УО «Белорусский государственный технологический университет»

Факультет информационных технологий

Кафедра «Информационных систем и технологий»

Специальность 1-40 05 01 «Информационные Системы и Технологии»

**Реферат**

**На тему «Текстовый 3D-движок на С++»**

|  |  |  |  |
| --- | --- | --- | --- |
| **Выполнил** |  |  |  |
| Студент 1 курса группы 1 |  |  | Д.И. Велютич |
|  | подпись, дата |  | инициалы и фамилия |
| **Проверил** |  |  | Н.И. Белодед |
|  | подпись, дата |  | инициалы и фамилия |

Минск 2023

Оглавление

[Итак, основные аспекты программы на C++: 2](#_Toc134067123)

[Полный программный код программы: 6](#_Toc134067124)

[Скриншоты и пояснения: 10](#_Toc134067125)

[Выводы: 12](#_Toc134067126)

На основе этой информации вы можете понять принципы работы программы, включая алгоритм трассировки лучей и методы рендеринга. Также вы можете описать процесс обработки пользовательского ввода и детали реализации, такие как использование символов для отображения различных расстояний до стен и форматирование вывода информации об игроке.

# Итак, основные аспекты программы на C++ включают следующее:

1. Инициализация переменных и констант для настроек игры и графики.

2. Создание и установка буфера экрана консоли.

3. Определение карты игрового мира.

4. Основной цикл игры.

5. Обработка пользовательского ввода для управления движением игрока.

6. Трассировка лучей для определения видимых стен на экране.

7. Рендеринг сцены с использованием символов в консоли.

8. Отображение информации об игроке, такой как положение, угол обзора и FPS.

9. Рендеринг мини-карты с текущим положением игрока.

Код представляет собой программу на C++, реализующую простой текстовый 3D-движок для отображения карты лабиринта. Она использует алгоритмы трассировки лучей для определения видимых объектов на экране. Ниже приведена подробная документация для каждого раздела программы.

1. Включение библиотек:

#include <iostream>

#include <vector>

#include <utility>

#include <algorithm>

#include <chrono>

Включение стандартных библиотек C++ для работы с вводом/выводом, контейнерами, алгоритмами и временем.

2. Включение библиотек Windows:

#include <stdio.h>

#include <Windows.h>

Используются для работы с функциями Windows API, управления консолью и получения состояния клавиш.

3. Объявление переменных:

int nScreenWidth = 120;

int nScreenHeight = 40;

int nMapWidth = 16;

int nMapHeight = 16;

float fPlayerX = 14.7f;

float fPlayerY = 5.09f;

float fPlayerA = 0.0f;

float fFOV = 3.14159f / 4.0f;

float fDepth = 16.0f;

float fSpeed = 5.0f;

Объявление глобальных переменных, которые определяют размеры экрана, размеры карты, позицию игрока, угол обзора, максимальную дистанцию прорисовки и скорость движения.

4. Функция `main`:

int main()

{

// Создание экранного буфера

wchar\_t\* screen = new wchar\_t[nScreenWidth \* nScreenHeight];

HANDLE hConsole = CreateConsoleScreenBuffer(GENERIC\_READ | GENERIC\_WRITE, 0, NULL, CONSOLE\_TEXTMODE\_BUFFER, NULL);

SetConsoleActiveScreenBuffer(hConsole);

DWORD dwBytesWritten = 0;

Определение функции `main`, инициализация экранного буфера и создание консольного экранного буфера с помощью Windows API.

5. Задание карты:

wstring map;

map += L"#########.......";

map += L"#...............";

...

map += L"################";

Определение лабиринта в виде строки символов, где символ `#` обозначает стену, а точка (`.`) - свободное пространство.

6. Инициализация переменных времени:

auto tp1 = chrono::system\_clock::now();

auto tp2 = chrono::system\_clock::now();

Используются для измерения прошедшего времени между кадрами и определения FPS.

7. Главный цикл программы:

while (1)

{

// Обновление времени

...

