PV227 GPU programming

Marek Vinkler

Department of Computer Graphics and Design

GLSL

- officialy OpenGL Shading Language,
- part of OpenGL standard (from OpenGL 2.0),
- high-level procedural language (based on C and C++),
- independent on hardware,
- performance oriented (through custom compilers).

GLSL

- single set of instructions for all shaders (almost),
- native support for vectors and matrices,
- no pointers (hurray :D) and strings,
- strict with types,
- no length limit (language part).

GLSL

- part of the OpenGL driver – graphics driver,
- common front-end (should be), different (optimized) back-ends,
- **shaders** are combined into **programs**,
- linking resolves cross shader references,
- shaders are strings, not files (no **#include**).

Scalar data types

- **float**,
- **int**,
- **uint**,
- **bool**,
- declarations may appear anywhere.

Scalar data types

```
float f;
float h = 2.4; // float constant in GLSL 3.3 and below
f = 0.2f;
float ff = 1.5e10;
ff -= 1.E-3;

uint n = 5;
n = 15u;
int a = 0xA;
a += 071;

bool skip = true;
skip = skip && false;
```

Vector data types

- **vec2**, **vec3**, **vec4** – **float**,
- **ivec2**, **ivec3**, **ivec4** – **int**,
- **uvec2**, **uvec3**, **uvec4** – **uint**,
- **bvec2**, **bvec3**, **bvec4** – **bool**,
- two, three or four component vectors of scalar types.

Vector data types

- field selection or array access,
- **x**, **y**, **z**, **w** – for positions or directions,
- **r**, **g**, **b**, **a** – for colors,
- **s**, **t**, **p**, **q** – for texture coordinates,
- only for readability, all select certain vector coordinate (e.g. $v.x \equiv v.r \equiv v.s \equiv v[0]$).

Matrix data types

- only matrices of **floats**
- **mat2**, **mat3**, **mat4** – 2×2, 3×3, 4×4 matrices,
- **matmxn** – $m \times n$ (column $\times$ row) matix,
- column major order (first coordinate is column),
- as in OpenGL, unlike C/C++.

```
mat4 m;
vec4 v = m[3]; // Fourth column
float f = m[3][1]; // Second component (row) of the fourth
    column vector
```

Sampler data types

- for texture access,
- variants for **floats**, **ints**, **unsigned ints** (no **bool**),
- **[i|u]sampler{1|2|3}D** – access one, two or three dimensional texture,
- **[i|u]samplerCube** – access cube-map texture,
- **[i|u]sampler2DRect** – access two-dimensional rectangle texture,
- **[i|u]sampler{1|2}DArray** – access one or two dimensional texture array,
- **[i|u]samplerBuffer** – access texture buffer,

Sampler data types

- **sampler{1|2|3}DShadow** – access one, two or three dimensional depth texture with comparison,
- **sampler{1|2|3}DShadow** – access one, two or three dimensional depth texture with comparison,
- **sampler2DRectShadow** – access two-dimensional rectangle depth texture with comparison,
- **sampler{1|2}DArrayShadow** – access one or two dimensional depth texture array with comparison.

Sampler data types

- application initializes the samplers,
- passed to shaders through **uniform** variables,
- samplers cannot be manipulated in shader,
- shadow textures and color samplers must not be mixed $\rightarrow$ undefined behaviour.

```
uniform sampler2D sampler;
vec2 coord = vec2(0.f, 1.f);
vec4 color = texture(sampler, coord);
```

Structures

- C++ style (name of structure $\rightarrow$ name of type),
- can be embedded and nested, contain arrays,
- bit-fields not allowed, no **union**, **enum**, **class.**

```
struct vertex
{
  vec3 pos;
  vec3 color;
};
vertex v;
```

Arrays

- available for any type,
- zero indexed,
- no pointers $\rightarrow$ always declared with [] and size,
- the array must be declared with same size in all shaders.

