ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ОЛЕСЯ ГОНЧАРА

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

Дипломна робота
перший (бакалаврський) рівень вищої освіти
спеціальність 113 Прикладна математика
освітня програма: Комп’ютерне моделювання та технології програмування

ГЕНЕРАЦІЯ ТА ВІДОБРАЖЕННЯ 3D-МОДЕЛЕЙ ОБ'ЄКТІВ У КОМП'ЮТЕРНИХ ІГРАХ З ВИКОРИСТАННЯМ СПЛАЙНІВ

Виконавець
студент групи ПА-17-2
Панасенко Є. С._____ _____
(підпис)

Керівник
доц., канд. фіз.-мат. наук
Степанова Н. І._____ _____
(підпис)

Завідувач кафедри
комп’ютерних технологій
д-р фіз.-мат. наук, проф.,
Гук Н.А._____ _____
(підпис)

2021 р.

ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ОЛЕСЯ ГОНЧАРА

Факультет прикладної математики

Кафедра комп'ютерних технологій

Рівень перший (бакалаврський)

Спеціальність 113 Прикладна математика

Освітня програма Комп'ютерне моделювання та технології програмування

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп'ютерних технологій
_____ Гук Н.А.
(підпис) **(П.І.Б.)**
«22» березня 2021 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ

Панасенко Єгор Сергійович

1. Тема роботи Генерація та відображення 3D-моделей об'єктів у комп'ютерних іграх з використанням сплайнів
керівник роботи доц., канд. фіз.-мат. наук Степанова Наталія Іванівна,
затверджені наказом по Університету від «19» березня 2021 року № 332с

2. Строк подання роботи 04 червня 2021 року

3. Вхідні дані до роботи система просторових координат об'єктів, які підлягають відображенню

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) огляд проблеми та методи розробки програмного забезпечення щодо генерації 3D-моделей об'єктів з використанням сплайнів; проектування програмного додатку; виконання програмної реалізації; аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
33 рисунки, 12 слайдів

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Степанова Н. І., доцент кафедри комп'ютерних технологій	22.03.2021	23.04.2021
2	Степанова Н. І., доцент кафедри комп'ютерних технологій	23.04.2021	11.05.2021
3	Степанова Н. І., доцент кафедри комп'ютерних технологій	11.05.2021	25.05.2021

7. Дата видачі завдання 22 березня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів роботи	Примітка
1	Знайомство з предметною областю. Вивчення алгоритмів генерації зображень в комп'ютерних іграх	08.04.2021	виконано
2	Огляд методів програмної реалізації поставленої задачі, вивчення потрібних бібліотек	19.04.2021	виконано
3	Вибір алгоритму і розробка структури програми генерації зображення	27.04.2021	виконано
4	Виконання програмної реалізації	14.05.2021	виконано
5	Аналіз результатів роботи розробленого програмного додатку	28.05.2021	виконано
6	Оформлення дипломної роботи та супроводжуючої документації	31.05.2021	виконано
7	Надання до випускової кафедри електронної версії дипломної роботи для проходження нормоконтролю	01.06.2021	виконано
8	Надання паперового примірника дипломної роботи з власноручним підписом до випускової кафедри	07.06.2021	виконано

Студент _____ Панасенко Є. С.
(підпис)

Керівник роботи _____ Степанова Н. І.
(підпис)

РЕФЕРАТ

Дипломна робота складається з 68 с., 33 рис., 11 джерел, 1 додатку.

Об'єктом дослідження даної дипломної роботи є 3D-моделі об'єктів побудованих за допомогою сплайнів, програмне забезпечення для роботи з 3D-моделями.

Мета роботи: розробка програмного забезпечення для генерації та відображення 3D-моделей об'єктів у комп'ютерних іграх з використанням сплайнів у режимі реального часу.

Методика (метод) дослідження: порівняльний аналіз сучасних засобів збереження графічних даних, графічних бібліотек; проведення тестування програмного забезпечення на відомих тестових прикладах

Одержані висновки та їх новизна: У роботі запропоновані підходи до моделювання гладких об'єктів складної форми та розроблено програмне забезпечення для відображення й редагування вказаних об'єктів. Результати досліджень можуть бути використані для створення об'єктів складної форми у системі моделювання, при розробці комп'ютерних ігор.

Результати досліджень можуть бути застосовані при розробці 3D-моделей об'єктів у комп'ютерних іграх.

Перелік ключових слів: КРИВА БЕЗ'Є, ПОВЕРХНЯ БЕЗ'Є, IMGUI, ТЕСЕЛЯЦІЙНИЙ ШЕЙДЕР, 3D-МОДЕЛІ.

ANNOTATION

The diploma work of the 4th year student Yehor Panasenko (DNU, Faculty of Applied Mathematics, Department of Computer Technology) is devoted to rendering models with Bezier surfaces.

Modeling and rendering of complex shape spatial objects with a large amount of calculated data, as well as the ability to edit them - an important modern task. The problem of visualization arises in industrial design, graphical representation of the results of scientific experiments, virtual reality systems, in the computer game industry. These areas of human activity are constantly expanding and improving, so the relevance of work related to the study of modeling and visualization of complex objects will continue to grow. So in this work it is proposed to solve complex shape objects rendering problem with Bezier surfaces.

Bibliography – 11, pictures – 33

ЗМІСТ

ВСТУП

Створення просторових моделей об’єктів виконується сьогодні у багатьох галузях науки і промисловості, таких як архітектура, медицина, будівництво, дизайн. Особливої уваги заслуговують також засоби подання динамічних 3D-об’єктів, які широко використовуються у кінематографі, індустрії комп’ютерних ігор.

У сучасному світі спостерігається неймовірний приріст потужності обчислювальної техніки і розробники систем віртуальної реальності, комп’ютерних ігор намагаються використати цю потужність якомога ефективніше з метою отримання графіки, найбільш схожої на реальний світ.

Для досягнення максимального задоволення користувачів розробники також створюють велику кількість окремих об’єктів та приголомшливих ефектів, що супроводжується значним споживання дискового простору й оперативної пам’яті. Тому завжди є актуальними питання розробки більш ефективних методів моделювання об’єктів складної форми, які б використовували менше обчислювальних ресурсів.

Які можливості сьогодні мають розробники інтерактивних програмних продуктів для зберігання об’єктів? По-перше, майже двадцять років тому, коли потужність процесорів достатньо зросла, щоб швидко виконувати великі об’єми обчислень, було розроблено векторний формат SVG. Для побудови зображень формат використовує криві Безьє, які є окремим випадком B-сплайнів. Сьогодні формат SVG є поширеним, він підтримується всіма сучасними браузерами для настільних і мобільних пристроїв.

Формат SVG дозволяє зберігати як статичну, так і анімовану двовимірну графіку. Якщо розглядати використання SVG формату у інтерактивних системах, зокрема у комп’ютерних іграх, дуже цікавою є можливість закріплення за

об’єктом у даному форматі обробника подій, що дає користувачеві можливість керувати зображенням: міняти його форму, пересувати. Крім того, векторним форматам притаманні гарна масштабованість й незначне використання дискового простору за умови, що зображення складається з невеликої кількості простих елементів, що також сприяє популярності SVG формату у розробників інтерактивних графічних додатків.

З іншого боку, SVG як і всі векторні формати, має також і недоліки: у порівнянні з растровими аналогами побудова SVG-зображення потребує більше процесорного часу, а зображення, що складаються з великої кількості дрібних деталей, починають вимагати більше дискового простору ніж аналогічні растрові.

Також суттєвим обмеженням для використання формату SVG у індустрії комп’ютерних ігор є те, що він не підтримує опис тривимірної графіки.

У тривимірному просторі найбільш розповсюдженим форматом є OBJ – простий і гнучкий формат, що дозволяє створювати об’єкти за допомогою різних способів, у тому числі з використанням кривих Безьє і B-сплайнів.

Таким чином на даний час вже існують формати, які дозволяють зберігати окремі об’єкти компактно, забезпечувати їх легку масштабованість.

Але у реальному ігровому процесі, де об’єкти мають досить складні форми, постійно взаємодіють один з одним, а сцени є досить насиченими, виникає проблема: як найбільш просто зробити опис об’єктів і забезпечити їх подальшу динаміку з найменшим навантаженням на комп’ютерну систему?

Виходячи із цієї проблеми у ході роботи буде розглянуто існуючі математичні моделі поверхонь, які можуть використовуватися для відображення у комп’ютерних іграх. На основі цих математичних моделей буде вибрано оптимальну математичну моделі, досліджено як її можна використовувати для повноцінного рендеренгу. І далі буде написано програмне забезпечення, яке буде використовувати цю модель для рендеренгу і більшу того у режимі реального

часу за допомогою відеокарти.

Дипломна робота складається з таких частин:

- вступ, який обґрунтовує актуальність роботи; визначає цілі проведення наукового дослідження; галузь дослідження; методи дослідження або розрахунків;
- постановка задачі;
- аналітичний огляд літературних джерел;
- дослідження математичних моделей складних об’єктів;
- розробка програми для моделювання об’єктів;
- аналіз розробленої програми для моделювання об’єктів;
- висновки;
- список використаних джерел;
- додатки.

ПОСТАНОВКА ЗАДАЧІ

Метою цієї роботи є розробка програмного забезпечення для генерації та відображення 3D-моделей об’єктів у комп’ютерних іграх із застосуванням сплайн-технології у режимі реального часу. Виконання поставленої у роботі задачі передбачає розгляд існуючих підходів щодо моделювання кривих і поверхонь у комп’ютерній графіці, а також більш детальне вивчення сплайн-інтерполяції як одного з таких підходів. У загальному випадку були розглянуті 3D-моделі об’єктів, побудованих з використанням поверхонь Безьє.

Для досягнення мети необхідно виконати наступні задачі:

- дослідити математичні моделі просторових об’єктів;
- обрати оптимальний спосіб подання інформації про об’єкти, які потрібно відобразити;
- сформувати вимоги до програми відображення 3D-об’єктів і виконати програмну реалізацію;
- розробити інтерфейс для створення та редагування 3D-об’єктів;
- зробити програмне забезпечення придатним для компіляції та роботи у різних операційних система;
- розробити шейдер для генерації 3D-моделі за допомогою відеокарти;
- надати опис розробленого програмного забезпечення та створити схеми взаємодії його компонентів.

1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1.1 Моделі подання просторових об’єктів

Комп’терна графіка пропонує сьогодні різні засоби моделювання просторових форм і об’єктів. Геометричне моделювання - це математичний опис об’єктів у просторі певними атрибутами: координатами, розмірами, формою. При відображенні геометричних об’єктів потрібно враховувати також їх просторове розташування і поведінку: переміщення, повороти відносно координатних осей (шість ступенів свободи), зіткнення з перешкодами або іншими об’єктами. Крім того для отримання образів просторових форм на площині екрану необхідно використовувати ще одне геометричне перетворення - проеціювання. Для дослідження математичних моделей поверхонь та об’єктів було використано посібники авторів Порев [5] та Нікулін [6]

1.1.1 Математичний опис моделей поверхонь та об’єктів

У комп’ютерній графіці прийнята така класифікація моделей поверхонь і об’єктів:

- Каркасні - на екрані візуалізуються не всі точки поверхні, а лише невелика їх кількість, достатня, щоб передати характер поверхні. Пари точок утворюють систему ліній і формують каркас моделі;
- Точкові - на екрані відображаються точки з відповідним забарвленням;
- Кінематичні - поверхня будується неперервним рухом у просторі лінії по заданій траєкторії;
- Кусочні – поверхня складається з окремих фрагментів, при обмеженому наборі даних у поверхні присутні розриви і злами;

- Сплайнові – моделі використовуються для побудови гладких поверхонь на основі обчислення координат за допомогою розв’язання систем лінійних алгебраїчних рівнянь [8, с. 487];
- Фрактальні – при побудові поверхні використовується властивість об’єктів до самоподібності в залежності від масштабу;
- Графічні - використовуються у разі, якщо не можливо виділити певний закон для побудови і поверхня заповнюється деякими дискретними елементами.

У загальному випадку не можна стверджувати, що одна математична модель краща за іншу. Так, наприклад, каркасна модель зручна для виконання швидкої візуалізації поверхні, кінематична підходить для об’єктів з природною симетрією, графічна дає більш реалістичне уявлення про об’єкт. Тому необхідно обирати математичну модель з урахуванням потрібного ступеня реалістичності, обчислювальних можливостей комп’ютерної системи, особливостей задачі, для якої застосовується моделювання об’єктів.

1.1.2 Опис розташування об’єктів у сцені

Сцена у комп’ютерній графіці - це сукупність об’єктів, які підлягають відображенню, описана за допомогою деякої математичної моделі. Візуалізацією називають процес перетворення математичної моделі сцени у вигляд, придатний для показу на наявних пристроях виведення.

Для подання об’єктів сцени у графіці використовують декілька координатних систем: об’єктну (жорстко зв’язана з об’єктом), світову (нерухома система, призначена для визначення взаємного розташування всіх об’єктів сцени), видову (система спостерігача, визначає напрямок камери і ракурс показу).

