МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
БУРЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Институт математики и информатики
Кафедра прикладной математики

Курсовая работа

Рендеринг псевдотрёхмерного пространства методом бросания лучей

Выполнил : студент 4 курса группы 05230
Шорников Александр Евгеньевич
Научный руководитель: асс.
Брагин Александр Фёдорович

Улан-Удэ
2016

Оглавление

Стр.

Введение **3**

1 Технологии рендеринга **5**

2 Описание метода **7**

3 Реализация нетекстурированного движка **15**

Заключение **30**

Список литературы **31**

Приложение 1 **33**

ВВЕДЕНИЕ

Представление данных на мониторе компьютера в графическом виде впервые было реализовано в середине 50-х годов для больших ЭВМ, применявшихся в научных и военных исследованиях. С тех пор графический способ отображения данных стал неотъемлемой принадлежностью подавляющего числа компьютерных систем, в особенности персональных. Графический интерфейс пользователя сегодня является стандартом “де-факто” для программного обеспечения разных классов, начиная с операционных систем[10].

Существует специальная область информатики, изучающая методы и средства создания и обработки изображений с помощью программно-аппаратных вычислительных комплексов - компьютерная графика[2]. Она охватывает все виды и формы представления изображений, доступных для восприятия человеком либо на экране монитора, либо в виде копии на внешнем носителе (бумага, кинопленка, ткань и прочее). Без компьютерной графики невозможно представить себе не только компьютерный, но и обычный, вполне материальный мир. Визуализация данных находит применение в самых разных сферах человеческой деятельности. Для примера назовем медицину (компьютерная томография), научные исследования (визуализация строения вещества, векторных полей и других данных), моделирование тканей и одежды, опытно-конструкторские разработки.

Отдельным предметом считается трехмерная (3D) графика, изучающая приемы и методы построения объемных моделей объектов в виртуальном пространстве. Как правило, в ней сочетаются векторный и растровый способы формирования изображений.

Также, заметное место в компьютерной графике отведено развлечениям. Появилось даже такое понятие, как механизм графического представления данных (Graphics Engine). Рынок игровых программ имеет оборот в десятки

миллиардов долларов и часто инициализирует очередной этап совершенствования графики и анимации.

Хотя компьютерная графика служит всего лишь инструментом, ее структура и методы основаны на передовых достижениях фундаментальных и прикладных наук: математики, физики, химии, биологии, статистики, программирования и множества других. Это замечание справедливо как для программных, так и для аппаратных средств создания и обработки изображений на компьютере. Поэтому компьютерная графика является одной из наиболее бурно развивающихся отраслей информатики и во многих случаях выступает “локомотивом”, тянущим за собой всю компьютерную индустрию.

Задачи данной курсовой работы:
- проанализировать и обработать теоретические и экспериментальные данные по теме «Метод бросания луча»;
- анализ собранной информации;
- иллюстрация методов;
- разработка программы, реализующая данное семейство методов.

Объектом исследования данной курсовой работы является задача построения псевдотрёхмерной картинки.

Предметом исследования данной курсовой работы является изучение основных принципов построения псевдотрёхмерной картинки методом бросания луча.

Глава 1

Технологии рендеринга

Метод бросания лучей(англ. Raycasting, "Рейкастинг") - это технология получения изображения по модели с помощью компьютерной программы, позволяющая создавать 3D перспективу в 2D картах[15]. В те времена, когда компьютеры были намного медленнее, чем сейчас, и механизмы 3D невозможно было запустить в реальном времени, рейкастинг был единственным возможным решением. Рейкастинг может работать очень быстро, поскольку он предполагает только выполнение необходимых вычислений для каждой вертикальной линии на экране. Самая известная игра с применением рейкастинга - это, конечно же, компьютерная игра *Wolfenstein 3D*.

Рис. 1.1: Скриншот игры *Wolfenstein 3D*

Wolfenstein 3D engine — псевдотрёхмерный игровой движок, разработанный для игры *Wolfenstein 3D*, вышедшей 5 мая 1992 года. Движок разрабатывался преимущественно Джоном Кармаком, главным программистом компании id Software. Движок *Wolfenstein 3D engine* реализует VGA графику (рейтранслитинговая), звук (WAV и IMF), физику и управление. Написан на Си и ассемблере x86.

Возможности компьютеров с процессором Intel 80286, которые были тогда распространены, были крайне ограничены[11]. Для рендеринга изображения при помощи рейкастинга в игре *Wolfenstein 3D* движок игры был специальным образом оптимизирован для слабых вычислительных машин. В результате чего все стены в этой игре имеют одинаковую высоту, и представляют собой взаимно перпендикулярные ячейки 2D сети, как видно на рисунке 1.2:

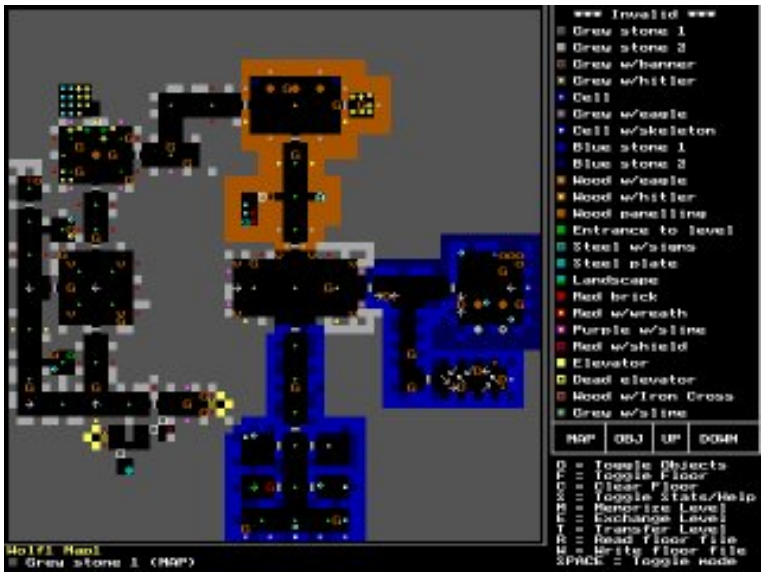

Рис. 1.2: Скриншот игры *Редактор уровней Wolfenstein 3D*

К сожалению, графические движки на основе технологии рейкастинга слишком слабы, что бы реализовать такие элементы, как лестницы или прыжки с разницей высот. Независимые графические объекты, свободно перемещающиеся по экрану (противники, внутреигровые объекты и прочее) представляют собой не трёхмерные объекты, а двухмерные картинки-спрайты. Более поздние игры на этой технологии, такие как *Doom* и *Duke Nukem 3D* были гораздо более продвинутыми, и позволяли создавать наклонные стены (поверхности), разницу высот, текстурированные полы и потолки, прозрачные стены и т.д., но в них были использованны разные технологии помимо рейкастинга, поэтому в рамках данной курсовой работы они не представляют интереса[12].