Declarations and scopes

- variable name format same as in C/C++ (case sensitive),
- names begining with “gl_” or “__” are reserved,
- scoping similar to C++.

```
float f; // Declared from this point until the end of the block
for(int i = 0; i < 3; ++i) // i is declared only in this cycle
  f *= f;

if(i == 1) // Invalid
{
  ...
}
```

Initializers and constructors

- scalar variables may be initialized in declaration,
- constants must be initialized,
- **in** and **out** variables may not be initialized,
- **uniform** variables may be initialized.

```
int a = 0, b, c = 1;
const float eps = 1e-3f;
uniform float temp = 36.5f;
```

Initializers and constructors

- aggregate types are initialized/set with constructors,
- the number of components in vectors need not match.

```
vec2 v = vec2(0.f, 1.f);
v = vec2(1.f, 0.f);
vec3 v3 = vec3(v, 0.f);

v3 = vec3(1.f); // vec3(1.f, 1.f, 1.f);
v = vec2(v3); // vec2(1.f, 1.f);

float array[4] = float[4](0.f, 1.f, 2.f, 3.f);

struct person
{
  struct attrib
  {
    vec3 color;
    bool active;
  };
  vec3 pos;
} person1 = person(attrib(vec3(0.5f, 0.5f, 0.5f), true), v3);
```

Matrix constructors

- matrix components are read and written in column major order,
- matrices cannot be constructed from matrices.

```
mat2 matrix = mat2(1.f, 2.f, 3.f, 4.f); // 1.f, 3.f
                                        // 2.f, 4.f);
mat2 identity = mat2(1.f);  // Initializes diagonal
                            // mat2(1.f, 0.f, 0.f, 1.f);

vec2 v = vec2(1.0f);
mat2 identity2 = mat2(v);
```

Type matching and promotion

- strict matching (prevents ambiguity),
- assigned types, functions parameters must match exactly,
- scalar integers may be implicitely converted to scalar floats,
- may force the programmer to use explicit conversion.

Type conversions

- performed with constructors,
- no C-style typecast,
- no way to reinterpret a value,
- conversions to boolean $\rightarrow$ non-zero as **true**, zero as **false**,
- conversions from boolean $\rightarrow$ **true** as 1 (1.f), **false** as 0 (0.f).

GLSL qualifiers

- tell compiler where the value comes from,
- **in** – vertex attribute (vertex shader) vertex data (geometry shader) or interpolated value (fragment shader),
- **uniform** – constant variable in all shaders,
- **out** – varying variable passed from one shader to another, output to frame buffer,
- **const** – compile time constant variables,
- **in**, **uniform**, **out** are always global variables,
- qualifier are specified before variable type.

Uniform qualifier

- cannot be modifed from shader,
- less frequently updated, max once per primitive,
- all data types supported,
- used for samplers,
- all shaders inside a program share uniform variables.

In qualifier (vertex shader)

- vertex attributes,
- can be changed as often as a single vertex,
- not all data types supported:
  - boolean scalars and vectors,
  - structures,
  - arrays.

Out qualifier (vertex shader/geometry shader)

- output to the geometry shader / rasterizer,
- interpolation qualifiers for computing fragments:
  - **smooth out** – perspective-correct interpolation,
  - **noperspective out** – interpolation without perspective correction,
  - **flat out** – no interpolation.
- floating point scalars, vectors, matrices and arrays,
- with **flat out**: [unsigned] integer scalars, vectors, arrays,
- no structures.

In qualifier (fragment shader)

- interpolated values from the rasterizer,
- must match the definition of **out** variables in vertex / geometry shader,
    - interpolation qualifier, type, size, name.

Out qualifier (fragment shader)

- passed to per fragment fixed-function stage,
- floating point/integer/unsigned integer scalars, vectors and arrays,
- no matrices or structures,
- can be preceeded with `layout(location = x)`, where `x` is the number of the render target.

Constant qualifiers

- compile time constant,
- not visible outside the shader,
- individual structure items may not be constants,
- initializers may contain only literal values or other **const** variables.

No qualifiers

- can be both read and written,
- unqualified global variables,
  - shared between shader of the same type, not between shaders of different types,
  - not visible outside program,
  - lifetime limited to a single run of a shader (no “**static**”),
  - different variables for different processors $\rightarrow$ do NOT use.