Для здійснення переходу від однієї координатної системи до іншої ви-

користовуються матриці базових геометричних перетворень (зсуву, обертання, масштабування). Складні перетворення визначаються шляхом перемноження матриць відповідних елементарних перетворень між собою. Таким чином спочатку відбувається перехід від об'єктної системи координат до світової, а потім зі світової до видової.

Після отримання видових координат об'єктів сцени виконується проектування сцени на екранну площину. Для цього використовується матриця проективного перетворення (паралельне або центральне проектування), яка дає змогу отримати екранні (двовимірні) координати об'єктів сцени. Третя видова координата зазвичай зберігається; з її допомогою визначають взаємне розташування об'єктів сцени за глибиною.

На останньому кроці відбувається перетворення координат об'єктів з урахуванням особливостей системи графічного виводу.

1.2 Бібліотека Visualization Library

У ході роботи було знайдено таку біблиотеку, як Visualization Library [10]. Ця біблиотека написана на мові C++ і може використовуватись для графіки у 2D або 3D. Вона дозволяє моделювати різні види поверхонь, фрактали, та багато іншого. Проаналізувавши можливості використання бібліотеки було отримано такі висновки:

- Бібліотека написана на мові C++ та з використанням виключень, таким чином це робить неможливим її використання іншими мовами програмування.
- Бібліотека самостійно реалізує свою матрицю та вектор, таким чином закривають можливість оптимізувати операції над матрицями. Більш того бібліотека не використовує команди SSE, які дають приріст у швидкості,

як це зроблено у бібліотеці CGLM.

- Бібліотека вже не підтримується розробниками, останній внесення змін у код було 20 лютого 2020 року, у порівнянні з бібліотекою CGLM, яка активно розвивається.
- Якщо подивитися на реалізацію кривих Безьє, то ми побачимо, що бібліотека не використовує матричний спосіб отримання вершин з поверхні Безьє, таким чином ми знову не можемо використати оптимізацію за допомогою команд SSE.
- Також перерірено спосіб знаходження нормалей для поверхні, біблиотека знаходить нормалі по отриманим трикутникам при будуванні поверхні, хоча для поверхні Безьє існує значно швидший та дешевший спосіб знаходження нормалі, цей спосіб будується на знаходження похідних до кривої Безьє з різних сторін.

1.3 Програмні інтерфейси для роботи з моделями

Важливим питанням для даної роботи є правильний вибір програмного інтерфейсу для рендерингу складних моделей. Ці програмні інтерфейси допомагають використовувати відеокарту для рендернгу, а із розвитком цих інтерфейсів з’являються нові можливості. Загалом існує три популярні програмні інтерфейси: DirectX, OpenGL, Vulkan.

У цій роботі розглянуто програмний інтерфейс OpenGL, так як він є найпопулярнішим із кроссплатформенних програмних інтерфейсів. Також OpenGL має теселяційний шейдер [9] який дозволяє генерувати нові вершини у режимі реального часу, що дуже корисно якщо використовувати математичні моделі поверхонь з контрольними точками [11].

2 ДОСЛІДЖЕННЯ МАТЕМАТИЧНИХ МОДЕЛЕЙ СКЛАДНИХ ОБ'ЄКТІВ

Для подальшої програмної реалізації серед існуючих математичних моделей поверхонь об'єктів було обрано поверхню Безьє, яка є частинним випадком B-сплайнів.

2.1 Сплайни і сплайн-інтерполяція

Існує досить велика кількість геометричних конструкцій, які називають сплайнами. Наприклад, експоненціальні (напружені) сплайни, тригонометричні, раціональні сплайни. У комп'ютерній графіці найбільш широке застосування знайшли кубічні сплайни та метод інтерполяції кубічними сплайнами.

Особливість сплайн-інтерполяції полягає в тому, що сплайнова крива складається з кількох поліномів третього ступеня, а їх кількість дорівнює кількості інтервалів, всередині яких ми виконуємо інтерполяцію. Гладкість побудованої інтерполяційної кривої забезпечується безперервністю першої похідної на всьому інтервалі інтерполяції.

Розглянемо загальний випадок спланової кривої. Нехай у тривимірному просторі існують вектори $u_i = [x_i\ y_i\ z_i], i = \overline{0, n}$, ці вектори визначають вузлові точки сплайнової кривої. Будемо вважати, що вузлові точки пронумеровані у порядку з'єднання кривої.

Параметричне подання сплайнової кривої має вигляд:

$$\begin{cases} x_i(t) = s_{3x_i}t^3 + s_{2x_i}t^2 + s_{0x_i}t + s_{1x_i} \\ y_i(t) = s_{3y_i}t^3 + s_{2y_i}t^2 + s_{0y_i}t + s_{1y_i} \\ z_i(t) = s_{3z_i}t^3 + s_{2z_i}t^2 + s_{0z_i}t + s_{1z_i} \end{cases},$$

або у векторній формі:

$$\begin{cases} p_i(t) = [t^3 \quad t^2 \quad t \quad 1]\, S_i \\ \forall t \in [0, d_i],\ i = \overline{0, n} \end{cases}, \text{ де } S_i = \begin{bmatrix} s_{3x_i} & s_{3y_i} & s_{3z_i} \\ s_{2x_i} & s_{2y_i} & s_{2z_i} \\ s_{1x_i} & s_{1y_i} & s_{1z_i} \\ s_{0x_i} & s_{0y_i} & s_{0z_i} \end{bmatrix}.$$

З урахуванням властивостей сплайнів будується система лінійних алгебраїчних рівнянь відносно невідомих коефіцієнтів сплайну:

$$QS = U \to S = Q^{-1}U,$$

де $Q \in R^{4n \times 4n}$ - матриця яка задає необхідні умови для системи, $S = \begin{bmatrix} S_1 \\ \dots \\ \S n \end{bmatrix} \in R^{4n \times 3}, U \in R^{4n \times 3}$.

2.1.1 Крива Безьє

Окремим випадком сплайнів є крива Безьє. Кубічну криву Безьє можна побудувати з використанням чотирьох опорних точок $P_i, i = \overline{0, 3}$. У параметричному вигляді отримаємо:

$$B(t) = (1-t)^3 P_0 + t(1-t)^2 P_1 + t^2(1-t)P_2 + t^3 P_3,$$

де t – параметр, $t \in [0, 1]$.

Розглянемо графічний спосіб побудови кривої Безьє із застосуванням алгоритму де Кастельє (рис. 2.1).

Для отримання точки кривої, яка відповідає, наприклад, значенню параметра $t = 0.25$ потрібно відкладасти 0.25 шляху на відрізках $P_i P_{i+1}$. В результаті

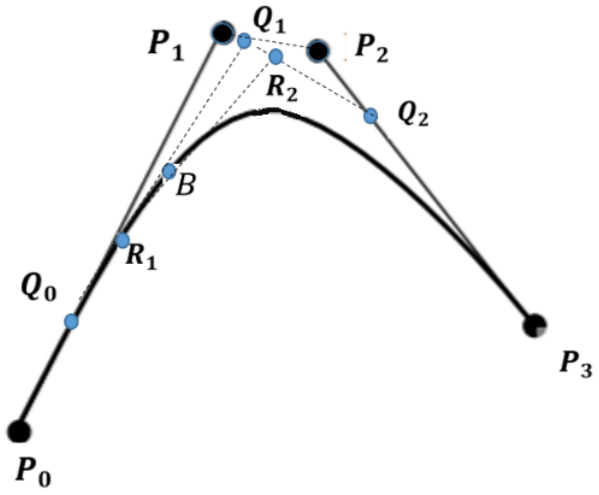

Рисунок 2.1 – Побудова кривої Безьє

отримаємо точки $Q_j, j = \overline{0,2}$, на наступному кроці зробимо теж саме і отримаємо R_0 та R_1.У такий спосіб чином ми отримали дотичну до кривої, ця властивість буде використана для побудови нормалі у поверхні кривої Безьє. І знову прокладемо 0.25 шляху на відрізку R_0R_1 отримаємо нашу точку B, яка знаходиться на кривій. Якщо ми будемо послідовно обирати t, наприклад з кроком 0.1, та з'єднувати у відрізки, то ми отримаємо ламану. Зі зменшенням кроку ламана буде ставати все більш схожою на криву. Таким чином можна підібрати такий крок, при якому на екрані комп'ютера буде відображатися крива.

Крива Безьє задається формулою:

$$B(t) = \sum_{i=0}^{n} P_i b_{i,n}(t),$$

де P_i - контрольні точки, а $b_{k,n}(t)$ - поліноми Бернштейна, базисні функції кривої Безьє.

$$b_{k,n}(t) = C_i^n t^k (1-t)^{n-k},$$

де C_i^n число поєднань з n по k

$$C_i^n = \frac{n!}{k!(n-k)!}.$$

Побудуємо формулу кубічної кривої Безьє:

$$B(t) = (1-t)^3 P_0 + t(1-t)^2 P_1 + t^2(1-t)P_2 + t^3 P_3.$$

Цю формулу можна отримати побудувавши криву графічним способом. Прокласти шлях від однієї контрольної точки до іншої можна таким чином $(1-t)P_i + tP_{i+1}$, якщо ми послідовно проробимо ті самі кроки, що і у графічному будуванні, отримаємо:

$$\begin{aligned} B(t) =& t(t((1-t)P_2 + tP_3) + (1-t)((1-t)P_1 + tP_2)) + \\ &+ (1-t)(t((1-t)P_1 + tP_2) + (1-t)((1-t)P_0 + tP_1)) \end{aligned}$$

Спростимо формулу:

$$B(t) = -t^3P_0 + 3t^3P_1 - 3t^3P_2 + t^3P_3 + 3t^2P_0 - 6t^2P_1 + 3t^2P_2 - 3tP_0 + 3tP_1 + P_0. \quad (1)$$

Тепер ми можемо записати формулу у матричному вигляді:

$$B(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix},$$

або нехай $P_i = [p_{ix}\ p_{iy}\ p_{iz}\ 1]$,

$$B(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{0x} & p_{0y} & p_{0z} & 1 \\ p_{1x} & p_{1y} & p_{1z} & 1 \\ p_{2x} & p_{2y} & p_{2z} & 1 \\ p_{3x} & p_{3y} & p_{3z} & 1 \end{bmatrix}.$$

Враховуючи що на сучасних комп'ютерах завдяки кешуванню рядків, то множити матрицю на вектор швидше ніж вектор на матрицю, то більш опти-

мальною формулою буде:

$$B(t) = \begin{bmatrix} p_{0x} & p_{0y} & p_{0z} & 1 \\ p_{1x} & p_{1y} & p_{1z} & 1 \\ p_{2x} & p_{2y} & p_{2z} & 1 \\ p_{3x} & p_{3y} & p_{3z} & 1 \end{bmatrix}^T \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}.$$

Тепер знайдемо похідну до вираження (1), для того щоб знайти дотичну, отримаємо:

$$B(t) = -3t^2P_0 + 9t^2P_1 - 9t^2P_2 + 3 * t^2P_3 + 6tP_0 - 12tP_1 + 6tP_2 - 3P_0 + 3P_1. \quad (2)$$

Запишемо у матричному вигляді:

$$B(t) = \begin{bmatrix} p_{0x} & p_{0y} & p_{0z} & 1 \\ p_{1x} & p_{1y} & p_{1z} & 1 \\ p_{2x} & p_{2y} & p_{2z} & 1 \\ p_{3x} & p_{3y} & p_{3z} & 1 \end{bmatrix}^T \begin{bmatrix} 0 & 0 & 0 & 0 \\ -3 & 9 & -9 & 0 \\ 6 & -12 & 6 & 0 \\ -3 & 3 & 0 & 0 \end{bmatrix}^T \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}.$$

Властивості кривої Безьє:

- неперервність заповнення сегменту між початковою та кінцевою точками,
- крива завжди знаходитися у фігурі утвореній контрольними точками, у кубічній кривій це буде деякий чотирикутник, цю властивість можна використати для того щоб перевірити чи не перетинаються дві криві на початковому етапі,
- якщо контрольні точки знаходяться на одній прямій, то утворюється пряма лінія,
- крива симетрична, тобто якщо переставити вектор контрольних точок у

зворотьому порядку, то отримаємо ту саму форму,

- крива афінно інваріантна,
- зміна однієї контрольної точки приводить до зміни всієї кривої,
- будь який сегмент кривої є крива Безьє.

2.1.2 Поверхня Безьє

Як і крива Безьє, поверхня Безьє визначається набором контрольних точок. Розглянемо графічний спосіб побудови кубічної поверхні Безьє з 16 контрольними точками (рис. 2.2).

