Supporting New YUV Formats in Mesa

Eric Smith

XDC 2025

No need to fear weird movie formats

Existing YUV Formats

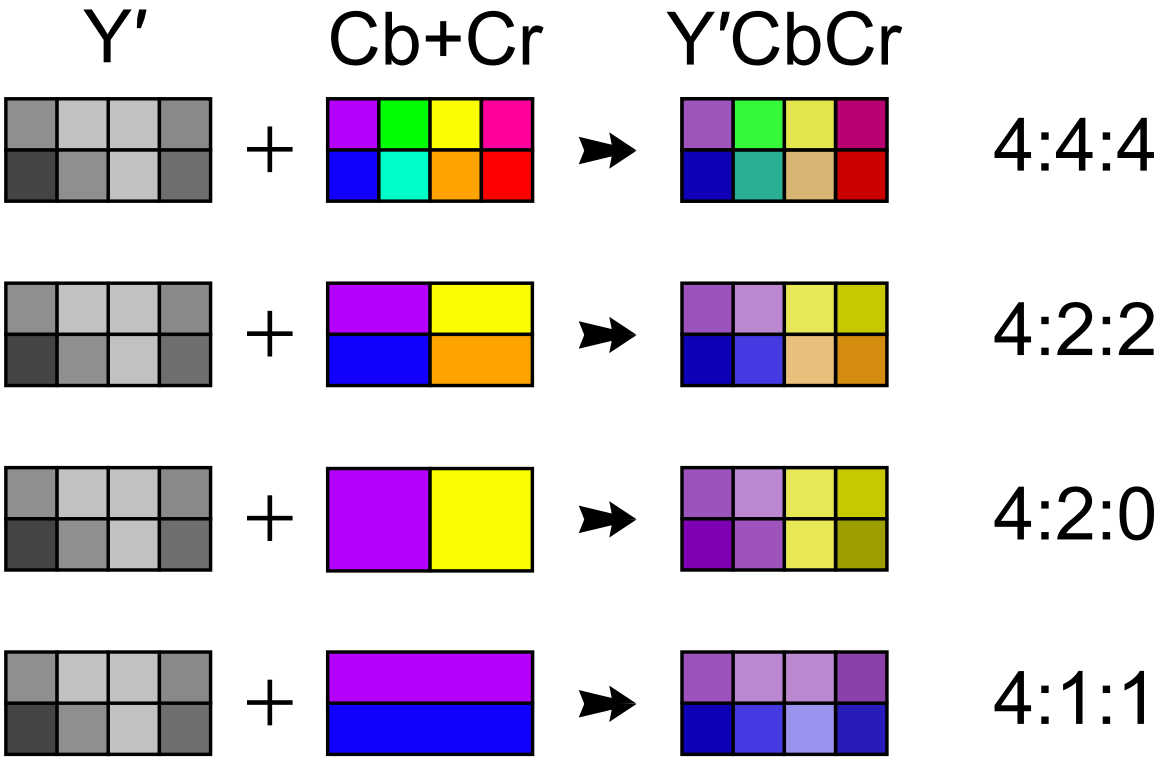
YUV, YCbCr, 4:2:2, and all that

- YUV historically referred to a specific way of encoding color for broadcast television
  - Y is the luminance (black & white), U and V encode color
  - These days “YUV” is used generically for luma + chroma representations
- YCbCr (or Y’CbCr) is a digital representation
  - Various specific standards for converting from RGB to YCbCr: ITU-R BT.601, ITU-R BT.709, SMTPE 240M, etc.
  -

Subsampling

- Eyes tend to be less sensitive to chroma than luma
- So subsample chroma to reduce storage & bandwidth
- Terminology: J:a:b, where J is reference width (typically 4), a is number of chroma samples horizontally, b is factor for next line (usually same as a, or 0)
  -

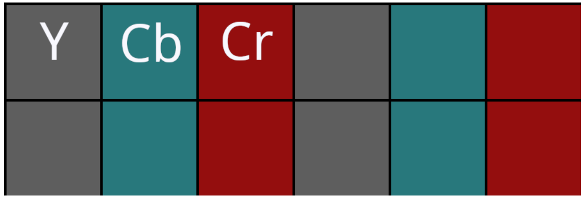
Common Subsampling Formats



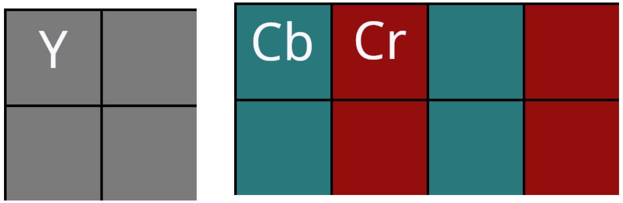
By Mackenziemacaroni - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=147346822

