ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

Бинарное дерево поиска

|  |  |
| --- | --- |
| Выполнил студент | Бойваленко Никита Евгеньевич |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИВ-221 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | ст. преп. Кафедры ВС Д. М. Берлизов |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2023

[ВВЕДЕНИЕ 3](#__RefHeading___4)

[ХОД РАБОТЫ 6](#__RefHeading___7)

[Данные 6](#__RefHeading___8)

[Функции bsp-tree 6](#__RefHeading___10)

[Дополнительные функции 13](#__RefHeading___12)

[ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ АЛГОРИТМА 16](#__RefHeading___31)

[Результат выполнения программы 16](#__RefHeading___14)

[Графики функций initialization и traverse. 16](#__RefHeading___16)

[Результаты исследования 17](#__RefHeading___33)

[ЗАКЛЮЧЕНИЕ 18](#__RefHeading___20)

[ПРИЛОЖЕНИЕ 20](#__RefHeading___24)

# 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:
   1. Если этот многоугольник полностью находится перед плоскостью, содержащей P *,* переместите этот многоугольник в список узлов перед *P.*
   2. Если этот многоугольник полностью находится за плоскостью, содержащей P *,* переместите этот многоугольник в список узлов за *P.*
   3. Если этот многоугольник пересекается плоскостью, содержащей P *,* разделите его на два многоугольника и переместите их в соответствующие списки многоугольников позади и перед *P.*
   4. 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* .

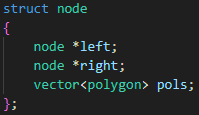
Следующая диаграмма иллюстрирует использование этого алгоритма для преобразования списка линий или многоугольников в дерево BSP. На каждом из восьми шагов описанный выше алгоритм применяется к списку строк и к дереву добавляется один новый узел.

|  |  |  |
| --- | --- | --- |
|  | 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. |  |
| **1.** | 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 . |  |
| **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. |  |

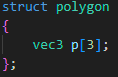
# ХОД РАБОТЫ

## Данные

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



Структура polygon является массивом из 3 векторов.

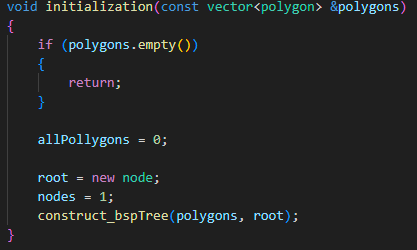


Структура vec3 вмещает в себя 3 координаты x,y,z. Структура plane является разбивающей плоскостью и имеет тип данных vec4(x,y,z,w).

