



Mobile devices as development platform in Broken Age

Oliver Franzke

Lead Programmer, Double Fine Productions



p1xelcoder

GAME DEVELOPERS CONFERENCE
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

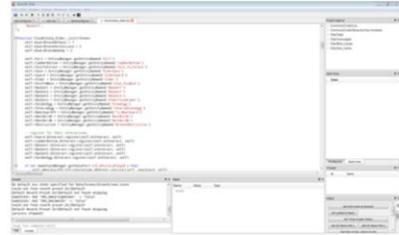
This talk is about how we integrated iOS and Android devices into our development pipeline at Double Fine, so that they can be used not only by engine coders but also by game-play programmers and even artists.



What makes a dev platform?

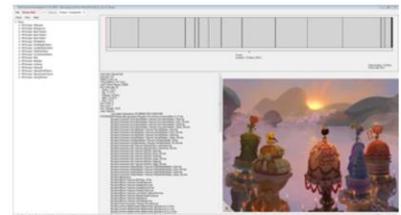
- Fast iteration time

- Content 
- Native code 



- Debuggability

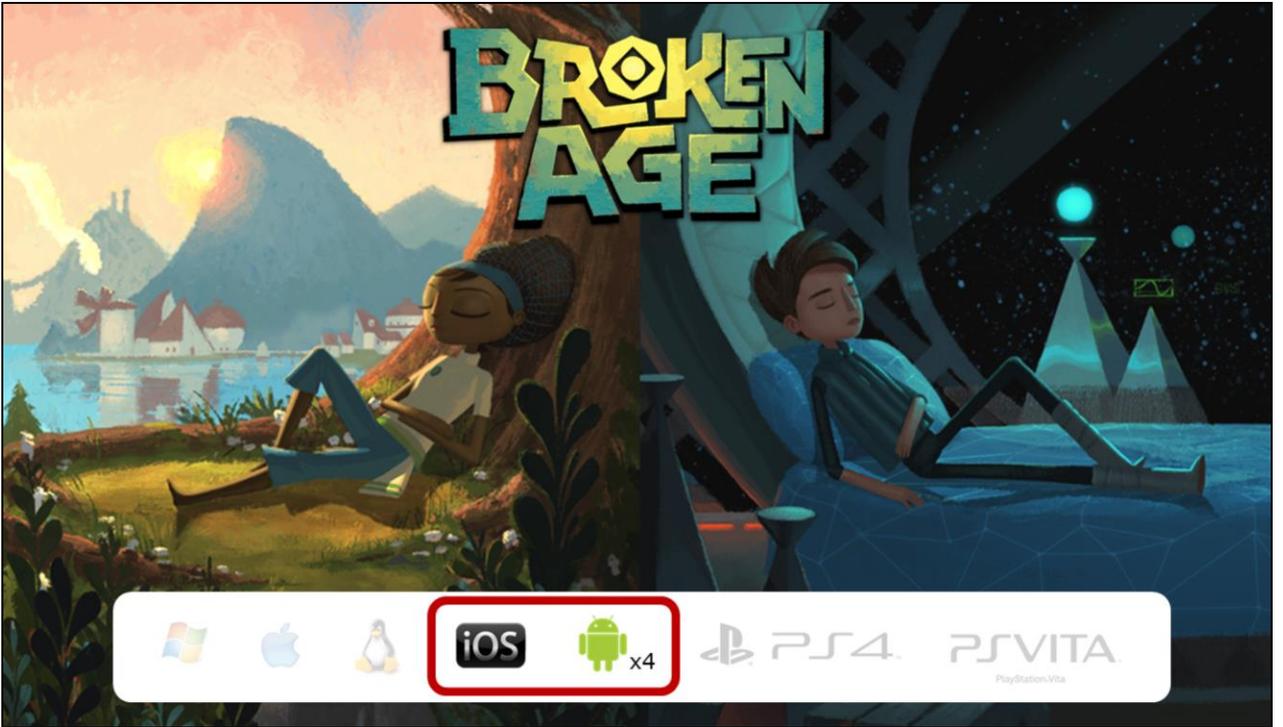
- Game-play script 
- Native code
- Graphics 



In my opinion a development platform is categorized by two major factors: Fast iteration time and debuggability.

