Материалы к занятию

Для начала создадим пустое окно

```
import arcade

SCREEN_WIDTH = 1024
SCREEN_HEIGHT = 768
SCREEN_TITLE = "GPU Particle Explosion"


class MyWindow(arcade.Window):
   def __init__(self):
       super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT,
SCREEN_TITLE)

   def on_draw(self):
       self.clear()

   def on_update(self, dt):
       pass

   def on_mouse_press(self, x: float, y: float, button: int,
modifiers: int):
       pass


if __name__ == "__main__":
   window = MyWindow()
   window.center_window()
   arcade.run()
```

Мы будем рисовать точку каждый раз, когда пользователь щелкает мышью на экране.
Для каждого клика мы создадим экземпляр класса, который в конечном итоге превратится в полный взрыв. Каждый экземпляр пакета будет добавлен в список.

Произведем необходимые импорты

```
from array import array
from dataclasses import dataclass
import arcade
import arcade.gl
```

Создадим класс данных. Для каждого взрыва нам нужно отслеживать объект Vertex Array Object, который хранит информацию о нем. Внутри будет объект буфера вершин (VBO), где мы будем хранить местоположения, цвета, скорость и т. д.

```
@dataclass
class Burst:
   buffer: arcade.gl.Buffer
   vao: arcade.gl.Geometry
```

Создадим пустой атрибут списка. Также создадим нашу программу шейдеров OpenGL. Программа будет представлять собой коллекцию из двух шейдерных программ. Они будут храниться в отдельных файлах, сохраненных в одном каталоге.

```
class MyWindow(arcade.Window):
   def __init__(self):
       super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT,
SCREEN_TITLE)
       self.burst_list = []

       self.program = self.ctx.load_program(
           vertex_shader="vertex_shader_v1.glsl",
           fragment_shader="fragment_shader.glsl",
       )

       self.ctx.enable_only()
```

OpenGL (GLSL) — это язык в стиле C, который работает на вашей видеокарте (GPU), а не на процессоре.
У нас будет два шейдера. Вершинный шейдер и фрагментный шейдер. Вершинный шейдер выполняется для каждой вершинной точки геометрии, которую мы визуализируем, а шейдер фрагмента выполняется для каждого пикселя. Например, вершинный шейдер может выполняться четыре раза для каждой точки на прямоугольнике, а шейдер фрагмента будет выполняться для каждого пикселя на экране.
Вершинный шейдер принимает положение нашей вершины. Мы установим и передадим эти данные этому шейдеру.
Вершинный шейдер выводит цвет нашей вершины. Цвета выполнены в формате Красно-Зеленый-Синий-Альфа (RGBA) с дробными числам в диапазоне от 0 до 1.

Создадим файл vertex_shader_v1.glsl и добавим в него следующий код

```
#version 330

// (x, y) position passed in
in vec2 in_pos;

// Output the color to the fragment shader
out vec4 color;
```

```
void main() {

   // Set the RGBA color
   color = vec4(1, 1, 1, 1);

   // Set the position. (x, y, z, w)
   gl_Position = vec4(in_pos, 0.0, 1);
}
```

Создадим файл fragment_shader.glsl и добавим код

```
#version 330

// Color passed in from the vertex shader
in vec4 color;

// The pixel we are writing to in the framebuffer
out vec4 fragColor;

void main() {

   // Fill the point
   fragColor = vec4(color);
}
```

Каждый раз, происходит нажатие кнопку мыши, создается взрыв в этом месте.
Данные будут храниться в экземпляре класса Burst
Классу нужен буфер данных. Буфер данных содержит информацию о каждой частице. В этом случае у нас есть только одна частица, и нам нужно только хранить x, у этой частицы в буфере. Однако в конечном итоге у нас будут сотни частиц, каждая из которых имеет положение, скорость, цвет. Чтобы приспособиться к созданию этих данных, создадим функцию генератора.

```
def on_mouse_press(self, x: float, y: float, button: int,
modifiers: int):
   def _gen_initial_data(initial_x, initial_y):
       yield initial_x
       yield initial_y

   x2 = x / self.width * 2. - 1.
   y2 = y / self.height * 2. - 1.

   initial_data = _gen_initial_data(x2, y2)

   buffer = self.ctx.buffer(data=array('f', initial_data))

   buffer_description = arcade.gl.BufferDescription(buffer,
```

```
                                                        '2f',
                                                        ['in_pos'])
    vao = self.ctx.geometry([buffer_description])

    burst = Burst(buffer=buffer, vao=vao)
    self.burst_list.append(burst)
```

