FTF | FREESCALE TECHNOLOGY FORUM
POWERING INNOVATION

Graphics Development on the i.MX 6 Series

FTF-CSD-F0049

Brian Murray
i.MX Graphics Architect

June 2012

Session Introduction

This session covers the basics of using the GPUs and graphics APIs on i.MX 6 and gives the audience a hands on experience building these applications.

What you need to know:

- C & C++ programming skills
- Basic Linux / GCC / Make usage (for building the samples)

This will give the audience the basis for further graphics application development and debugging. This course is a broad introduction to these topics and not a comprehensive computer graphics course.

This session is 2 hours.

Agenda

- Overview of i.MX 6x GPUs
- Quick Overview of 3D Graphics
- EGL and Windowing Systems
- OpenGL ES 1.1 Overview and Samples
- OpenGL ES 2.0 Overview and Samples
- OpenVG 1.1 Overview and Samples
- Overview of GPGPU
- OpenCL Overview and Samples
- PC Tools
- Graphics Debug Tips & Tricks

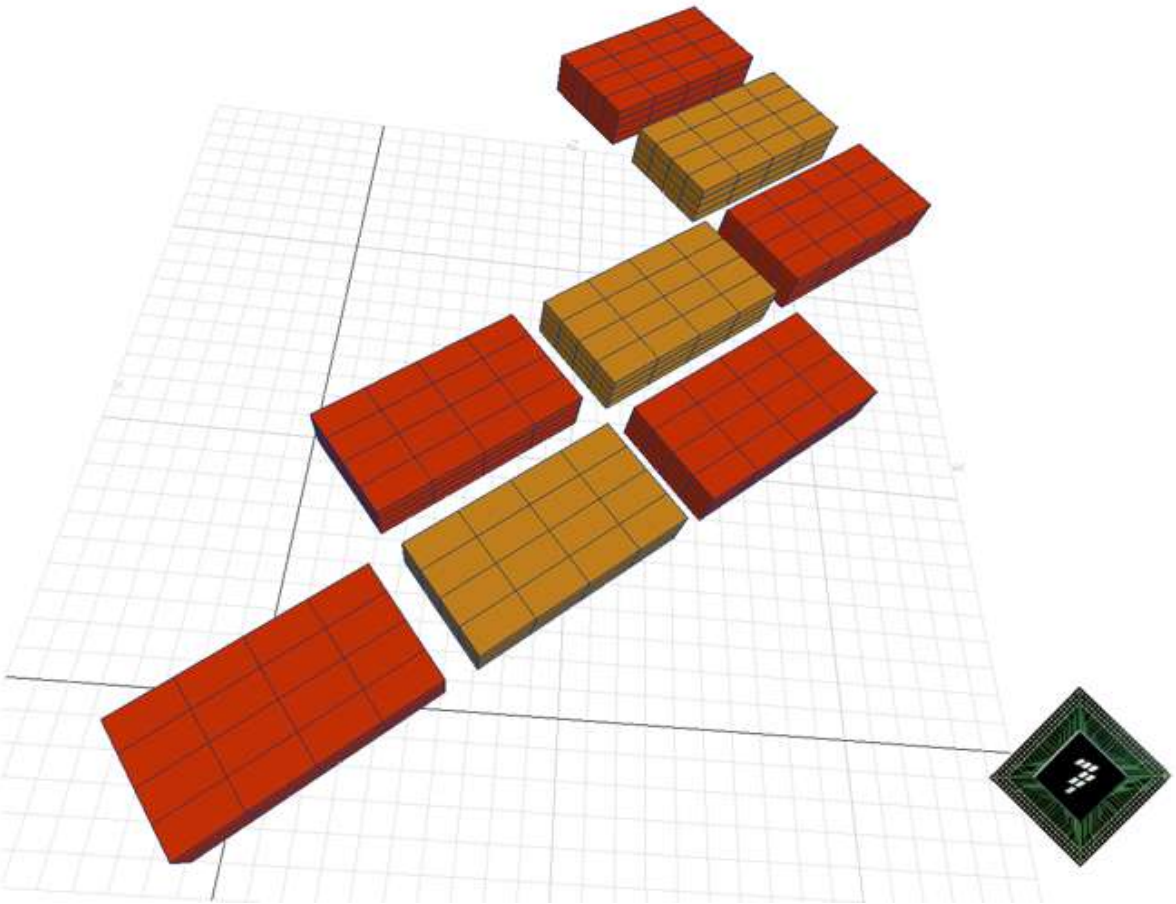
Hands-on Workshop + Lecture

- This course is both a lecture on GPU programming and a Hands-on workshop
- Throughout the course, I will point out API samples in the slides and give you a chance to build the code
- While you are building, I may proceed to talk more about the fundamentals for the sake of time
  - Do not feel obligated to stop tinkering with the code
- Toward the end of the course, we will try some more active debugging and development

Samples Intro

- OpenGL ES 1.1, 2.0, OpenVG, and OpenCL Samples on your i.MX 6Q Sabre-Lites
- Use editor of choice to edit source files
- Make files have a separate rules file
  - Easier for porting / cross compiling

Overview of i.MX 6x GPUs



i.MX 6 Series: Graphics Processing Goals

- **Diverse markets** and applications call for **multiple GPUs** :

- **Accelerate** existing **windowing environments** (Android, X11, etc.) and base user interfaces with a low-power high-performance composition engine

- Providing **compelling vector capabilities** for high-quality fonts and automotive clusters in a manner that assures quality of service in performance

- Accelerate **next-generation 3D** and GPGPU applications with a feature-rich high-performing unified graphics processor

i.MX 6D/Q Graphics Processing Units

Introducing the **i.MX** Triple-Play Graphics Solution

- **High-Performance OpenGL/OpenCL GPU Core**
  - Vivante GC2000 @ 528MHz (600 MHz shader)
  - 88M triangles / sec (176MT/s @ 50% cull)
  - 4 Shader Cores: 21.6 GFLOPS
  - 1056 M Pixels/s, 1056 M Texels/ s

- **High-Performance Composition Engine 2D Core**
  - Up to 633M pixels / sec raw performance
  - Vivante GC320

- **High-Performance Vector Graphics Core**
  - 264M Pixels / sec raw performance
  - Native OpenVG 1.1 vector acceleration with hardware tessellation
  - Vivante GC355

i.MX 6DL/S Graphics Processing Units

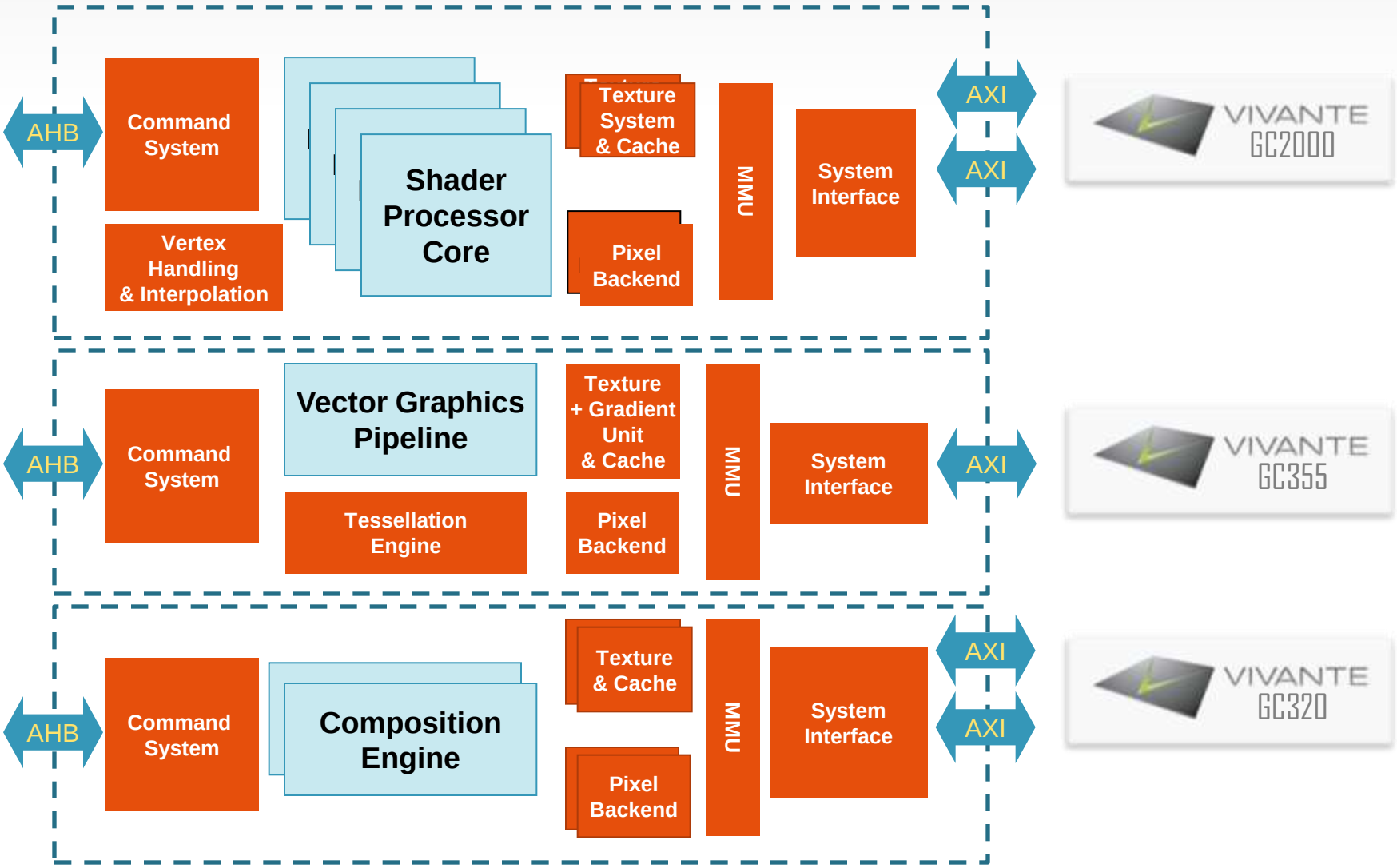
- **High-Performance OpenGL GPU Core**
  - Vivante GC880 @ 533MHz
  - 27M triangles / sec (53MT/s @ 50% cull)
  - 1 Shader Cores: 4.8 GFLOPS
  - 266 M Pixels/s, 266 M Texels/ s

- **High-Performance Composition Engine 2D Core**
  - Up to 633M pixels / sec raw performance
  - Vivante GC320

i.MX 6 Series: Graphics Processing Advantages

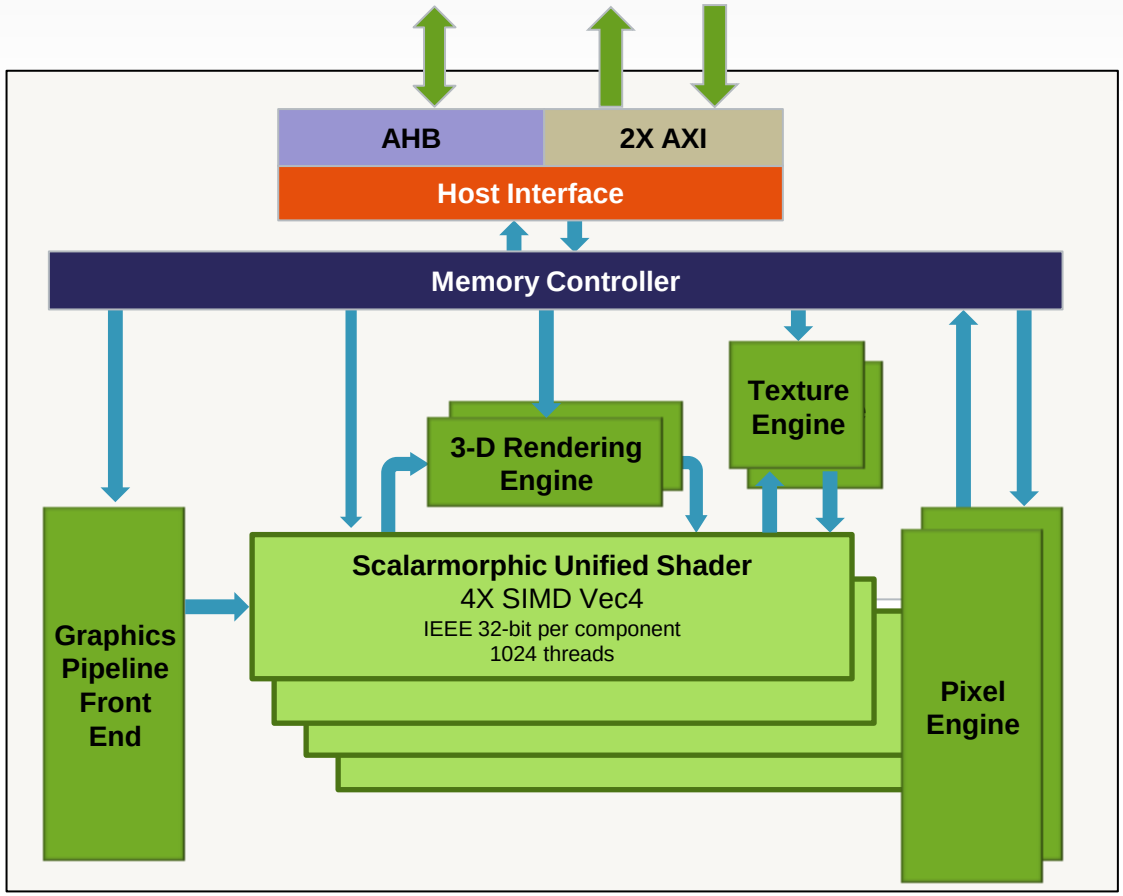
- **Independently clocked and powered**
  - i.e. 3D / VG clocked and powered only when needed
- **Completely separate bus masters allow OSs and users to define QoS at a core / API level**
  - Supplements driver level job interleaving / prioritization
- **Maximizes system bandwidth / processing**
  - Data is sharable between cores and operable in parallel
- **Best of both worlds**
  - Customer no longer has to choose between doing GPGPU functions and accelerating their UI
- **Complete customer freedom in UI acceleration strategy**

High-Level GPU Block Diagram



Vivante GC2000 ScalarMorphic™ Architecture

- Architectural Features
- CoolThreads™ multilevel clock gating
- Massive data parallelism
- 4X SIMD Vec-4 unified shaders (IEEE single precision FP)
- High-performance dependent texture operation
- Point sample, bi-linear/tri-linear and cubic/volumetric textures
- Balanced performance/ bandwidth
- Memory locality
- Early/hierarchical z
- Fast clear
- Texture compression
- Frame buffer/Z compression



