

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
БУРЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Институт математики и информатики

Кафедра прикладной математики

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Кроссплатформенное приложение для
3D-визуализации

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.
Введение	3
1 Постановка задачи	5
2 Математическая модель	6
2.1 Способ представления карты	6
2.2 Рейкастинг как алгоритм отрисовки	10
2.3 Алгоритм текстурирования	14
2.4 Паттерн для интерактивного взаимодействия	16
3 Реализация	19
3.1 Инструментарий	19
3.2 Организация работы над проектом	22
3.3 Структура проекта (UML)	23
3.4 Реализация на целевых платформах	24
3.5 Внедрение ссылка на сайт ИМИ БГУ	27
Заключение	28
Список литературы	29
Приложение 1	31

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).

Для построения трехмерной визуализации выбран метод бросания лучей. Данный метод, являясь одним из простейших для трехмерной отрисовки, имеет ряд свойств согласующихся с задачами проекта: малую вычислительную сложность и простую реализацию без использования готовых библиотек трехмерной 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.

Метод бросания лучей (англ. Raycasting, рейкастинг) - это один из ме-

тодов 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.

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 C++;

Объектом исследования является задача построения псевдотрёхмерной картинки.

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

Глава 1

Постановка задачи

Итак, перед нами встала задача: создание интерактивного плана помещения методом бросания лучей.

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.

