Power Management

v. 0.4

A current-generation mobile device is amazingly powerful for something that you can stick in your pocket - you can reasonably expect to find four 2.6 Ghz CPU cores and a 600 MHz GPU. Fully utilized, they can actually deliver more performance than an XBOX 360 or PS3 in some cases.

A governor process on the device monitors an internal temperature sensor and tries to take corrective action when the temperature rises above certain levels to prevent malfunctioning or scalding surface temperatures. This corrective action consists of lowering clock rates.

If you run hard into the limiter, the temperature will continue climbing even as clock rates are lowered, and CPU clocks may drop all the way down to 300 MHz. The device may even panic under extreme conditions. VR performance will catastrophically drop along the way.

The default clock rate for VR applications is 1.8 GHz on two cores, and 389 MHz on the GPU. If you consistently use most of this, you will eventually run into the thermal governor, even if you have no problem at first. A typical manifestation is poor app performance after ten minutes of good play. If you filter logcat output for "thermal" you will see various notifications of sensor readings and actions being taken. (For more on logcat, see *Android Debugging*.)

A critical difference between mobile and PC/console development is that no optimization is ever wasted. Without power considerations, if you have the frame ready in time, it doesn't matter if you used 90% of the available time or 10%. On mobile, every operation drains the battery and heats the device. Of course, optimization entails effort that comes at the expense of something else, but it is important to note the tradeoff.

1. Fixed Clock Level API

On the current device, the CPU and GPU clock rates are completely fixed to the application set values until the device temperature reaches the limit, at which point the CPU and GPU clocks will change to the power save levels. This change can be detected (see “3. Power State Notification and Mitigation Strategy” below), and some apps may choose to continue operating in a degraded fashion, perhaps by changing to 30 FPS or monoscopic rendering. Other apps may choose to put up a warning screen saying that play cannot continue.

The Fixed Clock API allows the application to set a fixed CPU level and a fixed GPU level. These are abstract quantities, not MHz / GHz, so some effort can be made to make them compatible with future devices. For the initial hardware, the levels can be 0, 1, 2, or 3 for CPU

and GPU. 0 is the slowest and most power efficient; 3 is the fastest and hottest. Typically the difference between the 0 and 3 levels is about a factor of two.
Not all clock combinations are valid for all devices. For example, the highest GPU level may not be available for use with the two highest CPU levels. If an invalid matrix combination is provided, the system will not acknowledge the request and clock settings will go into dynamic mode. VrLib will assert and issue a warning in this case.

The following chart illustrates clock combinations for the supported Samsung device:

	GPU	240	300	389	500
CPU	Level	0	1	2	3
884	0				
1191	1				
1498	2				Caution
1728	3				Caution

Note: For the initial release of the device, the combinations 2 CPU / 3 GPU (2,3) and 3 CPU / 3 GPU (3,3) are allowed. However, we discourage their use, as they are likely to lead quickly to overheating.

2. Power Management and Performance

How long your game will be able to play before running into the thermal limit is a function of how much work your app is doing, and what the clock rates are. Changing the clock rates all the way down only yields about a 25% reduction in power consumption for the same amount of work; most power saving has to come from actually doing less work in your app. This is critically important – there are no magic settings in the SDK to fix power consumption.

If your app can run at the (0,0) setting, it should never have thermal issues. This is still two cores at around 1 GHz and a 240 MHz GPU, so it is certainly possible to make sophisticated applications at that level, but Unity-based applications might be difficult to optimize for this setting.

There are effective tools for reducing the required GPU performance – don't use chromatic aberration correction on TimeWarp, don't use 4x MSAA, and reduce the eye target resolution. Using 16 bit color and depth buffers can also help some. It is probably never a good trade to go below 2x MSAA – you should reduce the eye target resolution instead. These are all quality tradeoffs which need to be balanced against things you can do in your game, like reducing overdraw (especially blended particles) and complex shaders. Always make sure textures are compressed and mipmapped.

In general, CPU load seems to cause more thermal problems than GPU load. Reducing the required CPU performance is much less straightforward. Unity apps should use the multithreaded renderer option, since two cores running at 1 GHz do work more efficiently than one core running at 2 GHz.

If you find that you just aren't close, then you may need to set MinimumVsyncs to 2 and run your game at 30 FPS, with TimeWarp generating the extra frames. Some things work out okay like this, but some interface styles and scene structures highlight the limitations. For more information on how to set MinimumVsyncs, see the TimeWarp technical note.

So, the general advice is:

If you are making an app that will probably be used for long periods of time, like a movie player, pick very low levels. Ideally use (0,0), but it is possible to use more graphics if the CPUs are still mostly idle, perhaps up to (0,2).

If you are okay with the app being restricted to ten-minute chunks of play, you can choose higher clock levels. If it doesn't work well at (2,2), you probably need to do some serious work.

With the clock rates fixed, observe the reported FPS and GPU times in logcat. The GPU time reported does not include the time spent resolving the rendering back to main memory from on-chip memory, so it is an underestimate. If the GPU times stay under 12 ms or so, you can probably reduce your GPU clock level. If the GPU times are low, but the frame rate isn't 60 FPS, you are CPU limited.

Always build optimized versions of the application for distribution. Even if a debug build performs well, it will draw more power and heat up the device more than a release build.

Optimize until it runs well. For more information on how to improve your application's performance, see the following documents: *Performance Guidelines* and, for Unity developers, *Unity Performance Best Practices*.

3. Power State Notification and Mitigation Strategy

The mobile SDK provides power level state detection and handling. Power level state refers to whether the device is operating at normal clock frequencies or if the device has risen above a thermal threshold and thermal throttling (power save mode) is taking place. In power save mode, CPU and GPU frequencies will be switched to power save levels. The power save levels are equivalent to setting the fixed CPU and GPU clock levels to (0, 0). If the temperature continues to rise, clock frequencies will be set to minimum values which are not capable of supporting VR applications.

Once we detect that thermal throttling is taking place, the app has the choice to either continue operating in a degraded fashion or to immediately exit to the Oculus Menu with a head-tracked error message.

In the first case, when the application first transitions from normal operation to power save mode, the following will occur:

- The Universal Menu will be brought up to display a dismissible warning message indicating that the device needs to cool down.
- Once the message is dismissed, the application will resume in 30Hz TimeWarp mode with correction for chromatic aberration disabled.
- If the device clock frequencies are throttled to minimum levels after continued use, a non-dismissible error message will be shown and the user will have to undock the device.

In this mode, the application may choose to take additional app-specific measures to reduce performance requirements. For Native applications, you may use the following AppInterface call to detect if power save mode is active: `GetPowerSaveActive()`. For Unity, you may use the following plugin call: `OVR_IsPowerSaveActive()`. See Moonlight/OVRModeParms.cs for further details.

In the second case, when the application transitions from normal operation to power save mode, the Universal Menu will be brought up to display a non-dismissible error message and the user will have to undock the device to continue. This mode is intended for applications which may not perform well at reduced levels even with 30Hz TimeWarp enabled.

You may use the following calls to enable or disable the power save mode strategy:

For Native, set `modeParms.AllowPowerSave` in `ConfigureVrMode()` to `true` for power save mode handling, or `false` to immediately show the head-tracked error message.

For Unity, you may enable or disable power save mode handling via `OVR_VrModeParms_SetAllowPowerSave()`. See Moonlight/OVRModeParms.cs for further details.

Last Update: Nov 10, 2014