



Vertex animation

in OpenGL

The background is a complex 3D visualization. It features a dense field of small, rectangular blocks or cubes. These blocks are colored in a gradient: dark browns and greys on the left, transitioning through oranges and reds in the center, and then into blues, greens, and purples on the right. Two prominent, semi-transparent planes intersect diagonally across the scene. One plane is a solid light blue, and the other is a dark grey. They appear to be slicing through the 3D block structure.

Introduction

Vertex animation

Introduction

- Shader ability too use parallelism in the modern graphics card makes them suitable to modify the vertex position directly, basically implement animation on the graphic card
- This can provide a boost in efficiency if the animation algorithm can be parallelized appropriately
- A shader must not only compute the positions, but often it must write out the updated positions for use in the next frame and it can do so via a number of techniques including shader storage buffer objects and image load/store
- It can send the values of the vertex or geometry shader's output variables to an arbitrary buffer (or buffers). This feature is called **transform feedback**, and is particularly useful for particle systems




Animating a surface

using vertices

Animating a surface

- A straightforward way to leverage shaders for animation is to simply transform the vertices within the vertex shader based on some time-dependent function.
- The OpenGL application supplies static geometry, and the vertex shader modifies the geometry using the current time (supplied as a uniform variable).
- This moves the computation of the vertex position from the CPU to the GPU, and leverages whatever parallelism the graphics driver makes available.



Animating a surface

- We'll create a waving surface by transforming the vertices of a tessellated quad based on a sine wave.
- We'll send down the pipeline a set of triangles that make up a flat surface in the x-z plane.
- In the vertex shader, we'll transform the y coordinate of each vertex based on a time-dependent sine function, and compute the normal vector of the transformed vertex.

Animating a surface

- We'll transform the y coordinate of the surface as a function of the current time and the x coordinate. To do so, we'll use the basic plane wave equation:

$$y(x, t) = A \sin\left(\frac{2\pi}{\lambda} (x - vt)\right)$$

A – wave's amplitude (the height of peaks)

λ – wavelength (the distance between successive peaks)

v – wave velocity

t – time

Animating a surface

- In order to render the surface with proper shading, we also need the **normal vector** at the transformed location. We can compute the normal vector using the (partial) derivative of the previous function:

$$\mathbf{n}(x, t) = \left(-A \frac{2\pi}{\lambda} \cos\left(\frac{2\pi}{\lambda} (x - vt) \right), 1 \right)$$

A – wave's amplitude (the height of peaks)

λ – wavelength (the distance between successive peaks)