Глава 2

Описание метода

Основная идея рейкастинга состоит в следующем: карта представляет собой 2D-решетку с квадратными ячейками (двухмерный массив), где значением каждой ячейки может быть равно 0, что означает отсутствие стены, либо положительное число, означающее стену определенного цвета или текстуры.

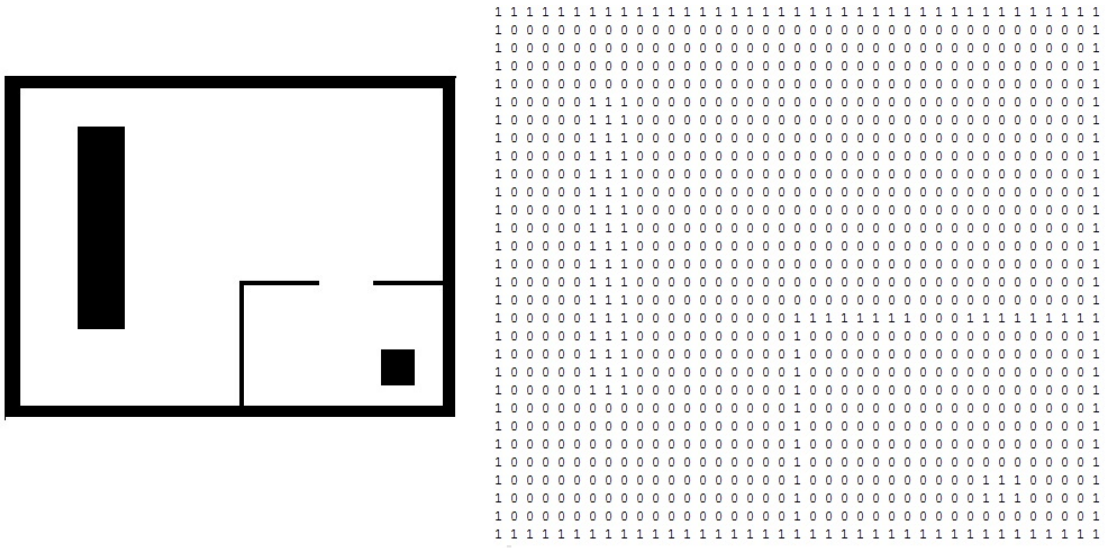

Рис. 2.1: Пример уровня карты

Каждому значению x на экране (для каждой вертикальной линии на экране) соответствует луч, который исходит из местонахождения игрока и направление которого зависит от двух критериев: направление взгляда игрока и координата x на экране. Затем данный луч начинается двигаться вперед по 2D карте до тех пор, пока не упрется в ячейку карты, которая является стеной. Если он пересечётся со стеной, то будет рассчитываться расстояние от этой точки соприкосновения со стеной до игрока, которое поможет определить, насколько высоко стену нужно будет переместить на экране: чем дальше расположена стена, тем меньше будет ее изображение на экране, и наоборот. Это - все 2D расчёты[4]. На рисунке 2.2 в ракурсе «сверху вниз» представлены

два луча (выделены красным), которые исходят от игрока зеленя точка) и упираются в синюю стену.

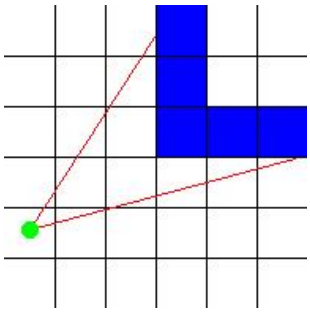

Рис. 2.2: Бросание луча

Чтобы обнаружить первую стену, которую луч встречает на своем пути, необходимо, чтобы он исходил из точки местоположения игрока, а затем нужно все время проверять, не находится ли луч внутри стены. Если он оказывается внутри стены (упирается в неё), цикл можно завершить, рассчитать расстояние и нарисовать стену правильной высоты. Если же луч не упирается в стену, необходимо продолжать вести его: добавить определённую величину к его положению, в направлении направления данного луча, и проверить, не находится ли луч в новом положении внутри стены. Проделывать данные действия необходимо до тех пор, пока луч не коснется стены.

Человек может сразу же увидеть, касается ли луч стены, но невозможно сразу же рассчитать, какой именно ячейки луч касается, используя всего одну формулу, поскольку компьютер может осуществить проверку ограниченного количества положений луча. Многие рейкастовые движки добавляют постоянную величину к лучу на каждом этапе, но в этом случае существует вероятность того, что луч может "промахнуться"и не коснуться стены. Например, положение этого красного луча проверялось в каждой красной точке.

Как вы можете видеть на рисунке 2.3, луч проходит прямо через синюю стену, но компьютеру не удалось это определить, поскольку он осуществлял только проверки красных точек. Чем большее количество положений прове-

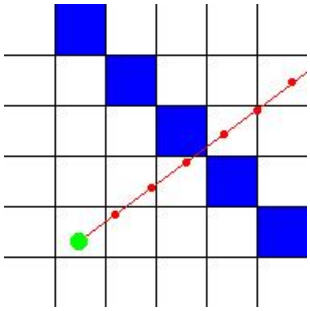

Рис. 2.3: Луч дошёл до стены

ряется, тем менее вероятность того, что компьютер не определит стену, но тем больше вычислений понадобится выполнить. На рисунке 2.4 показано, что расстояние между шагами было уменьшено вдвое, и компьютером было определено, что луч проходит через стену, хотя его положение и не совсем верно.

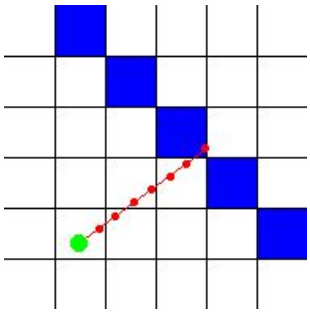

Рис. 2.4: Уменьшение числа проверок на столкновение

Для повышения точности данного метода необходимо бесконечно малое расстояние между точками, и, таким образом, необходимо будет производить бесконечное количество вычислений. Это неудобно, но, к счастью, существует более подходящая методика, предполагающая выполнение лишь нескольких вычислений и позволяющая определить каждую из стен. Идея состоит в следующем: проверять факт наличие луча на каждой из сторон стены. При ширине каждой ячейки, равной 1, каждая из сторон стены будет целым числом, а места между стенами будут равны некоему числу с цифрами после

запятой. В этом случае размер шага не является постоянной величиной, он зависит от расстояния до следующей стороны ячейки.

