Open GL ES

Introduction

- 3D graphics for embedded systems
  - Smart phones
  - Pads
  - Portable Multimedia Systems
  - Gaming consoles (both portable and stationary)
  - Settop boxes

- Motivation
  - Mobile gaming (iOS, Android) fast growing market
  - Portable gaming consoles (Nintendo 3DS, Playstation Vita)
  - Also: stationary consoles (Ouya)

Introduction

- Example architectures
    - Imagination Technologies PowerVR (market leader)
    - ARM Mali
    - Qualcomm Adreno (former: by ATI)
    - NVIDIA Tegra (caution: no unified architecture!)

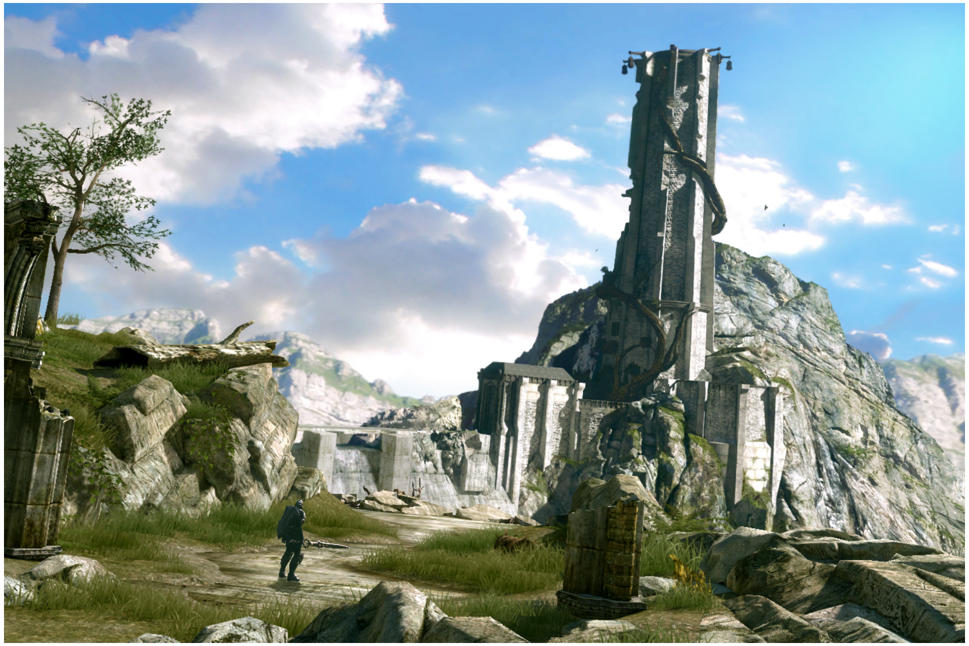
- Two flavors
    - OpenGL ES 1.x: fixed pipeline
    - OpenGL ES 2.0/3.0: shaders
    - Not compatible with each other!