Components in planes

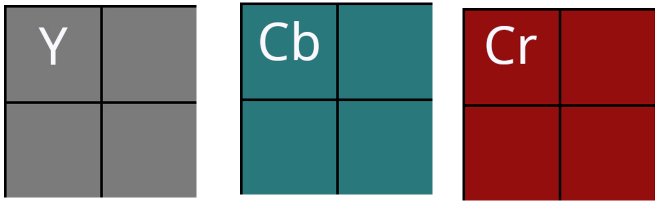
- Interleaved



- Semi-planar



- Planar



Existing Mesa Formats

Mesa Formats

- Mesa can use YUV formats for textures
  - May be imported as “External” textures from video decode HW
  - Or video decode could be done in software or in custom shaders
- Much of what we say here will also apply to RGB textures too (or compressed textures)

YUV Formats Supported by Mesa

- Single Plane, Two Planes (Y and UV subsampled), or Three Planes (Y, U, V, possibly with subsampling)
- 4:4:4, 4:2:2, and 4:2:0 variants
- Components:
  - 8 bit and 10 bit are common
  - 12 and 16 bits per component are also seen
- Various ways to tile and interleave the components

Existing YUV Formats

- Around 44 YUV related formats in Mesa now, including:
  - A8Y8U8V8_444, Y8_U8_V8_444, etc.
  - Y8U8Y8V8_422, U8Y8V8Y8_422, etc.
  - Y8_U8V8_420, Y10_U10V10_420, etc.
- Some of these have aliases reflecting common usage
  - E.g. NV12, aka Y8_U8V8_420_UNORM

Adding New Formats to Mesa

New FourCC code (if necessary)

- FourCCs are 4 byte identifiers consisting of 4 ASCII characters and indicating the overall pixel format, e.g. “YU08” == 0x30385559
- If a new one is necessary, add it to drm-uapi/drm_fourcc.h
- Also need to update the kernel
  - Upstreaming kernel patches can take a while

src/util/format/u_format.yaml

Entries look like:

```
- name: Y8_U8V8_420_UNORM
  alias: NV12
  layout: planar2
  colorspace: YUV
  block: {width: 1, height: 1, depth: 1}
  channels: []
  swizzles: [X, Y, Z, W]
```

src/util/format/u_format.yaml

Fields to fill out:

- Name: group components together by plane
    - Nowadays we try to include subsampling
- Alias (e.g. common FOURCC)
- Layout (often “subsampled” or “planar2” for YUV)
- Colorspace (may need both RGB and YUV versions)

src/util/format/u_format.yaml

Continued

- Block: size is based on access patterns
  - 1x1 for if individual texel elements can be accessed
  - 4x1 for 10 bits where 4 texels are packed in 5 bytes
- Channels (bits per channel, usually UN8 or UN10)
- Swizzle

src/util/format/u_format.yaml

- Repeat for the RGB equivalent of the YUV format, if necessary
  - This usually is: we’ll want to use the RGB version for texturing, with colorspace conversion in the shader

Digression: RGB and YUV

- For each YUV format we typically have an RGB equivalent, used to access the raw component data
  - E.g. for Y8_U8V8_420_UNORM we have R8_G8B8_420_UNORM, and so on
- Color space conversion is done in the shader this way
  - Can get precise conversion coefficients, which often isn’t possible in HW

src/util/format/u_format_table.py

- Add the new format(s) to the noaccess_formats list
- If you skip this step, you will have to provide conversion functions in src/util/format/u_format_yuv.c
  - Usually we don’t bother with this, if the format is included for hardware reasons

DRI Considerations

- dri2_format_mapping_table indicates when we can use an RGB format to support a YUV one
- dri_create_image_from_winsys checks this; if the HW does not directly support sampling from YUV (common!) we need to know the RGB equivalent
  - Even if HW supports some YUV → RGB, it probably doesn’t support all the variations

YUV Conversion code in shader

- Texture loading and conversion is generated in src/compiler/nir/nir_lower_tex.c
  - Existing code can handle most situations
  - Driven by external sampler key set up by state tracker

