Компьютерная графика
Современные технологии компьютерной графики и рендеринга

Системы частиц в WebGL

Лекция 6

02.04.02 ФИИТ

Разработка мобильных приложений и компьютерных игр

2025-2026

Что такое система частиц?

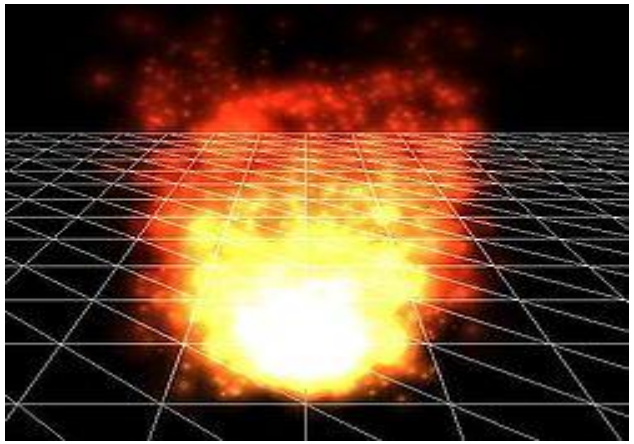
Определение:
Система частиц — это метод компьютерной графики для моделирования объектов,
не имеющих чёткой геометрической формы.

Ключевая идея:
Вместо моделирования сложного объекта целиком, мы создаём множество простых элементов (частиц), поведение которых в совокупности формирует нужный визуальный эффект.

Системы частиц могут быть реализованы как в двумерной, так и в трёхмерной графике

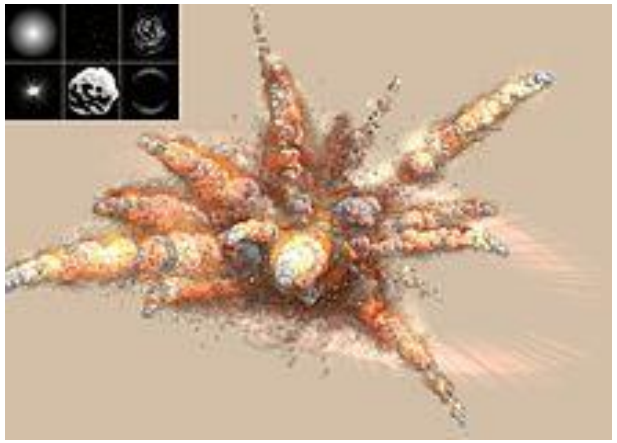
Области применения

- Природные явления: огонь, дым, снег, дождь, туман, различные облака
- Спецэффекты: взрывы, искры, фейерверки, шлейфы, струи пара, **бенгальский огонь (рассмотрим)**
- Научная визуализация: потоки жидкости, газов, поля частиц
- Атмосферные эффекты в играх и виртуальных средах
- Простые системы частиц применяются практически во всех современных компьютерных играх и пакетах 3D моделирования

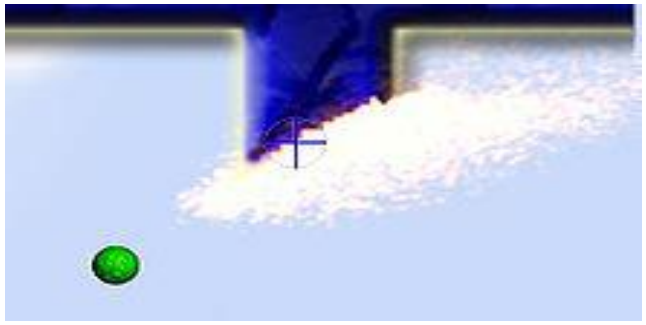


Система частиц, симулирующая огонь, создана в 3dengfx

Система частиц, симулирующая галактику, создана в 3dengfx

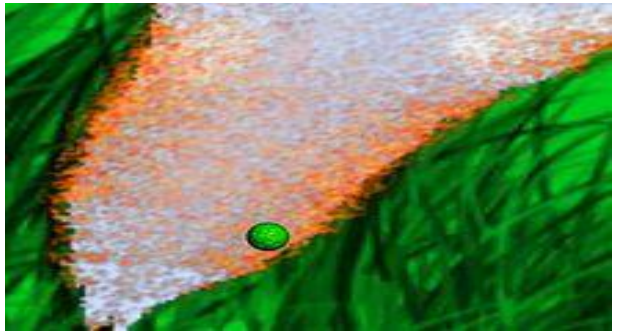


Система частиц, симулирующая взрыв бомбы, создана в particleIllusion



Пример системы частиц в двумерной графике

Пример системы частиц в двумерной графике



Пример системы частиц в двумерной графике

Принципы построения системы частиц

Система частиц состоит из определённого (фиксированного или произвольного) количества частиц

Математически каждая **частица** представляется как **материальная точка с** дополнительными **атрибутами**, такими как внешний вид, скорость, ориентация в пространстве, угловая скорость, и т. п.

В ходе работы программы каждая **частица изменяет своё состояние по** определённому, **общему** для всех частиц системы, **закону**.
Например, частица может подвергаться воздействию гравитации, менять размер, цвет, скорость и т.д.

После проведения всех расчётов, **частица визуализируется**. Частица может быть визуализирована точкой, треугольником, спрайтом, или даже полноценной трёхмерной моделью

При моделировании физики жидкостей часто используются метасферы, которые «сливаются» между собой

Математическая модель частицы

Абстракция частицы
Каждая частица — это материальная точка с набором атрибутов

Динамика
Состояние частицы обновляется каждый кадр по законам физики или процедурным правилам.