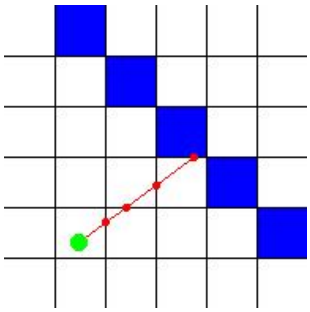

Рис. 2.5: Проверка на каждой из сторон стены

Как видно на рисунке 2.5, луч касается стены именно там, где нам это необходимо. В способе используется алгоритм, основанный на цифровом дифференциальном анализе. Цифровой Дифференциальный Анализ(Digital Differential Analysis, DDA) - скоростной алгоритм, обычно применяемый при использовании квадратной решётки, позволяющий определить, какие ячейки задевает луч (например, чтобы нарисовать линию на экране, состоящем из решётки квадратных пикселей)[5]. Таким образом, можно использовать этот метод, чтобы определить, какие ячейки решётки на нашем экране оказываются задеты лучом, и приостановить алгоритм, как только луч коснется ячейки, являющейся стеной.

Некоторые рейтрейсеры работают с Евклидовыми углами, которые представляют направление взгляда игрока и лучей, и позволяют задать Обзор (Поле Зрения) с помощью другого угла. Тем не менее, я обнаружил, что вместо этого намного проще работать с векторами и камерой: положение игрока всегда является вектором (с координатами x и y), но теперь можно также определить направление вектора: его направление определяется с помощью двух величин (координат x и y направления). Вектор направления можно визуализировать следующим образом: если нарисуем линию в направлении взгляда игрока, проходящую через точку нахождения игрока, то каждая точ-

ка данной линии будет являться суммой показателей положения игрока и будет кратна направлению вектора (показателям направления вектора). Длина вектора направления не имеет особого значения, важно лишь его направление. Умножение показателей x и y на одну и ту же величину изменяет длину, но сохраняет направление вектора[9].

Данный векторный метод также требует наличие дополнительного вектора, представляющего собой вектор плоскости камеры. В настоящем 3D движке также имеется плоскость камеры, и там она является настоящей 3D плоскостью, которую должны представлять два вектора (u и v). Рейкастинг рассчитан на 2D карты, потому в этом случае плоскость камеры не является плоскостью, это скорее линия, представленная одним вектором. Плоскость камеры всегда должна быть перпендикулярна вектору направления. Плоскость камеры представляет собой плоскость на компьютерном экране, в то время как вектор направления расположен перпендикулярно по отношению к этой плоскости и направлен внутрь экрана. Положение игрока представлено одной точкой, расположенной перед плоскостью камеры. Определённый луч определённых x-координат на экране является, в таком случае, лучом, который начинается в точке нахождения игрока и проходит через это положение или через плоскость камеры.

На рисунке 2.6 представлена такая 2D камера. Зеленая точка - это положение

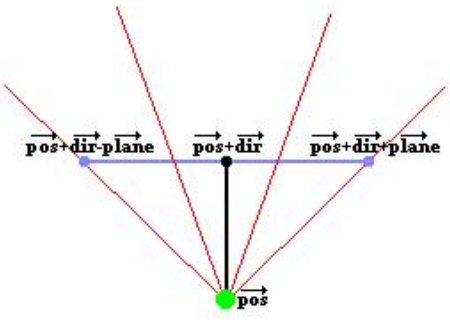

Рис. 2.6: Визуализация 2D-камеры

(вектор $\overrightarrow{pos}$). Чёрная линия, оканчивающаяся чёрной точкой, представляет

вектор направления (вектор $\overrightarrow{dir}$). Таким образом, положение чёрной точки - это вектор $\overrightarrow{pos} + \overrightarrow{dir}$. Синяя линия представляет полную плоскость камеры. Вектор, проходящий от чёрной точки к правой синей точке представляет вектор $\overrightarrow{plane}$. Таким образом, положение правой синей точки - $\overrightarrow{pos} + \overrightarrow{dir} + \overrightarrow{plane}$, а положение левой синей точки - $\overrightarrow{pos} + \overrightarrow{dir} - \overrightarrow{plane}$ (все это - векторное сложение).

Красные линии на изображении - это несколько лучей. Направление этих лучей легко можно рассчитать с помощью камеры: это - сумма вектора направления камеры и части вектора плоскости камеры. Например, третий красный луч на рисунке проходит сквозь правую часть плоскости камеры в точке, составляющей приблизительно $1/3$ от её длины. Таким образом, направление этого луча рассчитывается по формуле $\overrightarrow{dir} + \overrightarrow{plane} \cdot 1/3$. Направление этого луча - это вектор $\overrightarrow{rayDir}$, а компоненты x и y данного вектора далее будут использоваться в алгоритме цифрового дифференциального анализа.

Две другие лини являются левой и правой границей экрана, а угол между этими двумя линиями называется углом обзора. Угол обзора определяется по соотношению длины вектора направления и длины плоскости. Вот несколько примеров различных углов обзора:

Если вектор направления и вектор плоскости камеры имеют одинаковую длину, угол обзора будет равен $90°$:

Если вектор направления намного длиннее вектора плоскости камеры, значение угла обзора будет намного меньше $90°$, и поле обзора будет очень маленьким, игрок будет видеть все более детально, но с меньшей глубиной, что напоминает приближение камеры для получения изображения крупным планом.

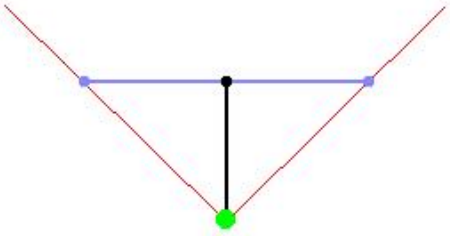

Рис. 2.7: Угол обзора 90°

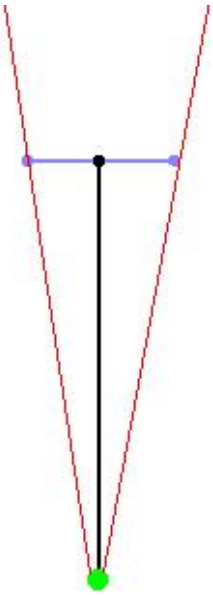

Рис. 2.8: Угол обзора меньше 90°

Если вектор направления короче вектора плоскости камеры, значение угла обзора будет превышать 90° (180° - максимальная величина, если вектор направления равен 0). В этом случае игрок будет иметь гораздо более широкое поле обзора, как при отдалении камеры.

