Город в бутылке — движок Raycasting всего на 256 байт

Автор оригинала: Frank Force

Привет любителям микро-кодирования. Вот вам кое-что невероятное: крошечный движок raycasting (метод “бросания лучей”) и генератор городов, который помещается в отдельный 256-байтовый HTML-файл.

В этой статье я поделюсь всеми секретами работы этой волшебной программы.

Город в бутылке

*<canvas style=width:99% id=c onclick=setInterval('for(c.width=w=99,++t,i=6e3;i–;c.getContext`2d`.fillRect(i%w,i/w|0,1-d*Z/w+s,1))for(a=i%w/50-1,s=b=1-i/4e3,X=t,Y=Z=d=1;++Z<w&(Y<6-(32<Z&27<X%w&&X/9^Z/8)*8%46||d|(s=(X&Y&Z)%3/Z,a=b=1,d=Z/w));Y-=b)X+=a',t=9)>* pic.twitter.com/N3WElPqtMY

Эта удивительная программа использует множество разных концепций в очень-очень малом объёме, и понимание её похоже на решение головоломки. В ней несколько основных частей: html-код, цикл обновления кадров, система рендеринга, движок raycasting и сам город.

Вы можете подумать, что для разгадки потребуется какая-то продвинутая математика, но на самом деле нет, код довольно прост и использует только базовую алгебру, не задействуются и какие-либо тригонометрические функции. Однако есть несколько хитростей, благодаря которым все линии сходятся в потрясающую картинку.

Весь код

Прежде, чем глубоко погрузиться в код, давайте бегло просмотрим его целиком. Это не просто фрагмент JavaScript, а целая действующая HTML-программа.

```
<canvas style=width:99% id=c onclick=setInterval('for(c.width=w=99,++t,i=6e3;i--;c.getContext`2d`.fillRect(i%w,i/w|0,1-d*Z/w+s,1))for(a=i%w/50-1,s=b=1-i/4e3,X=t,Y=Z=d=1;++Z<w&(Y<6-(32<Z&27<X%w&&X/9^Z/8)*8%46||d|(s=(X&Y&Z)%3/Z,a=b=1,d=Z/w));Y-=b)X+=a',t=9)>
```

Это невероятно плотный блок минимизированного кода размером 256 байт, поэтому потребуется некоторая работа, чтобы сделать его читабельным.

HTML-код

Прежде чем перейти к JavaScript, давайте посмотрим на HTML-часть кода. Вот только HTML сам по себе…

```
<canvas style=width:99% id=c onclick=setInterval('',t=9)>
```

Это просто элемент canvas с событием onclick. Я не поскупился и установил ширину CSS на 99%, хотя он прекрасно работает и без этого, так что есть дополнительное пространство для будущих ремиксов. Идентификатор canvas установлен на **c**, что даёт нам возможность получить к нему доступ из JavaScript.

Событие onclick запускает программу. Вызов setInterval — это фрагмент JavaScript, который создаёт цикл обновления. Время интервала составляет 9 миллисекунд, что немного быстрее, чем 60, но достаточно близко к этому значению, чтобы не оказывать значительного влияния. Переменная времени **t** здесь также инициализируется значением 9 для экономии места.

Существует небольшая ошибка, которая возникает, если кликнуть по canvas несколько раз; интервал также будет запускаться несколько раз, из-за чего всё замедлится. Это не такая уж большая проблема, но об этом следует знать. Существует несколько других способов создания HTML-части этого кода, каждый из которых имеет свои недостатки. Обычный блок сценария тоже работает нормально, просто для этого потребуется немного больше места.