Отрисуем все в on_draw

```
def on_draw(self):
    self.clear()

    self.ctx.point_size = 2 * self.get_pixel_ratio()

    for burst in self.burst_list:
        burst.vao.render(self.program, mode=self.ctx.POINTS)
```

Займемся тем, чтобы было более одной частицы и заставить частицы двигаться. Мы сделаем это, создав частицы и рассчитав, где они должны быть на основе времени с момента создания. Это немного отличается от того, как мы перемещаем спрайты.
Произведем импорт библиотек

```
import random
import time
```

Также создадим константу хранящую количество частиц.

```
PARTICLE_COUNT = 300
```

Нам нужно будет добавить время в на класс взрыва. Это будет дробное число.

```
@dataclass
class Burst:
   buffer: arcade.gl.Buffer
   vao: arcade.gl.Geometry
   start_time: float
```

Теперь, когда мы создается взрыв, нам нужно несколько частиц, и каждая частица также нуждается в скорости. Добавим цикл для каждой частицы, а также выведем дельта x и y.
Важно: Из-за того, как мы устанавливаем дельту x и дельту y, частицы будут расширяться в прямоугольник, а не в круг. Мы исправим это далее.
Поскольку мы добавим скорость, нашему буферу теперь нужны две пары плавающих чисел.
Изменим метод нажатия на мышь

```
def on_mouse_press(self, x: float, y: float, button: int,
modifiers: int):
   def _gen_initial_data(initial_x, initial_y):
       for i in range(PARTICLE_COUNT):
           dx = random.uniform(-.2, .2)
           dy = random.uniform(-.2, .2)
           yield initial_x
           yield initial_y
           yield dx
           yield dy

   x2 = x / self.width * 2. - 1.
   y2 = y / self.height * 2. - 1.
   initial_data = _gen_initial_data(x2, y2)
   buffer = self.ctx.buffer(data=array('f', initial_data))
```

```
   buffer_description = arcade.gl.BufferDescription(buffer,
                                                    '2f 2f',
                                                    ['in_pos',
'in_vel'])
   vao = self.ctx.geometry([buffer_description])
   burst = Burst(buffer=buffer, vao=vao, start_time=time.time())
   self.burst_list.append(burst)
```

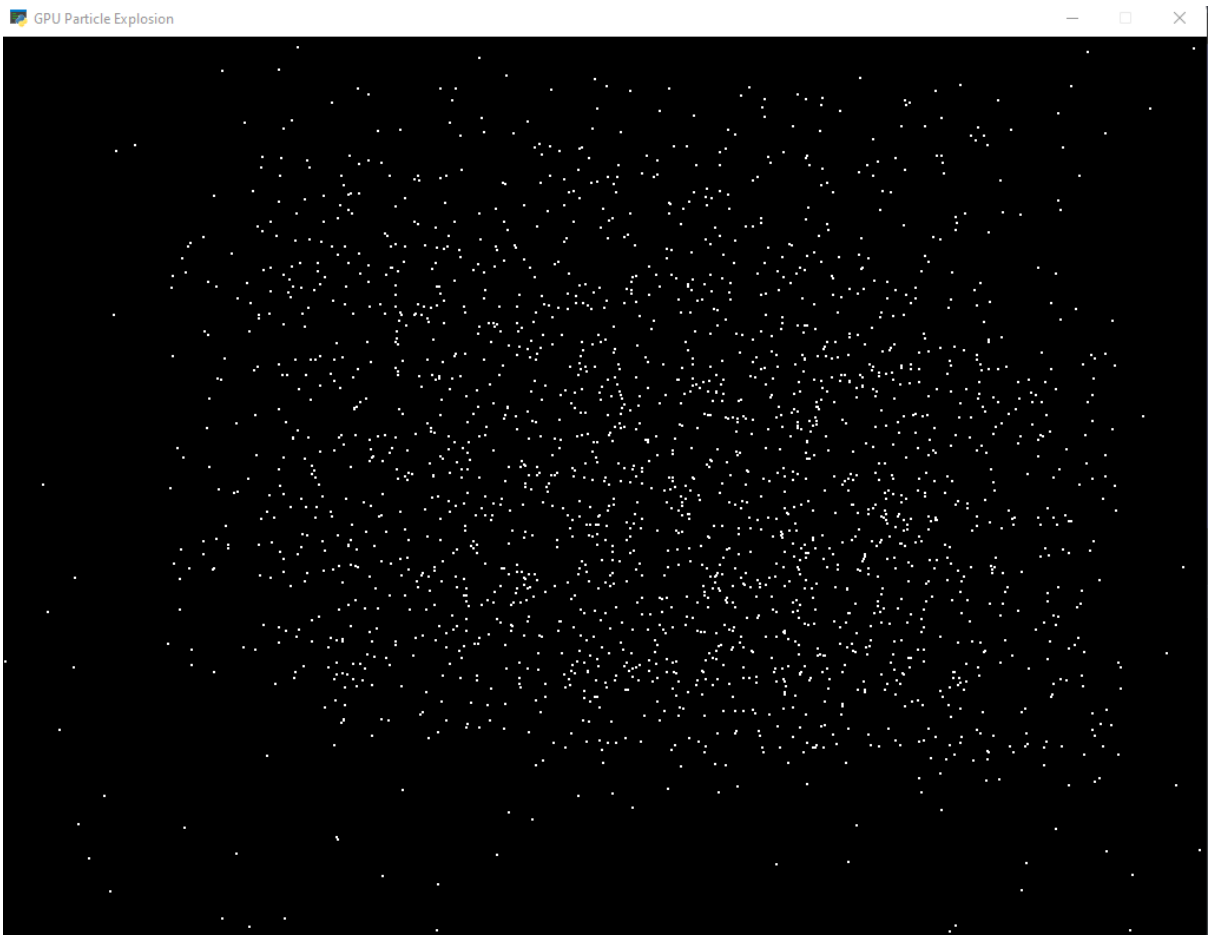
Теперь также изменим on_draw

```
def on_draw(self):
   self.clear()

   self.ctx.point_size = 2 * self.get_pixel_ratio()

   for burst in self.burst_list:
       self.program['time'] = time.time() - burst.start_time
       burst.vao.render(self.program, mode=self.ctx.POINTS)
```

