## **TЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**



**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

**по професия код 481020 „Системен програмист“**

**специалност код 4810201**  **„Системно програмиране“**

Тема: Компютърна графика - 3D engine

Дипломант: Дипломен ръководител:

*Денислав Ивайлов Иванов Александър Ангелов*

СОФИЯ

2 0 2 2

**TЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**



**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 14.12.2021 г. Утвърждавам:..............................

Дата на предаване: 14.03.2022 г. /проф. д-р инж. Т. Василева/

**ЗАДАНИЕ**

**за дипломна работа**

**ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ**

**по професия код 481020**  **„Системен програмист“**

**специалност код 4810201**  **„Системно програмиране“**

на ученика Денислав Ивайлов Иванов от 12В клас

1. Тема: Компютърна графика - 3D engine
2. Изисквания:

2.1. Графичен engine, който визуализира 3D сцени, базиран на C++ и OpenGL

2.2. Подвижна камера, която обхожда сцената

2.3. Поддръжка на Anti aliasing

2.4. Поддръжка на 2D модели

1. Съдържание

3.1 Теоретична част

3.2 Практическа част

3.3 Приложение

Дипломант :...........................................

/ Денислав Ивайлов Иванов /

Ръководител:..........................................

/ Александър Ангелов /

Директор:...............................................

/ доц. д-р инж. Ст. Стефанова /

**Мнение на дипломен ръководител**

УВОД

3D двигателят (3D engine) е система, използвана за създаване на виртуални компютърни симулации, която се използва най-често за разработката на видео игри. Такива компютърни програми се наричат още платформи за създаване на 2D и 3D видеоигри. В зависимост от стила и жанра на играта, се формира графика и се вграждат компоненти и различни графични обекти и алгоритми за извършване на определени действия. В много случаи 3D двигателите предоставят набор от инструменти за визуална разработка в допълнение към софтуерни компоненти за многократна употреба. Тези инструменти обикновено се предоставят в интегрирана среда за разработка, за да позволят опростено и бързо разработване на игри. Първите 3D двигатели представлявали прости програми за запис на информацията на пиксели във видео буферите на операционната система на машините, на които работели. В днешно време двигателите стават все по-сложи и по-сложни. Резолюцията и детайлността стават все по-високи, набляга се на повече ефекти и възможности и съответно програмите и езикът, с който се компилират стават по-сложни с много повече термини и функционални характеристики.

Главната цел на един 3D двигател е изобразяването на сцени и съпътстващите ги изчисления с акцент върху производителността.

Целта на дипломната работа е да се създаде 3D графичен двигател под формата на Native Windows приложение, който визуализира 3D сцени в реално време. Приложението има способността да зарежда 3D, както и 2D, обекти и сцени и позволява на потребителя да ги обхожда с помощта на клавиатурата и мишката.

# Първа глава

Преглед на използвани технологии и

подобни решения

**1.1 Преглед на използвани технологии**

1.1.1 C++



**Фиг. 1.1**

С++ е език за програмиране от високо ниво, използван за разработката на операционни системи, браузъри, игри, големи програми, както и други системи, където производителността е от особено значение.

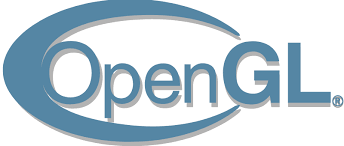
Датският програмист Бярне Строуструп разработва C++ през 1983 г. в Лабораториите „Бел“ като разширение на езика C – езикът е базиран на C, но в него са добавени редица допълнителни възможности и са направени няколко промени. Основната разлика между C и C++ е, че C++ съдържа вградена в езика поддръжка на обектно ориентирано програмиране. В C++ са добавени класове, множествено наследяване, виртуални функции, overloading, шаблони (templates), обработка на изключения (exceptions) и вградени оператори за работа с динамична памет. Езиковият стандарт на C++ е ратифициран през 1998 като *ISO/IEC 14882:1998*, през 2003 година има преразглеждане на стандарта – *ISO/IEC 14882:2003*, а от 2011 стандартът се обновява на всеки три години. Последната ревизия е от 2020г.

Повечето програми на C могат директно или със съвсем малки модификации да бъдат компилирани с компилатор за езика C++. Поради заимстване на множество концепции от C++ езикът Java има много общи черти със C++.

Езикът C++ има два основни компонента: директен достъп до хардуера, осигурен предимно от подмножеството C, и абстракции с нулеви надписи, базирани на тези съпоставяния. Строуструп описва C++ като "лек език за програмиране, изграждане и използване на ефективни и елегантни абстракции"; и "предлагането както на хардуерен достъп, така и на абстракция е основата на C++. Правейки това ефективно го отличава от други езици".

Една голяма част от приложните програми на много операционни системи, както и някои от самите операционни системи, са написани на този език.

1.1.2 OpenGL



**Фиг. 1.2**

**OpenGL** (**Open Graphics Library**) е мощен приложно-програмен интерфейс (API) за реализиране на лесно преносими графични приложения. Създаден през 1992 г., OpenGL бързо става един от най-популярните програмни интерфейси за реализиране на 2D и 3D графики. За това допринасят широката му достъпност, съвместимостта му с различни операционни системи и с различни компютърни платформи. Тази библиотека е подходяща за приложения, изискващи високо качество на изображението, комбинирано с добра производителност, за да бъде възможно използването му в реално време.

Спецификацията на OpenGL описва абстрактен API за визуализиране на 2D и 3D графики. Въпреки че е възможно API-ът да бъде имплементиран изцяло чрез софтуер, той е проектиран да бъде приложен предимно или изцяло чрез хардуер.

API се дефинира като набор от функции, които могат да бъдат извикани от клиентската програма, заедно с набор от наименувани целочислени константи (например константата GL\_TEXTURE\_2D, която съответства на десетичното число 3553). Въпреки че дефинициите на функциите са повърхностно подобни на тези на езика за програмиране C, те са независими от него. Това позволява

на OpenGL да бъде използван от множество езици: някои от най-забележителните са JavaScript чрез WebGL (API, базиран на OpenGL ES 2.0, за 3D изобразяване през браузъра); C връзките WGL, GLX и CGL; Обвързването с езика С, предоставено от iOS; и чрез интерфейсите в Java и C, предоставени от Android.

Освен че е независим от езика, OpenGL е и кросплатформен. Спецификацията не казва нищо по въпроса за получаване и управление на контекст на OpenGL, оставяйки това на основната система за прозорци. По същата причина OpenGL се занимава изцяло с изобразяването, като не предоставя API за други входно-изходни устройства (пр. клавиатура, мишка, аудио).

OpenGL е активно разработван API. Нови версии на спецификациите на OpenGL се пускат редовно от Khronos Group, всяка от които разширява API-а, за да поддържа различни нови функции. Подробностите за всяка версия се решават с консенсус между членовете на Khronos Group, включително производителите на графични карти, дизайнерите на операционни системи и общи технологични компании като Mozilla и Google. В допълнение към функциите, изисквани от основния API, доставчиците на графични процесори (GPU) могат да предоставят допълнителна функционалност под формата на разширения. Разширенията могат да въвеждат нови функции и нови константи и могат да облекчат или премахнат ограниченията върху съществуващите функции на OpenGL. Доставчиците могат да използват разширения за излагане на персонализирани API, без да се нуждаят от поддръжка от други доставчици или Khronos Group като цяло, което значително увеличава гъвкавостта на OpenGL. Всички разширения се събират и дефинират от OpenGL регистъра.

Всяко разширение е свързано с кратък идентификатор, базиран на името на компанията, която го е разработила. Например, идентификаторът на Nvidia е NV, който е част от името на разширението GL\_NV\_half\_float, константата GL\_HALF\_FLOAT\_NV и функцията glVertex2hNV(). Ако множество доставчици се съгласят да внедрят една и съща функционалност, използвайки един и същ API, споделено разширение може да бъде пуснато, използвайки идентификатора EXT. В такива случаи може да се случи също така, че Съветът за преглед на архитектурата на Khronos Group да даде изричното си одобрение за разширението, в който случай се използва идентификаторът ARB. Функциите, въведени от всяка нова версия на OpenGL, обикновено се формират от комбинираните характеристики на няколко широко внедрени разширения, особено разширения от тип ARB или EXT.

**1.2 Преглед на подобни решения**

1.2.1 Unreal3D



**Фиг. 1.3**

1.2.1.1 Въведение

Unreal3D е игрови двигател, разработен от Epic Games. За пръв път е демонстриран през 1998 година в играта Unreal. Въпреки че е разработен предимно за FPS

(First Person Shooter) игри, той е използван успешно и в редица игри от различни жанрове като Stealth, MMORPG и RPG. Софтуерният код на Unreal Engine е написан на езика C++, което му осигурява много висока степен на портативност и производителност. Днес се използва от много и различни разработчици на игри. Текущата версия на софтуера е Unreal Engine 4, предназначена за технологията DirextX 11 и 12 на Microsoft (за Microsoft Windows, Xbox One и Windows RT); OpenGL (за MAC OS X, Linux, PlayStation 4, iOS, Android, Ouya и Windows XP) и JavaScript/WebGL (за уеб браузъри, поддържащи HTML 5).

