ТЕХНОЛОГИЧЕСКО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Д И П Л О М Н А Р А Б О Т А

Тема: Фреймуърк за разработка на 3D игри, развиващи се в космическа среда

Дипломант:
Даниел Николаев Шаприн

Научен ръководител:
Мая Милушева

С О Ф И Я

2 0 1 1

Увод

В създаването на програмните продукти е от голямо значение програмиста да не върши една и съща работа по няколко пъти и работата, която я е свършил да може да бъде използвана от други програмисти. За това, за определена група програми, се обособява код, който често ще се използва за разработването следващите продукти. Това спестява време и средства на разработчиците и им позволява да се съсредоточат в направата на по-добра функционалност на програмния продукт.

Разработката на игри не прави изключение. Всяка фирма и всеки програмист си създават база на която да стъпят, за да не се налага всеки проект да го започват от нулата. Това позволява игрите да се развиват по-бързо и играчите да играят все по-интересни и добре изглеждащи игри.

С тази дипломна работа се цели да се създаде основа, която ще помогне за разработката на различни игри, в които играча е с космически кораб в космоса.

ПЪРВА ГЛАВА

Проучване на известните развойни среди и средствата за разработка на фреймуърк за 3D игри

1.1 Сравнение на водещите библиотеки за 3D графика - OpenGL и DirectX

В 3D графиката са се наложили 2 продукта – OpenGL и DirectX. OpenGL е графична библиотека, докато DirectX е група от библиотеки, като Direct3D е тази, която служи за триизмерната графика. И двете библиотеки позволяват да се използват възможностите на видео картата за по-бързо и качествено изрисуване.

1.1.2 Преносимост

DirectX е насочено към Windows операционните системи и XBox докато разработчиците OpenGL се целят да постигнат мултиплатформеност. Въпреки че OpenGL е съвместимо с повече операционни системи (като Windows, Linux, Mac OS) и други, в платформите за игри рядко се използва. Единствено на PlayStation 3 може да се използва OpenGL, но не предлага добра производителност.

1.1.3 Сложност

Ранните версии на Direct3D са били изключително сложни за използване, защото дори и прости операции изисквали създаването и поддържането на обекти, наречени „execute buffers”. За сравнение, в OpenGL, в повечето случаи, е било достатъчно да се извика една функция. Direct3D модела е разгневил много програмисти. До излизането на версия 9 той е бил подлаган на постоянна критика от разработчиците за неговата сложност. [1]

Въпреки солидните критики, Microsoft продължили работа върху DirectX и много от хората, които първоначално са критикували това API са се съгласили, че Direct3D е също толкова добра, ако не и по-добра библиотека от OpenGL. [2]

1.1.4 Структура и функционалност на двете API: OpenGL и DirectX

Direct3De реализиран на базата на COM технологията. Това подсигурява съвместимостта на код написан за по-стара версия на библиотеката с код написан за по-нова. Това че Direct3D е базиран на COM означава, че той може да се използва във който и да е поддържащ тази технология език, например (Microsoft Visual C++, Delphi и .NET езиците).

OpenGL е базиран на езика C. Библиотеката е разработена около идеята за Стейт машината, като по-новите версии на това API имат по-обектно базирана система. Въпреки че, OpenGL е написан на C, той може да бъде използван с всеки един програмен език.

Direct3De създаден като абстрактен слой над графичния хардуер, за да няма нужда програмиста да се нагажда към него. От друга страна, OpenGL е проектиран да бъде 3D хардуерно-ускорена рендерираща система, която може да бъде емулирана софтуерно

Тези две API са проектирани по два напълно различни начина. Поради това има разлики в начина, по който двете библиотеки работят. Direct3D очаква от приложението да управлява хардуерните ресурси; OpenGL изпълнява тази задача, но това усложнява създаването на имплементация или драйвер, който е с добра производителност. С DirectX, на разработчика се налага да управлява хардуерните ресурси, но има възможността да използва ресурсите по най-ефективния начин.

Като изключим няколко незначителни функционални разлики, двете библиотеки предлагат почти една и съща функционалност. Производителите на хардуер реагират бързо на промените в DirectX, докато новите функции в OpenGL са създадени първо в хардуерните устройства и след това са добавени към стандарта.

1.1.5 Производителност

В производителността на двете библиотеки има голяма разлика, дължаща се в разликата в структурата на хардуерните драйвери. Под DirectX, IHV драйверите са на ниво ядро и са инсталирани в операционната система. Потребителската част на библиотеката се обработват от DirectX runtime, осигурено от Microsoft. Под OpenGL IHV драйвера е разделен на две части - потребителска част която имплементира API то и системна част, която се ползва от потребителската. Това е недостатък, защото ползвайки системни операции от потребителското ниво изисква системно извикване(например да се превключи

процесора в защитен режим). Това е бавна операция, отнемаща микросекунди да завърши. През това време, процесора не може да изпълнява никакви изчисления. Поради това е необходимо да се минимизира броя системни извиквания. Например, ако командния буфер е пълен с данни за рендериране, съответното API може да запази извикванията за рендериране в временен буфер и когато командния буфер е почти празен превключва в защитен режим и изпълнява всичките извиквания на веднъж.

Тъй като Direct3D IHV драйверите са на системно ниво и кода на потребителското ниво е извън тях, е невъзможно да се извърши такава оптимизация. Заради Direct3D runtime, потребителската част която имплементира библиотеката не може да знае за вътрешната организация на драйвера, тя не може ефективно да използва гореспоменатата техника. Това означава че всяко D3D извикване което изпраща команди към хардуера трябва да направи превключване в привилегирован режим. Това е довело до нуждата да се подават голями масиви от данни в едно системно извикване.[5]

Direct3D 10 позволява част от драйверите да вървят в потребителски режим, това позволяват IHV да използва техниките на OpenGL да се намалят системните извиквания.

1.1.6 Област на приложение

OpenGL винаги се е използвал повече за професионалния графични приложения, докато DirectX се използва в електронните игри. (Понятието *професионален* в случая се отнася към приложение за 3D графика, в анимационни филми и научни визуализации).

Причината OpenGL да завземе професионалния пазар е частично историческа. Много професионални графични приложения (на пример, Softimage|3D, Alias PowerAnimator) са били написани на IRIS GL и след това са преминали към OpenGL.

Многото допълнителни функции на OpenGL, които не са приложими в разработката на игри, са приложими в професионалните графични приложения.

Другата причина OpenGL да се използва в такъв тип програми е маркетинговата стратегия и дизайна на библиотеката. DirectX е API проектирано за високо производителен достъп до хардуера на ниско ниво за целите на създаването на игри. OpenGL е с доста по-общо предназначение, за това то предлага функционалност, която не е необходима за разработчиците на игри.

В момента електронни игри се играят предимно на конзоли или на PC под Windows. Единствената конзола, която поддържа OpenGL е PlayStation 3, но това API предлага много по-ниска производителност сравнение с стандартното API за PS3. Конзоли като Xbox и Dreamcaster използват DirectX като стандартно API. Въпреки че, DirectX е съвместимо с по-малко платформи от OpenGL, платформите за игри ползващи DirectX са повече от тези ползващи OpenGL.

Другата причина DirectX да държи пазара на игрите е, че DirectX е проектиран специално за разработка на игри, докато OpenGL е с по-общо предназначение.

1.2 Езици за шейдъри

Шейдърите са малки програми, написани на език на високо ниво, които се изпълняват на видеокартата. С тях програмиста пише код, който се изпълнява в някой от етапите на т. нар. Rendering pipeline. Това е поредица от процеси, които имат за цел да създадат двумерно изображение от тримерните обекти в сцената. Има няколко езика за писане на шейдъри: HLSL, GLSL и CG. Всичките те са базирани на езика C и споделят подобен синтаксис. Част от функционалността на езика C е променена и са добавени нови типове данни, за да бъдат подходящи за програмиране на GPU(graphics processing unit). Те са специализирани езици от високо ниво и са предназначени единствено за програмиране на видеоконтролери.

HLSL означава High Level Shader Language или High Level Shading Language . Този език е разработен от Microsoft и може да се използва само в Direct3D приложения. Той е изключително подобен на Nvidia CG езика, тъй като е бил разработван заедно с него. HLSL е започнал разработката си с DirectX8, за да се програмира Rendering Pipeline. В DirectX 8, тя е била програмирана с комбинация от инструкции на асемблер и HLSL инструкции. С излизането на Direct3D 10 API, Rendering Pipeline вече е напълно програмируем с HLSL и асемблер повече не се ползва.

GLSL е езика, който се използва в OpenGL. GLSL означава openGL Shader Language и е разработен от OpenGL ARB. Той може да се използва само в OpenGL базирани приложения. Недостатък пред HLSL е липсата на възможност за многопасово рендиране, което присъства в HLSL и CG.

Cg или C for Graphics също е език за шейдъри на високо ниво. Той е разработен от Nvidia

със сътрудничество с Microsoft. За разлика от GLSL и HLSL, Cg не е зависим от конкретно API и може да се използва както с OpenGL така и с DirectX.

Последната версия на шейдърите е версия 5. Тя е разширение на версия 4 и предлага подобни функции. За разлика от предходните, версия 5 има възможност за обектно ориентирано програмиране.[3] Друга възможност е динамичното избиране на това кой шейдър да се ползва. Графичните разработчици, понякога създават огромни шейдъри с общо предназначение. Те могат да бъдат ползвани с голям брой елементи на сцената. По време на изпълнение шейдъра изпълнява код, за да установи съответната ситуация. За нещастие, тези големи шейдъри неефикасно използват регистрите с общо предназначение и могат да бъдат много по-бавни от по- специализираните шейдъри. Шейдър моделът 5 предлага решение на този проблем, като предоставя динамично свързване на шейдърите. Динамичното свързване разделя фрагментите код на шейдъра, използвайки интерфейси и виртуални функции и позволява приложението да избере фрагмента, който ще се ползва по време на изчертаване. Това повишава производителността, като използва само необходимия шейдър код, а не целия голям шейдър с общо предназначение. [4]

1.2.1 Видове шейдъри

Шейдърите се делят на няколко вида:

- Vertex Shader – този тип шейдър се изпълнява по веднъж за всеки вертекс, който е подаден на графичния процесор. Целта му е да се трансформират тримерните координати на вертексите в двумерни координати, за да могат да се изобразят на екрана и да се запази информация за дълбочината в Z-буфера. Вертекс шейдърите могат да манипулират вертексите, като им променят позицията, цвета, текстурните координати, нормалата, но не могат да създават или премахват вертекси. Изхода от Vertex шейдъра отива като вход на следващия етап в rendering pipeline.
- Geometry shader- този шейдър може да създава и премахва геометрия към тримерните модели. Geometry shader не е задължителен. В версия 5 на шейдърите той предлага инстанциране, което увеличава производителността, когато останалите примитиви модела нямат значение. Друга възможност на шейдъра са множеството изходни потоци. Така можем да предадем точки, към следващия стадии в rendering

pieline, в няколко потока.[4]

- PixelShader или Fragment shader – служи за изчисляването на цвят на отделните пиксели. Входа на този шейдър идва от растеризатора, който попълва полигоните, които са били изпратени през graphics pipeline. Pixel Shader обикновено се използват за ефекти свързани с осветлението. Такива са попикселово осветление, bump mapping и други. При пиксел шейдърите няма едно към едно връзка между броя извиквания на пиксел шейдърите и броя на пикселите на екрана.
- Compute Shader – само в Direct3D 11. Compute Shader разширява възможностите на Direct3D11 отвъд графичното програмиране. Тази технология е позната още като DirectCompute технология. Както и останалите програмируеми шейдъри(vertex и geometry shader на пример), compute shader е проектиран и имплементиран с HLSL, но това е мястото, където общите неща свършват. Compute Shader предоставя възможност висикоскоростни изчисления с общи предназначение и се възползва от големия брой паралелни процесори на GPU(graphical process unit).Compute shader осигурява споделяне на памет и синхронизация на нишки за да позволи по-ефективните паралелни програмни методи.

1.3 Физични Енджини

1.3.1 Причина да се ползва физичен енжин

Освен графиката в игрите е важно и усещането за динамика. Игрите, които не притежават реалистична физика, създават неприятно, изкуствено чувство по време на игра. Колкото и реалистично да изглежда графиката, ако телата не се движат по начин подобен на тези в реалния свят, всичко изглежда неестествено. Това налага използването на физичен енджин.

1.3.2 Видове физични енджини

Физичните енджини се делят на два типа:

- Енджини в реално време
- Енджини с голяма прецизност

Енджините с голяма прецизност са приложими в симулации, където е важно да се изчислят

данни с голяма точност. Те използват прекалено много ресурси и нямат приложение в направата на игри. В игрите е важно, грешката в изчисленията да е достатъчно малка, да не може да се хване на око. При тяхната разработка има едно правило - „Ако изглежда правилно, значи е правилно“. За това в тях се използват Енджини в реално време, тъй като те са достатъчно леки и бързи, за да може да се играе играта на нормален персонален компютър.

1.3.3 Работа с физични енджини

С повечето физични енджини се работи по сравнително един и същ начин. Тялото се състои от два обекта. Единия е високо детайлен и служи за рендиране, а за физиката се създава доста по опростен модел, за да не се използват много ресурси за изчисление. Обектите който отговаря за физиката се добавят в обект, който представлява физическия свят в приложението. Там се засичат сблъсъци, прилагат се различни сили и така нататък. При рендирането се проверяват свойствата на обекта като местоположение, ротация и така нататък. Обекта, който се рендира, се транслира и завърта спрямо тях.

1.3.4 Избор на физичен енжин

Трябва да се вземат в предвид няколко критерия, когато избираме физичен енджин. Това са прецизността, производителността и това колко добре е документиран. Енджини като Bullet physics са с доста добра производителност и прецизност, но са доста зле документирани. Други енджини като ODE са с по-лоша производителност, но имат доста солидна документация. Документацията на даден продукт в случая е от ключово значение, когато даден програмист почне да се занимава с него. Липсата на документация може да забави доста завършването на проекта, в който се използва.. Друг критерии при избора на физичен енжин са неговите перспективи в бъдеще. Добре е да се вземе в предвид това, защото след време може да се наложи да се премине към другата технология, а това би коствало допълнителни средства.

1.4 Среди за разработка на C++ приложения

1.4.1 Visual C++

За разработка на C++ приложения под windows има голям брой среди. Една от тях е Visual Sdudio на Microsoft. Това е най-популярната среда за разработка на C++ приложения под Windows. Microsoft са изминали дълъг път свързан с поддръжката на C++ стандарта от Visual c++ 6.0. От версиите Visual Studio 2005/Visual C++ 2005 Express Edition стандарта се поддържа толкова добре, колкото и всеки друг популярен C++ компилатор. Със всяка версия Microsoft пуска различни издания в зависимост от нуждите на съответния програмист или фирма. За версия 2010 те са:

- Ultimate – 11,899$
- Premium – 5,469$
- Professional - 549$
- Express – безплатен

Всяко издание се различава по своята цена и възможности. Visual Studio за момента е най-използваната среда за разработка на C++ под Windows в софтуерните компании.