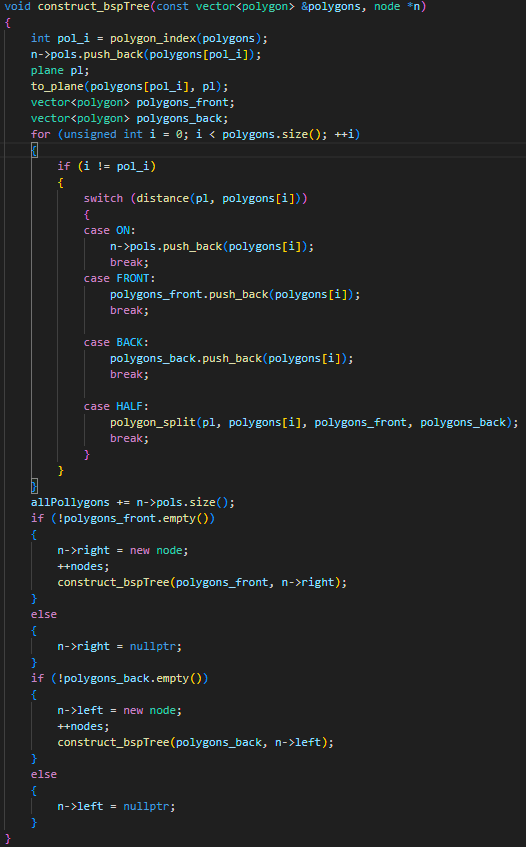
## 

## Функции bsp-tree

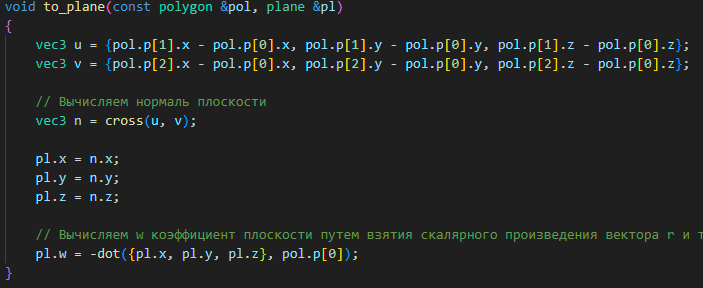
Функция initializtion принимает на вход список полигонов, создает корень дерева и вызывает рекурсивную функцию для построения bsp-tree.

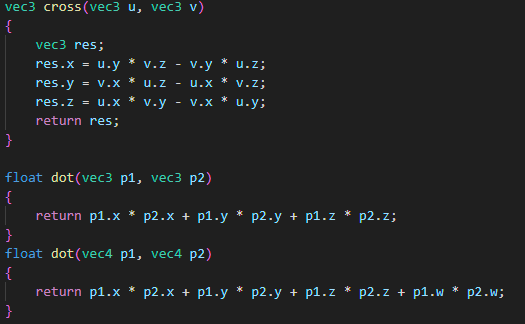


Функция construct\_bspTree принимает на вход список полигонов и структуру node. В начале выбирается случайный полигон из списка переданных, этот полигон добавляется в структуру node. Затем с помощью функции to\_plane этот полигон становится разбивающей плоскостью. В середине функции перебираются все 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.

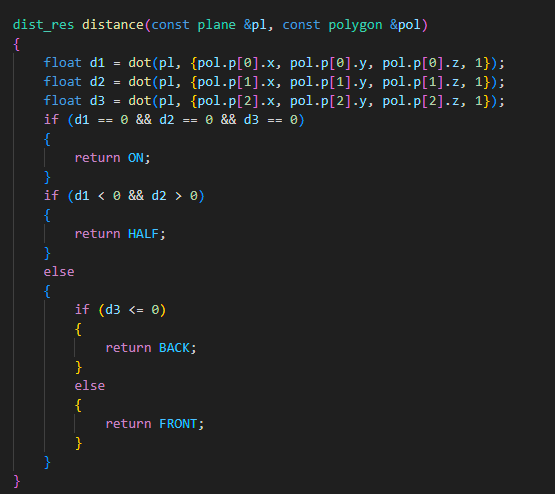


Функция to\_plane принимает на вход полигон и разбивающую плоскость, куда будет записывать результат выполнения программы. В начале вычисляются 2 вектора, которые лежат в полигоне, затем с помощью функции cross, которая находит векторное произведение, находится нормаль плоскости. В конце вычисляется коэффициент плоскости с помощью функции dot, которая вычисляет скалярное произведение векторов.

