

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ  
БУРЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Институт математики и информатики

Кафедра прикладной математики

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

3D-визуализация 2D модели  
плана помещения методом бросания лучей

Выполнил : студент 4 курса группы 05230  
Шорников Александр Евгеньевич

Научный руководитель: к.ф.-м.н., ст. преп.  
Трунин Дмитрий Олегович

Научный консультант: асс. каф. ИТ  
Брагин Александр Фёдорович

Улан-Удэ

2017

# Оглавление

Стр.

<b>Введение</b>	<b>3</b>
<b>1 Постановка задачи</b>	<b>5</b>
<b>2 Математическая модель</b>	<b>14</b>
2.1 Способ представления карты . . . . .	14
2.2 Рейкастинг как алгоритм отрисовки . . . . .	18
2.3 BSP-дерево . . . . .	22
2.4 Алгоритм текстурирования . . . . .	29
2.5 Паттерн для интерактивного взаимодействия . . . . .	31
2.6 Поиск оптимального пути . . . . .	33
2.7 Движение сегментов . . . . .	42
<b>3 Реализация</b>	<b>43</b>
3.1 Инструментарий . . . . .	43
3.2 Организация работы над проектом . . . . .	46
3.3 Структура проекта (UML) . . . . .	50
3.4 Реализация на целевых платформах . . . . .	51
3.5 Производительность . . . . .	53
3.6 Внедрение ссылки на сайт ИМИ БГУ . . . . .	58
<b>Заключение</b>	<b>61</b>
<b>Список литературы</b>	<b>63</b>
<b>Приложение 1</b>	<b>68</b>

## ВВЕДЕНИЕ

Интерактивные планы помещений - доступный и современный способ разобраться в незнакомом здании. Сегодня интерактивная карта в отличие от обычной плоской в виде изображения отличается тем, что ее элементами можно управлять. Пользователь, находясь на странице, может свободно перемещаться по карте, находить объекты, схему прохода и просматривать информацию. При необходимости элементы карты могут включать в себя помимо реальных физических объектов (дома, улицы, парки, дороги и т.д.) дополнительно текстовую информацию, видеозаписи и ссылки на сайты.

К таким интерактивным планам предъявляются весьма суровые системные требования: поскольку они предназначены для самых разных платформ, и должны запускаться не только на мощных десктопах, но и на мобильных устройствах, слабых нодах типа Raspberry Pi, встраиваемых системах типа Arduino, а так же крутиться на сайтах. Не на всех из вышеперечисленных платформ может запускаться стандартная для современных 3D-визуализаций библиотека OpenGL(или WebGL в случае сайтов).

Поэтому разработка проекта для 3D визуализации здания по 2D плану помещения весьма актуальна. Вопросы рассмотренные в данной работе требуют крайне серьёзных исследований, ибо за подобными технологиями будущее[40].

**Цель:** создание кроссплатформенного псевдотрёхмерного движка для 3D визуализации здания по 2D плану.

**Задачи:**

–Модификация алгоритма рейкастинга для вещественных координат;

- Разработка математической модели для рейкастового рендерера;
- Изучение и освоение технологии кросс-компиляции *Emscripten*;

**Объектом** исследования является задача построения псевдотрёхмерной картинки.

**Предметом** исследования является изучение основных принципов построения псевдотрёхмерной картинки методом бросания лучей.

# Глава 1

## Постановка задачи

Для визуализации плана помещения существует достаточно много решений. Есть *HTMLMaps* - 2,5D подобный план помещений, работающий на *Joomla* 1.5 поверх *CSS* 3. Его принцип заключается в создании на основе помеченных координат на прямоугольной сетке вида "сверху" и текстур создание плоского вида "сверху" с возможностью показа планов при наведении. Есть множество подобных ему планов для подобного визуализирования, отличающиеся только подробностью и способом отрисовки[43]. Но на данный момент мне не известны методы создания визуализации плана помещения в 3D подобно трёхмерным играм в жанре "шутер". Можно решить данный недостаток полумерами в духе карт помещения на движке игр, например, Counter Strike. Но подобные действия - не выход. И мой проект создания подобного плана помещения призван заполнить образовавшуюся нишу на рынке.

Но к подобному плану будут предъявляться весьма суровые требования - с одной стороны они должны крутиться на любой встраиваемой системе университета - начиная от консолей на входе, и заканчивая зачастую слабыми компьютерами в аудиториях. Кроме того такой план должен быть на сайте института, быть на телефонах, планшетах и нетбуках сотрудников, преподавателей, студентов и абитуриентов. С другой стороны, такой план должен устраивать всех своей графикой, чтобы было сразу же понятно где находится та или иная аудитория, а в идеале должна быть система поиска маршрута в духе "как пройти из одной аудитории в другую кратчайшим маршрутом". Именно поэтому к проекту были суровые требования выполнения всех вышеперечисленных пунктов.

В соответствии с требованиями озвученными выше были рассмотрены несколько вариантов рендеринга трёхмерной картинки. Для начала необходимо было определиться с 3D-моделированием. 3D-моделирование - это процесс создания трёхмерной модели объекта. Задача 3D-моделирования - разработать визуальный объёмный образ желаемого объекта. Графическое изображение трёхмерных объектов отличается тем, что включает построение геометрической проекции трёхмерной модели сцены на плоскость с помощью специализированных программ.

Трёхмерная графика обычно имеет дело с виртуальным, воображаемым трёхмерным пространством, которое отображается на плоской, двухмерной поверхности дисплея или листа бумаги. В настоящее время известно несколько способов отображения трёхмерной информации в объёмном виде, хотя большинство из них представляет объёмные характеристики весьма условно, поскольку работают со стереоизображением[5]. Однако и 3D-дисплеи по-прежнему не позволяют создавать полноценной физической, осязаемой копии математической модели, создаваемой методами трёхмерной графики.

Для получения трёхмерного изображения на плоскости требуются следующие шаги:

- моделирование - создание трёхмерной математической модели сцены и объектов в ней;
- текстурирование - назначение поверхностям моделей растровых или процедурных текстур (подразумевает также настройку свойств материалов - прозрачность, отражения, шероховатость и пр.);
- освещение - установка и настройка источников света;
- анимация (в некоторых случаях) - приданье движения объектам;
- динамическая симуляция (в некоторых случаях) - автоматический расчёт взаимодействия частиц, твёрдых/мягких тел и пр. с моделируемыми силами гравитации, ветра, выталкивания и др., а также друг с другом;

- рендеринг (визуализация) - построение проекции в соответствии с выбранной физической моделью;
- композитинг (компоновка) - доработка изображения;
- вывод полученного изображения на устройство вывода - дисплей или специальный принтер[42].

Моделирование сцены (виртуального пространства моделирования) включает в себя несколько категорий объектов:

- Геометрия (построенная с помощью различных техник (напр., создание полигональной сетки) модель, например, здание);
- Материалы (информация о визуальных свойствах модели, например, цвет стен и отражающая/преломляющая способность окон);
- Источники света (настройки направления, мощности, спектра освещения);
- Виртуальные камеры (выбор точки и угла построения проекции);
- Силы и воздействия (настройки динамических искажений объектов, применяется в основном в анимации);
- Дополнительные эффекты (объекты, имитирующие атмосферные явления: свет в тумане, облака, пламя и пр.).

Задача трёхмерного моделирования - описать эти объекты и разместить их в сцене с помощью геометрических преобразований в соответствии с требованиями к будущему изображению.

Назначение материалов: для сенсора реальной фотокамеры материалы объектов реального мира отличаются по признаку того, как они отражают, пропускают и рассеивают свет; виртуальным материалам задается соответствие свойств реальных материалов - прозрачность, отражения, рассеивания света, шероховатость, рельеф и пр.

Текстурирование подразумевает проецирование растровых или процедурных текстур на поверхности трёхмерного объекта в соответствии

с картой UV-координат, где каждой вершине объекта ставится в соответствие определённая координата на двухмерном пространстве текстуры. освещение заключается в создании, направлении и настройке виртуальных источников света. При этом в виртуальном мире источники света могут иметь негативную интенсивность, отбирая свет из зоны своего "отрицательного освещения". Как правило, пакеты 3D-графики предоставляют следующие типы источников освещения:

- Omni light (Point light) - всенаправленный;
- Spot light - конический ( прожектор), источник расходящихся лучей;
- Directional light - источник параллельных лучей;
- Area light (Plane light) - световой портал, излучающий свет из плоскости;
- Photometric - источники света, моделируемые по параметрам яркости свечения в физически измеримых единицах, с заданной температурой накала.

Существуют также другие типы источников света, отличающиеся по своему функциональному назначению в разных программах трёхмерной графики и визуализации. Некоторые пакеты предоставляют возможности создавать источники объемного свечения (Sphere light) или объемного освещения (Volume light), в пределах строго заданного объёма. Некоторые предоставляют возможность использовать геометрические объекты произвольной формы.

На этапе рендеринга 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:

- Z-буфер (используется в OpenGL и DirectX 10);
- Ray casting ("метод бросания лучей упрощенный алгоритм обратной трассировки лучей) - он же Сканлайн (scanline) - расчёт цвета каждой точки картинки построением луча из точки зрения наблюдателя через воображаемое отверстие в экране на месте этого пикселя "в сцену" до пересечения с первой поверхностью. Цвет пикселя будет таким же, как цвет этой поверхности (иногда с учётом освещения и т. д.);
- Трассировка лучей (рейтреисинг, англ. raytracing) - то же, что и сканлайн, но цвет пикселя 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;
- Глобальное освещение (англ. global illumination, radiosity) - расчёт взаимодействия поверхностей и сред в видимом спектре излучения с помощью интегральных уравнений[5].

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 самосвещения) света, но и

сглаженные тени. По этой причине чаще понятие Raycasting относится к обратной трассировке лучей, а Raytracing - к прямой.

Вследствие большого объёма 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.

Сравнив все методы рендеринга представленные выше, было принято решение использовать метод "бросания лучей" ввиду его удобства для поставленных целей. Метод бросания лучей (англ. Raycasting, рейкастинг) это один из методов рендеринга в компьютерной графике, при котором изображение строится на основе замеров пересечения лучей с визуализируемой поверхностью. Грубо говоря, из точки обзора бросается множество лучей, каждый из которых пересекает какой-нибудь отрезок на карте или вообще ничего не пересекает. На экране эти точки пересечения отображаются как вертикальные отрезки, рассчитанные от расстояния до визуализируемой поверхности. Если точки пересечения нет, то длина вертикального отрезка равна нулю, то есть вырисовывается горизонт.

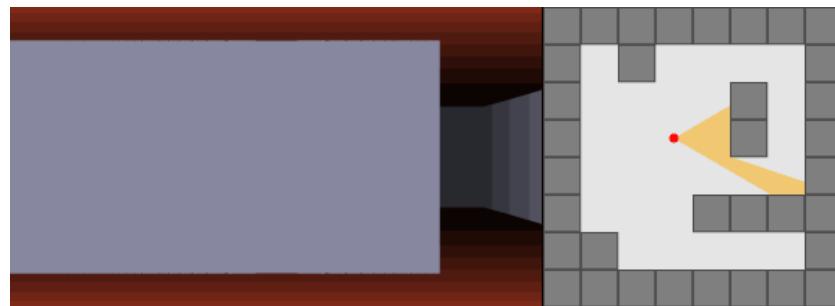


Рис. 1.1: Принцип работы метода "бросания лучей"

Данный метод, являясь одним из простейших для трёхмерной отрисовки, имеет ряд свойств согласующихся с задачами проекта: малую вычислительную сложность и простую реализацию без использования готовых библиотек трёхмерной графики. Плюсами данного метода так же является то, что метод работает с плоской двумерной моделью помещения. То есть для визуализации помещения не надо строить полностью трёхмерную карту для рендеринга - достаточно дать плоскую карту.

В работах по компьютерной графике метод бросания лучей впервые был рассмотрен для отрисовки моделей конструктивной блочной геометрии в публикациях Скотта Ди Рота в 1982 году.

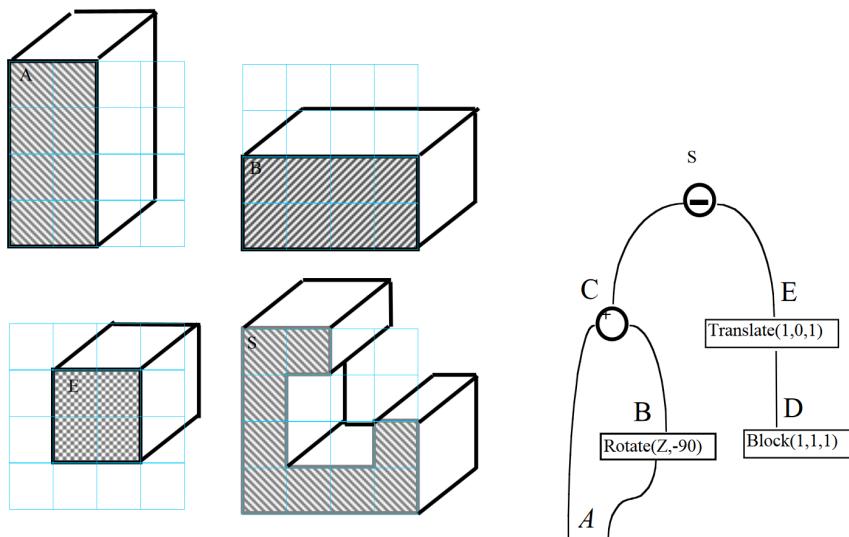


Рис. 1.2: Roth, Scott D. (February 1982), "Ray Casting for Modeling Solids Computer Graphics and Image Processing T. 18: 109–144

После выхода игры *Wolfenstein 3D* в 1992 году технология рейкастинга была широко использована для компьютерных игр.

*Wolfenstein 3D engine* — псевдотрёхмерный игровой движок, разработанный для игры *Wolfenstein 3D*, вышедшей 5 мая 1992 года. Дви-



Рис. 1.3: Скриншот игры *Wolfenstein 3D*

движок разрабатывался преимущественно Джоном Кармаком, главным программистом компании id Software. Движок *Wolfenstein 3D engine* реализует VGA графику (рейтранслитинговая), звук (WAV и IMF), физику и управление. Написан на Си и ассемблере x86[48].

Возможности компьютеров с процессором Intel 80286, которые были тогда распространены, были крайне ограничены. Для рендеринга изображения при помощи рейкастинга в игре *Wolfenstein 3D* движок игры был специальным образом оптимизирован для слабых вычислительных машин[49]. В результате чего все стены в этой игре имеют одинаковую высоту, и представляют собой взаимно перпендикулярные ячейки 2D сетей, как видно на рисунке:

К сожалению, графические движки на основе технологии рейкастинга слишком слабы, что бы реализовать такие элементы, как лестницы или прыжки с разницей высот. Независимые графические объекты, свободно перемещающиеся по экрану (противники, внутреигровые объекты и прочее) представляют собой не трёхмерные объекты, а двухмерные



Рис. 1.4: Скриншот игры *Редактор уровней Wolfenstein 3D*

картинки-спрайты. Более поздние игры на этой технологии, такие как *Doom* и *Duke Nukem 3D* были гораздо более продвинутыми, и позволяли создавать наклонные стены (поверхности), разницу высот, текстурированные полы и потолки, прозрачные стены и т.д., но в них были использованы разные технологии помимо рейткастинга[50]. Поэтому в рамках нашего проекта необходимо было видоизменить алгоритм рейткастинга таким образом, что бы уйти от ограничений старых технологий и внести новизну в довольно древнюю технологию. А именно требуется преобразовать алгоритм рейткастинга для работы в вещественных координатах, создать новую структуру данных для задания карт, возможность динамического движения элементов карты "на лету" и многое другое[43].

