Computer Graphics – Transform Feedback*

J.-Prof. Dr. habil. Kai Lawonn

Introduction

- Up to now, we have always sent vertex data to the GPU to generate drawn pixels
- What if we want to get the vertices back after, we processed them in a vertex shader?
- In this lecture, we will learn one way to do this, known as transform feedback

Basic

- Let‘s say, we want to compute the sin of an array
- The vertex shader would be:

```
const char* tf_shader_src =
"#version 330 core\n"
"layout (location = 0) in float inFloat;\n"
"out float outFloat;\n"
"void main()\n"
"{\n"
"   outFloat=sin(inFloat);\n"
"}\n\0";
```

Basic

- Interestingly, this vertex shader does not have the output gl_Position
- We only get inFloat as an in variable and get a float outFloat out

Basic

- Just like the initiation of a normal vertex shader, we set up a shader for this transform feedback object

```
// vertex shader
int tf_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(tf_shader, 1, &tf_shader_src, NULL);
glCompileShader(tf_shader);

// link shaders
unsigned int tf_Program = glCreateProgram();
glAttachShader(tf_Program, tf_shader);
```

- Nothing new…
- We do not link the program yet

Basic

- Now, we tell OpenGL, which is the output value (outFloat)

```
const GLchar* feedbackVaryings[] = { "outFloat" };
glTransformFeedbackVaryings(tf_Program, 1, feedbackVaryings,
GL_INTERLEAVED_ATTRIBS);
```

- `glTransformFeedbackVaryings:`
- 1.: target program
- 2.: number of varying variables
- 3.: names of varying variables
- 4.: specifies the mode used to capture the varying variables: GL_INTERLEAVED_ATTRIBS (written to a single buffer object) or GL_SEPARATE_ATTRIBS (written to a multiple buffer objects)

Basic

- Link, delete and use:

```
glLinkProgram(tf_Program);
glDeleteShader(tf_shader);
glUseProgram(tf_Program);
```

- Now, we can create the VAO:

```
unsigned int tf_vao;
glGenVertexArrays(1, &tf_vao);
glBindVertexArray(tf_vao);
```

Basic

- Then, we need an array and a corresponding buffer:

```
float tf_data[] = {0.0f, 0.5f, 1.0f, 1.5f, 2.0f, 2.5f, 3.0f};

unsigned int  tf_vbo;
glGenBuffers(1, &tf_vbo);
glBindBuffer(GL_ARRAY_BUFFER, tf_vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(tf_data), tf_data, GL_STATIC_DRAW);
```

- tf_data is the array where we want to compute the sin of

Basic

- Similar to previous lectures, set the attribute location and enable it (inFloat)

```
glVertexAttribPointer(0, 1, GL_FLOAT, GL_FALSE, sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Basic

- The output is the array outFloat, but we need a buffer that stores the resulting values

```
unsigned int tf_vbo2;
glGenBuffers(1, &tf_vbo2);
glBindBuffer(GL_ARRAY_BUFFER, tf_vbo2);
glBufferData(GL_ARRAY_BUFFER, sizeof(tf_data), NULL, GL_STATIC_READ);
```

- We set NULL, because no data is copied (data store of specified size is created, but is uninitialized)
- GL_STATIC_READ: data store is modified and data is copied to this buffer, which can be read by our application

Basic

- We will only use the vertex shader for computational purposes, but nothing will be drawn:

```
glEnable(GL_RASTERIZER_DISCARD);
```

- This call will discard the rasterization

Basic

- Now, we bind the buffer as a transform feedback buffer by using glBindBufferBase:

```
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tf_vbo2);
```

- 1.: target of the bind operation
- 2.: index of the binding point within the array (not so interesting for now)
- 3.: buffer object

Basic

- We are entering the transform feedback mode:

```
glBeginTransformFeedback(GL_POINTS);
```

- Other options are possible, but we consider the float values as points such that every point is processed on their own, and thus, is calculated
- Virtually, we have to tell OpenGL to start/draw:

```
glDrawArrays(GL_POINTS, 0, 7);
```

- Seven values are in the array

Basic

- Empty all buffers and cause that all operations are executed as quickly as possible

```
glFlush();
```

- It does not tell you anything about the execution status, but should be called after the operations are finished
- “For example, call glFlush before waiting for user input that depends on the generated image.”

Basic

- Finally, copy the results back:

```
float feedback[7];
glGetBufferSubData(GL_TRANSFORM_FEEDBACK_BUFFER, 0, sizeof(feedback),
feedback);
```

- glGetBufferSubData:
- 1.: target of the buffer
- 2.: offset of the buffer object's data
- 3.: size of the data store
- 4.: pointer of the buffer object data that is returned

Basics

- Print it:

```
printf("%f %f %f %f %f %f %f\n", feedback[0], feedback[1], feedback[2],
feedback[3], feedback[4], feedback[5], feedback[6]);
```

```
0.000000 0.479426 0.841471 0.997495 0.909298 0.598472 0.141120
```

- We did it!

Basic

- Do not forget to delete everything afterwards and activate the rasterizer again:

```
glDeleteProgram(tf_Program);
glDeleteShader(tf_shader);

glDeleteBuffers(1, &tf_vbo);
glDeleteBuffers(1, &tf_vbo2);

glDeleteVertexArrays(1, &tf_vao);



glDisable(GL_RASTERIZER_DISCARD);
```

All together

```
const char* tf_shader_src = "#version 330 core\n"
"layout (location = 0) in float inFloat;\n"
"out float outFloat;\n"
"void main()\n"
"{\n"
"    outFloat=sin(inFloat);\n"
"}\n\0";
```

https://open.gl/feedback (another example)

All together

```
int tf_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(tf_shader, 1, &tf_shader_src, NULL);
glCompileShader(tf_shader);

unsigned int tf_Program = glCreateProgram();
glAttachShader(tf_Program, tf_shader);

const GLchar* feedbackVaryings[] = { "outFloat" };
glTransformFeedbackVaryings(tf_Program, 1, feedbackVaryings, GL_INTERLEAVED_ATTRIBS);

glLinkProgram(tf_Program);
glDeleteShader(tf_shader);
glUseProgram(tf_Program);

unsigned int  tf_vao;
glGenVertexArrays(1, &tf_vao);
glBindVertexArray(tf_vao);

float tf_data[] = {0.0f, 0.5f, 1.0f, 1.5f, 2.0f, 2.5f, 3.0f};

unsigned int  tf_vbo;
glGenBuffers(1, &tf_vbo);
glBindBuffer(GL_ARRAY_BUFFER, tf_vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(tf_data), tf_data, GL_STATIC_DRAW);
```

All together

```
glVertexAttribPointer(0, 1, GL_FLOAT, GL_FALSE, sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

unsigned int tf_vbo2;
glGenBuffers(1, &tf_vbo2);
glBindBuffer(GL_ARRAY_BUFFER, tf_vbo2);
glBufferData(GL_ARRAY_BUFFER, sizeof(tf_data), NULL, GL_STATIC_READ);

glEnable(GL_RASTERIZER_DISCARD);

glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tf_vbo2);

glBeginTransformFeedback(GL_POINTS);
glDrawArrays(GL_POINTS, 0, 7);
glEndTransformFeedback();

glFlush();

