Atomic Mode-Setting

Thierry Reding

NVIDIA Corporation

February 1, 2015

Table of Contents

1. A Bit of History
   - Pre-KMS Era
   - Kernel Mode-Setting Era
2. Atomic Mode-Setting
   - Building Blocks
3. Driver Conversion
   - Preparatory Work
   - Three Phases
   - Follow-up Work
4. Future Work
   - Userspace
   - Drivers
5. Summary

NVIDIA

Prehistory

- user-space mode-setting
- X driver has direct access to graphics card registers
- X needs superuser privileges

The Middle Ages

- DRM allows multiple processes to access a single graphics card
- attempt to provide a common API
  - buffer management
  - command submission
- user-space still performs mode-setting

Renaissance

- kernel mode-setting (KMS) is introduced
- kernel drivers are now in charge of mode-setting
- kernel drivers also manage other resources
  - buffer objects
  - output configuration
  - hotplug

Renaissance - Kernel Mode-Setting

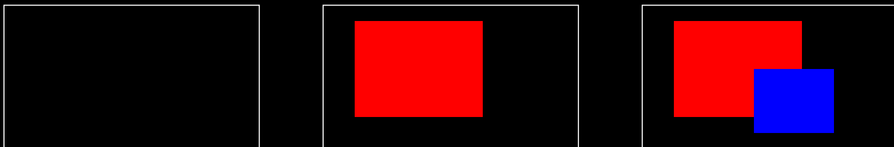
- CRTCs, encoders and connectors
  - query configurations and capabilities
  - set modes and framebuffers
  - page-flip buffers
- planes
  - query number of available planes and supported formats
  - set a plane (position, size, framebuffer)
  - ...
- object properties

Renaissance - Not everything is perfect

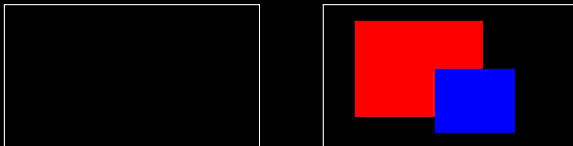
- mode sets fail too late, there's no way to rollback
- KMS does not guarantee that a configuration is applied in totality for the next frame
- no VBLANK-synchronized page-flips for planes (!)
- compositors want perfect frames

Perfect Frames?

Suppose we want to show two planes on the screen:



Whereas we really want this:



Atomic Mode-Setting

- allows an output configuration to be validated before it is applied:
  - no hardware is touched before the driver acknowledges that the configuration is valid and can be applied
  - no need for rollback
- allows atomic updates of an output configuration:
  - multiple planes updated at the same time
  - perfect frames
- allows for unification and simplification of drivers
  - legacy code can be removed from drivers
  - much of the handling moves into helpers

Atomic Mode-Setting - Building Blocks

- Universal Planes
- Properties
- Atomic State

Universal Planes

- root window is special in legacy KMS
  - becomes a regular plane
  - hardware treats them the same anyway
    - code becomes simpler
- cursor exposed as plane
  - exports a list of supported formats
- helpers are available
  - implement legacy IOCTLs using universal planes
  - implement primary plane using legacy callbacks

Properties and Atomic State

- interface parameters are converted to object properties
- properties are stored in atomic state objects
    - blob properties are read-only, no way to change the video mode from userspace (yet)
- atomic state objects are validated and applied

Universal Planes

- relatively easy to implement
  - matches what modern hardware exposes
- many drivers are already converted
  - good references

VBLANK Handling

- atomic mode-setting has somewhat strict requirements
  - plane updates are synchronized to VBLANKs to make sure framebuffers are unused before unreferenced
  - good news because drivers don't have to worry anymore
- drivers must disable VBLANK machinery when the display controller is off (`drm_crtc_vblank_off()`)
- when the display controller is enabled the VBLANK machinery must be turned on again (`drm_crtc_vblank_on()`)

Helpers fall over if the hardware state isn't accurately mirrored in the VBLANK machinery.

Driver Rewrite

- atomic mode-setting has fairly high expectations
- especially important for atomic DPMS
- `->prepare()` is called when the CRTC or encoder is disabled
- `->mode_set()` is called when the mode changes
- `->commit()` is used when the CRTC or encoder is enabled

On Tegra everything was done in `->dpms()`, requiring an almost complete rewrite.

