2024-09-06-백민호(rex)

rex tutorials

viewer

- shader vsQuad(in vec3 position : 0, in vec3 normal : 1, in vec2 texcoord : 2, out PSIN vout)
  - vertex shader와 비슷한 개형을 가지고 있다.
  - gl_Position, vout.tex = texcoord으로 vsQuad의 out으로 넘겨준다
  - interface PSIN { vec2 tex; }
    - cg에서 배웠던 out vec2 tc 대신 interface 사용
  - GSLS
    - uniform : threads가 cpu에 받는 input, 일정(uniform)하고 read only, global하기 때문에 모든 shader에서 접근 가능
    - thread built-in output : gl*FragColor... input : gl_FragCoord... 주로 gl* 붙는다
      - built-in value는 varying type으로 thread마다 다르고 편집 가능하다
    - layout(location = ?) : CPU에 vertex data를 일정한 형태의 입력을 받아야 하는데, location meta data를 통해 vertex attribute를 구성할 수 있다. 우리가 VAO(또는 VBO) 만들 때 vertex의 구조와 순서를 정해준다. (cgut 참고)
    - static const size_t attrib_sizes[] = { sizeof(vertex::pos), sizeof(vertex::norm), sizeof(vertex::tex) };
    - interface block : gsls in, out, uniform 등을 그룹화 한것, struct의 gsls 버전
- shader psInvert(in PSIN pin, out vec4 pout)
  - 이전에 vs는 vertex shader의 줄임말이라 예상했는데 ps는? (fragment shader? fs?)
  - texel을 받아와서 pout으로 보낸다, pout.rgb를 통해 rgb값을 바꿀 수 있다
    - cg에서는 out vec4 fragColor(fragment shader에서)를 밖으로 보냈다
- program Invert { vs(440) = vsQuad(); fs(440) = psInvert(); };
  - Invert라는 program으로 vertex shader와 fragment shader에 위 두 함수를 붙여줌
  - 440?? : glsl 버전
- effect->bind("Invert"), effect->set_uniform("name", var_name)

    - effect : GL의 program 집합체, bind를 통해서 program을 붙여줄 수 있고, get&set_uniform을 통해서 uniform 변수를 다룰 수 있다. 추가적으로 image texture bind, vertex array를 통해서 primitive draw 연산을 수행한다.
- effect->draw_quads(); cg sample과 동일한 방식, color, depth buffer 비우고 vertex_array bind한 뒤 draw array 또는 draw element, GL_TRIANGLE_STRIP 사진은 네모

