PV227 GPU programming

Marek Vinkler

Department of Computer Graphics and Design

Motivation

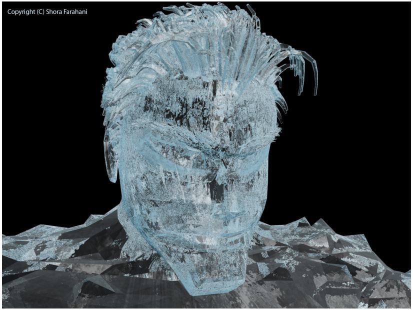

Figure: Taken from shoraspot.com

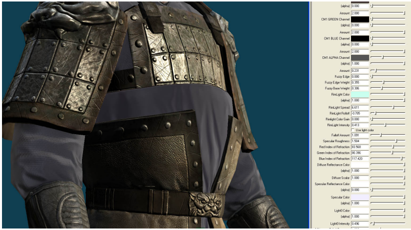

Figure: Taken from cgsociety.org

Course

- no more than 2 absences,
- final test (on the spot programming),
- first lectures more theoretical, then mostly practical.

Course

- new course $\rightarrow$ active participation,
- only major language features are introduced,
- graphics change fast $\rightarrow$ help me ;-)

Contact

- Office C420
- xvinkl@fi.muni.cz

Why GPU?

- graphics computations are costly,
- graphics are "embarrassingly parallel",
- increasing model complexity, screen resolution, ...
- GPU is parallel co-processor.

Why GPU?

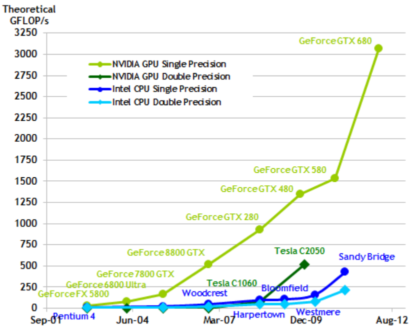

Figure: Taken from docs.nvidia.com

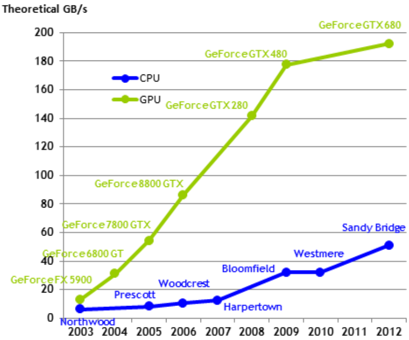

Figure: Taken from docs.nvidia.com

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.

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.

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,
- more CPU like,
- emphasis on better programming languages and tools,
- merge of graphics and general purpose APIs.

Graphics pipeline

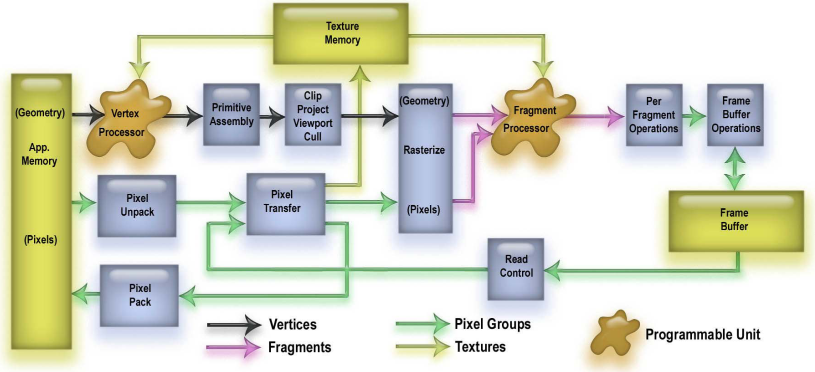

Figure: Taken from goanna.cs.rmit.edu.au

Graphics pipeline

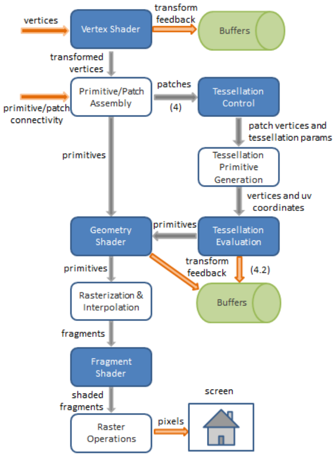

Figure: Taken from lighthouse3d.com

- 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.