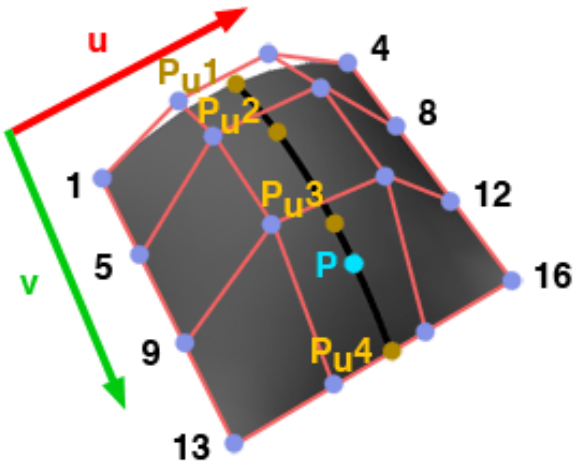

Рисунок 2.2 – Побудова поверхні Безьє

Спочатку будуємо 4 кубічні криві Безьє через контрольні точки 1-4, 5-8, 9-12, 13-16 використовуючи дійсне число *v*, далі використовуючи точки відповідних *v* на отриманих кривих як контрольні точки наступної кривої будуємо наступну криву використовуючи дійсне число *u*, таким чином ми отримаємо поверхню побудованої з багатьох кривих, причому як ми все знаємо відрізок отриманий в останньому кроці при побудові кривої це дотична, якщо ми побудуємо поверхню будуючи криві по контрольним точкам 1, 5, 9, 13 і так далі до

4, 8, 12, 16, то ми отримаємо ще одну дотичну, але в деякому іншому напрямку, і якщо ми знайдемо векторний добуток отриманих дотичних, ми отримаємо нормаль до поверхні у даній точці.

Поверхня Безьє задається формулою:

$$p(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{n} b_{i,n}(u) b_{j,n}(v) P_{ij}$$

В комп'ютерній графіці поверхні Безьє використовують для подання гладких поверхонь. Вони досить компактні, ними легко маніпулювати, вони мають гарні властивості безперервності. Крім того, такі канонічні поверхні, як сфери і циліндри, можна добре апроксимувати невеликим числом кубічних поверхонь Безьє.

3 РОЗРОБКА ПРОГРАМИ ДЛЯ МОДЕЛЮВАННЯ ОБ'ЄКТІВ

На початку розробки графічного додатку розробник повинен визначитися, з якого рівня починати писати власний програмний код. Програмуванням на рівні графічного обладнання, як правило, займаються лише його виробники. Графічна бібліотека, яка реалізує певний стандарт абстрагування від обладнання, безпосередньо взаємодіє з драйвером. Стандартами абстрагування є, наприклад, бібліотеки OpenGL (відкрита графічна бібліотека для настільних комп'ютерів під керуванням різних ОС), Direct3D (призначена для різних ЕОМ під управлінням Windows і Windows Phone), Metal (для мобільних пристроїв під керуванням iOS). [7]

У даній роботі для розробки програмного продукту обрано кросплатформовий програмний інтерфейс OpenGL, що забезпечує незалежність програмного додатку від операційної системи.

3.1 Вимоги до розробленого програмного забезпечення

Для розробки програмного забезпечення висунуто такі вимоги:

- Програмне забезпечення повинно мати відкритий вихідний код та ліцензію вільного програмного забезпечення.

  Це дозволить будь якому досвідченому користувачу зкомпілювати програмне забезпечення під будь яку платформу та операційну систему, або навіть дасть можливість модифікувати код під свої потреби.

  Також ліцензія повинна бути сумісна з ліцензіями використаних бібліотек. Загалом були використані бібліотеки CGLM, ImGui, GLFW, EASTL та прогрмний інтерфейс OpenGL. Перші біблиотеки CGLM та ImGui використовують ліцензію MIT, бібліотека GLFW використовує ліцензію

ZLib, а EASTL – ліцензію BSD. А програмний інтерфейс OpenGL має ліцензію подібну до ліцензії BSD. Усі ці ліцензії є сумісними з ліцензією LGPLv3, яка є подібною до GPL, але дозволяє використовувати програмне забезпечення у пропріетарних проектах.

- Програмне забезпечення повинно працювати у режимі реального часу.
  Саме таким чином було вибрано мову C++ та бібліотеку CGLM, які дозволяють досягти найбільшої швидкості роботи програми у порівнянні з іншими мовами програмування, причому практично не знижуючи швидкості розробки коду. Більш того CGLM автоматично компілюється з використанням SSE команд, якщо є така можливість, що ще дає приріст у швидкості.
- Програмне забезпечення повинно дати можливість використання бібліотеки якомога більшому колу розробників.
  Саме тому було обрано мову програмування C++ та бібліотеку EASTL, яка на відміну від стандартної бібліотеки STL дозволяє розробляти без використання виключень. Таким чином за допомогою інструментів можна на основі цього програмного забезпечення згенерувати C код, який у подальшому можна обернути у більшість мов програмування і таким чином програмне забезпечення зможуть використати і розробники, які не знають C++, але знають деяку іншу мову програмування.
- Програмне забезпечення повинно бути якомога простим та легким, та залежати від простих та легких бібліотек.
  Програмне забезпечення повинно розроблятися по принципу KISS (акронім для “Keep it simple, stupid”), що означає що проектування повинно бути якомога простішим. Таким чином можна уникнути багатьох помилок пов’язані з тим що неможливо розробник не може охопити структуру вихідного коду складного програмного забезпечення, а також таке про-

грамне забезпечення має дуже малий розмір зкомпільованої програми, що підвищує легкість розповсюдження. А також саме тому було вибрано саме такий набір бібліотек, а загалом графічну бібліотеку ImGui, яка має досить невеликий обсяг коду, приблизно 30 тисяч строк коду разом з коментарями.

3.2 Опис програмного забезпечення

Було розроблено програмне забезпечення на мові C++ для моделювання поверхні Безьє за допомогою програмного інтерфейсу OpenGL, що використовується для відображення 2D та 3D векторної графіки на екран, основна особливість, чому була вибрано саме OpenGL це те що інтерфейс має вільну ліцензію подібну до BSD, її підтримують більшість оперативних систем та інтерфейс на мові Сі. Також були використані бібліотеки:

- CGLM - математична бібліотека написана на мові Сі. Використовує ліцензію MIT.

  У программі загалом використовується для афінних перетворень та арифметичними операціями між матрицями. За замовчанням використовує команди SSE, що дозволяють прискорити швидкість обчислення завдяки повному виковистанню особливостей обчислення процесорів.

- ImGui - бібліотека для графічного інтерфейсу написана на мові C++. Використовує ліцензію MIT.

  Бібліотека має невелику кодову базу порівняно з аналогічними графічними бібліотеками та фреймворками, такими як GTK або QT, та дозволяє створювати динамічні віджети.

- GLFW - бібліотека для відображення вікна з OpenGL та обробки вводу. Використовує ліцензію ZLib.

- EASTL - бібліотека для заміни стандартного STL. Використовує ліцензію BSD.

  Бібліотека EASTL дозволяє замінити стандартну бібліотеку STL для того щоб уникнути виключень, що не оброблюється деякими мовами програмування. Також бібліотека цікава тим що вона реалізує оптимізовані версії контейнерів, що мають такий же самий інтерфейс, що і звичайні контейнери.

  Код програмного забезпечення складається з таких компонентів:

- Вихідний код програми, який зберігається у директорії “osdo”. Тут знаходиться бібліотека “osdo”, яка не використовує STL, таким чином її можна використовувати іншими мовами програмування. Загалом тут знаходяться файли заголовків з розширенням “.h” та з реалізацією з розширенням “.cpp”, кожен файл заголовків у цій директорії утворює окремий клас.

- Вихідний код програми, який зберігається у директорії “druidengine”. Тут знаходиться інтерфейс програми, так як деякі компоненти ImGui, такі як файловий менеджер, використовує STL, це унеможливлює використання іншими мовами програмування, хоча це і не потрібно, так как інтерфейс програми не потібен для розробки. Загалом тут знаходяться файли заголовків з розширенням “.h” та з реалізацією з розширенням “.cpp”, кожен файл заголовків у цій директорії утворює окремий клас.

- Ресурси програми, які зберігаються у директорії “res” (скорочено “resource”). Тут знаходяться шейдери та тестові моделі чайнику, моделі машини та деякої еліпсоподібної моделі.

- Файл з правилами компіляції для CMake. CMake дозволяє компілювати програму незалежно від платформи, більш того дозволяє створити інсталяційний файл на цю платформу.

  Після компіляції ми отримаємо нашу програму у директорії “bin” та ре-

сурси у директорії “share/osdo”, ця структура директорії Unix подібна.

3.3 Огляд роботи програми

Для тестування розробленого програмного забезпечення використано відому модель “Чайник з Юти”, за допомогою якої перевіряють відображення складних об’єктів. Результат генерації моделі продемонстровано на рис 3.1.

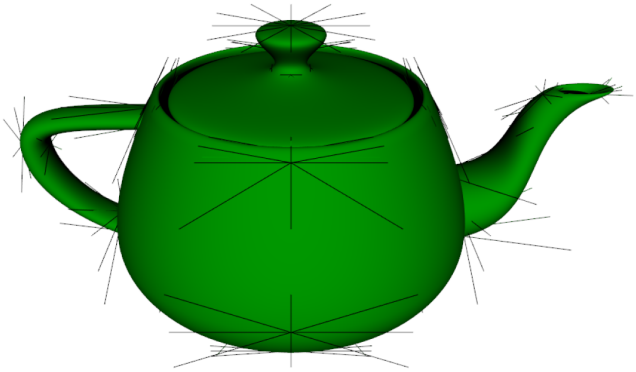

Рисунок 3.1 – Тестування програми на стандартній моделі

Інтерфейс користувача розробленої програми наведено на рис. 3.2

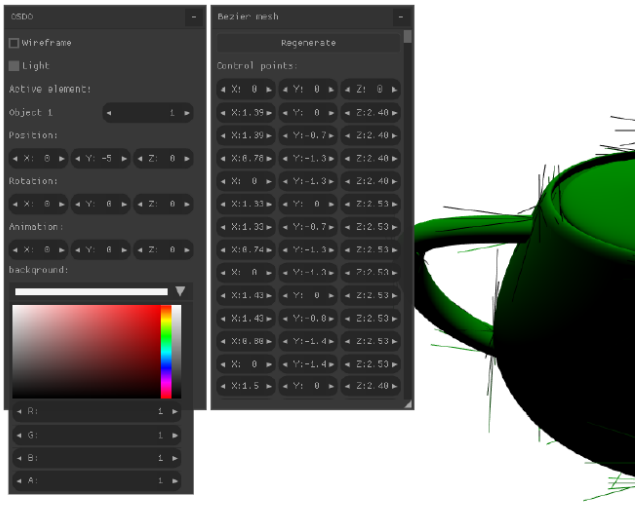

Рисунок 3.2 – Інтерфейс користувача програми

На рис. 3.3 продемонстровано побудовану з використанням розробленого програмного додатку каркасну модель тестового прикладу.

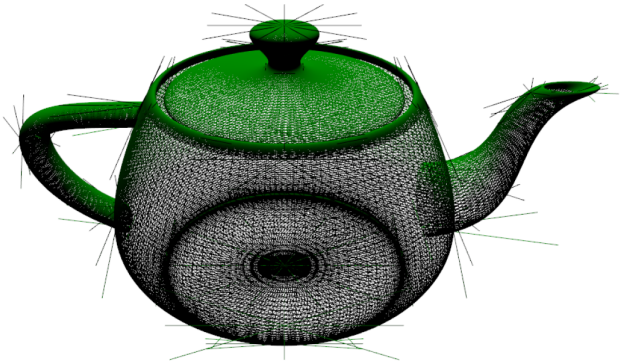

Рисунок 3.3 – Генерація каркасної моделі тестового прикладу

Далі розглянемо приклад створення об’єкту за допомогою розробленої програми. При початковому завантаженні програми ми отримаємо вікно, яке продемонстровано на рис. 3.2. Для налаштування області виводу зображення користувач може скористуватися головним та допоміжним вікнами. У допоміжному вікні є можливість перемкнутися у режим каркасу, перемкнути режим світла. Зробивши камеру джерелом світла також можна вибрати активний елемент з наявних (за замовчанням це камера).

У активному елементі ми можемо задати позицію, поворот та анімацію повороту. Якщо перемкнутися на деякий об’єкт отримаємо наступні екрани (рис. 3.4, 3.5).

