PV227 GPU programming

Marek Vinkler

Department of Computer Graphics and Design

Tools

- shaders are just strings → any editor you desire,
- RenderMonkey (http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/rendermonkey-toolsuite/),
- FX Composer (https://developer.nvidia.com/fx-composer),
- OpenGL Shader Designer (http://www.opengl.org/sdk/tools/ShaderDesigner/),
- and many more, mostly discontinued,
- shader programming got diverse, only IDEs for specialized tasks.

Tools

- NVIDIA NSight,
  - for Registered developers,
- AMD CodeXL,
  - directly downloadable,
- gDEBugger (`http://www.gremedy.com/`),
  - directly downloadable, up to OpenGL 3.2
- VS2010,
  - use what is already there,
  - syntax highlighting, IntelliSense.

Project setup

- create folder *H:\PV227* (not Desktop, Documents, . . .),
- crate subfolders *Templates* and *Final*,
- unzip the libraries into the *Templates* folder (optionally also to the *Final*),
- unzip the source codes into these folders,
- launch the projects with **Ctrl-F5** (keeps the console open).

GLUT

- multiplatform windowing system for OpenGL,
- not updated, alternatives exist: FreeGLUT (`http://freeglut.sourceforge.net/`),
- download built libraries at `http://www.transmissionzero.co.uk/software/freeglut-devel/`.

GLEW

- library for accessing OpenGL core and extension functionality,
- download built libraries at `http://glew.sourceforge.net/`.

VS2010 setup

Project properties $\rightarrow$ Set All Configurations:

- VC++ Directories,
  - Include Directories: <path>\freeglut\include;<path>\glew-1.10.0\include;
  - Library Directories: <path>\freeglut\lib;<path>\glew-1.10.0\lib\Release\Win32;
- Debugging,
  - Environment: PATH=<path>\freeglut\bin;<path>\glew-1.10.0\bin\Release\Win32;

VS2010 setup

Syntax highlighting:

- Tools → Options,
- Text Editor → File Extension,
- add vert, geom, frag with Microsoft Visual C++ syntax,
- update usertype.dat in the VS2010 directory C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE.

Workflow

- same operation exactly once for every vertex/patch/primitive/fragment,
- independent states, no communication,
- program is for the entire pipeline,
- data can be passed between shaders.

Workflow

Which shader to use for a given task?

- depends on the modified data,
- per vertex $\rightarrow$ vertex shader,
- per patch $\rightarrow$ tessellation shader,
- per primitive $\rightarrow$ geometry shader,
- per fragment $\rightarrow$ fragment shader,
- no idea $\rightarrow$ compute shader.

Workflow

Which shader to use for a given task?

- may depend on special properties of the processors:
  - cancel computation $\rightarrow$ fragment or geometry shader,
  - some build-in functions are defined only for certain processors.

Workflow

- Shaders replace entire fixed pipeline.
- If we want to modify the vertex transformation behaviour, we also have to write code for lighting, texture generation, ...
- This may be tedious when small changes are desired.
- In bigger projects you usually rewrite it anyway ;-).

Vertex processor

Replaces the following fixed functionality:

- Vertex transformation by modelview and projection matix.
- Texture coordinates transformation by texture matrices.
- Transformation of normals to eye coordinates.
- Rescaling and normalization of normals.
- Texture coordinate generation.
- Per vertex lighting computations.
- Color material computations.
- Point size distance attenuation.

Vertex processor

Fixed functionality applied to the result:

- Perspective division on clip coordinates.
- Viewport mapping.
- Depth range scaling.
- View frustum clipping.
- Front face determination.
- Culling.
- Flat-shading.
- Associated data clipping.
- Final color processing.

Vertex processor

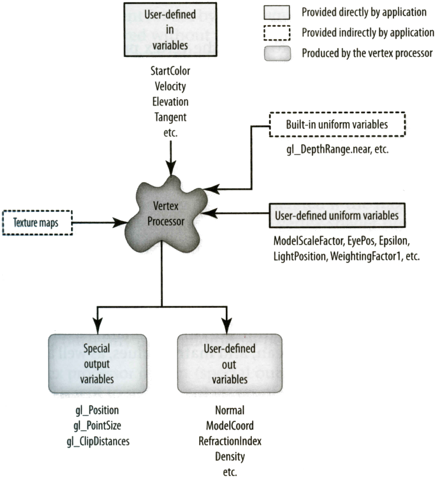

Figure: Scan from OpenGL Shading Language 3rd edition

Input data

- vertex attributes (user-defined),
- uniforms (built-in, user-defined),
- textures,
- special built-in variables.