Being able to iterate very quickly on code and game assets is paramount, which is why this presentation will focus on this aspect.

Debuggability helps to identify what happened if things go wrong.





Developing BA on mobile devices

- Some of the stuff we did on device
 - Game-play programmer wrote touch controls
 - VFX artist checked and optimized effects
 - Fix shader bugs

Considering the scope of Broken Age and the amount of target platforms the development team was tiny.

We only had one systems (or engine) programmer assigned to the project, which was me. I knew from the beginning that I won't have the time to handle the mobile devices by myself, so making sure that mobile devices can be used by other team members was critical. I think this work definitely paid off.

Broken Age is a point and click adventure game and we wanted to make sure that the game plays just as nice using touch controls. A game-play programmer was able to tune the input controls directly on device.

In addition to that our VFX artist worked directly on device to find areas with performance issues and tweak effects in order to achieve the target frame rate.

There were also a bunch of shader bugs that only showed up on specific mobile GPUs which we fixed pretty quickly by tweaking and hot reloading the problematic shaders.



Double Fine development pipeline

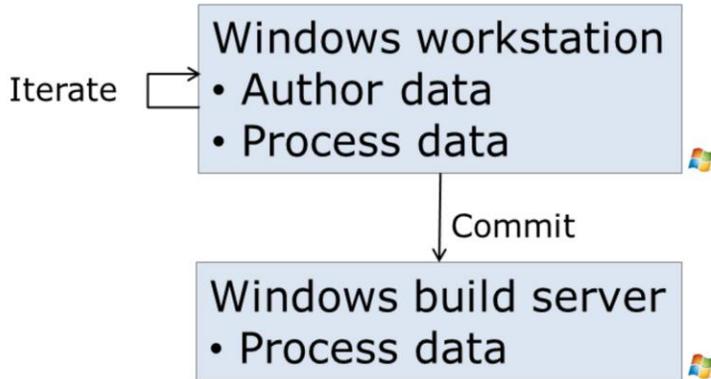
- Target platform:   
 - Code: compiled on target platform
 - Data: shared, authored on Windows
- Same hardware
 - Performance characteristics similar
 - No per-platform content

At Double Fine we traditionally use Windows as our main development platform. All of our content creation tools are running on Windows and we use Visual Studio as our primary IDE.

This changed a bit when we started to ship games on OSX and Linux, since we had to build and debug the native code differently. The data remained the same though, since all three operating systems (Windows, OSX and Linux) are PCs in the end of the day. In other words we didn't need to create and special assets and therefore we didn't need to port any tools to OSX and Linux.