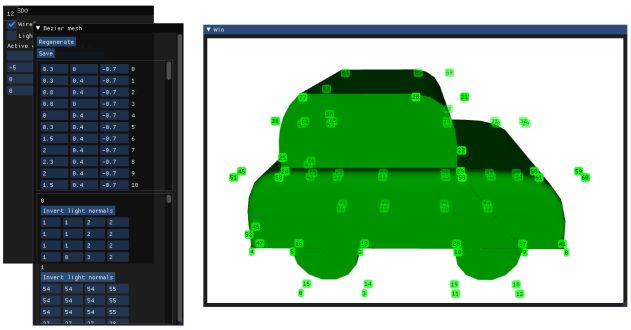

Рисунок 3.4 – Генерація моделі (режим редагування)

У даному режимі з’являється можливість редагування об’єкту побудованого за допомогою поверхонь Безьє. Загалом на головному вікні з’являються номери контрольних точок. Також з’являється третє вікно у якому присутні та-

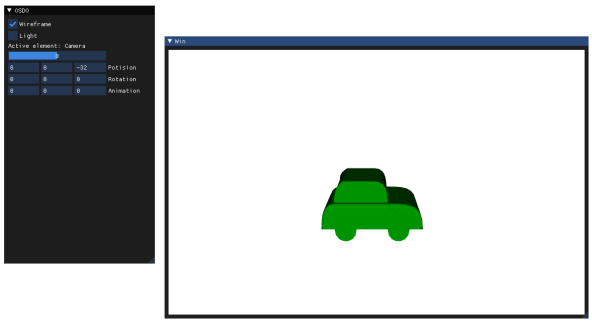

Рисунок 3.5 – Генерація моделі (режим перегляду)

кі елементи:

- Кнопка ”Regenerate дозволяє перебудувати об’єкт
- Кнопка ”Save зберігає об’єкт на диск
- Підвікно з можливістю знайти контрольну точку за її номером та змінити її координати
- Підвікно з можливістю знайти одну з поверхонь та відредагувати такими елементами:
- Кнопка ”Invert light normals дозволяє змінити порядок контрольних точок поверхні для того щоб нормалі поверхні дивилися в протилежну сторону.
- 16 полів з номерами контрольних точок.

4 АНАЛІЗ РОЗРОБЛЕНОЇ ПРОГРАМИ ДЛЯ МОДЕЛЮВАННЯ ОБ'ЄКТІВ

Для реалізації поставленої задачі у програмному забезпеченні було створено класи, кожен з яких міститься у окремому файлі. Далі буде описано перелік класів, опис класів, а у кінці розділу буде описано файли.

4.1 Клас Beziator

Клас який зберігає та оброблює модель утворену через поверхні Безьє. Схему успадкувань можна розглянути на рис. 4.1, а діаграму зв'язків на рис. 4.2.

```
#include <beziator.h>
```

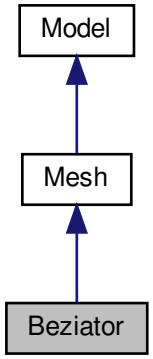

Рисунок 4.1 – Схема успадкувань для Beziator

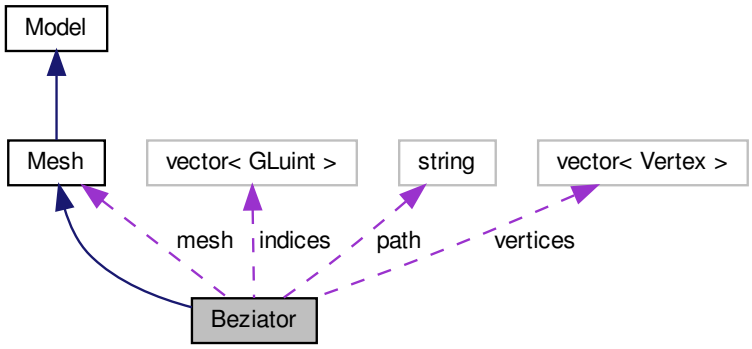

Рисунок 4.2 – Діаграма зв'язків класу Beziator

Загальнодоступні типи

- typedef surfacei_t ∗ surfaces_vector

  Тип позначаючий вказівнкик на массив з поверхнями Безьє.

Загальнодоступні елементи

- Beziator (const string &path)

  Конструктор до Beziator, який зберігає шлях до файлу з моделлю.
- ∼Beziator () override
- bool init ()

  Завантажує модель у пам'ять.
- void draw (Shader &shader, bool pre_generated) override

  Відображує модель.
- void generate (size_t d=8) override

  Генерує деталізований меш моделі.
- bool save ()

  Зберігає модель у файл, вказаний у полі `path`.
- void rotate (size_t i)

  Інвертує порядок індесів поверхні, щоб нормалі дивилися у протиле-

жний бік.

- vector< Vertex > * get_vertices () override

  Видає список вершин моделі.

Захищені дані

- const string path

  Шлях до файлу у якому зберігається модель.

- Mesh mesh

  Згенерований за допомогою CPU меш моделі.

- vector< Vertex > vertices

  Масив вершин/вузлів моделі.

- vector< GLuint > indices

  Масив індексів, що утворюють поверхні Безье.

Детальний опис Клас який зберігає та оброблює модель утвореню через поверхні Безьє.

Див. визначення в файлі beziator.h, рядок 22

Опис типів користувача

surfaces_vector `typedef surfacei_t* Beziator::surfaces_vector`

Тип позначаючий вказівнкик на массив з поверхнями Безьє.

Див. визначення в файлі beziator.h, рядок 27

Конструктор(и)

Beziator() `Beziator::Beziator (`

`const string & path )`

Конструктор до Beziator, який зберігає шлях до файлу з моделлю.

Обов'язково потібно запустити метод `Beziator::init` для того щоб завантажити модель у пам'ять.

Аргументи

path Шлях до файлу у якому зберігається модель.

Див. визначення в файлі beziator.cpp, рядок 18

∼Beziator() `Beziator::∼Beziator ( ) [override]`

Див. визначення в файлі beziator.cpp, рядок 58

Опис методів компонент

draw() `void Beziator::draw (`

`Shader & shader,`

`bool pre_generated ) [override], [virtual]`

Відображує модель.

За допомогою флагу `pre_generated` можна задати яким чином потібно відображати, якщо задати `false`, то у буде використаний меш із поверхнями Безье 4x4, а якщо задано `true`, то відобразиться сгенерований деталізований меш моделі.

Аргументи

shader Шейдер який використовуєтсья для відображення моделі.

pre_generated Флаг, який позначає який з мешів відображати.

Переозначення з Model.

Див. визначення в файлі beziator.cpp, рядок 61, також можна розглянути граф всіх викликів на рис. 4.3

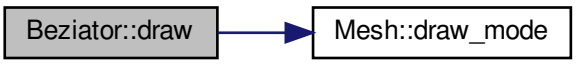

Рисунок 4.3 – Граф всіх викликів Beziator::draw

generate() `void Beziator::generate (`

`size_t d = 8 ) [override], [virtual]`

Генерує деталізований меш моделі.

Ступінь деталізаії d позначає скільки вершин буде створено по двом осям, за заммовчанням задано 8, таким чином поверхня буде складатися з 8x8=64 вершини.

Аргументи

d ступінь деталізації.

Переозначення з Model.

Див. визначення в файлі beziator.cpp, рядок 136, також можна розглянути граф всіх викликів на рис. 4.4

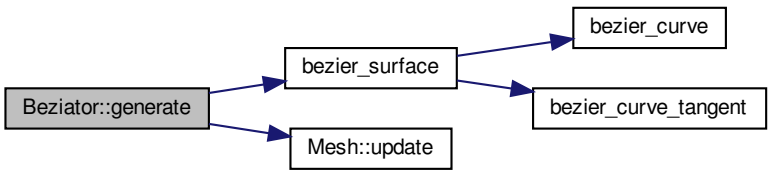

Рисунок 4.4 – Граф всіх викликів Beziator::generate

get_vertices() `vector< Vertex > * Beziator::get_vertices ( ) [override], [virtual]`

Видає список вершин моделі.

Повертає

Вказівник на поле `vertices`.

Переозначення з Model.

Див. визначення в файлі beziator.cpp, рядок 287

init() `bool Beziator::init ( )`

Завантажує модель у пам'ять.

Повертає

Статус, чи успішно була завантажена модель.

Див. визначення в файлі beziator.cpp, рядок 20, також можна розглянути граф всіх викликів на рис. 4.5

Рисунок 4.5 – Граф всіх викликів Beziator::init

rotate() `void Beziator::rotate (`

`size_t i )`

Інвертує порядок індесів поверхні, щоб нормалі дивилися у протилежний бік.

Аргументи

i номер поверхні.

Див. визначення в файлі beziator.cpp, рядок 277

save() `bool Beziator::save ( )`

Зберігає модель у файл, вказаний у полі `path`.

Повертає

Статус зберігання файлу.

Див. визначення в файлі beziator.cpp, рядок 110

Компонентні дані

indices `vector<GLuint> Beziator::indices [protected]`

Масив індексів, що утворюють поверхні Безье.

Індекси розташовані у масиві по 16 елементів, які утворюють поверхню з контрольними точками 4x4. Масив легко інтерпретуєтсья у `surfaces_vector`:

```
surfacei_t *surfaces = reinterpret_cast<surfacei_t*>(indices.data());
```

Див. визначення в файлі beziator.h, рядок 52

mesh `Mesh Beziator::mesh [protected]`

Згенерований за допомогою CPU меш моделі.

Див. визначення в файлі beziator.h, рядок 36

path `const string Beziator::path [protected]`

Шлях до файлу у якому зберігається модель.

Див. визначення в файлі beziator.h, рядок 32

vertices `vector<Vertex> Beziator::vertices [protected]`

Масив вершин/вузлів моделі.

Див. визначення в файлі beziator.h, рядок 42

Документація цих класів була створена з файлів:

- osdo/beziator.h
- osdo/beziator.cpp

4.2 Клас Bijective

Інтерфейс до об'єктів, що можуть можуть бути переміщені та повернуті у просторі. Схему успадкувань можна розглянути на рис. 4.6.

`#include <bijective.h>`

Схема успадкувань для Bijective

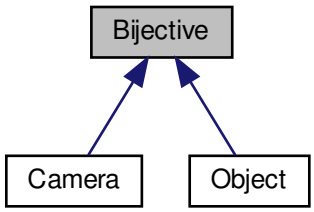

Рисунок 4.6 – Схема успадкувань для Bijective

Загальнодоступні елементи

- virtual ∼Bijective ()
- virtual void get_position (vec4 position)

  Забирає поточну позицію об'єкта у просторі.
- virtual void set_position (vec4 position)

  Задає нову позицію об'єкта у просторі.
- virtual void get_rotation (vec3 rotation)

  Забирає поточний нахил об'єкта.
- virtual void set_rotation (vec3 rotation)

  Задає новий нахил об'єкта.

- virtual void get_animation (vec3 rotation)

  Забирає поточну анімацію обернення об'єкта.
- virtual void set_animation (vec3 rotation)

  Задає нову анімацію обернення об'єкта.
- virtual void get_mat4 (mat4 matrix)

  Забирає матрицю лінійних перетворень над об'єктом.
- virtual void translate (vec3 distances, float delta_time)

  Переміщує об'єкт у просторі.
- virtual void rotate (enum coord_enum coord, float delta_time)

  Обертає об'єкт.
- virtual void rotate_all (vec3 angles)

  Обернути об'єкт по всім осям.
- virtual void add_animation (vec3 angles, float delta_time)

  Додає швидкість анімації обертання об'єкту.

Детальний опис Інтерфейс до об'єктів, що можуть можуть бути переміщені та повернуті у просторі.

Див. визначення в файлі bijective.h, рядок 13

Конструктор(и)

~Bijective() `virtual Bijective::~Bijective ( ) [inline], [virtual]`

Див. визначення в файлі bijective.h, рядок 15

Опис методів компонент

add_animation()

```
virtual void Bijective::add_animation (
    vec3 angles,
    float delta_time ) [inline], [virtual]
```

Додає швидкість анімації обертання об'єкту.

Аргументи

`in` *angles* вектор швидкостей анімацій обертання по трьом осям

`in` *delta_time* скільки часу пройшло з останнього кадру

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 81

get_animation() `virtual void Bijective::get_animation (`

`vec3` *rotation* `) [inline], [virtual]`

Забирає поточну анімацію обернення об'єкта.

Аргументи

`out` *rotation* поточна анімація обернення об'єкта

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 43

get_mat4() `virtual void Bijective::get_mat4 (`

`mat4` *matrix* `) [inline], [virtual]`

Забирає матрицю лінійних перетворень над об'єктом.

Аргументи

`out` *matrix* матриця лінійних перетворень

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 54

get_position() `virtual void Bijective::get_position (`

`vec4` *position* `) [inline], [virtual]`

Забирає поточну позицію об'єкта у просторі.

Аргументи

`out` *position* поточна позицію об'єкта

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 21

get_rotation() `virtual void Bijective::get_rotation (`

`vec3` *rotation* `) [inline], [virtual]`

Забирає поточний нахил об'єкта.

Аргументи

`out` *rotation* поточний нахил об'єкта

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 32

rotate() `virtual void Bijective::rotate (`

`enum coord_enum` *coord,*

`float` *delta_time* `) [inline], [virtual]`

Обертає об'єкт.

Аргументи

`in` *coord* позначає координатну вісь навколо якої обертати

`in` *delta_time* скільки часу пройшло з останнього кадру

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 70

rotate_all() `virtual void Bijective::rotate_all (`

`vec3` *angles* `) [inline], [virtual]`

Обернути об'єкт по всім осям.

Аргументи

in *angles* вектор кутів у радіанах на кожну вісь

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 75

set_animation() `virtual void Bijective::set_animation ( vec3 rotation ) [inline], [virtual]`

Задає нову анімацію обернення об'єкта.

Аргументи

in *rotation* нова анімація обернення об'єкта.

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 48

set_position() `virtual void Bijective::set_position ( vec4 position ) [inline], [virtual]`

Задає нову позицію об'єкта у просторі.