rgb2gray

- ntsc = vec3(0.299f, 0.587f, 0.114f); 사람이 인식하는 rgb 보정값

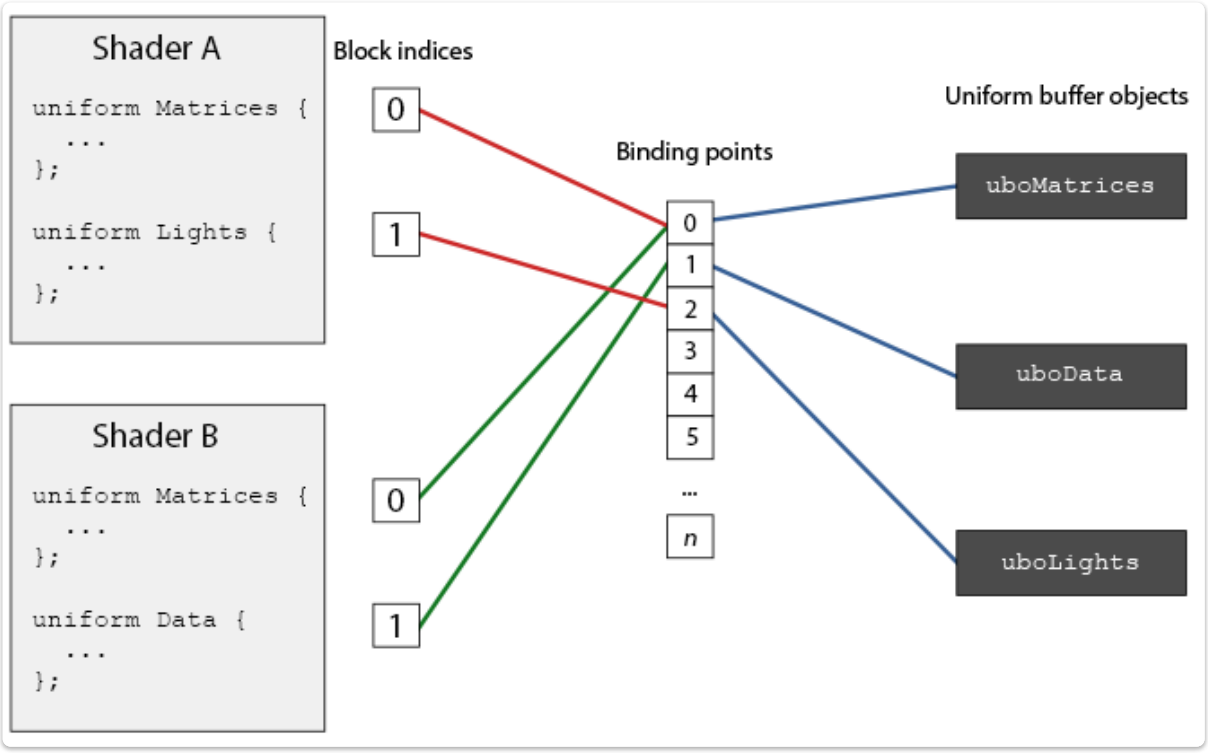
box-blur

- 2 pass를 활용한 blur, SRC -> TMP -> DST
- 커널rgb의 평균 값으로 픽셀 값 blur

moving-least-square

- 평면 이미지 tex 파일에서 추가적으로 원형 점 추가
- layout(std140, binding = 1), 기존에는 location을 이용하여 vertex attribute를 지정
    - interface처럼 uniform block, uniform buffer object를 만들어서 여러 shader program에 걸쳐 동일한 global uniform 변수 모음이다. 연관 uniform은 한번에 설정해야 함, glGenBuffers로 생성가능하고 GL_UNIFORM_BUFFER에 binding(userdata.cpp 참고)
    - std140 : 현재 정의된 uniform block은 지정된 memory layout을 사용하고 있다는 의미, 메모리에 적당한 offset이나 allign을 맞추기 위한 규칙들이 있는데 그 중에서 std140 layout을 쓰겠다는 것이다.
    - binding = ? : uniform block을 특정 binding point에 연결했다는 뜻

- 출처 : Learn OpenGL



  - 여러 uniform buffer를 여러 uniform point에 binding하여 여러 shader가 share

- cg course 학습하면서 두 오브젝트에 다른 효과를 주고싶었을 때 하나의 셰이더에 boolean을 이용하여 가지를 나눴는데, fx에서는 여러 shader program 분리
- on_dirty_image() : pp_image로 받은 이미지를 텍스쳐화? 한다. SRC에는 width와 height에 해당하는 img, DST는 빈 w, h 값을 가진 텍스쳐만 들어간다.
- fsDeform(in PSIN psin, out vec4 pout)
  : 1. 기준점의 개수(CONTROL_COUNT) 만큼 루프하면서 가중치 Wi를 기존의 Pi 위치와 계산하여 평균 Ps, Qs를 구함. pimv는 tex 좌표와 기준점 좌표의 차이 당연히 vertex와 가까운 기준점이 영향을 크게 미친다
  2.P0와 Q0 각각 P와 Q 위치에 P, Q 중앙점 차이 벡터, PWP와 PQ는 각각 PP, PQ 외적 *W*
  3.PQ 역행렬 PWP 곱하여 역방향 매핑을 위한 invM구하기, 현재 좌표 v를 변환한 새로운 좌표 f 계산 -> f = (v - Qs) * invM + Ps;

