Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «Кубанский государственный технологический университет»

(ФГБОУ ВО «КубГТУ»)

Институт\_компьютерных систем и информационной безопасности (ИКСиИБ)\_\_\_\_\_\_\_\_\_

Кафедра\_\_информационных систем и программирования\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Направление подготовки /специальность 09.03.04. Программная инженерия\_\_\_\_\_\_\_\_

*(код и наименование направления подготовки/специальности)*

Профиль/специализация\_\_Проектирование и разработка программного обеспечения\_\_\_\_

*(наименование профиля/специализации)*

**КУРСОВОЙ ПРОЕКТ**

по дисциплине\_\_\_Алгоритмы компьютерной графики\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(наименование дисциплины)*

на тему: «Разработка Quake-подобного игрового движка на OpenGL»\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_

*(тема курсовой работы)*

Выполнила студентка Орехова София Михайловна курса\_3\_\_группы 19-ЗКБс-ПР1\_\_\_\_\_

*(фамилия, имя, отчество)*

Допущена к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата)

Руководитель (нормоконтролер) работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ст.преп. Урвачев П.М.

*(должность, подпись, дата)*

Защищена\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(дата)*

Члены комиссии:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_к.т.н., доцент Попова О.Б.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_к.т.н., доцент Тотухова К.Е.

*(должность, подпись, дата, расшифровка подписи)*

Краснодар

2022 г.

ФГБОУ ВО «Кубанский государственный технологический университет»

(ФГБОУ ВО «КубГТУ»)

Институт\_компьютерных систем и информационной безопасности (ИКСиИБ)\_\_\_\_\_\_\_\_\_

Кафедра\_\_информационных систем и программирования\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Направление подготовки /специальность 09.03.04. Программная инженерия\_\_\_\_\_\_\_\_

*(код и наименование направления подготовки/специальности)*

Профиль/специализация\_\_Проектирование и разработка программного обеспечения\_\_\_\_

*(наименование профиля/специализации)*

УТВЕРЖДАЮ

Зав. кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2022г.

**ЗАДАНИЕ**

на курсовой проект

Студентуке\_Ореховой Софии Михайловны\_\_ курса\_\_3\_\_ группы\_19-ЗКБс-ПР1 \_

Тема проекта/работы: «Разработка Quake-подобного игрового движка на OpenGL»

(утверждена указанием директора института № \_ от \_\_.\_\_ 2022 г.)

План проекта/работы:

1. Изучить возможности OpenGL на основе игры Quake;

2. Выполнить доработку в реализации камеры объекта;

3. Изучить и применить на практике метод коллизии объектов;

4. Установить управление, согласно трехмерной графике.

Объем проекта/работы:

а) пояснительная записка\_30 с.

Рекомендуемая литература:

1. Баранов, С.Н. Основы компьютерной графики : учеб. пособие / С.Н. Баранов, С.Г. Толкач. - Красноярск : Сиб. федер. ун-т, 2018. - 88 с. - ISBN 978-5-7638-3968-5. - Текст : электронный.
2. Вольф, Д. OpenGL 4. Язык шейдеров. Книга рецептов / Д. Вольф ; пер. с англ. А.Н. Киселева. - Москва : ДМК Пресс, 2015. - 368 с. - ISBN 978-5-97060-255-3. - Текст : электронный.

Срок выполнения: с «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_ по «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

Срок защиты: «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

Дата выдачи задания: «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

Дата сдачи проекта/работы на кафедру: «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

Руководитель проекта/работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ст.преп. Урвачев П.М.

(*должность, подпись)*

Задание приняла студентка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Орехова С.М.

*(подпись)*

ФБГОУ ВО «Кубанский государственный технологический университет»

(ФГБОУ ВО «КубГТУ»)

Кафедра Информационных систем и программирования

Институт Компьютерных систем и информационной безопасности

**Реферат**

Пояснительная записка курсового проекта 30 с., 13 рис., 2 табл., 5 источника, 5 прил.

ГРАФИЧЕСКАЯ СИСТЕМА, КОМПЬЮТЕРНАЯ ГРАФИКА, ТЕХНОЛОГИЯ OpenGL, ИГРОВОЙ ДВИЖОК, QUAKE ENGINE

В данном курсовом проекте решена задача создания графического движка трехмерного пространства с применением технологии OpenGL, на основе quake-подобного игрового движка.

Проработаны и реализованы физические зависимости в игровой среде с применением графической реализации. Добавлена возможность управления игровым персонажем.

Созданный игровой движок позволил закрепить навыки применения технологий компьютерной графики на практике.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 5](#_Toc104995806)

[Термины, определения и сокращения 7](#_Toc104995807)

[1 Анализ проектируемого Quake-подобного игрового движка 8](#_Toc104995808)

[1.1 Принцип разработки 2.5D и 3D пространства на примере игры Quake, и версий разработанных движков поколения Quake 8](#_Toc104995809)

[1.2 Эволюция разработки рендринга компьютерной графики 10](#_Toc104995810)

[2 Разбор и применение графических методов 13](#_Toc104995811)

[2.1 Выбор пакета аппаратно-программной прорисовки OpenGL 13](#_Toc104995812)

[2.2 Использование двоичного разбиения пространства или Binary space partitioning, дополнение к разбрасыванию лучей 13](#_Toc104995813)

[2.3 Техническое задание проекта 15](#_Toc104995814)

[3 Разработка проекта графической системы 17](#_Toc104995815)

[3.1 Изучение и использование свободных библиотек для конструкции уровня 17](#_Toc104995816)

[3.2 Доработка и внесение изменений к движку 18](#_Toc104995817)

[3.2.1 Обработка камеры 18](#_Toc104995818)

[3.2.2 Обнаружение столкновения с объектом 20](#_Toc104995819)

[3.3 Создание местности и рендер мира 22](#_Toc104995820)

[3.4 Установка управления 29](#_Toc104995821)

[ЗАКЛЮЧЕНИЕ 33](#_Toc104995822)

[Список использованных источников 34](#_Toc104995823)

[ПРИЛОЖЕНИЕ А Код части программы – Camera 35](#_Toc104995824)

[ПРИЛОЖЕНИЕ Б Код коллизии объектов 43](#_Toc104995825)

[ПРИЛОЖЕНИЕ В Рендер мира 48](#_Toc104995826)

[ПРИЛОЖЕНИЕ Г Управление 53](#_Toc104995827)

[ПРИЛОЖЕНИЕ Д Результаты выполненного проекта 57](#_Toc104995828)

ВВЕДЕНИЕ

Изучение дисциплины и самого курса компьютерной графики основывается на исследовании основных алгоритмов, на которых строится базовое построение машинной графики. Возникнув из потребностей рынка, развития информатики и вычислительной техники, компьютерная графика изучает методы построения изображений различных геометрических объектов и сцен.

Главными этапами построения изображения являются:

* моделирование математически описанного объекта и сцены;
* визуализация построения изображения объемного мира.

Целью данного курсового проекта является изучение такого понятия, как «игровой движок», и применить полученные знания для реализации quake-подобного игрового движка. Для достижения цели необходимо изучить методы обработки трехмерной графики в реальном времени, и применить основные технические характеристики, а так же главный алгоритм используемый для рендеринга сцены – рейкастинг или метод «бросания лучей».

Следовательно, задачи курсового проекта можно обозначить в следующих пунктах:

* изучить методы разработки quake engine для применения паттернов при разработке собственного движка;
* поэтапно выстроить сцену для реализации выбранного алгоритма:

1. применить первый этап: пробросить лучи для отображения препятствий;
2. применить второй этап: внести отображения «шейдеров» для препятствий;

* обозначить «нижнее» ограничение и «верхнее», прописать основные выполняемые действия;
* обозначить основное управления на сцене.

