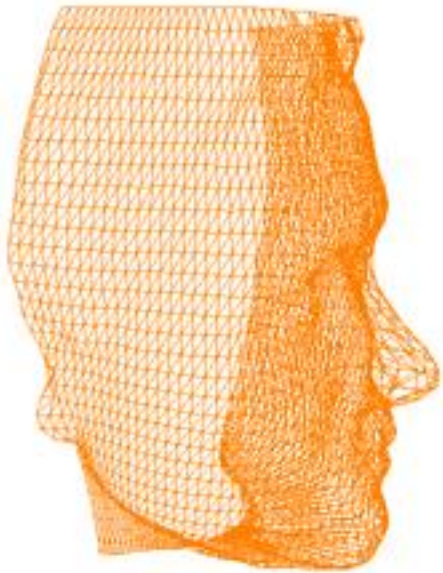


COSC422 Advanced Computer Graphics

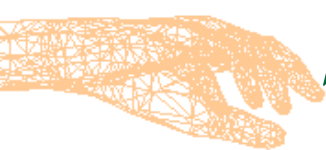


4 Transform Feedback

Semester 2
2021



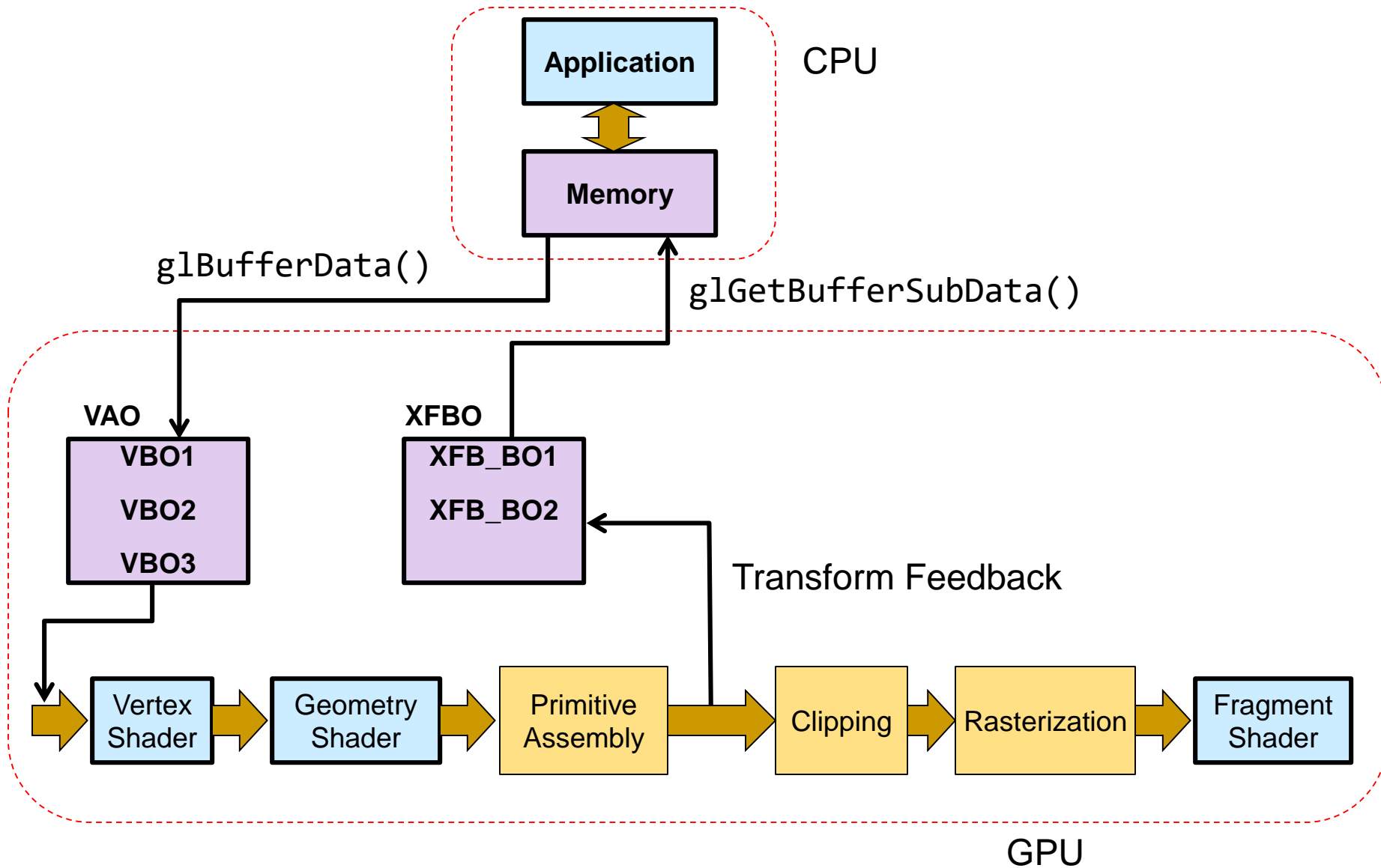
R. Mukundan (mukundan@canterbury.ac.nz)
Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