И обновим наш файл vertex_shader_v1.glsl

```
#version 330


// Time since burst start

uniform float time;


// (x, y) position passed in
in vec2 in_pos;


// Velocity of particle

in vec2 in_vel;


// Output the color to the fragment shader
out vec4 color;

void main() {

   // Set the RGBA color
   color = vec4(1, 1, 1, 1);


   // Calculate a new position
```

```
    vec2 new_pos = in_pos + (time * in_vel);

    // Set the position. (x, y, z, w)
    gl_Position = vec4(new_pos, 0.0, 1);
}
```



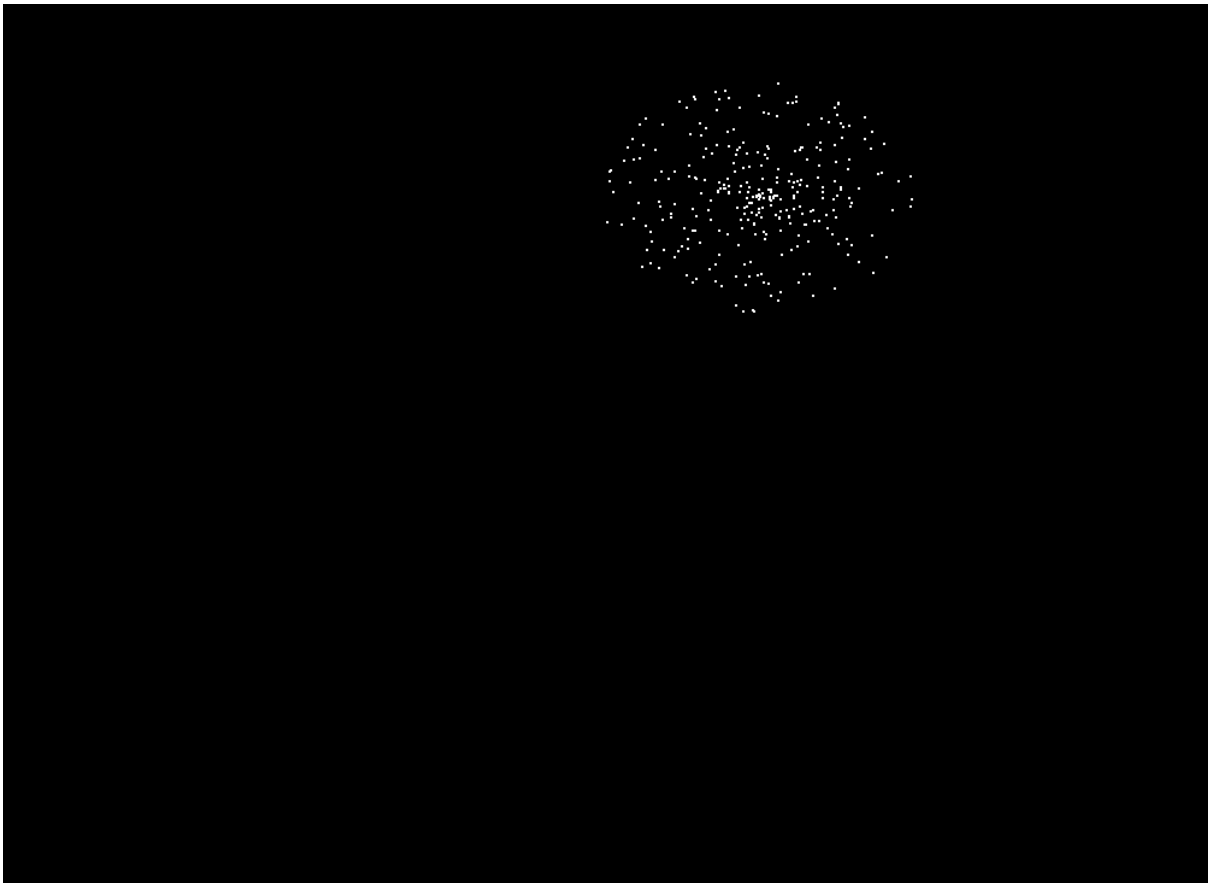
Теперь исправим проблему с прямоугольным взрывом и превратив его в круг

```
import math
```

