Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Космических и Информационных Технологий
институт

Информатики
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой

_____ _____
подпись инициалы, фамилия
« ____» _____ 20 __ г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

09.03.04 Программная инженерия
код и наименование специальности

Разработка игрового 3D приложения
тема

Пояснительная записка

Руководитель	_____	доцент, к. т. н.	Кукарцев В.В.
	подпись, дата	должность, ученая степень	фамилия, инициалы
Выпускник	_____		Щербинин Е.О.
	подпись, дата		фамилия, инициалы
Нормоконтролер	_____		Антамошкин О.А.
	подпись, дата		фамилия, инициалы

Красноярск 2016

Аннотация

Тема дипломной работы – «Разработка игрового 3D приложения» - в рамках которой были рассмотрены и изучены основы разработки игровых приложений и работы с пакетом библиотек DirectX.

Было реализовано построение игрового ландшафта, возможность перемещения по игровому миру, загрузка 3D моделей в игровой мир, проверка столкновений некоторых объектов в игре и возможность стрелять по передвигающимся персонажам в игровом мире.

Annotation

Thesis - "Development of 3D gaming application" - in which were reviewed and learned the basics of developing gaming applications, and working with the package DirectX libraries.

There have been implemented the gaming landscape generation, the ability to walk in the game world, load 3D models to the gaming world, collision detection of some objects in the game and the ability to shoot at the moving characters in this game.

СОДЕРЖАНИЕ

АННОТАЦИЯ 2
СОДЕРЖАНИЕ 4
ВВЕДЕНИЕ 5
1. Общие сведения 6
1.1. Актуальность игровых приложений 6
1.2. Анализ современного состояния рынка игровых 3D приложений 6
1.3. Программные средства для реализации игрового приложения 8
2. Описание приложения 10
2.1. Функционал приложения 10
2.2. Требования к аппаратным и программным средствам 12
3. Техническая реализация ..... **Ошибка! Закладка не определена.**
3.1. Описание алгоритма (блок-схема) и диаграмма классов**Ошибка! Закладка не**
3.2. Создание окна приложения 17
3.3. Модуль D3DDevice 20
3.4. Модуль Camera ..... **Ошибка! Закладка не определена.**
3.5. Модуль Terrain 31
3.6. Проверка коллизий (столкновений) 42
ЗАКЛЮЧЕНИЕ ..... **Ошибка! Закладка не определена.**
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**Ошибка! Закладка не определена.**

ВВЕДЕНИЕ

В современном мире практически каждый человек, независимо от возраста, время от времени пользуется компьютером и современными средствами коммуникации и обработки информации. Огромное влияние на развитие вычислительной техники оказали видеоигры. Игровая индустрия еще очень молода, но за пару-тройку десятков лет своего существования стала частью нашей жизни и имеет огромное влияние на людей. В компьютерные игры играют почти все. Прежде всего, игровая индустрия это такая же сфера искусства как литература, кино и живопись, но в сравнении с ними она более обширна, даже можно сказать, что игровая культура вобрала в себя все остальные виды творчества. Ведь для создания хорошей игры требуется усердная и качественная работа художников, сценаристов, музыкантов. Все эти люди искусства задействованы в создании игр. Без красочных детальных концептов, грамотного сюжета и поддерживающего атмосферу саундтрека мы не прочувствуем мир за монитором, не вживемся в роль подконтрольного персонажа. Многие исследования доказывают положительное влияние игр на концентрацию внимания, реакцию, логическое мышление, на способность общаться с другими людьми. Влияние зависит от жанра игры.

1. Общие сведения

1.1 Актуальность игровых приложений

В современном мире практически каждый человек, независимо от возраста, время от времени пользуется компьютером и современными средствами коммуникации и обработки информации. Огромное влияние на развитие вычислительной техники оказали видеоигры. Многие из нас проводят досуг за прохождением квеста или получением уровня в ролевой игре. Так же разработка игр – очень выгодная сфера деятельности, на фоне других областей программного обеспечения. Главная особенность игр в том, что их они никогда не потеряют свою популярность. Каждый день выходят новые игры, новые жанры и, чтобы удовлетворять запросам современных игр, в свою очередь, выходят новые технологические средства и технологии. Таким образом игры приносят прибыль не только своим непосредственным разработчикам, но и косвенно помогают развитию отрасли аппаратных средств. Многие люди утверждают, что компьютерные игры могут положительно влиять на концентрацию внимания, реакцию, логическое мышление, на способность общаться с другими людьми.

1.2 Анализ современного состояния рынка игровых 3D приложений

На сегодняшний день рынок игр во всем мире является самым большим сегментом мирового рынка цифрового контента, ежегодно генерируя многомиллиардные доходы и привлекая огромную аудиторию.

Рисунок 1 - Рынок игр в 2014 - 2016 гг., млрд. долл.

Как свидетельствует статистика, на 98 процентов персональных компьютеров мира установлена хотя бы одна игра. В США они уже превратились фактически в неотъемлемую часть повседневной жизни, известно, что средний американец тратит на игры 6-7 часов в неделю. Что же до россиян, то около миллиона из них проводят за компьютерными играми от 3 до 7 часов в сутки(!).

В последние годы появились новые возможности для динамичного роста мирового игрового рынка:

- доступность компьютеров, широкополосный доступ в интернет
- появились разные платформы для игр – игровая приставка, социальные сети, в которые встроены игры, планшет, мобильный телефон, смартфон и т.д.;
- распространение мобильных устройств (планшетных компьютеров, смартфонов).
- компьютеры, мобильные устройства и игровые приставки становятся дешевле, а значит, доступнее для самых широких масс.

В общем, индустрия компьютерных игр с каждым новым годом будет только расти, принося их разработчикам и издателям всё больше прибыли.

Вот несколько примеров популярных игровых 3D приложений GTA V, Far Cry 3, GTA San Andreas , Xenus 2, Crysis, Stalker, Call of Duty, Postal 2 и др.

Вот некоторые крупнейшие компании по разработке видеоигр: Rockstar Games, Electronic Art, Ubisoft.

1.3 Программные средства для реализации игрового приложения

При разработке игрового приложения были использованы следующие программные средства:

- Microsoft Visual Studio Express 2012 - среда разработки приложения
- Blender - редактор трехмерных моделей
- GIMP - редактор двухмерных изображений
- DirectX - это набор API, разработанных для решения задач, связанных с программированием под Microsoft Windows. Преимущественно для работы с трехмерной графикой

Практически все части DirectX API представляют собой наборы COM-совместимых объектов.

В целом, DirectX подразделяется на:

- DirectX Graphics, набор интерфейсов, ранее (до версии 8.0) делившихся на: DirectDraw: интерфейс вывода растровой графики. (Его разработка давно прекращена) и Direct3D (D3D): интерфейс вывода трёхмерных примитивов.
- DirectInput: интерфейс, используемый для обработки данных, поступающих с клавиатуры, мыши, джойстика и пр. игровых контроллеров.
- DirectPlay: интерфейс сетевой коммуникации игр.
- DirectSound: интерфейс низкоуровневой работы со звуком (формата Wave)
- DirectMusic: интерфейс воспроизведения музыки в форматах Microsoft.
- DirectShow: интерфейс, используемый для ввода/вывода аудио и/или видео данных.
- DirectX Instruments — технология, позволяющая на основе мультимедийного API DirectX создавать и использовать программные синтезаторы. В отличие от DX-плагинов, такие программы могут полностью управляться по MIDI и служат главным образом не для обработки, а для синтеза звука. Технология DXi была популярна в 2001—2004 гг., особенно в программных продуктах Cakewalk, но со временем проиграла «войну форматов» технологии VST от Steinberg.
- DirectSetup: часть, ответственная за установку DirectX.
- DirectX Media Objects: реализует функциональную поддержку потоковых объектов (например, кодировщики/декодировщики)

- Direct2D: интерфейс вывода двухмерной графики

Приложение было написано на языке программирования C++.

2. Описание приложения

2.1 Функционал приложения

Программа должна выполнять следующие основные действия:

Инициализация графики. В DirectX этот процесс достаточно объемный и трудоемкий. Будет удобнее вынести его в отдельную функцию.

Загрузка сетки из X-файла. В трехмерном пространстве существуют всего три вида примитивов - это точка, отрезок и треугольник. Все остальные примитивы можно построить с помощью этих трех составляющих. Основным примитивом в Direct3D, конечно, является треугольник. Почему в качестве основы используется именно треугольник, и нет более сложных фигур? Дело в том, что сами видео-ускорители при формировании графики оперируют именно треугольниками, поэтому логичнее будет, если программа будет «разговаривать» с устройством на одном языке и оперировать одними и теми же данными.

Для построения одного треугольника нужно три точки, каждая из которых должна иметь, как минимум 3 параметра - это координаты в трехмерном пространстве (x,y,z) и цвет вершины. Для создания объемной фигуры понадобится создание буфера индексов и буфера вершин. Каждую точку, определяющую вершину треугольника нужно четко рассчитать. Из всего этого следует вывод, чтобы построить фигуру сложнее куба, нужно потратить не один час времени. И как, например, нарисовать фигуру человека таким образом, да еще и правильно натянуть на рассчитанную сетку текстуру. Для решение проблемы корпорация Microsoft разработала формат X. В файле с расширением «X» может храниться информация о сетке, текстуре, скелете и даже анимации. DirectX имеет ряд очень удобных функций для работы с X-файлами и это очень упрощает программирование графики. Чтобы создать сетку и сохранить ее в файле с расширением «X», нужно, чтобы этот трехмерный редактор поддерживал экспорт в этот формат.

В данном случае используется трехмерный редактор Blender, включающий в себя специальный плагин для экспорта моделей в формат «X».

Итак, этот проект должен загружать сетки и, если есть, текстуры из X-файла.

Расчеты положения игровых объектов. Игровым объектом является: источник освещения, камера, сетка (загруженная из X-файла или описанная в ручную). Расчет положения объекта производится с помощью векторов (в случае с источником света) и матриц (с помощью матриц рассчитывается положение камеры и всего игрового мира). Что касается сетки, то тут все достаточно просто - есть матрица положения, с помощью которой можно деформировать и перемещать объект. С камерой все несколько сложнее. Камера в игровой программе - это персонаж, который должен перемещаться по игровому пространству по команде игрока (пользователя). Приложение должно рассчитывать положение и направление камеры в игровом пространстве оперируя матрицами.

Диалог с пользователем. Все выше перечисленные действия можно реализовать с помощью одного только модуля Direct3D. Для диалога с пользователем понадобится модуль DirectInput, который будет производить чтение с устройств ввода. Чтобы узнать какое действие произвел пользователь можно было бы воспользоваться WinAPI функциями. Но WinAPI работает через Windows, и это значительно снижает производительность. Компьютерные игры - это та область программирования, где нет лишних ресурсов и программы должны работать максимально быстро. К тому же есть еще один нюанс в работе WinAPI, если зажать и не отпускать одну клавишу (например, при движении игрока прямо на некоторое расстояние) Windows выдаст сообщение о повторном нажатии одной и той же клавиши, что не допустимо в игре. DirectInput работает с устройством в обход Windows и достигает максимальной скорости считывания информации.