Сред характеристиките на Unreal3D са collision detection, цветно осветление и ограничена форма на филтриране на текстури. Двигателят също така интегрира редактор на нива, UnrealEd, който имаше поддръжка за операции с конструктивна твърда геометрия в реално време още през 1996 г., позволявайки на картографите да променят оформлението на нивата в движение. Въпреки че Unreal е проектиран да се конкурира с id Software (разработчикът на Doom и Quake), съоснователят Джон Кармак похвали играта за използването на 16-битов цвят и отбеляза нейното прилагане на визуални ефекти като обемна мъгла. "Съмнявам се, че някоя важна игра ще бъде проектирана с 8-битов цвят оттук нататък. Unreal направи важно нещо в насочването към директен цвят и това дава на художниците много повече свобода", каза той в статия, написана от Джеф Кийли за GameSpot. „Светлинните цъфтежи [сферите на светлината], обемите на мъглата и композитните небеса бяха стъпки, които планирах да предприема, но Epic стигна дотам първи с Unreal“, казва той и добавя: „Двигателят на Unreal вдигна летвата за това, което екшън геймърите очакват от бъдещи продукти. Визуалните ефекти, които се виждат за първи път в играта, ще се очакват от бъдещите игри."

1.2.1.2 История

Първото поколение на двигателя е разработено от Тим Суиини, основателят на Epic Games. Суиини започва работа по двигателя през втората половина на 1995 година по време на продукцията на играта Unreal.

След години на разработка двигателят прави своя дебют през 1998 година с излизането на играта на пазара. Първото поколение Unreal Engine интегрира рендериране, collision detection, изкуствен интелект, видимост и полезрение, работа в мрежата, скриптиране, и управление на файлова система в един цялостен двигател.

По време на интервю със списание Maximum PC, създателят на Epic Games – Тим Суини заявява, че от самото начало двигателят е проектиран така, че да бъде лесно разширяван и подобряван за бъдещи игри.

**Unreal 1**

* Unreal Engine 1 предоставя напреднал софтуерен растеризатор и хардуерно-ускорен рендъринг използвайки Glide API, специално разработен за 3dfx графични процесори. По-късно е добавена поддръжка за OpenGL и Direct3D. Значителни части от играта са имплементирани чрез вградения скриптов език, наречен UnrealScript. Epic използват двигателя за Unreal и Unreal Tournament. С излизането на Unreal Tournament се отбелязва голям напредък в мрежовата производителност и поддръжката на Direct3D и OpenGL. Двигателят става популярен благодарение на модулната си архитектура и вградения скриптов език, който улеснява модифицирането. Tactical Ops е игра, започнала като мод за Unreal Tournament и превърнала се в самостоятелно заглавие след цялостна модификация.

**Unreal 2**

* Това поколение отбеляза значителен напредък в условията на изобразяване, както и нови подобрения на набора от инструменти. Способен да изпълнява нива, почти 100 пъти по-подробни от тези в Unreal 1.0 двигателят интегрира различни функции, включително инструмент за кинематографично редактиране, системи за частици, експортни плъгини за 3D Studio Max и Maya и система за скелетна анимация, представена за първи път във версията за PlayStation 2 на Unreal Tournament. В допълнение, потребителският интерфейс за UnrealEd бива пренаписан на C++ с помощта на инструментариума wxWidgets, който според Суини е „най-доброто налично“ по това време.

**Unreal 3**

* Първоначално Unreal Engine 3 е наличен само за платформите Windows, PlayStation 3 и Xbox 360, докато iOS (първо демонстрирано с Epic Citadel) и Android биват добавени по-късно през 2010 г., като Infinity Blade е първото заглавие за iOS, а Dungeon Defenders първото заглавие за Android. През 2011 г. е обявено, че двигателят ще поддържа Adobe Flash Player 11 чрез Stage 3D хардуерно ускорени API и че се използва в две игри на Wii U, Batman: Arkham City и Aliens: Colonial Marines. През 2013 г. Epic си партнира с Mozilla, за да внесе Unreal Engine 3 в мрежата; използвайки подезика asm.js и компилатора Emscripten, те успяват да пренесат двигателя за четири дни.

**Unreal 4**

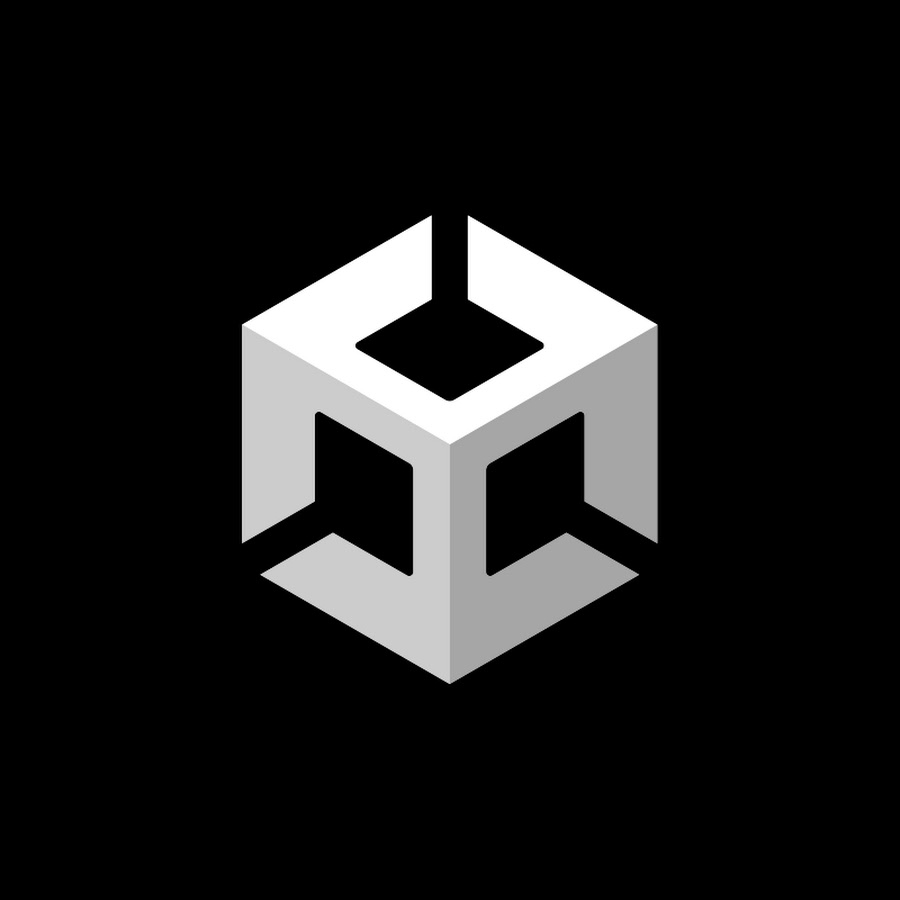
* Една от основните функции, планирани за UE4, е глобалното осветление в реално време, използващо проследяване на вокселен конус, елиминирайки предварително изчисленото осветление. Тази функция обаче, наречена Sparse Voxel Octree Global Illumination (SVOGI) и демонстрирана с Elemental, бива заменена с подобен, но по-малко скъп алгоритъм поради проблеми с производителността. UE4 включва и новата система за визуални скриптове "Blueprints" (наследник на "Kismet" на UE3), която позволява бързо развитие на логиката на играта без използване на код, което води до по-малко разделение между дизайнери и програмисти.

**Unreal 5**

* Една от основните му характеристики е Nanite, двигател, който позволява в игрите да бъде импортиран високодетайлен фотографски изходен материал. Технологията за виртуализирана геометрия Nanite позволява на Epic да се възползва от предишното си придобиване на Quixel, най-голямата библиотека за фотограметрия в света от 2019 г. Целта на Unreal Engine 5 е да улесни разработчиците възможно най-лесно да създават подробни игрови светове, без да се налага да прекарват прекалено много време за създаване на нови подробни активи. Nanite може да импортира почти всяко друго съществуващо триизмерно представяне на обекти и среди, включително ZBrush и CAD модели, което позволява използването на активи с качество на филма. Nanite автоматично обработва нивата на детайлност ( LODs) на тези импортирани обекти, подходящи за целевата платформа и изчертаване на разстояние, задача, която художникът би трябвало да изпълни иначе. Lumen е друг компонент, описан като „напълно динамично глобално решение за осветяване, което незабавно реагира на промените на сцената и светлината“. Lumen елиминира необходимостта художници и разработчици да изработят светлинна карта за дадена сцена, но вместо това изчислява светлинните отражения и сенките в движение, като по този начин позволява поведението на източниците на светлина в реално време. Virtual Shadow Maps е друг компонент, добавен в Unreal Engine 5, описан като „нов метод за картографиране на сенките, използван за осигуряване на последователно засенчване с висока разделителна способност, което работи с активи с филмово качество и големи, динамично осветени отворени светове“.

