Lesson 12 – Modern OpenGL
Vulkan

PV227 – GPU Rendering

Jiří Chmelík, Jan Čejka
Fakulta informatiky Masarykovy univerzity

3. 12. 2019

Modern OpenGL and Vulkan

- Theory
  - Modern OpenGL
    - ⋆ Briefly look at some not-yet-covered areas
  - Vulkan
    - ⋆ Briefly look at the basic concepts
- Practice
  - Try some methods in OpenGL

Modern OpenGL

- Separate shader objects
- Immutable storage for buffers/textures
- Texture views
- Separating format of vertex shader inputs
- Indirect drawing
- Direct State Access (DSA)

Separate shader objects

- Since OpenGL 4.1, see extension *GL_ARB_separate_shader_objects*
- Allows the programmer to use separate shaders without combining them into shader programs
- No linking – checking the input/output correctness on the fly

Immutable storage

- New way of allocating the memory for buffers/textures
  - Memory allocated only when the object is created
  - Delete and recreate the object to reallocate the memory
- Saves many checks of the driver
- Buffers
  - Since OpenGL 4.4, see extension *GL_ARB_buffer_storage*
  - The type of memory to be allocated is specified better than with *glBufferData*
    - ★ Memory accessible by CPU and GPU (for copies)
    - ★ Memory accessible only by GPU (for rendering)
- Textures
  - Since OpenGL 4.2, see extension *GL_ARB_texture_storage*
  - Allocates the texture with all mipmaps
  - Texture is always complete
  - The data is uploaded with *glTexSubImage**

Texture views

- Since OpenGL 4.3, see extension *GL_ARB_texture_view*
- Treat a part of a texture as a separate texture
  - 2D texture from a slice of an array of 2D textures
  - Cube texture from six slices of an array of 2D textures
  - ...
- Change the interpretation of the pixel data
  - Treat *GL_RGBA32F* as *GL_RGBA32UI*
  - ...
- No allocation of memory, uses the memory of the original texture
- Saves number of combination of shaders, ...

Separating vertex format

- Since OpenGL 4.3, see extension *GL_ARB_vertex_attrib_binding*
- Separates the format of vertex shader input (e.g. 3 floats without normalization) and the buffer in which the data is stored
- Binds separately the format and the buffers
- Changing the format is more complicated for the driver than setting the buffers
- Many geometries have the same format – when being rendered, only the buffers are changed

Indirect drawing

- Since OpenGL 4.0, see extension *GL_ARB_draw_indirect*
- Stores the parameters of the draw commands (first vertex to draw, number of vertices to draw, etc.) on the GPU.
- No need to transfer the parameters from CPU to GPU every frame
- The buffers can be changed from GPU, e.g. by compute shaders