Программа должна инициализировать основные устройства ввода (клавиатура, мышь), считывать из них информацию, и, в соответствии с введенными пользователем данными, производить определенные действия (например, при нажатии клавиши «W» переместить камеру вперед, по направлению ее взгляда). Необходимо также реализовать модуль, позволяющий главному игроку перемещаться по игровому миру (управление с помощью клавиатуры и компьютерной мышки):

- вперед (клавиша «W»)
- назад (клавиша «S»)
- влево (клавиша «A»)
- вправо (клавиша «D»)
- взгляд направо (мышка вправо)
- взгляд налево (мышка влево)
- взгляд вверх (мышка вверх)
- взгляд вниз (мышка вниз)

В приложении также должен генерироваться игровой ландшафт по заданной карте высот в 32 разрядном файле формата BMP.

Необходимо выполнить освещение ландшафта для увеличения реализма сцены.

Нужно также добавить возможность перемещать камеру таким образом, чтобы она имитировала ходьбу по ландшафту.

Должна быть реализована проверка коллизий (столкновений) некоторых объектов, например игрока и стены.

2.2 Требования к аппаратным и программным средствам

На компьютере должна быть установлена операционная система Windows и DirectX 9 последней версии.

Для использования программы необходимы заголовочные файлы для DirectX 9 и среда разработки Microsoft Visual C++ 2012.

Проект занимает 12 Mb свободного места на диске. Плюс к этому 4 Mb понадобятся для размещения «заголовочных файлов».

3. Техническая реализация

3.1 Описание алгоритма (блок-схема) и диаграмма классов

При запуске приложения происходят следующие действия:

1. Построение окна – задание его свойств и создание с помощью функции WinAPI.

2. Инициализация DirectX – инициализация графики, проектирование окна, создание устройства для работы с Direct3D

3. Бесконечный цикл обработки сообщений – сообщения в цикле обрабатываются с помощью WinAPI функций. Если сообщений нет, то происходит вызов функций главного модуля приложения, в котором происходит формирование сцены и обработка данных, полученных с устройств ввода.

Выход из программы происходит либо после прихода сообщения, равного WM_QUIT, либо во время выполнения функций движка, которые отвечают за работу с клавиатурой, если была нажата клавиша Esc.

Краткая блок-схема проекта:

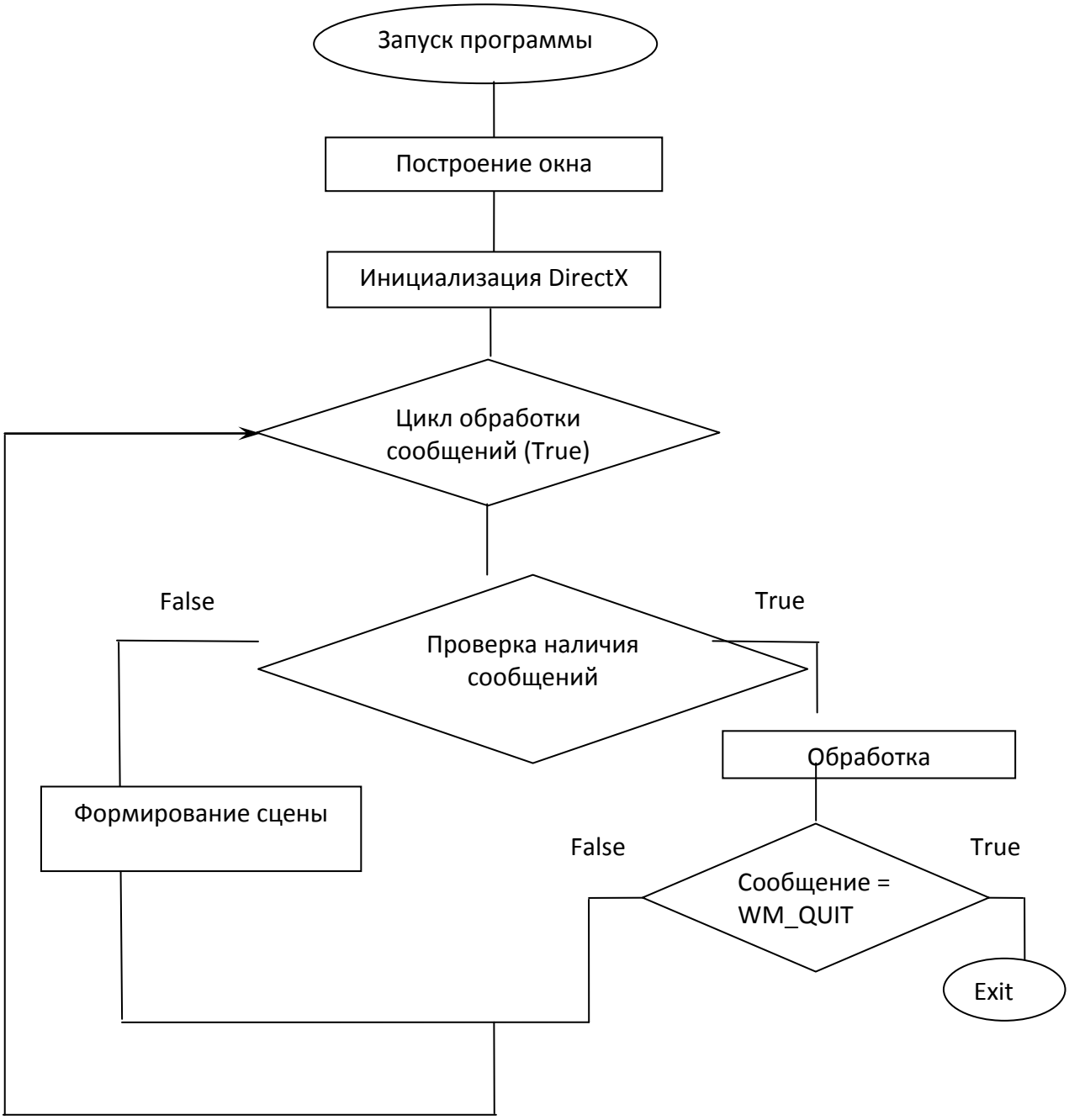

Диаграмма классов (иерархия основных модулей проекта):

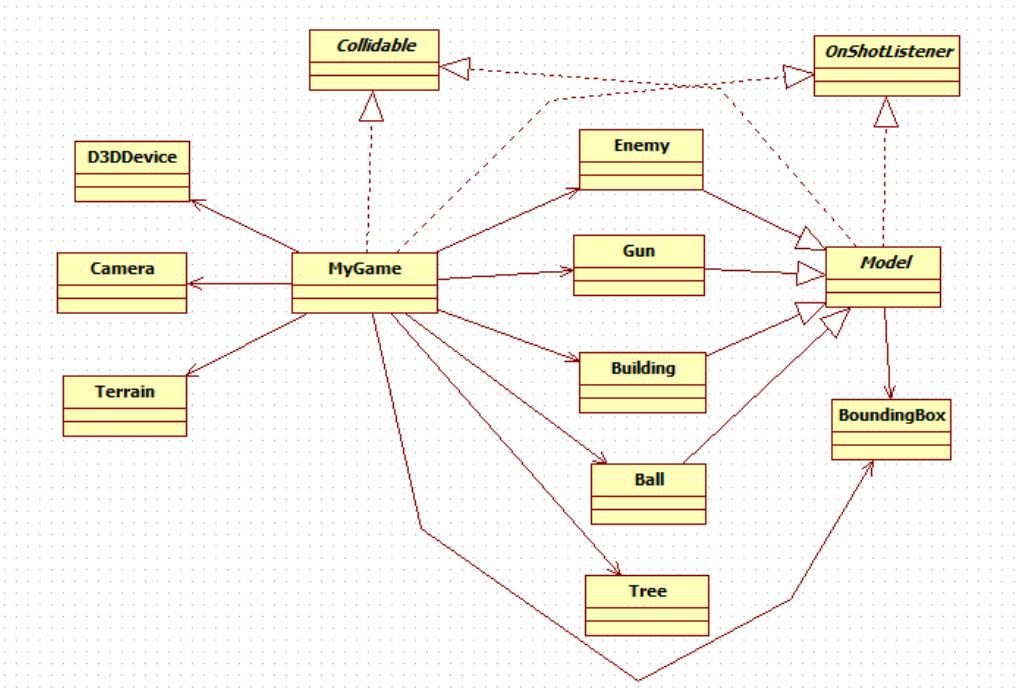

Рисунок 2 - Диаграмма классов

3.2 Создание окна приложения

Все программы под Windows используют специальный интерфейс программирования WinAPI. Это набор функций и структур на языке C, благодаря которым ваша программа становится совместимой с Windows.

Windows API обладает огромными возможностями по работе с операционной системой. Можно даже сказать - безграничными.

Мы не рассмотрим даже один процент всех возможностей WinAPI. Первоначально я хотел взять больше материала, но это заняло бы слишком много времени, и увязнув в болоте WinAPI, до DirectX'а мы добрались бы через пару лет. Описание WinAPI займёт два урока (включая этот). В них мы рассмотрим только каркас приложения под Windows.

Программа под Windows точно так же как и программа под DOS, имеет главную функцию. Здесь эта функция называется WinMain.

Функция WinMain

Программа под Windows состоит из следующих частей (всё это происходит внутри WinMain):

Создание и регистрация класса окна. Не путайте с классами C++. WinAPI написана на C, здесь нет классов в привычном для нас понимании этого слова.
Создание окна программы.
Основной цикл, в котором обрабатываются сообщения.
Обработка сообщений программы в оконной процедуре. Оконная процедура представляет собой обычную функцию.
Вот эти четыре пункта - основа программы Windows.

Теперь разберём всё это подробно:

WinAPI: Структура WNDCLASS

Прежде всего нужно создать и заполнить структурную переменную WNDCLASS, а затем на её основе зарегистрировать оконный класс.

Вот как выглядит эта структура:

```
typedef struct {
  UINT style;          // стиль окна
  WNDPROC lpfnWndProc; // указатель  на оконную процедуру
  int cbClsExtra;      // дополнительные байты после класса. Всегда ставьте 0
  int cbWndExtra;      // дополнительные байты после экземпляра окна. Всегда
ставьте 0
  HINSTANCE hInstance; // экземпляр приложения. Передаётся  в виде
параметра  в WinMain
```

```
  HICON hIcon;          // иконка приложения
  HCURSOR hCursor;      // курсор приложения
  HBRUSH hbrBackground;  // цвет фона
  LPCTSTR lpszMenuName;  // имя меню
  LPCTSTR lpszClassName; // имя класса
} WNDCLASS, *PWNDCLASS;
```

Структура WNDCLASS в составе WinAPI определяет базовые свойства создаваемого окна: иконки, вид курсора мыши, есть ли меню у окна, какому приложению будет принадлежать окно...

После того как вы заполните эту структуру, на её основе можно зарегистрировать оконный класс. Речь идёт не о таких классах как в C++. Скорее можно считать, что оконный класс это такой шаблон, вы его зарегистрировали в системе, и теперь на основе этого шаблона можно создать несколько окон. И все эти окна будут обладать свойствами, которые вы определили в структурной переменной WNDCLASS.