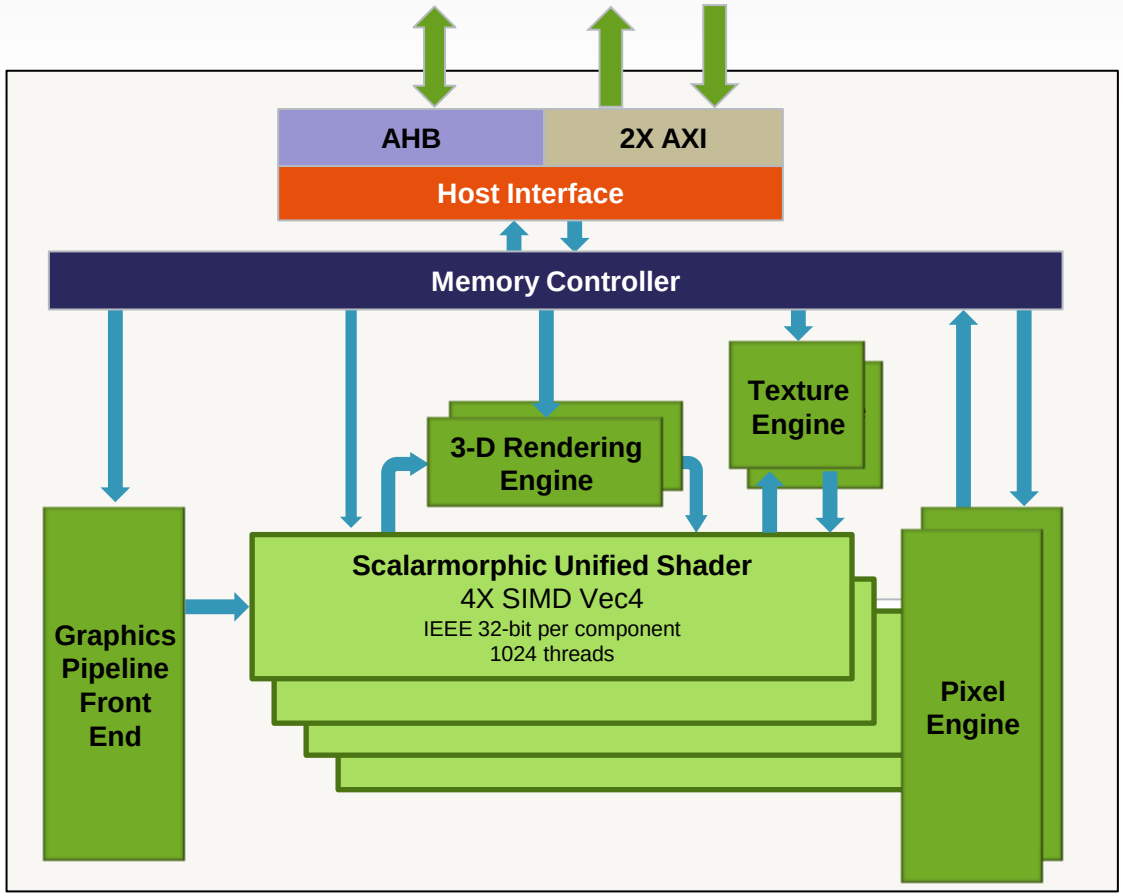
Vivante GC2000 ScalarMorphic Architecture

Architectural Features

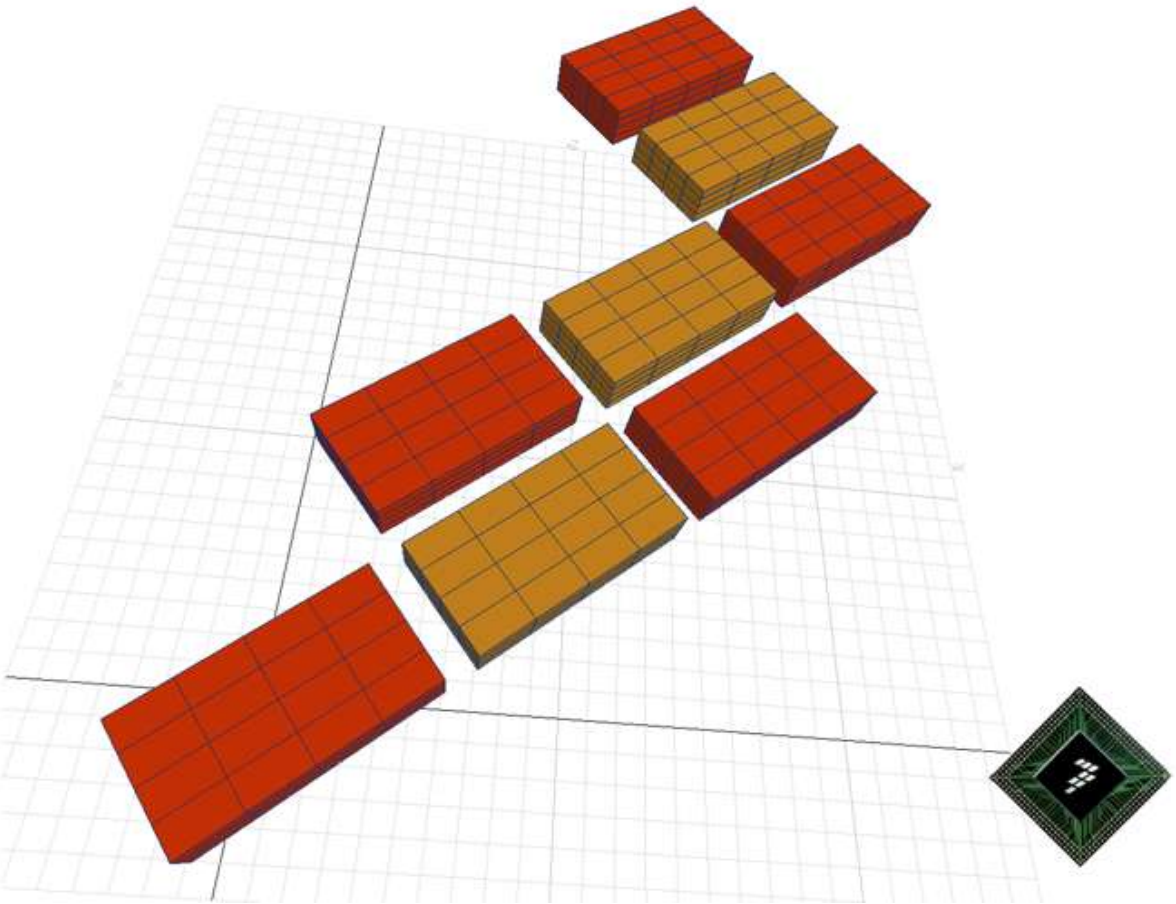
- Native OpenGL ES 2.0 rendering
- Quality 4x MSAA anti-aliasing
- Depth buffering
- Multi-format texture compression

System Functionality

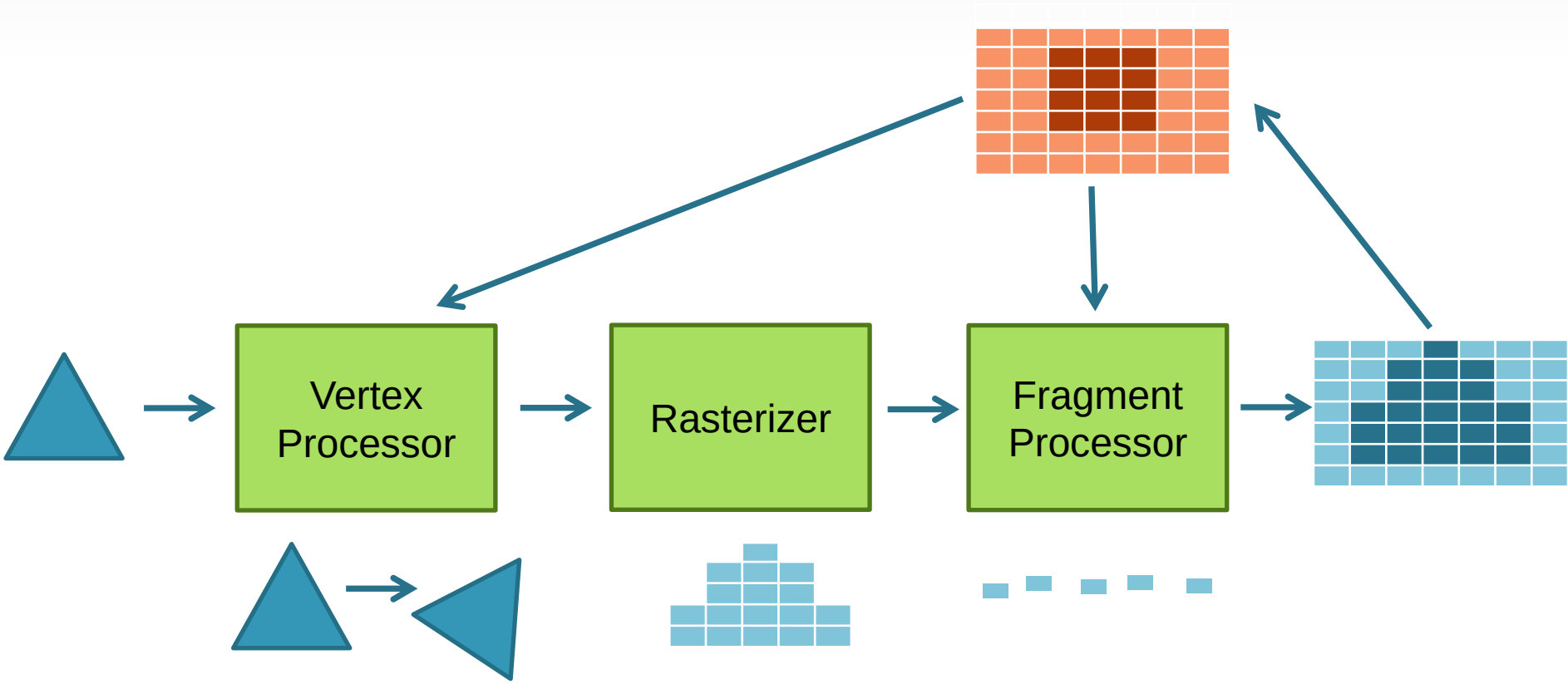
- Up to 1024 independent threads
- Low bandwidth characteristics for both simple and complex scenes
- Comprehensive API support
- OpenGL ES 1.1 / 2.0
- OpenVG 1.1
- OpenGL 2.1 & 3.0 (w/ GLX)
- OpenCL 1.1 EP



Quick Overview of 3D Graphics Fundamentals

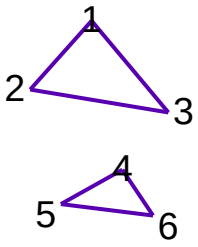


General ‘Basic’ 3D Pipeline

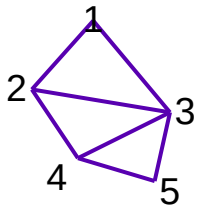


Triangles in OpenGL

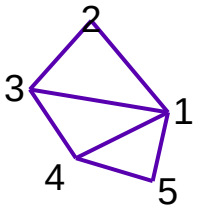
- 3D objects are built from one or more series or “strips” of triangles
- Typically generated by 3D artist tools from higher-level representations



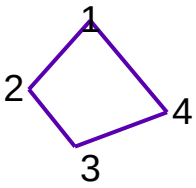
GL_TRIANGLES



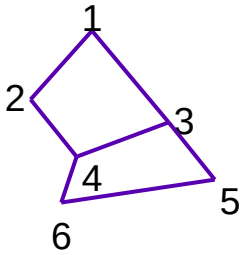
GL_TRIANGLE_STRIP



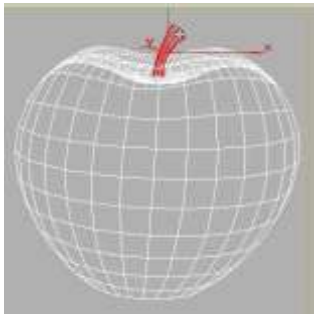
GL_TRIANGLE_FAN



GL_QUADS



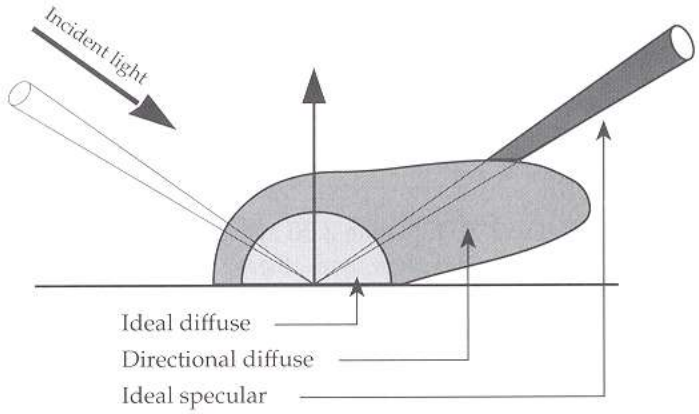
GL_QUAD_STRIP



OpenGL Lighting

- Lighting is calculated at each vertex

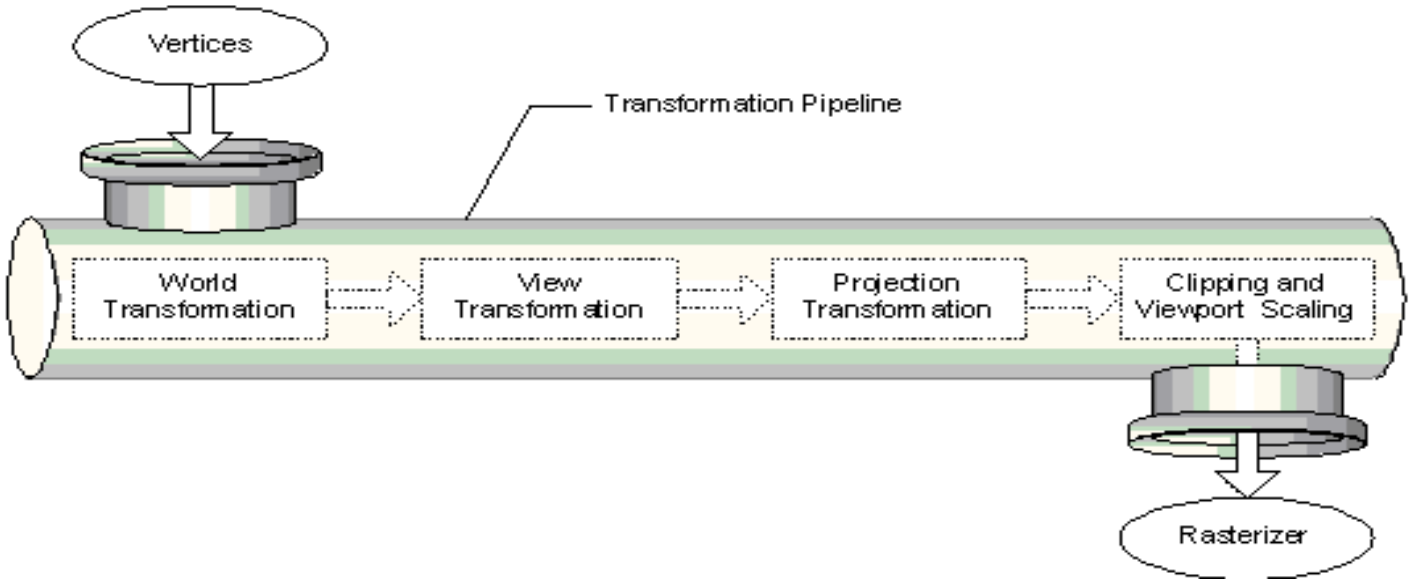
- Phong Lighting 3 components:
  - Ambient
    - Do not depend of the light source.
  - Diffuse
    - The light is evenly reflected in all direction.
  - Specular
    - The reflection is predominant in a given direction.
    - The specular coefficient can be varied.



- The light intensity can decrease according to the distance (constant, linear or quadratic).