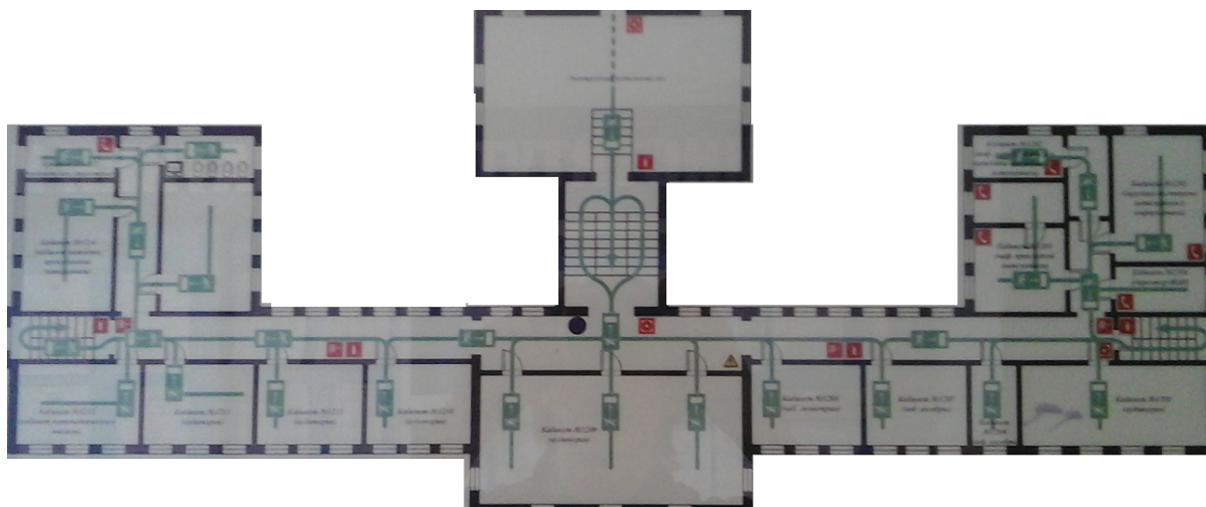


Рис. 2.1: Обработанное изображение 2 этажа 1 корпуса БГУ

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 внутренне формат *SVG* представляет собой *XML*-документ, содержащий помимо встроенной разметки координаты точек отрезков. Необходимо было только написать парсер для фильтрации полезной информации из *SVG*. Пример внутреннего 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>

```

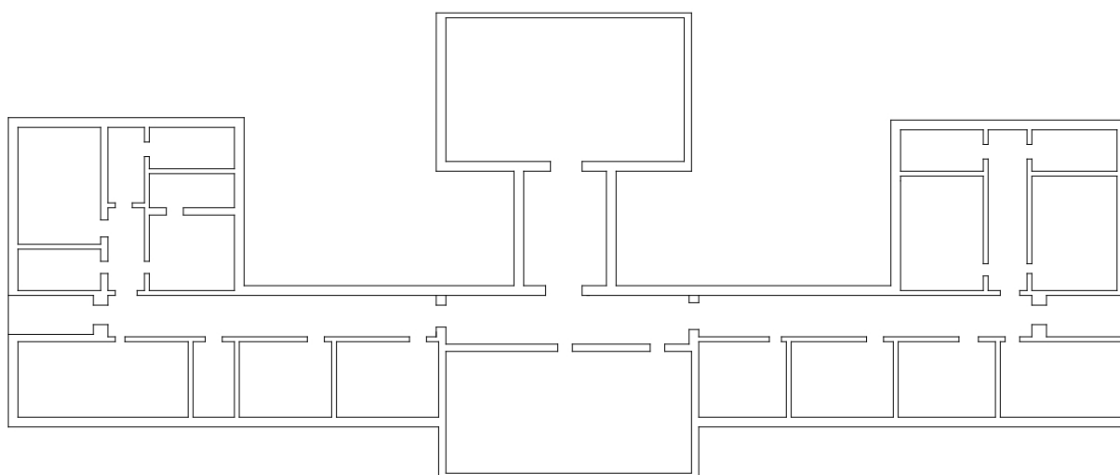


Рис. 2.2: Векторное представление изображения карты

В итоге, общая задача оцифровки 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 "скармливание" 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.

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.

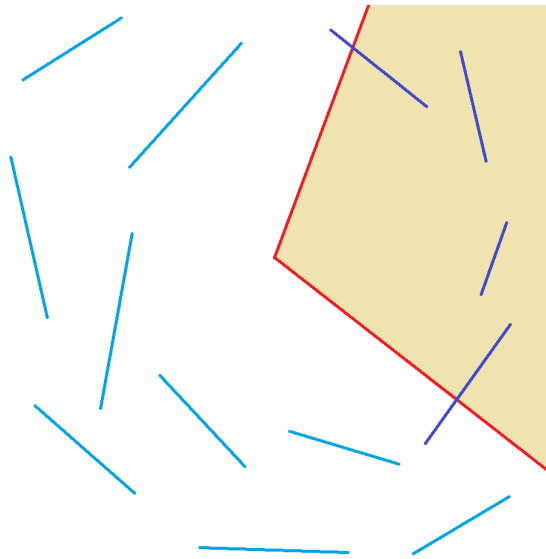


Рис. 2.3: Общий случай описания карты

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-

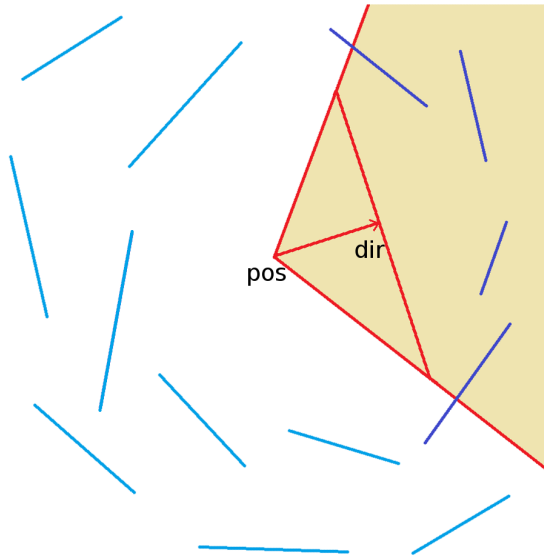


Рис. 2.4: Задание векторов

ризонталь разрешения экрана.

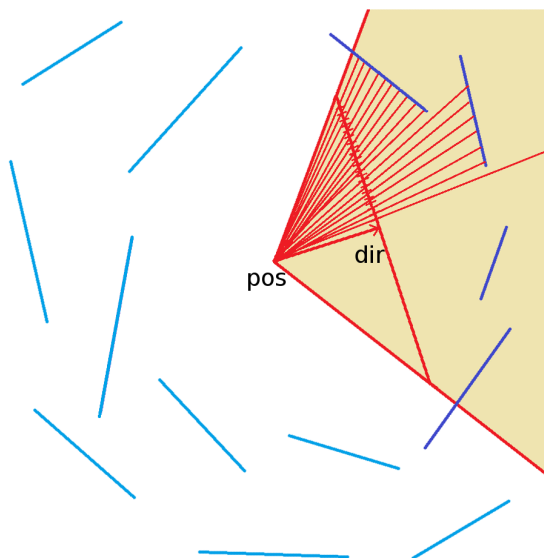


Рис. 2.5: Бросание лучей

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.

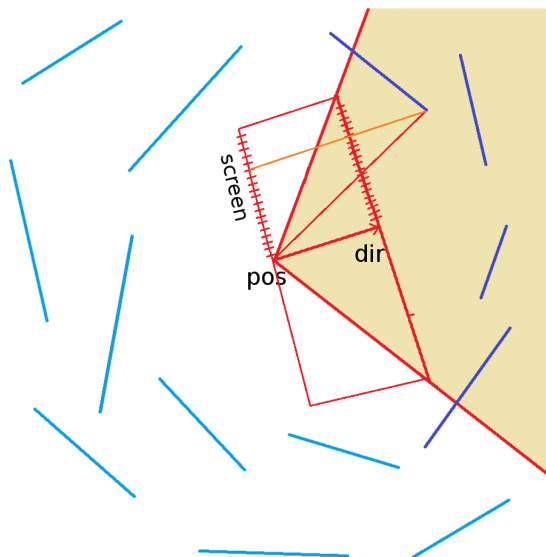


Рис. 2.6: Формирование изображения

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 изменялось 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.