Обновим on_mouse_press

```
def on_mouse_press(self, x: float, y: float, button: int,
modifiers: int):
   def _gen_initial_data(initial_x, initial_y):
       for i in range(PARTICLE_COUNT):
           angle = random.uniform(0, 2 * math.pi)
           speed = random.uniform(0.0, 0.3)
           dx = math.sin(angle) * speed
           dy = math.cos(angle) * speed
           yield initial_x
           yield initial_y
           yield dx
           yield dy

   x2 = x / self.width * 2. - 1.
   y2 = y / self.height * 2. - 1.
   initial_data = _gen_initial_data(x2, y2)
   buffer = self.ctx.buffer(data=array('f', initial_data))
   buffer_description = arcade.gl.BufferDescription(buffer,
                                                    '2f 2f',
                                                    ['in_pos',
'in_vel'])
   vao = self.ctx.geometry([buffer_description])
   burst = Burst(buffer=buffer, vao=vao, start_time=time.time())
   self.burst_list.append(burst)
```

Проверяем! Все отлично



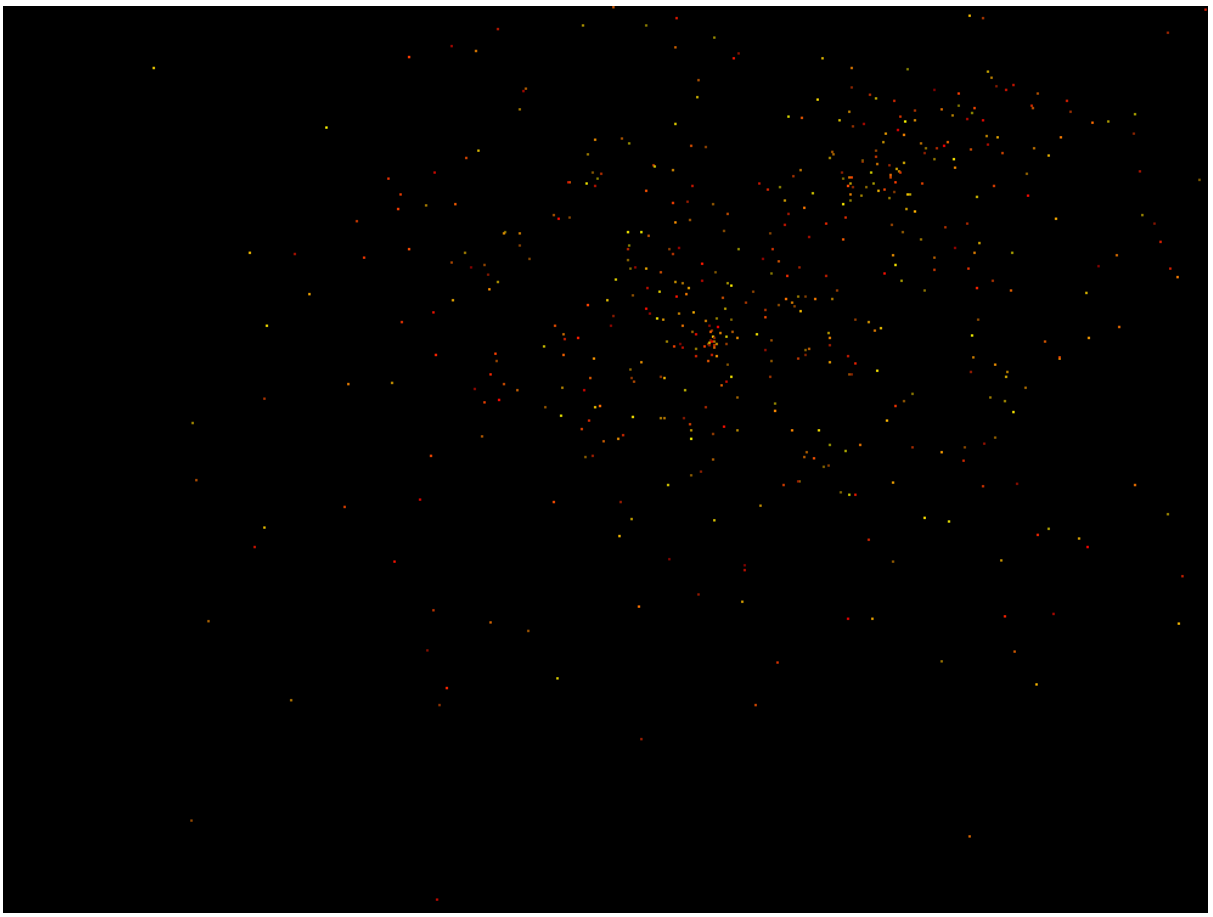
До сих пор все наши частицы были белыми. Как мы добавляем цвет? Нам нужно будет сгенерировать его для каждой частицы. Шейдеры принимают цвета в виде RGB, поэтому мы сгенерируем случайное число для красного и добавим немного зеленого, чтобы получить желтый. Наконец, передадим значения в буфер шейдера (VBO).

```
def on_mouse_press(self, x: float, y: float, button: int,
modifiers: int):
   def _gen_initial_data(initial_x, initial_y):
       for i in range(PARTICLE_COUNT):
           angle = random.uniform(0, 2 * math.pi)
           speed = abs(random.gauss(0, 1)) * .5
