

УДК 004.925

Разработка и оптимизация программы визуализации трёхмерных сцен

***Н.И. Амиантов**, студент*

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Программное обеспечение ЭВМ и информационные технологии»*

Научный руководитель: И.В. Ломовской, ассистент

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Программное обеспечение ЭВМ и информационные технологии»*

irudakov@bmstu.ru

Введение

Существует много примеров программ, отрисовывающих в реальном времени трёхмерные сцены с той или иной функциональностью. В качестве одной из них можно привести старую игру Quake, которая работала на мощностях центрального процессора на старых компьютерах. Её программный визуализатор поддерживал отрисовку в реальном времени 3D-моделей, достаточно просторных карт, текстурирование в реальном времени на не самых мощных компьютерах того поколения.

В рамках курсовой работы по машинной графике была разработана программа визуализации, способная в реальном времени отрисовывать сложные сцены, содержащие несколько текстурированных объектов с расчётом освещения. В статье описываются идеи и решения, использованные в работе над этой программой, измеряется и сравнивается производительность при разных опциях программы.

1. Оптимизации средствами языка программирования

При написании программы использовался язык C++, который, с одной стороны, является современным языком программирования с поддержкой многих удобных конструкций, а с другой, обеспечивает скорость выполнения на уровне Си. Использовались средства шаблонного метапрограммирования в языке. Идея этого метода заключается в использовании шаблонов для написания функций или классов, которые выполняют некий набор методов из статического класса-аргумента шаблона по заданному алгоритму. Такой метод обеспечивает большую безопасность и удобство написания по сравнению с использованием макросов за счёт обработки шаблонных конструкций на этапе компиляции. В частности, обеспечивается проверка типов, более ясно выделяются

ошибки, код выглядит красивее чем в макросах. Грамотное использование этого приёма и кодового слова `inline` даёт возможность писать очень гибкий и оптимальный код, и позволяет практически полностью избежать повторений. Недостатком данного метода является, в первую очередь, понижение читаемости, хотя код остаётся понятнее и чище, чем при использовании макросов. Пример использования шаблонного метапрограммирования приведён в листинге 1. Результат после прохода компилятором приведён в листинге 2. В работе этот метод использовался для написания шаблонных функций отсечения, которые на этапе компиляции генерируются для каждой отсекаемой плоскости. (см. раздел 2.2) Также это используется для написания «универсального обходчика треугольников по пикселям». (см. раздел 2.4).

Листинг 1: Пример использования шаблонного метапрограммирования в C++

```
struct Adder {  
    static inline int Apply(const int a, const int b) {  
        return a + b;  
    }  
};  
  
struct Subtractor {  
    static inline int Apply(const int a, const int b) {  
        return a - b;  
    }  
};  
  
template <class Operator> int ApplyBinary(const int a, const int b) {  
    return Operator::Apply(a, b);  
}
```

2. Описание и оптимизации программы визуализации

Основная особенность программы заключается в том, что её функционал настраивается на этапе компиляции при помощи макросов. Настраивать можно множество параметров, например, освещение, оптимизацию нормалей, многопоточный Z-буфер, включение отсечений по нормальям, использование сторонних библиотек. В проекте интенсивно используются макросы, что позволяет генерировать эффективный код для каждой конфигурации, избегая лишних вычислений за счёт проверок на включённость той

или иной опции во время выполнения программы, в частности, возможности оптимизировать код в циклах без подобных проверок.

Листинг 2: Результат к примеру 1

```
ApplyBinary<Adder>:
    leal (%rdi,%rsi), %eax
    ret
ApplyBinary<Subtractor>:
    movl %edi, %eax
    subl %esi, %eax
    ret
```

Рассмотрим общую последовательность действий, необходимую для растеризации трёхмерной модели [6][5]:

1. Преобразование координат
2. Отсечение линий за границей экрана
3. Отсечение невидимых линий
4. Заливка

Ниже каждый из этих этапов рассматривается подробнее, обсуждаются вопросы его оптимизации.

2.1. Преобразование координат.

Канонический способ преобразования координат — домножение вектора координат на матрицу M размером 4×4 (слева или справа — зависит от того, представляется ли вектор строкой или столбцом), которая переводит вектор в систему координат камеры. Однако возможно несколько ускорить этот процесс, а также добавить несколько дополнительных шагов.

