LOW LEVEL 3D EMBEDDED GRAPHICS

WITH LINUX, OPENGL ES AND BUILDROOT

1st > 3rd July 2019

SophiaConf.fr

The Open Source conferences & workshops cycle of the French Riviera

by TelecomValley

JULIEN OLIVAIN

SOPHIACONF
JULY 2ND 2019

PUBLIC

SECURE CONNECTIONS FOR A SMARTER WORLD

INTRODUCTION

Who am I?

- Julien Olivain
  `<julien.olivain@nxp.com>`
- Technical lead, software architect at NXP

- Interests:
  - Computer Graphics
  - Parallel computing
  - Security
  - Functional Safety
  - Operating systems

- **Disclaimer**: opinions and statements are those of the presenter, which are not necessarily those of NXP.

A Position of Strength to Better Serve Our 26,000+ Customers

Employees in
30+ Countries
Headquartered in Eindhoven, Netherlands

~30,000
Employees

9,000
Patent Families

$9.41B
Annual Revenue[1]

60+
Year History

~9,000
R&D Engineers

[1] Posted revenue for 2018 – Please refer to the Financial Information page of the Investor Relations section of our website at www.nxp.com/investor for additional information

Together With Our Valued Customers,

We Are Creating Secure Connections for a Smarter World

Enhanced Security
Mesh Connectivity
Sensors
Smart LED Lighting
Gateway & Cyber Security
Access

Cloud Computing
Gateways
Routers & Switches
Security Appliances
Wireless Base Stations
Smartphones

Car Entertainment
In-vehicle Networking
Car Access
Car2X and Radar

Smart Parking
Smart Roads
Traffic Congestion
Smart Lighting
Waste Management

Manufacturing Automation
Temperature Monitoring
Machine Diagnosis & Control
Remote Asset Control
Fleet Tracking

Distribution
Energy Consumption & Monitoring
Water Pressure Measurement

e-passport
Smart eID
Health Card

Secure Bank Cards
Mobile Transactions
Loyalty- Reward
RFID Tags/Labels
Supply Chain Monitoring
NFC Readers

Smart Watch
Activity Tracking
Smart Glass

Contactless
Transport Card
Access Solutions
Micro-payments

Hospital & at-home Patient Monitoring
Fall Detection
Personal Health & Fitness Monitoring

NXP Semiconductors – Sophia Antipolis

- Wide range of activities in NXP Sophia Antipolis

- 250 people working on:
  - Audio
  - Security
  - Wireless
  - i.MX8 Processor Architecture and Design
  - Software solution development and support

NXP Semiconductors – My Team

- NXP Professional Services
  - Embedded Linux / Development experts based in Sophia Antipolis
  - Customized software development / integration / optimization / maintenance on NXP i.MX processors
  - Bootloaders, Linux Kernel, User Space development, Build systems (Buildroot / Yocto)
  - GPU experts (2D, 3D, Vector Graphics)
  - Other specific area of expertise: Blackberry QNX, Greenhills INTEGRITY, Android, Bare-metal software, Security and Trusted Environments…

- See: https://nxp.com/proservices

Agenda / Summary

- What this workshop is NOT about:
  - We will NOT program a full high-end game engine today (sorry)
  - Code example are NOT covering standard modern 3D programming (many resources are available on that, links are provided though)
- This workshop is about the “simplest” ways of programming embedded GPUs

- **Part 1**: How to program a GPUs with the smallest possible dependencies
  - For simplicity, concepts are explained with OpenGL ES 2.0, which is the simplest, omnipresent and well known standard
  - Small examples of rendering / image or data processing, natively on a host computer

- **Part 2**: How to easily move the host native examples of part 1 on embedded systems?
  - Introduction of Buildroot: to totally control our Linux development environment

- **Part 3**: How emulate a system with GPU from part 2 without actual hardware
  - Introduction of QEMU and Virgl 3D GPU

Preparation for code and samples

- Up-to-date Ubuntu 18.04 LTS is assumed here:

```
sudo apt update
sudo apt dist-upgrade
```

- Fetch data and install dependencies:

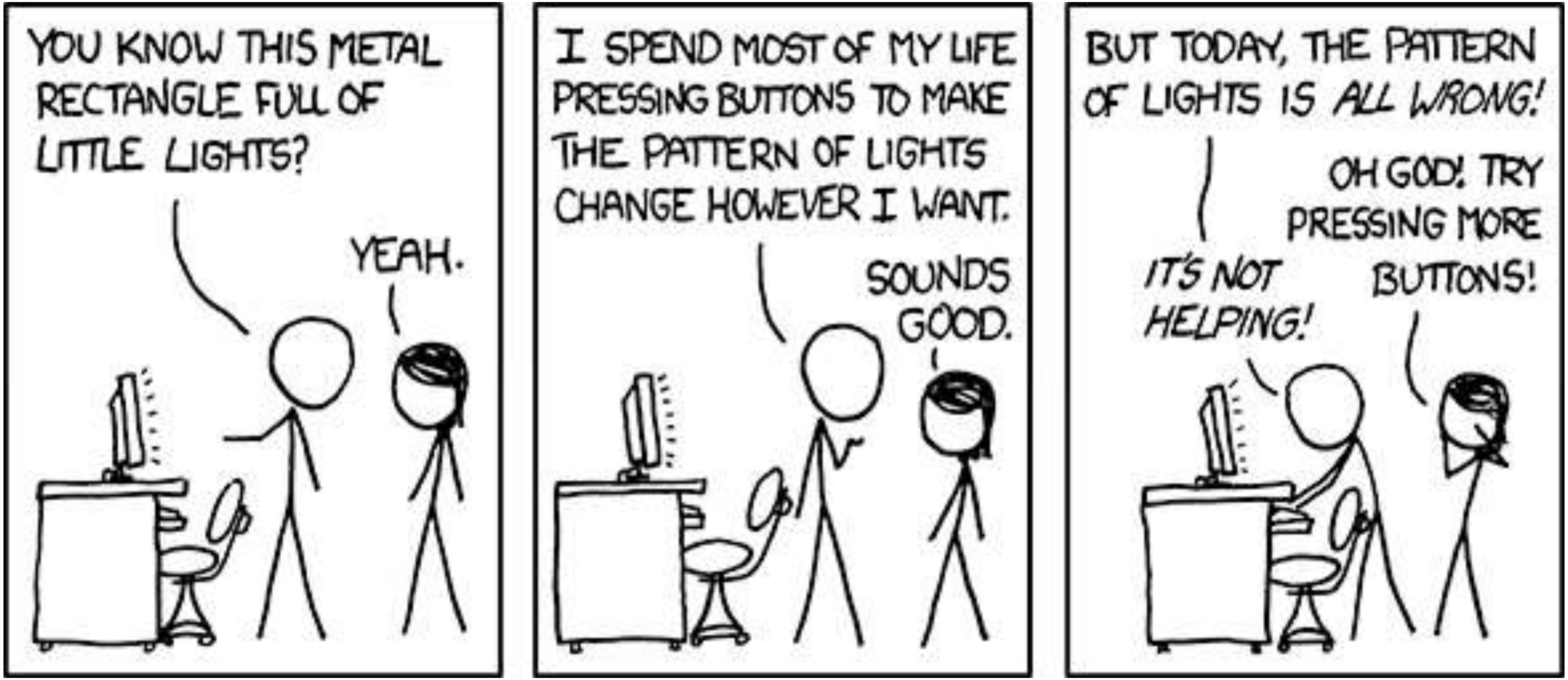
```
sudo apt install git
git clone https://github.com/jolivain/SophiaConf2019
cd SophiaConf2019
# check the script content, the presenter decline any responsibility ;)
sudo ./install-deps-ubuntu-18.04lts.sh
```

- People wanting to do the Buildroot Lab are invited to prepare their environment:

```
git clone -b SophiaConf2019 https://github.com/jolivain/buildroot
cd buildroot
make sc19_qemu_aarch64_defconfig
nice make
```