Content workflow





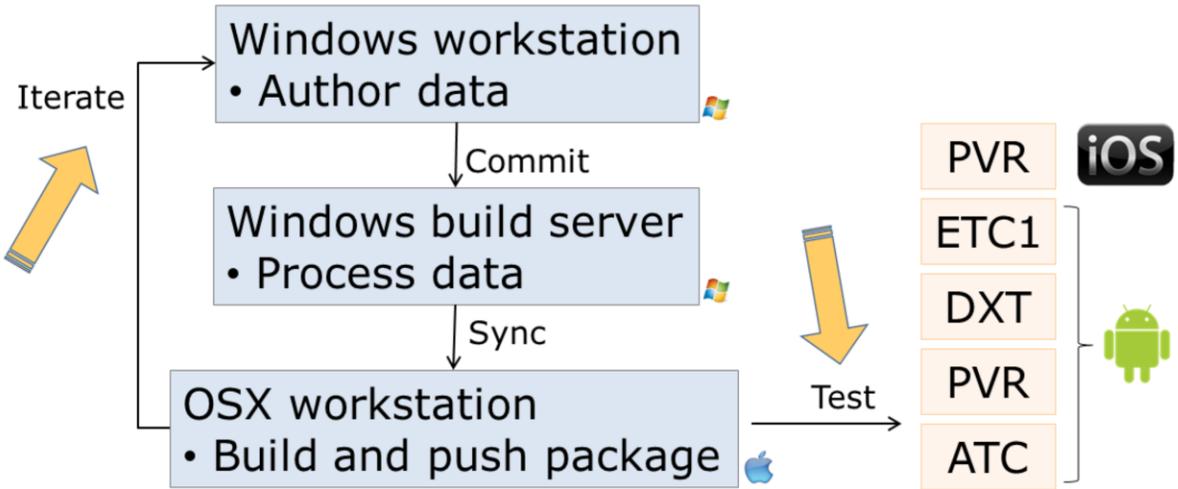
Double Fine development pipeline

- Target platform:  
 - Code: compiled on OSX
 - Data: **not shared**, authored on Windows
- Hardware is very different
 - Performance varies greatly
 - Platform / GPU specific data necessary

Broken Age was our first big cross-platform project that had to run on iOS and various Android platforms. We used OSX to author and debug the native code, but the content was still authored and processed on Windows. Since mobile devices are very different from a hardware perspective this became a big problem for us. Porting the tools to work on OSX would have been too much work and so we had to come up with another solution to solve the asset tweaking issue.



Content workflow





Mobile as development platform

- Broken Age Act 1 package size: ~1.2GB
- Reboot time was ridiculous!
 -  ~10 minutes*
 -  8 – 21 minutes
- Infeasible for development!



* Recent Xcode update reduced reboot time to 2 – 3 minutes

By the time we shipped Broken Age Act 1 on iOS and OUYA the game was already around 1.2GB big.

This posed a huge problem for us though, since the turnaround time for any minor script, asset or code change was very long.

During big parts of the development it took 10 minutes on iOS to simply reboot the game after a minor change. The most recent version of Xcode (6) fixed some of these issues, but it still takes between 2 and 3 minutes to restart.

On Android things are even worse as it takes between 8 and 20 (!) minutes to re-launch the app after a tweak.

Act 2 roughly doubled the amount of content, which means that the turnaround time also doubles.

Obviously this makes it impossible to use mobile devices directly during production. In fact this was a big risk for the project since we ran into various shader bugs on different mobile devices.



Too slow

- What is going on? 
 - Relink and update staging data: 10+ seconds
 - Data sync and install: ~10 minutes
- Sync (comparison and copy) is bottleneck

We wanted to make mobile devices a regular development platform and so we started to analyze the problem.

On iOS the root of the issue is that the length of the data sync through Xcode depends directly on the number (and size) of files in the staging directory. It doesn't matter how many files were actually changed. Xcode data sync is conservative and doesn't remove files.

I should also mention that the data sync in the latest version of Xcode is much faster than the earlier version as it 'only' takes between 2 and 3 minutes now.



Incomprehensibly slow

- What is going on? 
 - Package build time: 4+ minutes
 - Data transfer and install: 4 - 17 minutes
- USB speed is bottleneck



On Android things get really bad. The APK (android package) is essentially a ZIP file that contains the code and all of the data used by the app. Even a minor change to a asset, script or source code file makes it necessary to re-create and re-install the package on the device.

The biggest problem here is the data transfer and the app install which can take between 4 and 17 minutes for a package of 1GB .

The transfer speed depends heavily on the USB speed supported by the device. The numbers on this slide were measured on a NVIDIA Shield (USB 2) and a Shield Tablet (USB 3).



Slow content update

- Same symptom but different cause
- Different solutions required
 -  Minimize APK size
 -  Work around Xcode data sync

The bottom line is that it's impossible to iterate on content in a meaningful way. Waiting 3 to 20 minutes until you can see the effects of a tweak is obviously ridiculous and has to be fixed in order to make mobile devices a development platform.