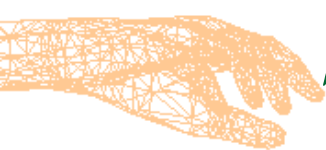


Transform Feedback

- ❑ Transform feedback allows the capture of vertex attributes as they are assembled into primitives, usually triangles (**before rasterization**).
- ❑ The attributes can be recorded into one or more transformation feedback buffer objects (XFB_BO).
 - ❑ The contents of a feedback buffer object are generally used as inputs to the vertex shader in a subsequent rendering pass.
 - ❑ The buffer objects can be read back by the application (e.g., for further processing of the outputs)

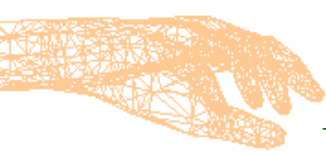
Transform Feedback





Transform Feedback

- ❑ Transform feedback can only capture outputs for primitive types points, lines and triangles.
- ❑ Transform feedback can capture outputs of a vertex shader only if a geometry shader is not active.
- ❑ If a geometry shader is active, only its outputs can be captured in the transform feedback buffer. In order to capture a data generated by the vertex shader, it must be passed through the geometry shader.
- ❑ If a geometry shader outputs a triangle strip with 6 vertices, the transform feedback will receive 12 vertices corresponding to 4 triangles.



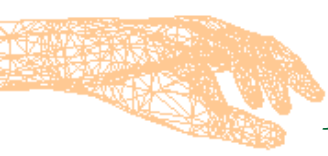
Applications: Particle Systems

- A particle's position P_n and velocity v_n within a particle system may change with time, or undergo random perturbations, requiring incremental updates.

$$P_{n+1} = P_n + (v_n + r_1)\Delta t + r_2 \quad \text{where, } v_n \text{ denotes velocity.}$$

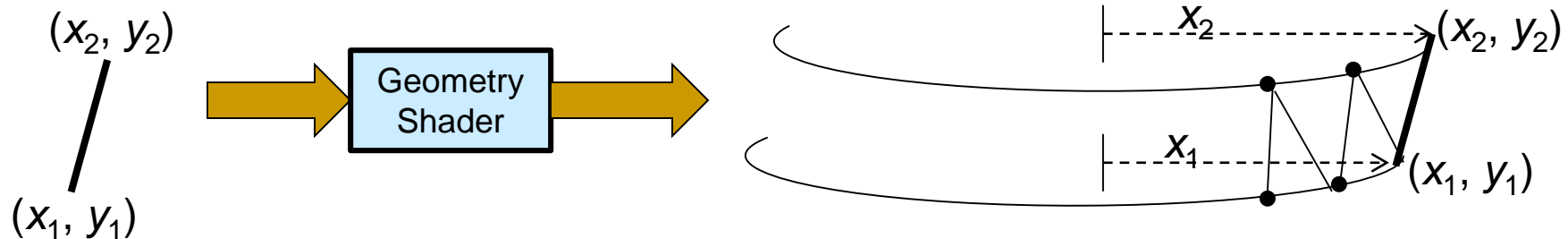
r_1, r_2 denote random numbers.

- We will need to store each particle's updated position and velocity (in transform feedback buffers) for use in the next iteration.
- The geometry shader can be used to easily create new particles or delete existing particles. The entire particle system can be stored in transformation feedback buffers and managed inside the GPU.

