Lesson 1 – Introduction

PV227 – GPU Rendering

Jiří Chmelík, Jan Čejka
Fakulta informatiky Masarykovy univerzity

22. 9. 2015

Outline

Outline

1. Organization

2. Introduction, history
   - Motivation
   - History
   - Pipeline
   - Shading Languages
   - Coordinate Spaces and Transforms
   - Homework

Course

- First lectures more theoretical, then mostly practical.
- Graphics is changing fast $\rightarrow$ only major language features will be introduced.
- Advanced features of OpenGL will be NOT covered.
- Teaching method = seminars $\rightarrow$ **active** participation ...

Course

- First lectures more theoretical, then mostly practical.
- Graphics is changing fast $\rightarrow$ only major language features will be introduced.
- Advanced features of OpenGL will be NOT covered.
- Teaching method = seminars $\rightarrow$ **active** participation ...

Figure: Taken from weebly.com

Requirements

To successfully pass the course:

- no more than 2 absences,
- success in final test (on the spot programming!),

Expectations:

- programming skills: C, C++
- knowledge of OpenGL (PV112)
- basic knowledge of basics principles of computer graphics (PB009)

Requirements

To successfully pass the course:

- no more than 2 absences,
- success in final test (on the spot programming!),

Expectations:

- programming skills: C, C++
- knowledge of OpenGL (PV112)
- basic knowledge of basics principles of computer graphics (PB009)

Contacts

- Jiří Chmelík
  - office: A412
  - e-mail: jchmelik@mail.muni.cz
- Jan Čejka
  - office: A419
  - e-mail: 324987@mail.muni.cz

Want to know more about GPUs?

There is a “parallel” course:

J006 – **Advanced GPU programming with Unity**, Mathieu Le Muzic
…only this year

- deferred rendering
- ambient occlusion
- marching-cubes
- …

Want to know even more about GPUs?

J006 – Advanced GPU programming with Unity, Mathieu Le Muzic
PV197 – GPU Programming, Jiří Filipovič:

- Introduction: motivation for GPU programming, GPU architecture, overview of parallelism model, basics of CUDA, first demonstration code
- GPU hardware and parallelism: detailed hardware description, synchronization, calculation on GPU – rate of instruction processing, arithmetic precision, example of different approaches to matrix multiplication – naive versus block-based
- Performance of GPUs: memory access optimization, instructions perormance, example of matrix transposition
- CUDA, tools and libraries: detailed description of CUDA API, compilation using nvcc, debugging, profiling, basic libraries, project assignment
- Optimization: general rules for algorithm design for GPU, revision of matrix multiplication, parallel reduction
- Parallelism in general: problem decomposition, dependence analysis, design analysis, parallel patterns
- Metrics of efficiency for GPU: parallel GPU and CPU usage, metrics for performance prediction of GPU code, demonstration using graphics algorithms, principles of performance measurement
- OpenCL: introduction to OpenCL, differences comparing to CUDA, exploiting OpenCL for hardware not accessible from CUDA
- ...

Outline

1. Organization

2. Introduction, history
   - Motivation
   - History
   - Pipeline
   - Shading Languages
   - Coordinate Spaces and Transforms
   - Homework

Outline

Motivation

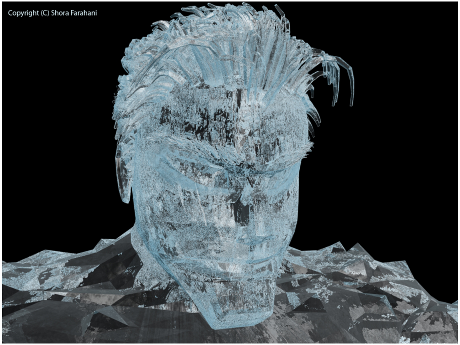

Figure: Taken from shoraspot.com

Figure: Taken from cgsociety.org

Why GPU?