Код JavaScript

Далее у нас есть 199-байтовая полезная нагрузка JavaScript, которая запускается при нажатии на canvas.

```
 for(c.width=w=99,++t,i=6e3;i--;c.getContext`2d`.fillRect(i%w,i/w|0,1-
d*Z/w+s,1))for(a=i%w/50-1,s=b=1-i/4e3,X=t,Y=Z=d=1;++Z<w&(Y<6-(32<Z&27<X%w&&
X/9^Z/8)*8%46||d|(s=(X&Y&Z)%3/Z,a=b=1,d=Z/w));Y-=b)X+=a
```

Разбиваем JavaScript

Финальное изображение с городом, текстурами и тенями

Первое, что мы сделаем, это разобьём этот код на строки, чтобы его было легче читать. JavaScript в основном игнорирует пробелы, так что мы можем переставить части кода и добавить немного дополнительного пространства. Точки с запятой обычно не требуются для завершения оператора, поэтому они используются только внутри структуры цикла for.

```
c.width = w = 99

++t

for (i = 6e3; i--;)

{

  a = i%w/50 - 1

  s = b = 1 - i/4e3

  X = t
```

```
  Y = Z = d = 1

  for(; ++Z<w &

    (Y < 6 - (32<Z & 27<X%w && X/9^Z/8)*8%46 ||

    d | (s = (X&Y&Z)%3/Z, a = b = 1, d = Z/w));)

  {

    X += a

    Y -= b

  }

  c.getContext`2d`.fillRect(i%w, i/w|0, 1 - d*Z/w + s, 1)

}
```

Идём по коду шаг за шагом

Давайте теперь пройдёмся по каждой строке кода…

```
c.width = w = 99
```

Сначала мы очищаем canvas, устанавливаем его ширину 99 пикселей и сохраняем значение 99 в w. Эта цифра будет использоваться повторно много раз. Высота canvas по умолчанию равна 150, что хорошо здесь подходит. Всё, что находится под нашим рисунком просто оставляем пустым.

```
++t
```

Для каждого следующего кадра мы должны увеличивать переменную времени на единицу, чтобы анимировать сцену.

```
for (i = 6e3; i--;)
```

Этот цикл будет повторяться с использованием переменной цикла **i** и в конечном итоге определять яркость каждого отдельного пикселя.

Для этого мы запустим луч из камеры, используя положение этого пикселя для управления углом луча. Затем, если луч наткнётся на что-то, она перенаправит луч на солнце, чтобы проверить, находится ли этот пиксель в тени. Звучит сложнее, чем есть на самом деле!

Получение вектора камеры

Сначала нам нужно получить представление луча камеры, запущенного из начала координат.

```
a = i % w / 50 - 1
```

Горизонтальная координата вектора камеры хранится в **a**. Мы можем вычислить её из

переменной **i**, сначала получив остаток деления **i** на ширину **w**, которая, как мы помним, равна 99. Затем мы делим это на 50, чтобы получить значение от 0 до 2, и вычитаем 1, чтобы привести его к значению между -1 и 1. К счастью, скобки не нужны, что помогает сэкономить место.

```
b = s = 1 - i / 4e3
```

Вертикальная координата вектора камеры хранится в **b**. Она рассчитывается аналогично a. Чтобы вычислить процент по вертикали корректно, нужно сначала разделить i на ширину, затем округлить в меньшую сторону, а затем разделить на высоту.

Однако, если мы примем, что есть почти незаметный наклон, можно упростить вычисления. Просто разделить **i** на половину количества пикселей и вычесть 1, чтобы попасть в диапазон между -1 и 1. Значение 4e3 было выбрано для смещения горизонта ниже центра. Вы можете поиграть с этими значениями, чтобы увидеть, как это повлияет на результат.

Также обратите внимание, что для **s** устанавливается то же значение, что и **b**, чтобы создать вертикальное линейное затухание фона, если луч ни с чем не столкнулся. Значение **s** в конечном итоге будет использоваться для управления затенением сцены.

Затухание фона хранится в s

Определение положения камеры

Чтобы сцена выглядела анимированной при движении вправо, в качестве начальной позиции X используется значение времени **t**.

```
X = t
```

Мы также должны определить компоненты Y и Z, а также **d**, который используется для создания тумана вдали. Для всего этого отлично подходит значение 1.

```
Y = Z = d = 1
```

Система Raycasting

Этот внутренний цикл — самая сложная часть всей программы, в которой шаги системы Raycasting выполняются до тех пор, пока что-то не встретится на пути, а затем луч отскакивает, чтобы проверить наличие теней.

```
for(; ++Z<w &
```

Часть проверки условия цикла for делает большую работу, поэтому для наглядности мы

разделим её на несколько строк. Первая часть просто прибавляет к Z по единице, пока значение не превысит **w**, что для нас будет означать повторное использование переменной **w**, равной 99. Переменные X и Y будут обновляться внутри цикла.

