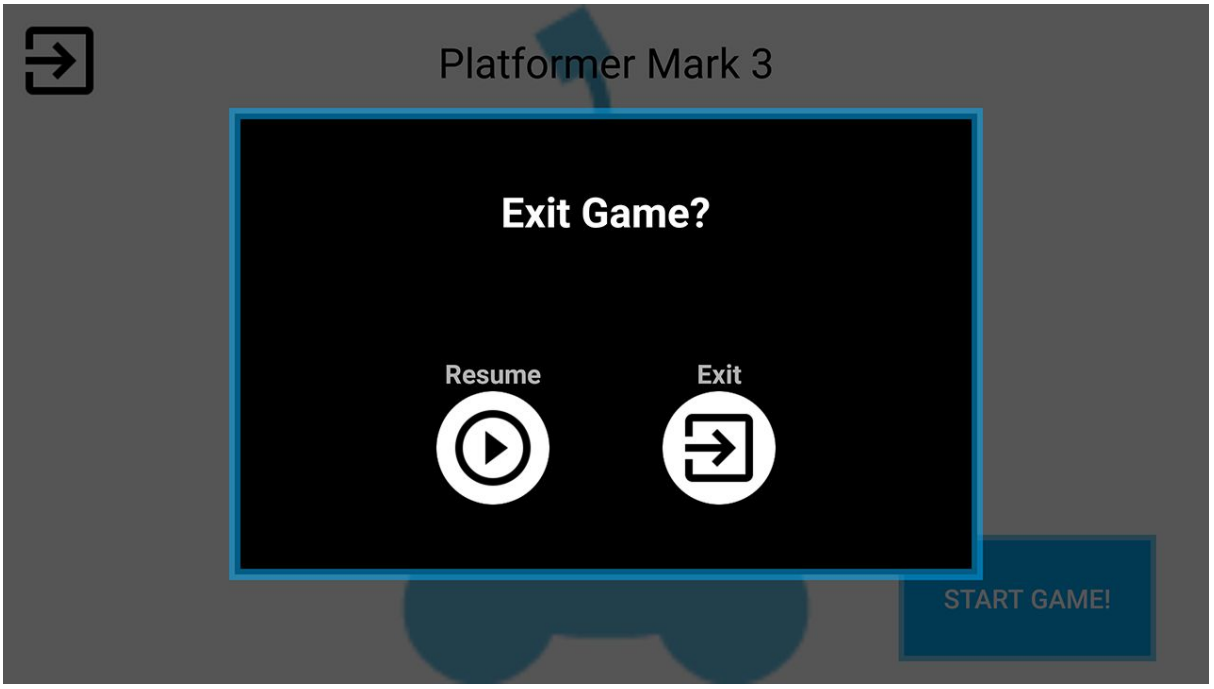
After pressing Start Game you are immediately dropped into the demo level and can start moving around. Both the virtual joystick, gamepads and accelerometer controls are active at the same time so you can either rotate the device or press a finger on the left side of the screen to start moving. If connecting a gamepad the game will pause and show you a help-text.

The game world has a few different entities; coins that you can collect, animated spears (they do no harm, but simply exercise the physics and animation system), "walkers" that simply glide back-and-forth, etc. Feel free to jump around and explore, and try to break the engine.

Press the back-key to pause the game or exit any dialogue. Exiting the app requires confirmation by pressing the Exit-button.

