МГТУ им. Н.Э. Баумана

Факультет информатики и систем управления

Кафедра теоретической информатики и компьютерных технологий

Курсовая работа по дисциплине «Компьютерная графика»

«Реализация системы частиц на примерах визуализации огня и дыма»

Выполнил:

студент группы ИУ9-51

Выборнов А.И.

Руководитель:

Вишняков И.Э.

Москва 2012 г.

Оглавление

Введение.

Компьютерная графика — область деятельности, в которой компьютеры выступают в качестве инструмента как для создания изображений, так и для обработки визуальной информации, полученной из реального мира.

Разработки в области компьютерной графики сначала двигались лишь академическим интересом и шли в научных учреждениях. Постепенно компьютерная графика прочно вошла в повседневную жизнь, появилась возможность вести коммерчески успешные проекты в этой области. В настоящее время компьютерная графика широко применяется во многих областях. К основным сферам применения технологий компьютерной графики относятся:

- Графический интерфейс пользователя.
- Спецэффекты, визуальные эффекты, цифровая кинематография.
- Цифровое телевидение, Интернет, видеоконференции.
- Цифровая фотография и существенно возросшие возможности по обработке фотографий.
- Цифровая живопись.
- Визуализация научных и деловых данных.
- Компьютерные игры, системы виртуальной реальности.
- Системы автоматизированного проектирования.
- Компьютерная томография.
- Компьютерная графика для кино и телевидения.
- Лазерная графика.

Компьютерная графика является также одной из областей научной деятельности. В области компьютерной графики защищаются диссертации, а также проводятся различные конференции. Современная компьютерная графика это тщательно разработанная дисциплина. В ней исследованы основные элементы геометрических преобразований и описания кривых и поверхностей. Также изучены, но всё ещё продолжают развиваться методы растрового сканирования, отсечения, удаления невидимых линий и поверхностей, закраски, текстурирования, модулирования эффектов прозрачности и преломления etc.

Данная работа поднимает задачу, которая требуется практически во всех областях связанных с визуализацией явлений естественного характера(дым, снег, дождь, огонь etc) и некоторых других(к примеру, визуализация данных), а именно, разработка системы частиц.

1. Формализация задачи.

1.1. Базовые понятия.

Система частиц — используемый в компьютерной графике способ представления объектов, не имеющих чётких геометрических границ (различные облака, туманности, взрывы, струи пара, шлейфы от ракет, дым, снег, дождь etc). Системы частиц могут быть реализованы как в двумерной, так и в трёхмерной графике.

Система частиц состоит из определенного (фиксированного или произвольного) количества частиц. Математически каждая частица представляется как материальная точка с дополнительными атрибутами, такими как скорость, цвет, ориентация в пространстве, угловая скорость, и т. п. В ходе работы программы, моделирующей частицы, каждая частица изменяет своё состояние по определенному, общему для всех частиц системы, закону. Например, частица может подвергаться воздействию гравитации, менять размер, цвет, скорость и прочие характеристики, а после проведения всех расчётов частица визуализируется. Частица может быть визуализирована точкой, треугольником, спрайтом, билбордом, или даже полноценной трехмерной моделью. Не существует общепринятой реализации систем частиц. В разных играх и программах 3D моделирования свойства, поведение и внешний вид частиц могут принципиально отличаться[1].

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. На протяжении жизни частица редко остаётся в покое. Частицы могут двигаться, вращаться, менять свой цвет, прозрачность, и сталкиваться с трёхмерными объектами. Часто у частиц задана максимальная продолжительность жизни, по истечении которого частица исчезает.

В трёхмерных приложениях реального времени (например в компьютерных играх) обычно считается, что частицы не отбрасывают тени друг на друга, а также на окружающую геометрию, и что они не поглощают, а излучают свет. Без этих упрощений обсчёт системы частиц будет

требовать больше ресурсов: в случае с поглощением света потребуется сортировать частицы по удалённости от камеры, а в случае с тенями каждую частицу придётся рисовать несколько раз.

1.2. Требования к реалистичности.

