ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ОЛЕСЯ ГОНЧАРА
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
КАФЕДРА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

КУРСОВА РОБОТА
ЗА ФАХОВИМ СПРЯМУВАННЯМ

на тему: Комп'ютерне моделювання поведінки динамічних об'єктів в комп'ютерних іграх

Освітньо-професійна програма: Комп'ютерне моделювання та технології програмування

Спеціальність: 113 Прикладна математика

Галузь знань: 11 Математика і статистика

Рівень вищої освіти: перший (бакалаврський)

Студента 4 курсу групи ПА–17–2
Панасенко Єгор Сергійович
(прізвище та ініціали)

Керівник: канд. фіз.-мат. наук
Степанова Наталія Іванівна
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Кількість балів: _____

Національна шкала: _____

Члени комісії:

_____ Зайцева Т.А.
(підпис) (прізвище та ініціали)

_____ Сердюк М.Є.
(підпис) (прізвище та ініціали)

_____ Дзюба П.А.
(підпис) (прізвище та ініціали)

м. Дніпро, 2020 р.

РЕФЕРАТ

Курсова робота: __ с., _ рис., _ джерел, _ додаток.

Об’єкт дослідження: динамічні гладкі об’єкти у комп’ютерній графіці.

Мета роботи: Розробити програмне забезпечення для моделювання зіткнень динамічних гладких об’єктів у режимі реального часу.

Одержані висновки та їх новизна: запропоновані підходи до моделювання динамічних гладких об’єктів та розроблено програмне забезпечення для відображення цих об’єктів.

Результати досліджень можуть бути використані для розробки комп’ютерних ігор та симуляції бою у єдиноборствах для перевірки ефективності прийомів.

Перелік ключових слів: OPENGL, КРИВІ БЕЗЬЄ, ПОВЕРХНІ БЕЗЬЄ, C, GLM, NUKLEAR, МАТЕМАТИЧНА МОДЕЛЬ ПОВЕРХНІ.

ЗМІСТ

ВСТУП

У сучасному світі спостерігається неймовірний приріст потужності обчислювальної техніки і розробники ігор намагаються використати цю потужність найбільш ефективно з метою отримання графіки, найбільш схожої на реальний світ.

Для досягнення найбільшого задоволення розробники також намагаються створити якомога більше ігрових об’єктів та приголомшливих ефектів, що супроводжується значним споживання дискового простору й оперативної пам’яті. Тому завжди є актуальним питання розробки більш ефективних методів моделювання ігрових об’єктів, які б використовували менше обчислювальних ресурсів.

Деякий час тому, коли потужність обчислювальних засобів достатньо виросла, було створено векторний формат зберігання зображень SVG, призначений для зберігання як статичної, так і анімованої двомірної графіки. Для побудови зображень SVG формат використовує криві Безьє, які у свою чергу є окремим випадком B-сплайна. Формат SVG підтримується всіма сучасними браузерами для настільних і мобільних пристроїв.

Як відомо, векторним форматам притаманні гарна масштабованість й незначне використання дискового простору за умови, що зображення складається з невеликої кількості простих елементів. З іншого боку, векторні формати мають також і недоліки: у порівнянні з растровими аналогами для побудови векторних зображень використовується більше процесорного часу, а складні зображення починають використовувати більше дискового простору ніж аналогічні растрові.

У трьохвимірному просторі найбільш розповсюдженим форматом є OBJ - гнучкий формат, що дозволяє створювати об’єкти за допомогою різних спосо-

бів, у тому числі з використанням кривих Безьє і B-сплайнів.

Таким чином вже існують формати, які дозволяють зберігати об’єкти компактно, забезпечувати їх легку масштабованість. Але у реальному ігровому процесі, де об’єкти сцени постійно взаємодіють один з одним, виникає проблема: як визначити коли об’єкти перетнулися або зіткнулися, під яким кутом це сталося?

У даному дослідженні поставлено задачу розробки ефективного алгоритму, що дозволить у режимі реального часу визначати місце та кут зіткнення декількох об’єктів, побудованих за допомогою сплайн-технологій. Також потрібно виконати програмну реалізацію, яка цей алгоритм використовує.