При вращении игрока камера также должна вращаться(рис. 2.10), следовательно, и вектор направления, и вектор плоскости камеры также должны поворачиваться вместе с ними. Далее все остальные лучи будут вращаться автоматически.

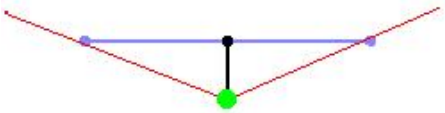

Рис. 2.9: Угол обзора больше 90°

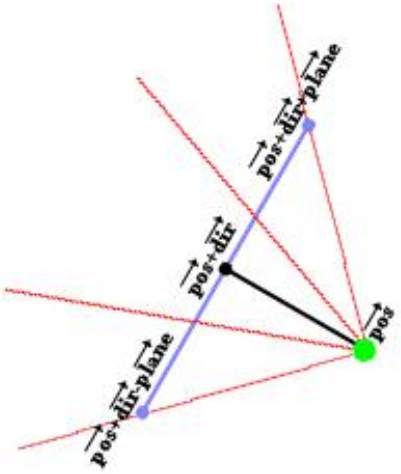

Рис. 2.10: Поворот 2D камеры

Чтобы повернуть вектор, необходимо рассчитать его по следующему шаблону вращения:

$$\begin{vmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{vmatrix}$$

К сожалению, нельзя использованию плоскость камеры, которая не перпендикулярна направлению, поскольку в результате мы получите «косой», искаженный мир.

Глава 3

Реализация нетекстурированного движка

Начнем с основ, то есть с нетекстурированного рейкастера. Данный пример также включает в себя счётчик fps (число кадров в секунду) и ключи ввода с обнаружением столкновений для движения и вращения.

Для реализации данного движка я использовал язык C++ и графические библиотеки SDL. Работоспособность была проверена в среде разработки *Qt* 5.7 с компилятором *gcc* под операционными системами *Linux Mint* 18 и *Windows* 7[1].

Как было сказанно в главе 2, карта мира представляет собой двумерный массив, где значение каждого элемента массива является квадратом мира. Если значение ячейки равно 0, то квадрат оказывается пустым, и через него можно пройти. Если же значение больше 0, квадрат представляет собой стену определённого цвета или текстуры. Карта, представленная здесь, очень мала, всего 24x24 квадрата, и непосредственно определяется кодом. В реальной игре, такой, как *Wolfenstein 3D*, будет использоваться карта большего размера, которая вместо этого будет загружаться из специального файла. Все нули в сетке являются пустыми пространствами, потому, в исходном варианте, мы увидим очень большую комнату, обнесённую по периметру стеной (значения, равные 1), и маленькую комнатку внутри нее (значения, равные 2), несколько столбов/колонн (значения, равные 3), и коридор с комнатой (значения, равные 4).

```
#define mapWidth 24
#define mapHeight 24

int worldMap[mapWidth][mapHeight]= \\карта
```

```
{
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,2,2,2,2,0,0,0,0,3,0,3,0,3,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,3,0,0,0,3,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,2,0,2,2,0,0,0,0,3,0,3,0,3,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,4,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,0,0,0,5,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,4,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
```

Несколько первых переменных заявлены как $posX$ и $posY$, и представляют позиционный вектор игрока (вектор его положения), в то время как переменные $dirX$ и $dirY$ задают направление игрока, а переменные $planeX$ и $planeY$ определяют положение камеры игрока. Необходимо убедиться в том, что плоскость камеры расположена перпендикулярно направлению (при

этом их длину можно варьировать). Соотношение длины вектора направления и плоскости камеры определяет поле обзора, при этом вектор направления немного длиннее длины плоскости камеры, так что поле обзора будет менее 90° (точнее, поле обзора будет равно $2 \cdot \arctan(0.66/1.0) = 66°$), что идеально для шутеров от первого лица). В дальнейшем, при вращении с помощью входных ключей, значения векторов направления и плоскости будут меняться, но всегда будут оставаться перпендикулярными, и их длина также будет оставаться постоянной.

Переменные *time* и *oldTime* будут использоваться для сохранения времени текущего и предыдущего кадров. Разница во времени между этими двумя показателями может использоваться для определения того, как далеко можно продвинуться при нажатии одной клавиши (чтобы двигаться с постоянной скоростью, независимо от того, как много времени занимает расчёт кадров), а также для счётчика в шутерах от первого лица.

```
int main(int /*argc*/, char */*argv*/[]) {
  double posX = 22, posY = 12;  //начальная позиция x и y
  double dirX = -1, dirY = 0; //начальный вектор направления
  double planeX = 0, planeY = 0.66; //ключи 2D камеры

  double time = 0; //время текущего кадра
  double oldTime = 0; //время предыдущего кадра
```

Оставшаяся часть майновской функции начинается здесь. Сначала экран создаётся с учётом оптимального разрешения. Если поставить большое разрешение, например, 1920*1080, эффекты будут применяться довольно медленно, не потому, что алгоритм рейкастинга не обладает достаточной скоростью, а по той простой причине, что загрузка всего экрана с центрального процессора на видеокарту идет достаточно медленно(вычисления производятся в ЦП а не в видеокарте).

```
screen(512, 384, 0, "Raycaster");
```

После настройки экрана начинается сам цикл игры. Во время этого цикла «рисуется» весь кадр, и идет постоянное считывание водимой информации.

```
while(!done()) {
```

Здесь и начинается сам рейкастинг. Цикл рейкастинга рассчитан на цикл, проходящий через каждый x, потому расчёт производится не для каждого пикселя экрана, а только для каждой вертикальной полосы, что, по сути, совсем не много. Чтобы запустить цикл рейкастинга, необходимо задать и рассчитать некоторые переменные:

Положение луча ($rayPos$) изначально устанавливается на основании положения игрока ($posX, posY$). $cameraX$ - это координата на векторе плоскости камеры, которую представляет текущая -координата экрана таким образом, что правая сторона экрана получает координату 1, центр - координату 0, а левая сторона - снова координату 1. В результате этого направление луча может рассчитываться, как было описано ранее, как сумма вектора направления и части вектора плоскости. Это должно быть выполнено для координат и вектора (поскольку сложение двух векторов означает сложение их и координат)[7].

```
    for(int x = 0; x < w; x++) {
//вычисление положение и направление луча
      double cameraX = 2 * x / double(w) - 1;
      //x-координата в пространстве камеры
      double rayPosX = posX;
      double rayPosY = posY;
      double rayDirX = dirX + planeX * cameraX;
      double rayDirY = dirY + planeY * cameraX;
```

В следующем отрывке кода, указывается и рассчитывается переменные для алгоритма цифрового дифференциального анализа. $mapX$ and $mapY$ представляют текущую ячейку массива карты, в которой сейчас находится луч. Само положение луча является числом с плавающей точкой, и содержит как информацию о том, в какой ячейке карты мы сейчас находимся, так и о том, в какой ячейке мы находились, но $mapX$ и $mapY$ являются всего лишь координатами этой ячейки.

