OpenGL® Programming Guide

Ninth Edition

The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V

John Kessenich • Graham Sellers • Dave Shreiner

The Khronos OpenGL ARB Working Group

Praise for previous editions of OpenGL® Programming Guide

"Wow! This book is basically one-stop shopping for OpenGL information. It is the kind of book that I will be reaching for a lot. Thanks to Dave, Graham, John, and Bill for an amazing effort."

—Mike Bailey, professor, Oregon State University

"The most recent Red Book parallels the grand tradition of OpenGL; continuous evolution towards ever-greater power and efficiency. The eighth edition contains up-to-the minute information about the latest standard and new features, along with a solid grounding in modern OpenGL techniques that will work anywhere. The Red Book continues to be an essential reference for all new employees at my simulation company. What else can be said about this essential guide? I laughed, I cried, it was much better than Cats—I'll read it again and again."

—Bob Kuehne, president, Blue Newt Software

"OpenGL has undergone enormous changes since its inception twenty years ago. This new edition is your practical guide to using the OpenGL of today. Modern OpenGL is centered on the use of shaders, and this edition of the Programming Guide jumps right in, with shaders covered in depth in Chapter 2. It continues in later chapters with even more specifics on everything from texturing to compute shaders. No matter how well you know it or how long you've been doing it, if you are going to write an OpenGL program, you want to have a copy of the *OpenGL® Programming Guide* handy."

—Marc Olano, associate professor, UMBC

"If you are looking for the definitive guide to programming with the very latest version of OpenGL, look no further. The authors of this book have been deeply involved in the creation of OpenGL 4.3, and everything you need to know about the cutting edge of this industry-leading API is laid out here in a clear, logical, and insightful manner."

—Neil Trevett, president, Khronos Group

feedback is paused, it is still considered active but will not record any data into the transform feedback buffers. There are also several restrictions about changing state related to transform feedback while transform feedback is active but paused:

- The currently bound transform feedback object may not be changed.
- It is not possible to bind different buffers to the **GL_TRANSFORM_FEEDBACK_BUFFER** binding points.
- The current program object cannot be changed.[3]

void **glPauseTransformFeedback**(void);

Pauses the recording of varyings in transform feedback mode. Transform feedback may be resumed by calling **glResumeTransformFeedback()**.

glPauseTransformFeedback() will generate an error if transform feedback is not active or if it is already paused. To restart transform feedback while it is paused, **glResumeTransformFeedback()** must be used (not **glBeginTransformFeedback()**). Likewise, **glResumeTransformFeedback()** will generate an error if it is called when transform feedback is not active or if it is active but not paused.

void **glResumeTransformFeedback**(void);

Resumes transform feedback that has previously been paused by a call to **glPauseTransformFeedback()**.

When you've completed rendering all of the primitives for transform feedback, you change back to normal rendering mode by calling **glEndTransformFeedback()**.

void **glEndTransformFeedback**(void);

Completes the recording of varyings in transform feedback mode.

3. Actually, it is possible to change the current program object, but an error will be generated by **glResumeTransformFeedback()** if the program object that was current when **glBeginTransformFeedback()** was called is no longer current. So be sure to put the original program object back before calling **glResumeTransformFeedback()**.

Transform Feedback Example—Particle System

This section contains the description of a moderately complex use of transform feedback. The application uses transform feedback in two ways to implement a particle system. On a first pass, transform feedback is used to capture geometry as it passes through the OpenGL pipeline. The captured geometry is then used in a second pass along with another instance of transform feedback in order to implement a particle system that uses the vertex shader to perform collision detection between particles and the rendered geometry. A schematic of the system is shown in Figure 5.19.