PART 1
GPU PROGRAMMING

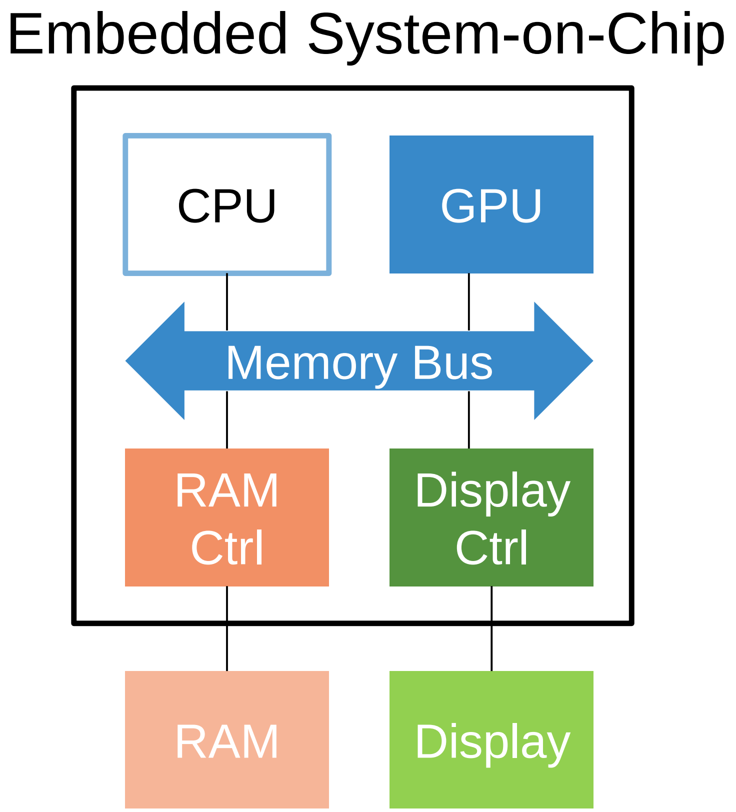
Computer Problems

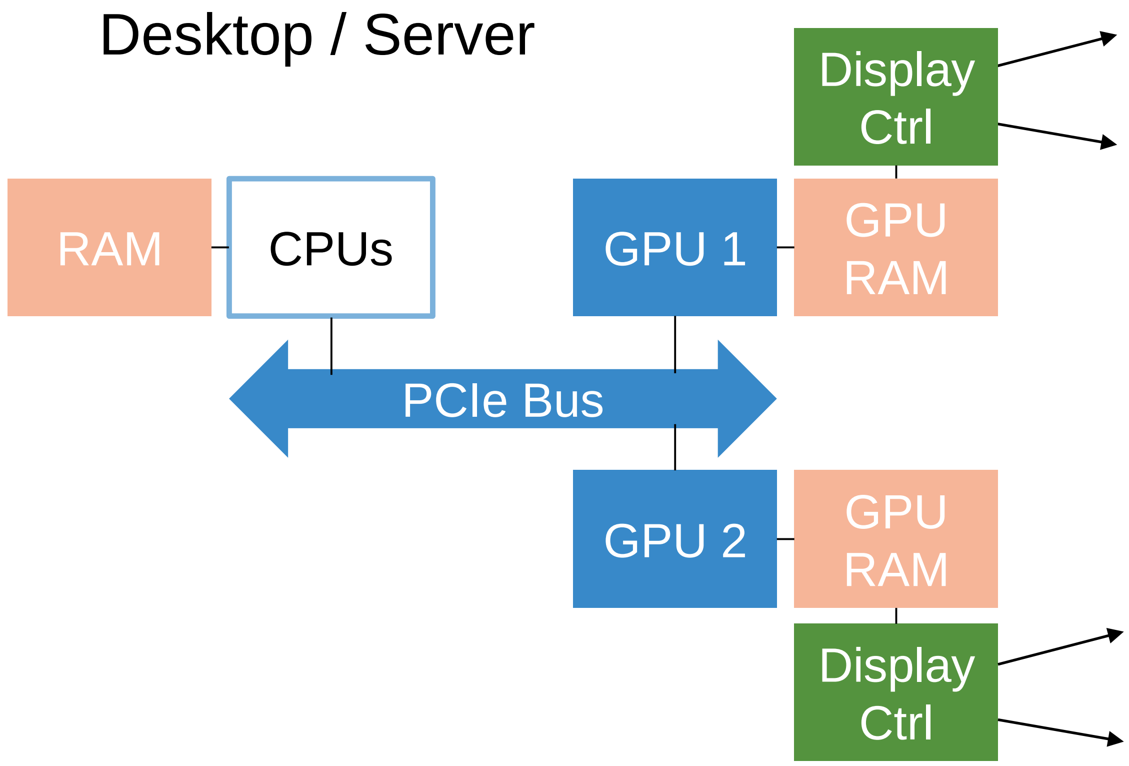


This is how I explain computer problems to my cat.
My cat usually seems happier than me.

http://www.xkcd.com/722

System GPU Architectures and Topologies





- Many topologies, use cases and implementation details to support
- Common standard and portable ways to program GPUs are needed: OpenGL

Khronos APIs

- OpenGL 1.0 .. 4.6
  - Windowing API: GLX, WGL, CGL, ...
- **<u>OpenGL ES</u>** 1.0 .. **<u>2.0</u>** .. 3.2
  - Windowing API: EGL
- OpenGL SC 1.0 / 2.0
- WebGL 1.0 / 2.0
- Vulkan 1.0 / 1.1
- OpenCL 1.0 .. 2.2

- Royalty-free and open standards for 3D graphics, Virtual and Augmented Reality, Parallel Computing, Neural Networks, and Vision Processing.

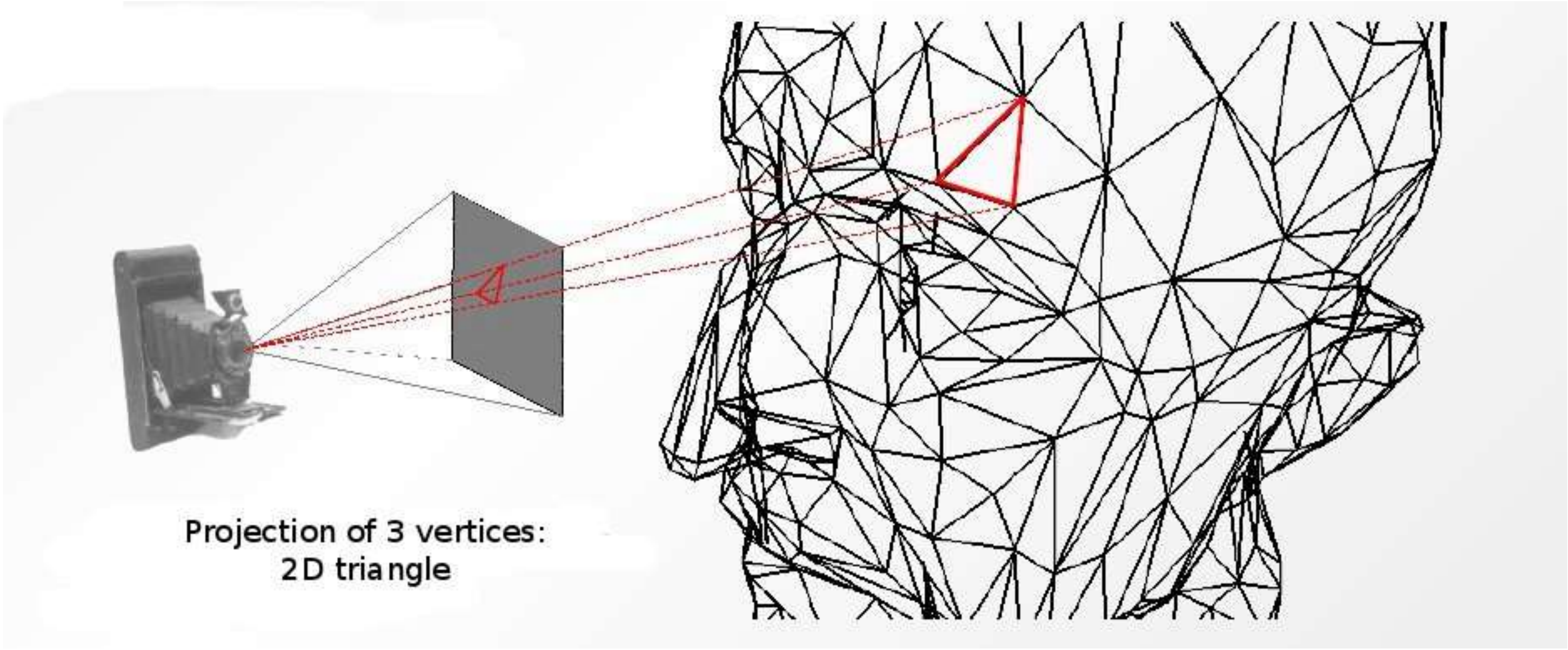
Why do we need a GPU, anyway?