Conversion in Three Phases

- Phase 1 - Transitional Helpers
- Phase 2 - Atomic State Object Scaffolding
- Phase 3 - Rolling out Atomic Support

Transitional and atomic helpers are very modular and the conversion is suprisingly painless.

Phase 1 - Transitional Helpers

- implement legacy entry points in terms of new atomic callbacks
  - CRTCs
    - `->atomic_check()` - validate state
    - `->atomic_begin()` - prepare for updates
    - `->atomic_flush()` - apply updates atomically
    - `->mode_set_nofb()` - apply CRTC timings

    It is possible to set a mode without a primary plane.
  - planes
    - `->prepare_fb()` - e.g. pin backing storage
    - `->cleanup_fb()` - e.g. unpin backing storage
    - `->atomic_check()` - validate state
    - `->atomic_update()` - plane updates
    - `->atomic_disable()` - disable plane
- caveat: `->atomic_destroy_state()` needs to be wired up here, otherwise the transitional helpers will leak state objects

Phase 2 - Atomic State Object Scaffolding

- wire up state callbacks for planes, CRTCs and connectors:
  - `->reset()`
    - `drm_atomic_helper_*_reset()`
  - `->atomic_duplicate_state()`
    - `drm_atomic_helper_*_duplicate_state()`
  - `->atomic_destroy_state()`
    - `drm_atomic_helper_*_destroy_state()`
- default helpers are good enough for starters

Phase 3 - Rolling out Atomic Support

- Step 1 - switch to atomic helpers internally:
  - Planes:
    - `drm_atomic_helper_update_plane()`
    - `drm_atomic_helper_disable_plane()`
  - Driver:
    - `drm_atomic_helper_check()`
    - `drm_atomic_helper_commit()`

Driver now uses only atomic interfaces internally.

Phase 3 - Rolling out Atomic Support

- Step 2 - switch to atomic helpers for userspace IOCTLs:
  - `drm_atomic_helper_set_config()`
    - `DRM_MODE_IOCTL_SET_CRTC`
  - provide a custom `->atomic_commit()` implementation
    - to support asynchronous commits, required for page-flipping
  - `drm_atomic_helper_page_flip()`
    - `DRM_MODE_IOCTL_PAGEFLIP`

Driver is now fully atomic (semantically).

Rip out Cruft

- `->mode_set()` and `->mode_set_base()` are no longer used
  - `->mode_set_nofb()` does what is necessary to set a mode
- atomic DPMS
  - DPMS standby and suspend are no more (w00t)
  - `->disable()` and `->enable()` callbacks
  - atomic DPMS is a full off or on cycle

Isn't this exactly what Tegra used to do before the almost complete rewrite? Almost, except the atomic helpers now keep track of everything.

Where the fun begins

- Subclassing atomic state:
  - embed `struct drm_*_state` in device-specific structures
  - move device-specific data into these structures

  Allows `->atomic_check()` to cache information already computed. Simplifies other code because it doesn't need to recompute.
- Implement truly atomic updates:
  - e.g. `->atomic_flush()`
  - "GO" bits

Userspace IOCTL

- brand new in Linux v3.20
- still hidden behind drm.atomic parameter
- amalgamation of objects and properties:
  - array of object IDs
  - array of per-object property counts
  - array of properties
  - flags
    - page-flip
    - test-only
    - non-block
    - allow modeset

More cool stuff

- hardware readout:
  - `->reset()` reads state of hardware at driver load time
  - no transition if no state is changed
  - seamless transition between firmware or bootloader and kernel
- unify asynchronous commits
  - possibly via generic helpers
- asynchronous page-flips
  - `eglSwapInterval()`
  - because everybody loves benchmarks
  - generic flip-queue that works for all drivers
    - can fast-forward over a bunch of updates if supported by the driver
    - it's like benchmarking page-flips, but without the tearing
- plane rotation via standard properties

NVIDIA

Summary

- Atomic mode-setting is not only necessary but allows a bunch of nice and long overdue cleanup and unification.
- Atomic mode-setting is nowhere near as complicated as it sounds.
- Before converting your driver, make sure to have well-behaved callbacks and VBLANK handling.
- Given that the conversion is almost trivial because of excellent helpers.

Bottom Line

If you maintain a KMS driver, go convert it now, otherwise you are going to miss out on all the good stuff coming up.