Interface blocks

- common names for several variables,
- different meaning for each qualifier, same syntax,
- used for passing data between shaders, loading uniform variables.

```
storage_qualifier block_name
{
  <members defition>
} [instance_name];
```

- block_name used from OpenGL,
- instance_name optional to create named instances inside GLSL,
- possible arrays of instances.

Inter shader communication

Name based matching:

```
// vertex shader
out vec4 color;

________________________
// geometry shader
in vec4 color[];
out vec4 colorFromGeom;

__________________________
// fragment shader
in vec4 colorFromGeom;
```

- names in shaders must match,
- **in** and **out** cannot be named the same,
- cannot use the same shader for vertex $\rightarrow$ fragment and vertex $\rightarrow$ geometry $\rightarrow$ fragment.

Inter shader communication

Location based matching:

```
// vertex shader
layout (location = 0) out vec3 normalOut;
layout (location = 1) out vec4 colorOut;

______________________________
// geometry shader
layout (location = 0) in vec3 normalIn[];
layout (location = 1) in vec4 colorIn[];

layout (location = 0) out vec3 normalOut;
layout (location = 1) out vec4 colorOut;

______________________________
// fragment shader
layout (location = 0) in vec3 normalIn;
layout (location = 1) in vec4 colorIn;
```

- locations in shaders must match,
- location is per max vec4 item (not aggregate types),
- difficulty with assigning location numbers.

Inter shader communication

Interface based matching:

```
// vertex shader
out Data {
    vec3 normal;
    vec3 eye;
    vec2 texCoord;
} DataOut;
---------------------
// geometry shader
in Data {
    vec3 normal;
    vec3 eye;
    vec2 texCoord;
} DataIn[];

out Data {
    vec3 normal;
    vec3 eye;
    vec2 texCoord;
} DataOut;
----------------------------
```

Inter shader communication (cont.)

```
// fragment shader
in Data {
    vec3 normal;
    vec3 eye;
    vec2 texCoord;
} DataIn;
...
DataOut.normal = normalize(someVector);
```

- block names in shaders must match,
- data manipulation through instance name,
- same members in blocks.

Uniform interface blocks

- sharing uniforms between programs,
- setting multiple uniforms at once,
- named blocks of uniform variables (individual items are globally scoped),
- backed by buffers for data transfer,
- for setting transform matrices, common variables in shader families etc.

Uniform interface blocks

```
layout (xxx) uniform ColorBlock {
    vec4 diffuse;
    vec4 ambient;
};
...
out vec4 outputF;

void main() {
    outputF = diffuse + ambient;
}
```

- layout specifies storage (default is implementation dependent),
- **std140** – OpenGL specified layout, blocks can be shared between shaders,
- **shared** – implementation dependent layout, blocks can be shared between shaders,
- **packed** – unused variables are optimized-out, not shareable.

Uniform interface blocks

- uniform blocks are connected with buffers through binding points,
- block indices are assigned during program link,
- multiple blocks can be bound to the same binding point.

```
GLuint bindingPoint = 1, buffer, blockIndex;
float myFloats[8] = {1.0, 0.0, 0.0, 1.0,  0.4, 0.0, 0.0, 1.0};

// Assign the uniform block to the binding point
blockIndex = glGetUniformBlockIndex(p, "ColorBlock");
glUniformBlockBinding(p, blockIndex, bindingPoint);

glGenBuffers(1, &buffer);
glBindBuffer(GL_UNIFORM_BUFFER, buffer);

// Assign the buffer to the bindong point
glBufferData(GL_UNIFORM_BUFFER, sizeof(myFloats), myFloats,
    GL_DYNAMIC_DRAW);
glBindBufferBase(GL_UNIFORM_BUFFER, bindingPoint, buffer);
```

