an introduction to opengl

opengl api

- interactive graphics API
  - rendering only API
  - no interaction, math, windowing, etc.
- pipeline architecture
  - programmable and fixed stages
- heavily parallel design
  - "limited" programmability to ensures parallel efficiency
- C language API
  - with specialize language for GPU kernels

opengl api

- z-buffer rasterization
- supports points, lines, triangles
  - emulate quads as 2 triangles
  - other shapes need to be converted
- programmable operations on shape vertices and pixel fragments
  - only access the current primitive, not whole scene

opengl api

- supported on multiple platforms
  - OpenGL: desktop/laptop
  - OpenGL ES: mobile
  - WebGL: web brower
  - slight differences between platforms
- many versions available
  - mostly backward compatible
  - extension for vendor specific feature
- bad tool integration
  - notoriously hard to debug

opengl api

- mixed API design
- state
  - sets flags to configure the library
- objects
  - sets flags on "objects" to configure them
- binding parameters
  - send data to the library to process it
- shaders
  - specify programmable code
  - GLSL: C-like language

opengl on gpu

- hardware accelerated on GPU
  - supports modern GPU features
  - originally designed for general processors
- API maps well to hardware
  - programmable stages executed on GPU cores
  - fixed stages implemented on non-programmable hardware
- parallelism implicit in API design
  - achieves best efficiency

opengl pipeline

- we will present a *simplified* pipeline
  - focus on important aspects of API/GPU model
  - show CPU/GPU interop
  - avoid features less used
- we will introduce common practice in a tutorial fashion

opengl pipeline

- computation split in stages
  - both fixed and programmable
- stream/pipeline model
  - stage input/output are streams of same type data
  - also access read-only memory (e.g. params and images)
  - starts with CPU vertex data
  - ends with framebuffer image data

opengl pipeline

- fixed stages
  - configured by flags
- programmable stages
  - kernels in GLSL shaders
  - parameters bound from CPU
  - same shaders for all elements in an object

opengl pipeline

vertex data

[prog. vertex processing] transformations

transformed vertex data

[clipping and rasterization] convert to pixels

fragments w/ interpolated data

[prog. fragment processing] compute final colors

fragments color and depth

[framebuffer processing] blending hidden-surface

framebuffer

glsl shaders

- C-like language
- primitive data for graphics
  - e.g. `vec3`, `mat4`
- builtin ops for graphics
  - vector ops, matrix ops
- same language for vertex and fragment

vertex shaders

- perform same operation on each vertex
  - simple function
- input: vertex attributes
  - e.g. vertex positions, normals
- params: uniform variables
  - e.g. mesh frames/transforms, camera inverse frame, camera projection
- output:
  - [necessary] vertex position after projection
  - per-vertex data passed to other stages

fragment shaders

- perform same operation on each fragment
  - simple function
- input: fragment attributes
  - e.g. positions, normals, colors
- params: uniform variables
  - e.g. lights and material parameters
- output:
  - [necessary] fragment color

opengl tutorial

typical vertex shaders (glsl)

```
<vertex attribute declaration>
<uniform variables declaration>
<vertex output declaration>

void main() {
    <transform position to clip space>
    <transform other data>
}
```

typical vertex shaders (glsl)

- vertex attributes
  - `attribute` specifier
  - e.g. `attribute vec3 vertex_pos;`
- shader parameters
  - `uniform` specifier
  - e.g. `uniform mat4 mesh_xform;`
- output values
  - `varying` specifier
  - e.g. `varying vec3 pos;`

typical vertex shaders (glsl)

- transform vertex position in clip-space
  - assign to special variable `gl_Position`
- transform and pass vertex data down the pipeline
  - assign to output variables
- example:

```
void main() {
    gl_Position = projection * inv_camera_xform *
        mesh_xform * vec4(vertex_pos,1);
    pos = mesh_xform * vec4(vertex_pos,1);
    ...
}
```

typical fragment shaders (glsl)

```
<fragment input declaration>
<uniform variables declaration>

void main() {
    vec3 c = <compute color>
    gl_FragColor = vec4(c,1);
}
```

typical fragment shaders (glsl)

- fragment input
  - must match vertex output
  - `varying` specifier
  - e.g. `varying vec3 pos;`
- shader parameters
  - `uniform` specifier
  - e.g. `uniform int lights_num;`
  - e.g. `uniform vec3 lights_pos[16];`
  - e.g. `uniform vec3 material_kd;`
  - params arrays have fixed size

typical fragment shaders (glsl)

- compute final fragment color
  - assign to special variable `gl_FragColor`
- example:

```
void main() {
    vec3 n = normalize(n);
    vec3 c = ambient * kd;
    for(int i = 0; i < num_lights; i ++) {
        c += <compute lighting>
    }
    gl_FragColor = c;
}
```

typical opengl program

```
init_shaders()

while True:
    setup_flags()
    clear_framebuffer()
    foreach mesh:
        bind_shaders()
        bind_shaders_parameters()
        bind_vertex_attributes()
        draw_elements()
```

typical opengl program

- shader initialization
  - load shader code from file
  - create shaders `glCreateShader`
  - send code to the GPU `glShaderCode`
  - compile on the fly `glCompileShaders`
  - create program `glCreateProgram`
  - attach shaders into a program `glAttachShader`
  - bind vertex attribute locations `glBindAttribLocation`
  - link program `glLinkProgram`

typical opengl program

- set up OpenGL flags
  - change the way you want to draw
  - typically, enable depth test (`glEnable`,`GL_DEPTH_TEST`), disable culling (`glDisable`,`GL_CULL_FACE`)
  - specify drawable area in pixels `glViewport`
- clear framebuffer
  - specify background color `glClearColor`
  - clear buffers `glClear` with `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`

typical opengl program

- bind program
  - activate a program `glUseProgram`
- bind shader `uniform` parameters
  - get "location" from name `glUniformLocation`
  - sets locations' values `glUniform*`
    - for matrices `mat4`, set transpose flag
    - for arrays, include the indices in the name when finding location, e.g. `light_intensity[0]`

typical opengl program

- bind vertex attributes
  - get location `glVertexAttribLocation`
  - same value for all vertices (rare) `glVertexAttrib`
  - un value for each vertex stored as an array (common)
    - enable array use `glEnableAttribPointer`
    - specify C++ pointer `glAttribPointer`
    - disable when done `glDisableAttribPointer`

typical opengl program

- draw shape elements
  - for all method, determine element type

    (`GL_POINTS`,`GL_LINES`,`GL_TRIANGLES`,`GL_QUADS`)
  - draw arrays directly if they are sequence of elements

    `glDrawArrays`
  - or draw using indices (common) `glDrawElements`