Data Flow

1. Application
2. Modeling Transformations
3. Viewing, Projection, Clipping
   * Vertices and Fragments
4. GPU VRAM
5. Framebuffer
6. Displays
**CPU-GPU Boundary**

- **CPU**
  - 3D Application
  - 3D API
    - OpenGL
    - DirectX/3D
  - API Commands/Calls
  - GPU Command/Data Stream
  - GPU Command
    - Primitive Assembly
      - Pre-transformed Vertices
      - Transformed Vertices
      - Programmable Vertex Processors
  - Rasterization & Interpolation
    - Triangles
    - Lines
    - Points
    - Rasterized Pre-transformed Fragments
    - Programmable Fragment Processors
  - Raster Operations
    - Transformed Fragments
  - Frame Buffer
    - Pixels
  - Pixel Location Stream
  - Pre-transformed Fragments
  - Transformed Vertices

- **GPU Boundary**
  - GPU Command/Data Stream
  - Vertex Index Stream
  - Pre-transformed Vertices
  - Transformed Vertices
  - Programmable Vertex Processors
  - Programmable Fragment Processors
1-Bit FrameBuffer

Graphical representation of 1 bit color
24-Bit FrameBuffer

Graphical representation of 24 bit color

The 24-bits associated with each pixel in the frame buffer are split into three 8-bit groups to specify the pixel's red, green, and blue (RGB) components.
Framebuffer-Display

Row Ctr \rightarrow Col Ctr \rightarrow CLK > 1024x768x50Hz

Frame Buffer

0 1 2 3 4 ...... 1023

0 1 2

767

8x3=24 Bits

LCD Display

1024x768 Pixel LCD

Refresh screen 50 times a Sec
Displays
Large Displays

and Very Large
Small Displays

• Small

• and very small

IEEE Spectrum, 2009
Babak A. Parviz
Univ. Washington, Seattle
The Ultimate Display

• Is obviously the eye
• Retinal Projection
LCD Displays

- Most common
- High production volume
- Lower costs
- Energy efficient
- Space efficient
LC = Liquid Crystals

• What are LC’s?
  ▪ Substance exhibiting states between a liquid and a solid.
  ▪ Crystals re-align when an electric field is created by applying a voltage between two electrode plates.
Basic Principle

- Nematic Liquid Crystals
- Natural state is to align with grooves
Enhancements

- **Transmissive** = light from a **backlight** passes once thru the pixels out to your eye. Notebook computers use transmissive displays.

- **Reflective** = light from the **environment** is reflected at the back of the display and passes back thru the pixels out to your eye.

**Transflective** = transmissive + reflective.
How they look...

iPod transmissive rows only

Nokia 6230i transmissive only

iPod reflective rows only

Nokia 6230i reflective only
Newer Displays

• Different screens under the microscope.

iPad2 9.7”
1024x768
132ppi

Samsung Galaxy S4
HTC, Nexus
1280 x 768

Galaxy S5 AMOLED
432ppi
What is PPI?

• PPI = Pixels Per Inch

10 PPI  
2,54 cm

20 PPI  
2,54 cm

• However, manufacturers also use standard TV scanline refresh.

• Example 1080p is a full HD, 1080 lines progressive scan.

• That is: 1920x1080
Transflective TFT Display

• Employs two complimentary display techniques to improve visibility across a wider range of ambient light conditions.

• Each pixel has separate transmissive and reflective areas can be seen in the close-up views of a single pixel.
LCD Types

1. TN: Twisted Nematic (1971)
2. MVA: Multi-domain Vertical Alignment (1999)
Twisted Nematic

• TN first introduced by Schadt, Helfrich and Fergason in 1971 (ILIXCO)
• Traditional LCD

Prior to this: Helfrich Hoffman in 1969
Le Locle Central Lab in Switzerland;
rotation nematic liquid crystal effect
Multi-domain Vertical Alignment

1999 Fujitsu.

Can give very good viewing angles
Simpler manufacturing – no rubbing
In-Plane Switching

• Hitachi 1992 and 2002
• Voltage is applied in the same plane as the substrate

AS-IPS
- IPS-Pro:
  • High Image Quality
  • Very good contrast
  • High brightness
  • Wide aperture
  • Variations:
All FP Displays: Raster

• All flat panel displays have the same basic characteristics:
  
  - Active Matrix = address
  
  - Picture Element = Pixel
  
  - Sub-pixel elements: R, G, B
Inside the Engine
Basic Rendering Pipeline

Geometry Data
3D Transformations
Triangulation
3D-2D Projection
Rasterization
Frame Buffer
Short History

Graphics Rendering Pipelines

- Pre-1980’s Customized Software Rendering
- Pre 2001 SGI
- DX8-DX10 Sun Microsystems
- Larrabee Customized Pipeline
CPU-GPU Boundary