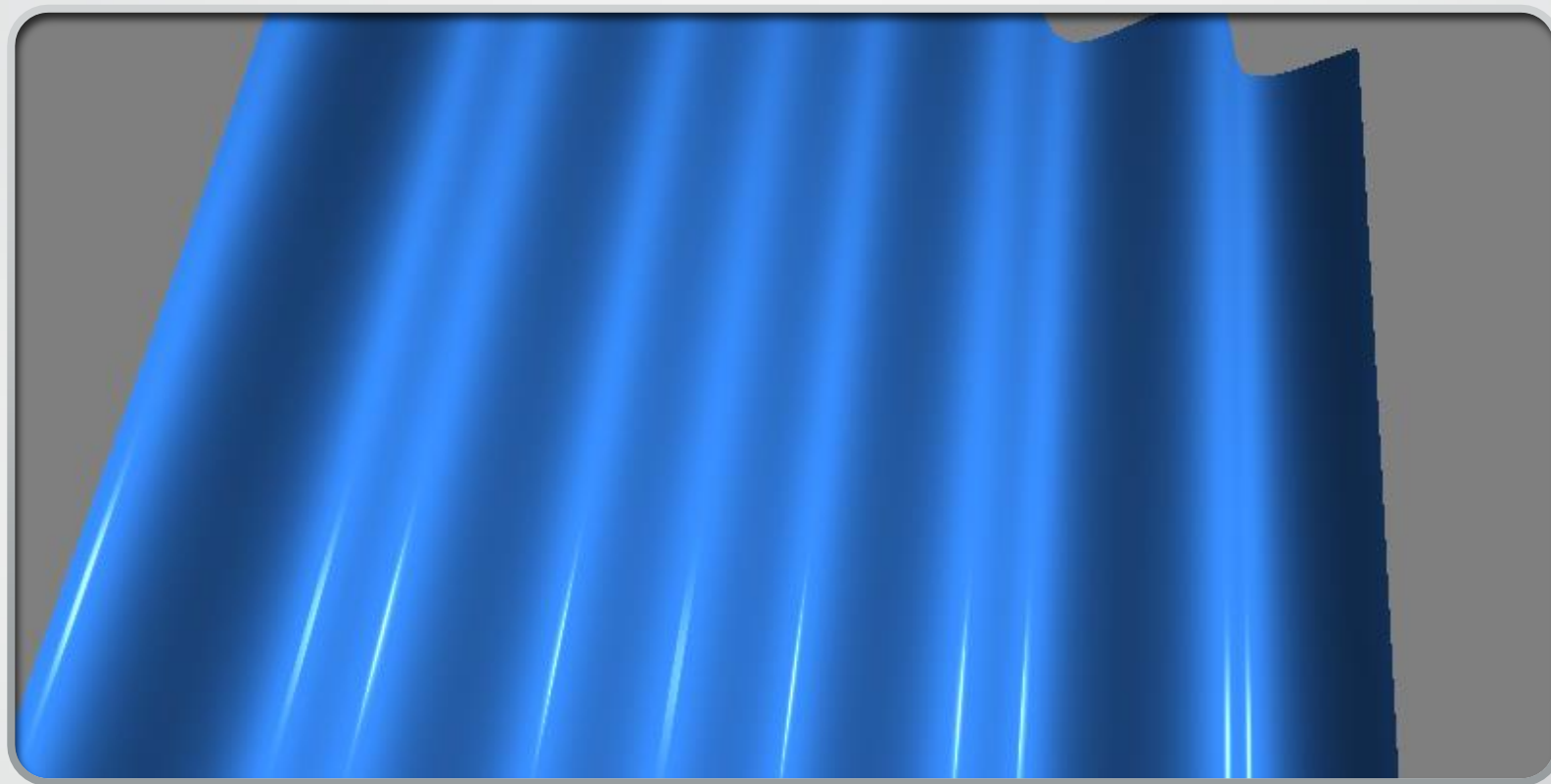
v – wave velocity

t – time

Make sure you normalise the vector after calculation.

Animating a surface

- Set up your OpenGL application to render a flat, tessellated surface in the x-z plane. The results will look better if you use a large number of triangles.
- Provide the current time to the vertex shader via a uniform variable called **Time**
- We use **k** as the wavenumber ($\frac{2\pi}{\lambda}$)
- **Velocity** is the wave velocity
- **Amp** is the wave's amplitude



Animating a surface



Particle fountain

animation

Particle fountain

- A **particle system** is a group of objects that are used to simulate a variety of fuzzy systems such as smoke, liquid spray, fire, explosions, or other similar phenomena.
- Each particle is considered to be a point object with a **position**, but no size.
- Each particle has a **lifetime**: it is born, animates according to a set of rules, and then dies. The particle can then be resurrected and go through the entire process again.

Particle fountain

- During the lifetime of a particle, it is animated according to a set of rules. These rules include the basic kinematic equations that define the movement of a particle that is subjected to constant acceleration (such as a gravitational field).
- In addition, we might take into account things such as wind, friction, or other factors. The particle may also change shape or transparency during its lifetime.
- Once the particle has reached a certain age (or position), it is considered to be dead and can be recycled and used again.

A particle fountain simulation showing a spray of particles emerging from a blue and grey nozzle on the left. The particles are represented as small white dots against a black background, forming a fan-like shape as they travel upwards and to the right. The nozzle is a 3D-like structure with blue and grey surfaces.

Particle fountain

- The particles in this example will not be recycled. Once they have reached the end of their lifetime, we'll draw them as fully transparent so that they are effectively invisible. This gives the fountain a finite lifetime, as if it only has a limited supply of material.
- To animate the particles, we'll use the standard kinematics equation for objects under constant acceleration:

$$P(t) = P_0 + v_0 t + \frac{1}{2} a t^2$$

P_0 - initial position

v_0 - initial velocity

a - acceleration

Particle fountain

- We'll define the initial position of all particles to be the origin (0,0,0). The initial velocity will be determined randomly within a range of values.
- Each particle will be created at a slightly different time, so the time that we use in the previous equation will be relative to the start time for the particle.
- Since the initial position is the same for all particles, we won't need to provide it as an input attribute to the shader. Instead, we'll just provide two other vertex attributes: **the initial velocity** and the **start time** (the particle's time of birth).
- Prior to the particle's birth time, we'll render it completely transparent. During its lifetime, the particle's position will be determined using the previous equation with a value for t that is relative to the particle's start time (**Time – StartTime**)

Particle fountain

- To render our particles, we'll use a technique called **instancing**, along with a simple trick to generate screen-aligned quads.
- We don't actually need any vertex buffers for the quad itself! Instead, we'll just invoke the vertex shader six times for each particle in order to generate two triangles (a quad). In the vertex shader, we'll compute the positions of the vertices as offsets from the particle's position. If we do so in screen space, we can easily create a screen-aligned quad. We'll need to provide input attributes that include the particle's **initial velocity** and **birth time**.
- This technique makes use of the vertex shader to do all the work of animating the particles. We gain a great deal of efficiency over computing the positions on the CPU. The GPU can execute the vertex shader in parallel, and process several particles at once.

