Министерство образования Республики Беларусь

Учреждения образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ

И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

К защите допустить:

Заведующая кафедрой информатики

\_\_\_\_\_\_\_\_\_\_\_\_Н. А. Волорова

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему

**ПОСТРОЕНИЕ И АНАЛИЗ СИМУЛЯТОРА 3Д МИРА**

БГУИР

БГУИР КП 1-40 40 01 113 ПЗ

Студенты: Михайловский А.А.

Беззубенко И. С.

Руководитель: Давыдчик А. В..

Минск 2023

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ 3**](#_Toc131539001)

**1** [**СПОСОБЫ ПРЕДСТАВЛЕНИЯ ИЗОБРАЖЕНИЙ В ПАМЯТИ ЭВМ 5**](#_Toc131539002)

**2** [**АЛГОРИТМ, ИСПОЛЬЗУЮЩИЙ БУФЕР 10**](#_Toc131539003)

**3** [**МЕТОД ТРАССИРОВКИ ЛУЧЕЙ (RAY CASTING) 15**](#_Toc131539004)

**4** [**РЕНДЕРИНГ 16**](#_Toc131539005)

**5** [**ЧТО ТАКОЕ RAY-CASTING 17**](#_Toc131539006)

**6** [**ОБЗОР 20**](#_Toc131539007)

**7** [**ПРИМЕРЫ 22**](#_Toc131539008)

**8** [**ОБОБЩЕННЫЕ ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ 22**](#_Toc131539009)

# **ВВЕДЕНИЕ**

**Актуальность.** На сегодняшний день трудно представить жизнь без компьютеров. Это мощный и многофункциональный инструмент, который предоставляет больше возможности для поисков информации, общения, учебы, работы и развлечения. Благодаря этому незаменимому инструменту многие области деятельности стали более оптимизированными в сфере ведения учета, хранения документов, поиска необходимой информации и совершенствования своих знаний.

Однако в большинстве случаев на компьютерах изначально устанавливается такая операционная система как Windows от компании Microsoft. Так уж получилось, что сама операционная система не может выдавать больше информации о внутренности (аппаратной начинке) компьютера, поэтому и создается программа-утилита для отображения большей технической информации о персональном компьютере пользователя. Так тема «Программа для мониторинга технической информации об устройствах ПК» актуальна на сегодняшний день.

Если заглянуть в историю, то можно проследить, как с момента появления первых ЭВМ люди стремятся разно­образить способы общения человека и машины, приблизившись к уровню общения человека с человеком. Это общение было бы гораздо более ограничен­ным, если бы не использовало один из наиболее простых способов — язык изображе­ний, образов. Сегодня графические изображения на экране монитора современного пер­сонального компьютера стали для нас нормой, совершенно неотъемлемым атрибутом интерфейса. Спектр применения ком­пьютерной графики, помимо средства интерфейса «человек-машина», чрезвычайно широк: от создания реклам­ный роликов, компьютерных мультфильмов и игр, кроя одежды, малых и монументальных форм дизайна, ком­пьютер­ной живописи до визуализации результатов научных изысканий. Можно с уверенностью сказать, что популярность Internet, и в частно­сти WWW, во многом объясняется широ­ким применением графики.

Рынок программного и аппаратного обеспечения компьютерной графики – один из са­мых динамичных. Об этом можно судить по объему литературы и числу сервисов Internet, посвященных так или иначе компьютерной гра­фике. Можно с уверенностью сказать, что как минимум половина из продаваемой литературы по программному обес­печению, посвящена описанию графических пакетов.

Предметом данной работы является обширная область компьютерных наук, посвящен­ная представлению дан­ных в памяти ЭВМ в графической форме. Это самое общее опре­деление, т.к. под данными можно понимать как непосредственно хранящееся в виде файла изображение в одном из графических форматов, так и протокол об­мена командами между пользователем и ЭВМ – то, что мы называем графическим интерфейсом; так и би­товую последовательность, сформированную для вывода на экран или печатающее уст­ройство. Методы и способы представления и манипуляции этим видом данных относится к компетенции компьютерной графики.

**Цель работы.** Исследовать принципы построения 3д миров. Изучение архитектуры современной графики компьютера, рассмотрение основных компонентов графики современного ПК, их предназначения, функционирования во всей системе, их взаимосвязи и взаимодействия, обеспечивающих эффективную работу ПК. Разработка программы, которая позволяет продемонстрировать ray casting в действии.

# **СПОСОБЫ ПРЕДСТАВЛЕНИЯ ИЗОБРАЖЕНИЙ В ПАМЯТИ ЭВМ**

Формальное определение компьютерная (машинная) графика – это создание, хранение и обработка моделей объектов и их изображений с помощью ЭВМ. Под инте­рактивной компьютерной графикой понимают раздел компьютерной графики, изучающий вопросы динамического управления со стороны пользователя содержанием изображения, его формой, размерами и цветом на экране с помощью интерактивных уст­ройств взаимодействия.

Под компьютерной геометрией понимают математический аппарат, применяемый в компьютерной графике.

Рынок программного и аппаратного обеспечения компьютерной графики – один из са­мых динамичных. Об этом можно судить по объему литературы и числу сервисов Internet, посвященных так или иначе компьютерной гра­фике. Можно с уверенностью сказать, что как минимум половина из продаваемой литературы по ПО, посвящена гра­фическим программам.

Необходимо отметить следующую отличительную черту компьютерных изображений. Изображения, которые мы встречаем в нашей повседневной жизни, реальные картины природы, можно бесконечно детализировать, вы­являть все новые цвета и оттенки. Изо­бражения, хранящиеся в памяти компьютера, независимо от способа их по­лучения и представления, всегда являются усеченной моделью картины реального мира. Их детали­зация воз­можна лишь с той степенью, которая была заложена при их создании или полу­чении, и их цветовая гамма будет не шире заранее оговоренной.

Одно и тоже изображение может быть представлено в памяти ЭВМ двумя принципи­ально различными спосо­бами и получено два различных типа изображения: растровое и векторное. Рассмотрим подробнее эти способы представления изображений, выделим их основные параметры и определим их достоинства и недостатки.

Что такое растровое изображение?

Возьмём фотографию (например, рис. 1.1). Конечно, она тоже состоит из маленьких эле­ментов, но будем счи­тать, что отдельные элементы мы рассмотреть не можем, и она представляется для нас, как реальная картина природы.