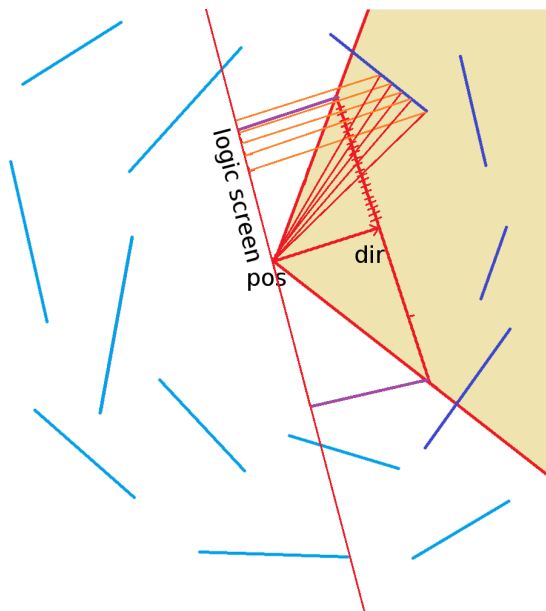


Рис. 2.7: Избавление от "эффекта рыбьего глаза"

2.3. 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 y 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 придуманном нами 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).

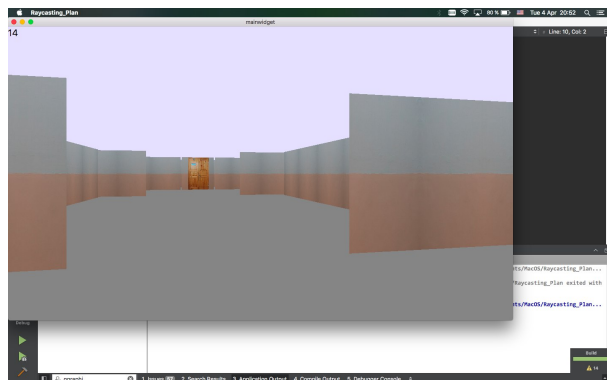


Рис. 2.8: Неоптимизованное текстурирование

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. Для этого после генерирования текстур необходимо поменять их значения x и y .

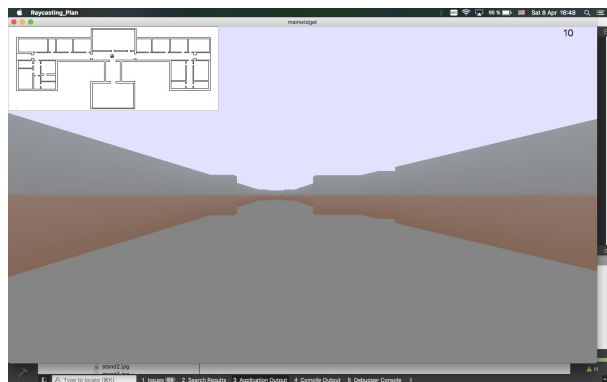


Рис. 2.9: Оптимизованное текстурирование

2.4. 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, изменению 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 Δdir .

Функция описывающая изменение координат называется *Update*. При вызове этой функции, которая получает новые координаты в зависимости от времени изменения кадров, 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 вышеперечисленных 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 широта 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 широта 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 отпуске клавиш изменения прекращаются, и координаты не меняются. Всё это проделывается каждый раз для каждого фрейма(кадра).

Глава 3

Реализация

3.1. Инструментарий