Transform

- The 3D Cartesian coordinates that define the triangle vertices are multiplied by a matrix to move it from the object's space to the "screen" space



Simple 3D – Affine Transforms

Scaling

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Rotations

X-Axis

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

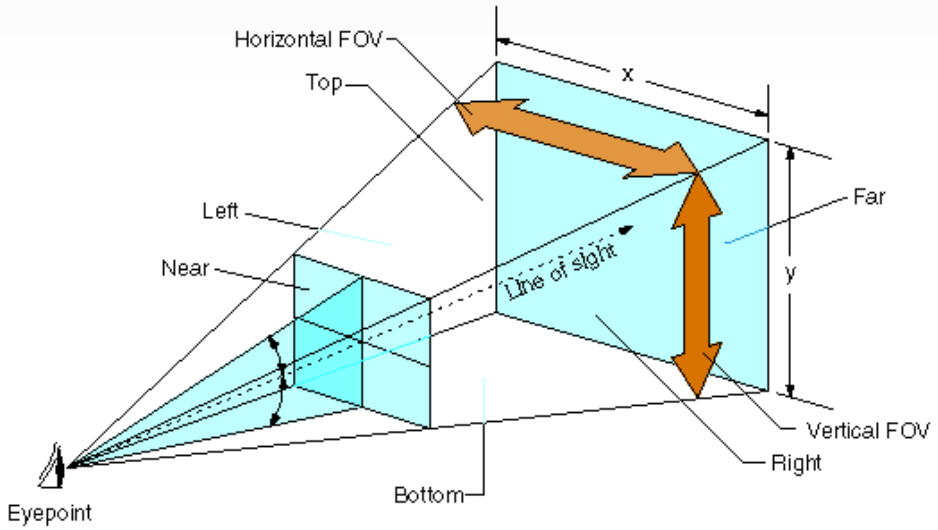
Y-Axis

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

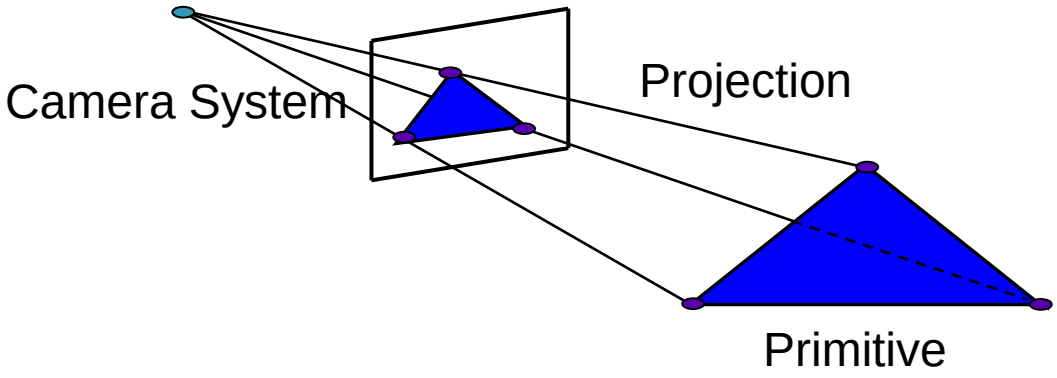
Z-Axis

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Transform



$$\text{Aspect Ratio} = \frac{y}{x} = \frac{\tan(\text{vertical FOV}/2)}{\tan(\text{horizontal FOV}/2)}$$



$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -QZ_n & 0 \end{bmatrix}$$

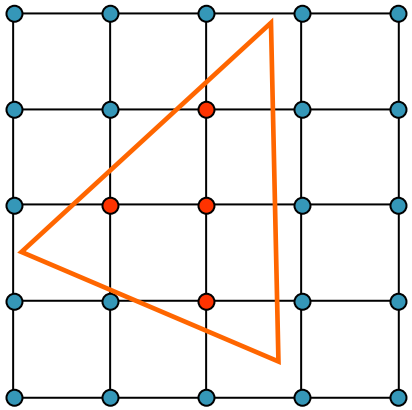
$$w = \cot\left(fov_w/2\right)$$

$$h = \cot\left(fov_h/2\right)$$

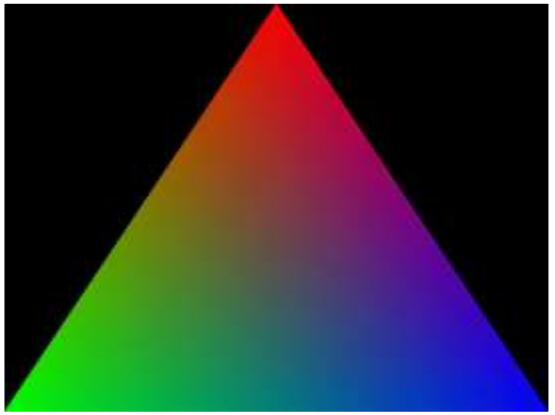
$$Q = \frac{Z_f}{Z_f - Z_n}$$

Rasterization

- After the 3 vertex coordinates are transformed into screen space, they are then bilinearly interpolated and aliased to ‘fragments’
- Other data associated with the vertices is also interpolated (colors, normals, texture coordinates, etc.)



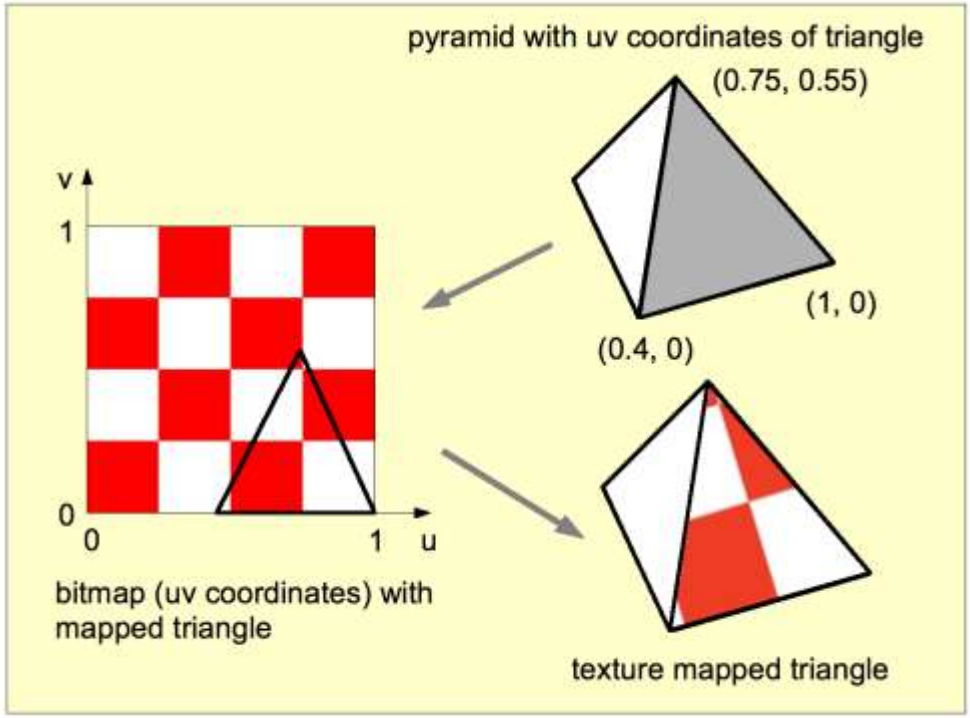
Rasterization



Linear Color Interpolation

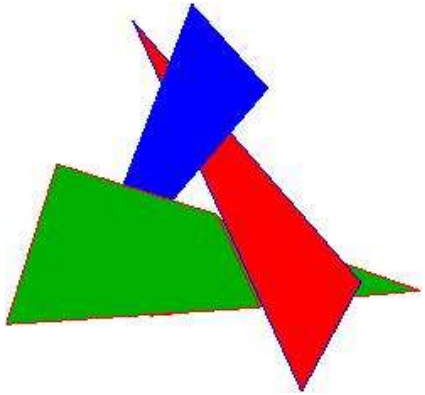
Texture Mapping

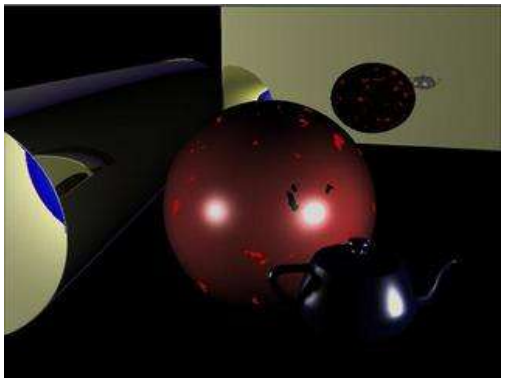
- If a texture (2D still image) is being applied to the surface, the interpolated texture coordinates are used to derive what color should be sampled from the texture.



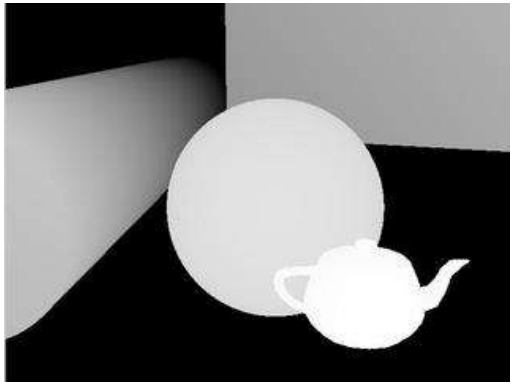
Depth Test / Z Test

- After interpolation, fragments have a ‘depth’ into the screen that is tested against the fragments that have already been rasterized.
  - If a fragment has already been produced that is “closer” to the screen, then the new value is not store.
  - Otherwise, the new color value overwrites the old and the new depth is written to a “Depth Buffer” or “Z Buffer”.



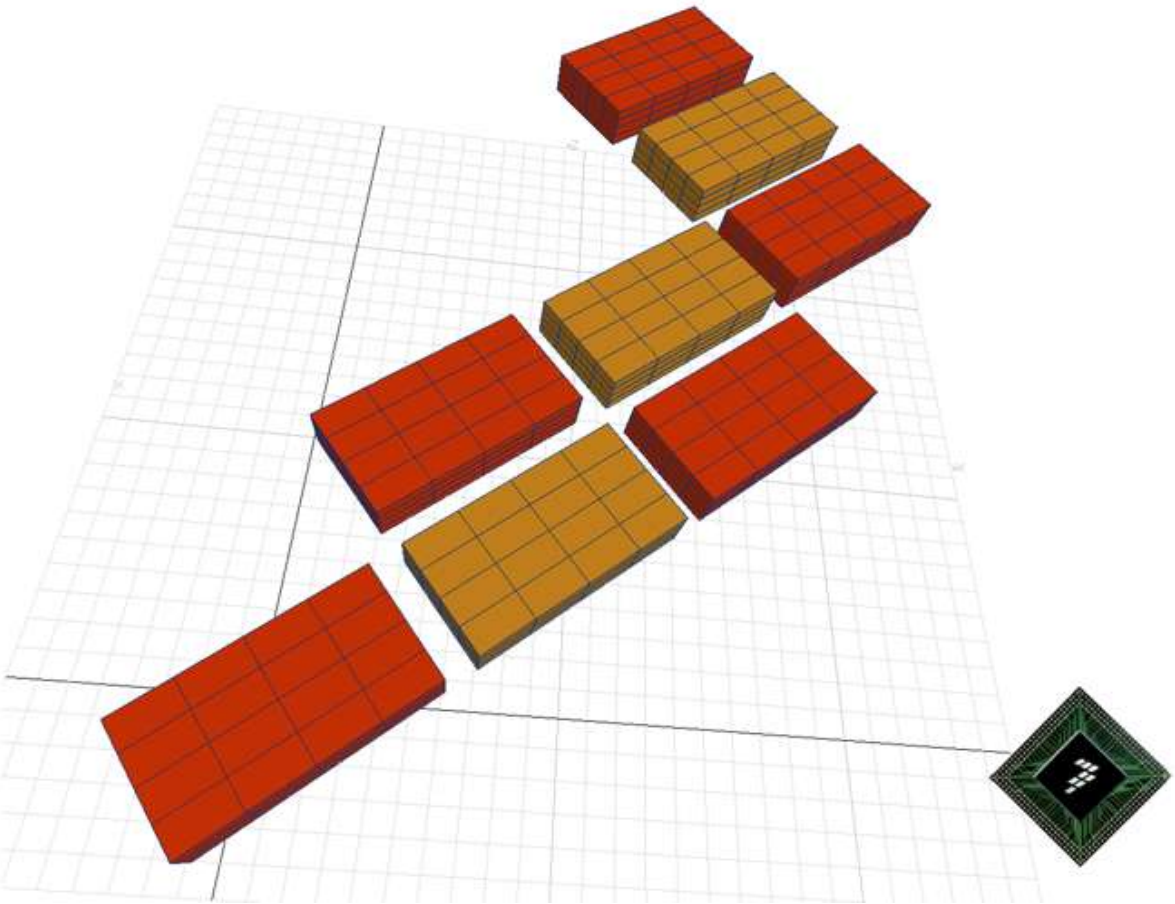


Final image



Final z-buffer

First Steps: EGL and Windowing Systems



EGL

- EGL– portable layer for graphics resource management
  - Created by the Khronos Group to use with GPU APIs
  - Graphics context management
  - Surface/buffer binding
  - Rendering synchronization
- Primary usage is to get the API context and allows user to map API buffers (e.g. OpenGL framebuffers) to Windows

More on EGL

- EGL Controls:
  - Creation of surfaces and API contexts
    - In windowed systems (i.e. X11) this is a ‘window’ surface
    - In other systems, this just aliases to a full screen
    - Allows the user to specify bit depth, resolution, etc.
  - Which API (OpenGL, OpenVG, OpenMAX, etc) is able to draw to the surface
    - Not setting your active API correctly will result in nothing drawn!
  - When the buffer is “flipped”
    - You won’t see anything until you eglSawpBuffers
- Also allows buffers to be shared between APIs
  - Typically for texturing and composition

EGL Sample: Display Initialization and API Bind

```
//The ‘display’ is an OS centric term
//The EGLDisplay is a way of restricting the EGL scope
//EGL_DEFAULT_DISPLAY for framebuffer
//or XOpenDisplay(NULL) for X11
egldisplay = eglGetDisplay ( display );
eglInitialize(egldisplay, NULL, NULL);

//EGL just controls the surface.  The API that draws to
//the surface must be set for the current thread
eglBindAPI(EGL_OPENGL_ES_API);
// Or
eglBindAPI(EGL_OPENVG_API);
```

EGL Sample: Surface Attributes and Config

// To get a surface you can render to, you need to request one from EGL. The 'config' is the set of valid surface properties. For a variety of reasons, GPUs support a limited set of 'configs'. The following structure defines a set of surface attributes for requesting a config.

```
static const EGLint s_configAttribs[] =
{
        EGL_RED_SIZE,      5,  //Colors are a minimum value
        EGL_GREEN_SIZE,  6,
        EGL_BLUE_SIZE,     5,
        EGL_ALPHA_SIZE,  0,
        EGL_SAMPLES,       0, //Samples define the number of points used in anti-aliasing (0 means no AA, 4 means 4xAA)
        EGL_NONE //Defines the end of the config
};
```

