

ВВЕДЕНИЕ.....	3
ХОД РАБОТЫ.....	6
Данные	6
Функции bsp-tree.....	6
Дополнительные функции	13
ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ	
АЛГОРИТМА.....	16
Результат выполнения программы	16
Графики функций initialization и traverse.	16
Результаты исследования.....	17
ЗАКЛЮЧЕНИЕ	18
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	19
ПРИЛОЖЕНИЕ.....	20

ВВЕДЕНИЕ

Разделение двоичного пространства (BSP) — это метод разделения пространства , который рекурсивно разделяет евклидово пространство на два выпуклых множества , используя гиперплоскости в качестве разделов. Этот процесс подразделения приводит к представлению объектов в пространстве в форме древовидной структуры данных , известной как BSP дерево.

Разделение двоичного пространства возникло из-за необходимости компьютерной графики быстро рисовать трехмерные сцены, состоящие из многоугольников. Простым способом рисования таких сцен является алгоритм художника , который создает полигоны в порядке удаления от зрителя, задом наперед, закрашивая фон и предыдущие полигоны с каждым более близким объектом. У этого подхода есть два недостатка: время, необходимое для сортировки полигонов в обратном порядке, и возможность ошибок при перекрытии полигонов. Построение BSP-дерева решает обе эти проблемы, предоставляя быстрый метод сортировки полигонов относительно заданной точки обзора и путем разделения перекрывающихся полигонов на избежать ошибок, которые могут возникнуть в алгоритме художника. Недостатком разделения двоичного пространства является то, что создание дерева BSP может занять много времени. Поэтому обычно он выполняется один раз для статической геометрии на этапе предварительного расчета перед рендерингом или другими операциями в реальном времени на сцене.

Рекурсивный алгоритм построения дерева BSP из этого списка многоугольников:

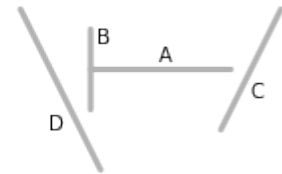
1. Выберите полигон P из списка.
2. Создайте узел N в дереве BSP и добавьте P в список полигонов в этом узле.
3. Для каждого другого полигона в списке:
 - a. Если этот многоугольник полностью находится перед плоскостью, содержащей P , переместите этот многоугольник в список узлов перед P .
 - b. Если этот многоугольник полностью находится за плоскостью, содержащей P , переместите этот многоугольник в список узлов за P .
 - c. Если этот многоугольник пересекается плоскостью, содержащей P , разделите его на два многоугольника и переместите их в соответствующие списки многоугольников позади и перед P .
 - d. Если этот многоугольник лежит в плоскости, содержащей P , добавьте его в список многоугольников в узле N .
4. Примените этот алгоритм к списку многоугольников перед P .
5. Примените этот алгоритм к списку многоугольников позади P .

Следующая диаграмма иллюстрирует использование этого алгоритма для преобразования списка линий или многоугольников в дерево BSP. На каждом из восьми шагов описанный выше алгоритм применяется к списку строк и к дереву добавляется один новый узел.

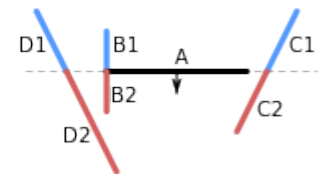
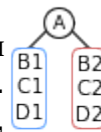
Начните со списка линий (или в 3D многоугольников), составляющих сцену. На древовидных диаграммах списки обозначаются прямоугольниками с закругленными углами, а узлы дерева BSP — кружками. На пространственной диаграмме линий направление, выбранное в качестве «передней» линии, обозначается стрелкой.

Следуя шагам алгоритма выше,

1. Мы выбираем строку A из списка и...
2. ...добавьте его в узел.
3. Мы разделяем оставшиеся строки в списке на те, что перед A (т.е. B2, C2, D2) и те, что позади (B1, C1, D1).
4. Сначала мы обрабатываем строки перед A,...
5. ... за которыми следуют те, кто стоит сзади .

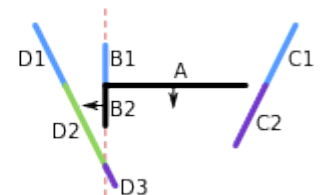
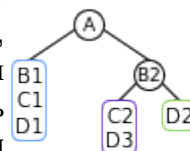


1.



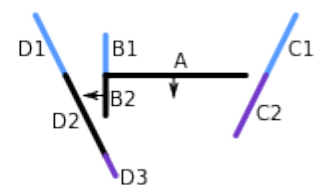
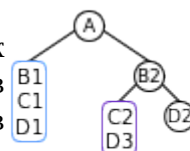
2.

Теперь мы применим алгоритм к списку строк перед A (содержащему B2, C2, D2). Мы выбираем строку B2, добавляем ее в узел и разбиваем остальную часть списка на те строки, которые находятся перед B2 (D2), и те, которые находятся за ним (C2, D3).



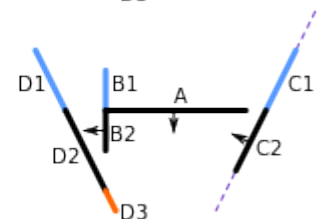
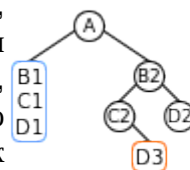
3.

Выберите линию D2 из списка строк перед B2 и A. Это единственная линия в списке, поэтому после добавления ее в узел больше ничего делать не нужно.



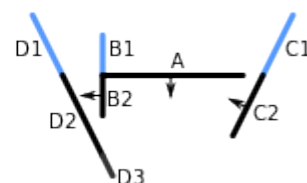
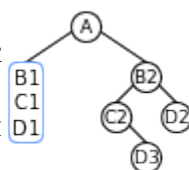
4.