Even though both iOS and Android have the same problem the cause is different, so a unique solution is necessary for the two platforms.



Fast content update

- Minimal APK: No data, just code
 - Reduced package build time
 - Fast APK transfer and installation
- Deal with data separately
 - Only copy added or changed files

Let's look at Android first as it is much slower than iOS. The core observation is that building, copying and installing a package is extremely slow. The only way around this issue seems to be to reduce the amount of data in the package.

Since assets take up a majority of the package size we decided to remove them from the package during the development. The native and compiled Java code has to be part of the APK, so our minimal package contains only that.

Obviously the game still needs to load assets, so we need to handle it differently.



Fast content update

- Load assets from `sdcard`
 - Supported by (almost) all Android devices
 - Remap file location

```
bool RemapFilename( const char* filename, char* remapped ) {  
    #if _DEV  
        sprintf(remapped, "/sdcard/dfp/dfa/%s", filename);  
        return FileExists(remapped);  
    #else  
        return false;  
    #endif  
}
```

We decided to move all of our assets into a sub-folder of the sdcard directory, which is supported by all devices we encountered.

Data in the sdcard folder can be read and written by the app, so it's easy to load the game assets from there.

All IO in our engine is routed through a file manager anyway, so adding file remapping to the new location was trivial to add. The code on this slide shows our remap function and as you can see there really isn't much to it.



Fast content update

- Data sync
 - 1st approach: Consistent file database
 - Sync changes using ADB (e.g. adb push ...)
 - Keep track of files on device
 - Update database during sync
 - Slow and inconvenient
 - Multiple devices: Per-device database?!

That's great now we have to find a efficient way to move all of the assets into the sdcard folder.

Our first attempt was to launch adb push commands from python to copy files to the device.

Obviously we don't want to copy all files every time we change a asset, so we need to find a way to apply these incremental changes. In theory you could query file stats using adb commands, but that is very slow.

Instead we created a database with information about all files on the device. This way it was easy to identify which files were added, removed or changed. The database is updated during the sync.

This worked but it was somewhat cumbersome.

Launching a shell command for each file update is pretty slow and managing the database can be tricky especially when using multiple devices.



Fast content update

- Data sync
 - 2nd approach: Scan device files
 - Naïve implementation is sllloooooowwww....
 - Re-implement ADB protocol based on OS source
 - https://android.googlesource.com/platform/system/core.git/+/_master/adb/



Thankfully the Android OS is open source, so we were able to analyze the code of the ADB client that runs on the device.

Turns out that it uses a socket to execute the ADB commands using a very simple communication protocol. Rather than spawning a new shell process for each IO command we simply reimplemented the protocol in order to be able to efficiently communicate with the device.



Fast content update

- Data sync
 - 2nd approach: Scan device files (cont.)
 - Example: List directory

```
socket.send(pack("LIST",15,"/sdcard/dfp/dfa"))
while True:
    id, mode, size, time, namelen = unpack(socket.recv(16))
    name = '' if namelen == 0 else _recvall(socket,namelen)
    if id == "DONE": break
    if stat.S_ISDIR(mode):
        dirs.append(...)
    elif stat.S_ISREG(mode):
        files.append(...)
return (dirs, files)
```

The python (pseudo) code on this slide shows the code we used to scan all files on the device in order to be able to find all added, removed or changed files.



Fast content update

- Data sync
 - 2nd approach: Scan device files (cont.)
 - Compare files and compute diff
 - Sync files using ADB protocol
 - Very fast
 - No database

Once we have the list of changes we apply them by using the ADB protocol directly.

First we delete all removed files and folders and then we copy all added or changed files.

This worked very well for us. The communication speed with the device is optimal and no database has to be created and kept up to date.



Fast content update

- Results
 - Data sync: 15 - 25 seconds
 - APK build, copy and install: 30 - 40 seconds
- 10 - 20+ speedup!