WinAPI: Функция CreateWindow

После регистрации оконного класса, на его основе создаётся главное окно приложения (мы сейчас приступили ко второму пункту). Делается это с помощью функции CreateWindow. Она имеет следующий прототип:

```
HWND CreateWindow(
  LPCTSTR lpClassName,  // имя класса
  LPCTSTR lpWindowName, // имя окна (отображается  в заголовке)
  DWORD dwStyle, // стиль окна
  int x,        // координата  по горизонтали  от левого края экрана
  int y,        // координата  по вертикали  от верхнего края экрана
  int nWidth,    // ширина окна
  int nHeight,   // высота окна
  HWND hWndParent, // родительское окно
  HMENU hMenu,     // описатель меню
  HINSTANCE hInstance, // экземпляр приложения
  LPVOID lpParam       // параметр; всегда ставьте NULL
);
```

Если в оконном классе (структуре WNDCLASS) задаются базовые свойства окна, то здесь - более специфические для каждого окна: размер окна, координаты...

Данная функция возвращает описатель окна. С помощью описателя можно обращаться к окну, это примерно как идентификатор.

Обратите внимание, что здесь очень много новых типов. На самом деле они все старые, просто переопределены. Например: HWND - это переопределение типа HANDLE, который в свою очередь является переопределением PVOID, который в свою очередь является переопределением void*. Как глубоко закопана правда! Но всё же тип HWND - это указатель на void.

Окно состоит из нескольких частей. Практически в каждой программе вы увидите: заголовок окна, системное меню (если нажать на иконку приложения в левой верхней части окна), три системные кнопки для работы с окном: свернуть, развернуть на весь экран и закрыть. Также, практически всегда в приложении присутствует меню. Вот как раз последнего у нас точно не будет. И, конечно же, большую часть окна занимает т.н. клиентская область, в которой обычно и работает пользователь.

Это что касается оконного режима. Довольно долго мы будем практиковаться с DiectX именно в окне - не будем пользоваться полноэкранным режимом.

Обработка сообщений (Message handling)

В программах под Windows обязательно реализована обработка сообщений.

Например, когда пользователь нажимает какую-нибудь клавишу на клавиатуре, генерируется сообщение, что была нажата клавиша. Затем это сообщение поступает в приложение, которое было активным, когда пользователь нажал клавишу.

Здесь у нас произошло событие (event) - была нажата клавиша.

Событием может быть: перемещение курсора мыши, смена фокуса приложения, нажатие клавиши клавиатуры, закрытие окна.

Так вот, когда происходит какое-либо событие, операционная система создаёт сообщение: была нажата такая-то клавиша, координаты курсора мыши изменились, открылось новое окно.

Сообщения может создавать как операционная система, так и различные приложения.

Сообщение представляет собой структуру, и выглядят следующим образом:

```
typedef struct tagMSG {
HWND hwnd;      // окно, которое получит это сообщение
UINT message;   // код сообщения
WPARAM wParam;  // параметр
LPARAM lParam;  // параметр
```

```
DWORD time;     // время, когда произошло сообщение
POINT pt;       // координаты курсора мыши
} MSG;
```

Оконная процедура (Window procedure - WndProc)

Как только до него дошла очередь, оно обрабатывается. Для обработки сообщений в каждой программе должна существовать специальная функция - оконная процедура. Обычно она называется WndProc (от Window Procedure).

Вызов оконной процедуры расположен в основном цикле программы и выполняется при каждой итерации цикла.

Сообщения (в виде структурных переменных MSG) попадают в данную функцию в виде параметров: описатель окна, идентификатор сообщения и два параметра. Внутри оконной процедуры расположено ветвление switch, в котором идёт проверка идентификатора сообщения.

И последнее - основной цикл. Он очень прост. Каждую итерацию цикла проверяется очередь сообщений приложения. Если в очереди сообщений есть сообщение, оно вытаскивается из очереди. Затем в теле цикла происходит вызов оконной процедуры, чтобы обработать взятое из очереди сообщение.

3.3 Модуль D3DDevice

В данном модуле происходит инициализация графического окна Direct3D для прорисовки игровой сцены, а также интерфейса IDirectInput8 для обработки нажатий клавиш и мыши.

Инициализация Direct3D начинается с запроса указателя на интерфейс IDirect3D9. Это выполняется с помощью простого вызова специальной функции Direct3D, как показано в приведенном ниже фрагменте кода:

```
IDirect3D9* _d3d9;
_d3d9 = Direct3DCreate9(D3D_SDK_VERSION);
```

Единственным параметром функции Direct3DCreate9 всегда должна быть константа D3D_SDK_VERSION, гарантирующая, что при построении приложения будут использованы правильные заголовочные файлы. Если при работе функции возникли ошибки, она возвращает нулевой указатель.

Объект IDirect3D9 используется для двух вещей: перечисления устройств и создания объекта IDirect3DDevice9. Перечисление устройств подразумевает определение возможностей, видеорежимов, форматов и другой информации о каждой установлденной в системе видеокарте. Например, чтобы создать

представляющий физическое устройство объект IDirect3DDevice9, необходимо указать поддерживаемую устройством конфигурацию видеорежима и формата. Чтобы найти такую работающую конфигурацию, используются методы перечисления IDirect3D9.

Следующий этап процесса инициализации — заполнение экземпляра структуры D3DPRESENT_PARAMETERS. Эта структура используется для задания ряда характеристик объекта IDirect3DDevice9, который мы будем создавать, и объявлена следующим образом:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
   UINT      BackBufferWidth;
   UINT      BackBufferHeight;
   D3DFORMAT BackBufferFormat;
   UINT      BackBufferCount;
   D3DMULTISAMPLE_TYPE MultiSampleType;
   DWORD     MultiSampleQuality;
   D3DSWAPEFFECT SwapEffect;
   HWND      hDeviceWindow;
   BOOL      Windowed;
   BOOL      EnableAutoDepthStencil;
   D3DFORMAT AutoDepthStencilFormat;
   DWORD     Flags;
   UINT      FullScreen_RefreshRateInHz;
   UINT      PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

- BackBufferWidth — Ширина поверхности вторичного буфера в пикселах.
- BackBufferHeight — Высота поверхности вторичного буфера в пикселах.
- BackBufferFormat — Формат пикселей во вторичном буфере (т.е. для 32-разрядного формата укажите D3DFMT_A8R8G8B8).
- BackBufferCount — Количество используемых вторичных буферов. Обычно мы задаем значение 1, чтобы указать, что нам нужен только один вторичный буфер.
- MultiSampleType — Используемый для вторичного буфера тип множественной выборки. Подробная информация об этом параметре содержится в документации SDK.
- MultiSampleQuality — Уровень качества множественной выборки. Подробная информация об этом параметре содержится в документации SDK.
- SwapEffect — Член перечисления D3DSWAPEFFECT, указывающий, как будет осуществляться переключение буферов в цепочке переключений. Для

большинства случаев можно указать значение D3DSWAPEFFECT_DISCARD.

- hDeviceWindow — Дескриптор связанного с устройством окна. Укажите дескриптор того окна приложения, в котором вы хотите выводить изображение.
- Windowed — Укажите true для запуска приложения в оконном режиме или false — для работы в полноэкранном режиме.
- EnableAutoDepthStencil — укажите значение true, чтобы Direct3D автоматически создал и поддерживал буферы глубины и трафарета.
- AutoDepthStencilFormat — формат буферов глубины и трафарета (например, для 24-разрядного буфера глубины с 8 разрядами, зарезервированными для буфера трафарета, укажите D3DFMT_D24S8).
- Flags&nbnsp;— Дополнительные параметры. Укажите ноль (если флаги отсутствуют) или член набора D3DPRESENTFLAG. Полный список допустимых флагов приведен в документации. Здесь мы рассмотрим два наиболее часто используемых:
- D3DPRESENTFLAG_LOCKABLE_BACKBUFFER — Указывает, что вторичный буфер может быть заблокирован. Обратите внимание, что использование блокируемого вторичного буфера снижает производительность.
- D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL — указывает, что буфер глубины и трафарета после показа вторичного буфера становится некорректным. Под словом «некорректным» мы подразумеваем, что данные, хранящиеся в буфере глубины и трафарета могут стать неверными. Это может увеличить производительность.
- FullScreen_RefreshRateInHz — Частота кадров; используйте частоту кадров по умолчанию, указав значение D3DPRESENT_RATE_DEFAULT.
- PresentationInterval — член набора D3DPRESENT. Полный список допустимых значений приведен в документации. Наиболее часто используются следующие два:
- D3DPRESENT_INTERVAL_IMMEDIATE — Показ выполняется немедленно.
- D3DPRESENT_INTERVAL_DEFAULT — Direct3D сам выбирает частоту показа. Обычно она равна частоте кадров.

Заполнив структуру D3DPRESENT_PARAMETERS мы можем создать объект IDirect3DDevice9 с помощью следующего метода:

```
HRESULT IDirect3D9::CreateDevice(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
```

```
    HWND hFocusWindow,
    DWORD BehaviorFlags,
    D3DPRESENT_PARAMETERS *pPresentationParameters,
    IDirect3DDevice9** ppReturnedDeviceInterface
);
```

- Adapter — указывает физический видеоадаптер, который будет представлять создаваемый объект IDirect3DDevice9.
- DeviceType — задает тип используемого устройства (т.е. аппаратное устройство (D3DDEVTYPE_HAL) или программное устройство (D3DDEVTYPE_REF)).
- hFocusWindow — дескриптор окна с которым будет связано устройство. Обычно это то окно, в которое будет выводиться изображение и для наших целей мы здесь задаем тот же дескриптор, который указан в члене d3dpp.hDeviceWindow структуры D3DPRESENT_PARAMETERS.
- BehaviorFlags — в этом параметре указывается значение D3DCREATE_HARDWARE_VERTEXPROCESSING либо D3DCREATE_SOFTWARE_VERTEXPROCESSING.
- pPresentationParameters — указывается инициализированный экземпляр структуры D3DPRESENT_PARAMETERS, задающий параметры устройства.
- ppReturnedDeviceInterface — возвращает указатель на созданное устройство.

Инициализация DirectInput включает следующие шаги:

- Получение интерфейса IDirectInput8.
- Создание устройства.
- Установка формата данных.
- Установка уровня взаимодействия с операционной системой.
- Получение доступа к устройству ввода.

После инициализации программа сможет принимать ввод от устройства.

3.4 Модуль Camera

Здесь реализовано гибкое управление камерой игрока для его перемещения по игровому миру.