Мы закончили с линиями перед B2, поэтому рассмотрим линии за B2 (C2 и D3). Выберите один из них (C2), добавьте его в узел и поместите другую строку из списка (D3) в список строк перед C2.



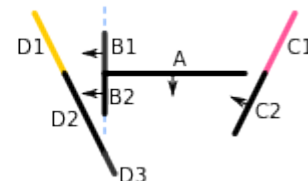
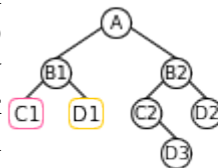
5.

Теперь посмотрите на список строк перед C2. Есть только одна линия (D3), поэтому добавьте ее в узел и продолжайте.



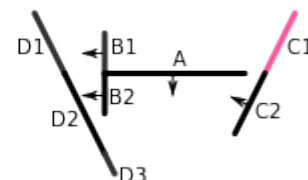
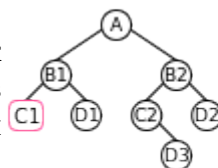
6.

Теперь мы добавили все строки перед A в дерево BSP, поэтому начинаем со списка строк после A. Выбрав строку (B1) из этого списка, мы добавляем B1 к узлу и разделяем остаток список на строки перед B1 (т.е. D1) и строки за B1 (т.е. C1).



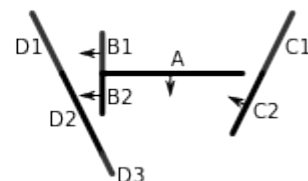
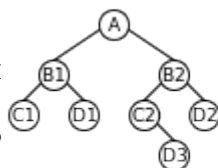
7.

Сначала обрабатывается список строк перед B1, D1 — единственная строка в этом списке, поэтому добавьте ее в узел и продолжайте.



8.

Далее, взглянув на список строк за B1, можно увидеть, что единственной строкой в этом списке является C1, поэтому добавьте ее в узел, и дерево BSP будет завершено.



ХОД РАБОТЫ

Данные

Структура node представляет узел bsp-tree, который содержит список полигонов(pols) и указатели на правое(right) и левое(left) поддереву.

```
struct node
{
    node *left;
    node *right;
    vector<polygon> pols;
};
```

Структура polygon является массивом из 3 векторов.

```
struct polygon
{
    vec3 p[3];
};
```

Структура vec3 вмещает в себя 3 координаты x,y,z. Структура plane является разбивающей плоскостью и имеет тип данных vec4(x,y,z,w).

```
struct vec3
{
    float x;
    float y;
    float z;
};
struct vec4
{
    float x;
    float y;
    float z;
    float w;
};
typedef vec4 plane;
```

Функции bsp-tree

Функция initialization принимает на вход список полигонов, создает корень дерева и вызывает рекурсивную функцию для построения bsp-tree.

```

void initialization(const vector<polygon> &polygons)
{
    if (polygons.empty())
    {
        return;
    }

    allPolygons = 0;

    root = new node;
    nodes = 1;
    construct_bspTree(polygons, root);
}

```

Функция `construct_bspTree` принимает на вход список полигонов и структуру `node`. В начале выбирается случайный полигон из списка переданных, этот полигон добавляется в структуру `node`. Затем с помощью функции `to_plane` этот полигон становится разбивающей плоскостью. В середине функции перебираются все полигоны из начального списка и проверяется, где находятся эти полигоны(`distance`) относительно разбивающей плоскости: спереди, сзади, в плоскости или же пересекаются разбивающей плоскостью. Если сзади, то полигон добавляется в список `polygons_back`, если спереди, то в список `polygons_front`, если в плоскости, то добавляется в список того же `node`, если пересекается, то вызывается функция `polygon_split()`. В конце функции происходит рекурсивный вызов функции, в которую передается ранее сформированные списки и правый или левый `node`.

```

void construct_bspTree(const vector<polygon> &polygons, node *n)
{
    int pol_i = polygon_index(polygons);
    n->pols.push_back(polygons[pol_i]);
    plane pl;
    to_plane(polygons[pol_i], pl);
    vector<polygon> polygons_front;
    vector<polygon> polygons_back;
    for (unsigned int i = 0; i < polygons.size(); ++i)
    {
        if (i != pol_i)
        {
            switch (distance(pl, polygons[i]))
            {
                case ON:
                    n->pols.push_back(polygons[i]);
                    break;
                case FRONT:
                    polygons_front.push_back(polygons[i]);
                    break;
                case BACK:
                    polygons_back.push_back(polygons[i]);
                    break;
                case HALF:
                    polygon_split(pl, polygons[i], polygons_front, polygons_back);
                    break;
            }
        }
    }
    allPolygons += n->pols.size();
    if (!polygons_front.empty())
    {
        n->right = new node;
        ++nodes;
        construct_bspTree(polygons_front, n->right);
    }
    else
    {
        n->right = nullptr;
    }
    if (!polygons_back.empty())
    {
        n->left = new node;
        ++nodes;
        construct_bspTree(polygons_back, n->left);
    }
    else
    {
        n->left = nullptr;
    }
}

```

Функция to_plane принимает на вход полигон и разбивающую плоскость, куда будет записывать результат выполнения программы. В начале