Рис. 1.1 Исходное изображение

Теперь разобьём это изображение на маленькие квадратики. Маленькие, но всё-таки чётко различимые, и каж­дый квадратик закрасим цветом, преобладающем в нём (на са­мом деле программы при оцифровке генерируют некий "средний" цвет, то есть если у нас была одна чёрная точка и одна белая, то квадратик будет иметь серый цвет).

Как мы видим, изображение стало состоять из конечного числа квадратиков определён­ного цвета. Эти квадра­тики называют pixel (от PICture ELement) - пиксель или пиксель.

Теперь каким-либо методом занумеруем цвета. Конкретная реализация этих методов нас пока не интере­сует - для нас сейчас важно то, что каждый пиксель на рисунке стал иметь определённый цвет, обозначен­ный цифрой (рис. 1.2).

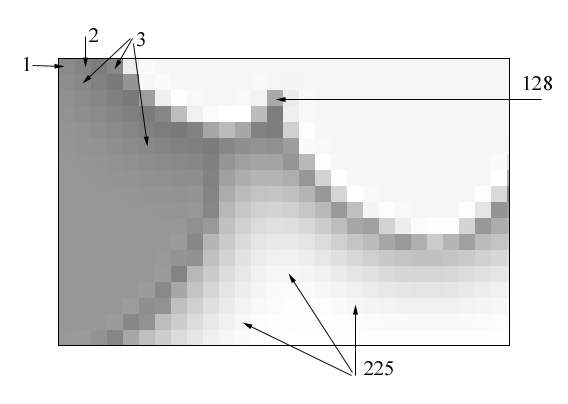
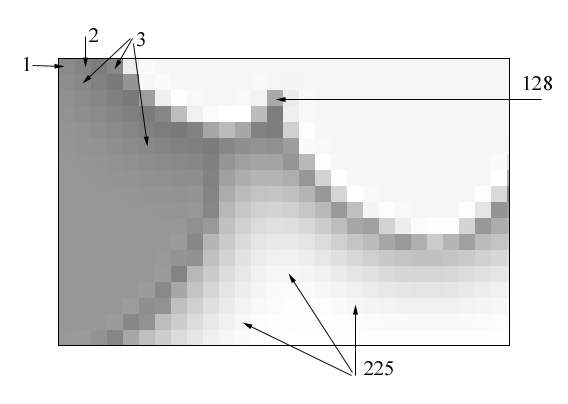


Рис. 1.2 Фрагмент оцифрованного изображения и номера цветов

Теперь пойдём по порядку слева направо и сверху вниз и будем в строчку выписывать номера цветов встречающихся пикселей. Получится строка примерно следующего вида:

1 2 8 3 212 45 67 45 127 4 78 225 34 ...

Вот эта строка и есть наши оцифрованные данные. Теперь мы можем сжать их (так как несжатые графические данные обычно имеют достаточно большой размер) и сохранить в файл.

Итак, под растровым (bitmap, raster) понимают способ представления изображения в виде совокупности от­дельных точек (пикселей) различных цветов или оттенков. Это наиболее простой способ представления изобра­жения, ибо таким образом видит наш глаз.

Достоинством такого способа является возможность получения фотореалистичного изображения высокого ка­чества в различном цветовом диапазоне. Недостатком – высо­кая точность и широкий цветовой диапазон требуют увеличения объема файла для хра­нения изображения и оперативной памяти для его обработки.

Для векторной графики характерно разбиение изображения на ряд графических при­митивов – точка, прямая, ломаная, дуга, полигон. Таким образом, появляется возможность хранить не все точки изображения, а коорди­наты узлов примитивов и их свойства (цвет, связь с другими узлами и т.д.). Вернемся к изображению на рис. 1.1. Взглянем на него по-другому. На изображении легко можно выделить множе­ство простых объектов — отрезки прямых, ломанные, эл­липс, замкнутые кривые. Представим себе, что пространство рисунка существует в неко­торой координатной системе. Тогда можно описать это изображение, как совокупность простых объектов, вышеперечисленных типов, координаты узлов которых заданы векто­ром, отно­сительно точки начала координат (рис. 1.3).

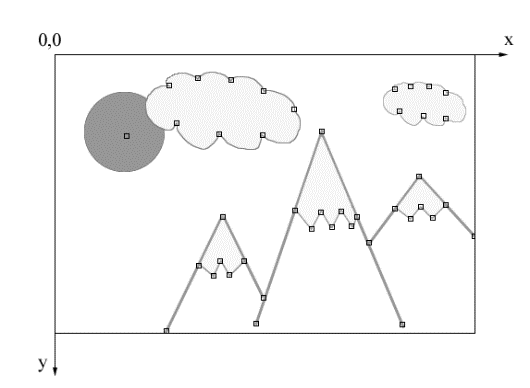


Рис. 1.3 Векторное изображение и узлы его примитивов

Проще говоря, чтобы компьютер нарисовал прямую, нужны координаты двух точек, ко­торые связываются по кратчайшей, для дуги задается радиус и т.д. Таким образом, век­торная иллюстрация - это набор геометрических примитивов. Важной деталью является то, что объекты задаются независимо друг от друга и, следо­вательно, могут перекры­ваться между собой.

При использовании векторного представления изображение хранится в памяти как база данных описаний при­митивов. Основные графические примитивы, используемые в век­торных графических редакторах: точка, прямая, кривая Безье, эллипс (окружность), по­лигон (прямоугольник). Примитив строится вокруг его узлов (nodes). Ко­ординаты узлов задаются относительно координатной системы макета.

А изображение будет представлять из себя массив описаний - нечто типа:

отрезок (20,20-100,80);

окружность(50,40-30);

кривая\_Безье (20,20-50,30-100,50).

Каждому узлу приписывается группа параметров, в зависимости от типа примитива, ко­торые задают его гео­метрию относительно узла. Например, окружность задается одним узлом и одним параметром – радиусом. Такой набор параметров, которые играют роль коэффициентов и других величин в уравнениях и аналитических соот­ношениях объекта данного типа, называют аналитической моделью примитива. Отрисовать примитив – значит построить его геометрическую форму по его параметрам согласно его аналитической модели.

Векторное изображение может быть легко масштабировано без потери деталей, т.к. это требует пересчета срав­нительно небольшого числа координат узлов. Другой термин object-oriented graphics.

Самой простой аналогией векторного изображения может служить аппликация. Все изображение состоит из от­дельных кусочков различной формы и цвета (даже части рас­тра), «склеенных» между собой. Понятно, что таким образом трудно получить фотореалистичное изображение, т.к. на нем сложно выделить конечное число примити­вов, однако существенными достоинствами векторного способа представления изображения, по сравнению с рас­тровым, являются:

векторное изображение может быть легко масштабировано без потери каче­ства, т.к. это требует пересчета срав­нительно небольшого числа координат узлов;

графические файлы, в которых хранятся векторные изображения, имеют суще­ственно меньший, по сравне­нию с растровыми, объем (порядка нескольких кило­байт).