By removing all assets from the APK and manually updating changed content we were able to reduce the turnaround time for a build to 45 to 60 seconds. The USB speed still has a big impact on the content update speed.

Overall we were able to achieve a 10x speedup on USB 3 devices like the Shield Tablet and more than 20x speedup on USB 2 devices like the OUYA or other older Android devices.

The numbers on this slide were measured on a Shield (USB 2) and a Shield Tablet (USB 3).

This is a significant improvement and allowed us to iterate on game-play scripts (e.g. touch and gamepad controller) much more quickly (and w/o going insane).



Fast content update

- Minimal staging folder
 - Delete unchanged files
 - Copy only added or changed files
 - Use sync timestamp
- Reduced work for Xcode
- No run-time changes necessary

Since iOS is a closed platform there is no easy way to sidestep Xcode in order to manually updated changed assets. It would be great to have a equivalent to ADB, but in the meantime another solution was necessary.

The key observation with Xcode's data sync is that copying changed files to the device is optimal, but identifying which files require updates is quite slow. Interestingly enough scanning all files on the device in order to compute the diffs is independent of whether files were changed or not.

The only way around this behavior is to remove all files from the staging directory that didn't change. Our solution therefore clears the staging folder before adding only files that were either added or changed.

During development very few files change, so the staging folder is almost empty and because of that Xcode's data sync is minimal.

The other great thing about this approach is that no run-time changes are necessary since the assets are copied to the real location.



Fast content update

- Results
 - Relink and update staging data: 15+ seconds
 - Data sync and install: 30 seconds
- 13 speedup (latest Xcode 2 – 4 speedup)



Updating the staging data is slightly slower than the naïve approach, but the data sync, app install and sandboxing are minimal.

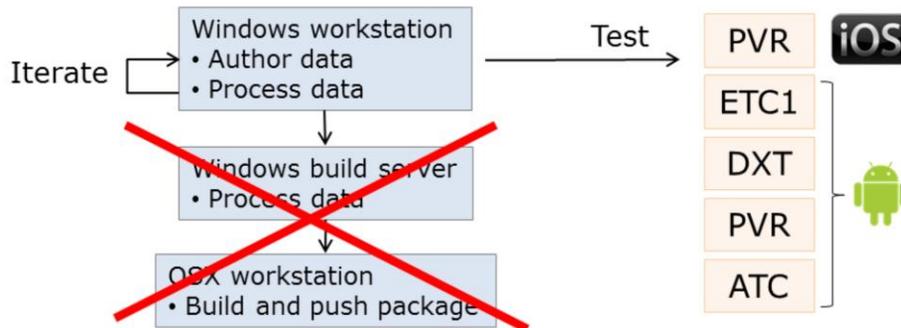
Using our solution we were able to achieve a 13x speedup for asset tweaks.

As mentioned earlier the most recent version of Xcode does improve the data sync time and the reboot is therefore 'only' 2 – 4 times as fast. It's nice to see that Apple has obviously recognized and fixed this problem, but using incremental asset updates is still beneficial.



Mobile as development platform

- Ideal workflow



Using the fast content update methods described up to this point was a huge win for us. It allowed us to implement and test a lot of our mobile-only scripts like the input controller or menus.

But it still doesn't allowed us to use mobile devices during production. What I mean by that is that we can't give a iPad to a artist or VFX dude, because in order to be able to see their work they would need to run the changed assets through the data pipeline, push them to the device and re-launch the app. This process is too cumbersome and slow for artists and therefore unusable for them.

What we really want is to give an iPad to any member of the team and they can change files on their workstation which will then be reflected on the device in real-time.

In other words we want to be able to stream assets from a workstation to mobile devices. This is already pretty common in console development (e.g. PS4, Xbone).