// The attributes are used to request a surface configuration

```
eglChooseConfig(egldisplay, s_configAttribs, &eglconfig, 1, &numconfigs);
```

// There is no guarantee the driver will return a config that will meet your attribute requirements, so you must check one is return

```
assert(numconfigs == 1);
```

EGL Sample: Window

```
//Create and x window
window = XCreateSimpleWindow(display, rootwindow, 0, 0, 400,
   533, 0, 0, WhitePixel (display, screen));

XMapWindow(display, window);

//get an eglsurface from the display using our egl configuration
eglsurface = eglCreateWindowSurface(egldisplay, eglconfig,
   window, NULL);
```

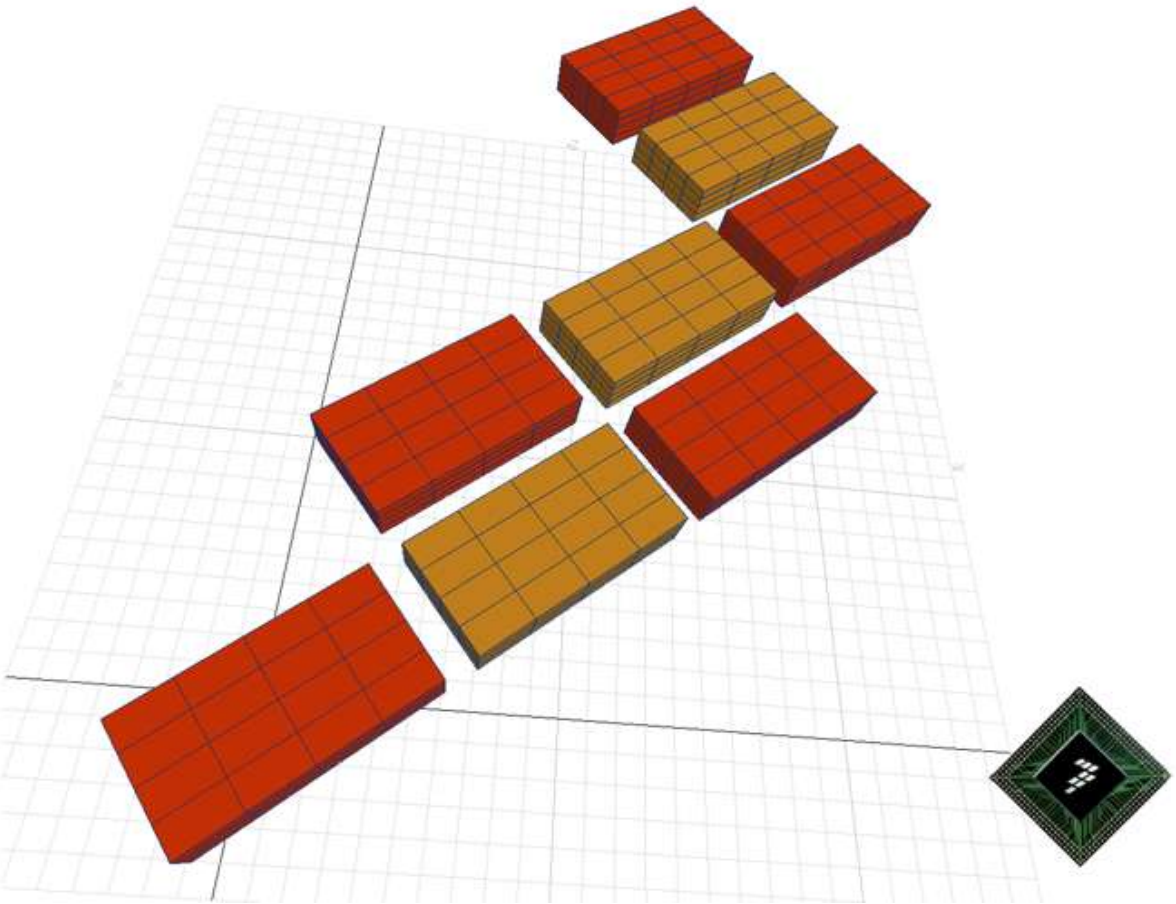
i.MX 6 Linux Framebuffer EGL Porting

```
// i.MX6 Vivante EGL initialization
EGLNativeDisplayType native_display = fbGetDisplay();
EGLNativeWindowType  native_window  =
   fbCreateWindow(native_display, 0, 1, WIDTH, HEIGHT);

// i.MX5 AMD EGL initialization
EGLNativeWindowType  native_window = open("/dev/fb0",
   O_RDWR);


eglWindowSurface = eglCreateWindowSurface(eglDisplay,
   eglConfig, native_window, NULL);
```

EGL Sample: Context Creation

```
//create the egl graphics context
eglcontext = eglCreateContext(egldisplay, eglconfig, NULL,
   NULL);

//make the context current
eglMakeCurrent(egldisplay, eglsurface, eglsurface, eglcontext);
```

OpenGL ES 1.1 Overview and Samples



OpenGL & OpenVG Programming model

- Low level hardware interface
  - May see high-level when looking at registers / abstractions, but the APIs are really the lowest useful level an application could feasibly use

- OpenGL and OpenVG are APIs designed around a State Machine based client-server model
  - User ‘sets’ and ‘gets’ variables in the machine (enable/disable, bind, etc.)
  - *OpenVG*: Cache handles to paints, paths, gradients, etc. to avoid the re-preprocessing each frame
  - *OpenGL:* Cache handles to shaders to avoid recompilation and track texture allocation to control memory usage

- Data types in a function are determined by a i, f, or v postfix
  - Integer, float, or vector

- Execution is only guaranteed when user blocks (i.e. [vg/gl]Finish) or implicitly flips (i.e. eglSwapBuffers)

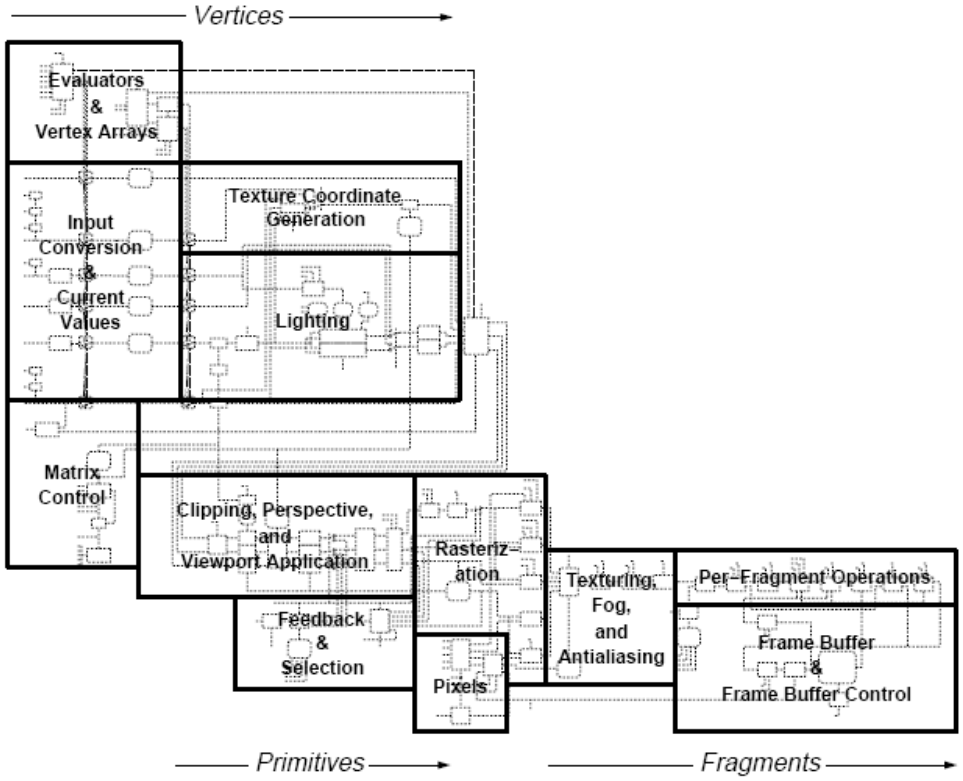
OpenGL ES Overview

- Defined as a subset of the full OpenGL API
  - OpenGL ES 1.0 is derived from OpenGL X.x
  - OpenGL ES 1.1 is derived from OpenGL X.x

- 'Embedded' and mobile focused subset
  - Removes redundant ways to do the same functionality
    - E.G. drawing can only be done with arrays, no glBegin / end
  - Keeps all the same state machine / API model, data types, etc. of OpenGL to maintain application portability

- OpenGL ES 1.x has two 'profiles'
  - Common: The full OpenGL ES API
  - Common-Lite: A fixed-point focused subset of OpenGL ES
  - Almost no devices on the market use Common-Lite

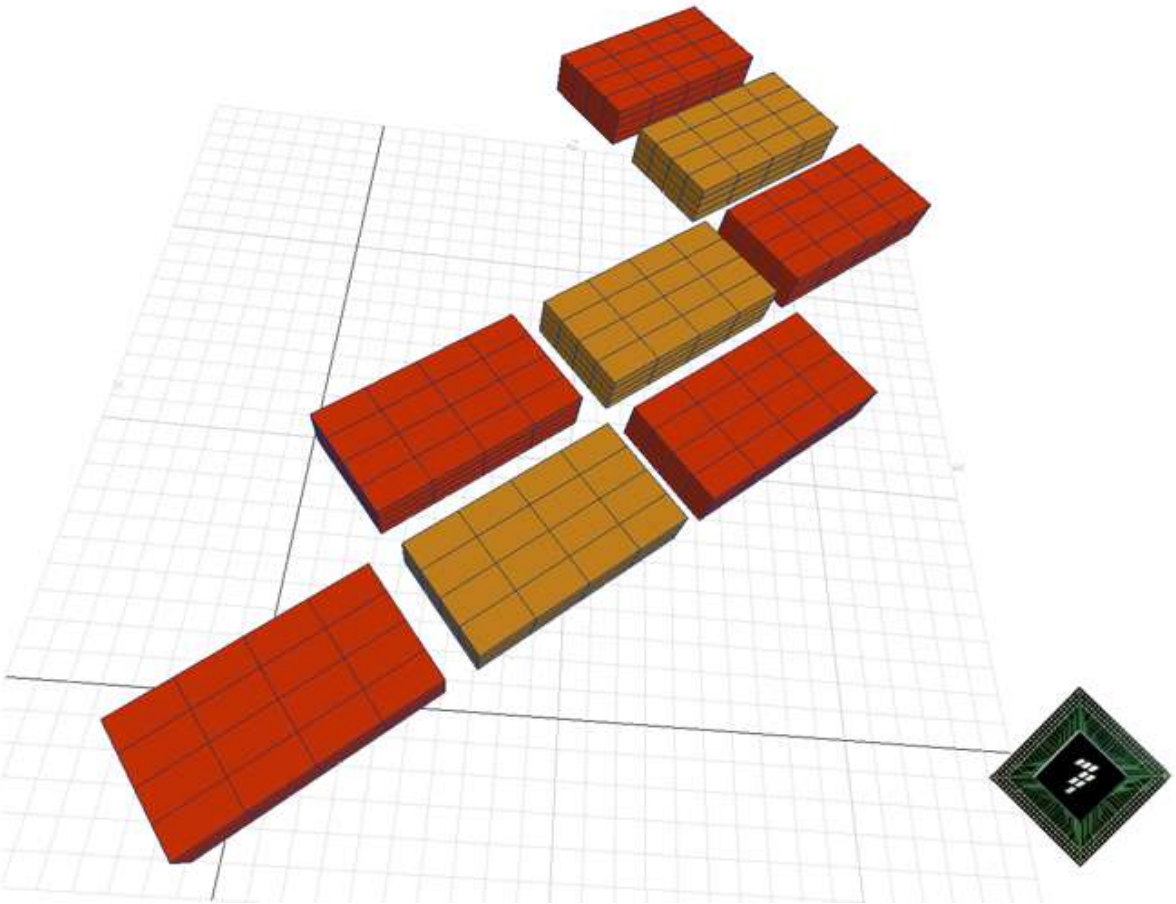
The OpenGL State Machine



Key to OpenGL Operations

OpenGL ES 1.1 Code Walkthrough

OpenGL ES 2.0 Overview and Samples

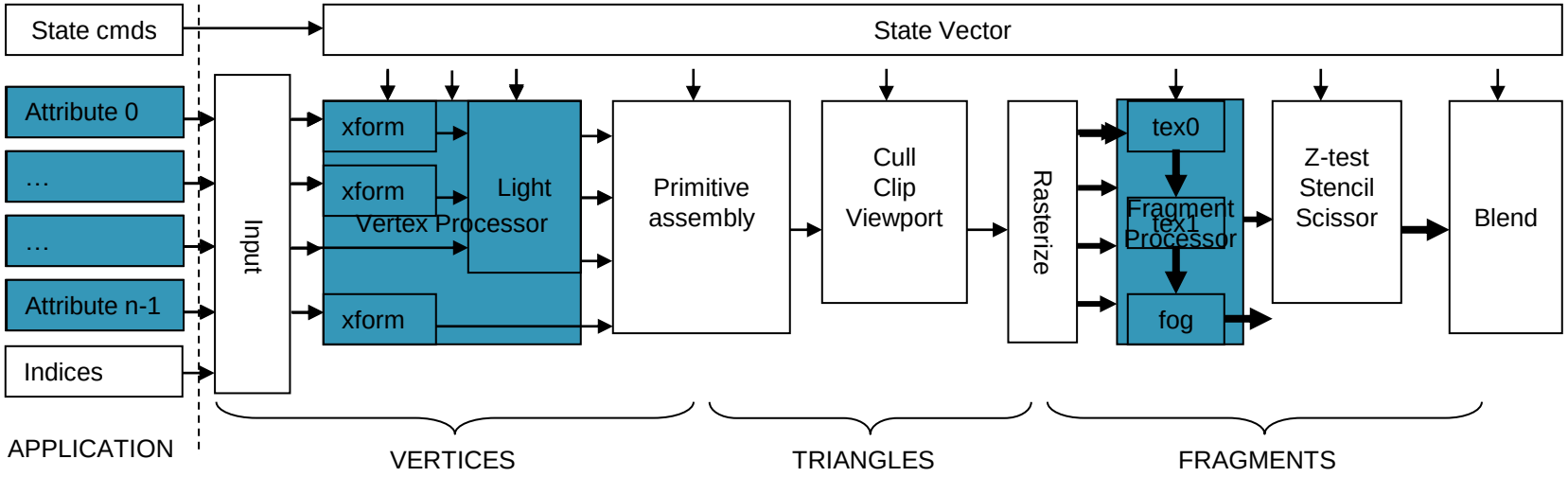


Changes in OpenGL ES 2.0

- Eliminates large pieces of the OpenGL ES 1.1 pipeline
  - Derived from OpenGL 2.x full API
  - Removes much of the ‘fixed’ pipeline (more on this in a moment)
  - OpenGL ES 2.0 is NOT backward compatible with OpenGL ES 1.1
  - For i.MX5x and i.MX 6x, we ship both 1.1 and 2.0 libraries for compatibility
- Users can insert ‘C’ like code into the pipeline
  - Replaces large sections of the 1.1 state machine
  - Called ‘shaders’
  - See next slide
- Standardized Texture Compression
  - Ericsson Texture Compression (ETC)
  - Cube Maps
  - Minimum of eight texture units
- Frame Buffer Objects
  - Easier to deal with rendering to a texture
  - Exchange data within the pipeline without copies