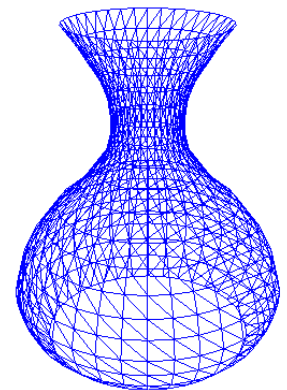


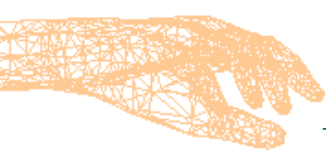
Applications: Mesh Generation

- A geometry shader can be used to generate a complex model from a set of simple primitives.



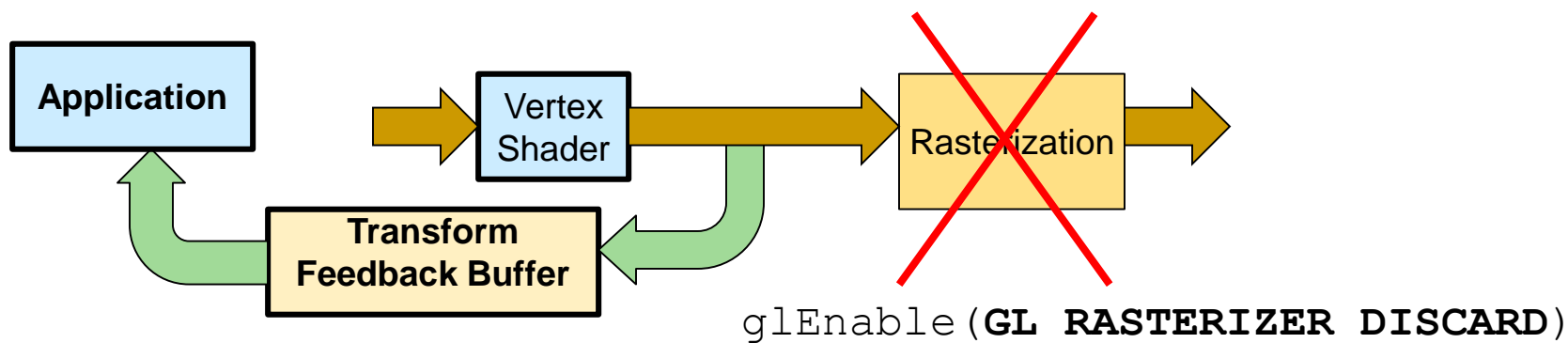
- The vertices of the generated mesh object can be stored in a transform feedback buffer for
 - generating a model definition file
 - subsequent rendering of the generated model.

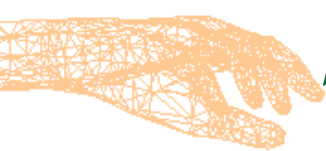




Applications: GPGPU

- ❑ Elements of vertex buffer objects are accessed and processed parallelly in vertex shaders.
- ❑ The output of some computation performed in the vertex shader using the array values can be stored in a transformation feedback buffer.
- ❑ Enabling `GL_RASTERIZER_DISCARD` prevents any primitives (points, triangles) from being rasterized and displayed.





Transform Feedback (Step 1)

Specify the vertex attributes to be captured as “out” variables.

```
layout (location = 0)
    in vec3 position;
out vec3 xfbPos;
out vec3 xfbVel;

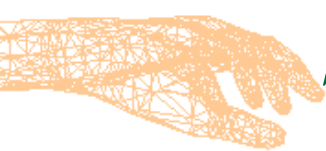
void main() {
    ...
    gl_Position = ...
    xfbPos = ...
}
```

Vertex Shader

OR

```
#version 400
layout (lines) in;
layout (triangle_strip) out;
out vec3 xfbPos;
out vec3 xfbVel;
...
void main() {
    xfbPos = ...
    xfbNor = ...
    EmitVertex();
    ...
}
```

Geometry Shader

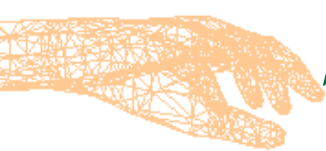


Transform Feedback (Step 2)