Мы определяем местоположение и ориентацию камеры относительно мировой системы координат с помощью четырех *векторов камеры* (*camera vectors*): *правого вектора* (*right vector*), *верхнего вектора* (*up vector*), *вектора взгляда* (*look vector*) и *вектора местоположения* (*position vector*), как показано на рис. 3 Эти векторы образуют локальную систему координат камеры, описанную в

мировой системе координат. Поскольку правый вектор, верхний вектор и вектор взгляда описывают ориентацию камеры в мировом пространстве, мы иногда будем называть их *векторами ориентации* (*orientation vectors*). Векторы ориентации должны быть *ортонормальными*. Набор векторов называется ортонормальным, если каждый вектор перпендикулярен остальным и длина всех векторов равна единице. Причина данного ограничения в том, что позже мы подставим эти векторы в строки матрицы, а матрица в которой векторы-строки являются ортонормальными будет ортогональной.

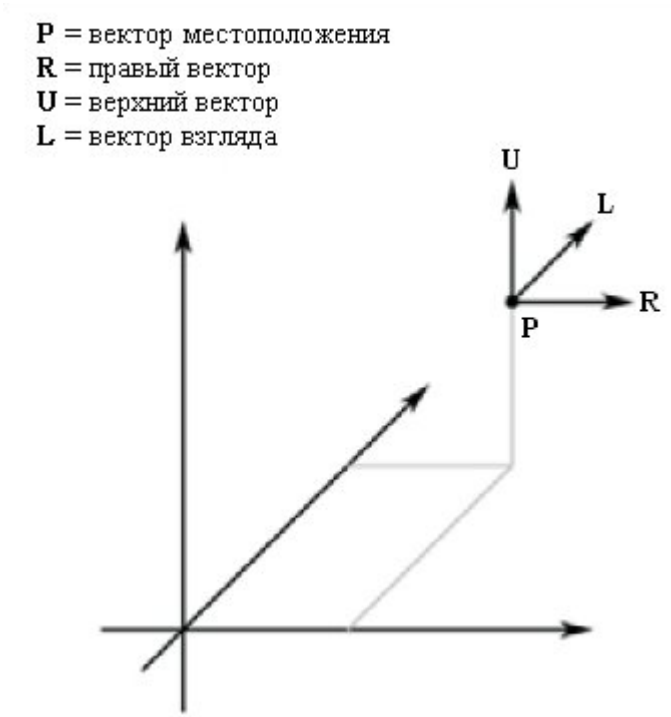

Рисунок 3 - Мировое пространство.

Эти четыре описывающих камеру вектора позволяют выполнять с камерой следующие шесть операций:

- Поворот относительно правого вектора (наклон).
- Поворот относительно верхнего вектора (отклонение).
- Поворот относительно вектора взгляда (вращение).
- Сдвиг вдоль правого вектора.
- Подъем вдоль верхнего вектора.
- Передвижение вдоль вектора взгляда.

Эти шесть операций позволяют нам перемещать камеру вдоль трех осей и вращать ее вокруг этих же осей, что дает в совокупности шесть степеней свободы.

Вычисление матрицы вида

Сейчас мы покажем, как можно вычислить матрицу преобразования вида, на основании заданных векторов камеры. Предположим, что векторы p = (p_x, p_y, p_z), r = (r_x, r_y, r_z), u = (u_x, u_y, u_z) и d = (d_x, d_y, d_z) являются соответственно вектором местоположения, правым вектором, верхним вектором и вектором взгляда камеры.

Преобразование пространства вида трансформирует геометрию мира таким образом, что камера помещается в начало координат и ее оси совпадают с осями мировой системы координат (рис. 4).

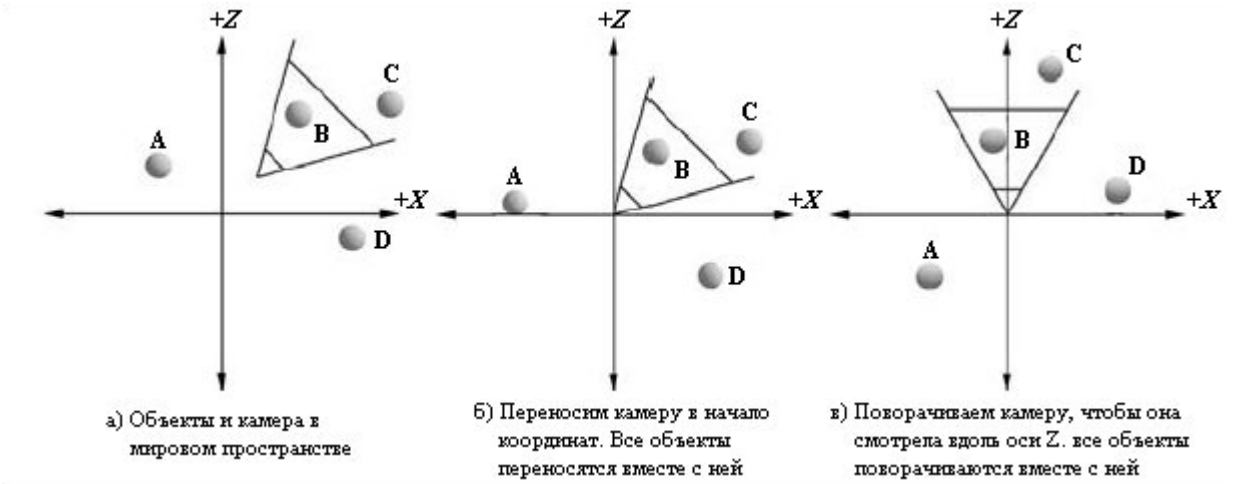

Рисунок 4 - Преобразование пространства вида

Следовательно, нам нужна матрица преобразования V, отвечающая следующим требованиям:

- pV = (0, 0, 0) — Матрица преобразования V перемещает камеру в начало координат.
- rV = (1, 0, 0) — Матрица V преобразует правый вектор камеры таким образом, чтобы он совпадал с осью X мировой системы координат.
- uV = (0, 1, 0) — Матрица V преобразует верхний вектор камеры таким образом, чтобы он совпадал с осью Y мировой системы координат.
- dV = (0, 0, 1) — Матрица V преобразует вектор взгляда камеры таким образом, чтобы он совпадал с осью Z мировой системы координат.

Задачу нахождения такой матрицы можно разделить на две части: 1) перемещение, в результате которого камера окажется в начале системы координат и 2) поворот, в результате которого оси камеры будут совпадать с осями мировой системы координат.

Перемещение

Чтобы точка p совпадала с началом координат, ее необходимо переместить на –p, поскольку p – p = 0. Так что отвечающая за перемещение часть матрицы преобразования вида задается следующей матрицей:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{bmatrix}$$

Вращение

Выравнивание трех векторов камеры по осям мировой системы координат требует большего объема работ. Нам необходима матрица поворота A, размером 3 × 3, которая выравнивает правый вектор, верхний вектор и вектор взгляда камеры с осями X, Y и Z мировой системы координат. Такая матрица должна удовлетворять следующим трем выражениям:

$$\mathbf{rA} = \begin{bmatrix} r_x, & r_y, & r_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1, & 0, & 0 \end{bmatrix}$$

$$\mathbf{uA} = \begin{bmatrix} u_x, & u_y, & u_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 0, & 1, & 0 \end{bmatrix}$$

$$\mathbf{dA} = \begin{bmatrix} d_x, & d_y, & d_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 0, & 0, & 1 \end{bmatrix}$$

Мы работаем с матрицами размера 3 × 3 потому что для представления вращения нам не нужны однородные координаты. Позднее мы вернемся к привычным матрицам размера 4 × 4.

Поскольку во всех трех приведенных выше формулах коэффициенты матрицы A одни и те же, можно свести все три формулы в одну, переписав их следующим образом:

$$\mathbf{BA} = \begin{bmatrix} r_x & r_y & r_z \\ u_x & u_y & u_z \\ d_x & d_y & d_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Существует несколько способов вычисления матрицы A, но легко заметить, что матрица A является инверсией матрицы B потому что $BA = BB^{-1} = I$. Поскольку матрица B является ортогональной (ее векторы-строки являются ортонормальными), ее инверсия совпадает с результатом транспонирования. Следовательно, преобразование, выравнивающее векторы ориентации по осям мировой системы координат, выглядит так:

$$\mathbf{B}^{-1} = \mathbf{B}^{\mathrm{T}} = \mathbf{A} = \begin{bmatrix} r_x & u_x & d_x \\ r_y & u_y & d_y \\ r_z & u_z & d_z \end{bmatrix}$$

Комбинирование обеих частей

Теперь мы дополняем A до матрицы 4 × 4 и комбинируем матрицу перемещения с матрицей вращения, получая матрицу преобразования вида V:

$$\mathbf{TA} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{bmatrix} \begin{bmatrix} r_x & u_x & d_x & 0 \\ r_y & u_y & d_y & 0 \\ r_z & u_z & d_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r_x & u_x & d_x & 0 \\ r_y & u_y & d_y & 0 \\ r_z & u_z & d_z & 0 \\ -\mathbf{p}\cdot\mathbf{r} & -\mathbf{p}\cdot\mathbf{u} & -\mathbf{p}\cdot\mathbf{d} & 1 \end{bmatrix} = \mathbf{V}$$

Вращение относительно произвольной оси

Чтобы реализовать методы для поворота нашей камеры, нам необходима возможность вращать ее относительно произвольной оси. Для этой цели библиотека D3DX предоставляет следующую функцию:

D3DXMATRIX *D3DXMatrixRotationAxis(

```
    D3DXMATRIX *pOut,      // возвращает матрицу вращения
    CONST D3DXVECTOR3 *pV, // ось вращения
    FLOAT Angle            // угол поворота в радианах
);
```

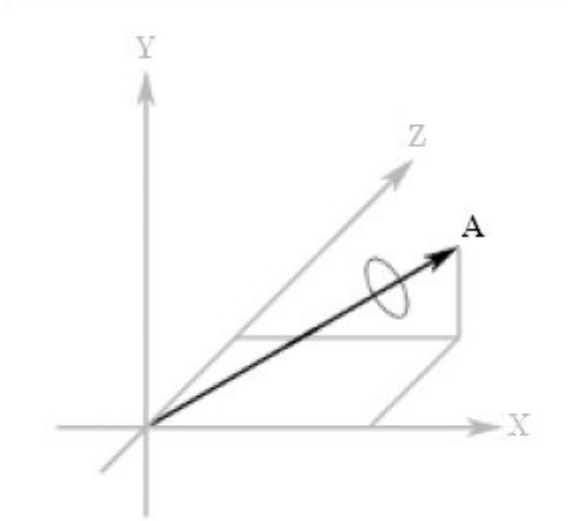

Рисунок 5 - *Вращение относительно произвольной оси, определенной вектором* ***A***