The OpenGL ES Pipeline

- What Changes From ES 1.1 to ES 2.0?
  - General-purpose attributes replace fixed input arrays
  - Vertex shader programs replace transform and lighting
  - General-purpose uniforms replace fixed lighting & texture state
  - General-purpose varyings replace fixed fragment attributes
  - Fragment shader programs replace texture / fog / alpha test



Vertex Shader Functions

- The vertex shader can do:
  - Transformation of position using model-view and projection matrices
  - Transformation of normals, including renormalization
  - Texture coordinate generation and transformation
  - Per-vertex lighting
  - Calculation of values for lighting per pixel

- The vertex shader cannot do:
  - Anything that requires information from more than one vertex
  - Anything that depends on connectivity.
  - Any triangle operations (e.g. clipping, culling)
  - Access color buffer

Example Vertex Shader

Diffuse lighting

```
uniform mat4 ModelViewProjectionMatrix, NormalMatrix;
uniform vec4 LightSourceDiffuse, LightSourcePosition, MaterialDiffuse;

attribute vec4 InputPosition, InputNormal, InputTextureCoordinates;

varying vec4 VertexColour;
varying vec4 TextureCoordinates;

void main()
{
   vec3 normal, lightDirection;
   vec4 diffuse;
   float NdotL;

   normal = normalize(NormalMatrix * Normal);
   lightDirection = normalize(vec3(LightSourcePosition));
   NdotL = max(dot(normal, lightDirection), 0.0);
   diffuse = MaterialDiffuse * LightSourceDiffuse;
   VertexColor = NdotL * diffuse;

   TextureCoordinates = InputTextureCoordinates;

   gl_Position = ModelViewProjectionMatrix * position;
}
```

Fragment Shader Functions

- The fragment shader can do:
  - Texture blending
  - Fog
  - Alpha testing
  - Dependent textures
  - Pixel discard
  - Bump and environment mapping

- The fragment shader cannot do:
  - Blending with colour buffer
  - ROP operations
  - Depth or stencil tests
  - Write depth

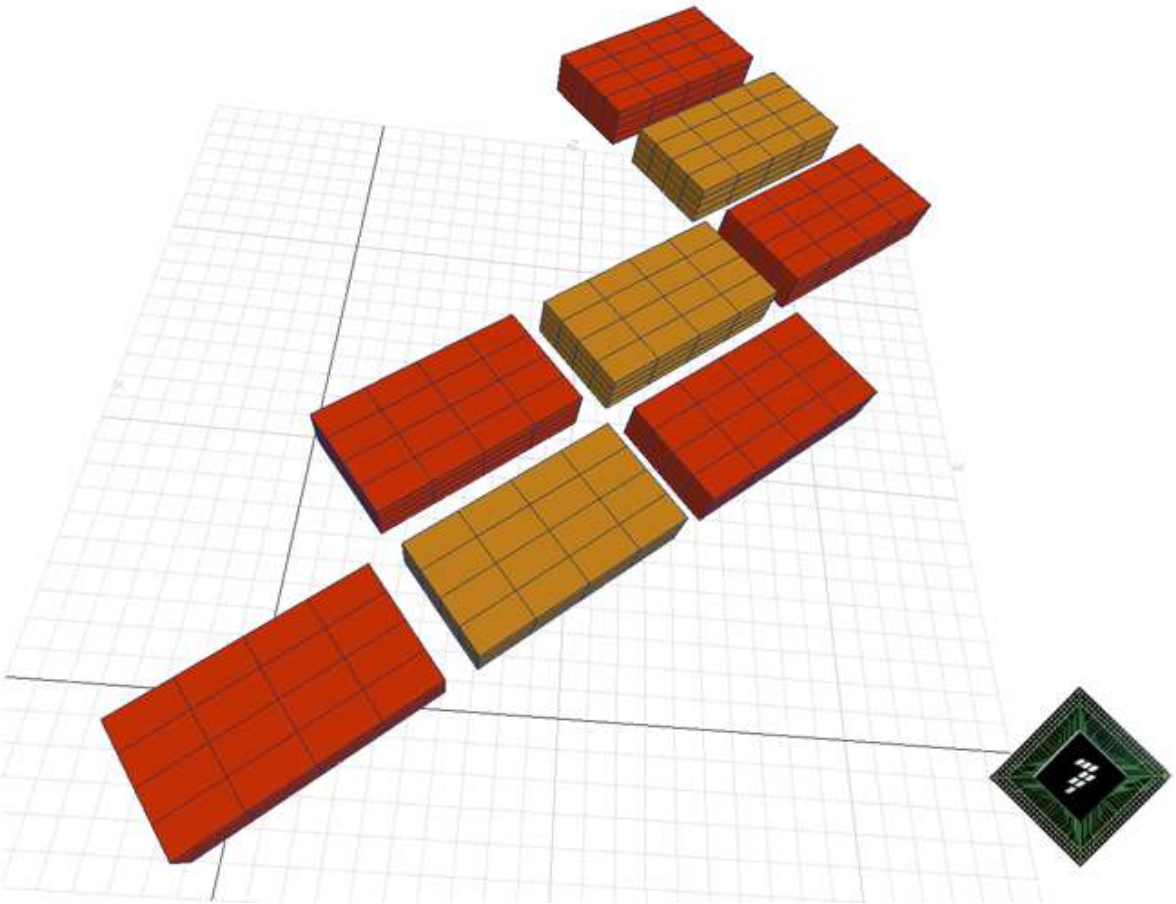
Example Fragment Shader

Simple Texture Blend

```
uniform sampler2D TextureHandle;
varying vec2 TextureCoordinates;
varying vec4 VertexColour;
void main()
{
  vec4 texel = texture2D (TextureHandle,
                          TextureCoordinates);
  gl_FragColor = texel * VertexColour;
}
```

OpenGL ES 2.0 Code Walkthrough

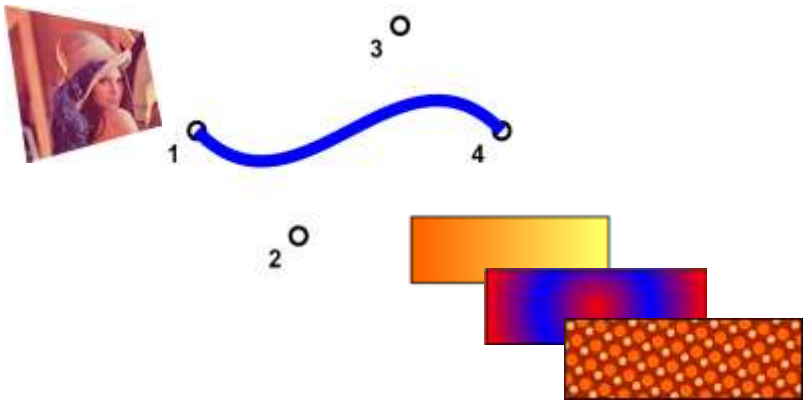
OpenVG 1.1 Overview and Samples



OpenVG Features

- Design Philosophy
  - Expands the OpenGL Programming model to 2D Vector Graphics
  - Provides a low-level hardware acceleration abstraction layer
  - Uses OpenGL-style syntax where possible
  - Allows flexibility in the way acceleration can be provided
  - Enables hardware vendors to use their own preferred internal representations

- The VGU Utility Library
  - Higher-level Geometric Primitives
  - Image Warping

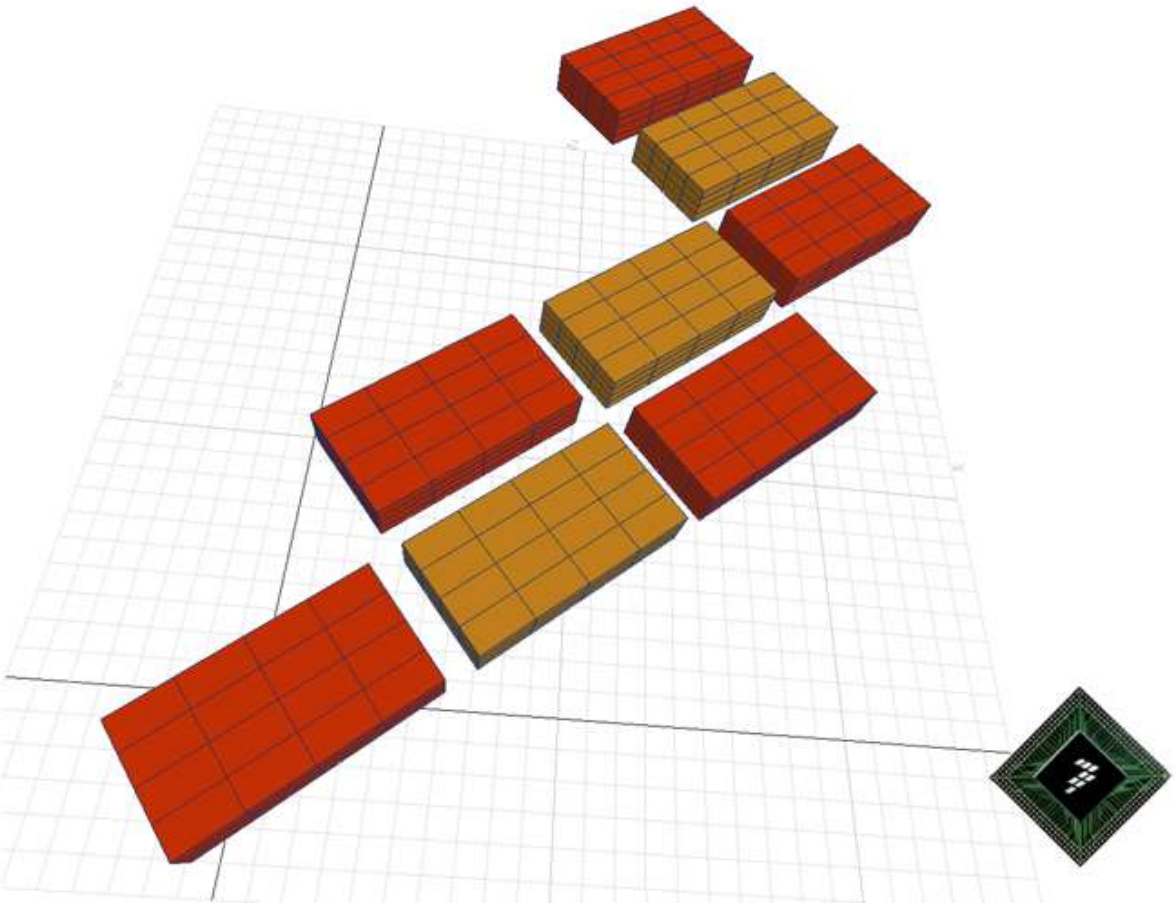


The OpenVG Pipeline

- OpenVG defines a hardware pipeline for paths and images
  - Path Definition
  - Stroking
    - Line width, joins & caps, dashing, etc.
  - Transformation
    - 2x3 (paths) and 3x3 (images) matrices
  - Rasterization
  - Clipping & Masking
    - Scissor rectangles
  - Paint Generation
  - Blending
  - Dithering
  - Image Filters

OpenVG Code Walkthrough

Overview of GPGPU



GPGPU

- GPGPU – Leveraging the SIMD and highly parallel GPU hardware to process non-graphics generation use cases.

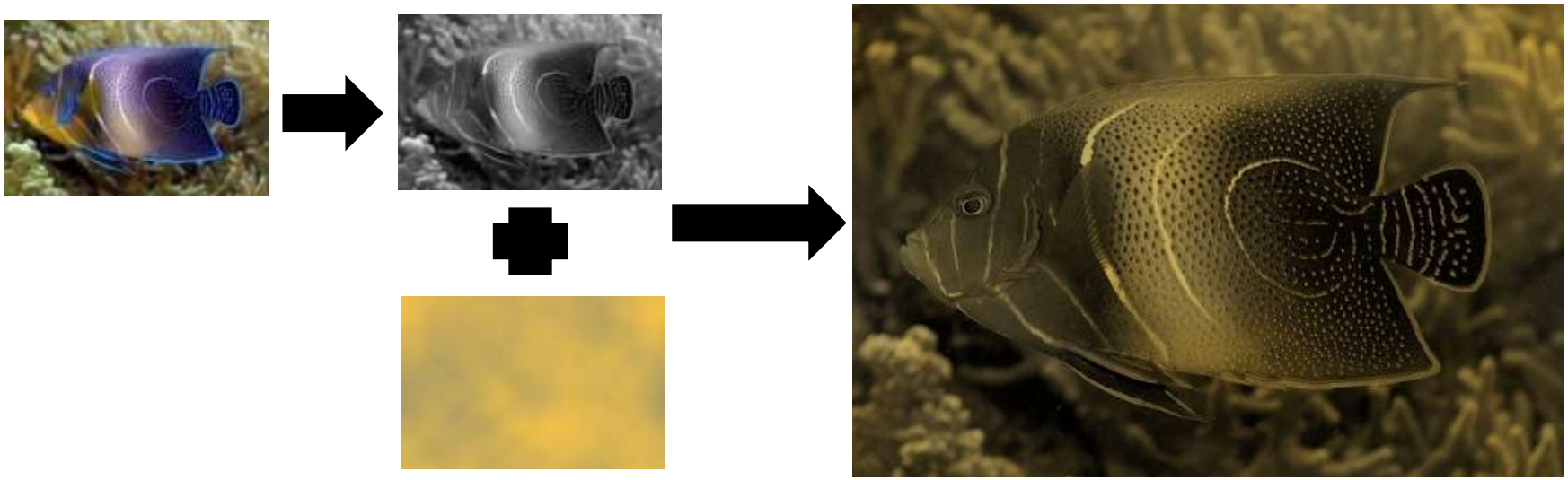
- Image filtering is common example of easy-to-enable GPGPU programming. The filters below run on OpenGL ES 2.

emboss warp sepia sharpen sobel (line detection)

Example – Sepia on OpenGL ES 2

RGB → Grayscale, Modulated with N-color Noise Cloud

```
void main (void){
    Color = texture2D(sampler2d_Image, myTexCoord);
    Intensity = 0.3 * Color.r + 0.59 * Color.g + 0.11 * Color.b;
    gl_FragColor = Intensity * texture2D(sampler2d_SepiaNoise, myTexCoord)
}
```