1.4.2 Dev C++

Друга популярна среда за разработка на C++ приложения е Dev-C++. Тя има голяма популярност сред учениците и е изключително слабо популярна сред софтуерните компании. Това, може би, се дължи на факта, че Dev C++ е безплатно IDE, но и не предлага големи възможности. Средата използва MinGW порт на gcc като компилатор. Популярността Dev C++ намалява, зашото неговата поддръжка е спряла. Недостатък е, че не поддържа auto completion и не подчертава грешките в процеса на въвеждане на код. Това може доста да удължи времето за написване на съответната програма. Чрез DevC++ може да се ползва VCS директно от средата. Единствения проблем е, че има поддръжка само за CVS.

1.4.2 MinGW studio

MinGW studio е среда подобна на DevC++. Тя също използва MinGW gcc компилатора и има добра интеграция с wxWidgets SDK. Както DevC++, така и MinGW studio поддържат сравнително стари версии на g++, което кара потребители да компилират g++ от сорс кода, ако те искат да използват по-актуална негова версия. Недостатък на MinGW studio е, че разработката му е спряла от 2005 година.

1.4.3 Eclipse for C++

Eclipse for C++ with CDT е open source IDE. Тази среда изисква инсталиран g++ компилатор. Eclipse for C++ бързо набира популярност и има шанс да стане един от водещите среди за разработка на C++ приложения. Eclipse е мултиплатформена среда и това е голям плюс спрямо Visual Studio. Освен това Eclipse разполага със солидна общност и потребителите бързо могат да намерят отговор на техните въпроси. Недостатък е, че понякога има проблеми с инсталацията при отделни хора. Eclipse for c++ е бързо развиващ се програмен продукт и откритите проблеми се отстраняват своевременно.

1.4.4 Code::Block

Code::Block е олекотена среда за разработка на C++ приложения. Тя зарежда и върви бързо дори и на по слаби машини. Тя също използва MinGW като компилатор. Code::Block е open source проект, който се разработва активно. Средата е мултиплатформена и може да се стартира на Windows, Linux и Mac

1.5 Системи за контрол на версиите

1.5.1 Примитивни методи

Често в разработката на един проект е полезно да могат да се създават отделни негови версии и когато възникне някакъв проблем да възможно проекта да се върне към предишната версия. Също така, когато един проект се разработва от повече от един програмист, е необходимо да има система, която да синхронизира тяхната работа. Първоначално са се правили ръчни копия на проекта в различните му етапи. Програмистът добавя коментари в сорс файловете свързани с историческото им развитие.

Синхронизирането на работа та става чрез вербални или e-mail споразумения.

Скритата цена на този метод е висока, особено когато(както често се случва) се повреди. Тази процедура отнема време и концентрация. Има голям шанс за грешка, особено когато програмиста е под напрежение или проекта не е в добро състояние, а точно тогава най-много се нуждаем от стабилната работа на такава система.

1.5.2 Автоматизирани системи за контрол на версиите

С времето се установило, че горния метод е неподходящ и неудобен, за това са създадени автоматизирани системи за контрол на версиите. Те пазят всички версии на файловете по-удобен за обработка начин. Поддържат разклонения, така няколко програмиста могат да работят с общи файлове и накрая да обединят работата си. Поддържат и средства за сливане на код от отделни разклонения.

1.5.3 Централизирани и децентрализирани системи за контрол на версиите

Автоматизираните системи за контрол на версиите се делят на два типа:

- централизирани
- децентрализирани

При централизираните има едно централно хранилище, където се пазят всички версии на файловете. Програмистите работят в/у checkout, който представлява състоянието на проекта в текущия момент. Правото да се пише в хранилището имат само ограничен брой добре познати програмисти, на които може да се има пълно доверие. Такива системи са CVS и SVN.

При децентрализираните системи има множество локални хранилища. Основни операции при тях са push и pull. Чрез pull всеки програмист, които работи по съответния проект, копира хранилището при себе си. Така се елиминира проблема с правото да се пише в хранилището, защото всеки разработчик притежава локално копие и може да пише всеки път когато пожелае в него. Когато приключи работа, програмиста може да качи своята работа в друго хранилище, посредством операцията push. При децентрализираните VCS,

най-често се обявява едно хранилище, което е главно за съответния проект и само определени програмисти имат право да изпълняват операцията push в/у него. Даже в някои случай, тази операция е напълно забранена и когато трябва да се качи код се изпълнява pull от главното хранилище. Пример за децентрализирана с-ма е Mecurial.

ВТОРА ГЛАВА

Изисквания към фреймуърка, използвани език и софтуерни средства

2.1 Изисквания към фреймуърка

В разработката на триизмерните игри винаги се налага да има възможност да се използват 3D модели, създадени от програми за моделиране, да се рендират сцени по удобен начин и други. Освен това игрите от определен жанр имат голямо сходство в разработката.

В космическите игри винаги присъстват елементи, като космически кораби, станции, планети и др. За това е необходимо да има готова база за създаването на такива обекти. Нужно е тези обекти да могат да се добавят по удобен начин в играта.

В почти всички жанрове игри е необходимо да могат да се бият и убиват различни участници в играта(били те хора или обекти с изкуствен интелект). За тази цел е необходимо в фреймуърка да има бойна с-ма.

При създаването на такава игра е хубаво физиката да е съобразена със средата, в която ще се развива действието. При игрите, които не се развиват в открития космос, гравитацията се описва като един вектор, който е с еднаква големина и посока за всяко тяло, независимо от неговото положение. В игрите, които се развиват в космоса, гравитацията трябва да зависи от положението на обектите. Тя трябва да сочи в определена точка и да зависи от разстоянието на обекта спрямо тялото, което създава тази гравитация. В фреймуърка ще се задава гравитация от втория тип.

Поради липса на съпротивление в космоса, човек може да контролира движението на телата само чрез реактивни двигатели разположени на определени места на тялото, което да бъде придвижвано и в зависимост от това кой двигател работи и с каква сила тялото може да се придвижва и да маневрира. Заради това е нужно програмиста, който разработва такава игра лесно да може да създава тела които да използват точно такъв начин на придвижване. В фреймурка ще има класове, които ще съдържат информация за местоположението и вида на двигателите.

Получаването на входни данни от периферните устройства в една игра, се различава много от същия процес в едно стандартно приложение. При него обработката на входните данни е

съобразена с процеси като въвеждане на текст и използване на различни графични компоненти чрез мишката. Когато се налага да се получава вход от клавиатурата чрез стандартните съобщения от операционната с-ма, при задържане на бутон се генерира първо едно съобщение и след определена пауза започват да се генерира последователност от такива съобщения. Това е удобно, когато въвеждаме текст, защото се избягва нежелано дублиране на символи, но в игра това би попречило. Ако използваме стандартните съобщения за вход от клавиатура, за да придвижваме обект, например космически кораб, той ще се придвижи леко на пред, ще спре и чак след известно време ще продължи да се движи нормално. Често в графичната среда на операционната с-ма се прилагат алгоритми променящи движението на курсора с цел, по-лесното уцелване на различни графични компоненти. В средата на една игра, обаче, тези алгоритми биха попречили на нормалната игра поради нелинейната зависимост между движението на мишката и изместването на курсора/камерата. За да се избегнат тези нежелани ефекти е необходимо в фреймуърка да има система за получаване на вход от периферните устройства на по-ниско ниво.

В създаването на геймплея на всяка игра е нужно да се се следи за събития като убит враг, играч стигнал някъде, играч взел нещо и т. н. Това е добре да се реализира със събития които да се прихващат и да се изпълнява определено действие, когато събитието настъпи. В фреймурка ще има такава система, която ще ще прихваща такива събития и ще вика методи да ги обработят.

В зората на библиотеките за 3D графика, така наречения Graphics pipeline е бил фиксиран. Програмиста не е имал достъп до него, но за графиката по онова време това не е било проблем. В последствие програмистите започнали да се стремят към по-реалистична и ефектна графика. Тогава фиксирания Rendering pipeline се оказал проблем. Програмистите желаели да добавят свой код в различните етапи на рендерирането. Първоначално това ставало чрез писане на код на асемблер за конкретния видеоконтролер. Това не е било особено удобно, защото както всеки асемблерен език и този за видеокартите е специфичен за всяка марка и всеки модел. За да се справят с това, организациите занимаващи се с 3D библиотеки разработили шейдърите. Това са малки програми, написани на език на високо ниво, които се изпълняват в/у видеокартата. Те набрали голяма популярност пред фиксирания Graphics pipeline и от версия 10 на DirectX това е единствения начин да се рендира 3D графика. За това се налага един фреймуърк да разполага със система за

зареждане на ефекти. Въпреки широката употреба на шейдърите, хора, които от скоро се занимават с 3D програмиране, често не са запознати с тях. Други програмисти може да желаят да отложат писането на своите шейдъри за по късен етап от разработката на приложението. За това се налага един фреймуърк да разполага с готови ефекти, които да ползват, но и лесно да могат да добавят други.

Всяка програма се нуждае от интерфейс, през който потребителя може да я управлява и следи нейното състояние. В игрите не могат да се използват стандартните средства за създаване на графичен интерфейс, защото външния му вид трябва да бъде съобразен с изгледа на играта. За това фреймуърка трябва да разполага със средства за създаване на графичен интерфейс за менютата и за самата игра.

2.1 Използван език и софтуерни средства

За реализацията на тази дипломната работа е използван езика C++. Една от причините да бъде избран е, че в игрите бързодействието е от голямо значение. Освен това, той е основния език, който се използва за разработка на игри и могат да се намерят много на брой изпитани софтуерни средства.

За среда за разработка се използва Visual C++ Express Edition 2010. Това е безплатната версия на Visual Studio 2010. Създадена е за учебни цели. Предпочетена е тя, тъй като Visual Studio е най-разпространената в софтуерните фирми среда за разработка на C++ приложения под Windows. Избрах Express Edition, защото е безплатна и предлага всички необходими средства за реализирането на дипломния проект.

За графиката се използва DirectX 10. Това е COM базирана библиотека на Microsoft. Предпочетена е пред OpenGL, защото DirectX е създадена специално за разработката на игри и е библиотеката, която се използва масово от почти всички студия за игри. Използвам версията 10 поради наличието на Geometry Shader. С не го може да се създава и унищожава геометрия което позволява направата на различни интересни ефекти като таселация (увеличаване на фасетите в дадено изображение чрез криви на Безие).

За писането на шейдъри се използва HLSL. Това е стандартната технология за създаване на ефекти, когато се използва DirectX. Предпочетох нея пред Cg, защото съм по-запознат с HLSL и има повече информация и учебни материали за нея. HLSL и Cg предлагат абсолютно еднаква функционалност, така че нямаше причина да бъде избирана Cg.

За физиката в фреймуърка се използва Bullet physics. Това е open source енджин. Предпочетох него, защото изчислява физиката с доста голяма точност. Освен това има по-добра производителност от някои от конкурентите му, например ODE. Другата причина да бъде избран е, че се развива доста бързо и се очаква да завзема все по-голяма част от пазара.

За да поддържката на кода се използва децентрализираната система за контрол на версиите Mercurial. Въпреки че, за момента има само един човек, който работи в/у този проект, тази система помага предоставяйки възможността, когато проекта се “счупи“ да се върне в момент когато е наред. Избрах Mecrurial, защото е децентрализирана и може да има локално хранилище, където да се пазят различните версии. Използва се и хранилище в Bitbucket, с цел да има резервно копие ако локалното хранилище бъде загубено.

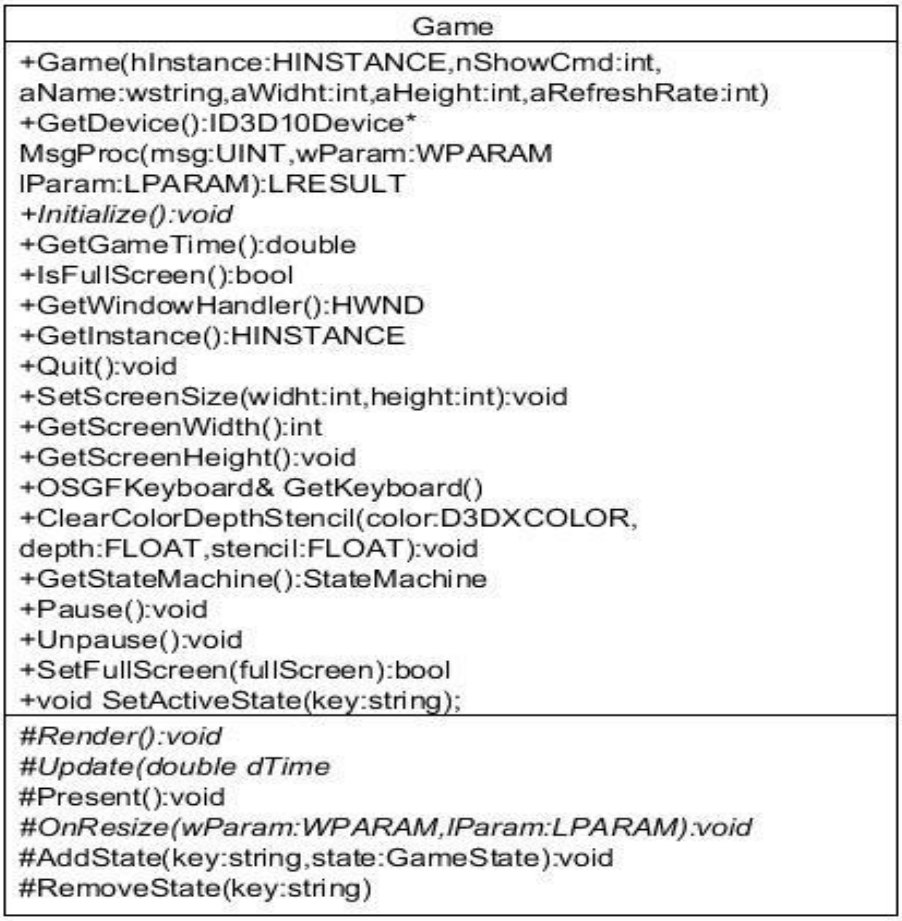
ТРЕТА ГЛАВА

Реализация на фреймуърка

3.1 Класът Game

Базовият клас за играта е класа Game. Той отговаря за създаването на прозорец за играта, инициализацията на DirectX, обработка на съобщенията, създава се главният цикъл, вика методите за обновяване на обектите в играта и тяхното рендиране. Това е абстрактен клас и играта, която бива създавана с този фреймуърк трябва да наследи този клас и предефинира част от методите му.