Наклон, рыскание и вращение

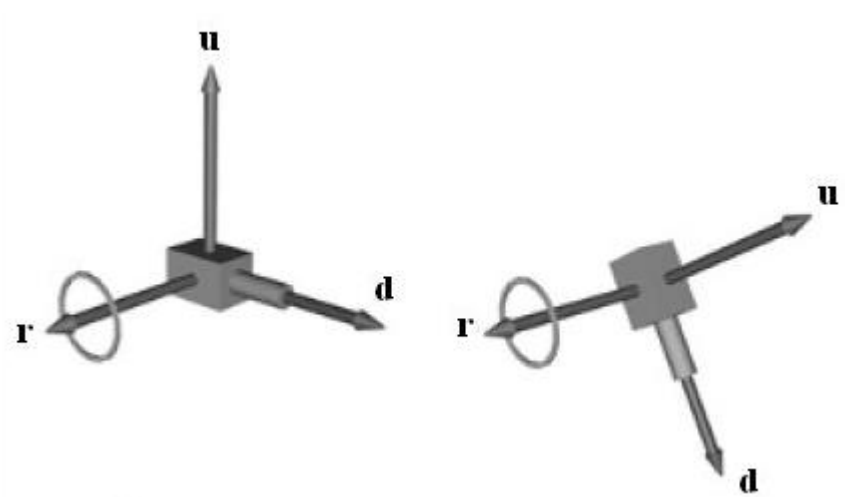

Рисунок 6 - Наклон, или поворот относительно правого вектора камеры

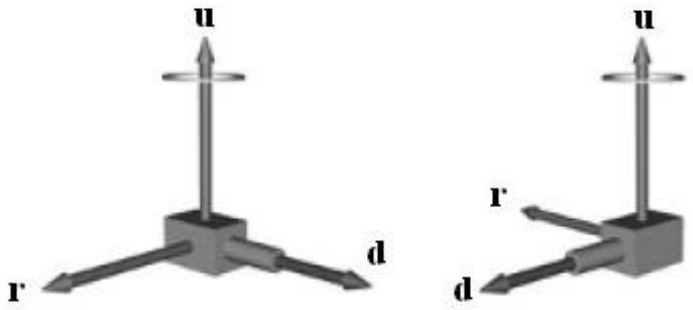

Рисунок 7 - Рыскание, или поворот относительно верхнего вектора камеры

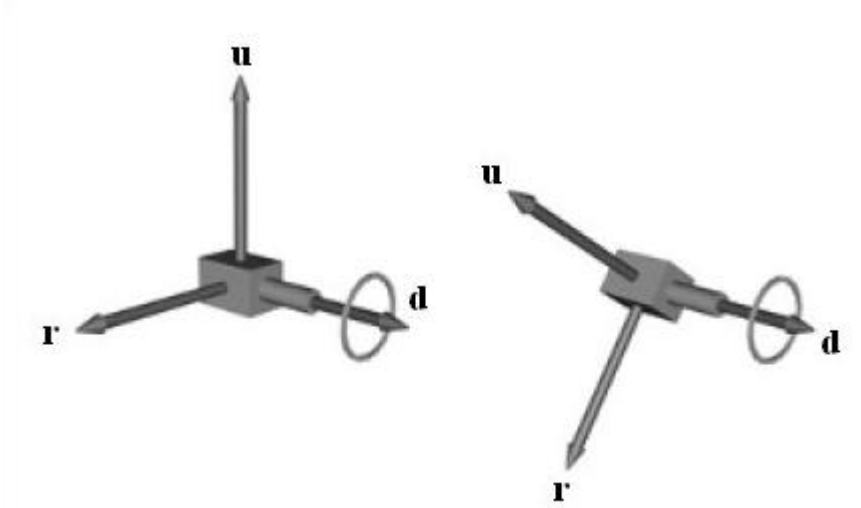

Рисунок 8 - Вращение, или поворот относительно вектора взгляда камеры

Как видите при наклоне камеры мы должны повернуть верхний вектор и вектор взгляда относительно правого вектора на заданный угол. Аналогично при рыскании мы поворачиваем правый вектор и вектор взгляда относительно верхнего вектора на указанный угол. А при вращении камеры мы поворачиваем на заданный угол верхний и правый векторы относительно вектора взгляда.

Теперь вы видите, почему нам необходима функция D3DXMatrixRotationAxis, ведь любой из трех векторов относительно которых выполняется поворот, в мировой системе координат может иметь произвольную ориентацию.

Реализация методов для наклона, рыскания и вращения соответствует приведенному выше описанию. Однако для типа камеры LANDOBJECT добавлен ряд ограничений. В частности, неестественно выглядит отклонение от заданного курса при наклоне или вращении наземного объекта. Поэтому при рыскании в модели LANDOBJECT мы выполняем поворот относительно

оси Y мировой системы координат, а не относительно верхнего вектора камеры. Кроме того, мы полностью запрещаем вращение наземных объектов.

Ходьба, сдвиг и полет

Говоря о ходьбе мы подразумеваем перемещение в направлении взгляда (то есть вдоль вектора взгляда). Сдвиг — это перемещение в сторону относительно направления взгляда, то есть перемещение вдоль правого вектора. Ну и когда мы говорим о полете, подразумевается перемещение вдоль верхнего вектора. Чтобы переместиться вдоль одной из этих осей мы просто прибавляем к вектору местоположения камеры вектор заданной длины, указывающий в том же направлении, что и ось, вдоль которой перемещается камера (рис. 8).

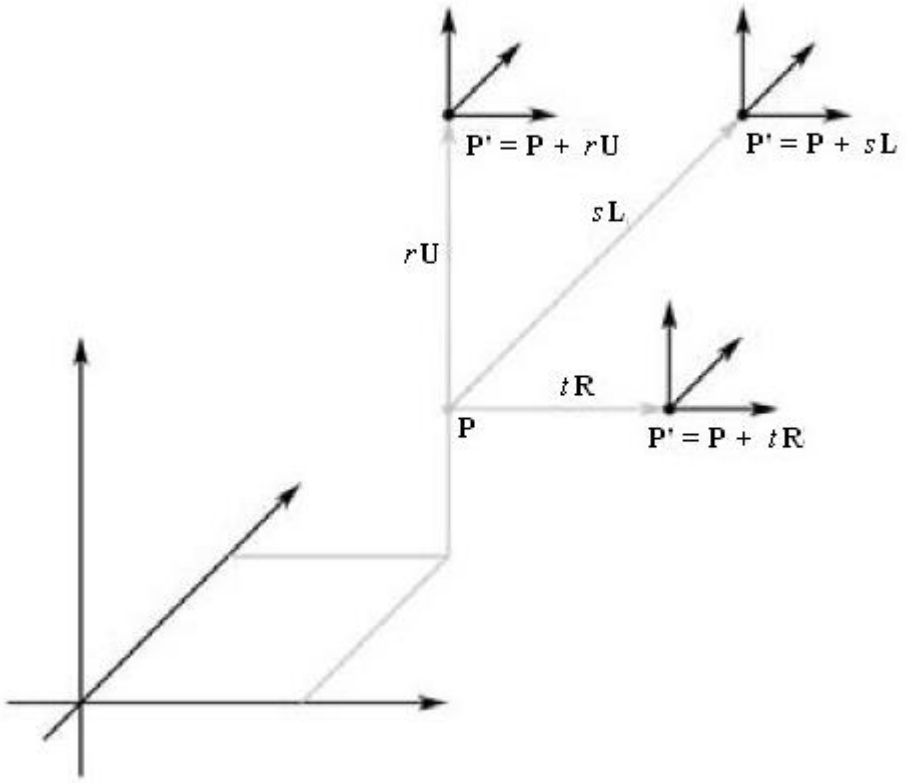

Рисунок 9 - Перемещение вдоль векторов ориентации камеры

Как и в случае с поворотами камеры, для наземных объектов мы вносим ряд ограничений. К примеру, объекты LANDOBJECT не могут находиться в воздухе, даже если верхний вектор изменяется в результате движения вперед или сдвига вбок. Следовательно, мы должны ограничить их перемещение плоскостью XZ. Но, поскольку наземные объекты могут изменять свою высоту, взбираясь на лестницы или холмы, мы предоставляем метод Camera::setPosition, позволяющий вручную разместить камеру в требуемом месте и на требуемой высоте.

3.5 Модуль Terrain

Здесь реализована генерация ландшафта.

Сетка ландшафта представляет собой обычную сетку с треугольными ячейками, подобную показанной на рис. 10 (а), у которой для каждой вершины сетки высота задана таким образом, чтобы сетка моделировала плавный переход от гор к долинам, подобный природному ландшафту (рис. 10(б)). И, конечно же, мы накладываем тщательно подобранные текстуры, изображающие песчаные пляжи, покрытые травой склоны и заснеженные вершины (рис. 10(в)).

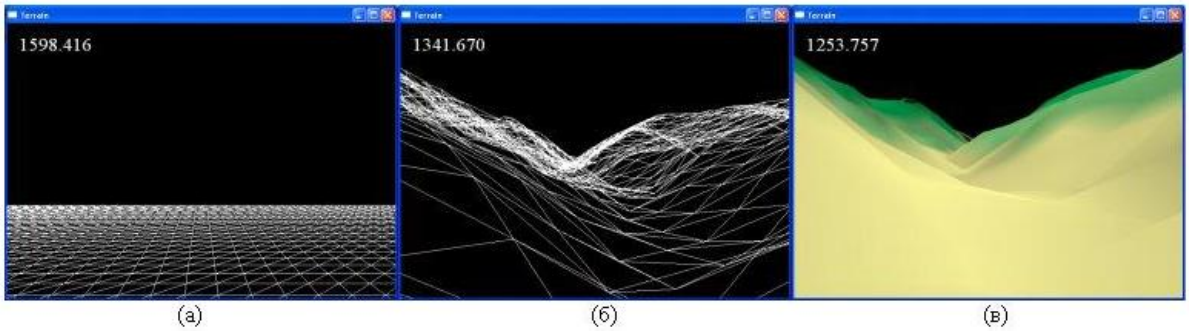

Рисунок 10 - (а) Сетка с треугольными ячейками. (б) Сетка с плавным изменением высот. (в) Освещенный и текстурированный ландшафт

В классе Terrain реализована генерация ландшафта, методом грубой силы. Под этим подразумевается, что он просто хранит данные вершин и индексов для всего ландшафта и визуализирует их. Для игр с ландшафтом небольшого размера такой подход работает, при том условии, что в компьютере установлена современная видеокарта поддерживающая аппаратную обработку вершин. Однако для тех игр, которым требуется большой ландшафт, мы должны выполнить дополнительную работу по изменению уровня детализации или отбрасыванию невидимых граней, поскольку при использовании метода грубой силы огромное количество геометрических данных, необходимых для моделирования обширных ландшафтов приведет к перегрузке видеокарты.

Карты высот

Мы используем карты высот, чтобы указать, где на нашем ландшафте находятся горы, а где — долины. Карта высот (heighmap) — это массив, в котором каждый элемент задает высоту отдельной вершины сетки ландшафта. (Существует и альтернативная реализация, в которой элемент карты высот соответствует квадрату ландшафта.) Обычно мы представляем карту высот в виде матрицы, каждый элемент которой однозначно соответствует вершине сетки ландшафта.

Когда мы сохраняем карту высот на диске, для каждого ее элемента обычно отводится один байт, так что значения высоты могут находиться в

диапазоне от 0 до 255. Хотя диапазон от 0 до 255 достаточен для хранения перепадов высот нашего ландшафта, в приложении может потребоваться масштабирование этого значения, чтобы оно соответствовало масштабу нашего трехмерного мира. Предположим, что в качестве единиц измерения для нашего трехмерного мира мы выбрали сантиметры, тогда диапазона значений от 0 до 255 будет недостаточно, чтобы изобразить что-нибудь интересное. Поэтому при загрузке данных в приложение мы выделяем для каждого элемента карты высот переменную типа int или float. Это позволяет масштабировать значения за пределы диапазона от 0 до 255, чтобы они соответствовали любому требуемому масштабу.