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 continued

Typical tasks done in shaders:

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

Shader languages

- Cg (C for Graphics), NVIDIA,
- HLSL (High Level Shading Language), Microsoft,
- GLSL (OpenGL Shading Language), 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

- 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.

Shader languages comparison

We will use GLSL:

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

Will will use GLSL 3.30 for OpenGL 3.3 (NVIDIA 9600 GT is a OpenGL 2.1/3.3 card). Newer features will be mentioned but not demonstrated.

OpenGL evolution

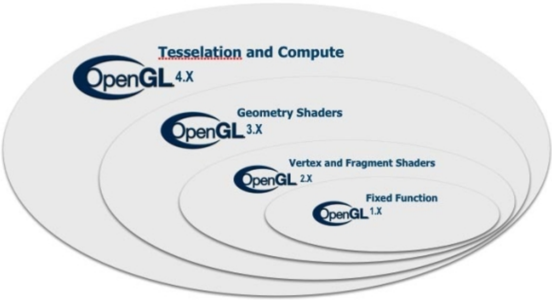

Figure: Taken from news.cnet.com

Hands-on shading

```
http://pixelshaders.com/
http://glsl.heroku.com/
http://www.kickjs.org/example/shader_editor/
shader_editor.html
http://www.iquilezles.org/default.html
http://www.iquilezles.org/live/index.htm
```

Coordinate spaces and transforms

- 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

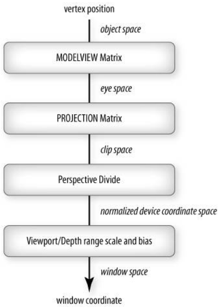

Figure: Taken from yaldex.com

- 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

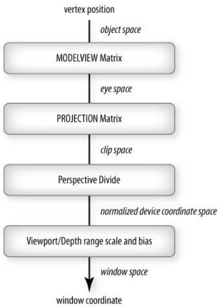

Figure: Taken from yaldex.com

- 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

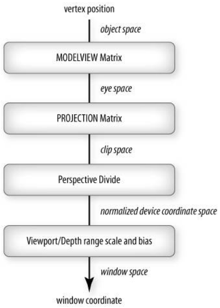

Figure: Taken from yaldex.com

- 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

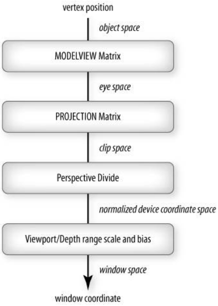

Figure: Taken from yaldex.com

- 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 arbitrarly sized viewports.

Coordinate spaces and transforms

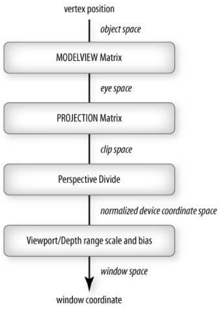

Figure: Taken from yaldex.com

- pixels coordinates are of form 0 – (width-1) and 0 – (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

- 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.**

GLSL shader setup

```
#include <GL/glew.h>
#include <GL/glut.h>

void main(int argc, char **argv)
{
  glutInit(&argc, argv);
  ...
  glewInit();

  if (glewIsSupported("GL_VERSION_3_3"))
  {
    printf("Ready for OpenGL 3.3\n");
  }
  else
  {
    printf("OpenGL 3.3 not supported\n");
    exit(1);
  }
  setShaders();
  initGL();

  glutMainLoop();
}
```