Проверка высоты здания

Этот код представляет форму города. Именно здесь мы создаём здания, переулки и красивую пляжную недвижимость. Это *очень плотный* фрагмент кода!

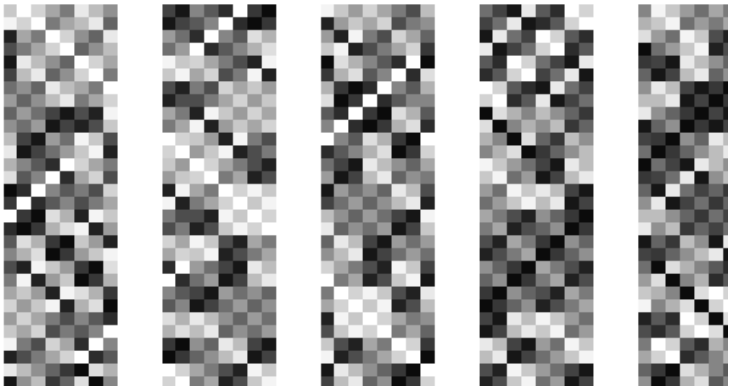
```
(Y < 6 - (32<Z & 27<X%w && X/9^Z/8)*8%46 ||
```

Чтобы проверить, находится ли луч внутри коллизии (в столкновении с объектом), мы проверяем, меньше ли значение Y, чем высота в данной позиции. Город формируется путём контроля высоты для каждого положения XZ

6 – эта часть просто перемещает результат высоты ниже центра и переворачивает все, чтобы земля оказалась внизу, как и должно быть.

Внутри круглых скобок происходит настоящее волшебство…

- Оставьте некоторое пространство между камерой и первым рядом зданий, убедившись, что Z сдвинулся как минимум на 32 единицы.
- Создайте переулки и побережье, проверив, что X mod w (константа 99) больше 27. Это позволит периодически оставлять пустые промежутки, такие как дороги, чтобы город делился на кварталы. В качестве дополнительного бонуса эта проверка всегда возвращает false при отрицательных значениях, создавая случайный океан.
- Сгенерируйте функцию шума для верхушек зданий, используя X/9^Z/8. Функция побитового исключающего «ИЛИ» используется здесь для получения интересного разброса значений, Таким образом здания будут иметь разную высоту.
- Деление используется для масштабирования, чтобы здания были 9 юнитов в ширину и 8 юнитов в глубину. Использование более крупных единиц приведёт к увеличению размеров зданий.
- Значение X/9 здесь также связано с шириной боковой улицы от 27 до 99, все эти числа кратны 9. Это предотвращает создание очень тонких зданий по краям.



Вид сверху вниз на здания, изображённые в оттенках серого

Результат всего, что находится в скобках, в итоге умножается на 8. Максимальной высотой будет остаток деления этого числа на 46. Эти значения были выбраны после экспериментов, чтобы здания получились разными по высоте. Так смотрится интереснее.

На этом можно было бы остановиться и не повторять цикл второй раз. Нарисовать то, что у нас есть, используя значение Z, чтобы затемнить здания в дали.

