2. Сегодня мы будем говорить о компиляторах, а точнее о том,
что происходит межу двумя понятными любому программисту этапами.

3. Чтобы было не скучно, начнем издалека и с чего нибудь доброго и светлого. Например, с Квейка-третьего (хотя по правде, первый мне нравится больше).

Любой кадр, отображаемый на экране надо предварительно нарисовать, или, говоря терминами CG, отрендерить. Рендеринг включает уйму операций, среди которых заметное место занимает работа с векторами в целом и с нормалями в частности.

4. Нам поможет… внезапно мистер Пинхед! Ибо вся его суровая брутальность на деле всего-лишь модель нормалей к поверхности… Даже меш нарисовать не забыл… ходячий экспонат, в общем.

5. В самом деле, между «чайником» Пинхеда и чайником Юта есть гораздо больше общего, чем кажется на первый взгляд.

6. Так вот, для правильной работы с векторами нормалей, их часто приходится приводить к единичному размеру, то есть нормализовать. Под часто я понимаю ЧАСТО, очень часто, миллионы раз за кадр.

Сама по себе операция сравнительно простая, вот только операций этих очень много.

7. Во времена, когда дискеты были большими, а мониторы маленькими, производительность математического сопроцессора сильно уступала по скорости работе с целочисленными операциями.

В то же время, при описании непрерывных процессов одними целыми обойтись сложно. Поэтому в usenet часто делились способами организации арифметики с фиксированной точкой, которая реализуется целиком в целых числах.

Как бы то ни было. Одна светлая голова (до сих пор неизвестно, кто именно; раньше приписывали Кармаку, но корни уходят аж в SGI) придумала хитрый способ, который позволяет выполнять нормализацию существенно быстрее, чем это делает сопр.

Как именно, это делается…

8. Сама операция нормализации банальна — надо взять вектор и покомпонентно поделить его на свою длину. Длина вектора определяется с помощью Евклидовой нормы — обобщения понятия расстояния в Евклидовом пространстве.

Хитрость в том, что выгодно свести операцию нормализации к функции обратного корня.

9. А вот обратный корень оказывается можно быстро вычислить с помощью особой уличной магии, которая опирается на особенности представления чисел с плавающей точкой в формате стандарта IEEE754.

10. Означенный стандарт представляет числа в виде битовой последовательности, поделенной на три части: красная часть хранит мантиссу, зеленая — экспоненту. Наконец, старший бит последовательности хранит знак.

Фокус в том, что взятие обратного корня можно выполнить с помощью серии целочисленных

операций.

11-16. Итак, алгоритм

17. Ничего непонятно, ну да ладно. Посмотрим на реализацию алгоритма в коде Quake 3. Комментарии автора сохранены, я всего лишь добавил переносов… и звездочку влепил.

18-23. Если присмотреться, то можно понять, как оно работает.

Основная идея реализации в том, что с данными одного типа мы работаем сквозь призму другого типа. Похожий, хотя и отличающийся прием используется в библиотеке сокетов Беркли для представления адресов.

24. Сокеты Беркли являются стандартом описания сетевого взаимодействия в мире Posix. Для передачи параметров привязки сокета и задания сетевого адреса используется прием, известный как «Type Punning» или «каламбур типизации».

Дело в том, что сетевых протоколов существует много: IPv4, IPv6, IPX, X25 и у каждого своя система адресации. Задание параметров происходит через один и тот же интерфейс, который должен это учитывать.

Поэтому осуществляется своеобразный «полиморфизм структур» адресов, когда прототип функции задает обобщенную структуру, а клиент передает фактически тип. Оба варианта имеют одинаковый заголовок и представление в памяти.

Однако, чтобы *гарантировать* корректную работу, приходится приплясывать. Заглянем в файл sys/socket.h и…

25. Если читать это все вслух вероятно можно вызвать … поэтому побережемся и перейдем к следующему слайду.

26. Здесь я развернул все макросы и выкинул лишние детали. Объявляется transparent union и набор структур адресов, помеченных тэгами, понятными компилятору.

Transparent union — это такой хитрый системный костыль, часто использующийся в коде библиотек.

27. Он позволяет делать своеобразную перегрузку функций, которую компилятор проглотит без возмущения.

Вообще, если делают такие камлания, то наверное это не просто так. Разработчики ОС и библиотек не дураки. А вот что может случиться, если программист считает себя умнее компилятора.

