LK-Tris:
A embedded game on a phone

from Michael Zimmermann

Index

1) Project Goals

Every project needs a goal. The goal of this project is to get a look at how embedded software works and which problems can occur. Also this might be interesting to understand how the hardware works and how software rendering engines actually work.

1.1) Must-Haves

- Framebuffer-Access(drawing pixels)
- Key-Input
- Working basic version of TETRIS game

1.2) Nice-to-Haves

- Highscores
- Run the game on Linux computers with the same code

1.3) What I realized

I realized all Must-Haves and the possibility to run the game on any Linux computer with the same code, by using compiler directives. I call this pseudo emulator.

2) What is embedded Software?

A software is embedded if it's executed directly on the hardware - without a underlying operating system or any other type of abstraction or interpretation.
This means, that by default there isn't any memory separation and the concept of processes just doesn't exist. All parts of the application has full read and write access to the whole system memory.

2.1) Little Kernel (LK)

Little Kernel is a project founded by Travis Geiselbrecht which aims to provide a small and solid base for embedded applications.
It tries to make development as simple as possible and keep the code small so it can run on devices with less resources, too.
LK gives you a very basic environment by initializing the most important hardware like the MMU, UART, timers and interrupts and sets up useful techniques like stack, cache and exception handling.

The company Qualcomm uses LK as base for their Android Bootloader on all their ARM chipsets - which are very common for a huge amount of smartphones and tablets.
Qualcomm added support for some hardware parts to LK which are very important for realizing this project. The most important ones are display, buttons, and flash memory.
This gives us all we need to develop a game on top of LK. Fortunately Qualcomm published the sourcecode of their changes.

3) Hardware

Since Qualcomm devices are very widespread and LK has already been ported to these, using such a device makes it much easier to realize this project.
I used two phones for testing and development. The “Xiaomi MI2” and the “Motorola Moto E”.
Furthermore, I wrote a pseudo emulator to be able to run the game on my computer.
Pseudo means, that it doesn’t really emulator the hardware. Instead it set’s up a window using the SDL graphics library and compiles the game code itself using customized C header files.

4) Sideloading LK

Since LK is already installed on our devices one could think you could easily replace it with the game. But this is a bad idea for several reasons:

1) On most devices the bootloader is signed, which means that you need the manufacturer’s private key to be able to execute your own version of LK. For security reasons the manufacturer will never publish this key.

2) Qualcomm does not publish all of the changes they applied to LK - only the most basic ones. Also do the device manufacturers(like Motorola or Xiaomi) need to add some changes for hardware specific support like display panels. This means that many important parts of the hardware would not work with the published version of Qualcomm’s LK.
3) The Android bootloader is the first point in the whole boot process of the device where you can simply install our own software do the device. If you replace it with a broken version it gets very complicated to restore the device.

3.1) Android Boot Image

Since we know, that replacing the existing LK is not a good idea, we take a look at how the Linux kernel gets booted on these devices.
The bootloader supports specially packed “Android Boot Images” only. These files contain the kernel, a ramdisk(root filesystem) and a few other things.
Among other information the file header of these images includes the sizes of all packaged files and the loading address of the kernel.

This header has a size of 0x8000 bytes followed by the kernel image. That's why the loading address needs to be set to "KERNEL_LOADING_ADDRESS - 0x8000" to get LK to the right place. As kernel Image you can use the raw binary generated during the compiled process of LK.
Also you need to use "/dev/null" for the non-optional ramdisk to get a ramdisk with a size of 0 bytes.

The Android boot image which was packaged this way now can be used to boot LK from the existing LK-based Android bootloader.

5) Display

Instead of investing time to write working display drivers for all target devices, the fact that the android bootloader initialized the screen already can be used as our advantage.
Once the memory address of the framebuffer is known you can easily write into this location to show pixels on the screen.

5.1) Framebuffer format

On both the pseudo emulator and the "Motorola Moto E" the color format of the framebuffer is BGR888. That means the all three colors have a size of 8 bit(0-254) and the order of the colors is Blue, Green and Read.
The "Xiaomi Mi2" uses RGB888. The pixels are the same size but the order of the color is different.