Particle fountain

- The core of this technique involves the use of the **glDrawArraysInstanced** function. This is similar to **glDrawArrays** but instead of just drawing once it does it repeatedly.
- Normally, we move to the next element with each invocation of the vertex shader (essentially once per vertex). However, with instanced drawing, we don't always want that. We might want several of invocations to get the same input value.
- Each particle in our particle system has six vertices (two triangles). For each of these six vertices, we want the same **velocity**, (particle) **position**, and other per-particle parameters.

Particle fountain implementation

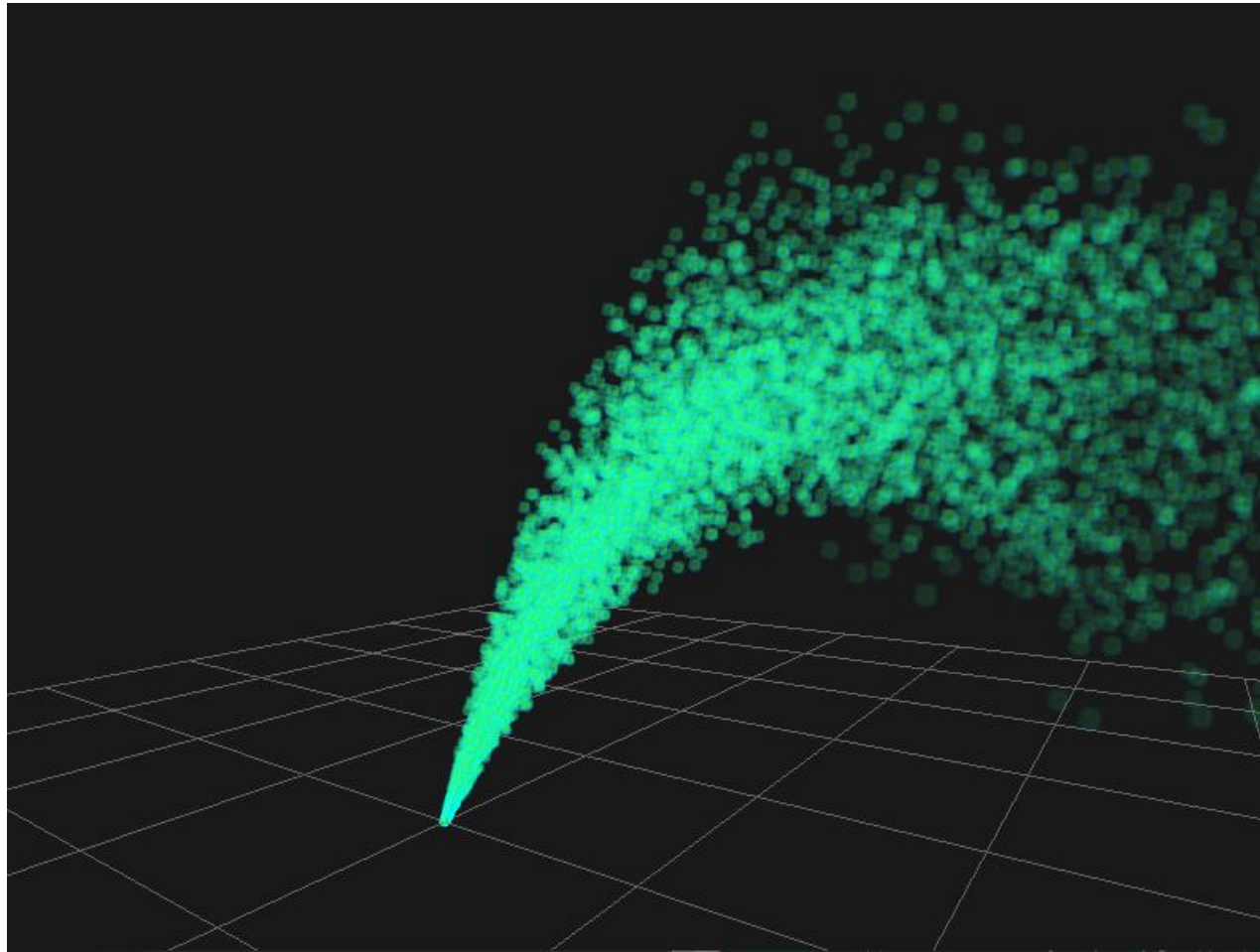
- We'll create two buffers (or a single interleaved buffer) to store our input attributes. The first buffer will store the **initial velocity** for each particle. We'll choose the values randomly from a limited range of possible vectors.
- To create the cone of particles, we'll choose randomly from a set of vectors within the cone. We will tilt the cone toward some direction by applying a rotation matrix
- To pick vectors from within our cone, we utilize spherical coordinates.
 - **theta** determines the angle between the centre of the cone and the vector.
 - The value of **phi** defines the possible directions around the **y** axis for a given value of **theta**