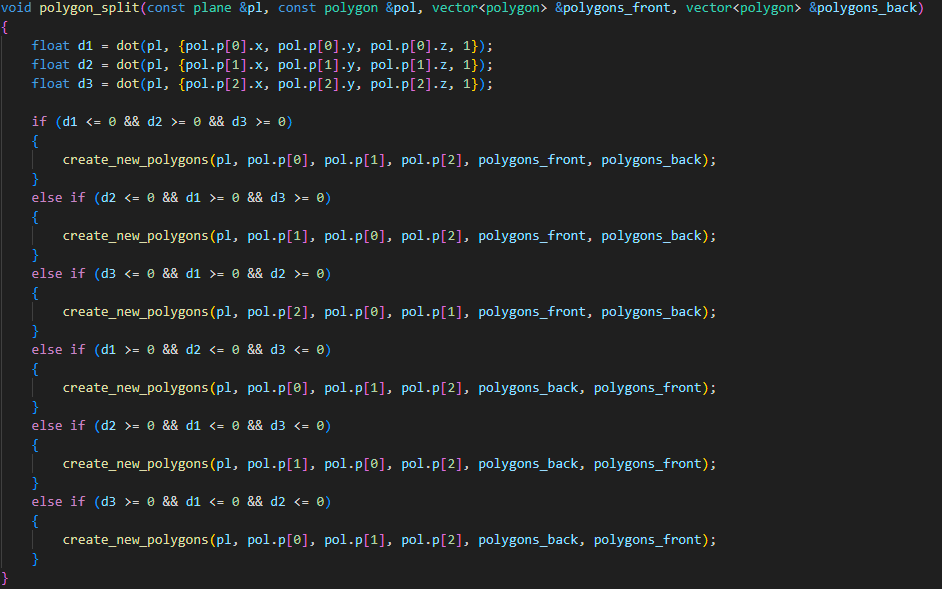




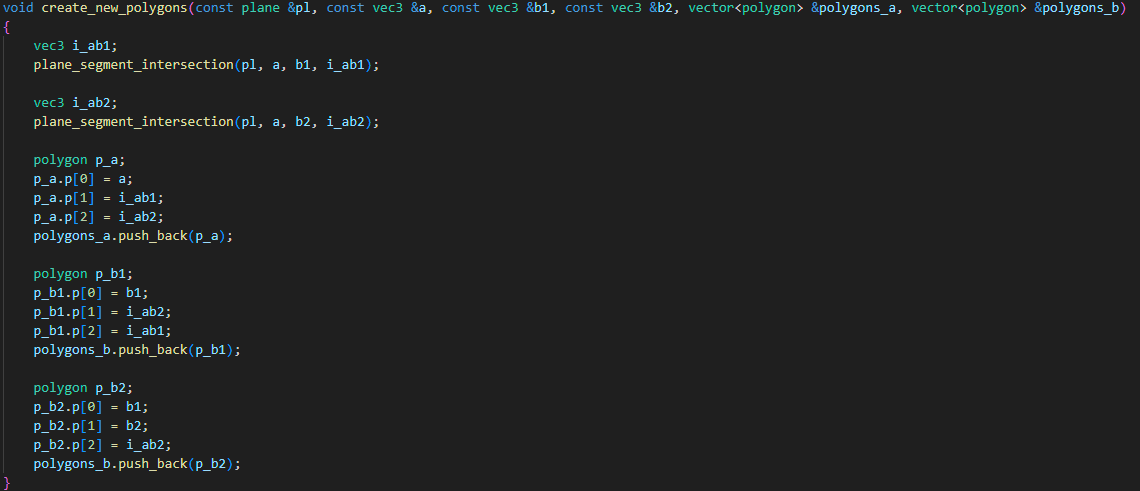
Функция distance принимает на вход полигон и разбивающую плоскость и высчитывает скалярное произведение между каждым вектором полигона и разбивающей плоскостью. Если все 3 точки больше нуля, то полигон находится спереди, если меньше, то сзади, если все три веткора равны 0, то полигон находится в плоскости.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



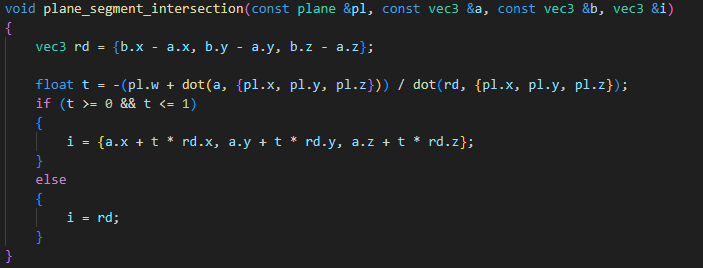
Функция polygon\_split в зависимости от того, какой из векторов разбивается плоскостью, выбирается в каком порядке нужно передавать аргументы в функцию polygon\_split.