Виртуалните карти на сянка се различават от обичайната реализация на карта на сянка по своята изключително висока разделителна способност, по-детайлни сенки и липсата на изскачащи и излизащи сенки, които могат да бъдат намерени в по-често срещаната техника на карти на сянка поради каскади на сенките. Допълнителните компоненти включват Niagara за динамика на течности и частици и Chaos за физичен двигател

1.2.2 Unity



**Фиг 1.4**

1.2.2.1 Въведение

Unity е мултиплатформен игрови двигател, разработен от Unity Technologies, за първи път обявен и пуснат през юни 2005 г. на световната конференция на разработчиците на Apple Inc. като ексклузивен двигател за Mac OS X. Оттогава двигателят постепенно се разширява, за да поддържа различни настолни, мобилни, конзолни и платформи за виртуална реалност. Той е особено популярен за разработка на мобилни игри за iOS и Android и се използва за игри като Pokémon Go, Monument Valley, Call of Duty: Mobile, Beat Sabre и Cuphead. Смята се, че е лесен за използване от начинаещи разработчици и е популярен за разработка на инди игри.

Двигателят може да се използва за създаване на триизмерни (3D) и двуизмерни (2D) игри, както и интерактивни симулации и други преживявания. Unreal3D е възприет от индустрии извън видео игрите, като филми, автомобилостроене, архитектура, инженерство, строителство и въоръжените сили на Съединените щати.

1.2.2.2 История

Игровият двигател Unity стартира през 2005 г., целящ да „демократизира“ разработването на игри, като го прави възможно за повече разработчици да го използват. Следващата година Unity беше избран за второ място в категорията за най-добро използване на графиката на Mac OS X в наградите за дизайн на Apple Inc. на Apple. Първоначално Unity е пуснат за Mac OS X, по-късно Unreal добавят поддръжка за Microsoft Windows и уеб браузъри.

### Unity 2.0 (2007)

* Добавен е оптимизиран терен за детайлни 3D среди, динамични сенки в реално време, насочени светлини и прожектори, възпроизвеждане на видео и други функции.

### Unity 3.0 (2010)

* В допълнение към поддръжката на Android, Unity 3 включва интеграция на инструмента Beast Lightmap на Illuminate Labs, отложено изобразяване, вграден редактор на дървета, собствено изобразяване на шрифтове, автоматично UV картографиране и аудио филтри, наред с други неща.

### Unity 4.0 (2012)

* Тази версия добавя поддръжка на DirectX 11 и Adobe Flash, нови инструменти за анимация, наречени Mecanim, и достъп до визуализацията на Linux

### Unity 5.0

* С Unity 5 двигателят подобрява осветлението и звука. Чрез WebGL разработчиците на Unity могат да добавят своите игри към съвместими уеб браузъри, без да се изискват добавки за играчите. Unity 5.0 предлага глобално осветление в реално време, визуализации на светлинните карти, Unity Cloud, нова аудио система и физическия двигател на Nvidia PhysX 3.3. Петото поколение на двигателя на Unity също въведе кинематографични ефекти на изображението, за да помогне на игрите на Unity да изглеждат по-малко общи. Unity 5.6 добави нови ефекти на осветление и частици, актуализира цялостната производителност на двигателя и добави естествена поддръжка за Nintendo Switch, Facebook Gameroom, Google Daydream и Vulkan graphics API.

Втора глава

Oписание на функционалните изисквания и структура на приложението

**2.1 Функционални изисквания**

**2.1.1 Графичен engine, който визуализира 3D сцени, базиран на C++ и OpenGL**

Native Windows приложение, написано на С++ и използващо графичния API – OpenGL. Приложението трябва да има способността да зарежда 3D и 2D сцени в .GLTF формат и да ги визуализира върху екрана.

**2.1.2 Подвижна камера, която обхожда сцената**

Приложението трябва да съдържа виртуална камера, която да позволява на потребителя да обхожда 3D пространството, в което се намира сцената, с помощта на клавиатурата и мишката.

**2.1.3 Поддръжка на Anti aliasing**

Приложението трябва да поддържа MSAA (**Multi-Sample Anti-Aliasing**) тип anti aliasing.

**2.1.4 Поддръжка на 2D модели**

Приложението трябва да поддържа както 3D, така и 2D модели.

**2.2 Избор на програмни средства и** **развойна среда**

**2.2.1 C++**

За език за програмиране е избран C++, защото се преобразува директно до машинен код, което го прави изключително бърз. Поддържа обектно-ориентиран и императивен стил на програмиране, което позволява използването на по-гъвкави методи на разработка. Бързината на C++ в комбинация с гъвкавата му природа го правят перфектния език за разработката на 3D двигател.

**2.2.1 GLFW**

GLFW е лека помощна библиотека, която перфектно допълва OpenGL. GLFW е съкращение от Graphics Library Framework. GLFW предоставя на програмистите възможността да създават и управляват прозорци и OpenGL контекстове, както и да управляват въвеждането с джойстик, клавиатура и мишка.

GLFW е малка C библиотека, която позволява създаването и управлението на прозорци с OpenGL контексти, което прави възможно използването на множество монитори и видео режими. Той осигурява достъп до въвеждане от клавиатурата, мишката и джойстиците. API осигурява тънък, мултиплатформен абстракционен слой, предимно за приложения, чийто единствен графичен изход е чрез OpenGL API. Докато GLFW е много полезен при разработването на многоплатформени OpenGL приложения, разработчиците на една платформа също могат да се възползват от избягването на работа с различни специфични за платформата API. Възможна причина, поради която библиотеки като GLFW са необходими, е, че OpenGL сам по себе си не предоставя никакви механизми за създаване на необходимия контекст, управление на прозорци, въвеждане от потребителя, време и т.н. Има няколко други налични библиотеки за подпомагане на OpenGL разработването.

Най-често срещаните са FreeGLUT (реализация с отворен код на GLUT) и SDL. Въпреки това, FreeGLUT се занимава най-вече с осигуряването на стабилен клонинг на GLUT, докато SDL е твърде голям за някои хора и никога не е имал OpenGL като основен фокус. GLFW се основава върху идеята за лека, модерна библиотека за управление на OpenGL контексти, прозорци и вход.

**2.2.1 GLEW**



**Фиг 2.1 Лого**

OpenGL Extension Wrangler Library (GLEW) е междуплатформена библиотека за зареждане на C/C++ разширения за OpenGL с отворен код. GLEW предоставя ефективни механизми по време на изпълнение за определяне кои OpenGL разширения се поддържат на целевата платформа. Функционалността на ядрото и разширението на OpenGL са изложени в един заглавен файл. GLEW е тестван на различни операционни системи, включително Windows, Linux, Mac OS X, FreeBSD, Irix и Solaris.

**2.2.2 GLM**



**Фиг. 2.2 Лого**

OpenGL Mathematics (GLM) е само заглавна математическа библиотека на C++ за графичен софтуер, базирана на спецификациите на OpenGL Shading Language (GLSL). GLM предоставя класове и функции, проектирани и внедрени със същите конвенции за именуване и функционалност като GLSL, така че всеки, който познава GLSL, може да използва GLM и в C++. Система за разширение, базирана на конвенциите за разширение GLSL, предоставя разширени възможности: матрични трансформации, кватерниони, пакетиране на данни, произволни числа, шум и т.н.

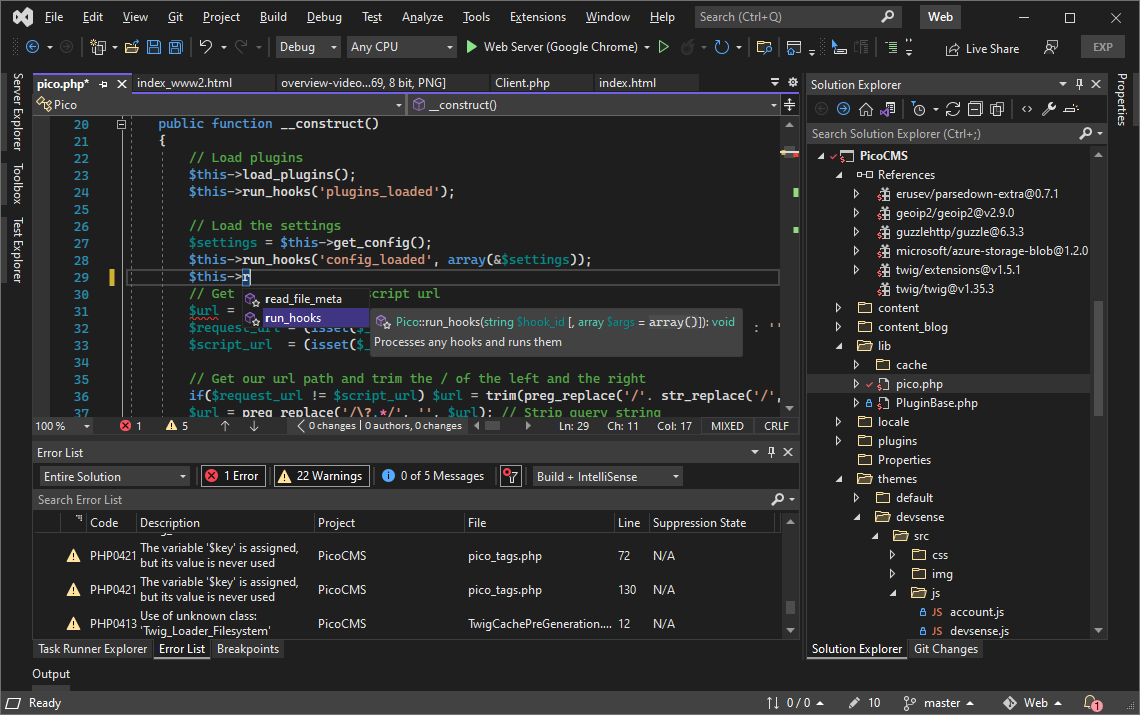
Библиотеката е направена да работи с OpenGL, но също така гарантира оперативна съвместимост с други библиотеки на трети страни и SDK (software development kit). Тя е добър кандидат за софтуерно изобразяване (проследяване на лъчи / растеризация), обработка на изображения, симулации на физика и всеки контекст на разработка, който изисква проста и удобна математическа библиотека.

