上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

1896 1935 1987 2006

游戏设计与开发

GPU实时图形管线

GPU: Real-time Graphics Pipelines

上海交通大学软件学院
数字艺术实验室
Digital ART LAB

Outline

- Real Time Requirement
- A Conceptual Rendering Pipeline
- Evolution of GPU
  - —Hardware
  - —APIs
  - —Shaders

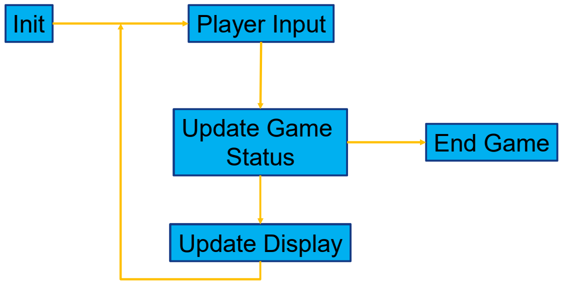
REAL TIME REQUIREMENT

Building A Game

- Background Scene (e.g. sky, terrain)
- Static Objects
- Movement of Objects
- Users' Control
- Collision and Response
- Others

3D Game Loop

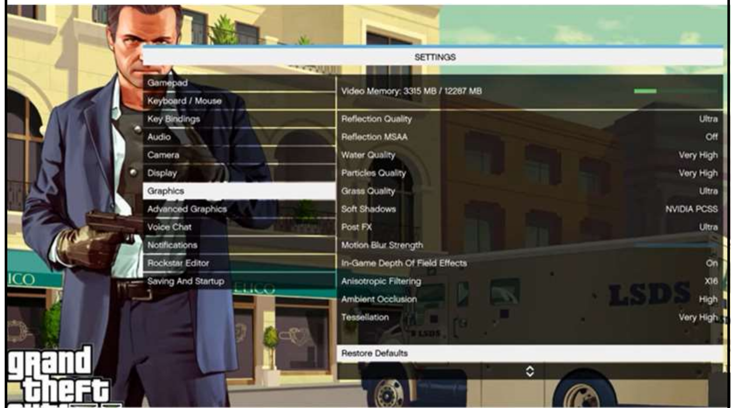
- **The “Game Loop” (Main Event Loop) :**



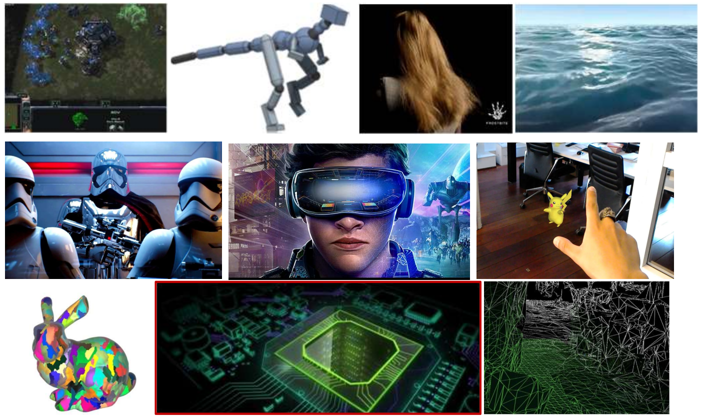
How Fast Does my Game Loop Need to Run?

- ANSWER: *“It depends…”*
  - Visual displays: 25-30 Hz or higher，90~120Hz for VR display
  - Head-tracking for HMDs: 60 Hz, but even only 2-5ms of latency yields *display lag*, which often quickly causes users to lose their lunch…
  - Haptic displays: much higher update rates (500 - 1000 Hz)
  - Multitasking / Multiprocessing: allows for different update rates for different types of output displays
- Main requirement: Real Time 3D Graphics

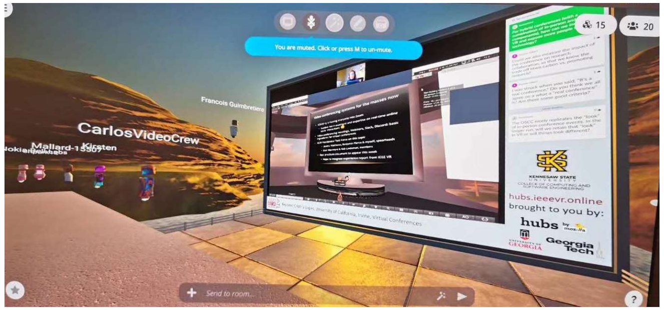
E.g. Game Graphics Setting



Game Programming Techniques



VR in VR (IEEE VR 2020, Mar.23-26)



ACM A.M. Turing Award (2019)

- **PIONEERS OF MODERN COMPUTER GRAPHICS: 3D CGIs**

Patrick M. Hanrahan

- Volume rendering; light field rendering
- RenderMan Shading Language: Shader, →GLSL
- Brook (a language for GPUs) → Nvidia's CUDA
- GPU enabled:
  - CGI for animated films and games
  - VR/AR
  - high performance computing
  - machine learning for AI

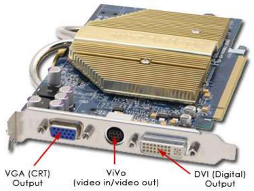
Edwin E. Catmull

Curved patches displaying
Z-Buffering
Texture Mapping
Catmull-Clark Subdivision

Lucas Films → Pixar (1986)

