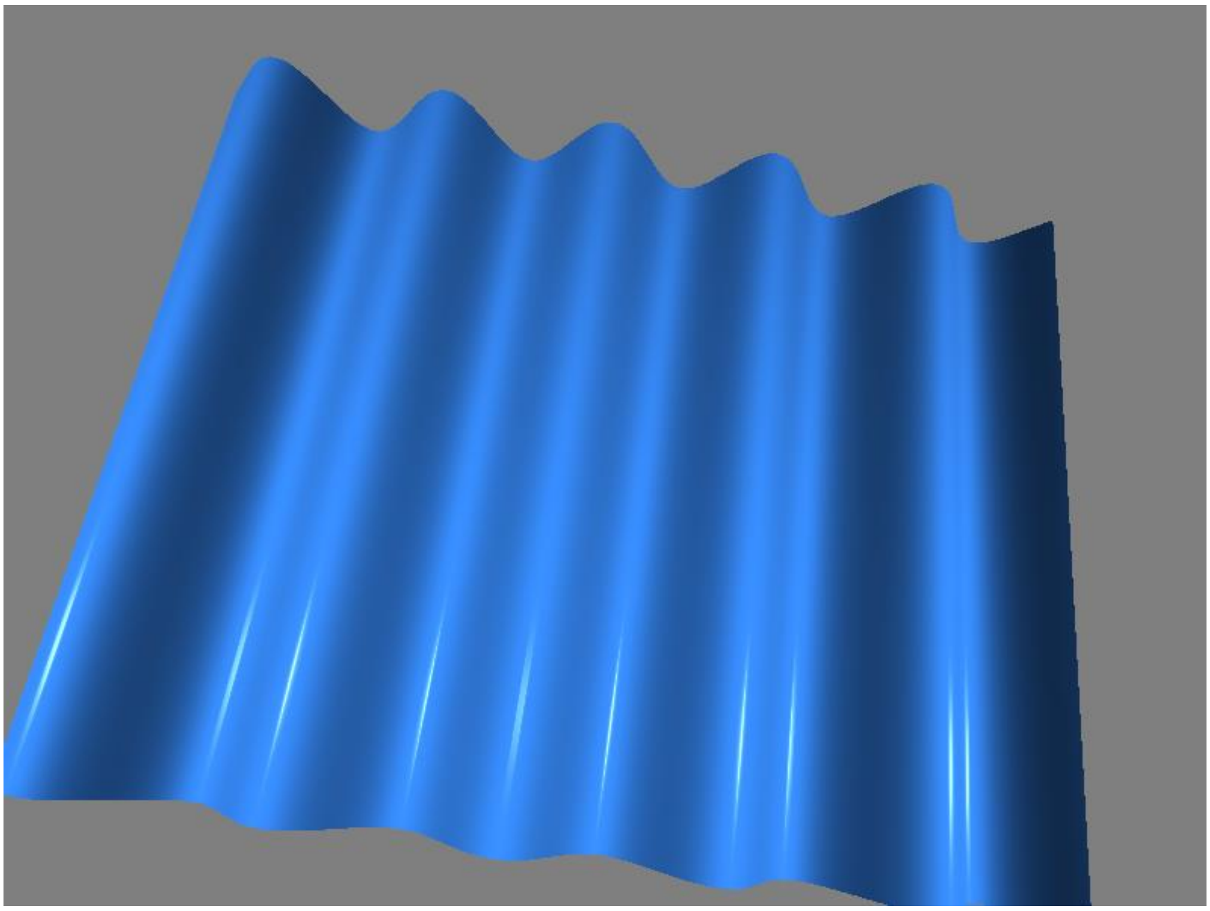
Surface animation

The end result:

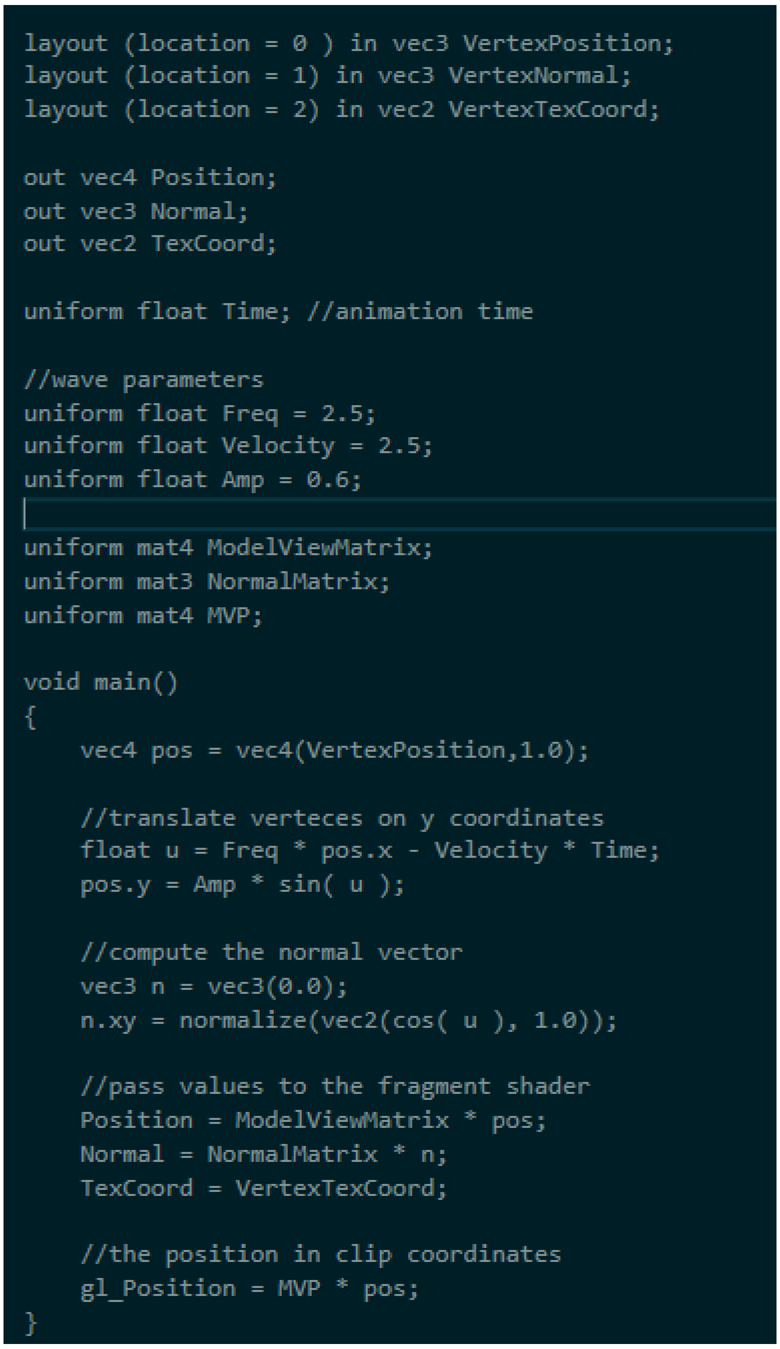


Vertex shader:

The vertex shader takes the position of the vertex and updates the y coordinate using the wave equation discussed in the lecture.

Modifying the vertex position within the vertex shader is a straightforward way to offload some computation from the CPU to the GPU. It also eliminates the possible need to transfer vertex buffers between the GPU memory and main memory in order to modify the positions.

The main disadvantage is that the updated positions are not available on the CPU side. For example, they might be needed for additional processing (such as collision detection).

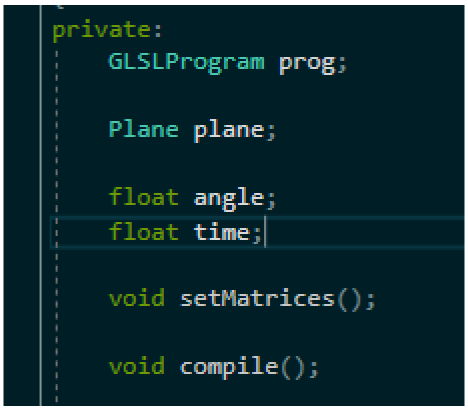


```
layout (location = 0 ) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;

out vec4 Position;
out vec3 Normal;
out vec2 TexCoord;

uniform float Time; //animation time

//wave parameters
uniform float Freq = 2.5;
uniform float Velocity = 2.5;
uniform float Amp = 0.6;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

void main()
{
    vec4 pos = vec4(VertexPosition,1.0);

    //translate verteces on y coordinates
    float u = Freq * pos.x - Velocity * Time;
    pos.y = Amp * sin( u );

    //compute the normal vector
    vec3 n = vec3(0.0);
    n.xy = normalize(vec2(cos( u ), 1.0));

    //pass values to the fragment shader
    Position = ModelViewMatrix * pos;
    Normal = NormalMatrix * n;
    TexCoord = VertexTexCoord;

    //the position in clip coordinates
    gl_Position = MVP * pos;
}
```

Fragment shader:

Implement a simple phong or blinn-Phong shader.

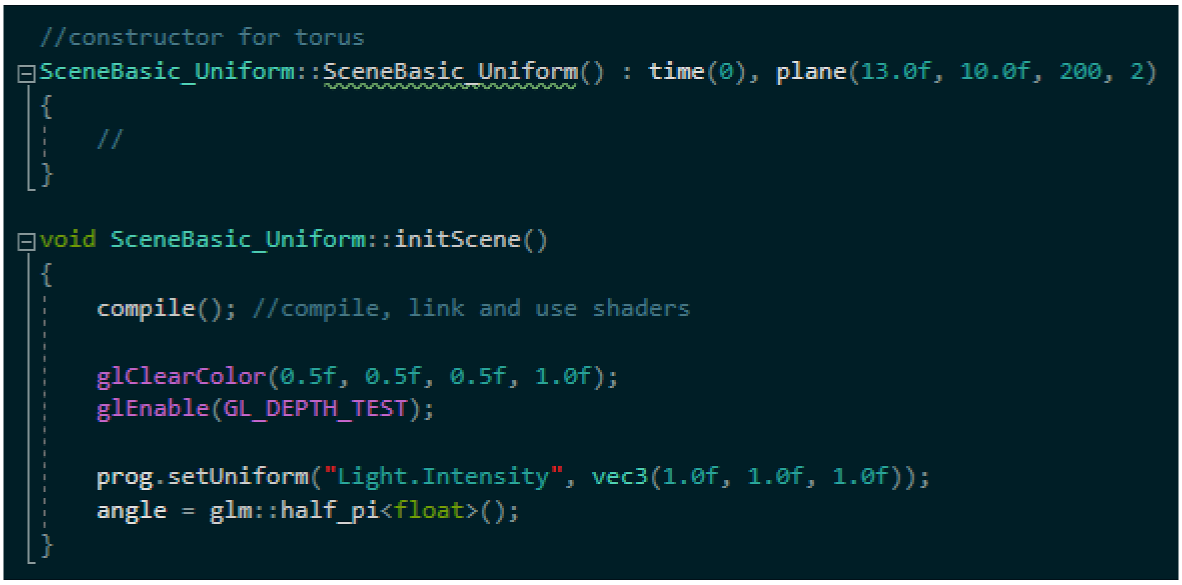
scenebasic_uniform.h:



```
private:
    GLSLProgram prog;

    Plane plane;

    float angle;
    float time;

    void setMatrices();

    void compile();
```

scenebasic_uniform.cpp:

Constructor and initScene() (pretty straight forward):

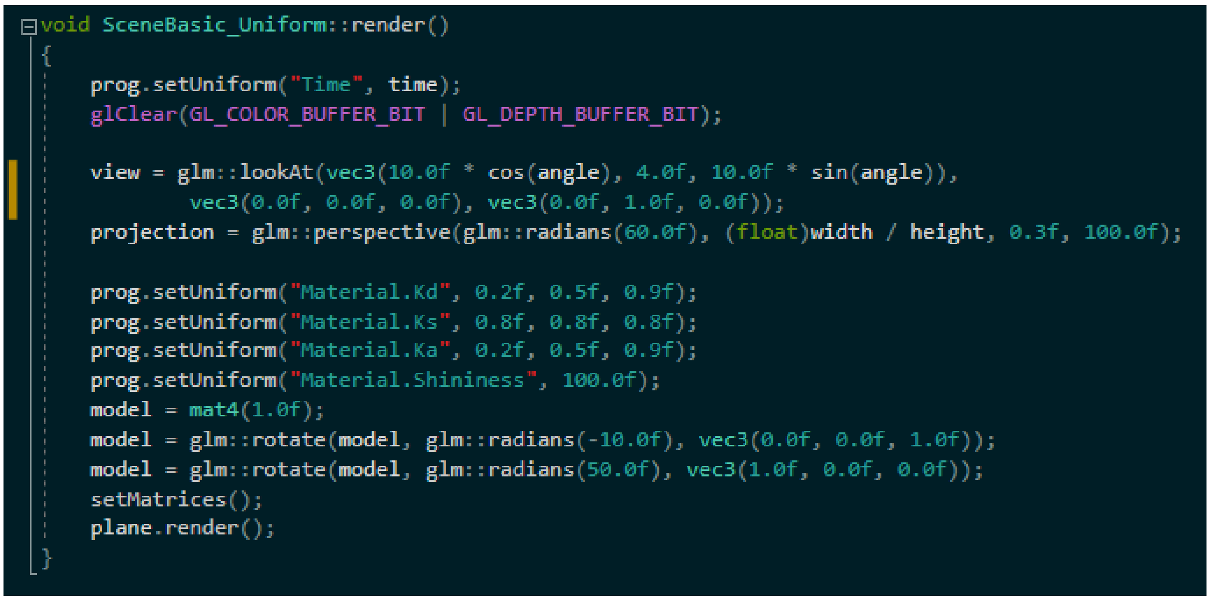


```
//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : time(0), plane(13.0f, 10.0f, 200, 2)
{
    //
}

void SceneBasic_Uniform::initScene()
{
    compile(); //compile, link and use shaders

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glEnable(GL_DEPTH_TEST);

    prog.setUniform("Light.Intensity", vec3(1.0f, 1.0f, 1.0f));
    angle = glm::half_pi<float>();
}
```

In the update() just pass value t to time variable:

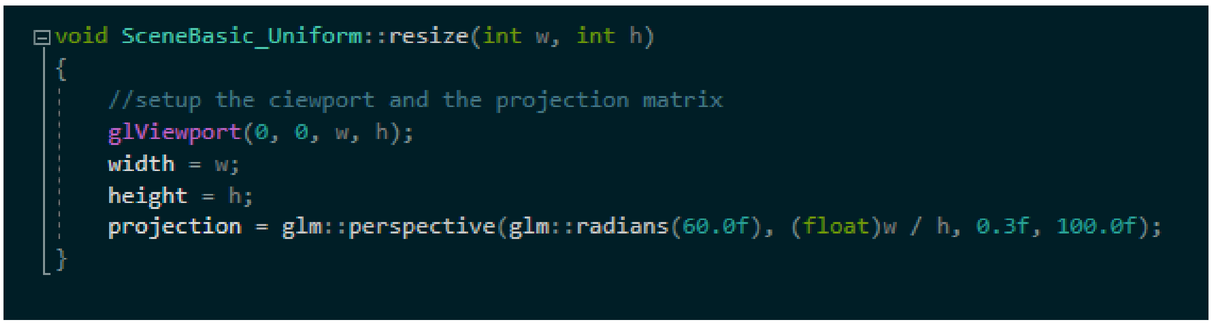
```
time = t;
```

In the render, do the following, if anything not clear, ask me:



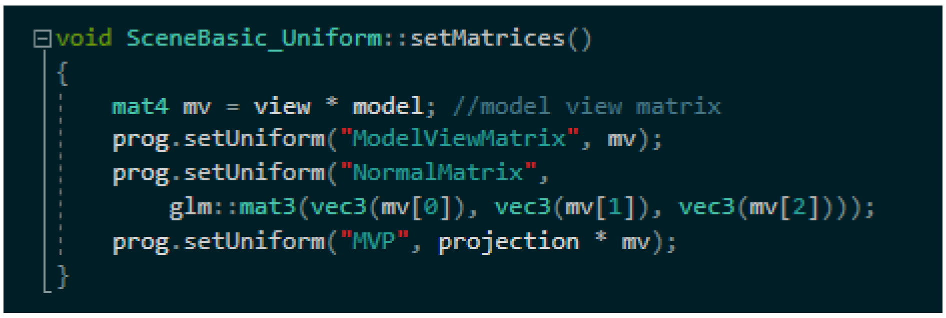
```cpp
void SceneBasic_Uniform::render()
{
    prog.setUniform("Time", time);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    view = glm::lookAt(vec3(10.0f * cos(angle), 4.0f, 10.0f * sin(angle)),
            vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f));
    projection = glm::perspective(glm::radians(60.0f), (float)width / height, 0.3f, 100.0f);

    prog.setUniform("Material.Kd", 0.2f, 0.5f, 0.9f);
    prog.setUniform("Material.Ks", 0.8f, 0.8f, 0.8f);
    prog.setUniform("Material.Ka", 0.2f, 0.5f, 0.9f);
    prog.setUniform("Material.Shininess", 100.0f);
    model = mat4(1.0f);
    model = glm::rotate(model, glm::radians(-10.0f), vec3(0.0f, 0.0f, 1.0f));
    model = glm::rotate(model, glm::radians(50.0f), vec3(1.0f, 0.0f, 0.0f));
    setMatrices();
    plane.render();
}
```

For setMatrices(), just pass to the shader the ModelViewMatrix, NormalMatrix and MVP:



```cpp
void SceneBasic_Uniform::resize(int w, int h)
{
    //setup the ciewport and the projection matrix
    glViewport(0, 0, w, h);
    width = w;
    height = h;
    projection = glm::perspective(glm::radians(60.0f), (float)w / h, 0.3f, 100.0f);
}
```

In the resize(), set ta projection matrix with a perspective of 60 radians, near of 0.3 and a far of 100:

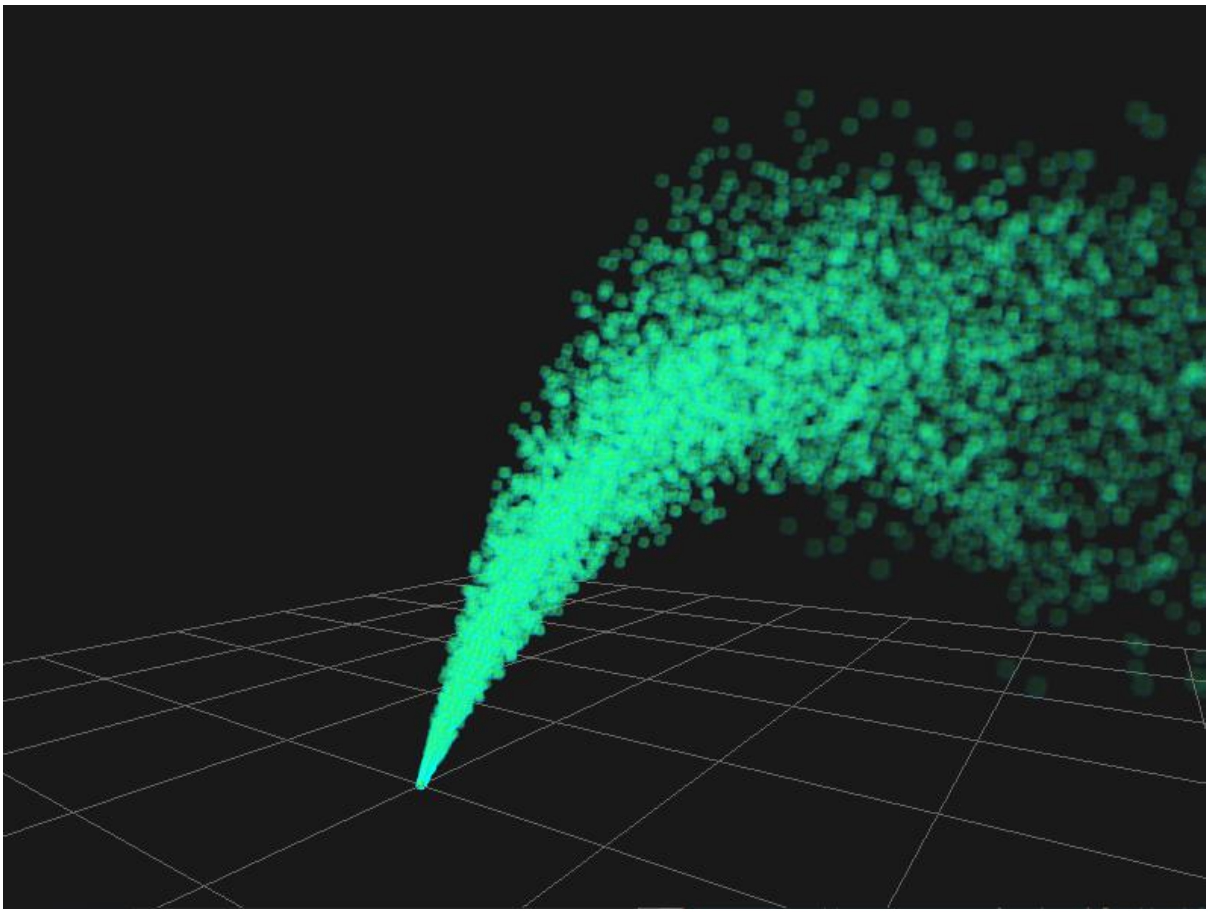


```cpp
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model; //model view matrix
    prog.setUniform("ModelViewMatrix", mv);
    prog.setUniform("NormalMatrix",
        glm::mat3(vec3(mv[0]), vec3(mv[1]), vec3(mv[2])));
    prog.setUniform("MVP", projection * mv);
}
```

That's it.

Particle fountain

The end result:



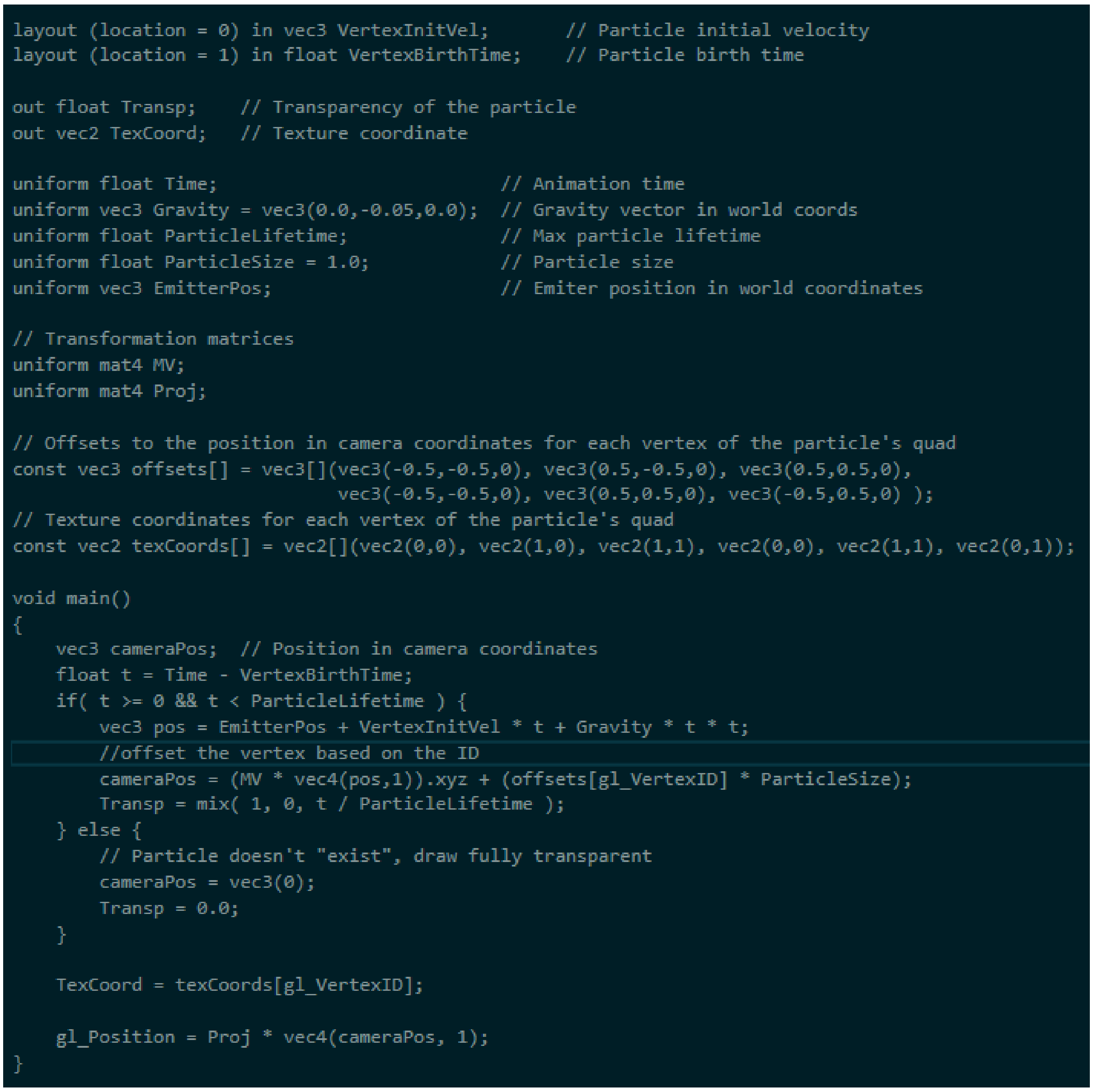
We'll define the initial position of all particles to be the origin (0,0,0). The initial velocity will be determined randomly within a range of values. Each particle will be created at a slightly different time, so the time that we use in the previous equation will be relative to the start time for the 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

To render our particles, we'll use a technique called instancing, along with a simple trick to generate screen-aligned quads. With this technique, 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.

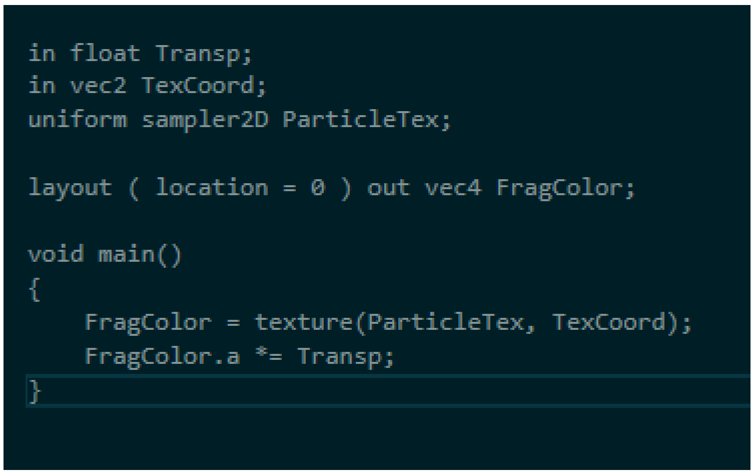
Vertex shader:

Use this vertex shader:



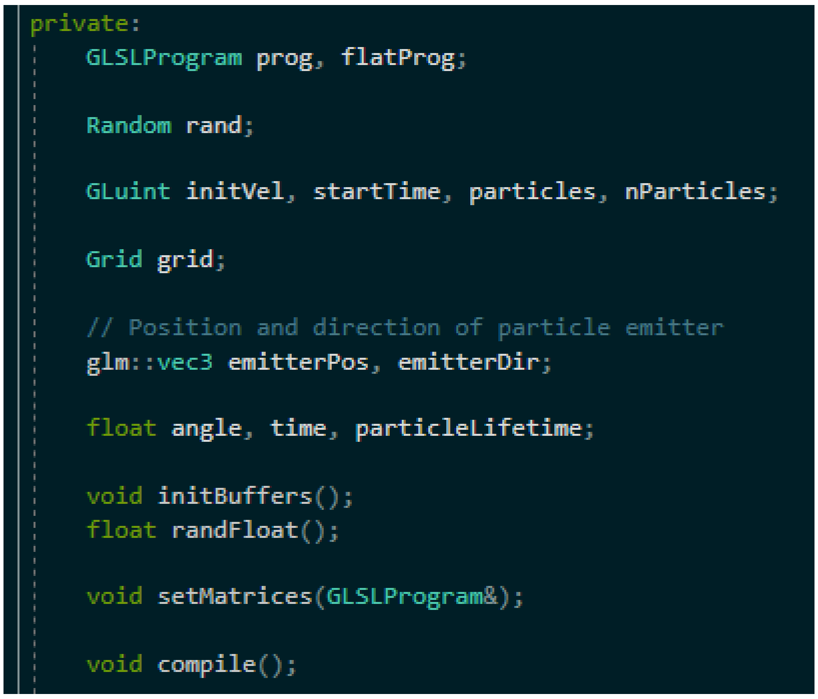
```
layout (location = 0) in vec3 VertexInitVel;       // Particle initial velocity
layout (location = 1) in float VertexBirthTime;    // Particle birth time

out float Transp;    // Transparency of the particle
out vec2 TexCoord;   // Texture coordinate

uniform float Time;                             // Animation time
uniform vec3 Gravity = vec3(0.0,-0.05,0.0);  // Gravity vector in world coords
uniform float ParticleLifetime;                 // Max particle lifetime
uniform float ParticleSize = 1.0;               // Particle size
uniform vec3 EmitterPos;                        // Emiter position in world coordinates

// Transformation matrices
uniform mat4 MV;
uniform mat4 Proj;

// Offsets to the position in camera coordinates for each vertex of the particle's quad
const vec3 offsets[] = vec3[](vec3(-0.5,-0.5,0), vec3(0.5,-0.5,0), vec3(0.5,0.5,0),
                              vec3(-0.5,-0.5,0), vec3(0.5,0.5,0), vec3(-0.5,0.5,0) );
// Texture coordinates for each vertex of the particle's quad
const vec2 texCoords[] = vec2[](vec2(0,0), vec2(1,0), vec2(1,1), vec2(0,0), vec2(1,1), vec2(0,1));

void main()
{
    vec3 cameraPos;  // Position in camera coordinates
    float t = Time - VertexBirthTime;
    if( t >= 0 && t < ParticleLifetime ) {
        vec3 pos = EmitterPos + VertexInitVel * t + Gravity * t * t;
        //offset the vertex based on the ID
        cameraPos = (MV * vec4(pos,1)).xyz + (offsets[gl_VertexID] * ParticleSize);
        Transp = mix( 1, 0, t / ParticleLifetime );
    } else {
        // Particle doesn't "exist", draw fully transparent
        cameraPos = vec3(0);
        Transp = 0.0;
    }

    TexCoord = texCoords[gl_VertexID];

    gl_Position = Proj * vec4(cameraPos, 1);
}
```

Fragment shader:



```
in float Transp;
in vec2 TexCoord;
uniform sampler2D ParticleTex;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
    FragColor = texture(ParticleTex, TexCoord);
    FragColor.a *= Transp;
}
```

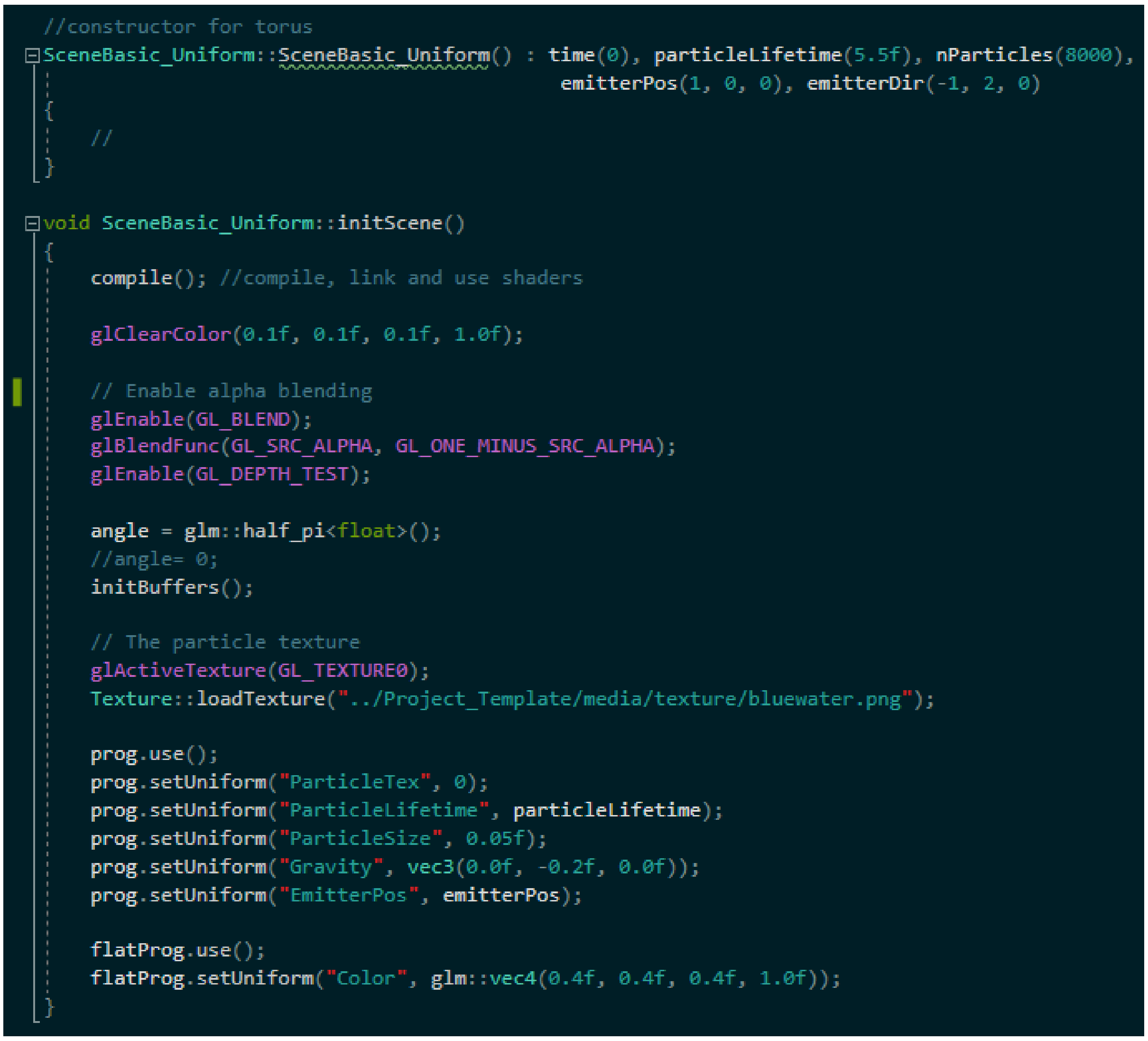
scenebasic_uniform.h:



```
private:
    GLSLProgram prog, flatProg;

    Random rand;

    GLuint initVel, startTime, particles, nParticles;

    Grid grid;

    // Position and direction of particle emitter
    glm::vec3 emitterPos, emitterDir;

    float angle, time, particleLifetime;

    void initBuffers();
    float randFloat();

    void setMatrices(GLSLProgram&);

    void compile();
```