Screenshots / app walkthrough:

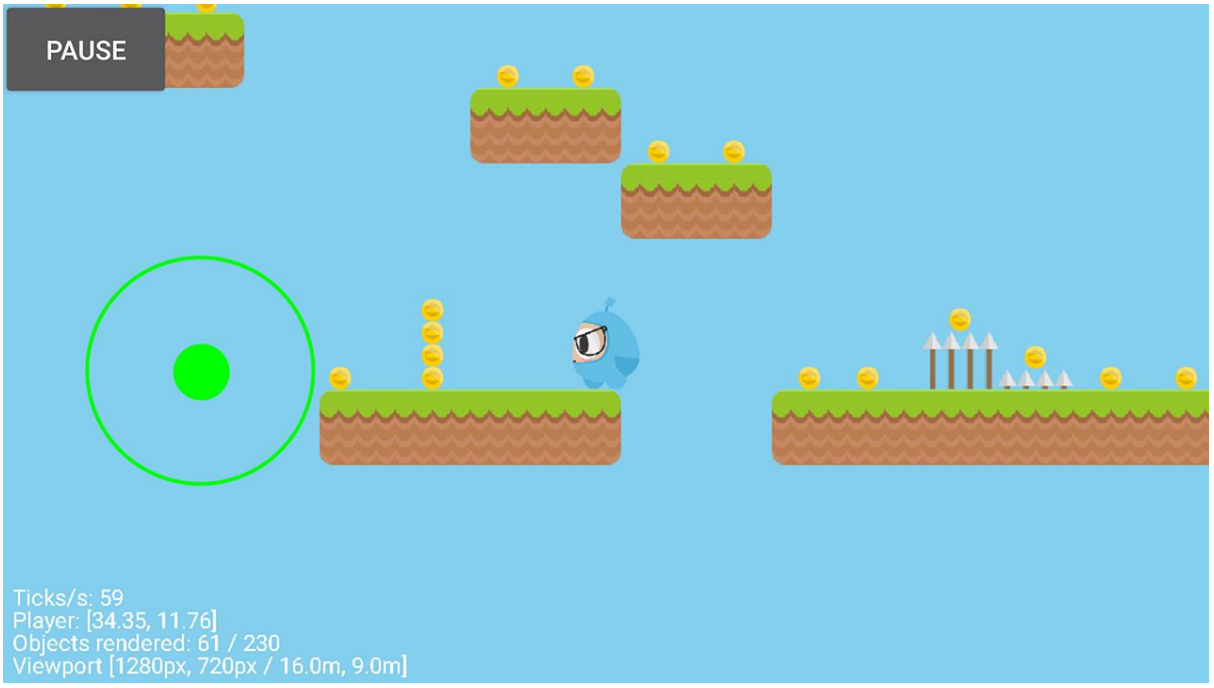
The start menu is a fragment (`MainMenuFragment.java`) which offers 2 customized buttons; one to start the game, and one to open the `ExitDialog` to confirm closing the app. The `Activity` which hosts the entire application is listening for Gamepad connections, and will pop-up an alert when a useable device connects or disconnect, whether the user is in the menu-fragment or the game-fragment.



The `ExitDialog` is simply two `Drawables` - one grey overlay and the dialog box. Some developers[6][7] argue that it's an anti-feature to ask for confirmation before exiting and I agree in the general case. But for apps that takes a long time to load - such as many games - I feel