- graphics computations are costly,
- graphics are "embarrassingly parallel",
- increasing model complexity, screen resolution, ...
- GPU is parallel co-processor.
- Nice demo: `http://youtu.be/-P28LKWTzrI`

Performance

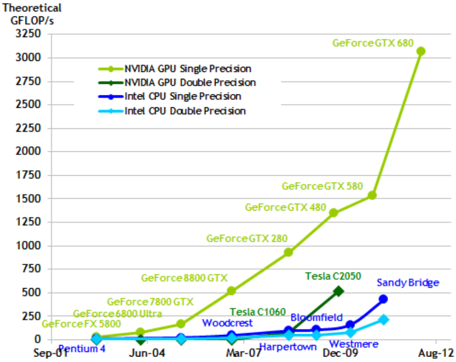

Figure: Taken from docs.nvidia.com

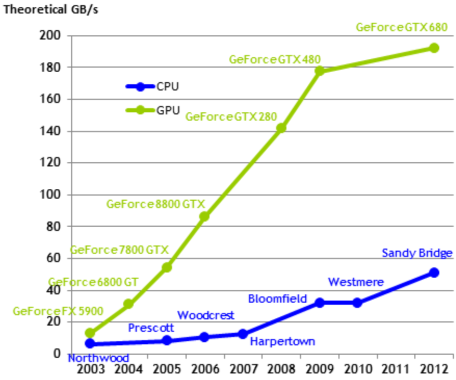

Figure: Taken from docs.nvidia.com

Performance

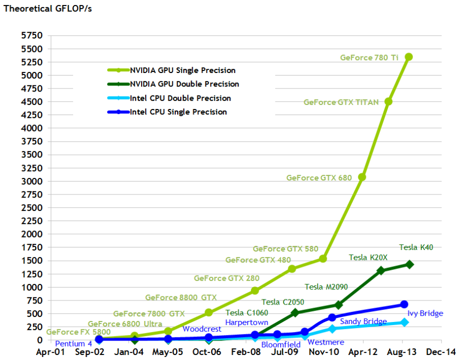

Figure: Taken from docs.nvidia.com

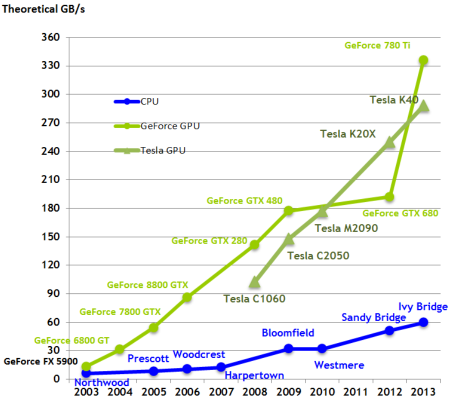

Figure: Taken from docs.nvidia.com

Shaders

Shaders are small programmes, that can alter the processing of the input data. The hardware units they target are called processors. They come in various flavours:

- vertex shader: modifies individual vertices,
- geometry shader: operates on whole primitives, can create new primitives,
- tessellation shader: similar to geometry shader, specific for tesselation,
- fragment shader: modifies individual pixel fragments,
- compute shader: arbitrary parallel computations.

Fragment vs Pixel

- A pixel represents the contents of the frame buffer at a specific location.
- A fragment is the state required to potentially update a particular pixel.
- A fragment has an associated pixel location, a depth value, and a set of interpolated parameters.

Outline

1. Organization
2. Introduction, history
   - Motivation
   - History
   - Pipeline
   - Shading Languages
   - Coordinate Spaces and Transforms
   - Homework

Brief History: 1980's

- integrated framebuffer,
- draw to display,
- tightly CPU controlled,
- addition of shaded solids, vertex lighting, rasterization of filled polygons, depth buffer,
- OpenGL in 1989, beginning of graphics pipeline.

Brief History: 1990's

Generation 0

- fixed graphics pipeline,
- half the pipeline on CPU, half on GPU,
- 1 pixel per cycle, easy to overload $\rightarrow$ multiple pipelines,
- dawn of "cheap" game hardware: 3DFX Voodoo, NVIDIA TNT, ATI Rage,
- developement driven by games: Quake, Doom, ...

Brief History: 1990's

Generation I

- no 2D graphics acceleration; only 3D,
- transform part of the pipeline on CPU,
- rendering part on GPU (texture mapping, z-buffering, rasterization),
- 3DFX Voodoo, 3DFX Voodoo2.