mipmap

- 2^n 사이즈의 n level로 이루어진 텍스쳐(eg. 256 x 256 -> 128 x 128, 64 x 64, 32 x 32 ...)들을 미리 만들어서 메모리에 올려 놓는 것으로 랜더링 속도가 향상된다.
- frame buffer object setting, depth나 cull 등등 T/F, bind(DST) : FBO에 텍스쳐 bind
- SRC texture 복사본으로 pout 생성(non-pot tex를 pot(2^n)로 바꿈, mipmap을 위해서!!)

- shader psBuildMipmap(in PSIN pin, out vec4 pout)
  - tc = ive2(gl_FragCoord.xy)<<1 : 픽셀 조정 2배씩 -> 해상도는 2배 감소
  - mipmap 생성 루프(render function)
    - FBO->bind(DST, 0, k) k-level rendering 대상으로 생성
    - DST->view(k-1, 1) 이전 레벨의 mipmap을 소스로 줄여나간다
- 이후 texelFetch(MIP, tc+offset, 0) 이전 레벨 텍스쳐 픽셀 값을 추출하고 유효 값들만 pout에 저장한다(offset은 현재 texel 주변 2 x 2(0,0), (1,0), (0,1), (1,1))
- create_control을 이용해서 키보드 입력 값에 대한 변화를 줄 수 있다.

minmax-mipmap

- 이전에는 주변 2x2 커널의 평균값을 이용 이번에는 minmax
- Framebuffer FBO, Effect effect, Texture DST(output) 추가로 Texture MMX(minmax mipmap 2 pass), Buffer GMX(global minmax uniform buffer)
- 1. 첫 init은 power of 2 없이 mipmap 생성
  - fbo에 mmx(texture), 0 level로 bind, init_mipmap program, sampler는 SRC
  - tc는 이전과 동일하게 픽셀 값 2배, ts는 텍스쳐 크기에 맞게
  - tc 주변 4개의 픽셀에 대하여 최솟값과 최댓값을 비교하여 pout에 저장
  - ivec2 t = tc+ivec2(0,0);
  - if(t.x<ts.x&&t.y<ts.y)
    { float z=texelFetch(SRC,t,0).x; pout=vec2(min(pout.x,z),max(pout.y,z)); }
- 2. level 값에 따른 mipmap 생성
  - for(k) loop를 통해 k-level의 mipmap texture를 fbo에 bind
  - 이전과 같은 방식에서 sampler를 MMX로 반복해서 mipmap으로 줄이기
  - effect uniform mmx는 k-1 level의 mipmap 이용
- 3. last level mipmap을 GMX로 copy
- 4. output DST로 랜더링

holegen

- on_mouse로 object interaction
- hole(x, y, size, valid?), b_selected(onClick method)
- render
  - tc = gl_FragCoord