Значения расстояний, используемые для создания тумана

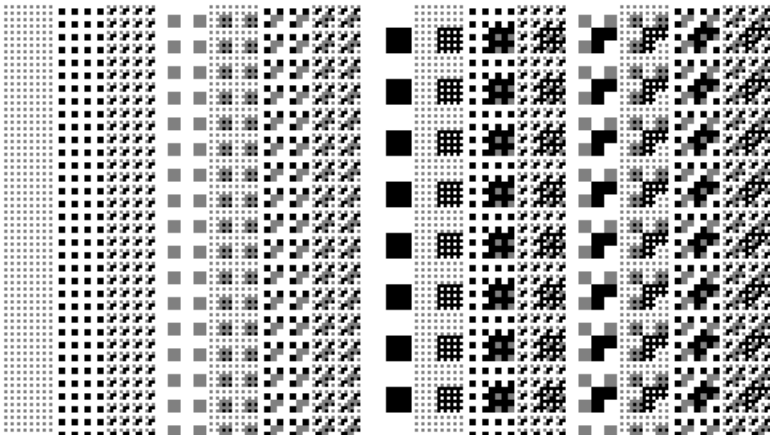
Создание тени и текстуры

Если этот код выполнен, значит, первый тест пройдён, и луч, должно быть, с чем-то столкнулся. Здесь мы получим текстуру объекта столкновения, отразим свет к солнцу, чтобы создать тень. Это немного сложнее, потому что на самом деле это два цикла в одном.

```
d | (s = (X&Y&Z)%3/Z, a = b = 1, d = Z/w));)
```

Первая часть **d |** нужна для того, чтобы проверить, запускаем ли мы луч из камеры или направляем луч на свет, чтобы проверить наличие тени. Мы уже установили **d** равным 1 до начала цикла, и в конце этой строки ему будет присвоено значение меньше 1, которое в сочетании с побитовым “или” будет оценено как false. Это позволяет циклу запуститься второй раз, перемещаясь к свету и проверяя наличие тени. Поэтому, если он окажется в тени при следующем столкновении, он выйдет из цикла “for” и отрисует пиксель.

Значение текстуры в оттенках серого сохраняется в **s** и генерируется с помощью оператора & с X, Y и Z и остаток их деления на 3. Это создаёт эффект, похожий на различные типы окон. Результат также делится на Z, чтобы затенить текстуры в отдалении.



Вид сбоку на текстуры зданий

Чтобы направить луч на источник света, для **a** и **b** установлено значение 1. Это отлично работает для направленного источника света.

Значение тумана сохраняется в **d** путём деления текущего значения Z на w (99), которое будет использоваться для освещения зданий на расстоянии. Это же значение **d** теперь

гарантированно будет меньше 1, а значит, мы проверяем наличие тени, как упоминалось ранее.

```
X += a
```

```
Y -= b
```

Каждый компонент обновляется для перемещения конечной точки луча. Часть X и Y управляется **a** и **b** соответственно. Часть Z всегда перемещается вперёд на 1, поскольку направленный свет также направлен в сторону камеры, и его никогда не нужно менять.

Рисование каждого пикселя

В конечном итоге каждый пиксель рисуется с использованием простого вычисления i для получения координат X и Y. Яркость контролируется путём уменьшения размера пикселя, что является очень экономичным способом создания изображений в оттенках серого по пикселю за раз.

```
c.getContext`2d`.fillRect(i%w, i/w|0, 1 - d*Z/w + s, 1)
```

Существует несколько значений, которые объединяются для управления окончательным значением оттенков серого. Значение 1 здесь будет черным пикселем, поэтому для создания изображения в оттенках серого мы будем производить вычитание из 1.

Без текстурного компонента

Значение тумана **d** умножается на текущее расстояние Z/w, и именно так создаются тени. Если луч находится не в тени, он должен пройти максимальное расстояние w, поэтому Z/w будет равно 1. И наоборот, если он находится в тени, то Z будет меньше **w**, в результате чего эта область станет темнее. Это на самом деле создаёт своего рода модель затемнения, потому что чем ближе объект, блокирующий свет, тем гуще тень.

Наконец, мы добавляем **s** к результату, который представляет собой текстуру зданий, которые мы рассчитали ранее. Вы увидите, как оба этих важных компонента объединяются, чтобы создать реалистичную сцену.

Конечный результат, где объединено всё вышеописанное

Хотите верьте, хотите нет, но это вся программа! Этот крошечный 256-байтовый движок рейкастинга и генератор городов демонстрирует, как многого можно достичь с помощью минимального кода.

Спасибо за внимание.