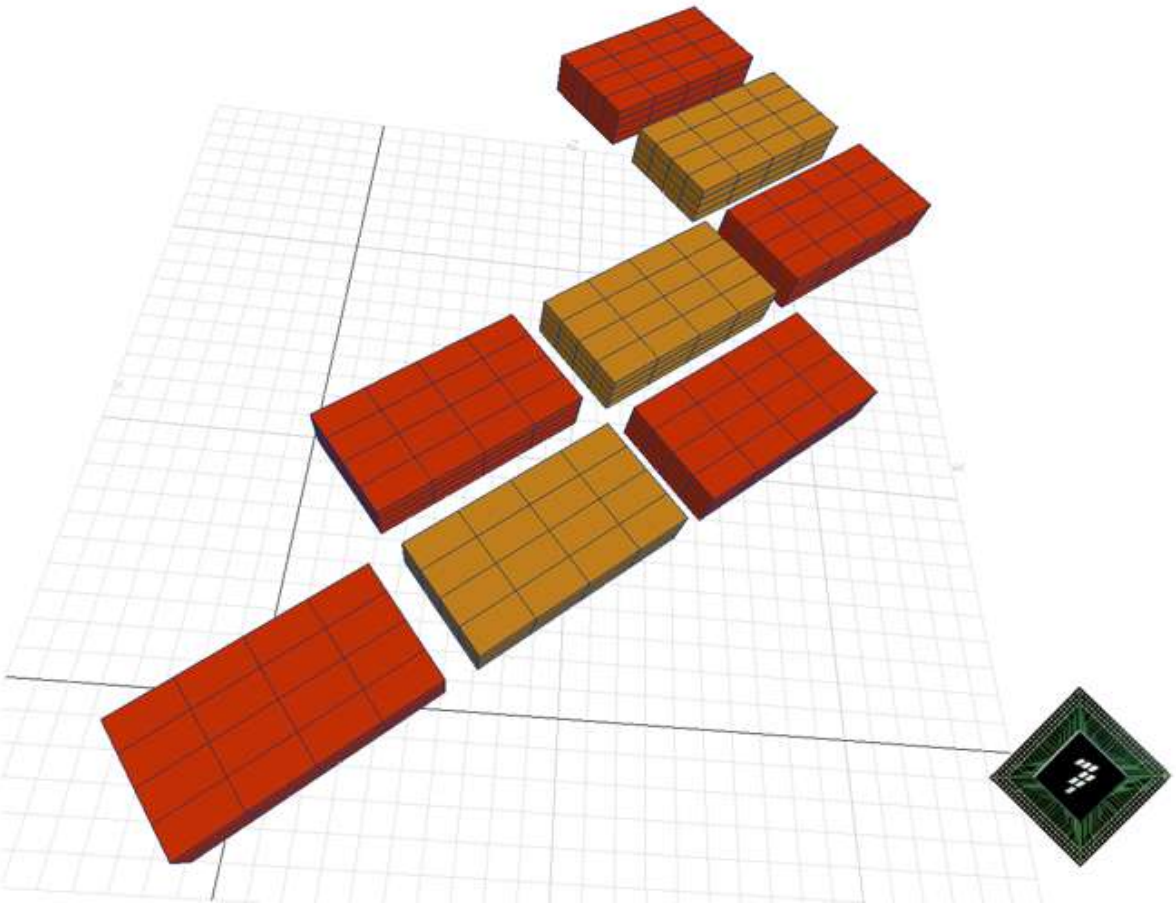


OpenGL Limitations

- Despite the high programmability, OpenGL ES 2 is still fairly limited for ‘general purpose’ programming
  - Very fixed pipeline inputs and outputs with interpolation between
  - Very restrictive shader programming model
  - Shaders only run over a very set range of input values

- Thus, GPGPU programming needs another API …

OpenCL Overview and Samples



OpenCL

- Move to higher compute facilities in i.MX 6 requires support for a more advanced GPGPU API than OpenGL ES
- OpenCL is an open specification API from the Khronos Group (same standards body that made OpenGL, OpenVG, OpenMAX, etc.)
- OpenCL enables asynchronous multi-core programming for heterogeneous computing environments
  - Allows the user to run work ‘kernels’ that run over large data sets and across many processors
  - Effectively makes GPU look like a cluster of DSPs executing on a task
  - i.MX 6Q implements the OpenCL 1.1 Embedded Profile
- Not in Android at this time. Google is pushing Render Script Compute API, which may be supported by GPUs in the future

OpenCL Example Use Cases

- Image Processing
  - Image detection
    - Active Safety (road sign detection, lane departure, driver drowsiness, etc.)
    - Facial Recognition / Eye Tracking
  - Custom Photo Filters
    - HDR image processing
    - Image Effects
- Video Processing
  - Pre and Post Encode Decode Filters
  - Custom scaling and image color conversion
  - Watermarking
- Audio Processing
  - Batch audio processing for leveling and effects
  - Speech recognition and synthesis

OpenCL efficient functions

- Haar-like Features / Wavelets
- Sobel Line Detection
- Computing Eigenvalues
- FFT
- Noise Generation
- DCT
- Histogram Generation
- Histogram of Oriented Gradients (HOGs)
- Demosiac

Anatomy of OpenCL

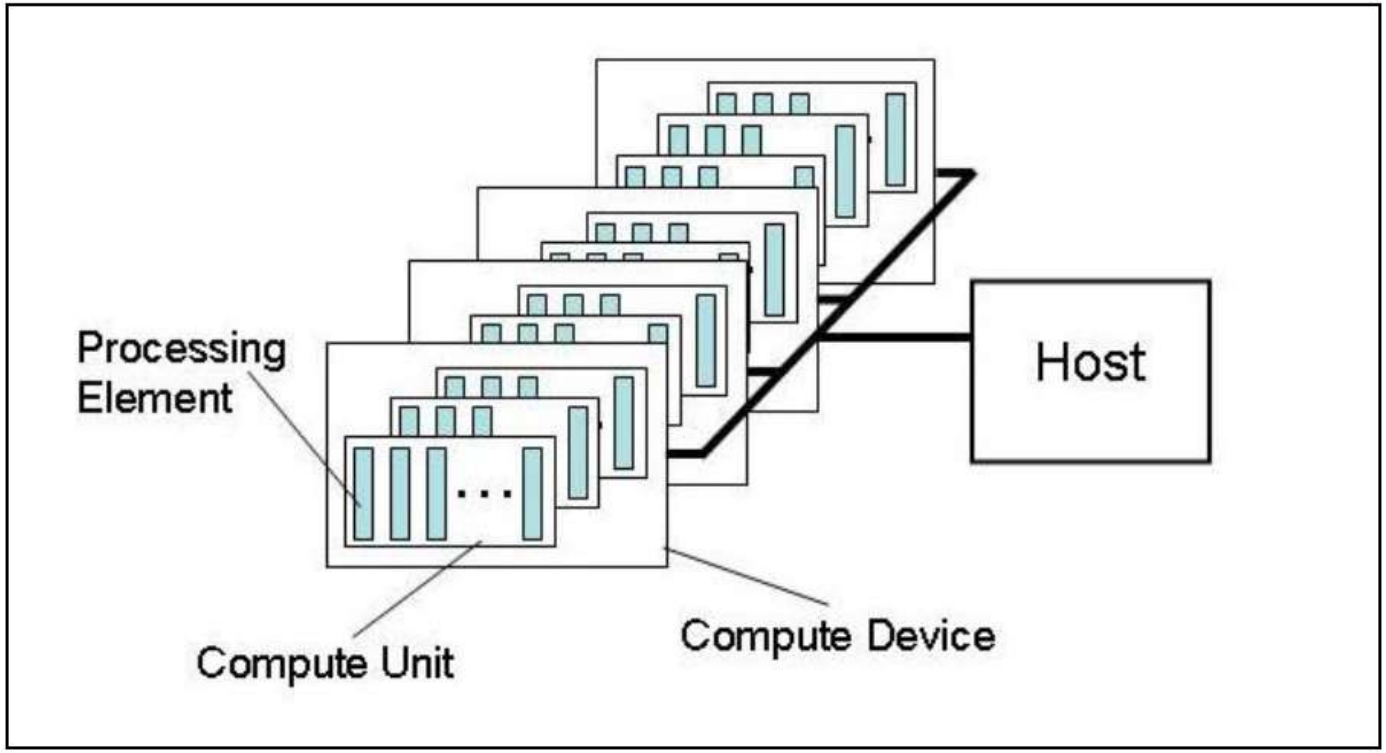
- Language Specification
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions - familiar to developers
  - Well-defined numerical accuracy (IEEE 754 rounding with specified max error)
  - Online or offline compilation and build of compute kernel executables
  - Includes a rich set of built-in functions

- Platform Layer API
  - A hardware abstraction layer over diverse computational resources
  - Query, select and initialize *compute devices*
  - Create *compute contexts* and *work-queues*

- Runtime API
  - Execute *compute kernels*
  - Manage scheduling, compute, and memory resources

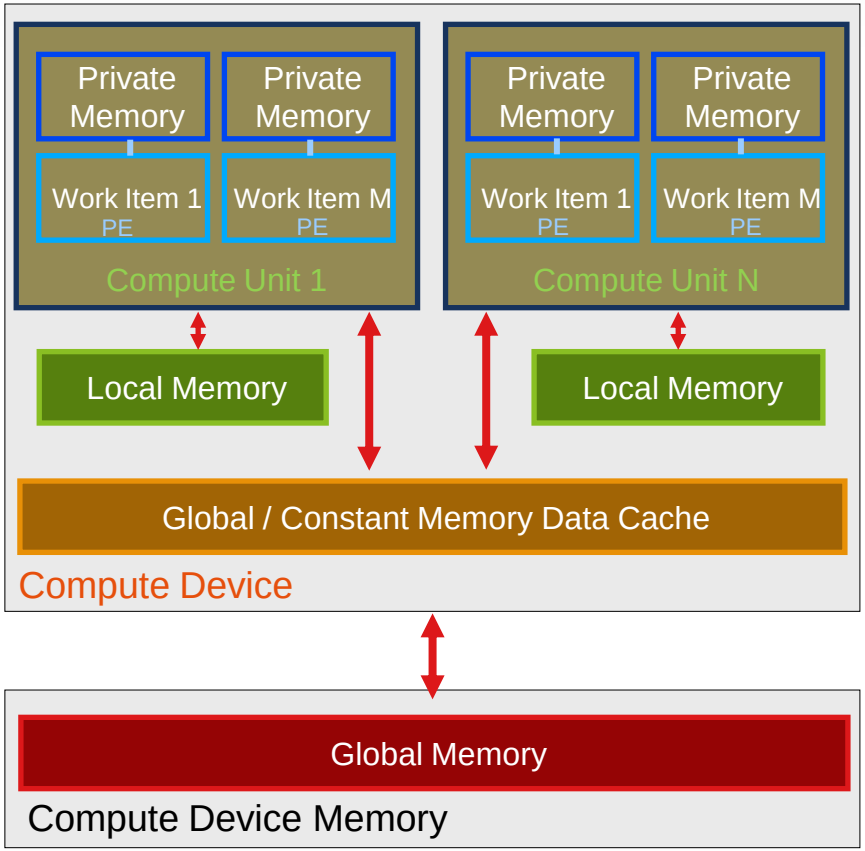
OpenCL Platform Model

- One Host + one or more Compute Devices
  - Each **Compute Device** is composed of one or more **Compute Units**
    - Each **Compute Unit** is further divided into one or more **Processing Elements**



OpenCL Memory Model

- Shared memory model
  - Relaxed consistency
- Multiple distinct address spaces
  - Address spaces can be collapsed depending on the device's memory subsystem
- Address spaces
  - Private - private to a *work-item*
  - Local - local to a *work-group*
  - Global - accessible by all work-items in all work-groups
  - Constant - read only global space
- Implementations map this hierarchy
  - To available physical memories



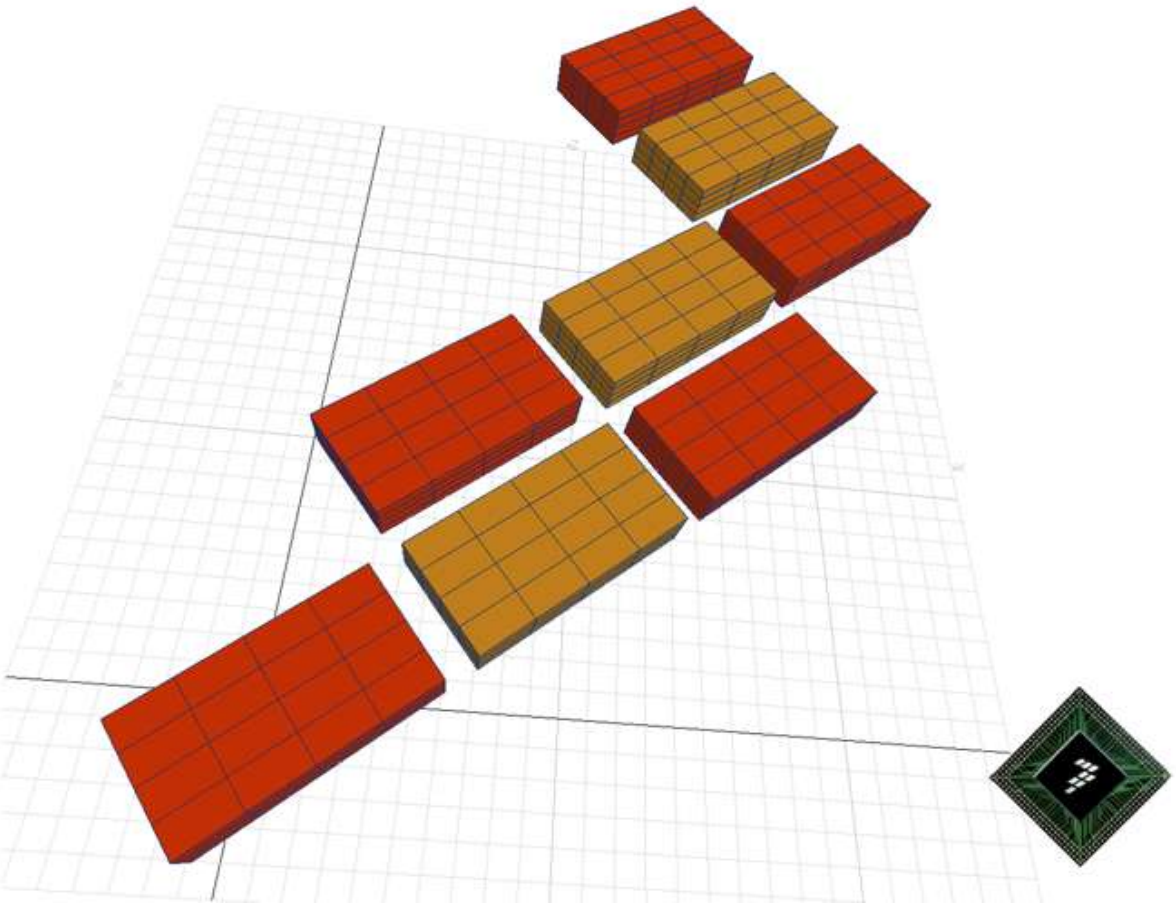
OpenCL Execution Model

- OpenCL *Program*:
  - Kernels
    - Basic unit of executable code — similar to C functions, CUDA kernels, etc.
    - Data-parallel or task-parallel
  - Host Program
    - Collection of compute kernels and internal functions
    - Analogous to a dynamic library

- *Kernel* Execution
  - The host program invokes a kernel over an index space called an ***NDRange***
    - NDRange, “N-Dimensional Range”, can be a 1D, 2D, or 3D space
  - A single kernel instance at a point in the index space is called a ***work-item***
    - Work-items have unique global IDs from the index space
  - Work-items are further grouped into ***work-groups***
    - Work-groups have a unique work-group ID
    - Work-items have a unique local ID within a work-group