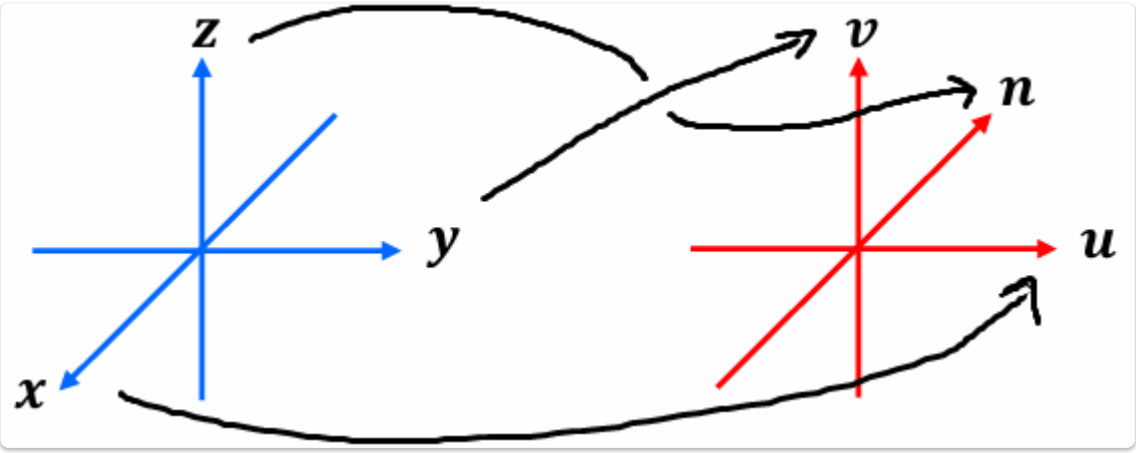
- uniform value : sampler2D SRC, vec3 hole(x, y, size)
- tex와 hole.xy 거리가 hole size보다 작으면 black or texelFetch(SRC, tc, 0)

pull-push-synth

- pull push synthesis??
  - mipmap처럼 다른 해상도를 이용해서 데이터를 보간하는 방법
- render
  - b_npot에 대해 npot texture -> pot texture 변환(padtopot)
    - ts : textureSize, 원본 texture의 w, h 저장
    - tc : 현재 픽셀 좌표(fragment 위치)
    - 현재 픽셀이 텍스쳐의 경계를 넘어가면 0,0,0,0으로 padding
  - pull phase
    - k level mipmap을 PULL texture에 저장(!b_npot 라면 k = 0 loop 1회 생략)
  - push phase
    - k = kn-2 부터 하는 이유는 kn-1은 가장 낮은 해상도라서 다룰 필요가 없기 때문
    - k-level의 PULL의 mipmap bind, PUSH k level에 넣어 줄 것(첫 loop 제외)
    - 추가적으로 k + 1 level의 PUSH texture bind(첫 loop에는 없으니 PULL texture)
    - tc 픽셀 값의 PULL texture를 가져와 pout에 저장, invalid pout에 대해 다음 처리
      - invalid한 것은 hole이고, 더 낮은 해상도를 가져와서 채우는 것
      - valid의 경우 PULL texture 그대로 pout 구멍에는 PUSH 채우기
      - PUSH(k+1) 주변 4개의 픽셀을 가져와서 pout에 더하기
      - 사각형 경계 검사, 정규화 등 진행
    - 마지막으로 PUSH 0 level DST에 draw_quads(!b_npot 경우 crop)

ubo

- uniform buffer object : uniform data를 저장하기 위한 buffer object, 다른 program 사이에서도 share 가능
  - struct 형태로 구현된 uniform data를 일괄적으로 set 가능
  - layout(std140, binding=0) uniform SPHERE{ sphere_t sphere; }; (GLSL)

    - gl::Buffer* ub_sphere = effect->bind_uniform_buffer("SPHERE");
      ub_sphere->set_data(sph_data);
    - layout(std140, binding=1, row_major) uniform MODEL{ model_t model; };
        - binding : GPU의 메모리에 저장될 위치, 각 uniform block 충돌 x
        - row_major :저장 방식에 대해 default column major, 행렬 쓸 때 cpu와 다른 방식
- effect->draw_quads() 가 아니라 VAO->draw_elements(0) + vertex shader 차이점
    - vertex에 대해 wpos, view_projection_matrix 추가 처리 필요
- view-projection matrix
    - model -> world -> view(camera가 원점) -> clip
    - {0,1,0,0, 0,0,1,0, -1,0,0,1, 0,0,0,1}, x -> y, y -> z, z -> -x에 축 맞추기
    - 

ssbo

- shader storage buffer object (openGL wiki)
    - buffer를 이용한 interface block의 일종
    - GLuint ssbo;
      glGenBuffers(1, &ssbo);
      glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
    - ubo와 차이점
        - ubo는 16KB, ssbo는 128KB까지 gpu memory를 할당할 수 있다
        - writable(even atomically), ubo는 읽기 전용 변수들을 저장, 읽고 쓰는데 incoherent memory accesses를 사용하기 때문에 memory barriers 필요
        - memory layout이나 binding qualities에 대한 special qualifier가 있음(eg. std430)
    - shader
        - struct image_data_t { ivec3 count; int pad; };