Attributes

- user-defined per vertex data,
- consist of a number of indexed locations called *current vertex state*,
- limited number of attributes,
- attributes are set with **glVertexAttrib** family of functions,
- one indexed location can hold a quadruple,
- matrix attributes are stored in column-major order in succesive attribute locations,
- the same value can be set for all vertices (that do not have it otherwise specified).

Attributes

```
void glBindAttribLocation(GLuint program, GLuint index, const GLchar *name);
        program – the handler to the program.
        index – index of the generic vertex attribute to be bound.
        name – string containing the name of the vertex shader attribute variable to
which index is to be bound.
```

- Used before linking to set the attribute name-index pairing.
- Automatic assignment of index+1, [index+2, [index+3]] for matrix name.
- Reserved variables (gl_*) must not be bound this way.
- May set the pairing of attributes from the same array for different shaders consistently.

Attributes

```
GLint glGetAttribLocation(GLuint program, const GLchar *name);
```

program – the handler to the program.

name – string containing the name of the vertex shader attribute variable to be queried.

- Used after linking to get the attribute name-index pairing.
- For matrix `name` the returned index is for the first column (index+1, [index+2, [index+3]]).
- For non-existent attributes or reserved variables (gl_*) −1 is returned.

Attributes

```
void glEnableVertexAttribArray(GLuint index);
void glDisableVertexAttribArray(GLuint index);
```

index – index of the generic vertex attribute to be enabled/disabled.

- Enabled/disable vertex attributes for use in the draw calls.
- By default all generic attributes are disabled.

Attributes

```
void glVertexAttribPointer (GLuint index, GLint size, GLenum type, GLboolean
normalized, GLsizei stride, const GLvoid *pointer);
void glVertexAttribIPointer (GLuint index, GLint size, GLenum type, GLsizei stride,
const GLvoid * pointer);
    index – index of the generic vertex attribute to be modified.
    size – the number of components of the generic attribute (1|2|3|4) .
    type – the type of each component.
    normalized – whether fixed–point data should be normalized.
    stride – byte offset between consecutive vertex attributes.
    pointer – offset of the first attribute in the buffer bound to
GL_ARRAY_BUFFER target.
```

- Specifies the location and format of vertex attributes.
- The I variant passes integer attributes unchanged.

Vertex arrays and buffers

- All attributes are bound to a single **vertex array object (VAO)**.
- This VAO consists of a number of **buffers** holding the individual attributes.
- The VAO holds all the information for the draw call e.g. **glDrawArrays** or **glDrawElements**.

Vertex arrays and buffers

```
GLuint vao;

// Create the VAO
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Create buffers for our vertex data
GLuint buffers[2];
glGenBuffers(2, buffers);

// Vertex coordinates buffer
glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);
glEnableVertexAttribArray(VERTEX_COORD_ATTRIB);
glVertexAttribPointer(VERTEX_COORD_ATTRIB, 4, GL_FLOAT, 0,0,0);

// Index buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(faceIndex),
    faceIndex, GL_STATIC_DRAW);

```

Vertex arrays and buffers (cont.)

```
// Unbind the VAO
glBindVertexArray(0);

...

// Render VAO
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, faceCount*3, GL_UNSIGNED_INT, 0);
```

Uniforms

- user-defined: read-only in all shaders,
- constant per draw call, changed per primitive at most (not recommended for performance),
- can be initialized inside the shader,
- location indices are assigned during link,
- limited number of uniforms (both build-in and user-defined),
- uniforms can be grouped into named blocks.

Uniforms

- all variables outside named block are in default block,
  - sampler variables must be in default block,
  - cannot be used for another program,
  - advantageous for variables tied to an individual shader/program.