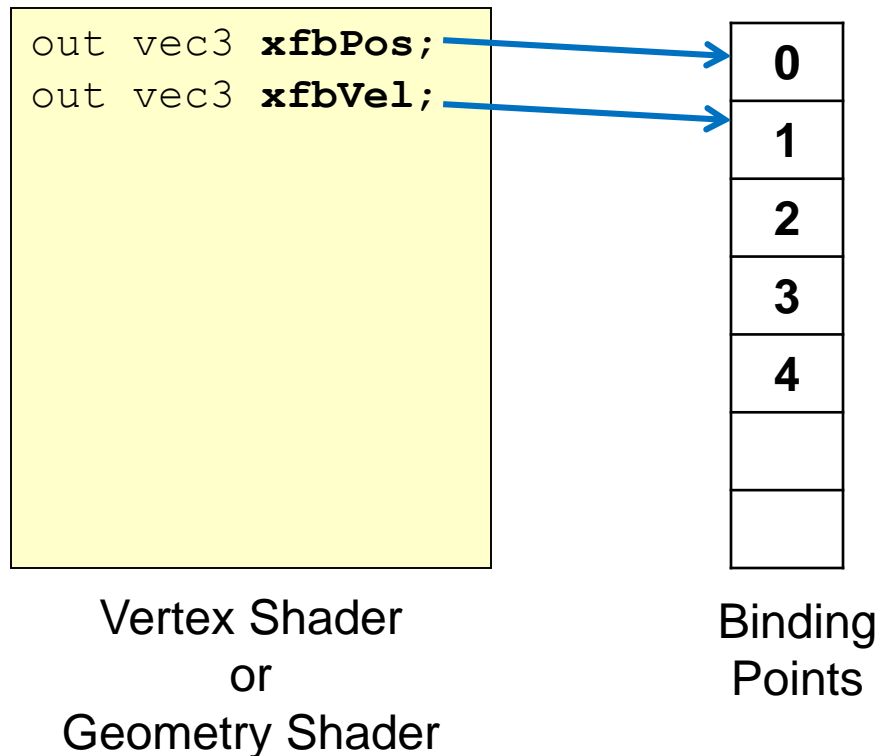
In the application, **define** the names of the shader variables as “transform feedback varyings”. This must be done after attaching shader objects, but before linking the program object.

```
GLuint program = glCreateProgram();  
glAttachShader(program, shader);  
const GLchar* xfbVars[] = {"xfbPos", "xfbVel" };  
glTransformFeedbackVaryings  
    (program, 2, xfbVars, GL_SEPARATE_ATTRIBS);  
glLinkProgram(program);
```

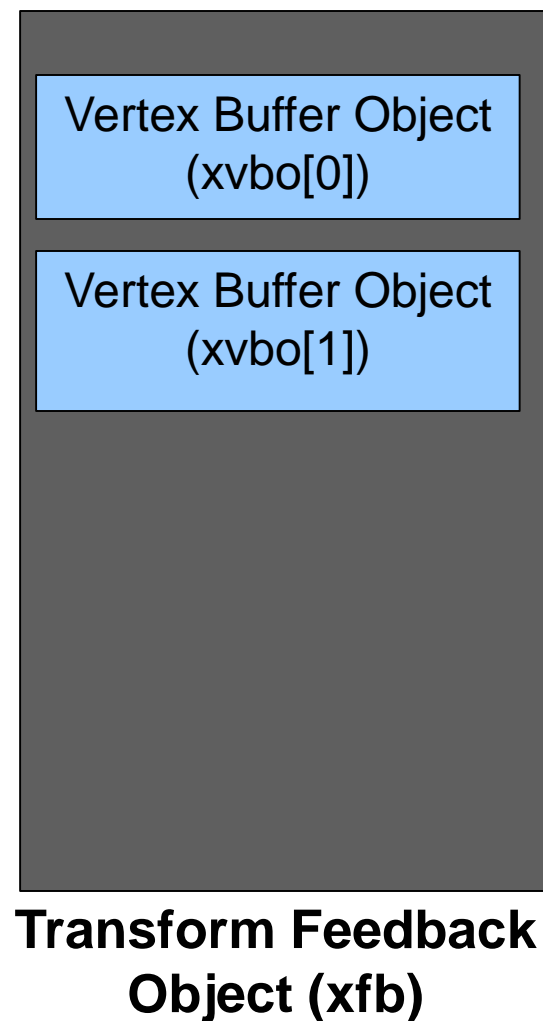
Variables “xfbPos”, “xfbVel” are now associated with xfb binding points 0, 1 (the order in which they are defined in the above array)