Всеки метод, който бива предефиниран, трябва да вика съответния метод на базовия клас. Изключение правят случаите, когато метода на базовия клас е абсолютно виртуален.



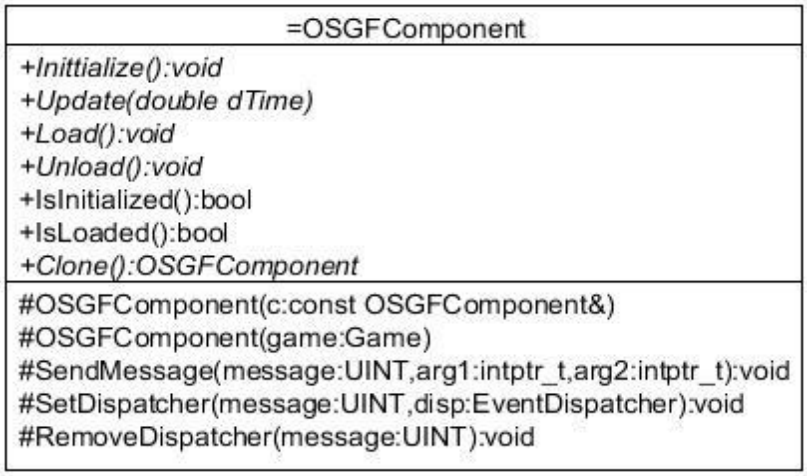
Фиг 3.1 UML диаграма на класа Game

Както се вижда от фигура 3.1 класа Game има конструктор с 6 аргумента. Първите 5 аргумента служат за да се получи необходимата информация за създаването на прозорец в който да се рендира играта. hInstance е хандлър на процеса, nShowCmd служи да определи какъв ще е прозореца, който ще се създаде (максимизиран, минимизиран, скрит), aName е името на класа на прозореца, aWidht и aHeight са размерите, а aRefreshRate определя с каква честота ще се обновява съдържанието на прозореца. Initialize() създава прозореца, инициализира DirectX, както и някой други части на играта. Самата игра се стартира от метода Run. Това отваря прозореца и играта влиза в главния цикъл. В него се следи за съобщения, предават се да се обработят от MsgProc, всеки кадър се обновява съдържанието на играта и в зависимост от опресняващата честота се обновява и съдържанието на екрана. Времето в играта се следи от класа GameTimer. От него може да се получи информация за изминалото време от стартирането на играта и за изминалото време от миналия кадър. Този

таймер се спира когато играта е спряна или минимизирана.

3.2 Класът OSGFComponent

Повечето класове в фреймурка са наследници на OSGFComponent. Това е базовия клас на повечето класове в проекта.



Фиг 3.2 UML диаграма на класа OSGFComponent

Конструктора на класа приема референция на обекта от тип Game, в който компонента участва. Така може всеки компонент в играта да получава необходимата информация от играта като размер на екрана и др.

Методът Initialize() служи за инициализация на обекта. Инициализацията се извършва в този метод а не в конструктора, защото понякога е удобно да създадем обект, който да се използва само в определена сценарии и не искаме да е заел значителни ресурси без да сме го използвали.

Методът Update() се вика при всяка итерация на главния цикъл. Аргумента на функцията сърдържа времето което е изминало преди предишното извиквне на този метод.

Методите Load() и Unload() служат за заделяне и освобождаване на ресурси когато даден обект влиза или излиза от употреба. Например елементите в менюто не е необходимо да заемат други ресурси, докато играча е извън него.

Методът Clone() е виртуален метод който връща копие на обекта. Наличието на този метод

се налага, защото C++ не разполага с виртуални конструктори и може да ни се наложи да копираме обект, който се съхранява в полиморфна променлива.

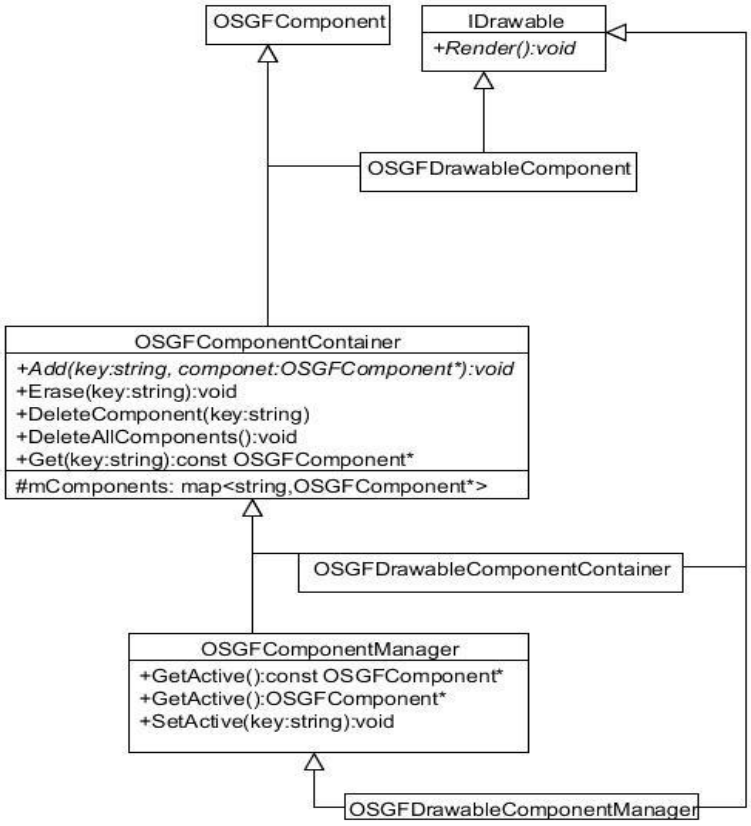
Методите SendMessage, SetDispatcher и RemoveDispatcher са свързани с системата за събития и ще бъдат обяснени по на долу в документацията.

3.3 Класът OSGFDrawableComponent

Класът OSGFComponent се използва, когато искаме да създадем клас за обект в играта, който не е необходимо да се визуализира(например камера).В случаите когато искаме да създаваме обекти, които да бъдат визуализирани използваме клас наследник на OSGFDrawableComponent. OSGFDrawableComponent е наследник на OSGFComponent и IDrawable. IDrawable е клас съдържащ само един абсолютно виртуален метод Render(). Причината поради която съм създал по такъв начин класа е избягването на т. нар Diamond Problem по-надолу в йерархията.

3.4 Класовете OSGFComponentContainer, OSGFDrawableComponentContainer, OSGFComponentManager и OSGFDrawableComponentManager.

За по-удобната работа с компонентите в играта са създадени два класа контейнери и два класа мениджъри. Класовете OSGFComponentContainer OSGFDrawableComponentContainer служат за да се съхраняват компоненти на едно място и да се изпълняват общи операции в/у тях с извикване на съответния метод на контейнера. Класовете мениджъри (OSGFComponentManager и OSGFDrawableComponentManager) служат когато искаме да пазим няколко компонента, но да работим само с един от тях.



Фиг. 3.3 UML диаграма на контейнери и мениджъри и техните базови класове.

В класовете контейнери и мениджъри се пази указател към съответните компоненти. Тъй като е възможно, когато се изтрива контейнерът и ли мениджърът, програмиста да не иска да изтрива компонентите които са в него това не става в деструктора а в отделна функция DeleteAll().

3.5 Реализацията на State Pattern

Класът StateMachine наследява OSGFDrawableComponentManager. Той служи за реализацията на т.нар State Pattern. При него се избягва писането на множество условни оператори за да се проверява в какво състояние е приложението. Всяко състояние е обект от класа GameState. Тези състояния се пазят в класа StateMachine и през него се викат методите за обновяване и рендиране на текущото състояние. Този подход увеличава модулността на приложението и лесно могат да се добавят нови режими на игра (например

главна игра или меню). Лесно се добавят и менюта и подменюта като се добави ново състояние. Идеята е, че играта не знае в какво състояние е и не се занимава с превключването им. State machine се грижи за това. Всяко състояние знае в кое друго може да премине. Играта има едно текущо състояние и се грижи за него. Рендира го, обновява го, грижи се за събитията му и т.н. В един момент състоянието се сменя. Например, менюто е получило за вход New Game и играта започва да обслужва ново състояние без да е разбрала за смяната. Единствено State machine знае за това. При всяка смяна на състояние се вика метода Load на новото състояние и метода Unload на старото. В клас StateMachine, освен наследените методи на OSGFDrawableComponentManager има метода *OnScreenResize(WPARAM wParam, LPARAM lParam).* Този метод се вика от играта всеки път когато се промени размера на екрана. Аргументите на функцията са аргументите на съобщението WM_RESIZE. Наличието на този метод се налага, защото за някои компоненти е от ключово значение дали екрана е променил размера си. Един от тях е камерата, която трябва да преизчисли отношението на височината на екрана към неговата широчина. Смяната на състоянията става с метода void SetActive(cosnt string& key). Всяко състояние се добавя с някакъв ключ от тип string и в последствие се процедира чрез него.[6]

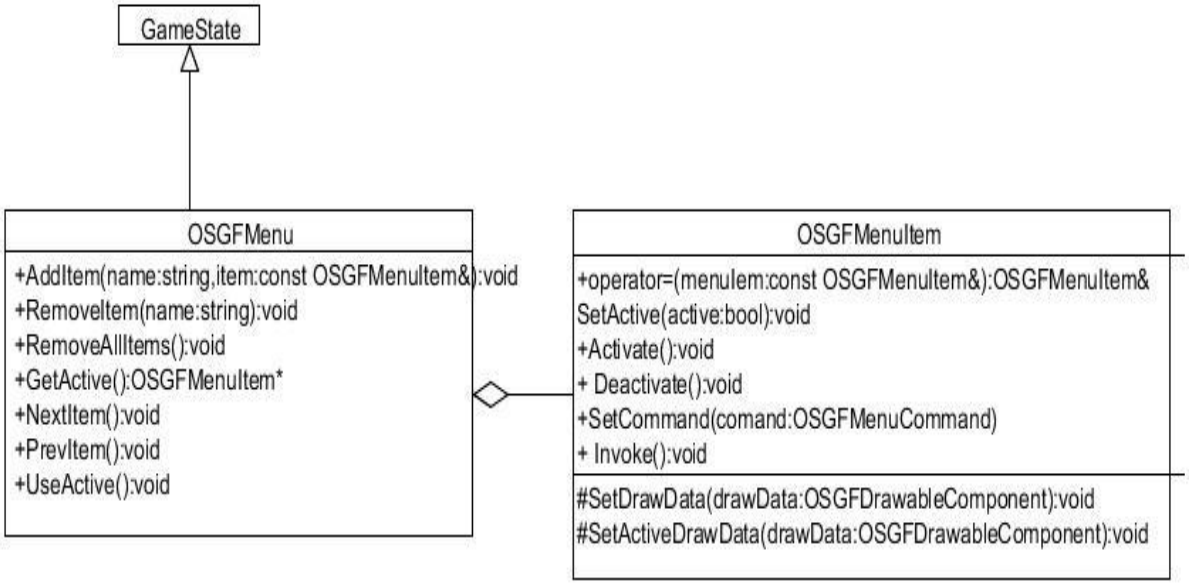
3.6 Класът OSGF2DDrawableComponent

Дори и в триизмерните игри се налага да има двумерни обекти. Всяка игра се нуждае от някакъв интерфейс. За 2D графиката е създаден класът OSGF2DDrawableComponent. Той наследява OSGF2DDrawableComponent и с него може да се зададе правоъгълник в който да се рендира двумерното изображение. Правоъгълника е обект от типа Rectangle и притежава възможност за проверка дали дадена точка е в него. Това става чрез метода *bool IsPointInside(LONG x,LONG y).* Наличието на тази функция улеснява създаването на потребителски интерфейс.

Един от неговите наследници е OSGFScreenText. Този клас се използва, когато трябва да се изпише някакъв текст на екрана. Освен наследените методи от OSGF2DDrawableComponent OSGFScreenText има методи SetColor,за задаване на цвета на текста, SetText за задаване на самия текст, и SetFontDesc, който получва като аргумент структурата `D3DX10_FONT_DESC.` Тя е дефинирана в DirectX 10 API и служи за описване

на шрифта, с който ще бъде изписван текстта.

3.6 Реализация на класа OSGFMenu



Фиг 3.4 UML диаграма на OSGFMenu и OSGFMenuItem

В фреймурка има класове за създаването на графичен интерфейс. Един от тях е OSGFMenu. Както се вижда от фигура 3.4, това е наследник на GameState. Този клас е базов за всички менюта и подменюта в играта. Той съдържа в себе си обекти от типа OSGFMenuItem. Те са наследници на OSGF2DDrawableComponent. Те имат две състояния за рендиране. Когато са маркирани и когато не са. Промяната на състоянието става от менюто чрез методите Activate() и Deactivate(). В менюто може да има само един селектиран елемент, който при натискане на бутон(например Enter) се изпълнява операцията за която е поставен елемента. Класът OSGFMenu е организиран като изображение(map).Елементите се добавят чрез AddItem() и се Махат чрез RemoveItem(). Могат да се премахнат всички елементи на веднъж чрез RemoveAllItems. Маркирания елемент се взима чрез GetActive(). Методите NextItem и PrevItem са удобни когато искаме менюто да може да бъде обхождано със клавишите. Те сменят елемента със следващия поред. Use Active се използва, когато човека ползващ менюто се е спрял на някой от

елементите му и иска да го използва. В менюто се поддържа и курсор, с който може да се взаимодейства с елементите в него. При всяко викане на Update се проверява каква е позицията на курсора и посредством *IsPointInside* на OSGF2DDrawableComponent се проверява дали е върху някой елемент. Ако е този елемент се маркира и при натискане на левия бутон се активира. Фиг 3.4 UML диаграма на класовете OSGFMenu и OSGFMenuItem

Класа OSGFMenu обработва вход от клавиатурата и мишката. Той предефинира метода Render, като изчертава курсора на мишката. Той има метод SetCursor, който приема като аргумент указател към OSGF2DDrawableComponent. Това е компонента, който се изчертава като курсор. Както се вижда от фигура 3.4, в менюто има обекти от типа OSGFMenuItem. Този тип се използва за да се представят елементите в менюто. Този клас се наследява от потребителя в зависимост от това какъв елемент иска да създаде. С защитените методи SetDrawData и SetActiveDrawData се задава OSGFDrawableComponent обекти, които да се рисуват, когато елемента е маркиран и когато не е. Дали е един елемент е активиран или е деактивиран се разбира чрез викането на методите Activate и Deactivate от менюто. С метода SetCommand() се предава указател към някой от методите на класа за менюто. Когато се извика метода Invoke() OSGFMenuItem вика метода, към който сочи указателя.