Изначально, $sideDistX$ и $sideDistY$ - это расстояние, которое нужно преодолеть лучу, чтобы пройти от его начальной позиции до первых координат и необходимой стороны. В дальнейшем это значение в коде немного поменяется. $deltaDistX$ и $deltaDistY$ - это расстояния, которое нужно преодолеть лучу, чтобы пройти от первой до следующей стороны , или от первой до следующей стороны . Рисунок 3.1 показывает исходные $sideDistX$, $sideDistY$ и $deltaDistX$, $deltaDistY$

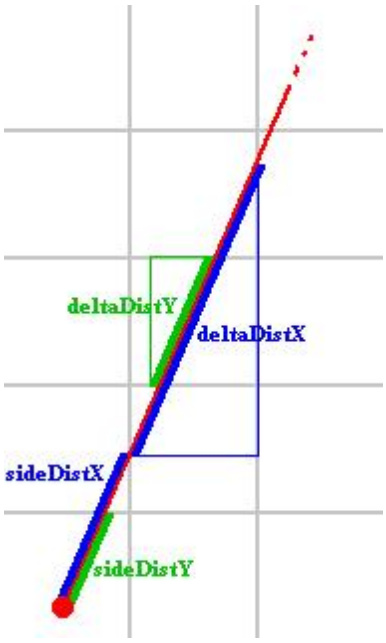

Рис. 3.1: Иллюстрация складывания луча

Переменная $perpWallDist$ будет использоваться в дальнейшем для расчета длины луча.

Алгоритм цифрового дифференциального анализа за один цикл всегда будет передвигаться ровно на одну ячейку в направлении оси или . В зависимости от направления луча, алгоритму может быть необходимо передвинуться в отрицательном или положительном направлении по оси или . Этот факт будет фиксироваться в *stepX* и *stepY*. Эти переменные всегда равны -1 либо +1.

Наконец, чтобы определить, может ли быть завершен данный цикл, используется факт достижения лучом одной из сторон какой-либо ячейки. Если луч достиг стороны по оси , сторона получает значение 0, если же луч достиг стороны по оси , она получает значение 1. Под сторонами по оси и подразумеваются линии решетки, которые являются границами между двумя ячейками[13].

```
//определение в каком квадрате карты мы находимся
int mapX = int(rayPosX);
int mapY = int(rayPosY);
//длина луча от текущей позиции до следующей координаты
double sideDistX;
double sideDistY;ч
 //длина луча от первой координаты к следующей
double deltaDistX = sqrt(1 + (rayDirY * rayDirY) /
(rayDirX * rayDirX));
double deltaDistY = sqrt(1 + (rayDirX * rayDirX) /
(rayDirY * rayDirY));
double perpWallDist;
//в каком направлении к шагу в направлении (+1 или -1)
int stepX;
int stepY;
int hit = 0; //был ли удар в стену?
int side;
```

Теперь, перед началом применения алгоритма цифрового дифференциального анализа, все же необходимо рассчитать первые $stepX$, $stepY$, а также исходные $sideDistX$ и $sideDistY$.

Если направление луча имеет отрицательный -компонент, $stepX$ = -1, если же направление имеет положительный -компонент, $stepX$ = +1. Если же компонент равен 0, значение $stepX$ неважно, поскольку в этом случае оно не будет использоваться. То же самое проделывается для -компонента.

Если же направление луча имеет отрицательный -компонент, $sideDistX$ представляет собой расстояние от исходной позиции луча до до первой стороны ячейки слева, а если направление луча имеет положительный -компонент - то до первой стороны справа соответственно. То же самое проделывается для -компоненты, но в этом случае берется первая сторон сверху и снизу от исходной позиции луча. Для этих значений используется целое значение $mapX$, из которого вычитается значение действительного положения, а затем в некоторых случаях прибавляется 1.0, в зависимости от того, где находится задействованная сторона луча (слева или справа, сверху или снизу). Затем получаем перпендикулярное расстояние до этой стороны, которое необходимо умножить на значение $deltaDistX$ или $deltaDistY$, чтобы получить значение действительного наклонного расстояния.

```
//вычисление шага и первоначального sideDist
if (rayDirX < 0){
  stepX = -1;
  sideDistX = (rayPosX - mapX) * deltaDistX;
}
else {
  stepX = 1;
  sideDistX = (mapX + 1.0 - rayPosX) * deltaDistX;
}
```

```
if (rayDirY < 0) {
  stepY = -1;
  sideDistY = (rayPosY - mapY) * deltaDistY;
}
else{
  stepY = 1;
  sideDistY = (mapY + 1.0 - rayPosY) * deltaDistY;
}
```

Теперь начинается действие действующего цифрового дифференциального анализа(DDA). Это - цикл, с помощью которого луч каждый раз поднимается на 1 ячейку, пока луч не достигнет стены. Каждый раз он делает скачок на расстояние, равное 1 ячейке, в направлении оси (при $stepX$) или оси (при $stepY$). За один раз луч «перепрыгивает» только одну ячейку. Если луч будет иметь направление , за один цикл луч будет передвигаться только в направлении оси , поскольку координата луча y в этом случае будет оставаться неизменной[3]. Если же луч немного наклонен в направлении оси , то при каждом его передвижении по оси луч будет передвигаться также на 1 ячейку в направлении оси . Если же луч имеет чистое направление , он вовсе не будет передвигаться по оси , и т.д.

Значения $sideDistX$ и $sideDistY$ увеличиваются на $deltaDistX$ при каждом передвижении в их направлении. Значения $mapX$ и $mapY$ также увеличиваются на $stepX$ и $stepY$ соответственно.