**2.2.3 STB\_IMAGE.H**

Stb библиотеките са колекция от еднофайлови библиотеки с заглавни файлове за C/C++ в публичното пространство, насочени предимно към разработчиците на игри. Те са проектирани така, че да бъдат лесни за интегриране, лесни за използване и лесни за освобождаване. Някои библиотеки са обикновени (зареждане на файлове с изображения), други са дълбоко езотерични (генериране на ниво). Stb\_image.h се използва за зареждането на изображения от форматите: JPG, PNG, TGA, BMP, PSD, GIF, HDR, PIC в буфер от байтове. Идеята зад файловите библиотеки с един заглавен файл е, че те са лесни за разпространение и внедряване, тъй като целият код се съдържа в един файл. По подразбиране .h файловете действат като свои собствени заглавни файлове, т.е. те декларират функциите, съдържащи се във файла, но всъщност не водят до компилиран код. Така че в допълнение, трябва да изберете точно един изходен файл на C/C++, който всъщност инстанциира кода, за предпочитане файл, който не бива редактиран често. Този файл трябва да дефинира специфично макро (това е документирано за всяка библиотека), за да активира дефинициите на функциите. Например, за да използвате stb\_image, трябва да имате точно един C/C++ файл, който не включва stb\_image.h редовно, но вместо това го добавя по следния начин:



Фиг. 2.3 stb\_image.h

**2.2.4 Visual Studio**

**Фиг. 2.4 Visual Studio**

Microsoft Visual Studio е интегрирана среда за разработка (IDE), създадена от Microsoft. Използва се за разработване на компютърни програми, както и на уебсайтове, уеб приложения, уеб услуги и мобилни приложения. Visual Studio използва платформи за разработка на софтуер на Microsoft като Windows API, Windows Forms, Windows Presentation Foundation, Windows Store и Microsoft Silverlight. Той може да произвежда както собствен код, така и управляван код. Visual Studio включва редактор на код, поддържащ IntelliSense (компонентът за завършване на код), както и рефакторинг на код. Интегрираният дебъгер работи както като дебъгер на ниво източник, така и като дебъгер на ниво машина. Други вградени инструменти включват инструмент за профилиране на код, дизайнер за изграждане на GUI приложения, уеб дизайнер, дизайнер на класове и дизайнер на схеми на база данни. Той приема добавки, които разширяват функционалността на почти всяко ниво – включително добавяне на поддръжка за системи за контрол на източници (като Subversion и Git) и добавяне на нови набори от инструменти като редактори и визуални дизайнери за специфични за домейна езици или набори от инструменти за други аспекти на разработката на софтуер жизнен цикъл (като клиента на Azure DevOps: Team Explorer). Visual Studio поддържа 36 различни езика за програмиране и позволява на редактора на код и дебъгера да поддържат (в различна степен) почти всеки език за програмиране, при условие че съществува специфична за езика услуга. Вградените езици включват C, C++, C++/CLI, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML и CSS. Поддръжката за други езици като Python, Ruby, Node.js и M, наред с други, е достъпна чрез плъгини.

Java (и J#) са се поддържали в миналото. Най-основното издание на Visual Studio, изданието за общността (Community Edition), е достъпно безплатно.

Visual Studio перфектно пасва на изискванията на дипломната работа с

неговата естествена поддръжка за операционната система Windows и вградения C++ компилатор.

**2.3 Описание на структурата на приложението**

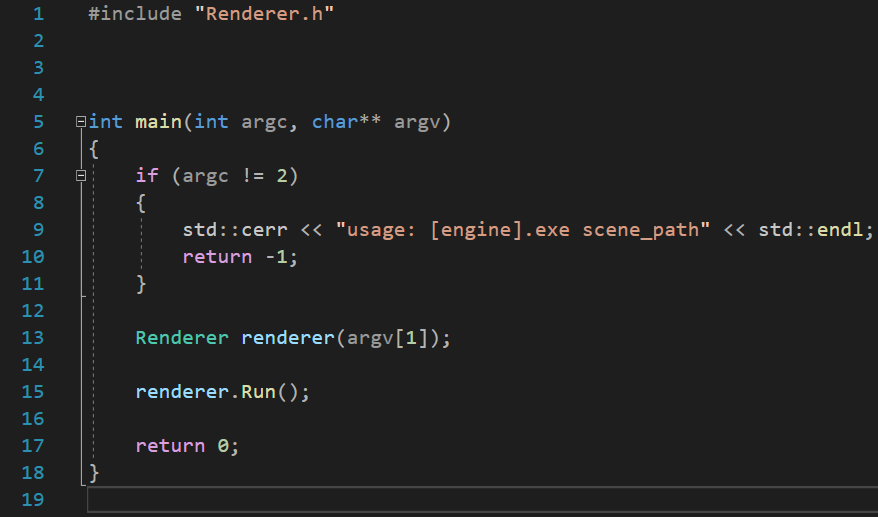
Приложението представлява изпълним файл за операционната система Windows, който се изпълнява през развойната среда. Двигателят приема папката на сцената/модела във файловата система като аргумент от командния ред и го визуализира върху прозореца на приложението.

# Трета глава

Програмна реализация на приложението

**3.1 Реализация на програмата**

**3.1.1 Началото**



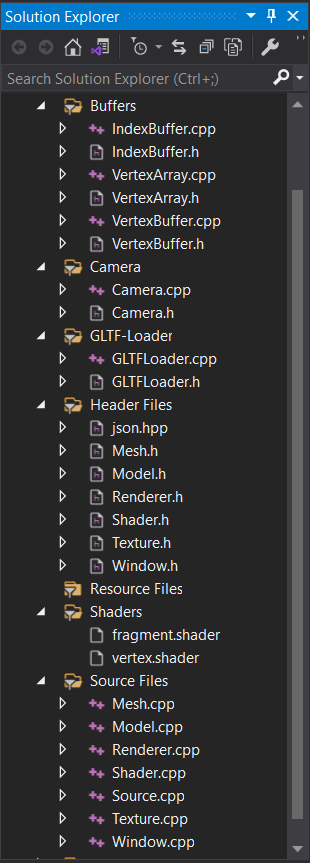
**Фиг. 3.1 Началото на програмата**

Всичко започва с “int main()” – началната точка на всяка C/C++ програма.

Функцията проверява за броя на подадените аргументи от командния ред и известява потребителя при несъответствие.

След което създава инстанция на клас “Renderer” с пътя към сцената като аргумент. След инициализацията на “Renderer” класа се извиква “Run()” метода, който задейства изпълнението на 3D двигателя.

**3.1.2 Файлова структура на приложението**

****

**Фиг. 3.2 Структура на проекта**

**3.1.3 OpenGL**

**3.1.3.1 Shaders**

Shader е дефинирана от потребителя програма, предназначена да се изпълнява на някакъв етап от графичния процесор. Шейдърите предоставят кода за определени програмируеми етапи на тръбопровода за изобразяване (graphics pipeline). Те могат да се използват и в малко по-ограничена форма за общи изчисления на GPU-то.

Конвейерът за изобразяване (rendering pipeline) дефинира определени секции да бъдат програмируеми. Всеки от тези раздели или етапи представлява определен тип програмируема обработка. Всеки етап има набор от входове и изходи, които се предават от предишни етапи и към следващите етапи (независимо дали са програмируеми или не). Шейдърите са написани на OpenGL Shading Language. Конвейерът за изобразяване на OpenGL дефинира следните етапи на шейдъра с тяхното име на изброител: Верхови шейдъри: GL\_VERTEX\_SHADER Шейдъри за управление и оценка на теселация: GL\_TESS\_CONTROL\_SHADER и GL\_TESS\_EVALUATION\_SHADER. (изисква GL 4.0 или ARB\_tessellation\_shader) Геометрични шейдъри: GL\_GEOMETRY\_SHADER Фрагментни шейдъри: GL\_FRAGMENT\_SHADER Изчисляване на шейдъри: GL\_COMPUTE\_SHADER. (изисква GL 4.3 или ARB\_compute\_shader) Програмният обект може да комбинира множество етапи на шейдър (изградени от шейдърни обекти) в едно, свързано цяло. Обектът на конвейера на програмата може да комбинира програми, които съдържат отделни етапи на шейдъра, в цял конвейер. Докато етапите на шейдъра използват един и същ език, всеки етап има отделен набор от входове и изходи, както и вградени променливи. Като такива, шейдърните обекти са създадени за конкретен етап на шейдъра. Така че докато програмните обекти могат да съдържат множество етапи, шейдърните обекти съдържат код само за един етап.