Аргументи

in *position* нова позиція об'єкта у просторі

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 26

set_rotation() `virtual void Bijective::set_rotation ( vec3 rotation ) [inline], [virtual]`

Задає новий нахил об'єкта.

Аргументи

`in` *rotation* новий нахил об'єкта

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 37

translate() `virtual void Bijective::translate (`

`vec3` *distances,*

`float` *delta_time* `) [inline], [virtual]`

Переміщує об'єкт у просторі.

Переміщує об'єкт у просторі на відстані з аргументу `distances`, де кожне значення вектору позначає відстань відповідної осі.

Аргументи

`in` *distances* відстані переміщення по осям

`in` *delta_time* скільки часу пройшло з останнього кадру

Переозначається в Object і Camera.

Див. визначення в файлі bijective.h, рядок 64

Документація цього класу була створена з файлу:

- osdo/bijective.h

4.3 Клас Model

Інтерфейс до деякої моделі, яку можна відобразити. Схему успадкувань можна розглянути на рис. 4.7.

`#include <model.h>`

Схема успадкувань для Model

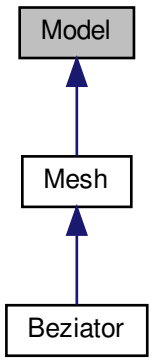

Рисунок 4.7 – Схема успадкувань для Model

Загальнодоступні елементи

- virtual ∼Model ()
- virtual void draw (Shader &shader, bool pre_generated=false)

  Відображує модель.
- virtual void generate (size_t d=8)

  Генерує деталізований меш моделі. Див. `Beziator::generate`
- virtual vector< Vertex > ∗ get_vertices ()

  Видає список вершин моделі.
- virtual void edit_panel ()

  Створює вікно редагування моделі.

Детальний опис Інтерфейс до деякої моделі, яку можна відобразити.

Див. визначення в файлі model.h, рядок 18

Конструктор(и)

∼Model() `Model::∼Model ( ) [virtual]`

Див. визначення в файлі model.cpp, рядок 4

Опис методів компонент

draw() `void Model::draw (`

`Shader & shader,`

`bool pre_generated = false )` `[virtual]`

Відображує модель.

Аргументи

shader Шейдер який використовуєтсья для відображення моделі.

pre_generated флаг, який позначає яким чином відображати модель.

Переозначається в Mesh і Beziator.

Див. визначення в файлі model.cpp, рядок 6

edit_panel() `void Model::edit_panel ( )` `[virtual]`

Створює вікно редагування моделі.

Див. визначення в файлі model.cpp, рядок 14

generate() `void Model::generate (`

`size_t d = 8 )` `[virtual]`

Генерує деталізований меш моделі. Див. `Beziator::generate`

Аргументи

d ступінь деталізації.

Переозначається в Beziator.

Див. визначення в файлі model.cpp, рядок 8

get_vertices() `vector< Vertex > * Model::get_vertices ( )` `[virtual]`

Видає список вершин моделі.

Повертає

Вказівник на поле `vertices`.

Переозначається в Beziator.

Див. визначення в файлі model.cpp, рядок 10

Документація цих класів була створена з файлів:

- osdo/model.h
- osdo/model.cpp

4.4 Структура Context

Контекст, який зберігає усі завантажені у пам'ять ресурси. Діаграму зв’язків можна розглянути на рис. 4.8.

```
#include <context.h>
```

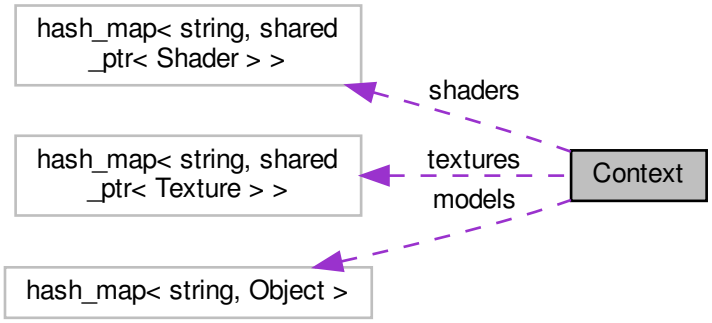

Рисунок 4.8 – Діаграма зв'язків класу Context

Загальнодоступні типи

- typedef hash_map< string, Object > Models

  Тип для зберігання моделей.

- typedef hash_map< string, shared_ptr< Texture > > Textures

  Тип для зберігання текстур.

Загальнодоступні елементи

- Context ()
- Models::iterator & next_active ()

  перехід до наступної моделі для редагування.
- void load_texture (const char ∗path)

  Завантажує текстуру у пам'ять.
- bool load_shader (const char ∗name, const Shader::shader_map &shaders)

  Завантажує та компілює шейдер.
- bool load_model (const string &path)

  Завантажує модель з поверхнями Безье

Загальнодоступні атрибути

- Models models

  Завантажені моделі.
- hash_map< string, shared_ptr< Shader > > shaders

  Зкомпіловані шейдери.
- Textures textures

  Завантажені текстури.
- Models::iterator active

  Вибрана модель для редагування.
- Textures::iterator active_texture

  Вибрана текстура для відображення.

Детальний опис Контекст, який зберігає усі завантажені у пам'ять ресурси.

Див. визначення в файлі context.h, рядок 26

Опис типів користувача

Models `typedef hash_map<string, Object> Context::Models`

Тип для зберігання моделей.

Див. визначення в файлі context.h, рядок 31

Textures `typedef hash_map<string, shared_ptr<Texture> > Context::Textures`

Тип для зберігання текстур.

Див. визначення в файлі context.h, рядок 35

Конструктор(и)

Context() `Context::Context ( )`

Див. визначення в файлі context.cpp, рядок 6

Опис методів компонент

load_model() `bool Context::load_model (`

`const string & path )`

Завантажує модель з поверхнями Безье

Аргументи

path шлях до файлу з моделлю

Повертає

статус успішності завантаження моделі

load_shader() `bool Context::load_shader (`

`const char * name,`

`const Shader::shader_map & shaders )`

Завантажує та компілює шейдер.

Аргументи

name назва шейдеру

shaders массив до файлів шейдеру

Повертає

статус успішності завантаження та компіляції шейдеру

Див. визначення в файлі context.cpp, рядок 26, також можна розглянути граф всіх викликів на рис. 4.9

Рисунок 4.9 – Граф всіх викликів Context::load_shader

load_texture() `void Context::load_texture (`

`const char * path )`

Завантажує текстуру у пам'ять.

Аргументи

`in` *path* шлях до файлу з текстурою

Див. визначення в файлі context.cpp, рядок 17

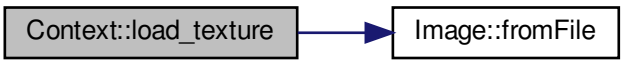

Рисунок 4.10 – Граф всіх викликів Context::load_texture

next_active() `Context::Models::iterator & Context::next_active ( )`

перехід до наступної моделі для редагування.

Повертає

ітератор моделі

Див. визначення в файлі context.cpp, рядок 10

Компонентні дані

active `Models::iterator Context::active`

Вибрана модель для редагування.

Див. визначення в файлі context.h, рядок 52

active_texture `Textures::iterator Context::active_texture`

Вибрана текстура для відображення.

Див. визначення в файлі context.h, рядок 56

models `Models Context::models`

Завантажені моделі.

Див. визначення в файлі context.h, рядок 39

shaders `hash_map<string, shared_ptr<Shader> > Context::shaders`

Зкомпіловані шейдери.

Див. визначення в файлі context.h, рядок 43

textures `Textures Context::textures`

Завантажені текстури.

Див. визначення в файлі context.h, рядок 47

Документація цих структур була створена з файлів:

- osdo/context.h
- osdo/context.cpp

4.5 Структура Scene

Сцена із об'єктами. Діаграму зв’язків можна розглянути на рис. 4.11.

`#include <scene.h>`

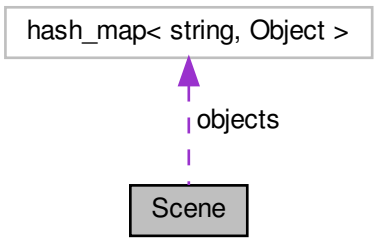

Рисунок 4.11 – Діаграма зв'язків класу Scene

Загальнодоступні елементи

- Scene (const Context::Models &objects)

  Конструктор, що створює об'єкти у сцені за заготовленими у аргументі `objects`

Загальнодоступні статичні елементи

- static shared_ptr< Scene > create (const Context::Models &objects)

Створює сцену

Загальнодоступні атрибути

- hash_map< string, Object > objects

  Об'єкти у сцені.

Детальний опис Сцена із об'єктами.

Див. визначення в файлі scene.h, рядок 16

Конструктор(и)

Scene() `Scene::Scene (`

`const Context::Models & objects )`

Конструктор, що створює об'єкти у сцені за заготовленими у агрументі `objects`

Аргументи

objects заготовлені об'єкти для додавання у сцену

Див. визначення в файлі scene.cpp, рядок 7

Опис методів компонент

create() `shared_ptr< Scene > Scene::create (`

`const Context::Models & objects ) [static]`

Створює сцену

Аргументи

objects заготовлені об'єкти для додавання у сцену

Повертає

Розумний вказівник на об'єкт сцени.

Див. визначення в файлі scene.cpp, рядок 10

Компонентні дані

objects `hash_map<string, Object> Scene::objects`

Об'єкти у сцені.

Див. визначення в файлі scene.h, рядок 20

Документація цих структур була створена з файлів:

- osdo/scene.h
- osdo/scene.cpp

4.6 Структура Vertex

Структура вершини, для передачі у відеокарту.

`#include <vertex.h>`

Загальнодоступні атрибути

- vec4 position

  Позиція вершини у просторі.
- vec3 normal

  Нормаль вершини.
- unsigned char color [4]

  Колір вершини.
- vec2 uv

  Координати вершини на текстурі.

Детальний опис Структура вершини, для передачі у відеокарту.

Див. визначення в файлі vertex.h, рядок 12

Компонентні дані

color `unsigned char Vertex::color[4]`

Колір вершини.

Див. визначення в файлі vertex.h, рядок 24

normal `vec3 Vertex::normal`

Нормаль вершини.

Див. визначення в файлі vertex.h, рядок 20

position `vec4 Vertex::position`

Позиція вершини у просторі.

Див. визначення в файлі vertex.h, рядок 16

uv `vec2 Vertex::uv`

Координати вершини на текстурі.

Див. визначення в файлі vertex.h, рядок 28

Документація цієї структури була створена з файлу:

- osdo/vertex.h

4.7 Файл osdo/beziator.h

Клас який зберігає та оброблює модель утворену через поверхні Безьє. Діаграма включених заголовочних файлів можна переглянути на рис. 4.12, а граф файлів, які включають файл на рис. 4.13

```
#include <EASTL/string.h>
#include "osdo.h"
#include "mesh.h"
```

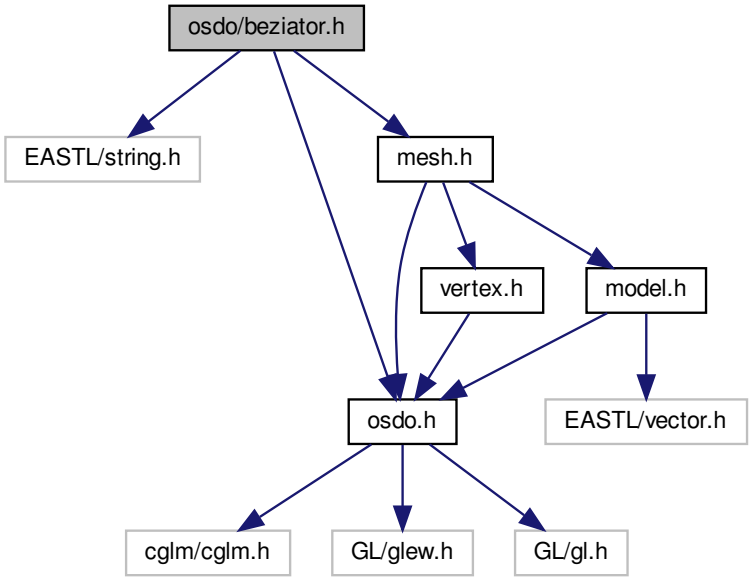

Рисунок 4.12 – Діаграма включених заголовочних файлів для beziator.h

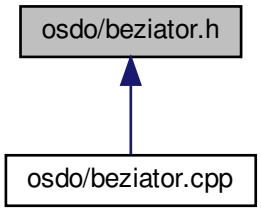

Рисунок 4.13 – Граф файлів, які включають файл beziator.h

Класи

- class Beziator

  Клас який зберігає та оброблює модель утворену через поверхні Безьє.

Визначення типів

- typedef GLuint surfacei_t[4][4]

  Набір індексів на вершини, що утворюють поверхню 4x4.

Детальний опис Клас який зберігає та оброблює модель утвореню через поверхні Безьє.

Див. визначення в файлі beziator.h

Опис визначень типів

surfacei_t `typedef GLuint surfacei_t[4][4]`

Набір індексів на вершини, що утворюють поверхню 4x4.