В компьютерной графике часто возникает проблема визуализации объектов с неточным границами. Обычно такие объекты описываются сложными физическими или математическими законами. Ввиду того, что системы частиц обычно пренебрегают сложными математическими вычислениями физических законов, необходимо учитывать тот факт, что надо создать высоко-реалистичное изображение.

1.3. Требования к производительности.

Для того, чтобы получить реалистичные эффекты, система частиц должна содержать и уметь обрабатывать на лету многие тысячи частиц. Что требует большого количества вычислений. Несмотря на то, что с совершенствованием техники эта проблема становится всё менее существенной, трудности всё равно возникают. Они возникают ввиду сферы применения систем частиц. А это обычно движки рендеринга, в которых не только обрабатывается множество сложных для вычисления методов, но и одновременно задействуется множество различных систем частиц.

Из этого возникает такое требование к любой системе частиц, как быстродействие.

1.4. Требования к расширяемости.

Написать систему частиц одновременно реализующую все возможные эффекты достаточно объёмная и бессмысленная задача, ввиду того, что постоянно появляются новые сферы применения со своей спецификой. Поэтому обычно пишется прототип системы частиц — некоторая структура, позволяющая на своей основе создавать системы частиц под каждый конкретный случай. Расширив и адаптировав который, можно получить требуемую систему частиц.

1.5. Итог.

Необходимо разработать и реализовать быстро работающую легко-расширяемую real-time

систему частиц. И на её основе создать как можно более фото реалистичные системы частиц для огня и дыма.

2. Выбор подхода к решению.

Выбор подхода к решению задачи, включает в себя множество факторов, которые повлияют на эффективность решения данной задачи в пределах сформулированных условий. Вот основные: архитектура проекта, язык программирования, способ отрисовки. Рассмотрим различные подходы.

2.1. Язык программирования.

В данное время существует не одна сотня языков программирования. Выбрать язык, подходящий для реализации задачи, не представляет трудностей. Необходимость быстрого конструирования новых систем частиц из уже существующего ядра может быть легко решена с использованием ООП парадигмы. А всем требованиям по реализации высокопроизводительной системы полностью удовлетворяет C++, который эту парадигму поддерживает.

2.2. Архитектура.

Основное различие среди возможных реализаций архитектуры, заключается в способе получения системы частиц для каждого конкретного случая из написанного прототипа. Существует три основных подхода: реализация с использованием наследования, реализация с использованием шаблона проектирования «стратегия» и реализация новой системы частиц в каждом отдельном случае.

2.2.1. Наследование.

При подходе с использованием наследования, пишется базовый абстрактный класс. А при решении конкретной задачи, мы создаём систему частиц наследуясь от базового класса переопределяя необходимые и добавляя новые методы.

Плюсами этого подхода является то, что полученная система частиц получается удобочитаемой и предоставляет практически неограниченный доступ к внутреннему состояния методов, реализующих поведение(к примеру при таком подходе легко реализуется ветер).

Но есть и существенный минус. При наследовании переопределяются виртуальные методы, а использование таблицы виртуальных вызовов, при тысячах обрабатываемых частицах, может

серьезно повлиять на общую производительность.

2.2.2. Реализация новой системы частиц в каждом конкретном случае.

Возможно создание отдельных классов систем частиц. По производительности этот метод самый оптимальной, но скорость и удобство разработки страдает, ведь для каждой новой системы надо создавать все механизмы работы с частицами с нуля.

2.2.3. Шаблон проектирования «стратегия».

При подходе с использованием этого шаблона проектирования, пишется базовый класс, в котором реализованы все основные методы, предоставляющие интерфейс для работы с системой частиц, а с помощью шаблонов (template) передаются методы, отвечающие за поведение. При решении конкретной задачи, мы пишем необходимые поведения и определяем новый класс на основе старого с подстановкой в шаблон конкретных методов, отвечающих за поведения[2].

При данном подходе мы получаем, как простоту разработки и написания новых систем частиц, так и повышенную быстродействительность, ввиду всего этого для разработки выбирается именно он.

2.3. Способ отрисовки.

Система частиц требует достаточно быстрой отрисовки большого количества объектов. Так что лучше использовать готовые решения по визуализации. Из множества всех решений мой выбор пал на OpenGL, ввиду возможности бесплатного использования и изучения его в рамках курса по компьютерной графике.