Атрибуты частицы

Атрибут	Тип	Назначение
Позиция	vec3	Местоположение в пространстве
Скорость	vec3	Вектор движения (м/с)
Ускорение	vec3	Влияние сил (гравитация, ветер)
Время жизни	float	Оставшееся время существования
Цвет	vec3 / vec4	Текущий цвет и прозрачность
Размер	float	Визуальный масштаб (в пикселях или юнитах)
Угол поворота	float	Ориентация спрайта (если не billboarding)
Угловая скорость	float	Скорость вращения

Жизнь частицы

На протяжении жизни частица редко остаётся в покое

Частицы могут:
- двигаться
- вращаться
- менять свой цвет и/или прозрачность
- сталкиваться с трёхмерными объектами

Часто у частиц задана максимальная продолжительность жизни, по истечении которого частица исчезает

Жизненный цикл частицы

Этапы жизни частицы:

1. Рождение (Spawn)
   - Создаётся эмиттером в заданной области (точка, линия, объём, поверхность меша)
   - Получает начальные атрибуты (часто со случайными вариациями)
2. Жизнь (Update)
   - Позиция изменяется по формуле: pos += velocity * dt
   - Скорость меняется под действием сил: vel += acceleration * dt
   - Время жизни уменьшается: life -= dt
   - Цвет, размер, прозрачность интерполируются по кривой от времени жизни
3. Смерть (Die)
   - Удаляется из активного списка
   - Может породить дочерние частицы (Sub-Emitters), например, искры от фейерверка

Важно:
Для стабильной анимации все обновления должны учитывать разницу во времени между кадрами (delta time), чтобы эффект работал одинаково на разной частоте кадров.

Эмиттер

В большинстве реализаций, новые частицы испускаются так называемым «эмиттером»

Эмиттером может быть точка, тогда новые частицы будут возникать в одном месте.
Так можно смоделировать, например, взрыв: эмиттером будет его центр

Эмиттером может быть отрезок прямой или плоскость: например частицы дождя или снега должны возникать на высоко расположенной горизонтальной плоскости

Эмиттером может быть и произвольный геометрический объект: в этом случае новые частицы будут возникать на всей его поверхности

Типы эмиттеров

Тип	Описание	Пример применения
Точечный	Все частицы рождаются из одной точки	Взрыв, искра, источник огня
Линейный	Частицы рождаются на отрезке	Огненный меч, линия дождя
Плоскостной	Частицы рождаются на прямоугольной плоскости	Снегопад, травяной покров
Объёмный	Частицы рождаются внутри объёма (сфера, куб, цилиндр)	Облако, рой мух
Mesh-эмиттер	Частицы рождаются на поверхности или внутри 3D-модели	Искры от движущегося автомобиля, кровь из раны

Параметры эмиттера

- Скорость эмиссии (частиц в секунду)
- Максимальное количество частиц
- Форма области рождения
- Начальные вариации скорости, размера, цвета

Расширение функциональности

К частицам могут быть применены **пространственные деформации** — силовые поля, которые могут изменять:

- вектор движения
- скорости
- и другие параметры частиц

Примеры таких деформаций:

- ветер
- гравитация
- ударная волна

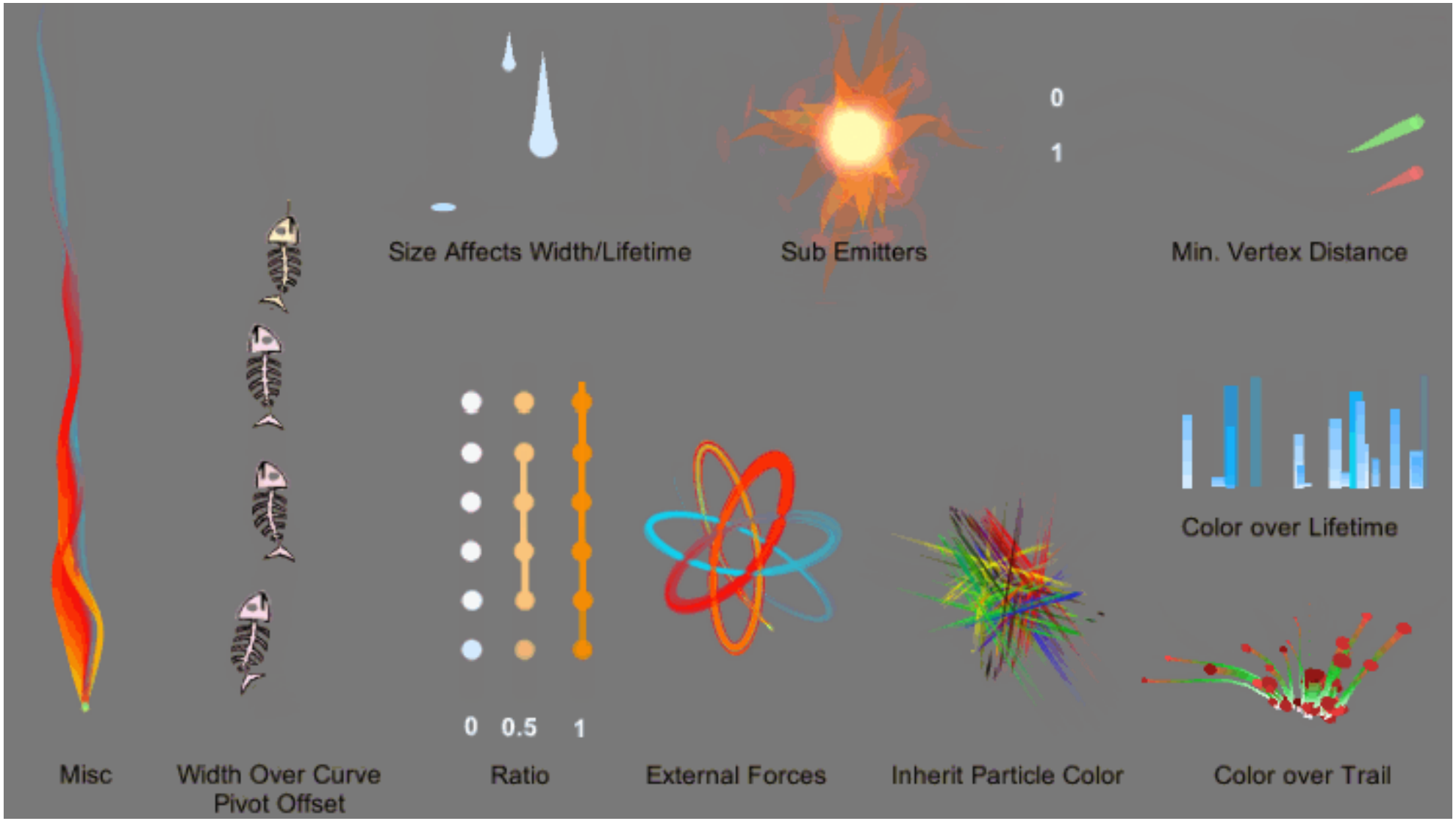
Пространственные деформации имеют визуальное представление только в программе для их редактирования, но они изменяют параметры управляемых ими частиц