# Глава 2

## Математическая модель

### 2.1. Способ представления карты

Поскольку метод бросания лучей работает с двумерной моделью помещения, необходимо было придумать способ представления карты для его адекватного считывания подпрограммой считывания уровня и представления функцией обработки. Для классического рейкастинга уровень представляет собой двумерный массив, где значение каждого элемента массива является квадратом мира. Если значение ячейки равно 0, то квадрат оказывается пустым, и через него можно пройти. Если же значение больше 0, квадрат представляет собой стену определённого цвета или текстуры.

Для нашего проекта мы изменили алгоритм метода бросания лучей для работы в вещественных координатах. Благодаря этому мы избавились от ограничений старого задания уровней. Все графические примитивы на карте мы стали обозначать отрезками задаваемыми двумя вещественными точками, благодаря чему размер карты стал ограничиваться только ресурсами платформ, а так же увеличилась точность самого рендеринга карты.

Для того что бы представить карту каждого этажа в уровне нашего движка необходимо составить модель этого этажа. Как уже говорилось выше, каждый из графических примитивов в карте задаётся отрезком. И для представления карты внутри программы мы можем создать  $C +$ -класс, описывающий тип данных отрезков в качестве совокупности координат  $(x_1, y_1) - (x_2, y_2)$ , а так же флага принадлежности текстуры к данному отрезку. Таким образом мы полностью описываем программно

уровень карты.

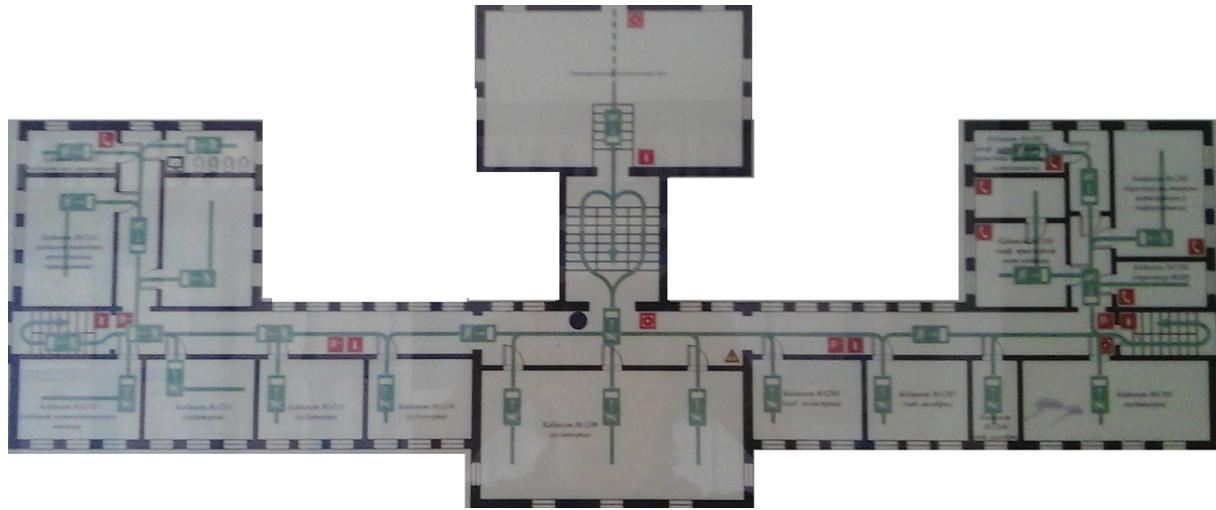


Рис. 2.1: Обработанное изображение 2 этажа 1 корпуса БГУ

Теперь для задания необходимого изображения этажа в карту можно представить её в векторном формате *SVG*. *SVG* (от англ. Scalable Vector Graphics — масштабируемая векторная графика) — язык разметки масштабируемой векторной графики, предназначенный для описания двумерной векторной и смешанной векторно/растровой графики в формате *XML*. Выбор пал именно на этот формат, поскольку внутрене формат *SVG* представляет собой *XML*-документ, содержащий помимо встроенной разметки координаты точек котрэзков. Необходимо было только написать парсер для фильтрации полезной информации из *SVG*. Пример внутренего содержания *SVG*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 2017-04-04 23:45:02 Generated by QCAD SVG Exporter -->
<svg width="460mm" height="191mm" viewBox="0 0 460 191"
version="1.1" xmlns="http://www.w3.org/2000/svg"
style="stroke-linecap:round; stroke-linejoin:round; fill:none">
<g transform="scale(1,-1)">
```