Brief History: 1990's

Generation II

- entire pipeline on GPU,
- term "GPU" introduced for GeForce 256,
- AGP instead of PCI bus,
- new features: multi-texturing, bump mapping, hardware T&L,
- fixed function pipeline.

Brief History: 2000–2002

Generation III

- programmable pipeline (NVIDIA GeForce 3, ATI Radeon 8500),
- parts of the pipeline can be change with custom programme,
- only vertex shaders,
- small assembly language “kernels”.

Brief History: 2002–2004

Generation IV

- "fully" programmable pipeline (NVIDIA GeForce FX, ATI Radeon 9700),
- vertex and fragment (pixel) shaders,
- dedicated vertex and fragment processors,
- floating point support, advanced texture processing $\rightarrow$ GPGPU.

Brief History: 2004–2006

Generation V

- faster than Moore's law growth,
- PCI-express bus (NVIDIA GeForce 6, ATI Radeon X800),
- multiple rendering targets, increased GPU memory,
- high level GPU languages with dynamic flow control (Brook, Sh).

Brief History: 2006–2009

Generation VI

- massively parallel processors,
- unified shaders (NVIDIA GeForce 8),
- streaming multiprocessor (SM),
- addition of geometry shaders,
- new general purpose languages: CUDA, OpenCL.

Unified Shaders

- before – different instruction set, capabilities,
- now they can do the same (almost – differences of pipeline position),
- gradient merging of instruction sets,
- HLSL perspective (`http://en.wikipedia.org/wiki/High-level_shader_language`),
- currently Shader model 5.0 (compute).

Brief History: 2009–?

Generation VII

- even more programmability,
- cache hierarchy, ECC, unified memory address space,
- focus on general computations,
- debuggers and profilers.

Brief Future :D

Generation Vxx

- slower rate of performance growth,
- focus on the energy efficiency (GFLOP/W),
- more CPU like,
- emphasis on better programming languages and tools,
- merge of graphics and general purpose APIs.

Outline

Graphics Pipeline OpenGL 4.2

- The graphics pipeline is a sequence of stages operating in parallel and in a fixed order.
- Each stage receives its input from the prior stage and sends its output to the subsequent stage.

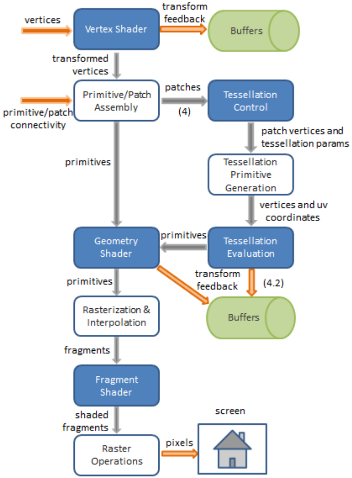

Figure: Taken from lighthouse3d.com

Graphics Pipeline

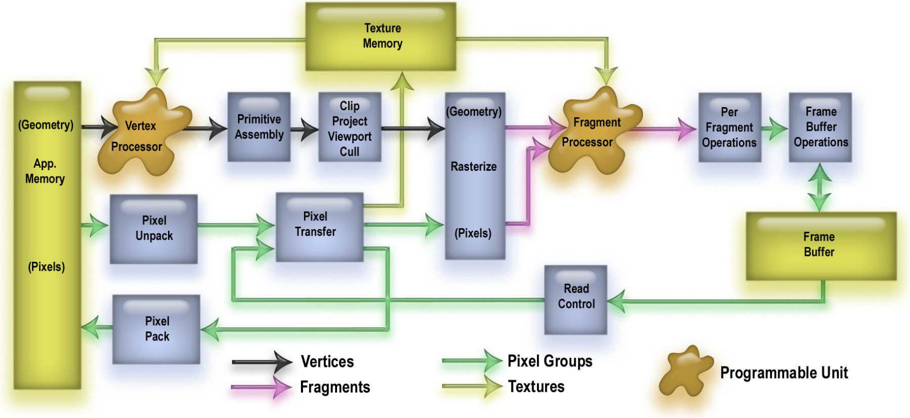

Figure: Taken from goanna.cs.rmit.edu.au

For more, detailed diagrams, see:
http://openglinsights.com/pipeline.html

Why Programmable Pipeline?