- Answers:
  - To accelerate rendering fast enough to create animation, historically
  - Also, to efficiently perform complex calculations on data (Vision, Machine learning, …)

- Examples:
  - FullHD: 1920x1080 32bpp 60 Hz = 474 MB/s
  - 4K UHD-1: 3840x2160 32bpp 60 Hz = 1.9 GB/s

- Acceleration is achieved with:
  - Parallelism,
  - Accelerating frequent and costly operations (e.g. trigonometry, vector ops, filtering...)
  - Adding fast caches to frequently accessed data

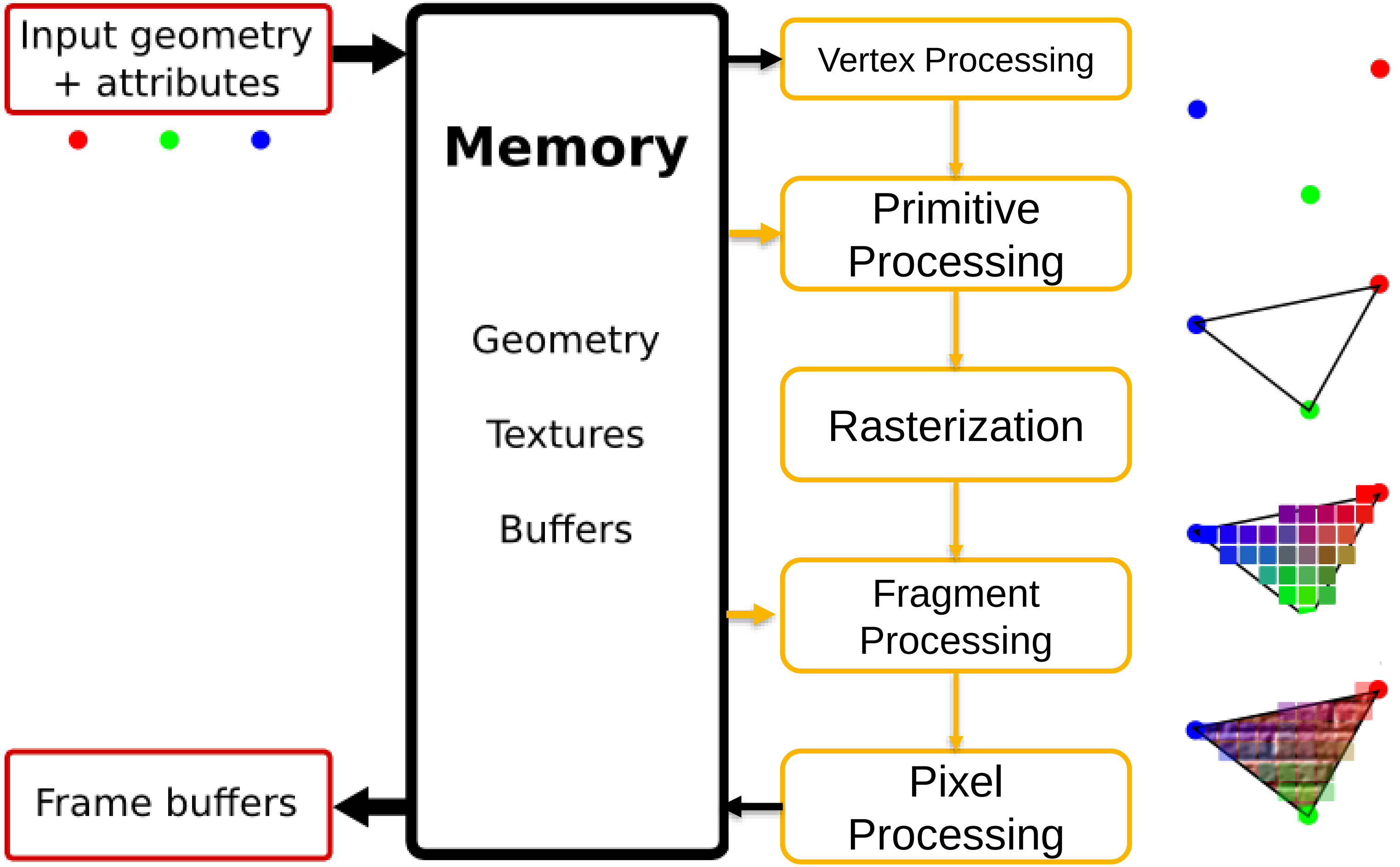
How to Render a 3D Frame with a GPU?



- Mesh triangles are projected on the screen plane
- Each pixel/fragment is then processed to compute its color

Figure taken from Romain Vergne online course -- http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

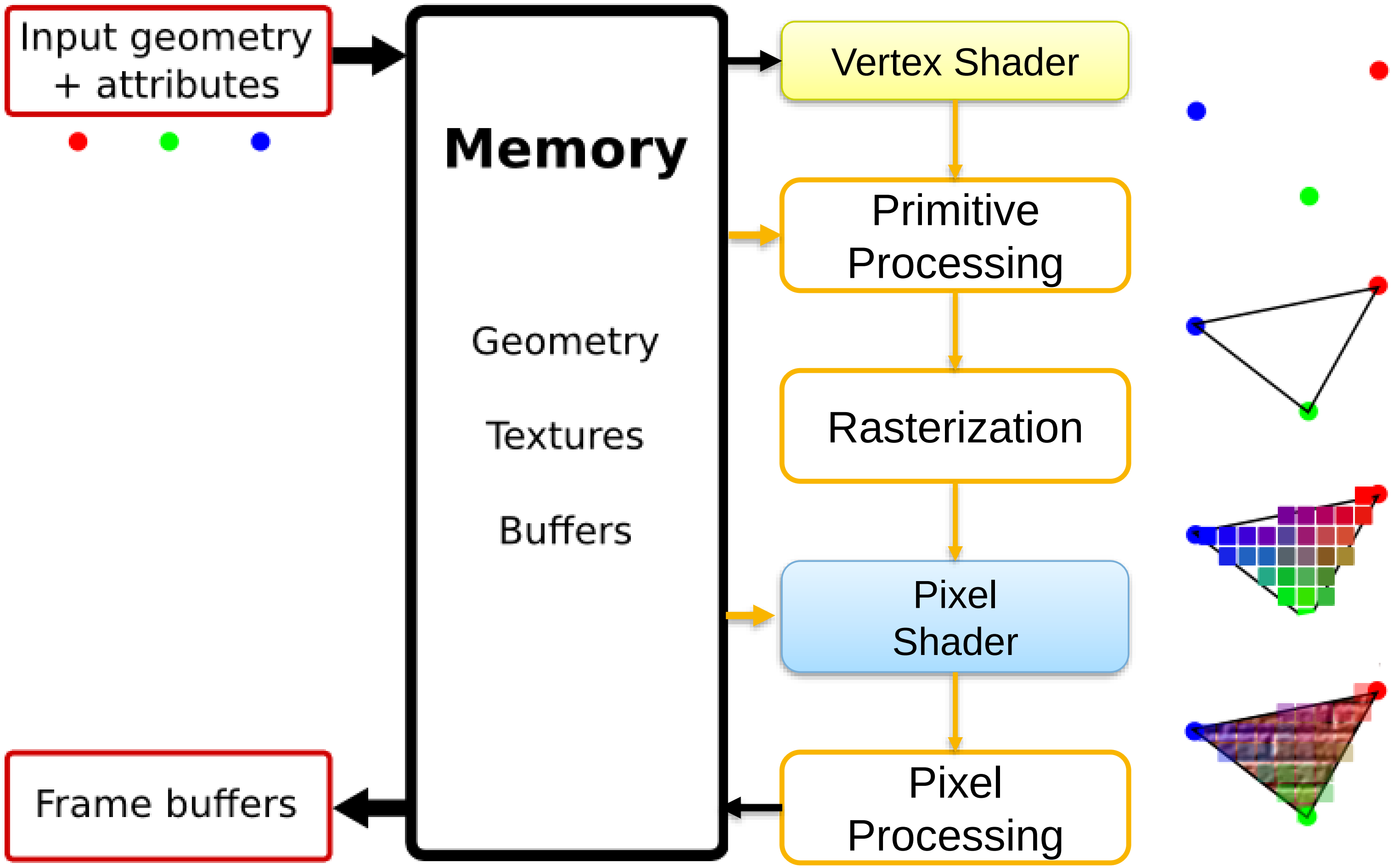
Fixed Function 3D Graphics Pipeline (OpenGL ES 1.x)



