# The Ray-Trace Game

## Часть I. Общие сведения

**Предыстория**

Микропроцессорная техника развивается невероятными темпами. То, что вчера ещё считалось фантастикой, сегодня уже является частью нашей повседневной жизни.

Тем не менее, геймеры всего мира, затаив дыхание, который год ждут революции в игровой индустрии: когда наконец-то графика станет реалистичной, когда настанет эра raytracing'a?

Современные технологии отображения, используемые в играх, лишь имитируют освещение и тени, прозрачность и туман. И каждый шаг на пути к реалистичности сопровождается дюжиной новых усложнений процесса имитации.

Чтобы хоть немного приблизить чудесное будущее, было решено создать игру с простой графикой, но на основе метода трассировки лучей.

Про что игра? Про трассировку луча, естественно.

**Краткое описание**

На каждом уровне есть источник лазерного луча, приёмник, зеркала, "фишки", стены и другие объекты. Цель – поворачивая зеркала, необходимо "поразить" лучом все фишки (пораженная фишка исчезает с поля), после этого исчезает объект, препятствующий прохождению луча к приёмнику. Для завершения уровня необходимо направить луч в приёмник.

Сложности: лазер имеет ограниченный заряд (по времени); установка может выйти из строя из-за перегрева (когда отраженный луч возвращается в источник) и другие.

**Интерфейс и управление**

На основном экране изображена схема установки и параметры (заряд, температура). Можно включить дополнительный экран – "Микроскоп". На этом экране будет отображаться участок схемы методом трассировки лучей.

Перемещение игрового курсора производится кнопками (стрелками) вверх, вниз, вправо, влево на клавиатуре. Клавиша ctrl поворачивает зеркало под курсором по часовой стрелке, а клавиша shift – против.

Для более детального (чем на схеме) изучения установки можно воспользоваться “микроскопом”.

Клавиши delete / page down, home / end изменяют угол обзора, а клавиши insert / page up регулируют масштаб.

Клавиша F1 переключает режим отображения прозрачных тел. В этом режиме квантовые точки отображаются как сферические линзы.

Клавиша F2 переключает режим сглаживания (anti-aliasing). При включении режима изображение становится более реалистичным, но частота кадров уменьшается в 4 раза.

Оба режима можно переключать при помощи кнопок в правом нижнем углу панели.

Для своей работы микроскоп потребляет большое количество энергии. Поэтому, если Вы проводите эксперименты на портативном устройстве, то при отключении сетевого питания микроскоп автоматически выключится.

Для удобства работы на сокращённых клавиатурах можно использовать альтернативные клавиши управления:

- перемещение курсора – a / d, w / s
- поворот зеркала – q / e
- управление микроскопом – g / j, y / h, t / i

По умолчанию микроскоп следит за игровым курсором. Если щёлкнуть мышкой на интересующем фрагменте схемы, она отобразится в окне микроскопа. При этом “слежка” отключается, а исследуемая точка отмечается на схеме мигающей белой точкой.

Перемещение мыши, при зажатой правой кнопке, приводит к изменению угла обзора микроскопа. Вращение колёсика изменяет масштаб.

**Установка**

Игра не требует установки, можно запускать сразу после распаковки. Для работы не требуется наличия DirectX, OpenGL и других дополнительных библиотек.

**О технологии**

Главным желанием в процессе разработки было продемонстрировать технологию ray-tracing (которая используется при создании мультфильмов и спецэффектов в кинофильмах) и проверить, готовы ли современные процессоры к широкому её внедрению. Оказалось, что для несложных сцен (как в данной игре), процессорной мощности современных компьютеров вполне достаточно. Основной плюс технологии – реалистичное освещение, блики и тени, корректное отображение прозрачных (линз) и зеркальных тел.

## Часть II. О разработке

**Идея**

С детства мечтал стать разработчиком игр. Как только освоил основы бейсика на ZX-Spectrum, сразу же принялся писать что-то типа “сапёра”. Шли годы, я постигал секреты программирования, иногда писал несложные игры, но мечта стать профессиональным разработчиком и написать ”самую лучшую игру” отодвигалась всё дальше и дальше в будущее.

Теперь я Java Developer, и про мечту свою уже успел позабыть… Осталась лишь страсть к графическим эффектам.

Этой весной передо мной встала следующая задача: отобразить в апплете трёхмерный текстурированный куб. Эту задачу можно было решить с использованием OpenGL, но решение получалось неуклюжим. Попытка использовать упрощённый ray-tracing на удивление оказалась более чем успешной!