Див. визначення в файлі beziator.h, рядок 17

4.8 Файл osdo/bijective.h

Інтерфейс до об'єктів, що можуть можуть бути переміщені та повернуті у просторі. Діаграма включених заголовочних файлів можна переглянути на рис. 4.14, а граф файлів, які включають файл на рис. 4.15

```
#include "osdo.h"
```

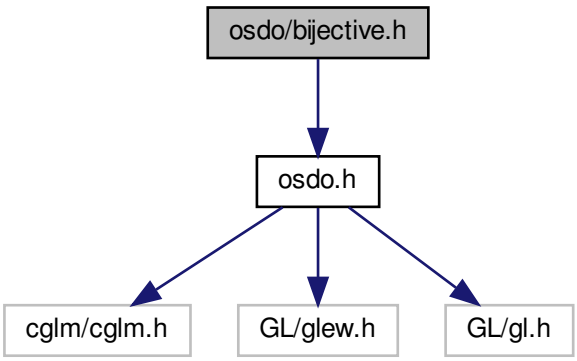

Рисунок 4.14 – Діаграма включених заголовочних файлів для bijective.h

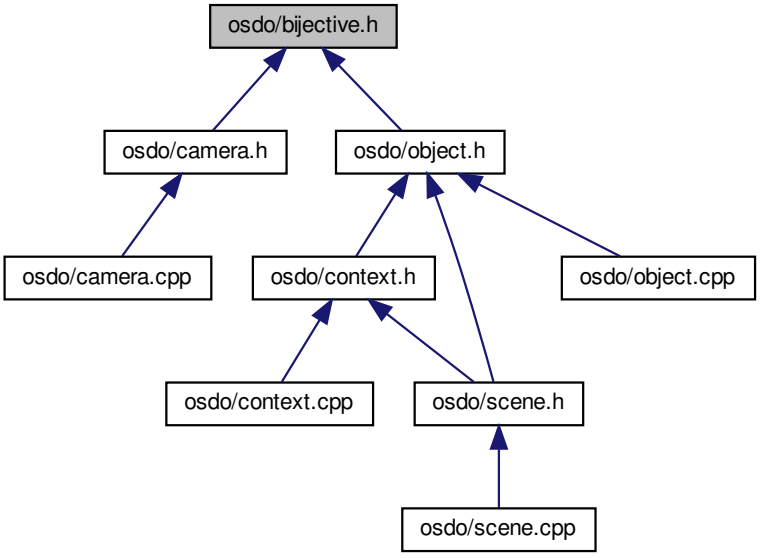

Рисунок 4.15 – Граф файлів, які включають файл bijective.h

Класи

- class Bijective

  Інтерфейс до об'єктів, що можуть можуть бути переміщені та по-

вернуті у просторі.

Детальний опис Інтерфейс до об'єктів, що можуть можуть бути переміщені та повернуті у просторі.

Див. визначення в файлі bijective.h

4.9 Файл osdo/context.h

Контекст, який зберігає усі завантажені у пам'ять ресурси. Діаграма включених заголовочних файлів можна переглянути на рис. 4.16, а граф файлів, які включають файл на рис. 4.17

```
#include "osdo.h"
#include "object.h"
#include "EASTL/hash_map.h"
#include "EASTL/string.h"
#include "EASTL/shared_ptr.h"
```

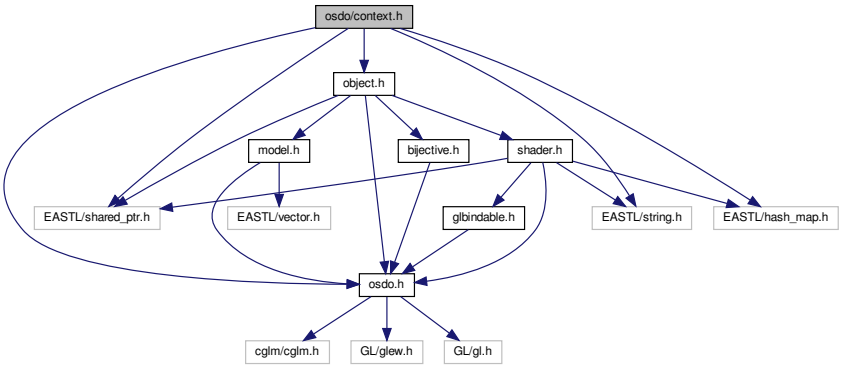

Рисунок 4.16 – Діаграма включених заголовочних файлів для context.h

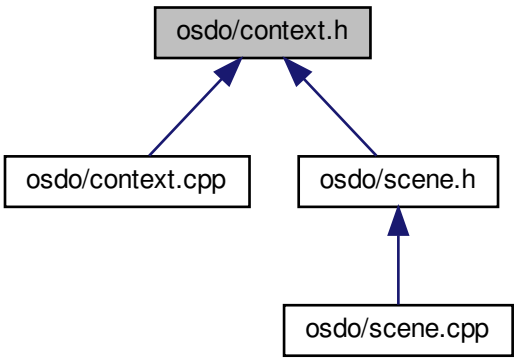

Рисунок 4.17 – Граф файлів, які включають файл context.h

Класи

- struct Context

  Контекст, який зберігає усі завантажені у пам'ять ресурси.

Детальний опис Контекст, який зберігає усі завантажені у пам'ять ресурси.

Див. визначення в файлі context.h

4.10 Файл osdo/context.cpp

```
#include "context.h"
#include "glbinder.h"
#include "image.h"
#include "texture.h"
```

Детальну діаграму включених заголовочних файлів для файлу context.cpp можна переглянути на рис. 4.18.

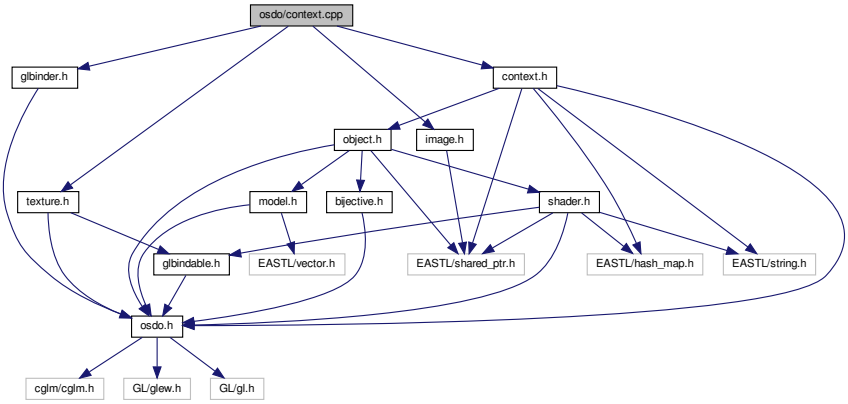

Рисунок 4.18 – Діаграма включених заголовочних файлів для context.cpp

4.11 Файл osdo/scene.h

Задає клас сцени із об'єктами. Діаграма включених заголовочних файлів можна переглянути на рис. 4.19, а граф файлів, які включають файл на рис. 4.20

```
#include "osdo.h"
#include "object.h"
#include "context.h"
```

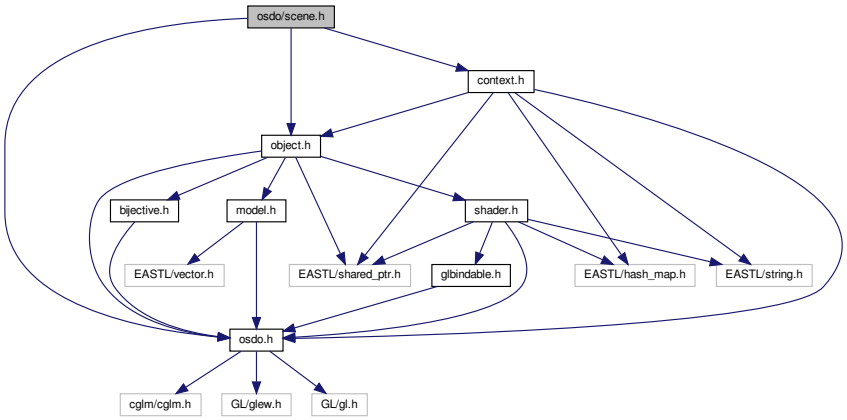

Рисунок 4.19 – Діаграма включених заголовочних файлів для scene.h

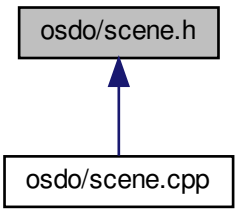

Рисунок 4.20 – Граф файлів, які включають файл scene.h

Класи

- struct Scene

  Сцена із об'єктами.

Детальний опис Задає клас сцени із об'єктами.

Див. визначення в файлі scene.h

4.12 Файл osdo/scene.cpp

```
#include "scene.h"
#include "conf.h"
#include "EASTL/algorithm.h"
```

Детальну діаграму включених заголовочних файлів для файлу scene.cpp можна переглянути на рис. 4.21.

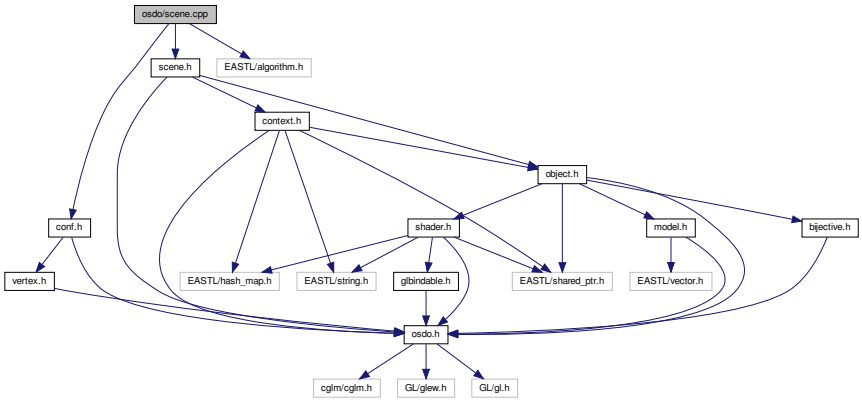

Рисунок 4.21 – Діаграма включених заголовочних файлів для scene.cpp

4.13 Файл osdo/model.h

Задає інтерфейс моделі, яку можна відобразити. Діаграма включених заголовочних файлів можна переглянути на рис. 4.22, а граф файлів, які включають файл на рис. 4.23

```
#include <EASTL/vector.h>
#include "osdo.h"
```

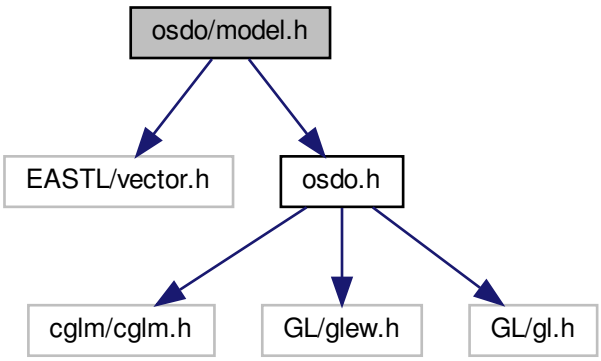

Рисунок 4.22 – Діаграма включених заголовочних файлів для model.h

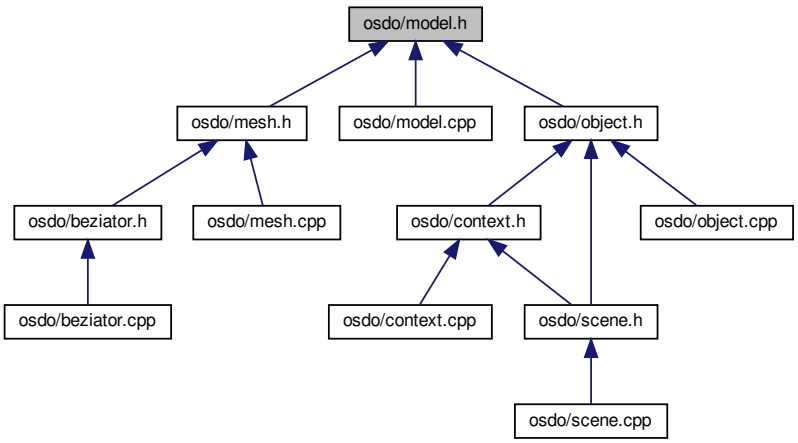

Рисунок 4.23 – Граф файлів, які включають файл model.h

Класи

- class Model

  Інтерфейс до деякої моделі, яку можна відобразити.

Детальний опис Задає інтерфейс моделі, яку можна відобразити.

Див. визначення в файлі model.h

4.14 Файл osdo/model.cpp

```
#include "model.h"
#include "vertex.h"
```

Детальну діаграму включених заголовочних файлів для файлу model.cpp можна переглянути на рис. 4.24.