Graphics Hardware and APIs

- **GPU(Graphics Processing Unit):**
  - **Nvidia, AMD, Intel**



- **API:**
  - **Microsoft: DirectX 9, DirectX 10, DirectX 11,…**
  - **OpenGL ARB: OpenGL 1.0, 2.0, 3.0, 4.0,…**

3D Graphics Software Tools

- Low-level 3D graphics APIs:
  - hardware-independent, transparent access
    - window-system and OS independent
    - network-transparent
  - Commonly-Used 3D Graphics APIs:
    - Realtime: OpenGL, Direct3D (*part of* **DirectX**), Vulkun
    - Offline: Renderman
- High-level 3D graphics tools:
  - — 3D Graphics Engine: OGRE, OpenSceneGraph, VTK
  - — 3D Game Engine: Unity, Unreal

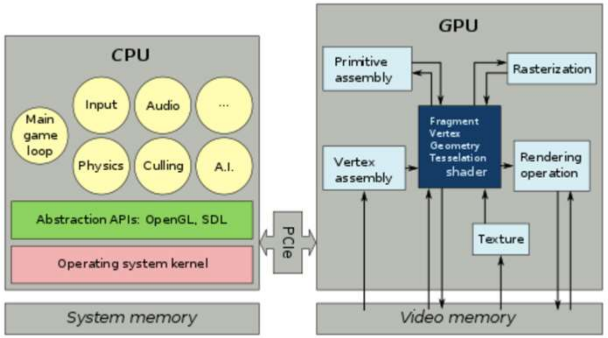
OpenGL vs. Vulkan

Vulkan: lower-level; multi-core; parallel tasking
- lower overhead
- more direct control over the GPU
- and lower CPU usage

OpenGL	Vulkan
One single global state machine	Object-based with no global state
State is tied to a single context	All state concepts are localized to a command buffer
Operations can only be executed sequentially	Multi-threaded programming is possible
GPU memory and synchronization are usually hidden	Explicit control over memory management and synchronization
Extensive error checking	Vulkan drivers do no error checking at runtime; there is a validation layer for developers

Digital ART Lab,SJTU

CPU-GPU Cooperation



Digital ART Lab,SJTU

GPU vs. CPU

- **GPU**
  - —Data Parallelism
  - —More cores, but low clock speed
  - —VRAM, fast but small size
  - —Low cache memory
- **CPU**
  - —Task parallelism
  - —Less cores, but high clock speed
  - —External RAM, slow but large in size
  - —High cache memory

Digital ART Lab,SJTU

Graphics Computations on CPU

- **Vertices to pixels:**
  - —Transformations done on CPU
  - —Compute each pixel in series…slow!
- **Example:**
  - 1 million triangles
  - x 100 pixels per triangle
  - x 10 lights
  - x 4 cycles per light computation

= 4 billion cycles

Digital ART Lab,SJTU

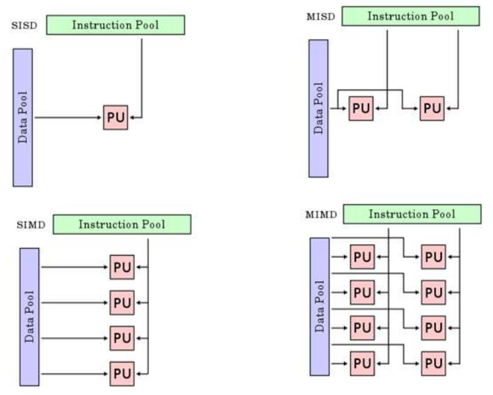
Flynn's Taxonomy

- A taxonomy of computer architecture
  - —Micheal Flynn in 1966
- Based on two things:
  - —Instructions
  - —Data

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Digital ART Lab,SJTU

Which one is closest to GPU?



Digital ART Lab,SJTU

A CONCEPTUAL RENDERING PIPELINE

Digital ART Lab,SJTU

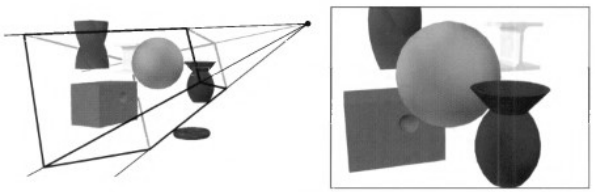
The Graphics Rendering Pipeline

- *rendering pipeline* or *the pipeline*
  - The core component of real time graphics
  - main function: generate, or render, a 2D image, given a virtual camera, 3D objects, light sources, lighting models, textures, and more.

Digital ART Lab,SJTU

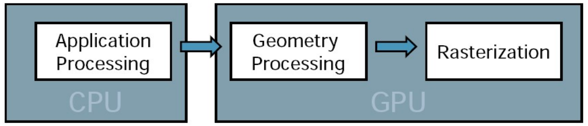
Using the pipeline

- Object (model, color, texture, material)
- Camera/view
- Light source



The Basic Construction

- A conceptual rendering pipeline
- 3 conceptual stages:



Digital ART Lab,SJTU

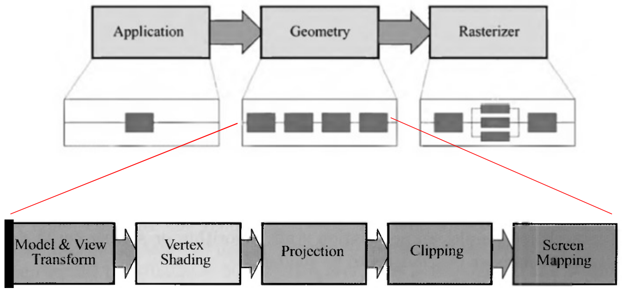
High-Level Pipeline

Application	Geometry (a.k.a. "vertex pipeline")	Rasterization (a.k.a. "pixel pipeline" or "fragment pipeline")
Handle input	Transform	Rasterize (fill pixels)
Simulation & AI	Lighting	Interpolate vertex parameters
		Look up/filter textures
Culling	Skinning	Z- and stencil tests
LOD selection	Calculate texture coords	Blending
Prefetching		

Digital ART Lab,SJTU

Substages of Geometry Stage

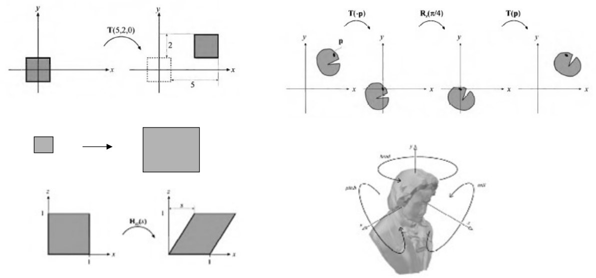
- A pipeline consists of several stages (N)
  —ideally give a speedup of a factor of N

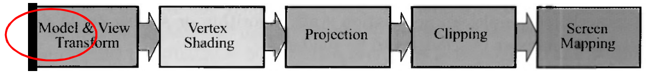


Digital ART Lab,SJTU

Model Transform

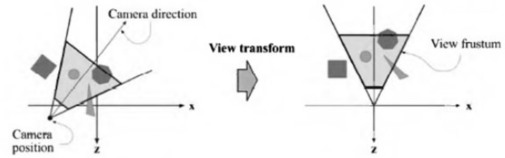
- model transform： model space → world space

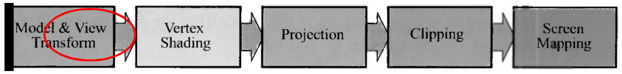




View Transform

- Purpose: place camera at the origin and aim it to look in –z , with y upward, x to the right
- Camera space (or eye space)
- right-hand or left-hand (API specific)





Transform Matrix

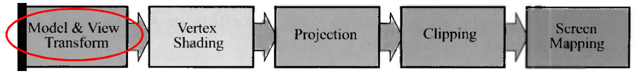
- Both are implemented as 4x4 matrices
  - —Model Transform
  - —View Transform

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & m \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

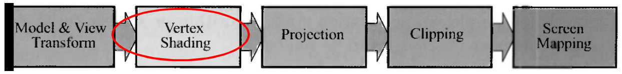
$$\begin{bmatrix} S_x x \\ S_y y \\ S_z z \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & 0 \\ e & f & g & 0 \\ i & j & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



Vertex Shading

- Modeling object appearance:
  - —Object's materials
  - —Effects of light sources
- ***Shading*** :
  - —Determining the effect of a light on a material
  - —By computing ***a shading equation*** at points on the object
    - Per-vertex operation
    - or Per-pixel operation

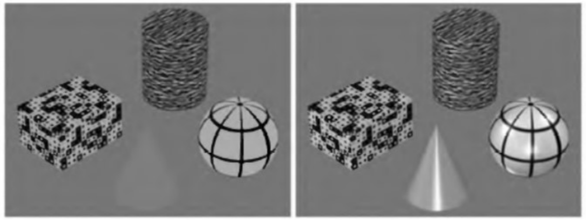


Vertex Shading

- Material data can be stored at each vertex:
  - —Points’ location
  - —A normal
  - —A color
  - —Other numerical information
- Shading can be performed in
  - —World space
  - —Model space
  - —Eye space

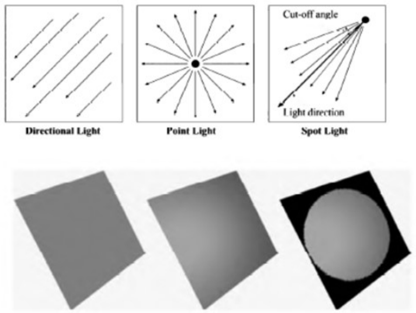
Lighting

- Lighting in fixed-function pipeline



Lighting

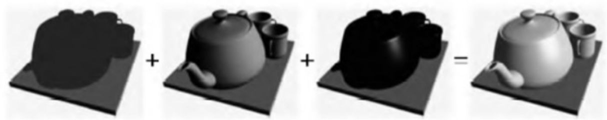
- Light sources



Lighting

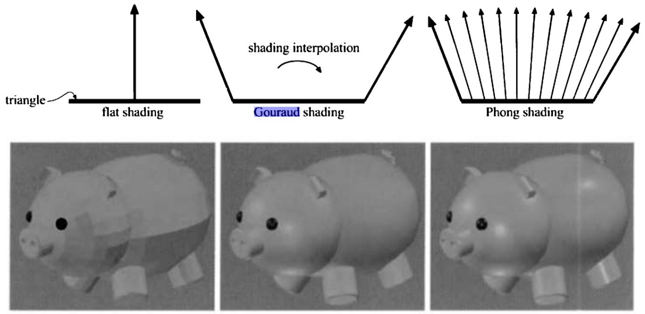
- Lighting Equation in fixed-function pipeline

$$\mathbf{i}_{tot} = \mathbf{i}_{amb} + \mathbf{i}_{diff} + \mathbf{i}_{spec} \qquad (4.14)$$