Когда луч достигнет стены, цикл окончится, и программа узнаёт, с какой стороной стены (или) соприкоснулся луч в переменной $side$, а также какой именно стены луч коснулся с $mapX$ и $mapY$. Программе не удастся точно узнать, где именно луч коснулся стены, но в данном случае в этом нет необходимости, потому что в этом случае программа не пользуется текстурированной стеной.

```
//выполнить цифровой дифференциальный анализ
      while (hit == 0){
        //перейти на следующий квардрат в направлении x или y
        if (sideDistX < sideDistY){
          sideDistX += deltaDistX;
          mapX += stepX;
          side = 0;
        }
        else{
          sideDistY += deltaDistY;
          mapY += stepY;
          side = 1;
        }
        //Проверка если луч врезался в стену
        if (worldMap[mapX][mapY] > 0)
            hit = 1;
      }
```

После выполнения цифрового дифференциального анализа необходимо рассчитать расстояние от луча до стены, чтобы можно было вычислить, насколько высокую стену необходимо изобразить после этого. При этом не используется наклонное расстояние: вместо этого нам необходимо значение расстояния, перпендикулярного плоскости камеры (проецируемое на вектор направления камеры), чтобы избежать эффекта рыбьего глаза. Этот эффект возникает в том случае, если используется реальное расстояние, так что все стены оказываются скругленными, что может вызвать головокружение игрока при вращении[**?**].

Важно отметить, что эта часть кода не является «правкой рыбьего глаза», поскольку данная поправка не нужна в представленном здесь случае использования рейкастинга. Эффекта рыбьего глаза удается избежать, просто рассчитав расстояние, как указано выше. Данное перпендикулярное расстояние

рассчитывается даже проще, чем реальное расстояние, так как нам не нужно знать точное места, в котором луч коснулся стены.

Во-первых, $(1 - stepX)/2$ равно 1, если $stepX$ = -1, и равно 0, если $stepX$ равна +1. Эти данные необходимы, поскольку нам нужно добавлять 1 к длине, когда $rayDirX < 0$, по той же причине, по которой в одном случае мы добавляли 1.0 к исходному значению $sideDistX$, а в другом случае - нет.

Затем расстояние рассчитывается следующим образом: если луч касается стороны , $mapX - rayPosX + (1 - stepX)/2)$ - это количество ячеек, которые луч пересек в направлении оси . если луч перпендикулярен оси , это значение уже является правильным. Но поскольку в большинстве случаев направление луча различается, его реальное перпендикулярное расстояние будет больше, поэтому необходимо последовательно разделить его на координату вектора $rayDir$.

Нечто похожее производится в случае, если луч касается стороны . Рассчитанное расстояние никогда не будет являться отрицательным числом, поскольку значение $mapX - rayPosX$ будет отрицательным только в том случае, если $rayDirX$ было отрицательным. Затем снова будет выполнено последовательное деление этих двух значений друг на друга[6].

```
//Расчёт расстояния проекции на направление камеры
//(с поправкой на эффект рыбьего глаза)
if (side == 0)
perpWallDist = (mapX - rayPosX + (1 - stepX) / 2) / rayDirX;
else
perpWallDist = (mapY - rayPosY + (1 - stepY) / 2) / rayDirY;
```

На основании рассчитанного расстояния вычисляется высота линии, кото-

рую нужно нарисовать на экране: это значение противоположно значению $perpWallDist$, затем его необходимо умножить на h(высоту экрана в пикселях), чтобы перевести значение в пиксели. Также можно умножить его на другое значение, например, на $2*h$, чтобы стена была выше или ниже. Значение h позволит создать стены, напоминающие кубы равной высоты, ширины и глубины, в то время как крупные значения позволят создавать более высокие "коробки"(в зависимости от разрешения экрана).

Затем на основании значения $lineHeight$ (которое является высотой вертикальной линии, которую необходимо нарисовать) рассчитываются начальная и конечная точки положения, где необходимо рисовать объект. Центр стены должен при этом совпадать с центром экрана, а если эти точки находятся за пределами экрана, их максимальные значения должны быть ограничены значениями 0 или $h - 1$.

```
//расчёт высоты линии для отрисовки на экране
int lineHeight = (int)(h / perpWallDist);
//вычисление самого низкого и самого высокого
//пикселя для заполнения текущей полосы
int drawStart = -lineHeight / 2 + h / 2;
if(drawStart < 0)drawStart = 0;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h)drawEnd = h - 1;
```

Наконец, в зависимости от номера стены, которой коснулся луч, выбирается цвет. Если луч коснулся стороны , выбирается более темный оттенок, что создает хороший эффект. Затем проводится вертикальная линия с помощью метода $verLine$. После того, как данная операция была проделана, по крайней мере, в отношении всех значений , цикл рейкастинга завершается.

```
//выбрать цвет стен
```

```
      ColorRGB color;
      switch(worldMap[mapX][mapY]) {
        case 1:  color = RGB_Red;  break; //красный
        case 2:  color = RGB_Green;  break; //зелёный
        case 3:  color = RGB_Blue;  break; //синий
        case 4:  color = RGB_White;  break; //белый
        default: color = RGB_Yellow; break; //жёлтый
      }
      //делаем стороны x и y разной яркости
      if (side == 1) {color = color / 2;}
      //рисуем полосу пикселей в виде вертикальной линии
      verLine(x, drawStart, drawEnd, color);
    }
```

После выполнения петли рейкастинга рассчитывается время текущей и предыдущей рамок, и показатель FPS (количество кадров в секунду) рассчитывается и печатается; изображение на экране перерисовывается так, что все (все стены, а также значения счётчика кадров в секунду) становятся видимыми. После этого бэкбуфер очищается с помощью метода $cls()$, так, что когда мы снова перерисуем стены, следующая рамка, пол и потолок снова станут чёрными, и не будут содержать пиксели из предыдущей рамки.

Модификаторы скорости используют $frameTime$ и постоянную величину для определения скорости движения и вращения ключей ввода. Благодаря использованию $frameTime$, мы можем убедиться, что скорость движения и вращения не зависит от скорости работы процессора.

```
//синхронизации для входа и счетчика FPS
    oldTime = time;
    time = getTicks();
    double frameTime = (time - oldTime) / 1000.0;
```

```
//время принятия кадра, сек
print(1.0 / frameTime); //FPS счётчик
redraw();
cls();
//модификаторы скорости
double moveSpeed = frameTime * 5.0;
//константная величина в квадратах / сек
double rotSpeed = frameTime * 3.0;
//константная величина в радианах / сек
```

Последней частью является обработка нажатий клавиш. При нажатии клавиши «вверх» игрок будет двигаться вперед: прибавьте $dirX$ к $posX$, и $dirY$ к $posY$. Предполагается, что $dirX$ и $dirY$ являются нормированными векторами (их длина равна 1). Имеется также простое встроенное условие обнаружения столкновений: если игрок находитесь внутри стены, то он не сможете двигаться. Данное устройство обнаружения столкновений можно улучшить, например, проверив, не будет ли врезаться в стену окружность вокруг игрока вместо всего одной точки. То же самое происходит, если нажата клавишу «вниз», но в этом случае направление вычитается[14].