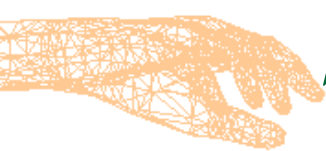


Transform Feedback



Transform feedback buffer is an indexed target

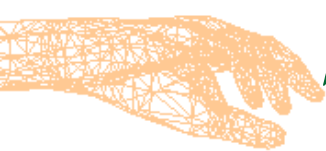




Transform Feedback (Step 3)

Create a “Transform Feedback Object” (similar to a VAO) and attach buffer objects to them:

```
glGenTransformFeedbacks(1, &xfb);  
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, xfb);  
  
GLuint xvbo[2];                                //2 VBOs  
glGenBuffers(2, xvbo);  
  
glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, xvbo[0]);  
glBufferData(GL_TRANSFORM_FEEDBACK_BUFFER,  
             size1, NULL, GL_DYNAMIC_COPY);  
glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, xvbo[1]);  
glBufferData(GL_TRANSFORM_FEEDBACK_BUFFER,  
             size2, NULL, GL_DYNAMIC_COPY);
```



Transform Feedback (Step 4)

Associate the buffer objects with the binding points

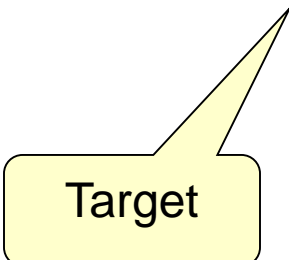
```
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, xfb);
```

```
glBindBufferBase
```

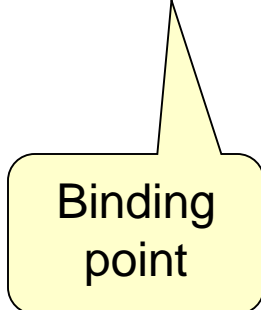
```
(GL_TRANSFORM_FEEDBACK_BUFFER, 0, xvbo[0]);
```

```
glBindBufferBase
```

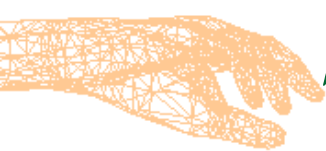
```
(GL_TRANSFORM_FEEDBACK_BUFFER, 1, xvbo[1]);
```