вычисляются 2 вектора, которые лежат в полигоне, затем с помощью функции cross, которая находит векторное произведение, находится нормаль плоскости. В конце вычисляется коэффициент плоскости с помощью функции dot, которая вычисляет скалярное произведение векторов.

```
void to_plane(const polygon &pol, plane &pl)
{
    vec3 u = {pol.p[1].x - pol.p[0].x, pol.p[1].y - pol.p[0].y, pol.p[1].z - pol.p[0].z};
    vec3 v = {pol.p[2].x - pol.p[0].x, pol.p[2].y - pol.p[0].y, pol.p[2].z - pol.p[0].z};

    // Вычисляем нормаль плоскости
    vec3 n = cross(u, v);

    pl.x = n.x;
    pl.y = n.y;
    pl.z = n.z;

    // Вычисляем w коэффициент плоскости путем взятия скалярного произведения вектора r и n
    pl.w = -dot({pl.x, pl.y, pl.z}, pol.p[0]);
}
```

```
vec3 cross(vec3 u, vec3 v)
{
    vec3 res;
    res.x = u.y * v.z - v.y * u.z;
    res.y = v.x * u.z - u.x * v.z;
    res.z = u.x * v.y - v.x * u.y;
    return res;
}

float dot(vec3 p1, vec3 p2)
{
    return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z;
}

float dot(vec4 p1, vec4 p2)
{
    return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z + p1.w * p2.w;
}
```

Функция distance принимает на вход полигон и разбивающую плоскость и высчитывает скалярное произведение между каждым вектором полигона и разбивающей плоскостью. Если все 3 точки больше нуля, то полигон находится спереди, если меньше, то сзади, если все три вектора равны 0, то полигон находится в плоскости. Если все предыдущие условия не выполняются, значит разбивающая плоскость разбивает полигон.

```

dist_res distance(const plane &pl, const polygon &pol)
{
    float d1 = dot(pl, {pol.p[0].x, pol.p[0].y, pol.p[0].z, 1});
    float d2 = dot(pl, {pol.p[1].x, pol.p[1].y, pol.p[1].z, 1});
    float d3 = dot(pl, {pol.p[2].x, pol.p[2].y, pol.p[2].z, 1});
    if (d1 == 0 && d2 == 0 && d3 == 0)
    {
        return ON;
    }
    if (d1 < 0 && d2 > 0)
    {
        return HALF;
    }
    else
    {
        if (d3 <= 0)
        {
            return BACK;
        }
        else
        {
            return FRONT;
        }
    }
}

```

Функция `polygon_split` в зависимости от того, какой из векторов разбивается плоскостью, выбирается в каком порядке нужно передавать аргументы в функцию `polygon_split`.

```

void polygon_split(const plane &p1, const polygon &pol, vector<polygon> &polygons_front, vector<polygon> &polygons_back)
{
    float d1 = dot(p1, {pol.p[0].x, pol.p[0].y, pol.p[0].z, 1});
    float d2 = dot(p1, {pol.p[1].x, pol.p[1].y, pol.p[1].z, 1});
    float d3 = dot(p1, {pol.p[2].x, pol.p[2].y, pol.p[2].z, 1});

    if (d1 <= 0 && d2 >= 0 && d3 >= 0)
    {
        create_new_polygons(p1, pol.p[0], pol.p[1], pol.p[2], polygons_front, polygons_back);
    }
    else if (d2 <= 0 && d1 >= 0 && d3 >= 0)
    {
        create_new_polygons(p1, pol.p[1], pol.p[0], pol.p[2], polygons_front, polygons_back);
    }
    else if (d3 <= 0 && d1 >= 0 && d2 >= 0)
    {
        create_new_polygons(p1, pol.p[2], pol.p[0], pol.p[1], polygons_front, polygons_back);
    }
    else if (d1 >= 0 && d2 <= 0 && d3 <= 0)
    {
        create_new_polygons(p1, pol.p[0], pol.p[1], pol.p[2], polygons_back, polygons_front);
    }
    else if (d2 >= 0 && d1 <= 0 && d3 <= 0)
    {
        create_new_polygons(p1, pol.p[1], pol.p[0], pol.p[2], polygons_back, polygons_front);
    }
    else if (d3 >= 0 && d1 <= 0 && d2 <= 0)
    {
        create_new_polygons(p1, pol.p[0], pol.p[1], pol.p[2], polygons_back, polygons_front);
    }
}

```

В функции `create_new_polygons` находятся точки пересечения полигона и разбивающей плоскости, создается три полигона и добавляются в списки `polygons back` и `polygons front`.

```

void create_new_polygons(const plane &p1, const vec3 &a, const vec3 &b1, const vec3 &b2, vector<polygon> &polygons_a, vector<polygon> &polygons_b)
{
    vec3 i_ab1;
    plane_segment_intersection(p1, a, b1, i_ab1);

    vec3 i_ab2;
    plane_segment_intersection(p1, a, b2, i_ab2);

    polygon p_a;
    p_a.p[0] = a;
    p_a.p[1] = i_ab1;
    p_a.p[2] = i_ab2;
    polygons_a.push_back(p_a);

    polygon p_b1;
    p_b1.p[0] = b1;
    p_b1.p[1] = i_ab2;
    p_b1.p[2] = i_ab1;
    polygons_b.push_back(p_b1);

    polygon p_b2;
    p_b2.p[0] = b2;
    p_b2.p[1] = b1;
    p_b2.p[2] = i_ab2;
    polygons_b.push_back(p_b2);
}

```

Функция `plane_segment_intersection` принимает на вход вектор `a`, вектор `b`, разбивающую плоскость и вектор `i`, куда будет записываться результат. В начале вычисляется вектор `rd`, путем вычитания вектора `b` из вектора `a`. Затем вычисляется коэффициент `t`. Если он входит в промежуток `[0;1]`, то точка