Pipeline Stage	Inputs Outputs	Details
Vertex	1 Vertex in 1 Vertex out	Transform and Lighting Attrib Assignment etc. Color, TexCoord
Primitive Processing	Vertices in Topology in Primitives out	Vertex assembly Perspective division Viewport transformation Clipping, Backface culling
Rasterization	Primitives in Fragments out	Fragment generation Multiple Fragment per pixel Assign fragment: color / texture / depth Interpolate btwn primitive
Fragment	Fragments in Pixels out	Apply Fog, Textures, Images Apply Stencil test Apply Depth test
Pixel Processing	Pixels in Pixels out	Apply Alpha Test Perform Blending Dithering

Taken from Romain Vergne online course -- http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

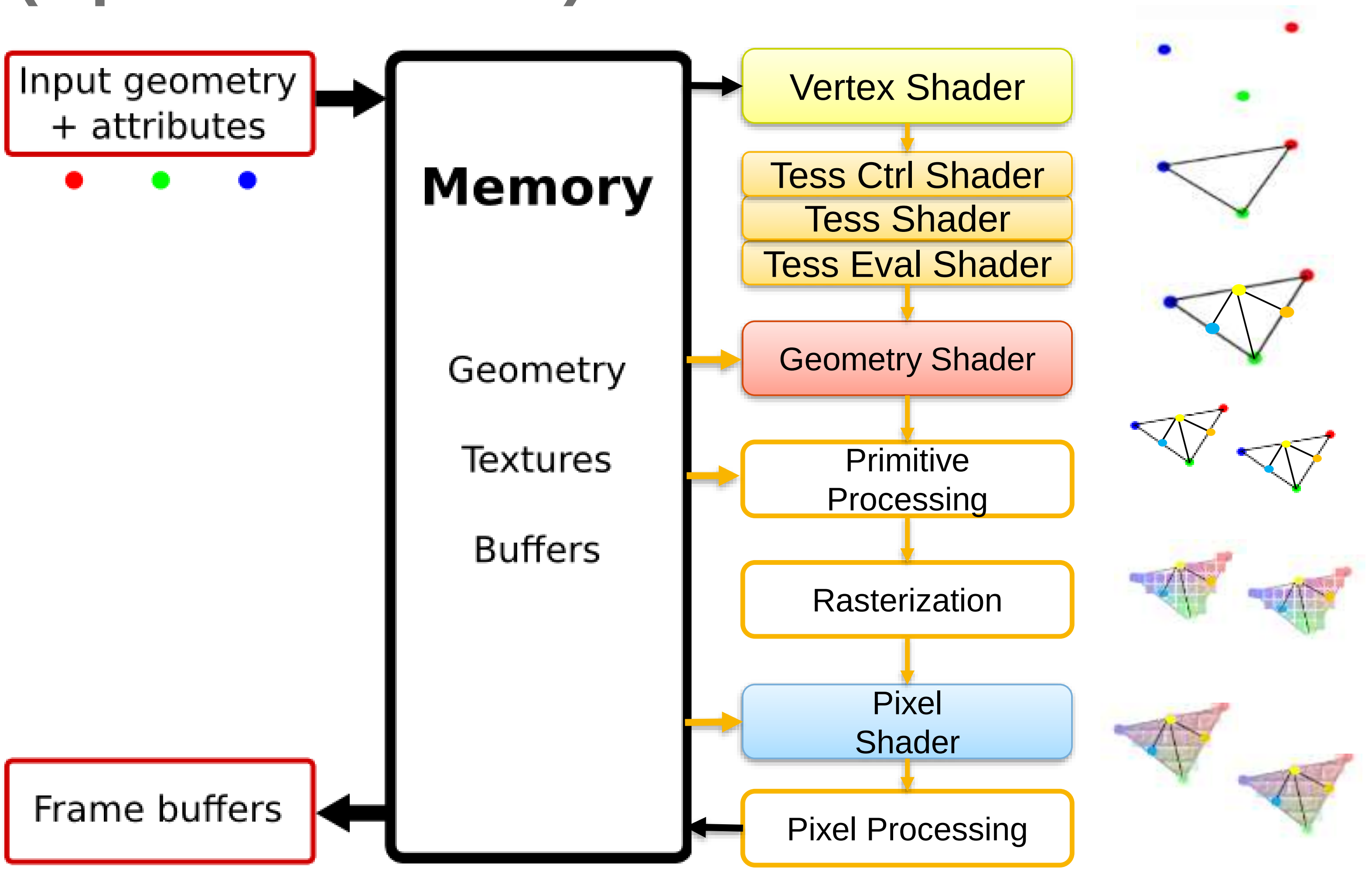
Shader Based 3D Graphics Pipeline (OpenGL ES 2.x)



Pipeline Stage	**Inputs Outputs**	**Details**
Vertex Shader	1 Vertex in 1 Vertex out	Transform and Lighting Attrib Assignment etc. Color, TexCoord **Programmable**
Primitive Processing	Vertices in Topology in Primitives out	Vertex assembly Perspective division Viewport transformation Clipping, Backface culling
Rasterization	Primitives in Fragments out	Fragment generation Multiple Fragment per pixel Assign fragment: color / texture / depth Interpolate btwn primitive
Pixel Shader	Fragments in Pixels out	Apply Fog, Textures, Images Apply Stencil test Apply Depth test **Programmable**
Pixel Processing	Pixels in Pixels out	Apply Alpha Test Perform Blending Dithering

Taken from Romain Vergne online course -- http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

Advanced Shader Based 3D Graphics Pipeline (OpenGL ES 3.1+)



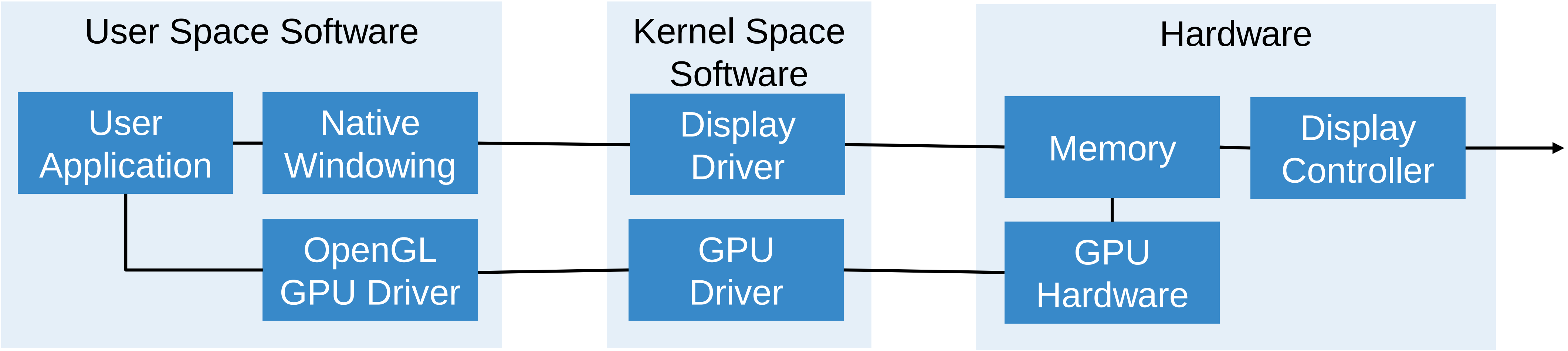
Pipeline Stage	Inputs Outputs	Details
Vertex Shader	1 Vertex in 1 Vertex out	Transform and Lighting Attrib Assignment etc. Color, TexCoord **Programmable**
Tessellation Shader	1 Vertex In 1 Control Param Many Vertex Out	Subdivide vertices using control parameters for higher LOD **Programmable**
Geometry Shader	1 Vertex in Many Vertex Out	Emit vertices relative to existing vertices. Add displacement to existing vtx **Programmable**
Primitive Processing	Vertices in Topology in Primitives out	Vertex assembly Perspective division Viewport transformation Clipping, Backface culling
Rasterization	Primitives in Fragments out	Fragment generation Multiple Fragment per pixel Assign fragment: color / texture / depth Interpolate btwn primitive
Pixel Shader	Fragments in Pixels out	Apply Fog, Textures, Images Apply Stencil test Apply Depth test **Programmable**
Pixel Processing	Pixels in Pixels out	Apply Alpha Test Perform Blending Dithering

Taken from Romain Vergne online course -- http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

How to write a simple GPU program?