OpenCL Sample Code Walkthrough

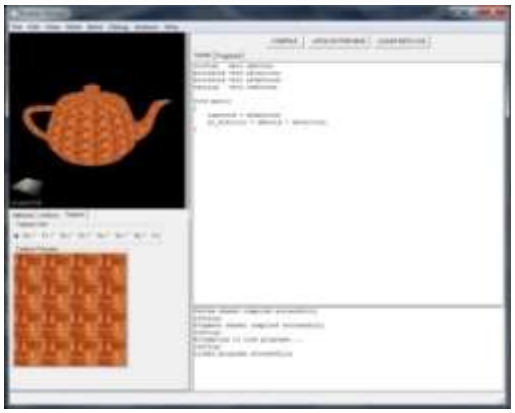
PC Tools



Vivante SDK Offerings

Graphics Application
(OES 1.1/2.0 OVG 1.1)

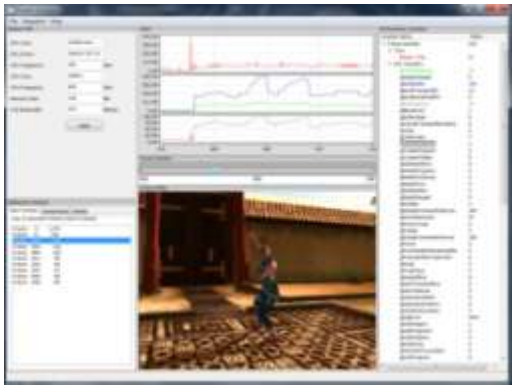
VShader
Rapidly prototype and optimize shader programs



VEmulator
Develop OpenGL ES 1.1 / 2.0 and embedded applications using PC graphics cards

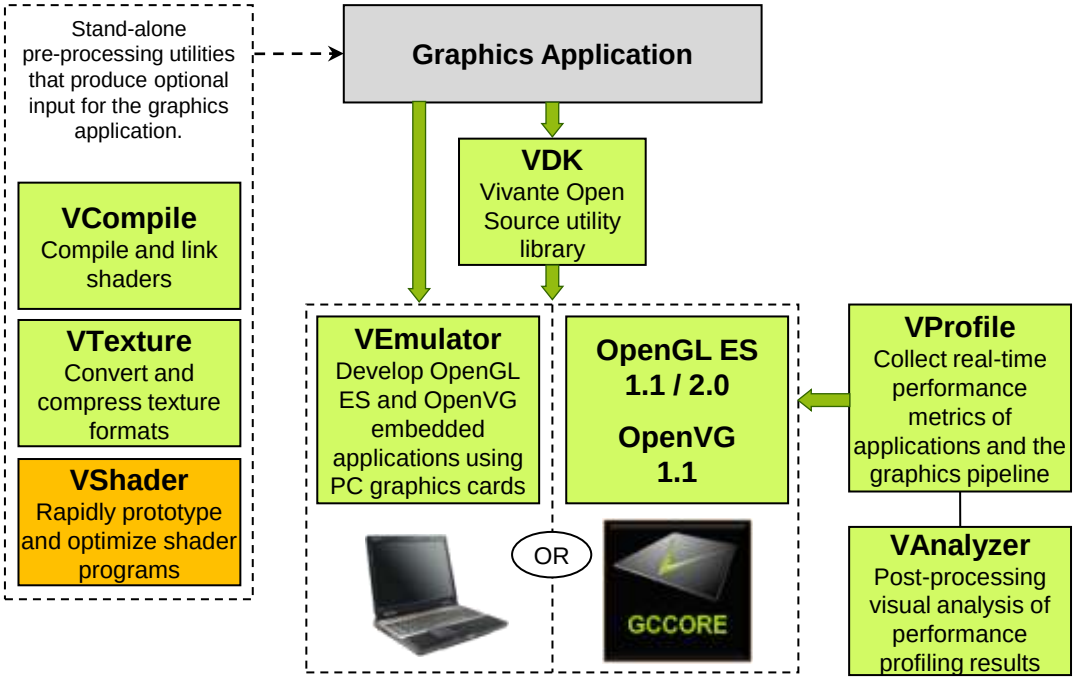
OES1.1/OES2.0/
OVG
Drivers

VProfiler/
VAnalizer
Visually analyze performance of applications and graphics pipeline



VShader

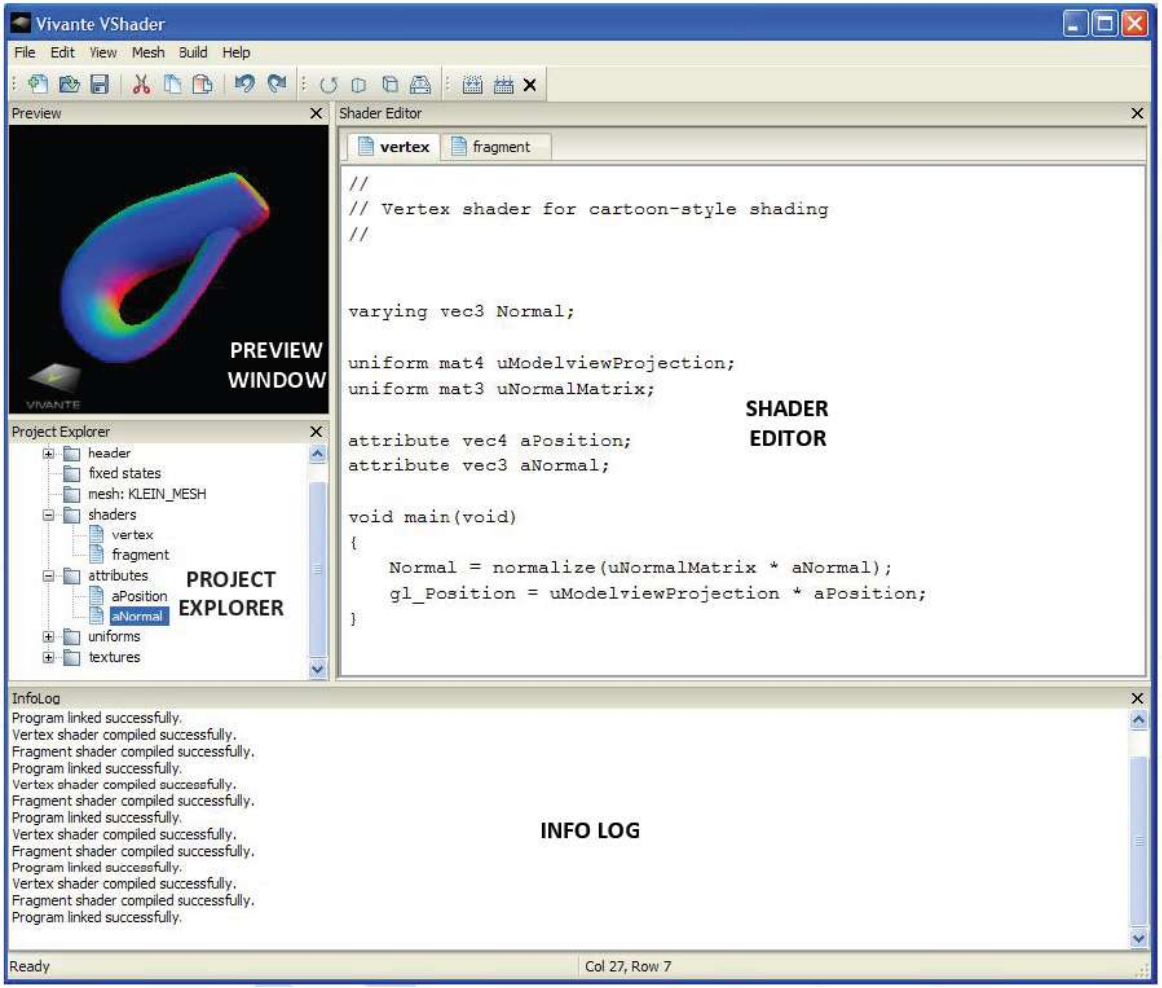
- VShader is a complete off-line environment for editing, previewing, analyzing, optimizing, and debugging shader programs.
- VShader runs on Windows computer platforms



VShader Features

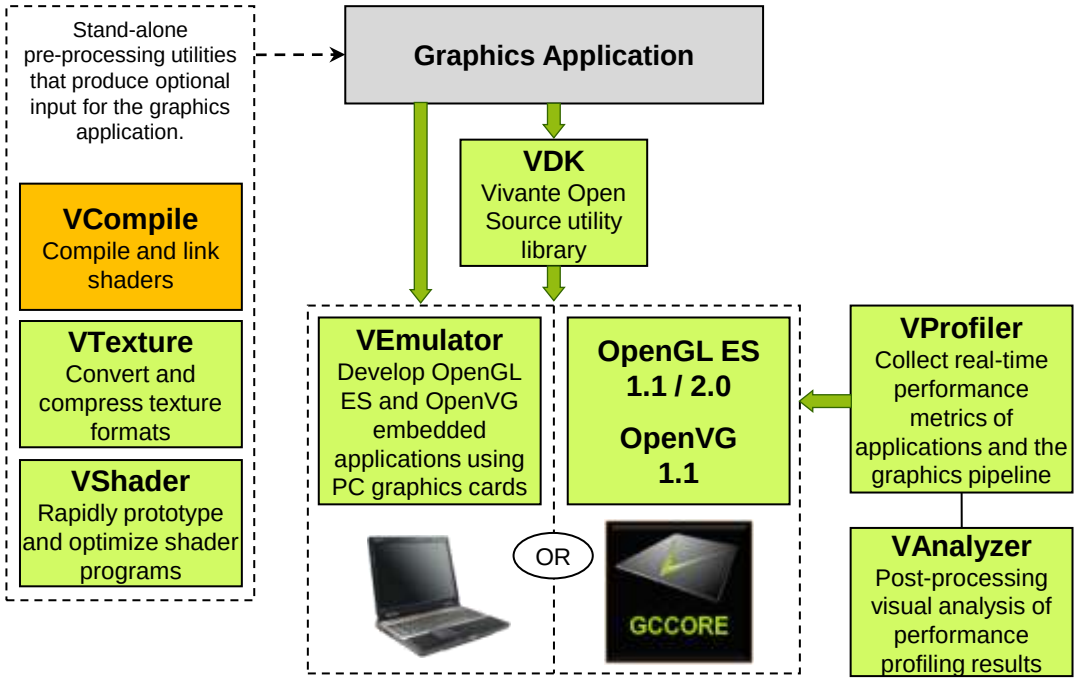
- Build trees of assets and prototype GLSL ES shaders on objects
  - Some built-in models, assets
- Map any texture to shaders
- Import user defined meshes
- Bind mesh attributes to shaders
- Set uniforms in shaders
- Analyze shader compiler output for optimization hints
- Predict hardware performance
- Debug shaders using techniques similar to debugging C code

VShader



VCompiler

- VCompiler is an off-line compiler and linker for targeting Vivante accelerated hardware platforms.



VCompiler

- Simple, command-line driven interface with the following options:

  -v verbose

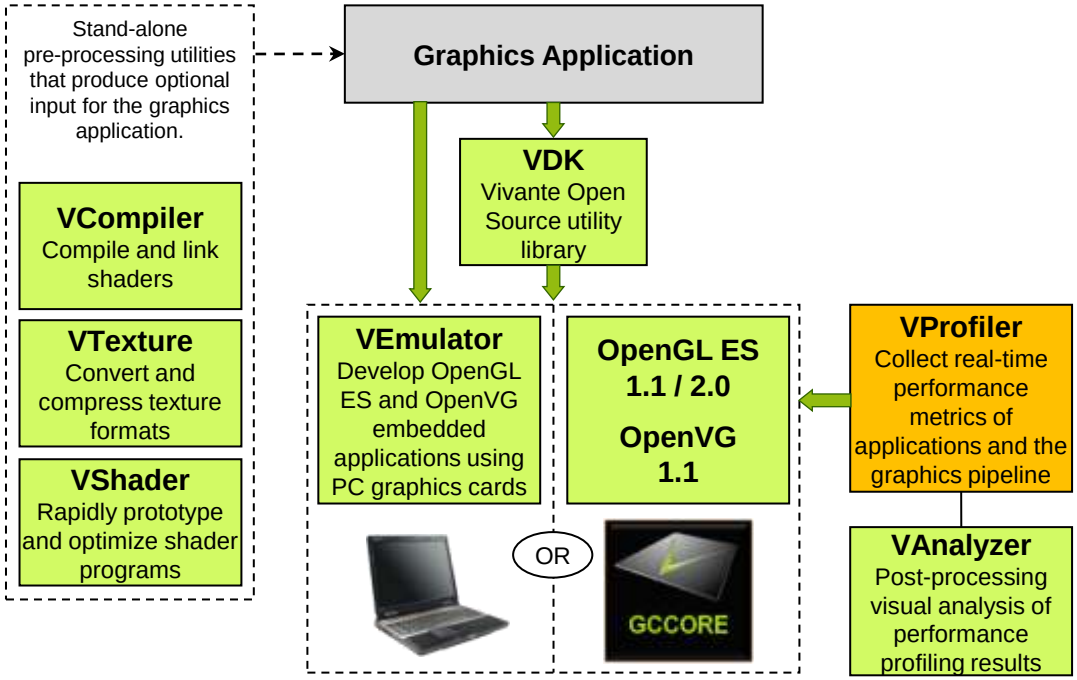
  -o0 do not optimize (optimization is on by default)

  -l create listing (generates ASCII intermittent format with .lst extension in the directory specified)

- VCompiler output file:
  - gcSH when a vertex shader (.vert) is compiled
  - gcSH when a fragment shader (.frag) is compiled
  - gcPGM when both vertex and fragment shaders are specified

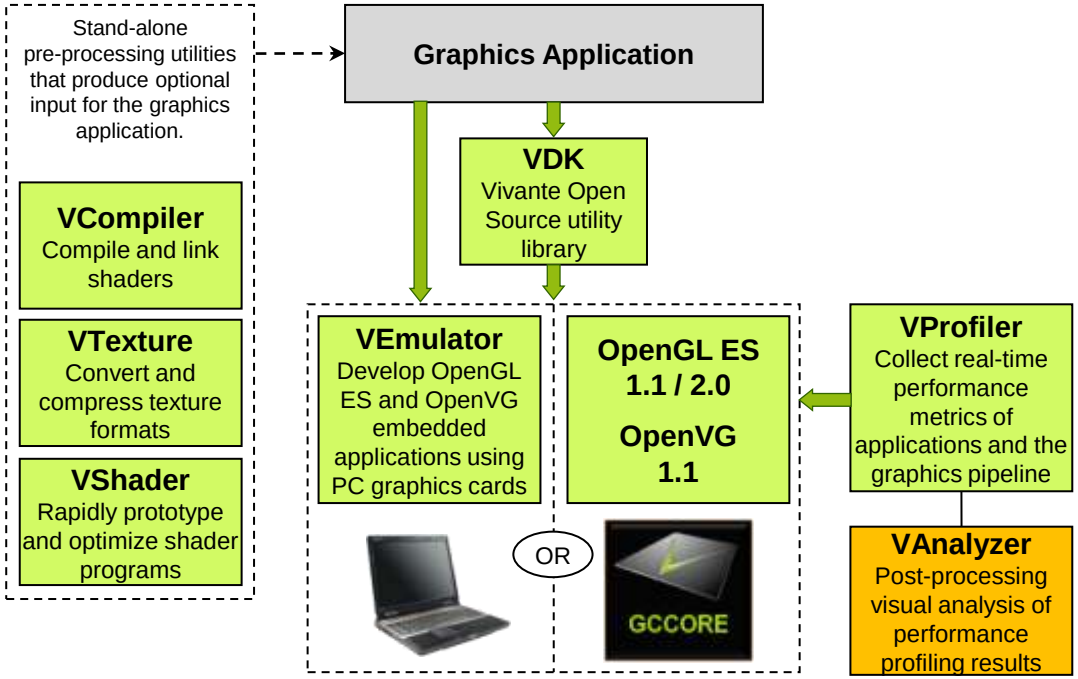
VProfiler

- VProfiler is a profiling tool which collects performance statistics of an application and the graphics pipeline during runtime.
- VProfiler can track GPU performance counters over any range of frames.



