

Мультимедиа

Практика №4

Рябинин Константин Валентинович

e-mail: icosaeder@ya.ru

jabber: icosaeder@jabber.ru

Пермь, 2013

Под оптимизацией процесса рендеринга понимается комплекс мер, которые можно предпринять, чтобы увеличить производительность графической программы

Различают несколько базовых подходов:

- Разумная экономия на разрешении графических данных (трёхмерных моделей и текстур)**
- Использование встроенных в конкретную систему визуализации средств оптимизации (например, использование буферных объектов в OpenGL)**
- Активное искусственное упрощение сцены без снижения качества результирующей картинки**
- Оптимизация расчётов (отказ от точных вычислений в пользу быстрых аппроксимаций)**
- . . .**

Под упрощением сцены понимается сокращение количества выводимых объектов и их геометрической сложности

Существует большое количество алгоритмов такого упрощения:

- **Z-буферизация**
- **Отсечение задних граней**
- **Отсечение по усечённой пирамиде видимости**
- **Отсечение заслонённых объектов**
- **Портальный рендеринг**
- **Уровни детализации**
- **Оптимизация шейдеров**
- **...**

- Данный алгоритм был подробно рассмотрен ранее
- Идея – в процессе рендеринга выполнять отсечение частей объектов, закрытых частями более близко расположенных объектов
- Повысить эффективность z-буферизации способен **алгоритм художника** – сортировка объектов по удалённости и вывод их, начиная с самых ближних
- Однако алгоритм художника не применим в том случае, если для объектов используется альфа-смешивание. В этом случае, для получения корректного результата необходимы прямо обратные действия (вывод объектов, начиная с самого дальнего)

Отсечение задних граней (*backface culling*) – это алгоритм, при котором

- Для каждого полигона определяется, внешней или внутренней стороной он повёрнут к наблюдателю (определение производится на основе порядка обхода вершин)
- В зависимости от результата определения принимается решение, выводить данный полигон, или нет
- Алгоритм позволяет сэкономить на задних гранях объекта, для которых делается предположение, что они закрыты передними
- Алгоритм применим только для «замкнутых» объектов (полностью ограничивающих какую-то часть пространства)
- Отсечение граней по их ориентации относительно наблюдателя поддерживается библиотеками визуализации. Например, в OpenGL его можно включить вызовами
`glEnable(GL_CULL_FACE);`
`glCullFace(GL_BACK); // либо glCullFace(GL_FRONT);`

Отсечение по усечённой пирамиде видимости
(*frustum culling*) – это алгоритм, при котором

- Для каждого объекта определяется, попадает ли он в поле зрения камеры (в роли поля зрения камеры выступает, в общем случае, усечённая пирамида видимости)
- Объект отправляется на графический конвейер только в том случае, если он находится в поле зрения камеры
- Для граничных случаев (часть объекта в поле зрения камеры, а часть – за пределами) существуют различные подходы, но чаще всего, чтобы не тратить время на разбиение объекта на множество подобъектов и вычленение из них видимых, объект полагается видимым и отрисовывается
- Определение попадания в поле зрения осуществляется чисто геометрически, на основе уравнений плоскостей, составляющих усечённую пирамиду видимости
- Геометрический тест позволяет определить, находится ли точка в поле зрения. Производить этот тест для всех вершин объекта неэффективно, поэтому используются т. н. **ограничивающие объёмы**

Отсечение по усечённой пирамиде видимости
(*frustum culling*) – это алгоритм, при котором

- **Ограничивающий объём** – это геометрическое тело простой формы, описанное вокруг объекта
 - Чаще всего в качестве такого тела используется **параллелепипед** (*bounding box*) или **сфера** (*bounding sphere*)
 - Использование параллелепипеда является более гибким, так как, обладая тривиальной формой, он способен, однако, достаточно точно ограничить произвольный объект
 - Тест видимости производится для ограничивающего объёма, а не для самого объекта
- Помимо этого, ограничивающие объёмы могут быть использованы для определения столкновений объектов, при моделировании физики твёрдого тела

Отсечение заслонённых объектов (*occlusion culling*) – это алгоритм, при котором

- Заранее определяются объекты, полностью закрытые другими объектами
- Невидимые объекты не отправляются на графический конвейер
- Отличие от z-буферизации состоит в том, что невидимые объекты *не отправляются на конвейер*
- Поиск заслонённых объектов – сам по себе довольно сложная задача. Поэтому использование этого алгоритма имеет смысл лишь
 - На сценах с большим количеством объектов разного размера
 - На сценах, моделирующих помещения (в которых, например, присутствуют стены и перегородки)

Портальный рендеринг (*portal rendering*) – это алгоритм визуализации закрытых пространств, при котором

