Advanced OpenGL
(versions $\geq 3.0$)

© 2012-2016 Josef Pelikán, Jan Horáček
CGG MFF UK Praha
pepca@cgg.mff.cuni.cz
http://cgg.mff.cuni.cz/~pepca/
Content

- OpenGL deprecation mechanisms
  - contexts, profiles
- New shader stages (geometry handling on GPU)
- Buffer Objects
- Shader Pipelines
- Shader subroutines
- Vertex Array Objects
- more little advances..
OpenGL contexts, deprecation

- “deprecation model” in OpenGL 3.0
  - removing outdated parts of the API

- OpenGL context
  - data structure in a driver holding all state information
  - context type defined in creation-time

- Full
  - complete functionality (even deprecated features)

- Forward Compatible
  - only non-deprecated features, similar to future OpenGL versions
Fixed-Functionality Pipeline

- **FFP deprecated** in OpenGL 3.1 (2009)
  - shaders are mandatory in current applications
  - FFP still usable in compatibility mode

- **server-side data** (stored on GPU: VBO, VAO, ..)
Context profiles

- introduced in OpenGL 3.2 (2009)
- better control of compatibility/deprecation..

<table>
<thead>
<tr>
<th>Context type</th>
<th>Profile</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>core</td>
<td>all from current version the whole OpenGL history</td>
</tr>
<tr>
<td></td>
<td>compatible</td>
<td></td>
</tr>
<tr>
<td>Forward Compatible</td>
<td>core</td>
<td>all “non–deprecated”</td>
</tr>
</tbody>
</table>
More GPU programming I

- new shader stage in OpenGL 3.2 (2009)
  - geometry shader
  - able to generate new geometry primitives on a GPU
  - optional shader stage
More GPU programming II

- two more shader stages in OpenGL 4.0 (March 2010)
  - tesselation shaders
    - tesselation control- & evaluation- shader
  - efficient but geometrically simple surface tesselation
  - optional
More GPU programming III

- **compute shaders** since OpenGL 4.3
- reaction to CUDA, OpenCL

**Shaders in OpenGL 4.x:**

<table>
<thead>
<tr>
<th>Shader type</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_VERTEX_SHADER</td>
<td>one vertex</td>
</tr>
<tr>
<td>GL_FRAGMENT_SHADER</td>
<td>one fragment</td>
</tr>
<tr>
<td>GL_TESSELATION_CONTROL_SHADER</td>
<td>input vertices of one patch</td>
</tr>
<tr>
<td>GL_TESSELATION_EVALUATION_SHADER</td>
<td>output vertices of one patch tesselation coordinates</td>
</tr>
<tr>
<td>GL_GEOMETRY_SHADER</td>
<td>vertices of one primitive</td>
</tr>
<tr>
<td>GL_COMPUTE_SHADER</td>
<td>data in buffers invocation indices</td>
</tr>
</tbody>
</table>
Buffer objects (version 4.x)

- **general unformatted memory** on GPU side
  - vertex data, pixel data, uniforms, compute inputs / results, ...

- **management** similar to textures/VBOs
  - glEnableBuffers(), glDeleteBuffers(), glBindBuffer(), ..

- **Immutable Storage** (unable to reallocate)
  - glBufferStorage(), ..
  - pure in-OpenGL buffers, static data, image reading, ..

- **Mutable Storage**
  - glBufferStorage(), ..
Immutable Storage

- always valid operations (server-side)
  - writing by rendering pipeline
    - Transform feedback, Image load store, Atomic counter, Shader storage buffer storage
  - clearing / copying / invalidating the buffer (server-side)
  - asynchronous pixel transfer into the buffer (pure-OpenGL)
  - glGetBufferSubData() is always available (GPU → CPU)

- client-side behavior controlled by flags:
  - mapping: GL_MAP_READ_BIT, GL_MAP_WRITE_BIT, GL_PERSISTENT_BIT (mapped & used simultaneously)
  - GL_COHERENT_BIT (explicit barrier not needed)
  - GL_CLIENT_STORAGE_BIT (client-side buffer)
Popular settings

- **pure in-OpenGL buffers**
  - pipeline data, flags set to zero

- **static data buffers**
  - e.g. input data from the application (set once – used many times), flags set to zero

- **image reading buffers**
  - for asynchronous pixel transfer operation (Pixel Buffer Objects), flags = GL_MAP_READ_BIT

- **modifiable buffers**
  - by setting flags = GL_MAP_WRITE_BIT, you promise not to use glBufferSubData()
Mutable Storage