VAnalyzer

- VAnalyzer is a stand-alone utility for graphically displaying the data gathered by VProfile and aiding the developer to understand the graphics performance.



VAnalyzer

A developer can display different combinations of counters from the .gcprof performance database to illustrate resource utilization for any portion of their application.

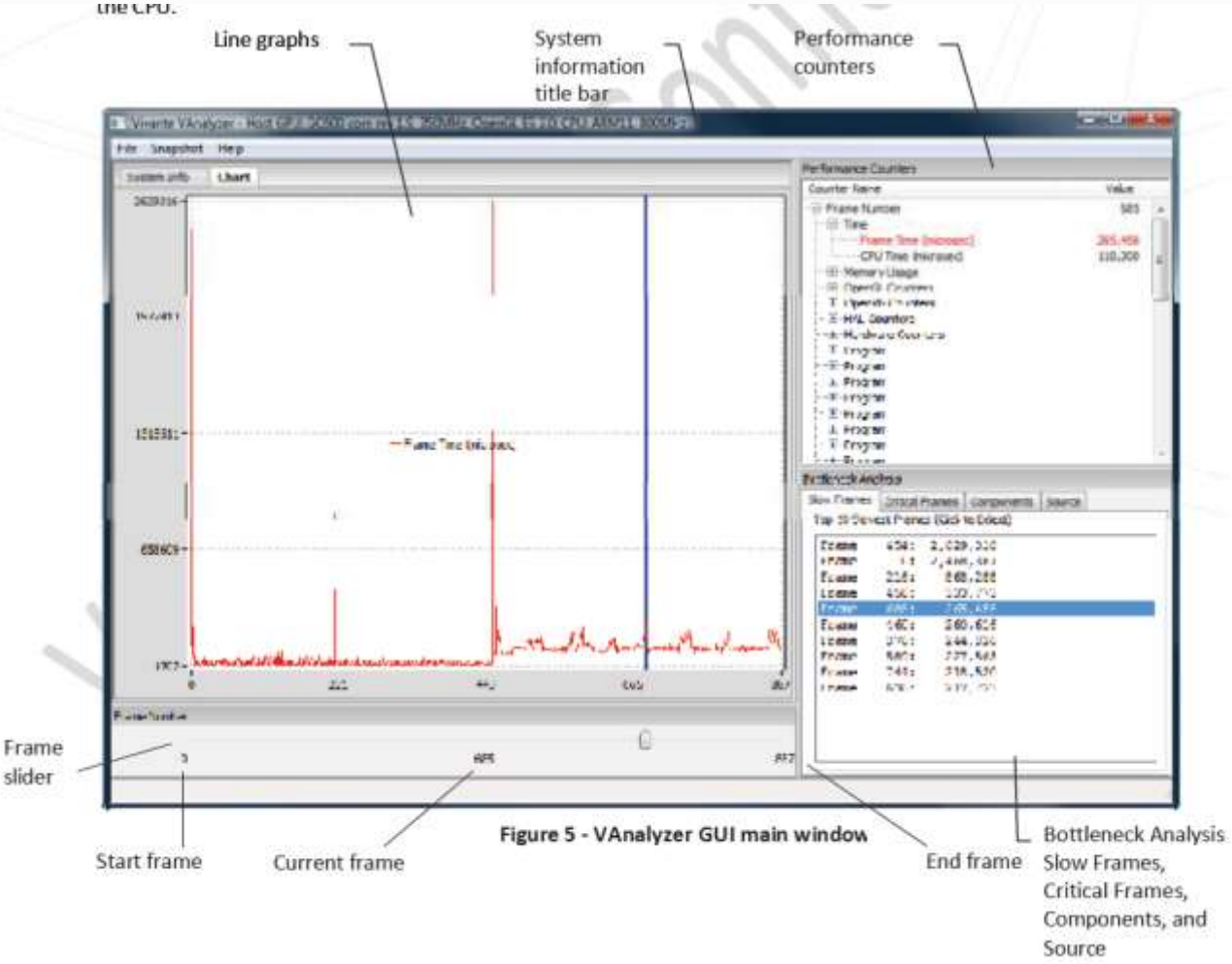


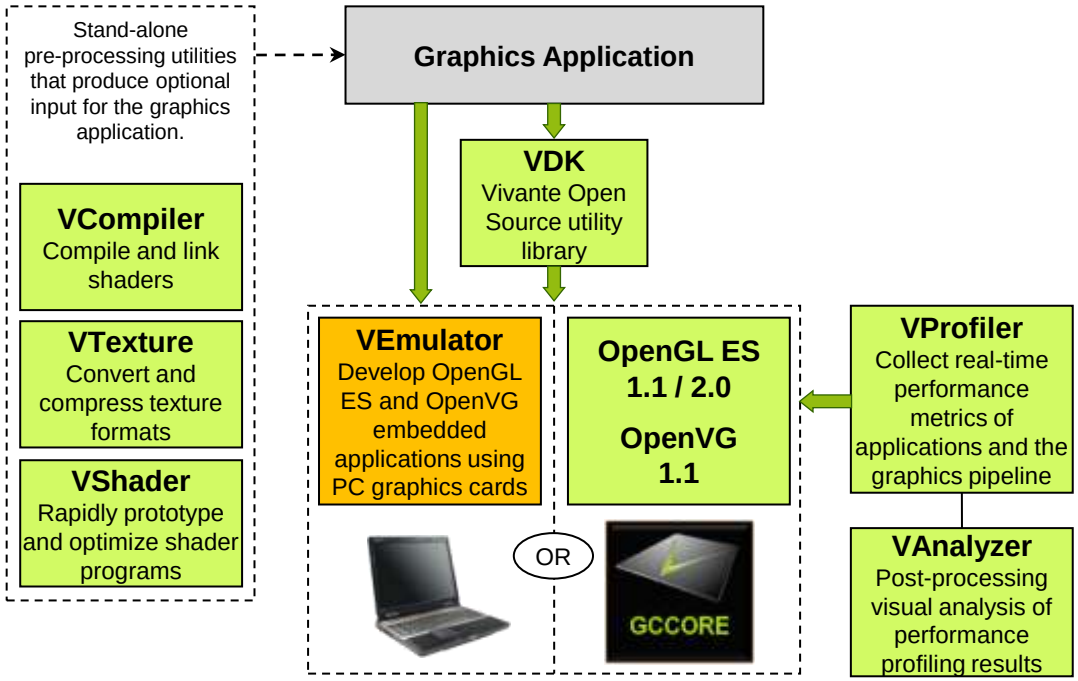
Figure 5 - VAnalyzer GUI main window

VProfiler + VAnalyzer Advantage

- Profile any frame from any application
- Browse through every counter and display them graphically
- Observe application performance bottlenecks
- Measure the system utilization for applications
- Realtime transfer of profile data over network connection

VEmulator

- VEmulator enables developers to develop and test applications on PC platforms that use popular graphics cards



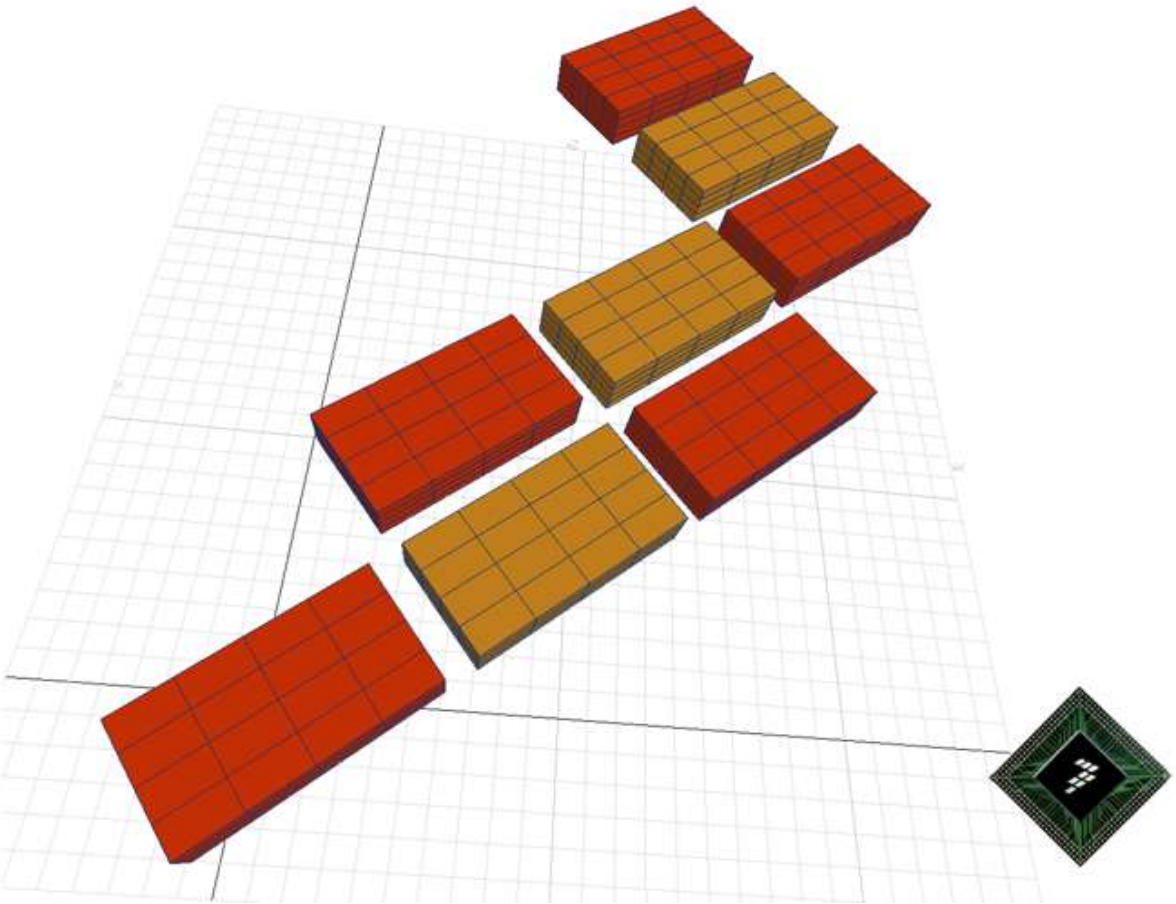
VEmulator

- PC Emulator for embedded graphics APIs. No performance throttling, just API remapping

- Currently supports
  - OpenGL ES 1.1
  - OpenGL ES 2.0
  - OpenCL EP 1.1

- Enables writing and testing applications on Windows PC platforms with popular graphics cards

Graphics Debug Tips & Tricks



OpenVG Tips and Tricks: i.MX5 to i.MX 6 differences

- Both GC2000 and GC355 can render OpenVG
  - Was also possible on i.MX5, but Z430 performance was too low
  - EGL extension will allow users to pick (details not available yet)
- Which to use?
  - GC2000: Software based tessellation + caching, but higher fillrate and texture rate; Only 8x line AA, lower quality
    - OpenVG with fewer or static lines (only matrix transforms) will have higher FPS performance (though will look more course)
  - GC355: Hardware tessellation with 16x AA dedicated line AA + render; HW gradients, etc.; ¼ the fill & texture rate of GC2000
    - OpenVG content that needs QoS (like in i.MX5); High quality needles, etc; VG content with new splines each frame (no tessellation penalty) or with a heavy amount of lines / splines;

OpenGL ES Tips and Tricks

- General Tips
  - Keep an eye on your config and state
  - Minimize state changes
  - Use compressed textures
  - Batch your calls as much as possible
  - Prevent uploads (FBOs, VBOs, TexImage2D, etc.)
  - Optimize your shaders (reduce branching, move to complexity to vertex shaders, etc.)

OpenGL ES Tips and Tricks: i.MX5 to i.MX 6 differences

- GC2000 is less sensitive to vertex complexity
  - Z430 chunking re-ran the vertex shader at each tile hit and could be sensitive to vertex complexity in high-geometry scenes.
  - Pushed complexity to pixel side
- Pipeline stalls (flushes) were more painful in Z430
  - Still BAD in GC2000 (as with all GPUs), but not as much
- GC2000 (and most IMRs) are more fillrate sensitive
  - Be careful with features that directly affect fillrate
  - AA has higher impact when it is not outweighed by vertex complexity
    - Many i.MX5 AA performance issues were actually vertex impacts
  - Use of more aggressive Z buffering (front to back sorting) has higher payoff

Other Graphics Debug Tips & Tricks

- Keep an eye on your CPU load
  - In Linux, use ‘top’
- High CPU load can be indicative of bad API usage
  - You should try to avoid data ‘uploads’ (either texture or vertex) on a frame-by-fame basis
  - Use VBOs instead of arrays in your draw calls
  - Use DirectVIV / PBOs / EGL Images instead of teximage data

OpenCL General Tips

- Try and eliminate serial portions of code. OpenCL architectures (GPUs specifically) really shine with unconstrained data parallel tasks.
- Try and eliminate any syncs if possible. Syncs cause waiting and execution bubbles and should be avoided. Some algorithms can use multiple data passes to avoid sync usage (Integral Image being a prime example) and may be much faster despite the increased computation and memory traffic of the additional pass.
- Similarly, data sharing and buffer passing back and forth to the CPU can cause performance loss due to GPU stalls and cache flushes. If you must switch, it is many times better to limit the break to one point (making a whole block execute on the GPU) even if small stages in the pipeline are more efficient on the other processor.
- Take advantage of the memory scoping. Properly group work-items into work-groups is key to getting efficient data access and speed.
- Reduce number of load/store requests for better latency hiding. Note that this may increase temporary register usage which may limit work group size.
- Precompute constants in application and pass as parameter or macro
- Use native built-in functions (e.g. native_divide)

GC2000 OpenCL Tips & Tricks

- GC2000 in MX6Q is a vec4 floating point SIMD engine, so vector math always prefers 4 inputs (or a multiple of 4) for maximum math throughput. Use vector addressing liberally (e.g. a4.xyzw)
- Both integer and floating point math is natively 32b. 8 and 16b primitives will still use 32b registers, so there is no gain (from the math computation) in going with lower sizes.
- For integer math, there is one 32-bit integer adder and one 32-bit integer multiplier per core
- Floating point computation supports “round-to-zero” only (round-to-nearest-even is not required if round-to-zero is supported). Use round-to-zero built-in functions (e.g. convert_float_rtz)
- For the most efficient use of the GPU L1 cache, data accesses should be 16B
- Work group size should be an integer multiple of the GPU's internal thread group size (16 on GC2000) to get the optimum hardware usage.
- Branching should be avoided if possible and branch penalties depend on the percentage of work items that go down the branch paths.

Q&A

- Thank you

Facebook.com/Freescale
Tag yourself in photos
and upload your own!

Tweeting?
Please use hashtag
#FTF2012

Session materials will be posted @ www.freescale.com/FTF
Look for announcements in the FTF Group on LinkedIn or follow Freescale on Twitter

freescale™


freescale™