5.2) Double buffering

The android bootloader did not enable double buffering for the display so a temporary software buffer needs to be implemented to prevent screen tearing if the lcd refreshes while the application is drawing into the buffer.
This can be accomplished by allocating memory with exactly the same size as the framebuffer and using this to draw to. After drawing each frame the content of this buffer has to be copied t the real framebuffer then.

On the "Xiaomi Mi2" this software buffer mechanism is not needed. This devices doesn't have double buffering enabled but a command needs to be send to the LCD to trigger flushing the framebuffer to finally show it on the display.

5.2.1) Hurdle: Xiaomi Mi2's framebuffer trigger

This software trigger caused some problems when calling it regularly in short intervals. I could see weird effects on the screen which looked like the buffer was flushed before the frame has been finished drawing.
I worked around this problem by adding a small delay of 10 milliseconds after calling the software trigger because the display probably won't be finished flushing when the trigger will be called next time.
Also I disabled interrupts before calling the software trigger to prevent any influence by timers or other threads.

6) Game Engine

I didn't use a ready-to-use game engine for this game because there isn't any compatible with LK and because the goal of this project is to get some experience with the the hardware.
In the following I describe what my little engine provides and the steps I made to abstract the hardware.

6.1) Game Loop

The minimum requirement for a game is a update functions which gets called regularly to create/remove objects and to update their positions. Also a render method is needed because 2d isn't as abstracted as 3d is and there is no engine which automatically draws objects created and registered in a special way.

In my first implementation I just called the update function followed by the render function in each iteration of the game loop. Since rendering need way more time than updating and it can be important to update more often than to render needed to parallelize these two functions.

6.1.1 Timers

So I used timers to parallelize updating and rendering into two separated loops which have different update frequencies. This worked great at lower framerates, but once I needed higher rates for a smoother gaming experience it started being as slow as the non parallelized version of the game loop.
After a talk with the creator of "Little Kernel" I found out, that timers are called by interrupts which can't run the same time. That means the the two game loops blocked each other until they were finished.

6.1.2 Threads

Unlike Timers, threads can run the same time without blocking others.
So I just started two threads - one for the update and for the render loop - to finally parallelize rendering and updating. What's important here is, that the endless loops inside the frames need to limit the iteration frequency to not keep the cpu at 100% usage all time.

After some testing I found 15 FPS for rendering and 20 FPS for updating to work pretty well for my implementation of the game tetris.

6.1.3 Delta Calculation

A small but very important thing is the calculation of the time between the last and the current frame. The result has to be passed to the update loop so the game can update the object position depending on the current time. If one frame takes a little bit more time or the engine changes the frame rate the game still runs at the same speed if it uses the delta time to calculate the nes object positions.

6.2) Framebuffer Abstraction

Abstracting the framebuffer into easy to use functions which let you set the color of pixels and use x/y coordinates simplifies drawing a lot.

6.2.1) Pixel Indexing

In the framebuffer all pixels are in one row and x/y coordinates can't be used to address them. Also the length and format of every pixel can vary on different devices.
To solve the second part I wrote a function which needs the frame buffer, a pixel index and a color in RGB format as arguments and set's the colors of the correct subpixels with these information.
To find the address of a pixel the formula "framebuffer_start + pixel_index * bytes_per_pixel" can be used.

6.2.2) X/Y coordinates

To be able to use X/Y coordinates instead of linear pixel indexing I wrote a function which needs the framebuffer, x/y coordinates and a color as arguments and set's the color of the correct pixel using the function I described above. It uses the formula "y*display_width + x" to map the coordinates to a index.

6.2.3) Hurdle: Rendering Speed

I noticed that the rendering speed even of one single frame takes very much time. Filling the buffer with black pixels via the frame buffer abstraction API took around 450ms.
The problem was, that for every pixel, the function “fbSetPixelXY” needs to be called which by itself calls “fbSetPixel” then.