```

<!-- Line -->
<path d="M208,-65 L208,-112 "
style="stroke:#000000;stroke-width:0.25;" />
<!-- Line -->
<path d="M212,-112 L212,-65 "
style="stroke:#000000;stroke-width:0.25;" />
<!-- Line -->
<path d="M180,-2 L278,-2 "
style="stroke:#000000;stroke-width:0.25;" />
<!-- Line -->
<path d="M278,-2 L278,-61 "
style="stroke:#000000;stroke-width:0.25;" />
<!-- Line -->
<path d="M236,-61 L236,-65 "
style="stroke:#000000;stroke-width:0.25;" />
</g>
</svg>

```

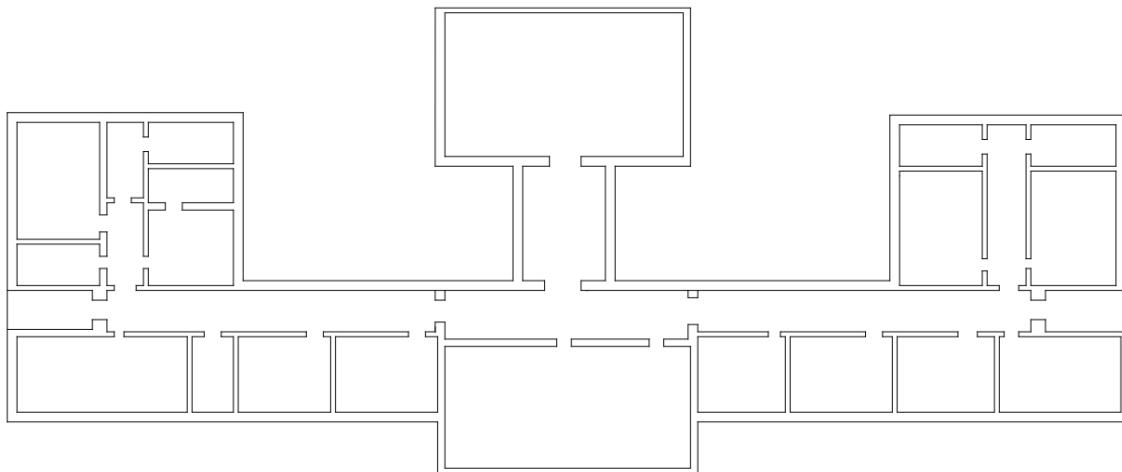


Рис. 2.2: Векторное представление изображения карты

В итоге, общая задача оцифрования уровня карты сводится к сканированию или простому фотографированию пожарного плана помеще-

ния, перевод данного изображения в векторный формат *SVG*, редактирование получившегося изображения при необходимости и "скрмливание" получившегося файла программе считывания уровня. Получение векторное изображение представляет собой уже размеченный *xml*-документ, который подпрограмма считывания уровня просто парсит, вычленяя необходимую информацию в виде координат точек отрезков[45].

Для удобства редактирования готовых *SVG* - файлов мы освоили систему автоматизированного проектирования(САПР, англ. CAD) *QCad*. *QCad* – 2-мерная САПР с открытым исходным кодом, предназначенная для создания чертежей. Работает под операционными системами *Windows*, *Mac OS X* и на *\*nix* системах. *QCad* предоставляет различные инструменты для черчения. Многие концепции интерфейса и приемы работы схожи с *AutoCAD*. *QCad* использует формат *DXF* для сохранения и импорта чертежей по умолчанию, однако в профессиональной версии имеет также поддержку *DWG* и импорт/экспорт в *SVG*.

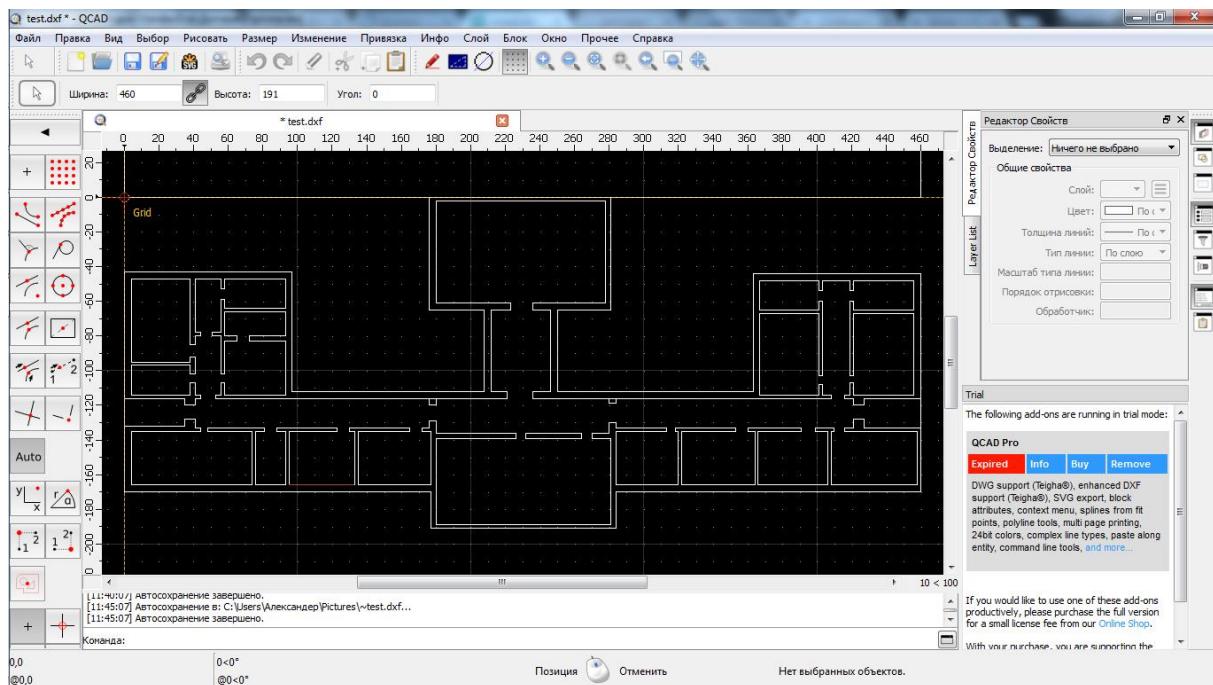


Рис. 2.3: План 2 этажа ИМИ БГУ в QCad

## 2.2. Рейкастинг как алгоритм отрисовки

Итак, в общем случае у нас на карте задан уровень в виде множества отрезков, а также задана точка, интерпретирующая положение камеры на карте. Также задан угол обзора, показывающий какие сегменты попали в наблюдение камеры.

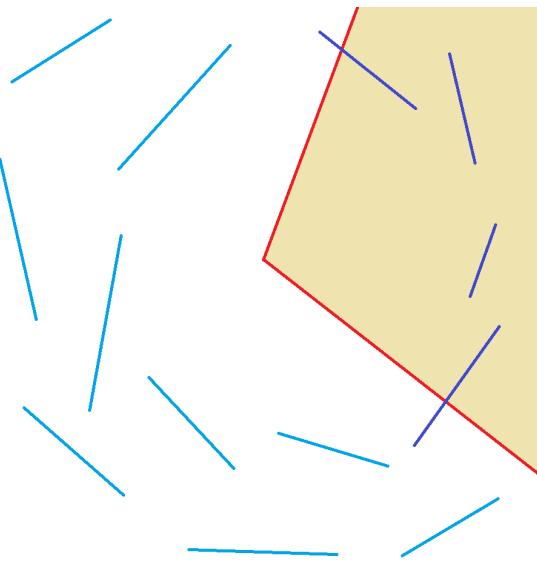


Рис. 2.4: Общий случай описания карты

Вектор описывающий положение камеры назовём  $\overrightarrow{pos}$ , а вектором  $\overrightarrow{dir}$  назовем направление угла обзора. Длина этого вектора не будет влиять на угол обзора.

При помощи оценивания знака векторного произведения определяется принадлежность сегмента к углу обзора. Из местоположения наблюдателя (вектор  $\overrightarrow{pos}$ ) через каждую точку отрезка экрана, представляющую собой одну колонку пикселей на мониторе, проводятся лучи. Их количество равно горизонтальному разрешению экрана. Расстояние между точками на отрезке равно полутора длинам вектора  $\overrightarrow{dir}$  деленное на го-

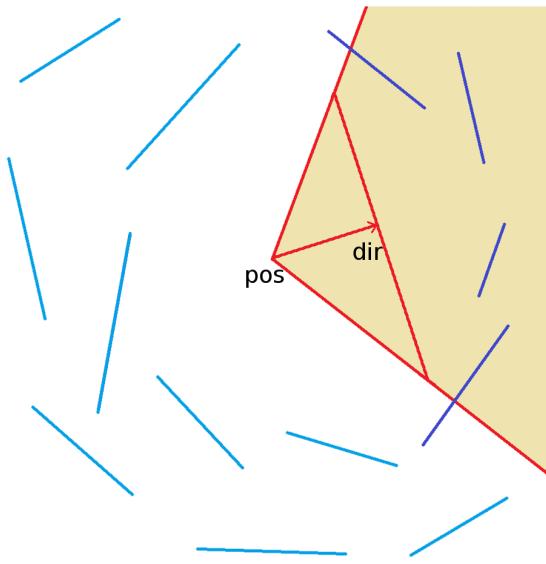


Рис. 2.5: Задание векторов

ризонталь разрешения экрана.

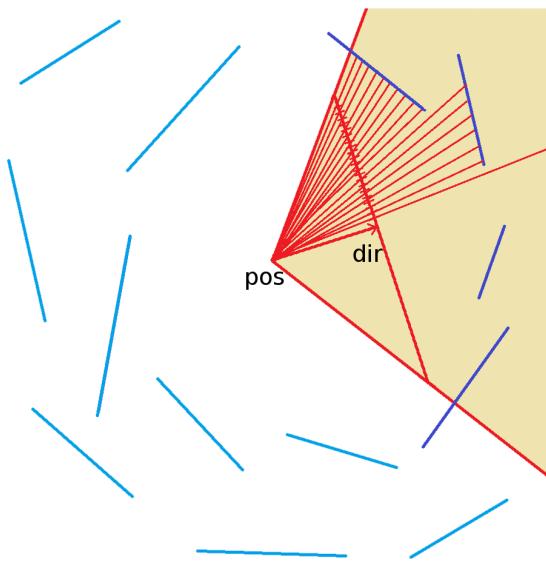


Рис. 2.6: Бросание лучей

Далее для каждой точки рассчитывается присутствие пересечения луча с сегментами по формуле Краммера и рассчитывается расстояние по

лучу до препятствия. В соответствии с полученным расстоянием отображаются линии которые формируют стены. Длина линии обратно пропорциональна найденному расстоянию. Т.е. чем дальше от нас объект, тем он меньше. Эти линии формируют стены[45].

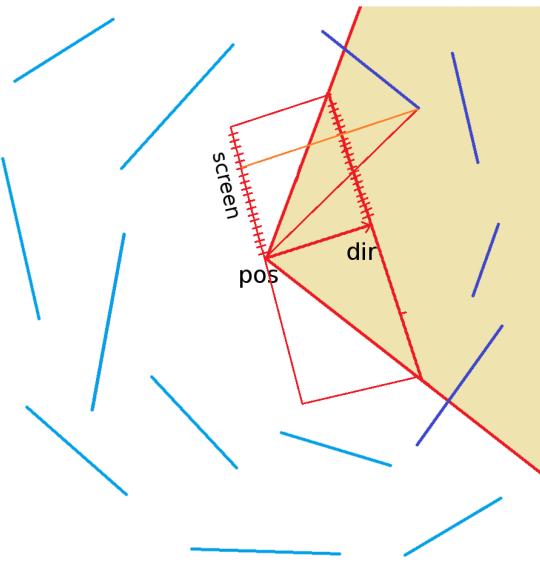


Рис. 2.7: Формирование изображения

К сожалению, при бросании луча из одной точки (точки где находится наблюдатель) возникает так называемый “эффект рыбьего глаза”, или широкоугольного объектива. Это происходит потому что расстояние пройденное по лучу, когда один конец луча неподвижен, а другой скользит вдоль прямой линии, изменяется по квадратичному закону относительно расстояния пройденного вдоль экрана. В результате этого границы стен на изображении описываются кривой второго порядка. Если мы хотим избежать этого эффекта, то нам нужно добиться того, чтобы расстояние пройденной лучом изменилось линейно от расстояния пройденного вдоль экрана. Для этого достаточно пускать лучи перпендикулярно экрану. Этого можно добиться минимальными изменениями в алгоритме, всего лишь изменив параметры передаваемые подпрограм-

ме по расчету расстояния - всего лишь умножив получение расстояние до пересечения на косинус угла между лучом и логическим экраном[36].

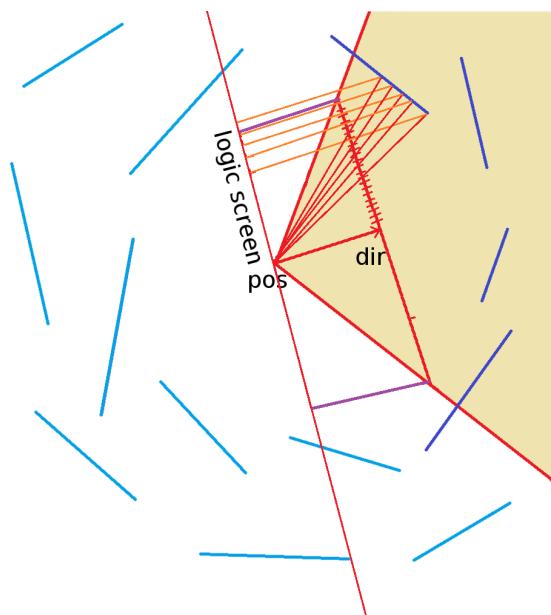


Рис. 2.8: Избавление от "эффекта рыбьего глаза"

### 2.3. BSP-дерево

После реализации алгоритма рейкастинга картинка была крайне нестабильной. Дело в том, что в угол обзора попадало разное количество обрабатываемых сегментов, и количество кадров в секунду было различным для разных мест уровня. Для решения этой проблемы было принято решение использовать оптимизирующий алгоритм, и одним из вариантов была реализация BSP-дерева[33]. Алгоритм BSP-дерева, или алгоритм двоичного разбиения пространства(англ. binary space partitioning) - это метод рекурсивного разбиения евклидова пространства на выпуклые множества и гиперплоскости. BSP-дерево используется для эффективного выполнения следующих операций в трёхмерной компьютерной графике:

- Сортировка визуальных объектов в порядке удаления от наблюдателя;
- Обнаружение столкновений.

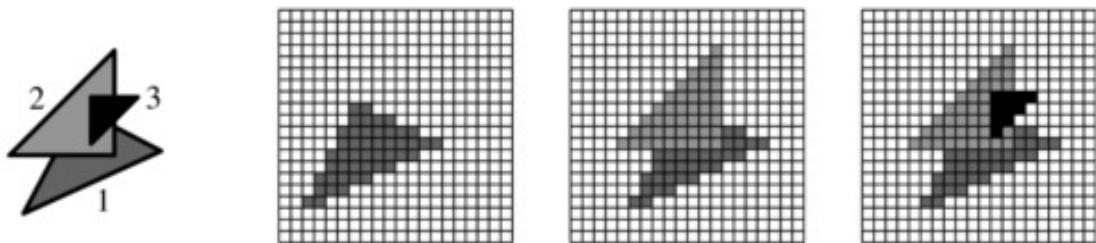
**Задача:** Данна сцена с 2D или 3D объектами и наблюдатель, который смотрит на сцену из своей точки обзора. Нужно отрисовать на сцене видимые наблюдателю части объектов.

#### Алгоритм z-буфера (z-buffer algorithm)

Для удаления невидимых частей объектов существует простой, но длительный метод - алгоритм z-буфера. В направлении просмотра проводится ось z-координат, затем определяется, какие пиксели покрывают проекции объектов. Алгоритм хранит информацию об уже обработанных объектах в двух буферах: буфере кадра и z-буфере. В буфере кадра для каждого пикселя хранится информация о цвете объекта, отображаемого им на данный момент. В z-буфере для каждого пикселя хранится z-координата видимого на данный момент объекта, точнее, в нем хранится z-координату точки такого объекта. Предположим, что мы выбрали пиксель и преобразовываем объект. Если z-координата объекта в

этом пикселе меньше, чем z-координата, хранимая в z-буфере, тогда новый объект лежит перед видимым на данный момент. Тогда запишем цвет нового объекта в буфер кадра, а его координату - в z-буфер. Если z-координата объекта в этом пикселе больше, чем z-координата, хранимая в z-буфере, то новый объект не видим, и буфера останутся без изменений. Алгоритм z-буфера легко реализовать, и он быстро работает. Поэтому именно этот метод используют чаще всего, но у него есть свой недостаток: для хранения z-буфера требуется большое количество памяти, кроме того, требуется дополнительная проверка каждого пикселя, покрываемого объектом[32].

### Алгоритм художника (painter's algorithm)



Алгоритм художника избегает дополнительных затрат памяти, изначально сортируя объекты по расстоянию от них до точки обзора. Тогда объекты проверяются в так называемом порядке глубины, начиная от самого дальнего. В таком случае при рассмотрении объекта уже не нужна проверка его z-координаты, мы всегда пишем цвет в буфер кадра. Значения, хранимые в буфере ранее, просто перезаписываются.

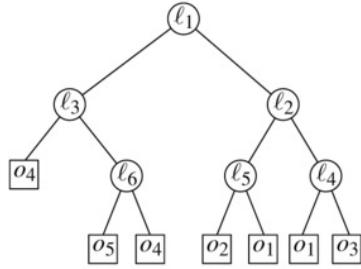
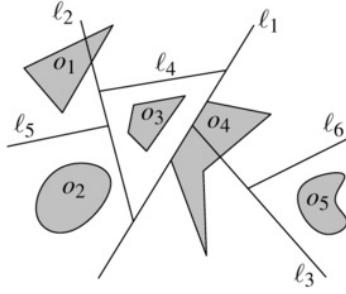
Чтобы успешно применять данный метод, нужно уметь быстро сортировать объекты. К сожалению, это не всегда просто. Кроме того, порядок глубины не всегда существует: отношение "перед" может содержать циклы. Когда такое циклическое перекрытие происходит, объекты не могут быть корректно отсортированы. В таком случае мы должны разорвать



циклы, разбив один или более объектов на части. Определение, какие объекты нужно разбить и где, затем сортировка их фрагментов - дорогой процесс, так как порядок зависит от положения точки обзора, и мы должны пересчитывать все при каждом ее смещении. Чтобы использовать этот алгоритм в реальной жизни, например, в симуляторе полета, мы должны предпосчитать сцену так, чтобы можно было быстро найти корректный порядок отображения объектов для любой точки обзора. Данную задачу можно элегантно решить при помощи техники двоичного разбиения пространства (англ. binary space partitioning, BSP).

Чтобы понять, что из себя представляет двоичное разбиение пространства, рассмотрим рисунок. На нем показано двоичное разбиение множества объектов на плоскости и дерево, которое этому разбиению соответствует. В двумерном случае BSP строится с помощью рекурсивного разбиения плоскости прямыми. В данном примере это происходит так: сначала проводим прямую  $l_1$ , разбивая полу平面 выше  $l_1$  прямой  $l_2$ , а ниже - прямой  $l_3$  и так далее.

Прямые разбивают на части не только плоскость, но и объекты, расположенные на ней. Разбиение продолжается до тех пор, пока внутри каждой грани плоскости окажется не более одного фрагмента объекта. Этот процесс можно представить с помощью двоичного дерева. Каждый лист дерева соответствует грани разбиения, в нем хранится фрагмент объекта, находящийся внутри этой грани. Каждый узел дерева



соответствует разбивающей прямой, которая хранится в этом узле.

**Определение.** *BSP-дерево (англ. binary space partition tree) - дерево, отвечающее заданному двоичному разбиению пространства.*

Рассмотрим гиперплоскость  $h : a_1 \cdot x_1 + a_2 \cdot x_2 + \dots + a_d \cdot x_d + a_{d+1} = 0$ . Пусть  $h^+$  - положительное полупространство, а  $h^-$  - отрицательное:  

$$h^+ = \{(x_1, x_2, \dots, x_d) \mid a_1 \cdot x_1 + a_2 \cdot x_2 + \dots + a_d \cdot x_d + a_{d+1} > 0\}$$
  

$$h^- = \{(x_1, x_2, \dots, x_d) \mid a_1 \cdot x_1 + a_2 \cdot x_2 + \dots + a_d \cdot x_d + a_{d+1} < 0\}$$

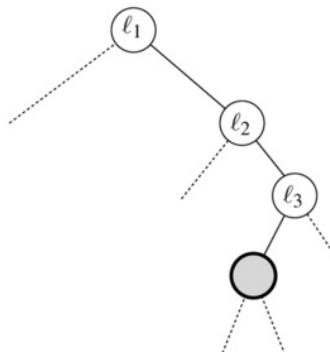
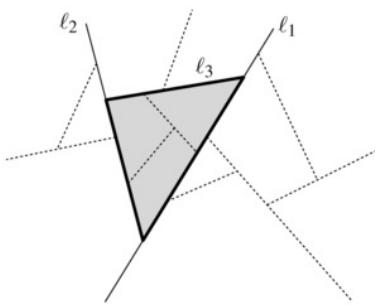
Пусть  $S$  - множество объектов, для которого мы строим разбиение в  $d$ -мерном пространстве. Пусть  $v$  - какая-то вершина дерева, тогда обозначим за  $S(v)$  множество объектов (возможно пустое), хранимых в этой вершине. BSP-дерево  $T$  для этого множества объектов обладает следующими свойствами:

если  $|S| \leq 1$ , то  $T$  - лист. Фрагмент объекта в  $S$ , если он существует, хранится в этом листе. если  $|S| > 1$ , то в корне дерева  $v$  хранится гиперплоскость  $h_v$  и множество  $S(v)$  объектов, которые полностью содержатся в  $h_v$ .

–левый ребенок  $v$  является корнем BSP-дерева  $T^-$  на множестве объектов  $S^- = \{h_v^- \cap s | s \in S\}$ ;

–правый ребенок  $v$  является корнем BSP-дерева  $T^+$  на множестве объектов  $S^+ = \{h_v^+ \cap s | s \in S\}$ .

Размер BSP-дерева равен суммарному размеру множеств во всех узлах. Другими словами, размер BSP-дерева – число фрагментов, на которые были разбиты объекты. Так как BSP-дерево не содержит бесполезные прямые (прямые, которые разбивают пустую грань), то количество узлов пропорционально размеру дерева[?].



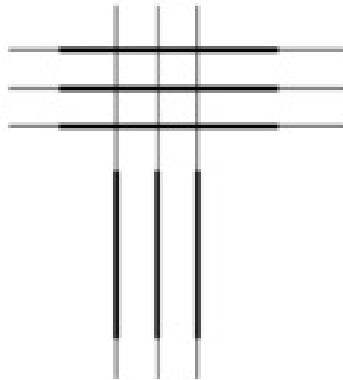
Листья BSP-дерева соответствуют граням, то есть мы можем каждой вершине  $v$  сопоставить полигональную область на плоскости, которая определяется как пересечение полуплоскостей  $h_\mu^\diamondsuit$ , где  $\mu$  – предок  $v$ , и

$$\diamondsuit = \begin{cases} - & \text{if } v \text{ – left child} \\ + & \text{if } v \text{ – right child} \end{cases}$$

Корню дерева соответствует все пространство. Таким образом, серая область на рисунке соответствует региону  $l_1^+ \cap l_2^+ \cap l_3^-$ . При построении BSP-дерева могут использоваться любые разбивающие гиперплоскости. В целях упрощения вычислений может быть удобно ограничить множество доступных разбивающих гиперплоскостей. обычно используют авто-разбиения.

**Определение.** В двумерном случае для множества отрезков разбиение, в котором используются разбивающие прямые, проходящие через один из данных отрезков, называется авто-разбивающим (англ. *auto-partition*).

В трёхмерном случае авто-разбиение использует плоскости, которые содержат грани многогранников.



Как видно из рисунка, размер авто-разбивающего дерева может быть не минимальным. Возможен случай, когда размер BSP-дерева может составлять  $\mathcal{O}(n^2)$ , где  $n = |S|$ .

### BSP-деревья и алгоритм художника

Предположим, что мы построили BSP-дерево  $T$  для множества объектов  $S$  в трехмерном пространстве. Как нам следует использовать его, чтобы получить порядок глубины для алгоритма художника. Пусть  $p_{view}$  - точка обзора, и она лежит над разбивающей плоскостью, хранимой в корне

$T$ . Тогда ни один из объектов, лежащих под этой плоскостью, не может перекрыть ни один из объектов, лежащих выше нее. Таким образом, мы можем безопасно отрисовать фрагменты объектов из поддерева  $T^-$  до отрисовки объектов из поддерева  $T^+$ . Порядок фрагментов объектов в поддеревьях определяется таким же способом.

Заметим, что мы не рисуем объекты из  $S(v)$ , когда  $p_{view}$  лежит на разбивающей плоскости  $h_v$ , потому что они являются плоскими двумерными полигонами. Эффективность данного алгоритма, как и любого другого алгоритма для BSP-деревьев, зависит от размера BSP-дерева. То есть необходимо выбирать разбивающие плоскости таким образом, чтобы фрагментация объектов была минимальной. BSP-деревья интересны тем, что позволяют достичь быстрой реализации удаления скрытых поверхностей для отрисовки сцены (будь то симулятор полёта или персонаж в игре, осматривающий окружающий мир). Так как скорость - главная цель, следует упростить вид объектов рассматриваемого пейзажа, поэтому далее будем считать, что в 3D мы работаем только с многоугольниками, грани которых уже триангулированы. Таким образом множество  $S$  в трёхмерном пространстве будет состоять только из треугольников[11].

## 2.4. Алгоритм текстурирования

При нанесении текстур алгоритм рейкастинга работает практически такая же, но в конце необходимо произвести несколько дополнительных вычислений текстуры; кроме того, петля в направлении у должна пройти через каждый пиксель для определения того, какой тексель (пиксель текстуры) данной текстуры необходимо использовать в этот раз.

Вертикальные полосы на этот раз нельзя нарисовать с помощью вертикальной командной линии. Вместо этого, каждый пиксель необходимо рисовать отдельно. Лучший способ - использовать в этот раз карту 2D в качестве буфера экрана, и сразу же копировать ее на экран.

Конечно же, нам необходим собственно, сам набор текстур, и поскольку функция *makeColumn*, которая наносит текстуры, работает с отдельными целыми значениями для цветов (вместо 3-х отдельных байтов для R, G и B), текстуры хранятся также в формате *RGB*.

Ширина и высота экрана для текстур определяется в самом начале, поскольку нам нужна одна и та же величина для функции экрана и для создания буфера экрана. Кроме того, новыми являются текстура ширины и высоты, определяемые здесь. Очевидно, это - ширина и высота в текселях (элементах) текстур.

Буфер экрана и наборы текстур - это набор динамических массивов-векторов *STD* (*std :: vectors*). Каждая из текстур обладает определенными показателями ширины и высоты (в пикселях). В придуманномами механизме рейкастинга текстуры могут иметь неограниченный размер, и ограничиваются только весом, ограничением времени и скоростью взаимодействия с ними. Наиболее оптимальным размером текстур оказался 512x512 пикселей, поскольку такое разрешение даёт достаточное

качество картинки без замедления FPS(частоты смены кадров).

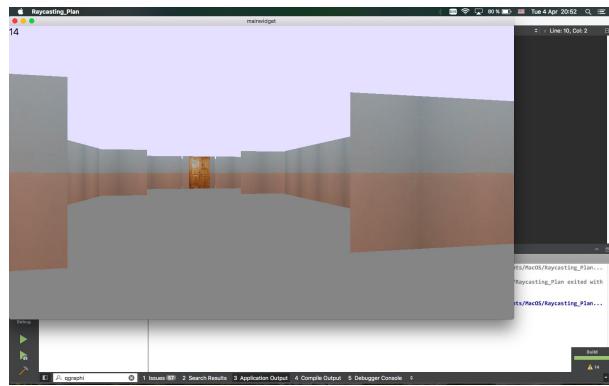


Рис. 2.9: Неоптимизированное текстурирование

Обычно сами текстуры отрисовываются в виде горизонтальных строк пикселей, но в случае рейкастинга, текстуры изображаются в виде вертикальных линий. Таким образом, чтобы оптимально использовать кэш центрального процессора, а также избежать пропусков страниц (avoid page misses), более эффективным может оказаться хранение текстур в памяти в виде набора вертикальных полос вместо горизонтальных строк пикселей[17]. Для этого после генерирования текстур необходимо поменять их значения  $x$  и  $y$ .

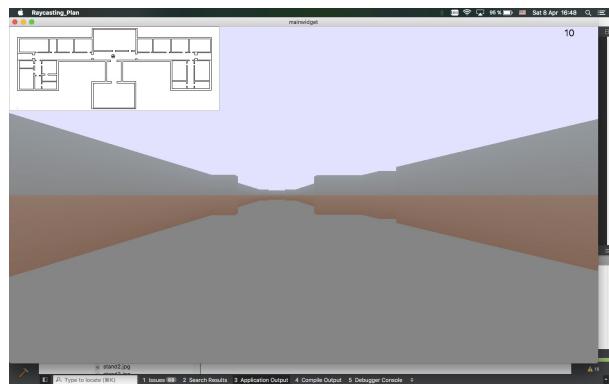


Рис. 2.10: Оптимизированное текстурирование

## 2.5. Паттерн для интерактивного взаимодействия

Для передвижения в среде полученного движка используется метод сдвига координат. Его суть банальна и сводится к следующим геометрическим манипуляциям: изменению координат  $x$  и  $y$  камеры на карте, изменения угла поворота камеры, замеры скорости перемещения камеры по пространству уровня для спидрана(англ. Speed Run - скоростное перемещение). Скорость перемещения камеры, а равно и игрока изменяется на константную величину, регулируя которую в файле конфигурации меняем скорость перемещения по карте.

Для передвижения по карте необходимо определить угол обзора относительно координатной оси, и, в зависимости от него изменить на соответствующий коэффициент координаты  $x$  и  $y$ . В нашем движке это определяется так: заранее в классе игрока Player определяется точка местоположения игрока *pos* и угол *dir*, задающийся нулём, что будет означать что камера смотрит точно по направлению оси  $Y$ . Так же задаются переменные *dx* и *dy*, определяющие изменение координат камеры, а так же переменную *ddir*, которая определяет изменение угла, математический смысл которой  $\Delta dir$ .

Функция описывающая изменение координат называется *Update*. При вызове этой функции, которая получает новые координаты в зависимости от времени изменения кардров, движения или отсутствия движения камеры, определяются изменения *dx* и *dy* и добавляются с определённой задержкой к основным координатам камеры, для получения плавного передвижения, а так же прибавляется изменение угла в зависимости от времени. После вышеперечисленых манипуляций переменные *dx*, *dy* и *ddir* обнуляются. В коде это определяется так:

```

void Player::update(double &time) {
    pos.setX(pos.x() + dx*time*0.1);
    pos.setY(pos.y() + dy*time*0.1);
    dir += (ddir*time*(-0.00001));
    ddir=0;
    dx=0;
    dy=0;
    time = 0;
}

```

Переменные  $dx$  и  $dy$  изменяются в части кода, отвечающей за интерактивное воздействие через нажатие клавиш и повороты мышью. Изменение угла зависит от поворота вектора  $\overline{dir}$ , который можно описать следующим шаблоном:

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

Развёртка клавиатурных сочетаний была выбрана стандартная игровая расстановка *WASD*, для лучшей адаптации. При нажатии клавиши *W* камера должна двинуться вперёд. Изменение координат  $dx$  и  $dy$  фиксируется следующим образом:

```

if (event->key() == Qt::Key_W) {
    RP->player->setDX( cos(RP->player->getDir()));
    RP->player->setDY( sin(RP->player->getDir()));
}

```

При нажатии клавиши *S* камера должна двинуться назад. Изменение координат  $dx$  и  $dy$  фиксируется следующим образом:

```

if (event->key() == Qt::Key_S) {

```

```

    RP->player->setDX(-cos(RP->player->getDir()));
    RP->player->setDY(-sin(RP->player->getDir()));
}

```

При нажатии клавиши *D* камера должна повернуться направо, и ширина угла поворота определяется длительностью нажатия. Изменение координат *dx* и *dy* фиксируется следующим образом:

```

if (event->key() == Qt::Key_D) {
    RP->player->setDX(sin(RP->player->getDir()));
    RP->player->setDY(-cos(RP->player->getDir()));
}

```

При нажатии клавиши *A* камера должна двинуться повернуться направо, и ширина угла поворота определяется длительностью нажатия. Изменение координат *dx* и *dy* фиксируется следующим образом:

```

if (event->key() == Qt::Key_A) {
    RP->player->setDX(-sin(RP->player->getDir()));
    RP->player->setDY(cos(RP->player->getDir()));
}

```

При отпускании клавиш изменения прекращаются, и координаты не меняются. Всё это проделывается каждый раз для каждого фрейма(кадра).

## 2.6. Поиск оптимального пути

Для поиска оптимального маршрута на карте задача была сведена к стандартной задаче поиска оптимального маршрута по графу. По своей сути алгоритм поиска пути ищет на графике, начиная с одной (стартовой) точки и исследуя смежные узлы до тех пор, пока не будет достигнут узел назначения (конечный узел). Кроме того, в алгоритмы поиска пути в большинстве случаев заложена также цель найти самый

короткий путь. Некоторые методы поиска на графе, такие как поиск в ширину, могут найти путь, если дано достаточно времени. Другие методы, которые "исследуют" граф, могут достичь точки назначения намного быстрее. Здесь можно привести аналогию с человеком, идущим через комнату. Человек может перед началом пути заранее исследовать все характеристики и препятствия в пространстве, вычислить оптимальный маршрут и только тогда начать непосредственное движение. В другом случае человек может сразу пойти в приблизительном или предполагаемом направлении цели и потом, уже во время пути, делать корректировки своего движения для избегания столкновений с препятствиями. К самым известным и популярным алгоритмам поиска пути относятся такие алгоритмы:

- алгоритм поиска A\*(A Star);
- алгоритм Дейкстры;
- Волновой алгоритм;
- Маршрутные алгоритмы;
- Навигационная сетка (Navmesh);
- Иерархические алгоритмы;
- обход препятствий;
- разделяй и властвуй;
- алгоритм поворота Креша.

Как наиболее простой и эффективный алгоритм не требующий серьёзных математических ресурсов был выбран Алгоритм Дейкстры. Алгоритм Дейкстры решает задачу о кратчайших путях из одной вершины для взвешенного ориентированного графа  $G = (V, E)$  с исходной вершиной  $s$ , в котором веса всех рёбер неотрицательны ( $\omega(u, v) \geq 0$ ) для всех  $(u, v) \in E$ [10].

### **Формальное объяснение:**

В процессе работы алгоритма Дейкстры поддерживается множество  $S \subseteq$

$V$ , состоящее из вершин  $\nu$ , для которых  $\delta(s, \nu)$  уже найдено. Алгоритм выбирает вершину  $u \in V \setminus S$  с наименьшим  $d[u]$ , добавляет  $u$  к множеству  $S$  и производит релаксацию всех рёбер, выходящих из  $u$ , после чего цикл повторяется. Вершины, не лежащие в  $S$ , хранятся в очереди  $Q$  с приоритетами, определяемыми значениями функции  $d$ . Предполагается, что граф задан с помощью списков смежных вершин.

### **Не формальное объяснение:**

Каждой вершине из  $V$  сопоставим метку - минимальное известное расстояние от этой вершины до  $a$ . Алгоритм работает пошагово - на каждом шаге он "посещает" одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

### **Инициализация:**

Метка самой вершины  $a$  полагается равной 0, метки остальных вершин - бесконечности. Это отражает то, что расстояния от  $a$  до других вершин пока неизвестны. Все вершины графа помечаются как непосещенные.

### **Шаг алгоритма:**

Если все вершины посещены, алгоритм завершается. В противном случае из еще не посещенных вершин выбирается вершина  $u$ , имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых  $u$  является предпоследним пунктом. Вершины, соединенные с вершиной  $u$  ребрами, назовем соседями этой вершины. Для каждого соседа рассмотрим новую длину пути, равную сумме текущей метки  $u$  и длины ребра, соединяющего  $u$  с этим соседом. Если полученная длина меньше метки соседа, заменим метку этой длиной. Рассмотрев всех соседей, пометим вершину  $u$  как посещенную и повторим шаг.

### **Пример работы алгоритма:**

Рассмотрим работу алгоритма на примере графа, показанного на рисун-

ке. Пусть требуется найти расстояния от 1-й вершины до всех остальных.

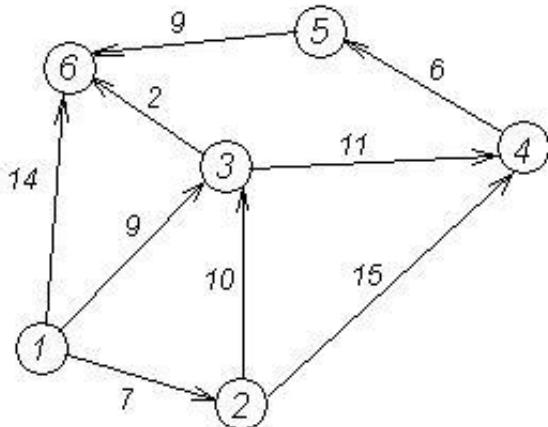
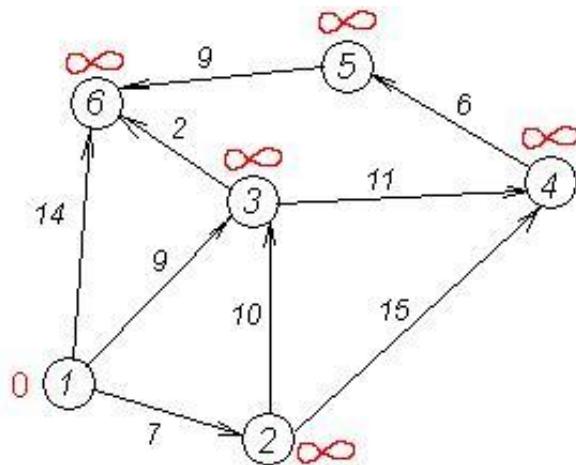


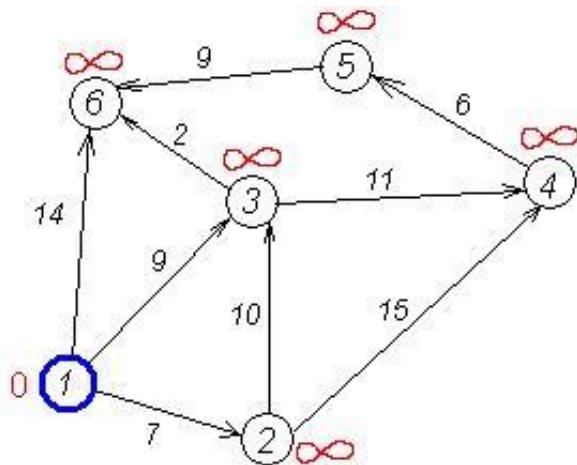
Рис. 2.11: Начальный граф

Кружками обозначены вершины, линиями - пути между ними (ребра графа). В кружках обозначены номера вершин, над ребрами обозначена их "цена" или "вес" длины пути. Рядом с каждой вершиной красным обозначена метка - длина кратчайшего пути в эту вершину из вершины 1.

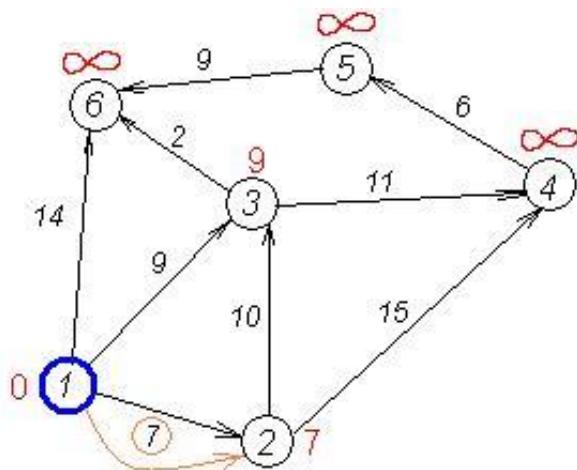


**Первый шаг.** Рассмотрим шаг алгоритма Дейкстры для нашего примера. Минимальную метку имеет вершина 1. Ее соседями являются вершины 2, 3 и 6.

Первый по очереди сосед вершины 1 - вершина 2, потому что длина пути до нее минимальна. Длина пути в нее через вершину 1 равна крат-



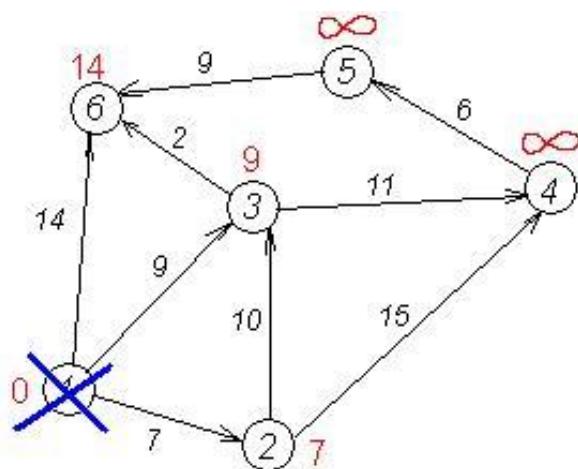
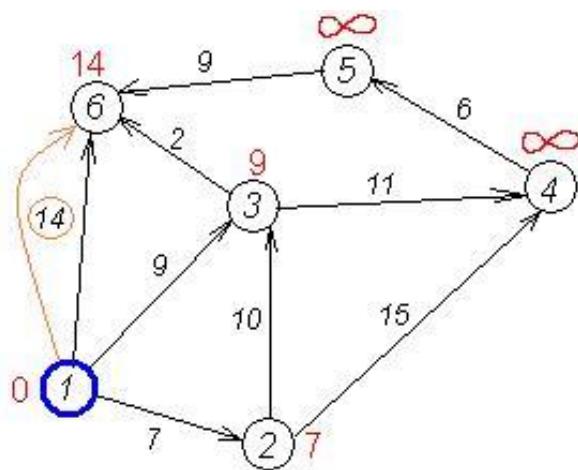
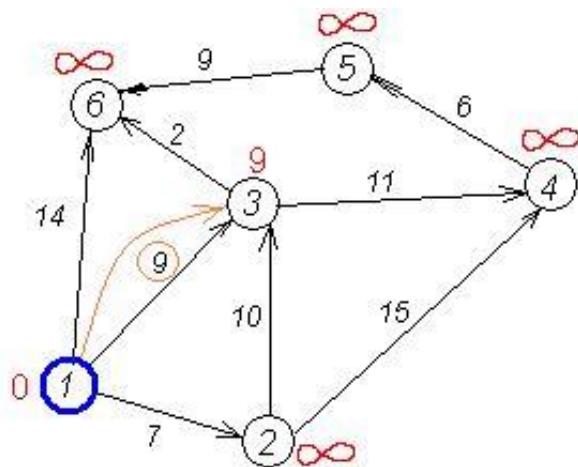
чайшему расстоянию до вершины 1 + длина ребра, идущего из 1 в 2, то есть  $0 + 7 = 7$ . Это меньше текущей метки вершины 2, поэтому новая метка 2-й вершины равна 7.



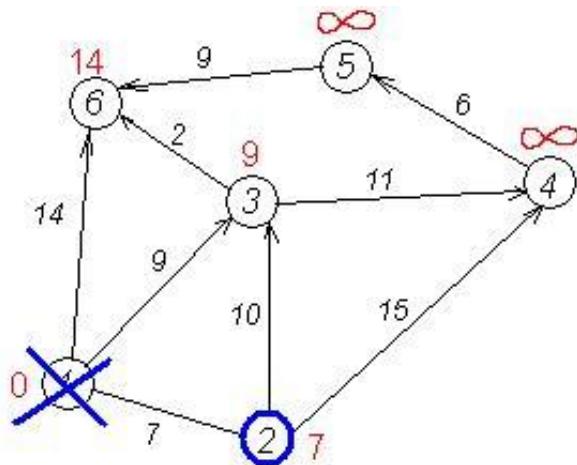
Аналогичную операцию проделываем с двумя другими соседями 1-й вершины - 3-й и 6-й.

Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит (то, что это действительно так, впервые доказал Дейкстра). Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.

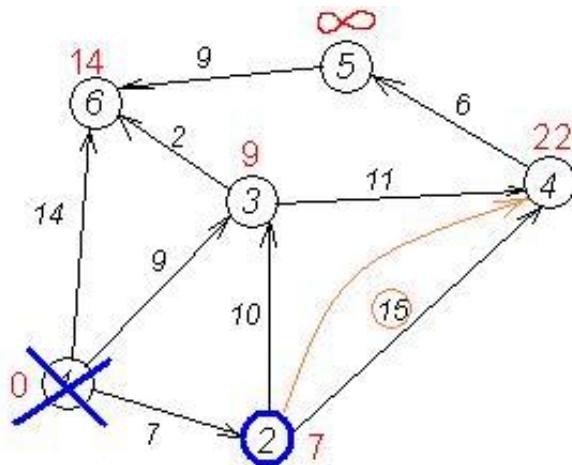
**Второй шаг.** Шаг алгоритма повторяется. снова находим "ближайшую" из непосещенных вершин. Это вершина 2 с меткой 7.



Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю. соседями вершины 2 являются 1, 3, 4. Пер-

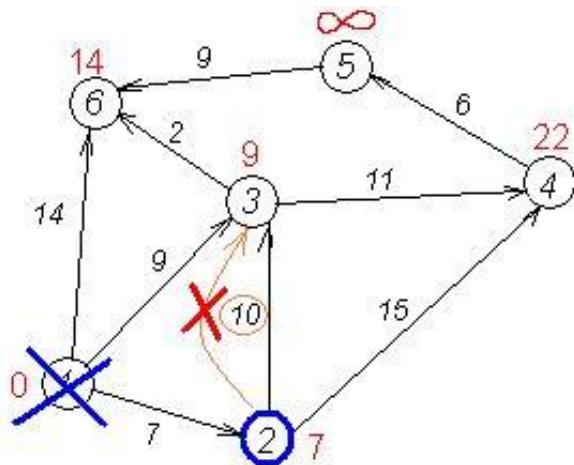


вый (по порядку) сосед вершины 2 - вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем. Следующий сосед вершины 2 - вершины 4 и 3. если идти в неё через 2-ю, то длина такого пути будет = кратчайшее расстояние до 2 + расстояние между вершинами 2 и 4 =  $7 + 15 = 22$ . Поскольку  $22 < \infty$ , устанавливаем метку вершины 4 равной 22.



Ещё один сосед вершины 2 - вершина 3. Если идти в неё через 2, то длина такого пути будет  $= 7 + 10 = 17$ . Но текущая метка третьей вершины равна  $9 < 17$ , поэтому метка не меняется.

Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем ее как посещенную.



**Третий шаг.** Повторяем шаг алгоритма, выбрав вершину 3.

**Дальнейшие шаги.** Повторяем шаг алгоритма для оставшихся вершин (Это будут по порядку 6, 4 и 5).

**Завершение выполнения алгоритма.** Алгоритм заканчивает работу, когда вычеркнуты все вершины. Результат его работы: кратчайший путь от вершины 1 до 2-й составляет 7, до 3-й - 9, до 4-й - 20, до 5-й - 20, до 6-й - 11.

Поскольку алгоритм Дейкстры всякий раз выбирает для обработки вершины с наименьшей оценкой кратчайшего пути, можно сказать, что он относится к жадным алгоритмам.

**Теорема.** Правильность алгоритма Дейкстры

Пусть  $G = (V, E)$  - взвешенный ориентированный граф с неотрицательной весовой функцией  $\omega : E \rightarrow R$  и исходной вершиной  $s$ . Тогда после применения алгоритма Дейкстры к этому графу для всех вершин  $u \in V$  будут выполняться равенства  $d[u] = \delta(s, u)$ .

**Следствие.** Пусть  $G = (V, E)$  - взвешенный ориентированный граф с неотрицательной весовой функцией  $\omega$  и исходной вершиной  $s$ . Тогда после применения алгоритма Дейкстры к этому графу подграф предшественников  $G_\pi$  будет деревом кратчайших путей с корнем в  $s$ .

## Время работы алгоритма Дейкстры

Сложность алгоритма Дейкстры зависит от способа нахождения вершины  $v$ , а также способа хранения множества непосещенных вершин и способа обновления меток. Обозначим через  $n$  количество вершин, а через  $m$  - количество ребер в графе  $G$ . В простейшем случае, когда для поиска вершины с минимальным  $d[v]$  просматривается все множество вершин, а для хранения величин  $d$  - массив, время работы алгоритма есть  $O(n^2 + m)$ . Основной цикл выполняется порядка  $n$  раз, в каждом из них на нахождение минимума тратится порядка  $n$  операций, плюс количество релаксаций (смен меток), которое не превосходит количества ребер в исходном графе.

Для разреженных графов (то есть таких, для которых  $m$  много меньше  $n^2$ ) непосещенные вершины можно хранить в двоичной куче, а в качестве ключа использовать значения  $d[i]$ , тогда время извлечения вершины из  $U$  станет  $\log n$ , при том, что время модификации  $d[i]$  возрастет до  $\log n$ . Так как цикл выполняется порядка  $n$  раз, а количество релаксаций не больше  $m$ , скорость работы такой реализации  $O(n \cdot \log n + m \cdot \log n)$ . Если для хранения непосещенных вершин использовать фибоначчиеву кучу, для которой удаление происходит в среднем за  $O(\log n)$ , а уменьшение значения в среднем за  $O(1)$ , то время работы алгоритма составит  $O(n \cdot \log n + m)$ . Благодаря такому нерастратному к времени алгоритму, а так же предрасчётом до визуализации, сам поиск оптимального расчёта не скажется на количестве кадров в секунду.

Для нашего проекта поиск оптимального маршрута будет проходить по массиву отрезков, задающих уровень. Фактически, граф строится по отрезкам означающим двери на карте. Благодаря такому удобному подходу, поиск маршрута сводится к заданию маршрута между различными дверями на карте[10].

## 2.7. Движение сегментов

Для движения сегментов в поле рисовки динамически изменяются координаты необходимого сегмента. Поскольку каждый сегмент задаётся двумя точками  $A(x_1, y_1)$  и  $B(x_2, y_2)$ , при перемещении новые координаты задаются уже другими точками:  $\tilde{A}(\tilde{x}_1, \tilde{y}_1)$  и  $B(x_2, y_2)$  при перемещении точки  $A$ ,  $A(x_1, y_1)$  и  $\tilde{B}(\tilde{x}_2, \tilde{y}_2)$  при перемещении точки  $B$ , и  $\tilde{A}(\tilde{x}_1, \tilde{y}_1)$  и  $\tilde{B}(\tilde{x}_2, \tilde{y}_2)$  при перемещении обоих точек. Длина сегмента не изменяется и задаётся формулой  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ . При повороте одной точки  $A(x_1, y_1)$ , изменённая координата  $\tilde{A}$  принимает координаты:  $(\tilde{x}_1 = x_1 + \cos \alpha, \tilde{y}_1 = y + \sin \alpha)$ , где угол  $\alpha$  - угол между осью  $oX$  и самим сегментом. При движении другой точки происходят аналогичные преобразования с учётом длины сегмента  $d$ . И эти манипуляции производятся каждый тик внутрисистемного времени с зависимостью от необходимой скорости вращения[16].