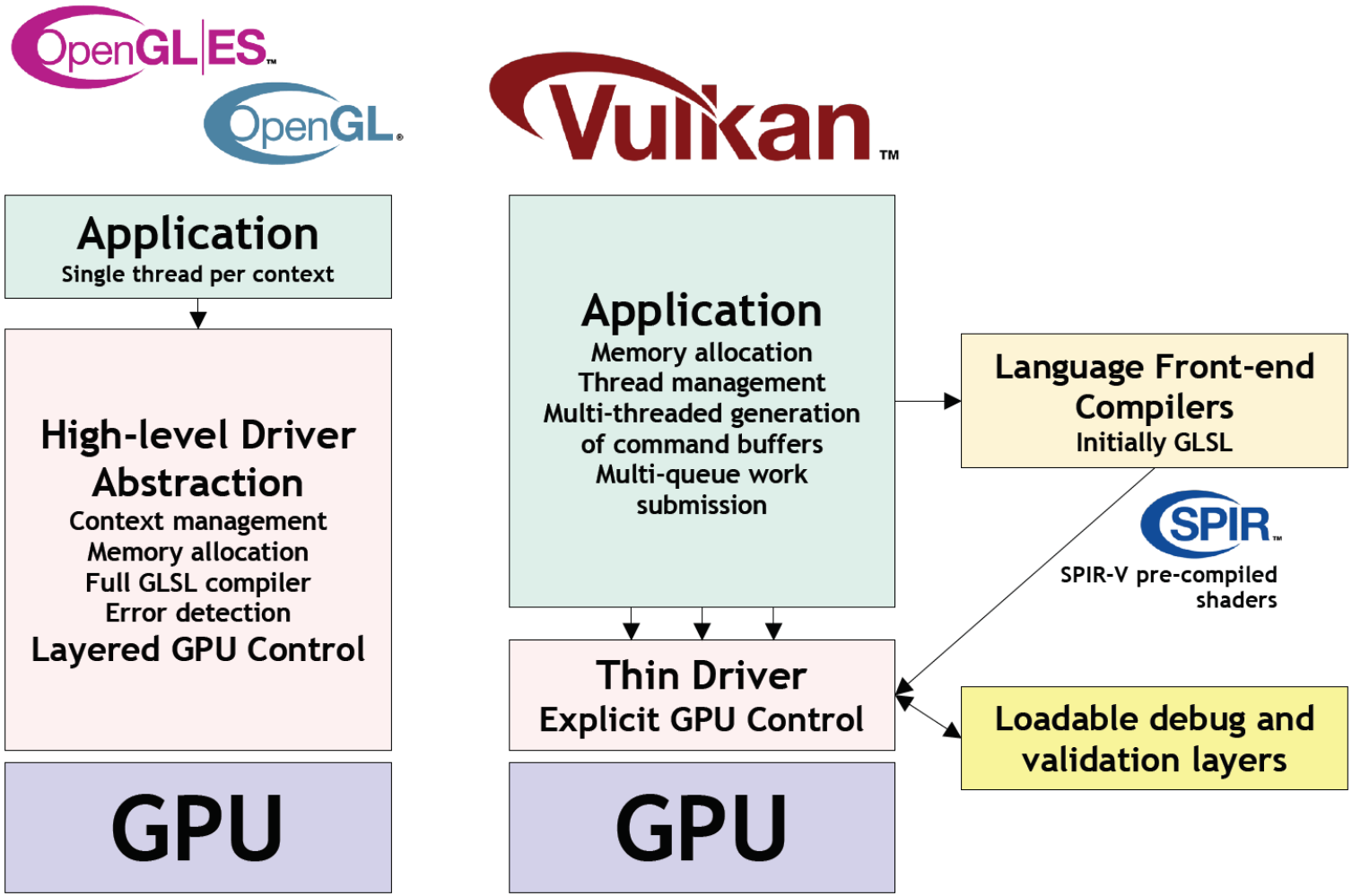
Direct State Access (DSA)

- Extension *GL_EXT_direct_state_access*
- Present OpenGL since version 4.5, but only subpart for the core profile and newest methods
- Allows us to query/change/. . . parameters of buffers/textures/. . . without binding them
  - Example: instead of
    glBindTexture(GL_TEXTURE_2D, my_tex);
    glTexParameteri(GL_TEXTURE_2D, xxx, yyy);
    use:
    glTextureParameteri(my_tex, xxx, yyy);
- Functions have very similar names

Vulkan

- Very brief introduction into Vulkan and similar APIs (Direct3D 12, Metal)
- Many concepts can be found in OpenGL via extensions
- Topics
  - Target platforms
  - Devices, rendering contexts, layers
  - Swap chain
  - Command queues and synchronization
  - Command lists
  - Pipeline state
  - Buffers and textures
  - Shaders

Vulkan vs. OpenGL



Vulkan vs. OpenGL, from Khronos: Vulkan Overview

Target platforms

- Cross-platform like OpenGL
- For desktops and mobiles (OpenGL and OpenGL ES together)
  - Mobiles (and NV Maxwell and newer) use tiled archtecture

Devices, rendering contexts, layers

- Choosing proper rendering device (graphics card)
  - Better cooperation between multiple devices
  - Can be done in OpenGL, but harder
- Vulkan uses layers as “plugins”
  - Debug layers for checking correctness of parameters
  - Layers for profiling
  - Third-party libraries, not a part of the driver
  - No layer – no checking, no debugging, fast code

Swap chain

- Mostly the same as swap chain in Direct3D
- Represents the back buffer of the window
- Accessible in rendering as a texture
- Parameters
  - Number of buffers in swap chain
  - What to do when the buffers swap