Particle fountain implementation

- In the second buffer, we'll store the **start time** for each particle. This will provide only a single float per vertex (particle). For this example, we'll just create each particle in succession at a fixed rate.
- The array is then copied into the buffer referred to by **startTime**. Set this buffer to be the input for vertex attribute one.
- We set the divisor for both attributes to one. This ensures that all vertices of a particle will receive the same value for the attributes. This should be executed while the vertex array object (VAO) is bound. The divisor information is stored within the VAO:

```
glVertexAttribDivisor(0,1);
```

```
glVertexAttribDivisor(1,1);
```



Particle
fountain



Particle fountain

with transform feedback

Particles with feedback

- Transform feedback provides a way to capture the output of the vertex (or geometry) shader to a buffer for use in subsequent passes.
- This feature is particularly well-suited for particle systems, because among other things, it enables us to do discrete simulations.
- We can update a particle's position within the vertex shader and render that updated position in a subsequent pass (or the same pass). Then the updated positions can be used in the same way as input to the next frame of animation.

Particles with feedback

- Instead of using an equation that describes the particle's motion for all time, we'll update the particle positions incrementally, solving the equations of motion based on the forces involved at the time each frame is rendered.
- A common technique is to make use of the **Euler method**, which approximates the position and velocity at time **t** based on the position, velocity, and acceleration at an earlier time

$$P_{n+1} = P_n + v_n h$$

$$v_{n+1} = v_n + a_n h$$

P- particle position

v – particle velocity

h – time step (amount of time elapsed between frames)

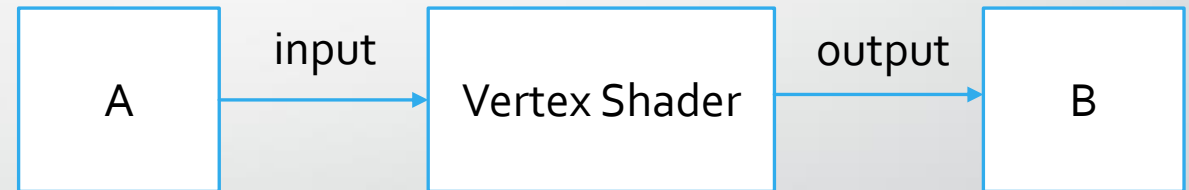
a – acceleration

n – frame number

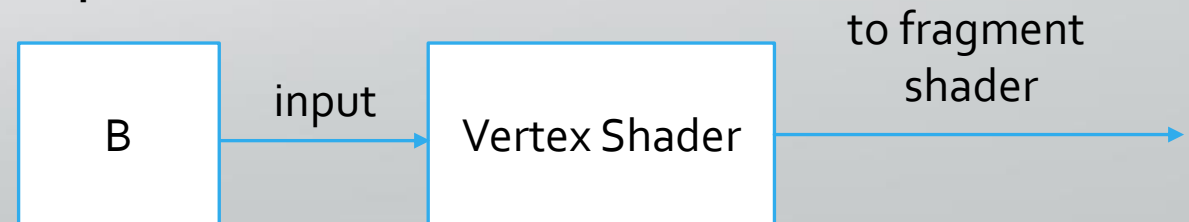
Particles with feedback

- We'll use a technique sometimes called **buffer ping-ponging**. We maintain two sets of vertex buffers and swap their uses each frame.
- For example, we use buffer A to provide the positions and velocities as input to the vertex shader. The vertex shader updates the positions and velocities using the Euler method and sends the results to buffer B using transform feedback. Then, in a second pass, we render the particles using buffer B:

Update pass (no rasterization)