```
void MovableSegment :: update( double &time ) {
    time = qBound( 1. , time , 200. );
    if ( 11 > 12 ) {
        if ( 1+du*time<=12 ) {
            l = 12 ;
            setB( QPointF( cos( l )*len + A().x() ,
                           sin( l )*len + A().y() ) );
            return ; } else {
        if ( 1+du*time>=12 ) {
            l = 12 ;
            setB( QPointF( cos( l )*len + A().x() ,
                           sin( l )*len + A().y() ) );
            return ; } } l+=du*time ;
    setB( QPointF( cos( l )*len + A().x() ,
                   sin( l )*len + A().y() ) ); }
```

# Глава 3

## Реализация

### 3.1. Инструментарий

Для создания проекта был выбран язык программирования *C++*‘14 с фреймворком *Qt* 5.8. *Qt* это кроссплатформенный инструментарий для разработки ПО на языке *C++*, в данной редакции используется язык *C++* в редакции *ISO/IEC JTC1* (полное название: *International Standard ISO/IEC 14882 : 2014(E) Programming Language C++*)[6].

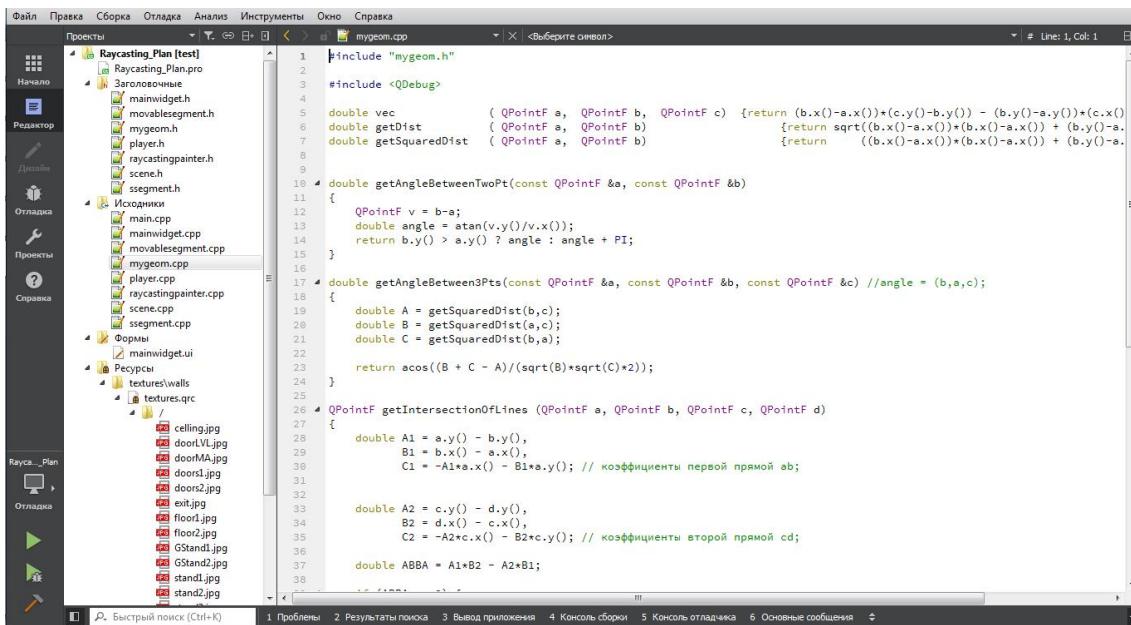


Рис. 3.1: Написание кода в *Qt*

*Qt* позволяет запускать написанное с его помощью ПО в большинстве современных операционных систем путём простой компиляции программы для каждой ос без изменения исходного кода. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и

*XML*. *Qt* является полностью объектно-ориентированным, легко расширяемым и поддерживающим технику компонентного программирования. Отличительная особенность *Qt* от других библиотек - использование *Meta Object Compiler (MOC)*. *MOC* - компилятор - это предварительная система обработки исходного кода. *MOC* позволяет во много раз увеличить мощь библиотек, вводя такие понятия, как слоты и сигналы. Кроме того, это позволяет сделать код более лаконичным. Утилита *MOC* ищет в заголовочных файлах на C++ описания классов, содержащие макрос *Q\_OBJECT*, и создаёт дополнительный исходный файл на C++, содержащий метаобъектный код[23].

Для переноса общей части на *JavaScript* был использован кросс-компилятор C++ в *JavaScript* под названием *Emscripten*. *Emscripten* — компилятор из *LLVM* байт-кода в *JavaScript*. C/C++ код может быть скомпилирован в *LLVM* байт-код с помощью компилятора *Clang*. Некоторые другие языки так же имеют компиляторы в *LLVM* байт-код. *Emscripten* на основе байт-кода генерирует соответствующий *JavaScript*-код, который может быть выполнен любым интерпретатором *JavaScript*, например современным браузером. *Emscripten* предоставляет: *emconfigure* – утилита настройки окружения и последующего запуска *./configure*; *emmake* – утилита для настройки окружения и последующего запуска *make*; *emcc* – компилятор *LLVM* в *JavaScript*[35].

Сам код на *Qt* был написан максимально неплатформозависимым, следующим стандартам языка и не использующим нестандартные библиотеки, для того что бы написанный код был без проблем перенесён на *JavaScript*. Для достижения такого эффекта даже была перенесена и в ручную доделана библиотека из *Boost*, использующаяся в геометрии для просчёта расстояния, пройденого лучём, а так же для работы с сегментами. Новые библиотеки получили имя *ssegment.h* и *tgeom.h*[34].

Для автоматической сборки проекта и компиляции использована утилита *CMake*. *CMake* - это универсальная кроссплатформенная утилита для автоматической сборки программы из исходных кодов. При этом сама *CMake* непосредственно сборкой кода не занимается, а выступает в качестве front-end'a для back-end компилятора. И в итоге для общей сборки проекта необходим скрипт сборки для *CMake*, который выглядит следующим образом:

```
cmake -G"MinGW Makefiles" -DCMAKE_TOOLCHAIN_FILE=
C:\cheerp\share\cmake\Modules\CheerpToolchain.cmake
.. / segments && mingw32-make
```

```
cmd <1> cmd
C:\cheerp\share\cmake\Modules\CheerpToolchain.cmake:12 (CMAKE_FORCE_CXX_COMPILER)
C:/Program Files/CMake/share/cmake-3.8/Modules/CMakeDetermineSystem.cmake:88 (include)
CMakeLists.txt:3 (project)

CMake Deprecation Warning at C:/Program Files/CMake/share/cmake-3.8/Modules/CMakeForceCompiler.cmake:69 (message):
The CMAKE_FORCE_C_COMPILER macro is deprecated. Instead just set
CMAKE_C_COMPILER and allow CMake to identify the compiler.
Call Stack (most recent call first):
C:\cheerp\share\cmake\Modules\CheerpToolchain.cmake:11 (CMAKE_FORCE_C_COMPILER)
D:/Project/agarb/test/CMakeFiles/3.8.0-rc2/CMakeSystem.cmake:6 (include)
CMakeLists.txt:3 (project)