За създаването на шейдър се използва следната функция:

glCreateShader(GLuint type), където “type” е типа на шейдъра.

Компилацията се извършва чрез следните функции:

glShaderSource(GLuint shader, GLsizei count,

const GLchar \*\*string,

const GLint \*length),

glCompileShader(GLuint shader\_id)

които добавят сорс код и компилират шейдъра респективно.

**3.1.3.2 Буфери**

**3.1.3.2.1 Vertex Buffer Object**

Обект на върхов буфер (VBO) е функционалност на OpenGL, която предоставя методи за качване на върхови данни (позиция, нормален вектор, цвят и т.н.) към видео устройството за изобразяване в не-непосредствен режим. VBO предлагат значителни печалби в производителността спрямо изобразяването в непосредствен режим главно защото данните се намират в паметта на видеоустройството, а не в системната памет, и така могат да бъдат изобразени директно от видео устройството. Те са еквивалентни на върховите буфери в Direct3D. Спецификацията на обекта на буфера на върховете е стандартизирана от Съвета за преглед на архитектурата на OpenGL от версия 1.5 на OpenGL (през 2003 г.). Подобна функционалност беше налична преди стандартизирането на VBO чрез създаденото от Nvidia разширение „vertex array range“ или разширението на ATI „vertex array object“.

Следните функции се използват за създаването и използването на върховите буфери:

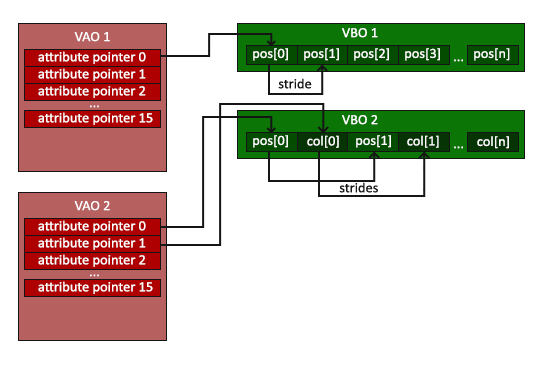
glGenBuffers(sizei n, uint\* buffers)

glBindBuffer(GLenum target, GLuint buffer)

glBufferData(GLenum target, GLsizeiptr size, const void\* data, GLenum usage)

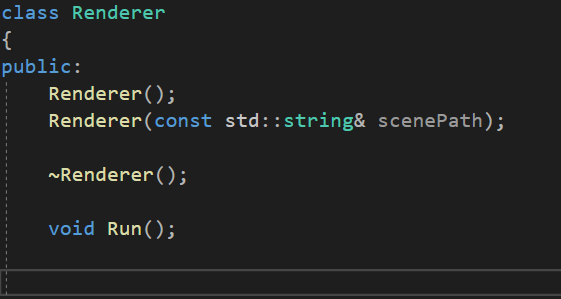
**3.1.3.2.2 Vertex Array Object**

Обект на масив от върхове (VAO) е OpenGL обект, който съхранява цялото състояние, необходимо за предоставяне на данни за върхове (с едно незначително изключение, отбелязано по-долу). Той съхранява формата на данните за върховете, както и буферните обекти (вижте по-долу), осигуряващи масивите от данни за върховете. Обърнете внимание, че VAO не копират, замразяват или съхраняват съдържанието на посочените буфери - ако промените някои от данните в буферите, на които се позовава съществуващ VAO, тези промени ще бъдат видени от потребителите на VAO. Като OpenGL обекти, VAO имат обичайните функции за създаване, унищожаване и свързване: glGenVertexArrays, glDeleteVertexArrays и glBindVertexArray. Последното е различно, тъй като няма "целеви" параметър; има само една цел за VAO и glBindVertexArray се свързва с тази цел. Забележка: VAO не могат да се споделят между контекстите на OpenGL. Атрибутите на върховете са номерирани от 0 до GL\_MAX\_VERTEX\_ATTRIBS - 1. Всеки атрибут може да бъде активиран или деактивиран за достъп до масив. Когато достъпът до масива на атрибут е забранен, всяко четене на този атрибут от шейдера на върха ще произведе постоянна стойност (виж по-долу) вместо стойност, извлечена от масив. Новосъздаденият VAO има деактивиран достъп до масив за всички атрибути. Достъпът до масива се активира чрез обвързване на въпросния VAO и извикване: void glEnableVertexAttribArray(GLuint индекс;); Има подобна функция glDisableVertexAttribArray за деактивиране на активиран масив. Запомнете: цялото състояние по-долу е част от състоянието на VAO, освен когато изрично е посочено, че не е. VAO трябва да бъде обвързан при извикване на някоя от тези функции и всички промени, причинени от тази функция, ще бъдат уловени от VAO. Профилът за съвместимост на OpenGL прави VAO обект 0 обект по подразбиране. Основният OpenGL профил прави VAO обект 0 изобщо не обект. Така че, ако VAO 0 е свързан в основния профил, не трябва да извиквате функция, която променя състоянието на VAO. Това включва обвързване на `GL\_ELEMENT\_ARRAY\_BUFFER` с glBindBuffer.

****

**Фиг. 3.3 Структура на буферите в OpenGL**

**3.1.3 Рендърър**

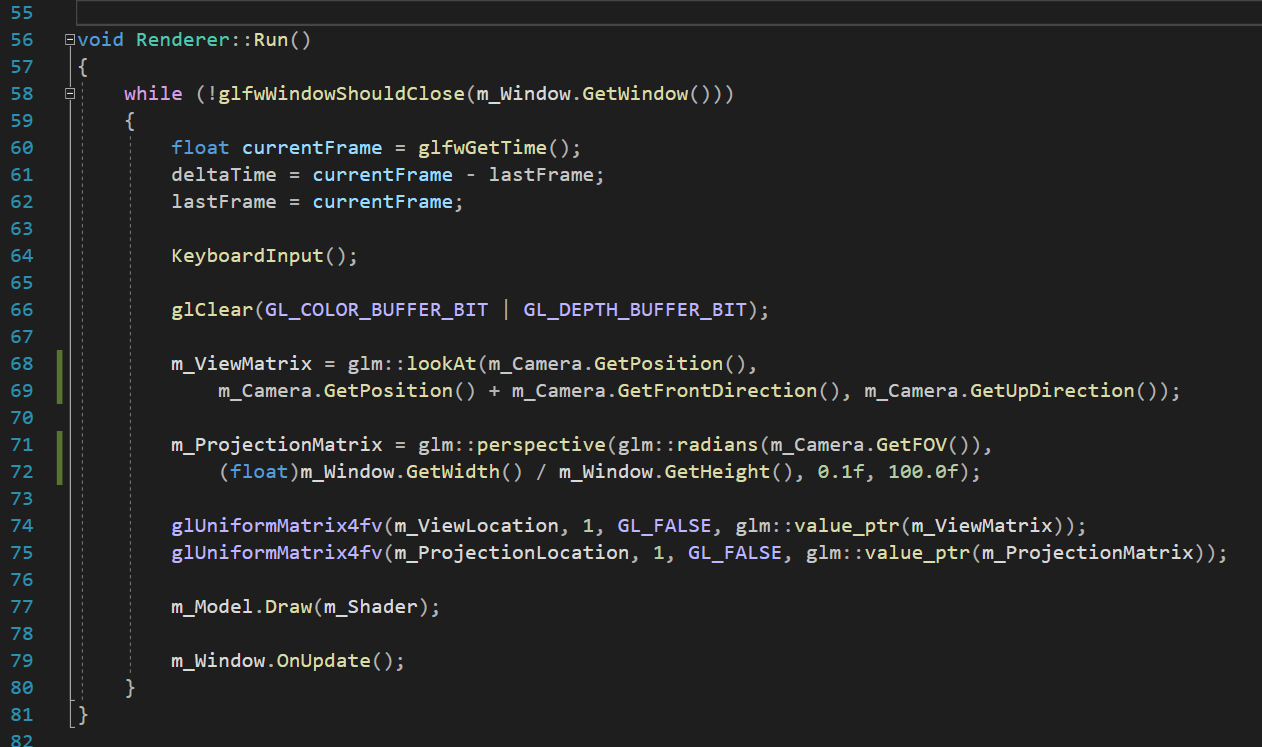


**Фиг 3.4 Публична функционалност на рендъръра**

Рендърърът е сърцето на 3D двигателя. Той се грижи за изобразяването на сцената върху екрана, получаването на входни данни и обработката им.