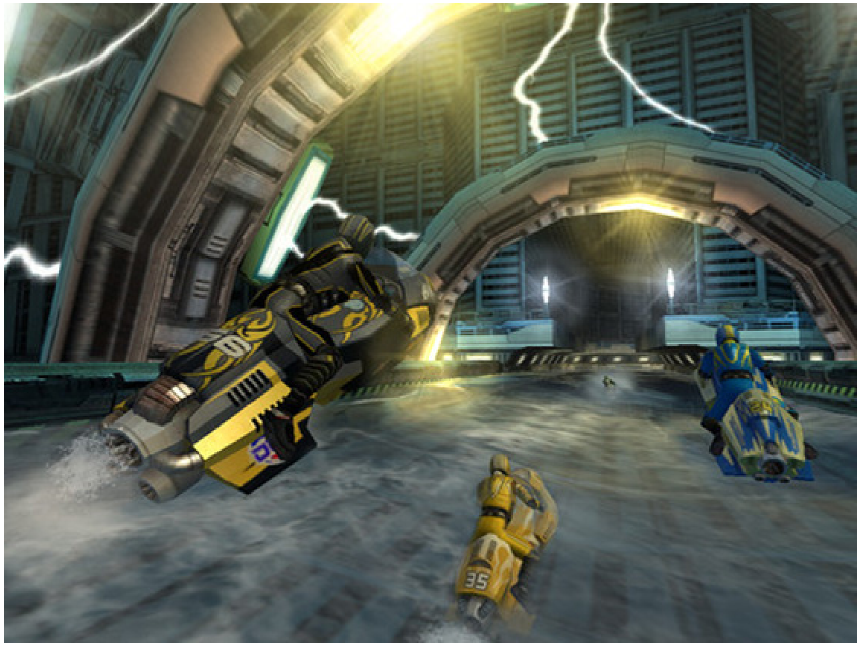
Introduction

- OpenGL ES 1.x
  - ❑ Android since 1.6
  - ❑ iOS
  - ❑ Nintendo 3DS
  - ❑ Playstation 3 (supports parts from Open GL ES 2.0 as well)

- OpenGL ES 2.0
  - ❑ iOs (since Iphone 3GS)
  - ❑ Android (since 2.0)
  - ❑ Playstation Vita
  - ❑ Chosen as basis for WebGL

Examples





Left: screenshot from Horn, right: screenshot from Riptide

Why OpenGL ES?

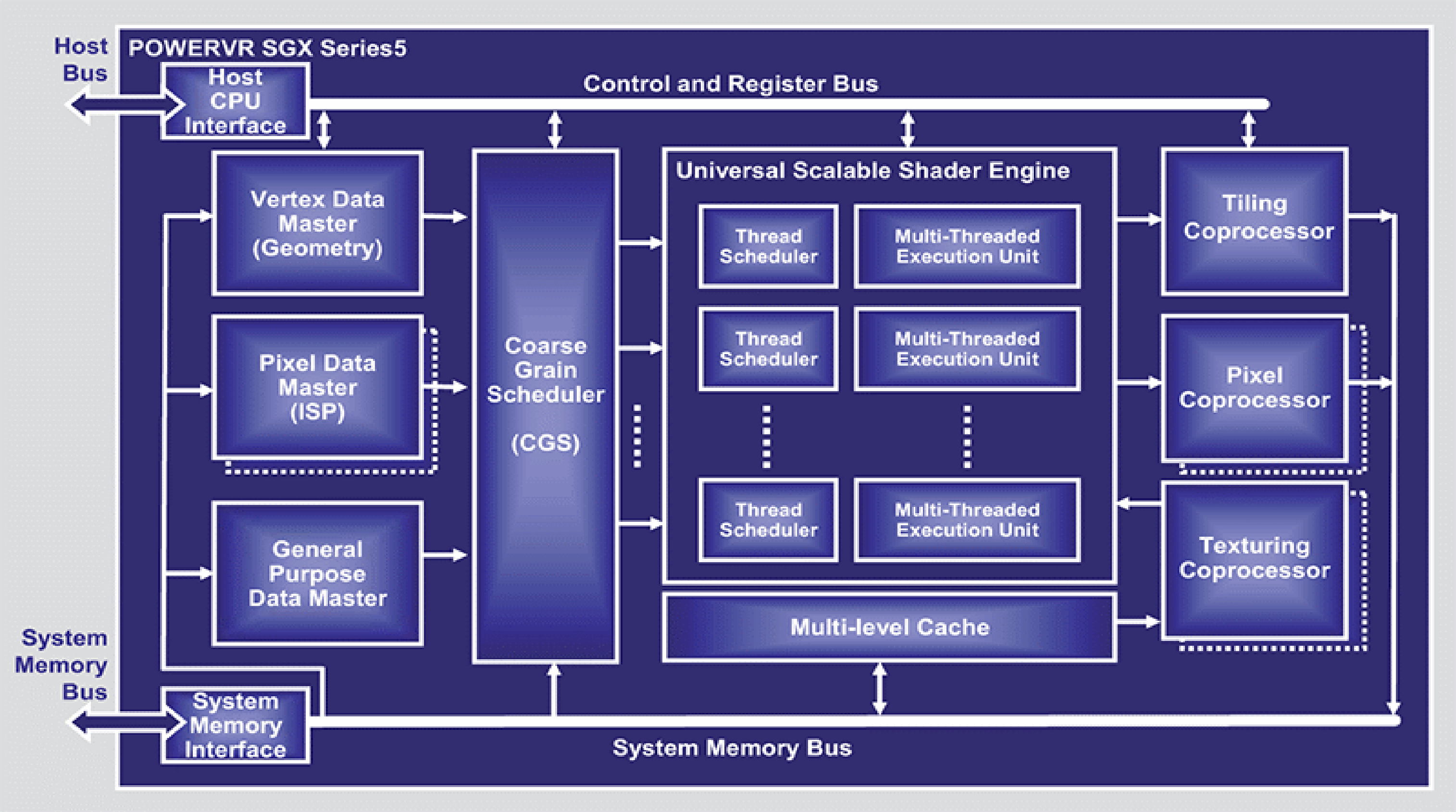
- Many of these systems support “normal” OpenGL as well, but...
  - Not all of them
  - OpenGL ES designed with embedded systems in mind
    => reaches higher performance
- The only reason to use “normal” OpenGL is when you need a feature not included in OpenGL ES
  - But beware: there is probably a good reason why it is absent