находится на векторе. Тогда вектор вектору i присваивается сумма a и вектора rd умноженному на t , иначе rd .

```
void plane_segment_intersection(const plane &pl, const vec3 &a, const vec3 &b, vec3 &i)
{
    vec3 rd = {b.x - a.x, b.y - a.y, b.z - a.z};

    float t = -(pl.w + dot(a, {pl.x, pl.y, pl.z})) / dot(rd, {pl.x, pl.y, pl.z});
    if (t >= 0 && t <= 1)
    {
        i = {a.x + t * rd.x, a.y + t * rd.y, a.z + t * rd.z};
    }
    else
    {
        i = rd;
    }
}
```

Функция `delete_bspTree` удаляет bsp-tree, путем рекурсивного обхода дерева и освобождения памяти.

```
void delete_bspTree(node *n)
{
    if (n->left != nullptr)
    {
        delete_bspTree(n->left);
    }

    if (n->right != nullptr)
    {
        delete_bspTree(n->right);
    }

    delete n;
}
```

Функция `traverse_tree` рекурсивно обходит каждый узел дерева и добавляет в список `partitions` все полигоны.

```

void traverse_tree(bsp_tree::node *tree, vector<bsp_tree::polygon> &partitions)
{
    if (tree == NULL)
        return;

    if (tree->right)
    {
        traverse_tree(tree->right, partitions);
    }
    if (tree->left)
    {
        traverse_tree(tree->left, partitions);
    }
    for (int i = 0; i < tree->pols.size(); i++)
    {
        partitions.push_back(tree->pols[i]);
    }
}

```

Дополнительные функции

Функция load_ply считывает данные с ply файла. В начале выполняются проверки, что файл соответствует ply формату. Затем считываются координаты и вершины. Каждой вершине сопоставляется координата, тем самым формируя полигон.

```

void load_ply(const string &filename, vector<bsp_tree::polygon> &polygons)
{
    ifstream file(filename);
    if (!file.is_open())
        throw logic_error("Failed to open file");

    string line;
    if (!getline(file, line))
        throw logic_error("Failed to read file");

    if (line != "ply")
        throw logic_error("The file does not contain ply");

    unsigned int vertices_size;
    unsigned int faces_size;

    bool header = true;
    while (header && getline(file, line))
    {
        string s;
        stringstream iss(line);

        if (!(iss >> s))
            throw logic_error("Failed to read file");

        if (s == "format")
        {
            if (!(iss >> s))
                throw logic_error("The file does not contain format");

            if (s != "ascii")
                throw logic_error("The file does not contain ascii");
        }
        else if (s == "element")
        {
            unsigned int n;
            if (!(iss >> s >> n))
                throw logic_error("Failed to read file");

            if (s == "vertex")
            {
                vertices_size = n;
            }
            else if (s == "face")
            {
                faces_size = n;
            }
        }
        else if (s == "end_header")
        {
            header = false;
        }
    }
}

```

```

vector<bsp_tree::vec3> vertices;
vertices.resize(vertices_size);

for (unsigned int i = 0; i < vertices_size; ++i)
{
    if (!(file >> vertices[i].x >> vertices[i].y >> vertices[i].z))
        throw logic_error("Failed to read coordinates in file");
}

struct ply_polygon
{
    unsigned int a, b, c;
};

vector<ply_polygon> faces;
faces.resize(faces_size);
polygons.resize(faces_size);
for (unsigned int i = 0; i < faces_size; ++i)
{
    unsigned int n;

    if (file >> n >> faces[i].a >> faces[i].b >> faces[i].c)
    {
        polygons[i].p[0] = vertices[faces[i].a];
        polygons[i].p[1] = vertices[faces[i].b];
        polygons[i].p[2] = vertices[faces[i].c];
    }
    else
        throw logic_error("Failed to read faces in file");

    if (n != 3)
        throw logic_error("All polygons must be triangles");
}
}

```

Функция experiment высчитывает время выполнения функций initialization и traverse tree.

```

void experiment(const vector<bsp_tree::polygon> &polygons)
{
    bsp_tree tree;
    double t;

    t = wtime();
    tree.initialization(polygons);
    t = wtime() - t;
    cout << "Time of construct bsp-tree: " << t << endl;

    vector<bsp_tree::polygon> partition;
    t = wtime();
    traverse_tree(tree.root, partition);
    t = wtime() - t;
    cout << "Time of traverse bsp-tree: " << t << endl;

    cout << "Polygons: " << polygons.size() << endl;
    cout << "Nodes: " << tree.get_nodes() << endl;
}

```

Функция `wtime()` возвращает нынешнее время в секундах.

```
double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    double res = (double)t.tv_sec + (double)t.tv_usec * 1E-6;
    return res;
}
```


ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ АЛГОРИТМА

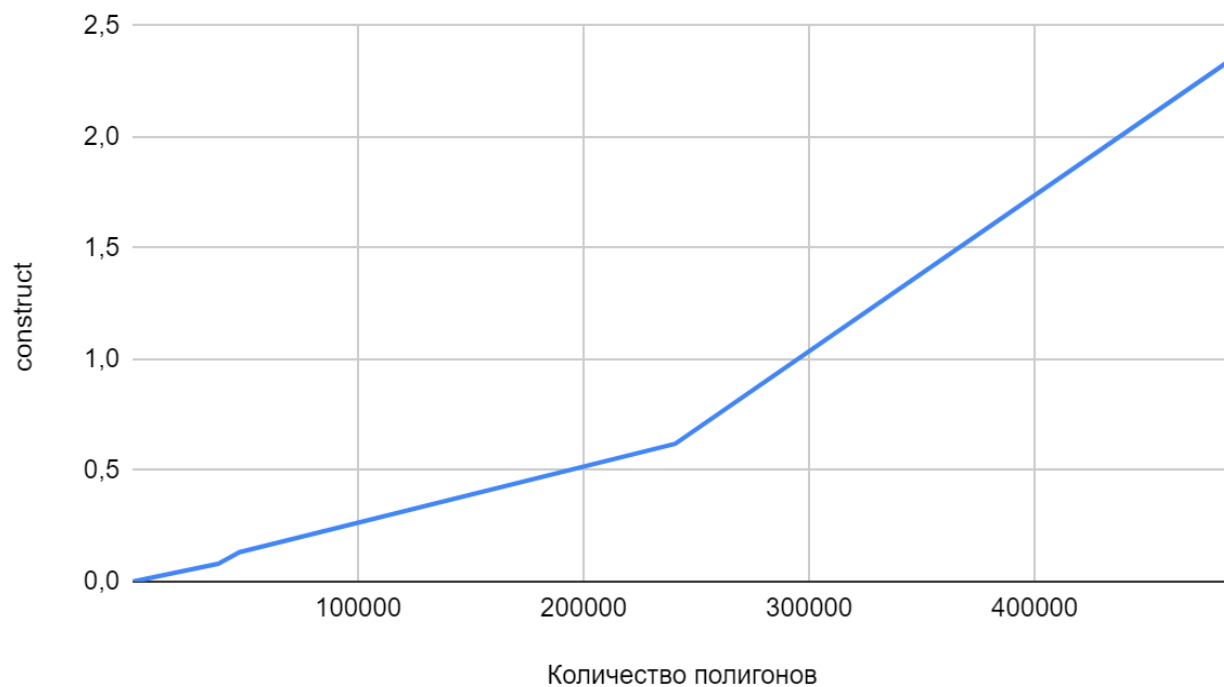
Результат выполнения программы

```
cube
Time of construct bsp-tree: 1.00136e-05
Time of traverse bsp-tree: 1.90735e-06
Polygons: 12
Nodes: 6
4cubes
Time of construct bsp-tree: 4.29153e-05
Time of traverse bsp-tree: 3.09944e-06
Polygons: 48
Nodes: 25
monkey
Time of construct bsp-tree: 0.00181985
Time of traverse bsp-tree: 9.48906e-05
Polygons: 967
Nodes: 2028
dragon.ply
Time of construct bsp-tree: 0.079174
Time of traverse bsp-tree: 0.004076
Polygons: 37986
Nodes: 88704
pharaon
Time of construct bsp-tree: 0.132076
Time of traverse bsp-tree: 0.00417399
Polygons: 47392
Nodes: 113543
highPolDragon
Time of construct bsp-tree: 0.619257
Time of traverse bsp-tree: 0.0284419
Polygons: 240600
Nodes: 666775
CheGuevara
Time of construct bsp-tree: 2.35613
Time of traverse bsp-tree: 0.0501192
Polygons: 487384
Nodes: 1119069
```

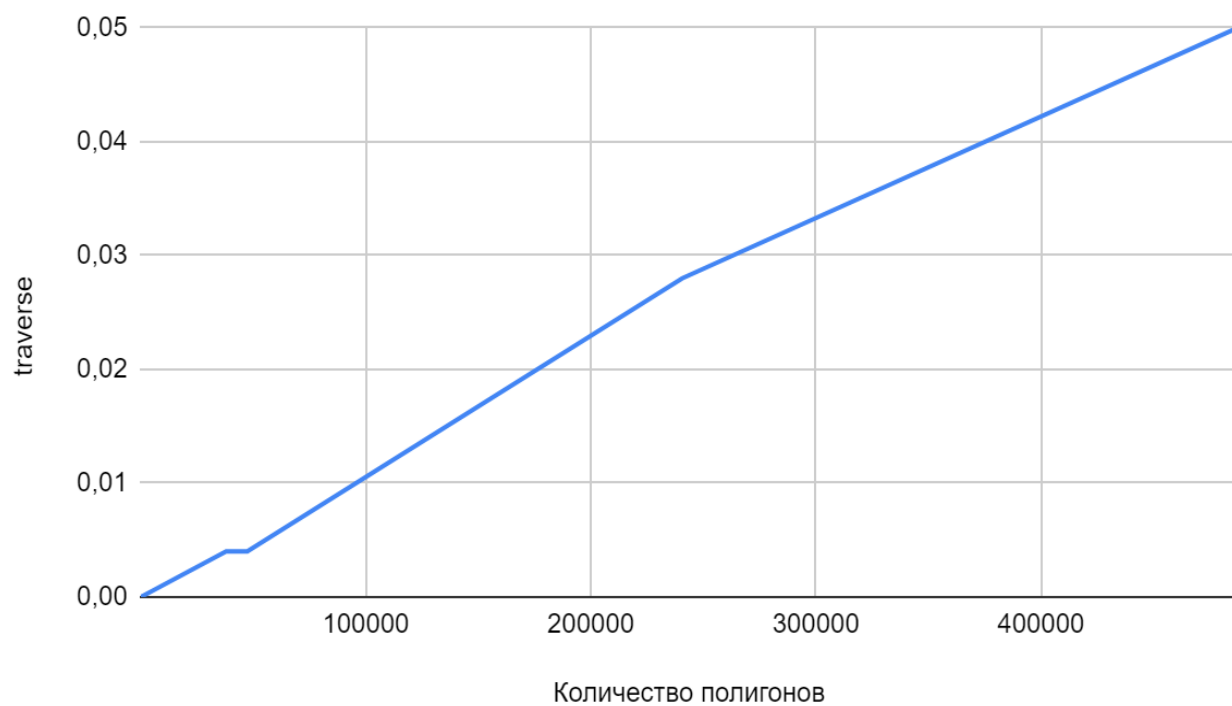
По полученным данным составим графики.

Графики функций initialization и traverse.