// Обработка пользовательского ввода:

if (GetAsyncKeyState((unsigned short)'A') & 0x8000)

fPlayerA -= (fSpeed \* 0.75f) \* fElapsedTime;

if (GetAsyncKeyState((unsigned short)'D') & 0x8000)

fPlayerA += (fSpeed \* 0.75f) \* fElapsedTime;

if (GetAsyncKeyState((unsigned short)'W') & 0x8000)

{

fPlayerX += sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY += cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

if (map.c\_str()[(int)fPlayerX \* nMapWidth + (int)fPlayerY] == '#')

{

fPlayerX -= sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY -= cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

}

}

if (GetAsyncKeyState((unsigned short)'S') & 0x8000)

{

fPlayerX -= sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY -= cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

if (map.c\_str()[(int)fPlayerX \* nMapWidth + (int)fPlayerY] == '#')

{

fPlayerX += sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY += cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

}

}

Этот блок кода обрабатывает пользовательский ввод с клавиатуры (W, A, S, D) и обновляет позицию и угол поворота игрока в соответствии с этим вводом.

8. Трассировка лучей:

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

{

...

while (!bHitWall && fDistanceToWall < fDepth)

{

...

if (map.c\_str()[nTestX \* nMapWidth + nTestY] == '#')

{

...

}

}

...

}

Этот блок кода реализует алгоритм трассировки лучей, который определяет, какие стены видны игроку на экране.

9. Рендеринг сцены:

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

{

...

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

{

...

}

}

Этот блок кода отвечает за отображение результатов трассировки лучей на экране консоли, используя разные символы для представления различных расстояний до стены.

10. Отображение информации об игроке и мини-карты:

swprintf\_s(screen, 40, L"X=%3.2f, Y=%3.2f, A=%3.2f FPS=%3.2f ", fPlayerX, fPlayerY, fPlayerA, 1.0f / fElapsedTime);

for (int nx = 0; nx < nMapWidth; nx++)

for (int ny = 0; ny < nMapWidth; ny++)

{

screen[(ny + 1) \* nScreenWidth + nx] = map[ny \* nMapWidth + nx];

}

screen[((int)fPlayerX + 1) \* nScreenWidth + (int)fPlayerY] = 'P';

screen[nScreenWidth \* nScreenHeight - 1] = '\0';

WriteConsoleOutputCharacter(hConsole, screen, nScreenWidth \* nScreenHeight, { 0,0 }, &dwBytesWritten);

Этот блок кода отвечает за отображение положения игрока, угла обзора и FPS на экране, а также за рендеринг мини-карты с текущим положением игрока.

# Полный программный код программы:

#include <iostream>

#include <vector>

#include <utility>

#include <algorithm>

#include <chrono>

using namespace std;

#include <stdio.h>

#include <Windows.h>

int nScreenWidth = 120; // Размер экрана консоли X (колонки)

int nScreenHeight = 40; // Размер экрана консоли Y (строки)

int nMapWidth = 16; // Мировые измерения

int nMapHeight = 16;

float fPlayerX = 14.7f; // Стартовая позиция игрока

float fPlayerY = 5.09f;

float fPlayerA = 0.0f; // Стартовое вращение игрокa

float fFOV = 3.14159f / 4.0f; // Поле зрения

float fDepth = 16.0f; // Максимальное расстояние рендеринга

float fSpeed = 5.0f; // Скорость ходьбы

int main()

{

// Создание экранного буфера

wchar\_t\* screen = new wchar\_t[nScreenWidth \* nScreenHeight];

HANDLE hConsole = CreateConsoleScreenBuffer(GENERIC\_READ | GENERIC\_WRITE, 0, NULL, CONSOLE\_TEXTMODE\_BUFFER, NULL);

SetConsoleActiveScreenBuffer(hConsole);

DWORD dwBytesWritten = 0;

// Создаем карту мирового пространства # = блок стены, . = пустота

wstring map;

map += L"#########.......";

map += L"#...............";

map += L"#.......########";

map += L"#..............#";

map += L"#......##......#";

map += L"#......##......#";

map += L"#..............#";

map += L"###............#";

map += L"##.............#";

map += L"#......####..###";

map += L"#......#.......#";

map += L"#......#.......#";

map += L"#..............#";

map += L"#......#########";

map += L"#..............#";

map += L"################";

auto tp1 = chrono::system\_clock::now();

auto tp2 = chrono::system\_clock::now();

while (1)

{

// Нам понадобится разница во времени на кадр для расчета модификации

// скорости движения, чтобы обеспечить последовательное движение, так как трассировка лучей является недетерминированным

tp2 = chrono::system\_clock::now();

chrono::duration<float> elapsedTime = tp2 - tp1;

tp1 = tp2;

float fElapsedTime = elapsedTime.count();

// Вращение рукоятки в направлении против часовой стрелки

if (GetAsyncKeyState((unsigned short)'A') & 0x8000)

fPlayerA -= (fSpeed \* 0.75f) \* fElapsedTime;