В рамках курсового проекта будет использоваться графическая библиотека OpenGL, согласно ранее выданному заданию. Разработка игрового движка будет осуществляться на языке программирования C++.

Так же, дополнительно необходимо отметить, что при разработке сцены при использовании графики OpenGL, используется долго поддерживающее расширение FreeGlut, которое работает с С-подобным языком, как в данном случае Си.

Актуальность данной работы заключается в исследовании методов для разработки собственного движка, основываясь на конкретных паттернах ранее революционных идей, которые на текущий момент требуют доработки при условии развитии алгоритмических возможностей и компьютерной техники.

Термины, определения и сокращения

BSP, binary space partitioning – метод двоичного разбиения пространства;

Raycasting – метод пробрасывания лучей;

1 Анализ проектируемого Quake-подобного игрового движка

1.1 Принцип разработки 2.5D и 3D пространства на примере игры Quake, и версий разработанных движков поколения Quake

Перед объяснением феномена псевдотрехмерной графики и полноценной трехмерной, необходимо определить, на чем основываются данные законы. Главной основой является понятие евклидова пространства, которое подразумевает наличие размерности равное трем, то есть трехмерное.

В основе игрового движка Quake или Quake engine лежит основа как псевдотрехмерной графики, так и основа полноценной 3D графики. Данное утверждение верно в отношении первого поколения разработанных игровых движков для DOOM, который использовал полноценный ray casting и малую часть BSP-деревьев, основа которых лежит в разбиении основного пространства на две плоскости. Данный принцип тянется из математической модели евклидова пространства. Основными показателями, при которых считается что метод ray casting, или пробрасывание лучей, и BSP-деревья относятся к методам трехмерной компьютерной графики – это определение столкновения и сортировка визуальных объектов.

Считать рейкастинг полноценным методом рендеринга, как 2.5D пространства нельзя, поскольку существуют принципиальные различия. Такие как, пример передвижение игрока лишь по двум осям, но и рейкастинг не поддерживает перемещение вверх и вниз, тогда как полноценная трехмерная графика подразумевает и данное передвижение. Исходя из данного факта, можно выделить следующее - рейкастинг является псевдотрехмерной графикой от первого лица, в которой лучи пробрасываются для каждого вертикального среза экрана, это выглядит следующим образом, представлено на рисунке 1.

Дополнительно следует отметить, что метод с векторами работает в том плане, при котором обязательным является использование 2D карты, которая является полноценной плоскостью для реализации пространства.

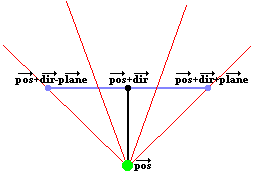


Рисунок 1 – Пробрасывание лучей по вертикали

Исходя из вышеописанных данных, следует рассмотреть маппинг, и где именно происходит передвижение в псевдопространстве, данное необходимо для рассмотрения следующего метода, который используется как в open source Quake engine, так и при проектировании курсового проекта.

В плане маппинга, в 2.5-мерном пространстве строится на 2D карте, которая и формирует полноценный уровень, пример на рисунке 2.

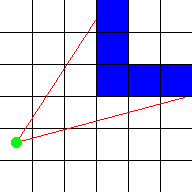


Рисунок 2 – Маппинг

Двумерная карта является как визуализацией уровня, так и формируемым пространством.

Далее необходимо вернуться, и полноценно дать определение рейкастингу, как основополагающему методу при разработке.

Основная идея рейкастинга заключается в следующем: карта представляет собой двухмерную квадратную сетку, и каждый квадрат может иметь либо 0 (= отсутствие стены), либо положительное значение (= стена с определенным цветом или текстурой). При разработке примера, в данном случае, шейдер использоваться не будет (добавление текстур для рендеринга).

Для каждого N экрана (то есть для каждой вертикальной полосы экрана) отправляется луч, который начинается в местоположении игрока и имеет направление, которое зависит как от направления взгляда игрока, так и от координаты N экрана. Следующим этапом, прописанными командами по 2D-карте позиция player или луч трассировки передвигается по плоскости, пока он не столкнется с квадратом карты, который является стеной. Если он ударился о стену, рассчитайте расстояние от этой точки попадания до игрока, то это расстояние используется для того, чтобы рассчитать, насколько высоко эта стена должна быть отрисована на экране: чем дальше стена, тем меньше она на экране и тем ближе, тем выше он кажется. Это все касается 2D расчетов, которые являются основными методом построения плоскости при управлении игроком.

Сама разработка игры Quake полноценно использует следующее поколение рендеринга, не основанного на разбиении двоичного бинарного дерева, то есть рассматривается пространство не как плоскость на 2D карте, а полноценное евклидово пространство.

1.2 Эволюция разработки рендринга компьютерной графики

Под эволюцией редринга используется основное понятие, как изменение графики из псевдотрехмерной в полноценную трехмерную графику. Данное стало возможным после применения рендеринга на 2D карте, как это рассматривалось ранее, в предыдущем пункте.

Рассматриваемая трехмерная игра, в которой происходит игра, называется картой или maps, то есть, сейчас она не является двумерной, как ранее. Программа редактора карт использует ряд простых выпуклых трехмерных геометрических объектов, известных как кисти или brushes, которые масштабируются и поворачиваются для создания среды. Кисти размещаются и ориентируются для создания замкнутого, пустого, объемного пространства, и когда дизайн завершен, карта проходит через препроцессор рендеринга. Препроцессор используется для обнаружения двух типов пустого пространства на карте: пустого пространства, окруженного кистями, где будет проходить игра, и другого пустого пространства за пределами кистей, которое игрок никогда не увидит. Затем препроцессор удаляет задние поверхности отдельных кистей, которые находятся за пределами игрового пространства, оставляя только несколько полигонов, определяющих внешний периметр закрытого игрового пространства.

Как правило после того, как карта была предварительно обработана, ее нельзя повторно отредактировать обычным образом, поскольку исходные кисти были разрезаны на мелкие кусочки. Вместо этого исходные данные редактора карт с кистями сохраняются и используются для создания новых версий карты. Но можно редактировать обработанную карту, открыв ее в специальном редакторе вершин и отредактировав исходные данные вершин, или добавить или удалить отдельные грани треугольников.

Этот этап предварительной обработки не может работать, если есть какие-либо небольшие дыры или «утечки», которые соединяют внутреннее игровое пространство с внешним пустым пространством, и часто сложные проекты по построению карты отбрасывались, потому что дизайнер карты не мог найти утечку в их карта. Во избежание протечек щетки должны перекрываться и слегка проникать друг в друга; попытка идеально выровнять края кистей необычной формы на сетке может привести к очень маленьким зазорам, которые трудно обнаружить.

Открытое небо в картах Quake на самом деле не открытое, а покрыто и окружено большими кистями и текстурировано специальной текстурой скайбокса, которая запрограммирована на использование сферического отображения, и поэтому всегда выглядит одинаково с любой точки зрения, давая иллюзия далекого неба, пример для разрабатываемого проекта на рисунке 3.

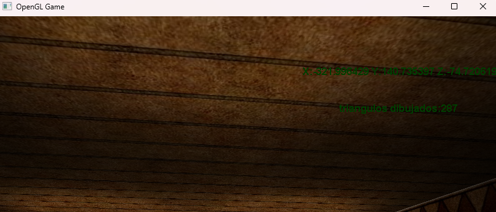


Рисунок 3 – Пример легкого использования «неба»

На рисунке 3 полноценно можно рассмотреть, что в качестве полноценного неба используется иллюзия, поскольку в рамках курсового проекта не предусмотрена работы с brushes, по этой причине будут использоваться стандартный набор свободно распространяемых текстур из одного из уровней игры Quake.