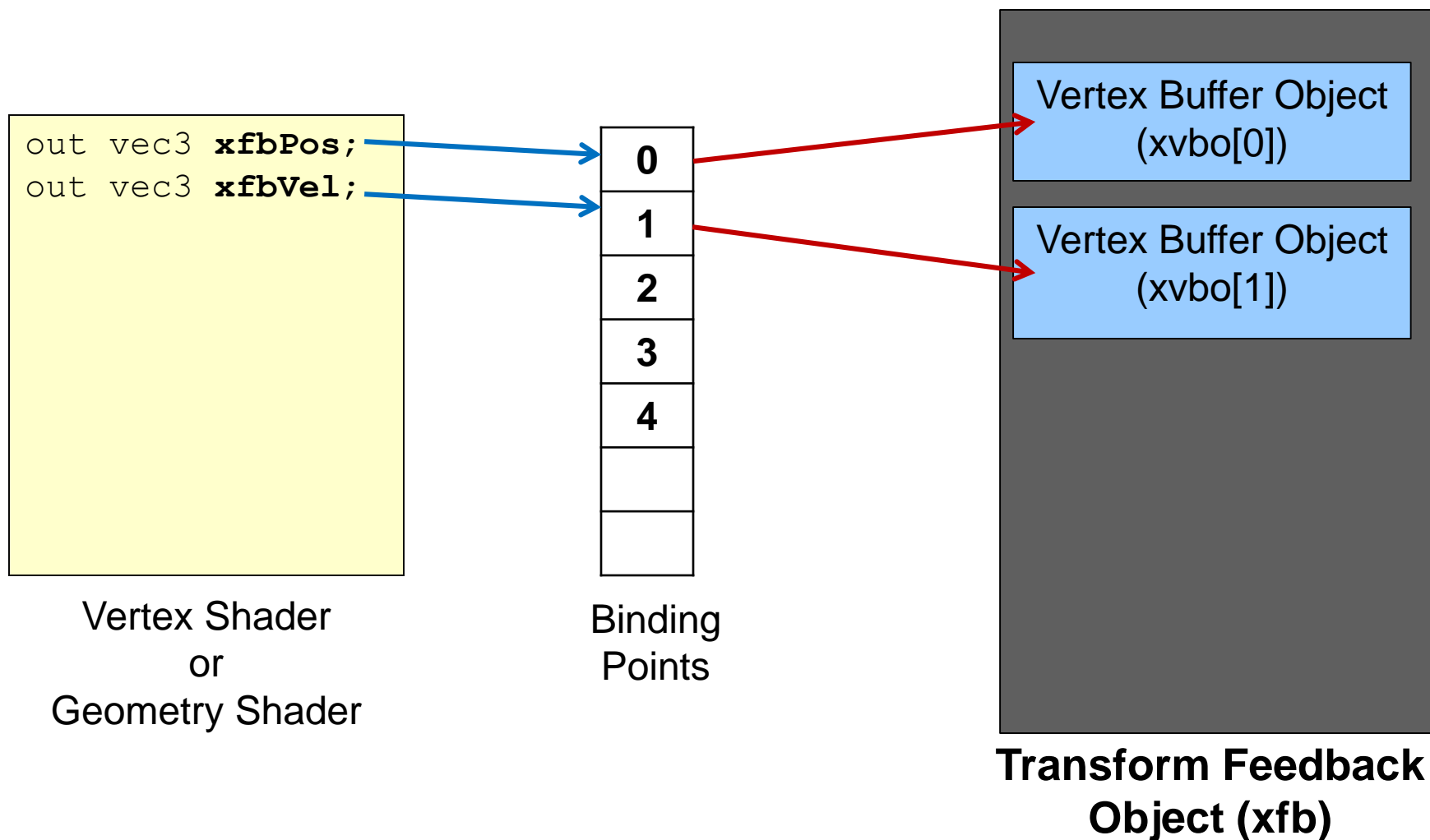
Target

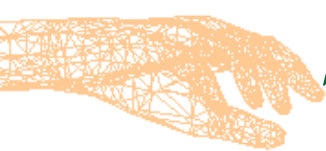


Binding
point



Transformation Feedback





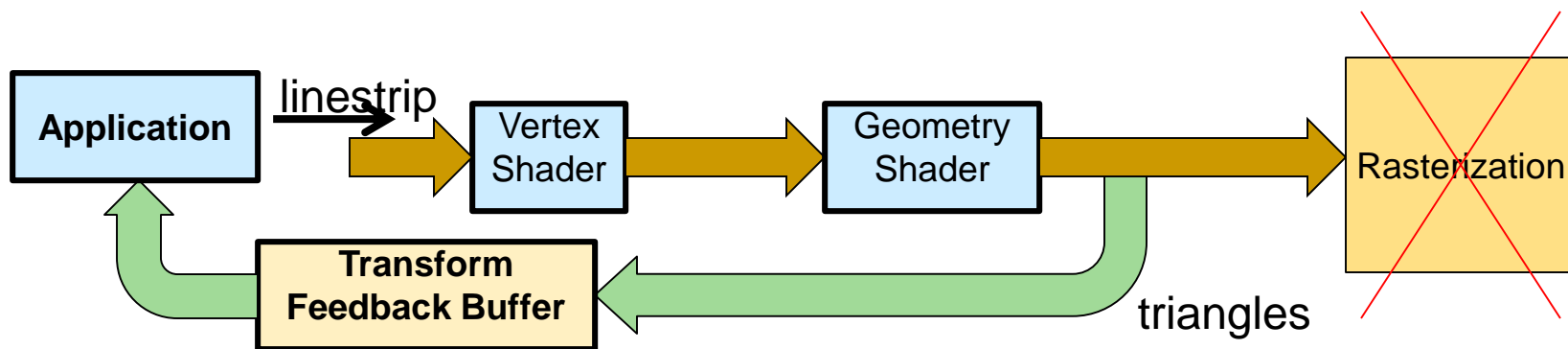
Transform Feedback (Step 5)