Одним из возможных представлений карты высот является карта с градациями серого, где темные области соответствуют низинам, а светлые — возвышенностям. Такая карта показана на рис. 11.

Рисунок 11 - Карта высот в виде градаций серого

Создание карты высот

Карта высот может быть создана либо программно, либо с помощью графического редактора, например GIMP или Photoshop. Использование графического редактора — более простой способ, позволяющий интерактивно создавать ландшафты и видеть полученный результат. Кроме того, можно создавать необычные карты высот, пользуясь дополнительными возможностями графического редактора, такими как фильтры.

По завершению рисования карты высот, сохраняем ее как 32-разрядный файл формата BMP. Далее необходимо считать данные о высотах вершин из этого файла в соответствующий массив.

Создание геометрии ландшафта

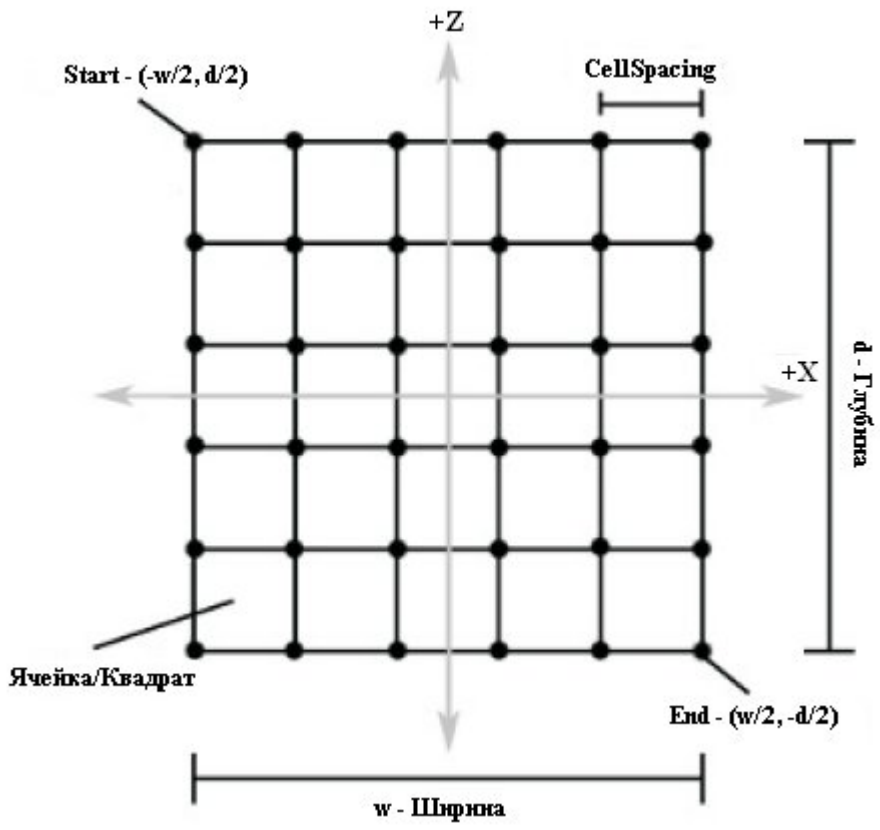

Рисунок 12 - Свойства размеченной треугольной сетки. Точки на пересечении линий сетки обозначают вершины

Рисунок 12 иллюстрирует свойства ландшафта, термины и специальные точки, на которые мы будем ссылаться. Размер ландшафта определяется путем указания количества вершин в строке, количества вершин в столбце и размера ячейки. Все эти значения мы передаем в класс Terrain. Кроме того, мы передаем указатель на связанное с ландшафтом устройство, строку с именем файла, содержащего карту высот и коэффициент масштабирования, используемый при масштабировании элементов карты высот.

Вычисление вершин

Во время последующего обсуждения смотрите на рис. 12. Чтобы создать вершины нашей сетки мы просто начинаем генерировать данные вершин с той, которая отмечена на рисунке словом *start* и заполняем вершины ряд за рядом, пока не дойдем до той, которая отмечена словом *end*; при этом расстояние между вершинами задается параметром *cellSpacing*. Это позволяет нам получить

координаты X и Z, но как насчет координаты Y? Координата Y берется из соответствующего элемента загруженной карты высот.

При вычислении координат текстуры, учитывая рис. 13, получаем простой сценарий, определяющий соответствие координат текстуры (u, v) вершине ландшафта (i, j) по следующей формуле:

$$u = j * uCoordIncrementSize$$

$$v = i * vCoordIncrementSize$$

где

$$uCoordIncrementSize = \frac{1}{numCellCols}$$

$$vCoordIncrementSize = \frac{1}{numCellRows}$$

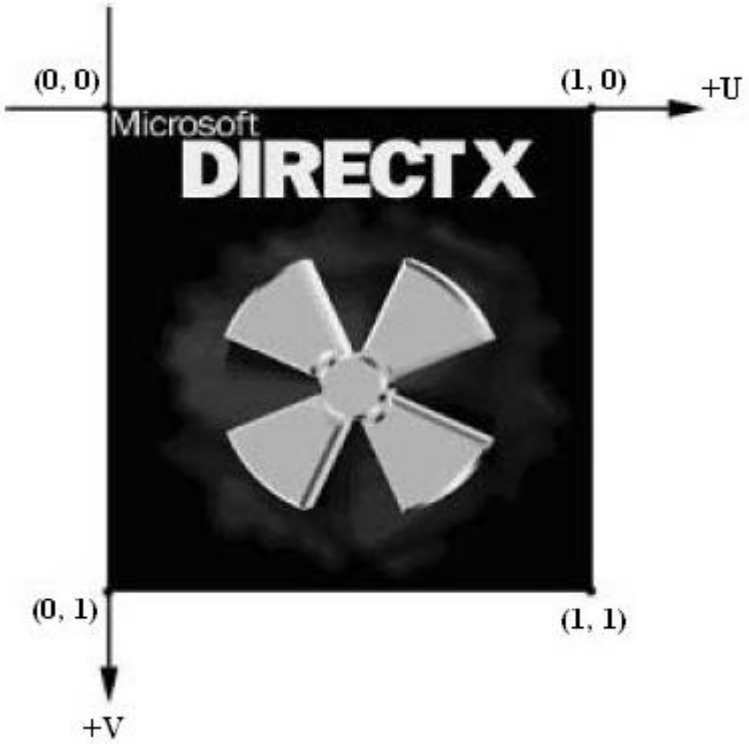

Рисунок 13 - Соответствие между вершинами ландшафта и координатами текстур

Вычисление индексов — определение треугольников

Чтобы вычислить индексы сетки мы просто в цикле перебираем все ее квадраты, начиная с верхнего левого и заканчивая правым нижним, как показано на рис. 14, и формируем два треугольника, образующие квадрат.

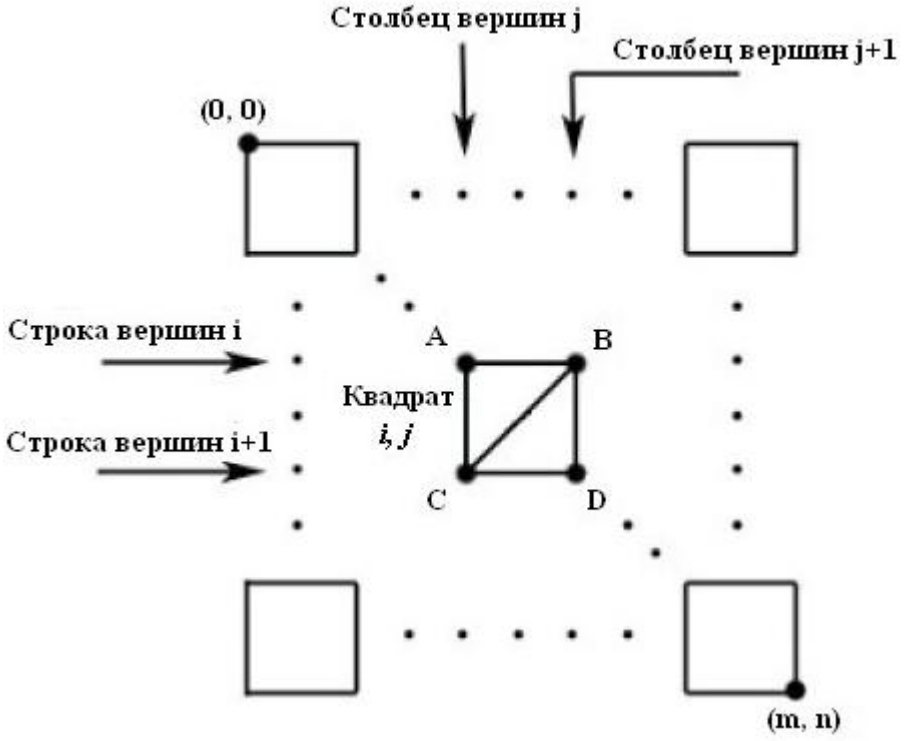

Рисунок 14 - Вершины квадрата

Главным трюком здесь является общая формула, позволяющая вычислить индексы треугольников, образующих квадрат в позиции (i, j). Рис. 14 позволяет вывести общую формулу для квадрата(i, j):

ΔABC = { i*numVertsPerRow + j,
i*numVertsPerRow + j + 1,
(i + 1)*numVertsPerRow + j }

ΔCBD = { (i + 1)*numVertsPerRow + j,
i*numVertsPerRow + j + 1,
(i + 1)*numVertsPerRow + j + 1 }

Текстурирование

Наиболее очевидный способ — загрузить ранее подготовленную текстуру из файла и использовать ее с помощью метода D3DXCreateTextureFromFile.

Альтернативным способом текстурирования ландшафта является программное вычисление текстуры; это означает, что мы создаем «пустую» текстуру и вычисляем цвет каждого ее текселя в коде на основании некоторых предопределенных параметров. В нашем примере таким параметром является высота вершины ландшафта.

Программная генерация текстуры выполняется в методе Terrain::genTexture. Сперва мы создаем пустую текстуру с помощью метода D3DXCreateTexture. Затем мы блокируем текстуру верхнего уровня (помните, что это детализируемая текстура и у нее есть несколько уровней детализации). После этого мы в цикле перебираем тексели и назначаем их цвет. Цвета текселей зависят от высоты вершин квадрата сетки, которому они принадлежат. Идея заключается в том, что низкие участки ландшафта окрашиваются в цвет песчанного пляжа, участки со средней высотой — в цвет травы, а высокие части ландшафта — в цвет снежных вершин. Мы считаем, что высота квадрата это высота его верхнего левого угла.

Назначив цвета всем текселям, мы должны сделать некоторые из них темнее или светлее в зависимости от того, под каким углом солнечный свет (моделируемый с помощью источника направленного света) падает на квадрат ландщафта, соответствующий данному текселю. Все это делается в методе Terrain::lightTerrain.