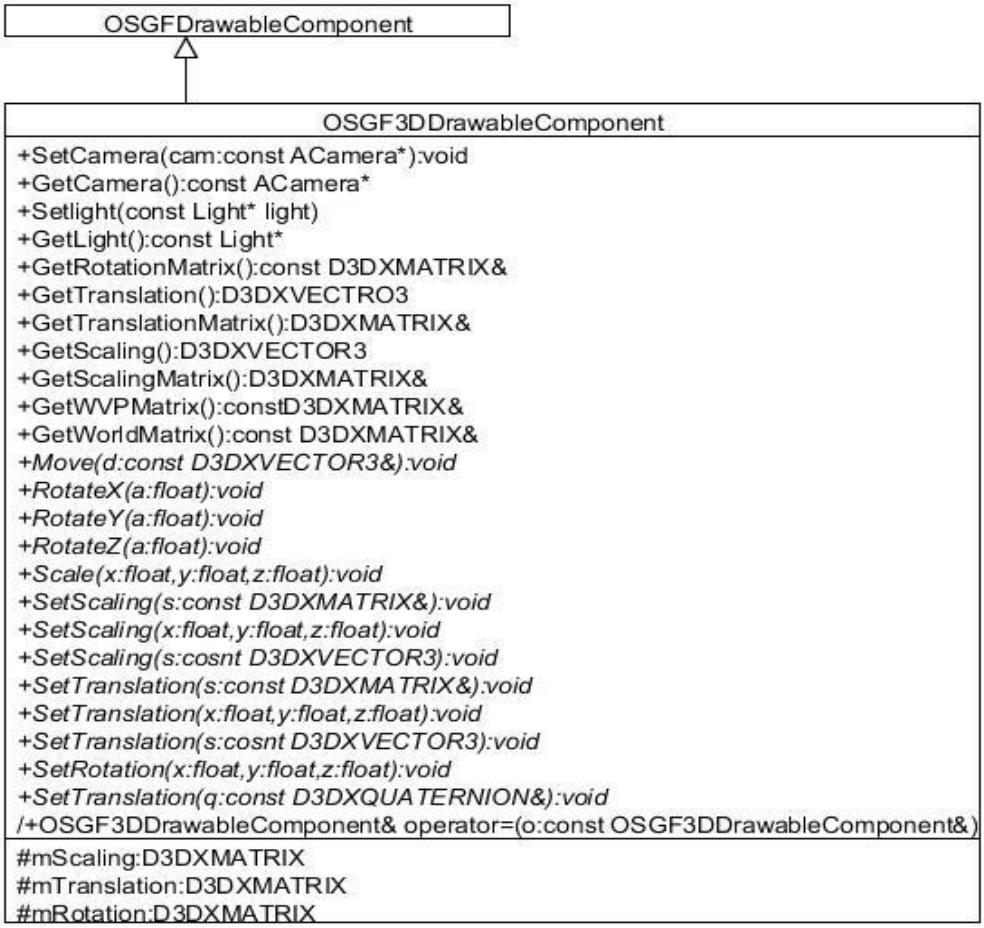
OSGFMenu има вградена логика за вход от мишка. Различните елементи в менюто могат да се избират като се постави курсора в/у тях. Класът предоставя възможност да се рисуват всякакви курсори. Те трябва да са обекти от клас наследник на OSGF2DDrawableComponent и се добавят чрез функцията *void SetCursor(OSGF2DDrawableComponent* cursor).* Ако не се извика тази функция обаче, няма да бъде визуализиран курсор.

За добавянето на текст в менюто има клас MenuText. Той е наследник на OSGFMenuItem и може да се използва за направата на бутони. В себе си съдържа два обекта от тип OSGFScreenText. Единият се вижда ако MenuText е селектиран, а другият – ако не е.

3.7 Реализация на класа OSGF3DDrawableComponent

За 3D графика е създаден класа OSGF3DDrawableComponent. Той наследява OSGFDrawableComponent. като добавя функционалност за трансформиране на обекта в

тримерното пространство. Тази трансформация е необходима, защото 3D обекта трябва да мине през няколко пространства докато може да се изобрази на екрана. Първото е локалното пространство. Там обекта е със размери и ориентация, както е създаден от програмата за моделиране. За да стои адекватно в сцената на играта трябва да се оразмери завърти и премести. Така обектът преминава в глобалното пространство и е с подходящия размер и положение в сцената която изобразяваме. След това е необходимо да се трансформира спрямо камерата и да се изчисли перспективата. Така той преминава в пространството на камерата. Последния стадий е да се проектира на двумерна равнина (пространство на екрана) и вече е готов да се покаже на екрана. Трансформации стават с помощта на матрици. За преминаване от локално в глобално пространство всяка точка се представя като вектор и се умножава по мащабираща матрица, ротационна матрица и транслационна матрица. Вектора се умножава по матриците точно в този ред. Причината за това е че умножението на матриците не е комутативно.



Фиг 3.5 UML Диаграма на OSGF3DDrawableComponent.

Тези матрици се пазят в OSGF3DDrawableComponent и имат няколко начина за задаване. Могат да се зададат директно, като се извика съответния сетър приемащ матрица. Транслационната и мащабиращата матрица могат да се зададат чрез трите компонента x,y,z или чрез вектор, който съдържа същите компоненти. Ротационната матрица може да бъде създадена чрез кватернинион или чрез задаване на завъртането по осите x, y, и z.

Освен сетърите, с които заменяме старата матрица с нова , независимо от старите и стойности, има методи които променят текущите стойности. Това са Scale, Move и Rotate*. Те умножават старите матрици по нови такива. Тези методи са удобни да се ползват ако едно тяло искаме непрекъснато да го променяме, като промените стават плавно. Например ако имаме един обект, който искаме да местим в сцената, можем да използваме метода Move, който ще прибави към текущата позиция вектор, който представлява отместването

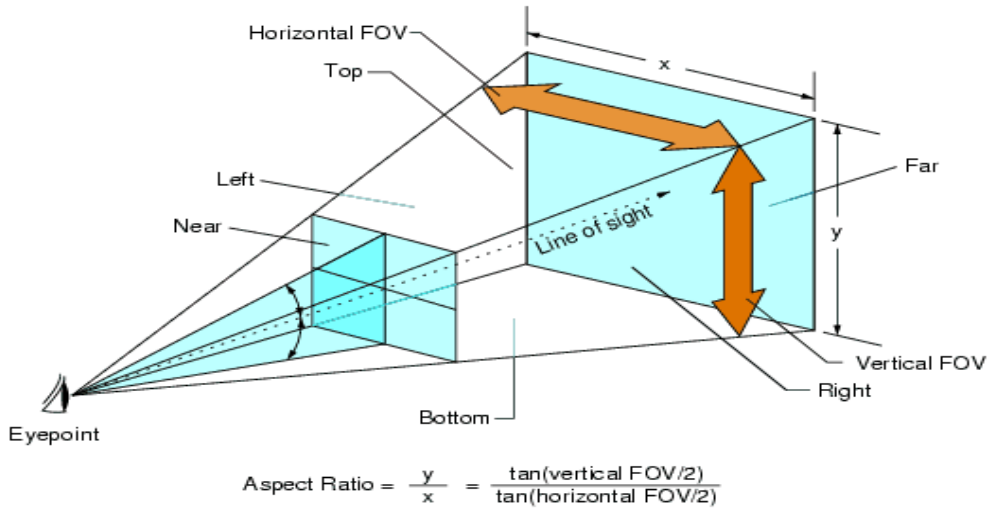
от старата позиция. Преместването става по-удобно така, защото иначе се налага всеки път да взимаме текущата позиция и да изчисляваме новата. Всички методи могат да се видят от Фиг 3.5, на която е изобразена UML диаграмата на този клас.

Както се вижда от UML диаграмата класа притежава сетъри и гетъри за оразмеряването,ротацията и транслацията на обекта. Сетърите имат предефинирани версии за да може да се задават съответните свойства по най-удобния за ситуацията начин. Освен задаването на транслация , ротация и оразмеряване има две двойки, сетър и гетър, за Камерата и за Светлината в сцената.

3.8 Реализация на камерата в фреймуърка

3.8.1 Виртуалната камера в 3D сцените

Камерата е специален обект в играта, който наподобява на видео камерата в реалния живот. Тя заснема сцената от определено място и я проектира в/у двумерна повърхност. На камерата трябва да се зададе местоположение, посока на която е обърната,близка и далечна равнина съотношение на широчина към височина на екрана и ъгъл на видимост. Ъгълът на видимост определя каква част от сцената може да бъде проектирана на екрана. Ако той е прекалено малък, обектите изглеждат много близко, а ако е прекалено голям се получават изкривявания.



Фиг 3.6 Принцип на работа на камерата в играта.

Близката и далечната равнина определят колко далеч или близко може да бъде един обект и

да бъде рендиран. На Фиг 3.6 са отбелязани като Near и Far. Избора на далечна равнина зависи единствено от желаната производителност в играта. Няма да се влоши графиката, ако се избере много далечна равнина, която е на много голямо разстояние от камерата, но така в процеса на рендериране ще се обработват много повече обекти и това може значително да го забави.

3.8.2 Реализация на класа OSGFCamera

Фиг 3.7 UML Диаграма на класа OSGFCamera

Камерата е реализирана с класа OSGFCamera. Неговите методи могат да бъдат видqни на фигура 3.7.

При всяко викане на Update() метода се проверява дали информацията в viewMatrix и projectionMatrix е актуална и ако не е изчислява матриците на ново.

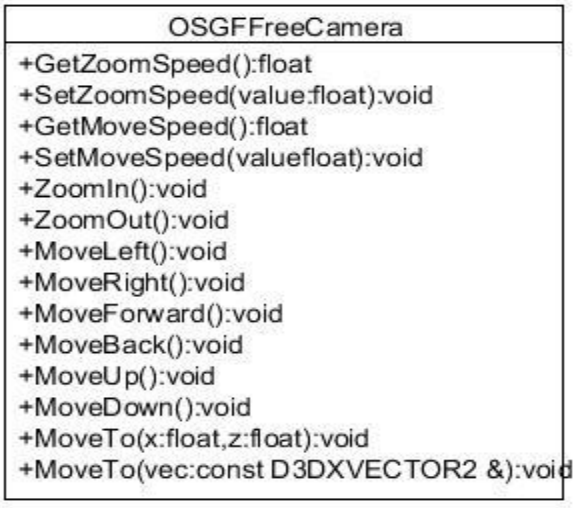
Когато се променя някой от параметрите на камерата се отбелязва в булева променлива че е

сменен и така не се налага да се преизчисляват матриците за всеки кадър или за всяка промяна на параметър.

С методите в този клас може да се променят всеки един от параметрите показани в фигура 3.6. Единствено ъгълът на видимост може да се задава само по y, но стойността му по x се изчислява спрямо съотношението на екрана. Така се избягват възможни изкривявания на обектите в тримерната сцена.

3.8.3 Класът OSGFFreeCamera

Обекта създаден чрез OSGFCamera е удобен, когато програмиста желае да създаде статична камера или камера, която ще се премества на точно определени позиции. Когато желаем да създадем камера, която да може да се движи свободно по екрана е по-удобно да се ползва OSGFFreeCamera.



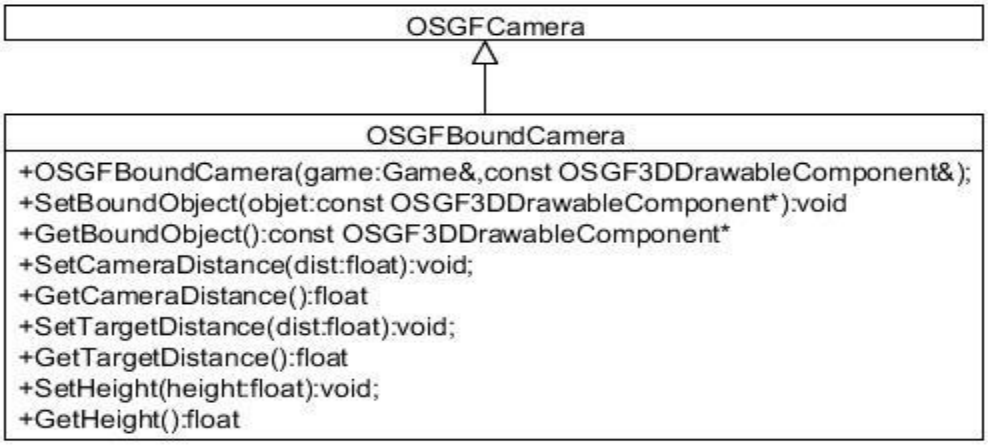
Фиг 3.8 UML диаграма на класа OSGFFreeCamera

Както се вижда от фигура 3.8, OSGFFreeCamera разполага с методи за преместване на камерата по тримерната сцена. Тези методи започват с Move и завършват с посоката, в която камерата трябва да се премести. Това с колко ще се премести зависи от скоростта на дивжение на камерата. Тя се задава чрез метода void SetMoveSpeed(float value). С методите ZoomIn и ZoomOut могат да се приближават или отдалечават обектите. Скоростта, с която това става се определя от SetZoomSpeed(). Камерата има възможност да бъде преместена

на определена позиция чрез MoveTo. Функцията е предефинирана, така че може да получава като аргумент позицията чрез две променливи от тип float или чрез DirectX вектор.

3.8.4 Класът OSGFBoundCamera

Често в игрите се налага да има камера, която да следи даден обект. Фреймуъркът разполага с такъв клас – OSGFBoundCamera.



Фиг 3.9 UML диаграма на класа OSGFBoundCamera

Както се вижда от Фиг 3.9, класът приема в конструктора си указател към OSGF3DDrawableComponent. Това е обекта към, който камерата ще бъде прикачена. Той може да бъде сменен в последствие чрез SetBoundObject().

Методите SetCameraDistance и GetCameraDistance служат да се определи или получи разстоянието на камерата от обекта по оста определена от позицията му и вектора, който показва на къде е обърнат. Стойностите подадени на тази функция изместват камерата в посока обратна на тази, на която тялото е обърнато.

Чрез SetTargetDistance и GetTargetDistance определяме или получаваме разстоянието на точката, в която е насочена камерата, от обекта. Това разстояние отново е по оста определена от позицията му и вектора, който показва на къде е обърнат, но в този случай точката се измества по посоката на която обектът е завъртян.