Для розробки програмного продукту обрано крос-платформовий програмний інтерфейс OpenGL, що забезпечує незалежність програмного додатку від операційної системи.

ПОСТАНОВКА ЗАДАЧІ

Метою цієї курсової роботи є розробка програмного забезпечення для моделювання зіткнень динамічних гладких об’єктів у режимі реального часу. У загальному випадку розглядаються динамічні об’єкти створені за допомогою поверхонь Безьє. Для досягнення поставленої мети були поставлені наступні задачі:

- дослідити математичні моделі динамічних гладких об’єктів та вибрати оптимальний спосіб для розв’язання поставленої задачі,
- розробити програмне забезпечення для відображення динамічних об’єктів,
- розробити програмне забезпечення для створення або редагування динамічних об’єктів.

1. АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1.1. Математичні моделі поверхонь та об’єктів

Комп’ютерна 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 цього для отримання образів просторових форм на площині екрану необхідно використовувати ще одне геометричне перетворення - проеціювання. Існує така класифікація моделей поверхонь і об’єктів:

Існує така класифікація поверхонь:

- Каркасні - на екран друкуються не всі точки поверхні, а лише невелика кількість, що реалістично передає характер поверхні. З цих точкок будуємо систему ліній на поверхні і отримаємо каркас.
- Точкові - на екран друкуються точки з відповідним забарвленням.
- Кінематичні - поверхня будується неперервним рухом у просторі лінії по деякій траєкторії.
- Кусочні, яка при обмеженому набору даних будує поверхню у якій присутні розриви ти злами.
- Сплайнові, використовується для побудови гладких поверхонь на основі обчислення координат за допомогою СЛАУ, або його модифікацію.

- Фрактальні, використовує властивість об’єктів до самоподібності в залежності від масштабу.
- Графічні використовується у разі, якщо не можливо виділити деякий закон для побудови і поверхня заповнюється деякими дискретними елементами, названими вокселами.

1.2. Дослідження математичних моделей динамічних гладких об’єктів

Серед існуючи математичних моделей поверхонь та об’єктів було вибрано саме поверхню Безьє, яка є частинним випадком B-сплайнів.

1.2.1. Сплайнові криві

Розглянемо загальний випадок спланової кривої. Нехай існують вектори $u_i, i = \overline{0, n}$, ці вектори називаються вузлові точки, довжина вектора залежить від розмірності простору у якому ми працюємо, для двомірного простору $u_i = [x_i\ y_i]$, для трьохмірного $u_i = [x_i\ y_i\ z_i]$. Надалі будемо розглядати трьохмірний простір. Нехай ці вузлові точки пронумеровані у порядку з’єднання кривої.

Існують координатний вид кривої

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

та векторний

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

Загалом на основі властивостей сплайнів будується СЛАУ

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

1.2.2. Крива Безьє

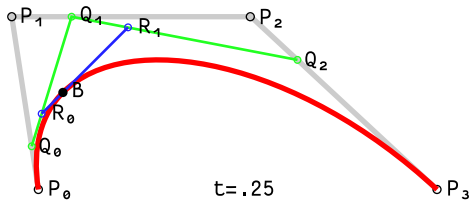

Рис. 1. Будування кривої Безьє

Розглянемо графічний спосіб побудови кубічної кривої Безьє, який проілюстрований на рис. 1. Нехай у нас є деяке дійсне число $t = 0.25$, загалом $t \in [0,1]$, і $P_i, i = \overline{0,3}$ - це контрольні точки кривої Безьє на першому кроці ми відкладаємо 0.25 шляху на відрізках P_iP_{i+1}, отримаємо точки $Q_j, j = \overline{0,2}$, на наступному кроці робимо теж саме і отримаємо R_0 та R_1, на даному етапі 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 вибирати t, наприклад з кроком 0.1, та з'єднувати у відрізки, то ми отримаємо ламану і з зменшенням кроку ми будемо отримувати ламану більш схожу 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:

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

Тепер ми можемо записати формулу у матричному вигляді:

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

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

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

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

1.2.3. Поверхня Безьє

Розглянемо графічний спосіб побудови кубічної поверхні Безьє з 16 контрольними точками. Спочатку ми 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