- Fixed pipeline is limited to algorithms hard-coded into the graphics chips $\rightarrow$ narrow class of effects.
- Programmability gives the developer almost limitless possibilities.
- We cannot combine fixed and programmable pipeline. Once shader is active it is responsible for the entire stage.

Shaders (cont.)

Typical tasks done in shaders:

- vertex shader: animation, deformation, lighting,
- geometry shader: mesh processing,
- tessellation shader: tessellation,
- fragment shader: shading ;-),
- compute shader: almost anything.

Outline

1. Organization
2. Introduction, history
   - Motivation
   - History
   - Pipeline
   - Shading Languages
   - Coordinate Spaces and Transforms
   - Homework

Shader Languages

- Cg (C for Graphics), by NVIDIA – no longer under active development,
- HLSL (High Level Shading Language), by Microsoft,
- GLSL (OpenGL Shading Language), by Khronos Group.

Shader Languages Comparison

- almost the same capabilities,
- conversion tools between them,
- Cg and HLSL very similar (different setup),
- HLSL DirectX only, GLSL OpenGL only, Cg for both $\rightarrow$ different platforms supported.

Shader Languages Comparison – Compilers

- HLSL needs DirectX, Cg needs Cg toolkit [DirectX], GLSL comes with driver,
- HLSL & Cg: toolkit compiler $\rightarrow$ “same” binary code for all vendors $\rightarrow$ translation to machine code,
- GLSL: vendor compiler $\rightarrow$ “faster” machine code, inconsistencies, harder to deal with varying hardware,
- Cg may have compiler issues on ATI cards.

Chosen Language

We will use GLSL in this course:

- open standard (same as OpenGL),
- no install needed,
- all platforms, all vendors.

Will will use GLSL 3.30 for OpenGL 3.3

- newer features will be mentioned but not demonstrated,
- e.g., NVIDIA 9600 GT (released 02/2008) is a OpenGL 2.1/3.3 card.

OpenGL Evolution

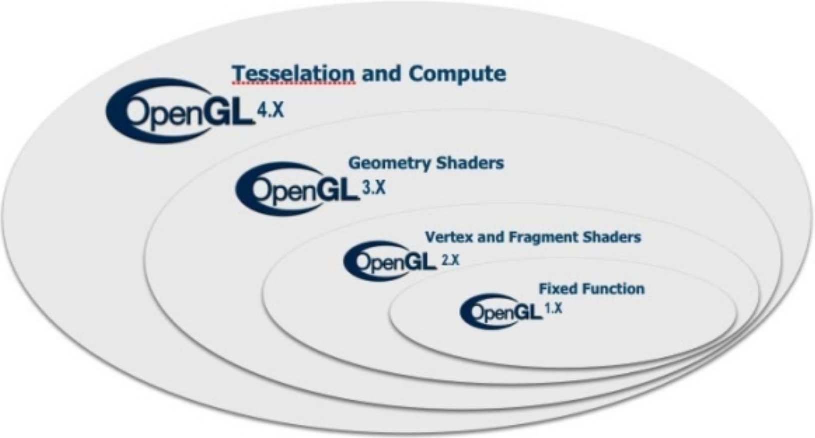

Figure: Taken from news.cnet.com

OpenGL Evolution

Released	OpenGL Version	GLSL Version
1992	1.0	—
2004	2.0	1.10
2006	2.1	1.20
2008	3.0	1.30
2009	3.1	1.40
2009	3.2	1.50
2010	3.3	3.30
2010	4.0	4.0
...	...	...
2014	4.5	4.5

Table: OpenGL and GLSL versions

For more, see, e. g., following: History of OpenGL

Outline

Coordinate Spaces and Transforms – Object Space

- the pipeline transforms 3D objects into 2D image,
- divided into several coordinate spaces beneficial for different tasks,
- transformation starts with polygon representation of the model,
- represented in **object space** (**local space**),
- origin and units chosen according to the model.