2 Разбор и применение графических методов

2.1 Выбор пакета аппаратно-программной прорисовки OpenGL

Проектирование игрового движка на свободно распространяемой библиотеки OpenGL была выбрана в следствии больших возможностей данной библиотеки, как в работе с освещением и рассеиванием света, поскольку основной задачей используемого алгоритма, как ray casting, так и его последующей эволюции в трехмерной графике – BSP, применяется работа со светом на генерируемой карте.

2.2 Использование двоичного разбиения пространства или Binary space partitioning, дополнение к разбрасыванию лучей

Основой двоичного разбиения послужила разработка предыдущей версий игр Doom и Quake – пробрасывание лучей. Данный метод модифицирован в стандартных библиотеках, в основу которых берется разработка рендеринга одного уровня (уровень в плоскости карты). Поскольку задача в курсовом проекте стоит иная, следует отметить, что в рамках разработки необходимо отдельно отметить вопрос об управлении, так как трехмерный движок, от разрабатываемого примера, управляется не только на клавиатуре, но и с использованием мыши – отличительная особенность поколения движков от предыдущих.

Если рейкастинг использует спрайты, то трехмерный мир использует полноценную модель и текстуры. Таким образом, необходимо определить пространство мира, и по какому принципу будет работать столкновение объекта (игрока) с другими объектами из пространства.

Так же перед проектированием отмечается следующая модель: по какому принципу игра будет формировать при загрузке мир.

На рисунке 4 представлена краткая схема отработки алгоритма при запуске движка:

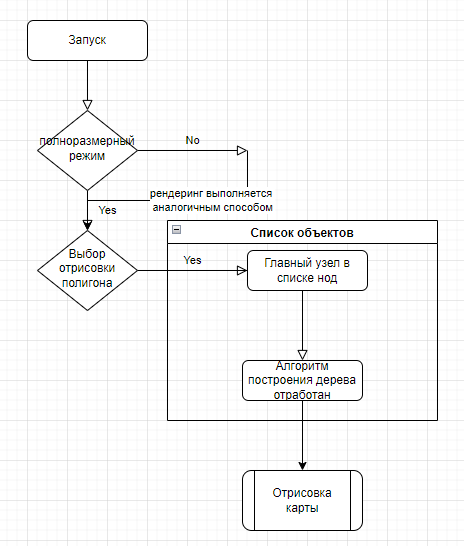


Рисунок 4 – Алгоритм отрисовки

Поскольку quake-подобный движок строится на алгоритме BSP-деревьев, как и для любого другого движка основой является построение бинарного дерева, то перед запуском в программе необходимо уточнение полно размерности. Связано это с количеством необходимых разбиений. Далее по алгоритму выбирается полигон, от которого будет проходить главный узел дерева при использовании гиперплоскостей. Таким образом, пространство, заложенное в саму сцену, будет разделено на два подпространства.

2.3 Техническое задание проекта

К техническому заданию проекта стоит отнести следующие проблемы при создании полноценного движка на основе имеющейся библиотеки:

1. Оптимизировать камеру таким образом, чтобы она вычисляла коллизию ;
2. При образовании коллизии – объект отталкивается;
3. При рендере мира, местность формируется на полноценной основе формирования бинарных деревьев – по фрагментно.

Три основных положения, которые позволят при разработке ориентироваться к доработке самой системе игрового движка.

На рисунке 5 показан процесс обращения между объекта мира к модели и обратно.

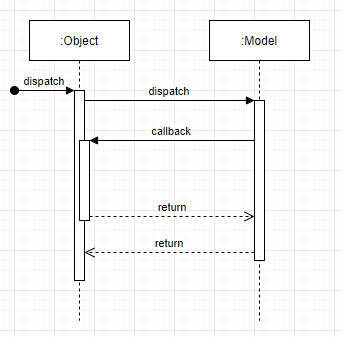


Рисунок 5 – Процесс обращения к объекту и модели

Данная схема описывает состояние объекта к модели. Что подразумевает следующее: от входной точки положения (переменной) объект осуществляет обращение к поиску моделей, обратное возвращение полученных данных от модели бинарного дерева. Return обращает связь.

Разбиение двоичного дерева осуществляется по следующему принципу: существует две плоскости – вертикальная и горизонтальная. Для проявления объекта на плоскости и проявления самой местности, алгоритм разбивает появление на фрагменты. Следовательно, сам граф будет иметь примерный вид:



Рисунок 6 – Образование фрагментов в графе

Каждый фрагмент будет проводится по поиску в графе, где начальный параметр – это входной или главный граф. К входному или начальному графу будет относится позиция главной модели.

3 Разработка проекта графической системы

3.1 Изучение и использование свободных библиотек для конструкции уровня

Первая такая библиотека с дополнительным кодом на cpp рассматривается в вопросе о стандартной Q3Loader, которая входит в стандартную разработку Quake. В данной библиотеке определяются такие моменты, как: флаги и задание переменной к вызову из хедера, при обработке кода. Поскольку сам файл cpp включает определенную модель подгрузки, которую будем использовать для системы.

Q3loader простой загрузчик карт Quake BSP. Файлы Quake 3 BSP это файлы IBSP, и поэтому имеют структуру, аналогичную предыдущим файлам BSP от id Software. Каждый файл IBSP начинается с заголовка, который, в свою очередь, содержит общий каталог. Каталог lump описывает расположение остальной части файла, который содержит некоторое количество lump. Каждый lump хранит определенный тип картографических данных.

Таблица 1 – Структура файла

|  |
| --- |
| Header / Directory |
| Lump |
| Lump |
| Lump |
| …. |

Так же стоит отметить, что файлы Quake 3 BSP содержат только четыре основных типа данных.

Таблица 2 – Типы данных

|  |  |
| --- | --- |
| Тип | Описание |
| ubyte | unsigned byte |
| int | 4-байтовое целое, обратный порядок байтов |
| float | 4-байтовый IEEE с плавающей запятой, обратный порядок байтов |
| string[n] | строка из n байтов ASCII, не обязательно заканчивающаяся нулем |

Основное понимание данных файлов, поможет разработать собственную сцену и включить основное понимание к доработке собственного мира.

3.2 Доработка и внесение изменений к движку

3.2.1 Обработка камеры

Изначальная разработка будет вносится к отделу world. Поскольку данный хедер распространяется свободно, то разработка к нему world.cpp составляет перенос bsp карты в сцену.

Исходя из доработок – первое, что необходимо сформировать – camera. Камера объекта сцены. Поскольку в стандартной библиотеке записаны хедеры camera и object, то camera.cpp будет содержать в себе именно их. Это связано с тем, что в данных библиотеках прописаны варианты передвижения объекта по времени и по его просмотру, то есть :

Листинг 1 – Методы камеры по просмотру и передвижению

void CCamera::UpdateLookAt()

{

CVector look = CVector(finalLookAt.x - lookAt.x,

finalLookAt.y - lookAt.y,

finalLookAt.z - lookAt.z);

lookAtVel = look \* 0.5;

}

void CCamera::UpdateMoveTo()

{

CVector pos = CVector(finalPosition.x - position.x,

finalPosition.y - position.y,

finalPosition.z - position.z);

velocity = pos \* 0.5;

Первый метод CCamera::UpdateLookAt принимает от класса обновление видимости объектов в зависимости от позиции объекта, когда второй от положения : pos \* 0.5.