Силы и пространственные деформации

- Гравитация: постоянное ускорение вниз (или к центру объекта)
- Ветер: постоянная сила в заданном направлении с возможной турбулентностью
- Ударная волна: радиальная сила от точки взрыва
- Затухание (Damping): постепенное замедление частиц (сопротивление среды)
- Вихри (Vortices): вращающие поля

Реализация в коде (GLSL)

```
// Применение гравитации
velocity.y -= gravity * deltaTime;

// Применение ветра
velocity += windDirection * windStrength * deltaTime;

// Затухание
velocity *= (1.0 - damping * deltaTime);
```


Size Affects Width/Lifetime
Sub Emitters
0
1
Min. Vertex Distance
Color over Lifetime
0
0.5
1
Misc
Width Over Curve
Pivot Offset
Ratio
External Forces
Inherit Particle Color
Color over Trail

Рендеринг частиц. Способы визуализации в WebGL

1. Точки (GL_POINTS)
    - Самая простая форма: каждый частица — одна вершина
    - Размер задаётся через gl_PointSize в вершинном шейдере
    - Всегда повёрнуты к камере (billboarding)
    - Идеально для искр, звёзд, снега
2. Спрайты (Quad с текстурой)
    - Два треугольника, образующих прямоугольник
    - Может быть повёрнут вручную (например, для листьев или лепестков)
    - Требует больше вершин, но даёт контроль над формой
3. 3D-модели
    - Каждая частица — полноценный меш (например, камешек или пуля)
    - Используется редко из-за высокой производительности
4. Метасферы (Metaballs)
    - Частицы «сливаются» при сближении, создавая эффект жидкости
    - Требует сложного рендеринга через marching cubes

Рендеринг частиц. Сравнение производительности

Способ	Вершин на 1000 частиц	Сложность
Точки	1000	Низкая
Спрайты	6000 (2 треуг.)	Средняя
3D-модели	100 000+	Высокая

Смешивание цветов и прозрачность

Проблема: Частицы часто имеют полупрозрачные текстуры (искры, дым, огонь).
WebGL должен правильно смешивать их с фоном.

Решение — включение смешивания:
javascript: gl.enable(gl.BLEND);

Два основных **режима смешивания**:

Режим	Код	Эффект
Аддитивное смешение	gl.blendFunc(gl.SRC_ALPHA, gl.ONE)	Яркие эффекты, искры, взрывы — частицы добавляют свет
Альфа-смешение	gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA)	Прозрачные объекты, дым, стекло

Важно:
При альфа-смешении частицы должны рендериться от дальних к ближним, иначе прозрачность будет выглядеть некорректно. Для аддитивного смешения порядок не важен.

Упрощения в real-time графике

Упрощение	Почему это важно
Частицы не отбрасывают тени	Тени требуют дополнительного рендеринга в карту теней
Частицы не получают теней от геометрии	Упрощает освещение, частицы всегда полностью освещены
Частицы не отражаются в зеркалах	Экономия на reflection-проходах
Частицы не влияют на освещение сцены	Нет необходимости в сложном освещении
Частицы не сортируются по глубине	Экономия CPU, но может давать артефакты прозрачности

Без этих упрощений обсчёт системы частиц будет требовать больше ресурсов:

- в случае с поглощением света потребуется сортировать частицы по удалённости от камеры
- в случае с тенями каждую частицу придётся рисовать несколько раз

Когда упрощения снимаются

- В кинематографичных сценах (pre-rendered)
- В высокобюджетных играх с мощными GPU
- При использовании compute shaders для сортировки на GPU

Нет единого решения для реализации

В настоящее время (2024-2025 г.г.) не существует общепринятой реализации систем частиц

В разных играх и программах 3D моделирования свойства, поведение и внешний вид частиц могут принципиально отличаться

Оптимизация — инстансинг (Instancing)

Проблема:
Рендеринг 10 000 частиц как отдельных draw calls уничтожит производительность.

Решение — инстансинг:
Отрисовка множества одинаковых объектов (частиц) одним вызовом.