Coordinate Spaces and Transforms – World Space

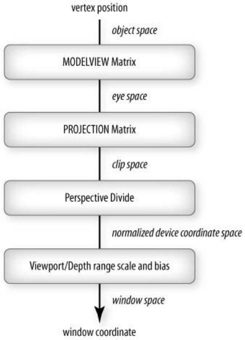

Figure: Taken from yaldex.com

- objects are composed in a single scene (share a single world),
- represented in **world space** (**model space**),
- origin and units chosen according to the scene,
- objects are transformed into this space by **modeling transformation** as defined by **model matrix**,
- spatial relations of objects are known afterwards.

Coordinate Spaces and Transforms – Eye Space

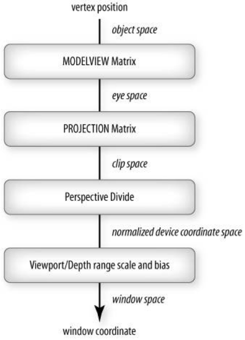

Figure: Taken from yaldex.com

- the scene is viewed by a camera,
- the view is represented in **eye space** (**camera space**),
- origin at the eye position, looking down the the negative Z axis,
- objects are transformed into this space by **viewing transformation** as defined by **view matrix**,
- spatial relations of objects are unchanged,
- model and view matrix are combined into **modelview matrix** $modelview = view \times model$.

Coordinate Spaces and Transforms – Clip Space

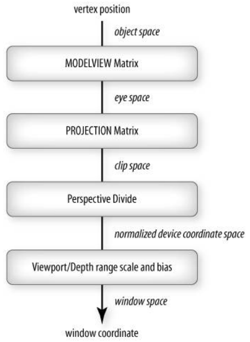

Figure: Taken from yaldex.com

- the camera defines a viewing volume, space visible in the final image,
- the view is represented as a axis-aligned cube in **clip space**,
- $-w \leq x \leq w, -w \leq y \leq w, -w \leq z \leq w$,
- objects are transformed into this space by **projection transformation** as defined by **projection matrix**,
- beneficial for **frustum clipping** polygons outside the axis-aligned cube.

Coordinate Spaces and Transforms – NDC Space

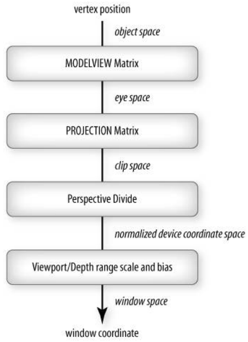

Figure: Taken from yaldex.com

- the clip space is compressed into [-1,1] range with the **perspective divide**,
- achieved by dividing with $w \rightarrow$ only 3 coordinates left,
- the resulting space is called **normalized device coordinate space**,
- beneficial for mapping visible primitives to arbitrarily sized viewports.

Coordinate Spaces and Transforms – Screen Space

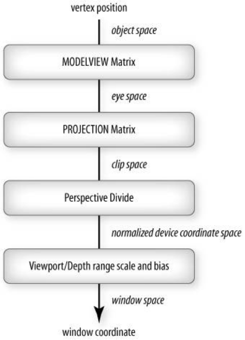

Figure: Taken from yaldex.com

- Pixels coordinates are of form 0 to (width-1), and from 0 to (height-1), i.e. **window coordinate system** (**screen space**).
- **Viewport transformation** transforms the [-1,1] range into this system.
- Primitives are rasterized in this system.

Coordinate Spaces and Transforms – Guidelines

- During computations the variables must be in the same space,
- E. g., vertices, normals and light positions in eye space,
- Vertex shader must output the **clip coordinates.**

Outline

Homework

- Recapitulate shader set-up process (shader & program creation; compilation, running, . . .),
  - from **PV112** 10th lecture,
  - from **PV227** setup materials.

- Try to compile and run examples in “homework” assignment:
  - most things should be already set – just open, compile and run (hopefully),
  - try to briefly look at different setups.

Sources

- interactive book about shaders: `http://pixelshaders.com/`
- “simple” shader sandbox: `http://glslsandbox.com/`
- advanced sandbox: `http://www.kickjs.org/example/shader_editor/shader_editor.html`
- shaders Guru’s (Ińigo Quílez) web: `http://www.iquilezles.org/default.html`