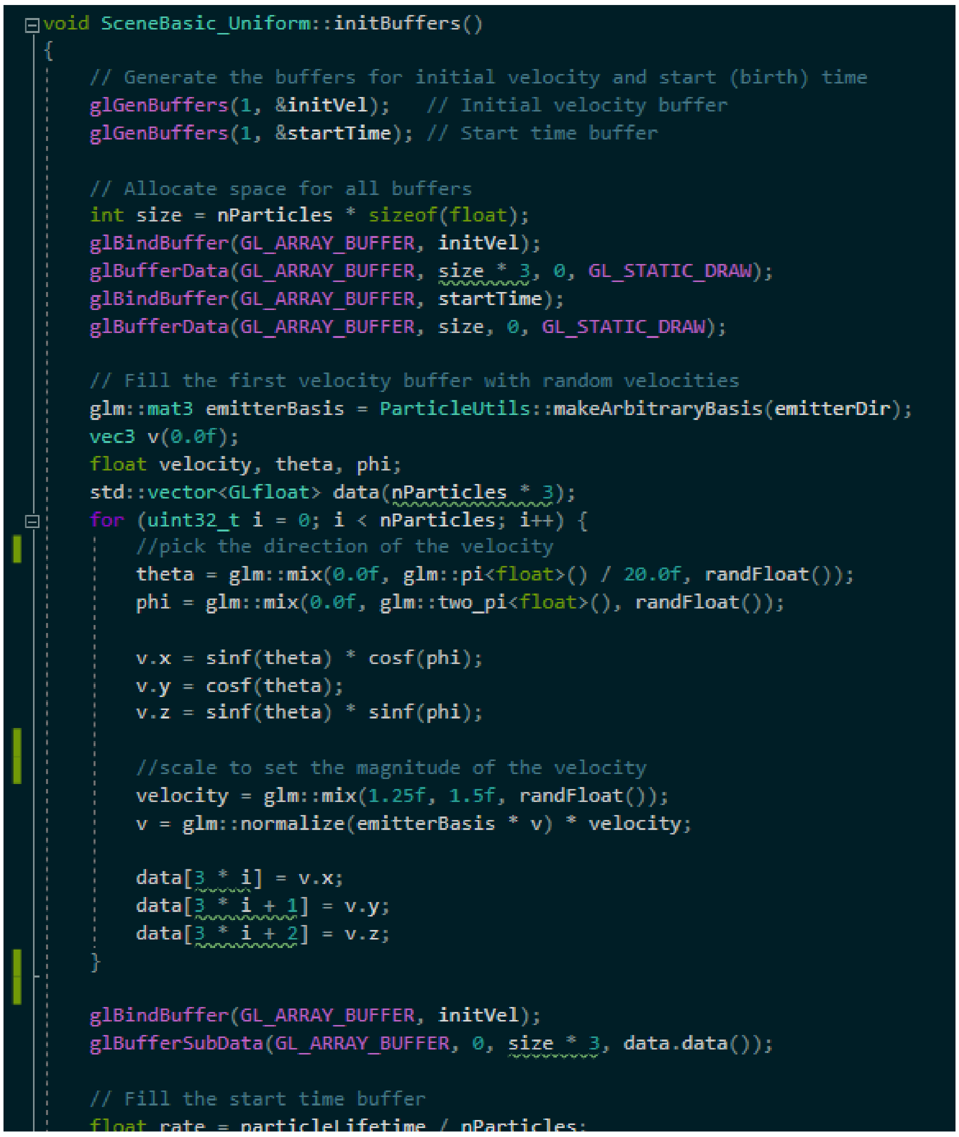
scenebasic_uniform.cpp:

Constructor and the initScene():

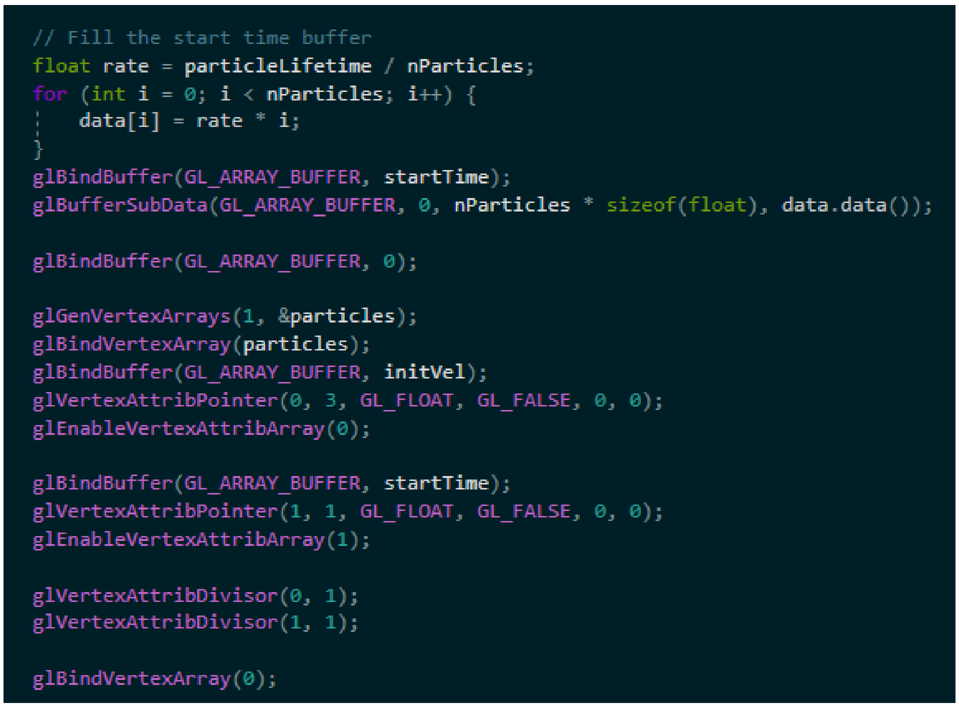


```
//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : time(0), particleLifetime(5.5f), nParticles(8000),
                                           emitterPos(1, 0, 0), emitterDir(-1, 2, 0)
{
    //
}

void SceneBasic_Uniform::initScene()
{
    compile(); //compile, link and use shaders

    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

    // Enable alpha blending
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_DEPTH_TEST);

    angle = glm::half_pi<float>();
    //angle= 0;
    initBuffers();

    // The particle texture
    glActiveTexture(GL_TEXTURE0);
    Texture::loadTexture("../Project_Template/media/texture/bluewater.png");

    prog.use();
    prog.setUniform("ParticleTex", 0);
    prog.setUniform("ParticleLifetime", particleLifetime);
    prog.setUniform("ParticleSize", 0.05f);
    prog.setUniform("Gravity", vec3(0.0f, -0.2f, 0.0f));
    prog.setUniform("EmitterPos", emitterPos);

    flatProg.use();
    flatProg.setUniform("Color", glm::vec4(0.4f, 0.4f, 0.4f, 1.0f));
}
```

Use this for initiating the buffers:

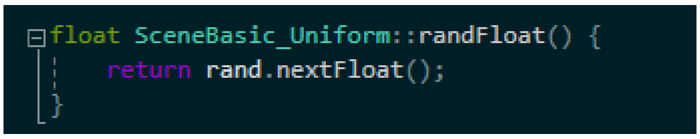


```cpp
void SceneBasic_Uniform::initBuffers()
{
    // Generate the buffers for initial velocity and start (birth) time
    glGenBuffers(1, &initVel);   // Initial velocity buffer
    glGenBuffers(1, &startTime); // Start time buffer

    // Allocate space for all buffers
    int size = nParticles * sizeof(float);
    glBindBuffer(GL_ARRAY_BUFFER, initVel);
    glBufferData(GL_ARRAY_BUFFER, size * 3, 0, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, startTime);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_STATIC_DRAW);

    // Fill the first velocity buffer with random velocities
    glm::mat3 emitterBasis = ParticleUtils::makeArbitraryBasis(emitterDir);
    vec3 v(0.0f);
    float velocity, theta, phi;
    std::vector<GLfloat> data(nParticles * 3);
    for (uint32_t i = 0; i < nParticles; i++) {
        //pick the direction of the velocity
        theta = glm::mix(0.0f, glm::pi<float>() / 20.0f, randFloat());
        phi = glm::mix(0.0f, glm::two_pi<float>(), randFloat());

        v.x = sinf(theta) * cosf(phi);
        v.y = cosf(theta);
        v.z = sinf(theta) * sinf(phi);

        //scale to set the magnitude of the velocity
        velocity = glm::mix(1.25f, 1.5f, randFloat());
        v = glm::normalize(emitterBasis * v) * velocity;

        data[3 * i] = v.x;
        data[3 * i + 1] = v.y;
        data[3 * i + 2] = v.z;
    }

    glBindBuffer(GL_ARRAY_BUFFER, initVel);
    glBufferSubData(GL_ARRAY_BUFFER, 0, size * 3, data.data());

    // Fill the start time buffer
    float rate = particleLifetime / nParticles;
```



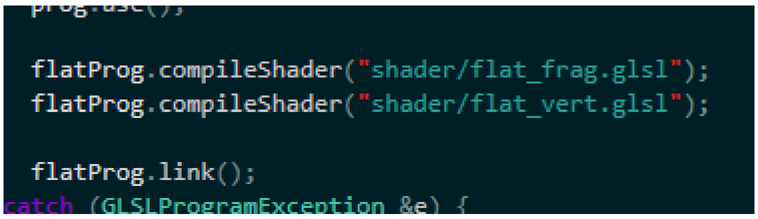
```cpp
// Fill the start time buffer
float rate = particleLifetime / nParticles;
for (int i = 0; i < nParticles; i++) {
    data[i] = rate * i;
}
glBindBuffer(GL_ARRAY_BUFFER, startTime);
glBufferSubData(GL_ARRAY_BUFFER, 0, nParticles * sizeof(float), data.data());

glBindBuffer(GL_ARRAY_BUFFER, 0);

glGenVertexArrays(1, &particles);
glBindVertexArray(particles);
glBindBuffer(GL_ARRAY_BUFFER, initVel);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, startTime);
glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

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

glBindVertexArray(0);
```

Use this to generate random floats:



```
float SceneBasic_Uniform::randFloat() {
    return rand.nextFloat();
}
```

In the compile() make sure you add the “flat” shaders (vertex and fragment). Add the shaders after you call prog.use() and before catch…



```
  flatProg.compileShader("shader/flat_frag.glsl");
  flatProg.compileShader("shader/flat_vert.glsl");

  flatProg.link();
catch (GLSLProgramException &e) {
```

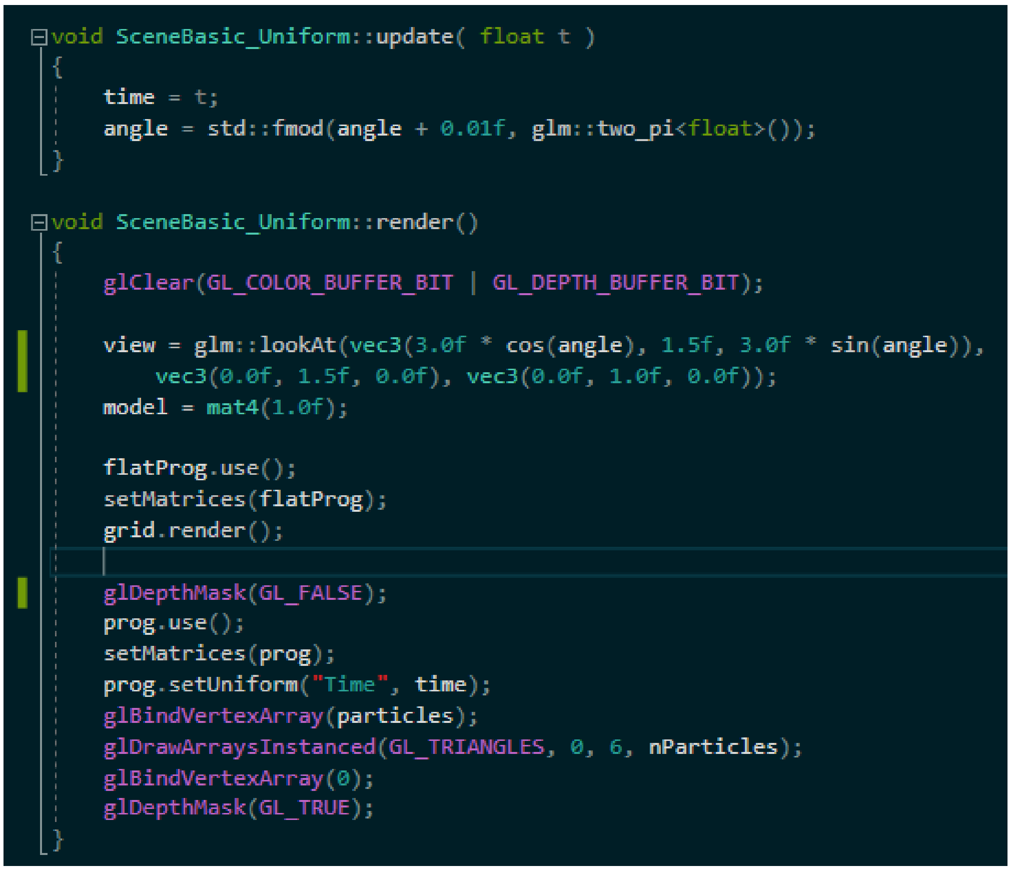
flat_vert shader just passes VertexPosition to gl_Position.

```
gl_Position = Proj * MV * vec4(VertexPosition, 1.0);
```

flat_frag shader just passes a colour uniform of type vec4 to FragColour

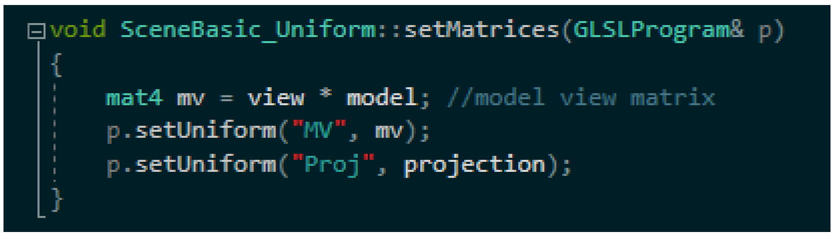
```
FragColor = Color;
```

For update() and render() use this code:



```
void SceneBasic_Uniform::update( float t )
{
    time = t;
    angle = std::fmod(angle + 0.01f, glm::two_pi<float>());
}

void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    view = glm::lookAt(vec3(3.0f * cos(angle), 1.5f, 3.0f * sin(angle)),
        vec3(0.0f, 1.5f, 0.0f), vec3(0.0f, 1.0f, 0.0f));
    model = mat4(1.0f);

    flatProg.use();
    setMatrices(flatProg);
    grid.render();

    glDepthMask(GL_FALSE);
    prog.use();
    setMatrices(prog);
    prog.setUniform("Time", time);
    glBindVertexArray(particles);
    glDrawArraysInstanced(GL_TRIANGLES, 0, 6, nParticles);
    glBindVertexArray(0);
    glDepthMask(GL_TRUE);
}
```

Use this code for setmatrices():



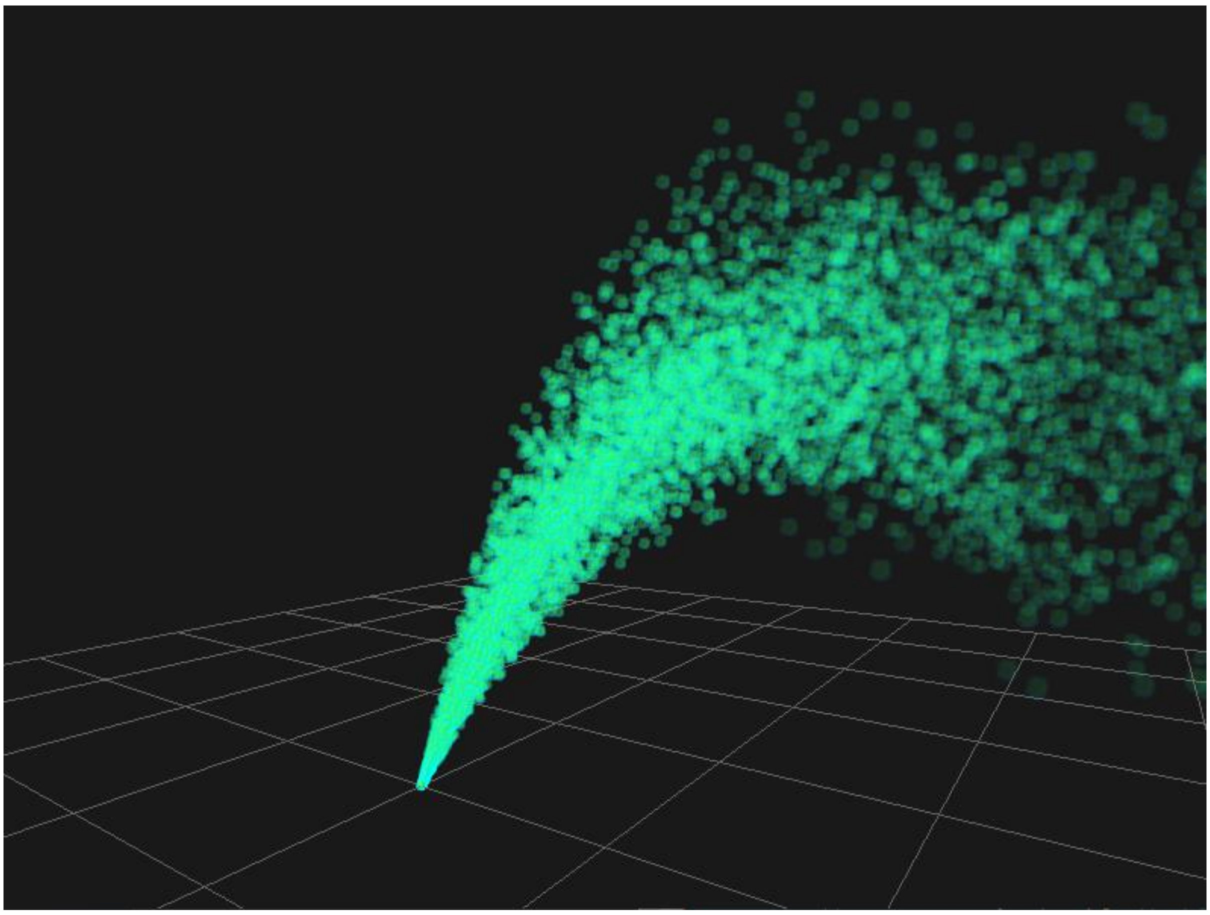
```
void SceneBasic_Uniform::setMatrices(GLSLProgram& p)
{
    mat4 mv = view * model; //model view matrix
    p.setUniform("MV", mv);
    p.setUniform("Proj", projection);
}
```

For resize() just the code from surface animation example done earlier.