float feedback[7];
glGetBufferSubData(GL_TRANSFORM_FEEDBACK_BUFFER, 0, sizeof(feedback), feedback);
printf("%f %f %f %f %f %f %f\n", feedback[0], feedback[1], feedback[2], feedback[3], feedback[4], feedback[5],
feedback[6]);
glDeleteProgram(tf_Program);
glDeleteShader(tf_shader);
glDeleteBuffers(1, &tf_vbo);
glDeleteBuffers(1, &tf_vbo2);
glDeleteVertexArrays(1, &tf_vao);
glDisable(GL_RASTERIZER_DISCARD);
```

Example: Vector Fields

Vector Fields

- 2D Vector field:

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

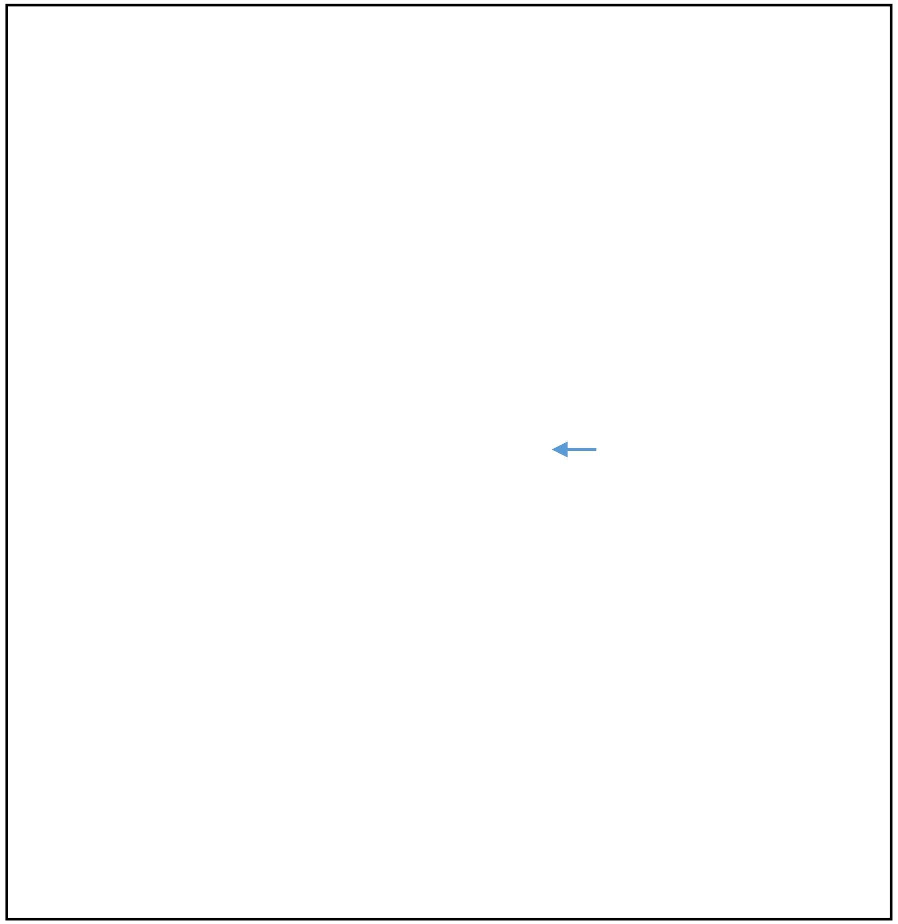
- 3D Vector field :

$$f : \mathbb{R}^3 \to \mathbb{R}^3$$

Vector Fields

- You can think of a vector field as a domain, where we have a vector at each point given
- Example:

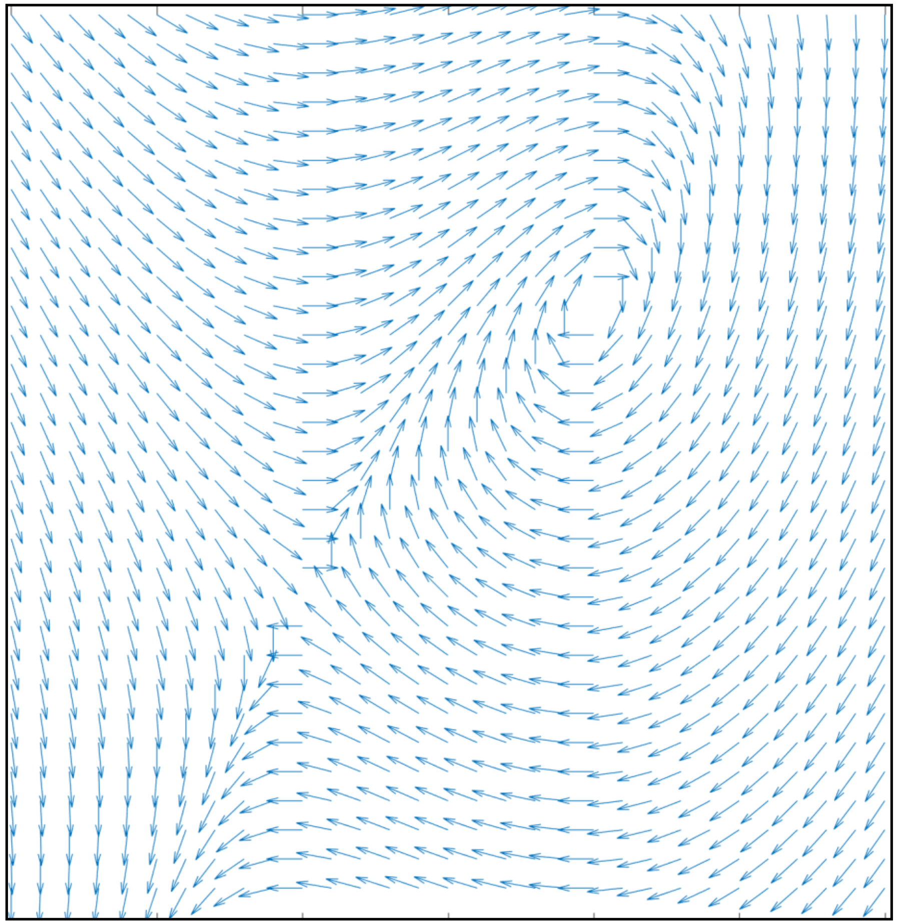
$$\boldsymbol{v}(x, y) = \left(-x + y,\ 1 - x^2\right)^T$$
$$\boldsymbol{v}(1,0) = (-1, 0)^T$$



Vector Fields

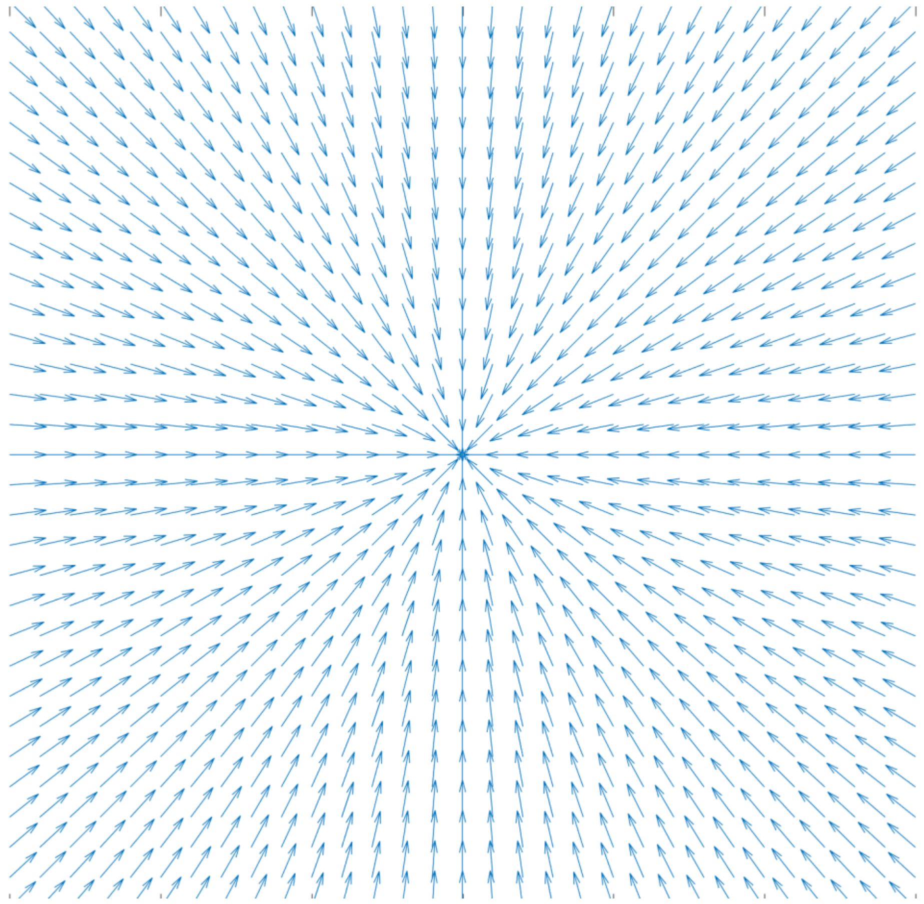
- You can think of a vector field as a domain, where we have a vector at each point given
- Example:

$$\boldsymbol{v}(x, y) = \left(-x + y,\ 1 - x^2\right)^T$$



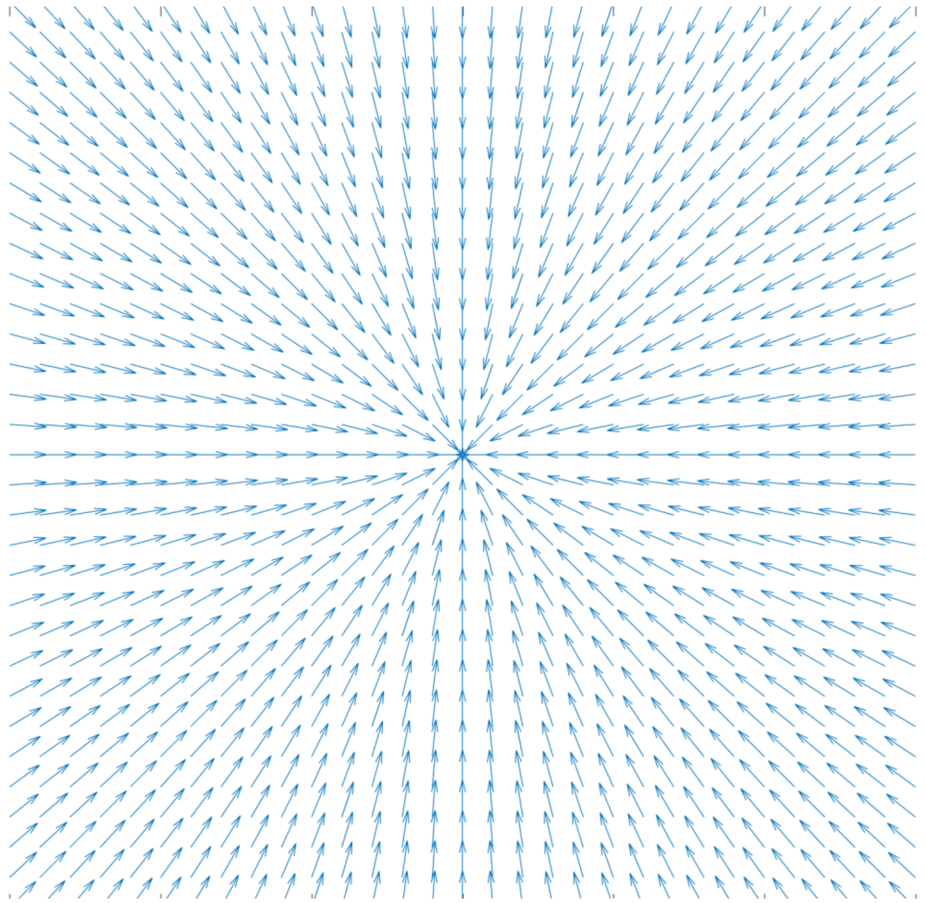
Walking along a Vector Field

- Example: 2D vector field $\boldsymbol{v}(x, y) = (-x, -y)^T$



Walking along a Vector Field

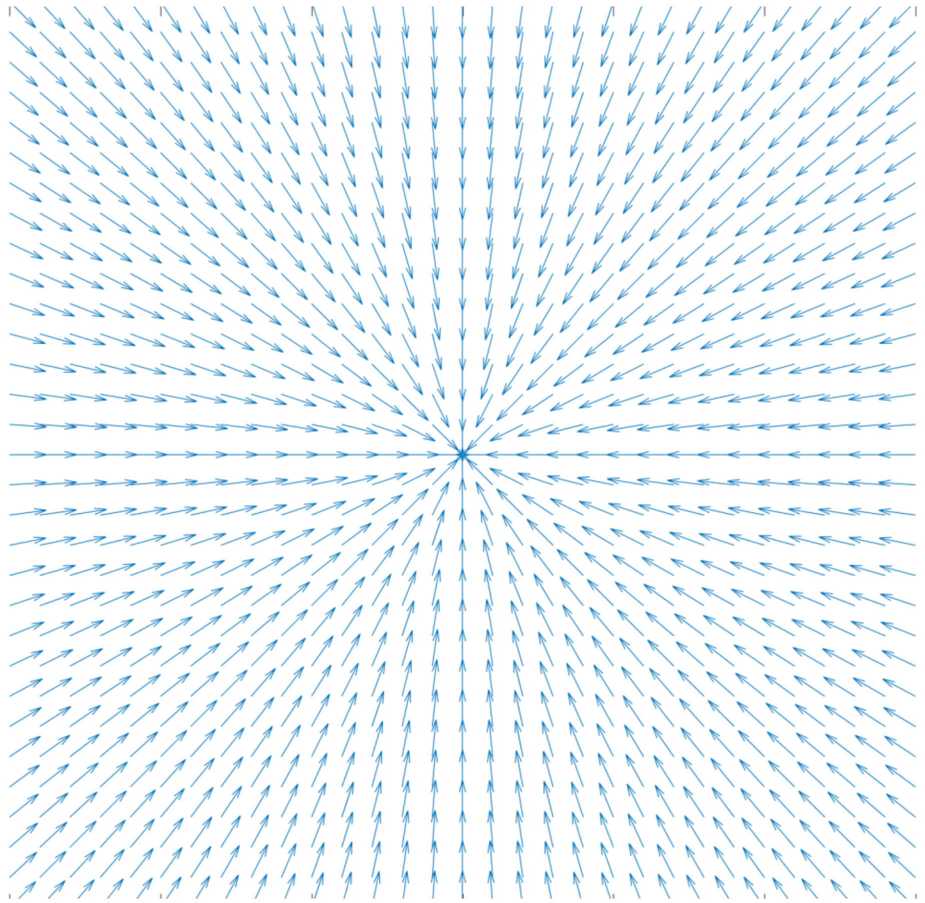
- Example: 2D vector field $\boldsymbol{v}(x, y) = (-x, -y)^T$
- Let's start at a point and walk along the vector field



Walking along a Vector Field

- Example: 2D vector field $\boldsymbol{v}(x, y) = (-x, -y)^T$
- Let's start at a point and walk along the vector field
- Start point: $\boldsymbol{x}_0 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$
- The vector field at $\boldsymbol{x}_0$ is:

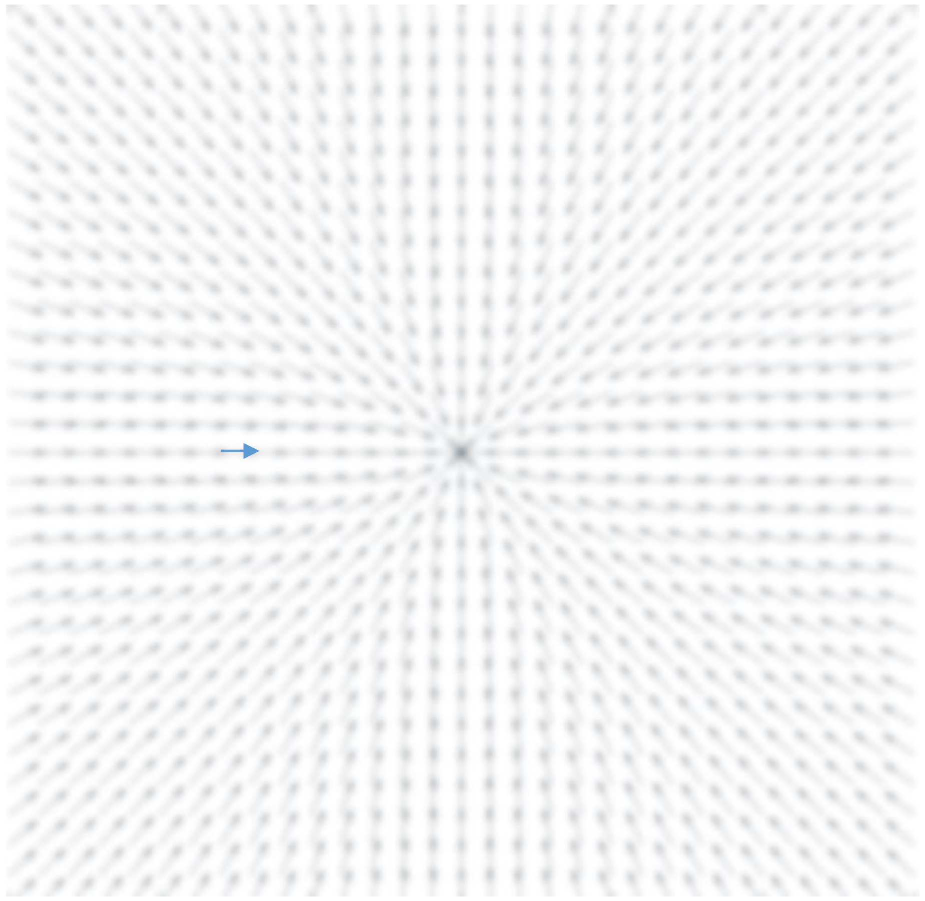
$$\boldsymbol{v}(\boldsymbol{x}_0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$



Walking along a Vector Field