File streaming

- No re-sync necessary*
- Hot-reload changed files
- No OSX workstation needed
- Artists can work on target device

* Code changes require re-sync

The big advantage of file streaming is that the app on a mobile device doesn't need to be updated often. Updates are still necessary when the (non-script) code changes, which happened relatively infrequently. In the end we usually re-synced devices every day or two.

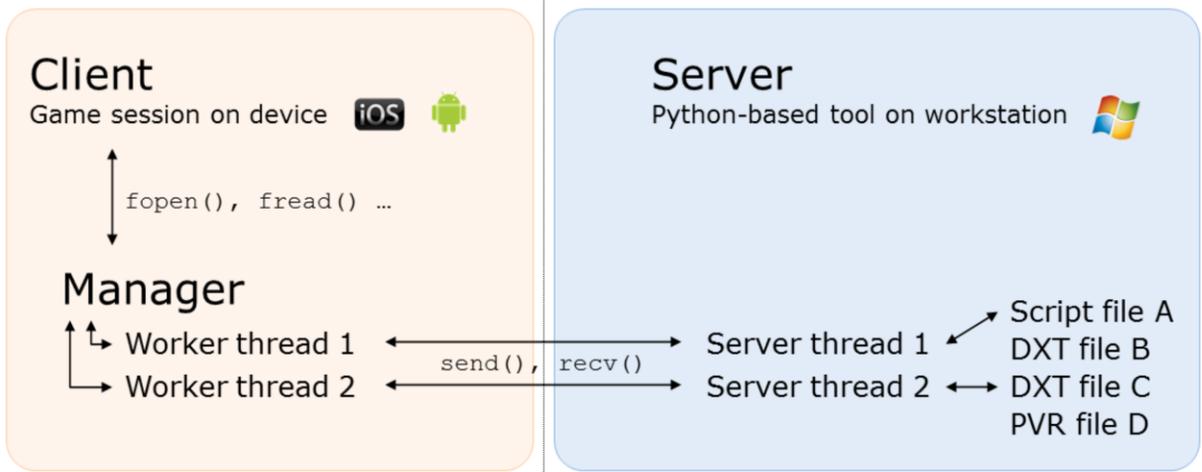
Since assets are loaded directly from the workstation it is also very easy to hot-reload changed content on the fly. This became very important when defining level-of-detail strategies for different devices, since it allowed us to tweak particle systems effectively.

In addition to that this feature was used heavily to fix shader bugs. We identify a shader that didn't work properly and start editing it and see the results of the tweaks in real-time on the device. Without hot-reloading my hair would probably be much more gray today.

Another benefit was that artists didn't need a OSX workstation to update the builds. We only have a few OSX machines and they tend to be on programmer desks, since they are responsible for the platform integration.



File streaming



Our file streaming is implemented by re-routing all IO operations to the file streaming manager.

The manager has multiple worker threads that are responsible to fulfill the incoming requests.

For example the game calls fopen to read a texture. This call will then be forwarded to one of the worker threads which sends the request via a socket to the server on the connected workstation.

The file server waits for incoming requests, executes them and then sends the resulting data back.

Once the worker thread receives the response from the server the results are returned to the original caller.



File streaming

- Latency optimization
 - Use local file cache



Intercepting IO operations and serving them from a remote source works great, but it's obviously much slower than local IO.

Because of that we implemented various optimizations to alleviate this problem.