        - layout(std430, binding=1) buffer HIST { image_data_t data[]; };
        - storage block var를 사용하기 위해 atomic function 필요
            - atomicAdd, Min, And etc...(int, uint type에서만 작동한다)
- render
    - 1. pp_SRC -> SRC -> DST에 image copy
    - 2. SRC rgb 값을 HIST에 누적 연산
        - b_atomic_sync -> atomicAdd : 항상 일정한 값 출력
        - !b_atomic_sync -> 단순 += 연산 -> 값이 일정하지 않고 작음(덮어 써짐)
        - compute shader와 fragment shader(openGL wiki)
            - compute shader : GPUI 범용 계산을 수행하는데 최적화, 무언가 그리기 보다는 히스토그램 계산 같은 작업을 수행하고 랜더링 과정이 아니기 때문에 frament shader가 필요 없게 된다. (FPS나 시간 변화 차이 확인하기)
            - ComputeShader는 work group(x, y, z)이 존재하는데 work group에 여러 threads가 존재할 수 있다.
            - 실행 할 때, 몇 개의 work groups, 몇 개의 threads를 실행할 지 결정해야 한다. 이렇게 구분하는 이유는 thread를 가변적으로 설정할 때가 있고, 동일 work group의 thread는 동시 실행, 언제 실행하든 같은 output을 내야하기 때문이다.(work group 내의 shared memory와 synchronizing)
            - tc = ivec2(gl_GlobalInvocationID.xy);
                - gl_WorkGroupID : 현재 work group id
                - gl_WorkGroupSize : work group 내 thread 수
                - gl_LocalInvocationID : work group 내 현재 thread id
                - gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID

minmax-compute

- minmax mipmap 연산을 compute shader 통해 parallel하게 계산
- compute shader(csReduceMinmax)
    - total Thread = WorkGroup 내의 Thread * total WorkGroup
    - gl_NumWorkGroups, gl_WorkGroupID, LocalInvocation, GlobalIncovation
    - 동일 WorkGroup 내의 Thread는 동시에 실행된다(동시에 실행되는 것과 다를 바 없는

계산 결과를 뽑아내야 한다)
    - 동일 WorkGroup내의 Thread는 Shared Memory 가지고 있다(Synchronizing)
    - shared vec2 sdata[] : local_x, y, z 만큼의 WorkGroup 별 shared memory
    - level 0에서는 texture에서, k level에서는 ssbo인 MMX에서 값을 읽는다
    - threads 마다 min, max를 비교하여 각 WorkGroup의 최종 결과를 수집,
      tid == 0인 0 thread가 최종 mmx 값을 sdata[0]를 이용하여 global memory로 옮긴다