Так же необходимо отметить, что для большей гибкости отрисовку лучше вынести за пределы системы частиц.

3. Построение системы частиц на базе шаблона проектирования «стратегия».

3.1. Построение класса Particle.

Система частиц управляет большим количеством объектов, так что достаточно важен выбор атрибутов для частицы. Вполне очевидно, что в фиксированный момент времени каждая частица задаётся своим положением в пространстве, цветом и размером. Но также необходимо не забывать про то, что требуется независимое изменение местоположения частиц. Это требует введения полей отвечающих за текущую скорость и, возможно, некоторую силу действующую на частицу. Так же свойства частицы могут изменятся с течением времени, так что необходимо добавить поле, отвечающее за жизненный цикл.

Вот пример реализации структуры данных, отвечающей за частицу.

```
struct Particle {
    double       Life;
    Vector3f     Velocity;
    Vector3f     Gravity;

    Color4f      Color;
    Vector3f     Position;
    Vector3f     Size;
};
```

Vector3f и Color4f - это классы для работы с векторами размерности 3 и 4 соответственно, их реализация в контексте задачи не столь существенна.

3.2. Реализация стратегий.

Под стратегией подразумевается функция, вызываемая при определённом сценарии событий. При реализации системы частиц необходимы стратегии как для инициализации частицы, а именно стратегии инициализации цвета, размера, положения etc, так и отвечающие за обработку частицы, к примеру стратегия, вычисляющая новое положение частицы.

Как пример, стратегия, отвечающая за инициализацию цвета в системе частиц реализующей дым.

```
struct Smoke_InitColor {
    inline void operator () (Particle *particle, double frameTime) {
            double a = 0.6 + ((rand() % 1000) * 1.0) * 0.0001;
            particle -> Color = Color4f(a, a, a, 0);
    }
};
```

Как видно из примера стратегия представляет собой структуру, с заданным оператором «скобки». То есть эту структуру можно рассматривать как функцию. В эту функцию передаётся два параметра: указатель на частицу (particle) и время прошедшее с предыдущей итерации (frameTime). Если необходимость передачи particle очевидна, то необходимость передачи frameTime вызывает вопросы. Это значение необходимо для того модулировать свойства, зависящие от хода времени, к примеру, ускорение частицы. В качестве него может выступать, например, время прошедшее с моменты последней обработки.

Так же, если какая-нибудь стратегия в заданной системе частиц отсутствует, то необходимо сохранить единообразность построения систем частиц, иначе теряется весь смысл шаблонов. Для этого будет использована пустая стратегия, изображённая ниже.

```
struct NullPolicy {
    inline void operator () (Particle *particle, double frameTime) {}
};
```

Для удобства будет использоваться агрегатор стратегий, предоставляющий интерфейс для одновременного вызова некоторого набора стратегий, продемонстрирован ниже.

```
template<class Life, class Position, class Size, class Velocity, class Color, class Gravity>
    class CompletePolicy {
    public:
        Life          life;
        Position      position;
        Size          size;
        Velocity      velocity;
        Color         color;
        Gravity       gravity;

        inline void operator () (Particle *particle, double frameTime) {
                life          (particle, frameTime);
                velocity      (particle, frameTime);
                position      (particle, frameTime);
```

```
        size          (particle, frameTime);
        color         (particle, frameTime);
        gravity       (particle, frameTime);
    }
};
```

3.3. Построение класса ParticleSystem.

Класс ParticleSystem строится с использованием шаблона с тремя параметрами:

- MaxParticleCount – максимально возможное количество частиц.
- InitPolicy – стратегии отвечающие за инициализацию частицы.
- UpdatePolicy – стратегии отвечающие за обработку частицы.

Он должен содержать методы, ответственные за обработку и эмитирование частиц, а также методы необходимые для отрисовки.

Ввиду того, что отрисовка реализуется внешними средствами, необходимо предоставить удобный интерфейс позволяющий это осуществить. Минимально необходимый интерфейс для этого — это агрегатор к массиву, содержащему частицы, и метод возвращающий общее количество живых частиц.