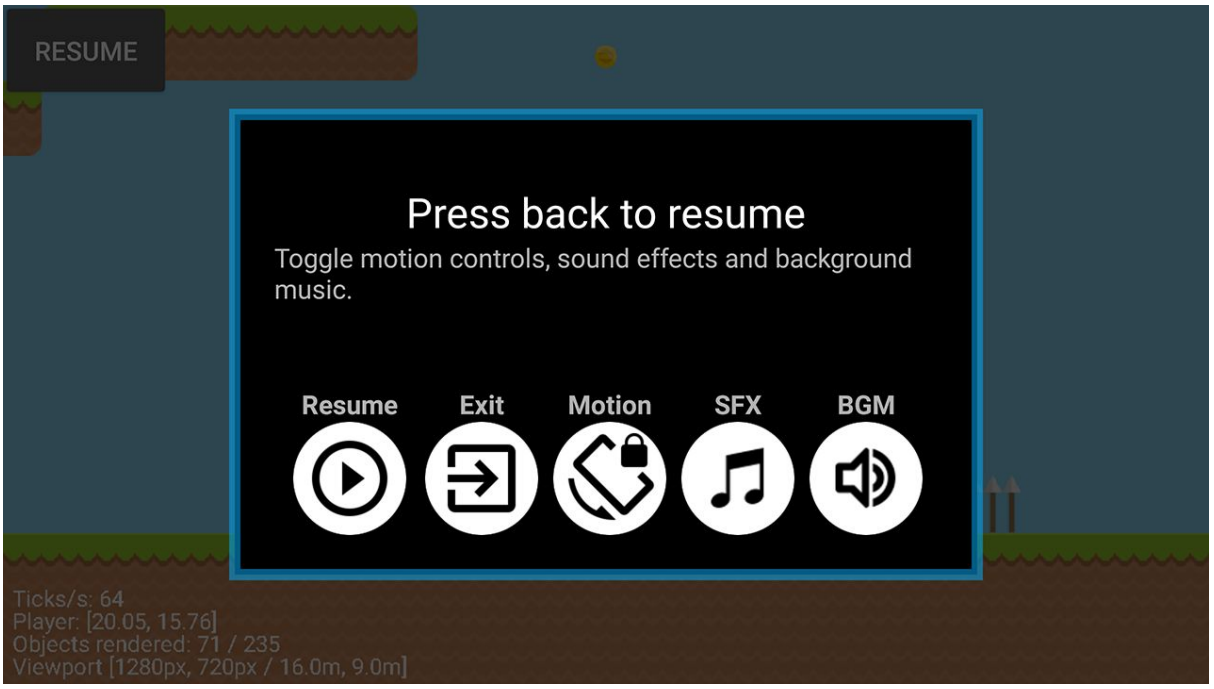
[6] https://plus.google.com/+CyrilMottier/posts/EiXqUDrr6jT
[7] http://www.androiddesignpatterns.com/2012/08/exit-application-dialogs-are-evil-dont.html

the extra safety net is motivated. For example, loading up a game like Hitman Sniper takes 10-15 seconds even on a high-end device, so you'd really want to make sure that the user intended to unload the app. My little demo is whip-fast to launch, but I wanted to try and implement the confirm-on-exit feature. It should be noted that there is always a way for the user to immediately get out of the app, by using the Home button.



This in-game view currently displays the virtual joystick. The entire left side of the screen is a joystick-region, and the entire right side is the jump-button. The debug text output provides some useful information: the number of frames per second, the player position (in meters), number of objects rendered (out of the total existing in the world) and the viewport configuration. The viewport output tells us that the framebuffer is 1280x720px (¼ of my device's native resolution!) and that our field of view is 16 meters wide, and 9 meters tall. If you were to enable app rotation, the view would will still be 9 meters tall but the visible width of the world would be more like 4-5 meters.

Not seen in screenshot: the spears and the player are both animated.



The `PauseDialog` is pretty much the same as the `ExitDialog`. I am particularly happy with managing focus in these dialogs. If you are using a physical gamepad you can navigate between the buttons and activate them without touching the screen. The buttons even make use of the "state list drawable" feature to highlight properly when focused or interacted with (see icon_button_bg.xml for an example).

Construction

In MK1 and MK2 of this this engine I used two Activities - one for the start menu and one for the game. MK3 uses a single Activity that hosts fragments instead - one fragment for the main menu and one for the game. There is no significant data passed between these states so it doesn't make any practical difference to the app which way you go here.

The core logic of the game itself is relatively simple. The GameEngine owns a list of GameObjects and a thread[8]. The thread is controlled by the Activity lifecycle events. When active the thread simply executes the "core loop" over and over again. This core loop does:

- "update" the input (check state of buttons, sensors etc)
- update the camera (viewport)
- update all game objects in the list
- collision checking and simple physics of all game objects
- render (draw) all visible game objects

The core loop sleeps when a dialogue is displayed or the app loses focus.

[8] I did experiment with multi-threading (one for update, one to render), but disabled this feature in the delivered project. Much of the code is still in there though.