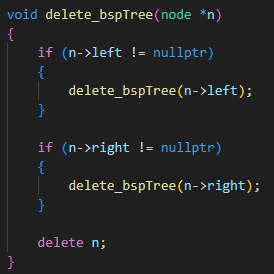
В функция create\_new\_polygons находятся точки пересечения полигона и разбивающей плоскости, создается три полигона и добавляются в списки polygons\_back и polygons\_front.



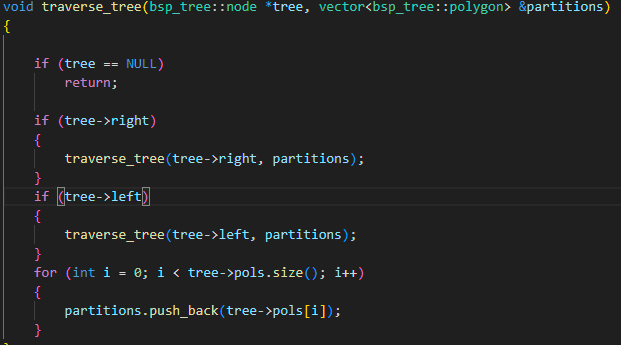
Функция plane\_segment\_intersection принимает на вход вектор a, вектор b, разбивающую плоскость и вектор i, куда будет записываться результат. В начале вычисляется вектор rd, путем вычитания вектора b из вектора a. Затем вычисляется коэффициент t. Если он входит в промежуток [0;1], то точка находится на векторе. Тогда вектор вектору i присваивается сумма a и вектора rd умноженному на t, иначе rd.



Функция delete\_bspTree удаляет bsp-tree, путем рекурсивного обхода дерева и освобождения памяти.

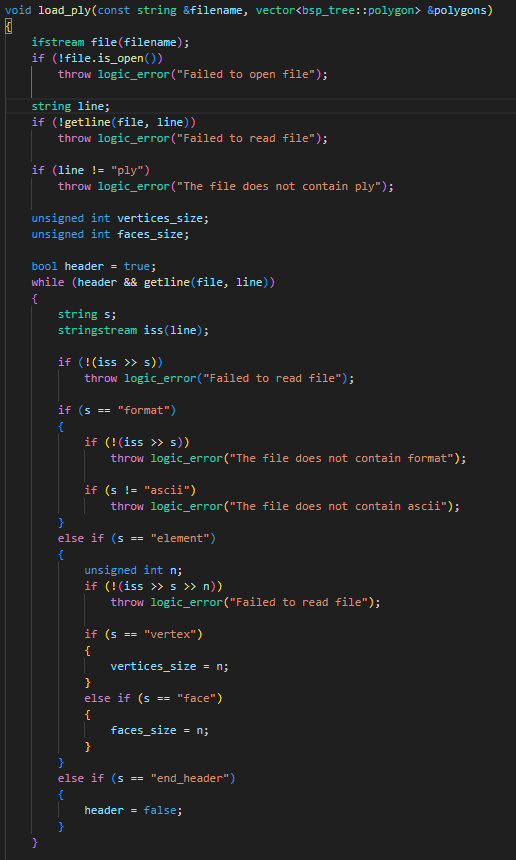


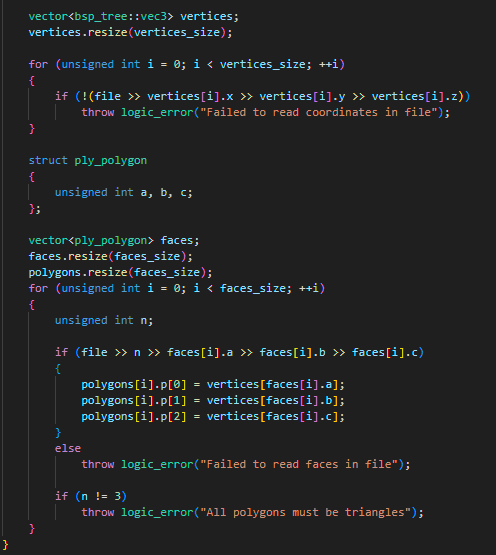
Функция traverse\_tree рекурсивно обходит каждый узел дерева и добавляет в список partitions все полигоны.