Хранение сцены и моделей Моделью называется описание формы визуализируемого объекта. Сценой называется набор моделей в виртуальном пространстве, которое проецируется на видовой экран. В работе используются полигональные модели, которые хранятся в виде массивов точек и треугольников, а сцену — как связанный список объектов, содержащих указатель на модель и некоторые дополнительные поля для кэширования. Структура данных «массив» позволяет с большой скоростью обходить все треугольники модели, к тому же в разработанном алгоритме

недопустимо изменения этих массивов после добавления модели на сцену. Подобный формат хранения моделей (в виде структур-треугольников с индексами точек) в частности используется в графических библиотеках OpenGL [1] и DirectX [2].

Матрицы. Классический способ хранения точек в виде вектора длиной 4 (X, Y, Z, W) подразумевает возможность проведения перспективных преобразований. Они служат для учёта перспективы — зависимости видимого размера объекта от его дальности. Можно сократить расходы памяти и процессорного времени, если будем хранить лишь 3 координаты для каждой точки без значения W и матрицы размером 3×4 . Таким образом, исключается возможность перспективного преобразования и проводим его отдельно заданной формулой.

Отсечение по направлению грани. Для ускорения вычисления кадра можно применить отсечение по направлению грани. Идея этого отсечения сводится к тому, что для каждой грани известна также нормаль к ней, и считается, что грань должна быть видима только с этой стороны. Соответственно, если проекция этой нормали на ось камеры положительна, то грань расположена тыльной стороной к наблюдателю, и возможно отбросить её целиком. Казалось бы, достаточно проверить значение скалярного произведения вектора направления камеры на вектор нормали. Однако, такая проверка не срабатывает из-за наличия перспективного преобразования, которое, в том числе, изменяет направления нормалей к поверхностям. Поэтому после всех преобразований необходимо высчитать новую нормаль к поверхности, и сравнивать её с направлением камеры в пространстве камеры, которое, по сути — единичный вектор Z .

Векторы для освещения. Для выбранной нами модели освещения требуются векторы направления от точек к камере, источникам освещения, нормали. С помощью макросов рассчитываются для каждой конфигурации только те векторы и величины, которые необходимы для выбранного способа закраски. Также оптимизируются все циклы, исходя из необходимых расчётов. Например, если мы рассчитываем один уровень освещённости для каждого треугольника, мы можем рассчитать необходимые величины только для одной точки в середине треугольника.

Кэширование. Многие операции по преобразованию достаточно выполнять лишь при изменении положения моделей. В работе полагается, что положение камеры изменяется постоянно, и оптимизации, предполагающие кэширование зависящих от положения камеры величин, не имеют смысла. Однако же, сами объекты перемещаются достаточно редко, и можно выполнять операции по преобразованию их координат в глобальные один раз для каждого изменения положения и хранить промежуточные значения в кэше. Такой подход существенно экономит процессорное время за счёт

большого количества занимаемой памяти.

Операции выделения и освобождения памяти достаточно трудоёмки. Они избегаются, используя пул уже доступной памяти. В частности, для каждого объекта создаётся массив с уже обработанными треугольниками и точками. Эти массивы используются на протяжении многих кадров. В итоге, удалось добиться отсутствия операций получения и освобождения памяти в куче во время отрисовки подавляющей части кадров. Исключения составляют кадры, в которые входят новые объекты. Стек используется только для массивов с фиксированным и заранее рассчитанным размером и некоторых небольших объектов (например, итераторов).

Использование сторонних библиотек для работы с матрицами. Для ускорения производимых вычислений можно использовать сторонние библиотеки, применяющие векторные инструкции процессора для быстрой работы с векторными и матричными типами, и оптимизирующими производимые операции. В качестве примера была добавлена возможность использовать стороннюю библиотеку Eigen3 [3], которая многократно ускоряет вычисления над векторными типами данных.

2.2. Отсечение линий за границей экрана.

После преобразований все треугольники отсекаются по плоскости Z_0 и плоскостям, ограничивающим видовой экран. Каждый треугольник обладает фиксированным количеством вариантов разбиений (0, 1 или 2 треугольника в результате), причём в большинстве случаев отсечение по всем плоскостям будет оставлять треугольник без изменений или отсекает его целиком (обычно объект будет либо помещаться на видовой экран целиком, либо отсутствовать на нём совсем). Был применён такой алгоритм — треугольники отсекаются по одному по каждой плоскости в отдельности с помощью поиска пересечений сторон треугольника с плоскостью, причём каждое отсечение может вернуть 0, 1 или 2 треугольника. Если обнаружено, что треугольник не разбивался и был полностью отсечён очередной плоскостью, работа останавливается.