CMake Deprecation Warning at C:/Program Files/CMake/share/cmake-3.8/Modules/CMakeForceCompiler.cmake:83 (message):
The CMAKE_FORCE_CXX_COMPILER macro is deprecated. Instead just set
CMAKE_CXX_COMPILER and allow CMake to identify the compiler.
Call Stack (most recent call first):
C:\cheerp\share\cmake\Modules\CheerpToolchain.cmake:12 (CMAKE_FORCE_CXX_COMPILER)
D:/Project/agarb/test/CMakeFiles/3.8.0-rc2/CMakeSystem.cmake:6 (include)
CMakeLists.txt:3 (project)

-- Configuring done
-- Generating done
-- Build files have been written to: D:/Project/agarb/test
Scanning dependencies of target segments
[ 1%] Building CXX object Chakelites/segments.dir/cheerapplication/webmain.cpp.obj
[ 2%] Building CXX object Chakelites/segments.dir/cheerimplementation/canvas.cpp.obj
[ 3%] Building CXX object Chakelites/segments.dir/cheerimplementation/imagebuffer.cpp.obj
[ 4%] Building CXX object Chakelites/segments.dir/mock/inputbuffer.cpp.obj
[ 5%] Building CXX object Chakelites/segments.dir/cone/canvas.cpp.obj
[ 6%] Building CXX object Chakelites/segments.dir/cone/color.cpp.obj
[ 7%] Building CXX object Chakelites/segments.dir/cone/imagebuffer.cpp.obj
[ 8%] Building CXX object Chakelites/segments.dir/cone/inputbuffer.cpp.obj
[ 9%] Building CXX object Chakelites/segments.dir/cone/scene.cpp.obj
[100%] Linking CXX executable segments.js
[100%] Built target segments