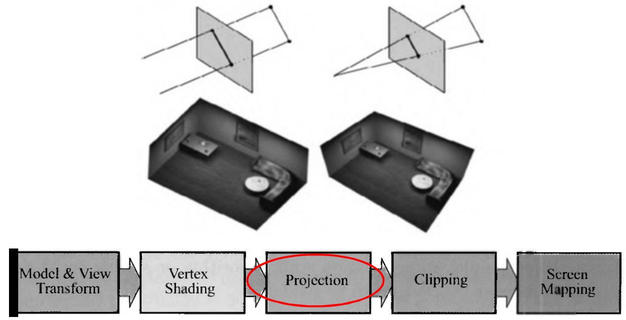
Shading Model

- **Shading Model:**
  - Flat shading, Gouraud shading, Phong Shading (not in fixed)

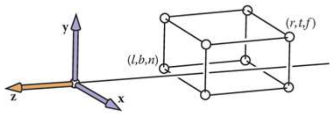


Projection

- View volume → unit cube (-1,-1,-1) and (1,1,1)
- Orthographic projection, perspective projection



Orthographic Projection

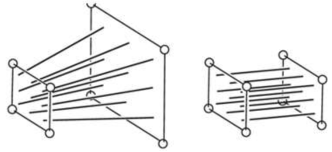


$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Orthographic → Perspective

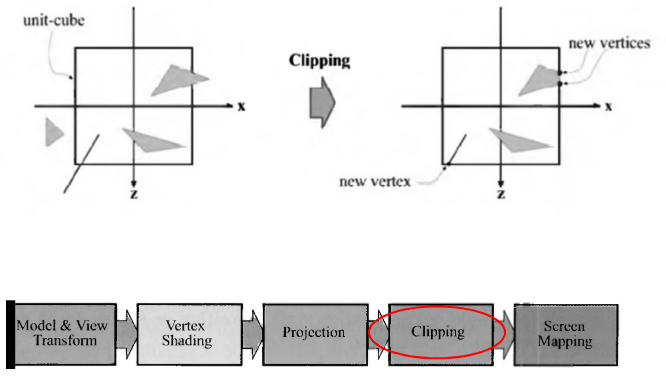
$$\mathbf{M}_{\text{per}} = \mathbf{M}_{\text{orth}}\mathbf{P}$$

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_{\text{per}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

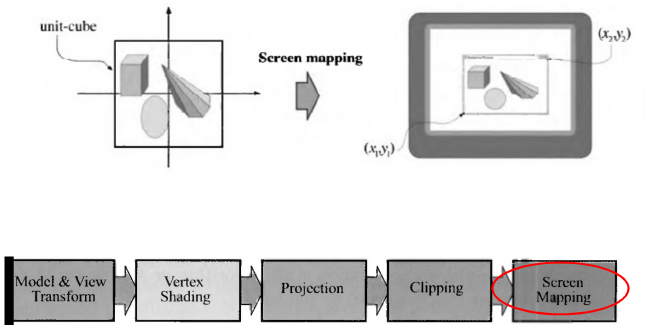


from Fundamentals of Computer Graphics, 4th Edition

Clipping

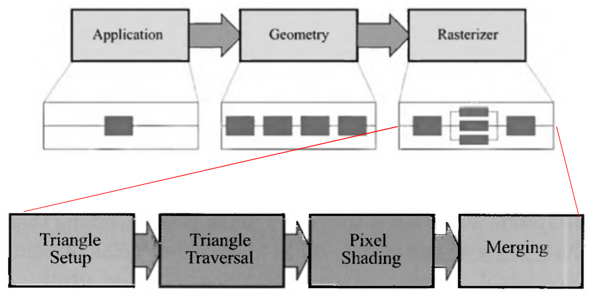


Screen Mapping

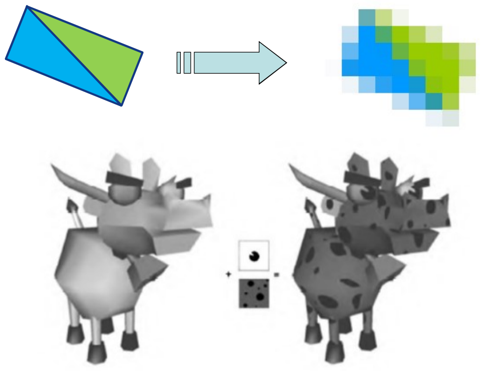


Substages of Rasterizer Stage

- Rasterizer: geometry → fragments
  - Continuous → discrete

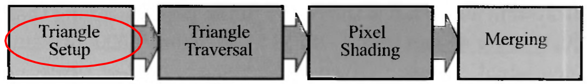


Rasterizer Stage



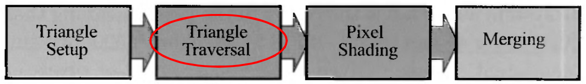
Triangle Setup

- Compute the differentials and other data for the triangle's surfaces
  - —used for scan conversion (next stage)
  - —for interpolation of various shading data produced by the geometry stage (next stages)
- Performed by fixed-operation hardware