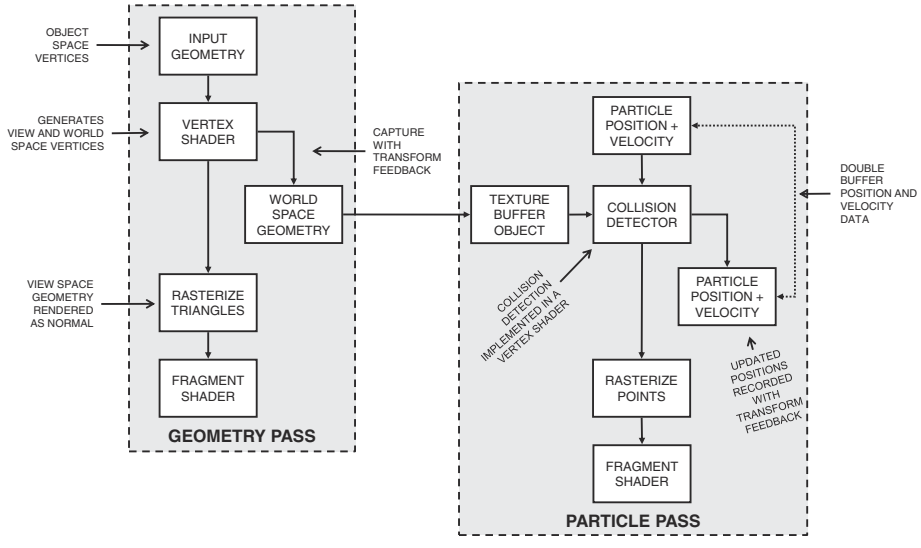

Figure 5.19 Schematic of the particle system simulator

In this application, the particle system is simulated in world space. In a first pass, a vertex shader is used to transform object space geometry into both world space (for later use in the particle system simulation) and into eye space for rendering. The world space results are captured into a buffer using transform feedback, while the eye space geometry is passed through to the rasterizer. The buffer containing the captured world space geometry is attached to a *texture buffer object* (TBO) so that it can be randomly accessed in the vertex shader that is used to implement collision detection in the second, simulation pass. Using this mechanism, any object that would normally be rendered can be captured so long as the vertex (or geometry) shader produces world space vertices in addition to eye space vertices. This allows the particle system to interact with multiple objects, potentially

with each rendered using a different set of shaders—perhaps even with tessellation enabled or other procedurally generated geometry.[4]

The second pass is where the particle system simulation occurs. Particle position and velocity vectors are stored in a pair of buffers. Two buffers are used so that data can be double-buffered, as it's not possible to update vertex data in place. Each vertex in the buffer represents a single particle in the system. Each instance of the vertex shader performs collision detection between the particle (using its velocity to compute where it will move to during the time-step) and all of the geometry captured during the first pass. It calculates new position and velocity vectors, which are captured using transform feedback and written into a buffer object ready for the next step in the simulation.

Example 5.11 contains the source of the vertex shader used to transform the incoming geometry into both world and eye space, and Example 5.12 shows how transform feedback is configured to capture the resulting world space geometry.

Example 5.11 Vertex Shader Used in Geometry Pass of Particle System Simulator

```
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out vec4 world_space_position;

out vec3 vs_fs_normal;

void main(void)
{
    vec4 pos = (model_matrix * (position * vec4(1.0, 1.0, 1.0, 1.0)));
    world_space_position = pos;
    vs_fs_normal = normalize((model_matrix * vec4(normal, 0.0)).xyz);
    gl_Position = projection_matrix * pos;
};
```

4. Be careful: Tessellation can generate a very large amount of geometry, all of which the simulated particles must be tested against, which could severely affect performance and increase storage requirements for the intermediate geometry.

Example 5.12 Configuring the Geometry Pass of the Particle System Simulator

```
static const char * varyings2[] =
{
    "world_space_position"
};

glTransformFeedbackVaryings(render_prog, 1, varyings2,
                            GL_INTERLEAVED_ATTRIBS);

glLinkProgram(render_prog);
```

During the first geometry pass, the code in Examples 5.11 and 5.12 will cause the world space geometry to be captured into a buffer object. Each triangle in the buffer is represented by three vertices[5] that are read (three at a time) during the second pass into the vertex shader and used to perform line segment against triangle intersection test. A TBO is used to access the data in the intermediate buffer so that the three vertices can be read in a simple for loop. The line segment is formed by taking the particle's current position and using its velocity to calculate where it *will be* at the end of the time step. This is performed for every captured triangle. If a collision is found, the point's new position is reflected about the plane of the triangle to make it "bounce" off the geometry.