// Обработка вращения по часовой стрелке

if (GetAsyncKeyState((unsigned short)'D') & 0x8000)

fPlayerA += (fSpeed \* 0.75f) \* fElapsedTime;

// Обработка движения вперед и столкновений

if (GetAsyncKeyState((unsigned short)'W') & 0x8000)

{

fPlayerX += sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY += cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

if (map.c\_str()[(int)fPlayerX \* nMapWidth + (int)fPlayerY] == '#')

{

fPlayerX -= sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY -= cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

}

}

// Обработка движения назад и столкновений

if (GetAsyncKeyState((unsigned short)'S') & 0x8000)

{

fPlayerX -= sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY -= cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

if (map.c\_str()[(int)fPlayerX \* nMapWidth + (int)fPlayerY] == '#')

{

fPlayerX += sinf(fPlayerA) \* fSpeed \* fElapsedTime;;

fPlayerY += cosf(fPlayerA) \* fSpeed \* fElapsedTime;;

}

}

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

{

// Для каждого столбца вычислите угол проецирования луча в мировое пространство

float fRayAngle = (fPlayerA - fFOV / 2.0f) + ((float)x / (float)nScreenWidth) \* fFOV;

// Найдите расстояние до стены

float fStepSize = 0.1f; // Увеличение размера для приведения лучей, уменьшение для увеличения

float fDistanceToWall = 0.0f; // разрешение

bool bHitWall = false; // Устанавливается, когда луч попадает в блок стены

bool bBoundary = false; // Устанавливается, когда луч попадает на границу между двумя стеновыми блоками

float fEyeX = sinf(fRayAngle); // Единичный вектор для луча в пространстве игрока

float fEyeY = cosf(fRayAngle);

// Инкрементально бросаем луч от игрока, вдоль угла луча, проверяя на пересечения с блоком

while (!bHitWall && fDistanceToWall < fDepth)

{

fDistanceToWall += fStepSize;

int nTestX = (int)(fPlayerX + fEyeX \* fDistanceToWall);

int nTestY = (int)(fPlayerY + fEyeY \* fDistanceToWall);

// Проверяем, не выходит ли луч за пределы границ

if (nTestX < 0 || nTestX >= nMapWidth || nTestY < 0 || nTestY >= nMapHeight)

{

bHitWall = true; // Просто установите расстояние на максимальную глубину

fDistanceToWall = fDepth;

}

else

{

// Луч находится в границах, поэтому проверяем, является ли ячейка луча блоком стены.

if (map.c\_str()[nTestX \* nMapWidth + nTestY] == '#')

{

// Луч ударился о стену

bHitWall = true;

// Чтобы выделить границы плитки, проведите луч от каждого угла плитки к игроку.

// Чем больше этот луч совпадает совпадает с лучом рендеринга, тем ближе мы к границе плитки.

// Границе, которую мы будем затенять, чтобы добавить детали к стенам.

vector<pair<float, float>> p;

// Протестируйте каждый угол плитки, сохранив расстояние до него от

// игрока, и вычисленное точечное произведение двух лучей

for (int tx = 0; tx < 2; tx++)

for (int ty = 0; ty < 2; ty++)

{

// Угол от угла до глаза

float vy = (float)nTestY + ty - fPlayerY;

float vx = (float)nTestX + tx - fPlayerX;

float d = sqrt(vx \* vx + vy \* vy);

float dot = (fEyeX \* vx / d) + (fEyeY \* vy / d);

p.push\_back(make\_pair(d, dot));

}

// Сортировка пар от ближайшей к наиболее удаленной

sort(p.begin(), p.end(), [](const pair<float, float>& left, const pair<float, float>& right) {return left.first < right.first; });

// Первые два/три - самые близкие (мы никогда не увидим всех четырех)

float fBound = 0.01;

if (acos(p.at(0).second) < fBound) bBoundary = true;

if (acos(p.at(1).second) < fBound) bBoundary = true;

if (acos(p.at(2).second) < fBound) bBoundary = true;

}

}

}

// Вычислить расстояние до потолка и пола

int nCeiling = (float)(nScreenHeight / 2.0) - nScreenHeight / ((float)fDistanceToWall);

int nFloor = nScreenHeight - nCeiling;