На **фиг. 3.4** е изобразена публичната функционалност, която рендърърът предоставя на потребителя:

* Renderer(): конструктор по подразбиране, който създава прозорец с размер 1280x1024 и заглавие “OpenGL” и зарежда модела “assets/cube/”
* Renderer(const std::string& path): конструктор, който изпълнява същата функция като този по подразбиране, но приема и път към сцена/модел.
* ~Renderer(): деструктор
* void Run(): метод, който започва изпълнението на двигателя: обновява буферите за рендериране, получава входни данни от потребителя, обновява матриците... Необходим е за правилното изпълнение на програмата.

****

**Фиг. 3.5 Renderer::Run()**

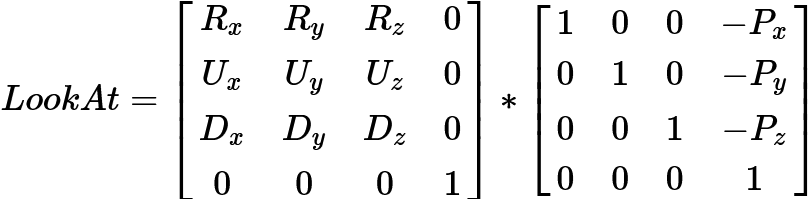
В основата на функцията е “while” цикъл, който проверява дали е подаден сигнал за затваряне на прозореца. Ако не е, се изчислява времето от предишния frame, получават се входни данни от клавиатурата чрез метода “KeyboardInput” , изчистват се цветния и дълбочинния буфер, още известен като Z-буфер, изчисляват се панорамната[1] и проекционната[2] матрица и се изобразява сцената/модела (m\_Model.Draw()).

[1] - Панорамната матрица преобразува координатите от 3D пространството на света в пространството на камерата. Формулата за изчисление е показана на Фиг. 3.7

[2] – Проекционната матрица преобразува координатите от пространството на камерата в пространството на OpenGL (в диапазона от -1 до 1).

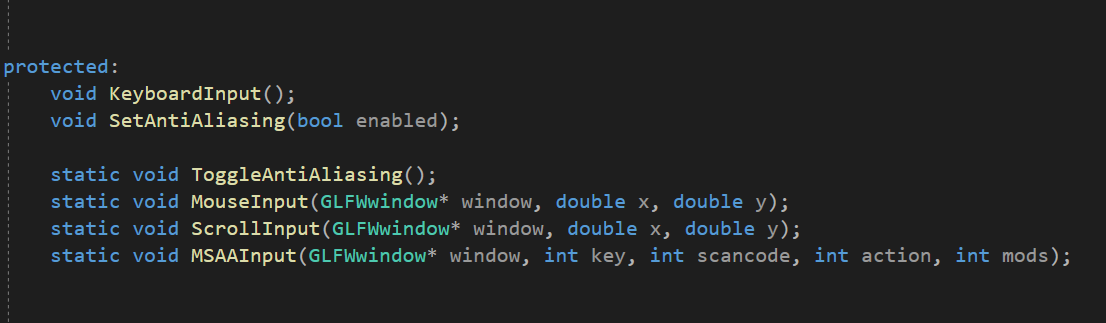


**Фиг 3.6 Панорамна и проекционна матрица**

****

**Фиг 3.7 Панорамна матрица**

**“R” – вектор на дясната посока, “U” – вектор на горната посока, “D” – вектор на посоката на насочване, “P” – вектор на позицията на камерата**

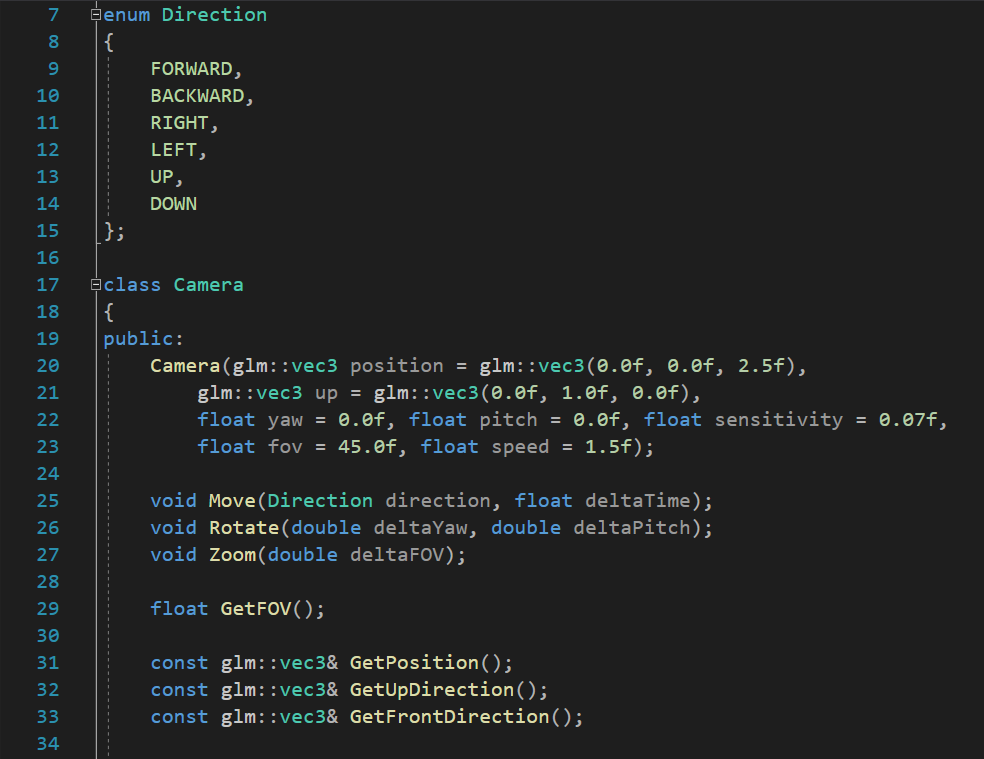
****

**Фиг. 3.8 Защитена функционалност**

**На фиг. 3.8 е показана защитената (protected) функционалност, която рендърърът предоставя на бъдещи класове, които го наследяват:**

* voidKeyboardInput()**:** метод, който се извиква на всеки frame от рендерирането и получава входни данни от клавиатурата
* voidSetAntiAliasing(bool enabled)**:** този метод включва или изключва Anti Aliasing функционалността в зависимост от подадения аргумент.
* static void ToggleAntiAliasing()**:** този метод включва или изключва Anti Aliasing функционалността в зависимост от последното състояние.
* static voidMouseInput(GLFWwindow\* window, double x, double y)**:** методът се извиква от операционната система при промяна в позицията на мишката с новите й координати
* static voidScrollInput(GLFWwindow\* window, double x, double y)**:** методът се извиква от операционната система при промяна в позицията на scroll wheel-a с новите му координати
* static voidMSAAInput(GLFWwindow\* window, int key, int scancode, int action, int mods)**:** методът се извиква при натиск на клавиш от клавиатурата, проверява дали въпросният клавиш е ‘Z’ и ако е**,** включва/изключва Anti Aliasing**.**

**3.1.4 Камерата**

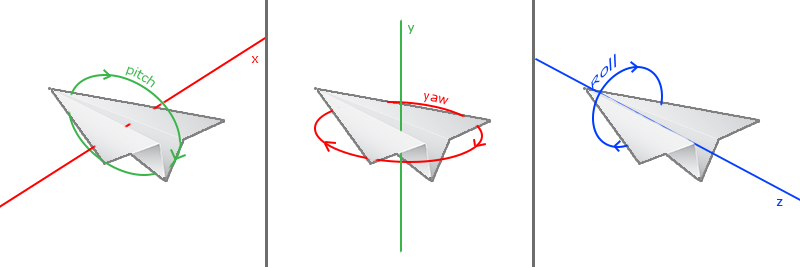
****

**Фиг. 3.9 Публична функционалност на камерата**

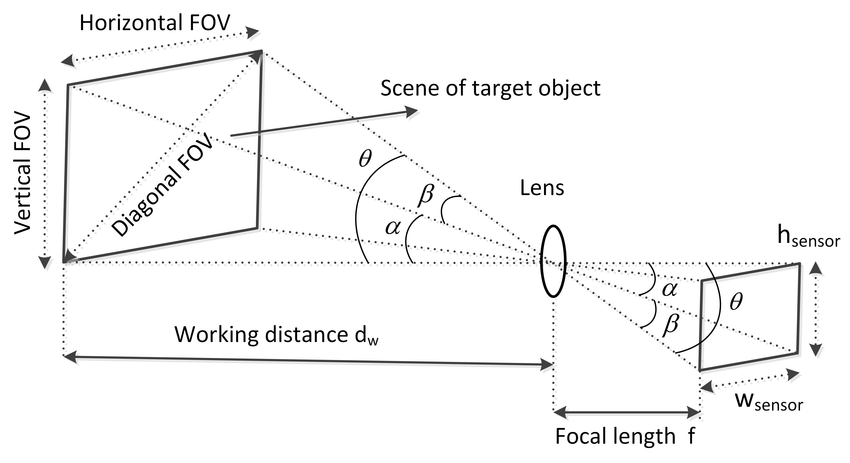
**На фиг. 3.9 е показана публичната функционалност, която камерата предоставя на потребителя:**