GLSL shader setup

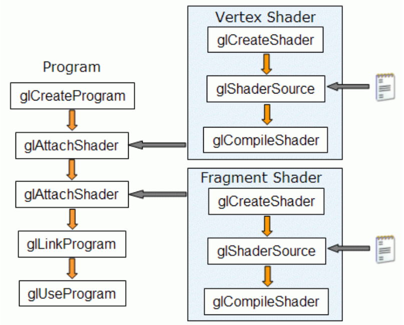

Figure: Taken from lighthouse3d.com

Creating shader

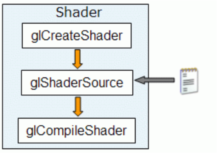

Figure: Taken from lighthouse3d.com

GLuint glCreateShader(GLenum shaderType);

shaderType – GL_{VERTEX|FRAGMENT|GEOMETRY|TESS_CONTROL|TESS_EVALUATION|COMPUTE}_SHADER.

- Creates shader object of a specified type that acts as a container.
- Returns the handle for that container.

Creating shader

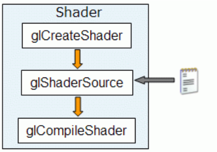

Figure: Taken from lighthouse3d.com

void glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length);

shader – the handler to the shader.

count – the number of strings in the arrays.

string – the array of strings.

length – an array with the length of each string; NULL, meaning that the strings are NULL terminated.

- Replaces a source code for the shader.
- Single string can be used instead of an array.
- Multiple strings can define common pieces of code, third-party library functions, ...

Creating shader

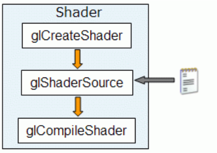

Figure: Taken from lighthouse3d.com

```
void glCompileShader(GLuint shader);
```

shader – the handler to the shader.

- Compiles the shader.
- Checks its validity.

Creating program

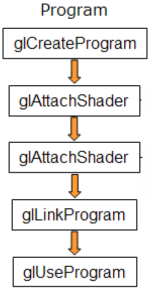

Figure: Taken from lighthouse3d.com

```
GLuint glCreateProgram(void);
```

- Creates program object that acts as a container.
- Returns the handle for that container.
- Any number of programs can be created and used in a single frame.
- Programes can be switched at runtime.
- No program used → fixed pipeline.

Creating program

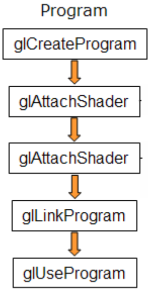

Figure: Taken from lighthouse3d.com

```
void glAttachShader(GLuint program, GLuint shader);
```

program – the handler to the program.

shader – the handler to the shader you want to attach.

- Attaches a shader into the program.
- The shaders need neither be compiled nor have source code.
- Any number of shaders can be attached, but only one main for each shader type.
- Single shader can be attached to many programes.

Creating program

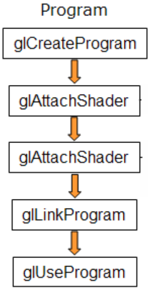

Figure: Taken from lighthouse3d.com

```
void glLinkProgram(GLuint program);
```

program – the handler to the program.

- Links the program, resolves cross-shader references.
- Shaders must be compiled at this point.
- Afterwards the shaders can be modified & recompiled.
- Uniform variables are assigned locations and set to 0.

Creating program

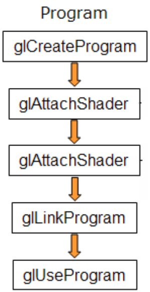

Figure: Taken from lighthouse3d.com

void glUseProgram(GLuint prog);

program – the handler to the program; zero to use fixed functionality .

- Sets the program for use in rendering.
- Relinking a used program also sets it for use.

Cleanup

```
void glDetachShader(GLuint program, GLuint shader);
```

program – the program to detach from.

shader – the shader to detach.