- From software application writer point of view, there is 3 big parts:
  - **Native display/window/buffer management:** how to create memory regions, and show it on a display
  - **Setup and Draw commands**: executed from client app on CPU (OpenGL calls)
  - **Instructions executed in the GPU**: Vertex / Fragment shader program, GLSL

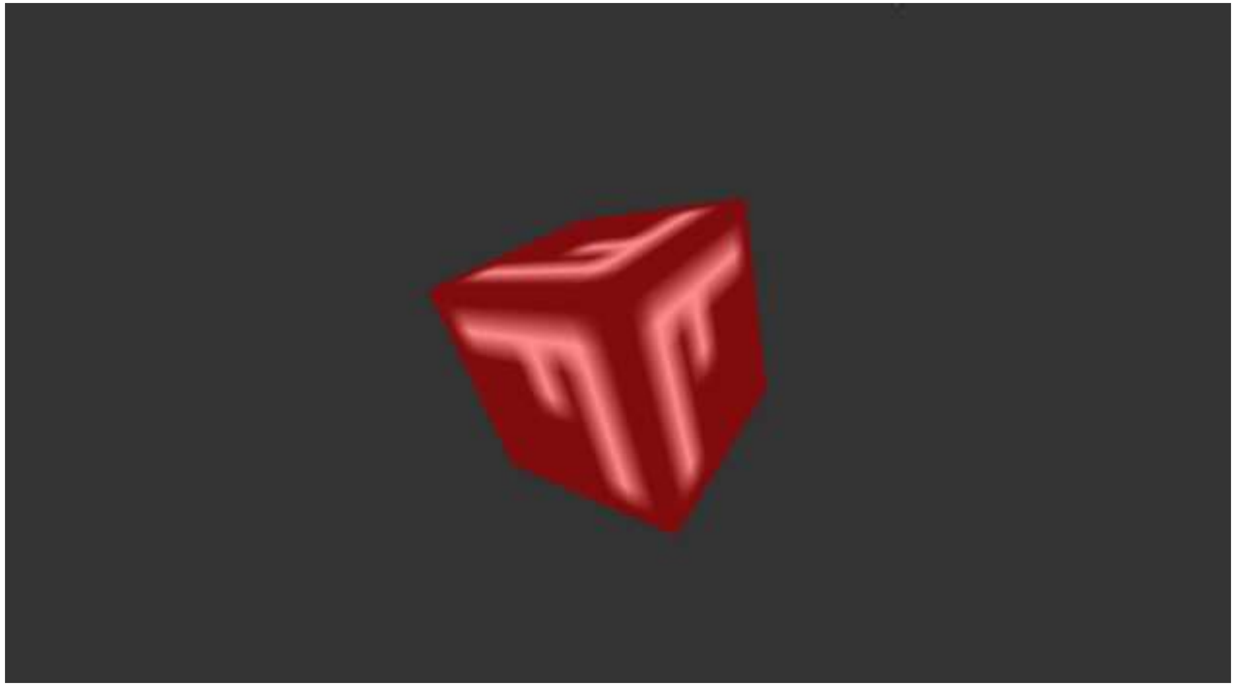
Components Involved in a Simple Rendering



Simple Cube Demo

- gles2cube: https://github.com/jolivain/gles2cube

```
git clone https://github.com/jolivain/gles2cube.git
cd gles2cube
autoreconf -vfi
./configure
make
./src/gles2cube
```



Rendering App Part 1: Native Windowing

- Before rendering anything, we need to get some memory space, and select a display
  - Example APIs: X11, Wayland, KMS/DRM/GBM, FBDev, proprietary systems, ...

- Khronos EGL API does the interface between the actual native windowing system, and an abstract one, which will be passed to rendering OpenGL ES APIs

- Usual windowing setup is:
  - Connect a native display
  - Create a native window and a surface buffer
  - Transform native data to EGL Abstract display/window
  - Create and setup an OpenGL rendering Context

Rendering App Part 1: Native Windowing – EGL

- See: https://github.com/jolivain/gles2cube/blob/master/src/egl_helper.c
- EGL Reference: https://www.khronos.org/registry/EGL/sdk/docs/man/
- All `native_gfx_*` functions are specific to the native windowing system
- Function `egl_init()`
  - Perform all native specific initialization to connect and get a display (`native_gfx_open_display()`) and `native_gfx_get_egl_native_display()`)
  - Convert it to an abstract "EGL Display" (`eglGetDisplay()`)
  - Initialize EGL (`eglInitialize()`)
  - Choose a configuration for your program (`eglChooseConfig()`)
  - Create and get a native window (`native_gfx_create_window()` and `native_gfx_get_egl_native_window()`)
  - Convert it to an abstract "EGL Surface" (`eglCreateWindowSurface()`)
  - Create a rendering context (`eglCreateContext()`)
  - Make it "current" (`eglMakeCurrent()`)

Rendering App Part 1: Native Windowing – X11 Example

- See: https://github.com/jolivain/gles2cube/blob/master/src/native_gfx_x11.c

- Open Display:
  - `XOpenDisplay()`

- Create Window:
  - `XCreateSimpleWindow()`

- More info: https://en.wikipedia.org/wiki/X_Window_System

Rendering App Part 1: Native Windowing - KMS / DRM / GBM Ex.

- See: https://github.com/jolivain/gles2cube/blob/master/src/native_gfx_kms.c

- Linux Kernel Subsystem exposing user APIs to interface with GPUs
- DRM: Direct Rendering Manager
- KMS: Kernel Mode Setting
- GBM: Generic Buffer Manager

- Exposes many features:
  - Multiple displays, connectors, planes, supported pixel formats, resolutions…
  - Accurate management of frame updates
- Preferred solution, when one application controls a display/plane
- More info: https://en.wikipedia.org/wiki/Direct_Rendering_Manager

Rendering App Part 1: Native Windowing – Wayland Example

- See: https://github.com/jolivain/gles2cube/blob/master/src/native_gfx_wl.c

- Open Display:
  - `wl_display_connect()`
  - `wl_display_get_registry()`

- Create Window:
  - `wl_compositor_create_surface()`
  - `wl_egl_window_create()`

- Preferred solution, when many applications are composited on display(s)
- More info: https://en.wikipedia.org/wiki/Wayland_(display_server_protocol)

Rendering App Part 2: OpenGL: Setup and Draw calls

- Configure rendering parameters
  - Provide data to be used by GPU: Textures, Mesh coordinates, shader programs
- Draw Calls, to start GPU instructions execution

- See: https://github.com/jolivain/gles2cube/blob/master/src/gles2cube.c
  - `setup()` function
  - `draw_frame()` function

- OpenGL is a state machine:
  - States changed only once can be set at setup time
  - Otherwise, they can changed at runtime during draw

Rendering App Part 3: Instructions running on GPU

- Draw commands from Part 2 will execute programmable stage on GPU
  - In our case: Vertex Shader, Fragment Shader

- See: https://github.com/jolivain/gles2cube/blob/master/src/gles2cube.c
  - Shader source code in variables:
    - `vertex_shader_g`
    - `fragment_shader_g`
  - Compiled and linked and loaded in:
    - `load_program()`, `load_shader()`
  - GPU Rendering started with "draw" calls, e.g. `glDrawElements()`

Rendering App Part 3: Instructions running on GPU

Vertex Shader Code

```
precision mediump float;

uniform mat4 u_mat;
attribute vec4 a_pos;
attribute vec2 a_uv;
varying vec2 v_uv;

void main() {
   gl_Position = u_mat * a_pos;
   v_uv = a_uv;
}
```

Fragment Shader Code

```
precision mediump float;

uniform vec4 u_color;
uniform float u_alpha;
uniform sampler2D u_tex;
varying vec2 v_uv;

void main() {
  vec4 t = texture2D(u_tex, v_uv);
  gl_FragColor = mix(u_color, t, u_alpha);
}
```

GPU are Complex systems

- GPUs are complex hardware that runs with complex software driver
  - Drivers expose and abstract low level functions
  - Application also needs complex engines on top of it

- An OpenGL driver is generally including a compiler for GPU shader cores
  - Ex: Mesa3D 19.1 is ~1.5M line-of-code

- The full stack (hardware + software) is hard to test

Testing and improving a GPU Driver and Hardware with Open Source Software

- Khronos OpenGL ES conformances: https://github.com/KhronosGroup/VK-GL-CTS
- Open source test suites: piglit: https://piglit.freedesktop.org/
- Video games, open source (ioquake, supertuxkart)
- Glmark2: https://github.com/glmark2/glmark2
- Web Browser based rendering:
  - glslsandbox: http://glslsandbox.com/
  - three.js: https://threejs.org/examples/
  - ShaderToy: https://shadertoy.com/
- ...
- Those systems requires many dependencies
  - Can be an issue for automated testing, or testing on production hardware

Why low level graphics?