Чрез SetHeight и GetHeight се определя или получава разстоянието на което камерата е

издигната спрямо обекта. Посоката на издигане се определя от вектора, който указва посоката горе за тялото.

При обновяването на камерата се изпълнява следния код:

```
1. OSGFCamera::Update(dTime);
2. D3DXVECTOR3 objectPos = mBoundObject->GetTranslation();
3. D3DXVECTOR3 forward=mBoundObject->GetFrontGlobalNormalized();
4. D3DXVECTOR3 camPos = objectPos - forward*mDistance;
5. camPos+= mBoundObject->GetUpGlobalNormalized()*mHeight;
6. SetPosition(camPos);
7. SetUpVector(mBoundObject->GetUpGlobal());
8. SetTarget(objectPos + forward*mTargetDistance);
```

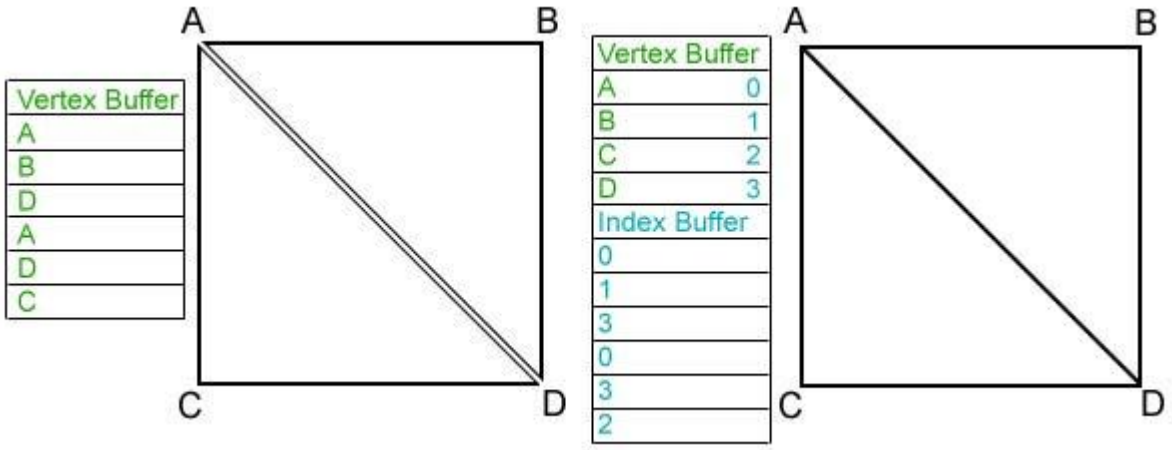
В втори и трети ред се взима позицията на обекта и посоката, на която е обърнат. И двете са представени с помощта на Direct3D вектори. На четвъртия ред се изчислява позицията на камерата. Това става като от позицията на обекта се извади единичния вектор. показващ на къде е обърнат., умножен по разстоянието. Така камерата се установява на определено разстояние зад обекта. В пети ред камерата се издига над обекта. Това става, като се прибави към позицията на камерата посоката горе за обекта, умножена по височината, зададена чрез SetHeight(). В редове 6, 7 и 8 се задават, чрез съответните сетъри, позицията, посоката на горе, на камерата, и точката, в която тя гледа. Тази точка се изчислява като от позицията на обекта се извади посоката напред, умножена по разстоянието на точката от обекта, зададена чрез SetTargetDistance.

3.9 Обекти съдържащи информация за тримерни модели

Класът OSGF3DDrawableComponent е абстрактен и не притежава логика за рендиране и съхраняване на 3D тялото. Той има няколко наследника. OSGF3DModel, OSGFMesh и OSGFDrawablePhysicalBody.

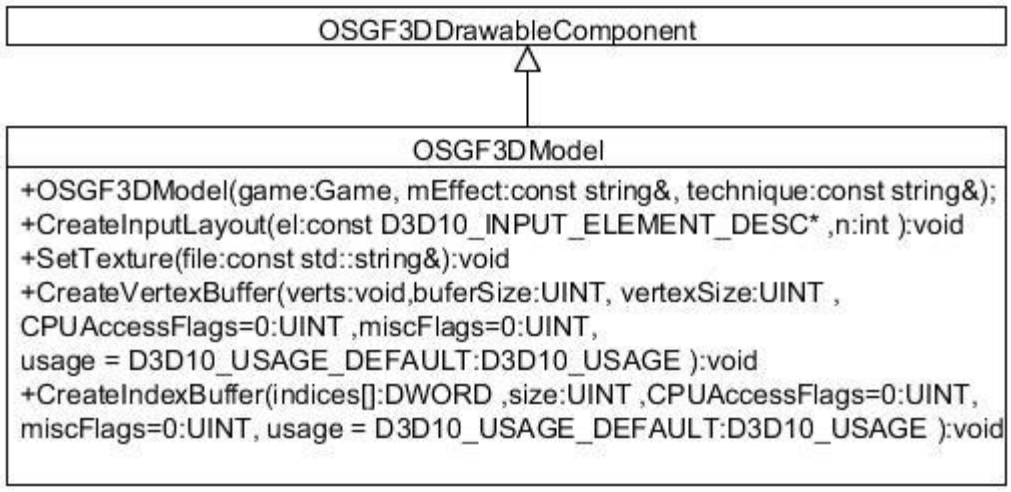
3.9.1 Класът OSGF3DModel

Класът OSGF3DModel пази информацията за модела в индекс и вертекс буфери. Идеята при тях е върховете на фасетите да се записват в вертекс буфер, а подредбата им да се пази в индекс буфер. Така се избягва дублирането на вертекси.



Фиг 3.10 Разликата при липсата на индекс буфер и при неговото наличие.

Както се вижда от 3.10, когато се използва само вертекс буфер се налага да има 6 вертекса за един квадрат, ако фасетите са триъгълници. При нарастване на сложността на модела броя на дублираните вертекси се увеличава с голяма скорост. Поради този факт, в



фреймуърка е използван само втория вариант с наличието на индекс буфер.

Фиг 3.11 UML диаграма на OSGF3DDrawableComponent

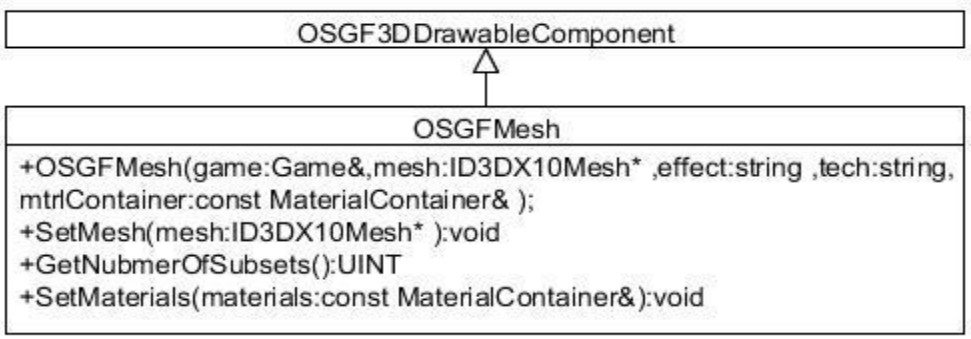
За да се използва обект от този клас, трябва да се извикат методите Initialize(),CreateVertexBuffer(), CreateIndexBuffer(), CreateInputLayout(). CreateVertexBuffer() и CreateIndexBuffer() имат два задължителни аргумента – масив с вертексите/индексите и брой на елементите в масива. На CreateVertexBuffer, освен това,

трябва да се подаде и размера на елементите в масива.

Тъй като формата на вертексите е различен. Някой могат да бъдат само позиция, други могат да имат текстура, цвят, нормала, като добавим иу факта, че и реда на елементите е от значене си проличава необходимоста да има начин тези вертекси да се описват. В DirectX това става с масив от D3D10_INPUT_ELEMENT_DESC елементи. Двата аргумента на CreateInputLayout са масив от D3D10_INPUT_ELEMENT_DESC елементи и техния брой.

Ползването на OSGF3DModel е приложимо когато искаме моделът се създава в самата програма, а не се зарежда от файл извън нея.

3.9.2 Класът OSGFMesh



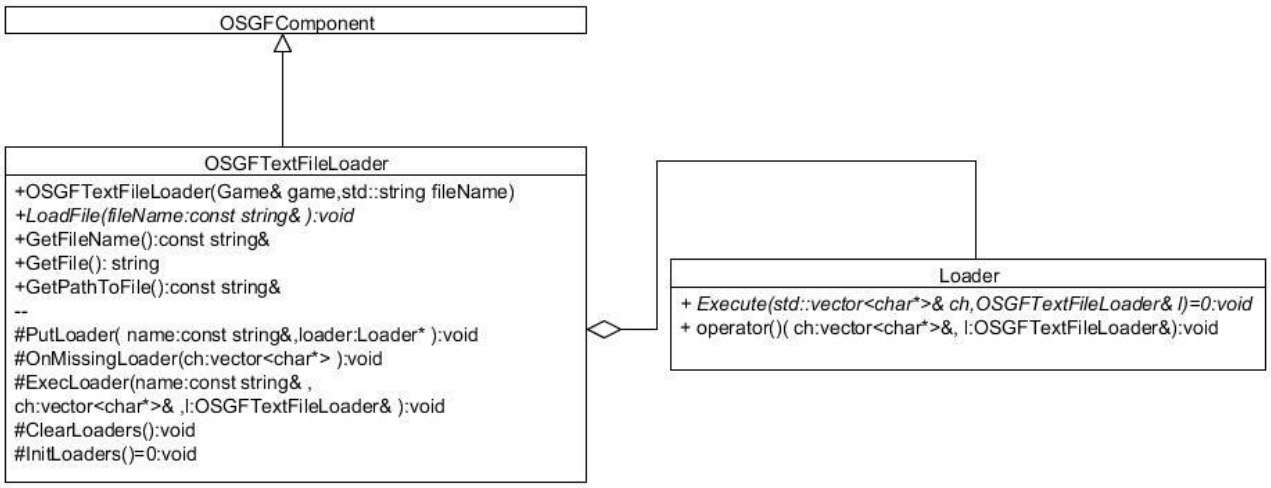
Фиг. 3.12 UML Диаграма на класа OSGFMesh

Класът OSGFMesh пази моделите в себе си чрез ID3DX10Mesh. Информацията за модела може да се подаде чрез конструктора или чрез метода SetMesh. Този клас удобен да се ползва, когато моделът трябва да се зареди от файл.

GetNumberOfSubsets() връща броя на подгрупите фасети в дадения модел. Подгрупите, най-често, са обособени части на модела с общи свойства, например, ако имаме модел на кола подгрупи ще бъдат прозорците, гумите, купето и други.

SetMaterials задава материалите на всяка една подгрупа. Материала определя как ще изглежда фасета когато се рендира. Чрез материалите се определят свойства, като как ще отразява светлина и каква текстура ще има. Материалите се съхраняват в обекти от типа Material. Тази структура съдържа информация за това каква светлина и каква част от нея

поглъща, каква текстура е приложена на фасета и алфа каналът, който определя прозрачността. Освен това има и поле от тип string в което се пази името на материала. Тези структури се пазят в обект от типа Material Container. Този клас е организиран като вектор, с тази разлика, че елементите му могат да се постъпват не само по пореден номер, а и по тяхното име.



Фиг 3.13 UML Диаграма на OSGFTextFileLoader.

3.10 Зареждане на модели от файл

Класът OSGFTextFileLoader се грижи за четенето на текстови файлове. Този клас чете файла ред по ред и разделя редовете с някакъв разделител. Първият елемент се търси в STL контейнер от тип map. Този контейнер използва за ключ string и съхранява в себе си функционални класове с базов клас Loader. Този клас има един абсолютно виртуален метод Execute и един оператор (), който вика този метод. Когато се създава нов клас за зареждане на файлове, се създават нови класове наследници на Loader и се имплементира метода Execute. Функцията Execute има 2 аргумента. Единия е вектор, който пази под низовете на реда, който е прочетен, а другия е псевдоним на OSGFTextFileLoader. Те се добавят в изображението и се викат в зависимост от срещнатия низ. Например при зареждане на OBJ файл се срещне ред започваш с vn се взима обекта от тип Loader ,който е с ключ „vn” и се вика метода Execute или се ползва оператора ().

Има няколко начина за работа с класовете наследници на OSGFTextFileLoader. Единия от

тях е, като името на файла, който ще четем се подаде на конструктора. Тогава при викането на метода Initialize() се зарежда файла. Друг вариант да се зареди файл е да се създаде обект без да се задава името на файла и след това да се викне метода LoadFile(string fileName).

В фреймуърка е създаден клас за зареждане на .obj файлове. Това е класът OSGFObjMeshLoader. Той наследява OSGFTextFileLoader, като добавя логиката, специфична за зареждането на този файлов формат. Овен наследените от базовия клас, OSGFObjMeshLoader съдържа и метод OSGFMesh* GetMesh(string effect,string tech). Чрез този метод се се взима новосъздаден и инициялизиран обект от типа OSGFMesh със зададени ефект и техника чрез аргументите на GetMesh.

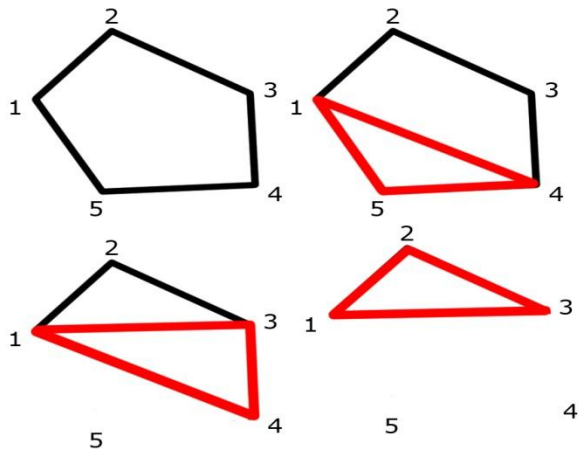
Особеност при OBJ файловете е че освен файла, който е с модела има отделен файл, който е за материалите с разширение .mtl. Това налага създаването на клас за зареждането на тези файлове. Това е класа OSGFMtlLoader. Той също е наследник на OSGFTextFileLoader. OSGFMtlLoader има още един метод - const MaterialContainer& GetMaterials().