Сферы применения векторной графики очень широки. В полиграфике - от создания красочных иллюстраций до работы со шрифтами. Все, что мы называем машинной гра­фикой, 3D-графикой, графическими средствами ком­пьютерного моделирования и САПР – все это сферы приоритета векторной графики, ибо эти ветви дерева ком­пьютерных наук рассматривают изображение исключительно с позиции его математического пред­ставления. Как видно, векторным можно назвать только способ описания изображения, а само изо­бражение для нашего глаза всегда растровое. Таким образом, задачами векторного гра­фического редактора являются растровая прорисовка графических примитивов и предос­тавление пользователю сервиса по изменению параметров этих примитивов. Все изобра­жение представляет собой базу данных примитивов и параметров макета (размеры хол­ста, единицы измерения и т.д.). Отрисовать изображение – значит выполнить последовательно процедуры прорисовки всех его деталей.

Для уяснения разницы между растровой и векторной графикой приведем простой при­мер. Вы решили отскани­ровать Вашу фотографию размером 10х15 см чтобы затем обра­ботать и распечатать на цветном принтере. Для получения приемлемого качества печати необходимо разрешение не менее 300 dpi. Считаем:

10 см = 3,9 дюйма 15 см = 5,9 дюймов

По вертикали: 3,9 \* 300 = 1170 точек.

По горизонтали: 5,9 \* 300 = 1770 точек.

Итак, число пикселей растровой матрицы 1170 \* 1770 = 2.070.900.

Теперь решим – сколько цветов мы хотим использовать. Для черно-белого изображения используют обычно 256 градаций серого цвета для каждого пикселя или 1 байт. Полу­чаем, что для хранения нашего изображения надо 2.070.900 байт или 1,97 Мб.

Для получения качественного цветного изображения надо не менее 256 оттенков для каждого базового цвета. В модели RGB соответственно их 3: красный, зеленый и синий. Получаем общее количество байт – 3 на каждый пиксел. Соответственно размер храни­мого изображения возрастает в три раза и составляет 5,92 Мб.

Для создания макета для полиграфии фотографии сканируют с разрешением 600 dpi, следовательно, размер файла вырастает еще вчетверо. С другой стороны, если изображение состоит из простых объектов, то для его хранения в векторном виде необходимо не более нескольких килобайт. Управлять редактированием растровых изображений высокой точности под силу только мощным ЭВМ с большим размером ОП. Поэтому персональные компьютеры стали ис­пользоваться для этих целей сравнительно недавно. Причем, зачастую это требует до­полни­тельного оснащения ПК мощными видеокартами, графическими акселераторами и профессиональными монито­рами.

Третьим способом описания и получения изображений на компьютере является использование фрактальной графикой. Базовым элементом фрактальной графики явля­ется сама математическая формула, то есть никаких объектов в памяти компьютера не хранится и изображение строится исключительно по уравнениям. Таким спо­собом стро­ятся как простейшие регулярные структуры, так и сложные иллюстрации, имитирующие природные ландшафты и трехмерные объекты.

Об аппаратном обеспечении средств компьютерной графики речь пойдет далее, а мы остановимся на программных продуктах для редактирования изображений при помощи ЭВМ.

# **АЛГОРИТМ, ИСПОЛЬЗУЮЩИЙ БУФЕР**

Это один из простейших алгоритмов удаления невидимых поверхностей. Впервые он был предложен Кэтмулом. Работает этот алгоритм в пространстве изображения. Идея z-буфера является простым обобщением идеи о бу­фере кадра. Буфер кадра используется для запоминания атрибутов (интенсивности) каждого пиксела в простран­стве изображения, z-буфер - это отдельный буфер глубины, используемый для запоминания координаты z или глубины каждого видимого пиксела в пространстве изображения. В процессе работы глубина или значение z ка­ждого нового пиксела, который нужно зане­сти в буфер кадра, сравнивается с глубиной того пиксела, который уже занесен в z-бу­фер. Если это сравнение показывает, что новый пиксел расположен впереди пиксела, на­ходя­щегося в буфере кадра, то новый пиксел заносится в этот буфер и, кроме того, про­изводится корректировка z-бу­фера новым значением z. Если же сравнение дает противоположный результат, то никаких действий не произво­дится. По сути, алгоритм является поиском по х и у наибольшего значения функции z (х, у).

Главное преимущество алгоритма - его простота. Кроме того, этот алгоритм решает за­дачу об удалении неви­димых поверхностей и делает тривиальной визуализацию пересе­чений сложных поверхностей. Сцены могут быть любой сложности. Поскольку габариты пространства изображения фиксированы, оценка вычислительной трудо­емкости алго­ритма не более чем линейна. Поскольку элементы сцены или картинки можно заносить в буфер кадра или в z-буфер в произвольном порядке, их не нужно предварительно сорти­ровать по приоритету глубины. Поэтому экономится вычислительное время, затрачивае­мое на сортировку по глубине.

Основной недостаток алгоритма - большой объем требуемой памяти. Если сцена под­вергается видовому преоб­разованию и отсекается до фиксированного диапазона коорди­нат z значений, то можно использовать z-буфер с фиксированной точностью. Информацию о глубине нужно обрабатывать с большей точностью, чем координатную информацию на плоскости (х, y); обычно бывает достаточно 20 бит. Буфер кадра разме­ром 512х512х24 бит в комбинации с z-буфером размером 512х512х20 бит требует почти 1.5 мегабайт памяти. Однако снижение цен на память делает экономически оправданным создание специализированных запоминающих уст­ройств для z-буфера и связанной с ним аппаратуры.

Альтернативой созданию специальной памяти для z-буфера является использование для этой цели оперативной или массовой памяти. Уменьшение требуемой памяти достигается разбиением пространства изображения на 4, 16 или больше квадратов или полос. В предельном варианте можно использовать z-буфер размером в одну строку раз­вертки. Для последнего случая имеется интересный алгоритм построчного сканирования. Поскольку каждый элемент сцены обрабатывается много раз, то сегментирование z-бу­фера, вообще говоря, приводит к увеличению времени, необходимого для обработки сцены. Однако сортировка на плоскости, позволяющая не обрабатывать все многоуголь­ники в каждом из квадратов или полос, может значительно сократить этот рост.