Для создания проекта был выбран язык программирования $C++14$ с фреймворком *Qt 5.8*. *Qt* это кроссплатформенный инструмент 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++*).

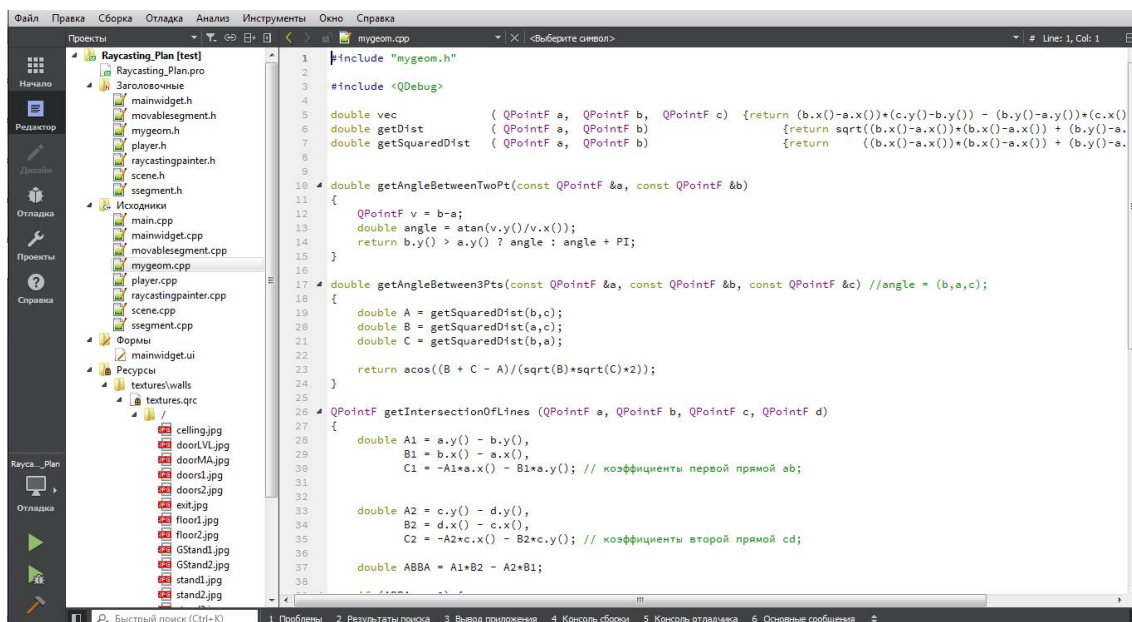


Рис. 3.1: Написание кода в Qt

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

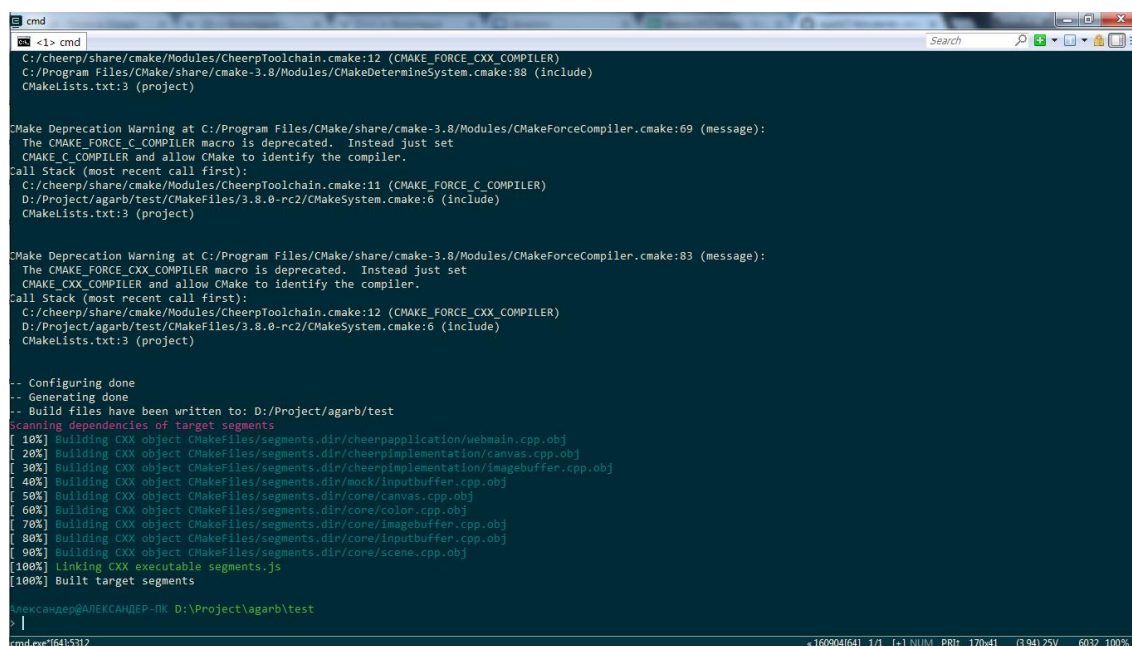
XML. *Qt* является полностью объектно-ориентированным, легко расширяемым и поддерживающим технику компонентного программирования. отличительная особенность *Qt* от других библиотек - использование *Meta Object Compiler* (*МОС*). *МОС* - компилятор - это предварительная система обработки исходного кода.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

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*.

Сам код на *Qt* был написан максимально платформозависимым, 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, пройденного лучём, а так же для работы с сегментами. Новые библиотеки получили имя *ssegment.h* и *mygeom.h*.

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'а 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
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
[ 16%] Building CXX object CMakeFiles/segments.dir/cheerpapplication/webmain.cpp.obj
[ 28%] Building CXX object CMakeFiles/segments.dir/cheerpimplementation/canvas.cpp.obj
[ 36%] Building CXX object CMakeFiles/segments.dir/cheerpimplementation/imagebuffer.cpp.obj