В конце метода Terrain::genTexture осуществляется вычисление текселей остальных уровней детализируемой текстуры. Это делается с помощью функции D3DXFilterTexture.

Освещение

Метод Terrain::genTexture обращается к функции Terrain::lightTerrain, которая, как говорит ее имя, выполняет освещение ландшафта для увеличения реализма сцены. Поскольку мы уже вычислили цвета текстуры ландшафта, нам осталось вычислить только коэффициент затенения, который делает отдельные участки темнее или светлее в зависимости от их расположения относительно источника света. В данном разделе мы исследуем такую технику. Вы можете недоумевать, почему мы занялись освещением ландшафта, а не позволили Direct3D все сделать за нас. У самостоятельного выполнения вычислений есть три преимущества:

- Мы экономим память, поскольку нам не надо хранить нормали вершин.
- Так как ландшафты статичны и мы не будем перемещать источники света, можно заранее рассчитать освещение, освободив то время, которое Direct3D тратил бы на расчет освещения ландшафта в реальном времени.
- Мы попрактикуемся в математике, познакомимся с базовыми концепциями освещения и поработаем с функциями Direct3D.

Техника освещения, которую мы будем использовать для вычисления оттенков ландшафта является одной из самых простых и известна как *рассеянное освещение* (*diffuse lighting*). Мы используем параллельный источник света, который описываем путем указания направления на источник света, являющегося противоположным тому направлению, в котором падают испускаемые источником лучи. Например, если мы хотим чтобы лучи света падали с неба вертикально вниз в направлении *lightRaysDirection* = (0, –1, 0), то направлением на источник света будет указывающий в противоположном направлении вектор *directionToLight* = (0, 1, 0). Обратите внимание, что мы используем единичные векторы.

Затем для каждого квадрата ландшафта мы вычисляем угол между вектором освещения и нормалью к поверхности квадрата.

На рис. 15 видно, что чем больше угол, тем больше поверхность квадрата отворачивается от источника света и тем меньше света попадает на поверхность. Соответственно, чем меньше угол, тем больше поверхность квадрата повернута к источнику света и тем больше света она получает. Кроме того, обратите внимание, что если угол между вектором освещения и нормалью поверхности больше 90 градусов, поверхность вообще не освещена.

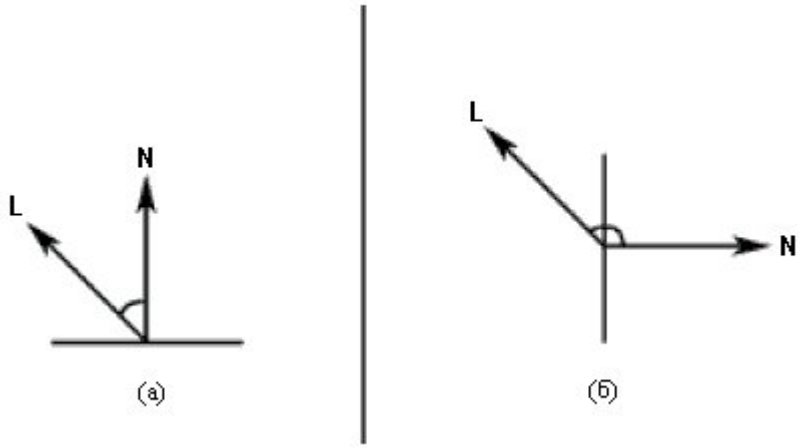

Рисунок 15 - Угол между вектором освещения L и нормалью поверхности N определяет сколько света получает поверхность. На рисунке (а) угол меньше 90 градусов. На рисунке (б) угол больше 90 градусов.

Используя угловые отношения между вектором освещения и нормалью поверхности можно вычислить коэффициент затенения, находящийся в диапазоне [0, 1], который определяет сколько света получает поверхность. Большие углы представляются близкими к нулю значениями коэффициента. Когда цвет умножается на близкий к нулю коэффициент затенения, он становится темнее, а это именно то, что нам надо. С другой стороны, малые углы представляются

близкими к единице значениями коэффициента, и умножение на этот коэффициент практически не меняет яркость цвета.

Вычисление затенения квадрата

Направление на источник света нам дано в виде нормализованного вектора L. Чтобы вычислить угол между L и нормалью квадрата N нам сперва необходимо получить N. Это тривиальная задача, решаемая с помощью векторного произведения. Но сначала мы должны получить два ненулевых и непараллельных вектора, лежащих в той же плоскости, что и квадрат. На рис. 15 эти два вектора обозначены u и v:

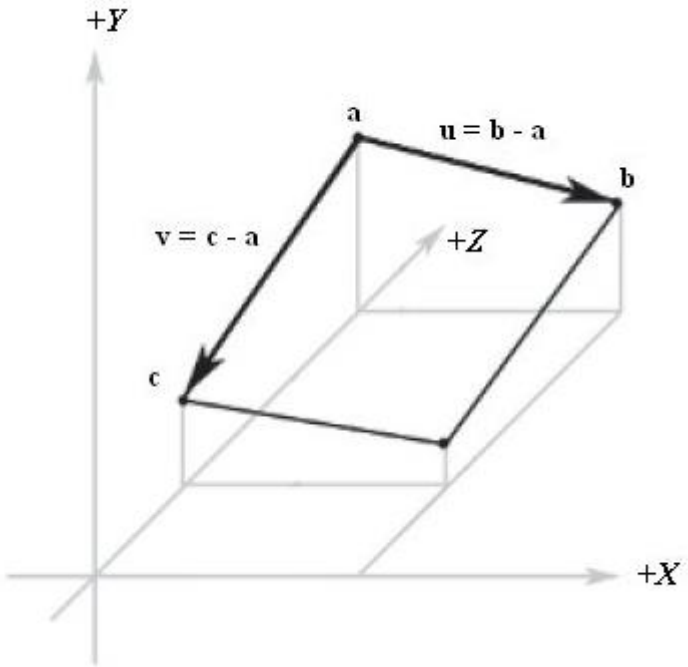

Рисунок 15 - Вычисление двух векторов, находящихся в одной плоскости с квадратом

Затенение ландшафта

Теперь, когда мы узнали как выполнить затенение отдельного квадрата, можно выполнить затенение всех квадратов ландшафта. Мы просто перебираем в цикле все квадраты ландшафта, вычисляем для каждого из них коэффициент затенения и умножаем цвет соответствующего квадрату текселя на полученный коэффициент. В результате квадраты, получающие мало света станут темнее.

«Ходьба» по ландшафту

После того, как мы сконструировали ландшафт, хорошо добавить возможность перемещать камеру таким образом, чтобы она имитировала ходьбу по ландшафту. То есть нам надо менять высоту камеры (координату Y) в зависимости от того, в каком месте ландшафта мы находимся. Для этого мы

сначала должны определить по координатам X и Z камеры квадрат ландшафта в котором мы находимся. Все это делает функция Terrain::getHeight; в своих параметрах она получает координаты X и Z камеры и возвращает высоту, на которой должна быть расположена камера, чтобы она оказалась над ландшафтом.

Сперва мы выполняем перемещение в результате которого начальная точка ландшафта будет совпадать с началом координат. Затем мы выполняем операцию масштабирования с коэффициентом равным единице деленной на размер клетки; в результате размер клетки ландшафта будет равен 1. Затем мы переходим к новой системе координат, где положительное направление оси Z направлено «вниз». Конечно, вы не найдете кода меняющего систему координат, просто помните, что ось Z направлена вниз. Все эти этапы показаны на рис. 16.

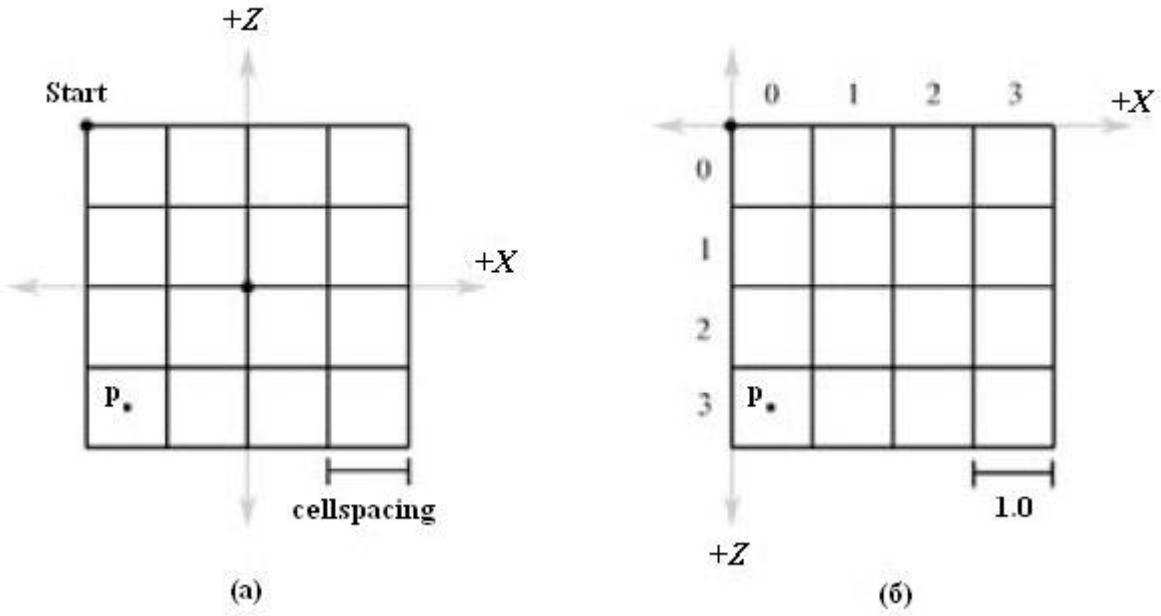

Рисунок 16 - Исходная сетка ландшафта и сетка после переноса начальной точки ландшафта в начало координат, масштабирования, делающего размер квадрата равным 1 и смены направления оси Z

Мы видим, что измененная координатная система соответствует порядку элементов матрицы. То есть верхний левый угол — это начало координат, номера столбцов увеличиваются вправо, а номера строк растут вниз. Следовательно, согласно рис 16, и помня о том, что размер ячейки равен 1, мы сразу видим что номер строки и столбца для той клетки на которой мы находимся вычисляется так:

```
float col = ::floorf(x);
float row = ::floorf(z);
```

Другими словами, номер столбца равен целой части координаты X, а номер строки — целой части координаты Z. Также вспомните, что результатом функции floor(t) является наибольшее целое число, которое меньше или равно t.

Теперь, когда мы знаем в какой ячейке находимся, можно получить высоты ее четырех вершин:

```
// A   B
// *---*
// | / |
// *---*
// C   D
float A = getHeightmapEntry(row,   col);
float B = getHeightmapEntry(row,   col+1);
float C = getHeightmapEntry(row+1, col);
float D = getHeightmapEntry(row+1, col+1);
```

Теперь мы знаем в какой ячейке находимся и высоты всех четырех ее вершин. Нам надо найти высоту (координату Y) точки ячейки с указанными координатами X и Z, где находится камера. Это нетривиальная задача, поскольку ячейка может быть наклонена, как показано на рис. 17.