Пример реализации класса ParticleSystem.

```
template<int MaxParticleCount, class InitPolicy, class UpdatePolicy>
class ParticleSystem {
private:
    InitPolicy    particleInit;                    //инициализация частицы
    UpdatePolicy  particleUpdate;                  //обновление состояния частицы
    Particle      particles[MaxParticleCount];     //массив частиц
    int           particleCount;                   //количество живых частиц
    bool          isAlive;                         //существование системы частиц
    double        lastFrameTime;                   //время последнего отрисованного кадр

public:
    inline void Emit(EmitManager emitPoints, int num) {…} //эмиссия новых частиц
    inline void Update(double frameTime) {…}              //обновление состояния системы
    Particle &operator [](int i) {…}                      //доступ к частице по индексу
    int Count() {…}                                       //общее количество частиц.
```

```
    void Init() {…}
};
```

В функцию Emit передаётся два аргумента, первый, имеющий тип EmitManager, это набор точек в которых эмитируются частицы, и второй, имеющий тип int, задающий количество точек, эмитируемых в каждой точке на данной итерации. Эта функция должна добавить и проинициализировать все точки из EmitManager.

EmitManager — это класс, предоставляющий удобный интерфейс для работы с точками для эмитирования. Его реализация не существенна.

В функцию Update передаётся единственный параметр, имеющий тип int и отвечающий за текущее время с момента создания системы частиц. Эта функция обновляет текущее состояние каждой частицы с учётом переданного времени.

Так как необходимо постоянно создавать и уничтожать множество частиц, то для удаления частицы без перевыделения памяти можно использовать одну хитрость. Она заключается в том, что при удалении частицы, на её место копируется частица, находящаяся в конце массива, и на единицу уменьшается счётчик количества частиц.

3.4. Пример создания системы частиц.

Рассмотрим пример создания системы частиц для огня, на базе описанного выше подхода.

```
typedef ParticleSystem
    <12000, CompletePolicy<Fire_InitLife, NullPolicy, Fire_InitSize, Fire_InitVelocity,
Fire_InitColor, NullPolicy>,
    CompletePolicy<Fire_ProcLife, Fire_ProcPosition, NullPolicy, Fire_ProcSize,
Fire_ProcColor, NullPolicy> > FireSystem;
```

В этой системе может быть до 12 тысяч частиц, для которых инициализация описывается следующим агрегатором.

```
CompletePolicy<Fire_InitLife, NullPolicy, Fire_InitSize, Fire_InitVelocity, Fire_InitColor,
NullPolicy>
```

А обработка осуществляется следующим агрегатором.

```
    CompletePolicy<Fire_ProcLife, Fire_ProcPosition, NullPolicy, Fire_ProcSize,
Fire_ProcColor, NullPolicy>
```

Рассмотрим агрегатор, отвечающий за обработку. В него передано шесть стратегий:

стратегии обработки времени жизни, положения в пространстве, размера, цвета, а также две пустые стратегии, чем учитывается факт того, что не производятся обработки как скорости, так и внешних воздействий.

Полученный класс FireSystem уже можно использовать. Подробное описание этого находится в пункте «Отрисовка системы частиц».

4. Создание систем частиц, визуализирующих огонь и дым.

Ввиду того, что в большинстве случаев модулирование физических закон требует огромного количества вычислений, реализация эффекта представляет собой набор методик, дающий приемлемый результат. Приведу набор методик, который использую при визуализации. Также при написании системы присутствует достаточно много задаваемых по умолчанию значений. Выбор этих значений зависит от конкретной реализации.

4.1. Визуализация огня.

4.1.1. Цвет.

Для получения реалистичного цвета огня необходимо задать каждой частице цвет, в котором красная компонента максимальна, зеленая содержит небольшое значение, а синяя отсутствует. Также необходимо использовать аддитивное смешивание. Всё это обусловлено тем, что в местах пересечения частиц, красная компонента останется максимальной, а зелёная увеличится. То есть будет получаться равномерная градация цвета от красного на границах, до желтого в наиболее насыщенном участке пламени.

Также при реализации цвета необходимо учитывать альфа-канал, который должен при инициализации частицы быть нулевым, а в момент смерти быть максимальным. Сценарий его изменения существенной роли не играет, главное чтобы он изменялся по непрерывной функции. Альфа-канал необходим для более гладких границ системы частиц[4].