- Example: 2D vector field $\boldsymbol{v}(x, y) = (-x, -y)^T$
- Let's start at a point and walk along the vector field
- Start point: $\boldsymbol{x}_0 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$
- The vector field at $\boldsymbol{x}_0$ is:

$$\boldsymbol{v}(\boldsymbol{x}_0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$



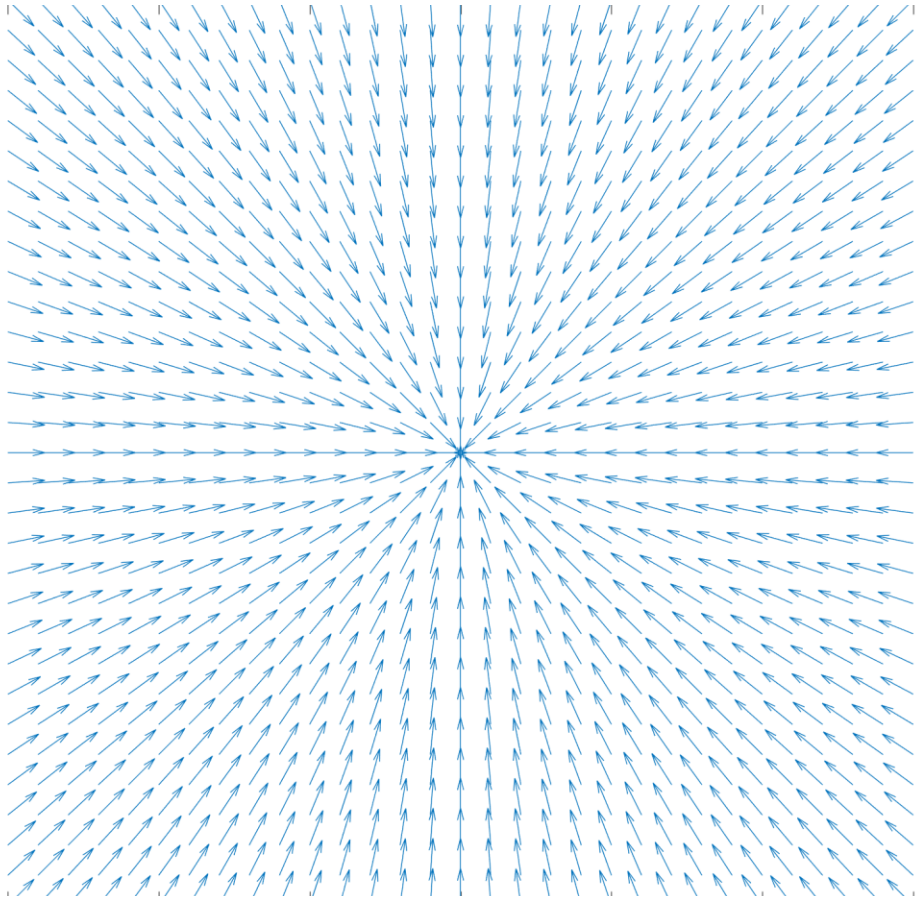
Walking along a Vector Field

- Example: 2D vector field $\boldsymbol{v}(x, y) = (-x, -y)^T$
- Start point: $\boldsymbol{x}_0 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$
- The vector field at $\boldsymbol{x}_0$ is:

$$\boldsymbol{v}(\boldsymbol{x}_0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

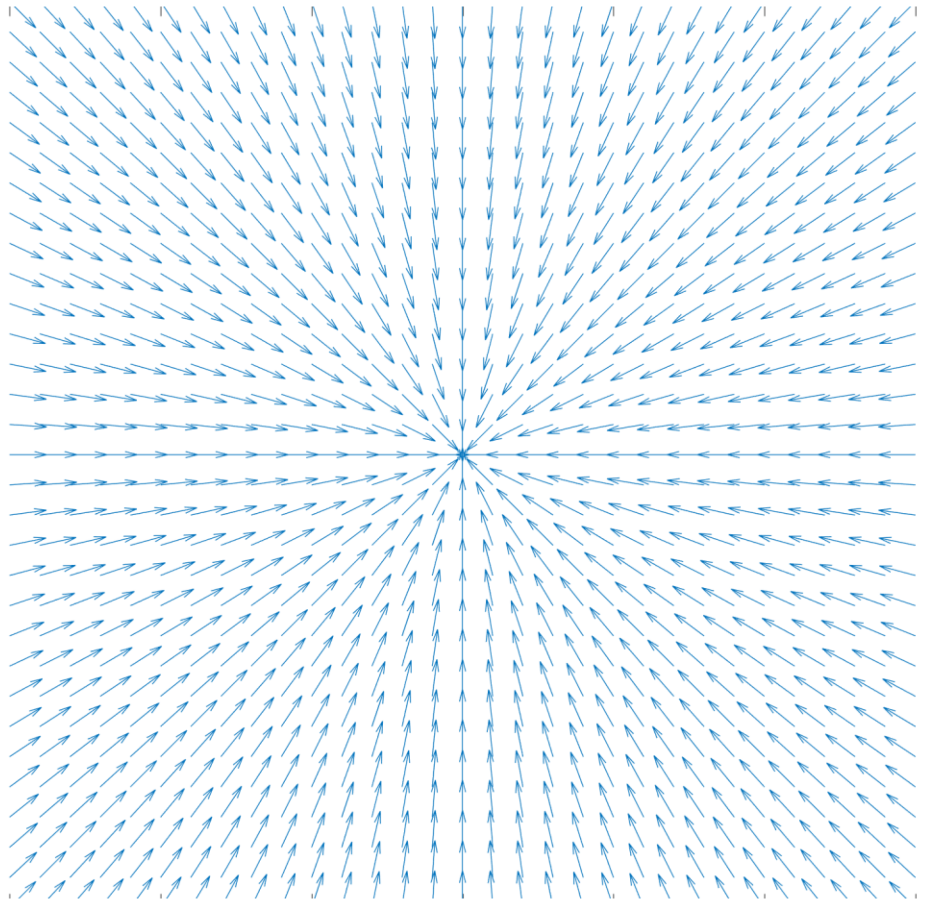
- Let's walk along this direction with $\Delta s = 0.5$:

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \Delta s \cdot \boldsymbol{v}(\boldsymbol{x}_0, t_0) = \begin{pmatrix} 0.5 \\ -1 \end{pmatrix}$$



Walking along a Vector Field

- Example: 2D vector field $\boldsymbol{v}(x, y) = (-x, -y)^T$
- If we continue with this, we land at the origin, which happens to be a sink in this vector field



Walking along a Vector Field

- Of course, we can calculate the sinks in a vector field
- As an example, we assume that the sinks cannot be determined, we want to find some sinks by walking along a vector field
- Let’s say we place hundreds of particles at different start positions in a vector field and all of them walk along the vector field
- At the end some of them may be stuck in a sink

Walking along a Vector Field

- As a simple example, we use the following vector field:

$$v(x,y) = \begin{pmatrix} \frac{1}{4}x - x^3 \\ \frac{1}{4}y - y^3 \end{pmatrix}$$

$$v(x,y) = \begin{pmatrix} \frac{1}{4}x - x^3 \\ \frac{1}{4}y - y^3 \end{pmatrix}$$

Walking along a Vector Field

- First, we define the vertex shader:

```
#version 330 core
layout(location = 0) in vec3 point;
out vec3 res;

vec3 vectorfield(vec3 p)
{
    float x = p.x;
    float y = p.y;
    vec3 res = vec3(0.25 * x - x * x * x, 0.25 * y - y * y * y, 0);
    return res;
}

void main()
{
    float stepSize = 0.1;
    res = point + stepSize * vectorfield(point);
}
```

Walking along a Vector Field

- We set the vertex shader as `const char* tf_shader_src`
- `Then, we create the shader, compile, attach, etc.`

```
int tf_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(tf_shader, 1, &tf_shader_src, NULL);
glCompileShader(tf_shader);

unsigned int tf_Program = glCreateProgram();
glAttachShader(tf_Program, tf_shader);

const GLchar* feedbackVaryings[] = { "res" };
glTransformFeedbackVaryings(tf_Program, 1, feedbackVaryings,
GL_INTERLEAVED_ATTRIBS);

glLinkProgram(tf_Program);
glDeleteShader(tf_shader);
glUseProgram(tf_Program);
```

Walking along a Vector Field

- Now, create two VAOs and two VBOs (nothing new so far):

```
unsigned int tf_vao[2];
glGenVertexArrays(2, tf_vao);

unsigned int tf_vbo[2];
glGenBuffers(2, tf_vbo);
```

Walking along a Vector Field

- To create a random number, we have to initialize the randomizer with srand:

```
srand(static_cast <unsigned> (time(0))); // #include <ctime>
```

- Afterwards, rand() create a random number in the range [0,RAND_MAX]
- This creates a random number in [-1,1]

```
-1.0f + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / (2.0f)));
```

Walking along a Vector Field

- Create 10 particles with random positions:

```
const int numParticles = 10;
float tf_data[3 * numParticles];

float tmp1, tmp2;
srand(static_cast <unsigned> (time(0)));
for (int i = 0; i < numParticles; i++)
{
    tmp1 = -1.0f + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (2.0f)));
    tmp2 = -1.0f + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (2.0f)));
    tf_data[i * 3 + 0] = tmp1;
    tf_data[i * 3 + 1] = tmp2;
    tf_data[i * 3 + 2] = 0;
}
```

Walking along a Vector Field

- Bind the VAOs and the VBOs and enable the positions:

```
for (int i = 0; i < 2; i++)
    {
        glBindVertexArray(tf_vao[i]);
        glBindBuffer(GL_ARRAY_BUFFER, tf_vbo[i]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(tf_data), tf_data, GL_DYNAMIC_COPY);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
        glEnableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindVertexArray(0);
    }
```

Walking along a Vector Field

- In the next step, we iterate 500 times: bind the VAO and apply the walking in the vector field

```
for (int i = 0; i < 500; i++)
{
    glBindVertexArray(tf_vao[i&1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tf_vbo[(i+1) & 1]);

    glBeginTransformFeedback(GL_POINTS);
    glDrawArrays(GL_POINTS, 0, numParticles);
    glEndTransformFeedback();
}
```

Walking along a Vector Field

- i&1 is a bitwise and operator:
- 11 & 1 → 1011 & 0001 = 1; 6 & 1 → 110&001 = 0

```
for (int i = 0; i < 500; i++)
{
    glBindVertexArray(tf_vao[i&1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tf_vbo[(i+1) & 1]);

    glBeginTransformFeedback(GL_POINTS);
    glDrawArrays(GL_POINTS, 0, numParticles);
    glEndTransformFeedback();
}
```

Walking along a Vector Field

- Bind the VAO und bind the VBO of the other VAO as the result of the transform feedback

```
for (int i = 0; i < 500; i++)
{
    glBindVertexArray(tf_vao[i&1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tf_vbo[(i+1) & 1]);

    glBeginTransformFeedback(GL_POINTS);
    glDrawArrays(GL_POINTS, 0, numParticles);
    glEndTransformFeedback();
}
```

Walking along a Vector Field

- Begin with the transform feedback and execute it for every particle

```
for (int i = 0; i < 500; i++)
{
    glBindVertexArray(tf_vao[i&1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tf_vbo[(i+1) & 1]);

    glBeginTransformFeedback(GL_POINTS);
    glDrawArrays(GL_POINTS, 0, numParticles);
    glEndTransformFeedback();
}
```

Walking along a Vector Field

- Finally, we end the transform feedback
- We do not delete the buffer that contains tf_vbo[0] because we want to display the particles

```
glFlush();

glDeleteProgram(tf_Program);
glDeleteShader(tf_shader);
glBindVertexArray(0);

glDeleteBuffers(1, &tf_vbo[1]);
glDeleteVertexArrays(2, tf_vao);

glDisable(GL_RASTERIZER_DISCARD);
```

Walking along a Vector Field

- For displaying the result, we use a simple vertex and fragment shader:

```
const char* vertexShaderSource = "#version 330 core\n"
"layout (location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\0";
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n\0";
```

Walking along a Vector Field

- We only need to bind the buffer with tf_tbo:

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, tf_vbo[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);

glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);
```