Подметоды принимающие полноценное положение рассчитанные в листинге 1, обновляют позицию в камере:

Листинг 2 – Полноценное обновление объектов по времени

// просмотр объекта во времени

void CCamera::LookAt(CObject \*object)

{

CVector v = CVector(object->position - lookAt);

initLookAt = lookAt;

finalLookAt = object->position;

lookAtAccel = -lookAt \* 0.25f;

UpdateLookAt();

}

// двигаться к объекту с течением времени

void CCamera::MoveTo(CObject \*object)

{

CVector v = CVector(object->position - position);

initPosition = position;

finalPosition = object->position;

acceleration = -position \* 0.25f;

UpdateMoveTo();

}

3.2.2 Обнаружение столкновения с объектом

Обнаружение столкновения с миром неотъемлемая часть в 3D представлениях и трехмерных движках. В оптимизации данного движка используется пространственное разбиение. Именно пространственное разбиение, BSP дерево используется в обнаружении столкновений.

При поиске столкновений BSP-дерево используется для поиска плоскости, расположенной ближе всего к объекту. Чаще всего границы объекта задаются ограничивающий сферой (или окружностью) для упрощения вычислений. Выполняется обход BSP-дерева от корня до плоскости, расположенной ближе всего к объекту.

Листинг 3 – Анимация

void CEntity::OnAnimate(float deltaTime)

{

float cosYaw = (float)cos(DEG2RAD(direction));

float sinYaw = (float)sin(DEG2RAD(direction));

float speed = velocity.z \* deltaTime;

if ((direction >= 360.0f) || (direction <= -360.0f))

direction = 0.0f;

position.x += float(cosYaw)\*speed;

position.z += float(sinYaw)\*speed;

deltaT = deltaTime;

}

На листинге 3 представлена анимация обнаружения, данный алгоритм работает на основе определения позиции и позиции точки на плоскости и предоставляет саму анимацию объекта: отталкивается от точки на плоскостях.

Самостоятельный метод коллизии выведен в следующий метод:

void CEntity::OnCollision(CObject \*collisionObject)

{

if (typeid(\*collisionObject) == typeid(CEntity))

{

modelState = MODEL\_IDLE;

velocity = CVector(0.0, 0.0, 0.0);

}

if (typeid(\*collisionObject) == typeid(CTerrain))

{

position.y = ((CTerrain\*)collisionObject)->GetHeight(position.x, position.z) + size;

}

}

Где на объекте просматривается самостоятельная точка в двух плоскостях. Позиция точки определяется сравнением позиции объекта к позиции самой местности.

3.3 Создание местности и рендер мира

Под применением текстур объекта подразумевается формирование классов и методов, которые отвечают за выведение BSP моделей и обычных текстур на сформированные локации сцены.

Подгрузка местности сформирована на вызове стандартных методов, обозначенных из библиотеки Quake:

// подгрузка местности

terrainTex[0].LoadTexture("textures/ground.tga");

glGenTextures(1, &terrainTex[0].texID);

glBindTexture(GL\_TEXTURE\_2D, terrainTex[0].texID);

То есть glBindTexture метод, который принимает индексы текстур.

Сам метод BuildTerrain наследуется из класса CTerrain::BuildTerrain, где принимает следующие методы:

width = w;

heightMap = new float[width\*width];

MakeTerrainPlasma(heightMap, width, rFactor);

Каждый метод определяет размерность карты.

Сам метод рендеринга мира так же передается от класса CTerrain, только если при создании мира участвовал чисто прописанный образец передачи метода с дополнительными заголовками текстур.

Тогда как при рендеринге используется так же наследуемый метод, который определяет в себя положение камеры, которая в свою очередь рассчитывает положение главного объекта и вопрос коллизии.

Листинг 4 – Рендер мира

void CTerrain::OnDraw(CCamera \*camera)

{

for (z = (int)(camera->position.z / terrainMul - scanDepth), z=z<0?0:z; (z < camera->position.z / terrainMul + scanDepth) && z < width-1; z++)

{

glBegin(GL\_TRIANGLE\_STRIP);

for (x = (int)(camera->position.x / terrainMul - scanDepth); (int)(x < camera->position.x / terrainMul + scanDepth); x++)

for (x = (int)(camera->position.x / terrainMul - scanDepth), x=x<0?0:x; (x < camera->position.x / terrainMul + scanDepth) && x < width-1; x++)

{

glTexCoord2f(textureMul \* x, textureMul \* z);

glVertex3f((float)x\*terrainMul, (float)heightMap[x + z\*width]\*heightMul, (float)z\*terrainMul);

glTexCoord2f(textureMul \* (x+1), textureMul \* z);

glVertex3f((float)(x+1)\*terrainMul, (float)heightMap[x+1 + z\*width]\*heightMul, (float)z\*terrainMul);

glTexCoord2f(textureMul \* x, textureMul \* (z+1));

glVertex3f((float)x\*terrainMul, (float)heightMap[x + (z+1)\*width]\*heightMul, (float)(z+1)\*terrainMul);

glTexCoord2f(textureMul \* (x+1), textureMul \* (z+1));

glVertex3f((float)(x+1)\*terrainMul, (float)heightMap[x+1 + (z+1)\*width]\*heightMul, (float)(z+1)\*terrainMul);

}

glEnd();

}

Первоначально задаются переменные, которые будут принимать на себя запись значений положения, которая передается к самой камере. Поскольку от камеры зависит положение генерируемого дерева в рассмотрении данных бинарных деревьев, то рассчет положения объекта и объектов на плоскостях одно из необходимых к формированию местности. На сколько было упомянуто ранее, сама местность образуется моментально, а именно не зависит от камеры. Что не касается текстур.

Поскольку сам мир и подгрузка карт основана на BSP деревьях, то следующим этапом является формирование world.cpp, в котором обновляется модель мира, взятая из свободных библиотек.

Сама конструкция CWorld::CWorld(CCamera \*c) наследуема от класса из заголовка. Данный заголовок, в формировании и рендеринге мира, управляется в последнюю очередь. Так как в работе необходимо обозначить формирование камеры и объектов местности. Самостоятельный класс мира основан на самих подгрузках, полностью представлено в Приложении А. Аналогично и для прорисовки сцены.

К поведению главного объекта в мире относится и хедер md2, который вносит в себя такие моменты, как стандартные операции и фреймы из glut (свободно распространяемая библиотека OpenGL). Данные фреймы образуют на основе ранее построенной модели рейкастинг образование мира и поведение камеры («эффект рыбий глаз»).

За данный класс стоит обратиться к полноценному методу, который занимается непосредственной загрузкой:

int CMD2Model::Load(char \*modelFile, char \*skinFile)

Метод вытаскивает модель файла, образуемая в CMD2, так же как и сам skin.