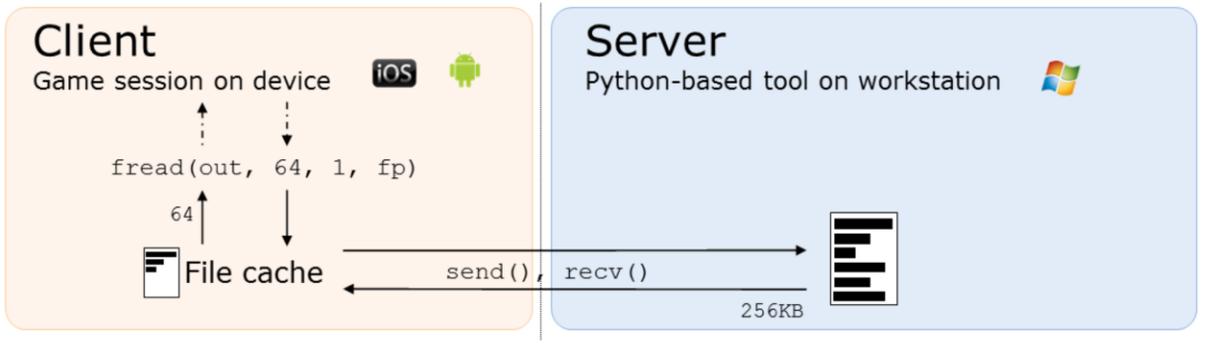
One way to hide some of the additional latency is by exploiting data locality. The most common file access pattern in Broken Age is to open a file and then read data in chunks of various sizes. We rarely move the file pointer by seeking, so caching data next to the original read makes the following reads much faster.

Our cache lines are 256 KB, which is a arbitrary size that worked best in our experiemnts.



File streaming

- Latency optimization (cont.)
 - Read at least one cache line



Intercepting IO operations and serving them from a remote source works great, but it's obviously much slower than local IO.

Because of that we implemented various optimizations to alleviate this problem.

One way to hide some of the additional latency is by exploiting data locality. The most common file access pattern in Broken Age is to open a file and then read data in chunks of various sizes. We rarely move the file pointer by seeking, so caching data next to the original read makes the following reads much faster.

Our cache lines are 256 KB, which is an arbitrary size that worked best in our experiments.



File streaming

- Latency optimization (cont.)
 - Exploit data locality

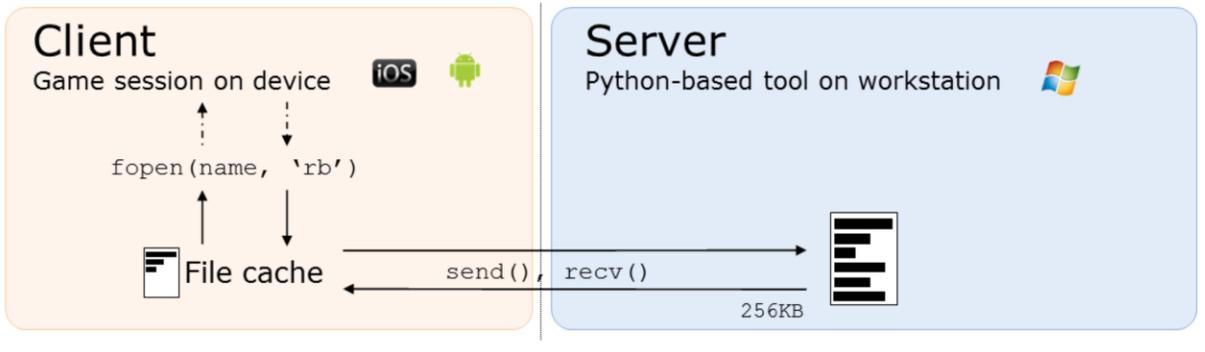


One way to hide some of the additional latency is by exploiting data locality. The most common file access pattern in Broken Age is to open a file and then read data in chunks of various sizes. We rarely move the file pointer by seeking, so caching data next to the original read makes the following reads much faster.



File streaming

- Latency optimization (cont.)
 - `fstat()` & `fopen()` read first cache line



Opening a file will read the first cache line, because the next operation will almost always be a read.