Чтобы вращаться/поворачиваться, необходимо нажать кнопки «влево» или «вправо». И вектор направления, и вектор плоскости вращаются благодаря использованию формул умножения с матрицей вращения (с учетом угла *rotSpeed*).

```
 readKeys();
//двигаться вперед, если впереди нет стены
if (keyDown(SDLK_UP)) {
if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false)
posX += dirX * moveSpeed;
if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false)
```

```
posY += dirY * moveSpeed;
}//двигаться в обратном направлении, если сзади нет стены
if (keyDown(SDLK_DOWN)) {
if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false)
posX -= dirX * moveSpeed;
if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false)
posY -= dirY * moveSpeed;}
//повернуть направо
if (keyDown(SDLK_RIGHT)) {
//как направление и плоскость камеры должны быть повернуты
double oldDirX = dirX;
dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
double oldPlaneX = planeX;
planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
} //повернуть влево
if (keyDown(SDLK_LEFT)) {
//как направление и плоскость камеры должны быть повернуты
double oldDirX = dirX;
dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
double oldPlaneX = planeX;
planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
} } }
```

Так завершается код нетекстурированного рейкастера. Результат на рисунке 3.2 :

На рисунке 3.3 приводится пример того, что происходит, если плоскость камеры не перпендикулярна направлению вектора: в этом случае мир "на-

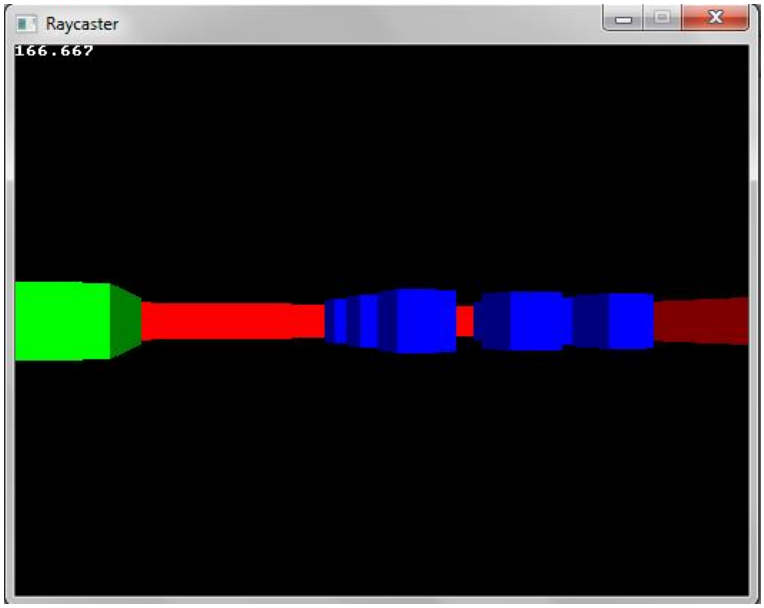

Рис. 3.2: Пример работы приложения

клоняется".

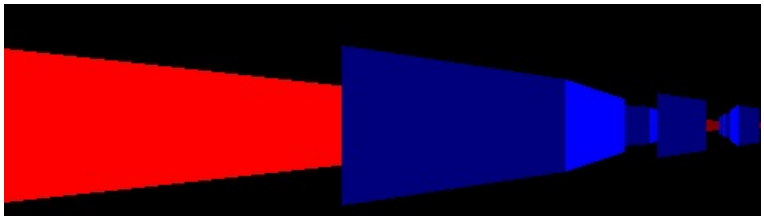

Рис. 3.3: Пример "неправльной"работы приложения

ЗАКЛЮЧЕНИЕ

Для рендеринга трёхмерной картинки существует много методов, однако, методов бросания лучей являются одними из самых простых и быстрых. С его помощью можно быстро и без особых усилий отрендерить псевдотрёхмерную картинку, получить результат быстро и без больших вычислительных можностей.

Во многим, технология рейкастинга и устарела. С её помощью нельзя сделать картинку "реалестичной пол и потолок всегда константной высоты, плохое и малоразмерное текстурирование и тд. Но несмотря на это, метод ещё не изжил себя полностью. с его помощью можно легко и непринуждённо переводить двухмерную карту в трёхмерный вид, например, для пожарных схем.

В моей курсовой работе был произведен тщательный анализ, исследование и реализация данных семейств методов. Результатом своей работы я считаю получение основы для программного обеспечения, реализованного на данном опыте. В дальнейшем я собираюсь и дальше развивать эту тему, и, в итоге получить программное обеспечение, по своей функциональности и возможностям не уступающее современным инди-разработкам.

Литература

1. Бьерн Страуструп. Язык программирования C++(3 издание). -СПб.: Невский Диалект, 2008. - 504 с.

2. Вельтмандер П.В. Машинная графика. Основные алгоритмы. Книга 2. – Новосибирск: НГУ, 1997. -197 с.

3. Котов Ю. В. Как рисует машина. — М.: Наука, 1988. — 224 с.

4. Ласло М. Вычислительная геометрия и компьютерная графика на C++. — М.: БИНОМ, 1997. — 304 с.

5. Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики - СПб.: БХВ-Петербург, 2003. - 554 с.

6. Павлидис Т. Алгоритмы машинной графики и обработки изображений: Пер. с англ. - М.: Радио и связь, 1986. – 400 с.

7. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989. — С. 50-54

8. Чириков С. В. Алгоритмы компьютерной графики (Методы растрирования кривых). Учебное пособие — СПб: СПбГИТМО(ТУ), 2001. — 120 с.