That’s it.

Particle fountain with feedback

The end result:

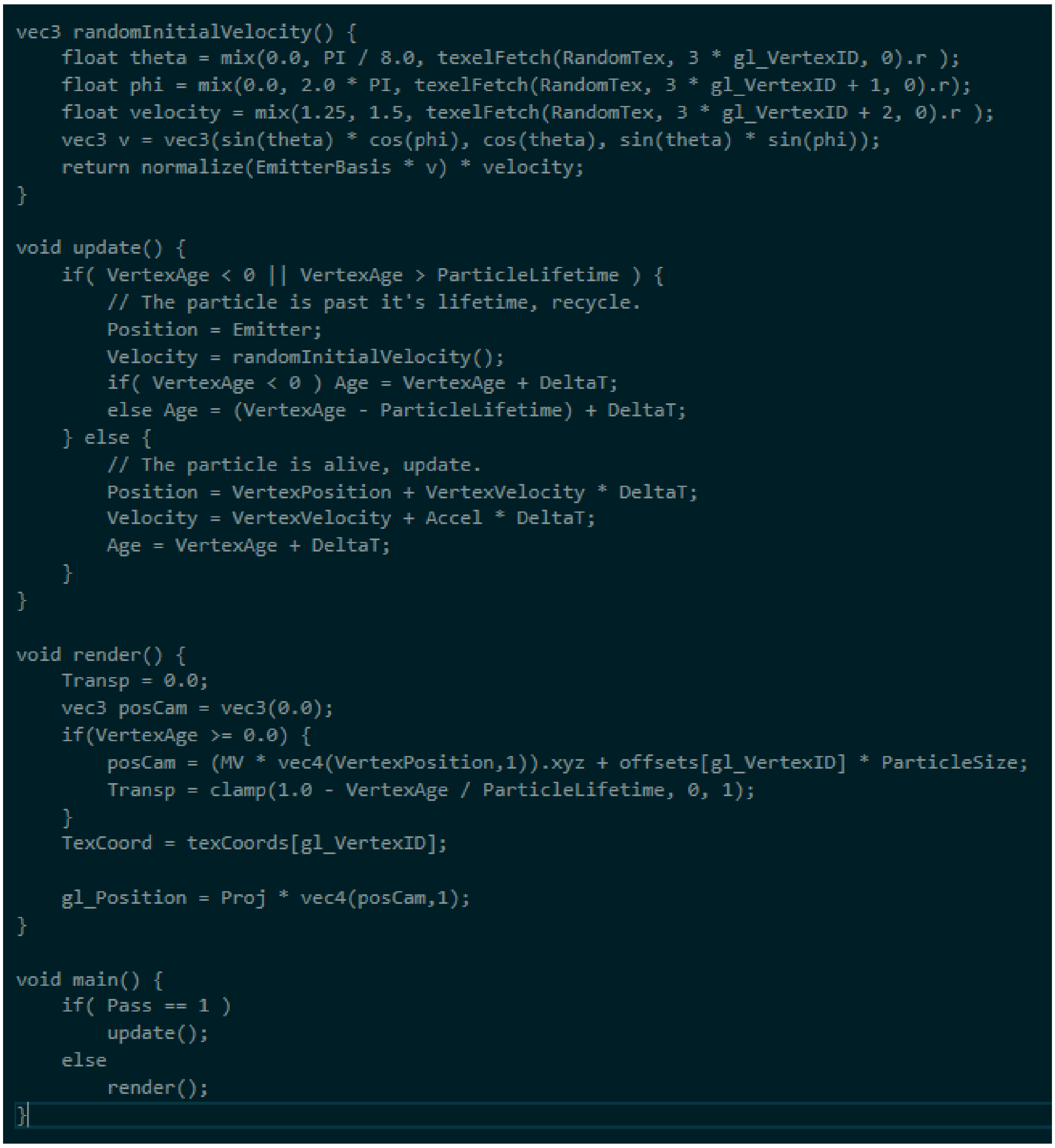


Vertex shader:

The vertex shader, we have code that supports two passes: the update pass where the particles' position, age, and velocity are updated, and the render pass where the particles are drawn. Here are the variables used in the shader:

```
const float PI = 3.14159265359;

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexVelocity;
layout (location = 2) in float VertexAge;

// Render pass
uniform int Pass;

// Output to transform feedback buffers Update pass
//   (Layout specifiers only available in OpenGL 4.4+)
/*layout( xfb_buffer = 0, xfb_offset=0 )*/ out vec3 Position;
/*layout( xfb_buffer = 1, xfb_offset=0 )*/ out vec3 Velocity;
/*layout( xfb_buffer = 2, xfb_offset=0 )*/ out float Age;

// Out to fragment shader
out float Transp;           // Transparency
out vec2 TexCoord;          // Texture coordinate

uniform float Time;                 // Simulation time
uniform float DeltaT;               // Elapsed time between frames
uniform vec3 Accel;                 // Particle acceleration (gravity)
uniform float ParticleLifetime;     // Particle lifespan
uniform vec3 Emitter = vec3(0);     // World position of the emitter.
uniform mat3 EmitterBasis;          // Rotation that rotates y axis to the direction of emitter
uniform float ParticleSize;         // Size of particle

uniform mat4 MV;    // View * Model
uniform mat4 Proj;  // Projection matrix

uniform sampler1D RandomTex;

// Offsets to the position in camera coordinates for each vertex of the particle's quad
const vec3 offsets[] = vec3[](vec3(-0.5,-0.5,0), vec3(0.5,-0.5,0), vec3(0.5,0.5,0),
                              vec3(-0.5,-0.5,0), vec3(0.5,0.5,0), vec3(-0.5,0.5,0) );
// Texture coordinates for each vertex of the particle's quad
const vec2 texCoords[] = vec2[](vec2(0,0), vec2(1,0), vec2(1,1), vec2(0,0), vec2(1,1), vec2(0,1));
```

And here is the rest of the shader:

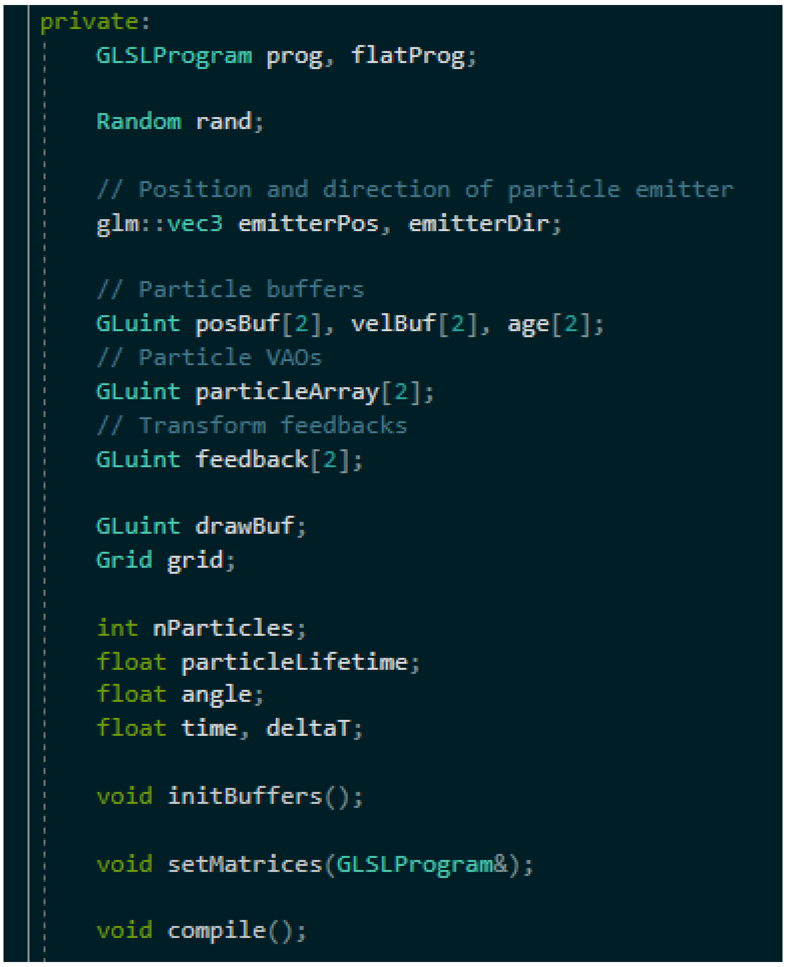


```
vec3 randomInitialVelocity() {
    float theta = mix(0.0, PI / 8.0, texelFetch(RandomTex, 3 * gl_VertexID, 0).r );
    float phi = mix(0.0, 2.0 * PI, texelFetch(RandomTex, 3 * gl_VertexID + 1, 0).r);
    float velocity = mix(1.25, 1.5, texelFetch(RandomTex, 3 * gl_VertexID + 2, 0).r );
    vec3 v = vec3(sin(theta) * cos(phi), cos(theta), sin(theta) * sin(phi));
    return normalize(EmitterBasis * v) * velocity;
}

void update() {
    if( VertexAge < 0 || VertexAge > ParticleLifetime ) {
        // The particle is past it's lifetime, recycle.
        Position = Emitter;
        Velocity = randomInitialVelocity();
        if( VertexAge < 0 ) Age = VertexAge + DeltaT;
        else Age = (VertexAge - ParticleLifetime) + DeltaT;
    } else {
        // The particle is alive, update.
        Position = VertexPosition + VertexVelocity * DeltaT;
        Velocity = VertexVelocity + Accel * DeltaT;
        Age = VertexAge + DeltaT;
    }
}

void render() {
    Transp = 0.0;
    vec3 posCam = vec3(0.0);
    if(VertexAge >= 0.0) {
        posCam = (MV * vec4(VertexPosition,1)).xyz + offsets[gl_VertexID] * ParticleSize;
        Transp = clamp(1.0 - VertexAge / ParticleLifetime, 0, 1);
    }
    TexCoord = texCoords[gl_VertexID];

    gl_Position = Proj * vec4(posCam,1);
}

void main() {
    if( Pass == 1 )
        update();
    else
        render();
}
```

Fragment shader:

Use the fragment shader from Particle fountain example.

scenebasic_uniform.h:



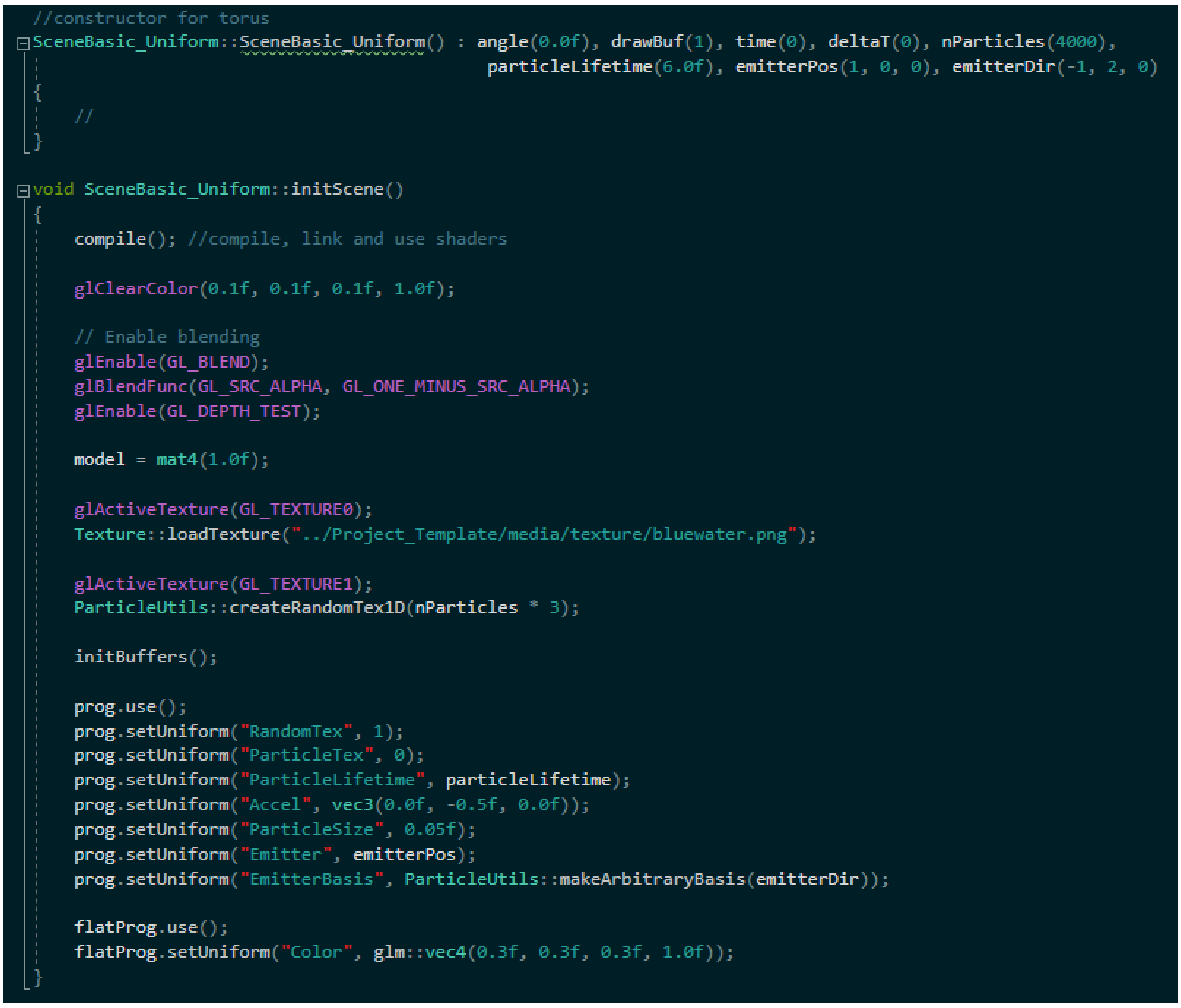
```
private:
    GLSLProgram prog, flatProg;

    Random rand;

    // Position and direction of particle emitter
    glm::vec3 emitterPos, emitterDir;

    // Particle buffers
    GLuint posBuf[2], velBuf[2], age[2];
    // Particle VAOs
    GLuint particleArray[2];
    // Transform feedbacks
    GLuint feedback[2];

    GLuint drawBuf;
    Grid grid;

    int nParticles;
    float particleLifetime;
    float angle;
    float time, deltaT;

    void initBuffers();

    void setMatrices(GLSLProgram&);

    void compile();
```

scenebasic_uniform.cpp:

Make sure you add particleutils.h to the project and you use it in here:

```
#include "helper/particleutils.h"
```

Use this for constructor and initScene():



```
//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : angle(0.0f), drawBuf(1), time(0), deltaT(0), nParticles(4000),
                                            particleLifetime(6.0f), emitterPos(1, 0, 0), emitterDir(-1, 2, 0)
{
    //
}

void SceneBasic_Uniform::initScene()
{
    compile(); //compile, link and use shaders

    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

    // Enable blending
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_DEPTH_TEST);

    model = mat4(1.0f);

    glActiveTexture(GL_TEXTURE0);
    Texture::loadTexture("../Project_Template/media/texture/bluewater.png");

    glActiveTexture(GL_TEXTURE1);
    ParticleUtils::createRandomTex1D(nParticles * 3);

    initBuffers();

    prog.use();
    prog.setUniform("RandomTex", 1);
    prog.setUniform("ParticleTex", 0);
    prog.setUniform("ParticleLifetime", particleLifetime);
    prog.setUniform("Accel", vec3(0.0f, -0.5f, 0.0f));
    prog.setUniform("ParticleSize", 0.05f);
    prog.setUniform("Emitter", emitterPos);
    prog.setUniform("EmitterBasis", ParticleUtils::makeArbitraryBasis(emitterDir));

    flatProg.use();
    flatProg.setUniform("Color", glm::vec4(0.3f, 0.3f, 0.3f, 1.0f));
}
```

Next we 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.