Embedded systems

- Low performance (compared to PC)
  - Cost
  - Runtime (note that battery technique hasn't improved much in recent years
  - No active cooling! (Otherwise too big)
- High Resolution
  - iPad(4): 2048x1536
  - Nexus 10: 2569x1600

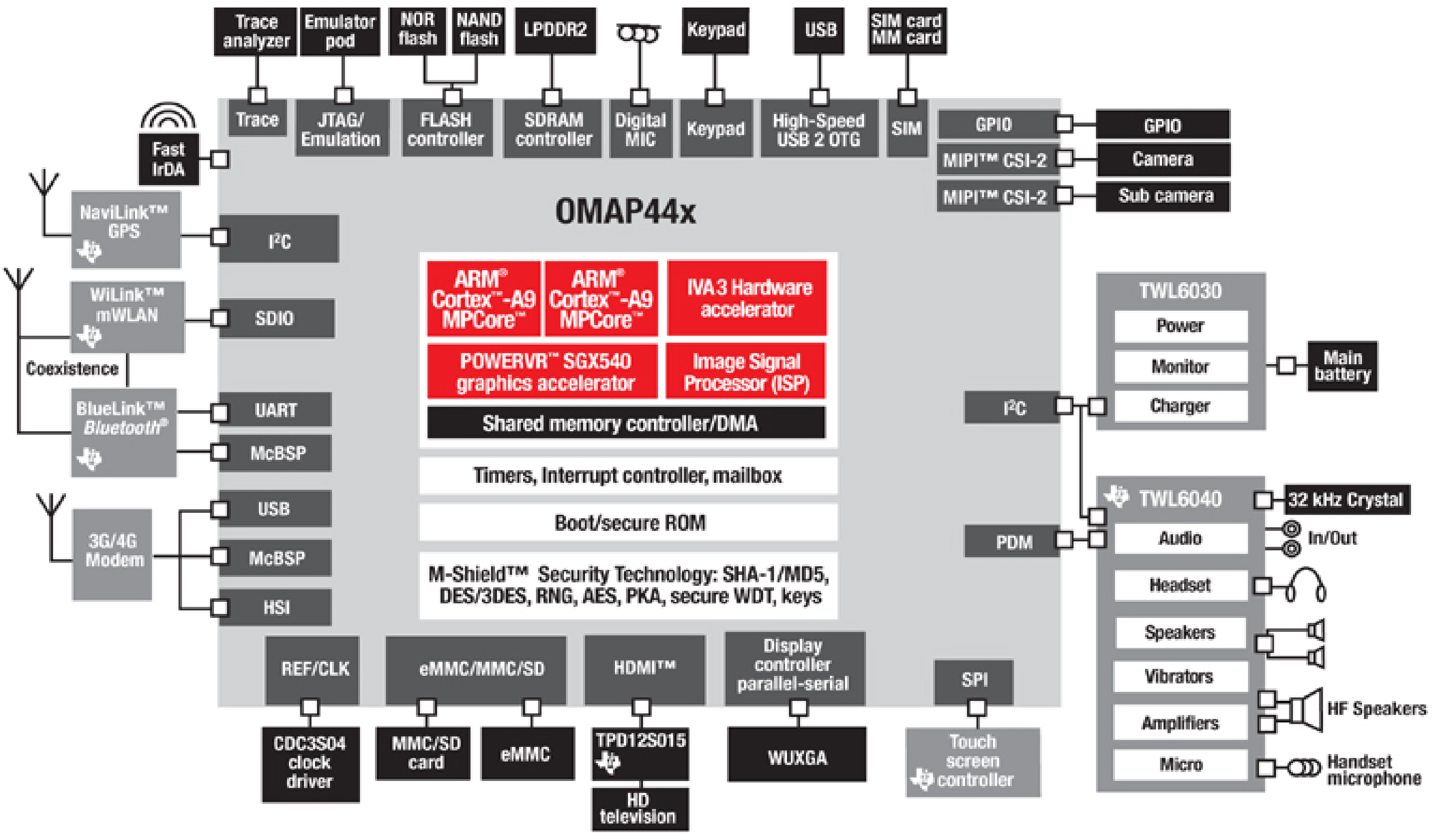
=> Specialized solution needed!