In C, to call a function, the CPU has to backup all registers on the stack, and restore them when the function returned. This takes a lot of time if you need to do it for every single pixel on the screen(two times!).

I did not want to give up the abstraction because the game code would get very messy then. So I looked into available code optimization flags the GCC compiler offers. The default optimization level of LK is O2 which optimizes minor things without risky changes to the code. In kernels and bootloaders O2 is the default level because there’s no need to use the more aggressive optimization level O3 and risk broken code.

For a game O3 can be very interesting though. The most important optimization flag enabled by O3 is “finline-functions”. This one merges multiple functions into one to remove the need of backing up and restoring the stack. This speeds up applications a lot but also increases the code size.

After enabling this flag the rendering time of the same frame was decreased from 450ms to less than 10ms. Less, because the precision of the system clock is 10ms and I got 0 as a result.

6.3) Input Events

Qualcomm’s implementation allows us to read the status of the volume keys and the power key. This status needs to be checked in every iteration of the game loop though, because we don’t receive interrupts if they changed.

For this to work I created a small keymap which uses key codes as the index and 1 or 0 as values to indicate if the key currently is pressed or not.
As soon as the game read the key it will be zeroed to prevent indicating one keypress as multiple. Also there has to be a limit how often a key will be set to true, otherwise a keypress of 500ms would be treated as 10 if the update loop gets called every 50ms.

6.4) Grid

In a game like tetris it can be very helpful to use a grid-based object positioning system.
That's why I implemented functions to set the grid size and and fill a specific tile in the grid with a color.

Since this "engine" doesn't need to be common and needs to work for tetris only - where every grid tile is a place for a stone - I could use the same function to make these stones more pretty by adding a 3D effect.

This works by first drawing a black border, then drawing a inner border on the top and right side with a brighter version of the color and finally a border at the bottom and left side with a darker version of the color.

6.5) Font

To display text, a font is needed. Because complicated libraries are needed to render vector fonts - and it probably would cost too many resources - I decided to use a bitmap font.
I generated these fonts with a gimp plugin and then exported the resulting bitmap as a C header file which then can be used in my code. **[1]**

I wrote simple functions to abstract printing text on the screen which work exactly the same way as the standard input/output functions in the libc library. I created functions like "putc", "puts", and "printf" which take additional arguments for the framebuffer and the position.

Additionally I wrote a function called "printf_centered" which does the same as printf but aligns the text horizontally centered. This is very useful for the the main menu.

6.5.1) Hurdle: Booting on "Moto E"

The bitmap gimp generates needs to be converted on runtime into a rw format with a simple loop. For some reason the CPU of the Moto E stopped as soon as I tried to write the pixels to the allocated memory.

After a lot of time spent in testing I noticed, that the bootloader forces the loading address of the Android boot image to 0x0 and completely ignores the loading address set in the header.
That shouldn't be a problem though because LK relocates itself to the right position in memory after loading. Otherwise I would have never been able to run LK at all.

I did not investigate further into the problem because I don't on low level debugging connections like UART or JTAG on this device and it probably would cost more time than I have for this project.

6.6) Transparent Color

Instead of supporting alpha transparency which could cost some resources which aren't available I chose to use a color which I never draw and will appear as 100% transparent. This transparent color is used for fonts and grid tiles only and doesn't cause any problems this way. I chose black as the transparent color because both the font background and the map background are black

6.7) States

States are very important in games to be able to easily switch between several screens. The states I used are "menu", "ingame", and "gameover".
At the beginning I render the state menu, once the user has pressed a key, the state will be switched to "ingame", which renders the tetris game itself. once the user lost the game, the "gameover" screen will be shown.

I did not put the code for adding and switching states into the main engine code where the game loops are, because these files basically abstract the loop itself only so I can run the game on both LK and the pseudo emulator. Instead I put the code into the file with the init, update and render loops and created a separate file for every state which implement callback functions for handling state switches and updating and rendering.

6.8) Tetris Stones

Our tiles are used as stones, but there's much more needed to combine and rotate them, to let them fall down and to check the collision.