- Detaches shader from a program.

```
void glDeleteShader(GLuint id);
void glDeleteProgram(GLuint id);
```

id – the handler of the shader / program to erase.

- When attached shader/program is deleted, it is only "marked for deletion" and is fully deleted when no longer used.
- Shaders may be deleted as soon as they are attached, everything will be cleaned up when program is deleted.

GLSL setup example

```
void setShaders()
{
  char *vs, *fs;

  // Setup
  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);

  vs = textFileRead("simple.vert");
  fs = textFileRead("simple.frag");

  const char * vv = vs;
  const char * ff = fs;

  glShaderSource(v, 1, &vv, NULL);
  glShaderSource(f, 1, &ff, NULL);

  free(vs);
  free(fs);

  glCompileShader(v);
  glCompileShader(f);
```

GLSL setup example (cont.)

```

    p = glCreateProgram();

    glAttachShader(p, v);
    glAttachShader(p, f);

    glLinkProgram(p);
    glUseProgram(p);

    ...

    // Clean up
    glDetachShader(p, v);
    glDetachShader(p, f);

    glDeleteShader(v);
    glDeleteShader(f);

    glUseProgram(0);
    glDeleteProgram(p);
}
```

State query

```
void glGetShaderiv(GLuint shader, GLenum pname, GLint *params);
```

shader – the shader to query.

pname – parameter to query.

params – queried state.

pname:

- **GL_SHADER_TYPE** – type of the shader,
- **GL_DELETE_STATUS** – marked for deletion?,
- **GL_COMPILE_STATUS** – last compile successful?,
- **GL_INFO_LOG_LENGTH** – length of the information log,
- **GL_SHADER_SOURCE_LENGTH** – length of the concatenated shader.

State query

```
void glGetProgramiv(GLuint program, GLenum pname, GLint *params);
```

program – the shader to query.

pname – parameter to query.

params – queried state.

pname (not all shown):

- **GL_LINK_STATUS** – last link successful?,
- **GL_DELETE_STATUS** – marked for deletion?,
- **GL_VALIDATE_STATUS** – last validation successful?,
- **GL_INFO_LOG_LENGTH** – length of the information log,
- information on number of shaders attached, number of attribute values and uniform variables.

State query

void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei ∗length, GLchar ∗infoLog);

shader – the shader to query.

maxLength – maximal length of output buffer.

length – actual length of the log.

infoLog – the shader log.

- updated during shader compile,
- may contain diagnostic messages, errors, warnings etc. (implementation specific).

State query

void glGetProgramInfoLog(GLuint program, GLsizei maxLength, GLsizei ∗length, GLchar ∗infoLog);

program – the program to query.

maxLength – maximal length of output buffer.

length – actual length of the log.

infoLog – the shader log.

- updated during program validation or link,
- may contain diagnostic messages, errors, warnings etc. (implementation specific).

State query

```
void glValidateProgram(GLuint program);
```

program – the program to validate.

- checks whether program can execute given current OpenGL state,
- updates the program log,
- only for developement (slow).

GLSL query example

```
void printShaderInfoLog(GLuint obj)
{
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;

    glGetShaderiv(obj, GL_INFO_LOG_LENGTH, &infologLength);

    if(infologLength > 0)
    {
        infoLog = (char *)malloc(infologLength);
        glGetShaderInfoLog(obj, infologLength, &charsWritten,
            infoLog);
        printf("%s\n",infoLog);
        free(infoLog);
    }
}
```

GLSL query example

```
void printProgramInfoLog(GLuint obj)
{
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;

    glGetProgramiv(obj, GL_INFO_LOG_LENGTH, &infologLength);

    if(infologLength > 0)
    {
        infoLog = (char *)malloc(infologLength);
        glGetProgramInfoLog(obj, infologLength, &charsWritten,
            infoLog);
        printf("%s\n",infoLog);
        free(infoLog);
    }
}
```