[ 44%] Building CXX object CMakeFiles/segments.dir/mock/inputbuffer.cpp.obj
[ 50%] Building CXX object CMakeFiles/segments.dir/core/canvas.cpp.obj
[ 60%] Building CXX object CMakeFiles/segments.dir/core/color.cpp.obj
[ 70%] Building CXX object CMakeFiles/segments.dir/core/imagebuffer.cpp.obj
[ 80%] Building CXX object CMakeFiles/segments.dir/core/inputbuffer.cpp.obj
[ 90%] Building CXX object CMakeFiles/segments.dir/core/scene.cpp.obj
[100%] Linking CXX executable segments.js
[100%] Built target segments

alexander@ALEKSKANDER-ПК 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.

3.2. Организация работы над проектом

3.3. Структура проекта (UML)

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, аудиоокнтент, 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 платформ вошли конкретные реализации канвы и буфера кадров. Сейчас специальная часть для платформ занимают лишь 30% от общей кодовой базы, и нет причин для увеличения этой части с ростом проекта.

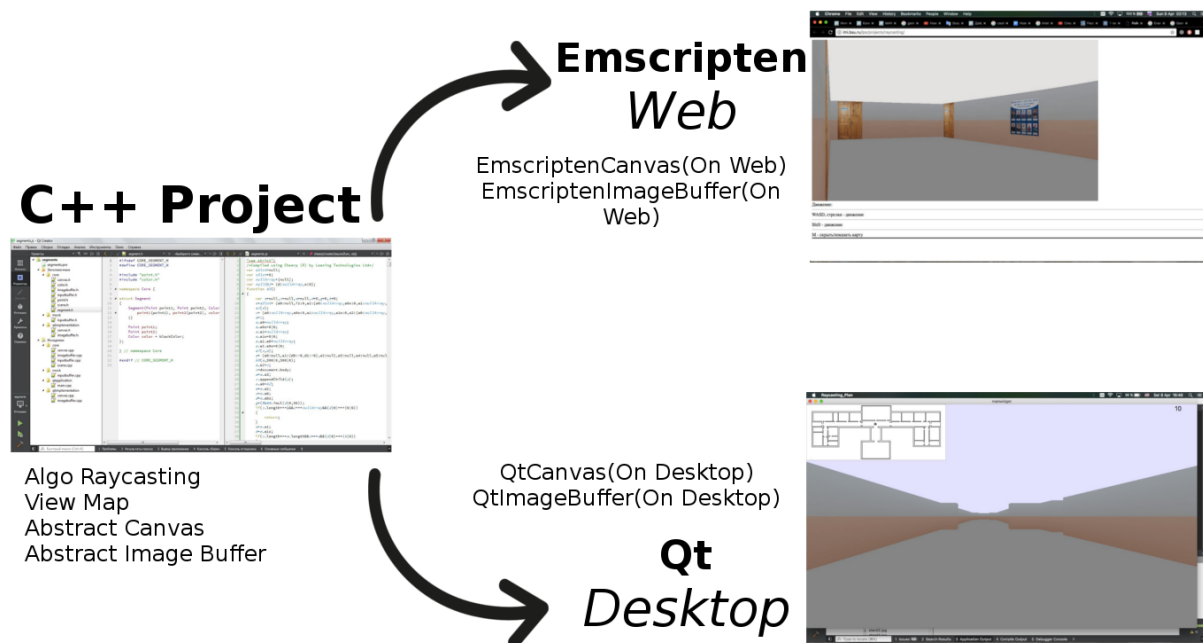


Рис. 3.3: Разделение конкретных реализаций

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

ствующий *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*.

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'а 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
```