* Camera(glm::vec3 position, glm::vec3 up, float yaw, float pitch, float sensitivity, float fov, float speed)**:** конструкторът на класа “Camera” приема списък от аргументи, всеки от които има стойност по подразбиране:

1. **position (X: 0, Y: 0. Z: 2.5):** местоположението на камерата в 3D пространството
2. **up (X: 0, Y: 1, Z: 0)**: векторът, който представлява ординатната ос в координантната система на камерата
3. **Yaw (0 deg):** ъгълът спрямо YZ равнината в пространството
4. **Pitch (0 deg):** ъгълът спрямо XZ равнината в пространството
5. **Sensitivity (0.07)** : чувствителността, с която мишката завърта камерата
6. **FOV (45 deg):** полезрението на камерата в градуси
7. **Speed (1.5)**: скоростта на придвижване на камерата

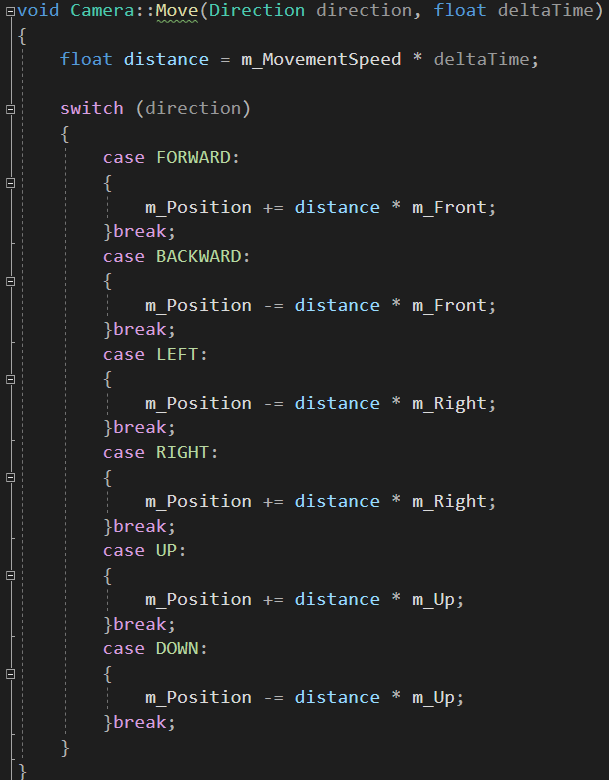
****

**Фиг. 3.10 Ойлеровите ъгли в 3D пространството**

****

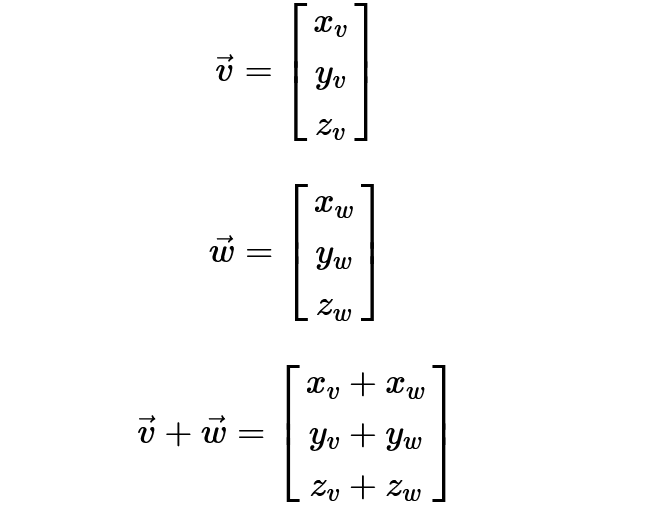
**Фиг. 3.11 FOV (ъгъл на полезрение)**

* voidMove(Direction direction, float deltaTime)**:** този метод приема посока и изминалото време от предишния frame и премества камерата в дадената посока.

****

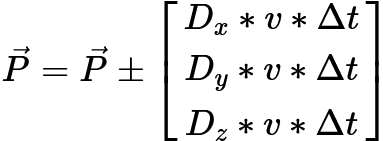
**Фиг. 3.12 Camera::Move()**

**Преместването се осъществява чрез събиране на вектори:**

****

**Фиг. 3.13 Събиране на вектори**

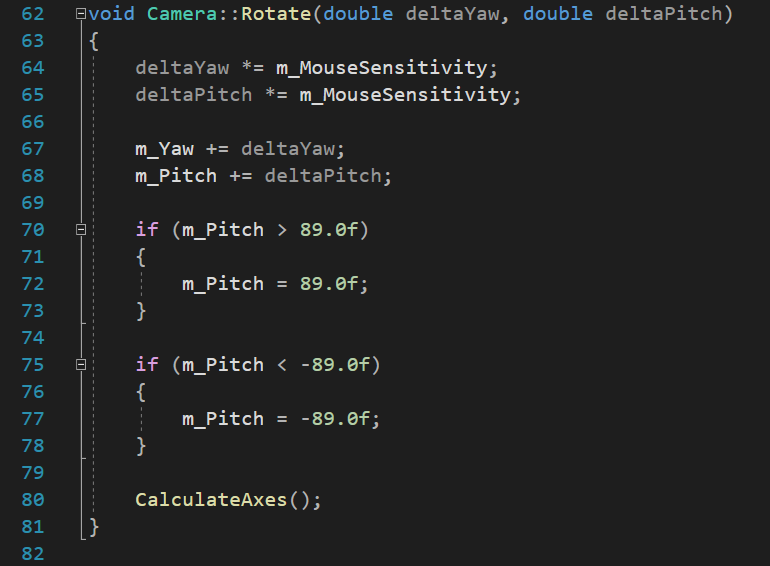
**Изминатото разстояние в дадена посока се изчислява по дадената формула:**

****

**Фиг. 3.14 Обновяване на позицията**

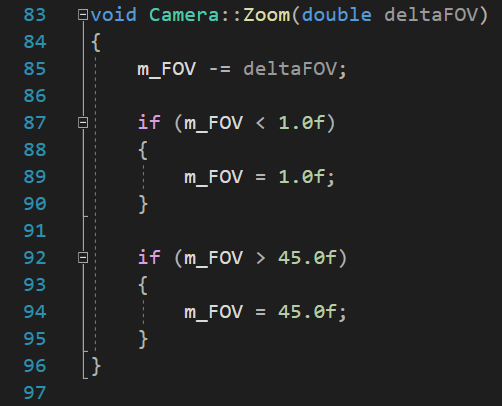
**Където “P” e позицията на камерата , а “D” е векторът на посоката.**

* void Rotate(double deltaYaw, double deltaPitch)**:** този метод приема разликите в ъглите на камерата и я завърта спрямо тях.



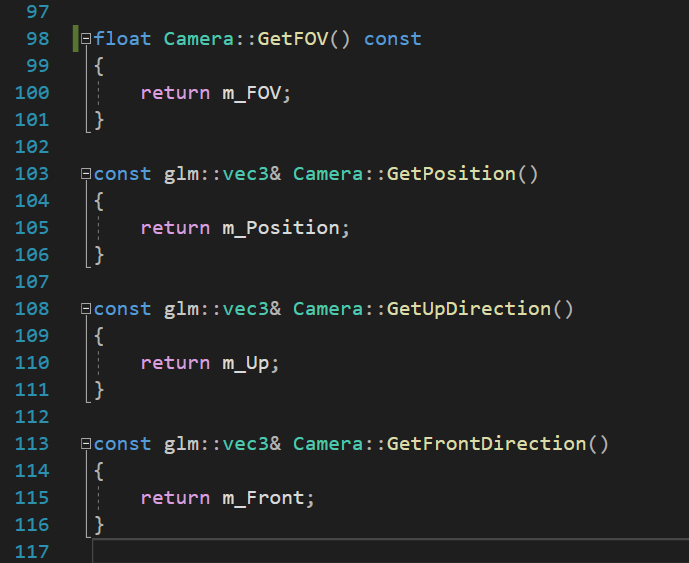
**Фиг. 3.15 Camera::Rotate()**

* void Zoom(double deltaFOV)**:** този метод “приближава“ или “отдалечава“ камерата в зависимост от подадената стойност, като увеличава или намалява ъгъла на полезрението респективно.

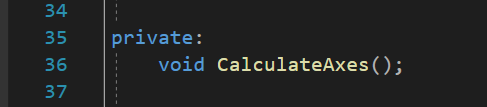
****

**Фиг. 3.16 Camera::GetFov()**

* float GetFOV()**:** “getter” метод, който връща стойността на ъгъла на полезрението
* const glm::vec3& GetPosition()**:** “getter” метод, който връща референция към вектора, който представлява позицията на камерата в 3D пространството.
* const glm::vec3& GetUpDirection()**:**“getter” метод, който връща референция към вектора, който представлява ординатната ос на камерата в 3D пространството.
* const glm::vec3& GetFrontDirection()**:** “getter” метод, който връща референция към вектора, който представлява предната посока на камерата в 3D пространството.