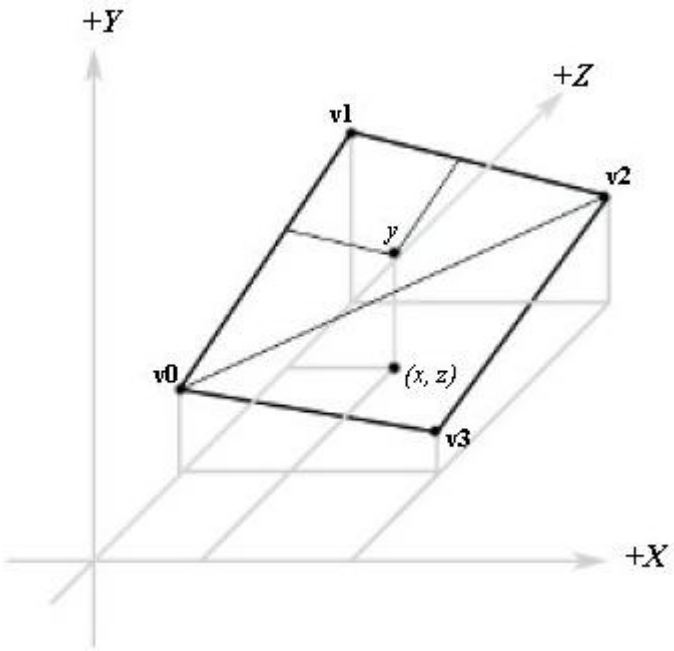

Рисунок 17 - Высота ячейки (координата Y) для заданных координат местоположения камеры X и Z

Чтобы определить высоту мы должны узнать в каком треугольнике ячейки мы находимся. Вспомните, что каждая ячейка визуализируется как два треугольника. Чтобы определить в каком треугольнике мы находимся, мы берем тот квадрат сетки в котором находимся и перемещаем его таким образом, чтобы верхняя левая вершина совпадала с началом координат.

Поскольку переменные row и col определяют местоположение левой верхней вершины той ячейки где мы находимся, необходимо выполнить перемещение на –col по оси X и –row по оси Z. Преобразование координат X и Z выполняется так:

```
float dx = x - col;
float dz = z - row;
```

Ячейка после выполнения преобразования показана на рис. 18

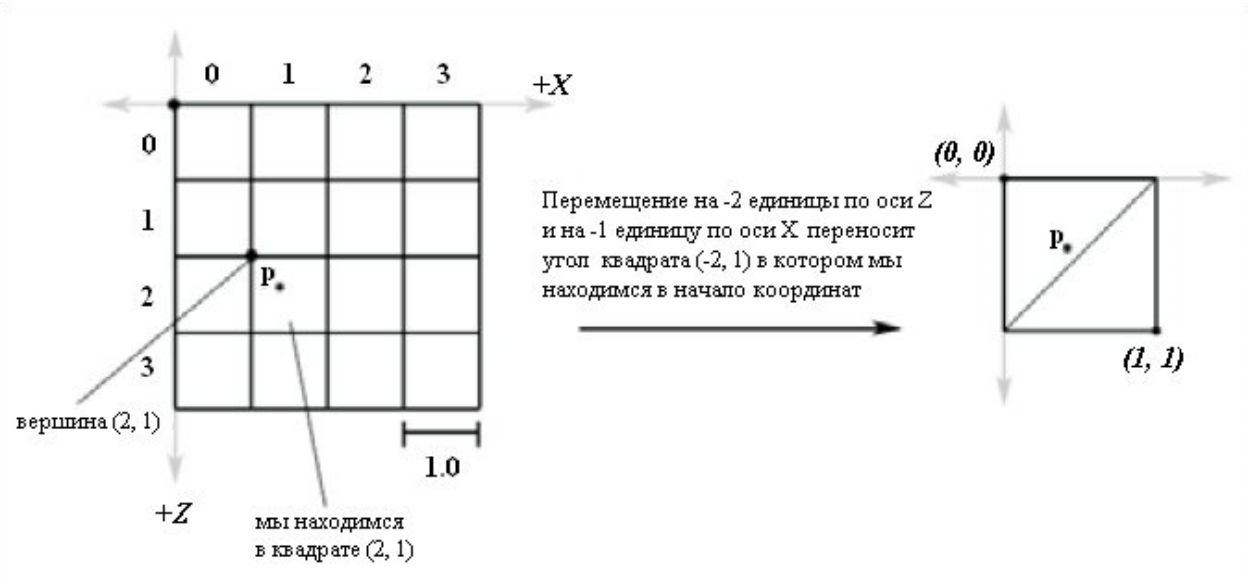

Рисунок 18 - Ячейка до и после преобразования, переносящего ее верхнюю левую вершину в начало координат

Теперь, если $dz < 1.0 - dx$ мы находимся в «верхнем» треугольнике Δv0v1v2. В ином случае мы находимся в «нижнем» треугольнике Δv0v2v3 (рис. 18).

Теперь мы посмотрим как определить высоту, если мы находимся в «верхнем» треугольнике. Для «нижнего» треугольника процесс аналогичен и ниже будет приведен код для обоих вариантов. Чтобы найти высоту когда мы находимся в «верхнем» треугольнике, надо сформировать два вектора $u = (cellSpacing, B - A, 0)$ и $v = (0, C - A, -cellSpacing)$, совпадающих со сторонами треугольника и начинающихся в точке, заданной вектором $q = (q_x, A, q_z)$, как показано на рис. 19(а) Затем мы выполняем линейную

интерполяцию вдоль вектора u на *dx* и вдоль вектора v на *dz*. Эти интерполяции показаны на рис. 19(б). Координата Y вектора (q + *dx*u + *dz*v) дает нам высоту для заданных координат X и Z; чтобы увидеть, как это происходит, вспомните геометрическую интерпретацию сложения векторов.

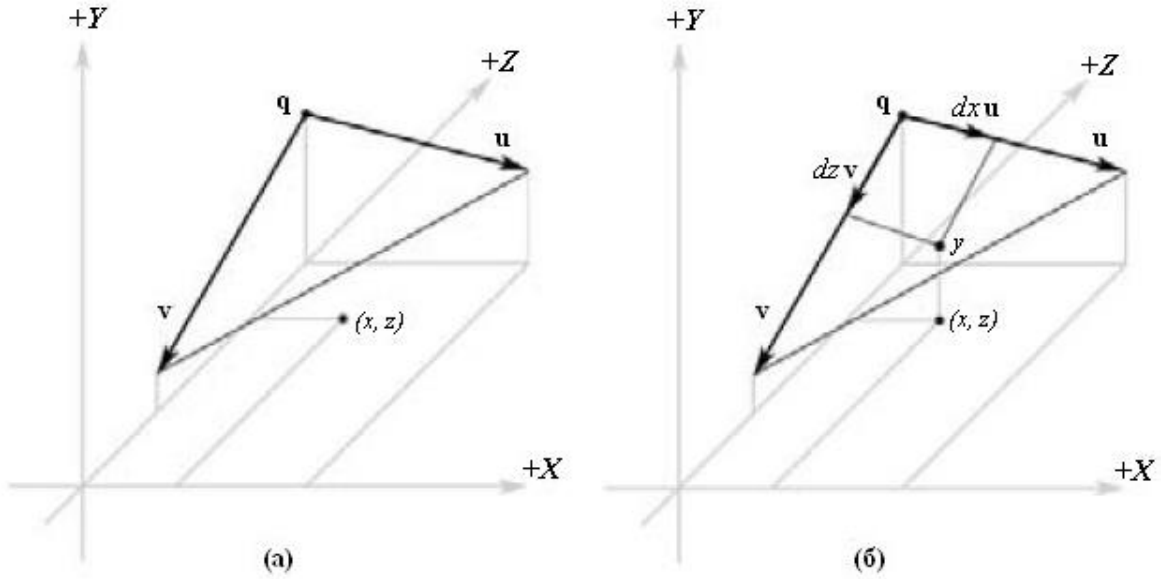

Рисунок 19 - (а) Вычисляем два вектора, совпадающих со сторонами треугольника. (б) Высота вычисляется путем линейной интерполяции **u** на dx и **v** на dz

Обратите внимание, что поскольку нас интересует только значение высоты, мы можем выполнять интерполяцию только для компоненты *y* и игнорировать остальные компоненты. В этом случае высота определяется по формуле $A + dxu_y + dzv_y$.

3.6 Проверка коллизий (столкновений)

Для определения коллизий, класса объектов, которые участвуют в этом, реализуют интерфейс Collidable:

```
class Collidable {
public:
virtual void onCollide() = 0;
virtual void someObjectMoved(Collidable *c1, BoundingBox *box, D3DXVECTOR3 pos, D3DXVECTOR3 dir) = 0;
};
```

Когда объект двигается, например главный игрок, он сообщает всем остальным объектам, об этом через метод someObjectMoved.

Вот пример использования данного метода:

```
void Model::someObjectMoved(Collidable *c, BoundingBox *_box, D3DXVECTOR3 pos, D3DXVECTOR3 dir) {
   if(mesh != 0) {
         computeBoundingBox();
         if(this != c && box->intersects(_box)) {
               c->onCollide();
         }
   }
}
```

Метод computeBoundingBox() вычисляет ограничивающий параллелепипед для объекта того класса, в котором он вызывается.

Вот структура ограничивающего параллелепипеда:

```
struct BoundingBox {
   BoundngBox() {/* инициализация */}
   bool isPointInside(D3DXVECTOR3 p) {/* проверяет попадает ли точка в заданный параллелепипед*/      }
   bool intersects(BoundingBox *b) {/*проверка пересечения данного параллелепипеда с тем, что указан в аргументе*/}
   bool intersects(D3DXVECTOR3 pos, D3DXVECTOR3 at, float* result) {/* проверка пересечения луча, с текущим параллелепипедом */}
   D3DXVECTOR3 _min; // минимальная вершина параллелепипеда в мировой системе координат
   D3DXVECTOR3 _max; // максимальная вершина параллелепипеда в мировой системе координат
   float EPSILON;
};
```

ЗАКЛЮЧЕНИЕ

В процессе работы была реализована компьютерная мини-игра, позволяющая игроку перемещаться по игровому миру и стрелять в двигающихся персонажей. Теперь этот проект поможет молодым программистам, желающим работать в этом направлении. С ее помощью они смогут освоить программирование 3D игр под DirectX на C++ и использовать эти знания и опыт в своих проектах.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Франк Луна, Введение в программирование трехмерных игр с DX9. Wordware Publishing Inc. 2003
2. Грег Снук, Создание 3D-ландшафтов в реальном времени с использованием C++ и DirectX 9. КУДИЦ-Образ. 2007
3. Фленов Михаил, Искусство программирования игр на C++. БХВ-Петербург. 2006
4. Джим Адамс, DirectX: продвинутая анимация. КУДИЦ-Образ. 2004