Поскольку сама библиотека ссылается и на объект, и на текстуры подгружаемые к местности, то перед задачей данного файла – подгрузить модель:

Листинг 5 – Загрузка модели местности

// открыть моедль

filePtr = fopen(modelFile, "rb");

if (filePtr == NULL)

return false;

// найти длину файла

fseek(filePtr, 0, SEEK\_END);

fileLen = ftell(filePtr);

fseek(filePtr, 0, SEEK\_SET);

// прочитать весь файл в буфер

buffer = new char [fileLen+1];

fread(buffer, sizeof(char), fileLen, filePtr);

// извлечь заголовок файла модели из буфера

modelHeader = (modelHeader\_t\*)buffer;

vertexList = new vector\_t [modelHeader->numXYZ \* modelHeader->numFrames];

numVertices = modelHeader->numXYZ;

numFrames = modelHeader->numFrames;

frameSize = modelHeader->framesize;

for (j = 0; j < numFrames; j++)

{

frame = (frame\_t\*)&buffer[modelHeader->offsetFrames + frameSize \* j];

vertexListPtr = (vector\_t\*)&vertexList[numVertices \* j];

for (i = 0; i < numVertices; i++)

{

vertexListPtr[i].point[0] = frame->scale[0] \* frame->fp[i].v[0] + frame->translate[0];

vertexListPtr[i].point[1] = frame->scale[1] \* frame->fp[i].v[1] + frame->translate[1];

vertexListPtr[i].point[2] = frame->scale[2] \* frame->fp[i].v[2] + frame->translate[2];

}

}

Данный метод обобщен для загрузки самой модели или текстуры объекта, как и загрузка к местности. пример, образованный список, в котором записывается положение некоторого фрагмента на плоскости: vertexListPtr[i].point[0] = frame->scale[0] \* frame->fp[i].v[0] + frame->translate[0];

Если при поиске обнаружена следующая точка, то из списка извлекается массив фрагментов.

На рисунке ниже представлен результат формируемого мира при подгрузке основных фрагментов:

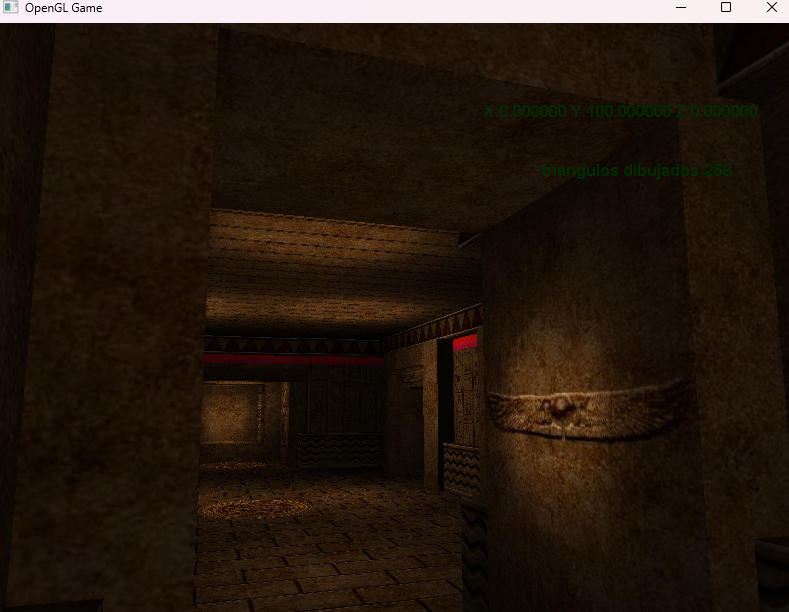


Рисунок 7 – Результат рендеринга мира

Согласно перемещению объекта местности, используются два варианта:

* Угловой шаг;
* «Бриллиантовый» шаг.

Смысл первого шага основывается на следующем: необходимо найти значения на левой и верхней сторонах каждого прямоугольника, к прямоугольникам относится комната. Правая и нижняя стороны — это левая и верхняя стороны соседних прямоугольников, таким образом вычислять их не нужно. Поле высоты переносится, для того чтобы объект передвижения (игрок) не завис. Правая сторона последнего прямоугольник в строке — это левая сторона первого прямоугольника в строке. Конечная сторона последнего прямоугольника в столбце является верхней стороной первого прямоугольника в столбце.

Реализация в коде данных переменных следующая:

g = (d+f+a+b)/4 + random

h = (a+c+e+f)/4 + random

In the code below:

a = (i,j)

b = (ni,j)

c = (i,nj)

d = (mi,pmj)

e = (pmi,mj)

f = (mi,mj)

g = (mi,j)

h = (i,mj)

На рисунке далее показано каким образом выбираются в переменные:

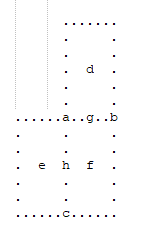


Рисунок 8 – Процесс определения куба

Следующий алгоритм работает иначе: необходимо найти значения в центре прямоугольников путем усреднения значений к углам и добавление случайного смещения, рисунок .



Рисунок 9 – Поиск центральной точки

Данный алгоритм действует по основе алгоритма Кэтмелла –Кларка. Алгоритм использует расчет центральной точки для сращения объекта в целое. После обнаружения средней центральной точки, для соединения и проведения линии проводится расчет средних точек на ребрах. Преобразование таким образом, помогает улучшить визуальную картинку.

3.4 Установка управления

Последним этапом для передвижения мира является само управление. Поскольку, трехмерное пространство подразумевает перемещение не только по одной плоскости, но так же и по плоскости горизонтальной – управление мыши.

Управление и подключение самой мыши данные возьмутся из стандартного набора mouse.h и mouse.cpp – который устанавливает к переменным сигнал о нажатой кнопке.

Реализацию и выгрузку в переменные будут осуществлены в engine.cpp.

Сам хедер engine, так же является стандартным, но управляемый файл необходимо сформировать по текущим потребностям.

Дополнительно стоит отметить, что к данному файлу будут вытащены данные по управлению и на клавиатуре.

Листинг 6 – Входящие данные

inputSystem->GetMouseMovement(mouseDeltaX, mouseDeltaY);

OnMouseMove(mouseDeltaX, mouseDeltaY);

if (inputSystem->KeyDown(SDLK\_w))

OnKeyDown(SDLK\_UP);

if (inputSystem->KeyDown(SDLK\_s))

OnKeyDown(SDLK\_DOWN);

if (inputSystem->KeyDown(SDLK\_a))

OnKeyDown(SDLK\_LEFT);

if (inputSystem->KeyDown(SDLK\_d))

OnKeyDown(SDLK\_RIGHT);

if (inputSystem->KeyDown(SDLK\_f))

OnKeyDown(SDLK\_f);

if (inputSystem->KeyDown(SDLK\_g))

OnKeyDown(SDLK\_g);

if (inputSystem->KeyDown(SDLK\_PLUS))

OnKeyDown(SDLK\_PLUS);

if (inputSystem->KeyDown(SDLK\_MINUS))

OnKeyDown(SDLK\_MINUS);

if (inputSystem->KeyDown(SDLK\_ESCAPE))

OnKeyDown(SDLK\_ESCAPE);

if (inputSystem->ButtonDown(SDL\_BUTTON\_LEFT))

{

if (buttonDelta == 0.0f)

{

OnMouseDownL(0,0);

buttonDelta = 0.5f;

}

}

}

К входящим данным относится получение отклика не только кнопок мыши, но и клавиатуры для управления.

После получения отклика inputSystem->KeyDown(SDLK\_g))

OnKeyDown(SDLK\_g);

Система обращает положение объекта к движению.

То есть, при получении данных от камеры и положения объекта, в наследуемом методе объект перемещается и определяет действия согласно следующему сценарию:

void CEngine::GameCycle(float deltaTime)

{

CCamera \*camera = OnGetCamera(); // get the camera

CWorld \*world = OnGetWorld(); // get the world

if (useInput)

CheckInput(deltaTime);

// подгрузить фреймы opengl (clear, identity)

OnPrepare();

// подготовить объекты и выполнить столкновения

world->Prepare();

// перемещение и ориентирование камеры в пространстве

camera->Animate(deltaTime);

// так же объекты

world->Animate(deltaTime);

// прорисовка объектов

world->Draw(camera);

Где прорисовка мира затрагивает положение точек на плоскостях.