- **glBufferData()**
  - update data + **reallocate the buffer!**
  - hints (usage): **DRAW**, **READ**, **COPY**
  - hints (frequency of a change): **STATIC**, **DYNAMIC**, **STREAM**

- **glBufferSubData()**
  - setting data **without reallocating** the buffer

- **glClearBufferSubData() / glClearBufferData()**
  - fills the part/whole buffer with the given pixel value

- **glCopyBufferSubData()**

- **glMapBufferRange()**
Selected Buffer Object Targets

- **GL_ARRAY_BUFFER**
  - vertex data

- **GL_ELEMENT_ARRAY_BUFFER**
  - vertex indices

- **GL_COPY_READ_BUFFER / GL_COPY_WRITE_BUFFER**
  - no explicit meaning

- **GL_PIXEL_UNPACK_BUFFER / GL_PIXEL_PACK_BUFFER**
  - async pixel transfer operations

- **GL_TEXTURE_BUFFER**

- **GL_TRANSFORM_FEEDBACK_BUFFER**

- **GL_UNIFORM_BUFFER**
Indexed targets

- **glBindBufferRange**( `target`, `index`, `buffer`, `offset`, `size` )
  - internally: `target[ index ]`
  - data: `buffer + offset ... buffer + offset + size`
  - `offset` is in bytes
  - `size` is in bytes and can be zero (unlimited)

- **Uniform Buffer Object**
  - uniforms stored in one GPU object
  - efficiency
  - GLSL: `uniform blocks` (vs. storage blocks)
Shader pipelines (OpenGL ≥ 4.0)

- independent shader program with only **one stage**
  - `glProgramParameteri( program, GL_PROGRAM_SEPARABLE, GL_TRUE )`

- `glGenProgramPipelines() / glDelete*()`

- `glBindProgramPipeline(pipeline)`
  - use the pipeline for subsequent calls, use it for rendering

- `glUseProgramStages(pipeline, stages, program)`
  - configure (import) individual stage[s] from program

- `glActiveShaderProgram(pipeline, program)`
  - activate the program for the pipeline
in vec3 n;
out vec4 fragColor;
uniform int mode;

vec4 AmbientColor ( vec3 n )
{
    ... 
}

vec4 DiffuseColor ( vec3 n )
{
    ... 
}

void main ()
{
    if ( mode == 0 )
        fragColor = AmbientColor( n );
    else
        fragColor = DiffuseColor( n );
}
Shader subroutines II

more GPU-friendly solution using shader subroutines (“function pointers”)

```glsl
#version 400 core

subroutine vec4 LightFunc ( vec3 n ); // subroutine type
subroutine (LightFunc) vec4 AmbientColor ( vec3 n ) { ... }
subroutine (LightFunc) vec4 DiffuseColor ( vec3 n ) { ... }
subroutine uniform LightFunc matShader; // current subroutine

void main ()
{
    matShader( n );
}
```
Shader subroutines III – application

// subroutine indices:
GLint ambient = glGetSubroutineIndex( program,
      GL_FRAGMENT_SHADER, "AmbientColor" );
GLint diffuse = glGetSubroutineIndex( program,
      GL_FRAGMENT_SHADER, "DiffuseColor" );

// matShader subroutine uniform index
GLint materialShaderLoc = glGetSubroutineUniform( program,
      "matShader" );

// set subroutine indices:
GLint numSubroutines;
glGetIntegerv( GL_ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS,
      &numSubroutines );
GLint* indices = new Gluint[ numSubroutines ];
indices[ materialShaderLoc ] = ambient;
glUniformSubroutinesuiv( GL_FRAGMENT_SHADER,
      numSubroutines, indices );
Vertex Array Objects (VAOs)

- more compact approach to setting/usage of geometry & index data
  - all VBO bindings concentrated in one call

- `glGenVertexArrays()`

- `glBindVertexArray()`
  - update all the VBOs associated with it, including index buffer

- `glBindVertexArray()` - sets everything up in one call
  - subsequent `glDraw*()` calls use the current VAO
Survey of more advances I

- **'double' arithmetic** (OpenGL 4.0)
  - not common on commodity GPUs
  - specialized GPU cards for computing (NVIDIA Tesla..)

- **GL_ARB_debug_output** (4.1)
  - callback-function based error handling, can replace `glGetError()`

- **GL_ARB_get_program_binary** (4.1)
  - not portable feature!
  - useful for shader-binary caching..?
Survey of more advances II

- **GL_ARB_shader_image_load_store** (4.2)
  - loading & storing of texture data
  - atomic texture operations

- **atomic counters** (4.2)
Sources