           dx = math.sin(angle) * speed
           dy = math.cos(angle) * speed
           red = random.uniform(0.5, 1.0)
           green = random.uniform(0, red)
           blue = 0
           yield initial_x
           yield initial_y
           yield dx
           yield dy
           yield red
           yield green
           yield blue

   x2 = x / self.width * 2. - 1.
   y2 = y / self.height * 2. - 1.

   initial_data = _gen_initial_data(x2, y2)

   buffer = self.ctx.buffer(data=array('f', initial_data))
   buffer_description = arcade.gl.BufferDescription(buffer,
                                                    '2f 2f 3f',
                                                    ['in_pos',
'in_vel', 'in_color'])
   vao = self.ctx.geometry([buffer_description])
   burst = Burst(buffer=buffer, vao=vao, start_time=time.time())
   self.burst_list.append(burst)
```

обновим наш файл vertex_shader_v1.glsl

```
#version 330

// Time since burst start
uniform float time;
```

```
// (x, y) position passed in
in vec2 in_pos;

// Velocity of particle
in vec2 in_vel;


// Color of particle

in vec3 in_color;


// Output the color to the fragment shader
out vec4 color;

void main() {


   // Set the RGBA color

   color = vec4(in_color[0], in_color[1], in_color[2], 1);


   // Calculate a new position
   vec2 new_pos = in_pos + (time * in_vel);

   // Set the position. (x, y, z, w)
   gl_Position = vec4(new_pos, 0.0, 1);
}
```



Сейчас частицы взрыва летают вечно. Заставим их исчезнуть. Как только взрыв исчезнет, удалим его.
Добавим константы для минимального и максимального времени

```
MIN_FADE_TIME = 0.25
MAX_FADE_TIME = 1.5
```