Схема работы:
1. Создаём один буфер с геометрией (квадрат для спрайта)
2. Создаём второй буфер с per-instance данными (позиция, цвет, размер)
3. Используем gl.drawArraysInstanced или gl.drawElementsInstanced

Пример кода WebGL

javascript

```
// Настройка атрибутов для инстансинга
gl.vertexAttribDivisor(locationPosition, 0); // общая геометрия
gl.vertexAttribDivisor(locationInstancePos, 1); // меняется на частицу
gl.vertexAttribDivisor(locationInstanceColor, 1);

gl.drawArraysInstanced(gl.TRIANGLES, 0, 6, particleCount);
```

Выигрыш:
10 000 частиц → 1 draw call вместо 10 000. Скорость возрастает в десятки раз.

gl.vertexAttribDivisor(locationPosition, 0);

gl.vertexAttribDivisor(locationPosition, 0); // общая геометрия

Что делает:

- vertexAttribDivisor устанавливает «делитель» для атрибута вершин
- Когда делитель = 0, атрибут обновляется каждую вершину, а не каждый экземпляр
- Это означает, что данные из этого атрибута будут одинаковыми для всех экземпляров

Для locationPosition:

- Здесь хранятся координаты вершин одного квадрата (спрайта)
- Делитель = 0 означает, что WebGL будет использовать одни и те же 6 вершин для каждой частицы
- Все частицы будут иметь одинаковую форму, но разное положение

gl.vertexAttribDivisor(locationInstancePos, 1);

javascript
gl.vertexAttribDivisor(locationInstancePos, 1); // меняется на частицу

Что делает:

- Делитель = 1 означает, что атрибут обновляется каждый экземпляр
- Каждая новая частица будет использовать следующее значение из буфера

Для locationInstancePos:

- Здесь хранятся позиции всех частиц в виде массива
- Первая частица получает первую позицию, вторая — вторую и т.д.

Как это работает внутри

1. WebGL берет геометрию (6 вершин)
2. Для экземпляра 0: подставляет instancePos[0] и instanceColor[0]
3. Отрисовывает 6 вершин
4. Для экземпляра 1: подставляет instancePos[1] и instanceColor[1]
5. Отрисовывает следующие 6 вершин
6. И так далее для всех particleCount экземпляров

Сравнение производительности

Количество частиц	Без инстансинга	С инстансингом
100	100 draw calls	1 draw call
1 000	1 000 draw calls	1 draw call
10 000	10 000 draw calls	1 draw call
100 000	Невозможно (FPS ~1)	1 draw call (FPS ~60)

Почему инстансинг быстрее:

- Каждый draw call имеет накладные расходы (смена состояния GPU, проверки)
- При 10 000 частиц без инстансинга CPU занят подготовкой 10 000 вызовов
- Инстансинг передает все данные за один раз, GPU обрабатывает их параллельно

GPU-частицы (Transform Feedback / Compute Shaders)

Идея:
Обновление позиций частиц происходит не на CPU, а на GPU, чтобы избежать передачи данных между памятью.