Для оптимизации функций использовалось шаблонное метапрограммирование — была написана общая процедура отсечения, которая принимала в качестве шаблонов классы с методами для проверки на выход за границу и для нахождения пересечения. В качестве структуры хранения данных использовались массивы на стеке с заранее аналитически рассчитанным размером для максимально возможного количества данных, полученных из одного треугольника после всех отсечений. Расположение массива на стеке даёт мгновенное выделение памяти, а поскольку размер заранее вычислен и не слишком велик, угроза переполнения стека отсутствует.

2.3. Отсечение невидимых линий.

В качестве алгоритма для отсечения невидимых линий был выбран Z-буфер — один из самых быстрых на современной аппаратуре из-за большой скорости работы с памятью, а также тривиальный в написании. Однако, можно попытаться оптимизировать как очистку буфера, так и работу с ним. Было сделано несколько оптимизаций:

- Использование функции C `memset`, в которой применяются инструкции процессора, ускоряющие заполнение большого объёма данных.
- Реализация многопоточности.
- Оптимизация обхода.

Будут описаны последние две.

Реализация многопоточности. Эта оптимизация позволяет использовать ресурсы многоядерного процессора для подготовки буфера к следующему кадру во время отрисовки текущего. Для работы используются два буфера — для текущего и следующего кадров. В начале работы программы создаётся поток для очистки буфера, в котором выполняется чистка следующего буфера, в то время как программа работает с текущим буфером. После окончания отрисовки текущего кадра программа ждёт окончания работы другого потока, и буферы меняются местами. Такая оптимизация, к сожалению, не дала серьёзных улучшений в производительности.

Оптимизация обхода. Из-за особенностей, связанных с методом обхода треугольников для заливки (подробнее см. [2.4](#)), вызовы к буферам будут производиться для длинных последовательностей точек. Соответственно, нет необходимости вычислять полностью адрес для каждой точки в буфере. Достаточно вычислить его один раз для первой точки в последовательности, а дальше лишь увеличивать её на единицу. Это позволяет обойтись без множества операций умножения для каждого кадра и заметно ускоряет работу.

2.4. Заливка.

Задача заливки в данном случае будет иметь следующую формулировку. Дан треугольник ABC с точками в трёхмерном пространстве. Для каждой точки треугольника необходимо выполнить некоторую операцию над этой точкой и некоторыми величинами, причём и точка, и величины являются билинейно интерполированными, исходя из знания значений на вершинах треугольника.

Обход треугольника. В основе алгоритма заливки в программе используется модифицированная версия алгоритма списка активных рёбер (SAP), оптимизированная

для работы с треугольниками.

В случае треугольника имеется три линии — AB , BC и AC . Для обхода и билинейной интерполяции на таком треугольнике был придуман следующий алгоритм:

1. Отсортируем точки треугольника по высоте (пусть отсортированные точки — pn)
2. Линия p_0p_2 — длинная, линии p_0p_1, p_1p_2 — короткие
3. Вплоть до значения высоты точки p_1 интерполируем билинейно по прямым p_0p_2, p_0p_1 , далее — по прямым p_0p_2, p_1p_2 .

Архитектура обходчика. Из-за множества различных конфигураций возникает задача написания некоего универсального «обходчика треугольников», который будет одновременно интерполировать несколько различных величин и выполнять некую операцию над каждой точкой внутри треугольника. При этом надо учитывать, что функции обхода линии и обработки точек будут вызываться сотни тысяч раз, поэтому в этом месте необходима серьёзная оптимизация. Однако, использование шаблонного метапрограммирования позволяет нам написать некую функцию «абстрактного обходчика». Она принимает в качестве шаблонного аргумента классы, содержащие функции шага и выполнения некоей операции для каждой точки. Сами классы собираются из необходимых составляющих «кирпичиков» — классов, отвечающих за интерполяцию и содержание какой-то одной величины. Структура используемых обработчиков представлена на рисунке 1. В зависимости от выбранного функционала отрисовщика используется один из этих обработчиков. Таким образом, повторение кода минимально, и за счёт шаблонного метапрограммирования и использования директивы `inline` достигается практически полное отсутствие лишних инструкций в выходном коде.