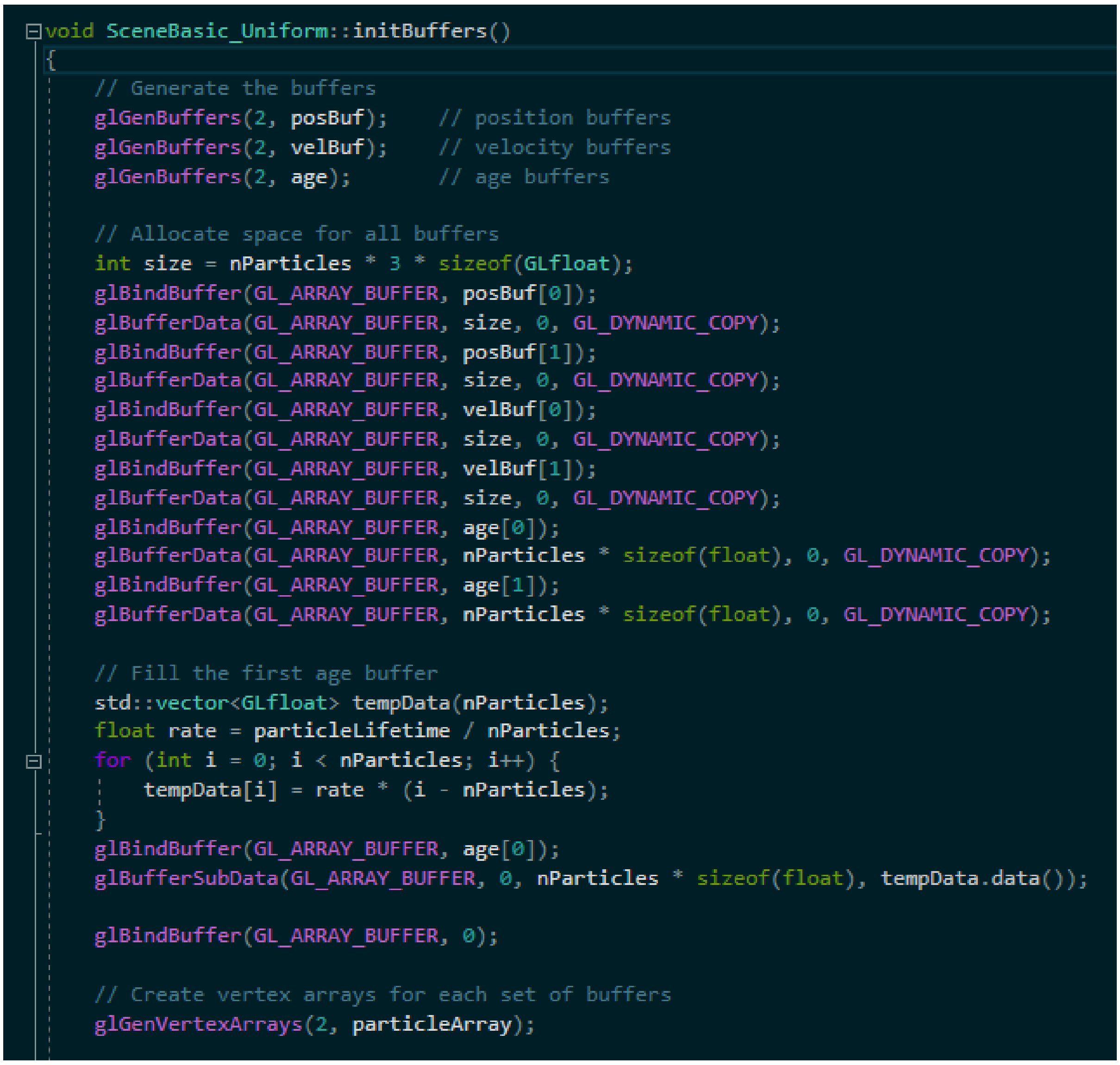
Create two vertex arrays. The first vertex array should link the A position buffer with the first vertex attribute (attribute index 0), the A velocity buffer with vertex attribute one, and the A age buffer with vertex attribute two.

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.

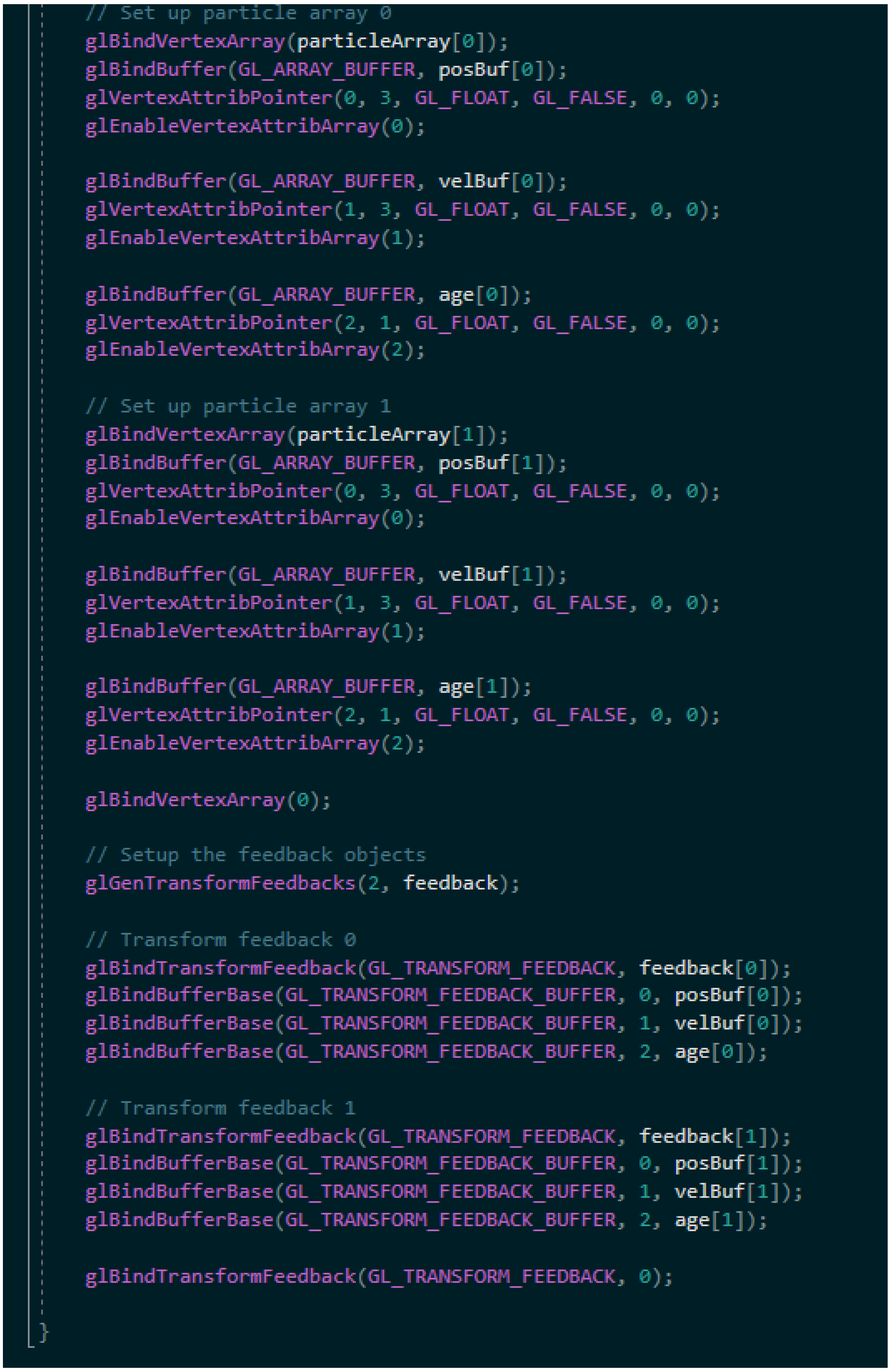
Initialize the A buffers with appropriate initial values. For example, all of the positions could be set to the origin, and the velocities and start times could be initialized in the same way as described in the previous particle fountain example. The initial velocity buffer could simply be a copy of 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. To help simplify things, we'll make use of transform feedback objects.

You do all this in the initBuffers():

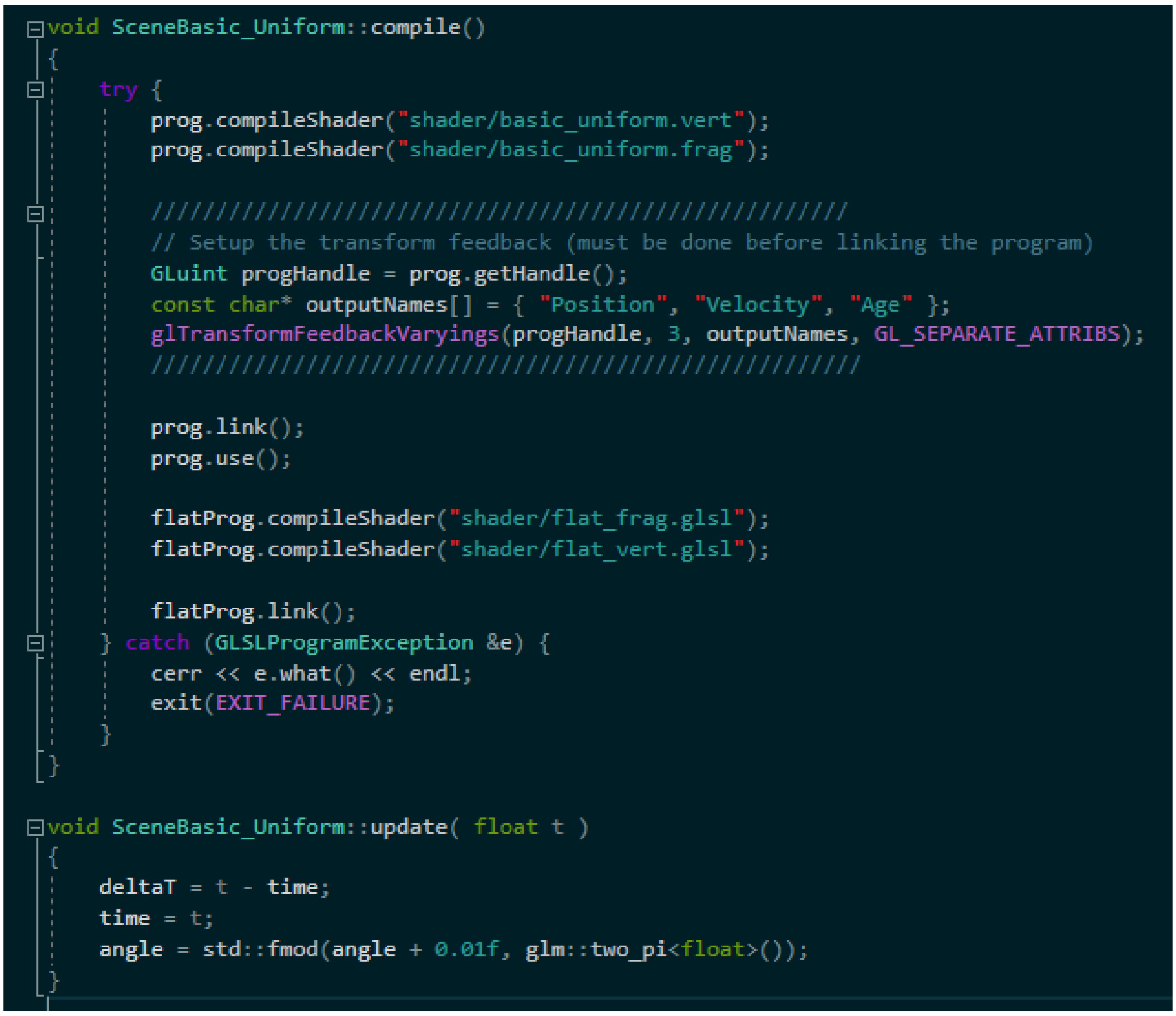


```
void SceneBasic_Uniform::initBuffers()
{
    // Generate the buffers
    glGenBuffers(2, posBuf);    // position buffers
    glGenBuffers(2, velBuf);    // velocity buffers
    glGenBuffers(2, age);       // age buffers

    // Allocate space for all buffers
    int size = nParticles * 3 * sizeof(GLfloat);
    glBindBuffer(GL_ARRAY_BUFFER, posBuf[0]);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, posBuf[1]);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, velBuf[0]);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, velBuf[1]);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, age[0]);
    glBufferData(GL_ARRAY_BUFFER, nParticles * sizeof(float), 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, age[1]);
    glBufferData(GL_ARRAY_BUFFER, nParticles * sizeof(float), 0, GL_DYNAMIC_COPY);

    // Fill the first age buffer
    std::vector<GLfloat> tempData(nParticles);
    float rate = particleLifetime / nParticles;
    for (int i = 0; i < nParticles; i++) {
        tempData[i] = rate * (i - nParticles);
    }
    glBindBuffer(GL_ARRAY_BUFFER, age[0]);
    glBufferSubData(GL_ARRAY_BUFFER, 0, nParticles * sizeof(float), tempData.data());

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // Create vertex arrays for each set of buffers
    glGenVertexArrays(2, particleArray);
```



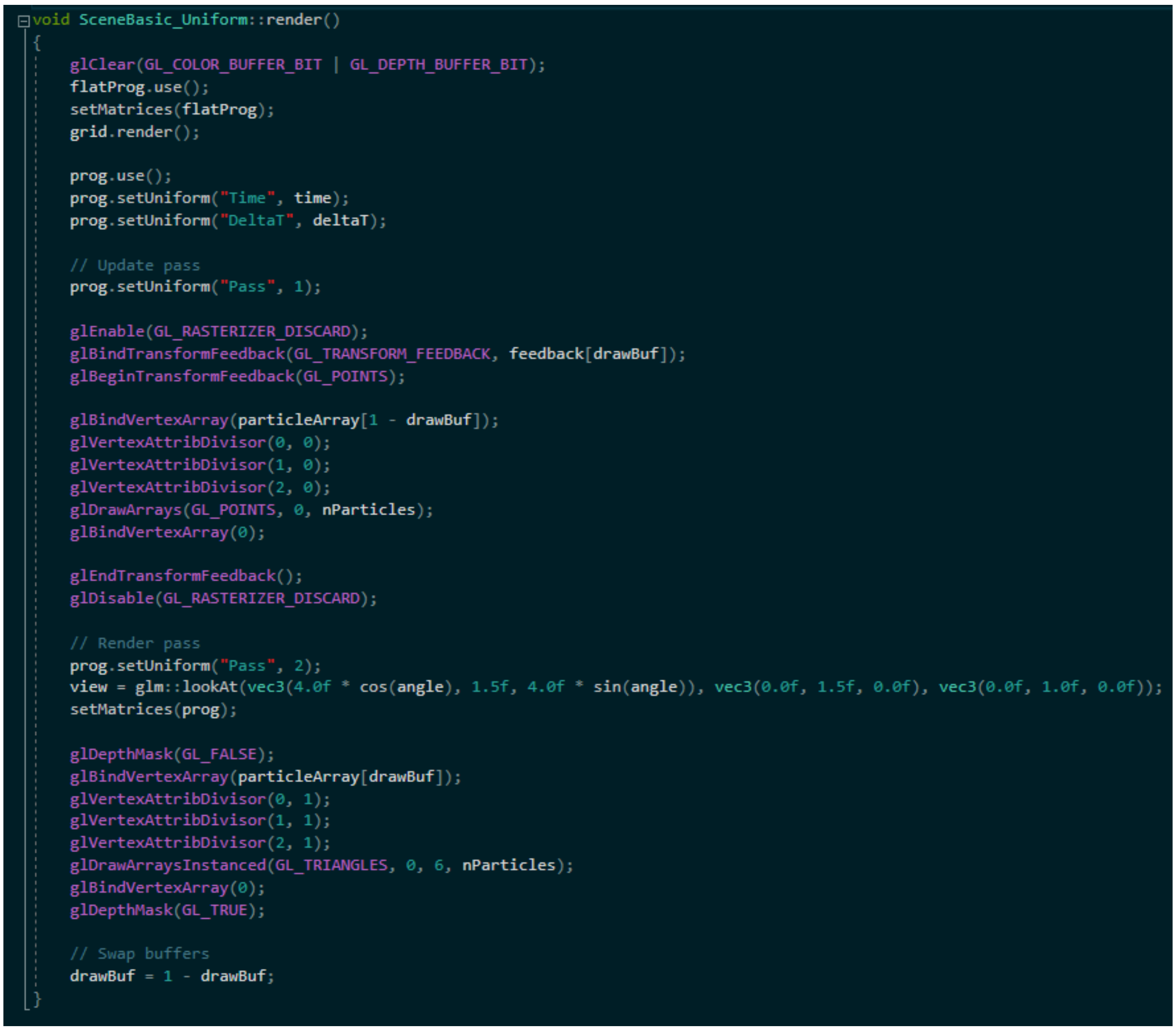
```
    // Set up particle array 0
    glBindVertexArray(particleArray[0]);
    glBindBuffer(GL_ARRAY_BUFFER, posBuf[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, velBuf[0]);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    glBindBuffer(GL_ARRAY_BUFFER, age[0]);
    glVertexAttribPointer(2, 1, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(2);

    // Set up particle array 1
    glBindVertexArray(particleArray[1]);
    glBindBuffer(GL_ARRAY_BUFFER, posBuf[1]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, velBuf[1]);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    glBindBuffer(GL_ARRAY_BUFFER, age[1]);
    glVertexAttribPointer(2, 1, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(2);

    glBindVertexArray(0);

    // Setup the feedback objects
    glGenTransformFeedbacks(2, feedback);

    // Transform feedback 0
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, posBuf[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, velBuf[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2, age[0]);

    // Transform feedback 1
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, posBuf[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, velBuf[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2, age[1]);

    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, 0);

}
```

The compile() and update() method should look like this:



```
void SceneBasic_Uniform::compile()
{
    try {
        prog.compileShader("shader/basic_uniform.vert");
        prog.compileShader("shader/basic_uniform.frag");

        //////////////////////////////////////////////////////
        // Setup the transform feedback (must be done before linking the program)
        GLuint progHandle = prog.getHandle();
        const char* outputNames[] = { "Position", "Velocity", "Age" };
        glTransformFeedbackVaryings(progHandle, 3, outputNames, GL_SEPARATE_ATTRIBS);
        //////////////////////////////////////////////////////

        prog.link();
        prog.use();

        flatProg.compileShader("shader/flat_frag.glsl");
        flatProg.compileShader("shader/flat_vert.glsl");

        flatProg.link();
    } catch (GLSLProgramException &e) {
        cerr << e.what() << endl;
        exit(EXIT_FAILURE);
    }
}

void SceneBasic_Uniform::update( float t )
{
    deltaT = t - time;
    time = t;
    angle = std::fmod(angle + 0.01f, glm::two_pi<float>());
}
```

And render method should look like this:



```
void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    flatProg.use();
    setMatrices(flatProg);
    grid.render();

    prog.use();
    prog.setUniform("Time", time);
    prog.setUniform("DeltaT", deltaT);

    // Update pass
    prog.setUniform("Pass", 1);

    glEnable(GL_RASTERIZER_DISCARD);
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[drawBuf]);
    glBeginTransformFeedback(GL_POINTS);

    glBindVertexArray(particleArray[1 - drawBuf]);
    glVertexAttribDivisor(0, 0);
    glVertexAttribDivisor(1, 0);
    glVertexAttribDivisor(2, 0);
    glDrawArrays(GL_POINTS, 0, nParticles);
    glBindVertexArray(0);

    glEndTransformFeedback();
    glDisable(GL_RASTERIZER_DISCARD);

    // Render pass
    prog.setUniform("Pass", 2);
    view = glm::lookAt(vec3(4.0f * cos(angle), 1.5f, 4.0f * sin(angle)), vec3(0.0f, 1.5f, 0.0f), vec3(0.0f, 1.0f, 0.0f));
    setMatrices(prog);

    glDepthMask(GL_FALSE);
    glBindVertexArray(particleArray[drawBuf]);
    glVertexAttribDivisor(0, 1);
    glVertexAttribDivisor(1, 1);
    glVertexAttribDivisor(2, 1);
    glDrawArraysInstanced(GL_TRIANGLES, 0, 6, nParticles);
    glBindVertexArray(0);
    glDepthMask(GL_TRUE);

    // Swap buffers
    drawBuf = 1 - drawBuf;
}
```