Nbuffers

- std::string fxname = format("%s.%s",wtoa(get_project_name()),get_name());
  출력결과) 4-nbuffers.TTGL_NBuffers
    - project_name : 4-nbuffers
    - get_name : TTGL_NBuffers(RexPlugImpl name)
- query size에 맞게 픽셀 범위를 잡고 min max 계산
- N-Buffers : n-buffer를 사용하여 tiling 없이 layer 확인, 이미지를 작은 타일로 나누지 않고 각 버퍼에 레이어를 두어서 여러 개의 레이어를 가져가는 것
- Y-Map Buffers : Y-map buffer를 사용하여 tiling 없이 1 level layer에서 layer 확인
  2d 이미지에 대해 remap 할 때 rgb가 아닌 float 좌표 값들로 x-map, y-map buffer 생성
  x, y map 값을 보고 원본 이미지에 어떤 rgb 값을 가져올 지 결정할 수 있다
- render
    - ybuffer.texture, nbuffer.texture
    - FBO->bind(texture, layer, miplevel)
    - init_buffers
        - source texture의 픽셀값의 r 값만(gray scale) 가져오기
        - pout[0], [1] 둘 다 vec2(1 - gray, gray) 값으로(min max 계산에 쓰기)
    - psQuery + NBuffer & YBuffer
        - tc(frag의 x,y 좌표) 주변 query_size 만큼 minmax 계산
        - min_or_max flag에 맞게 pout 출력
        - nbuffer, ybuffer query_buffer
            - texelFetch(NBF or YBF, ivec3(tc, 0). 0).xy로 NBF, YBF에 맞는 값

simple-fixed

- render chain에 light, camera, dynamics 추가(obj source, mesh와 배경 cube)
- UserData
  - background cube map
    - texture BG, VertexArray skydome
  - FrameBuffer, Effect ,Texture(DST, OUTPUT)
- GUIData(control box 같은 gui controls, gui에 쓰는 것들, 또는 boolean)
  - b_background
- ImportData
  - p_mesh, p_cam, p_bgimage, p_light
  - on_dirty~ callback을 통해 image 파일을 texture로 import, VA setting
- cg camera 다시 체크
- render
  - draw_background(p_cam), background.fx
    - cam이 생겼기 때문에 global position만 쓰던 vertex shader에서 cam의 projection matrix를 기준으로 frag position setting 해야 한다
    - bg_scale = const 10, 1로 하면 skydome이 너무 작게 나옴, 적당한 수로 설정
    - pout = vec4(texture(BGCUBE, normalize(pin.normal)).rgb, (-pin.epos.z - cam.dnear) / (cam.dfar - cam.dnear));
      - texture(BGCUBE, normalize(pin.normal))은 normal 벡터 방향의 샘플 rgb를 가져온다(pin.normal은 cube map으로 정규화된 좌표)
      - (-pin.epos.z - cam.dnear) / (cam.dfar - cam.dnear) 부분은 현재 픽셀의 depth를 계산. 근거리 원거리 clipping plane 사이를 정규화
  - simplefixed.fx
    - variables
      - bbox : bounding box
      - LIT : light[max_light] -> pos, color, normal
      - MAT : material[max_material] -> color, bsdf(빛의 산란), metal, rough 등
        표면 재질 + TEX, NRM, PBR
      - GEO(std430) : geometry[]
      - p_mesh->geometries.ID로 object들 확인 가능(이빨, 몸, 손톱이 따로 있음, 배경이랑 뼈 구조물 등등...)
    - fixed vertex shader
      - model_matrix = geometries[vout.draw_id].mtx, wpos = model_matrix * pos

        - mtx -> trans, rotate, scale 등 trans matrix 포함
        - epos = cam.view_matrix * wpos
        - gl_Position = cam.projection_matrix * epos, NDC로 이동하는 좌표계
    - fragment phong shading + bump mapping
        - bump_normal
            - 기존의 폴리곤을 표시하기 위해서는 최소 점 3개를 기준으로 내부 rgb 값을 vertex 사이를 보간하면서 채워넣었다. 이렇게 하면 울퉁불퉁한 효과를 내기는 어려움
            - normal 값을 적당히 조절하면서 표면은 부드럽지만 phong reflect 값을 바꿀 수 있게 하기 -> bump normal값과 기존 normal 적당히 계산
            - 기존 normal 벡터와 y, z축 벡터와의 외적을 통해 y, z 방향 법선 벡터 중 길이가 큰 값을 선정
            - 법선 벡터와 normal 외적으로 binormal 계산
            - [0, 1] 범위의 bump normal을 [-1, 1] 구간으로 정규화 -> dnorm
            - world-space bumpnormal = mat3(tangent, binorm, norm) * dnorm
        - blinn-phong shading(ambient 제외)
            - k개의 light_count로 light마다 계산
            - (vertex에서 light source로 나가는 방향) L = light.pos - vertex.pos
            - 나머지 부분은 기존과 동일
- on_size : 화면 조작 관련 기능