Embedded systems



Found in: iPhone (since version 4), iPad (since version 2), Nexus S, Samsung Galaxy S, Samsung Galaxy Tab, Sony Ericsson Vivaz, Nokia N900, Playstation Vita, among many others

Just to make a point



Embedded systems

- CPU, GPU, hardware accelerators, interfaces, ...
  - All share same bus and memory
    => bottleneck!
  - Not likely to change: energy optimized architecture

OpenGL ES

- “streamlined” OpenGL
  - ❑ Removed obscure methods
  - ❑ Optimize existing methods for low pow performance hardware
  - ❑ Introduce new specialized methods and data structures

- Based on OpenGL 1.3 (OpenGL ES 1.x) resp. OpenGL 2.0 (OpenGL ES 2.0, but is closely related to OpenGL 3.0)

- OpenGL ES 3.0: basically OpenGL ES 2.0, but with extensions to make it more flexible

Differences to OpenGL3.0

- No geometry or tesselation shader
  - OpenGL ES 1.x: no shader at all
- No anti-alias (would cost too much memory)
- Scissor buffer
  - Like stencil buffer, but only for rectangles => much faster

Differences to OpenGL3.0

- Only 2D textures
  - No 3D textures for particle effects like smoke, fire, water
  - 3D textures introduced in OpenGL ES 3.0, but I discourage strongly to use them
- Better support for texture compression
  - Lossy compression, typically 30 db PSNR @ 1:6 compression
  - Very low decoding complexity, decoding “on-the-fly”
  - Most architectures support it in hardware

Differences to OpenGL3.0

- No geometry or tesselation shader
  - OpenGL ES 1.x: no shader at all
- Need to declare precision for shader variables and functions

Example

Open GL 3.0

```
uniform sampler2D tex;
in vec2 coord;

out vec4 outColor;

void main(void)
{
  outColor=texture(tex,coord);
}
```