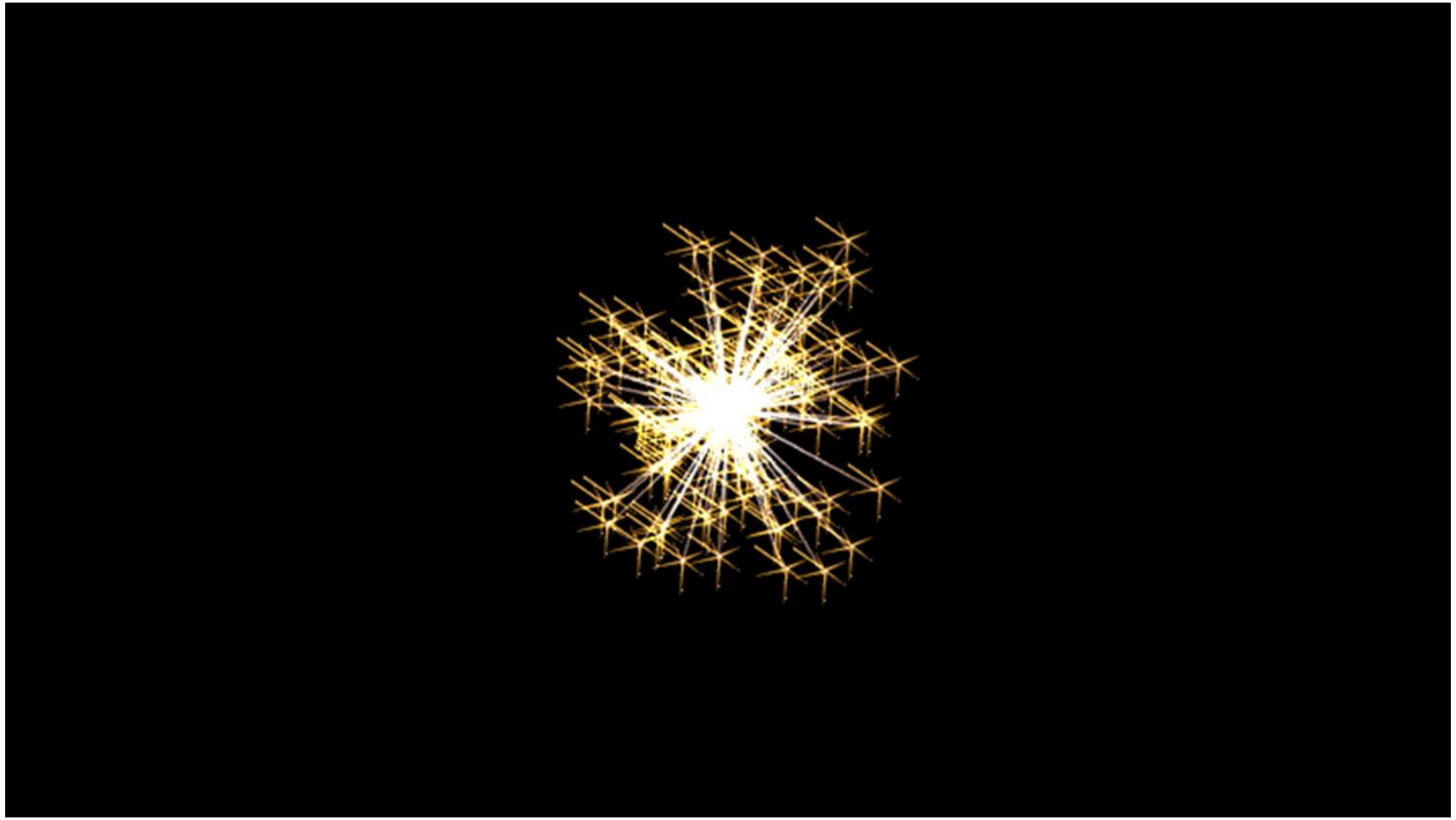
Вариант 1: **Transform Feedback**

- Вершинный шейдер обновляет позиции и записывает их обратно в буфер
- Следующий кадр читает обновлённый буфер как входные данные

Вариант 2: **Compute Shaders** (WebGL 2.0 / WebGPU)

- Пишем compute-шейдер, который параллельно обновляет тысячи частиц
- Данные хранятся в текстурах или SSBO

Бенгальский огонь



Характеристики системы частиц «бенгальский огонь»

- источник появления частиц (эмиттер)
- каждая частица будет:
  - иметь свою скорость
  - иметь своё направление
  - иметь своё время жизни
  - будет оставлять за собой след
- но в то же время все частицы подчиняются определённому закону, общему для системы

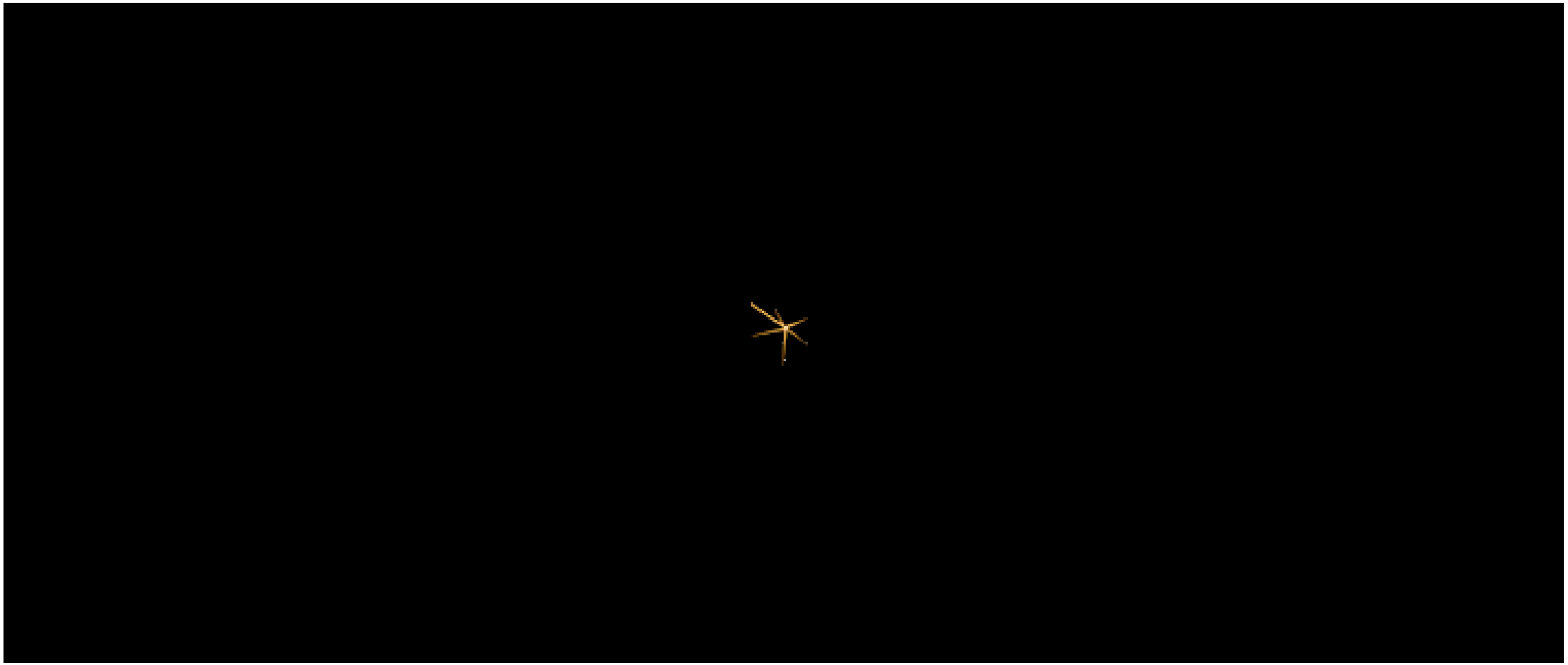
Объекты, используемые для частиц

- полноценный 3D-объект
- 2D-объект с заливкой или текстурой
- точка (спрайт)

Для искр подходит спрайт, поскольку

- для каждой частицы достаточно всего одной вершины
- спрайт по определению всё время повёрнут в сторону наблюдателя, независимо от поворота сцены, и искра всегда остаётся искрой
- размер задаётся в пикселях, не зависит от расстояния

Одна искра



Одна искра

текстура для точки – изображение с прозрачным фоном 

зададим размер точки с помощью gl_PointSize в вершинном шейдере,
тогда точка превращается в квадрат указанного размера с центром в этой точке

Различия:

- квадрат из двух треугольников будет подвержен трансформациям матриц
- заданный одной точкой квадрат будет всегда направлен к наблюдателю и будет иметь размер gl_PointSize

Желательно, чтобы размер точки совпадал с размером изображения искры

Вершинный шейдер

```
attribute vec3 a_position;

uniform mat4 u_mvMatrix;
uniform mat4 u_pMatrix;

void main() {
    gl_Position = u_pMatrix * u_mvMatrix * vec4(a_position, 1.0);

    // размер искры
    gl_PointSize = 32.0;
}
```

Фрагментный шейдер

```
precision mediump float;

   uniform sampler2D u_texture;

   void main() {
      gl_FragColor = texture2D(u_texture, gl_PointCoord);
   }
```

Загрузка изображения и инициализация текстуры

```
var texture = gl.createTexture();
var image = new Image();
image.src = "spark.png";

image.addEventListener('load', function() {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.bindTexture(gl.TEXTURE_2D, null);

    // отрисовку сцены начинаем только после загрузки изображения
    requestAnimationFrame(drawScene);
});
```

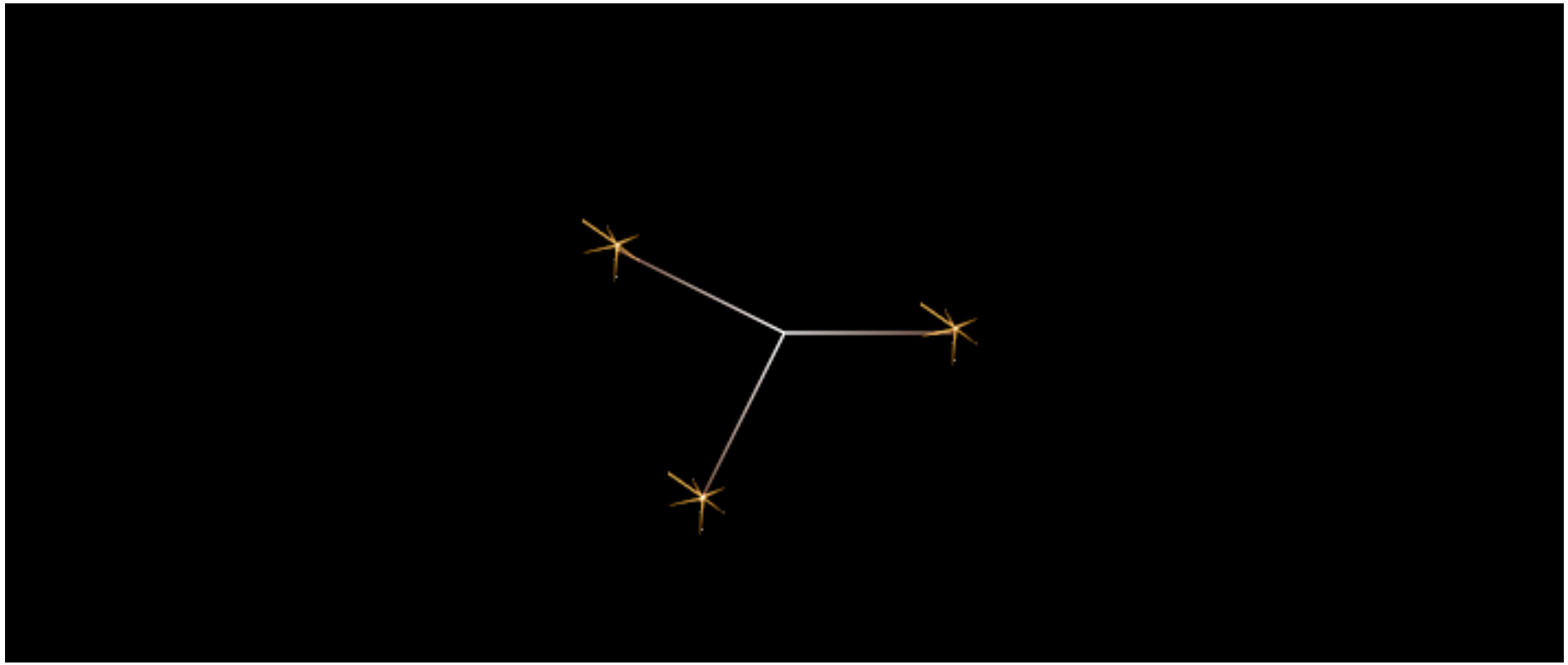
NEAREST LINEAR

Смешивание цветов

Чтобы WebGL понимал прозрачность и избавлялся от фона искры, нам необходимо включить смешивание цветов

```
//включает смешивание
gl.enable(gl.BLEND);

//определяет взаимодействие рисуемых и уже отрисованных пикселей
gl.blendFunc(gl.SRC_ALPHA, gl.ONE);
```

Добавляем следы искр



Ещё один вершинный шейдер

```
attribute vec3 a_position;
attribute vec3 a_color;

out vec3 v_color;

uniform mat4 u_mvMatrix;
uniform mat4 u_pMatrix;

void main() {
    v_color = a_color;
    gl_Position = u_pMatrix * u_mvMatrix * vec4(a_position, 1.0);
}
```

Ещё один фрагментный шейдер

```
precision mediump float;

in vec3 v_color;

void main() {
    gl_FragColor = vec4(v_color, 1.0);
}
```

Инициализация искр

```
//инициализация программы искр

var programSpark = webglUtils.createProgramFromScripts(gl, ["vertex-shader-spark", "fragment-shader-spark"]);

var positionAttributeLocationSpark = gl.getAttribLocation(programSpark, "a_position");
var textureLocationSpark = gl.getUniformLocation(programSpark, "u_texture");
var pMatrixUniformLocationSpark = gl.getUniformLocation(programSpark, "u_pMatrix");
var mvMatrixUniformLocationSpark = gl.getUniformLocation(programSpark, "u_mvMatrix");
```