Incomplete or missing features

Action bar
The assignment instructions asks for us to implement an action bar, but my app is a full screen game and I spent considerable effort researching how to properly make it full-screen - including backwards-compatible "immersive mode". I hope that's an okay trade-off. The pause dialog gives access to the few runtime settings the game provides; such as toggling audio and accelerometer controls.

Multi-threading
When looking through the code you will come across many signs of my late-abandoned multi-threading attempt - many synchronized regions and both the RenderThread and UpdateThread are still in the source tree, despite UpdateThread being the only one in use in the current version. My multi-threading *did* work and gave massive performance improvements (massive, as in from 50-60 ticks per second, to over 2000 ticks per second in the virtual device. many hundred ticks per second on the phone), but it also caused minor visual artefacts. Since performance was a non-issue (I get a solid 60fps on most devices) I disabled threading in the final version, rather than spend time implementing some kind of render-state-que[9].

App rotation
The game is locked to landscape mode but works perfectly fine in portrait mode. The reason for disabling rotation was simply to avoid having to serialize the game state. If you enable rotation (setRequestedOrientation in MainActivity.java, and android:screenOrientation in the manifest) you get a good demonstration of how flexible the Viewport is. It is currently set to always display 9 meters of the game world (on the y-axis), and will always fill the screen correctly no matter the orientation. However, since the app is re-created on rotation the game state will be reset.

Maximize use of the resource system
Android has a wonderful system for moving values out of code - using the XML resource systems. I use it for the GUI and non-debug strings. Ideally I would keep all gameplay settings and configuration in the resource files as well - not having to recompile to test various settings. Currently I have all settings in named and typed constants, which is the second best thing. No magic values allowed. :)

In-elegant animation system
The biggest unsolved problem for me was how to implement animation in my GameObjects. My class hierarchy goes: `GameObject -> DynamicGameObject -> InputGameObject`. Where the base `GameObject` is any visual object with position and dimension, but it doesn't move. The `Dynamic` child class extend the `update()` function to implement basic physics. The `Input` grandchild extends that again, to allow player input to affect the physics.

---

[9] such as http://blog.slapware.eu/game-engine/programming/multithreaded-renderloop-part1/

Animation is an orthogonal feature in that lineup. Any of them **could** be animated, but most of them wouldn't be. So ideally this would be solved by composition, so that the "visual representation" of an object is handled by it's own class - a `Drawable`. Unfortunately the Animation and it's owner needs to communicate quite a lot. Unlike a plain `BitmapDrawable`, the `Animation` needs;

1. to be updated and given the current delta time
2. to be stopped and started
3. playback speed controls (think; synching a walk-cycle to the physics)
4. (preferably be able to emit `GameEvents`, for sound effects and such)

That interface is a lot more involved than the `BitmapDrawable`, requiring substantial rewiring of the `GameObject`. If I keep working on the project, this will be my next task.

I should add too, that my `Animation` class make use of Android's `AnimatedDrawable`, despite it providing almost no useful features. It gives me a list of bitmaps and for how long each frame should be displayed. But I still need to scale each frame to fit my game object dimensions, I need to support flipping the animation (which means manually flipping each frame), I need to support measuring each frame, and so on. My `Animation.java` is a bit of a hack, is what I'm saying, and should probably be rewritten to make use of a well packed sprite sheet instead of `AnimatedDrawable` assets.

Credits:

The tracks for the background music are all courtesy of Eric Skiff, used under a Creative Commons Attribution license. The track names are **Prologue**, **All of Us** and **Chibi Ninja.** Are all available from: https://soundcloud.com/eric-skiff/sets/resistor-anthems and http://ericskiff.com/music/

All sprite graphics are courtesy 1001.com, also under a Creative Commons Attribution license. The full sprite set is available at: http://opengameart.org/content/platform-pack

The sound effects have been created by me, using the BFXR Sound Effect Generator.

Icons are from Android's Material icon set, used under the Apache License Version 2.0. The full Material icon set is available at: https://material.io/icons/