Далее изменим строчку

```
self.ctx.enable_only()
```

на

```
self.ctx.enable_only(self.ctx.BLEND)
```

Добавим скорость затухания наших частиц в on_mouse_press

```
def on_mouse_press(self, x: float, y: float, button: int,
modifiers: int):
   def _gen_initial_data(initial_x, initial_y):
```

```
        for i in range(PARTICLE_COUNT):
            angle = random.uniform(0, 2 * math.pi)
            speed = abs(random.gauss(0, 1)) * .5
            dx = math.sin(angle) * speed
            dy = math.cos(angle) * speed
            red = random.uniform(0.5, 1.0)
            green = random.uniform(0, red)
            blue = 0
            fade_rate = random.uniform(1 / MAX_FADE_TIME, 1 /
MIN_FADE_TIME)

            yield initial_x
            yield initial_y
            yield dx
            yield dy
            yield red
            yield green
            yield blue
            yield fade_rate

    x2 = x / self.width * 2. - 1.
    y2 = y / self.height * 2. - 1.

    initial_data = _gen_initial_data(x2, y2)

    buffer = self.ctx.buffer(data=array('f', initial_data))
    buffer_description = arcade.gl.BufferDescription(buffer,
                                                     '2f 2f 3f f',
                                                     ['in_pos',
                                                      'in_vel',
                                                      'in_color',

'in_fade_rate'])
    vao = self.ctx.geometry([buffer_description])
    burst = Burst(buffer=buffer, vao=vao, start_time=time.time())
    self.burst_list.append(burst)
```

Обновим наш шейдер

```
#version 330

// Time since burst start
uniform float time;

// (x, y) position passed in
in vec2 in_pos;

// Velocity of particle
in vec2 in_vel;

// Color of particle
```

```
in vec3 in_color;

// Fade rate
in float in_fade_rate;

// Output the color to the fragment shader
out vec4 color;

void main() {

   // Calculate alpha based on time and fade rate
   float alpha = 1.0 - (in_fade_rate * time);
   if(alpha < 0.0) alpha = 0;

   // Set the RGBA color
    color = vec4(in_color[0], in_color[1], in_color[2], alpha);


   // Calculate a new position
   vec2 new_pos = in_pos + (time * in_vel);

   // Set the position. (x, y, z, w)
   gl_Position = vec4(new_pos, 0.0, 1);
}
```

Добавим on_update

```
def on_update(self, dt):
   temp_list = self.burst_list.copy()
   for burst in temp_list:
       if time.time() - burst.start_time > MAX_FADE_TIME:
          self.burst_list.remove(burst)
```

Ну и напоследок добавим гравитацию в наш шейдер

```
#version 330

// Time since burst start
uniform float time;

// (x, y) position passed in
in vec2 in_pos;

// Velocity of particle
in vec2 in_vel;

// Color of particle
in vec3 in_color;
```

```
// Fade rate
in float in_fade_rate;

// Output the color to the fragment shader
out vec4 color;

void main() {

   // Calculate alpha based on time and fade rate
   float alpha = 1.0 - (in_fade_rate * time);
   if(alpha < 0.0) alpha = 0;

   // Set the RGBA color
   color = vec4(in_color[0], in_color[1], in_color[2], alpha);

   // Adjust velocity based on gravity
   vec2 new_vel = in_vel;
   new_vel[1] -= time * 1.1;

   // Calculate a new position
   vec2 new_pos = in_pos + (time * new_vel);

   // Set the position. (x, y, z, w)
   gl_Position = vec4(new_pos, 0.0, 1);
}
```

Давайте воспользуемся еще одним уже готовым шейдером и рассмотрим результат работы

Создадим основной файл

```
import arcade
from arcade.experimental import Shadertoy


class MyGame(arcade.Window):

   def __init__(self):
       super().__init__(width=800, height=600)

       self.time = 0.0

       file_name = "explosion.glsl"
       self.shadertoy = Shadertoy(size=self.get_size(),

main_source=open(file_name).read())

   def on_draw(self):
       self.clear()
       self.shadertoy.program['pos'] = self.mouse["x"],
self.mouse["y"]
```

```
        self.shadertoy.render(time=self.time)

    def on_update(self, delta_time: float):
        self.time += delta_time


if __name__ == "__main__":
    window = MyGame()
    window.center_window()
    arcade.run()
```