AccDOF(Accumulation Depth Of Field)

- 줌에 따른 초점 변화
- Texture ACC, sampler_t sampler(blur 섞는 개수) 추가
- 기존의 rendering 과정을 sampler size만큼 반복(시간 점유율 높음)
    - FBO bind : OUTPUT->slice(0)
    - PD = sampler[k].xy -> lens
    - 카메라 렌즈의 왜곡이나 초점을 바꾸기 위해 미세한 위치 변화
    - shader에 bg_df, bg_E, bg_PD?? 없애도 실행되는 걸로 보임
    - 각 geometry에 대해 draw(accdof)
    - 기존 vertex shader에서 epos.xy 값을 조정, 마구 흔들어서 나중에 쌓으면서 blur
        - samples 값을 2나 3으로 했을 때 효과가 잘보인다

        - vout.epos.xy += cam.E *PD* (cam.df + vout.epos.z) / cam.df;
        - cam.E : 카메라의 줌, 초점 조정 요소
        - PD : 렌즈 위치 offset
        - cam.df : 카메라의 초점 거리(5321.0으로 constant)
        - 카메라 초점 거리와 vertex의 z 값
    - phong shading에 epos 넣을 때는 xy에 cam.E *PD* sign(cam.df+pin.epos.z) 빼주기
    - 나머지 동작은 동일
- rgb 값 accumulate
    - FBO bind : ACC->slice(0)
    - blend를 true 해줘야 섞인다
    - OUTPUT texture의 rgb 값을 a값은 1.0인 상태로 fragColor에 texelFetch
        - 이때 vs의 gl_Position은 PD로 흔들지 않은 정상 좌표
- normalize
    - FBO bind : OUTPUT->view(0, 1, 0, 1)

quad-wraping

- mouse interaction(with anchor)
- update : fx bind 한 뒤, anchor point attribute만 업데이트 해줌
- geometry shader
    - vertex shader, frag shader 사이 선택적 shader
    - 기본 primitive의 vertex array를 받아서 적당하게 가공한다
    - vertex 추가나 다른 primitive로 변환 가능, EmitVertex를 통해 해당 순간의 정점 정보들이 output buffer로 전달, EndPrimitive 함수로 primitive를 만들 수 있다
    - wireframe 썼을 때, anchors와 VAO(texture) 차이점 확인해보기
    - quad_wraping에서는 texture 모서리에 위치한 anchor를 위해 모서리를 기준으로 4방향으로 복제하는 역할을 한다
    - 근데 이게 main image에도 적용이 되는 것 같다, 이를 구분하기 위해 gl_VertexID를 사용, GPU에서 자동으로 할당해주고 b.draw_anchor에 대해서는 0,1,2,3의 고유 인덱스를 할당해준다
- on_mouse
    - selected.index -> 누른 anchor index return, else -1
    - seleted activate 상태라면 move_anchor(selected는 global)
    - 떼면 selected.reset
- userdata

- on_dirty_src
  - SRC size를 재고, DST texture를 세팅
  - src width, height : 1024, 681 (image 크기), factory : 1920, 1080
    - 각각 image_aspect, viewport_aspect --> relative aspect(1:0.xx)
    - anchor의 네 꼭짓점을 -1, 1사이의 x, y 좌표로 분배
- on_dirty_anchor_size()
  - 화면 비율에 맞춰 anchor의 size 조절
  - size x, y = 0.0167, 0.0230
  - vec2 size 의미가 무엇인가??
- intersect : mouse pos를 잡고 anchor에 닿았는지
  - mouse pos를 local viewport 좌표로 변환(canonical_in_local_viewport)
    - [0,1]의 x,y pos를 [-1, 1] 기준으로 변환
    - a.pos(박스 중심) 좌표가 각각 [-1, 1]사이에 사각형으로 퍼져 있고 a.size를 통해 박스 위에 있는지 감지하고 해당 인덱스 리턴