// Шейдер стен на основе расстояния

short nShade = ' ';

if (fDistanceToWall <= fDepth / 4.0f) nShade = 0x2588; // Очень близко

else if (fDistanceToWall < fDepth / 3.0f) nShade = 0x2593;

else if (fDistanceToWall < fDepth / 2.0f) nShade = 0x2592;

else if (fDistanceToWall < fDepth) nShade = 0x2591;

else nShade = ' '; // Слишком далеко

if (bBoundary) nShade = ' '; // Затемнить

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

{

// Каждая строка

if (y <= nCeiling)

screen[y \* nScreenWidth + x] = ' ';

else if (y > nCeiling && y <= nFloor)

screen[y \* nScreenWidth + x] = nShade;

else // Пол

{

// Затенение пола в зависимости от расстояния

float b = 1.0f - (((float)y - nScreenHeight / 2.0f) / ((float)nScreenHeight / 2.0f));

if (b < 0.25) nShade = '#';

else if (b < 0.5) nShade = 'x';

else if (b < 0.75) nShade = '.';

else if (b < 0.9) nShade = '-';

else nShade = ' ';

screen[y \* nScreenWidth + x] = nShade;

}

}

}

// Отображение статистики

swprintf\_s(screen, 40, L"X=%3.2f, Y=%3.2f, A=%3.2f FPS=%3.2f ", fPlayerX, fPlayerY, fPlayerA, 1.0f / fElapsedTime);

// Отображение карты

for (int nx = 0; nx < nMapWidth; nx++)

for (int ny = 0; ny < nMapWidth; ny++)

{

screen[(ny + 1) \* nScreenWidth + nx] = map[ny \* nMapWidth + nx];

}

screen[((int)fPlayerX + 1) \* nScreenWidth + (int)fPlayerY] = 'P';

// Рамка дисплея

screen[nScreenWidth \* nScreenHeight - 1] = '\0';

WriteConsoleOutputCharacter(hConsole, screen, nScreenWidth \* nScreenHeight, { 0,0 }, &dwBytesWritten);

}

return 0;