Создадим шейдер с именем explosion.glsl

```
#version 330

// Time since burst start
uniform float time;

// (x, y) position passed in
in vec2 in_pos;

// Velocity of particle
in vec2 in_vel;

// Color of particle
in vec3 in_color;

// Fade rate
in float in_fade_rate;

// Output the color to the fragment shader
out vec4 color;

void main() {

    // Calculate alpha based on time and fade rate
    float alpha = 1.0 - (in_fade_rate * time);
    if(alpha < 0.0) alpha = 0;

    // Set the RGBA color
    color = vec4(in_color[0], in_color[1], in_color[2], alpha);

    // Adjust velocity based on gravity
    vec2 new_vel = in_vel;
    new_vel[1] -= time * 1.1;

    // Calculate a new position
    vec2 new_pos = in_pos + (time * new_vel);

    // Set the position. (x, y, z, w)
    gl_Position = vec4(new_pos, 0.0, 1);
```

```
}
```

На текущий момент частицы не двигаются, а просто перемещаются вслед за мышью.

Добавим движение частицам. Для этого модифицируем шейдер

```
// Origin of the particles
uniform vec2 pos;

// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;

// Time for each burst cycle, in seconds.

const float BURST_TIME = 2.0;

const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an
input seed.
// The result is in polar coordinates, to make the points random
in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
 float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
 float distance = fract(sin((t + angle) * 724.3) * 341.2);
 return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
   // Normalized pixel coordinates (from 0 to 1)
   // Origin of the particles
   vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
   // Position of current pixel we are drawing
   vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

   // Re-center based on input coordinates, rather than origin.
   uv -= npos;

   // Default alpha is transparent.
   float alpha = 0.0;
```

```
   // 0.0 - 1.0 normalized fraction representing how far along in
the explosion we are.
   // Auto resets if time goes beyond burst time. This causes the
explosion to cycle.
   float timeFract = fract(iTime * 1 / BURST_TIME);

   // Loop for each particle
   for (float i= 0.; i < PARTICLE_COUNT; i++) {
       // Direction of particle + speed
       float seed = i + 1.0;
       vec2 dir = Hash12_Polar(seed);
       // Get position based on direction, magnitude, and
explosion size

       // Adjust based on time scale. (0.0-1.0)

      vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE *
timeFract;

       // Distance of this pixel from that particle
       float d = length(uv - particlePosition);
       // If we are within the particle size, set alpha to 1.0
       if (d < PARTICLE_SIZE)
           alpha = 1.0;
   }
   // Output to screen
   fragColor = vec4(1.0, 1.0, 1.0, alpha);
}
```

Теперь взрыв также двигается за мышью, а частички летят. Отлично! добавим затухание частиц

```
// Origin of the particles
uniform vec2 pos;

// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
// Time for each burst cycle, in seconds.
const float BURST_TIME = 2.0;
const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an input seed.
// The result is in polar coordinates, to make the points random in a circle
```

```
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
 float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
 float distance = fract(sin((t + angle) * 724.3) * 341.2);
 return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
   // Normalized pixel coordinates (from 0 to 1)
   // Origin of the particles
   vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
   // Position of current pixel we are drawing
   vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

   // Re-center based on input coordinates, rather than origin.
   uv -= npos;

   // Default alpha is transparent.
   float alpha = 0.0;

   // 0.0 - 1.0 normalized fraction representing how far along in the explosion we are.
   // Auto resets if time goes beyond burst time. This causes the explosion to cycle.
   float timeFract = fract(iTime * 1 / BURST_TIME);

   // Loop for each particle
   for (float i= 0.; i < PARTICLE_COUNT; i++) {
      // Direction of particle + speed
      float seed = i + 1.0;
      vec2 dir = Hash12_Polar(seed);
      // Get position based on direction, magnitude, and explosion size
      // Adjust based on time scale. (0.0-1.0)
      vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE * timeFract;
      // Distance of this pixel from that particle
      float d = length(uv - particlePosition);
      // If we are within the particle size, set alpha to 1.0
      if (d < PARTICLE_SIZE)
         alpha = 1.0;
   }
   // Output to screen

   fragColor = vec4(1.0, 1.0, 1.0, alpha * (1.0 - timeFract));

}
```