Uniforms

GLint glGetUniformLocation(GLuint program, const GLchar *name);

program – the handler to the program.

name – string containing the name of uniform variable to be queried.

- Returns the memory location of a uniform variable.
- Must be called after linking the program (location may change with each link).
- Not usable for structures, arrays, subcomponents of vectors and matrices.
- For structures and arrays, its elements can be set with "." and "[]".
- For non-existent uniforms or reserved names (gl_*) -1 is returned.

Uniforms

- during link uniforms are set to 0,
- their value can be modified only when their program is used,
- the values are preserved when the program is switched off and on,
- uniforms are set with **glUniform** family of functions.

Uniforms

```
void glUniform{1|2|3|4}{f|i|ui}(GLint location, TYPE v);
    location - the location of the uniform variable.
    v - 1|2|3|4 component value of the uniform.

void glUniform{1|2|3|4}{f|i|ui}v(GLint location, GLsizei count, const TYPE* v);
    location - the location of the uniform variable.
    count - number of array elements to be specified.
    v - array of values to be loaded.

void glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv(GLint location, GLsizei count,
GLboolean transpose, const GLfloat* v);
    location - the location of the uniform variable.
    count - number of matrices to be specified.
    transpose - load from row major order?
    v - array of values to be loaded.
```

Uniforms

- types and sizes of the uniform variables must match the functions,
- locations for array elements and other variables cannot be computed: `loc("A[n]") != loc("A")+n`.

Uniforms

```
uniform struct
{
  struct
  {
    float a;
    float b[10];
  } c[2];
  vec2 d;
} e;
```

```
loc1 = glGetUniformLocation(prog, "e.d"); // valid: vec2
loc2 = glGetUniformLocation(prog, "e.c[0]"); // invalid: struct
loc3 = glGetUniformLocation(prog, "e.c[0].b"); // valid: array
loc4 = glGetUniformLocation(prog, "e.c[0].b[2]"); // valid:
    array element

glUniform2f(loc1, 1.0f, 2.0f); // valid: vec2
glUniform2i(loc1, 1, 2);       // invalid: not ivec2
glUniform2f(loc1, 1.0f);       // invalid: not float
glUniform2fv(loc3, 10, &f);    // valid: b[0] (+10)
glUniform2fv(loc4, 10, &f);    // invalid: out of range
glUniform2fv(loc4, 8, &f);     // valid: b[2] (+8)
```

Samplers

- only **glUniform1i** and **glUniform1iv** can be used to load samplers,
- the loaded value is the index of the texture unit to be used,
- the same unit cannot be loaded into samplers of different types.

Special built-in variables

- **gl_VertexID** – implicit vertex index passed by e.g. **DrawArrays**,
- **gl_InstanceID** – implicit primitive index passed by instanced draw calls e.g. **glDrawArraysInstanced**,

Output data

- special built-in variables,
- varying variables (user-defined),

Special built-in variables

- in vec4 gl_Position;
  - homogeneous position in clip space (modelview, projection),
  - must be set, used by the rest of the pipeline,
- in float gl_PointSize;
  - size of the rasterized points,
  - must be set if points are rasterized,
- in float gl_ClipDistance[];
  - array of distances to user clipping planes,
  - must be set if user clipping is enabled.

Varying variables

- passed from vertex processor to rasterizer,
- anything can be passed,
- more variables can be outputed than used by follow-up shader,
- interpolation type can be set,
- limited number of interpolated values.

Vertex processor example

- Project triangle!
- Rotate and project triangle!

Geometry processor

- Optional (no fixed pipeline equivalent).
- Receives assembled primitives, outputs zero (culling) or more primitives.
- May receive adjacency information.
- The type of input and output primitives need not match (triangles $\rightarrow$ points).
- Designed for moderate geometry amplification, not tessellation.

Geometry processor

Input primitives:

- **points**,
- **lines**,
- **lines_adjacency**,
- **triangles**,
- **triangles_adjacency**.

Output primitives:

- **points**,
- **line_strip**,
- **triangles_strip**.

Input data

- interpolated varying variables (built-in, user-defined),
- uniforms (built-in, user-defined),
- textures,
- special built-in variables.