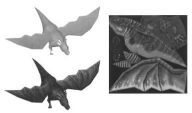
Triangle Traversal

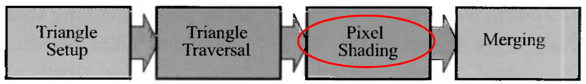
- Also called ***scan conversion***
- Find which samples or pixels are inside a triangle
  - —Check each pixel covered by the triangle
  - —Generate a ***fragment*** for the part of the pixel that overlaps the triangle
  - —Fragment properties: depth, and any shading data from the geometry stage:
    - generated using data interpolated among the 3 triangle vertices
- Performed by fixed-operation hardware



Pixel Shading

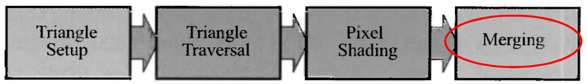
- Do any ***per-pixel shading*** computations
  - —Input: the interpolated shading data
  - —Output: one or more colors to be passed to next stage
- Executed by programmable GPU cores
- Many techniques can be employed here, e.g.
  - —Texturing: "glue" an image (1D,2D or 3D) onto the object





Merging

- Combine the fragment color produced with the color currently stored in color buffer
  - — *color buffer* : 2D array store color info for each pixel (e.g. RGB)
- Not fully programmable, but highly configurable (enable SFXs)
- Also resolving visibility: mostly depth test using Z-buffer (same size as the color buffer, store depth value)
  - — Depth test：compare z-value before rendering to a pixel → update Z-buffer and color buffer if closer
  - — Order-independent for opaque object, but need back-to-front for transparent object (major weakness of Z-buffer)



Merge stage: more buffers and operations

- *Color buffer*: colors
- *Z-buffer*: z-values (*depth test)*
- *Alpha channel* (color buffer): opacity values
  - *alpha test* (==, >=, …) optional before the *depth test*
  - *E.g. ensure fully transparent fragments not affect z-buffer*
- *Stencil buffer*
  - *Record locations of the rendered primitive*
  - *offscreen buffer (typically 8 bits/pixel)*
  - *Control rendering into the color buffer and Z-buffer*
  - *Powerful tool for SFXs: e.g. a circle window*
  - *Raster operations (ROP) or blend operations*
- *Frame buffer (all the buffers, or color + Z-buffer)*
- *Accumulation buffer (images accumulated using a set of ROP)*
  - *e.g. motion blur, depth of field, antialiasing, soft shadows, …*
- *Double buffering:* front buffer & back buffer, swapped during *vertical retrace* (avoid seeing uncompleted screen)

Digital ART Lab,SJTU

Various Pipelines

- *Real-time rendering pipelines:* decades of API and GPU evolution for real-time rendering applications
  - *fixed-function pipeline (e.g. Nitendo's Wii, maybe the last)*
    - *On-off configuration*
  - *Programmable GPUs (the modern way!)*
    - *Program exactly operations in substages*
- *Offline rendering pipelines*: different evolution paths
  - Film rendering: commonly *micropolygon pipelines*
  - Academic, and predictive rendering applications (Pre-Viz): *ray tracing renderers*

Digital ART Lab,SJTU

EVOLUTION OF GPU HARDWARE

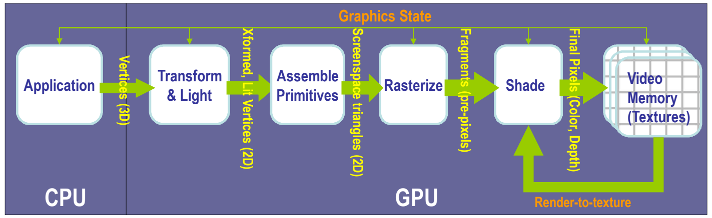
Digital ART Lab,SJTU

Graphics Processing Unit (GPU)

- Evolution of hardware graphics accelerations:
  - Started at the end of the pipeline
  - Worked back up the pipeline
  - Hardware accelerator for higher-level application-stage algorithms
- 1999, NVIDIA GeForce256, coined the term GPU (vs. AMD VPU)
  - Hardware T&L (CPU vs. GPU) ➔ 3D FPS Game Card (bad for 2D apps)
  - 4 pixel pipelines: each has 4 pixel units + 1 texture unit
  - Triangle throughput: 15 Million/sec
  - Pixel throughput: 480 Million/sec
  - 2300 transistors (> Pentium III)
  - 0.22 micro process (heat problem vs. 0.18)
  - 256-bit display architecture
  - GeForce256 vs. Quadro

Digital ART Lab,SJTU

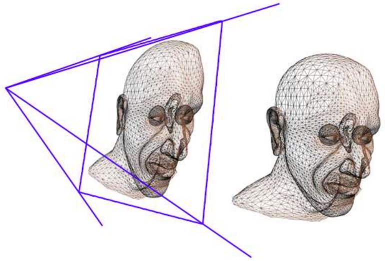
The Fixed-function Graphics Pipeline



- A simplified fixed-function graphics pipeline

Fixed-function GPU Pipeline: Transform

- Transform & light (a.k.a. vertex processor)
  - Transform from “world space” to “image space”
  - Compute per-vertex lighting