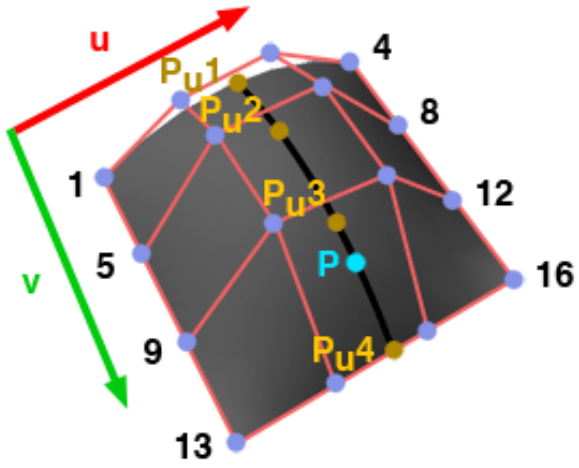

Рис. 2. Будування поверхні Безьє

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 данній точці.

Крива Безьє задається формулою:

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

1.3. Бібліотека Visualization Library

У ході роботи було знайдено таку бібліотеку, як Visualization Library. Це бібліотека написана на мові C++ і може використовуватись для графіки у 2D або 3D. Вона дозволяє моделювати різні види поверхонь, фрактали, та багато іншого.

Зробимо аналіз цієї бібліотеки.

- Бібліотека написана на мові C++, тобто це звужує кількість мов, що можуть використовувати цю бібліотеку.

- 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, бібліотека 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.

2. 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, Nuklear, GLFW, uthash та прогрмний інтерфейс OpenGL. Перші біблиотеки CGLM та Nuklear використовують ліцензію MIT, бібліотека GLFW використовує ліцензію ZLib, а uthash – 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 програмуванні, 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.

Це ще одна причина вибору мови С, яку можна обернути у більшість мов програмування і таким чином програмне забезпечення зможуть використати і розробники, які не знають С, 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 програмне забезпечення має дуже малий розмір зкомпільованої програми, що підвищує легкість розповсюдження. І це третя причина чому було вибрано саме мову програмування С, яка має дуже простий синтаксис та просту структуру. А також саме тому було вибрано саме такий набір бібліотек, а загалом графічну бібліотеку Nuklear, яка має досить невеликий обсяг коду, приблизно 30 тисяч строк коду разом з коментарями.

3. ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Було розроблено програмне забезпечення на мові Сі 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.

- Nuklear - бібліотека для графічного інтерфейсу написана на мові Сі. 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.

- uthash - біблиотека для динамічних масивів та хеш таблиць. Використовує ліцензію BSD.

  Завдяки тому що на мові Сі відсутні стандартні структури даних, ця бібліотека дозволяє створювати динамічні масиви, хеш таблиці, та багато іншого для різних типів даних та структур, завдяки цьому ми можемо

працювати так само як і на більшості мовах програмування та навіть з більшою швидкістю. Аналогічно як і бібліотека Nuklear, бібліотека uthash має дуже малу кодову базу, варіативність типів реалізується за допомогою макросів, що також зменшує кількість коду і відповідає принципу програмування DRY (акронім “don’t repeat yourself”).

Код програмного забезпечення складається з таких компонентів:

- Вихідний код програми, який зберігається у директорії “src” (скорочено “source”). Загалом тут знаходяться файли заголовків з розширенням “.h” та з реалізацією з розширенням “.c”, кожен файл заголовків у цій директорії утворює окремий модуль, подібний класу у парадигмі об’єктно орієнтованого програмування.
- Ресурси програми, які зберігаються у директорії “res” (скорочено “resource”). Тут знаходяться шейдери та тестові моделі чайнику 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 ресурси у директорії “share/osdo”, ця структура директорії Unix подібна.

4. ОПИС МОДУЛІВ ПРОГРАМИ

Програма складається з декількох модулів, загалом вони реалізуються деяку структуру за поняттям дуже подібною, що у парадигмі об’єктно орієнтованого програмування (далі ООП), навіть використовується принципи поліморфізму, як наприклад у модулях “Bijective” та “Model”, та інкапсуляції, як наприклад у модулі “Window”.