****

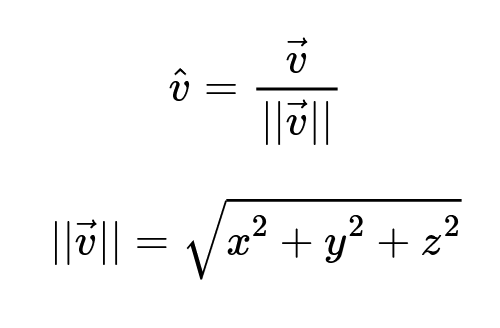
**Фиг. 3.17 Гетъри на класа “Camera”**

****

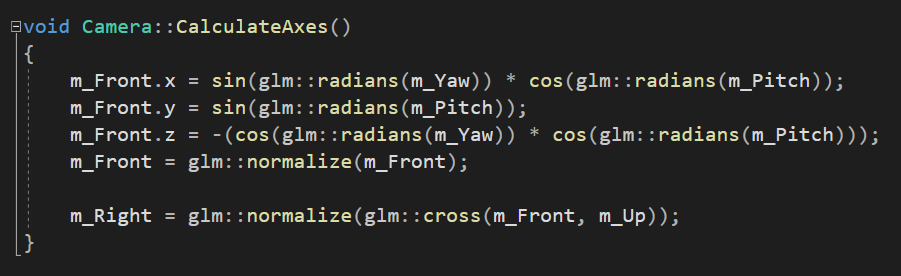
**Фиг. 3.18 Частни функции на класа “Camera”**

**На фиг. 3.18 е показана частната (private) функционалност на класа “Camera”:**

* void CalculateAxes()**:** този метод преизчислява векторите, дефиниращи координатната система на камерата, след което ги нормализира (скалира до дължина единица).



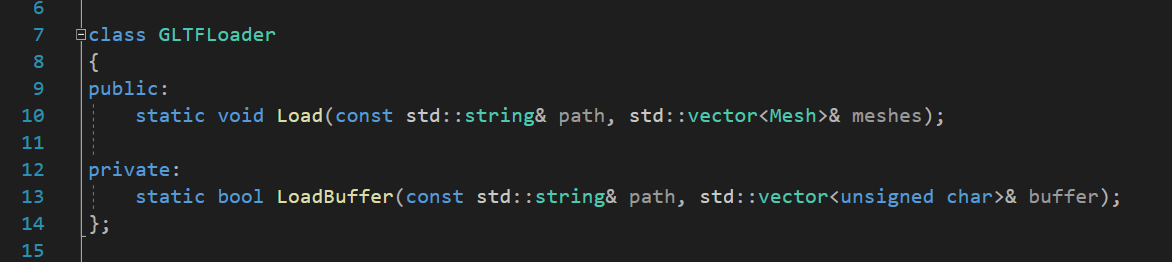
**Фиг. 3.19 Нормализация на вектор**



**Фиг. 3.20 Camera::CalculateAxes()**

**3.1.5 GLTF Loader**

Целта на glTF е да дефинира стандарт за представяне на 3D съдържание във форма, която е подходяща за използване в приложения по време на изпълнение. Съществуващите файлови формати не са подходящи за този случай на употреба: някои от тях не съдържат никаква информация за сцената, а само данни за геометрията; други са проектирани за обмен на данни между авторски приложения и основната им цел е да запазят възможно най-много информация за 3D сцената, което води до файлове, които обикновено са големи, сложни и трудни за анализиране. Освен това може да се наложи данните за геометрията да бъдат предварително обработени, за да могат да бъдат изобразени с клиентското приложение. Нито един от съществуващите файлови формати не е проектиран за случай на използване на ефективно прехвърляне на 3D сцени в мрежата и изобразяването им възможно най-ефективно. Но glTF не е „още друг файлов формат“. Това е определението за формат на предаване за 3D сцени: Структурата на сцената е описана с JSON, който е много компактен и може лесно да бъде анализиран. 3D данните на обектите се съхраняват във форма, която може да се използва директно от обикновените графични API, така че няма излишни разходи за декодиране или предварителна обработка на 3D данните. Различните инструменти за създаване на съдържание вече могат да предоставят 3D съдържание във формат glTF. И все по-голям брой клиентски приложения могат да консумират и изобразяват glTF. Така glTF може да помогне за преодоляване на пропастта между създаването на съдържание и изобразяването.

****

**Фиг. 3.21 Класът “GLTFLoader”**

**На фиг. 3.21 е показана функционалността, която съдържа GLTFLoader:**

* static void Load(const std::string& path, std::vector<Mesh>& meshes)**:** методът зарежда модели/сцени на база подадения път и референцията към вектор от обекти от тип “Mesh”.
* static void LoadBuffer(const std::string& path, std::vector<unsigned char>& buffer)**:** методът зарежда информацията от бинарни файлове във вектори от тип “unsigned char”

# Четвърта глава

Ръководство за потребителя

**4.1 Софтуерни изисквания и хардуерни изисквания**

Програмата изисква копие на операционната система Windows в комбинация с Visual Studio 2019, графичен процесор с 1 гигабайт VRAM памет, както и 4 гигабайта RAM памет.

**4.2 Инсталация**

След успешна инсталация на Visual Studio потребителят е необходимо да следва следната последователност от стъпки:

1. Потребителят трябва да кликне върху .sln файла
2. Потребителят трябва да натисне клавиша F7, за да компилира проекта
3. За да изпълни програмата, потребителят е необходимо да избере комбинацията от клавиши Ctrl+F5.

**4.3 Управление на програмата**

**4.3.1 Аргументи от командния ред**

За зареждане на сцени е необходимо да се подаде пътя през аргументите от командния ред. За тази цел е необходима следната последователност от стъпки от страна на потребителя:

Alt+F7 -> Configuration Properties -> Debugging -> Command Arguments

Аргументите се подават в следния формат:

Scene\_Path/

**4.3.2 Вход от клавиатурата**

W, A, S, D -> преместване на камерата съответно напред, наляво, назад, надясно.

SPACE -> преместване нагоре

LSHIFT -> преместване надолу

Z -> включване/изключване на Anti Aliasing

ESC -> затваряне на приложението

**4.3.3 Вход от мишката**

SCROLL UP -> Zoom In

SCROLL DOWN -> Zoom Out

# Заключение

Настоящата дипломна работа има завършен облик и е пример за 3D двигател, написан на С++ и използващ графичния API OpenGL. Имплементацията задоволява всички функционални изисквания, описани в началото на заданието и е реализирана с внимание към детайлите. Акцентът е поставен върху производителността, а гъвкавата природа на използвания език за програмиране допринася за лесната скалируемост на програмата.

Някои възможности за бъдещо развитие на проекта са:

* интерактивно меню за улеснение на потребителя
* различни видове осветление
* система за материали
* различни видове Anti Aliasing

# Използвана литература

1. <https://www.unrealengine.com/en-US/blog/unreal-engine-4-27-released>
2. [https://web.archive.org/web/20170809214002/http://www.gamasutra.com/view/feature/4035/from\_the\_past\_to\_the\_future\_tim\_.php](https://web.archive.org/web/20170809214002/http:/www.gamasutra.com/view/feature/4035/from_the_past_to_the_future_tim_.php)
3. <https://www.polygon.com/2012/10/1/3438196/better-with-age-a-history-of-epic-games>
4. <https://github.com/KhronosGroup/glTF-Tutorials/blob/master/gltfTutorial/gltfTutorial_001_Introduction.md>
5. <https://unity3d.com/beta/2022.1b>
6. <https://unity3d.com/unity/whats-new/unity-5.0>
7. [https://www.khronos.org/registry/OpenGL-Refpages/gl4//](https://www.khronos.org/registry/OpenGL-Refpages/gl4/)
8. <https://www.khronos.org/opengl/wiki/Category:Core_API_Reference>
9. <https://www.khronos.org/opengl/wiki/Shader>
10. <https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf>
11. <https://visualstudio.microsoft.com/downloads/>
12. [https://web.archive.org/web/20130312140345/http://docs.unity3d.com/Documentation/Manual/DirectX11.html](https://web.archive.org/web/20130312140345/http:/docs.unity3d.com/Documentation/Manual/DirectX11.html)
13. <https://books.google.bg/books?id=7AxTDwAAQBAJ&redir_esc=y>
14. <https://github.com/nlohmann/json>
15. <https://github.com/nothings/stb>

# Съдържание

Увод4

1. Първа глава6

1.1 Преглед на използвани технологии6

1.2 Преглед на подобни решения11

2. Втора глава21

2.1 Функционални изисквания21

2.2 Избор на програмни средства и развойна среда22

2.3 Описание на структурата на приложението29

3. Трета глава30

3.1 Реализация на програмата30

4. Четвърта глава54

4.1 Софтуерни изисквания и хардуерни изисквания54

4.2 Инсталация54

4.3 Управление на програмата55

Заключение57

Използвана литература58