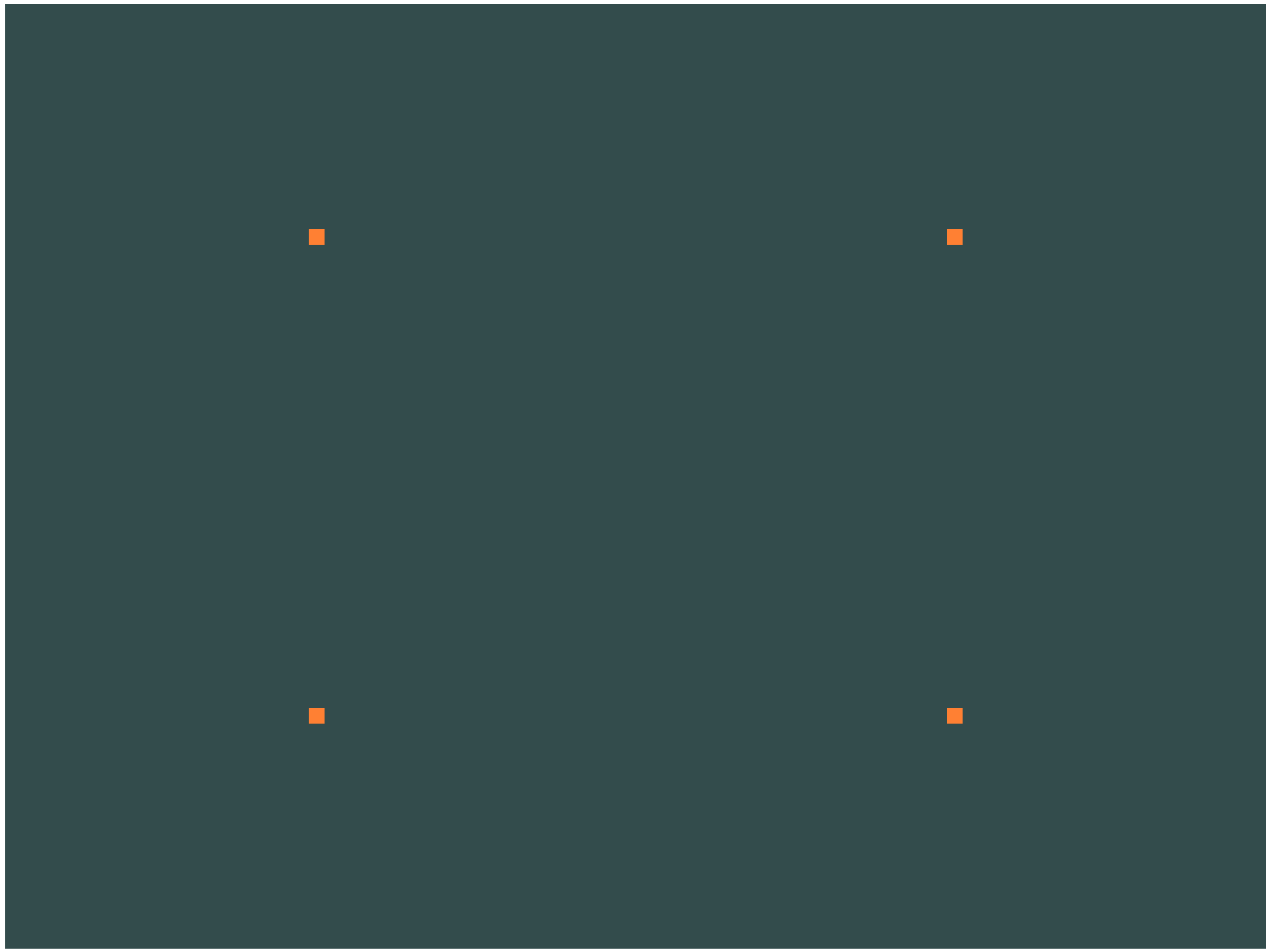
Walking along a Vector Field

- To show the results, we use points:

```
glPointSize(10);
while (!glfwWindowShouldClose(window))
{
    …
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(shaderProgram);
    glBindVertexArray(VAO);

    glDrawArrays(GL_POINTS, 0, numParticles);
    …
}
```

F5…

- … we get the correct positions



Remarks

- This yields a visual result of 10 particles
- This method does not ensure to find all sinks
- To get the positions, we have to read the values of the array, yielding (- 0.5, - 0.5); (0.5, - 0.5); (- 0.5, 0.5); (0.5, 0.5)
- This was just an example to show the use of transform feedback (although, we have other tools to determine the sinks)

Shader Storage Buffer

Introduction

- So far, we uploaded an array to the GPU, but we could only use the value of the current vertex
- For example, we generated vertex positions, but we had only access to the positions of the vertex that is currently processed in the vertex shader
- What if we want to have access to the whole array in the vertex shader, e.g., access to the position of the fifth vertex although the vertex shader process the first one

Introduction

- For this, we can use shader storage buffer
- Shader storage buffer allow to have access to the whole array in the vertex shader

Example

- Assume we have a surface/mesh that consists of vertices and the vertices have values that change over time
- You can think of the course of the temperature (how does the temperature change over the day at a certain position)
- We want to construct such an example and display the change over time

Example

- We start with an easy example
- As the basis, we draw a simple triangle (we already have this example in our portfolio – 2.1 Hello Triangle - LearnOpenGL)

Create Time Values

- We have a triangle, now we construct an array that stores the time values, e.g., the temperature over time

```
const int num_time_steps = 100;
float time_values[3 * num_time_steps];

float tmp1, tmp2, tmp3;
for (int i = 0; i < num_time_steps; i++)
{
        tmp1 = static_cast <float> (i) / (num_time_steps -1);
        tmp2 = tmp1 * tmp1;
        tmp3 = sqrt(tmp1);

        time_values[i * 3 + 0] = tmp1;
        time_values[i * 3 + 1] = tmp2;
        time_values[i * 3 + 2] = tmp3;
}
```

Create Time Values

- The first vertex has the values: 0, 1/100, 2/100, 3/100,…,1
- The second vertex has the values: 0, $(1/100)^2$,$(2/100)^2$, $(3/100)^2$,…,1
- The third vertex has the values: 0, $\sqrt{1/100}$, $\sqrt{2/100}$, $\sqrt{3/100}$,…,1

Create SSBO

- Now, we create the shader storage buffer object (SSBO):

```
unsigned int SSBO_time;

glBindBuffer(GL_SHADER_STORAGE_BUFFER, SSBO_time);
glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(time_values), time_values, GL_STATIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, SSBO_time);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

- That's all, `glBindBufferBase` uses the binding location (2. argument), we need this in the vertex shader

Vertex Shder

- We have to set the binding to 1 and, we have to set the version to 430 (SSBOs are available since OpenGL 4.3):

```
#version 430 core

layout(std430, binding = 1) buffer Time
{
    float time_values[];
};
```

Render Loop

- During rendering, we want to use a uniform that index the time value
- For this, we use `glfwGetTime`, but every 0.05s, we want to change the values

```
float lastFrame = 0.0f;
while (!glfwWindowShouldClose(window))
{
    float currentFrame = glfwGetTime();

    if (currentFrame - lastFrame > 0.05f)
    {
        …
    }
    …
}
```

Render Loop

- We use an int to address the time values:

```
int timeSteps = 0;
while (!glfwWindowShouldClose(window))
{
    if (currentFrame - lastFrame > 0.05f)
    {
        timeSteps++;
        lastFrame = currentFrame;
    }
    if (timeSteps > num_time_steps)
        timeSteps = 0;
    …
}
```

Render Loop

- A uniform is then used in the vertex shader:

```
int vertexColorLocation = glGetUniformLocation(shaderProgram,
        "timeSteps");
glUniform1i(vertexColorLocation, timeSteps);
```

- The vertex shader:

```
uniform int timeSteps;
```

Render Loop

- Render loop:

```
int timeSteps = 0;
float lastFrame = 0.0f;
while (!glfwWindowShouldClose(window))
{
    float currentFrame = glfwGetTime();
    int vertexColorLocation = glGetUniformLocation(shaderProgram,
        "timeSteps");
    glUniform1i(vertexColorLocation, timeSteps);
    if (currentFrame - lastFrame > 0.05f)
    {
        timeSteps++;
        lastFrame = currentFrame;
    }
    if (timeSteps > num_time_steps)
        timeSteps = 0;
…}
```

Vertex Shader

```
#version 430 core
layout (location = 0) in vec3 aPos;
layout(std430, binding = 1) buffer Time
{
    float time_values[];
};

uniform int timeSteps;
out float time;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    time=time_values[3*timeSteps+gl_VertexID];
}
```

Vertex Shader

```
time=time_values[3*timeSteps+gl_VertexID];
```

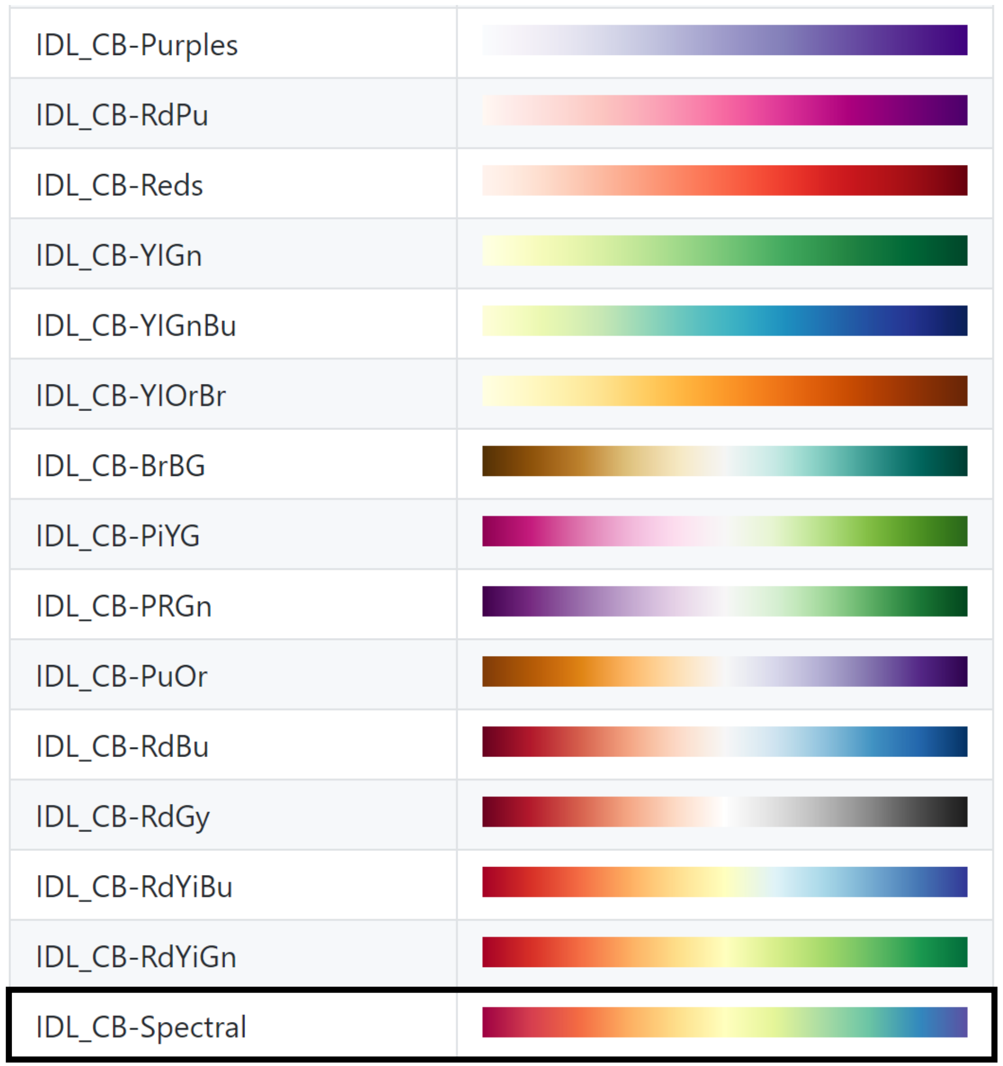
- Gl_VertexID is an in-built variable that addresses the index of the vertex
- If we are at time step 10, then the time value for vertex 0 is at `time_values[30+0]` and the value for vertex 2 is at `time_values[30+2]`

Fragment Shader