То есть для реализации цвета огня нужно использовать два типа смешивания, аддитивное и через альфа-канал. При реализации данного подхода на OpenGL возникает проблема. Она возникает ввиду того, что OpenGL не позволяет одновременно использовать два типа смешивания. Но эта проблема легко решаема. Можно самостоятельно рассчитывать смешивание через альфа-канал, а затем отдавать на аутсорс OpenGL расчёт аддитивного смешивания.

Если *bgColor* - цвет фона, *curColor* - цвет добавляемого пикселя, *alpha* - коэффициент альфа-канала (должен лежать в $[0,1]$), *resColor* - полученный результат, то самостоятельно рассчитать смешивание можно по следующей формуле:

$$resColor = bgColor * (1 - alpha) + curColor * (1 - alpha)$$

4.1.2. Способ отрисовки частицы

Частицу лучше всего отрисовывать с помощью текстуры. Текстуру необходимо задать таким образом, чтобы цвет был наиболее насыщенным в центре текстуры с плавным переходом в полностью прозрачный по краям[3]. Вид текстуры может быть произвольный, наиболее подходящий выбирается в частной ситуации, я использую следующую текстуру (рисунок 1).

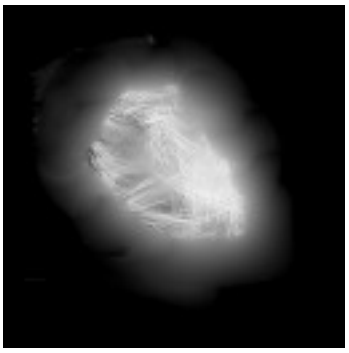

Рисунок 1: Текстура для визуализации частицы.

4.1.3. Положение частицы.

Каждой частице задаётся скорость, которая в последствие изменяется по линейному закону. Причём она направленна направлена вверх с небольшим отклонением в сторону. Отклонения в сторону выбираются случайно, но достаточно маленькие и из небольшого диапазона. Также необходим большой разброс скоростей по координате, отвечающей за подъём частицы. Всё это позволяет создать реалистичный эффект природного процесса. Также можно добавить небольшие колебания из стороны в сторону, это создаёт более дрожащее пламя.

4.1.4. Размер частицы.

Для реализации эффекта частицы должны создаваться различного размера, также можно уменьшать размер частицы в ходу жизни, это сделает языки пламени более острыми.

4.1.5. Итог

В итоге получается система частиц, с помощью которой можно реализовать огонь подобный

изображенному на рисунке 2.

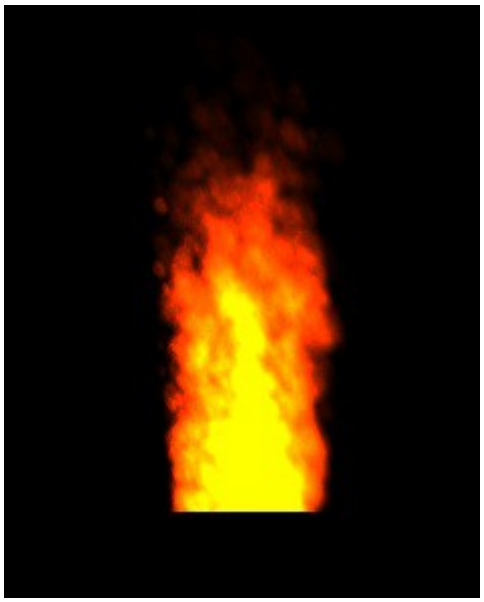

Рисунок 2: Огонь, полученный с помощью данного подхода.

4.2. Визуализация дыма.

4.2.1. Цвет.

Цвет дыма реализуется намного проще, чем цвет огня. Каждой компоненте цвета необходимо задать очень маленькое, значение. Причём значения компонент должны быть равны между собой. То есть задаётся практически черный, тёмно-серый цвет. В последствие, при визуализации, все частицы аддитивно смешиваются.

4.2.2. Способ отрисовки частицы

Частицы лучше всего отрисовывать с помощью текстуры округлой формы, других ограничений не накладывается. Текстура приведённая на рисунке 1, вполне подойдёт.