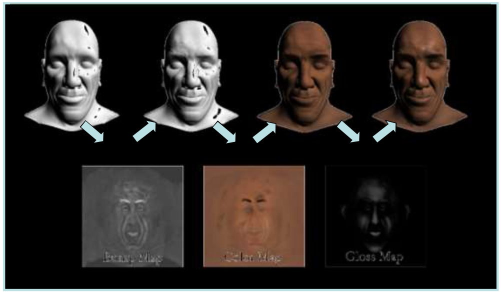
Courtesy Mark Harris

Fixed-function GPU Pipeline: Rasterize

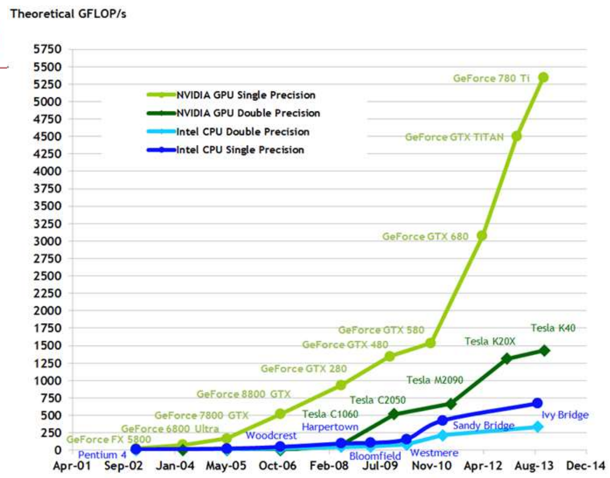
- Rasterizer
  - Convert geometric rep. (vertex) to image rep. (fragment)
    - Fragment = image fragment
      - Pixel + associated data: color, depth, stencil, etc.
  - Interpolate per-vertex quantities across pixels

Fixed-function GPU Pipeline: Shade

- Fragment processors (multiple in parallel)
  - Compute a color for each pixel
  - Optionally read colors from textures (images)

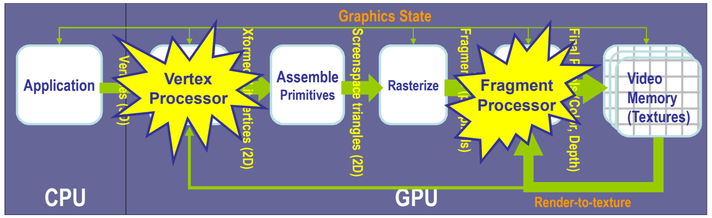


Courtesy Mark Harris



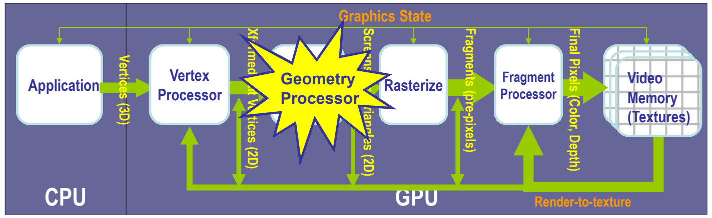
Floating-Point Operations per Second - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

The Modern Graphics Pipeline



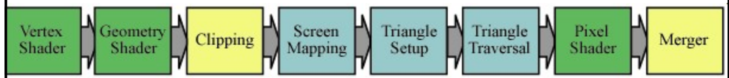
- Programmable vertex processor!
- Programmable pixel processor !

The Modern Graphics Pipeline



- Programmable primitive assembly!
- More flexible memory access!

Modern GPU implementation of rendering pipeline



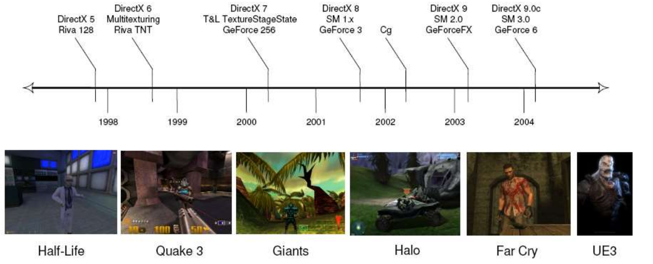
green : fully programmable
yellow: configurable but not programmable
blue: completely fixed in their function

Vertex Shader (fully programmable)
- Model and View Transform
- Vertex Shading
- Projection

- Geometry shader (optional) - (fully programmable)
  - Operate on the vertices of a primitive (point, line, or triangle)
- Clipping, screen mapping, triangle setup, triangle traversal stages (fixed)
- Pixel Shading - (fully programmable)
  - Pixel shading function
- Merge Stage (highly configurable)
  - Modifying the color
  - Z-buffer blend, stencil, and other buffers

Digital ART Lab,SJTU

DirectX Shader Model Timeline



Digital ART Lab,SJTU

Earlier GPU Architecture Evolution

	Product	New Features	OpenGL Version	Direct3D Version
2000	GeForce 256	Hardware transform & lighting, configurable fixed-point shading, cube maps, texture compression, anisotropic texture filtering	1.3	DX7
2001	GeForce3	Programmable vertex transformation, 4 texture units, dependent textures, 3D textures, shadow maps, multisampling, occlusion queries	1.4	DX8
2002	GeForce4 Ti 4600	Early Z culling, dual-monitor	1.4	DX8.1
2003	GeForce FX	Vertex program branching, floating-point fragment programs, 16 texture units, limited floating-point textures, color and depth compression	1.5	DX9
2004	GeForce 6800 Ultra	Vertex textures, structured fragment branching, non-power-of-two textures, generalized floating-point textures, floating-point texture filtering and blending	2.0	DX9c
2005	GeForce 7800 GTX	Transparency antialiasing	2.0	DX9c

Digital ART Lab,SJTU

DirectX 5 / OpenGL 1.0 and Before

- Hardwired pipeline
  - Simple API inputs
  - Small set of operations
- Example Hardware
  - NVIDIA RIVA 128
  - 3dfx Voodoo
  - S3 Virge
- Rigid data flow
  - No read-back from frame buffer