Render pass



Particles with feedback

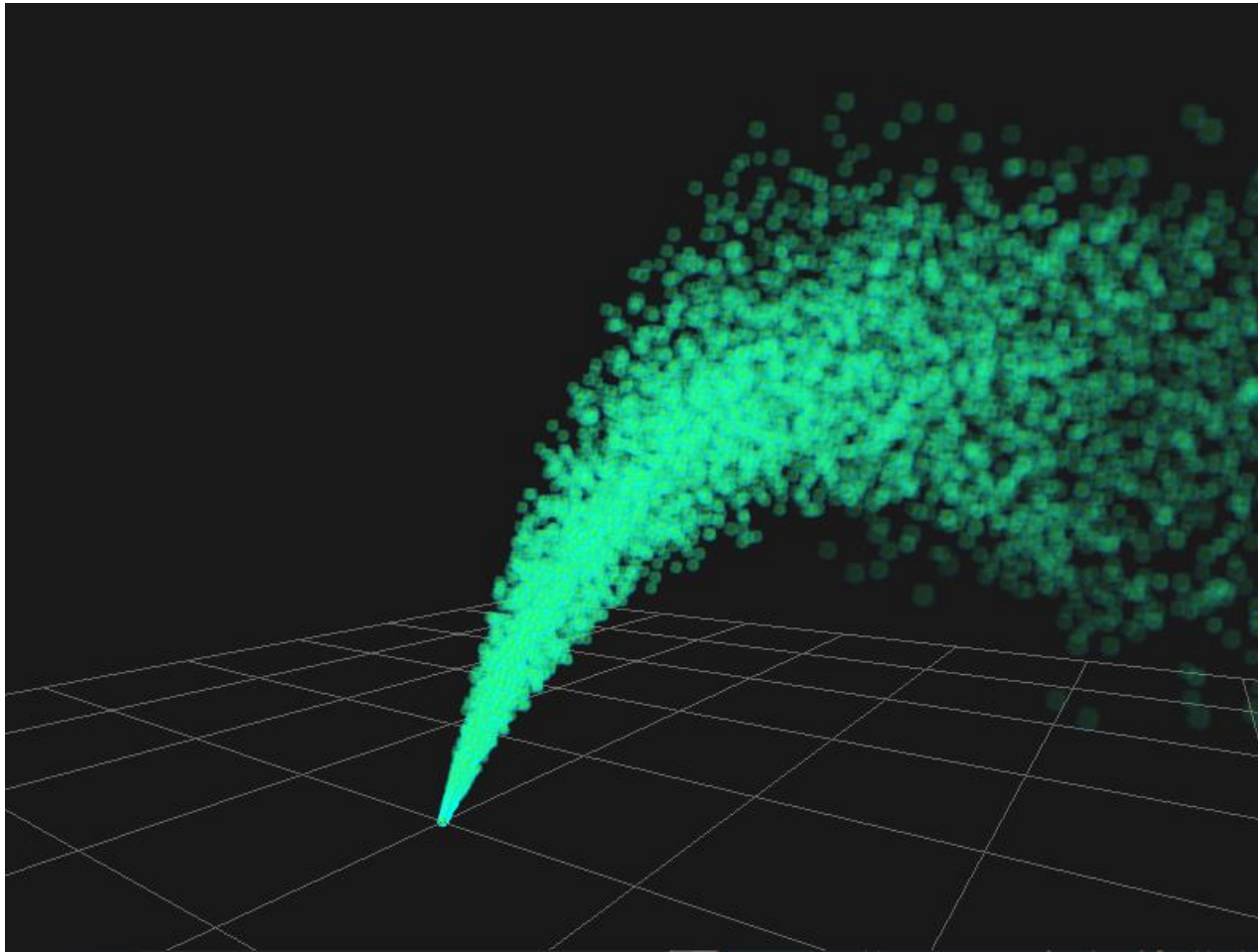
- Transform feedback allows us to define a set of shader output variables that are to be written to a designated buffer (or set of buffers).
- Just before the shader program is linked, we define the relationship between buffers and shader output variables using the `glTransformFeedbackVaryings` function
- During rendering, we initiate a transform feedback pass. We bind the appropriate buffers to the transform feedback binding points. (If desired, we can disable rasterization so that the particles are not rendered.)
- We enable transform feedback using the `glBeginTransformFeedback` function and then draw the point primitives.
- The output from the vertex shader will be stored in the appropriate buffers. Then we disable transform feedback by calling `glEndTransformFeedback`.

Particles with feedback implementation

- Create and allocate **three pairs of buffers**. The **first pair** will be for the **particle positions**, the **second** for the **particle velocities**, and the **third** for the **age of each particle**. For clarity, we'll refer to the **first buffer in each pair** as the **A buffer**, and the **second** as the **B buffer**.
- Create **two vertex arrays**. The **first vertex array** should link the **A position buffer** with the first vertex attribute (**attribute index 0**), the **A velocity buffer** with **vertex attribute one**, and the **A age buffer** with **vertex attribute two**.
- The second vertex array should be set up in the same way using the B buffers. The handles to the two vertex arrays will be accessed via the GLuint array named `particleArray`.

Particles with feedback implementation

- Initialize the A buffers with appropriate initial values. All of the **positions** could be set to the **origin**, and the **velocities** and **start times** could be initialized in the same way as described in the previous particle fountain example. The **initial velocity buffer** could simply be a copy of the **velocity buffer**.
- When using transform feedback, we define the buffers that will receive the output data from the vertex shader by binding the buffers to the indexed binding points under the `GL_TRANSFORM_FEEDBACK_BUFFER` target.
- The index corresponds to the index of the vertex shader's output variable as defined by `glTransformFeedbackVaryings`



Particles with
feedback

The background features a close-up of a copper mesh with a grid of small circular holes on the left side. On the right, there are abstract geometric shapes in blue and grey, separated by white lines, creating a modern, technical feel.

Instanced meshes

particle system

Instanced meshes particle system

- To give more geometric detail to each particle in a particle system, we can draw entire meshes instead of single quads. Instanced rendering is a convenient and efficient way to draw several copies of a particular object.
- OpenGL provides support for instanced rendering through the functions `glDrawArraysInstanced` and `glDrawElementsInstanced`.
- We'll modify the particle system introduced in the previous example. Rather than drawing single quads, we'll render a more complex object in the place of each particle.
- We'll also add another attribute to control the rotation of each particle so that each can independently spin at a random rotational velocity.