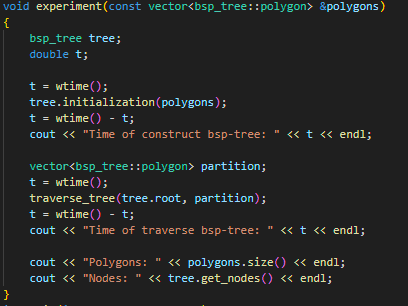
## Дополнительные функции

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.

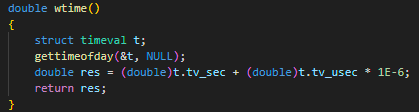




Функция experiment высчитывает время выполнения функций initialization и traverse\_tree.

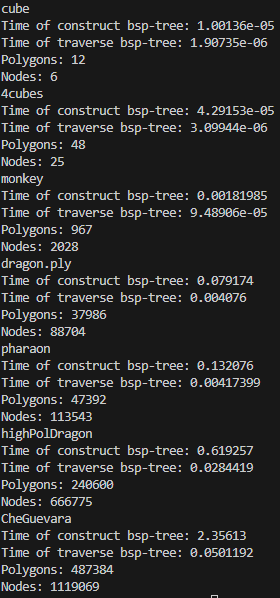


Функция wtime() возвращает нынешнее время в секундах.



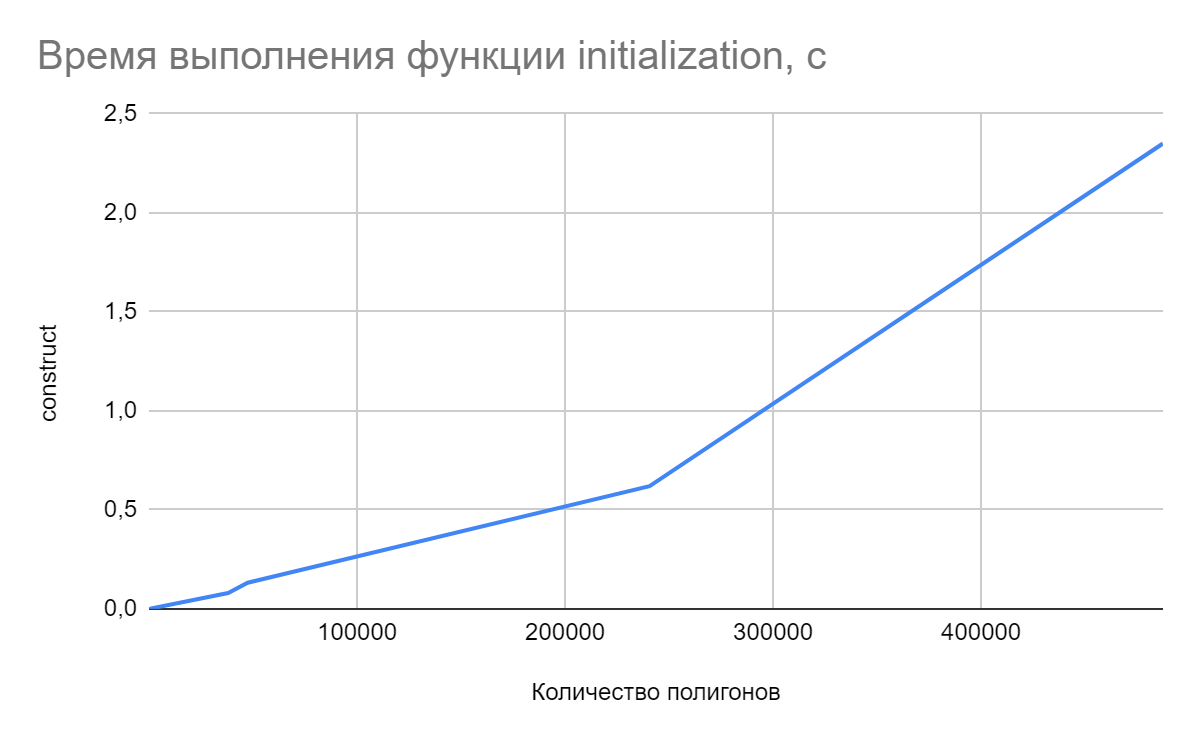
# ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ АЛГОРИТМА