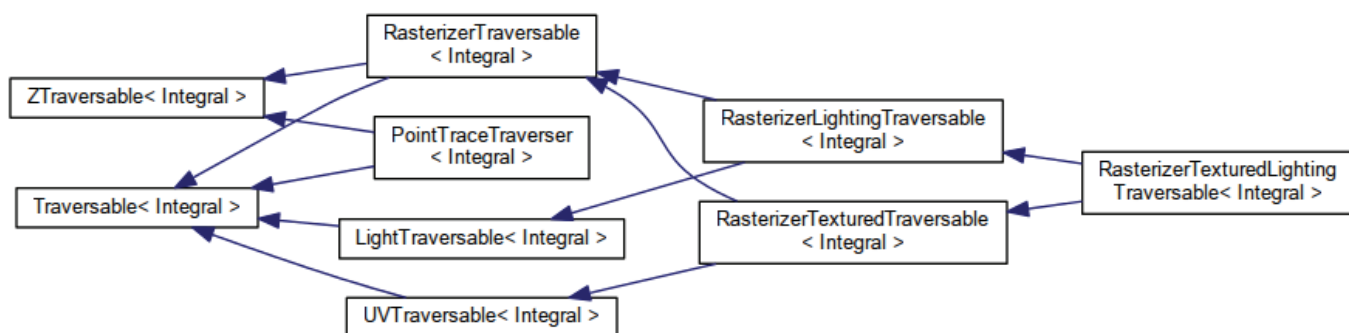


Рис. 1. Иерархия отрисовщиков

3. Результаты исследований

Полученная в результате программа способна обеспечивать 70-80 кадров в секунду для двух источников освещения с разрешением 1366x768 и тремя сложными объектами, занимающими в сумме шестую часть экрана на процессоре Intel Core i5 M450, используя лишь одно ядро. Была произведена серия замеров производительности в зависимости от различных настроек программы визуализации и параметров компиляции.

3.1. Оптимизации компилятора.

Компилятор позволяет собирать программу с различными опциями. Было изучено влияние некоторых из них на скорость программы — в частности, четыре уровня оптимизации скорости -O0, -O1, -O2 и -O3. Также была рассмотрена опция -ffast-math, включающая оптимизации работы с числами с плавающей запятой которые могут повлечь потерю точности, -funroll-loops, разворачивающая циклы с постоянным количеством шагов и -funswitch-loops, преобразующая циклы, в которых проверяется не зависящее от цикла условие, в два цикла для разных результатов проверки.

Параллельно проводилось сравнение скорости работы программы с использованием собственной библиотеки матриц и векторов по сравнению с библиотекой Eigen3.

Ключи компилятора:

```
-Fstack-protector -funsafe-loop-optimizations -fPIE -mtune=native -  
march=native -fomit-frame-pointer -DGOURAUD_SHADING -DAFFINE_TEXTURES  
-DDUMB_NORMAL_CLIPPING
```

Были произведены измерения работы программы на различных уровнях оптимизации с тестовой сценой. Результаты отражены на рисунке [2](#). Результаты показывают влияние уровня оптимизаций на качество конечного машинного кода при активном использовании шаблонного метапрограммирования. Особенно хорошо это видно на активно использующей эти возможности внешней библиотеке векторов и матриц. Также этот график демонстрирует прирост производительности при использовании векторных инструкций процессора для работы над матрицами и векторами.

На рисунке также видно, что уровень оптимизации -O0 даёт на выходе крайне медленный исполняемый файл, а уровни -O2 и -O3 практически неотличимы по скорости. Это связано с тем, что -O3 добавляет к -O2 лишь несколько небезопасных оптимизаций, которые проявляются хорошо только в специфических случаях.