Инициализация следов искр

```
// инициализация программы следов искр

var programTrack = webglUtils.createProgramFromScripts(gl, ["vertex-shader-track", "fragment-shader-track"]);

 var positionAttributeLocationTrack = gl.getAttribLocation(programTrack, "a_position");
 var colorAttributeLocationTrack = gl.getAttribLocation(programTrack, "a_color");
 var pMatrixUniformLocationTrack = gl.getUniformLocation(programTrack, "u_pMatrix");
 var mvMatrixUniformLocationTrack = gl.getUniformLocation(programTrack, "u_mvMatrix");
```

Отрисовка следов и самих искр

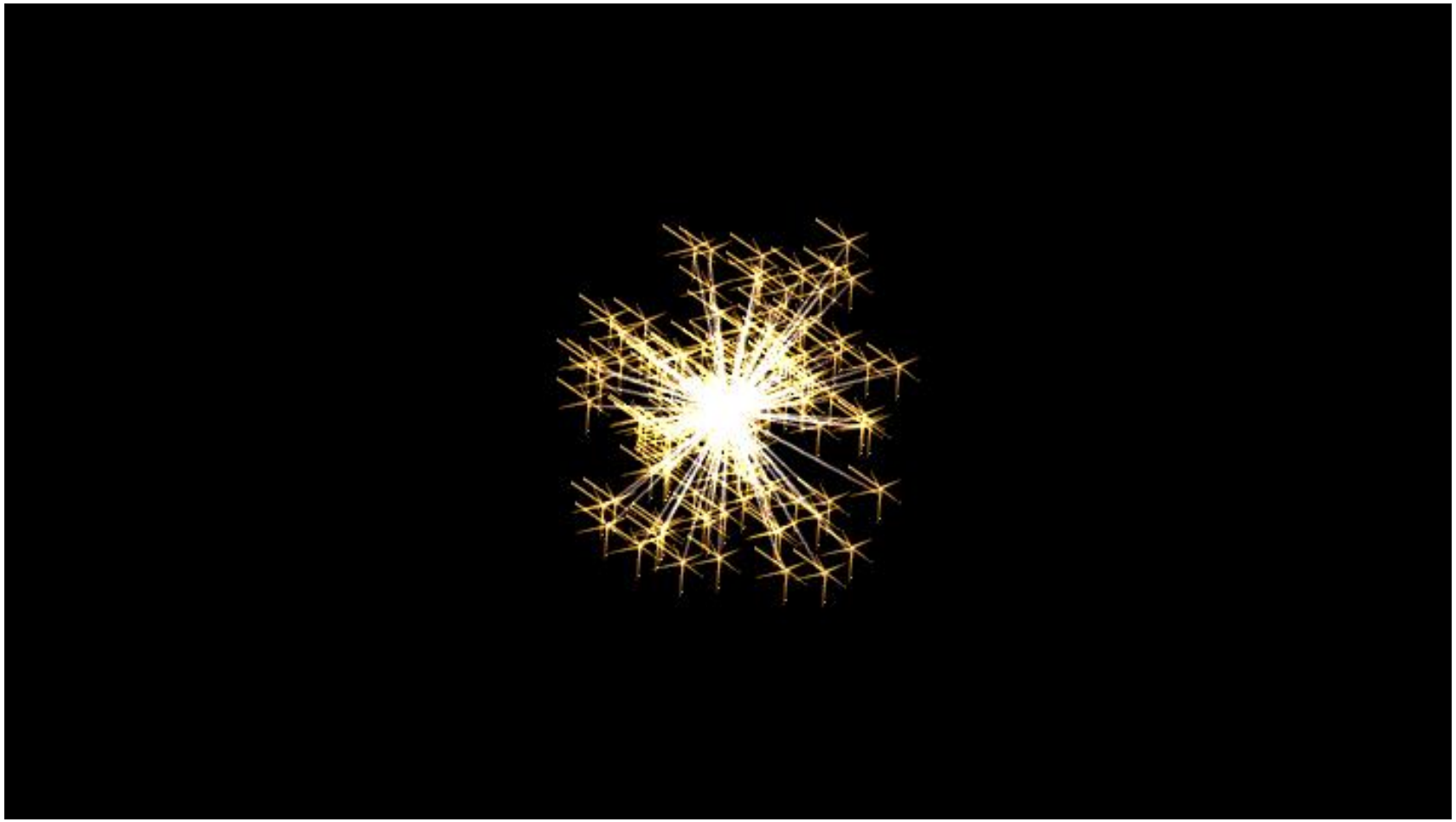
В каждую функцию будут передаваться координаты всех существующих на данный момент искр — сейчас три

```
var positions = [
       1, 0,   0,
      -1, 0.5, 0,
      -0.5, -1, 0
 ];

// В начале каждой функции отрисовки drawTracks и drawSparks
// будет вызываться gl.useProgram для установки текущей программы

 drawTracks(positions);  // будет активироваться программа programTrack
 drawSparks(positions); // будет активироваться программа programSpark
```

Отрисовка следов искр

```
var colors = [];
var positionsFromCenter = [];
for (var i = 0; i < positions.length; i += 3) {
   // для каждой координаты добавляем точку начала координат, чтобы получить след искры
   positionsFromCenter.push(0, 0, 0);
   positionsFromCenter.push(positions[i], positions[i + 1], positions[i + 2]);

   // цвет в начале координат будет белый (горячий), а дальше будет приближаться к оранжевому
   colors.push(1, 1, 1, 0.47, 0.31, 0.24);
 }
```