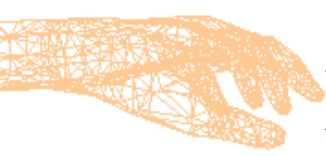
Begin capture of data. If rendering of the model is not required, rasterization of primitives may be disabled:

Note: The following can be done in “initialise()” function:

```
glEnable(GL_RASTERIZER_DISCARD);  
glBeginTransformFeedback(GL_TRIANGLES);  
glDrawArrays(GL_LINE_STRIP, 0, 19);  
glEndTransformFeedback();  
glFlush();
```

Primitive
type





Rendering Captured Data

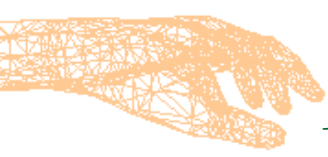
This is what we did for rendering data from a VAO:

```
glBindVertexArray(vao)
glDrawArrays(GL_TRIANGLES, 0, count);
```

For a transform feedback object, we can query the number of feedback primitives written, but this information is not always necessary. We can render the captured data directly (inside display function) using

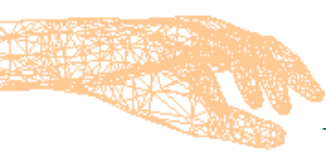
```
glDrawTransformFeedback(GL_TRIANGLES, xfb);
```

Feedback
object id



Application: Mesh Generation

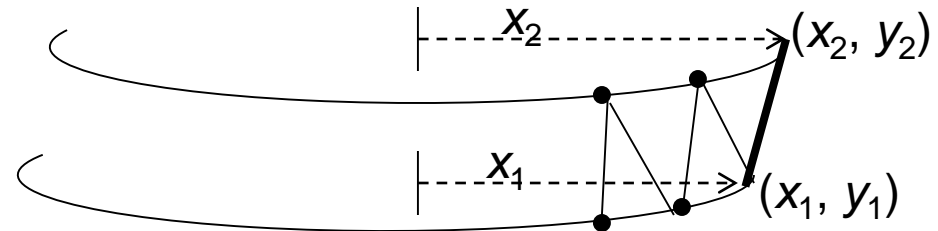
- ❑ The next slide shows the code for the geometry shader used in the construction of a surface of revolution.
- ❑ The vertices of the triangle strip are stored in the variable “xfbPosn” and their values are captured in the transformation feedback buffer.

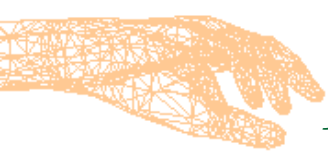


Application: Mesh Generation

```
layout (lines) in;
layout (triangle_strip, max_vertices = 80) out;
out vec3 xfbPosn;
uniform mat4 mvpMatrix;

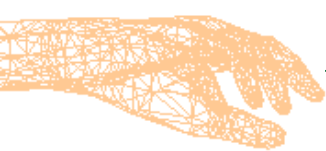
void main() {
    float rad1 = gl_in[0].gl_Position.x,    rad2 = gl_in[1].gl_Position.x;    //Top and bottom radii of the strip
    float y1 = gl_in[0].gl_Position.y,      y2 = gl_in[1].gl_Position.y;
    float x1, x2, z1, z2, theta, cthet, sthet;
    int i;
    for(i = 0; i <= 36; i++)
    {
        if(i == 0 || i == 36) theta = 0.0;
        else theta = i * 0.17453; //10 degs
        cthet = cos(theta);    sthet = sin(theta);
        x1 = rad1*cthет;    z1 = -rad1*sthet;
        xfbPosn = vec3(x1, y1, z1);
        gl_Position = mvpMatrix * vec4(fbPosn, 1);
        EmitVertex();
        x2 = rad2*cthет;    z2 = -rad2*sthet;
        xfbPosn = vec3(x2, y2, z2);
        gl_Position = mvpMatrix * vec4(fbPosn, 1);
        EmitVertex();
    }
}
```