Другой недостаток алгоритма z-буфера состоит в трудоемкости и высокой стоимости устранения лестничного эффекта, а также реализации эффектов прозрачности и просве­чивания. Поскольку алгоритм заносит пикселы в буфер кадра в произвольном порядке, то нелегко получить информацию, необходимую для методов устранения лестничного эффекта, основывающихся на предварительной фильтрации. При реализации эффектов прозрачности и просвечивания, пикселы могут заноситься в буфер кадра в некорректном порядке, что ведет к ло­кальным ошибкам.

Формальное описание алгоритма z-буфера таково:

Заполнить буфер кадра фоновым значением интенсивности или цвета.

Заполнить z -буфер минимальным значением z.

Преобразовать каждый многоугольник в растровую форму в произвольном по­рядке.

Для каждого Пиксел(x,y) в многоугольнике вычислить его глубину z(x,y).

Сравнить глубину z(х,у) со значением Zбуфер(х,у), хранящимся в z-буфере в этой же позиции.

Если z(х, у) > Zбуфер (х,у), то записать атрибут этого многоугольника (интенсивность, цвет и т. п.) в буфер кадра и заменить Zбуфер(х,у) на z(х,у). В противном случае никаких действий не производить.

На псевдокоде алгоритм можно представить так:

for all objects

for all covered pixels

compare z

В качестве предварительного шага там, где это целесообразно, применяется удаление нелицевых граней.

Если известно уравнение плоскости, несущей каждый многоугольник, то вычисление глубины каждого пиксела на сканирующей строке можно проделать пошаговым спосо­бом. Грань, при этом, рисуется последовательно строка за строкой. Для нахождения не­обходимых значений используется линейная интерполяция (рис 5.11).

*(xa, y, za)*

*(x1, y1, z1)*

*(x, y, z)*

*(x2, y2, z2)*

*(x3, y3, z3)*

*(xb, y, zb)*

Рис. 4.11 Сканирующая строка по грани

Для рисунка y меняется от y1 до y2 и далее до y3 при этом для каждой строке определя­ется xa, za, xb, zb.

xa = x1 + (x2 - x1)⋅

xb = x1 + (x3 - x1)⋅

za = z1 + (z2 - z1)⋅

zb = z1 + (z3 - z1)⋅ 

На сканирующей строке x меняется от xa до xb и для каждой точки строки определяется глубина z:

z = za + (zb - za)⋅ 

Реализация алгоритма вдоль сканирующей строки позволяет совместить алгоритм z – буфера с алгоритмами рас­тровой развертки ребер и алгоритмами закраски грани.

Проиллюстрируем работу алгоритма на примере для рис. 5.12.

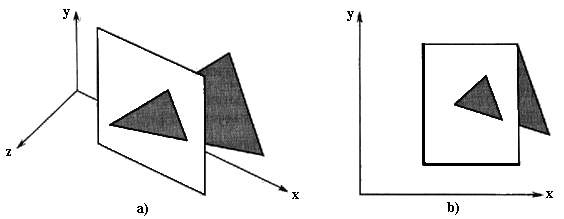


Рис. 4.12 Протыкающий треугольник

В начале в буфере кадра, и в z-буфере содержатся нули. После растровой развертки пря­моугольника содержи­мое буфера кадра будет иметь вид:

Содержимое z-буфера таково:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

При обработке треугольника преобразование его в растровую форму и сравнение по глубине дает новое значе­ние буфера кадра:

Новое содержимое z-буфера таково:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 5 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 5 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 6 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 6 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 11 | 10 | 10 | 10 | 10 | 6 | 5 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 12 | 11 | 10 | 10 | 10 | 10 | 7 | 5 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 13 | 12 | 11 | 11 | 10 | 10 | 10 | 7 | 6 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 14 | 13 | 12 | 12 | 11 | 10 | 10 | 10 | 7 | 6 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 15 | 14 | 13 | 12 | 12 | 11 | 10 | 10 | 10 | 7 | 6 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 14 | 13 | 12 | 12 | 11 | 11 | 10 | 10 | 8 | 7 | 5 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 12 | 11 | 11 | 10 | 10 | 8 | 7 | 5 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 8 | 7 | 6 |
| 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 6 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# МЕТОД ТРАССИРОВКИ ЛУЧЕЙ (RAY CASTING)

В этом методе для каждого пиксела картинной плоскости определяется ближайшая к нему грань, для чего через этот пиксел выпускается луч, находятся все его пересечения с гранями и среди них выбирается ближайшая. Алгоритм на псевдокоде можно кратко за­писать так:

for all pixels

for all objects

compare z

Алгоритм трассировки лучей несколько похож на алгоритм z-буфера, однако здесь циклы по пикселам и по объектам меняются местами.

Алгоритм использующий список приоритетов.

# **РЕНДЕРИНГ**

На этом этапе математическая (векторная) пространственная модель превращается в плоскую (растровую) картинку. Если требуется создать фильм, то рендерится последовательность таких картинок -- кадров. Как структура данных, изображение на экране представлено матрицей точек, где каждая точка определена по крайней мере тремя числами: интенсивностью красного, синего и зелёного цвета. Таким образом рендеринг преобразует трёхмерную векторную структуру данных в плоскую матрицу пикселов. Этот шаг часто требует очень сложных вычислений, особенно если требуется создать иллюзию реальности. Самый простой вид рендеринга -- это построить контуры моделей на экране компьютера с помощью проекции, как показано выше. Обычно этого недостаточно и нужно создать иллюзию материалов, из которых изготовлены объекты, а также рассчитать искажения этих объектов за счёт прозрачных сред (например, жидкости в стакане). Существует несколько технологий рендеринга, часто комбинируемых вместе. Например:

1. Z-буфер (используется в OpenGL и DirectX 10);

2. Сканлайн (scanline) -- он же Ray casting («бросание луча», упрощенный алгоритм обратной трассировки лучей) -- расчёт цвета каждой точки картинки построением луча из точки зрения наблюдателя через воображаемое отверстие в экране на месте этого пиксела «в сцену» до пересечения с первой поверхностью. Цвет пиксела будет таким же, как цвет этой поверхности (иногда с учётом освещения и т. д.);

3. Трассировка лучей (рейтрейсинг, англ. raytracing) -- то же, что и сканлайн, но цвет пиксела уточняется за счёт построения дополнительных лучей (отражённых, преломлённых и т. д.) от точки пересечения луча взгляда. Несмотря на название, применяется только обратная трассировка лучей (то есть как раз от наблюдателя к источнику света), прямая крайне неэффективна и потребляет слишком много ресурсов для получения качественной картинки;

4. Глобальное освещение (англ. global illumination, radiosity) -- расчёт взаимодействия поверхностей и сред в видимом спектре излучения с помощью интегральных уравнений.