Полноценная система частиц



Алгоритм работы бенгальского огня

- создаём искру с произвольными
  - направлением
  - скоростью
  - длиной пути

- при каждой отрисовке изменяем положение искры

- когда искра пройдёт весь отрезок пути, запускаем её заново из начала координат

Класс для искры

```
function Spark() {
    this.init();
};

// количество искр
Spark.sparksCount = 200;
```

С двумя функциями:

- функция init вызывается для создания и инициализации искры
- функция move вызывается при каждом цикле отрисовки для приращения координат искры

```
Spark.prototype.init = function() {
    // время создания искры
    this.timeFromCreation = performance.now();

    // задаём направление полёта искры в градусах, от 0 до 360
    var angle = Math.random() * 360;
    // радиус - это расстояние, которое пролетит искра
    var radius = Math.random();
    // отмеряем точки на окружности - максимальные координаты искры
    this.xMax = Math.cos(angle) * radius;
    this.yMax = Math.sin(angle) * radius;

    // dx и dy - приращение искры за вызов отрисовки, то есть её скорость,
    // у каждой искры своя скорость. multiplier подбирается эмпирически
    var multiplier = 125 + Math.random() * 125;
    this.dx = this.xMax / multiplier;
    this.dy = this.yMax / multiplier;

    // Для того, чтобы не все искры начинали движение из начала координат,
    // делаем каждой искре свой отступ, но не более максимальных значений.
    this.x = (this.dx * 1000) % this.xMax;
    this.y = (this.dy * 1000) % this.yMax;
};
```

```
Spark.prototype.move = function(time) {
    // находим разницу между вызовами отрисовки, чтобы анимация работала
    // одинаково на компьютерах разной мощности
    var timeShift = time - this.timeFromCreation;
    this.timeFromCreation = time;

    // приращение зависит от времени между отрисовками
    var speed = timeShift;
    this.x += this.dx * speed;
    this.y += this.dy * speed;

    // если искра достигла конечной точки, запускаем её заново из начала координат
    if (Math.abs(this.x) > Math.abs(this.xMax) || Math.abs(this.y) > Math.abs(this.yMax)) {
        this.init();
        return;
    }
};
```

Создаем необходимое количество искр при инициализации программы

```
var sparks = [];
  for (var i = 0; i < Spark.sparksCount; i++) {
     sparks.push(new Spark());
  }
```

```
//Вызываем смещение искр при каждой отрисовке
for (var i = 0; i < sparks.length; i++) {
        sparks[i].move(now);
    }

//получаем координаты искр для передачи в функции
var positions = [];
 sparks.forEach(function(item, i, arr) {
        positions.push(item.x);
        positions.push(item.y);
        // искры двигаются только в одной плоскости xy
        positions.push(0);
 });

 drawTracks(positions);
 drawSparks(positions);
```

Оптимизация для большого количества частиц

Метод	Когда использовать
Typed Arrays + буферы	Всегда, чтобы минимизировать накладные расходы
Web Workers	Обновление частиц в отдельном потоке, не блокирует UI
GPU-частицы (Transform Feedback)	10 000+ частиц, критично к производительности
Инстансинг	Отрисовка тысяч спрайтов одним вызовом

Расширение — Sub-Emitters и каскады эффектов

Идея:
Одна система частиц может порождать другие при определённых событиях (смерть частицы, столкновение).

Пример: фейерверк

1. Основной снаряд поднимается вверх
2. При достижении вершины — смерть основной частицы
3. Sub-emitter порождает 100 искр, разлетающихся во все стороны
4. Каждая искра через 0.5 секунды порождает микро-вспышку (ещё один Sub-emitter)

Дополнительные эффекты — шум и турбулентность

Для реалистичности движения частиц (дым, облака, листва) можно добавлять процедурный шум

Шум Перлина в движении:

```
// В шейдере обновления позиций
float noise = texture2D(u_noiseTexture, position.xz * 0.1).r;
velocity.x += sin(time + noise * 10.0) * 0.5;
velocity.z += cos(time + noise * 8.0) * 0.5;
```

Эффект:

Частицы движутся не по прямой, а плавно извиваются
Создаётся ощущение турбулентности воздуха

Что мы узнали

- Математическую модель частицы и её жизненный цикл
- Типы эмиттеров и силовых полей
- Способы рендеринга: точки, спрайты, инстансинг
- Оптимизации: от CPU до GPU-симуляций
- Реализацию конкретного эффекта — бенгальский огонь

Базовый уровень

1. Что такое система частиц и для каких визуальных эффектов она применяется? Назовите не менее 5 примеров из лекции.
2. Объясните, как в WebGL реализуется отрисовка искры с помощью GL_POINTS. Что делает gl_PointSize и gl_PointCoord?
3. Зачем в системах частиц используется смешивание цветов (blending)? Какой режим применяется для бенгальского огня и почему?

Средний уровень

1. Опишите жизненный цикл частицы. Какие изменения могут происходить с частицей на протяжении её существования?
2. Что такое эмиттер? Назовите три типа эмиттеров и приведите примеры, где каждый из них применяется.
3. В чём заключается алгоритм работы бенгальского огня? Как задаётся направление, скорость и дальность полёта каждой искры?

Продвинутый уровень

1. Какие упрощения используются в real-time графике при работе с системами частиц? Почему без этих упрощений обсчёт системы требует больше ресурсов?
2. Сравните три способа визуализации частиц в WebGL: точка (GL_POINTS), спрайт из двух треугольников и полноценная 3D-модель. Каковы преимущества и недостатки каждого?
3. В чём заключается принцип работы инстансинга (instancing) и compute-шейдеров для систем частиц? Какие задачи решает каждый из этих подходов и при каком количестве частиц их применение становится необходимым?