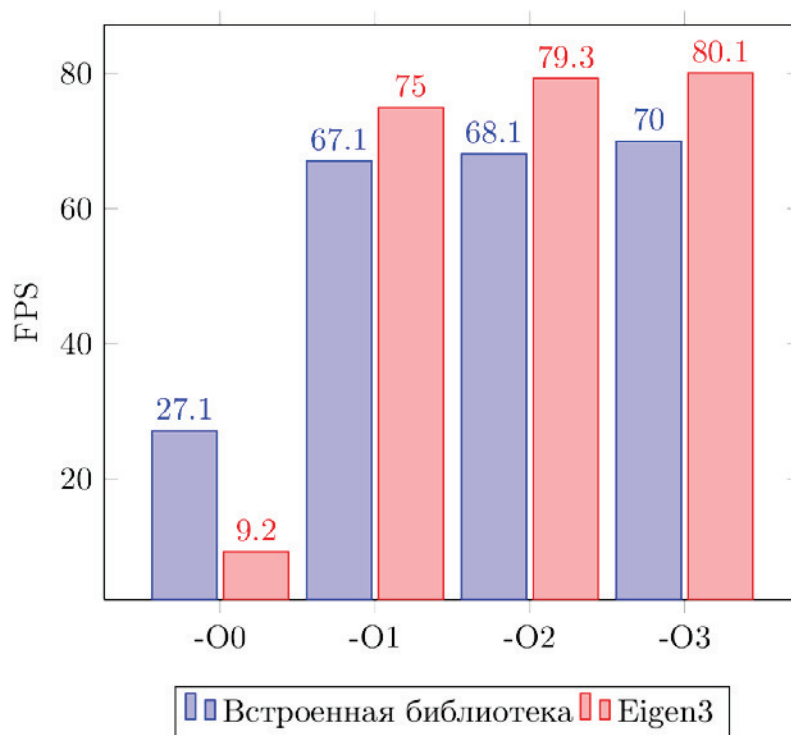


Рис. 2. Сравнение оптимизаций скорости

В дальнейшем все измерения проводились на уровне оптимизации -O3 с применением внешней библиотеки, что позволяло эффективно разворачивать код, сгенерированный с помощью шаблонов, встраивать небольшие функции прямо в код вызывающей процедуры, и оптимизировало прочие узкие но неинтересные для нас места (в частности операции над векторами и матрицами, которые во внешней библиотеке оптимизированы с применением ассемблерных вставок).

Была также рассмотрена роль дополнительных оптимизаций для выбранной конфигурации. Результаты применения различных оптимизаций приведены на рисунке [3](#).

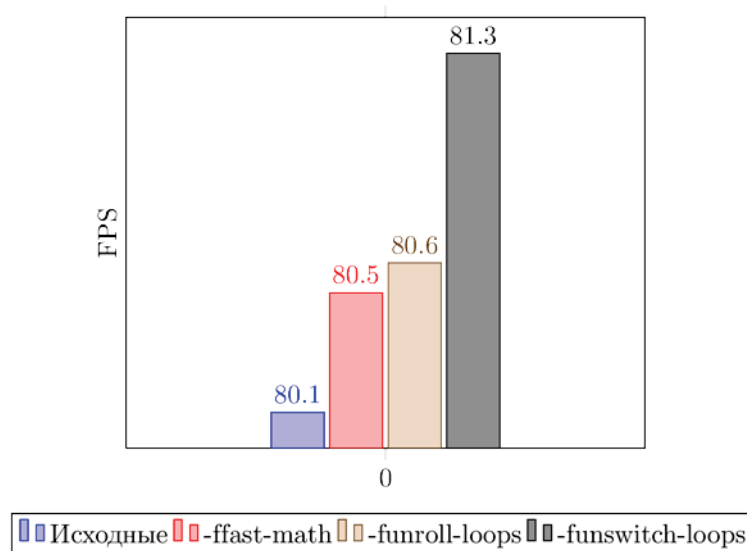


Рис. 3. Дополнительные оптимизации компилятора

Оптимизация `-funroll-loops` считается спорной, поскольку программа начинает занимать больше места в памяти (и в кэше процессора), и современные процессоры конвейерно оптимизируют подобные циклы. В итоге в финальные параметры сборки были добавлены `-ffast-math` `-funswitch-loops`, что согласно исследованию положительно влияет на скорость программы.

3.2. Интерполяция текстур.

Сравнивались качество картинки и скорость работы программы при использовании разных способов интерполяции текстур — аффинного и перспективного. Сравнение скорости приведено на графике 4.