В ID3DX10Mesh моделите могат да бъдат съставени само един вид фасети (триъгълници, четириъгълници). В OBJ файла, обаче, фасетите могат да бъдат всякакви. Може в един модел да има триъгълници, четириъгълници, петоъгълници и тн. За това когато се зарежда модела, полигоните се разделят на триъгълници. Използва се точно тази геометрична фигура, защото всеки многоъгълник може да бъде разделен на триъгълници.

```
1.  void OSGFObjMeshLoader::LoadF::Execute(std::vector<char*> &v,
2.                      OSGFTextFileLoader& l)
3.  {
4.       int i = v.size();
5.      OSGFObjMeshLoader& ol =reinterpret_cast<OSGFObjMeshLoader&>(l);
6.      while(i>3)
7.      {
8.          ol.mFaces.push_back(v[1]);
9.          ol.mFaces.push_back(v[i-2]);
10.         ol.mFaces.push_back(v[i-1]);
11.         i--;
12.         ol.mAttributeData.push_back(ol.mCurrentSubSet);
13.    }
14. }
```

В OBJ формата лице се описва с буквата f и след това изброени върховете на полигона. В случая в променливата v има, разделен на под низове, точно такъв ред. Основната идея на

този алгоритъм е да се създават триъгълници от първия, последния и предпоследния връх. След се маха последния връх и алгоритъма са повтаря докато не остане само 1 триъгълник.



Фиг 3.14 Разделяне на многоъгълник на триъгълници

Както се вижда от фигура 3.12 това действа за произволен четриъгълник, който няма ъгли над 180 градуса. Това ограничение не е голям недостатък на алгоритъма, защото в моделите изключително рядко може да се срещне такъв ъгъл. Тази забрана за многоъгълници, които имат ъгли над 180 градуса присъства в голяма част от софтуерните продукти. Например в OpenGL функцията за изчертаване на многоъгълник също не може да изчертае такъв тип фигура.

3.11 Физиката в фреймуърка

Третия наследник на OSGF3DDrawableComponent е OSGFDrawablePhysicsBody. Той опростява процедирането с твърдо тяло с физичния енджин. За да се направи такова тяло в Bullet трябва да се създадат няколко обекта:

- btCollisionShape
- btDefaultMotionState
- btRigidBody

btCollisionShape е формата на тялото(куб,сфера ид.р). Това служи за улавяне на колизии (сблъсъци). btDefaultMotionState служи за силите които действат на тялото и неговото

движение. btRigidBody е самото тяло. Конструктора на класа има два аргумента, Играта и физическия свят. Когато се създава обект от типа OSGFDrawablePhysicsBody метода Initialize трябва да се извика след като се зададът btCollisionShape и btMotionState със съответните сетъри.

Физическия свят е обект от типа OSGFPhysicsWorld. Този клас съдържа в себе си всички обекти нужни за инициализация на физичния свят. С викането на метода Initialize той инициализира цялата физика. Освен това самия клас се занимава със специфичната гравитация в фреймуърка, тъй като тя не трябва да бъде, както е описана в физичните енджини, един вектор, който да указва посоката и силата в която ще действа тя на всяко тяло, а трябва да се изчислява по реалната формула $G = (m_1 * m_2)/r^{2,}$ където m_1 и m_2 е обозначена масата на първия и масата на втория обект, а r е разстоянието м/у тях.

3.12 Получаване на вход от клавиатура и мишка

3.12.1 Метод за получаване на информация за устройствата

За получаване на вход от клавиатурата и мишката в играта е използван т.нар Raw input. Това става чрез обработката на съобщението WM_INPUT. Този начин на получаване на вход има доста предимства пред стандартните Windows съобщения. Те са предвидени за по лесна работа с програми с графичен интерфейс и по-лесно въвеждане на текст. При движението на мишката в стандартно Windows приложение се прилагат алгоритми, които променят движението така че, то да бъде по плавно и по-лесно да се уцелват определени бутони. В игра, обаче, този алгоритъм пречи. При малки движения на мишката, курсора се измества по-бавно от колкото когато този алгоритъм го няма. В игрите движението на курсора, мерника ит.н трябва да е в линейна зависимост от движението на мишката. Друга причина да се използва точно този метод за получаване на вход е входа от клавиатура. Със стандартния начин на вход,при задържане на бутон първо се генерира едно съобщение за натиснат бутон и след определена пауза почват да се генерират останалите. Това е удобно за въвеждане на текст, защото ако го няма това много често ще се дублират ненужно букви, но в игрите това би било доста неудобно. Ако се взима входа от клавиатурата по такъв начин, когато натиснем стрелка на пред или 'W' за да преместим играча си, той би направил една стъпка и чак след известно време да почне да се движи

нормално.

За получаването на вход е създаден класа InputSystem. Той притежава метода HandleInput. Той се вика от класа Game при получаване на WM_INPUT. В него се вижда от кое устройство е получен входа и се праща към съответния клас за това устройство. Базовия клас за остройствата е OSGFInputItem. Това е абстрактен клас с един абсолютно виртуален метод *virtual void HandleInput(const RAWINPUT* ri)=0;* Класът, който управлява клавиатурата се нарича OSGFKeyboard. Чрез него може да получаваме информация за състоянието на всеки един бутон в настоящия момент. Има два типа функции. Първият проверяват дали бутона е натиснат или не, а вторият тип проверяват дали бутона е установен в съответното състояние преди последното викане на функцията Update(). Методите от първия тип са bool *IsKeyDown(USHORT key)const;* и *bool IsKeyUP(USHORT key)const;*. Те са полезни когато използваме някакъв вход, който да служи за операции като придивижване на кораб, стрелба с автоматично оръгие и др. Вторият тип функции са *bool IsKeyReleased(USHORT vKey)const; и bool IsKeyJustPressed(USHORT vKey)const;* Те са удобни за навигиране в менюта, използване на умения ит.н. Ако използваме за навигация в меню IsKeyDown() когато изберем някоя от опциите в главното меню, при преминаване в подменюто има голям шанс да се отчете още веднъж, че е натиснато и да се изпълни операция от него. За вход от мишката е на разположение класа OSGFMouse. Той предоставя методи за следене на състоянието на всички бутони на мишката. Както при клавиатурата така и при мишката има методи, които следят дали бутона е натиснат или не и има методи, които следят дали е установен в съответното състояние преди последното обновяване на компонента. Методите са съответно *bool IsButtonDown(SHORT button)const* и *bool IsButtonUp(SHORT button)const* за проверка дали бутонът е в натиснато или отпуснато състояние и *bool IsButtonJustPressed(SHORT button)const* и *bool IsButtonReleased(SHORT button)const*, които проверяват дали бутонът е застанал в отпуснато или натиснато състояние преди обновяването на компонента. Класът OSGFMouse разполага с методи за проверка на позицията на курсора и промяната на позицията на курсора преди последното викане на Update(). Методите са съответно *LONG GetX()const; LONG GetY()const; LONG GetDeltaX()const; LONG GetDeltaY()const;* Методът *SHORT GetWheelRotation()const* връща завъртането на скрола преди последното обновяване. OSGFMouse притежава метод *void*

LimitTheCursorToScreenCoordinates(bool limt = true). Той задава дали координатите на мишката ще са в интервала определен от размерите на екрана или няма да са. Лимитирането на координатите е полезно когато трябва да може с мишката да се избират елементи които са на екрана, например бутони. Тогава би било голямо неудобство ако курсора е с някакви координати извън екрана и потребителя трябва да се опитва да го върне в нормално положение. Другата ситуация е удобна ако сцената е структурирана така че трябва да се взима отместването мишката и в зависимост от това да се извършва някаква операция (например завъртане на кораб). Тогава ако координатите са ограничени, когато мишката стигне края на екрана, няма да може да се взима отместването , ако то не е в посока средата на екрана.

3.13 Системата за събития

Удобен метод за комуникация м/у отделните модули е Системата за събития. Тя е реализирана с помощта на класа EventSystem. Не се използва стандартния начин за предаване на съобщения в windows приложение, защото при него съобщенията минават през различни нива докато стигнат до програмата. Това преминаване може да създаде забавяне, което да се отрази на играта.

Обработката на съобщения става чрез вътрешния клас за EventSystem – EventDispatcher. Този клас съдържа един абсолютно виртуален метод *virtual void DispatchEvent(std::intptr_t arg1,std::intptr_t arg2)=0;* Когато желаем да обработваме дадено съобщение, ние предефинираме този клас като имплементираме неговия този метод. Класа предефинира и оператора () което позволява да се изпълнава желаната операция по следния начин: event(message,arg1,arg2), където event е обект от клас наследник на EventDispatcher.

За да се създаде съобщение, трябва да се извика метода *void SendMessage(UINT message, std::intptr_t arg1,std::intptr_t arg2).*Той има три аргумента. UINT message е идентификатор на съобщението. Чрез него се определя кой EventDispatcher да се ползва. Чрез параметрите arg1, и arg2, се предава желана информация за съобщението. В тях може да се съхранява всякаква информация от координати на тяло, до указатели към обекти. Типа на аргументите е *std::intptr_t.* Това е целочислен тип. Неговия размер е размера на указателя на конкретната с-ма. Тъй като е използван този тип за аргументите, можем да

предаваме и указатели към обекти, чиито размер не се събира в целочислен примитив.

Когато искаме да използваме определен *EventDispatcher* ние го добавяме чрез метода UINT *SetDispatcher(, EventDispatcher* dipatcher).* Където a dispatcher е обекта от тип *EventDispatcher*, който ще обработва съобщението. Функцията връща номера с, който се идентифицира новото съобщение. Всяко съобщение получава един номер след предишното съобщение. Изключение прави случая когато са премахвани съобщения, тогава то получава номера на последното премахнато съобщение. Премахването става чрез функцията *void RemoveDispatcher(UINT message),* където чрез message указваме кое съобщеиние вече не искаме да обработваме.

В класа Game има обект от типа EventSystem. Той бива достъпван през метода *EventSystem& GetEventSystem().* Този обект е поставен там, защото всеки компонент има референция към Game и така може да бъде постъпвана лесно системата за събития.

В класа OSGFComponent има няколко метода, които спомагат работата със събития и съобщения. Те са UINT *SetDispatcher(EventDispatcher* dipatcher), void RemoveDispatcher(UINT message, EventDispatcher* dipatcher)* и void *SendMessage(UINT message, std::intptr_t arg1,std::intptr_t arg2).* Тяхната работа е да викат едноименните методи в EventSystem. Тези методи са защитени, защото е необходимо през всеки компонент да бъде достъпена системата за събития.

3.14 Класът OSGFShip

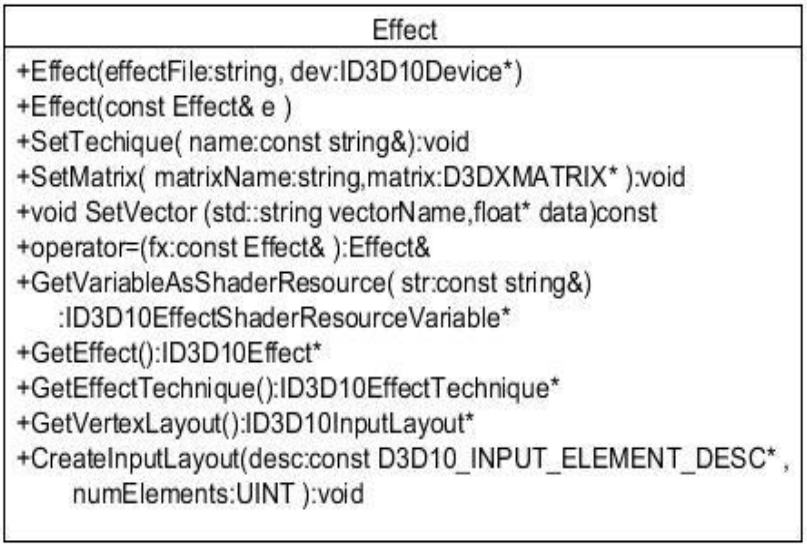
За създаването на космически кораб, фреймуърка предлага класа OSGFShip. Той е наследник на OSGFDrawablePhysicsBody. Освен наследената от базовия клас, OSGFShip предлага функционалност за добавяне на компоненти. Класът за компоненти е OSGFShipComponent. Този клас съдържа информация за позицията и посоката на която е обърнат съответния компонент. Тя се задава чрез методите *void SetPos(const btVector3& pos);void SetDirection(const btVector3& dir);* или могат да се зададе чрез конструктора. За да се вземе тази информация се използват съответните гетъри.

Двигателите са обекти от класа OSGFShipEngine. Той е наследник на OSGFShipComponent. Чрез тях се прилага сила на определено място и с определена посока в/у тялото на кораба. Това става чрез метода void RunEngine().

За оръдията има клас OSGFShipTurret. Както OSGFShipEngine той също е клас наследник на OSGFShipComponent. Този клас добавя метода Fire с който могат да се изстрелват снаряди. Снарядите са обекти от типа OSGFProjectile. Този клас наследява OSGFDrawablePysicslBody, като добавя свойството време за живот. То служи да се разбере кога снаряда трябва да се изчезне, тъй като, ако снарядите не се изтриват, след по-продължителна игра те ще заемат много излишни ресурси. В OSGFProjectile е предефиниран метода Update да намалява времето на живот всеки път когато се извика. Класът, който следи за живота на снарядите е OSGFProjectileSystem. Той се грижи да обновява снарядите и да проверява дали имат още време да стоят в играта. Когато види че времето им е изтекло той ги премахва.

3.14 Оперирane със шейдъри

Шейдърите са важна част от всяко едно 3D приложение. За това в фреймуърка има клас Effect. С него се улеснява зареждането и работата с ефекти.



Фиг 3.15 UML Диаграма на класа Effect

Конструктора на класа приема два аргумента, името на .fx файла, който искаме да заредим и указател към ID3D10Device. ID3D10Device е основния интерфейс в Direct3D, той представлява виртуален адаптер за Direct3D 10.0. Използва се да за рендиране и създаване на Direct3D ресурси. Класът Game създава такъв обект, който може да се вземе

чрез метода GetDevice().

Във всеки ефект има поне 1 техника. Техниката съдържа Pixel Shader, Vertex Shader и евентуално Geometry Shader. С метода SetTechnique задаваме коя техника да се ползва от съответния файл. За да работим с ефект, често се налага да подаваме информация от главната програма като позиция на източник на светлина или трансформиращи матрици. За това в класа OSGFEffect има методи SetMatrix и SetVector. Те приемат 2 аргумента, първия е името на променливата, на която искаме да дадем стойност, а другия са данните които ще зададем.

От класа могат да се вземат Direct3D интерфейсите за ефекта и за техниката. Това е необходимо ако потребителя иска да използва някой от техните методи, които ги няма в OSGFEffect.

Метода GetVariableAsShaderResourse(stiring name) връща указател към ID3D10EffectShaderResourceVariable. Този интерфейс се ползва когато искаме да подадем данни на ефекта, като на пример – текстура.