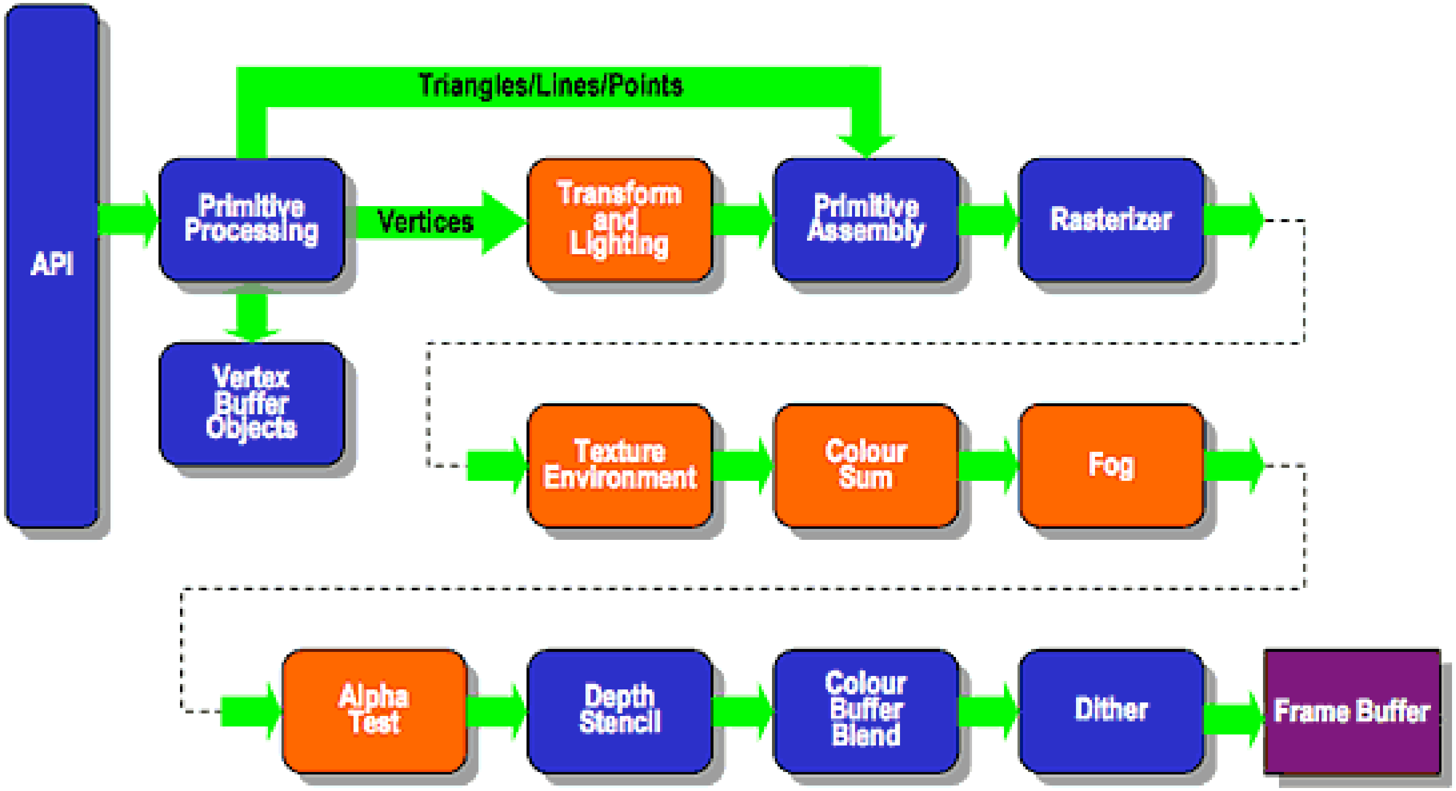
OpenGL ES 2.0

```
precision mediump float;
uniform sampler2D tex;
varying vec2 coord;


void main(void)
{
  gl_FragColor=texture2D(tex,coord);
}
```