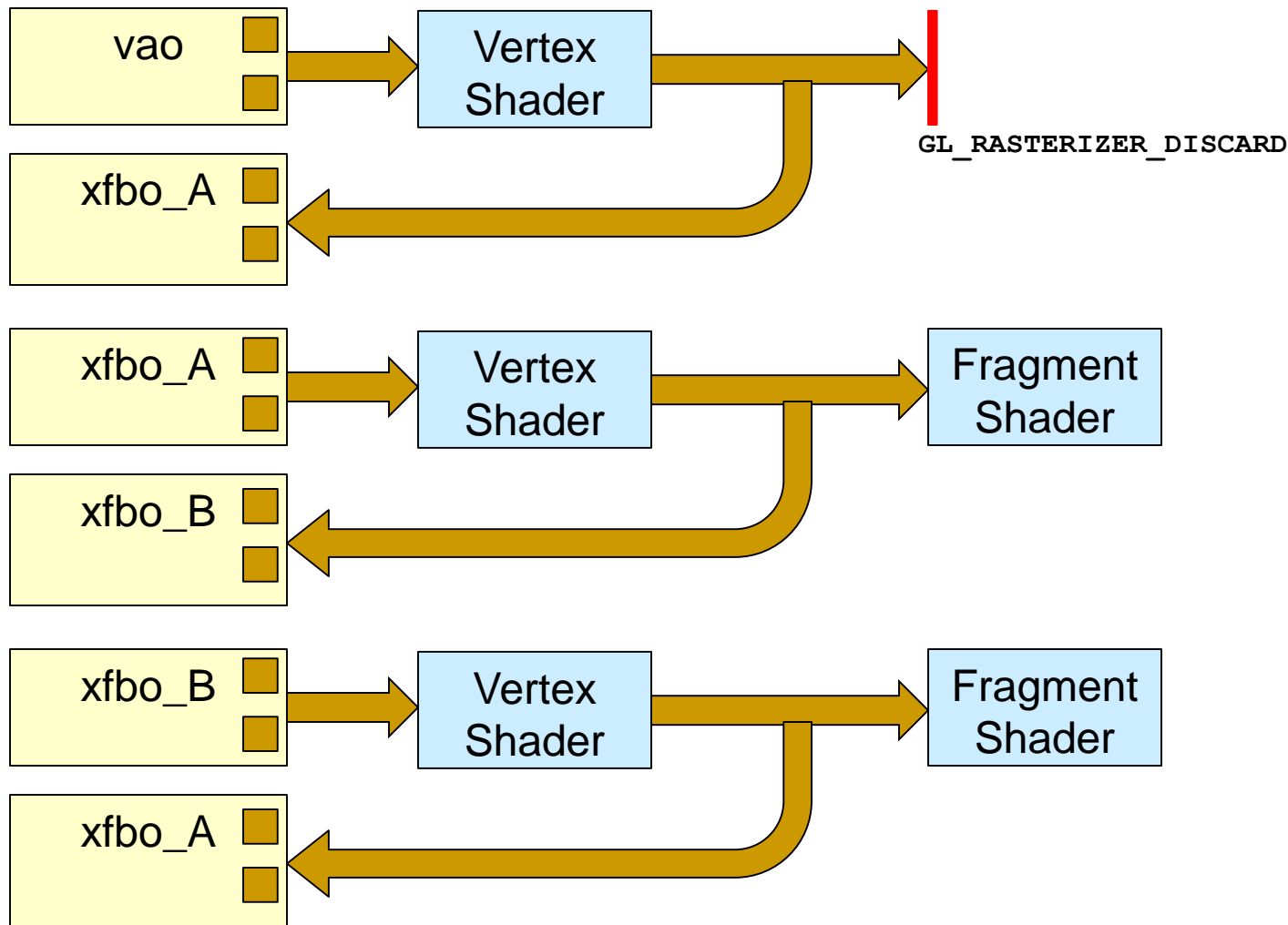
Application: Particle Systems

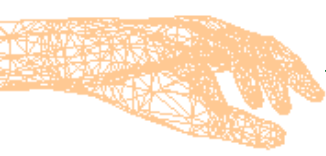
- ❑ We capture the transformed positions of the particles in a transformation feedback buffer.
- ❑ In the next step, we use the data from the feedback buffer as input, and capture their transformed values in another transformation feedback buffer.
- ❑ We can now switch the buffers as shown on the next slide. This process is called buffer ping-ponging.



Particle Systems: Transform Feedback

Buffer ping-ponging





Particle Systems: Geometry Shader

- ❑ Geometry shader receives points and their attributes (velocity, life time)
- ❑ Points are classified into emitter particles and regular particles