File streaming

- Latency optimization (cont.)
 - Cache expires after 5 ms to avoid stale data
 - Multiple concurrent requests
 - User filter to define streamed files
 - Local IO will always be faster



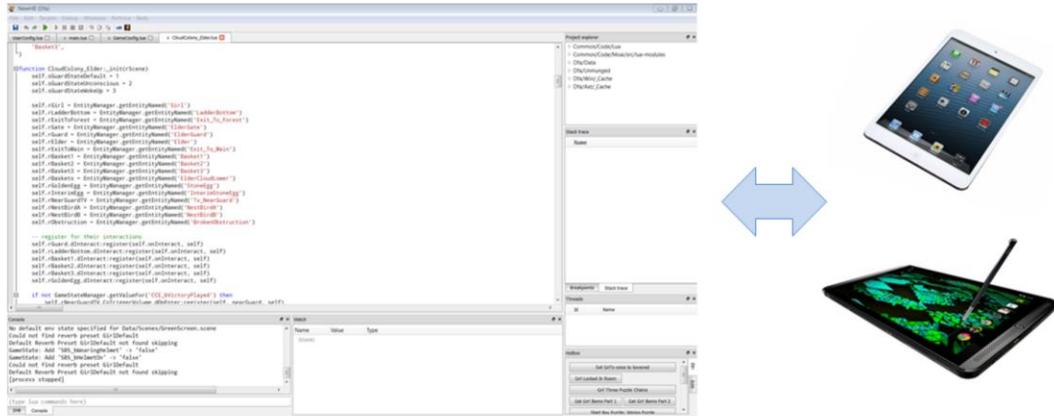
In order to avoid stale data the cache expires after 5 milliseconds or when the file is closed.

In addition to caching we also improve IO latency by serving multiple requests in parallel. Most of our data is loaded through IO worker threads anyway, so integrating this was relatively easy.

Another feature we implemented is to allow partial file streaming. Even with our latency optimizations using local IO is still much faster, so we allow the user to specify filter patterns. Only files that match the pattern will be streamed. This is helpful since an artist very often wants to concentrate on a few assets. For example there is no reason to stream all the big background textures when working on a particle effect or shader.

File streaming

- Fully integrated into our game editor 2HB



Overall implementing file streaming wasn't as straight forward as the fast content update, but it definitely paid off.

The file streaming is fully integrated into our game editor 2HB, which (in addition to other things) also allows remote Lua debugging on the device.



Conclusion

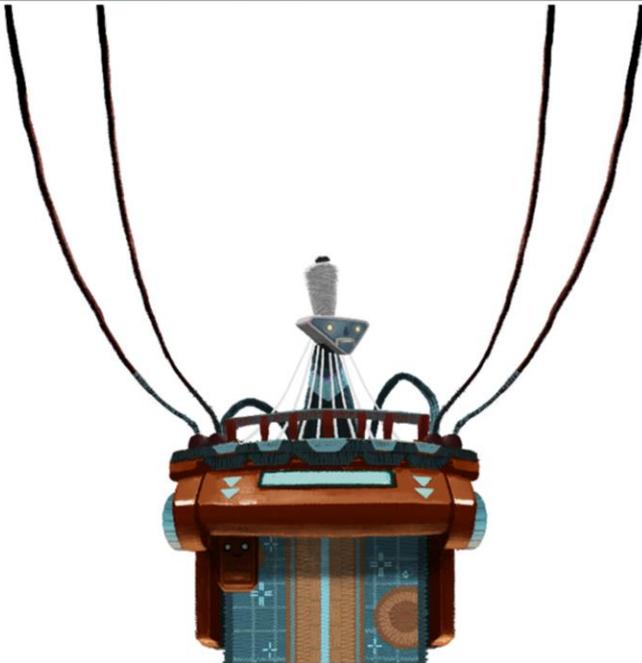
- Mobile as development platform is...
 - ...not trivial
 - ...worth your time
 - ...necessary for a ~~big~~ project
every



So if you are planning to ship games on iOS or Android, then making mobile devices a first-class development platform definitely pays off quickly.

Being able to quickly iterate on game-play scripts, shaders and other assets is just as important on mobile devices as it is on PC or consoles.

Unfortunately mobile devices don't make this easy out of the box, but with the techniques I presented today you should be able to improve your workflow significantly.



Thank you!
Questions?

 **p1xelcoder**