Example 5.13 contains the code of the vertex shader used to perform collision detection in the simulation pass.

Example 5.13 Vertex Shader Used in Simulation Pass of Particle System Simulator

```
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;
uniform int triangle_count;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 velocity;

out vec4 position_out;
out vec3 velocity_out;
```

5. Only triangles are used here. It's not possible to perform a meaningful physical collision detection between a line segment and another line segment or a point. Also, individual triangles are required for this to work. If strips or fans are present in the input geometry, it may be necessary to include a geometry shader in order to convert the connected triangles into independent triangles.

```
uniform samplerBuffer geometry_tbo;
uniform float time_step = 0.02;

bool intersect(vec3 origin, vec3 direction, vec3 v0, vec3 v1, vec3 v2,
               out vec3 point)
{
    vec3 u, v, n;
    vec3 w0, w;
    float r, a, b;

    u = (v1 - v0);
    v = (v2 - v0);
    n = cross(u, v);

    w0 = origin - v0;
    a = -dot(n, w0);
    b = dot(n, direction);

    r = a / b;
    if (r < 0.0 || r > 1.0)
        return false;

    point = origin + r * direction;

    float uu, uv, vv, wu, wv, D;

    uu = dot(u, u);
    uv = dot(u, v);
    vv = dot(v, v);
    w = point - v0;
    wu = dot(w, u);
    wv = dot(w, v);
    D = uv * uv - uu * vv;

    float s, t;

    s = (uv * wv - vv * wu) / D;
    if (s < 0.0 || s > 1.0)
        return false;
    t = (uv * wu - uu * wv) / D;
    if (t < 0.0 || (s + t) > 1.0)
        return false;

    return true;
}

vec3 reflect_vector(vec3 v, vec3 n)
{
    return v - 2.0 * dot(v, n) * n;
}

void main(void)
```

```
{
    vec3 acceleration = vec3(0.0, -0.3, 0.0);
    vec3 new_velocity = velocity + acceleration * time_step;
    vec4 new_position = position + vec4(new_velocity * time_step, 0.0);
    vec3 v0, v1, v2;
    vec3 point;
    int i;
    for (i = 0; i < triangle_count; i++)
    {
        v0 = texelFetch(geometry_tbo, i * 3).xyz;
        v1 = texelFetch(geometry_tbo, i * 3 + 1).xyz;
        v2 = texelFetch(geometry_tbo, i * 3 + 2).xyz;
        if (intersect(position.xyz, position.xyz - new_position.xyz,
                      v0, v1, v2, point))
        {
            vec3 n = normalize(cross(v1 - v0, v2 - v0));
            new_position = vec4(point
                                + reflect_vector(new_position.xyz -
                                                 point, n), 1.0);
            new_velocity = 0.8 * reflect_vector(new_velocity, n);
        }
    }
    if (new_position.y < -40.0)
    {
        new_position = vec4(-new_position.x * 0.3, position.y + 80.0,
                            0.0, 1.0);
        new_velocity *= vec3(0.2, 0.1, -0.3);
    }
    velocity_out = new_velocity * 0.9999;
    position_out = new_position;
    gl_Position = projection_matrix * (model_matrix * position);
};
```

The code to set up transform feedback to capture the updated particle position and velocity vectors is shown in Example 5.14.

Example 5.14 Configuring the Simulation Pass of the Particle System Simulator

```
static const char * varyings[] =
{
    "position_out", "velocity_out"
};

glTransformFeedbackVaryings(update_prog, 2, varyings,
                            GL_INTERLEAVED_ATTRIBS);

glLinkProgram(update_prog);
```