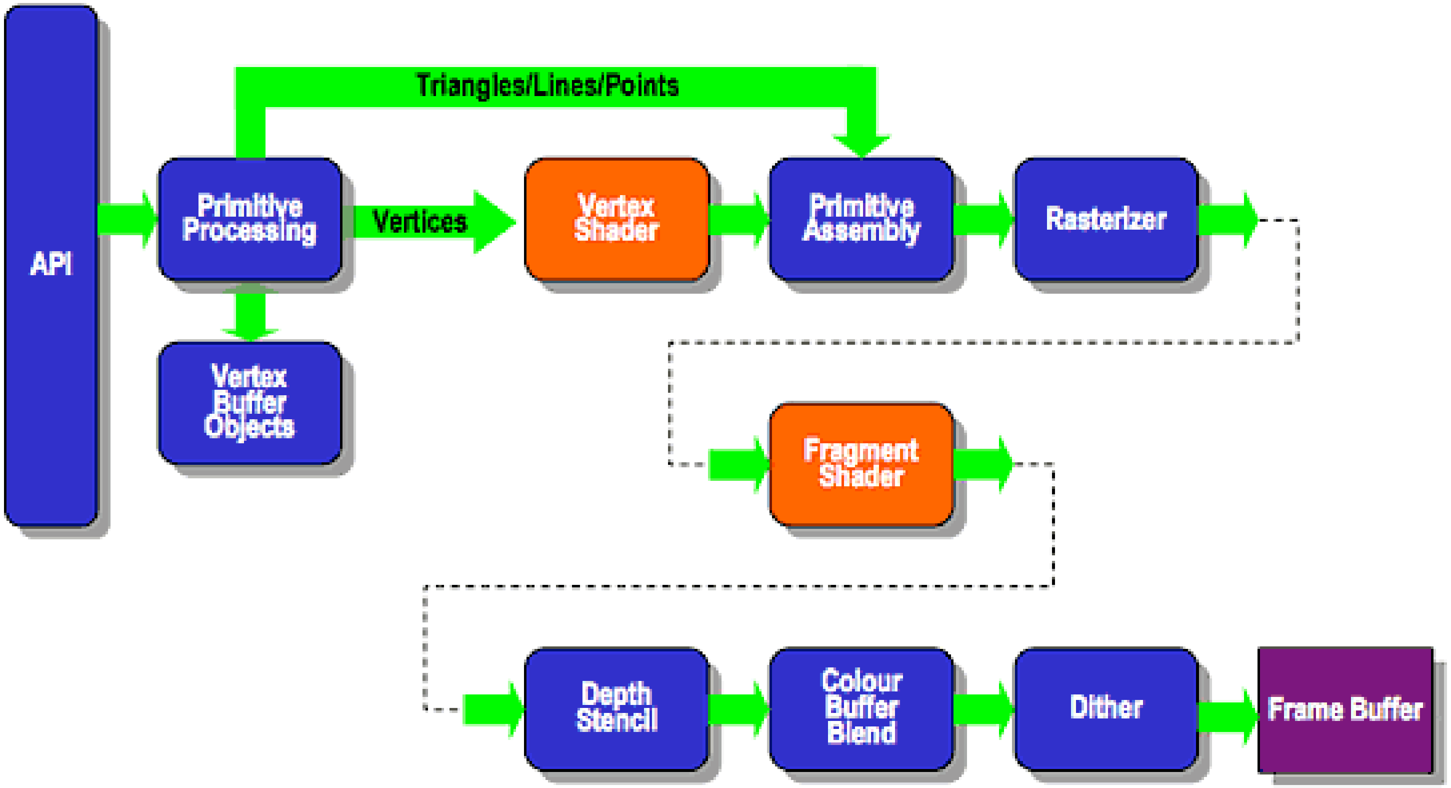
Which OpenGL ES?

OpenGL ES 1.x Pipeline



Which OpenGL ES?

OpenGL ES 2.0/3.0 Pipeline



Which OpenGL ES?

- OpenGL ES 1.x
  - For very low complex hardware
  - Might seem to be easier: no shader programming needed
  - But in reality: needs fiddling to get the right effect, if at all possible

- OpenGL ES 3.0
  - Not widely supported yet
  - You might need some of its new functionality though

=> OpenGL ES 3.0 safest bet right now

Design guideline

- Be much more performance aware
  - Reuse shaders whenever possible
  - Avoid branches (ifs), unroll loops
  - Often: rather recomputation than additional memory accesses
  - Texture compression often supported by hardware, therefore “for free”,but be careful if you are using the texture not as a picture, but as a cheap way to send data to the GPU
  - Use only as high precision as needed, prefer fixpoint
  - Don't use dynamic textures or array index calculation in the shader
  - Redraw only as much as needed
  - Think twice before using framebuffers, pingponging etc.
- But you can of course bend the rules, just make sure you know what you are doing!

Pictures from the demo