Время выполнения функции initialization, с



Время выполнения функции traverse, с



Результаты исследования

Сложность построения двоичного дерева поиска составляет $O(n \log n)$, где n - количество узлов в дереве. Сложность обхода двоичного дерева поиска составляет $O(n)$, где n - количество узлов в дереве. Таким образом, BSP дерево является эффективной структурой данных для хранения полигонов.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы было разработано бинарное дерево поиска и проведено исследование его времени выполнения операции в зависимости от объема данных. Вычислительная сложность построения бинарного дерева поиска составляет $O(n \log n)$, вычислительная сложность обхода бинарного дерева поиска составляет $O(n)$. Осуществлено моделирование разработанного алгоритма, которое подтвердило его эффективность и быстродействие. Таким образом можно сделать вывод, что бинарное дерево является полезной структурой данных для хранения полигонов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ахо А. В., Хопкрофт Д., Ульман Д. Д. Структуры данных и алгоритмы. – М.: Вильямс, 2001. – 384 с.
2. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ. – 3-е изд. – М.: Вильямс, 2013. – 1328 с.
3. Кормен Т. Х. Алгоритмы: Вводный курс. – М.: Вильямс, 2014. – 208 с.
4. Левитин А. В. Алгоритмы: введение в разработку и анализ. – М.: Вильямс, 2006. – 576 с.
5. Седжвик Р. Фундаментальные алгоритмы на С++. Анализ. Структуры данных. Сортировка. Поиск. – К.: ДиаСофт, 2001. – 688 с.
6. Скиена С. С. Алгоритмы. Руководство по разработке. – 2-е изд. – СПб: БХВ, 2011 – 720 с.
7. Макконнелл Дж. Основы современных алгоритмов. – 2-е изд. – М.: Техносфера, 2004. – 368 с.
8. Миллер Р. Последовательные и параллельные алгоритмы: общий подход. – М.: БИНОМ, 2006. – 406 с.
9. Сегаран Т. Программируем коллективный разум. – М.: Символ-Плюс, 2008. – 368 с.
10. https://en.wikipedia.org/wiki/Binary_space_partitioning

ПРИЛОЖЕНИЕ

Исходный код программы

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <fstream>
4  #include <vector>
5  #include <sstream>
6  #include <sys/time.h>
7  using namespace std;
8  class bsp_tree
9  {
10 public:
11     struct vec3
12     {
13         float x;
14         float y;
15         float z;
16     };
17     struct vec4
18     {
19         float x;
20         float y;
21         float z;
22         float w;
23     };
24     typedef vec4 plane;
25     struct polygon
26     {
27         vec3 p[3];
28     };
29     struct node
30     {
31         node *left;
32         node *right;
33         vector<polygon> polys; // список всех узлов, находящийся в одной плоскости
34     };
35     node *root;
36     enum dist_res
37     {
38         ON = 0,
39         FRONT = 1,
40         BACK = 2,
41         HALF = 3
42     };
43
44     int nodes;
45     int allPolygons;
```

```

46  vec3 cross(vec3 u, vec3 v)
47  {
48      vec3 res;
49      res.x = u.y * v.z - v.y * u.z;
50      res.y = v.x * u.z - u.x * v.z;
51      res.z = u.x * v.y - v.x * u.y;
52      return res;
53  }
54  float dot(vec3 p1, vec3 p2)
55  {
56      return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z;
57  }
58  float dot(vec4 p1, vec4 p2)
59  {
60      return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z + p1.w * p2.w;
61  }
62  int polygon_index(const vector<polygon> &polygons)
63  {
64      return rand() % polygons.size();
65  }
66  void plane_segment_intersection(const plane &pl, const vec3 &a, const vec3 &b, vec3 &i)
67  {
68      vec3 rd = {b.x - a.x, b.y - a.y, b.z - a.z};
69
70      float t = -(pl.w + dot(a, {pl.x, pl.y, pl.z})) / dot(rd, {pl.x, pl.y, pl.z});
71      if (t >= 0 && t <= 1)
72      {
73          i = {a.x + t * rd.x, a.y + t * rd.y, a.z + t * rd.z};
74      }
75      else
76      {
77          i = rd;
78      }
79  }
80  void create_new_polygons(const plane &pl, const vec3 &a, const vec3 &b1, const vec3
81  &b2, vector<polygon> &polygons_a, vector<polygon> &polygons_b)
82  {
83      vec3 i_ab1;
84      plane_segment_intersection(pl, a, b1, i_ab1);
85
86      vec3 i_ab2;
87      plane_segment_intersection(pl, a, b2, i_ab2);
88
89      polygon p_a;
90      p_a.p[0] = a;
91      p_a.p[1] = i_ab1;
92      p_a.p[2] = i_ab2;
93      polygons_a.push_back(p_a);
94

```