4.2.3. Положение частицы.

Каждой частице задаётся скорость, которая в последствие изменяется по линейному закону. Причём она направленна направлена вверх, с небольшим отклонением в сторону. Отклонения в сторону выбираются случайно, но достаточно маленькие и из небольшого диапазона. Также

необходим небольшой разброс скоростей по координате, отвечающей за подъём частицы. А вот колебания частицы лучше не добавлять, но при желании можно добавить медленное вращение.

4.2.4. Размер частицы.

Для реализации эффекта частицы должны создаваться различного размера, со временем жизни они должны увеличиваться.

4.2.5. Итог

В итоге получается система частиц, с помощью которой можно реализовать дым подобно изображенному на рисунке 3.

Рисунок 3: Огонь, полученный с помощью данного подхода.

5. Отрисовка системы частиц.

Отрисовка системы частиц с помощью OpenGL не представляет особой сложности.

Для начала необходимо объявить и проинициализировать систему частиц, внешний счётчик frameTime, который задаёт время прошедшее с начала жизни системы, а также добавить эмиттер.

```
int             frameTime;
FireSystem      fire;
EmitManager     fireEmit;

frameTime = 0;
fire.Init();
fireEmit.Add(Vector3f(3.5,1,0), Vector3f(4.5,1,0), 0.03);
```

Так же требуется постоянно обновлять состояние системы, что можно реализовать через функцию таймера следующим образом.

```
void timer(int value) {
   frameTime++;
   fire.Update(frameTime*1.0/2);
   fire.Emit(fireEmit, 5);
   glutPostRedisplay();
   glutTimerFunc(20, timer, 0);
}
```

Собственно для отрисовки системы частиц, нужно корректно инициализировать текстуру и в `glutDisplayFunc`, поместить код, аналогичный следующему.

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
glEnable(GL_TEXTURE_2D);

for (int i = 1; i < fire.Count(); ++i) {
   glColor3fWithAlpha(fire[i].Color.Red, fire[i].Color.Green, fire[i].Color.Blue,
fire[i].Color.Alpha);
   glPushMatrix ();
         glTranslatef(fire[i].Position.X, fire[i].Position.Y, fire[i].Position.Z);
         DrawParticle(fire[i].Size.X, fire[i].Size.Y);
   glPopMatrix ();
}
```