И если уж даже не самым мощным современным компьютерам посильно с использованием Java обсчитывать путём трассировки лучей, то на C++ точно можно добиться отображения несложных сцен!

Узнав о соревновании – решил попробовать свои силы и вспомнить старое. Тем интереснее было, что оставалось до сдачи всего 20 дней.

Во времена ZX-Spectrum я переиграл во множество разных игр. И хотя по уровню графики они значительно уступают современным, зато очень многие из них были действительно интересными и захватывающими. Одну из запомнившихся игр я решил использовать как образец и исполнить её на новый лад…

**Рабочее окружение и сроки**

Разработка была начата 9 июня, велась, в основном, по вечерам после работы. На разработку первой версии “микроскопа” ушло 5 вечеров.

Ещё несколько дней в июле и августе ушло на придание игре завершенного вида (отображение в микроскопе источника и приёмника, режим сглаживания).

В качестве рабочих инструментов использованы Intel C++ Compiler, FAR Manager, Paint.Net, Blender, Adobe Audition, SmartSVN.

Для хранения исходных кодов был создан проект на code.google.com.

Использованы звуковые материалы с сайта freesound.org и композиции группы Pink Floyd (что придаёт особый колорит).

**Начало проекта**

Как бы невероятно это не звучало, но самой главной трудностью на пути создания игры было создание рабочей среды! (Конечно, те, кто постоянно что-то разрабатывают в C++ могут лишь усмехнуться, но я-то последние несколько лет программирую на Java).

Итак, скачав компилятор Intel C++, я обнаружил, что этого совершенно недостаточно для написания программы. Самый простой и нелогичный (связывающий по рукам и ногам) путь решение этой проблемы – установка Microsoft Visual Studio. На домашнем компьютере всё прошло гладко, но на рабочем установка оказалась невозможной. Потратив некоторое время, удалось выяснить, какие файлы являются неотъемлемой частью среды для сборки и создать миниатюрный переносной “комплект разработчика”.

В дальнейшем выяснилось, что вместо Visual Studio можно установить Microsoft Windows SDK (но обязательно надо выбрать пункт ”компиляторы”).

Когда настало время для распараллеливания, встал выбор: OpenMP который я недавно освоил или Intel TBB, которого я ещё не знал.

Аргументы в пользу OpenMP:

- простота использования
- поддержка компиляторами

Решающий аргумент против: не слишком очевидный переход между исходным и генерируемым кодом.

Intel TBB напротив – прозрачен с точки зрения языковых конструкций.

Единственным препятствием к использованию этой библиотеки оказалось то, что с точки зрения архитектуры, Intel TBB должен подключаться динамически. При этом он не является частью поставки Windows, не имеет самодостаточных пакетов для установки, и зависит от MS VC Redistributable библиотек.

Поиск в интернете привёл на форумы ISN, где была опубликована неофициальная инструкция по сборке статической библиотеки. Сразу собрать библиотеку не удалось – в исходном коде содержались опечатки. Тем не менее – вскоре результат был достигнут.

**Первые трудности**

Поскольку было решено отказаться от использования «внешних» библиотек, то некоторые трудности появились ещё в период разработки графического интерфейса.

В основном меню игры, в схематическом отображении поля, в таблице рекордов, на экране "о лаборатории" используются различные графические эффекты (полупрозрачность, "частицы"). Замер производительности показал, что на обработку элементов меню уходило огромное количество процессорного времени. Первая оптимизация позволила сузить область применения эффекта, что незамедлительно дало четырехкратный прирост производительности.

Вторая оптимизация была хитрее. Анализ генерируемого компилятором кода показал, что выполнение простого действия — покомпонентного умножения цвета пикселя на число — влечёт за собой ненужные операции знакового расширения числа и знакового сужения. Это происходило из-за того, что промежуточный результат вычислений имел тип «байт».

Небольшое изменение алгоритма позволило отказаться от таких

преобразований и вместо 3-х операций умножения (по одной на каждую компоненту цвета) использовать две. В результате производительность повысилась ещё в несколько раз.

Пример кода, выполняющего смешение двух цветов:

```
DWORD vF0F = ((((v1 & 0xFF00FF) * zz) + ((v3 & 0xFF00FF) * z)) >> 8) & 0xFF00FF;
DWORD v0F0 = ((((v1 & 0xFF00) * zz) + ((v3 & 0xFF00) * z)) >> 8) & 0xFF00;
DWORD v = vF0F | v0F0;
```

**Многопоточность**

Трассировка лучей – ресурсоёмкая задача, которая получает огромный выигрыш при использовании многоядерных систем. Как уже было сказано ранее, для реализации многопоточной обработки используется статическая библиотека Intel TBB.