- To understand what happens under the hood (e.g. write small regression test cases)
- Small memory footprints constraints (few MB of flash or RAM)
- Boot time constraints (3D graphics ready < 1s from power on)
  - We sometimes need to start graphics as soon as possible (with no other dependencies)
- For specific scenarios, we just need a boot loader, a kernel
- For performance (e.g. can’t afford the overhead of a large framework)

- Need to work on many OSes (Linux, Blackberry QNX, Green Hills INTEGRITY, ...), with sometimes many other constraints, ex:
  - no file system, no shared libraries, ISO C90/99, strict POSIX 2001/2008

- In final system integration, we want to test in conditions near to the production (i.e. a striped out system without instrumentation/development tools).

Introducing glslsandbox-player tool

- Simple standalone tool for rendering shaders using the http://glslsandbox.com/ API
- WebGL 1.0 is using the same shader language as OpenGL ES 2.0
- For stressing:
  - the fragment shader compiler a lot
  - hardware shader core execution of the GPU
  - memory buses and RAMs loads
- Perform GPU rendering mainly using fragment shaders
  - i.e. can do raytracing, for example
- For the geometry, only 2 big triangles are drawn, in order to render fragment on all the selected surface
- It's a generic tool for experimenting fragment shader code

Introducing glslsandbox-player tool

- Can use http://glslsandbox.com/ shaders from the community
- Can read local files, for experimenting
- Can embed shaders into the program binary (i.e. deploy is just a file copy)
  - No need of file system (can be a ramfs embedded in the Kernel itself)
  - No need of network, keyboard/mouse/touchscreen
- Just needs a libc, EGL, GLES2 driver, and a native windowing library
- Can compare embedded systems and implementations
- The player program is Open Source 2-clauses BSD, available on GitHub: https://github.com/jolivain/glslsandbox-player
- Shader has their own license. It's user responsibility to check each shader license
  - If in doubt (i.e. commercial usage), write your own shaders and choose your own license!
- Can be used to experiment performances of a GPU
  - Very easy to add support for new OS / native windowing systems

Testing glslsandbox-player on an Ubuntu 18.04 Linux Computer

- **WARNINGs**:
  - Make sure to save all your work, as the GPU/Computer may totally crash
  - Some shaders can be intensive: computer may become hot
  - On a laptop, it can be intensive on the battery, make sure to have a power plugged in!

```
git clone -b SophiaConf2019 https://github.com/jolivain/glslsandbox-player
cd glslsandbox-player
autoreconf -vfi
./configure
make -j$(nproc)
./scripts/run-demos-random.sh
```

- For curious people: code of a specific fragment shader can be printed on the console with:
  `./src/glslsandbox-player -W320 -H180 -S SimpleMandel -p`

- Alternatively, the on-line URL is also printed, ex: http://glslsandbox.com/e#51406.0
- The shader code can be edited live in web browser, and saved in a fork (new ID).
  (ex: try to change a value of outCol1)

About the OpenGL ES 2.0 Shading Language

- It’s a programming language on its own
  - Spec: https://www.khronos.org/registry/OpenGL/specs/es/2.0/GLSL_ES_Specification_1.00.pdf
- Inspired from C and C++ programming languages
- Includes standard preprocessor directives (CPP “`#define`”, “`#ifdef`”, …)
- Add features dedicated for graphic processing, for ex:
  - Specific qualifier (“`precision`”, “`uniform`”, …)
  - New data types (“`vec4`”, “`mat4`”)
  - Specific built-in functions (“`inversesqrt()`”, “`length()`”, “`dot()`”, “`mix()`”)
  - New CPP directives (“`#extension`”, “`#version`”)
- Some restrictions, for ex:
  - No recursion
  - “for” loops must have specific forms

Shader Programming Introduction 1

- Simple case of GL ES 2
  - There is language variations in OpenGL from 2.0 to 4.6, GL ES 3.x and Vulkan
- All demo files are in the “SophiaConf2019” directory
- Render shader examples with command:

```
./src/glslsandbox-player -W320 -H180 -F SophiaConf2019/01_simple_color.frag
```

- 1. Simple Color

https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/01_simple_color.frag

- 2. Introduce "uniform" / Changing Color with time

https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/02a_simple_changing_color.frag

https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/02b_simple_changing_color.frag

Shader Programming Introduction 2

- How to know which fragment we're processing?
  - 3a. Checker board
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/03a_checker_board.frag
  - 3b. Changing checker board size with time
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/03b_moving_checker_board.frag
  - 3c. Changing color with fragment coordinate
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/03c_frag_coord_color.frag

- Adding properties to vertex, interpolated to fragment:
  Varying Color demo
  - 4a/b Changing color with position varying
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/04a_varying.frag
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/04b_varying.frag

Shader Programming Introduction 3

- A bit more complex shaders:

- 5. Mandelbrot Set Fractal
  - Algorithm: https://en.wikipedia.org/wiki/Mandelbrot_set#Escape_time_algorithm
  - GLSL Fragment Shader: https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/05_mandel.frag
  - Example of a CPU only implementation of this example: https://github.com/jolivain/SophiaConf2019/blob/master/soft-render/sw-render.c

- 6. Animated Perlin Noise

https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/06a_perlin.frag

Shader Programming Introduction 4

- Some image processing examples:

- 7a. mixing animated pictures
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/07a_mix_images.frag
- 7b. Adding deformation
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/07b_mix_images.frag
- 8 CMOS sensor de-bayer
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/08_debayering.glslf
- 9 Spiral deformation
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/09_image-spiral.frag

- 10 Introduction to raymarching with NXP Logo
  https://github.com/jolivain/glslsandbox-player/blob/SophiaConf2019/SophiaConf2019/10_raymarched-logo.frag
  - GPU Gems 2 Ch 8: https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter08.html
  - Íñigo Quílez Articles https://www.iquilezles.org/

Shader Programming Introduction 5

- Some video processing with Gstreamer multimedia framework
- Scripts in: https://github.com/jolivain/SophiaConf2019/tree/master/gst-shader
- Simple camera example (without GPU processing)

```
gst-launch-1.0 -v \
  v4l2src device=/dev/video0 ! 'video/x-raw,width=640,height=480,framerate=30/1' ! \
  videoconvert ! autovideosink
```

- Camera image processing with GPU

```
gst-launch-1.0 -v \
  v4l2src device=/dev/video0 ! 'video/x-raw,width=640,height=480,framerate=30/1' ! \
  glupload ! \
  glcolorconvert ! \
  glshader vertex="\"$(cat shader.vert)\"" fragment="\"$(cat shader.frag)\"" ! \
  glimagesinkelement
```

- More complex examples at: https://github.com/jolivain/gst-shadertoy

Additional Documentation

- Book of Shader https://thebookofshaders.com/

- Real-Time Rendering http://www.realtimerendering.com/
  - And its book references: http://www.realtimerendering.com/books.html

- OpenGL ES Programming Guide 2nd edition
  - http://opengles-book.com/index.html

PART 2
EMBEDDED LINUX WITH BUILDROOT

Part 2: Buildroot - Embedded Linux Made Easy

- How to easily build and test it on an embedded system?
- Need to track all change (ex: work with Git)
- Need to be easy and fast to change and debug
- In doubt, need to be restart from a clean state
- Need to automatize builds for continuous integration
- Easy to change from one system on another
- Easily share with other people
- Need to handle the full toolchain

- Buildroot to the rescue!

Buildroot

- https://buildroot.org/
- It is a tool that simplifies and automates the process of building a complete Linux system for an embedded system, using cross-compilation.

- Buildroot is a build system tracking many aspects:
  - Host dependencies
  - Toolchain / Libc
  - Kernel and user space programs on target system
  - Final image construction

- Easy for both manual development, and automated continuous integration.
- A full system build is generally only 3 or 4 commands!

- System configuration is made with Kconfig
  (config system from the Linux Kernel and U-Boot).