particle

- 이전과는 다르게 on_size() 부분에서 DST texture를 만들고 있다, default width, height가 변할 때, factory->WIDTH, HEIGHT 등으로 부르기도 가능
- CG course assn 1과 비슷하게 particles들의 pos, dir 물리량을 결정해주면 된다 여기서는 추가로 life와 재생성 효과가 있다.
  - particles와 particle_buffer가 따로 존재한다. particles은 N개의 particle 위치, 방향, life 값 등을 계속 보존하지만 particle_buffer는 매 프레임마다 초기화를 반복한다. 계산 파트와 렌더링 파트를 분리해서 하기 때문인 것 같다
  - partcle point는 PTS uniform buffer object에 저장, vec4의 좌표 값만 저장함
- array<particle_t, N>
- draw_points
  - texture를 쓰지 않을 때에는 단순 흰색으로,
  - pout = texture(SRC, pin.tex) 기본 텍스쳐 입히기
  - alpha : rgb grayscale로 바꾸기(NTSC 표준), alpha 미적용 시, 네모 배경 텍스쳐가 그대로 보여서 겹칠 때 가려짐, 검정색 배경이 alpha가 0이 되기 때문, 대신 뭔가 흐리멍텅해보임
  - geometry shader
    - 중심 point를 기준으로 texture를 입히기 위해 사각형 만들기
    - 새로운 vertex를 만들기 위해 gout.tex와 gl_Position을 설정하고 EmitVertex

        - 이전 shader 단계에서 출력을 얻는 방법 : glsl 내장 gl_in[] 내장 변수, gl_Position 가지고 있다
        - texture의 [0, 1]을 [-1, 1]과 aspect에 맞게 정규화 시키기

gbuffer-phong

- Deffered Shading, Geometry Buffer
    - 우선 opposite Forward Shading은 기존에 배운 rendering pipeilne 과정을 수행
    - deffered는 미룬이, 미룬이 셰이딩 기법이다, pipeline을 2번 거친다(2 pass)
    - 1st pass : GBuffer에 shading에 필요한 자원을 저장
        - pixel의 wpos, diffuse color, normal, light wpos etc...
        - 사용자가 원하는 만큼 Gbuffer에 데이터를 저장할 수 있다
        - GPU가 Multiple Render Target 기능을 지원해야 한다고 한다~
        - 본 코드에서도 SSBO를 통해 struct geometry 저장
    - 2nd pass : GBuffer에 저장된 자원들로 Shading 연산 수행
        - pixel shader lighting 연산만 하면 됨!
    - 1st pass에서 depth test를 진행하여 pixel에 하나의 obj를 골라내기 때문에 lighting
      연산이 크게 줄어든다
- vsFixed
    - GEO, geometries[] = p_mesh->buffer.geometry
    - geometries[id].mtx *vertex local pos => wpos, wpos* cam.view_matrix => epos
    - epos * cam.proj_matrix => gl_Position
    - normal도 vertex normal에서 geometries[id].mtx 곱해주기
- psGBuffer
    - 해당 geometry의 material 정보 m, gub
    - 알베도는 material color, normal은 cam.view_matrix 곱하기(일명 eye normal)
    - encode_gbuffer 의미??
        - pout으로 frag 하기 위한 정보들을 vec4 형태로 리턴
        - x : g.albedo의 rgba 값을 32비트로 압축하여 저장
        - y : normal.xy 값을 각각 sqrt(z + 1) 나누고 또 압축해서 저장
        - z : linear depth(epos.z 값과 카메라의 dnear, dfar을 통해 구한다)
        - w : 물체의 id 값