Uniform interface blocks

- individual uniforms may be aligned in memory,
- to set them correctly we need to compute their offset,
- queried with **glGetActiveUniformBlockiv** and **glGetActiveUniformsiv**,
- set with **glBufferSubData**.

```
layout (std140) uniform ColorBlock2 {
    vec3 diffuse;
    vec3 ambient;
};

GLuint bindingPoint = 1, buffer, blockIndex;
float myFloats[3] = {0.4, 0.0, 0.0};

glGenBuffers(1, &buffer);
glBindBuffer(GL_UNIFORM_BUFFER, buffer);

glBufferSubData(GL_UNIFORM_BUFFER, 4*sizeof(float), sizeof(
    myFloats), myFloats); // Notice the offset
```

Program flow

- similar to C++,
- **main** is the entry point for a shader,
- global variable are initialized before **main** is executed,
- looping
  - **for**, **while**, **do-while**, **break**, **continue**,
- selection
  - **if**, **if-else**, **if-else if-else**, **?:** and **switch**,
- expressions must be booleans,
- partial evaluation of **&&** and ||, **?:**,
- no **goto**,
- **discard** prevents fragment from updating frame buffer.

Functions

- support for C++ overload by parameter type,
- prototype declaration or definition before call to the function,
- exact matching of parameters, return values (**return**),
- no recursion,
- programs entry point is function `void main()`.

Calling conventions

- value-return,
- all input parameter values are copied to function before execution,
- all output parameter values are copied from the function after execution,
- parameter behaviour controlled by qualifiers **in** (default), **out** and **inout**,
- **in** parameters can be also **const** (not writeable inside function).

Functions continued

- arrays and structures are also copied by value,
- any return type (including structures).

```
void foo1(in vec3 normal, float eps, inout vec3 coord);
vec3 foo2(in vec3 normal, float eps, in vec3 coord);
void foo3(in vec3 normal, float eps, in vec3 coord, out vec3
    coordOut);

// Get coord
vec3 coord;
foo1(normal, eps, coord);
coord = foo2(normal, eps, coord);
foo3(normal, eps, coord, coord);
```

Swizzling

- used to select (rearrange) components of a vector,
- must use component names from the same set,
- must still be a valid type (no more than 4 components),
- R-values
  - any combination and repetition of components,
- L-values
  - no repetition of components.

```
vec4 pos = vec4(1.f, 2.f, 3.f, 4.f);
vec2 v1 = pos.xy;
vec3 v2 = pos.abb;
vec4 v3 = pos.xyrs; // Illegal different sets
vec4 o;
o.xw = v2; // (1.f, 2.f, 3.f, 4.f)
o.xx = vec2(0.f); // Illegal repetition
```

Operations on vectors and matrices

- mostly component-wise (independently for each component),
- vector sizes must match,
- vector $*$ matrix and matrix $*$ matrix are not component-wise,
- logical operations (**!**, **&&**, ||, **^^**) only on scalar boolean,
- **not** conmonent-wise logical not on boolean vectors.

Operations on vectors and matrices

- relational operators (**<**, **>**, **<=**, **>=**) on scalar floats and integers → scalar boolean,
- build-in functions like **lessThanEqual** do component-wise relational operations on vectors,
- **==** and **!=** operate on all types except arrays → scalar boolean,
- for component-wise comparision **equal** and **nonEqual** → boolean vector,
- **any** and **all** turn boolean vector into boolean scalar,
- **=** and its variants (**+=**, **-=**, ***=**, **/=**) operates on all types except structures and arrays.

Preprocessor

- basically the same as in C,
- macros begining with “GL_” or “__” are reserved,
- shaders should declare the GLSL version they are written for (**#version** number) as the first line of the code,
- usefull pragmas **optimize(on/off)** and **debug(on/off)**,
- language extensions can be accessed using **#extension.**

Build-in functions

- make shader programming easier,
- expose hardware functionality not writeable in the shader,
- provide optimized (possibly hardware accelerated) implementations of common functions,
- usually both scalar and vector variants,
- can be overriden by redeclaration,
- may be specific for a single shader type.

Shader specific functions

Geometry shader:

- void EmitVertex(void);,
  - use the current output state for a new vertex,
- void EndPrimitive(void);,
  - complete the current output primitive.

Keep up to date

- http://www.opengl.org/sdk/docs/man/
- http://www.opengl.org/sdk/docs/manglsl/
- http://www.opengl.org/registry/