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.

3.5. Внедрение ссылка на сайт ИМИ БГУ

Полученный проект

ЗАКЛЮЧЕНИЕ

Для рендеринга трёхмерной картинке существует много методов, однако, методов бросания лучей являются одними из самых простых и быстрых. С его помощью можно быстро и без особых усилий отрендерить псевдотрёхмерную картинку, получить результат быстро и без больших вычислительных возможностей.

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.

Но только БГУ не ограничивается применение полученного ПО. Благодаря удобному устройству задания уровней без проблем можно применить его к любому зданию, необходимо только сделать двумерный план помещения и сформировать текстуры. Во многом, технология рейкастинга и устарела. С её помощью нельзя сделать картинку "реалистичной пол и потолок всегда константной высоты, плохое и малоразмерное текстурирование и тд. Но несмотря на это, моя работа доказало его эффективность и в наше время.

Литература

1. Антонова Л.В., Бурзалова Т.В. Проективная геометрия : учеб. пособие.— Улан-Удэ : Бурятский государственный университет, 2016 .— 152 с.
2. Бьерн Страуструп. Язык программирования C++(3 издание). -СПб.: Невский Диалект, 2008. - 504 с.
3. Вельтмандер П.В. Машинная графика. Основные алгоритмы. Книга 2. – Новосибирск: НГУ, 1997. -197 с.
4. Котов Ю. В. Как рисует машина. — М.: Наука, 1988. — 224 с.
5. Ласло М. Вычислительная геометрия и компьютерная графика на C++. — М.: БИНОМ, 1997. — 304 с.
6. Макс Шлее. Qt 5.3. Профессиональное программирование на C++. -СПб.: БВХ-Петербург, 2015.
7. Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики - СПб.: БХВ-Петербург, 2003. - 554 с.
8. Павлидис Т. Алгоритмы машинной графики и обработки изображений: Пер. с англ. - М.: Радио и связь, 1986. – 400 с.
9. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989. — С. 50-54
10. Чириков С. В. Алгоритмы компьютерной графики (Методы растривания кривых). Учебное пособие — СПб: СПбГИТМО(ТУ), 2001. — 120 с.
11. Joseph O'Rourke. Computational Geometry in C. — Cambridge University Press, 1998. — 362 с.

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

П Р И Л О Ж Е Н И Е 1

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