Александр@ALEKSANDER-PC D:\Project\agarb\test
> |
cmd.exe[64]:5312 +160904[64] 1/1 [+/-] NUM PRI: 170x41 (3.94) 25V 6032 100% //
```

Рис. 3.2: Пример компиляции при помощи CMake

Подобного рода скрипты позволяют скомпилировать проект и сразу в нативную версию, и в версию для web-приложения с условием готового платформозависимого канваса и буфера кадров для странички *HTML*. Выходной *JavaScript*-файл встраивается на заранее установленную страничку автоматически[43].

### 3.2. Организация работы над проектом

Для организации работы над проектом была использована электронная доска *Trello* для упрощённой организации и управления проектом между всеми участниками проекта. *Trello* - это интерактивное веб-приложение, выглядящее как смесь электронной доски для стикеров и календаря. *Trello* использует парадигму для управления проектами, известную как канбан - "точно-в-срок".

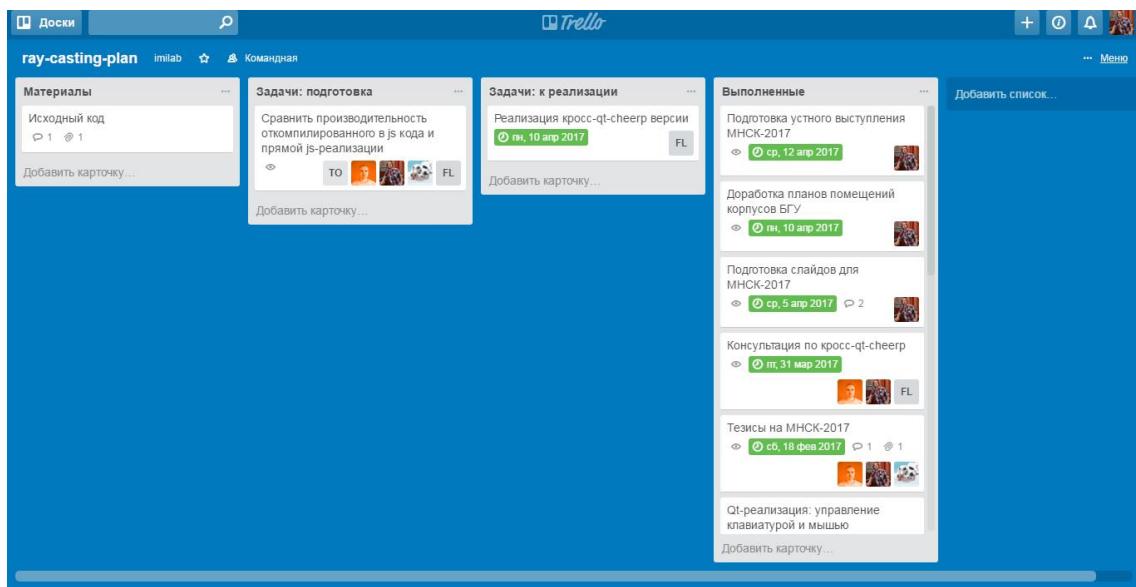


Рис. 3.3: Интерфейс электронной доски Trello

Суть *Trello* в том, что условной доске в виде сообщений висят задачи, которые необходимо реализовать обязательно в срок до заданной на карточке дате. При организации работы над проектом, в котором участвуют несколько человек, это приложение очень сильно облегчает организационные моменты.

Для упрощения организации при написании исходных кодов использовался *Git* - распределённая система управления версиями в виде централизованных репозиториев. Репозиторий *Git* представляет собой каталог файловой системы, в котором находятся файлы конфигурации репо-

зитория, файлы журналов, хранящие операции, выполняемые над репозиторием, индекс, описывающий расположение файлов и хранилище, содержащее собственно файлы. структура хранилища файлов не отражает реальную структуру хранящегося в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, при локальных изменениях или при получении патча от другого узла), оно создаёт в хранилище новые файлы, соответствующие новым состояниям изменённых файлов. существенно, что никакие операции не изменяют содержимого уже существующих в хранилище файлов. По умолчанию репозиторий хранится в подкаталоге с названием `".git"` в корневом каталоге рабочей копии дерева файлов, хранящегося в репозитории. Любое файловое дерево в системе можно превратить в репозиторий git, отдав команду создания репозитория из корневого каталога этого дерева (или указав корневой каталог в параметрах программы). репозиторий может быть импортирован с другого узла, доступного по сети. При импорте нового репозитория автоматически создаётся рабочая копия, соответствующая последнему зафиксированному состоянию импортируемого репозитория (то есть не копируются изменения в рабочей копии исходного узла, для которых на том узле не была выполнена команда `commit`).

Для упрощения работы над функционалом программы использовалась модель работы с ветками в *Git* под названием *Git Flow*. *Git Flow* - это модель, которая показывает, как можно проводить разработку в команде, используя возможности *Git*. Разработчики могут работать над задачами как индивидуально, так и в группах, не мешая друг другу. При этом в течение всего жизненного цикла разработки существуют только две основные ветки: *master* и *develop*, поэтому в репозитории поддерживается постоянный порядок, поскольку все другие ветки являются лишь временными. Данная модель проста и понятна, а применение расширения автоматизации делает ее очень удобной в использовании.

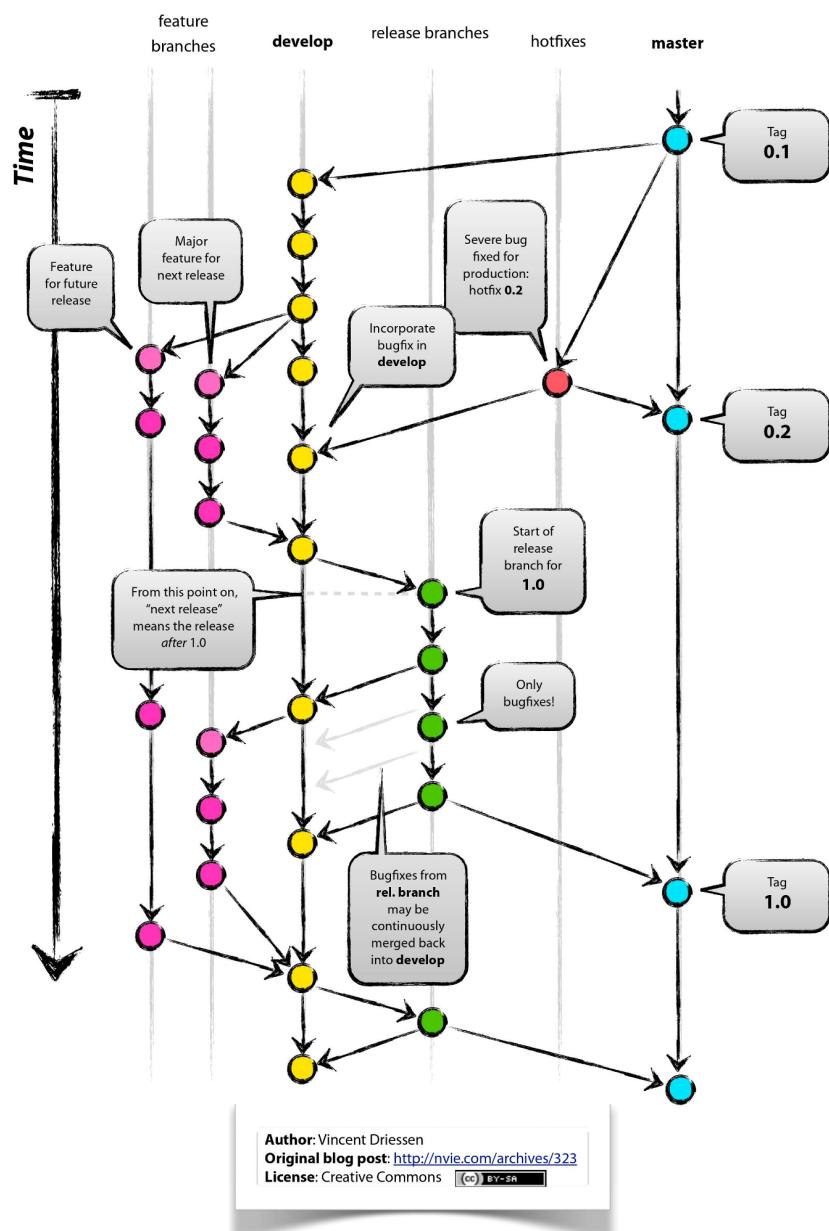


Рис. 3.4: Визуализация GitFlow

Центральный репозиторий содержит две главные ветви, существующие всё время - ветвь *master* и ветвь *develop*. Ветвь *master* создаётся при инициализации репозитория, что должно быть знакомо каждому пользователю *Git*. Параллельно ей также мы создаём ветку для разработки под названием *develop*. Мы считаем ветку *origin/master* главной. То есть, исходный код в ней должен находиться в состоянии *production* –

*ready* в любой произвольный момент времени. Ветвь *origin/develop* мы считаем главной ветвью для разработки. Хранящийся в ней код в любой момент времени должен содержать самые последние изданные изменения, необходимые для следующего релиза. Эту ветку также можно назвать "интеграционной". Когда исходный код в ветви разработки (*develop*) достигает стабильного состояния и готов к релизу, все изменения должны быть определённым способом влиты в главную ветвь (*master*) и помечены тегом с номером релиза. Следовательно, каждый раз, когда изменения вливаются в главную ветвь (*master*), мы по определению получаем новый релиз. Мы стараемся относиться к этому правилу очень строго, так что, в принципе, мы могли бы использовать хуки *Git*, чтобы автоматически собирать наши продукты и выкладывать их на рабочие сервера при каждом коммите в главную ветвь (*master*).

Помимо главных ветвей *master* и *develop*, наша модель разработки содержит некоторое количество типов вспомогательных ветвей, которые используются для распараллеливания разработки между членами команды, для упрощения внедрения нового функционала (*features*), для подготовки релизов и для быстрого исправления проблем в производственной версии приложения. В отличие от главных ветвей, эти ветви всегда имеют ограниченный срок жизни, и каждая из них в конечном итоге рано или поздно удаляется.

В итоге, мы используем следующие типы ветвей:

- Ветви функциональностей (*Feature branches*);
- Ветви релизов (*Release branches*);
- Ветви исправлений (*Hotfix branches*).

У каждого типа ветвей есть своё специфическое назначение и строгий набор правил, от каких ветвей они могут порождаться, и в какие должны влияться[37].

### 3.3. Структура проекта (UML)

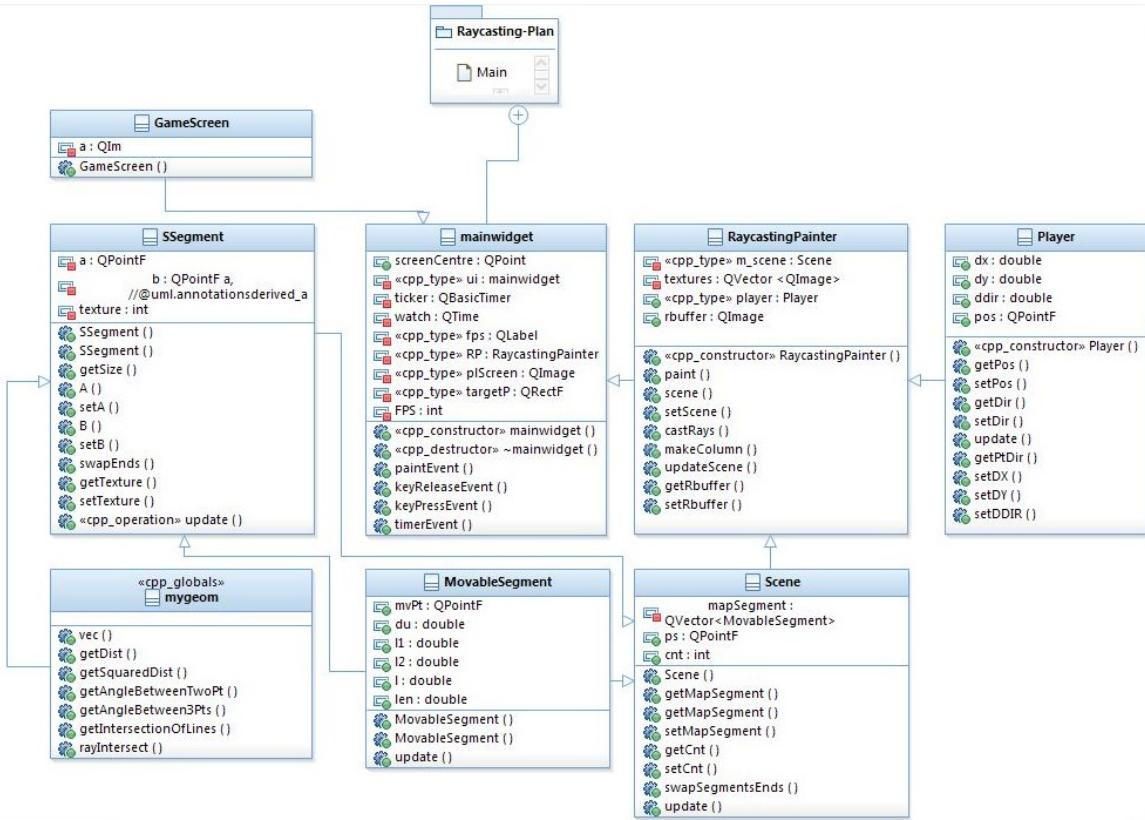


Рис. 3.5: UML - диаграмма проекта

У нас есть несколько объектов, самый главный - **MainWidget**. В нём запускается экземпляр класса **RaycastingPainter**, в котором происходит отрисовка кадра, а так же в нём определяется экран на котором происходит отрисовка **GameScreen**. Для работы **RaycastingPainter** создаются объекты **Player** и **Scene**. В **Player** задаются координаты камеры и направление камеры, в **Scene** задаются угол обзора, а так же определяется массив типа данных **SSegment**. **SSegment** - это сегмент карты с данными и текстурами, а так же определяется движение сегмента **MovableSegment**.

### 3.4. Реализация на целевых платформах

Поскольку современный человек предпочитает универсальные механизмы для любых платформ, мы так же озадачили себя кроссплатформенностью одного и того же кода, причём на таких казалось бы несовместимых платформах как *web* и нативная платформа(десктоп и мобильное приложение). Для достижения кроссплатформы на любой десктопной операционной системы, в разработке был применён язык *C++*'14 с фреймворком *Qt* 5.8.0. Использование именно этой среды и языка обеспечило полную совместимость как в *\*nix* системах (в частности, в *MacOS X*, системах на основе ядра *Linux*, *BSD* - системах и тд), так и в среде *Windows*. Так же фреймворк *Qt* позволяет с лёгкостью перекомпилировать тот же самый код без особых изменений на любое *Android* - устройство и устройство с *iOs*, например *iPhone*, что весьма актуально ввиду всеобщей распространённости таких гаджетов.

Однако, с платформой *Web* для встраивания на сайты пришлось повозиться. Любая *web*-страничка в Интернете представляет собой совокупность документа *HTML*, каскадную страницу стилей *CSS*, скриптовую часть на языке *JavaScript* и внутреннее содержание в виде текста, документов содержащих в себе картинки, аудиоконтент, видеоконтент, мультимедиа, апплеты и гиперссылки на другие страницы. И для реализации на сайте нашего приложения необходимо было создать скриптовый апплет. Ввиду абсолютной несовместимости *web*-платформы с языком *C++* в качестве скриптового языка для встраиваемых апплетов, необходимо было перенести реализацию на язык *JavaScript* с сохранением работоспособности десктопной версии.

Для решения этой проблемы мы разделили кодовую базу, сделав общей часть непосредственно решающую задачу, и отделив код специфичный для каждой из платформ. Код решающий задачу оперирует абстрактны-

ми классами канвы (*Core :: Canvas*) и буфера (*Core :: ImageBuffer*), которые в свою очередь конкретизируются для каждой платформы.

В итоге, в общую часть у нас вошло: Алгоритм рейкастинга, Представление карт уровней, Абстракции канвы и буфера кадров. И части для каждой платформы вошли конкретные реализации канвы и буфера кадров. Сейчас специальная часть для платформ занимает лишь 30% от общей кодовой базы, и нет причин для увеличения этой части с ростом проекта.

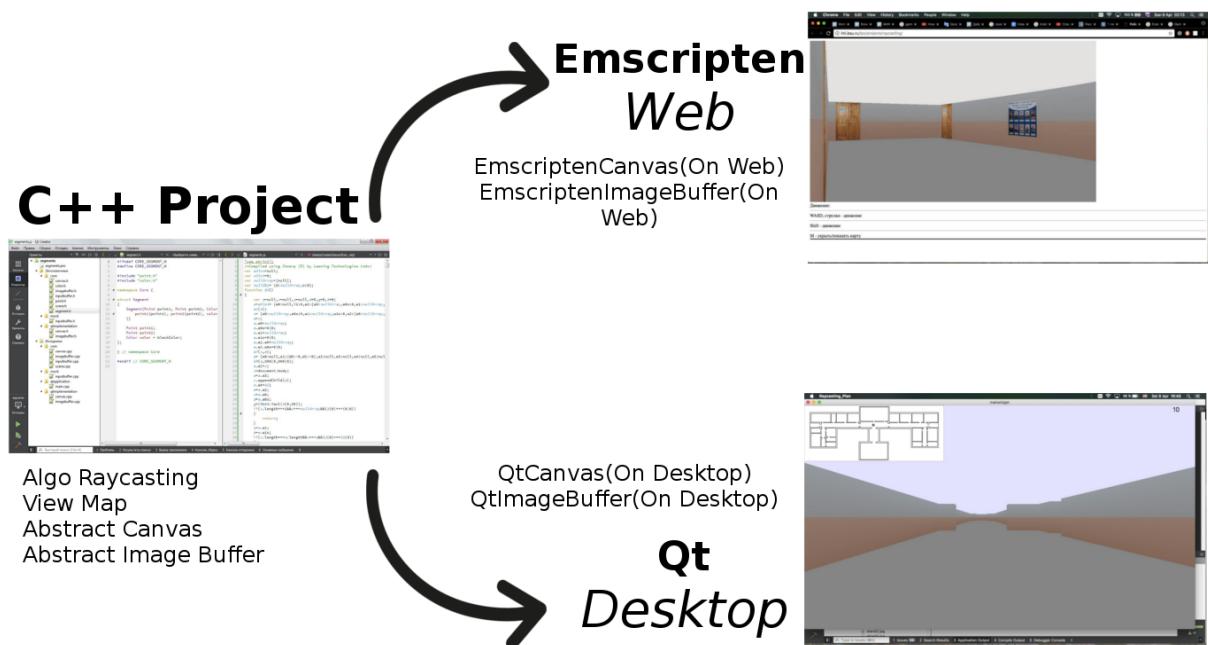


Рис. 3.6: Разделение конкретных реализаций

Для переноса общей части на *JavaScript* был использован кросс-компилятор *C++* в *JavaScript* под названием *Emscripten*, как было описано выше.

В общем итоге одна кодовая база делится на два подвида - для нативной платформы версия с *Qt* и для Web-платформы версия *JavaScript*. Подобные проекты с уникальной кодовой базой под разные платформы крайне редки в наше время, и это уникальная разработка для нас.

### 3.5. Производительность

После реализации нашего проекта были произведены замеры производительности на разных платформах с разным железом. Производительность нативной версии была произведена по показателям потребляемой памяти, процентов загрузки процессора и выдаваемым FPS(количества кадров в секунду). Отсутствие замеров производительности по видеокарте были излишними, поскольку наш проект работал только с процессором напрямую, и не затрагивал видеокарту, только для вывода самого изображения. Замеры версии для ПК были произведены на компьютерах с процессором *Intel Core i7* и 8 Гигабайтами оперативной памяти, ноутбуке с процессором *AMD A10 4600M* и 8 Гигабайтами оперативной памяти, а так же на старом компьютере с *AMD Athlon XP 2000+* и 512 Мегабайтами оперативной памяти. Замеры производились в операционных системах *Windows 10 Redstone 2 (1703)*

- *Creators Update*(*Windows XP SP3* для *AMD Athlon XP 2000+), *Ubuntu 16.04.2 LTS Xenial Xerus* и *FreeBSD 11.0*. Результаты бенчмарков вы видите ниже:*

- Для ПК с *Intel Core i7* и 8 ГБ ОЗУ данные указаны в таблице 3.1:

Таблица 3.1: Замеры производительности на 1 компьютере

OS	Memory	CPU	FPS
<i>Windows 10</i>	25 Mbyte	10%	27-41
<i>Ubuntu 16.04.2 LTS</i>	17.4 Mbyte	8%	34-44
<i>FreeBSD 11.0</i>	18 Mbyte	8%	32-47

- Для ПК с *AMD A10 4600M* и 8 ГБ ОЗУ данные указаны в таблице 3.2:

- Для ПК с *AMD Athlon XP 2000+* и 512 МБ ОЗУ данные указаны в

Таблица 3.2: Замеры производительности на 2 компьютере

<b>OS</b>	<b>Memory</b>	<b>CPU</b>	<b>FPS</b>
<i>Windows 10</i>	25 Mbyte	12%	25-39
<i>Ubuntu 16.04.2 LTS</i>	17.4 Mbyte	9%	33-40
<i>FreeBSD 11.0</i>	18 Mbyte	9%	33-42

таблице 3.3:

Таблица 3.3: Замеры производительности на 3 компьютере

<b>OS</b>	<b>Memory</b>	<b>CPU</b>	<b>FPS</b>
<i>Windows XP</i>	30 Mbyte	34%	16-23
<i>Ubuntu 16.04.2 LTS i386</i>	22.3 Mbyte	25%	19-26
<i>FreeBSD 11.0</i>	20 Mbyte	23%	20-26

Как видно из таблиц 3.1-3.3, приложение расходует крайне мало оперативной памяти вне зависимости от операционной системы, однако накладывает достаточные вычислительные затраты на процессор. Но в целом приложение хорошо идёт даже на слабых старых компьютерах. Под операционной системой *Windows* наше приложение работает хуже чем на аналогичном железе под *Linux* и *FreeBSD*, ввиду неоптимальности и несовершенности самой операционной системы *Windows*. Под *Linux* и *FreeBSD* приложение затрачивает примерно одинаковое количество ресурсов и выдаёт практически одинаковое значение количества кадров в секунду.

Так же замеры производительности были произведены на *MacBookAir* со стоковой *macOSSierra* и переносным самодельным дистрибутивом *Debian*

*Linux 8.7*. Результаты бенчмарков вы видите в таблице 3.4:

Таблица 3.4: Замеры производительности на *MacBook*

OS	Memory	CPU	FPS
<i>iOsX</i>	18 Mbyte	21%	33-42
<i>Debian 8.7</i>	18 Mbyte	20.9%	33-42

Как видно из таблицы 3.4, приложение расходует крайне мало оперативной памяти вне зависимости от операционной системы. Под *Linux* и *iOsX* приложение затрачивает примерно одинаковое количество ресурсов и выдаёт практически одинаковое значение количества кадров в секунду.

Для мобильных устройств тесты проводились на устройствах с операционной системой *Android*. При компиляции использовался *Android SDK* 18, поэтому полученное приложение запустится на любом устройстве с системой *Android* 4.0 и выше. Для теста были выбраны три аппарата: флагманский *Samsung Galaxy A5* с восьмиядерным процессором на 1900 *MHz* и 3 Гб оперативной памяти, *Highscreen Power Ice Evo* с четырёхядерным процессором на 1250 *MHz* и 2 Гб оперативной памяти и одноядерный *Samsung Galaxy Star Plus* с 800 *MHz* и 512 Мб оперативной памяти . Тестирование проводилось только на показатель FPS, поскольку точно установить производительность определённых приложений в Андроиде очень сложно. Результаты бенчмарков вы видите на таблице 3.5:

Как видно из таблицы, производительные многоядерные мобильные устройства имеют достаточно мощности что бы выдавать оптимальное количество кадров в секунду. Старые аппараты плохо тянут приложение, но это очевидно, поскольку такие аппараты уже считаются устаревшими, и уже не тянут приложения вроде последних версий *Google Chrome*.

Таблица 3.5: Замеры производительности на мобильных устройствах

Model	Antutu Test	FPS
<i>Samsung Galaxy A5</i>	59834	24-30
<i>Highscreen Power Ice Evo</i>	31672	20-26
<i>Samsung Galaxy Star Plus</i>	8012	12-20

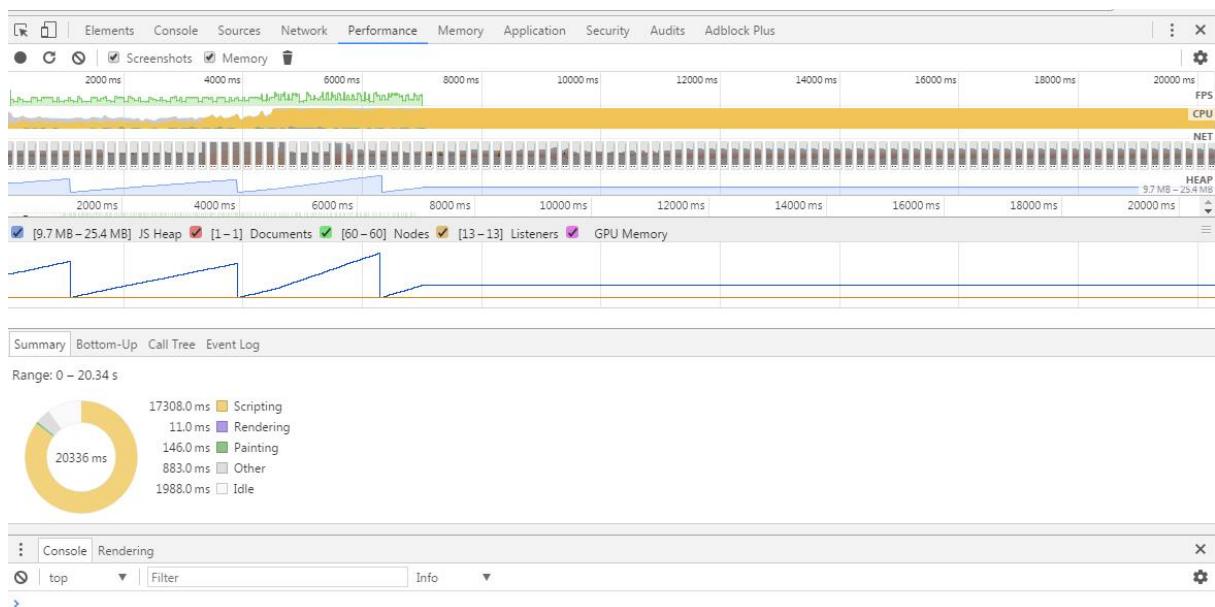


Рис. 3.7: Производительность web-версии в браузере Chromium

Мобильная версия приложения получилось универсальной, поскольку поддерживается любыми браузерами со встроенной поддержкой *HTML 5*, а таковыми являются почти все современные браузеры на любых компьютерах. Тестирование производилось в наиболее популярных ныне браузерах *Mozilla Firefox*, *Chromium*, *Safari* и *Microsoft Edge*. Тестирование браузеров *GoogleChrome*, *Opera* и прочих подобных браузеров было так же проведено, но поскольку все они используют движок и кодовую базу *Chromium*, то они получали аналогичные результаты. Тести-

рование производилось по результатам замеров используемой памяти и FPS. Результаты вы видите в таблице 3.6:

Таблица 3.6: Замеры производительности в различных браузерах

Browser	Memory	FPS
<i>Mozilla Firefox</i>	25.4 Mbyte	42-60
<i>Chromium</i>	28 Mbyte	39-56
<i>Safari</i>	28 Mbyte	39-56
<i>Microsoft Edge</i>	30.6 Mbyte	34-51

Как видно из таблицы, любой современный браузер выдаёт достаточное количество кадров в секунду для плавной работы, и браузерная версия гораздо менее ресурсозатратна ввиду меньшей точности при вычислениях. Результаты работы у браузера *Microsoft Edge* оказались наименее производительными, поскольку традиционно программные продукты от *Microsoft* отличаются низким качеством. Результаты *Chromium* и *Safari* оказались идентичными, поскольку они оба используют движок *WebKit*. Лучшим результатом оказался результат *Mozilla Firefox*, поскольку сам браузер и его движок очень качественные, производительные и на любых тестах показывает самые производительные результаты среди всех браузеров.

### 3.6. Внедрение ссылки на сайт ИМИ БГУ

Итоговой частью работы стало внедрение полученной веб-версии на сайт ИМИ. Поскольку сама реализация проекта выглядит как *JavaScript* - скрипт встраиваемый в *HTML* - форму, было решено отказаться от реализации сложного сайта в виде какой-либо *CMS* в сторону простой *HTML* страницы, содержащей в себе данный скрипт.

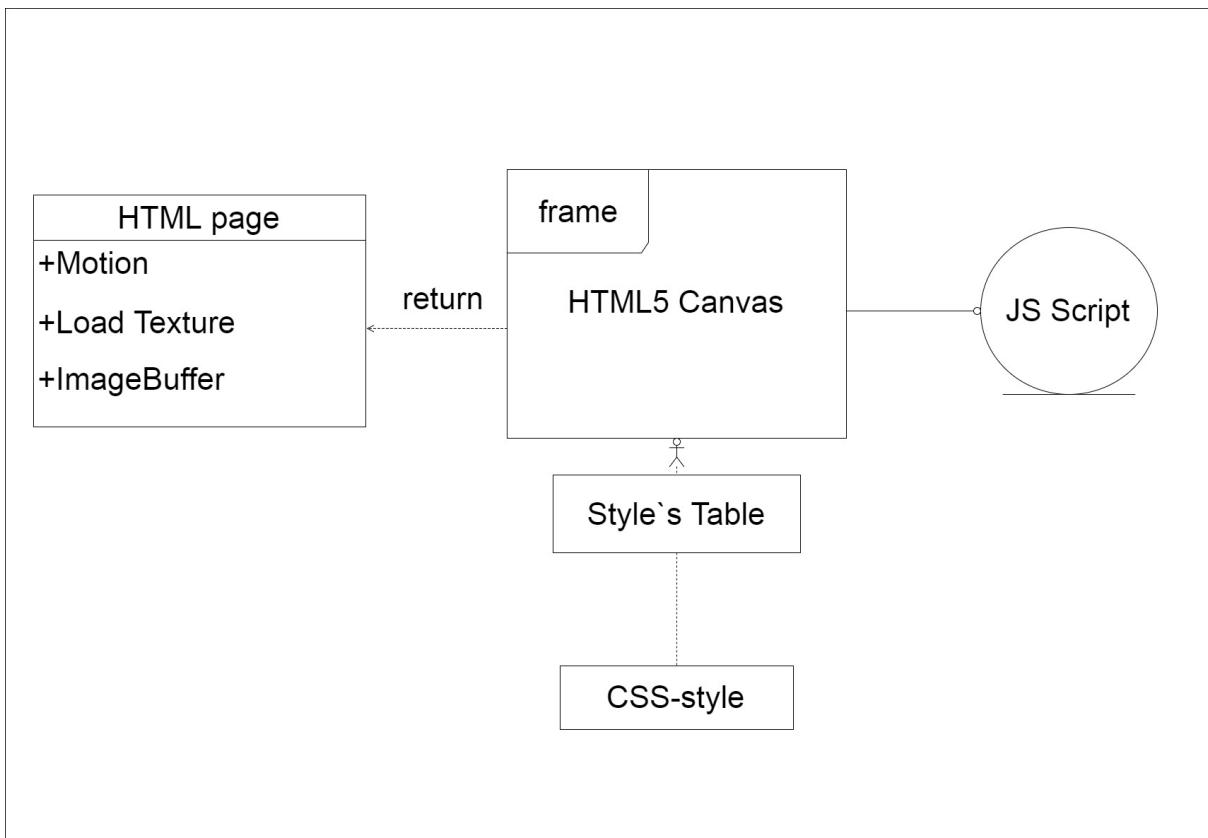


Рис. 3.8: Схема работы сайта

*HTML* (от англ. HyperText Markup Language - "язык гипертекстовой разметки") - стандартизованный язык разметки документов в Интернете. Большинство веб-страниц содержат описание разметки на языке *HTML* (или *XHTML*). Язык *HTML* интерпретируется браузерами; полученный в результате интерпретации форматированный текст отображается на экране монитора компьютера или мобильного устрой-

ства. В данном проекте был использован наиболее современный стандарт *HTML*, называемый *HTML5*. *HTML5* - новый стандарт *HTML*, поддерживаемый самыми совершенными браузерами. суть его отличий от старых версий *HTML* - в *HTML5* реализовано множество новых синтаксических особенностей. Например, элементы *< video >*, *< audio >* и *< canvas >*, а также возможность использования *SVG* и математических формул. Эти новшества разработаны для упрощения создания и управления графическими и мультимедийными объектами в сети без необходимости использования сторонних API и плагинов. Другие новые элементы, такие как *< section >*, *< article >*, *< header >* и *< nav >*, разработаны для того, чтобы обогащать семантическое содержимое документа (страницы). Новые атрибуты были введены с той же целью, хотя ряд элементов и атрибутов был удалён. Некоторые элементы, например *< a >*, *< menu >* и *< cite >*, были изменены, переопределены или стандартизированы. *API* и *DOM* стали основными частями спецификации *HTML5*. *HTML5* также определяет некоторые особенности обработки ошибок вёрстки, поэтому синтаксические ошибки должны рассматриваться одинаково всеми совместимыми браузерами.

В числе прочего, *HTML5* так же поддерживает такой элемент как *< canvas >*. *Canvas* (англ. canvas - "холст рус. канвас) - элемент *HTML5*, предназначенный для создания растрового двухмерного изображения при помощи скриптов, обычно на языке *JavaScript*. Начало отсчёта блока находится слева сверху. От него и строится каждый элемент блока. Размер пространства координат не обязательно отражает размер фактической отображаемой площади. По умолчанию его ширина равна трем стам пикселям, а высота ста пятидесяти. Используется, как правило, для отрисовки графиков для статей и игрового поля в некоторых браузерных играх. Но также может использоваться для встраивания видео в страницу и создания полноценного плеера.

Особенности:

- Изменение высоты или ширины холста сотрет всё его содержимое и все настройки, проще говоря он создаётся заново;
- Начало отсчёта (точка 0,0) находится в левом верхнем углу. Но её можно сдвигать;
- 3D контекста нет, есть отдельные разработки, но они не стандартизованы;
- Цвет текста можно указывать аналогично *CSS*, впрочем, как и размер шрифта.

Благодаря его свойствам, мы можем встроить полученный нами из *C++* скрипт в *JavaScript* и использовать <*canvas*>.