9. Joseph O'Rourke. Computational Geometry in C. — Cambridge University Press, 1998. — 362 с.

10. Kushner, David (2004-05-11). Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture. Random House.

11. Kent, Steven L. (2010-06-16). The Ultimate History of Video Games. Three Rivers Press.

12. Slaven, Andy (2002-07-01). Video Game Bible, 1985-2002. Trafford Publishing

13. Lode's Computer Graphics Tutorial. 2007. URL: http://lodev.org/cgtutor/index.html

14. Making a Basic 3D Engine in Java. 2009. URL: http://www.instructables.com/id/Making-a-Basic-3D-Engine-in-Java/

15. RAYCASTING - сделай себе немного DOOM'a. 2004. URL: http://zxdn.narod.ru/coding/ig5ray3d.htm

ПРИЛОЖЕНИЕ 1

Программный код приложения для реализации метода бросания лучей

```
#include <cmath>
#include <string>
#include <vector>
#include <iostream>
#include "quickcg.h"
using namespace QuickCG;
/*
g++ *.cpp -lSDL -O3 -W -Wall -ansi -pedantic
g++ *.cpp -lSDL
*/
#define mapWidth 24 //ширина карты
#define mapHeight 24 //высота карты
int worldMap[mapWidth][mapHeight]=
{
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,2,2,2,2,0,0,0,0,3,0,3,0,3,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,3,0,0,0,3,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,2,0,2,2,0,0,0,0,3,0,3,0,3,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
```

```
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,4,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,0,0,0,5,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,4,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
int main(int /*argc*/, char */*argv*/[]) {
  double posX = 22, posY = 12;  //начальная позиция x и y
  double dirX = -1, dirY = 0; //начальный вектор направления
  double planeX = 0, planeY = 0.66; //ключи 2D камеры
  double time = 0; //время текущего кадра
  double oldTime = 0; //время предыдущего кадра
  screen(512, 384, 0, "Raycaster");
  while(!done()) {
    for(int x = 0; x < w; x++) {
      //вычисление положение и направление луча
      double cameraX = 2 * x / double(w) - 1;
      //x-координата в пространстве камеры
      double rayPosX = posX;
      double rayPosY = posY;
      double rayDirX = dirX + planeX * cameraX;
      double rayDirY = dirY + planeY * cameraX;
      //длина луча от текущей позиции до следующей координаты
      int mapX = int(rayPosX);
```

```
int mapY = int(rayPosY);
//длина луча от текущей позиции до следующей координаты
double sideDistX;
double sideDistY;
 //длина луча от первой координаты к следующей
double deltaDistX =
sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX));
double deltaDistY =
sqrt(1 + (rayDirX * rayDirX) / (rayDirY * rayDirY));
double perpWallDist;
//в каком направлении к шагу в x или y-направлении
int stepX;
int stepY;
int hit = 0; //был ли удар в стену?
int side;
//вычисление шага и первоначального sideDist
if (rayDirX < 0){
  stepX = -1;
  sideDistX = (rayPosX - mapX) * deltaDistX;
}
else {
  stepX = 1;
  sideDistX = (mapX + 1.0 - rayPosX) * deltaDistX;
}
if (rayDirY < 0) {
  stepY = -1;
  sideDistY = (rayPosY - mapY) * deltaDistY;
}
else{
  stepY = 1;
  sideDistY = (mapY + 1.0 - rayPosY) * deltaDistY;
```

```
}
//выполнить цифровой дифференциальный анализ
while (hit == 0){
  //перейти на следующий квардрат в направлении x или y
  if (sideDistX < sideDistY){
    sideDistX += deltaDistX;
    mapX += stepX;
    side = 0;
  }
  else{
    sideDistY += deltaDistY;
    mapY += stepY;
    side = 1;
  }
  //Проверка если луч врезался в стену
  if (worldMap[mapX][mapY] > 0)
      hit = 1;
}
//Расчёт расстояния проекции на направление камеры
    //(устранение эффекта рыбьего глаза)
if (side == 0)
perpWallDist =
(mapX - rayPosX + (1 - stepX) / 2) / rayDirX;
else
perpWallDist =
(mapY - rayPosY + (1 - stepY) / 2) / rayDirY;
//расчёт высоты линии для отрисовки на экране
int lineHeight = (int)(h / perpWallDist);
//вычисление самого низкого и самого высокого
    //пикселя для заполнения текущей полосы
int drawStart = -lineHeight / 2 + h / 2;
```

```
    if(drawStart < 0)drawStart = 0;
    int drawEnd = lineHeight / 2 + h / 2;
    if(drawEnd >= h)drawEnd = h - 1;
    //выбрать цвет стен
    ColorRGB color;
    switch(worldMap[mapX][mapY]) {
      case 1:  color = RGB_Red;  break; //красный
      case 2:  color = RGB_Green;  break; //зелёный
      case 3:  color = RGB_Blue;  break; //синий
      case 4:  color = RGB_White;  break; //белый
      default: color = RGB_Yellow; break; //жёлтый
    }
    //делаем стороны x и y разной яркости
    if (side == 1) {color = color / 2;}
    //рисуем полосу пикселей в виде вертикальной линии
    verLine(x, drawStart, drawEnd, color);
}
//синхронизации для входа и счетчика FPS
oldTime = time;
time = getTicks();
double frameTime = (time - oldTime) / 1000.0;
//время принятия кадра, сек
print(1.0 / frameTime); //FPS счётчик
redraw();
cls();
//модификаторы скорости
double moveSpeed = frameTime * 5.0;
//константная величина в квадратах / сек
double rotSpeed = frameTime * 3.0;
//константная величина в радианах / сек
readKeys();
```

```
//двигаться вперед, если впереди нет стены
if (keyDown(SDLK_UP)) {
if(worldMap[int(posX + dirX * moveSpeed)][int(posY)]==false)
posX += dirX * moveSpeed;
if(worldMap[int(posX)][int(posY + dirY*moveSpeed)]==false)
posY += dirY * moveSpeed;
}
//двигаться в обратном направлении, если сзади нет стены
if (keyDown(SDLK_DOWN)) {
if(worldMap[int(posX - dirX * moveSpeed)][int(posY)]==false)
posX -= dirX * moveSpeed;
if(worldMap[int(posX)][int(posY - dirY*moveSpeed)]==false)
posY -= dirY * moveSpeed;
}
//повернуть направо
if (keyDown(SDLK_RIGHT)) {
  //как направление камеры и
  //плоскости камеры должны быть повернуты
  double oldDirX = dirX;
  dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
  dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
  double oldPlaneX = planeX;
  planeX=planeX*cos(-rotSpeed)-planeY*sin(-rotSpeed);
  planeY=oldPlaneX*sin(-rotSpeed)+planeY*cos(-rotSpeed);
}
//повернуть влево
if (keyDown(SDLK_LEFT)) {
  //как направление камеры и
  //плоскости камеры должны быть повернуты
  double oldDirX = dirX;
  dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
```

```
            dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
            double oldPlaneX = planeX;
            planeX=planeX*cos(rotSpeed)-planeY*sin(rotSpeed);
            planeY=oldPlaneX*sin(rotSpeed)+planeY*cos(rotSpeed);
        }
    }
}
```