```
#version 330 core
in float time;
out vec4 FragColor;

vec4 colormap(float x)
{…}

void main()
{
FragColor = colormap(time);
}
```

Fragment Shader

- We use kbinani's colormaps on github
- The shader code is available for the different colormaps
- We will go with IDL_CB-Spectral



IDL_CB-Purples
IDL_CB-RdPu
IDL_CB-Reds
IDL_CB-YlGn
IDL_CB-YlGnBu
IDL_CB-YlOrBr
IDL_CB-BrBG
IDL_CB-PiYG
IDL_CB-PRGn
IDL_CB-PuOr
IDL_CB-RdBu
IDL_CB-RdGy
IDL_CB-RdYiBu
IDL_CB-RdYiGn
IDL_CB-Spectral

https://github.com/kbinani/colormap-shaders

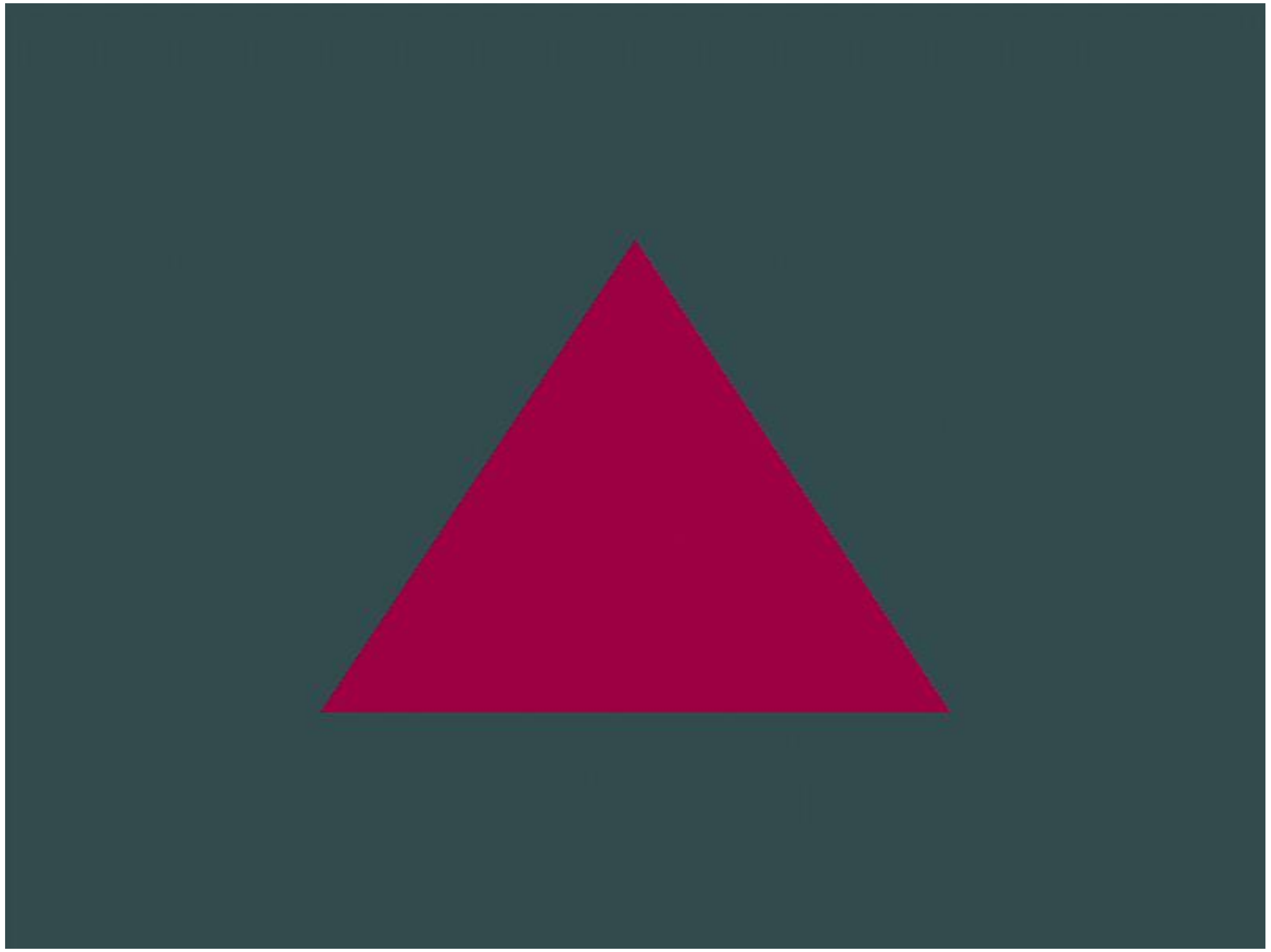
Fragment Shader

- Copy these functions to the fragment shader

```
float colormap_red(float x) {
if (x < 0.09752005946586478) {
return 5.63203907203907E+02 * x + 1.57952380952381E+02;
} else if (x < 0.2005235116443438) {
return 3.02650769230760E+02 * x + 1.83361538461540E+02;
} else if (x < 0.2974133397506856) {
return 9.21045429665647E+01 * x + 2.25581007115501E+02;
} else if (x < 0.5003919130598823) {
return 9.84288115246108E+00 * x + 2.50046722689075E+02;
} else if (x < 0.5989021956920624) {
return -2.48619704433547E+02 * x + 3.79379310344861E+02;
} else if (x < 0.902860552072525) {
return ((2.76764884219295E+03 * x - 6.08393126459837E+03) * x + 3.80008072407485E+03) * x - 4.57725185424742E+02;
} else {
return 4.27603478260530E+02 * x - 3.35293188405479E+02;
}
}

float colormap_green(float x) {
if (x < 0.09785836420571035) {
return 6.23754529914529E+02 * x + 7.26495726495790E-01;
} else if (x < 0.2034012006283468) {
return 4.60453201970444E+02 * x + 1.67068965517242E+01;
} else if (x < 0.302409765476316) {
return 6.61789401709441E+02 * x - 2.42451282051364E+01;
} else if (x < 0.4005965758690823) {
return 4.82379130434784E+02 * x + 3.00102898550747E+01;
} else if (x < 0.4981907026473237) {
return 3.24710622710631E+02 * x + 9.31717541717582E+01;
} else if (x < 0.6064345916502067) {
return -9.64699507389807E+01 * x + 3.03000000000023E+02;
} else if (x < 0.7987472620841592) {
return -2.54022986425337E+02 * x + 3.98545610859729E+02;
} else {
return -5.71281628959223E+02 * x + 6.51955082956207E+02;
}
}

float colormap_blue(float x) {
if (x < 0.0997359608740309) {
return 1.26522393162393E+02 * x + 6.65042735042735E+01;
} else if (x < 0.1983790695667267) {
return -1.22037851037851E+02 * x + 9.12946682946686E+01;
} else if (x < 0.4997643530368805) {
return (5.39336225400169E+02 * x + 3.55461986381562E+01) * x + 3.88081126069087E+01;
} else if (x < 0.6025972254407099) {
return -3.79294261294313E+02 * x + 3.80837606837633E+02;
} else if (x < 0.6990141388105746) {
return 1.15990231990252E+02 * x + 8.23805453805459E+01;
} else if (x < 0.8032653181119567) {
return 1.68464957265204E+01 * x + 1.51683418803401E+02;
} else if (x < 0.9035796343050095) {
return 2.40199023199020E+02 * x - 2.77279202279061E+01;
} else {
return -2.78813846153774E+02 * x + 4.41241538461485E+02;
}
}