Первая попытка использования оказалась неудачной. Анализ показал, что для корректной работы требуется создание экземпляра task_scheduler_init в каждом потоке.

Для синхронизации данных основного цикла игры и главного потока визуализации были задействованы неделимые (atomic) операции. Блоки synchronized языка Java оказались не намного страшнее в C++:

Семафор в tbb:

```
while (mutex.compare_and_swap(true,false) != false) {Sleep(0);}
// DO SOMETHING
while (mutex.compare_and_swap(false,true) != true) {Sleep(0);}
```

Параллельные потоки занимаются визуализацией строк. Память под строчки результатов выделена с выравниванием на строки кэша (cache aligned). Таким образом, выполнение потоков не сопровождается накладными расходами на поддержание когерентности кэша.

Для уменьшения времени "холостого хода" (когда нет новых задач, и некоторые потоки уже закончили работу) использована особенность "микроскопа". Микроскоп имеет круглое поле отображения. То есть строки имеют разную длину. Если раздавать задачи потокам в порядке уменьшения длины строки (и, в среднем, объёма вычислений), то разброс времени окончания вычислений заметно сокращается.

Для поиска объектов, пересекающихся с лучом отображения, использована особенность расположения объектов. Объекты располагаются в одномерном слое смежных кубов, что даёт хорошее разбиение пространства даже без построения дополнительных структур (kd-tree, …).

**Нестандартный подход**

В июле в ISN была опубликована статья, в которой рассказывалось о том, что использование PGO (profile guided optimization) позволяет в большинстве случаев добиться повышения производительности на 10%. Испытания показали, что использование этой технологии напрямую даёт желаемый результат не всегда.

Дело в том, что разные режимы отображения в игре задействуют разные участки кода. Это сбивало оптимизатор с толку. Было решено сделать следующее:

1. заменить условия, проверяющие опции отображения, на условную компиляцию
2. перед началом компиляции создавать копии файла с классом
3. подключать исходный код этих классов с разными определениями

Таким образом, исходный код одного класса порождал четыре класса (то есть умножался). В зависимости от опций отображения выбирался класс, осуществляющий отображение. При таком подходе PGO собирал отдельную статистику по методам разных классов, что положительно сказалось на результатах.

**Интернационализация, ресурсы и сжатие**

При разработке игры изначально закладывалась возможность локализации. Все текстовые строки и сообщения были вынесены в текстовый файл.

Изображения, как и файл локализации, расположены рядом с исполнимым файлом. Таким образом, создание игры с новым языком может быть осуществлено очень быстро, без необходимости пересборки программы. Для этого не потребуется специальных навыков, только умение пользоваться графическим редактором.

Изначально изображения хранились в виде bmp файлов. Однако когда объём изображений в 15 раз превысил размер программы, стало ясно, что надо ситуацию менять. Самый простой путь – это использовать готовую открытую библиотеку сжатия изображений. Наиболее подходящий кандидат для этого – png.

Библиотека png зависит от библиотеки zlib. Собрав воедино необходимые для компиляции исходные файлы, удалось скомпилировать статическую библиотеку способную распаковывать некоторое подмножество png файлов.

На удивление, размер библиотеки получится довольно небольшим.

Формат png осуществляет сжатие без потерь. Тем не менее, существует ряд параметров, влияющий на размер результирующего файла. Преобразования bmp файлов и для автоматического подбора параметров сжатия был использован набор утилит pngUtils.

Как уже было сказано ранее, формат png основывается на библиотеке сжатия zlib, то есть на сочетании алгоритма LZ77 и алгоритма Хаффмана. Как известно, большинство реализаций этой библиотеки выдают неоптимальный результат. Для дальнейшего уменьшения размеров файлов была использована утилита AdvanceCOMP, позволяющая, в некоторых случаях, уменьшить размер файла на 10%.

Комплект поставки игры состоит из исполнимого файла и файлов-ресурсов. Этот комплект сжимается при помощи 7zip в самораспаковывающийся архив. Дальше этот архив (фактически – исполнимый файл) обрабатывается программой UPX, для сжатия кода распаковки 7zip.

**Что дальше**

Дальнейшее развитие игры предполагает добавление новых уровней, опций визуализации и графических эффектов. Поскольку нет завязок на графические библиотеки, то, возможно, игра будет переписана под SDL и станет в кросс-платформенной.