```
<script type="text/javascript">
    var canvas = document.createElement('canvas');
    canvas.id = 'canvas';
    document.body.appendChild(canvas);
    canvas.width = 1024;
    canvas.height = 576;
    var ctx = canvas.getContext('2d');
</script>
```

Полученный холст канваса и рисуется на web-форме.

```
<canvas id="canvas" width="1024" height="576"></canvas>
<script type="text/javascript" src="rayc.js"></script>
```

## ЗАКЛЮЧЕНИЕ

Для человека, который приходит в ранее неизвестное ему здание, сложно быстро определиться в пространстве. Будучи первокурсником, я не раз думал о том, что когда-нибудь обязательно смоделирую план помещений ИМИ так, чтобы это было понятно и максимально доступно человеку с улицы.

В здании ИМИ часто проводятся семинары, конференции и симпозиумы различных уровней. Интерактивный план помещений – хорошее подспорье гостям института чтобы разобраться в расположении помещений. Данный план обязательно должен быть включен в паспорт антитеррористической защищенности здания ИМИ. С помощью этого плана определиться с расположением помещений очень легко, поскольку он содержит не только схему размещения здания ИМИ по отношению к объектам инфраструктуры, но и конкретно расположение помещений института. Отдельным приложением для представителей МЧС, МВД и ФСБ станет интерактивный план помещений ИМИ и возможные критические и чрезвычайные ситуации в здании ИМИ в результате проведения диверсионно-террористических акций или экстремистских проявлений.

Впереди время абитуриентов. Не секрет, что каждый вуз бьется за новых студентов буквально зубами. Для того, чтобы максимально заинтересовать абитуриентов, привлечь в Институт математики и информатики большее количество желающих обучаться здесь, есть отличное решение, практически инновационная технология в действии. Самые передовые технологии, существующие на бумаге, в форме цифр, формул и тд. Ничто, в сравнении с интерактивным планом помещений в корпусе института математики и информатики.

Абитуриент, который приходит в ИМИ, не знаком с планом помеще-

ний, его интересуют самые элементарные вещи. Для него первоочередной задачей является знакомство с расположением аудиторий, кабинетов, гардероба, буфета и даже туалета. С этой задачей справится интерактивный план помещений корпуса ИМИ. Абитуриент, имея такой план, с легкостью разберется в здании ИМИ, будет иметь возможность свободно перемещаться в здании, что послужит дополнительным способом коммуникации в незнакомом месте.

Универсальность этого способа заключается в том, что данный план абитуриент будет видеть у себя на мобильном устройстве в виде приложения при входе в здание ИМИ. Также план будет размещен на сайте ИМИ для свободного пользования. Каждый компьютер члена приемной комиссии будет оснащен данным планом, чтобы помогать абитуриенту на первых порах.

Итогом всей работы стал готовый кроссплатформенный интерактивный план помещений, который в настоящее время активно внедряется для использования в ИМИ БГУ, а в дальнейшем распространяется и на весь БГУ. Приложение является кроссплатформенным и универсальным, и в этом его преимущество. Создание интерактивного плана с сохранением кроссплатформенности - это очень сложная в технологическом смысле задача, и я рад что я справился.

В чем уникальность метода? В его универсальности и доступности. В наш век высоких технологий каждый школьник считает себя выдающимся специалистом в области компьютерной техники. И поэтому, имея на мобильном устройстве такой план, школьник будет стремиться попасть на обучение именно в ИМИ.

Ссылка на сайт - <http://imi.bsu.ru/lps/projects/raycasting/>.

## Литература

1. Антонова Л.В., Бурзалова Т.В. Проективная геометрия : учеб. пособие.— Улан-Удэ : Бурятский государственный университет, 2016 .— 152 с.
2. Амерал Л. Принципы программирования в машинной графике. - М. : Сол Систем, 1992. - 224 с.
3. Амерал Л. Машинная графика на языке С. - М. : Мир, 1982. - 184 с.
4. Бересков, А.В. Шикин, Е.В. Компьютерная графика. – М.: Юрайт, 2016. – 220 с.
5. Божко, А.Н. Компьютерная графика: учеб. пособие - М.: Изд-во МГТУ им. Н. Э. Баумана, 2007. - 389 с.
6. Бьерн Страуструп. Язык программирования C++(3 издание). -СПб.: Невский Диалект, 2008. - 504 с.
7. Вельтмандер П.В. Машинная графика. Основные алгоритмы. Книга 2. – Новосибирск: НГУ, 1997. -197 с.
8. Геворкян П. С. Высшая математика. Линейная алгебра и аналитическая геометрия: учеб. пособие. —Москва: Физматлит, 2011. —204 с.
9. Гилой В. Интерактивная машинная графика. - М.: Мир, 1981.- 380с.
10. Д. Кнут. Искусство программирования (3 тома). - М.: ВИЛЬЯМС, 2011.
11. Дёмин А.Ю. Практикум по компьютерной графике: уч.пособие; - Томск: Томский политехнический университет, 2014. -120 с.
12. Ильин В. А., Позняк Э. Г. Аналитическая геометрия: уч. —Москва: ФИЗМАТЛИТ, 2009. —223 с.

13. Кадомцев С. Б. Аналитическая геометрия и линейная алгебра.  
—Москва: Физматлит, 2011. —167 с.
14. Киселев А.П. Геометрия: учебник; под ред. и с доп. Н. А. Глаголева.  
—Москва: Физматлит, 2013. —328 с.
15. Киселев А.П. Геометрия. Планиметрия. Стереометрия: учебник.  
—Москва: ФИЗМАТЛИТ, 2013. —228 с.
16. Кустодиева Б.М., Голлербах Э.Ф. Графика. —Москва: Лань, 2013
17. Котов Ю. В. Как рисует машина. — М.: Наука, 1988. — 224 с.
18. Курош А.Г. Курс высшей алгебры: учеб. —Москва: Лань, 2013. —431 с.
19. Ласло М. Вычислительная геометрия и компьютерная графика на C++. — М.: БИНОМ, 1997. — 304 с.
20. Хейфец А.Л., Логиновский А.Н., Буторина И.В., Васильева В.Н. Инженерная 3D-компьютерная графика. —М.: Издательство Юрайт, 2015. —602 с.
21. Лорд Э.Э., Маккей А.Л., Ранганатан С. Новая геометрия для новых материалов; пер. с англ. Л.П. Мезенцевой под ред. В.Я. Шевченко, В.Е. Дмитриенко. —Москва: Физматлит, 2010. —263 с.
22. Шорников А. Е., Бадагаров Д. Ж. «Кроссплатформенное приложение для 3d-визуализации 2d-модели плана помещения методом бросания лучей», стр. 14 / Материалы 55-й Международной научной студенческой конференции МНСК-2017: Информационные технологии / Новосиб. гос. ун-т. —Новосибирск: ИПЦ НГУ, 2017. — 250 с.
23. Макс Шлее. Qt 5.3. Профессиональное программирование на C++. -СПб.: БВХ-Петербург, 2015.

24. Мозговой М. В. 85 нетривиальных проектов, решений и задач на языке C++. - СПб.: НАУКА И ТЕХНИКА, 2007.
25. Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики - СПб.: БХВ-Петербург, 2003. - 554 с.
26. Павлидис Т. Алгоритмы машинной графики и обработки изображений: Пер. с англ. - М.: Радио и связь, 1986. – 400 с.
27. Петров М.Н. Компьютерная графика (3-е изд.). - СПб.: Питер. 2011. – 544 с.
28. Перемитина Т.О. Компьютерная графика: Учебное пособие. - Томск. Эль Контент, 2012. - 144 с.
29. Постников М. М. Аналитическая геометрия. Лекции по геометрии: учеб. пособие. —Москва: Лань, 2009. —414 с.
30. Привалов И. И. Аналитическая геометрия: учебник. —Москва: Лань, 2007. —304 с.
31. Приступа А.В. Компьютерная графика. Алгоритмические основы и базовые технологии. —Томск : Издательство научно-технической литературы, 2012. — 258 с.
32. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989. — С. 50-54
33. Роджерс Д., Адамс Дж. Математические основы машинной графики. - М.: Машиностроение, 1980. - 240с.
34. Романов Е. Л. Практикум по программированию на C++. -СПб.: БВХ-Петербург, 2004.
35. Саттер. Решение сложных задач на C++. 87 головоломных примеров с решениями. - М.: ВИЛЬЯМС, 2016.

36. Селезнев В.А., Дмитриченко С.А. Компьютерная графика. —М.: Издательство Юрайт, 2016. —228 с.
37. Удачная модель ветвления для Git:интернет-ресурса. URL: <https://habrahabr.ru/post/106912/>
38. Фоли Дж., Ван Дэм А. Основы интерактивной машинной графики. Кн. 1 и 2 - М.: Мир, 1985.
39. Хейфец А.Л., Логиновский А.Н., Буторина И.В., Васильева В.Н. Инженерная 3D-компьютерная графика. —М.: Издательство Юрайт, 2015. —602 с.
40. Хитерхеева Н. С. Компьютерная графика/Н. С. Хитерхеева, И. Л. Дульчаева, Ч. Мунхбаяр. —Улан-Удэ: Изд-во Бурят. госун-та, 2009. —107 с.
41. Чердынцев Е.С. Математические основы машинной графики. Томск: 1997. - 92с.
42. Чекмарев А.А. Инженерная графика. —М.: Издательство Юрайт, 2016. —381 с.
43. Чириков С. В. Алгоритмы компьютерной графики (Методы растртирования кривых). Учебное пособие — СПб: СПбГИТМО(ТУ), 2001. — 120 с.
44. Шелестов А.А. Компьютерная графика: Учебное пособие. Томск: ТУСУР, 2012. - 121 с.
45. Шикин Е. В. Начала компьютерной графики. - М. : Диалог-МИФИ, 1993. - 138 с.
46. Шилдт Г. С++: Базовый курс. -М.: ВИЛЬЯМС, 2007.
47. Joseph O'Rourke. Computational Geometry in C. — Cambridge University Press, 1998. — 362 с.

48. Kushner, David (2004-05-11). Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture. Random House.
49. Kent, Steven L. (2010-06-16). The Ultimate History of Video Games. Three Rivers Press.
50. Slaven, Andy (2002-07-01). Video Game Bible, 1985-2002. Trafford Publishing
51. Lode's Computer Graphics Tutorial. 2007. URL:  
<http://lodev.org/cgtutor/index.html>
52. Making a Basic 3D Engine in Java. 2009. URL:  
<http://www.instructables.com/id/Making-a-Basic-3D-Engine-in-Java/>
53. RAYCASTING - сделай себе немного DOOM'a. 2004. URL:  
<http://zxdn.narod.ru/coding/ig5ray3d.htm>

# *ПРИЛОЖЕНИЕ*

**Программный код приложения для реализации  
метода бросания лучей**