Грань между алгоритмами трассировки лучей в настоящее время практически стёрлась. Так, в 3D Studio Max стандартный визуализатор называется Default scanline renderer, но он считает не только вклад диффузного, отражённого и собственного (цвета самосвечения) света, но и сглаженные тени. По этой причине, чаще понятие Raycasting относится к обратной трассировке лучей, а Raytracing -- к прямой.

Вследствие большого объема однотипных вычислений рендеринг можно разбивать на потоки (распараллеливать). Поэтому для рендеринга весьма актуально использование многопроцессорных систем. В последнее время активно ведётся разработка систем рендеринга использующих GPU вместо CPU, и уже сегодня их эффективность для таких вычислений намного выше.

К таким системам относятся:

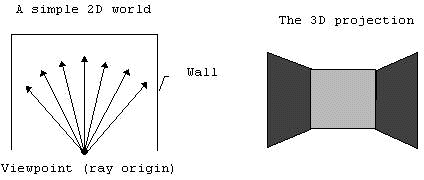
* § Refractive Software Octane Render
* § AAA studio FurryBall
* § RandomControl ARION (гибридная)

Многие производители систем рендеринга для CPU также планируют ввести поддержку GPU (LuxRender, YafaRay, mental images iray).

Самые передовые достижения и идеи трёхмерной графики (и компьютерной графики вообще) докладываются и обсуждаются на ежегодном симпозиуме SIGGRAPH, традиционно проводимом в США.

# **ЧТО ТАКОЕ RAY-CASTING**

Ray-casting (далее — бросание) — это технология, которая преобразует ограниченный набор данных (максимально упрощенная карта, либо же план этажа) в 3D проекцию путем «бросания лучей» из точки обзора по всей области видимости. К примеру, на картинке ниже показано, как бросание преобразует что-то двумерное A во что-то почти что трехмерное B.



Ray-casting и ray-tracing

Как и ray-casting, ray-tracing (далее — трассировка) «определяет видимую часть поверхности путем бросания воображаемых лучей света из точки обзора в сторону объекта на сцене».  
  
 Предыдущие два определения создают видимость, что бросание и трассировка — это одно и то же. На самом деле, некоторые книги вовсе используют оба термина, как взаимозаменяемые. Но тем не менее, с точки зрения gamedev-программистов, бросание лучей стоит рассматривать, как частный случай реализации трассировки.  
  
 Бросание быстрее трассировки. Это возможно за счет того, что оно использует некоторые геометрические ограничения для ускорения процесса рендеринга. К примеру, стены всегда перпендикулярны полу (это можно увидеть в таких играх как Doom или же Wolfenstein 3D). Если бы не такие ограничения, то технология бросания не была бы возможна.  
  
Далее по пунктам представлено глобальное сравнение бросания и трассировки. Главное запомнить, что «при бросании используется меньшее количество лучей, чем при трассировке, из-за определенных геометрических ограничений». Либо же, что «бросание — специализированная реализация трассировки».  
 Бросание: лучи выпускаются и движутся группами, основываясь на некоторых геометрических ограничениях. К примеру, на экране с разрешением 320х200 «генератор лучей» выпустит их только в количестве 320 штук (это число мы получаем из следствия, что экран, имеющий ширину в 320 пикселей, имеет и 320 вертикальных столбцов).  
  
 Трассировка: каждый луч вычисляется по отдельности, то есть каждая точка на экране монитора (обычно, пиксель) трассируется одним своим лучом. К примеру, на экране с разрешением 320х200 нам необходимы 64 тысячи (320 \* 200 = 64000) лучей, что в 200 раз медленнее в сравнение с бросанием.  
 Как правило, при бросании допустимы погрешности в вычислениях.

При трассировке же все должно быть максимально точным.  
  
 Бросание в разы быстрее трассировки и подходит для процессов реального времени. Трассировка же ни как не подходит (разве что у нас не имеется ПК с частотой в 500ГГц)

При бросании мы получаем изображение среднего / плохого качества. Можно заметить деление на блочные элементы. При трассировке же картинка максимально реалистичная — иногда даже слишком.



Сцена из Wolfenstein 3D. Заметьте, картинка делится на прямоугольные блоки. Объекты (оружие) и враги (собака) — это просто прозрачные растровые изображения, которые масштабированы и отрисованы поверх заднего фона.



Сцена из игры 7th Guest. Результат рендеринга поражает. Тем не менее, передвижения игрока ограничены ранее определенными путями (это обусловлено тем, что количество заранее отрендеренных картинок ограничено). При бросании мы геометрически ограничены и в распоряжении находятся только простые фигуры. При трассировке же любая фигура может быть отрендерена. Бросание: отрендеренные изображения не сохраняются на диск. Обычно, только сама карта сцены хранится на носителе, а соответствующие изображения генерируются «на лету».  
  
Трассировка: отрендеренные изображения сохраняются на диск и загружаются оттуда при необходимости. В настоящее время не существует такой аппаратной части, что была бы способна рендерить результат трассировки лучей «на лету».

Примеры

Бросание:

Wolfenstein 3D (iD Software)

Shadow Caster (Raven)

Arena (Bethesda)

Doom (iD Software)

Dark Forces (LucasArts)

Трассировка:

7th Guest (Trilobyte)

Critical Path (Mechadeus)

11th Hour (Trilobyte)

Myst (Cyan)

Cyberia (Xatrix)

# ОБЗОР

В основе проектировки мира лежал Ray casting. Один из [методов](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4) [рендеринга](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BD%D0%B4%D0%B5%D1%80%D0%B8%D0%BD%D0%B3) в [компьютерной графике](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0), при котором сцена строится на основе замеров пересечения лучей с визуализируемой поверхностью.

Использование: Рейкастинг может относиться к:

Общая проблема определения первого пересечения объекта с лучом[[4]](https://ru.wikipedia.org/wiki/Ray_casting#cite_note-4).

Методика для удаления невидимых поверхностей, основанная на поиске первого пересечения луча, который проектируется из точки обозрения к каждому пикселю изображения.

Не рекурсивный вариант [трассировки лучей](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D1%81%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BB%D1%83%D1%87%D0%B5%D0%B9), при котором «бросаются» только первичные лучи.