На следующем рисунке показан результат выполненного перемещения объекта от изначальной точки:

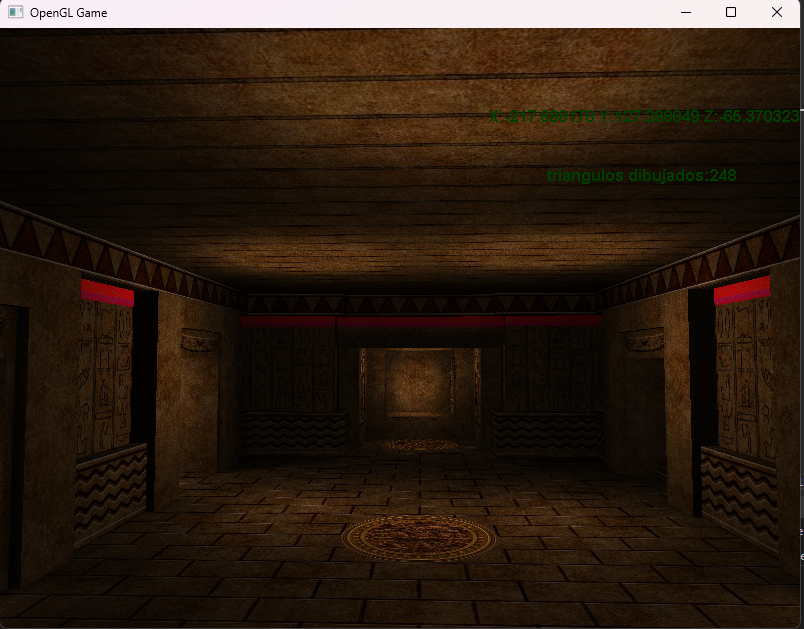


Рисунок 10 – Результат перемещения от начальной точки

Результаты тестирования движка представлены в Приложении Д.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта была достигнута следующая цель: средствами реализации quake-подобного игрового движка, были изучены такие понятия как методы бинарного разбиения, проброска лучей и основа формирования местности в трехмерном пространстве.

Так же были выполнены задачи по реализации: построена камера управления игроком, рассмотрен и реализован процесс коллизии объектов. Возможностями библиотеки OpenGl была использован метод наложения двумерной текстуры на объекты.

При выполнении курсового проекта также была рассмотрена история и разница формирования метода, по которому был спроектирован игровой движок – данные результаты позволили расширить возможность в реализации проекта.

Список использованных источников

1. Баранов, С.Н. Основы компьютерной графики : учеб. пособие / С.Н. Баранов, С.Г. Толкач. - Красноярск : Сиб. федер. ун-т, 2018. - 88 с. - ISBN 978-5-7638-3968-5. - Текст : электронный. - URL: https://znanium.com/catalog/product/1032167 (дата обращения: 01.06.2022).
2. Вольф, Д. OpenGL 4. Язык шейдеров. Книга рецептов / Д. Вольф ; пер. с англ. А.Н. Киселева. - Москва : ДМК Пресс, 2015. - 368 с. - ISBN 978-5-97060-255-3. - Текст : электронный. - URL: https://znanium.com/catalog/product/1027790 (дата обращения: 03.04.2022).
3. Корнеев, В. И. Программирование графики на С++. Теория и примеры : учебное пособие / В.И. Корнеев, Л.Г. Гагарина, М.В. Корнеева. — Москва : ФОРУМ : ИНФРА-М, 2021. — 517 с. + Доп. материалы [Электронный ресурс]. — (Высшее образование: Бакалавриат). — www.dx.doi.org/10.12737/23113. - ISBN 978-5-8199-0837-2. - Текст : электронный. - URL: https://znanium.com/catalog/product/1222414 (дата обращения: 03.04.2022).
4. Никулин, Е. А. Компьютерная геометрия и алгоритмы машинной графики: Пособие / Никулин Е.А. - СПб:БХВ-Петербург, 2015. - 554 с. ISBN 978-5-9775-1925-0. - Текст : электронный. - URL: https://znanium.com/catalog/product/940228 (дата обращения: 01.06.2022)
5. Сагитов, Р. В. Линейная алгебра. Часть II: Линейное программирование, динамическое программирование и теория игр : учебно-методическое пособие / Р. В. Сагитов, В. Г. Шершнев. - Москва : Менеджер, 2007. - 192 с. - Текст : электронный. - URL: https://znanium.com/catalog/product/347844 (дата обращения: 01.06.2022). – Режим доступа: по подписке.

ПРИЛОЖЕНИЕ А  
Код части программы – Camera

#include "camera.h"

#include "object.h"

CCamera::CCamera()

{

position = CVector(0.0, 0.0, 0.0);

lookAt = CVector(0.0, 0.0, 1.0);

forward = lookAt;

up = CVector(0.0, 1.0, 0.0);

right = CVector(1.0, 0.0, 0.0);

velocity = CVector(0.0, 0.0, 0.0);

acceleration = CVector(0.0, 0.0, 0.0);

yaw = 0.0;

pitch = 0.0;

}

CCamera::CCamera(CVector \*look)

{

position = CVector(0.0, 0.0, 0.0);

lookAt = look->UnitVector();

forward = lookAt;

up = CVector(0.0, 1.0, 0.0);

right = forward.CrossProduct(up); //CVector(1.0, 0.0, 0.0);

velocity = CVector(0.0, 0.0, 0.0);

acceleration = CVector(0.0, 0.0, 0.0);

yaw = 0.0;

pitch = 0.0;

}

CCamera::CCamera(CVector \*pos, CVector \*look)

{

position = \*pos;

lookAt = look->UnitVector();

forward = lookAt;

up = CVector(0.0, 1.0, 0.0);

right = CVector(1.0, 0.0, 0.0);

velocity = CVector(0.0, 0.0, 0.0);

acceleration = CVector(0.0, 0.0, 0.0);

yaw = 0.0;

pitch = 0.0;

}

CCamera::~CCamera()

{

}

void CCamera::UpdateLookAt()

{

CVector look = CVector(finalLookAt.x - lookAt.x,

finalLookAt.y - lookAt.y,

finalLookAt.z - lookAt.z);

lookAtVel = look \* 0.5;

}

void CCamera::UpdateMoveTo()

{

CVector pos = CVector(finalPosition.x - position.x,

finalPosition.y - position.y,

finalPosition.z - position.z);

velocity = pos \* 0.5;

}

void CCamera::RotatePitch(scalar\_t radians)

{

float sine = sinf(radians);

float cosine = cosf(radians);

up.y = cosine \* up.Length();

up.z = sine \* up.Length();

forward.y = -sine \* forward.Length();

forward.z = cosine \* forward.Length();

/\* x y z p

| 1 0 0 0 |

M = | 0 cos(A) -sin(A) 0 |

| 0 sin(A) cos(A) 0 |

| 0 0 0 1 |

\*/

}

void CCamera::RotateYaw(scalar\_t radians)

{

float sine = sinf(radians);

float cosine = cosf(radians);

right.x = cosine \* right.Length();

right.z = sine \* right.Length();

forward.x = -sine \* forward.Length();

forward.z = cosine \* forward.Length();

/\* x y z p

| cos(A) 0 -sin(A) 0 |

M = | 0 1 0 0 |

| sin(A) 0 cos(A) 0 |

| 0 0 0 1 |

\*/

}

void CCamera::RotateRoll(scalar\_t radians)

{

float sine = sinf(radians);

float cosine = cosf(radians);

right.x = cosine \* right.Length();

right.y = sine \* right.Length();

up.x = -sine \* forward.Length();

up.y = cosine \* forward.Length();