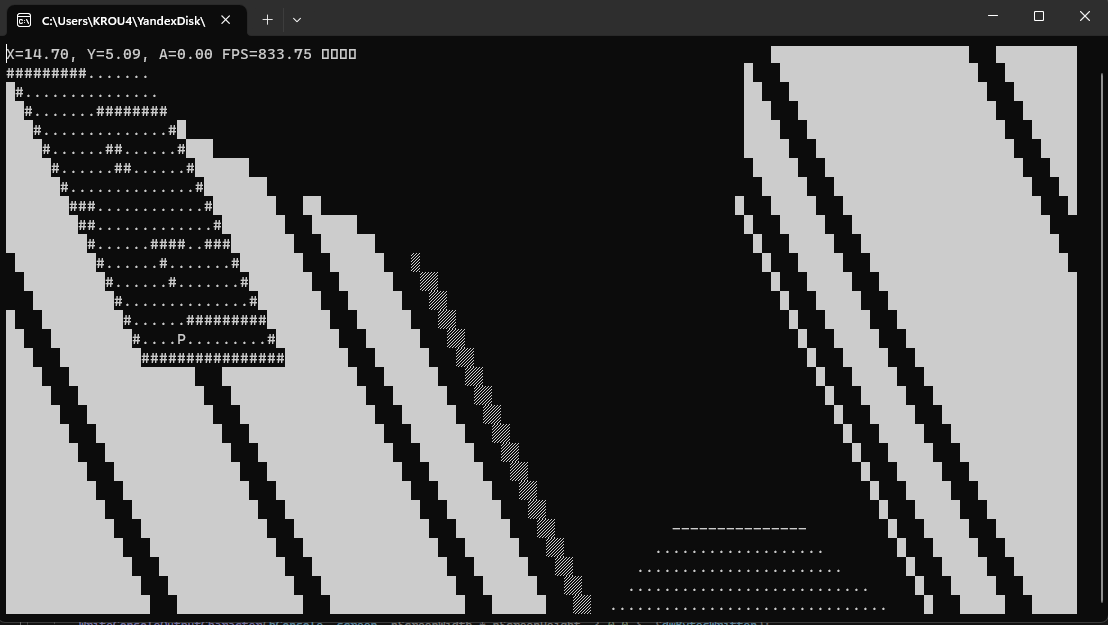
}

# Скриншоты и пояснения:

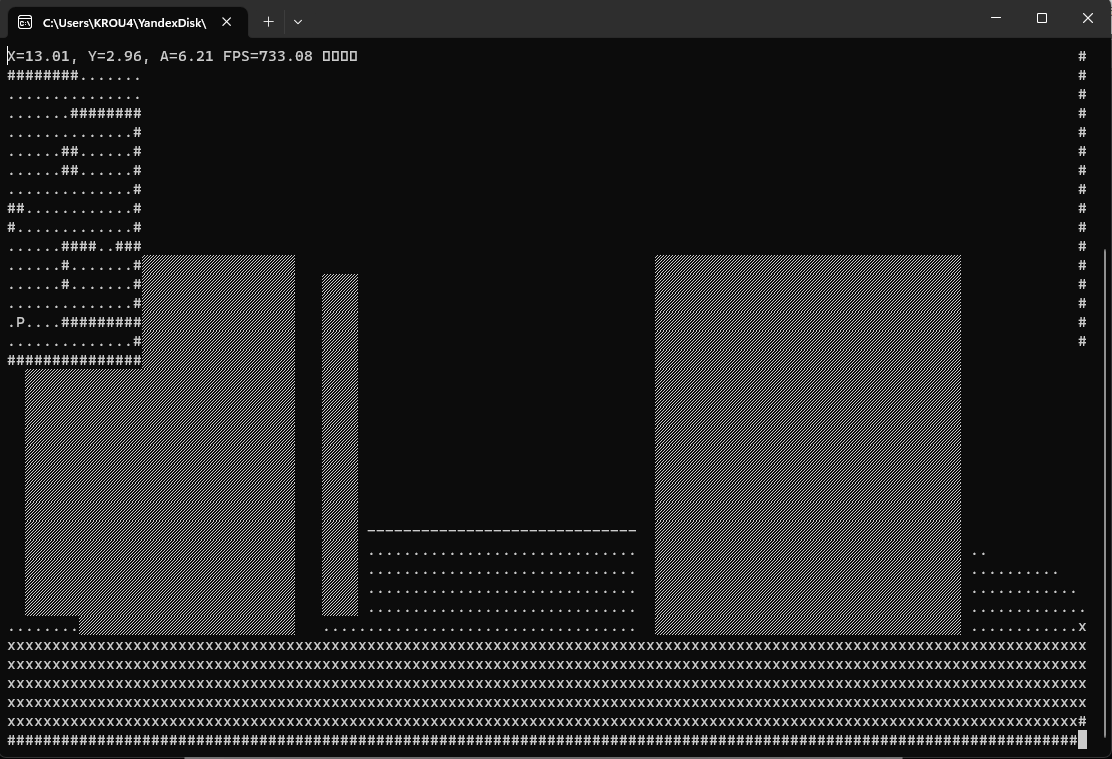
Во-первых, мы создаём нужную нам локацию путём создания карты (пункт 5 из документации).

Далее можем переназначить WASD при необходимости

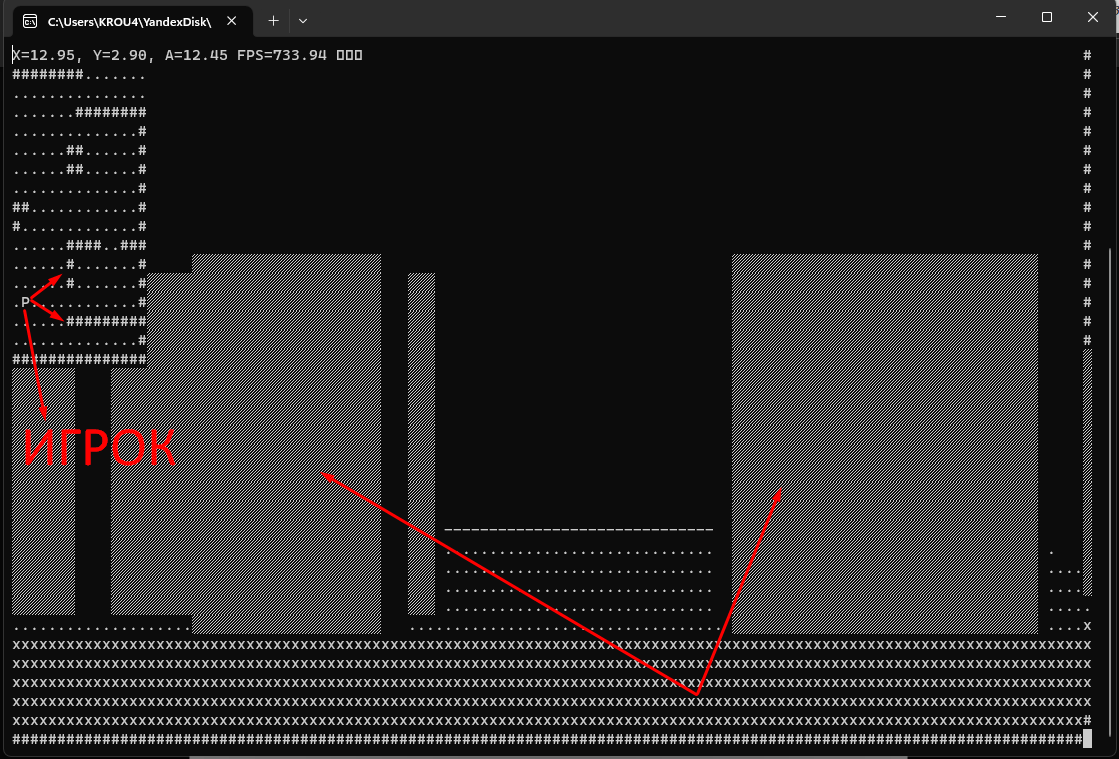
Далее запускаем приложение и выравниваем его по размеру окна консоли.