В фреймуръка има няколко файла с различни ефекти. Част от тях служат за дебъгване, докато друга част служат за създаването на истински тримерни сцени. Ефектите за дебъг, най често оцветяват цялата геометрия в 1 цвят(naprimer monoColor.fx) или я оцветяват в зависимост от някой от компонентите на вертекса (cool.fx и cool2.fx). Ефектите, с които се създават истински сцени са Light.fx, LightAndTexture.fx и Texture.fx. Light.fx създава попикселово осветление. При него светлината се разделя на 3 компонента:

- Ambient - светлината която обгръща тяло сцената. Тази светлина няма определен източник, тъй като това е общия светлинен фон на сцената. Тази светлина просто се добавя към цвета на обекта.[6]
- Diffuse – симулира осветление на грапави повърхности.

  Формулата за това осветление е следната: $outColor = (N * L)$ Където N е нормалата на вертекса, а L е позицията на вертекса спрямо светлинния източник. С * е обозначено скаларно произведение.[6]
- Specular – симулира осветление на гладки и "лъскави" повърхности. Формулата

е следната $outColor = (R * V)^S$. R - отразен около нормалата светлинен лъч.V - вектор от повърхността на обекта към окото
S - определя колко "лъскав" е един обект. [6]

Цветът на дадена точка се изчислява като се умножи цвета на матеряла й по цвета на светлината и получената стойност се умножи по outColor от някоя от горните формули. Матерялът е стойност, която указва колко червена, зелена и синия светлина отразява дадената повърхност, на която принадлежи точката.

Texture.fx поставя текстура в/у фасетите на тялото. Всеки вертекс има текстурни координати в интервала [0;1]. На базата на тези координати се взимат точки от двумерното изображение и на базата на тези точки се извършва интерполация за да се изчислят междинните цветове. Този алгоритъм го има вграден, ние трябва само да му подадем вертексите, текстурата и различните опции за това как да я сложи.

Ефекта LightAndTexture.fx е смес от горните два. Той извършва попикселово осветление и поставя текстури.

3.15 Sky box

В игрите, които се развиват на открито се създава т. нар. Sky box или Sky dome. Това е триизмерен обект, най-често куб или сфера, който обгръща останалите елементи на сцената. На не го се поставя текстура, чрез която се създава фон в играта. Така се създава усещането че в далечината има обекти, например звезди или облаци, докато всъщност това е едно двумерно изображение. Когато обектите са много далеч от камерата, паралаксът почти не се забелязва. Така може да се създаде илюзията, че двумерното изображение всъщност е триизмерно. Sky box трябва да бъде с много големи размери и неговия център да съвпада с позицията на камерата. Така играчът не може да достигне до края на Sky box и да забележи, че това е двумерно изображение поставено в/у куб или сфера.

3.16 Тестване на фреймуърка

За да се тества фреймуърка е разработена малка игра. В нея са създадени три състояния:

1. Main menu

2. Main game
3. Physical State

3.16.1 MainMenuState

В Main menu има 3 опции. Изход от играта, преминаване в Main game и преминаване в Physical . Състоянието MainGame е създадено като е наследен класа OSGFMenu. В него се поставят три бутона приблизително по средата на екрана. Бутоните са обекти на класа OSGFMenuItem. При промяна на размера на прозореца се променя и местоположението на елементите. Това става чрез съответния код в OnScreenResize метода на MainMenu.За бутоните в менюто е използван MenuText. Те са червени докато не са активни и жълти, когато са активни.

3.16.2 MainGameState

В това състояние се влиза, когато се избере бутона New Game от главното меню. В него се предимно неща свързани с графиката. В него се зарежда един модел на кораб, на който да се пробват различни ефекти. В това състояние има и SkyBox със звезди. В тази сцена има и OSGFFreeCamera чрез, която може да се разглежда цялата сцена. Камерата се движи чрез клавиатурата и мишката. При натсикане на бутона Esc играта се връща в състоянието MainMenu. В това състояние се използва и OSGFDrawableTextManager. Чрез него може да се променя текста, който се изписва в горния ляв ъгъл на екрана. При натискане на бутона 'H' на клавиатурата се извежда обикновено текстово съобщение, а ако се натисне 'J' се извежда информация за изминалото време от стартирането на играта, кадрите за секунда и времето, което изминава до смяната на кадъра. Така се тества GameTimer, дали работи правилно.

3.16.3 PhysicGameState

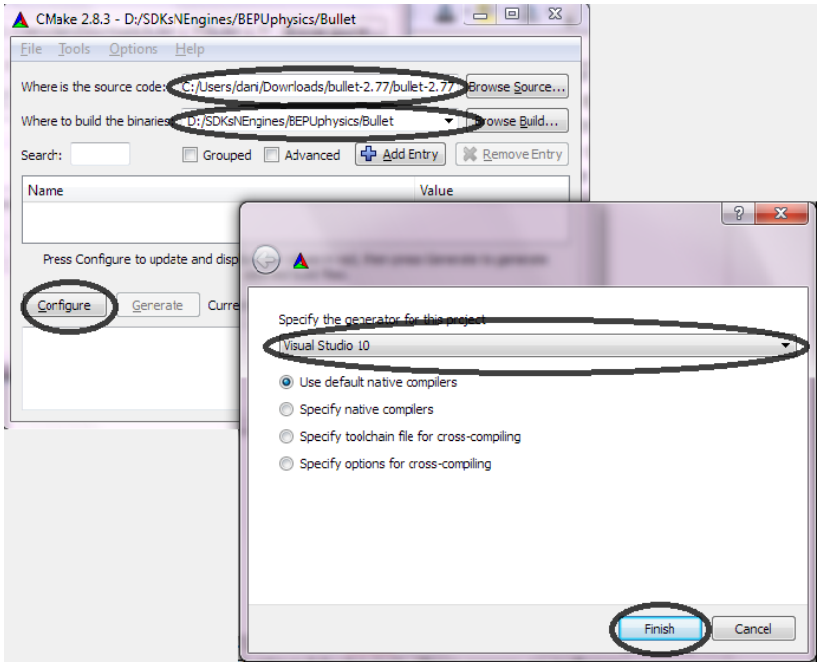
В това състояние се преминава, когато се избере бутона Physics Test. В това състояние има космически кораб, който се задвижва чрез физиката и може да стреля. Има и планета, която го привлича. Тук камерата е OSGFBoundCamera, за да може да се следи движението на кораба.

ГЛАВА4

Ръководство на потребителя

4.1 Инсталация

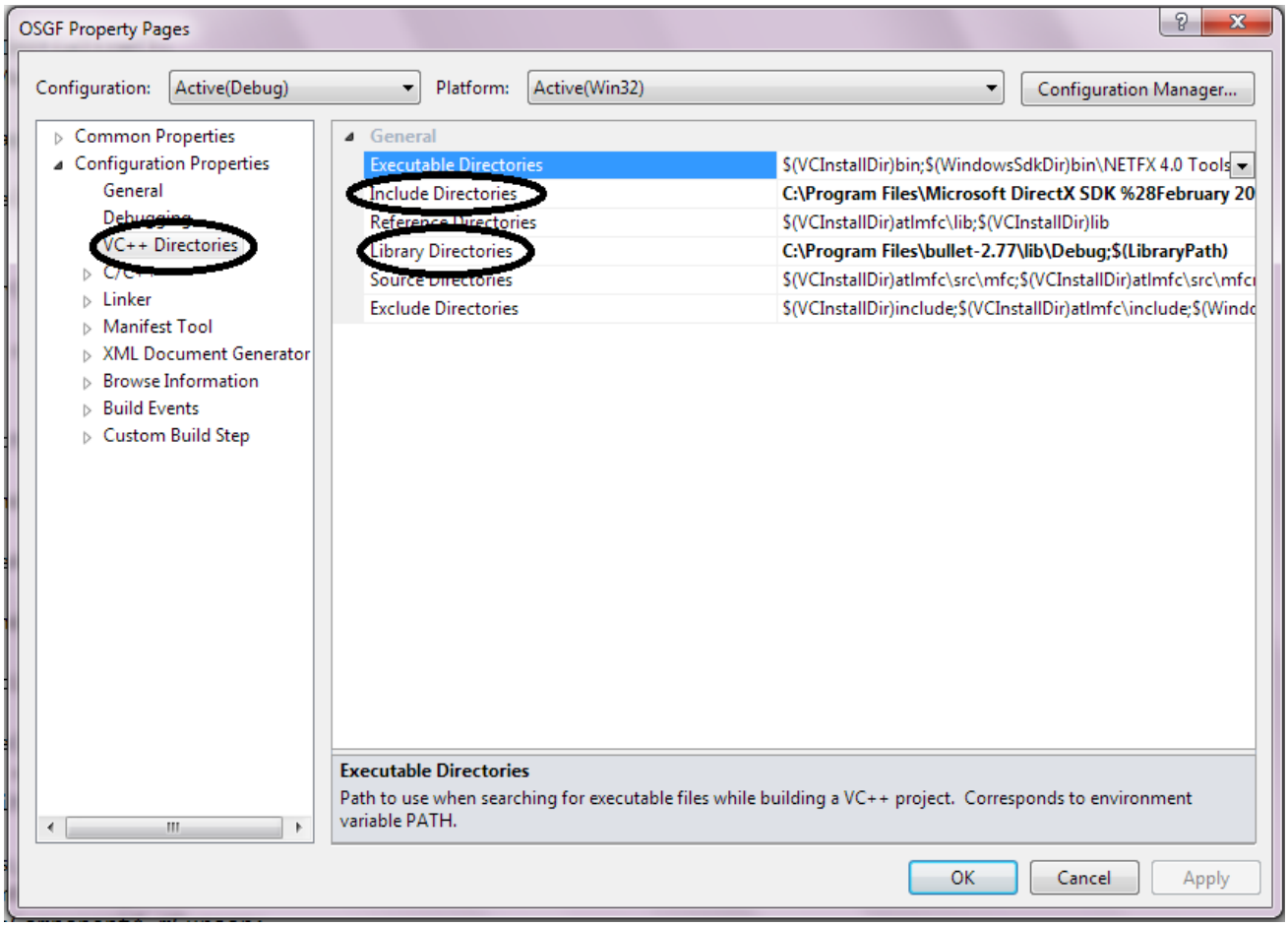
За да се използва фреймуърка трябва да бъде инсталиран DirectX 10 и Bullet Physics. За инсталацията на DirectX трябва да бъде ползван DXSDK_June10.exe. За да се ползва Bullet той трябва да се компилира на машината, на която ще бъде ползван за разработка на приложението, което ще го използва . За да стане това, трябва да се разкомпресира файла bullet-2.77.zip и да се създаде проект, който да бъде компилиран. Удобен начин за създаването на проект е използването на програмата cmake. Ако не е инсталирана на компютъра на потребителя, това може да стане от файла cmake-2.8.3-win32-x86.exe. За да се създаде проект с нея се стартира файла cmake-gui. В него се задава мястото на файловете с кода и мястото където потребителя желае да са компилираните файлове. След това се избира Configure и там избираме компилатора, които потребителя желае да ползва и кликаме в/у бутона Finish. Накрая се избира бутона Generate, който вече трябва да е активен и проекта е готов за компилира .



Фиг 4.1 създаване на проект чрез CMAKE

4.2 Използване на фреймуърка в проект

За да се създаде проект който ползва OSGF трябва да се укажат директориите, където се намират необходимите .h и .lib файлове. В Visual C++ 2010 Express Edition това става по следният начин. От стандартната лента се избира менюто Project. От падащото меню се избира опцията <името на проекта> Propertyes (или се използва клавишната комбинация Alt+f7). След като бъде избрана тази опция е появява изчакащ прозорец. В него се избира Configuration Properties >> VC++ Directories. Там се променят стойностите в Include Directories и Library Directories. Има два начина да стане това. Директно да се кликне върху текста и да се редактира или да изберем опцията Edit от падащото меню, което излиза след като се натисне на стрелката до текста.



Фиг 4.2 Указване на пътя до заглавните и библиотечните файлове в Visual C++ 2010 Express Edition

4.3 Създаване на игра

За да се създаде игра трябва да се създаде обект от тип Game или от негов наследник. След това трябва да се създадат сцени които да бъдат добавени в играта. Създаването на състояние става като се наследи класа GameState. След като се създаде логиката на сцената в класа, се създава обект, който се добавя в StateMachine чрез метода Add. Достъп до стейт машината можем да получим чрез метода StateMachine& Game::GetStateMachinie(). Друг начин да добавим състояние е да използваме метода AddState на класа Game. Последната стъпка е да се зададе активно състояние и да се извика метода Run на Game.

4.4 Получаване на вход от клавиатура и мишка

Получаване на вход от мишка и клавятура става чрез c*onst OSGFKeyboard& Game::GetKeyboard()const;* и *const OSGFMouse& GetMouse()const;* За вход от клавятурата се ползва върнатия обект от типа *OSGFKeyboard*. Когато трябва да се получи информация за състоянието на бутон, се викат методите *OSGFKeyboard::IsKeyDown(UINT vk)* и *OSGFKeyboard::IsKeyUp(UINT vk).* Тези методи връщат истина ако в момента бутона е натиснат/отпуснат и не зависят от предишното му състояние. Методите *OSGFKeyboard::IsKeyJustPressed(UINT vk)* и *OSGFKeyboard::IsKeyReleased(UINT vk),* проверяват дали бутона се е установил в съответното състояние в текущия момент. Аргумента на методите е един от кодовете на виртуалните клавиши. Подробна информация за това коя стойност за кой клавиш отговаря може да се получи от следната страница: http://msdn.microsoft.com/en-us/library/ms645540(v=vs.85).aspx. За вход от мишката се използва класа OSGFMouse. Той има методи, с които може да се получи информация за позицията на курсора, състоянието на бутоните, отместването на курсора и завъртането на скрола преди последното извикване на Update(). За да се получи позицията на курсора се използват методите *LONG GetX()const; LONG GetY()const;* Трябва да се обърне внимание на факта, че позицията на курсора получена чрез OSGFMouse по никакъв начин е е свързана с позицията на стандартния курсор в Windows. Тази разлика се дължи на факта, че OSGFMouse получава информация за входа чрез така наречения Raw input, докато информацията за курсора в Windows минава през различни филтри и алгоритми от курсора в OSGFMouse. Този клас притежава метод

Методите *LONG GetDeltaX()cosnt и LONG GetDeltaY()cosnt* получават информация за това с колко се се изместил курсора от преди последното викане на Update. Чрез метода *SHORT GetWheelRotation()const* можем да получим информация за завъртането на скрола.