/\*

| cos(A) -sin(A) 0 0 |

M = | sin(A) cos(A) 0 0 |

| 0 0 1 0 |

| 0 0 0 1 |

\*/

}

void CCamera::LookAtNow(CObject \*object)

{

lookAt = object->position;

}

void CCamera::MoveToNow(scalar\_t x, scalar\_t y, scalar\_t z)

{

position.x = x;

position.y = y;

position.z = z;

}

void CCamera::MoveToNow(CObject \*object)

{

position = object->position;

}

// просмотр объекта во времени

void CCamera::LookAt(CObject \*object)

{

CVector v = CVector(object->position - lookAt);

initLookAt = lookAt;

finalLookAt = object->position;

lookAtAccel = -lookAt \* 0.25f;

UpdateLookAt();

}

// двигаться к объекту с течением времени

void CCamera::MoveTo(CObject \*object)

{

CVector v = CVector(object->position - position);

initPosition = position;

finalPosition = object->position;

acceleration = -position \* 0.25f;

UpdateMoveTo();

}

void CCamera::Animate(scalar\_t deltaTime)

{

posicion\_antigua=position;

if ((yaw >= 360.0f) || (yaw <= -360.0f))

yaw = 0.0f;

if (pitch > 60.0f)

pitch = 60.0f;

if (pitch < -60.0f)

pitch = -60.0f;

float cosYaw = (scalar\_t)cos(DEG2RAD(yaw));

float sinYaw = (scalar\_t)sin(DEG2RAD(yaw));

float sinPitch = (scalar\_t)sin(DEG2RAD(pitch));

float speed = velocity.z \* deltaTime;

float strafeSpeed = velocity.x \* deltaTime;

if (speed > 15.0)

speed = 15.0;

if (strafeSpeed > 15.0)

strafeSpeed = 15.0;

if (speed < -15.0)

speed = -15.0;

if (strafeSpeed < -15.0)

strafeSpeed = -15.0;

if (velocity.Length() > 0.0)

acceleration = -velocity \* 1.5f;

velocity += acceleration\*deltaTime;

position.x += float(cos(DEG2RAD(yaw + 90.0)))\*strafeSpeed;

position.z += float(sin(DEG2RAD(yaw + 90.0)))\*strafeSpeed;

position.x += float(cosYaw)\*speed;

position.z += float(sinYaw)\*speed;

//поднять и опустить камеру (работа в верхней и нижней плоскости)

position.y += float(sinPitch)\*speed;

lookAt.x = float(position.x + cosYaw);

lookAt.y = float(position.y + sinPitch);

lookAt.z = float(position.z + sinYaw);

gluLookAt(position.x, position.y, position.z,

lookAt.x, lookAt.y, lookAt.z,

0.0, 1.0, 0.0);

}

ПРИЛОЖЕНИЕ Б  
Код коллизии объектов

#include "entity.h"

CEntity::CEntity()

{

interpol = 3.0f;

stateStart = 0;

stateEnd = 39;

animSpeed = 7.0f;

direction = 0.0f;

entitySound = NULL;

position = CVector(0,0,0);

velocity = CVector(0,0,0);

acceleration = CVector(0,0,0);

size = 6.25f;

}

CEntity::~CEntity()

{

if (entitySound)

{

delete entitySound;

entitySound = NULL;

}

}

void CEntity::OnAnimate(float deltaTime)

{

float cosYaw = (float)cos(DEG2RAD(direction));

float sinYaw = (float)sin(DEG2RAD(direction));

float speed = velocity.z \* deltaTime;

if ((direction >= 360.0f) || (direction <= -360.0f))

direction = 0.0f;

position.x += float(cosYaw)\*speed;

position.z += float(sinYaw)\*speed;

deltaT = deltaTime;

}

void CEntity::OnCollision(CObject \*collisionObject)

{

if (typeid(\*collisionObject) == typeid(CEntity))

{

modelState = MODEL\_IDLE;

velocity = CVector(0.0, 0.0, 0.0);

}

if (typeid(\*collisionObject) == typeid(CTerrain))

{

position.y = ((CTerrain\*)collisionObject)->GetHeight(position.x, position.z) + size;

}

}

void CEntity::OnDraw(CCamera \*camera)

{

glTranslatef(position.x, position.y, position.z);

glRotatef(-direction, 0.0, 1.0, 0.0);

glRotatef(90.0f, -1.0f, 0.0f, 0.0f);

glScalef(0.25, 0.25, 0.25);

AnimateModel(stateStart, stateEnd, deltaT\*animSpeed);

}

void CEntity::OnPrepare()

{

/\*

Frame# Action

----------------

0-39 idle

40-46 running

47-60 getting shot but not falling (back bending)

61-66 getting shot in shoulder

67-73 jumping

74-95 idle

96-112 getting shot and falling down

113-122 idle

123-135 idle

136-154 crouch

155-161 crouch crawl

162-169 crouch adjust weapon (idle)

170-177 kneeling dying

178-185 falling back dying

186-190 falling forward dying

191-198 falling back slow dying

\*/

srand((unsigned)time(NULL));

switch (modelState)

{

case MODEL\_IDLE:

stateStart = 0;

stateEnd = 39;

break;

case MODEL\_CROUCH:

break;

case MODEL\_RUN:

stateStart = 40;

stateEnd = 46;

velocity = CVector(0.0, 0.0, 15.0);

break;

case MODEL\_JUMP:

stateStart = 67;

stateEnd = 73;

break;

case MODEL\_DIE:

stateStart = 178;

stateEnd = 184;

break;

case MODEL\_FIRE:

stateStart = 0;

stateEnd = 5;

break;

}

// выполнить обнаружение столкновений объекта со всеми другими объектами в мире

ProcessCollisions(FindRoot());

}

void CEntity::LoadAudio(CAudioSystem \*audioSystem, char \*filename, bool is3DSound)

{

if (entitySound != NULL)

{

delete entitySound;

entitySound = new CAudio;

}

if (audioSystem != NULL)

entitySound = audioSystem->Create(filename, is3DSound);

else

{

delete entitySound;

entitySound = NULL;

}

}

ПРИЛОЖЕНИЕ В  
Рендер мира

#include "world.h"

//конструкция

CWorld::CWorld()

{

terrain = new CTerrain(256, 0.5);

}

//деструктор (деинициализация класса)

CWorld::~CWorld()

{

delete terrain;

delete audioSystem;

delete worldSound;

}

//конструкция

CWorld::CWorld(CCamera \*c)

{

camera = c;

terrain = new CTerrain(256, 0.6f);

enemy = new CEntity;

enemy2 = new CEntity;

enemy3 = new CEntity;

//nivel= new CNivel("models\\final3.bsp");

nivel= new CNivel("models\\final3.bsp");

msg=new CFont("Arial", 16);

frames\_por\_segundo=new CFont("Arial", 16);

temporizador=new CHiResTimer;

audioSystem = new CAudioSystem;

temporizador->Init();

enemy->AttachTo(terrain);

enemy2->AttachTo(terrain);

worldSound = audioSystem->Create("sounds\\musica.ogg", false);

audioSystem->Play(worldSound, true);

enemy->Load("models\\sodf8\\tris.md2", "models\\sodf8\\abarlith.pcx");

enemy->SetState(MODEL\_RUN);

enemy->direction = 45.0;

enemy2->Load("models\\ogro\\tris.md2", "models\\ogro\\ogrobase.pcx");

enemy2->SetState(MODEL\_RUN);

enemy2->direction = 225.0;

enemy2->position = CVector(200.0, 0.0, 200.0);

enemy3->Load("models\\ogro\\tris.md2", "models\\ogro\\ogrobase.pcx");

enemy3->position = CVector(0.0, 0.0, 0.0);

camera->position.y=100;

camera->position.x=0;

camera->position.z=00;