Buildroot

- Buildroot provide many configurations that boot a Kernel with a simple Busybox environment, for a large selection of hardware and architectures
  - Board specific documentation: https://git.busybox.net/buildroot/tree/board
  - Configs: https://git.busybox.net/buildroot/tree/configs

- Those configuration can be used as a starting point.

- When your project configuration is ready, typical usage is as simple as:

```
git clone ...your-buildroot-repo-url...
cd buildroot
make mysystem_defconfig
make
dd if=output/images/sdcard.img of=/dev/sdX bs=1M oflag=sync
```

Buildroot Demo

- Demo for a QEMU virtual Aarch64 machine
- If everything went well, the commands executed at the preparation slide should have generated a full system image

- A sample script is provided to start QEMU with console-only support:

```
./start-qemu-aarch64.sh
```

- Actual command line is:

```
qemu-system-aarch64 \
    -M virt -cpu cortex-a53 -nographic -smp 4 \
    -kernel output/images/Image \
    -append "root=/dev/vda console=ttyAMA0" \
    -drive file=output/images/rootfs.ext4,if=none,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0
```

Creating a Buildroot configuration

- Step 1 prepare the base of the system:

```
git clone -b SophiaConf2019 https://github.com/jolivain/buildroot
cd buildroot
```

- Starting from a working configuration:

```
cp configs/qemu_aarch64_virt_defconfig configs/mysophiaconf2019_defconfig
# You can create a Git commit as a starting point here
make mysophiaconf2019_defconfig
```

- Launch menu configuration with (for console interface) :

```
make menuconfig
```

- or, for graphical interface:

```
make xconfig
```

Creating a Buildroot configuration

- Changing existing parameters in Buildroot, in menus:
  - Toolchain -> Toolchain type: Select "External"
  - Build options: check "Enable compiler cache"
  - Build options -> gcc optimization level: select "level 2"
  - System configuration -> /dev management: select "Dynamic using devtmpfs + eudev"
  - Filesystem images -> ext 2/3/4 root filesystem -> exact size: enter "120M"
  - Kernel -> Linux Kernel -> Kernel version: select "Latest version (5.1)"
  - Kernel -> Linux Kernel -> Kernel configuration: select "Use the architecture default configuration"
  - Target packages -> Graphics libraries and applications: check "mesa3d"
  - Target packages -> Graphics libraries and applications -> mesa3d: check "gallium virgl driver"
  - Target packages -> Graphics libraries and applications -> mesa3d: check "OpenGL ES"
  - Target packages -> Libraries -> Graphics -> libdrm: check "install test programs"
  - Save and Exit

Creating a Buildroot configuration

- The "save" button only save the current configuration. Changes are not reflected in the defconfig file. To include those changes in the defconfig, use the command:

```
make savedefconfig
```

- It’s also a good time to commit your changes with Git

- Build the image:

```
make
```

- This will take some time (depending on download size/time, compiler cache, number of package to build)

- When the build is finished, the machine can be booted as in the demo.

```
./start-qemu-aarch64.sh
```

Adding New Things in Buildroot – Kernel Specific Configuration

- Adding Virtual GPU support
  - Simply by enabling support in Kernel
  - This is done with KConfig fragment file
- Step 1: add the file(s) with the needed config changes
  - In buildroot, file: board/qemu/aarch64-virt/drm-virtio-gpu.fragment

```
CONFIG_DRM=y
CONFIG_DRM_VIRTIO_GPU=y
```

- Step 2: in Buildroot config menu, add the fragment:
  - Kernel -> Linux Kernel -> Additional configuration files: enter "board/qemu/aarch64-virt/drm-virtio-gpu.fragment"

- It's recommended to create Git commit at each steps to keep track of changes

Adding New Things in Buildroot – New Package 1/2

- Adding a new package for glslsandbox-player
- It's very easy! It's even easier if the package use well known build systems (Makefile, autotools, CMake, meson, ...) as Buildroot includes template for those

- Step 1: the Config.in (for the KConfig menu options)
  - For a simple package only one option for the whole package is sufficient
  - Also declare dependencies and requirements
  - Also expose all build options for the package
  - Finally, the new Config.in needs to be added in main package/Config.in to appear

- See: https://github.com/jolivain/buildroot/blob/SophiaConf2019/package/glslsandbox-player/Config.in

Adding New Things in Buildroot – New Package 2/2

- Step 2: Create the package build recipe .mk file, the build recipe
  - Declare package URLs, version
  - How to pass Buildroot options to the package build
  - Declare build dependencies, for the build order
  - Other customizations, if any
  - Use a Buildroot templates, if possible. In our case: "autotools-package“

- See: https://github.com/jolivain/buildroot/blob/SophiaConf2019/package/glslsandbox-player/glslsandbox-player.mk

- More details at: https://buildroot.org/downloads/manual/manual.html#adding-packages

Adding New Things in Buildroot – Add Files on Target

- Create a directory in Buildroot, adding files in it

```
mkdir -p my-rootfs-overlay
```

- Commit in Git

- In config menu: System configuration -> Root filesystem overlay directories: enter "my-rootfs-overlay“

- More details at: https://buildroot.org/downloads/manual/manual.html#rootfs-custom

Using Buildroot During Development

- Since Buildroot downloads everything online, the code needs to be already published.
- When we develop, we want to test code before publishing it.
- But we still want the benefits of Buildroot
- It's possible to override source directories for some package
  - Just create a "local.mk" file at Buildroot top level directory and define variable, ex:

```
LINUX_OVERRIDE_SRCDIR = /home/user/work/linux/
GLSLSANDBOX_PLAYER_OVERRIDE_SRCDIR = /home/user/work/glslsandbox-player/
```

- For details, see Buildroot documentation: https://buildroot.org/downloads/manual/manual.html#_using_buildroot_during_development

Buildroot Conclusion

- More info at: https://buildroot.org/
- Full manual: https://buildroot.org/downloads/manual/manual.html

- Bootlin online training material on Buildroot:
  - https://bootlin.com/training/buildroot/
  - https://bootlin.com/doc/training/buildroot/buildroot-slides.pdf

- NOTE: Those compilations created the directory "`${HOME}/.buildroot-ccache/`“ for the compilation cache. When finished, you can remove this directory, to reclaim the storage space.

PART 3
QEMU GPU EMULATION

GPU Virtualization with QEMU

- QEMU is a generic and open source machine emulator and virtualizer

https://www.qemu.org/

- Why working in an emulator?
  - Work on software before having the final hardware ready
  - Scale up software team (some developer may work without the final hardware)
  - Automated testing
  - Faster deployment (some systems are long to flash)
  - Share system with people
  - …

GPU Virtualization with QEMU

- With the Virgil 3D project, it's possible to emulate a virtual 3D GPU in the guest OS, accelerated on the actual host GPU.

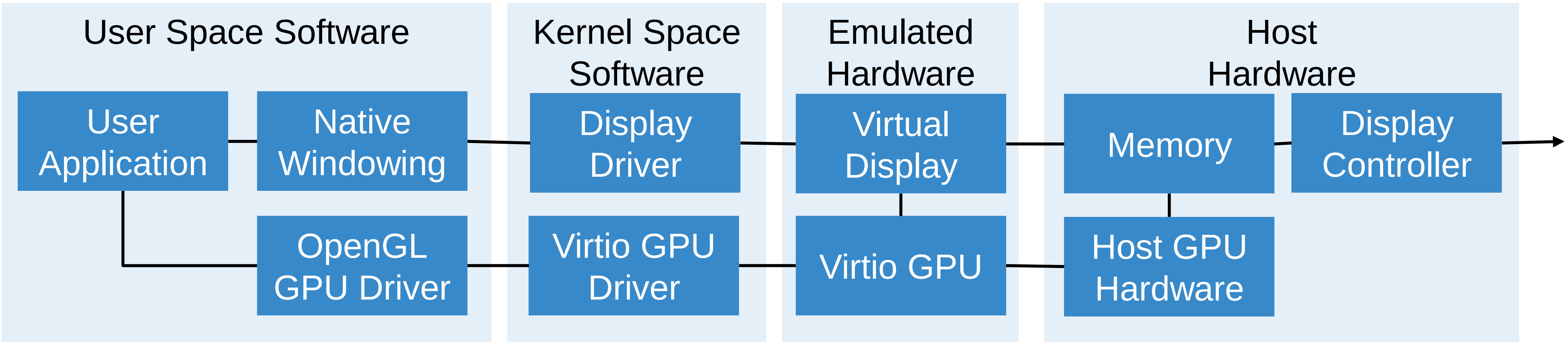
https://virgil3d.github.io/

- It is mostly based on OpenGL and work like remote rendering
  - There is no specific requirements regarding virtualization on host GPU

- Linux kernel: DRM VirtIO GPU
- Mesa3D: VirGL OpenGL driver

- Rendering may not be exactly the same as the actual hardware!
  - But this is not a problem, most of the time.