Метод прямого [объёмного рендеринга](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D1%91%D0%BC%D0%BD%D1%8B%D0%B9_%D1%80%D0%B5%D0%BD%D0%B4%D0%B5%D1%80%D0%B8%D0%BD%D0%B3), также называемый «[volume ray casting](https://ru.wikipedia.org/w/index.php?title=Volume_ray_casting&action=edit&redlink=1" \o "Volume ray casting (страница отсутствует)) ([англ.](https://en.wikipedia.org/wiki/volume_ray_casting))».

В реальной природе [источник света](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA_%D1%81%D0%B2%D0%B5%D1%82%D0%B0) испускает [луч](https://ru.wikipedia.org/wiki/%D0%9B%D1%83%D1%87) света, который, «путешествуя» по пространству, в конечном счёте «натыкается» на какую-либо преграду, которая прерывает распространение этого светового луча.

Луч света можно представить в виде потока [фотонов](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%82%D0%BE%D0%BD), который движется вдоль вектора луча. В какой-либо точке пути с лучом света может случиться любая комбинация трёх вещей: [поглощение](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B3%D0%BB%D0%BE%D1%89%D0%B5%D0%BD%D0%B8%D0%B5), [отражение](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) и [преломление](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%BB%D0%BE%D0%BC%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5). Поверхность может отразить весь световой луч или только его часть в одном или нескольких направлениях.

Поверхность может также поглотить часть светового луча, что приводит к потере интенсивности отраженного и/или преломлённого луча. Если поверхность имеет свойства [прозрачности](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%BE%D0%B7%D1%80%D0%B0%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0&action=edit&redlink=1), то она преломляет часть светового луча внутри себя и изменяет его направление распространения, поглощая некоторый (или весь) спектр луча (и, возможно, изменяя цвет).

Суммарная интенсивность светового луча, которая была «потеряна» вследствие поглощения, преломления и отражения, должна быть в точности равной исходящей (начальной) интенсивности этого луча. Поверхность не может, например, отразить 66 % входящего светового луча, и преломить 50 %, так как сумма этих порций будет равной 116 %, что больше 100 %.

Отсюда вытекает, что отраженные и/или преломлённые лучи должны «стыкаться» с другими поверхностями, где их поглощающие, отражающие и преломляющие способности снова вычисляются, основываясь на результатах вычислений входящих лучей. Некоторые из лучей, сгенерированных источником света, распространяются по пространству и, в конечном счете, попадают на [область просмотра](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%82_%D0%BF%D1%80%D0%BE%D1%81%D0%BC%D0%BE%D1%82%D1%80%D0%B0) (глаз человека, объектив фото- или видеокамеры и т. д.).

Попытка симулировать физический процесс распространения света путём трассировки световых лучей, используя компьютер, является чрезмерно расточительной, так как только незначительная доля лучей, сгенерированных источником света, попадает на область просмотра.

В программе использовался не рекурсивный вариант.

# **ПРИМЕРЫ**

Классический вид RayCasting осуществляемый в программе (Рис. 1):

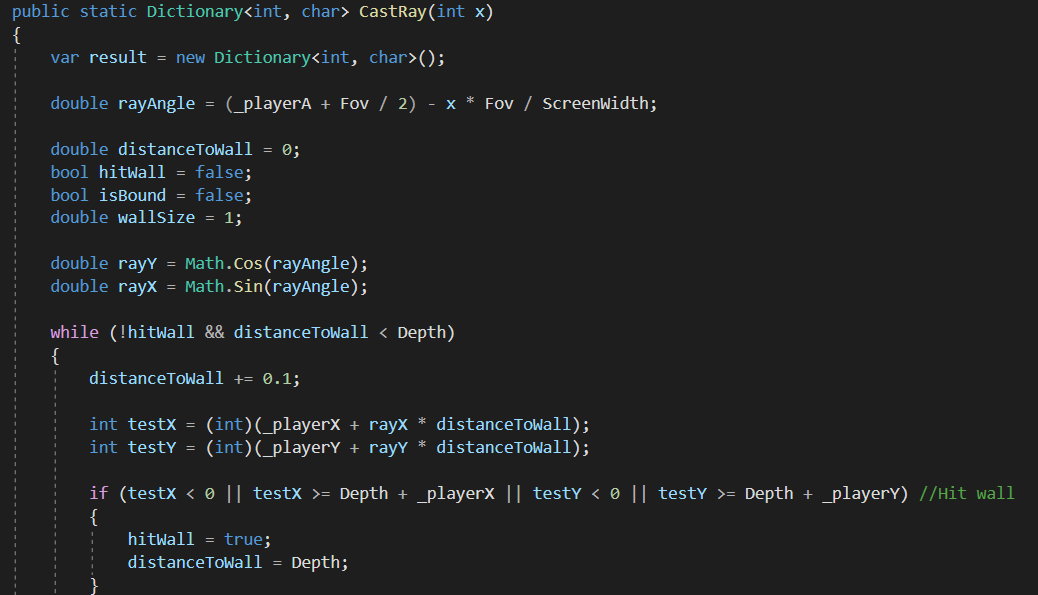


Рис. 1 — Модель RayCasting.

# **ОБОБЩЕННЫЕ ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ**

Где соответственно:

distanceToWall — расстояние от игрока до стены.

hitWall — упирается ли игрок в стенку.

rayY — координаты местоположения игрока по Y.

rayX — координаты местоположения игрока по X.

rayAngle — угол обзора игрока.

ScreenWidth – устанавливаемый размер экрана.

\_playerA — зона обзора игрока.

Fov — Pi / 3.5;

testX — обновляемые координаты игрока по Х.

testY — обновляемые координаты игрока по Y.

Depth — глубина до стены.

При передвижении игрока высчитывается расстояние от игрока до стены и осуществляется проверка hitWall.

var rayCastingTasks = new List<Task<Dictionary<int, char>>>();

for (int x = 0; x < ScreenWidth; x++)

{

var x1 = x;

rayCastingTasks.Add(Task.Run(() => CastRay(x1)));

}

foreach (Dictionary<int, char> dictionary in await Task.WhenAll(rayCastingTasks))

{

foreach (var key in dictionary.Keys)

{

screen[key] = dictionary[key];

}

}

cos(rayAngle);

sin(rayAngle);