}

//анимация объектов

void CWorld::Animate(float deltaTime)

{

CVector nueva\_posicion;

// устанавливаем высоту камеры в зависимости от положения на местности

// camera->position.y = Terrain->GetHeight(camera->position.x, camera->position.z) + 10.0f;

nueva\_posicion=nivel->TrazaEsfera(camera->posicion\_antigua,camera->position,15.0f);

//if(nivel->colision)

//camera->position=camera->posicion\_antigua;

camera->position=nueva\_posicion;

terrain->Animate(deltaTime);

}

//прорисовка сцены

void CWorld::Draw(CCamera \*camera)

{

msg->SetPos3D(0.25,0.25,0);

frames\_por\_segundo->SetPos3D(0.15,0.35,0);

msg->SetRGB(0,1,0);

frames\_por\_segundo->SetRGB(0,1,0);

msg->Print("tнарисованные треугольники:%d",nivel->triangulos);

frames\_por\_segundo->Print("X:%f Y:%f Z:%f",camera->position.x,camera->position.y,camera->position.z);

//terrain->Draw(camera);

enemy3->Draw(camera);

//glTranslatef(100,158,100);

/\* glPushMatrix();

glLoadIdentity();

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_COLOR, GL\_ONE\_MINUS\_SRC\_COLOR);

glDisable(GL\_TEXTURE\_2D);

glColor4f(1.0, 0.0, 0.0, 1.0);

glBegin(GL\_QUADS);

glVertex3f(-5, -5, -3.0);

glVertex3f(5, -5, -3.0);

glVertex3f(5, 5, -3.0);

glVertex3f(-5, 5, -3.0);

glEnd();

glColor4f(1.0, 1.0, 1.0, 1.0);

glEnable(GL\_TEXTURE\_2D);

glDisable(GL\_BLEND);

glPopMatrix();\*/

/\* glEnable(GL\_BLEND);

glBlendFunc(GL\_ONE\_MINUS\_SRC\_COLOR, GL\_ONE\_MINUS\_DST\_COLOR);\*/

glPushMatrix();

glBegin(GL\_LINE\_LOOP);

glColor3ub(255,0,0);

glVertex3f(0,0,0);

glColor3ub(0,255,0);

glVertex3f(0,10,0);

glColor3ub(255,255,255);

glVertex3f(0,0,10);

glEnd();

glPopMatrix();

nivel->Draw(camera);

//glDisable(GL\_BLEND);

}

void CWorld::OnPrepare()

{

glClearColor(1.0, 1.0, 1.0, 1.0);

//nivel->ComprobarNodo(0,1

//camera->position.y=100;

//camera->position.x=1050;

//camera->position.z=1050;

//terrain->Prepare();

}

ПРИЛОЖЕНИЕ Г  
Управление

#include <stdlib.h>

#include "engine.h"

#include "HiResTimer.h"

#include "camera.h"

#include "world.h"

void CEngine::CheckInput(float deltaTime)

{

static float buttonDelta = 0.0f;

int mouseDeltaX, mouseDeltaY; // положение мышки

// уменьшить количество времени до следующего возможного распознанного нажатия кнопки

buttonDelta -= deltaTime;

if (buttonDelta < 0.0f)

buttonDelta = 0.0f;

// обновит ьполучить последние движения мыши

inputSystem->Update();

//

inputSystem->GetMouseMovement(mouseDeltaX, mouseDeltaY);

OnMouseMove(mouseDeltaX, mouseDeltaY);

if (inputSystem->KeyDown(SDLK\_w))

OnKeyDown(SDLK\_UP);

if (inputSystem->KeyDown(SDLK\_s))

OnKeyDown(SDLK\_DOWN);

if (inputSystem->KeyDown(SDLK\_a))

OnKeyDown(SDLK\_LEFT);

if (inputSystem->KeyDown(SDLK\_d))

OnKeyDown(SDLK\_RIGHT);

if (inputSystem->KeyDown(SDLK\_f))

OnKeyDown(SDLK\_f);

if (inputSystem->KeyDown(SDLK\_g))

OnKeyDown(SDLK\_g);

if (inputSystem->KeyDown(SDLK\_PLUS))

OnKeyDown(SDLK\_PLUS);

if (inputSystem->KeyDown(SDLK\_MINUS))

OnKeyDown(SDLK\_MINUS);

if (inputSystem->KeyDown(SDLK\_ESCAPE))

OnKeyDown(SDLK\_ESCAPE);

if (inputSystem->ButtonDown(SDL\_BUTTON\_LEFT))

{

if (buttonDelta == 0.0f)

{

OnMouseDownL(0,0);

buttonDelta = 0.5f;

}

}

}

void CEngine::GameCycle(float deltaTime)

{

CCamera \*camera = OnGetCamera(); // get the camera

CWorld \*world = OnGetWorld(); // get the world

if (useInput)

CheckInput(deltaTime);

// подгрузить фреймы opengl (clear, identity)

OnPrepare();

// подготовить объекты и выполнить столкновения

world->Prepare();

// перемещение и ориентирвоание камеры в пространстве

camera->Animate(deltaTime);

// так же объекты

world->Animate(deltaTime);

// прорисовка объектов

world->Draw(camera);

//

SwapBuffers();

}

int CEngine::EnterMessageLoop()

{

// Message loop

SDL\_Event event;

timer = new CHiResTimer;

timer->Init();

for (;;)

{

GameCycle(timer->GetElapsedSeconds(1));

//timer->LockFPS(50);

while (SDL\_PollEvent(&event))

{

WndProcOGL(event);

}

}

delete timer;

return 0;

}

ПРИЛОЖЕНИЕ Д  
Результаты выполненного проекта



Рисунок Д1



Рисунок Д2

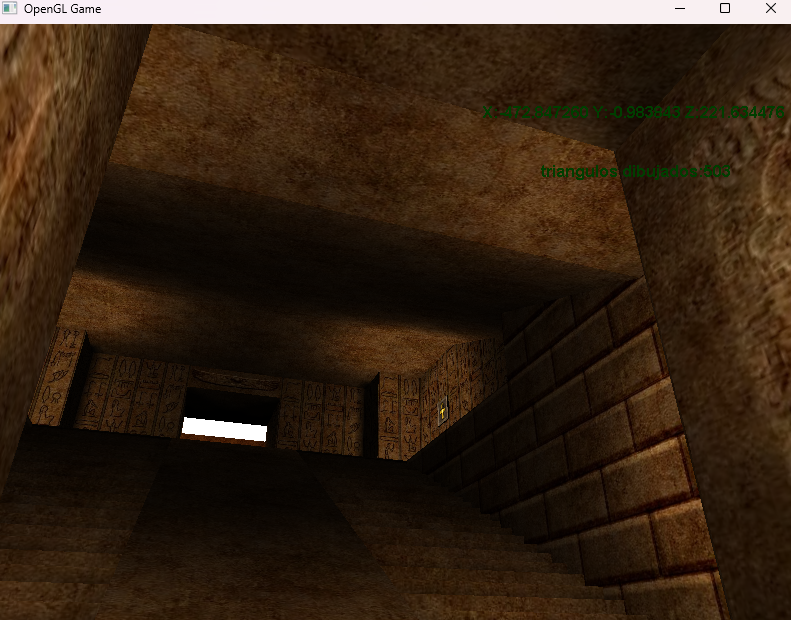


Рисунок Д3