## Результат выполнения программы



По полученным данным составим графики.

## Графики функций initialization и traverse.





## Результаты исследования

Сложность построения двоичного дерева поиска составляет O(nlogn), 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(nlogn), 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 C++. Анализ. Структуры данных.

Сортировка. Поиск. – К.: ДиаСофт, 2001. – 688 с.

1. Скиена С. С. Алгоритмы. Руководство по разработке. – 2-е изд. – СПб: БХВ, 2011 –720 с.
2. Макконнелл Дж. Основы современных алгоритмов. – 2-е изд. – М.: Техносфера, 2004.– 368 с.
3. Миллер Р. Последовательные и параллельные алгоритмы: общий подход. – М.: БИНОМ, 2006. – 406 с.
4. Сегаран Т. Программируем коллективный разум. – М.: Символ-Плюс, 2008. – 368 с.
5. <https://en.wikipedia.org/wiki/Binary_space_partitioning>

# ПРИЛОЖЕНИЕ

Исходный код программы

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 | #include <iostream> #include <cstdlib> #include <fstream> #include <vector> #include <sstream> #include <sys/time.h> **using** **namespace** std; **class** **bsp\_tree** { **public:**  **struct** vec3  {  **float** x;  **float** y;  **float** z;  };  **struct** vec4  {  **float** x;  **float** y;  **float** z;  **float** w;  };  **typedef** vec4 plane;  **struct** polygon  {  vec3 p[**3**];  };  **struct** node  {  node \*left;  node \*right;  vector<polygon> pols; // список всех узлов, находящизся в одной плоскости  };  node \*root;  **enum** dist\_res  {  ON = **0**,  FRONT = **1**,  BACK = **2**,  HALF = **3**  };   **int** nodes;  **int** allPollygons;  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;  }  **int** **polygon\_index**(**const** vector<polygon> &polygons)  {  **return** rand() % polygons.size();  }  **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;  }  }  **void** **create\_new\_polygons**(**const** plane &pl, **const** vec3 &a, **const** vec3 &b1, **const** vec3 &b2, vector<polygon> &polygons\_a, vector<polygon> &polygons\_b)  {  vec3 i\_ab1;  plane\_segment\_intersection(pl, a, b1, i\_ab1);   vec3 i\_ab2;  plane\_segment\_intersection(pl, 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**] = b1;  p\_b2.p[**1**] = b2;  p\_b2.p[**2**] = i\_ab2;  polygons\_b.push\_back(p\_b2);  }  **void** **polygon\_split**(**const** plane &pl, **const** polygon &pol, vector<polygon> &polygons\_front, vector<polygon> &polygons\_back)  {  **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**)  {  create\_new\_polygons(pl, pol.p[**0**], pol.p[**1**], pol.p[**2**], polygons\_front, polygons\_back);  }  **else** **if** (d2 <= **0** && d1 >= **0** && d3 >= **0**)  {  create\_new\_polygons(pl, pol.p[**1**], pol.p[**0**], pol.p[**2**], polygons\_front, polygons\_back);  }  **else** **if** (d3 <= **0** && d1 >= **0** && d2 >= **0**)  {  create\_new\_polygons(pl, pol.p[**2**], pol.p[**0**], pol.p[**1**], polygons\_front, polygons\_back);  }  **else** **if** (d1 >= **0** && d2 <= **0** && d3 <= **0**)  {  create\_new\_polygons(pl, pol.p[**0**], pol.p[**1**], pol.p[**2**], polygons\_back, polygons\_front);  }  **else** **if** (d2 >= **0** && d1 <= **0** && d3 <= **0**)  {  create\_new\_polygons(pl, pol.p[**1**], pol.p[**0**], pol.p[**2**], polygons\_back, polygons\_front);  }  **else** **if** (d3 >= **0** && d1 <= **0** && d2 <= **0**)  {  