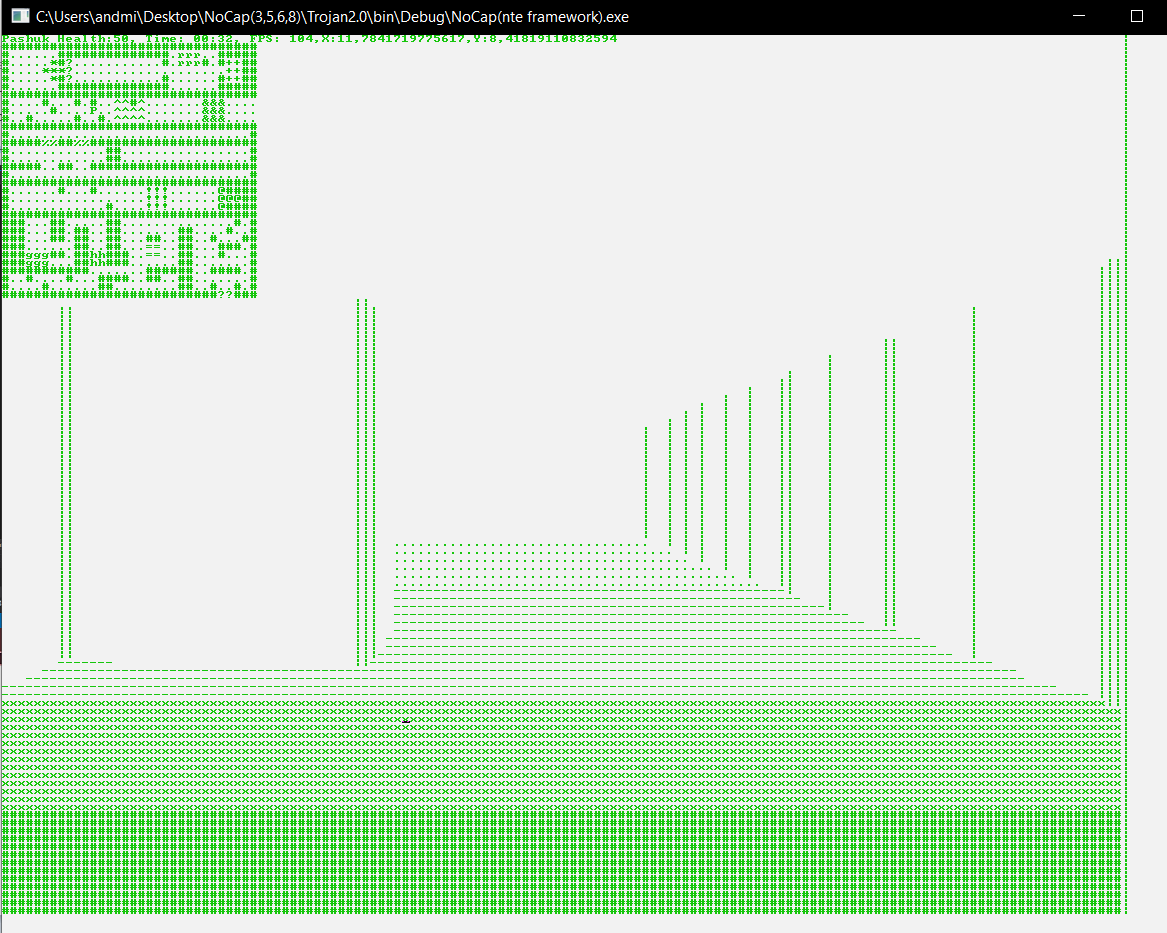
В итоге получается такой вывод (Рис. 5): 

Рис. 5 — экран.

Передвижение происходит с помощью клавиш WASD.

Соответственно:

При нажатии кнопки “A” или «D» происходит поворот игрока влево и вправо.

Где elapsedTime — длительность удерживания клавиши.

При нажатии кнопки «W» или «S» происходит изменение координат X и У,

что создает иллюзию передвижения (Рис. 2).



Рис. 2 — Клавиши «А», «D» и «W».

При нажатии клавиши «Esc» происходит завершение программы (Рис. 3).

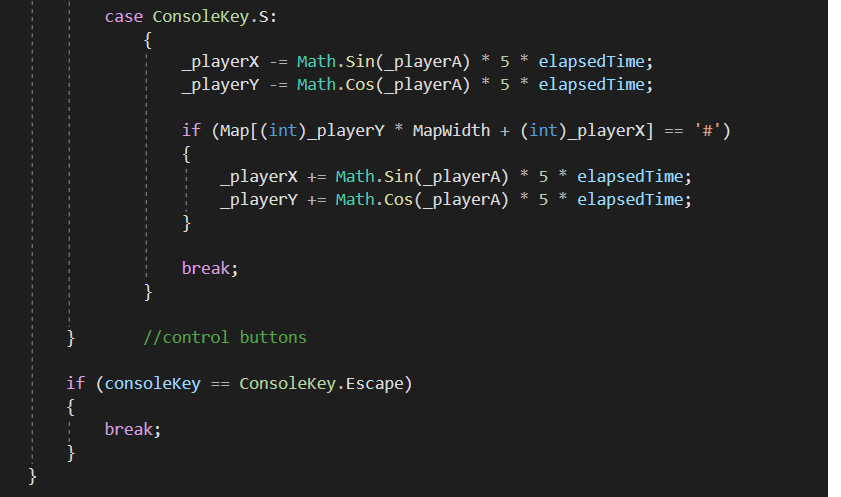


Рис. 3 — Клавиша «S» и «Esc».

Связь координат игрока и прорисовки объектов происходит с помощью мини-карты (Рис. 4).

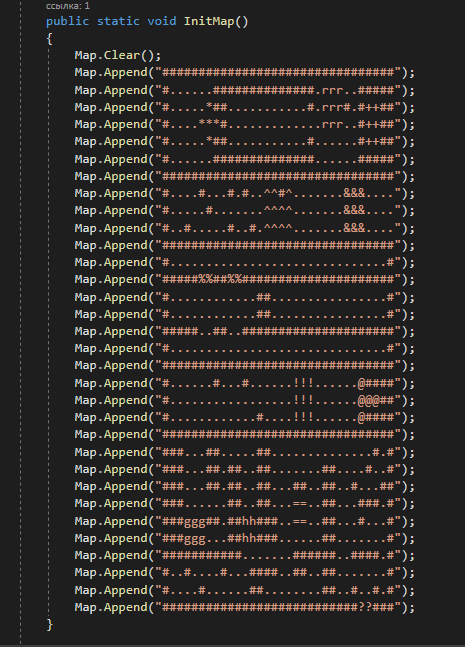


Рис. 4 — Мини-карта.

Где Map — переменная класса string.