Components Involved in a Virtual GPU Rendering



How to Fully Enable Virgl 3D in QEMU? The Host Part

- Host requirements:
  - A real GPU and its driver
  - Needs Mesa3D >= 10.6 with EGL_MESA_image_dma_buf_export
  - Needs libvirglrenderer

- QEMU >= 2.6 needs to be configured at build-time with
  - Option: `--enable-gtk` for GUI (also works with SDL)
  - Option: `--enable-opengl`
  - Option: `--enable-virglrenderer`

- When starting qemu, the GPU device need declared, and OpenGL display needs to be enabled too, with command line options:
  - Option: `-device virtio-gpu-pci,virgl`
  - Option: `-display gtk,gl=on`

How to Fully Enable Virgl 3D in QEMU? The Guest Part

- Guest Requirements:

- Linux Kernel needs:
  - Version >= 4.4 (for VirtIO GPU support)
  - DRM (Direct Rendering Manager) support enabled: `CONFIG_DRM`
  - virtio-gpu enabled in configuration: `CONFIG_DRM_VIRTIO_GPU`

- Mesa3D:
  - Version >= 11.1
  - compiled with option `--with-gallium-drivers=virgl`

How to Fully Enable Virgl 3D in QEMU?

- QEMU Aarch64 on Ubuntu 18.04 does not seem to have VirGL support
- So let’s compile it:

```
wget https://download.qemu.org/qemu-3.1.0.tar.xz
tar xf qemu-3.1.0.tar.xz
cd qemu-3.1.0
./configure \
  --target-list=aarch64-softmmu \
  --enable-opengl \
  --enable-virglrenderer \
  --enable-gtk \
  --enable-vte
make -j$(nproc)
```

- Output binary is in: `aarch64-softmmu/qemu-system-aarch64`
- Set the PATH to use this new binary: `export PATH=$PWD/aarch64-softmmu:$PATH`

Running Buildroot System in QEMU + VirGL

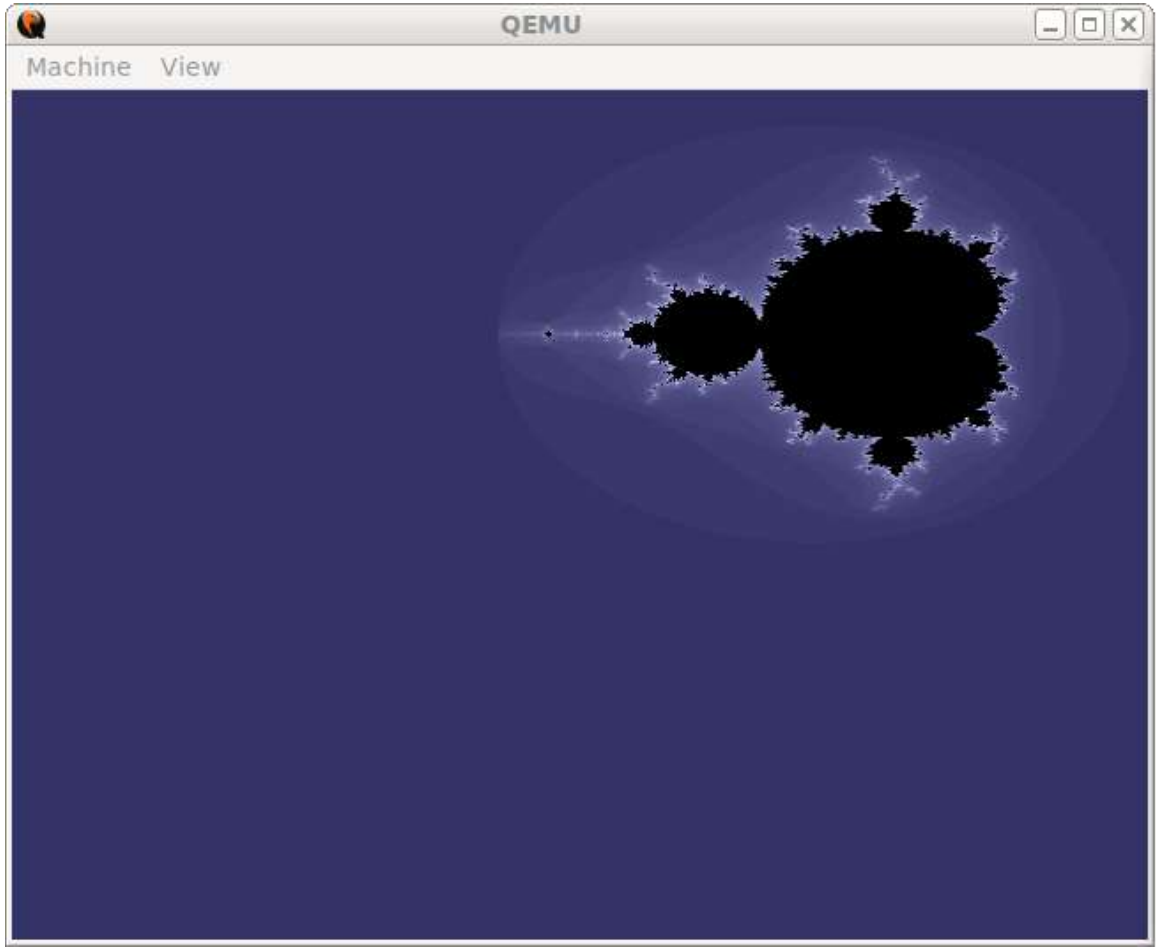
- Demo start script is:

```
./start-qemu-aarch64-virtio-gpu.sh
```

- Start QEMU with command line:

```
qemu-system-aarch64 \
     -M virt \
     -cpu cortex-a53 \
     -smp 4 \
     -m 2048 \
     -device virtio-gpu-pci,virgl \
     -kernel output/images/Image \
     -append "root=/dev/vda console=ttyAMA0" \
     -drive file=output/images/rootfs.ext2,if=none,format=raw,id=hd0 \
     -device virtio-blk-device,drive=hd0 \
     -display gtk,gl=on
```

Running QEMU + VirGL



QEMU: serial0

```
EGL_VERSION                    : 1.4

OpenGL ES driver information:
GL_VENDOR                      : Red Hat
GL_RENDERER                    : virgl
GL_VERSION                     : OpenGL ES 3.1 Mesa 19.0.3
GL_SHADING_LANGUAGE_VERSION : OpenGL ES GLSL ES 3.10

Running shader "SimpleMandel" (GLSL Sandbox ID: 51406.0, builtin ID: 0)
Available online at: http://glslsandbox.com/e#51406.0
PLEASE make sure to check original license and give credit to the original author(s).
Frame rate will be reported every 100 frames
Rendering on a 640x480 window
Setup time 0.365 sec (context init, shader compilation)
First frame time 19.745 ms (50.646 fps)
finished 3 warmup frames in 0.039 s (avg rate 77.218 fps).
Excluding first warmup frame: 2 warmup frames in 0.020 s (avg rate 101.062 fps).
Using origin of time: 1561036610.073620560
from_frame:0      to_frame:99     time:1.666   frame_rate:60.038    shadertime=1.685
from_frame:100    to_frame:199    time:1.663   frame_rate:60.120    shadertime=3.348
from_frame:200    to_frame:299    time:1.666   frame_rate:60.026    shadertime=5.014
from_frame:300    to_frame:399    time:1.665   frame_rate:60.051    shadertime=6.679
from_frame:400    to_frame:499    time:1.664   frame_rate:60.079    shadertime=8.344
from_frame:500    to_frame:599    time:1.664   frame_rate:60.092    shadertime=10.008
from_frame:600    to_frame:699    time:1.666   frame_rate:60.036    shadertime=11.673
from_frame:700    to_frame:799    time:1.665   frame_rate:60.062    shadertime=13.338
from_frame:800    to_frame:899    time:1.665   frame_rate:60.056    shadertime=15.004
```

QUESTIONS?

SECURE CONNECTIONS
FOR A SMARTER WORLD