Varying variables

- build-in and user-defined varying variables for each vertex,
  - in the form of array of structures (user-defined or gl_PerVertex),
  - definition must match vertex shader.

Uniforms

- defined the same way as for vertex shader,
- can be the same set of variables as in vertex shader,
- no need to setup uniforms for each shader,
- limited number of uniforms (both build-in and user-defined).

Output data

- same output as the vertex shader,
- definition of primitives,
- special built-in variables,
- varying variables (user-defined).

Fragment processor

Replaces the following fixed functionality:

- Texture environments and texture functions.
- Texture application.
- Color sum.
- Fog.

Fragment processor

Fragment shader does not change the following operations:

- Texture image specification.
- Alternate texture image specification.
- Compressed texture image specification.
- Texture parameters that behave as specified even when a texture is accessed from within a fragment shader.
- Texture state and proxy state.
- Texture object specification.
- Texture comparison modes.

Fragment processor

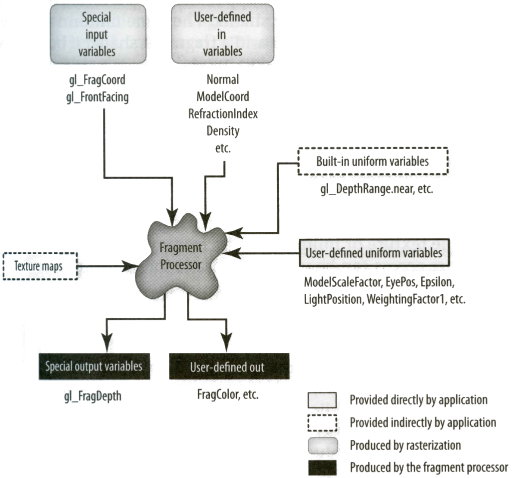

Figure: Scan from OpenGL Shading Language 3rd edition

Input data

- interpolated varying variables (built-in, user-defined),
- uniforms (built-in, user-defined),
- textures,
- special built-in variables.

Varying variables

- in vec4 gl_FragCoord;
  - window coordinate position (xy), fragment depth (z),
- in bool gl_FrontFacing;
  - whether the fragment originated from front facing primitive,
- in vec2 gl_PointCoord;
  - position of the fragment (only for point primitives),
- user defined varying variables,
  - definition must match vertex/geometry shader.

Uniforms

- defined the same way as for vertex/geometry shader,
- can be the same set of variables as in vertex/geometry shader,
- no need to setup uniforms for each shader,
- limited number of uniforms (both build-in and user-defined).

Output data

- special built-in variables,
- user-defined output,

Special built-in variables

- out float gl_FragDepth;
  - replaces fragment depth (can be also discarded),
  - fragments x,y position cannot be changed,

User-defined output

- output color or discard fragment,
- multiple buffers may be updated.

User-defined output

```
void glDrawBuffers(GLsizei n, const GLenum *bufs);
        n – number of render targets.
        bufs – array of output buffers.
```

- sets the output rendering targets,

```
void glBindFragDataLocation(GLuint program, GLuint colorNum, const char *name);
        program – the handler to the program.
        colorNum – the color number to bind the user–defined varying out variable to.
        name – the name of the varying out variable whose binding to modify.
```

- the index of the target as specified in **glDrawBuffers**,
- also possible to set from shader code.

Fragment processor example

- Shade triangle!

“Advanced” example

- Rotate triangle.
- Set varying attribute (color).
- Draw inverse color.

Build-in constants

- values accessible from OpenGL API by **glGet**,
- give minumum value for OpenGL conforming implementation.

```
const int gl_MaxVertexAttribs = 16;
const int gl_MaxVertexUniformComponents = 1024;
const int gl_MaxFragmentUniformComponents = 1024;
...
```

- glGetIntegerv(GL_MAX_{VERTEX|GEOMETRY|FRAGMENT} _UNIFORM_COMPONENTS, &nComponents);
- glGetIntegerv(GL_MAX_VARYING_FLOATS, &nFloats);
- glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nAttribs);
- glGetIntegerv(GL_MAX_DRAW_BUFFERS, &nBuffers);