```
glDisable(GL_BLEND);
glDisable(GL_TEXTURE_2D);
```

Где функция `glColor3fWithAlpha` отвечает за учёт альфа-канала, а функция `DrawParticle` отвечает за отрисовку прямоугольника заданного размера, с натягиванием на него текстуры.

6. Тесты

Качественным показателями системы частиц является скорость работы. Для этого целесообразно измерить время работы для различного количества частиц. В приведённых тестах меряется зависимость fps(кадров в секунду) от количества частиц. Все тесты проводятся для написанной системы частиц, визуализирующей огонь. Вычисления проводятся на процессоре Intel Core i7 3770K(3500MHz).

На диаграмме 1 построены графики зависимости fps от количества частиц для трёх случаев: полноценно работающая система частиц (график 1), система частиц с отключенными текстурами (график 2), система частиц с частицами визуализированными как точки (график 3).

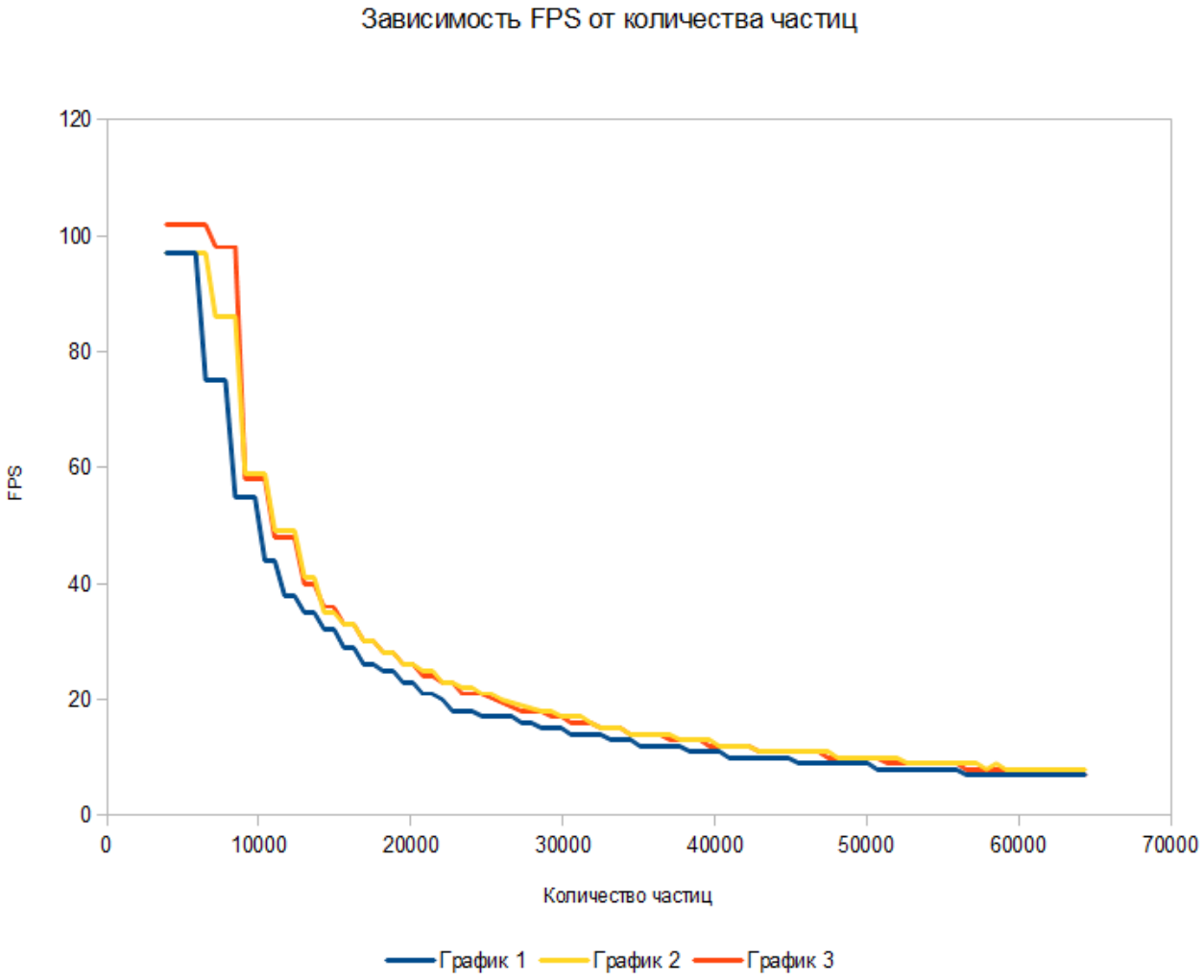

Рисунок 4: Сравнение производительности системы частиц при различных способах визуализации.

Как видно из рисунка 4, изменения связанные с примитивизацией отображаемых частиц не существенно влияют на производительность.

Намного более интересен рисунок 5. На ней показано сравнение производительности системы частиц с полноценной визуализацией (график 1) и без визуализации вообще (график 2).