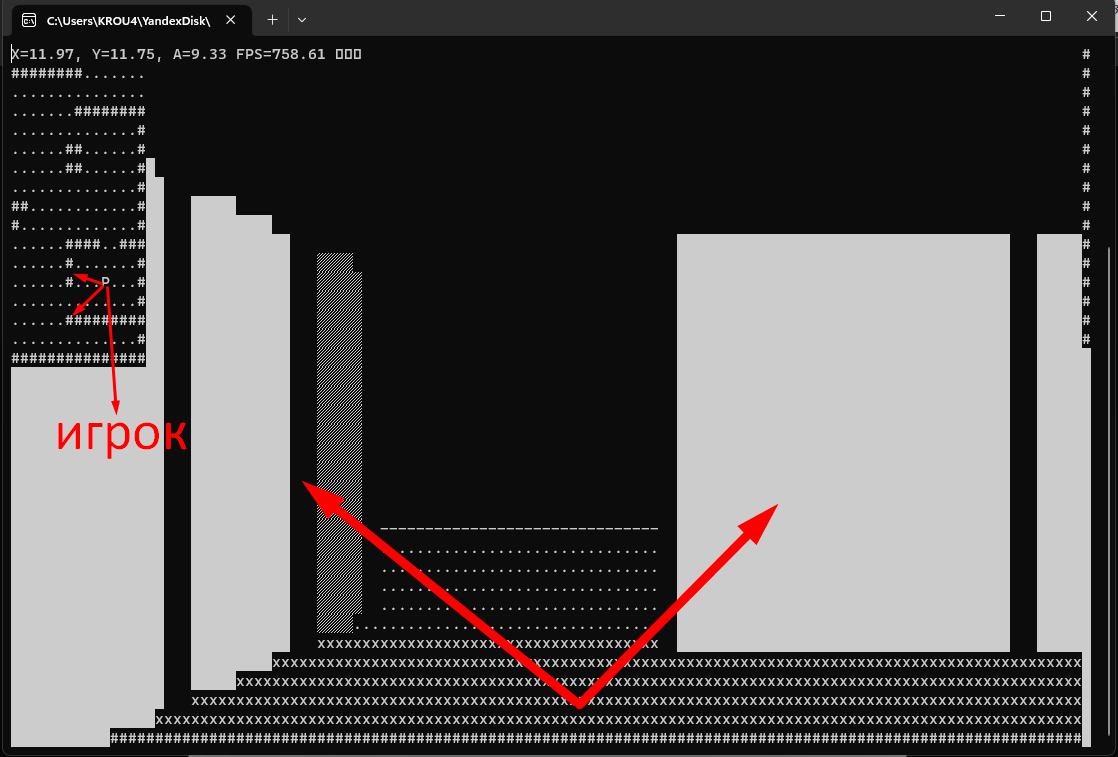
До вот такого



Как раз на этом скриншоте мы можем видеть динамическую отрисовку стен



Вот ещё пример с этими же стенами, но с другого угла



# Выводы:

По данной технологии можно создавать куда более впечатляющие карты/лабиринты, либо использовать данную технологию (технологию трассировки лучей, активно используемая в видеокартах GeForce RTX) в более масштабных проектах, начиная от игр, заканчивая наукой и медициной.