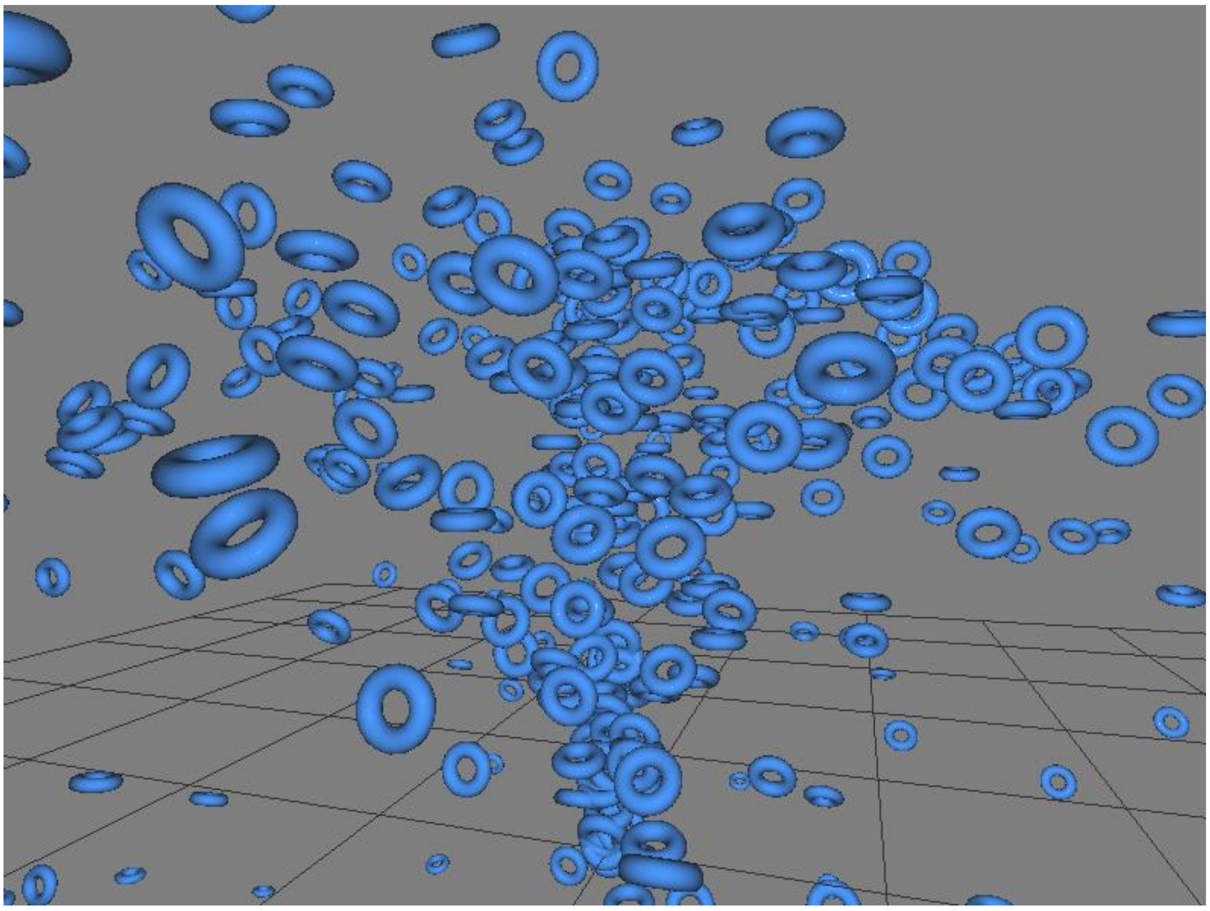
Use the setMatrices() and resize() methods from the particle fountain example.

That’s it.

Particle system using instanced meshes

Very similar to the previous example, except that we'll be using meshes instead of quads.

The end result:

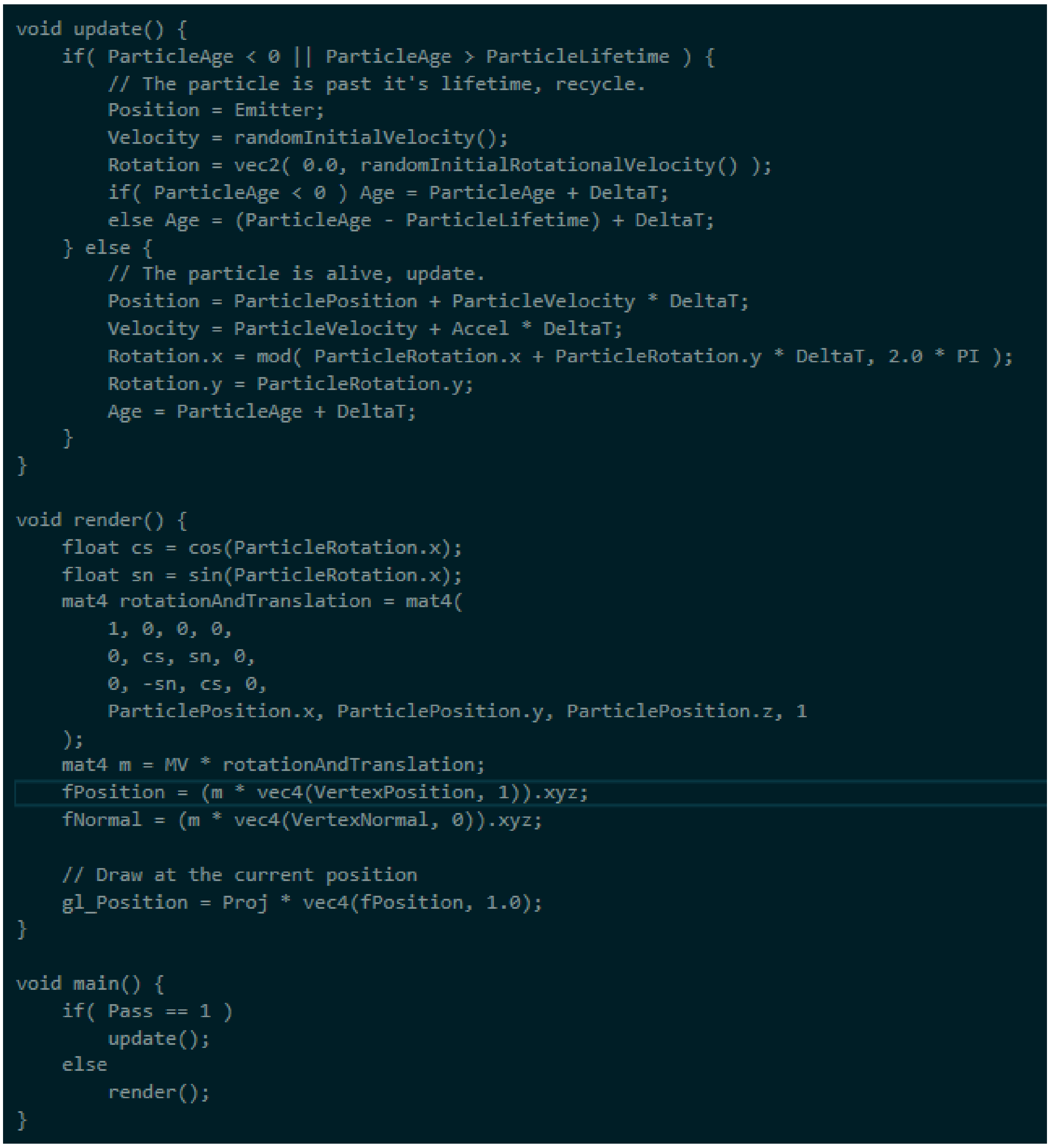


Vertex shader:

Part1 of the vertex shader:

```
const float PI = 3.14159265359;

uniform int Pass;

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

layout (location = 3) in vec3 ParticlePosition;
layout (location = 4) in vec3 ParticleVelocity;
layout (location = 5) in float ParticleAge;
layout (location = 6) in vec2 ParticleRotation;

// To transform feedback
/*layout( xfb_buffer = 0, xfb_offset=0 )*/ out vec3 Position;
/*layout( xfb_buffer = 1, xfb_offset=0 )*/ out vec3 Velocity;
/*layout( xfb_buffer = 2, xfb_offset=0 )*/ out float Age;
/*layout( xfb_buffer = 3, xfb_offset=0 )*/ out vec2 Rotation;

// To fragment shader
out vec3 fPosition;
out vec3 fNormal;

uniform float Time;              // Simulation time
uniform float DeltaT;            // Elapsed time between frames
uniform vec3 Accel;              // Particle acceleration (gravity)
uniform float ParticleLifetime;  // Particle lifespan
uniform vec3 Emitter = vec3(0);  // World position of the emitter.
uniform mat3 EmitterBasis;       // Rotation that rotates y axis to the direction of emitter
uniform float ParticleSize;      // Size of particle

// Transforms
uniform mat4 MV;    // View * Model
uniform mat4 Proj;  // Projection matrix

uniform sampler1D RandomTex;

vec3 randomInitialVelocity() {
    float theta = mix(0.0, PI / 6.0, texelFetch(RandomTex, 4 * gl_VertexID, 0).r );
    float phi = mix(0.0, 2.0 * PI, texelFetch(RandomTex, 4 * gl_VertexID + 1, 0).r);
    float velocity = mix(1.25, 1.5, texelFetch(RandomTex, 4 * gl_VertexID + 2, 0).r );
    vec3 v = vec3(sin(theta) * cos(phi), cos(theta), sin(theta) * sin(phi));
    return normalize(EmitterBasis * v) * velocity;
}

float randomInitialRotationalVelocity() {
    return mix(-15.0, 15.0, texelFetch(RandomTex, 4 * gl_VertexID + 3, 0).r );
}
```

Part 2 of the vertex shader:

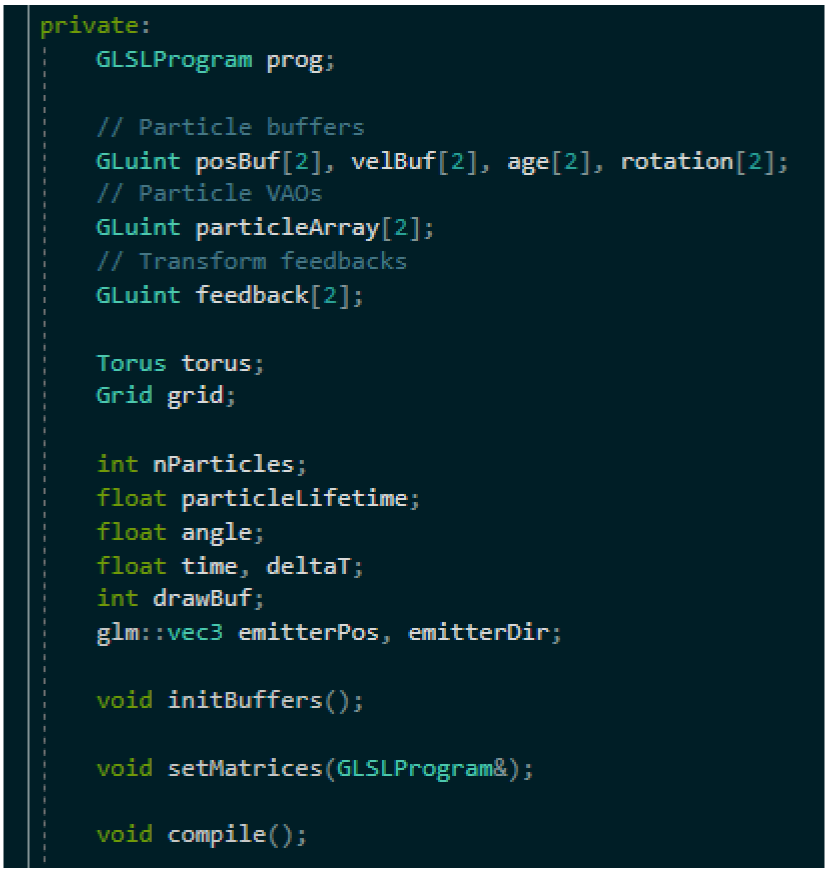


```
void update() {
    if( ParticleAge < 0 || ParticleAge > ParticleLifetime ) {
        // The particle is past it's lifetime, recycle.
        Position = Emitter;
        Velocity = randomInitialVelocity();
        Rotation = vec2( 0.0, randomInitialRotationalVelocity() );
        if( ParticleAge < 0 ) Age = ParticleAge + DeltaT;
        else Age = (ParticleAge - ParticleLifetime) + DeltaT;
    } else {
        // The particle is alive, update.
        Position = ParticlePosition + ParticleVelocity * DeltaT;
        Velocity = ParticleVelocity + Accel * DeltaT;
        Rotation.x = mod( ParticleRotation.x + ParticleRotation.y * DeltaT, 2.0 * PI );
        Rotation.y = ParticleRotation.y;
        Age = ParticleAge + DeltaT;
    }
}

void render() {
    float cs = cos(ParticleRotation.x);
    float sn = sin(ParticleRotation.x);
    mat4 rotationAndTranslation = mat4(
        1, 0, 0, 0,
        0, cs, sn, 0,
        0, -sn, cs, 0,
        ParticlePosition.x, ParticlePosition.y, ParticlePosition.z, 1
    );
    mat4 m = MV * rotationAndTranslation;
    fPosition = (m * vec4(VertexPosition, 1)).xyz;
    fNormal = (m * vec4(VertexNormal, 0)).xyz;

    // Draw at the current position
    gl_Position = Proj * vec4(fPosition, 1.0);
}

void main() {
    if( Pass == 1 )
        update();
    else
        render();
}
```

Fragment shader:

Just implement a Phong or a Blinn-Phong shader

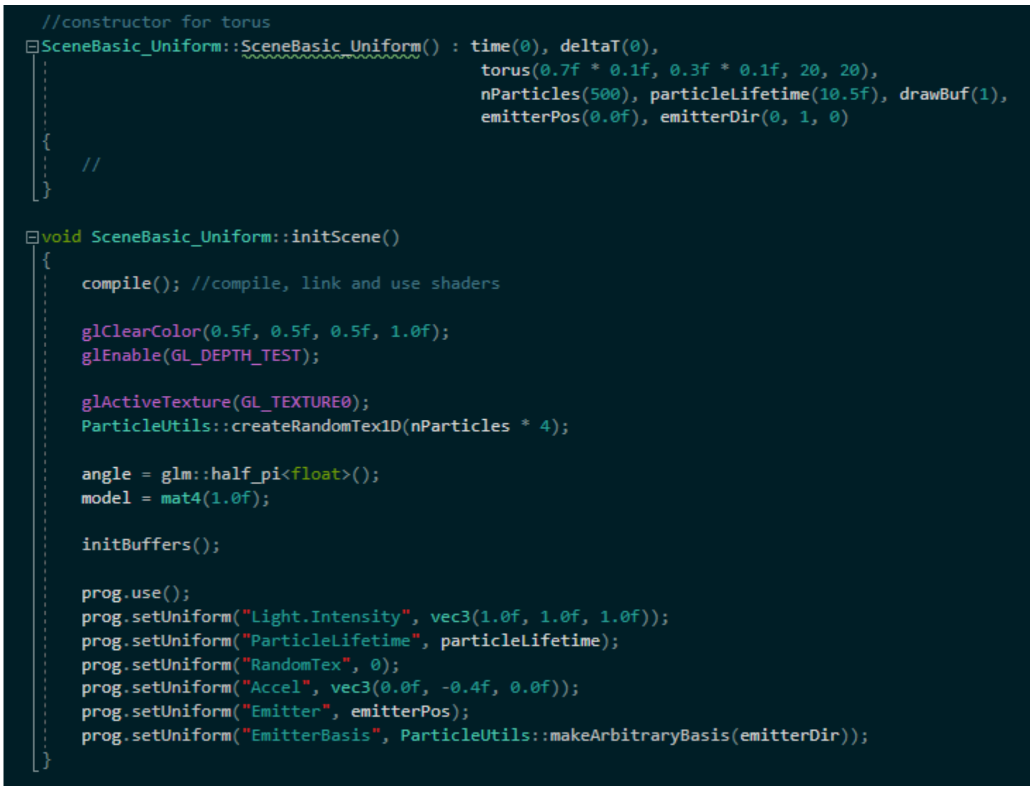
scenebasic_uniform.h:



```
private:
    GLSLProgram prog;

    // Particle buffers
    GLuint posBuf[2], velBuf[2], age[2], rotation[2];
    // Particle VAOs
    GLuint particleArray[2];
    // Transform feedbacks
    GLuint feedback[2];

    Torus torus;
    Grid grid;

    int nParticles;
    float particleLifetime;
    float angle;
    float time, deltaT;
    int drawBuf;
    glm::vec3 emitterPos, emitterDir;

    void initBuffers();

    void setMatrices(GLSLProgram&);

    void compile();
```

scenebasic_uniform.cpp:

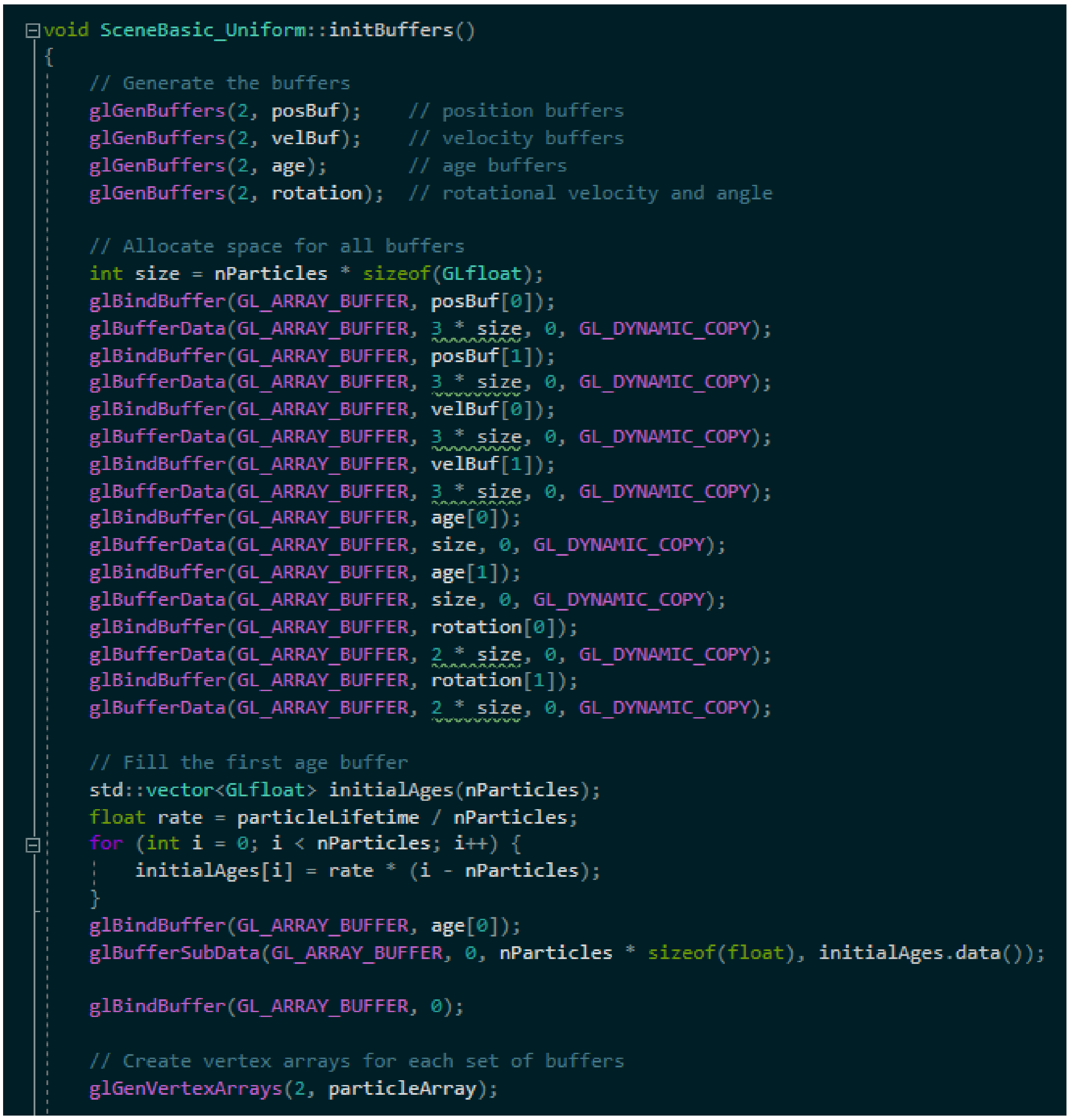
For constructor and initScene() use this code:



```
//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : time(0), deltaT(0),
                                            torus(0.7f * 0.1f, 0.3f * 0.1f, 20, 20),
                                            nParticles(500), particleLifetime(10.5f), drawBuf(1),
                                            emitterPos(0.0f), emitterDir(0, 1, 0)
{
    //
}

void SceneBasic_Uniform::initScene()
{
    compile(); //compile, link and use shaders

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glEnable(GL_DEPTH_TEST);

    glActiveTexture(GL_TEXTURE0);
    ParticleUtils::createRandomTex1D(nParticles * 4);

    angle = glm::half_pi<float>();
    model = mat4(1.0f);

    initBuffers();

    prog.use();
    prog.setUniform("Light.Intensity", vec3(1.0f, 1.0f, 1.0f));
    prog.setUniform("ParticleLifetime", particleLifetime);
    prog.setUniform("RandomTex", 0);
    prog.setUniform("Accel", vec3(0.0f, -0.4f, 0.0f));
    prog.setUniform("Emitter", emitterPos);
    prog.setUniform("EmitterBasis", ParticleUtils::makeArbitraryBasis(emitterDir));
}
```

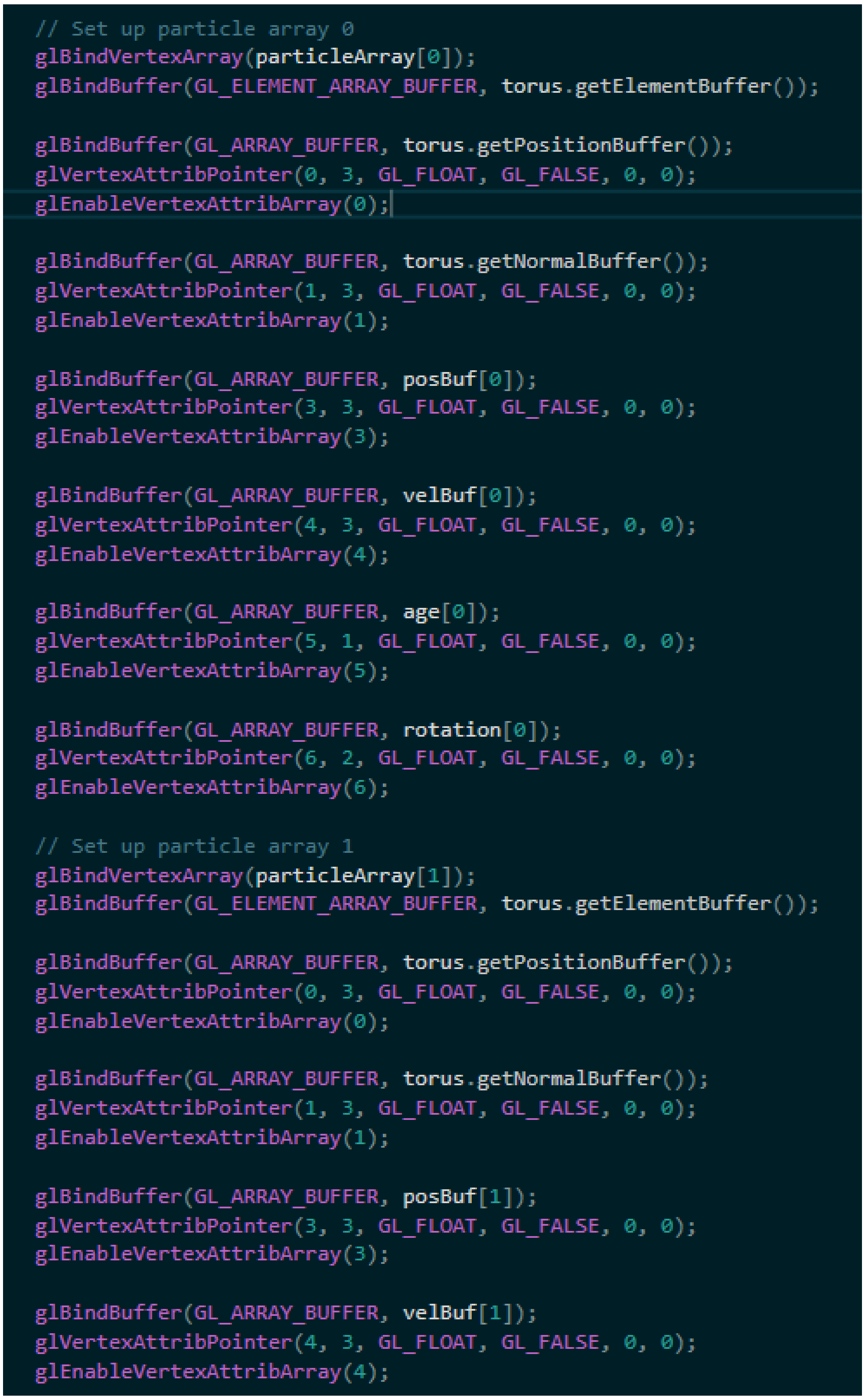
For initBuffers() use this code (part 1, 2 and 3):

Part 1:



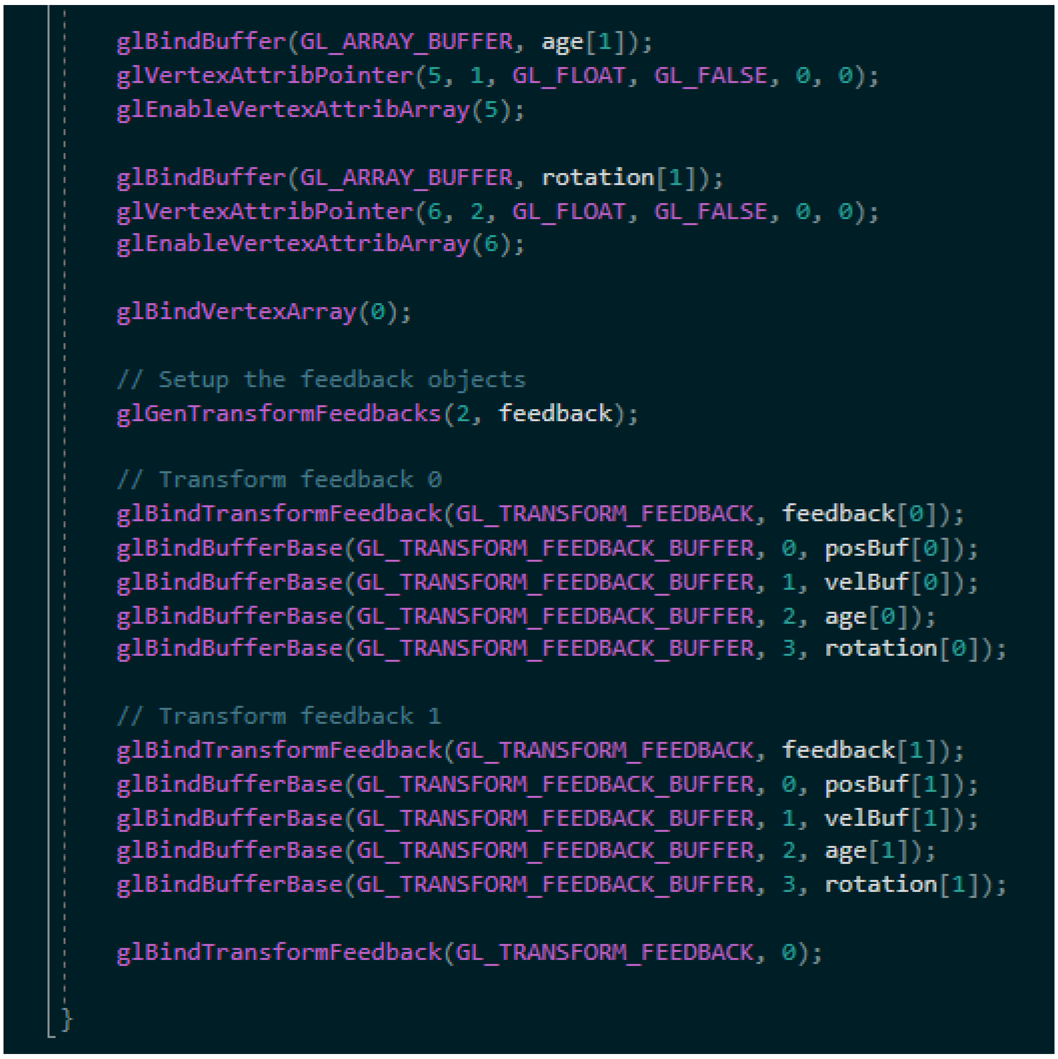
```
void SceneBasic_Uniform::initBuffers()
{
    // Generate the buffers
    glGenBuffers(2, posBuf);    // position buffers
    glGenBuffers(2, velBuf);    // velocity buffers
    glGenBuffers(2, age);       // age buffers
    glGenBuffers(2, rotation);  // rotational velocity and angle

    // Allocate space for all buffers
    int size = nParticles * sizeof(GLfloat);
    glBindBuffer(GL_ARRAY_BUFFER, posBuf[0]);
    glBufferData(GL_ARRAY_BUFFER, 3 * size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, posBuf[1]);
    glBufferData(GL_ARRAY_BUFFER, 3 * size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, velBuf[0]);
    glBufferData(GL_ARRAY_BUFFER, 3 * size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, velBuf[1]);
    glBufferData(GL_ARRAY_BUFFER, 3 * size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, age[0]);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, age[1]);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, rotation[0]);
    glBufferData(GL_ARRAY_BUFFER, 2 * size, 0, GL_DYNAMIC_COPY);
    glBindBuffer(GL_ARRAY_BUFFER, rotation[1]);
    glBufferData(GL_ARRAY_BUFFER, 2 * size, 0, GL_DYNAMIC_COPY);

    // Fill the first age buffer
    std::vector<GLfloat> initialAges(nParticles);
    float rate = particleLifetime / nParticles;
    for (int i = 0; i < nParticles; i++) {
        initialAges[i] = rate * (i - nParticles);
    }
    glBindBuffer(GL_ARRAY_BUFFER, age[0]);
    glBufferSubData(GL_ARRAY_BUFFER, 0, nParticles * sizeof(float), initialAges.data());

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // Create vertex arrays for each set of buffers
    glGenVertexArrays(2, particleArray);
```

Part 2:



```
// Set up particle array 0
glBindVertexArray(particleArray[0]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, torus.getElementBuffer());

glBindBuffer(GL_ARRAY_BUFFER, torus.getPositionBuffer());
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, torus.getNormalBuffer());
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

glBindBuffer(GL_ARRAY_BUFFER, posBuf[0]);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(3);

glBindBuffer(GL_ARRAY_BUFFER, velBuf[0]);
glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(4);

glBindBuffer(GL_ARRAY_BUFFER, age[0]);
glVertexAttribPointer(5, 1, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(5);

glBindBuffer(GL_ARRAY_BUFFER, rotation[0]);
glVertexAttribPointer(6, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(6);

// Set up particle array 1
glBindVertexArray(particleArray[1]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, torus.getElementBuffer());

glBindBuffer(GL_ARRAY_BUFFER, torus.getPositionBuffer());
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, torus.getNormalBuffer());
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

glBindBuffer(GL_ARRAY_BUFFER, posBuf[1]);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(3);

glBindBuffer(GL_ARRAY_BUFFER, velBuf[1]);
glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(4);
```

Part 3:



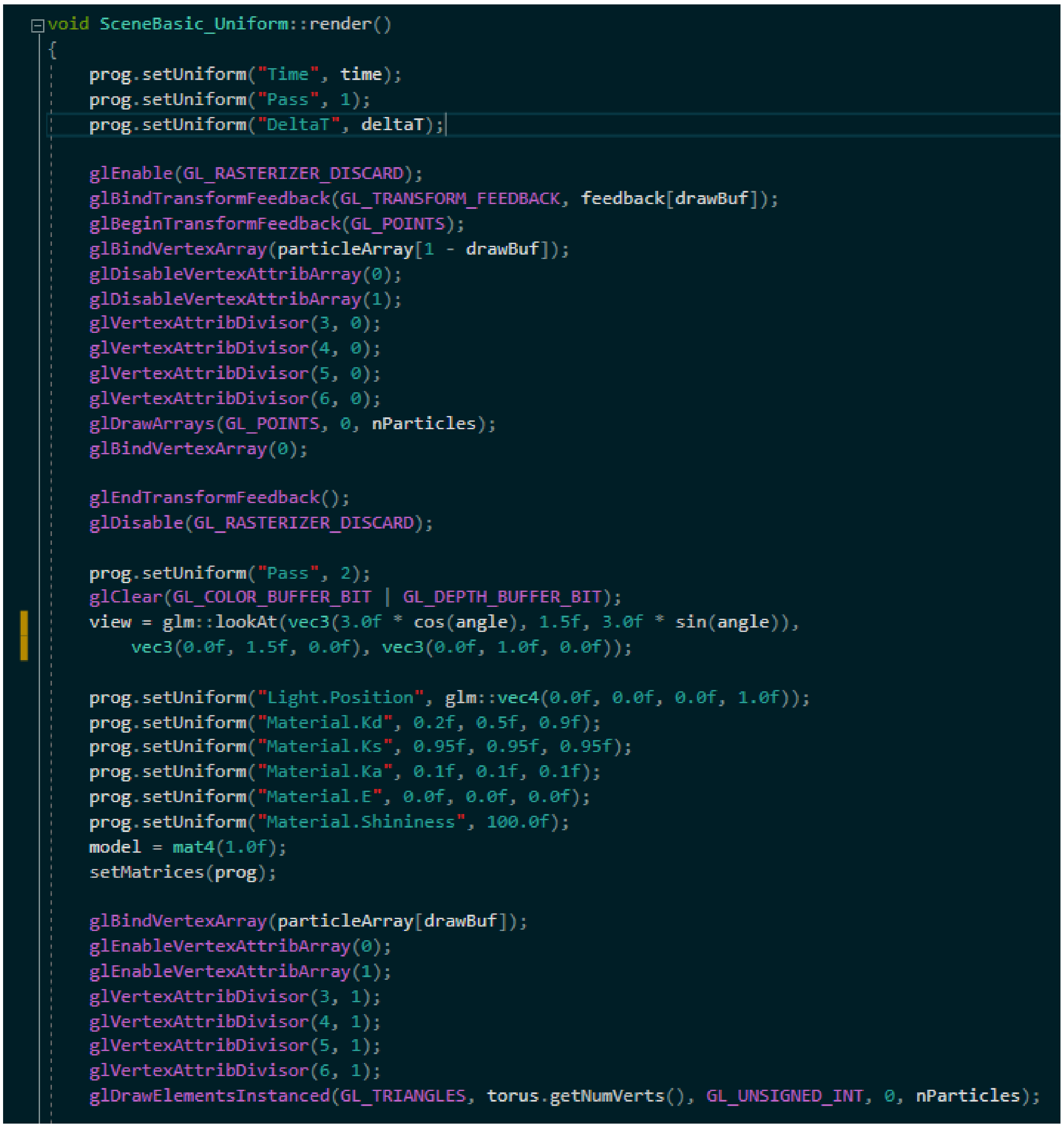
```
    glBindBuffer(GL_ARRAY_BUFFER, age[1]);
    glVertexAttribPointer(5, 1, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(5);

    glBindBuffer(GL_ARRAY_BUFFER, rotation[1]);
    glVertexAttribPointer(6, 2, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(6);

    glBindVertexArray(0);

    // Setup the feedback objects
    glGenTransformFeedbacks(2, feedback);

    // Transform feedback 0
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, posBuf[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, velBuf[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2, age[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 3, rotation[0]);

    // Transform feedback 1
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, posBuf[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, velBuf[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2, age[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 3, rotation[1]);

    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, 0);

}
```

For compile() method use the compile() from “Particle fountain with feedback” but:

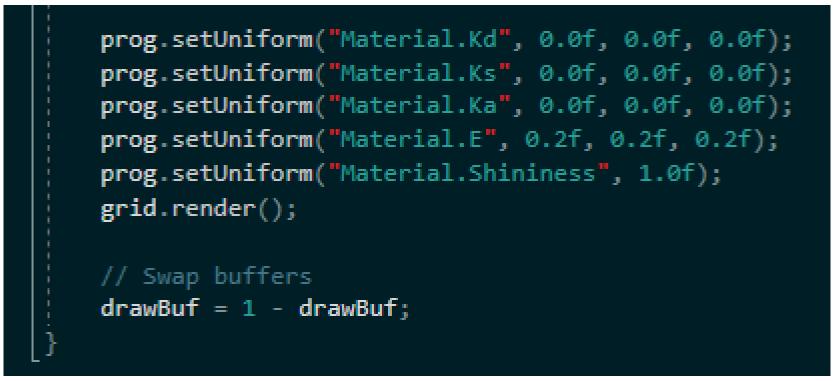
- For outputNames[] add a new name “Rotation”
- for `glTransformFeedbackVaryings` make sure it has 4 elements instead of 3.
- Remove flatProg as you don’t need it.