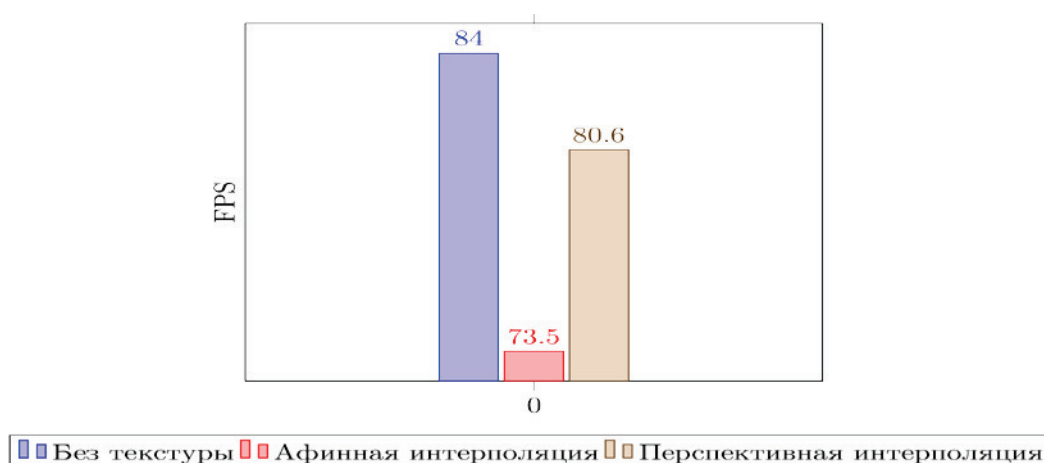
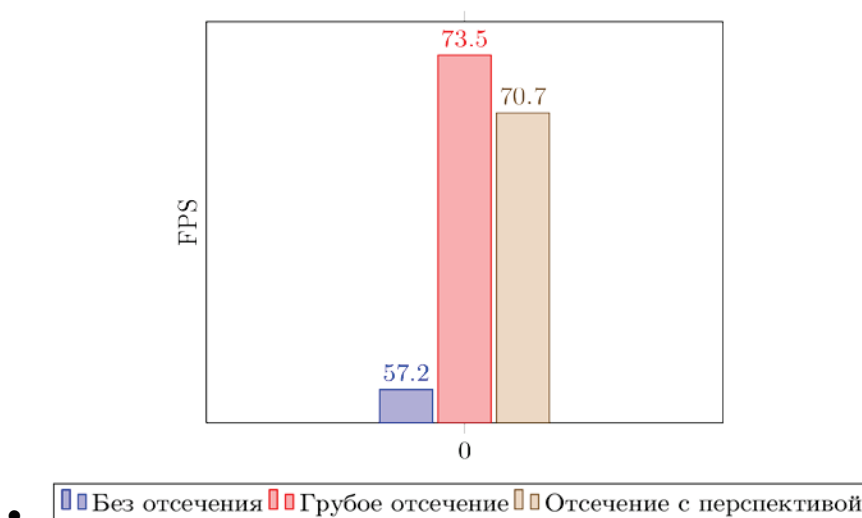


Рис. 4: Скорость интерполяции текстур

Сравнивалась скорость работы при использовании трёх видов отсечения по нормальям:

- Отсутствие отсечения
- Грубое отсечение (без учёта перспективы)
- Отсечение с учётом перспективы



• Рис. 5: Скорость с отсечением по нормальям

Результаты представлены на рисунке 5.

Из графика видно, насколько существенный прирост скорости даёт отсечение по направлению нормали. Следует учитывать, что при использовании грубого отсечения проявляются достаточно заметные дефекты исчезновения граней тел, которые проецируются на экран ближе к его краям. В итоге, лучше использовать отсечение с учётом перспективы, чтобы не терять в качестве картинки.

3.4. Тонирование.

В программе сравнивалось качество различных алгоритмов тонирования:

- Без учёта освещения
- Плоская тонировка
- Тонировка по Гуро
- Тонировка по Фонгу

Сравнивалось как качество картинки, так и скорость работы программы при выбранной тонировке. Результаты по скорости работы представлены на графике [6](#).