vec4 colormap(float x) {
float r = clamp(colormap_red(x) / 255.0, 0.0, 1.0);
float g = clamp(colormap_green(x) / 255.0, 0.0, 1.0);
float b = clamp(colormap_blue(x) / 255.0, 0.0, 1.0);
return vec4(r, g, b, 1.0);
}
```

F5…

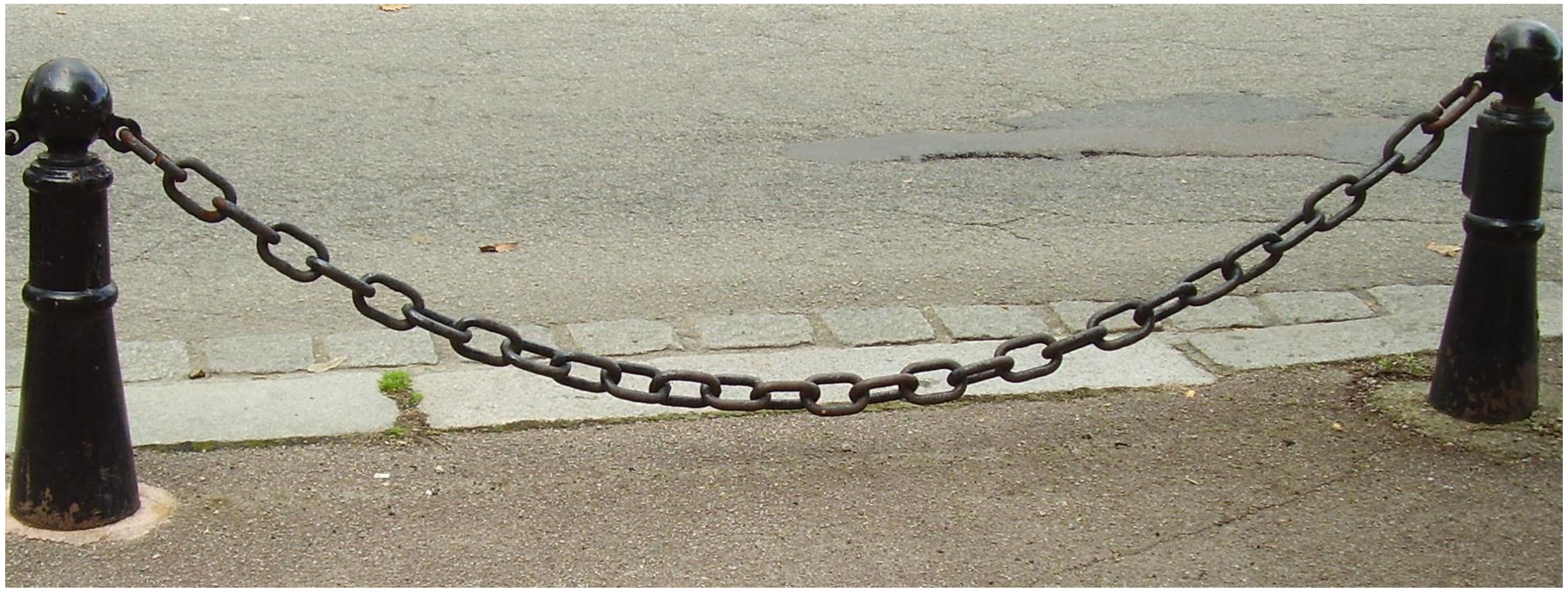
- … a color changing triangle!



Catenary

Introduction

- A chain line/catenary is a mathematical curve that describes the course of a chain with fixed ends under the influence of gravity



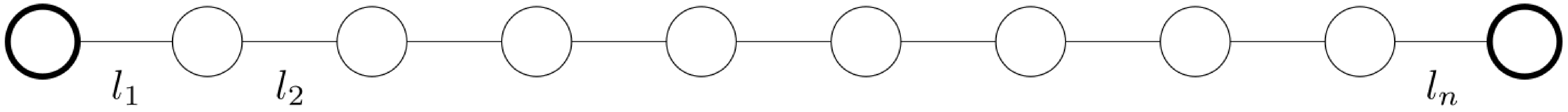
Introduction

- The solution of this can mathematically be solved
- We want to simulate this problem
- Using transform feedback and shader storage buffers, we want to get the solution, too

Simulation

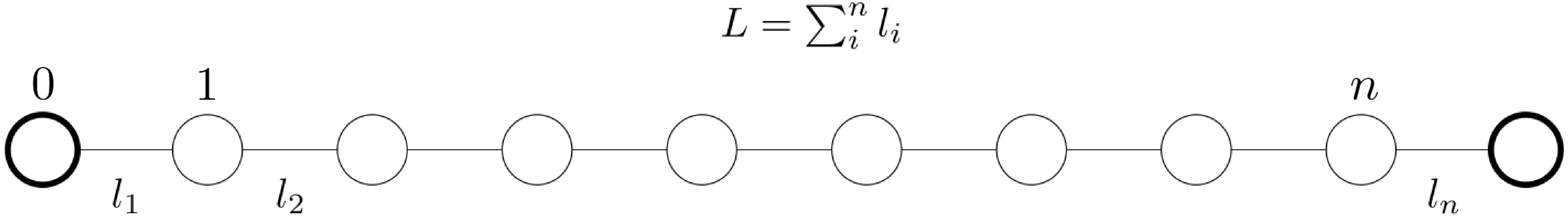
- We use segments, which are connected by two particles
- The whole line has a given length L
- We say that the particles are evenly distributed such that $l_i = L/n$

$$L = \sum_i^n l_i$$



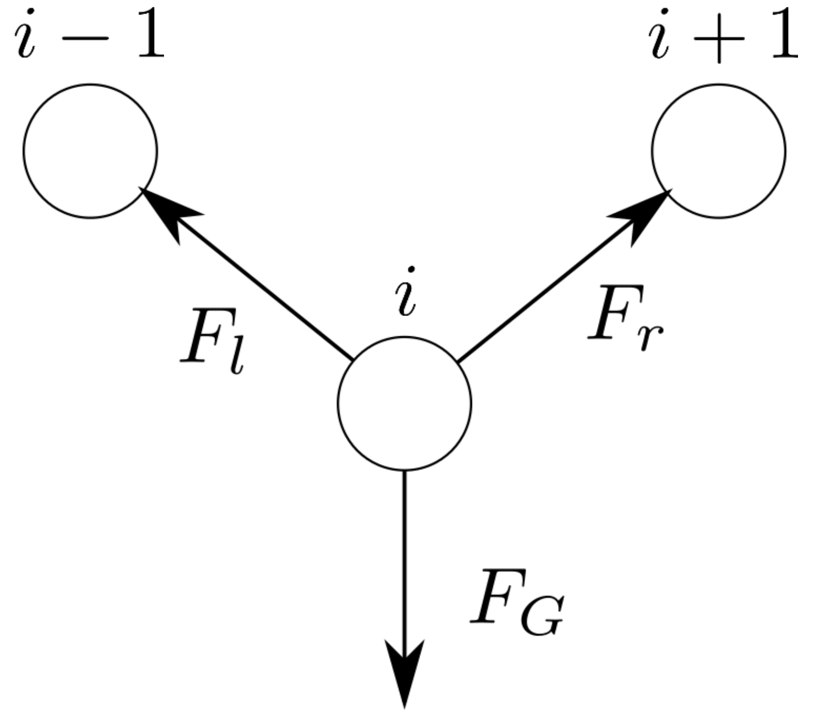
Simulation

- Every particle i ($\neq 0, n+1$) can be moved, but the length of incident segments (l_i, l_{i+1}) should be L/n



Simulation

- Each particle i is subject to forces: weight force, forces exerted by springs
- The forces F_l, F_r should ensure that the segments keep their lengths

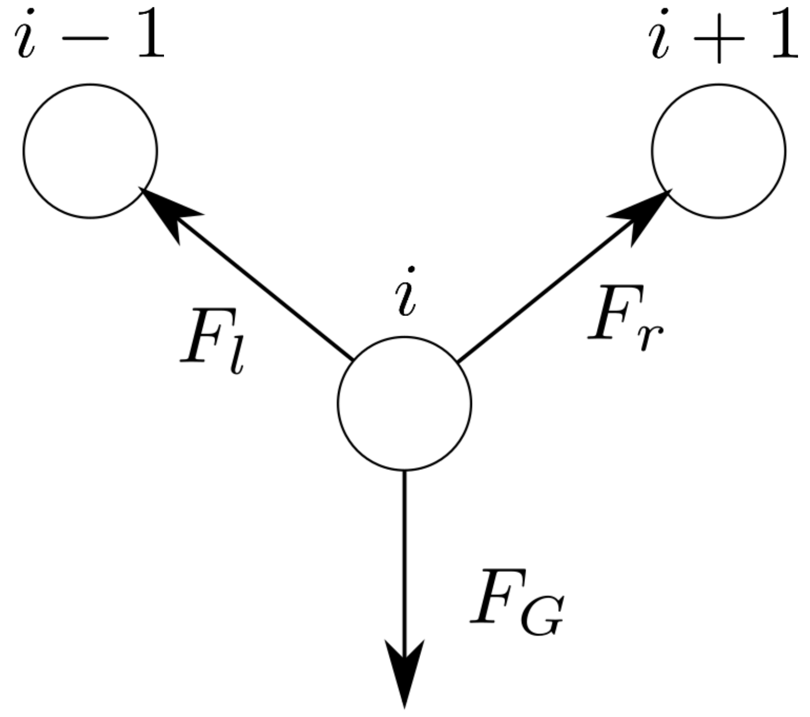


Simulation

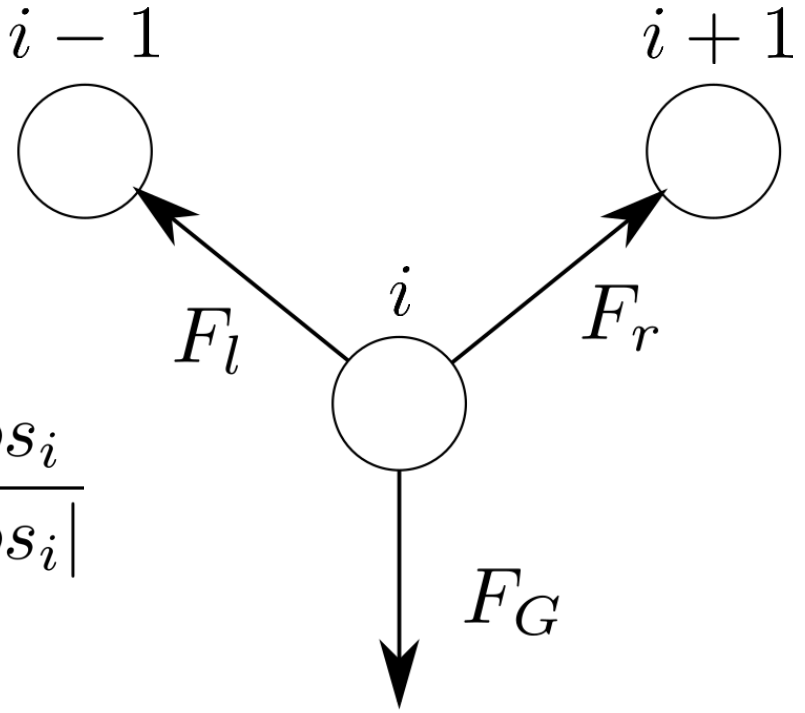
- We model the forces such that:

$$F_G = (0, -c, 0)$$

- Here, c is a constant such that the particle under the influence of gravity is pushed downwards



Simulation



- We model the forces such that:

$$F_{r/l} = \max\left(\frac{|pos_{i\pm1} - pos_i|}{L/n} - 1, 0\right) \cdot \frac{pos_{i\pm1} - pos_i}{|pos_{i\pm1} - pos_i|}$$

- Whenever the distance of a segment exceeds $\frac{L}{n}$, we have a positive argument in the max function, yielding a value greater than 0, 0 otherwise
- The force pushes the particle to the neighbor

Simulation – Vertex Shader

- For simplicity, we say:
  - $L/n = 1$
  - $c = 0.01$
  - Every 50. iteration, we move the particle downwards (otherwise correct the position of the particles such that the segment lengths are not violated)

Simulation – Vertex Shader

- Vertex shader:

```
#version 430 core
layout(std430, binding = 1) buffer Point
{
    vec4 pointSSBO[]; // DO NOT USE VEC3, only float, vec2 or vec4
};

uniform int numSegments;
uniform int iteration;

float len=1.0f;
out vec4 res;
```

Simulation – Vertex Shader

- Vertex shader:

```
void main()
{
   res=pointSSBO[gl_VertexID];

   if(gl_VertexID>0 && gl_VertexID<numSegments-1)
   {
      vec4 forceDown=vec4(0,-0.01,0,0);

      if(mod(iteration,50)==0)
         res +=forceDown;
      else
      {
```

Simulation – Vertex Shader

- Vertex shader:

```
          vec4 pos=pointSSBO[gl_VertexID];
          vec4 pos_l=pointSSBO[gl_VertexID-1];
          vec4 pos_r=pointSSBO[gl_VertexID+1];

          float dist_l=length(pos_l-pos);
          float dist_r=length(pos_r-pos);

          vec4 force_l=max(dist_l/len-1,0)*normalize(pos_l-pos);
          vec4 force_r=max(dist_r/len-1,0)*normalize(pos_r-pos);

          res += force_l + force_r;
          }
    }
}
```

Buffer

- Again, we generate two VAOs and two VBOs (SSBO):

```
unsigned int tf_vao[2];
glGenVertexArrays(2, tf_vao);

unsigned int tf_ssbo_vbo[2];
glGenBuffers(2, tf_ssbo_vbo);

for (int i = 0; i < 2; i++)
{
   glBindVertexArray(tf_vao[i]);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, tf_ssbo_vbo[i]);
   glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(tf_data), tf_data, GL_STATIC_READ);
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, tf_ssbo_vbo[i]);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
   glBindVertexArray(0);
}
```

Init Data

- Initialize the particles:

```
const int numSegments = 21;
float tf_data[4 * numSegments];

for (int i = 0; i < numSegments; i++)
{
    tf_data[i * 4 + 0] = -1.0f + 2.0f * static_cast<float>(i) / (numSegments - 1);
    tf_data[i * 4 + 1] = 0.0f;
    tf_data[i * 4 + 2] = 0.0f;
    tf_data[i * 4 + 3] = 0.0f;
}
```

Iteration

- Iterate over the vertex shader:

```
int numSegmentsLocation = glGetUniformLocation(tf_Program, "numSegments");
glUniform1i(numSegmentsLocation , numSegments);
glEnable(GL_RASTERIZER_DISCARD);
int iterationLocation= glGetUniformLocation(tf_Program, "iteration");
for (int i = 0; i < 100000; i++)
{
   glUniform1i(iterationLocation, i);
   glBindVertexArray(tf_vao[i & 1]);
   glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tf_ssbo_vbo[(i + 1) & 1]);
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, tf_ssbo_vbo[i & 1]);

   glBeginTransformFeedback(GL_POINTS);
   glDrawArrays(GL_POINTS, 0, numSegments);
   glEndTransformFeedback();
}
```

Delete

- Delete everything and activate the rasterizer:

```
glFlush();
glDeleteProgram(tf_Program);
glDeleteShader(tf_shader);
glDeleteBuffers(1, &tf_ssbo_vbo[1]);
glDeleteVertexArrays(2, tf_vao);
glDisable(GL_RASTERIZER_DISCARD);
```

Draw the Rest

- Generate VAO and bind VBO:

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, tf_ssbo_vbo[0]);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

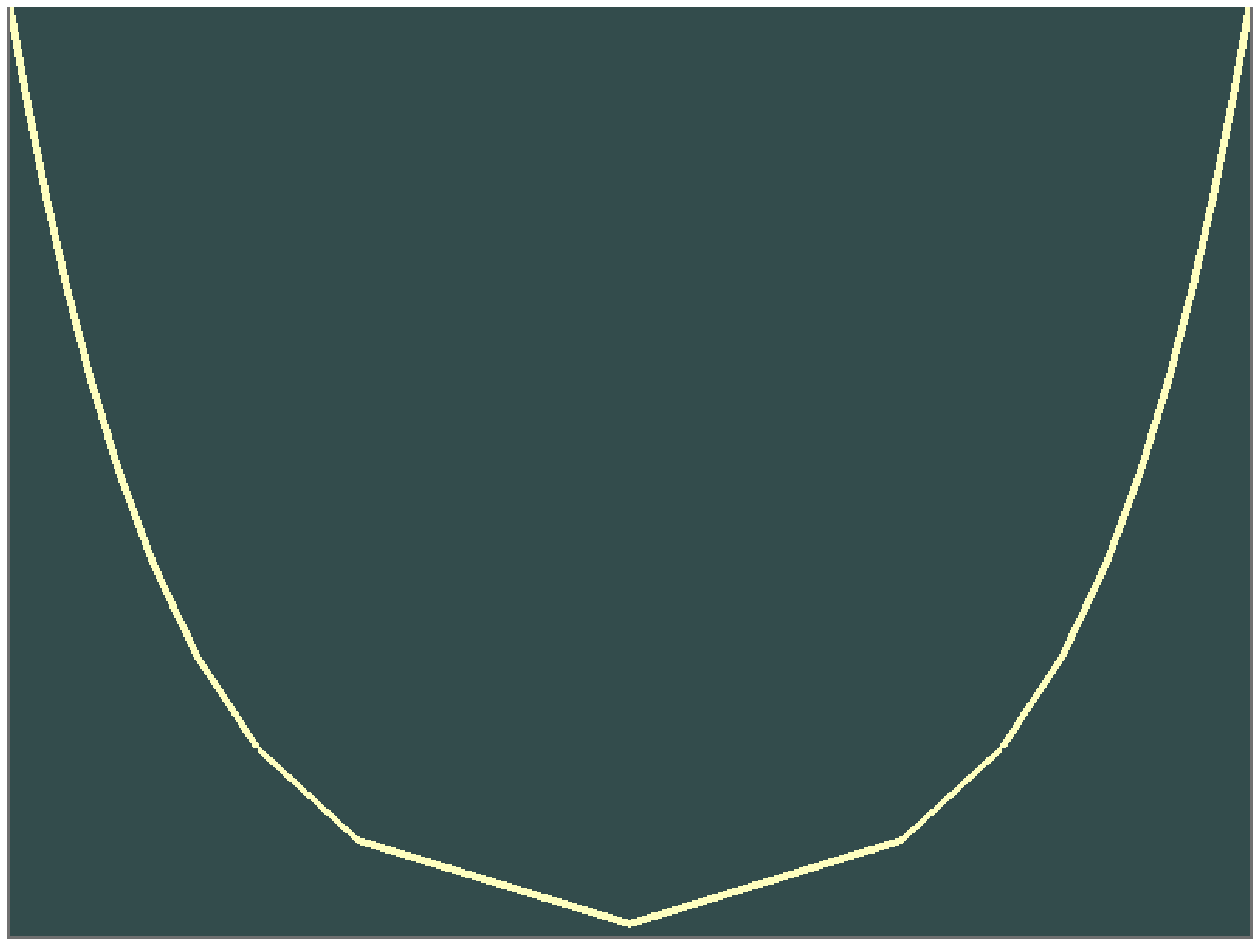
Draw the Rest

- Draw:

```
glLineWidth(5);
while (!glfwWindowShouldClose(window))
{
    …
    glDrawArrays(GL_LINE_STRIP, 0, numSegments);
    …
}
```

F5…

- … that’s nice!



Remarks

- This simulation is just a simplification (although the results are very close to the real solution)
- It is not very stable; it may be that for certain parameters we will run into nan values

Questions???