Instanced meshes particle system

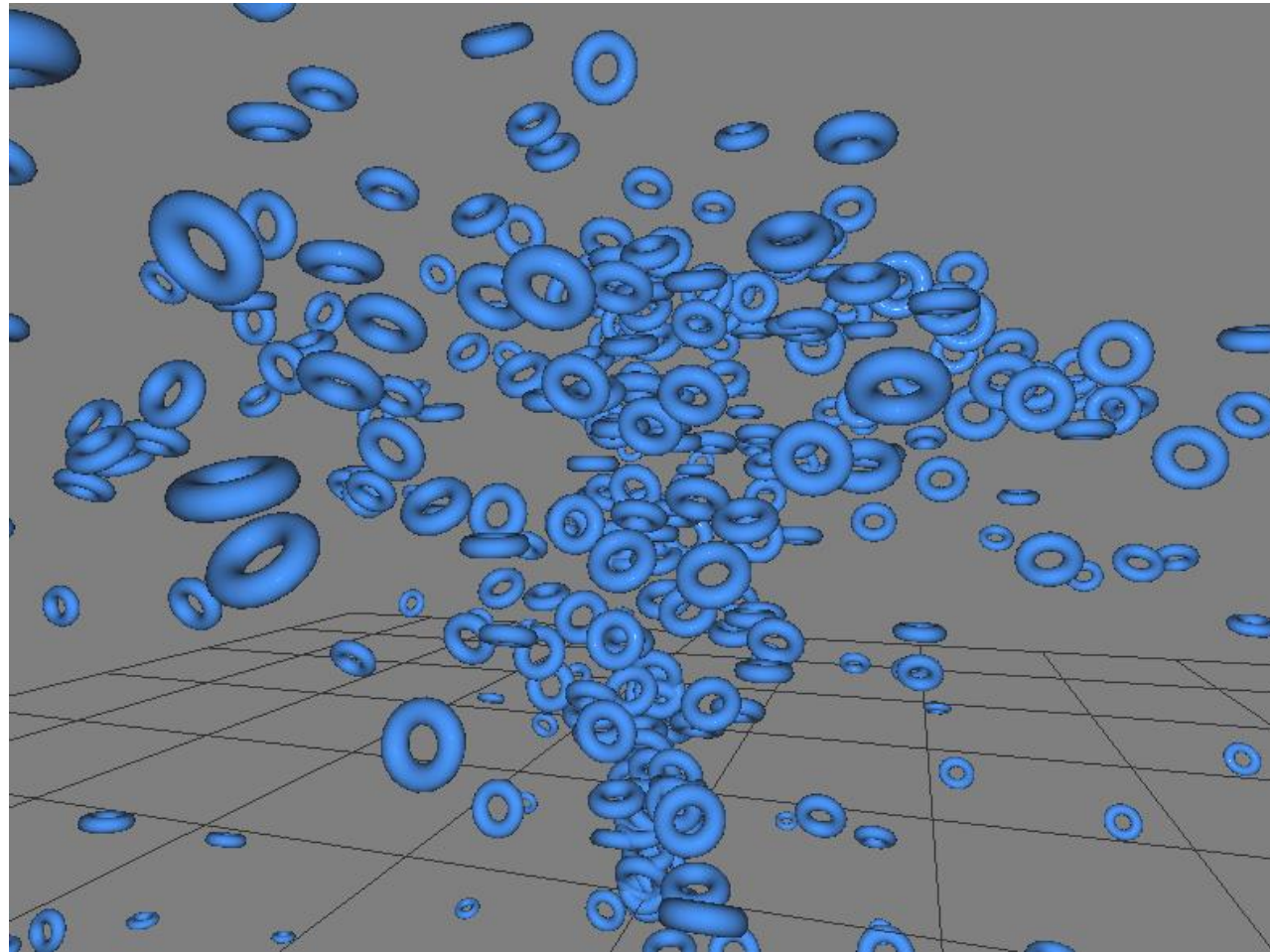
- Instead of three pairs of buffers, we'll use **four** this time. We'll need buffers for the **particle's position, velocity, age, and rotation**. The rotation buffer will store both the **rotational velocity** and the **angle of rotation** using the vec2 type.
- The x component will be the **rotational velocity** and the **y component** will be the **angle**. All shapes will rotate around the same axis.
- Set up the other buffers in the same way as in the previous recipe.

Instanced meshes particle system

- Because we are drawing full meshes, we need attributes for the **position** and **normal** of each vertex of the mesh.
- These attributes will have a **divisor of zero**, while the per-particle attributes will have a **divisor of one**. During the update pass, we will ignore the mesh vertex and normal attributes, focusing on the per-particle attributes. During the render pass, we'll use all attributes.
- We need pairs of buffers for the per-particle attributes, but we only need one buffer for our mesh data, so we'll share the mesh buffers with both vertex array objects.