- Для сцены, представляющей собой цепочку комнат, соединённых дверными и оконными проёмами (**порталами**), определяется видимая в данный момент площадь
- На конвейер отправляются только те объекты, которые принадлежат этой площади
- Для эффективного поиска видимых областей и объектов, принадлежащих этим областям, используется деление пространства (*space partitioning*)
- Деление пространства бывает двоичным (*BSP*), квадратичным (*quadtree*) и октарным (*octree*)
- Существует целый ряд алгоритмов деления пространства
- Результат деления сохраняется в древовидную структуру (бинарные, квадратичные и октарные деревья соответственно)
- Деление пространства используется для быстрого поиска объектов и определения их взаимного расположения
- Однако на динамических сценах, когда положение объектов постоянно изменяется, вместо прироста производительности можно получить её падение, так как деревья придётся всё время перестраивать

Уровни детализации (*LOD – level of detail*) – это изменение детализации объектов в зависимости от их удалённости

- Частным случаем LOD является мип-мэппинг текстур
- Идея состоит в том, что для удалённых объектов снижается детализация их представления
- Различные уровни детализации чаще всего генерируются автоматически на основе оригинала, разрешение которого берётся за максимальное
- Однако иногда разные уровни детализации подготавливаются заранее и загружаются из файлов
- Проблемой является переключение между уровнями. Если в случае текстур можно использовать интерполяцию цвета для сглаживания «скачка» с одного уровня на другой, то в случае трёхмерных моделей простая интерполяция невозможна
- Для трёхмерных моделей могут быть использованы алгоритмы **морфинга** (плавного перехода между формами), однако чаще всего используется скачкообразное изменение
- В качестве крайней меры снижения детализации трёхмерных объектов могут быть использованы импостеры, которые были рассмотрены ранее

- Не смотря на то, что шейдеры выполняются на GPU и предоставляют мощный аппарат ускорения вывода графики, их код также требует оптимизаций
- Хотя справедливо было бы ожидать, что это будет сделано автоматическим оптимизатором, полагаться на него не следует (так как на практике он далеко не всегда работает, если вообще работает)
- Для написания оптимального кода шейдеров необходимо иметь представление об особенностях GPU
- Из-за специфических оптимизаций код шейдеров становится сложным для чтения и понимания, поэтому его следует комментировать

- Использование операций с векторами там, где это ВОЗМОЖНО
- Использование «коктейлей» (swizzling):
`alpha.xy = beta.xy;`
- Использование встроенных функций там, где это ВОЗМОЖНО
 - Линейная интерполяция
`result = mix(alpha, beta, t);`
 - Скалярное произведение
`result = dot(alpha, beta);`
 - Нормировка вектора
`result = normalize(alpha);`
 - Длина вектора
`result = length(alpha);`

- Отражение вектора относительно вектора
`result = reflect(alpha, beta);`
- Преломление вектора
`result = refract(alpha, beta, ior);`
- ...
- Избежание ветвлений, так как они могут нарушить параллелизм (dynamic branching problem); во многих случаях ветвления на уровне логики могут быть заменены рядом функций:
 - Пороговая функция
`result = step(edge, x);` $step(edge, x) = \begin{cases} 0, & x_i < edge_i \\ 1, & x_i \geq edge_i \end{cases}$
 - Минимум / максимум
`result = min(alpha, beta);`
`result = max(alpha, beta);`

- Избежание операций над константами (!!!)

```
// result = alpha * 2 * 5 * sqrt(2)  
result = alpha * 14.14214;
```
- Избежание делений в пользу умножений (!!!)
- Объявление всех констант с ключевым словом `const`
- Избежание зависимого чтения текстур
- Использование MAD-инструкций (multiply-add)
 - Вычисление выражений вида $A * B + C$ за один такт процессора, например

```
result = 0.5 * alpha + 0.5;
```


лучше, чем

```
result = (alpha + 1.0) / 2.0;
```


и даже лучше, чем

```
result = (alpha + 1.0) * 0.5;
```

Задание: оптимизировать шейдер

15 / 15

```
varying vec4 v_color;
uniform float u_t;

void main()
{
    vec4 color;
    vec4 input = v_color;
    float length = sqrt(input.r * input.r + input.g * input.g +
                        input.b * input.b + input.a * input.a);

    input.r /= length;
    input.g /= length;
    input.b /= length;
    input.r += 3.0 / 2.0;
    input.g += 3.0 / 2.0;
    input.b -= 3.0 / 2.0;
    if (v_color.r > v_color.g)
        color.r = (1.0 - u_t) * (input.r / 2.0) + u_t * (input.g / 2.0);
    else
        color.r = (1.0 - u_t) * input.r + u_t * input.b;
    color.g = sin(input.g);
    color.b = sin(input.b);
    color.a = 1.0;
    gl_FragColor = color;
}
```