State tracker updates

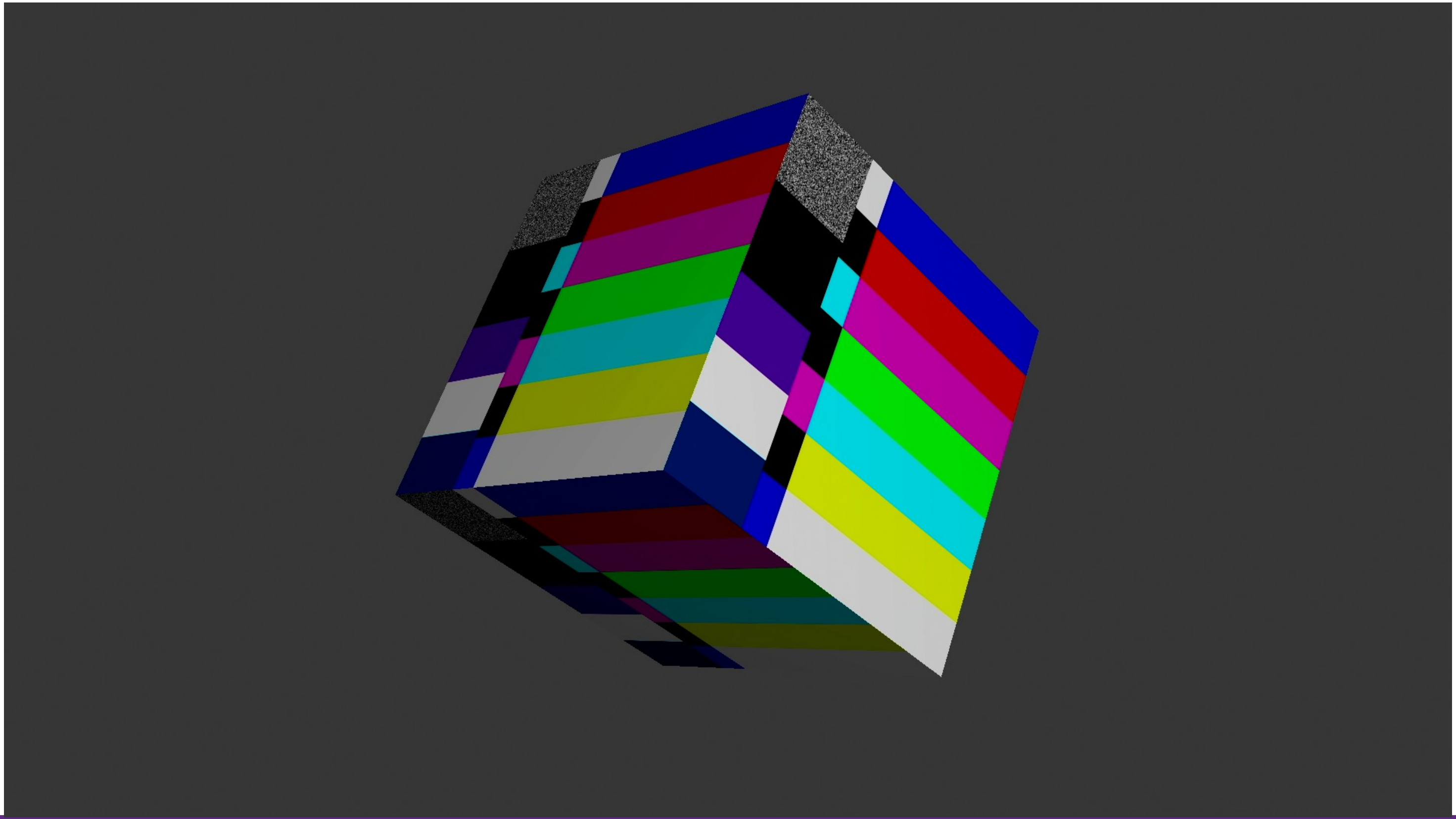
- st_program.h: st_get_external_sampler_key sets up fields to describe texture lowering
- st_cb_eglimage.c:
  - lower YUV to equivalent RGB in is_format_supported()
  - Set up texture object in st_bind_egl_image()

State tracker updates (cont’d)

- st_atom_texture.c: st_get_sampler_views() needs to set up views for additional planes
- Similarly for update_shader_samplers() in st_atom_sampler.c
- Also update st_get_sampler_view_format() in st_sampler_view.c, if necessary

Testing
(something always goes wrong)

Kmscube



Kmscube

- Relatively easy to add new formats
- I have a fork that also allows modifiers to be specified
  - https://gitlab.freedesktop.org/ericsmith/kmscube
  - nv15_nv20_p010 branch

Gstreamer

- Test with real movies
- Actually modifying gstreamer to handle new formats is a fairly big job
  - But presumably someone is going to do it if the format is interesting

Thank you!

We are hiring

col.la/careers