4.5 Работа със състоянията

Фреймуърка е базиран на т.нар State Machine. Така играта е разделена на състояния като всяко състояние се грижи да обработва входните данни и да извежда информация на екрана. Стейт машината се грижи да има само едно активно състояние, което да обновява и да изчертава. За да се добави състояние, програмиста трябва да предефинира един от класовете GameState, OSGFLevel, OSGFMenu. GameState е базовия клас на състоянията когато го наследвяваме трябва да се имплементира метода *void HandleInput(double dTime).*Този метод се вика при всяко обновяване на сцената и служи за получаване и интерпретиране на входни данни. Най-вероятно програмиста ще желае и да предефинира методите Update, Render, Load и Unload. Когато правим това, не трябва да се забравя да извикат същите методи в базовия клас.

Когато трябва да се смени състоянието се използваме метода SetActive на StateMachine. SetActive получава аргумент от тип string, чрез който се указва коя сцена да стане активна.

Другите два класа OSGFMenu и OSGFLevel наследяват GameState и предлагат по лено създаване на състояния, които са меню или ниво в играта. OSGFMenu разполага с вграден курсор и методи за добавяне на елементи в менюто. Менюто се състои от няколко на брой елемента, като има един селектиран. За да добави елемент към менюто прорамиста може да използва някои от наслениците на OSGFMenuItem (например MenuText) или да създаде свой като наследява този клас. Всеки елемент в менюто има две състояние Active и Inactive, за чиято смяна се грижи OSGFMenu. Със метода SetDrawData се задава компонента, който да бъде изрисуван като елемент в менюто. Ако се ползва и SetActiveDrawData може да се зададе втори компонент, който да се изрисува само ако елемента в менюто е селектиран.

Действието което трябва да извърши елемента се задава чрез указател на метод в класът за менюто чрез метода SetCommand. Метода трябва да връща void и да получава като

аргумент указател към OSGFMenuItem*, с който може да бъде достъпен компонента от който е извикан.

Когато трябва да се създаде състояние, което представлява 3D сцена е удобно да наследим класа OSGFLevel. Той съдържа логика за общи неща в тримерните сцени, като камера и светлинни източници. Освен това има сетъри и гетъри с които можем да четем и променяме параметри на тези компоненти на сцената.

4.6 Зареждане на тримерни модели

В играта програмистите, най-вероятно ще искат да зареждат триизмерни модели. Това става чрез съответните класове за зареждане на файлове. Те се използват по следния начин: първо се създава обект от такъв клас. Програмиста има избор дали още тук да зададе името на файла, който ще зарежда като използва конструктора с 1 или с 2 аргумента. След това се вика Initialize(). Ако програмиста не е задал името на файла още в конструктора на класа е необходимо да се извика метода void Load(string name), където задаваме кой файл искаме да заредим. Накрая се извиква метода GetMesh(string effect, string tech). Този метод връща указател към обект от типа OSGFMesh*. Този обект е инициализиран и със зареден ефект. Единственото което трябва да се направи преди да се ползва е да се подадат камерата и светлините на съответната сцена чрез съответните сетъри SetCamera и SetLight.

Ако поради някаква причина програмиста иска да зареди модел, който да го генерира чрез програмата може да ползва класа OSGF3DModel. За да се създаде обект трябва да се подаде Game обекта, името и пътя до файла с ефекта и използваната техника. За да създаде такъв модел трябва да укаже формата на вертексите чрез масив от D3D10_INPUT_ELEMENT_DESC, да създаде масив с вертексите и масив с индексите и да ги подаде съответно на – CreateInputLayout CreateIndexBuffer и CreateVertexBuffer. Когато е готов с това трябва да извика метода Initialize и модела е готов. Отново за да може да се използва в сцената трябва да се му се подадат камерата и светлините.

4.7 Използвне на физика

За използва физиката на разположение е класа OSGFPhysicsWorld. Той капсулира

физиката създадена с физичния енджин, и освобождава програмиста от някои особености при инициализацията на енджина. Освен това се грижи за гравитацията, така че да могат да се създават точки които да привличат телата към себе си. За създаването на обект от тип OSGFPhysicsWorld е необходимо само да се подаде обекта на играта. За да се работи с него се извиква Initialize() и след това с методите Add и AddGravityCenter се добавят съответно тела и гравитационни центрове. Add е наследен от OSGFComponentContainep и има за аргумент низ който е ключ, с който се достъпва елемента и указател към обекта, който добавяме. Въпреки че, аргумента е от тип OSGFComponent трябва да се подават обекти от тип OSGFDrawablePhysicsBody иначе се хвърля изключение.

4.8 Използвне на системата за събития

С класа OSGFDrawablePhysicsBody се създават твърди тела, които да се добавят в физичната симулация. За да се създаде такова тяло е необходимо да се викне конструктора на класа с аргументи играта и OSGFPhysicsWorld. След това трябва да се зададат различните свойства на обекта и на края да се извика Initialize, който да добави елемента в физичния свят.

За да се използва системата за събития в фреймуърка е необходимо единствено да се добави диспечер, който да следи за определено събитие. Диспечера се създава, като се наследи EventSystem::EventDispatcher и се имплементира virtual void DispatchEvent(std::intptr_t arg1, std::intptr_t arg2)=0;. В този метод се слага кода който искаме да се изпълни при постъпване на събитие. Когато създадем обект диспечер, трябва да го добавим към евент системата. Има два начина за това. Единият е са се добави като се извика метода UINT SetDispatcher(EventSystem::EventDispatcher* dispatcher); на класа обекта от тип EventSystem, който е в класа Game, а другия е с аналогичния му метод в класа OSGFComponent. И двата метода връщат цяло число без знак, което служи за идентификатор на съобщението. Чрез него се създава съобщение, използвайки метода SendMessage. Когато трябва да се предаде някаква допълнителна информация със съобщението, това може да се направи чрез втория и третия аргумент на SendMessage. В тези аргументи може да се постави целочислена стойност или указател към някакъв друг обект. Аргументите са 2, тъй като така могат да се предават и масиви, като в първия аргумент се постави самия масив, а във втория размера му.

4.9 Добавяне на космически кораби в играта

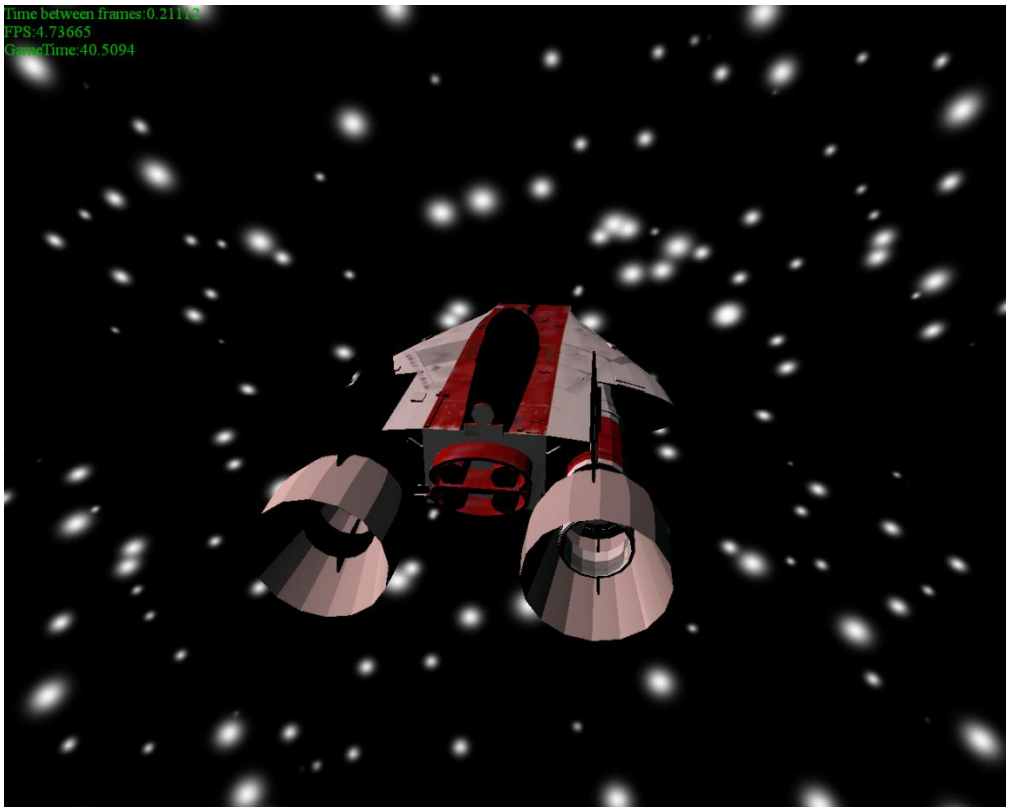
За да се добави кораб в играта се използва OSGFShip. Той е наследник на OSGFDrawablePhysicalComponent и се работи с него почти по същия начин. Разликата е че могат да се добавят компоненти към кораба. За момента има реализирани оръдия и двигатели. За да се добави компонент трябва да се извика съответния сетър. За Двигателите това е SetEngine, а за оръдията SetTurret. С методите RunEngnie и FireTurret се борави съответно с двигателите и с оръдията.

5.0 Примерната игра създадена с фреймуърка

За проверка на работоспособността и за демонстрация на способностите на фреймуърка е създадена игра. Нейният изпълним файл се намира в директорията Debug. Неговото име е OSGF.exe. Когато се стартира се зарежда сцената с главното меню.

Фиг 4.3 Главното меню

От фигура 4.3 се вижда че в менюто има 3 опции. Нова игра (New Game), тест на физиката (Physics Test) и изход от играта (Quit). Когато се избере нова игра се демонстрира зареждането на тримерни модели, създаването на сцена за играта и добавянето на двумерни графични компоненти в играта.



Фиг 4.4 Екрана когато се избере New Game от главното меню.

Както се вижда от фигура 4.4, е зареден триизмерен модел на космически кораб, небе със звезди и информация за състоянието на сцената в горния ляв ъгъл.

Когато бъде избран Physics Test играча има възможност да управлява космически кораб в открития космос. Клавишите, с които се използват са W, S, A, D – съответно за полет на пред, на зад , за вой на ляво и завой на дясно. Със стрелките на горе и на долу кораба се издига и снишава спрямо посоката избрана за нагоре. А с стрелките наляво и надясно се завърта кораба около оста определена от правата, по която се движи.

Когато е избран Physics Test или New Game с клавиша Esc потребителят може да се върне в главното меню. Когато е натиснат Esc в главното меню се излиза от приложението.

ЗАКЛЮЧЕНИЕ

В тази дипломната работа е разработена една основа, която би помогнала в разработката на компютърни игри, които се развиват в космоса. Тя е съобразена с особеностите на динамиката в космическата среда. Чрез разработения фреймуърк се намалява времето за разработка на една такава игра. Освен това програмистът може да се концентрира в разработката на по-интересен геймплей и да не се занимава със неща, като инициализация на прозорец или графична библиотека. Фреймуърка предлага добра скалируемост и това улеснява по-нататъшната му разработка. За бъдещо развитие е планирано да се добави система за по-удобно управление на корабите, модули за мултиплеър и изкуствен интелект, да се разшири броя файлови формати, които да е възможно да се зареждат.

ИЗПОЛЗВАНА ЛИТЕРАТУРА

1. http://www.bluesnews.com/archives/carmack122396.html
2. http://www.team5150.com/~andrew/carmack/johnc_interview_2007_CES_2007__John_Carmack_And_Todd_Hollenshead_Speak.html
3. http://www.tomshardware.com/reviews/opengl-directx,2019-9.html
4. http://msdn.microsoft.com/
5. http://nexe.gamedev.net/directknowledge/default.asp?p=Batching
6. http://vitaminche.wikidot.com

СЪДЪРЖАНИЕ

Увод 3

ПЪРВА ГЛАВА 4

1.1 Сравнение на водещите библиотеки за 3D графика - OpenGL и DirectX 4

1.1.2 Преносимост 4

1.1.3 Сложност 4

1.1.4 Структура и функционалност на двете API: OpenGL и DirectX 5

1.1.5 Производителност 5

1.1.6 Област на приложение 6

1.2 Езици за шейдъри 7

1.2.1 Видове шейдъри 8

1.3 Физични Енджини 9

1.3.1 Причина да се ползва физичен енжин 9

1.3.2 Видове физични енджини 9

1.3.3 Работа с физични енджини 10

1.3.4 Избор на физичен енжин 10

1.4 Среди за разработка на C++ приложения 11

1.4.1 Visual C++ 11

1.4.2 Dev C++ 11

1.4.2 MinGW studio 12

1.4.3 Eclipse for C++ 12

1.4.4 Code::Block 12

1.5 Системи за контрол на версиите 12
1.5.1 Примитивни методи 12
1.5.2 Автоматизирани системи за контрол на версиите 13
1.5.3 Централизирани и децентрализирани системи за контрол на версиите 13
ВТОРА ГЛАВА 15
2.1 Изисквания към фреймуърка 15
ТРЕТА ГЛАВА 19
3.1 Класът Game 19
3.2 Класът OSGFComponent 21
3.3 Класът OSGFDrawableComponent 22
3.4 Класовете OSGFComponentContainer, OSGFDrawableComponentContainer, OSGFComponentManager и OSGFDrawableComponentManager. 22
3.5 Реализацията на State Pattern 23
3.6 Класът OSGF2DDrawableComponent 24
3.6 Реализация на класа OSGFMenu 25
3.7 Реализация на класа OSGF3DDrawableComponent 26
3.8 Реализация на камерата в фреймуърка 29
3.8.1 Виртуалната камера в 3D сцените 29
3.8.2 Реализация на класа OSGFCamera 30
3.8.3 Класът OSGFFreeCamera 31
3.8.4 Класът OSGFBoundCamera 32
3.9 Обекти съдържащи информация за тримерни модели 33
3.9.1 Класът OSGF3DModel 33
3.9.2 Класът OSGFMesh 35

3.10 Зареждане на модели от файл....36
3.11 Физиката в фреймуърка....38
3.12 Получаване на вход от клавиатура и мишка....39
3.12.1 Метод за получаване на информация за устройствата....39
3.13 Системата за събития....41
3.14 Класът OSGFShip....42
3.14 Опериране със шейдъри....43
3.15 Sky box....45
3.16 Тестване на фреймуърка....45
3.16.1 MainMenuState....46
3.16.2 MainGameState....46
3.16.3 PhysicGameState....46
ГЛАВА4....47
4.1 Инсталация....47
4.2 Използване на фреймуърка в проект....48
4.3 Създаване на игра....49
4.4 Получаване на вход от клавиатура и мишка....49
4.5 Работа със състоянията....50
4.6 Зареждане на тримерни модели....51
4.7 Използвне на физика....51
4.8 Използвне на системата за събития....52
4.9 Добавяне на космически кораби в играта....53
5.0 Примерната игра създадена с фреймуърка....53

ЗАКЛЮЧЕНИЕ 55

ИЗПОЛЗВАНА ЛИТЕРАТУРА 56