3D Application or Game

3D API Commands

3D API: OpenGL or Direct3D

CPU - GPU Boundary

GPU Command & Data Stream

GPU Front End

Vertex Index Stream

Assembled Polygons, Lines & Points

Pixel Location Stream

Raster Operations

Frame Buffer

Pretransformed Vertices

Transformed Vertices

Rasterized Pretransformed Fragments

Transformed Fragments

Programmable Vertex Processor

Primitive Assembly

Rasterization & Interpolation

Programmable Fragment Processor

Pixel Updates
GPU Evolution

• **OpenCL:**
  - Open Computing Language
  - Runs on CUDA
  - Heterogeneous Computing

Future
CUDA, DX11 Compute
OpenCL

CUDA (PhysX, RT, AFSM...) 2008 - Backbreaker
Advanced Flow Simulation Modeling

DX7 HW T&L 1999 - Test Drive 6
DX8 Pixel Shaders 2001 - Ballistics
DX9 Prog Shaders 2004 - Far Cry
DX10 Geo Shaders 2007 - Crysis
Current GPU: GTX 1070

Based on NVIDIA GeForce Pascal Architecture
1920 Cuda Cores
120 Texture Units
256 GB/s Memory Bandwidth
150W TDP Total Power
PolyMorph Engine

New GF100 PolyMorph and Raster Engines

- Dedicated graphics hardware
- PolyMorph Engine
  - World space processing
- Raster Engine
  - Screen space processing

PolyMorph Engine
- Vertex Fetch
- Tessellator
- Viewport Transform
- Attribute Setup
- Stream Output

Raster Engine
- Edge Setup
- Rasterizer
- Z-Cull
GPUs Basics

[Diagram of GPU components including Shader Core, Tex, Input Assembly, Rasterizer, Output Blend, Video Decode, Work Distributor, HW or SW?]
A diffuse reflectance shader

Shader programming model:

Fragments are processed independently

But, there is no explicit parallel programming

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```
Compile the shader program

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```
Execute the shader

Fetch/Decode

ALU (Execute)

Execution Context

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
CPU core style execution

- Fetch/Decode
- ALU (Execute)
- Execution Context
- Data cache (a big one)
- Out-of-order control logic
- Fancy branch predictor
- Memory pre-fetcher
Slimming down

Idea #1:
Remove components that help a single instruction stream run fast
Two core – two fragments

fragment 1

Fetch/Decode

ALU (Execute)

Execution Context

fragment 2

Fetch/Decode

ALU (Execute)

Execution Context

<diffuseShader>: sample r0, v4, t0, s0
mul r3, v0, c0[0]
madd r3, v0, c0[1], r3
add r3, v2, c0[2], r3
cmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
Four cores – Four fragments
Share an instruction stream

• Many fragments should be able to share an instruction stream
A simple core

- Fetch/Decode
- ALU (Execute)
- Execution Context
Add ALUs

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD
Single Instruction Multiple Data
One fragment using scalar ops
Vectorized Shader

New Compiled Shader

8 fragments using vector ops

<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul vec_r3, vec_v0, cb[0]
VEC8_madd vec_r3, vec_v1, cb[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb[2], vec_r3
VEC8_clamp vec_r3, vec_r3, 1(0.0), 1(1.0)
VEC8_mul vec_00, vec_r0, vec_r3
VEC8_mul vec_01, vec_r1, vec_r3
VEC8_mul vec_02, vec_r2, vec_r3
VEC8_mov r03, 1(1.0)
Multiple Frags

Sequence of operations:

1. Fetch/Decode
2. ALU 1
3. ALU 2
4. ALU 3
5. ALU 4
6. ALU 5
7. ALU 6
8. ALU 7
9. ALU 8
10. Ctx
11. Ctx
12. Ctx
13. Ctx
14. Ctx
15. Ctx
16. Ctx
17. Ctx
18. Shared Ctx Data

Code snippet:

```cpp
<VEC8_diffuseShader>
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)
VEC8_mul vec_o0, vec_r0, vec_r3
VEC8_mul vec_o1, vec_r1, vec_r3
VEC8_mul vec_o2, vec_r2, vec_r3
VEC8_mov o3, 1(1.0)
```

Color mapping:

- Yellow: ALU 1
- Orange: ALU 2
- Red: ALU 3
- Blue: ALU 4
- Gray: ALU 5
- Yellow: ALU 6
- Orange: ALU 7
- Red: ALU 8
- Blue: Ctx
- Gray: Shared Ctx Data
8 Fragments

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)
VEC8_mul vec_o0, vec_r0, vec_r3
VEC8_mul vec_o1, vec_r1, vec_r3
VEC8_mul vec_o2, vec_r2, vec_r3
VEC8_mov o3, 1(1.0)
```
128 Fragments in Parallel

16 cores = 128 ALUs

16 instruction streams
128 Vertices/Fragments Primitives
Results
Display Map and Shading

Displacement Map and Shading
Create Visual Realism

Terrain Displacement Map
Real-Time Water Flow

Water (Demo)

NVIDIA embargoed information
Real-Time Hair

Hair (Demo)

- Combine tessellation, geometry shading and compute to generate hair
Summary

- CPU-GPU Framework
- GPU Organization
- Graphics Pipeline
- Frame Buffer
- Displays
The End