Instanced meshes particle system

- The first two input attributes to the vertex shader are not-instanced, meaning that they are advanced every vertex (and repeated every instance).
- The last four are instanced attributes and only update every instance. Therefore, the effect is that all vertices of an instance of the mesh are transformed by the same matrix, ensuring that it acts as a single particle.
- **gl_InstanceID** is a counter and takes on a different value for each instance that is rendered. The first instance will have an ID of zero, the second will have an ID of one, and so on.
- This can be useful as a way to index to texture data appropriate for each instance. Another possibility is to use the instance's ID as a way to generate some random data for that instance.



Instanced
meshes
particle
system



Simulating fire

with particles



Simulating fire

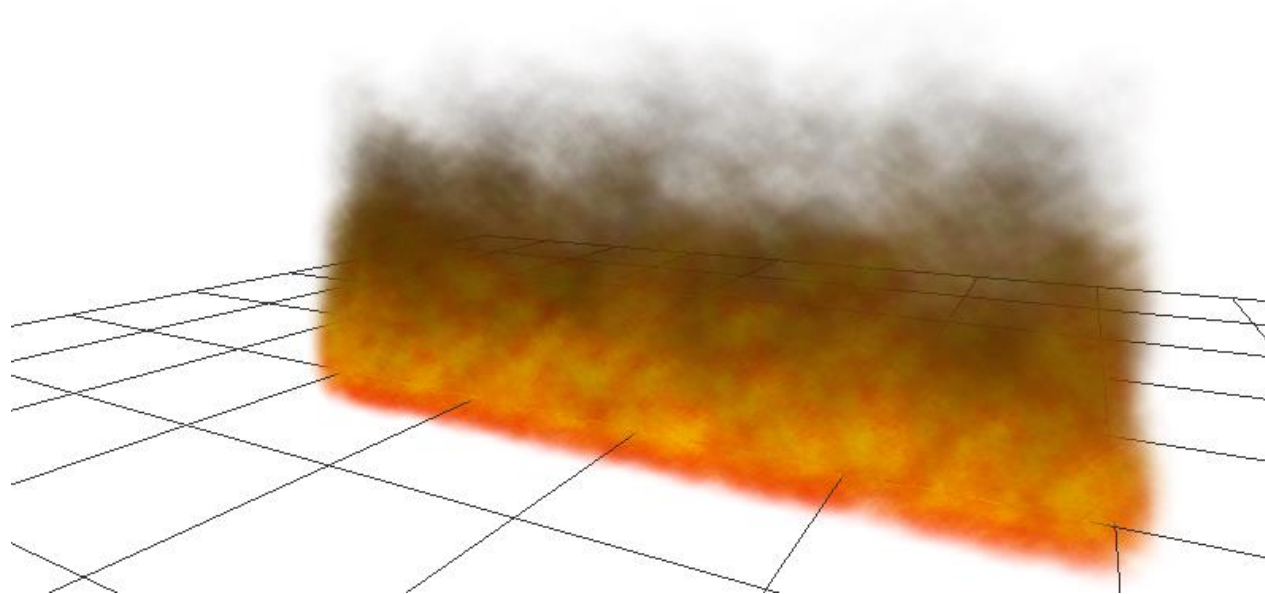
- We'll use the particles system with transform feedback as the base for this shader.
- Since fire is a substance that is only slightly affected by gravity, we don't worry about a downward gravitational acceleration. In fact, we'll actually use a slight upwards acceleration to make the particles spread out near the top of the flame.
- We'll also spread out the initial positions of the particles so that the base of the flame is not just a single point. We'll need to use a particle texture that has the red and orange colours associated with flame.

Simulating fire

- We'll use a texture filled with random values (two values per particle). The first value will be used to generate the initial velocity and the second, the initial position.
- For the initial positions, instead of using the emitter position for all particles, we offset that with random x location.
- When generating the initial velocities, we'll set the x and z components to zero and take the y component from the random texture.

Simulating fire

- We randomly distribute the **x** coordinate of the initial positions between -2.0 and 2.0 for all of the particles, and set the initial velocities to have a **y** coordinate between 0.1 and 0.5.
- Since the acceleration has only a **y** component, the particles will move only along a straight, vertical line in the **y** direction.
- The **x** or **z** component of the position should always remain at zero. This way, when recycling the particles, we can simply just reset the **y** coordinate to zero, to restart the particle at its initial position.