6.8.1) Hurdle: Random Stone

There are many different algorithms to improve randomness in games. Most of them rely on the functions srand() to set a seed and rand() to get the next "random" number though. **[2]**

That worked pretty good on the pseudo emulator but on the real hardware I always got the same stones. The problem was that, the time in nanoseconds is used for the seed, but the timer isn't accurate enough.
So I chose to only use the very last number of the timer as the random number. This seemed to be random enough for the algorithm to pick a random stone.

6.8.2) Collision Check

The advantage is, that I used a tile based map layout. This way it's very easy to check the collision by just checking if there is another stone at the coordinate where current stone is. Also some default checks like positions outside the map need to be checked.

6.8.3) Stone Rotation

Since the goal of this project is not to learn something about math, but to learn about embedded systems and performance optimization I used the best ready-to-use rotation algorithm I could find. **[3]**

What's important to note, is that I did not choose a in-place rotation, where the source array will be modified because I need to make collision checks after the rotation to check if this is possible or if I have to deny doing that.

7) Tetris rules

Tetris itself isn't that important for this project - it's just a example game to demonstrate the techniques. I think it's important to explain the basic rules here though.

7.1) Basics

Tetris is a game where you have one stone - consisting of multiple single stones - falling down the map. It's possible to move it to the left or to the right and to rotate it. In some implementations you also can speed up the falling process.

Once the stone can't continue falling down because it reached the floor or another stone, it will stay at it's position and a new stone gets spawned into the map.
If a row of the map is full of stones it will be cleared and all above stones will fall down one row.

7.2) Score

You get 100 points for removing one row, and 800 for removing four rows at once - which is called tetris. Back-to-back tetris clears are rewarded with 1200 points.

7.3) Levels

After reaching specific scores you go one level up. With every increase in the level the speed of the falling stone will be increased, too to make it harder to put the stones into the right place.

7.4) Goal

The goal is to keep the map clean as long as possible and get as many points as you can.

8) Game Performance

The performance of the game is very different among devices. The best results were shown off on the “Xiaomi MI2”. I measured up to 25FPS when the map is full of stones.

The performance of the “Motorola Moto E” on the other hand is very disappointing. This device can render 6-8 FPS only.

The first reason you may come through when searching for the problem may be the huge difference in Hardware. Xiaomi Mi2 has a Quad Core processor running at 1.5GHz while the Moto E has a Dual Core processor running at 1.2GHz only.

This cannot be the main cause though because LK isn’t capable of using multiple CPU cores. After comparing speed with the Linux kernel clocked at all common CPU frequencies I came to the conclusion that the CPU definitely isn’t clocked at full speed during boot up.
The linux kernel source code of the Moto E proves that, because they ramp up the cpu speed to increase boot speed. **[4]**

Summarized that means that the CPU of the Moto E doesn’t run at it’s full speed for some reason. It would be possible to set the clock speed from LK but this would require investing time into porting the clock driver over to LK and change the rate than. Obviously that exceeds the limits of this project.

9) Conclusion

I came through many hurdles like bringing up the hardware and optimizing performance.
I was able to see, that there’s much work needed to optimize a embedded environment - especially with a game (engine) - to the maximum.

After my work on this project I researched a little bit about software rendering and came across a library named “libpixelflinger”. This library directly uses the framebuffer like my project does, and emulates opengl on top of it.
Google uses this library for the Android emulator, and has been optimized very good to provide maximum performance.
They achieve this by using a pixel pipeline, and many assembler optimizations for specific processors.

That means that I got a great look into the way how embedded systems work but there still is very much space left for optimization.

10) References

[1] https://github.com/sole/snippets/blob/master/gimp/generate_bitmap_font/sole_generate_bitmap_font.py

[2] http://stackoverflow.com/questions/2509679/how-to-generate-a-random-number-from-within-a-range

[3] http://stackoverflow.com/questions/42519/how-do-you-rotate-a-two-dimensional-array

[4] https://github.com/MotorolaMobilityLLC/kernel-msm/blob/kitkat-mr1-rel-condor/arch/arm/mach-msm/acpuclock-cortex.c#L387