For update() stays the same as in previous example “Particle fountain with feedback”.

For render() use this code:



```
void SceneBasic_Uniform::render()
{
    prog.setUniform("Time", time);
    prog.setUniform("Pass", 1);
    prog.setUniform("DeltaT", deltaT);

    glEnable(GL_RASTERIZER_DISCARD);
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[drawBuf]);
    glBeginTransformFeedback(GL_POINTS);
    glBindVertexArray(particleArray[1 - drawBuf]);
    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glVertexAttribDivisor(3, 0);
    glVertexAttribDivisor(4, 0);
    glVertexAttribDivisor(5, 0);
    glVertexAttribDivisor(6, 0);
    glDrawArrays(GL_POINTS, 0, nParticles);
    glBindVertexArray(0);

    glEndTransformFeedback();
    glDisable(GL_RASTERIZER_DISCARD);

    prog.setUniform("Pass", 2);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    view = glm::lookAt(vec3(3.0f * cos(angle), 1.5f, 3.0f * sin(angle)),
        vec3(0.0f, 1.5f, 0.0f), vec3(0.0f, 1.0f, 0.0f));

    prog.setUniform("Light.Position", glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));
    prog.setUniform("Material.Kd", 0.2f, 0.5f, 0.9f);
    prog.setUniform("Material.Ks", 0.95f, 0.95f, 0.95f);
    prog.setUniform("Material.Ka", 0.1f, 0.1f, 0.1f);
    prog.setUniform("Material.E", 0.0f, 0.0f, 0.0f);
    prog.setUniform("Material.Shininess", 100.0f);
    model = mat4(1.0f);
    setMatrices(prog);

    glBindVertexArray(particleArray[drawBuf]);
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glVertexAttribDivisor(3, 1);
    glVertexAttribDivisor(4, 1);
    glVertexAttribDivisor(5, 1);
    glVertexAttribDivisor(6, 1);
    glDrawElementsInstanced(GL_TRIANGLES, torus.getNumVerts(), GL_UNSIGNED_INT, 0, nParticles);
```



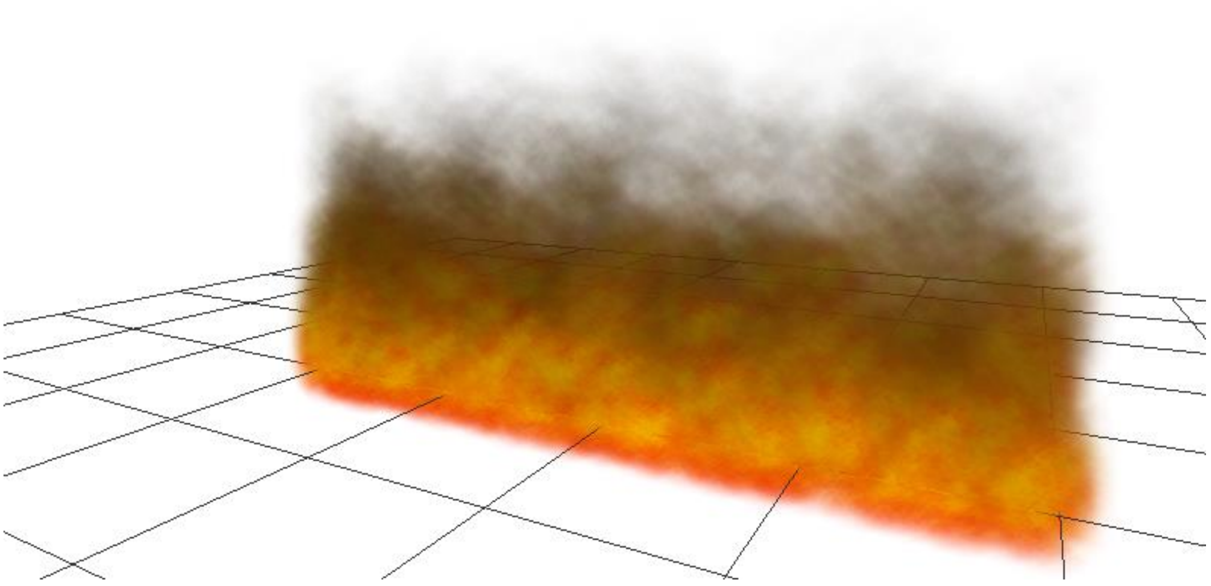
```
    prog.setUniform("Material.Kd", 0.0f, 0.0f, 0.0f);
    prog.setUniform("Material.Ks", 0.0f, 0.0f, 0.0f);
    prog.setUniform("Material.Ka", 0.0f, 0.0f, 0.0f);
    prog.setUniform("Material.E", 0.2f, 0.2f, 0.2f);
    prog.setUniform("Material.Shininess", 1.0f);
    grid.render();

    // Swap buffers
    drawBuf = 1 - drawBuf;
}
```

Use the setMatrices() and resize() methods from the “particle fountain with feedback” example.

That’s it.

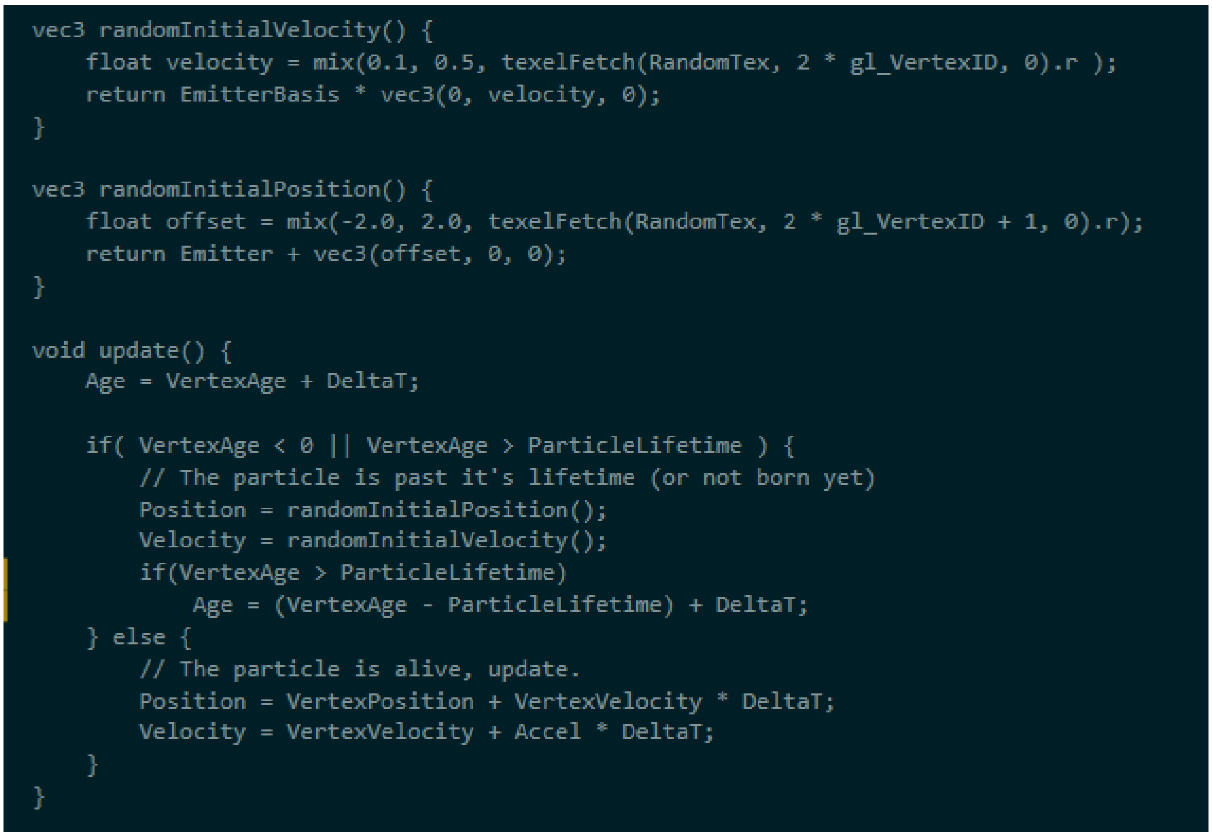
Fire particles

The end result:



Vertex shader:

Use vertex shader from the “particle fountain with feedback” example, but if you want a bit more control over the update(), replace it with this code:

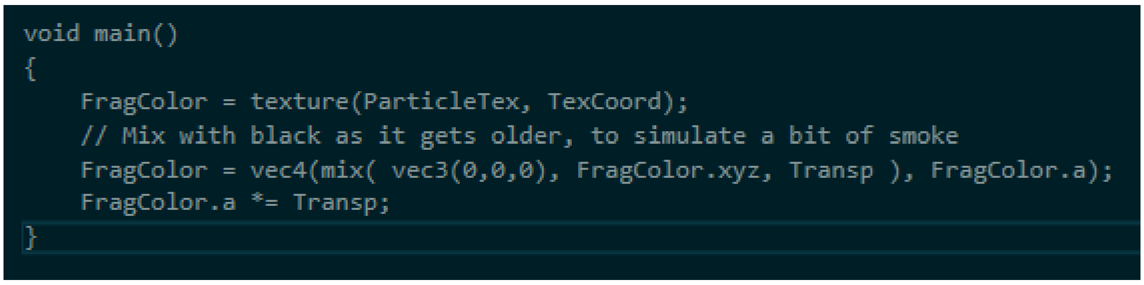


```
vec3 randomInitialVelocity() {
    float velocity = mix(0.1, 0.5, texelFetch(RandomTex, 2 * gl_VertexID, 0).r );
    return EmitterBasis * vec3(0, velocity, 0);
}

vec3 randomInitialPosition() {
    float offset = mix(-2.0, 2.0, texelFetch(RandomTex, 2 * gl_VertexID + 1, 0).r);
    return Emitter + vec3(offset, 0, 0);
}

void update() {
    Age = VertexAge + DeltaT;

    if( VertexAge < 0 || VertexAge > ParticleLifetime ) {
        // The particle is past it's lifetime (or not born yet)
        Position = randomInitialPosition();
        Velocity = randomInitialVelocity();
        if(VertexAge > ParticleLifetime)
            Age = (VertexAge - ParticleLifetime) + DeltaT;
    } else {
        // The particle is alive, update.
        Position = VertexPosition + VertexVelocity * DeltaT;
        Velocity = VertexVelocity + Accel * DeltaT;
    }
}
```

Make sure you're using these methods for randomInitialVelocity() and randomInitialPosition(), see in the image above.

Fragment shader:

Use fragment shader from the "particle fountain with feedback" example but mix the colour with black as the particles get older:



```
void main()
{
    FragColor = texture(ParticleTex, TexCoord);
    // Mix with black as it gets older, to simulate a bit of smoke
    FragColor = vec4(mix( vec3(0,0,0), FragColor.xyz, Transp ), FragColor.a);
    FragColor.a *= Transp;
}
```

scenebasic_uniform.h:

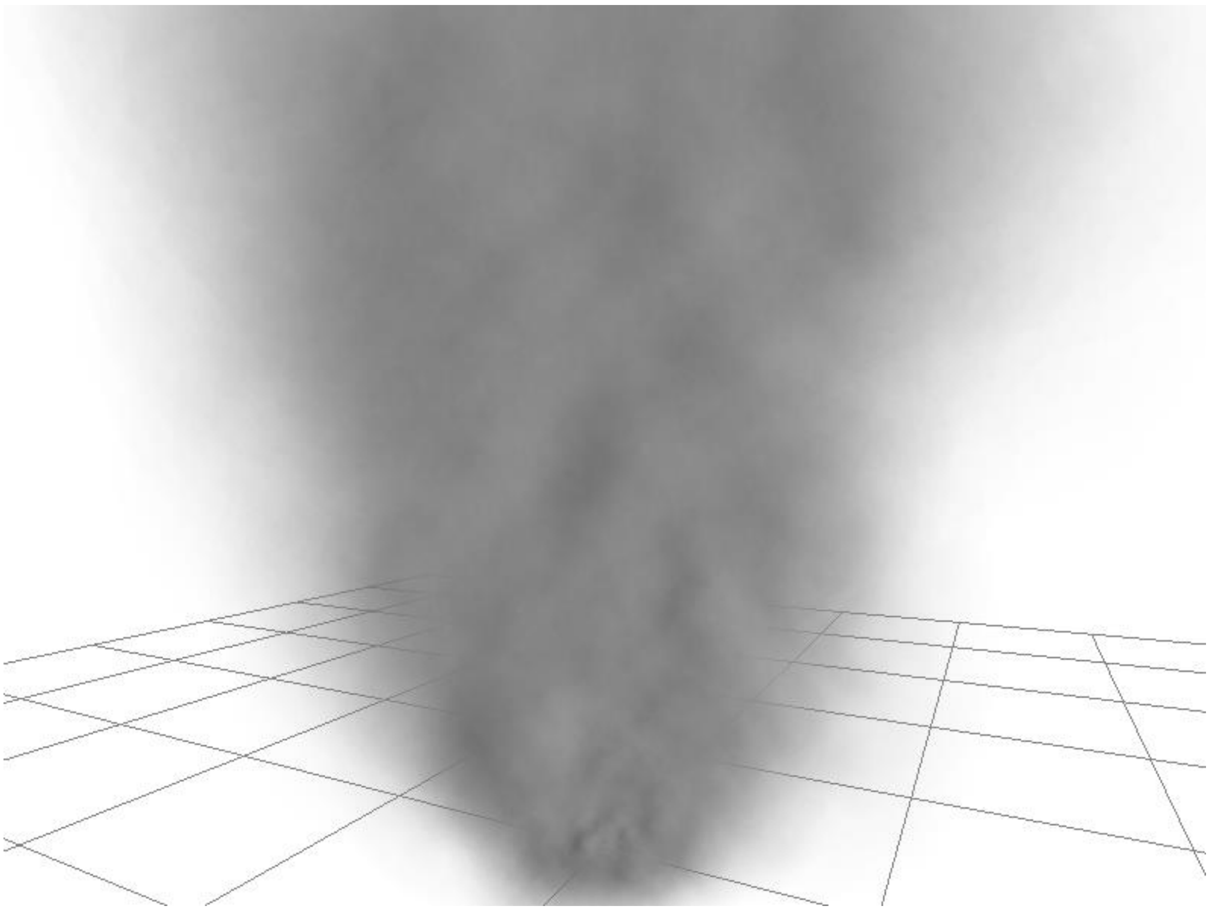
Use scenebasic_uniform.h from the "particle fountain with feedback" example.

scenebasic_uniform.cpp:

Use scenebasic_uniform.cpp from the "particle fountain with feedback" example.

Smoke particles

The end result:



Vertex shader:

Use the same shader as in Fire particle example, except you need to add 2 uniforms:

```
        uniform float MinParticleSize = 0.1;     // Minimum size
        uniform float MaxParticleSize = 2.5;     // Maximum size
```

and the render function becomes this:

```
void render() {
    Transp = 0.0;
    vec3 posCam = vec3(0.0);
    if( VertexAge >= 0.0 ) {
        float agePct = VertexAge / ParticleLifetime;
        Transp = clamp(1.0 - agePct, 0, 1);
        posCam =
            (MV * vec4(VertexPosition,1)).xyz +
            offsets[gl_VertexID] *
            mix(MinParticleSize, MaxParticleSize, agePct);
    }
    TexCoord = texCoords[gl_VertexID];

    gl_Position = Proj * vec4(posCam,1);
}
```

We basically grow the particles as they age.

The rest is identical to the fire particles example.