create\_new\_polygons(pl, pol.p[**0**], pol.p[**1**], pol.p[**2**], polygons\_back, polygons\_front);  }  }  // вычисляем уравнение плоскости на основе многоугольника,  // а функция расстояния определяет положение многоугольника относительно плоскости  **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 точки многоугольника  pl.w = -dot({pl.x, pl.y, pl.z}, pol.p[**0**]);  }   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;  }  }  }  **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**;  }  }  }   allPollygons += 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;  }  }  **void** **initialization**(**const** vector<polygon> &polygons)  {  **if** (polygons.empty())  {  **return**;  }   allPollygons = **0**;   root = **new** node;  nodes = **1**;  construct\_bspTree(polygons, root);  }   ~bsp\_tree()  {  delete\_bspTree(root);  }   **void** delete\_bspTree(node \*n)  {  **if** (n->left != nullptr)  {  delete\_bspTree(n->left);  }   **if** (n->right != nullptr)  {  delete\_bspTree(n->right);  }   **delete** n;  }   **int** get\_nodes()  {  **return** nodes;  }   **int** get\_fragments()  {  **return** allPollygons;  } }; **double** **wtime**() {  **struct** timeval t;  gettimeofday(&t, NULL);  **double** res = (**double**)t.tv\_sec + (**double**)t.tv\_usec \* **1E-6**;  **return** res; } **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");  } } **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]);  } } **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; } **int** **main**(**int** argc, **char** \*\*argv) {  srand(time(NULL));  cout << "cube" << endl;  vector<bsp\_tree::polygon> polygons;  try  {  load\_ply("models/cube.ply", polygons);  }  **catch** (**const** exception &ex)  {  cerr << "The file format must be ply. " << ex.what() << endl;  exit(**1**);  }  experiment(polygons);  polygons.clear();   cout << "4cubes" << endl;  try  {  load\_ply("models/4cubes.ply", polygons);  }  **catch** (**const** exception &ex)  {  cerr << "The file format must be ply. " << ex.what() << endl;  exit(**1**);  }  experiment(polygons);  polygons.clear();   cout << "monkey" << endl;  try  {  load\_ply("models/monkey.ply", polygons);  }  **catch** (**const** exception &ex)  {  cerr << "The file format must be ply. " << ex.what() << endl;  exit(**1**);  }  experiment(polygons);  polygons.clear();   cout << "dragon.ply" << endl; // 37986  try  {  load\_ply("models/dragon.ply", polygons);  }  **catch** (**const** exception &ex)  {  cerr << "The file format must be ply. " << ex.what() << endl;  exit(**1**);  }  experiment(polygons);  polygons.clear();   cout << "pharaon" << endl; // 47392  try  {  load\_ply("models/pharaon.ply", polygons);  }  **catch** (**const** exception &ex)  {  cerr << "The file format must be ply. " << ex.what() << endl;  exit(**1**);  }  experiment(polygons);  polygons.clear();  cout << "highPolDragon" << endl; // 240600  try  {  load\_ply("models/highPolDragon.ply", polygons);  }  **catch** (**const** exception &ex)  {  cerr << "The file format must be ply. " << ex.what() << endl;  exit(**1**);  }  experiment(polygons);  polygons.clear();  cout << "СheGuevara" << endl; // 487384  try  {  load\_ply("models/СheGuevara.ply", polygons);  }  **catch** (**const** exception &ex)  {  cerr << "The file format must be ply. " << ex.what() << endl;  exit(**1**);  }  experiment(polygons);  polygons.clear();  **return** **0**; } |