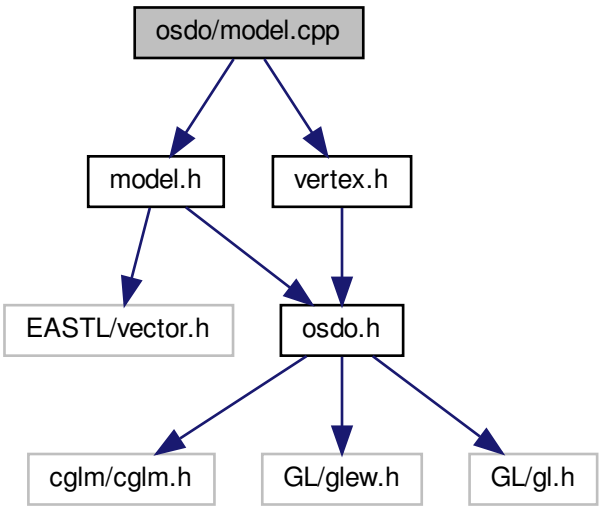

Рисунок 4.24 – Діаграма включених заголовочних файлів для model.cpp

4.15 Файл osdo/vertex.h

Задає структуру вершини. Діаграма включених заголовочних файлів можна переглянути на рис. 4.25, а граф файлів, які включають файл на рис. 4.26

```
#include "osdo.h"
```

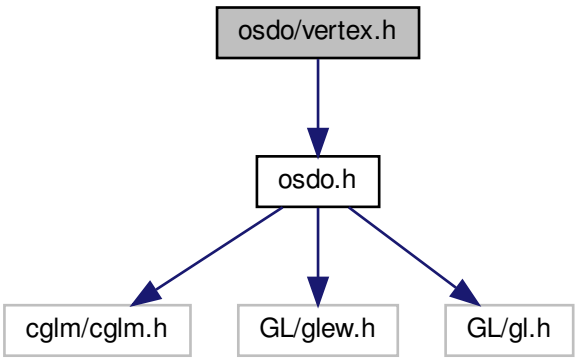

Рисунок 4.25 – Діаграма включених заголовочних файлів для vertex.h

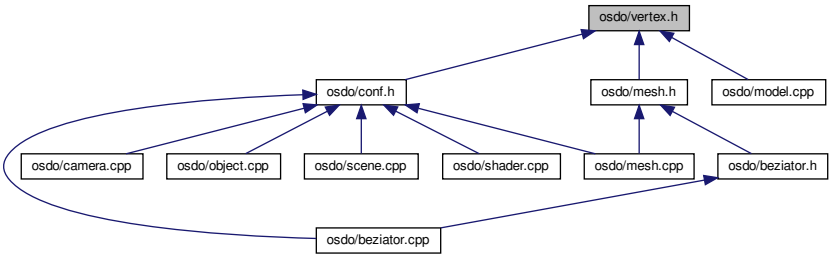

Рисунок 4.26 – Граф файлів, які включають файл vertex.h

Класи

- struct Vertex

  Структура вершини, для передачі у відеокарту.

Детальний опис Задає структуру вершини.

Див. визначення в файлі vertex.h

ВИСНОВКИ

В дипломній роботі було розглянуто проблему моделювання складних об’єктів у комп’ютерних іграх. Ця проблема виникає при потребі створити велику кількість дуже деталізованих об’єктів, що призводить до перевитрат ресурсів комп’ютеру або будь-якого ігрового пристрою.

Була розроблена програма, яка дозволялє створювати та переглядати такі складні об’єкти. Програма оптимізовано використовує пам’ять та графічний процесор комп’ютера при рендерінгу. Також у роботі була створена окрема бібліотека яка дозволяє завантажувати модель з диску та передавати у відеокарту, що у подальшому дозволяє використовувати її у ігрових рушіях.

У ході дипломної роботи були отримані такі результати:

- досліджено, які математичні моделі просторових об’єктів можуть використовуватися для нашої задачі;
- обрано сплаїни, як оптимальний спосіб подання інформації про об’єкти, які потрібно відобразити, загалом розглянуто поверхню Без’є з 16 контрольними точками;
- сформульовано вимоги до програми відображення 3D-об’єктів і виконано програмну реалізацію згідно цих вимог;
- розроблено інтерфейс для створення та редагування 3D-об’єктів за допомогою графічної бібліотеки ImGui;
- програмне забезпечення зроблене придатним для компіляції та роботи у операційних системах Linux та Windows, бінарні файли були опубліковані на сервісі github;
- розроблено теселяційний шейдер для генерації 3D-моделі за допомогою відеокарти;

- за допомогою інструменту Doxygen на основі коду програмного забезпечення надано його опис та створено схеми взаємодії його компонентів.

У подальшому це програмне забезпечення можна використовувати для створення повноцінний ігровий рушій.

Можливе подальше удосконалення розробленого програмного додатку. Наприклад, воно може бути доповнено кращою підтримкою текстур, створення формату для збереження на диск цілих сцен із об’єктами побудованими за допомогою поверхонь Без’є.

Повний вихідний код програмного забезпечення опубліковано на сервісі github із ліцензією LGPL, таким чином кожен розробник може доповнити існуючий код та удосконалити роботу програми.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційний сайт Міністерства освіти та науки України: http://mon.gov.ua/
2. СТП-02066747-009-01. Стандарт Дніпропетровського національного університету. Методика виконання випускних, курсових та дипломних проектів (робіт). Структура, правила оформлення та порядок узгодження і затвердження. Затверджено ректором ДНУ 31.10.2001 р.
3. СТП-02066747-010-01. Стандарт Дніпропетровського національного університету. Організація та проведення дипломування. Затверджено ректором ДНУ 1.11.2001 р.
4. http://www.dnu.dp.ua/docs/obgovorennya/Polozhennya_Antiplagiat_2016.doc
5. Порев. В.Н. Компъютерная графика – СПб: БХВ-Петербург, 2002 – 432 с.
6. Никулин Е. А. Компъютерная геометрия и алгоритмы машинной графики. - СПб: БХВ-Петербург, 2003 - 560 с.
7. Вычислительная геометрия и алгоритмы компьютерной графики. Работа с 3D-графикой средствами OpenGL: учеб. пособие / К. В. Рябинин; Перм. гос. нац. исслед. ун-т. – Пермь, 2017. – 100 с.
8. Р. Пэрент Компютерная анимация / Пер. с англ. – М: КУДИЦ-ОБРАЗ, 2004. – 560 с.
9. Tessellation - OpenGL Wiki – 2020 – Режим доступу: https://www.khronos.org/opengl/wiki/Tessellation – Tessellation
10. Visualization Library Reference Documentation – 2021 – Режим доступу:

https://www.visualizationlibrary.org/docs/2.1/html/index.html – Visualization Library

11. steps3D - Tutorial - Тесселяция в современном OpenGL – Alexey V. Boreskov 2003-2010 – Режим доступу:

http://steps3d.narod.ru/tutorials/tesselation-tutorial.html – Тесселяция в современном OpenGL

ДОДАТОК. ЛІСТИНГ ПРОГРАМИ

beziator.h

```
00001 /**
00002  * @file beziator.h
00003  * @brief                                          .
00004  */
00005 #ifndef BEZIATOR_H
00006 #define BEZIATOR_H
00007
00008 #include <EASTL/string.h>
00009 #include "osdo.h"
00010 #include "mesh.h"
00011
00012 using eastl::string;
00013
00014 /**
00015  * @brief                ,              4x4.
00016  */
00017 typedef GLuint surfacei_t[4][4];
00018
00019 /**
00020  * @brief                                          .
00021  */
00022 class Beziator : public Mesh {
00023 public:
00024     /**
00025      * @brief                                  .
00026      */
00027     typedef surfacei_t* surfaces_vector;
00028 protected:
00029     /**
00030      * @brief                            .
00031      */
00032     const string path;
00033     /**
00034      * @brief                CPU       .
00035      */
00036     Mesh mesh;
00037     //Mesh frame;
00038     //Mesh normals;
00039     /**
00040      * @brief       /       .
00041      */
00042     vector<Vertex> vertices;
00043     /**
00044      * @brief         ,                .
00045      *
00046      *                    16      ,
00047      *               4x4.
00048      *                   `surfaces_vector`:
00049      *
00050      *     surfacei_t *surfaces = reinterpret_cast<surfacei_t*>(indices.data());
00051      */
00052     vector<GLuint> indices;
00053 public:
00054     /**
00055      * @brief          Beziator,                        .
00056      *
00057      *   '                  `Beziator::init`
00058      *               '  .
00059      * @param path                           .
00060      */
00061     Beziator(const string& path);
00062     ~Beziator() override;
00063
00064     /**
00065      * @brief              '  .
00066      * @return    ,                    .
```

```
00067      */
00068     bool init();
00069
00070     /**
00071      * @brief           .
00072      *
00073      *             `pre_generated`
00074      *       ,         `false`,
00075      *           4x4,          `true`,
00076      *                              .
00077      * @param shader                                  .
00078      * @param pre_generated   ,                          .
00079      */
00080     void draw(Shader &shader, bool pre_generated) override;
00081
00082     /**
00083      * @brief                    .
00084      *
00085      *            `d`
00086      *   ,               8,
00087      * 8x8=64     .
00088      * @param d            .
00089      */
00090     void generate(size_t d = 8) override;
00091
00092     /**
00093      * @brief               ,           `path`.
00094      * @return               .
00095      */
00096     bool save();
00097
00098     /**
00099      * @brief                      ,                        .
00100      * @param i          .
00101      */
00102     void rotate(size_t i);
00103
00104     /**
00105      * @brief                .
00106      * @return            `vertices`.
00107      */
00108     vector<Vertex> *get_vertices() override;
00109 };
00110
00111 #endif // BEZIATOR_H
```

bezier.frag

```
00001 #version 420 core
00002 layout(location = 0) out vec4 FragColor;
00003
00004 struct Data {
00005     vec4 color;
00006     vec2 uv;
00007     vec3 normal;
00008     vec3 frag_pos;
00009 };
00010
00011 layout(location = 0) in Data data;
00012
00013 struct DirLight {
00014     vec3 direction;
00015
00016     vec3 ambient;
00017     vec3 diffuse;
00018     vec3 specular;
00019 };
00020
00021 uniform vec3 viewPos;
00022 uniform DirLight dirLight;
00023 uniform float materialShininess;
00024 uniform float alpha;
00025 uniform bool textured;
00026 uniform sampler2D textureSample;
00027
```

```
00028 // calculates the color when using a directional light.
00029 vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir, vec3 color)
00030 {
00031     vec3 lightDir = normalize(-light.direction);
00032     // diffuse shading
00033     float diff = max(dot(normal, lightDir), 0.0);
00034     // specular shading
00035     vec3 reflectDir = reflect(-lightDir, normal);
00036     float spec = pow(max(dot(viewDir, reflectDir), 0.0), materialShininess);
00037     // combine results
00038     vec3 ambient = light.ambient * color;
00039     vec3 diffuse = light.diffuse * diff * color;
00040     vec3 specular = light.specular * spec * color;
00041     return (ambient + diffuse + specular);
00042 }
00043
00044 void main()
00045 {
00046     vec3 norm = normalize(data.normal);
00047     vec3 viewDir = normalize(-viewPos - data.frag_pos);
00048     vec4 color = data.color;
00049     if (textured) {
00050         color = texture(textureSample, data.uv);
00051     }
00052     vec3 tmp = CalcDirLight(dirLight, norm, viewDir, vec3(color));
00053     FragColor = vec4(tmp, alpha);
00054 }
```

bezier.geom

```
00001 #version 420 core
00002 layout(triangles) in;
00003 layout(triangle_strip, max_vertices=16) out;
00004
00005 struct Data {
00006     vec4 color;
00007     vec2 uv;
00008     vec3 normal;
00009     vec3 frag_pos;
00010 };
00011
00012 in Data vertex[3];
00013 out Data geometry;
00014
00015 void main() {
00016     int i;
00017     for(i = 0; i < 16; i++) {
00018         gl_Position = gl_in[i].gl_Position;
00019         geometry.color = vertex[i].color;
00020         geometry.uv = vertex[i].uv;
00021         geometry.pos = vertex[i].pos;
00022         geometry.normal = vertex[i].normal;
00023         EmitVertex();
00024     }
00025     EndPrimitive();
00026 }
```

bezier.tesc

```
00001 #version 420 core
00002
00003 struct Data {
00004     vec4 color;
00005     vec2 uv;
00006     vec3 normal;
00007     vec3 frag_pos;
00008 };
00009
00010 layout(location = 0) in Data inData[];
00011 layout(location = 0) out Data outData[];
00012
00013 uniform int inner;
00014 uniform int outer;
00015
00016 layout(vertices = 16) out;
00017
```

```
00018 void main(void) {
00019     gl_TessLevelInner[0] = inner;
00020     gl_TessLevelInner[1] = inner;
00021     gl_TessLevelOuter[0] = outer;
00022     gl_TessLevelOuter[1] = outer;
00023     gl_TessLevelOuter[2] = outer;
00024     gl_TessLevelOuter[3] = outer;
00025 
00026     gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
00027     outData[gl_InvocationID].color = inData[gl_InvocationID].color;
00028     outData[gl_InvocationID].uv = inData[gl_InvocationID].uv;
00029     outData[gl_InvocationID].normal = inData[gl_InvocationID].normal;
00030     outData[gl_InvocationID].frag_pos = inData[gl_InvocationID].frag_pos;
00031 }
```