Добавим эффект свечения для наших частиц

```
// Origin of the particles
uniform vec2 pos;
```

```
// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
// Time for each burst cycle, in seconds.
const float BURST_TIME = 2.0;

// Particle brightness

const float DEFAULT_BRIGHTNESS = 0.0005;


const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an
input seed.
// The result is in polar coordinates, to make the points random
in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
 float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
 float distance = fract(sin((t + angle) * 724.3) * 341.2);
 return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
   // Normalized pixel coordinates (from 0 to 1)
   // Origin of the particles
   vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
   // Position of current pixel we are drawing
   vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

   // Re-center based on input coordinates, rather than origin.
   uv -= npos;

   // Default alpha is transparent.
   float alpha = 0.0;

   // 0.0 - 1.0 normalized fraction representing how far along in
the explosion we are.
   // Auto resets if time goes beyond burst time. This causes the
explosion to cycle.
   float timeFract = fract(iTime * 1 / BURST_TIME);

   // Loop for each particle
   for (float i= 0.; i < PARTICLE_COUNT; i++) {
       // Direction of particle + speed
       float seed = i + 1.0;
       vec2 dir = Hash12_Polar(seed);
```

```
        // Get position based on direction, magnitude, and
explosion size
        // Adjust based on time scale. (0.0-1.0)
        vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE *
timeFract;
        // Distance of this pixel from that particle
        float d = length(uv - particlePosition);

        // Add glow based on distance

        alpha += DEFAULT_BRIGHTNESS / d;

    }
    // Output to screen
    fragColor = vec4(1.0, 1.0, 1.0, alpha * (1.0 - timeFract));
}
```

И напоследок добавим мерцание

```
// Origin of the particles
uniform vec2 pos;

// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
// Time for each burst cycle, in seconds.
const float BURST_TIME = 2.0;
// Particle brightness
const float DEFAULT_BRIGHTNESS = 0.0005;
// How many times to the particles twinkle
const float TWINKLE_SPEED = 10.0;

const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an
input seed.
// The result is in polar coordinates, to make the points random
in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
 float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
 float distance = fract(sin((t + angle) * 724.3) * 341.2);
 return vec2(sin(angle), cos(angle)) * distance;
}
```

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
   // Normalized pixel coordinates (from 0 to 1)
   // Origin of the particles
   vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
   // Position of current pixel we are drawing
   vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

   // Re-center based on input coordinates, rather than origin.
   uv -= npos;

   // Default alpha is transparent.
   float alpha = 0.0;

   // 0.0 - 1.0 normalized fraction representing how far along in
the explosion we are.
   // Auto resets if time goes beyond burst time. This causes the
explosion to cycle.
   float timeFract = fract(iTime * 1 / BURST_TIME);

   // Loop for each particle
   for (float i= 0.; i < PARTICLE_COUNT; i++) {
       // Direction of particle + speed
       float seed = i + 1.0;
       vec2 dir = Hash12_Polar(seed);
       // Get position based on direction, magnitude, and
explosion size
       // Adjust based on time scale. (0.0-1.0)
       vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE *
timeFract;
       // Distance of this pixel from that particle
       float d = length(uv - particlePosition);
       // Add glow based on distance

       float brightness = DEFAULT_BRIGHTNESS * (sin(timeFract *
TWINKLE_SPEED + i) * .5 + .5);

       alpha += brightness / d;
   }
   // Output to screen
   fragColor = vec4(1.0, 1.0, 1.0, alpha * (1.0 - timeFract));
}
```

Ну и рассмотрим еще один вариант

```
import arcade
from arcade.experimental import Shadertoy

class MyGame(arcade.Window):

   def __init__(self):
       super().__init__(width=800, height=600)
```

```
        shader_file_path = "circle_6.glsl"
        window_size = self.get_size()
        self.shadertoy = Shadertoy.create_from_file(window_size,
shader_file_path)

    def on_draw(self):
        self.shadertoy.program['pos'] = self.mouse["x"],
self.mouse["y"]
        self.shadertoy.program['color'] =
arcade.get_three_float_color(arcade.color.LIGHT_BLUE)
        self.shadertoy.render()

if __name__ == "__main__":
    MyGame()
    arcade.run()
```

circle_6.glsl

```
uniform vec2 pos;

uniform vec3 color;


void mainImage(out vec4 fragColor, in vec2 fragCoord) {

    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    vec2 npos = pos/iResolution.xy;


    // Position of fragment relative to specified position

    vec2 rpos = npos - uv;

    // Adjust y by aspect ratio
    rpos.y /= iResolution.x/iResolution.y;

    // How far is the current pixel from the origin (0, 0)
    float distance = length(rpos);
    // Use an inverse 1/distance to set the fade
    float scale = 0.02;
    float fade = 1.1;
    float strength = pow(1.0 / distance * scale, fade);

    // Fade our orange color

    vec3 color = strength * color;


    // Tone mapping
    color = 1.0 - exp( -color );
```

```
    // Output to the screen
    fragColor = vec4(color, 1.0);
}
```