Digital ART Lab,SJTU

DirectX 6 / OpenGL 1.2

- Released 1998
- New Features
  - Multitexturing (in OpenGL since 1.3)
- Example Hardware
  - NVIDIA RIVA TNT
  - ATI Rage 128

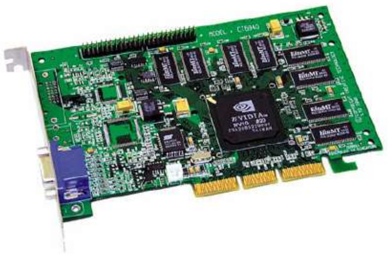
Digital ART Lab,SJTU

DirectX 7 / OpenGL 1.3

- Released 1999
- More work outsourced to GPU
  - Hardware Transformation and Lighting (T&L, Direct3D only)
- New Features
  - Texture Compression in OpenGL
- Example Hardware
  - NVIDIA GeForce 256
  - ATI Radeon 7200

Digital ART Lab,SJTU

Example: GeForce 256 (1999)



- Graphics Core:
  - 256-bit
- Memory Interface:
  - 128-bit
- Triangles/s:
  - 15 Million
- Pixels/s:
  - Up to 480 Million
- Memory:
  - Up to 128MB

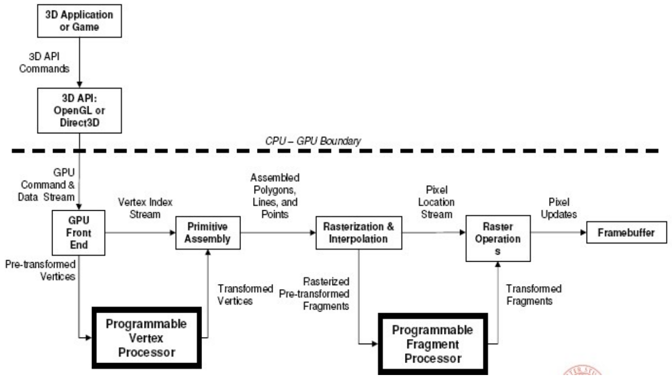
Digital ART Lab,SJTU

DirectX 8

- Released 2000
- Introduction of Shader Model 1.1
  - Shaders are GPU-run programs that manipulate Vertices or Pixels
  - Enables a plethora of new visual effects
  - Adds programmable processors to the graphics pipeline
- Example Hardware:
  - NVIDIA GeForce 3
  - ATI Radeon 9000

Digital ART Lab,SJTU

Adding Programmability

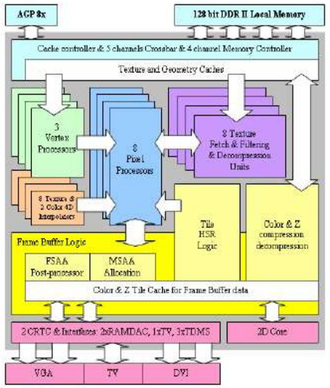


DirectX 9

- Released 2002
- Much more general programming paradigm
  - Branching
  - Floating point fragment programming
- Shader Model 3.0 (in 9.0c, 2004)
  - Big feature increment
- Example Hardware
  - NVIDIA GeForceFX (9.0)
  - NVIDIA GeForce 6200 (9.0c)

Example: Geforce FX (2003)

- **Vertex processor:**
  - execute vertex shaders
  - emulate T&L
- **Pixel processor:**
  - execute pixel shaders
  - emulate 2 integer or 1 float operations
  - Perform 2 Texture accesses
- **Texture & color interpolators**
  - interpolate texture coordinates and color values



OpenGL 2.0

- Released 2005
- OpenGL Shading Language (GLSL)
  - Vertex and fragment shaders
  - GLSL ties shaders to OpenGL API
- Point sprites
  - Particle effects
- Many more features

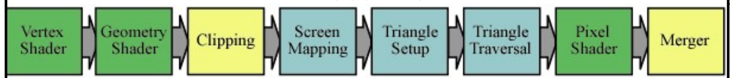
DirectX 10

- Released 2006
- Aligned with Windows Vista
- New Features
  - Geometry shaders; Streaming output; Arrays of surfaces and resource views; State encapsulation
- Break with Past
  - User mode drivers, even for DX9
  - Drivers do not implement Shader Models <4.0
  - No more fixed function (compiles legacy API calls to shaders)

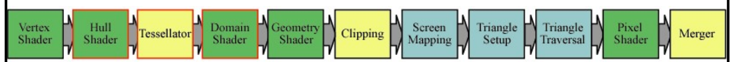
DirectX 11

- Direct3d 11, Released 2009
  - Windows Vista(With Patch)/Windows 7
  - Shader Model 5.0
  - Tessellation, Multithreaded rendering, Compute shaders, supported by hardware and software running Direct3D 9/10/10.1
- Direct3D 11.1
  - —Windows 8, Stereoscopic 3D Rendering, GPGPU
- Direct3D 11.2
  - —Windows 8.1, Tiled resources, GPGPU