```

95     polygon p_b1;
96     p_b1.p[0] = b1;
97     p_b1.p[1] = i_ab2;
98     p_b1.p[2] = i_ab1;
99     polygons_b.push_back(p_b1);
100
101     polygon p_b2;
102     p_b2.p[0] = b1;
103     p_b2.p[1] = b2;
104     p_b2.p[2] = i_ab2;
105     polygons_b.push_back(p_b2);
106 }
107 void polygon_split(const plane &pl, const polygon &pol, vector<polygon>
108 &polygons_front, vector<polygon> &polygons_back)
109 {
110     float d1 = dot(pl, {pol.p[0].x, pol.p[0].y, pol.p[0].z, 1});
111     float d2 = dot(pl, {pol.p[1].x, pol.p[1].y, pol.p[1].z, 1});
112     float d3 = dot(pl, {pol.p[2].x, pol.p[2].y, pol.p[2].z, 1});
113
114     if (d1 <= 0 && d2 >= 0 && d3 >= 0)
115     {
116         create_new_polygons(pl, pol.p[0], pol.p[1], pol.p[2], polygons_front, polygons_back);
117     }
118     else if (d2 <= 0 && d1 >= 0 && d3 >= 0)
119     {
120         create_new_polygons(pl, pol.p[1], pol.p[0], pol.p[2], polygons_front, polygons_back);
121     }
122     else if (d3 <= 0 && d1 >= 0 && d2 >= 0)
123     {
124         create_new_polygons(pl, pol.p[2], pol.p[0], pol.p[1], polygons_front, polygons_back);
125     }
126     else if (d1 >= 0 && d2 <= 0 && d3 <= 0)
127     {
128         create_new_polygons(pl, pol.p[0], pol.p[1], pol.p[2], polygons_back, polygons_front);
129     }
130     else if (d2 >= 0 && d1 <= 0 && d3 <= 0)
131     {
132         create_new_polygons(pl, pol.p[1], pol.p[0], pol.p[2], polygons_back, polygons_front);
133     }
134     else if (d3 >= 0 && d1 <= 0 && d2 <= 0)
135     {
136         create_new_polygons(pl, pol.p[0], pol.p[1], pol.p[2], polygons_back, polygons_front);
137     }
138 }
139 // вычисляем уравнение плоскости на основе многоугольника,
140 // а функция расстояния определяет положение многоугольника относительно
141 плоскости
142 void to_plane(const polygon &pol, plane &pl)
143 {

```



```

144     vec3 u = {pol.p[1].x - pol.p[0].x, pol.p[1].y - pol.p[0].y, pol.p[1].z - pol.p[0].z};
145     vec3 v = {pol.p[2].x - pol.p[0].x, pol.p[2].y - pol.p[0].y, pol.p[2].z - pol.p[0].z};
146
147     // Вычисляем нормаль плоскости
148     vec3 n = cross(u, v);
149
150     pl.x = n.x;
151     pl.y = n.y;
152     pl.z = n.z;
153
154     // Вычисляем w коэффициент плоскости путем взятия скалярного произведения
155     вектора r и точки многоугольника
156     pl.w = -dot({pl.x, pl.y, pl.z}, pol.p[0]);
157 }
158
159 dist_res distance(const plane &pl, const polygon &pol)
160 {
161     float d1 = dot(pl, {pol.p[0].x, pol.p[0].y, pol.p[0].z, 1});
162     float d2 = dot(pl, {pol.p[1].x, pol.p[1].y, pol.p[1].z, 1});
163     float d3 = dot(pl, {pol.p[2].x, pol.p[2].y, pol.p[2].z, 1});
164     if (d1 == 0 && d2 == 0 && d3 == 0)
165     {
166         return ON;
167     }
168     if (d1 < 0 && d2 > 0)
169     {
170         return HALF;
171     }
172     else
173     {
174         if (d3 <= 0)
175         {
176             return BACK;
177         }
178         else
179         {
180             return FRONT;
181         }
182     }
183 }
184 void construct_bspTree(const vector<polygon> &polygons, node *n)
185 {
186     int pol_i = polygon_index(polygons);
187     n->pols.push_back(polygons[pol_i]);
188
189     plane pl;
190     to_plane(polygons[pol_i], pl);
191
192     vector<polygon> polygons_front;

```

```

193     vector<polygon> polygons_back;
194
195     for (unsigned int i = 0; i < polygons.size(); ++i)
196     {
197         if (i != pol_i)
198         {
199             switch (distance(pl, polygons[i]))
200             {
201                 case ON:
202                     n->pols.push_back(polygons[i]);
203                     break;
204                 case FRONT:
205                     polygons_front.push_back(polygons[i]);
206                     break;
207
208                 case BACK:
209                     polygons_back.push_back(polygons[i]);
210                     break;
211
212                 case HALF:
213                     polygon_split(pl, polygons[i], polygons_front, polygons_back);
214                     break;
215             }
216         }
217     }
218
219     allPollygons += n->pols.size();
220
221     if (!polygons_front.empty())
222     {
223         n->right = new node;
224         ++nodes;
225         construct_bspTree(polygons_front, n->right);
226     }
227     else
228     {
229         n->right = nullptr;
230     }
231
232     if (!polygons_back.empty())
233     {
234         n->left = new node;
235         ++nodes;
236         construct_bspTree(polygons_back, n->left);
237     }
238     else
239     {
240         n->left = nullptr;
241     }

```

```

242     }
243     void initialization(const vector<polygon> &polygons)
244     {
245         if (polygons.empty())
246         {
247             return;
248         }
249
250         allPollygons = 0;
251
252         root = new node;
253         nodes = 1;
254         construct_bspTree(polygons, root);
255     }
256
257     ~bsp_tree()
258     {
259         delete_bspTree(root);
260     }
261
262     void delete_bspTree(node *n)
263     {
264         if (n->left != nullptr)
265         {
266             delete_bspTree(n->left);
267         }
268
269         if (n->right != nullptr)
270         {
271             delete_bspTree(n->right);
272         }
273
274         delete n;
275     }
276
277     int get_nodes()
278     {
279         return nodes;
280     }
281
282     int get_fragments()
283     {
284         return allPollygons;
285     }
286 };
287 double wtime()
288 {
289     struct timeval t;
290     gettimeofday(&t, NULL);

```