28. Наслушавшись подобных хакерских баек, юный программист кидается к клавиатуре и пишет что-то вроде:

```
float invert(float value) {
      uint32_t* const raw = (uint32_t*) &value;
      *raw ^= (1 << 31);  // меняем знак

      return * (float*) raw;
}
```

Будучи опытным камикадзе, программист даже убедился, что у него на машине все компилируется как надо. И, довольный собой, толкает код в продакшн, применив его во всех расчетах.

29. …После того, как летящие из глаз искры перестали заслонять обзор, программист, потирая ушибленные места, лезет в дебаггер на продуктовой машине и видит следующее:

```
fld     dword ptr [ebp+8]
sub     dword ptr [ebp+8], 2147483648
```

Почему-то компилятор поменял местами операции и вообще делает какой-то бред! Как так?

Отойдя от шока, программист спрашивает более опытного коллегу, что же все таки случилось и «почему у меня работало, в соседней библиотеке все работает, а тут взорвалось?!»

На что ему показывают скрипт сборки «работающей» библиотеки, где красуется параметр:

30. `-fno-strict-aliasing`

К слову, более новые версии gcc такой проблемы не имеют; clang мне вообще не удалось так поймать (LLVM несколько по другому работает с кодом); компилятор Интел вообще свел всю функцию к трем командам: выкинул ненужные пролог с эпилогом, оставив только **xor** аргумента, **ldr** того же аргумента и **ret**. С учетом кеширования и конвейеризации обе операции с памятью скорее всего будут выполнены очень эффективно.

Разумеется все это не извиняет программиста и так писать все равно нельзя, поскольку такой код провоцирует неопределенное поведение.

…Как вы уже могли заметить, все описанные мной случаи, так или иначе, включают магию указателей. При этом, получается, что несколько указателей одновременно ссылаются на один и тот же участок памяти.

31. В computer science такая ситуация называется *алиасингом*.

Можно считать его великим злом или великим благом, тем не менее он существует.
И существует он не просто так. Разумеется, все это делается ради оптимизаций. Для того, чтобы пользователи могли просмотреть *больше* котиков в единицу времени.

32. Надо больше котиков! Котики любят оптимизации!

33. Итак, оптимизации. Так или иначе все оптимизации решают одну задачу — как получить больше, делая меньше.

34. Существует огромное количество различных оптимизаций, каждая из которых работает на своем уровне

35. Поскольку сегодня мы говорим про память, то и рассматривать будем оптимизации, которые влияют на работу с памятью.

Постараемся выяснить, как alias analysis влияет на исход той или иной оптимизации.

36-38. В качестве примера возьмем такой вот код. Можно заметить, что в теле цикла присутствует большое количество операций с памятью.

39. Разумно было бы сгруппировать обращения к памяти, сократив их число.

40. Вот как выглядит результат компиляции подопытного кода, с небольшими правками (пролог). Всю «простыню» можно разделить на функциональные части, поддающиеся пониманию.

48. Обращаем внимание, что мы читаем input[i] дважды в теле цикла… **edi** не менялся, в **ebx** *уже́* лежит нужное значение, почему мы его не используем?

50. Почему нельзя сделать вот так? Ведь между чтениями массив не меняется!! …или меняется?

51. Выглядит по-идиотски, но вполне может случиться на практике. Компилятор не знает что подается на вход, а потому рассчитывает на худший сценарий, защищая свою честь.

52. Раз все портит **max**, ну давайте внесем его внутрь функции…

55. Листинг программы, написанной с учетом max (тот же результат может дать указание спецификатора **__restrict** в аргументе).

58. Запись результата уехала за пределы цикла.

61. Листинг той же программы, но в IR коде LLVM

64-66. Программа состоит из трех базовых блоков.

68-70. Фи-узел используется для задания значения на текущей итерации. Если управление пришло…

76. Дело в том, что результат AA напрямую влияет на оптимизации.

77. В общем виде, анализатор дает ответ на вопрос, могут ли два указателя указывать на одну и ту же ячейку памяти или нет.

78. Простой пример… Вопрос к залу.

92. Наконец strict aliasing и TBAA. В отличие от рассмотренного выше, TBAA непосредственно зависит от компилируемого языка, поскольку использует его систему типов для вынесения решений.

Посему проще всего показать это на примере конкретных языков.