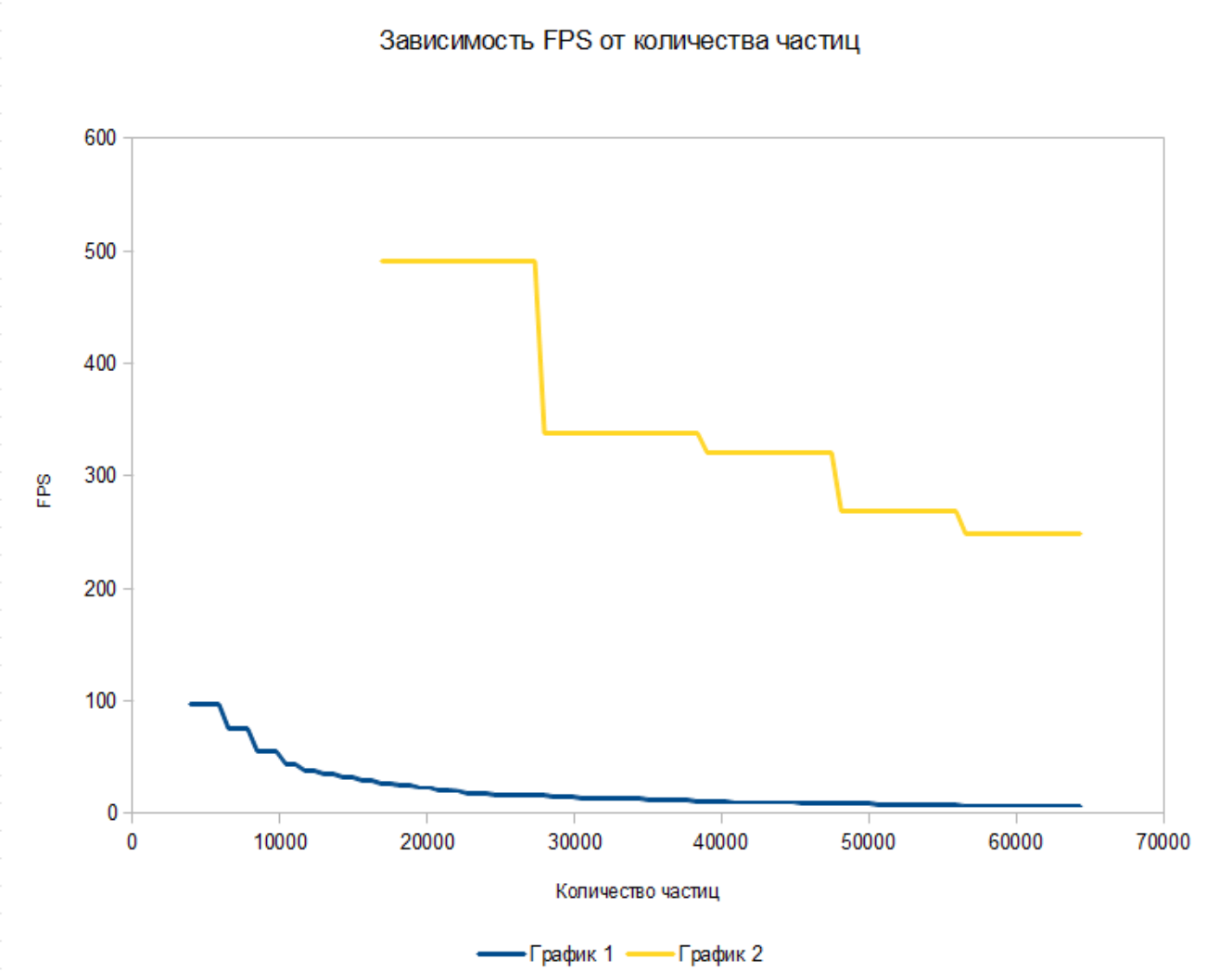

Рисунок 5: Сравнение производительности системы частиц с визуализацией и без.

Как видно из рисунка 5, при реализации системы частиц, визуализация намного более затратный процесс, чем обработка.

7. Заключение.

Разработана и реализована быстро работающая и легко-расширяемая система частиц. На её базе построены системы реализующие реалистичный огонь и дым. На основе тестов производительности выяснено, что примитивизация процесса отрисовки не существенно влияет на производительность, а также, что обработка системы частиц выполняется намного быстрее её визуализации.

8. Литература.

[1] John van der Burg. *Building an Advanced Particle System* // Gamasutra [сайт]. URL:http://www.gamasutra.com/view/feature/131565/building_an_advanced_particle_.php.

[2] Kent "_dot_" Lai. *Designing an Extensible Particle System using C++ and Templates* // GameDev.net [сайт]. URL:http://archive.gamedev.net/archive/reference/articles/article1982.html

[3] Hubert Nguyen. *Fire in the "Vulcan" Demo* // GPU Gems [сайт]. URL:http://http.developer.nvidia.com/GPUGems/gpugems_ch06.html

[4] Sanandanan Somasekaran. *Using Particle Systems to Simulate Real-Time Fire.* – Australia: The University of Western Australia, 2005.