```

291     double res = (double)t.tv_sec + (double)t.tv_usec * 1E-6;
292     return res;
293 }
294 void load_ply(const string &filename, vector<bsp_tree::polygon> &polygons)
295 {
296     ifstream file(filename);
297     if (!file.is_open())
298         throw logic_error("Failed to open file");
299
300     string line;
301     if (!getline(file, line))
302         throw logic_error("Failed to read file");
303
304     if (line != "ply")
305         throw logic_error("The file does not contain ply");
306
307     unsigned int vertices_size;
308     unsigned int faces_size;
309
310     bool header = true;
311     while (header && getline(file, line))
312     {
313         string s;
314         stringstream iss(line);
315
316         if (!(iss >> s))
317             throw logic_error("Failed to read file");
318
319         if (s == "format")
320         {
321             if (!(iss >> s))
322                 throw logic_error("The file does not contain format");
323
324             if (s != "ascii")
325                 throw logic_error("The file does not contain ascii");
326         }
327         else if (s == "element")
328         {
329             unsigned int n;
330             if (!(iss >> s >> n))
331                 throw logic_error("Failed to read file");
332
333             if (s == "vertex")
334             {
335                 vertices_size = n;
336             }
337             else if (s == "face")
338             {
339                 faces_size = n;

```

```

340     }
341 }
342 else if (s == "end_header")
343 {
344     header = false;
345 }
346 }
347
348 vector<bsp_tree::vec3> vertices;
349 vertices.resize(vertices_size);
350
351 for (unsigned int i = 0; i < vertices_size; ++i)
352 {
353     if (!(file >> vertices[i].x >> vertices[i].y >> vertices[i].z))
354         throw logic_error("Failed to read coordinates in file");
355 }
356
357 struct ply_polygon
358 {
359     unsigned int a, b, c;
360 };
361
362 vector<ply_polygon> faces;
363 faces.resize(faces_size);
364 polygons.resize(faces_size);
365 for (unsigned int i = 0; i < faces_size; ++i)
366 {
367     unsigned int n;
368
369     if (file >> n >> faces[i].a >> faces[i].b >> faces[i].c)
370     {
371         polygons[i].p[0] = vertices[faces[i].a];
372         polygons[i].p[1] = vertices[faces[i].b];
373         polygons[i].p[2] = vertices[faces[i].c];
374     }
375     else
376         throw logic_error("Failed to read faces in file");
377
378     if (n != 3)
379         throw logic_error("All polygons must be triangles");
380 }
381 }
382 void traverse_tree(bsp_tree::node *tree, vector<bsp_tree::polygon> &partitions)
383 {
384
385     if (tree == NULL)
386         return;
387
388     if (tree->right)

```

```

389     {
390         traverse_tree(tree->right, partitions);
391     }
392     if (tree->left)
393     {
394         traverse_tree(tree->left, partitions);
395     }
396     for (int i = 0; i < tree->pols.size(); i++)
397     {
398         partitions.push_back(tree->pols[i]);
399     }
400 }
401 void experiment(const vector<bsp_tree::polygon> &polygons)
402 {
403     bsp_tree tree;
404     double t;
405
406     t = wtime();
407     tree.initialization(polygons);
408     t = wtime() - t;
409     cout << "Time of construct bsp-tree: " << t << endl;
410
411     vector<bsp_tree::polygon> partition;
412     t = wtime();
413     traverse_tree(tree.root, partition);
414     t = wtime() - t;
415     cout << "Time of traverse bsp-tree: " << t << endl;
416
417     cout << "Polygons: " << polygons.size() << endl;
418     cout << "Nodes: " << tree.get_nodes() << endl;
419 }
420 int main(int argc, char **argv)
421 {
422     srand(time(NULL));
423     cout << "cube" << endl;
424     vector<bsp_tree::polygon> polygons;
425     try
426     {
427         load_ply("models/cube.ply", polygons);
428     }
429     catch (const exception &ex)
430     {
431         cerr << "The file format must be ply. " << ex.what() << endl;
432         exit(1);
433     }
434     experiment(polygons);
435     polygons.clear();
436
437     cout << "4cubes" << endl;

```

```

438 try
439 {
440     load_ply("models/4cubes.ply", polygons);
441 }
442 catch (const exception &ex)
443 {
444     cerr << "The file format must be ply. " << ex.what() << endl;
445     exit(1);
446 }
447 experiment(polygons);
448 polygons.clear();
449
450 cout << "monkey" << endl;
451 try
452 {
453     load_ply("models/monkey.ply", polygons);
454 }
455 catch (const exception &ex)
456 {
457     cerr << "The file format must be ply. " << ex.what() << endl;
458     exit(1);
459 }
460 experiment(polygons);
461 polygons.clear();
462
463 cout << "dragon.ply" << endl; // 37986
464 try
465 {
466     load_ply("models/dragon.ply", polygons);
467 }
468 catch (const exception &ex)
469 {
470     cerr << "The file format must be ply. " << ex.what() << endl;
471     exit(1);
472 }
473 experiment(polygons);
474 polygons.clear();
475
476 cout << "pharaon" << endl; // 47392
477 try
478 {
479     load_ply("models/pharaon.ply", polygons);
480 }
481 catch (const exception &ex)
482 {
483     cerr << "The file format must be ply. " << ex.what() << endl;
484     exit(1);
485 }
486 experiment(polygons);

```

```

487 polygons.clear();
488 cout << "highPolDragon" << endl; // 240600
489 try
490 {
491     load_ply("models/highPolDragon.ply", polygons);
492 }
493 catch (const exception &ex)
494 {
495     cerr << "The file format must be ply. " << ex.what() << endl;
496     exit(1);
497 }
498 experiment(polygons);
499 polygons.clear();
500 cout << "CheGuevara" << endl; // 487384
501 try
502 {
503     load_ply("models/CheGuevara.ply", polygons);
504 }
505 catch (const exception &ex)
506 {
507     cerr << "The file format must be ply. " << ex.what() << endl;
508     exit(1);
509 }
510 experiment(polygons);
511 polygons.clear();
512 return 0;
513 }
514

```