Simulating
fire

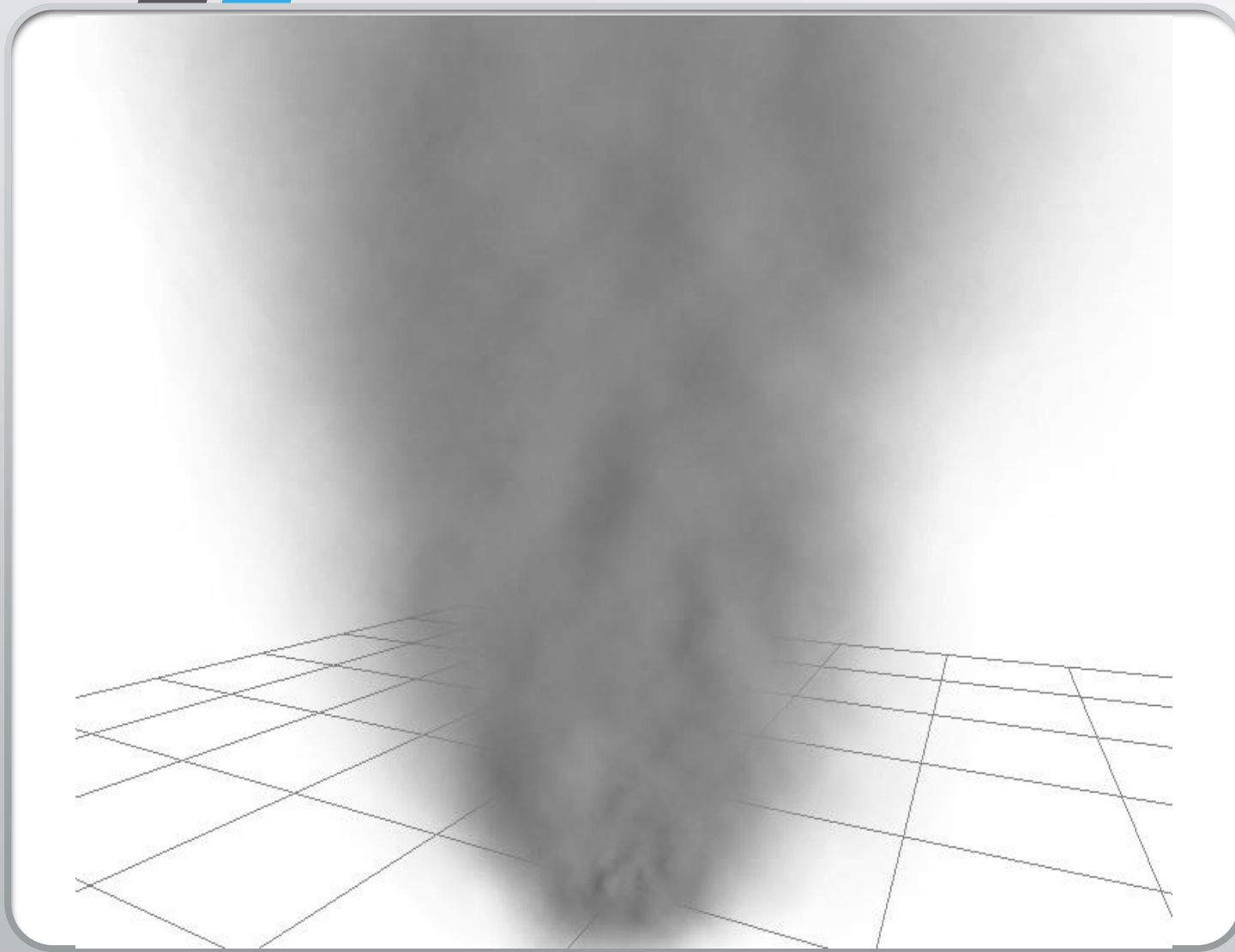


Simulating smoke

with particles

Simulating smoke

- Smoke is characterized by many small particles that float away from the source, and spread out as they move through the air.
- We can simulate the floatation effect with particles by using a small upwards acceleration (or constant velocity), but simulating the diffusion of each small smoke particle might be too expensive.
- Instead, we can simulate the diffusion of many small particles by making our simulated particles change their size (grow) over time.



Simulating
smoke

Useful links

- Wave equation explanation, Khan academy:
<https://www.khanacademy.org/science/physics/mechanical-waves-and-sound/mechanical-waves/v/wave-equation>
- To read - Chapter 11 Geometry shaders (OpenGL Superbible – see link on the DLE)
- To read – Geometry shaders: <https://learnopengl.com/Advanced-OpenGL/Geometry-Shader>
- To read: Particle systems and animation (OpenGL 4 Shading Language Cookbook).
- glDrawArraysInstanced: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArraysInstanced.xhtml>
- glDrawArrays: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArrays.xhtml>
- Spherical coordinates system:
https://en.wikipedia.org/wiki/Spherical_coordinate_system
- glVertexAttribDivisor: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribDivisor.xhtml>
- Transform feedback: https://www.khronos.org/opengl/wiki/Transform_Feedback
- glDrawElementsInstanced: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawElementsInstanced.xhtml>
- gl_InstanceID: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/gl_InstanceID.xhtml