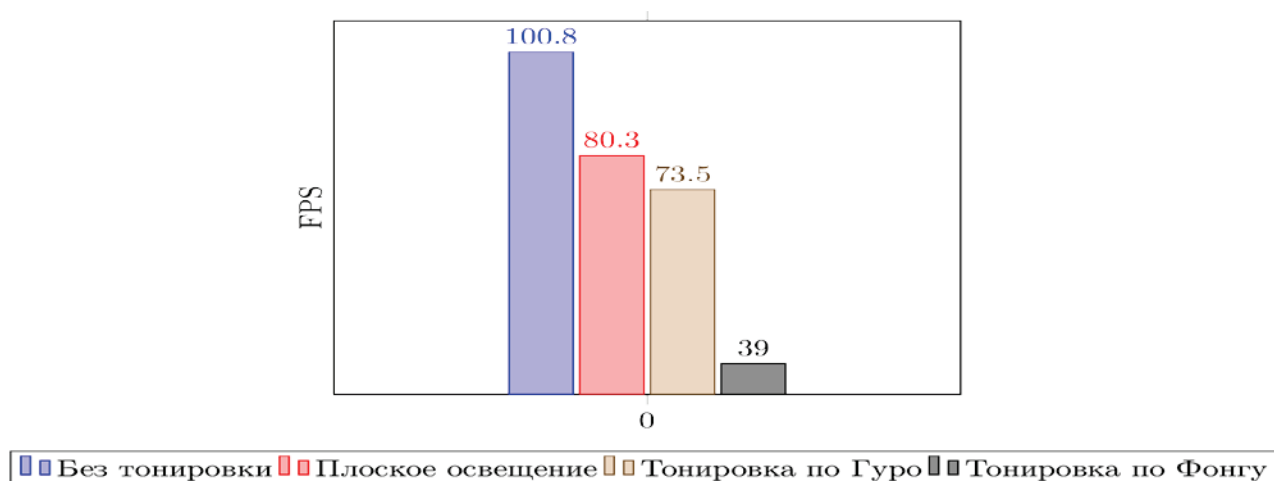


Рис. 6. Скорость тонировки

Для высокополигональной модели тонировки по Фонгу и по Гуро не будут визуально сильно различаться, хотя тонировка по Фонгу медленнее почти в два раза. Плоская тонировка не даёт большого прироста качества по сравнению с отсутствием тонировки, но при этом даёт большое падение производительности. В итоге, оптимальными для приложения реального времени оказались два режима — отсутствие освещения как такового и тонировка по Гуро, дающая хорошее качество тонировки.

4. Возможные пути улучшения

Существуют несколько возможных путей развития такого растеризатора. Во-первых, можно было предположить что последующие кадры не будут сильно отличаться от предыдущих и кэшировать их. Однако, это вряд ли дало бы большой прирост к производительности, поскольку камера пользователя обычно движется постоянно.

Другим путём оптимизации программы является распараллеливание визуализации. В частности, можно было бы создать пул потоков и распределять задачи визуализации объектов или даже треугольников между ними. Однако, написание программы для визуализации с параллельными вычислениями выходило за рамки курсовой работы, так как сопряжено с большими сложностями по оптимизации кэширования и достижения синхронизованности потоков во время визуализации (в частности, проверки Z-буфера), если учитывать что результат должен давать выигрыш в скорости. Наконец, можно было бы оптимизировать структуру хранения сцены, например, реализовать некоторый вид KD-дерева. В частности, рассматривалось BSP-дерево [4], однако, оно было сочтено неоптимальным по причинам необходимости полной регенерации при изменении положения объектов.

Заключение

В ходе работы над проектом была разработана программа для быстрой визуализации трёхмерных сложных сцен с освещением и текстурированием, описаны применённые оптимизации, произведены исследования влияния некоторых из проведённых оптимизаций на скорость работы программы. Намечены пути дальнейшего развития разработанной программы.

Список литературы

1. E. Angel. Interactive Computer Graphics: A Top-down Approach Using OpenGL. Computer graphics. Pearson/Addison-Wesley, 2006.
2. F.D. Luna. Introduction to 3D Game Programming With DirectX 10. Wordware Game and Graphics Library. Wordware Pub., 2008.
3. GaJel Guennebaud и др. Benoit Jacob. API Documentation for Eigen3. 2012. URL: <http://eigen.tuxfamily.org/dox/>. (дата обращения: 19.12.2012)
4. Henry Fuchs, Zvi M. Kedem и Bruce F. Naylor. «On visible surface generation by a priori tree structures». М.: SIGGRAPH Comput. Graph. 14.3 (июл. 1980), с. 124—133.
5. Wikipedia. Rasterization — Wikipedia, The Free Encyclopedia. 2012. URL: <http://en.wikipedia.org/w/index.php?title=Rasterisation&oldid=510412438>. (дата обращения: 18.12.2012)
6. Д.Ф. Роджерс. Алгоритмические основы машинной графики. Мир, 1989.