Command queues and synchronization

- Commands processed by multiple queues
  - Graphics queues (rendering)
  - Compute queues (compute shaders)
  - Transfer queues (copying the data)
- Queues run parallel between each other
- Synchronization objects
  - Synchronization between GPU and CPU
  - Synchronization between GPU queues
- The programmer cares about the synchronzation, not the driver

Command lists

- Individual commands for the API
  - Setting states
  - Draw commands
  - Copying data
  - ...
- Created on CPU, possibly in parallel
- Grouped into command lists
- Inserted into command queues to be processed

Pipeline state

- All rendering states in one pipeline state object
  - Shaders, vertex format
  - Parameters of blending, depth test, rasterization
  - ...
- The correctness is checked once when the object is created
- Very small amount of parameters can be changed after the creation
  - Viewports, scissors, stencil ref values, polygon offset, ...
- Contains the parameters of the data (e.g. vertex input format, number of attachments of FBO), but not the data itself
- Data (buffers, textures) are set separately

Buffers and textures

- Buffers and textures separated from the underlying memory
  - Memory allocated in large chunks
  - Buffers and textures are “bound” to subparts
  - The programmer manages suballocations, deals with fragmentation of the memory, . . .
  - The programmer handles updates of asynchronously used buffers.
- Sparse resources
  - Only a part of a buffer/texture has the underlying memory, the programmer must ensure that the regions accessed by shaders have the memory
  - Allows us to create very large textures (e.g. million $\times$ million pixels)
  - Useful e.g. for heightmaps – the whole heightmap is usually not accessed at the same time

Shaders

- Vulkan uses SPIR-V
  - Binary language
  - Basically any language can be compiled into SPIR-V
  - GLSL $\rightarrow$ SPIR-V compilers are available
- The code is precompiled – faster to load

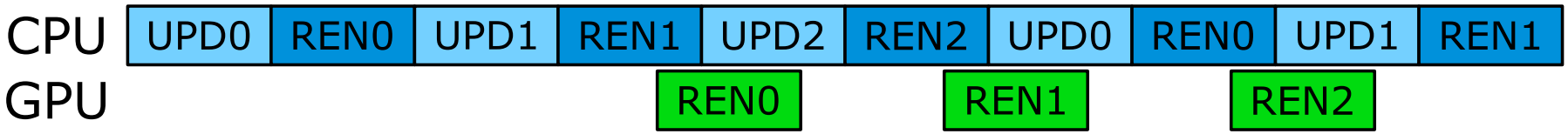
Vulkan – Conclusion

- It is not about new functions / shaders / hardware features
- It is more about better cooperation with the driver
- Many features available in OpenGL via extensions
  - Start using new way of setting input vertex format
  - Start using buffer/texture storage
  - Update the data from CPU to GPU via persistent buffers (accessible by both CPU and GPU, but not synchronized by the driver)
  - Look up bindless buffers and textures
  - Look up extension *GL_NV_command_list*
  - Look up presentations on “Approaching Zero Driver Overhead” (AZDO)

Practice

- Update the data of camera without implicit OpenGL synchronization
- Render the whole scene with a single draw command

Task: Update camera data



- Use multiple buffers, and switch them like with a circular buffer
- Use multiple fences to check that the data that you change is not used anymore

Task: Update camera data

- **Task 1:** Update the data of the camera without implicit OpenGL synchronization
  - Look into the code on how to use buffers in a new way
  - Look into the code on how to use fences
  - Set *TASK_ONE_METHOD* to *TASK_ONE_METHOD_NEW_WAY_NEW_UPDATE_CORRECT*
  - Use multiple buffers and multiple fences.

Task: Draw the whole scene with one draw command

- **Task 2:** Use NV extension and indirect drawing to create a list of draw calls and draw the whole scene with one draw command
  - Inspect the source code.
  - Set *TASK_TWO_METHOD* to *TASK_TWO_METHOD_USE*.
  - There are two places in shaders that needs to be changed.
  - Setup a new VAO object *VertexFormat_VAO* with the format of the geometry.
  - Create a rendering command for each object in the scene (including the floor)