bezier.tese

```
00001 #version 420 core
00002 
00003 layout(quads, equal_spacing) in;
00004 
00005 struct Data {
00006     vec4 color;
00007     vec2 uv;
00008     vec3 normal;
00009     vec3 frag_pos;
00010 };
00011 
00012 layout(location = 0) in Data inData[];
00013 layout(location = 0) out Data outData;
00014 
00015 mat4  b = mat4 ( 1,  0,  0, 0,
00016                 -3,  3,  0, 0,
00017                  3, -6,  3, 0,
00018                 -1,  3, -3, 1);
00019 
00020 void main(void) {
00021     float x = gl_TessCoord.x;
00022     float y = gl_TessCoord.y;
00023     vec4 u = vec4 (1.0, x, x*x, x*x*x);
00024     vec4 v = vec4 (1.0, y, y*y, y*y*y);
00025     vec4 uu = vec4 (0, 1.0, 2*x, 3*x*x);
00026     vec4 vv = vec4 (0, 1.0, 2*y, 3*y*y);
00027 
00028     vec4 bu = b * u;
00029     vec4 bv = b * v;
00030     vec4 buu = b * uu;
00031     vec4 bvv = b * vv;
00032 
00033     mat4 pu[4], pv[4], cu, cv;
00034     for (int i = 0; i < 4; i++) {
00035         for (int j = 0; j < 4; j++) {
00036             pv[i][j] = gl_in[j*4 + i].gl_Position;
00037         }
00038     }
00039     for (int i = 0; i < 4; i++) {
00040         cv[i] = pv[i] * bv;
00041     }
00042 
00043     gl_Position = cv * bu;
00044 
00045     for (int i = 0; i < 4; i++) {
00046         for (int j = 0; j < 4; j++) {
00047             pu[i][j] = vec4(inData[i*4 + j].normal, 1);
00048             pv[i][j] = vec4(inData[j*4 + i].normal, 1);
00049         }
00050     }
00051     for (int i = 0; i < 4; i++) {
00052         cu[i] = pu[i] * bu;
00053         cv[i] = pv[i] * bv;
00054     }
00055     vec4 du = cv * buu, dv = cu * bvv;
00056     outData.normal = cross(vec3(du), vec3(dv));
00057 
00058     for (int i = 0; i < 4; i++) {
```

```
00059         for (int j = 0; j < 4; j++) {
00060             pv[i][j] = vec4(inData[j*4 + i].frag_pos, 1);
00061         }
00062     }
00063     for (int i = 0; i < 4; i++) {
00064         cv[i] = pv[i] * bv;
00065     }
00066     outData.frag_pos = vec3(cv * bu);
00067 
00068 
00069     /*for (int i = 0; i < 4; i++) {
00070         for (int j = 0; j < 4; j++) {
00071             pv[i][j] = vec4(inData[i*4 + i].uv, 0, 1);
00072         }
00073     }
00074     for (int i = 0; i < 4; i++) {
00075         cv[i] = pv[i] * bv;
00076     }
00077     outData.uv = vec2(cv * bu);*/
00078     outData.uv = vec2(x, y);
00079 
00080     outData.color = inData[0].color;
00081 }
```

bezier.vert

```
00001 #version 420 core
00002 layout (location = 0) in vec3 position;
00003 layout (location = 1) in vec3 normal;
00004 layout (location = 2) in vec4 color;
00005 layout (location = 3) in vec2 uv;
00006 
00007 struct Data {
00008     vec4 color;
00009     vec2 uv;
00010     vec3 normal;
00011     vec3 frag_pos;
00012 };
00013 
00014 layout(location = 0) out Data data;
00015 
00016 uniform mat4 model;
00017 uniform mat4 camera;
00018 uniform mat4 projection;
00019 
00020 void main()
00021 {
00022     mat4 trans = projection * camera * model;
00023     vec4 pos = trans * vec4(position, 1.0);
00024     gl_Position = pos;
00025     data.color = color;
00026     data.uv = uv;
00027     data.frag_pos = vec3(model * vec4(position, 1.0));
00028     data.normal = mat3(transpose(inverse(model))) * vec3(normal);
00029 }
```

bijective.h

```
00001 /**
00002  * @file bijective.h
00003  * @brief          '   ,                                  .
00004  */
00005 #ifndef BIJECTIVE_H
00006 #define BIJECTIVE_H
00007 
00008 #include "osdo.h"
00009 
00010 /**
00011  * @brief          '   ,                                  .
00012  */
00013 class Bijective {
00014 public:
00015     virtual ~Bijective() {}
00016 
00017     /**
00018      * @brief                '         .
```

```
00019      * @param[out] position           '
00020      */
00021     virtual void get_position(vec4 position) {}
00022     /**
00023      * @brief              '         .
00024      * @param[in] position          '
00025      */
00026     virtual void set_position(vec4 position) {}
00027
00028     /**
00029      * @brief                '  .
00030      * @param[out] rotation            '
00031      */
00032     virtual void get_rotation(vec3 rotation) {}
00033     /**
00034      * @brief             '  .
00035      * @param[in] rotation          '
00036      */
00037     virtual void set_rotation(vec3 rotation) {}
00038
00039     /**
00040      * @brief                       '  .
00041      * @param[out] rotation                  '
00042      */
00043     virtual void get_animation(vec3 rotation) {}
00044     /**
00045      * @brief                    '  .
00046      * @param[in] rotation                '  .
00047      */
00048     virtual void set_animation(vec3 rotation) {}
00049
00050     /**
00051      * @brief                           '   .
00052      * @param[out] matrix
00053      */
00054     virtual void get_mat4(mat4 matrix) {}
00055
00056     /**
00057      * @brief        '         .
00058      *
00059      *        '                        `distances`,
00060      *                                   .
00061      * @param[in] distances
00062      * @param[in] delta_time
00063      */
00064     virtual void translate(vec3 distances, float delta_time) {}
00065     /**
00066      * @brief       '  .
00067      * @param[in] coord
00068      * @param[in] delta_time
00069      */
00070     virtual void rotate(enum coord_enum coord, float delta_time) {}
00071     /**
00072      * @brief       '          .
00073      * @param[in] angles
00074      */
00075     virtual void rotate_all(vec3 angles) {}
00076     /**
00077      * @brief                       '  .
00078      * @param[in] angles
00079      * @param[in] delta_time
00080      */
00081     virtual void add_animation(vec3 angles, float delta_time) {}
00082 };
00083
00084 #endif // BIJECTIVE_H
```

context.h

```
00001 /**
00002  * @file context.h
00003  * @brief     ,                    '       .
00004  */
00005 #ifndef CONTEXT_H
00006 #define CONTEXT_H
```

```
00007
00008 #include "osdo.h"
00009
00010 #include "object.h"
00011 #include "EASTL/hash_map.h"
00012 #include "EASTL/string.h"
00013 #include "EASTL/shared_ptr.h"
00014 using eastl::hash_map;
00015 using eastl::string;
00016 using eastl::shared_ptr;
00017 using eastl::pair;
00018 using eastl::make_shared;
00019
00020 class Shader;
00021 class Texture;
00022
00023 /**
00024  * @brief     ,                    '      .
00025  */
00026 struct Context
00027 {
00028     /**
00029      * @brief                .
00030      */
00031     typedef hash_map<string, Object> Models;
00032     /**
00033      * @brief                .
00034      */
00035     typedef hash_map<string, shared_ptr<Texture>> Textures;
00036     /**
00037      * @brief           .
00038      */
00039     Models models;
00040     /**
00041      * @brief            .
00042      */
00043     hash_map<string, shared_ptr<Shader>> shaders;
00044     /**
00045      * @brief            .
00046      */
00047     Textures textures;
00048
00049     /**
00050      * @brief                  .
00051      */
00052     Models::iterator active;
00053     /**
00054      * @brief                    .
00055      */
00056     Textures::iterator active_texture;
00057
00058 public:
00059     Context();
00060
00061     /**
00062      * @brief                          .
00063      * @return
00064      */
00065     Models::iterator &next_active();
00066
00067     /**
00068      * @brief               '  .
00069      * @param[in] path
00070      */
00071     void load_texture(const char *path);
00072
00073     /**
00074      * @brief                  .
00075      * @param name
00076      * @param shaders
00077      * @return
00078      */
00079     bool load_shader(const char *name, const Shader::shader_map& shaders);
```

```
00080
00081     /**
00082      * @brief
00083      * @param path
00084      * @return
00085      */
00086     bool load_model(const string& path);
00087 };
00088
00089 #endif // CONTEXT_H
```

context.cpp

```
00001 #include "context.h"
00002 #include "glbinder.h"
00003 #include "image.h"
00004 #include "texture.h"
00005
00006 Context::Context() : active(models.end()), active_texture(textures.end()) {
00007
00008 }
00009
00010 Context::Models::iterator &Context::next_active() {
00011     if (active == models.end()) {
00012         active = models.begin();
00013     } else active++;
00014     return active;
00015 }
00016
00017 void Context::load_texture(const char *path) {
00018     Image img = Image::fromFile(path);
00019     if (img.data) {
00020         auto tex = make_shared<Texture>();
00021         tex->update(img);
00022         textures[path] = tex;
00023     }
00024 }
00025
00026 bool Context::load_shader(const char *name, const Shader::shader_map& shaders) {
00027     auto shader = Shader::create(shaders);
00028     if (!shader)
00029         return false;
00030     this->shaders[string(name)] = shader;
00031     return true;
00032 }
```

scene.h

```
00001 /**
00002  * @file scene.h
00003  * @brief               '    .
00004  */
00005 #ifndef SCENE_H
00006 #define SCENE_H
00007
00008 #include "osdo.h"
00009
00010 #include "object.h"
00011 #include "context.h"
00012
00013 /**
00014  * @brief        '    .
00015  */
00016 struct Scene {
00017     /**
00018      * @brief  '        .
00019      */
00020     hash_map<string, Object> objects;
00021
00022     /**
00023      * @brief       ,         '                          `objects`
00024      * @param objects         '
00025      */
00026     Scene(const Context::Models& objects);
00027
00028     /**
```

```
00029      * @brief
00030      * @param objects         '
00031      * @return               '     .
00032      */
00033     static shared_ptr<Scene> create(const Context::Models& objects);
00034 };
00035
00036 #endif // SCENE_H
```

scene.cpp

```
00001 #include "scene.h"
00002 #include "conf.h"
00003 #include "EASTL/algorithm.h"
00004 using eastl::transform;
00005 using eastl::make_shared;
00006
00007 Scene::Scene(const Context::Models &objects) : objects(objects) {
00008 }
00009
00010 shared_ptr<Scene> Scene::create(const Context::Models &objects)
00011 {
00012     return make_shared<Scene>(objects);
00013 }
```

model.h

```
00001 /**
00002  * @file model.h
00003  * @brief              ,             .
00004  */
00005 #ifndef MODEL_H
00006 #define MODEL_H
00007
00008 #include <EASTL/vector.h>
00009 #include "osdo.h"
00010
00011 struct Vertex;
00012 class Shader;
00013 using eastl::vector;
00014
00015 /**
00016  * @brief                ,             .
00017  */
00018 class Model {
00019 public:
00020     virtual ~Model();
00021     /**
00022      * @brief          .
00023      * @param shader                                  .
00024      * @param pre_generated   ,                            .
00025      */
00026     virtual void draw(Shader &shader, bool pre_generated = false);
00027     /**
00028      * @brief                    .
00029      *   . `Beziator::generate`
00030      * @param d            .
00031      */
00032     virtual void generate(size_t d = 8);
00033     /**
00034      * @brief                .
00035      * @return            `vertices`.
00036      */
00037     virtual vector<Vertex> *get_vertices();
00038     /**
00039      * @brief                    .
00040      */
00041     virtual void edit_panel();
00042 };
00043
00044 #endif // MODEL_H
```

model.cpp

```
00001 #include "model.h"
00002 #include "vertex.h"
00003
00004 Model::~Model() {}
```

```
00005
00006 void Model::draw(Shader &, bool pre_generated) {}
00007
00008 void Model::generate(size_t d) {}
00009
00010 vector<Vertex> *Model::get_vertices() {
00011     return nullptr;
00012 }
00013
00014 void Model::edit_panel() {}
```

vertex.h

```
00001 /**
00002  * @file vertex.h
00003  * @brief     c          .
00004  */
00005 #ifndef VERTEX_H
00006 #define VERTEX_H
00007 #include "osdo.h"
00008
00009 /**
00010  * @brief           ,                .
00011  */
00012 struct Vertex {
00013     /**
00014      * @brief                .
00015      */
00016     vec4 position;
00017     /**
00018      * @brief          .
00019      */
00020     vec3 normal;
00021     /**
00022      * @brief         .
00023      */
00024     unsigned char color[4];
00025     /**
00026      * @brief                    .
00027      */
00028     vec2 uv;
00029 };
00030
00031 #endif // VERTEX_H
```