//Timer

date1 = date1.AddSeconds(0.005);

timeInGame = date1.ToString("mm:ss");

//Stats

char[] stats = $"Pashuk Health:{Health}, Time: {timeInGame}, FPS: {(int)(1 / elapsedTime)}"

.ToCharArray();

stats.CopyTo(screen, 0);

//Dialoge

stage = Map[(int)(\_playerY + 1) \* MapWidth + (int)\_playerX];

switch (stage)

{

case '\*':

Ramzes v = new Ramzes();

Console.Clear();

Console.SetCursorPosition(0, 0);

Map[4 \* MapHeight + 8] = '?';

Map[3 \* MapHeight + 8] = '?';

Map[2 \* MapHeight + 8] = '?';

Ramzes1.Play();

v.quest("Доброго дня! Сегодня хорошая погода!", "Доброго дня!", "Хорошая!", v);

Console.ForegroundColor = ConsoleColor.DarkGray;

Console.BackgroundColor = ConsoleColor.Yellow;

\_playerX = 12.0;

\_playerY = 3.0;

if (fast) {

\_playerX = 25.0;

\_playerY = 22.0;

Console.BackgroundColor = ConsoleColor.Red;

Console.ForegroundColor = ConsoleColor.DarkMagenta;

SoundTrack.Play();

timeInGame = "";

}

break;

case 'r':

Ramzes ramzes2 = new Ramzes();

Ramzes2.Play();

Console.Clear();

ramzes2.quest("Сегодня на лифте!Ах да у нас же нет лестницы в общежитии!", "Смешно", "Не смешно", ramzes2);

Console.Clear();

Console.SetCursorPosition(0, 0);

Console.ForegroundColor = ConsoleColor.DarkGray;

Console.BackgroundColor = ConsoleColor.Yellow;

Map[3 \* MapWidth + 27] = '.';

Map[3 \* MapWidth + 24] = '.';

Map[3 \* MapWidth + 23] = '.';

Map[3 \* MapWidth + 22] = '.';

\_playerX = 23;

\_playerY = 3;

break;

case '+':

\_playerY = 9.0;

\_playerX = 2.0;

//transition tran = new transition();

//tran.trans(0);

Console.Clear();

Console.SetCursorPosition(0, 0);

kuzya.Play();

Console.ForegroundColor = ConsoleColor.Green;

Console.BackgroundColor = ConsoleColor.White;

break;

case '&':

\_playerY = 14.0;

\_playerX = 5.0;

\_playerA = 2.7;

//transition apchi = new transition();

//apchi.trans(1);

train = true;

metro.Play();

Console.ForegroundColor = ConsoleColor.Gray;

Console.BackgroundColor = ConsoleColor.Blue;

break;

case '^':

kuzya.Stop();

Pashuk b = new Pashuk();

\_playerX = 21.0;

\_playerY = 9.0;

Console.Clear();

Console.SetCursorPosition(0, 0);

Pashuk1.Play();

if (b.quest("Я вам даю 40 вы мне даёте 60 %", "У меня есть леньги", "У меня нет денег"))

{

Health = 100;

}

Console.Clear();

Thanks.Play();

b.quest("Спасибо!!", "Пожалуйста", "Нет");

Console.ForegroundColor = ConsoleColor.Green;

Console.BackgroundColor = ConsoleColor.White;

break;

case '%':

//transition apchi1 = new transition();

//apchi1.trans(2);

cana.PlayLooping();

\_playerY = 19.0;

\_playerX = 2.0;

\_playerA = 1.5;

Console.Clear();

Console.SetCursorPosition(0, 0);

Console.ForegroundColor = ConsoleColor.Green;

Console.BackgroundColor = ConsoleColor.White;

break;

case '@':

\_playerY = 23.0;

\_playerX = 4.0;

\_playerA = 0.0;

//transition apchi3 = new transition();

//apchi3.trans(3);

Console.BackgroundColor = ConsoleColor.Red;

Console.ForegroundColor = ConsoleColor.DarkMagenta;

SoundTrack.Play();

timeInGame = "";

break;

case '!':

cana.Stop();

Alekseev alekseev = new Alekseev();

Console.Clear();

Console.SetCursorPosition(0, 0);

Alekseev1.Play();

alekseev.quest("Хахахах,ты пришёл но ты не сдашь никогда в жизни","...","...");

\_playerY = 19.0;

\_playerX = 24.0;

break;

case '=':

Alekseev alekseev2d = new Alekseev();

Console.Clear();

Console.SetCursorPosition(0, 0);

BALL.Play();

alekseev2d.quest("Сколькой баллов","7","что?");

\_playerY = 22.0;

\_playerX = 19.0;

\_playerA = 1.5;

SoundTrack.Play();

Console.BackgroundColor = ConsoleColor.Red;

Console.ForegroundColor = ConsoleColor.DarkMagenta;

break;

case 'g':

Alekseev alekseev3d = new Alekseev();

Console.Clear();

Console.SetCursorPosition(0, 0);

EXEPTION.Play();

alekseev3d.quest("А исключение это ошибка или ошибка это исключение??", "Вариант ответа один", "Вариант ответа два");

\_playerY = 27.0;

\_playerX = 7.0;

SoundTrack.Play();

Console.BackgroundColor = ConsoleColor.Red;

Console.ForegroundColor = ConsoleColor.DarkMagenta;

break;

case 'h':

Alekseev alekseev5d = new Alekseev();

Console.Clear();

Console.SetCursorPosition(0, 0);

QUESTION.Play();

alekseev5d.quest("Как поймать исключение??", "Нет", "Да");

\_playerY = 28.0;

\_playerX = 16.0;

SoundTrack.Play();

Console.BackgroundColor = ConsoleColor.Red;

Console.ForegroundColor = ConsoleColor.DarkMagenta;

break;

case 'b':

if (Health == 100)

{

Dialog dia = new Dialog();

Console.Clear();

Console.SetCursorPosition(0, 0);

dia.questDialog("Исключения не будет!!!");

}

break;

}

if (train)

{

ConsoleKeyInfo ki = Console.ReadKey(true);

if (ki.Key == ConsoleKey.W)

{

Map[11 \* (MapWidth) + (stepsTrain - 1)] = '#';

Map[11 \* (MapWidth) + (stepsTrain)] = '%';

stepsTrain--;

if (stepsTrain == 0)

{

train = false;

}

}

}

////map

for (int x = 0; x < MapWidth; x++)

{

for (int y = 0; y < MapHeight; y++)

{

screen[(y + 1) \* ScreenWidth + x] = Map[y \* MapWidth + x];

}

}

screen[(int)(\_playerY + 1) \* ScreenWidth + (int)\_playerX] = 'P';

Console.SetCursorPosition(0, 0);

Console.Write(screen, 0, ScreenWidth \* ScreenHeight);

}

}