DirectX 10



DirectX 11

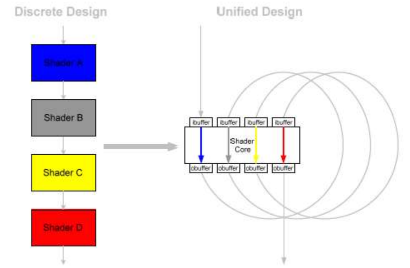


DirectX 12

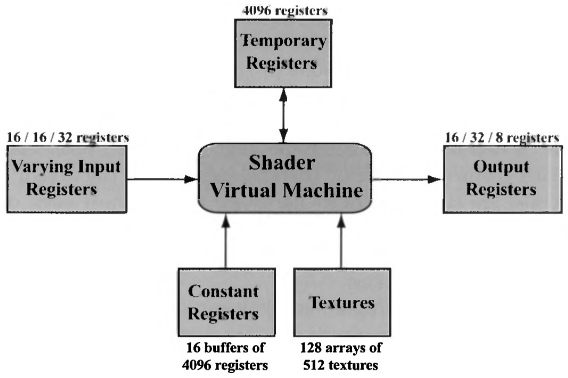
- July 2015
- Windows 10, Xbox One
- reduce driver overhead: “console-level efficiency”
- a lower level of hardware abstraction
  - —enabling future games to significantly improve multithreaded scaling and (decrease) CPU utilization
- claimed to be better than DirectX 11:
  - —50-70% faster
  - —>50% reduction in power consumption

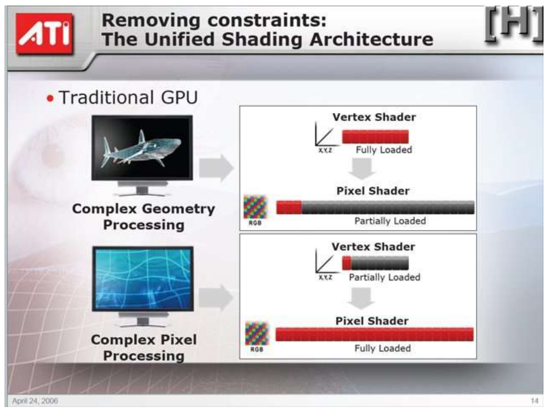
Unified Shader Design

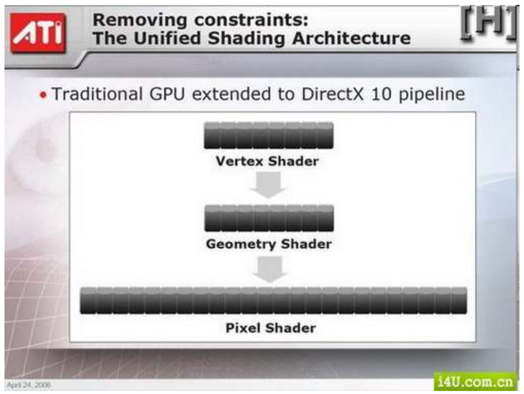
- Rendering pipeline no longer visible in hardware
- Unified shader cores process both vertices and pixels
- Shader Virtual Machine



Common Shader Core Virtual Machine





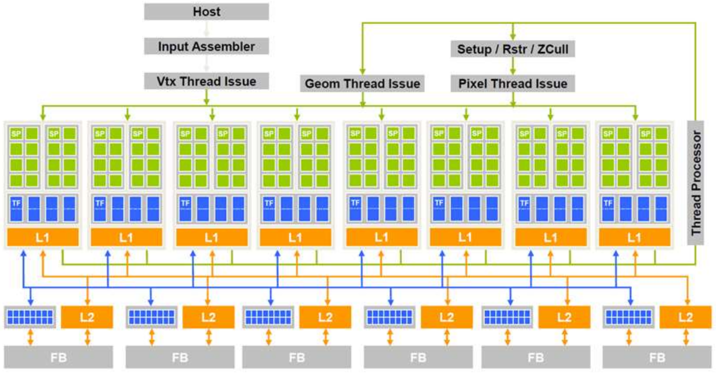


Impact of Unified Shaders

- All shading processes performed by a unified set of processors
- Fewer bottle-necks (i.e. in case of vertex or pixel dominant scenes)
- Better hardware utilization
- Hardware architecture no longer reflects the graphics pipeline
- Greater flexibility makes GPUs eligible for non-graphics applications (game physics, scientific applications)

→ Basically makes the GPU a massively parallel stream multiprocessor!

Example of GPU Architecture



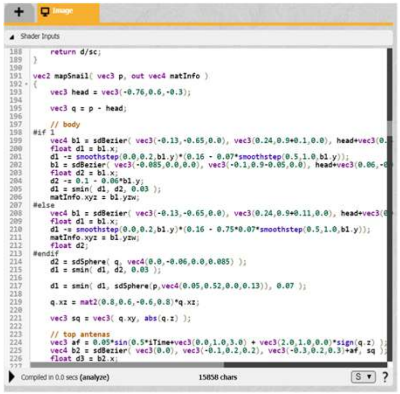
Digital ART Lab,SJTU

Programmable Shader Stage

- ***Common-shader core (API) (after DX10)***
  - ***Vertex, pixel and geometry shaders share a programming model***
  - ***Functional description seen by the application programmer***
- ***Unified shaders (GPU architecture)***
  - ***A GPU architecture that maps well to the common-shader core***
- ***Programming model:***
  - ***Shaders are programmed using C-like shading languages (e.g. Cg, HLSL, GLSL)***

Digital ART Lab,SJTU

ShaderToy Snail



- **GLSL pixel shader (~800 line codes)**
- **Procedural modeling and procedural lighting**
- **Raymatching**
- **distancefield**

Digital ART Lab,SJTU