- Модуль “App” (файли “app.h” та “app.c”) — компонує разом сцену, вікно, шейдери та об’єкти. Тут реалізується основна поведінка всієї програми, тобто головний виконуваний цикл.

- Модуль “Beziator” (файли “beziator.h” та “beziator.c”) — читає з файлу, назва якого була вказана у конструкторі, модель, побудовану з поверхонь Безьє. Модуль зберігає саму модель, дозволяє її змінювати та згенерувати з моделі новий меш, який відобразиться на екрані. Також у цьому модулі однією константою зафіксовано ступінь деталізації об’єкту.

- Модуль “Bijective” (файли “bijective.h” та “bijective.c”) — модуль за поняттям подібний до абстрактного класу з парадигми програмування ООП. Дозволяє трансформувати деякий об’єкт у просторі, використовується для того щоб переміщувати, обертати, анімувати камеру або об’єкт. Таким чином є батьківським модулем до модуля “Camera” та “Object”. Назва означає що об’єкт бієктивний відносно афінних перетворень.

- Модуль “Camera” (файли “camera.h” та “camera.c”) — реалізує камеру, зберігає матрицю повороту камери та вектор позиції камери, та дозволяє переміщуватись зміною вектора позиції та повертатись за допомогою афінних перетворень матриці.

- Модуль “Conf” (файли “conf.h”) — зберігає необхідні константи для робо-

ти програми, а також початкові значення для об’єктів та деякий приклад мешу куба.

- Модуль “Mesh” (файли “mesh.h” та “mesh.c”) — модуль зберігає меш об’єкту, та індекси які позначають у OpenGL індекс масиву вершин та сторін. Також модуль дозволяє відображати на екран меш, причому можна зазначити яким чином відображати, як об’єкт, або тільки каркас або вершини. Залежний від файлу заголовків <GL/glew.h>.

- Модуль “Model” (файли “model.h” та “model.c”) — реалізує абстрактного клас, який дозволяє зберігати повну модель об’єкту для відображення. Модуль є батьківським до модуля “Beziator” та “Mesh”, що дозволяє відображати на екран як прості поліноми, так і модель задану як поверхня Безьє.

- Модуль “Nkglfw” (файли “nkglfw.h” та “nkglfw.c”) — модуль потрібний для відображення графічного інтерфейсу Nuklear, історично склалося що nkglfw реалізовувався тільки у зв’язці Nuklear/GLFW/OpenGL, так і виникла назва, пізніше модуль дозволив працювати з власноруч реалізованим вікном з модуля “Window”, що дозволяє пізніше підключити різні вікна, наприклад SDL до цього самого модуля.

- Модуль “Object” (файли “object.h” та “object.c”) — утворює об’єкт з моделі (модуль “Model”), який можна переміщувати, збільшувати, та повертати.

- Модуль “Osdo” (файли “osdo.h” та “osdo.c”) — задається порядок імпортування бібліотек OpenGL, це пов’язано з тим, що OpenGL дуже строго відноситься до порядку імпортування, наприклад файл заголовків <GL/glew.h> повинен бути імпортований до <GL/gl.h>. Також модуль реалізує макрос ітерації по динамічному масиву з біблиотеки uthash.

- Модуль “Scene” (файли “scene.h” та “scene.c”) — зберігаються усі перетворення об’єктів та камери, таким чином можна скиданням на параметри за замовчанням ми приводимо сцену до початкового стану.

- Модуль “Shader” (файли “shader.h” та “shader.c”) — реалізує клас обгортку для завантаження з файлу та компіляції шейдеру. Залежний від файлу заголовків <GL/glew.h>.

- Модуль “Window” (файли “window.h” та “window.c”) — клас обгортка на вікно з бібліотеки “GLFW”, модуль приховує реалізацію “GLFW”, таким чином жоден з модулів не знає об існуванні “GLFW” і це дозволяє у подальшому полегшити процес впровадження нового типу вікна, такого як в бібліотеці “SDL”, наприклад.

5. ОГЛЯД РОБОТИ ПРОГРАМИ

Розглянемо роботу програми.

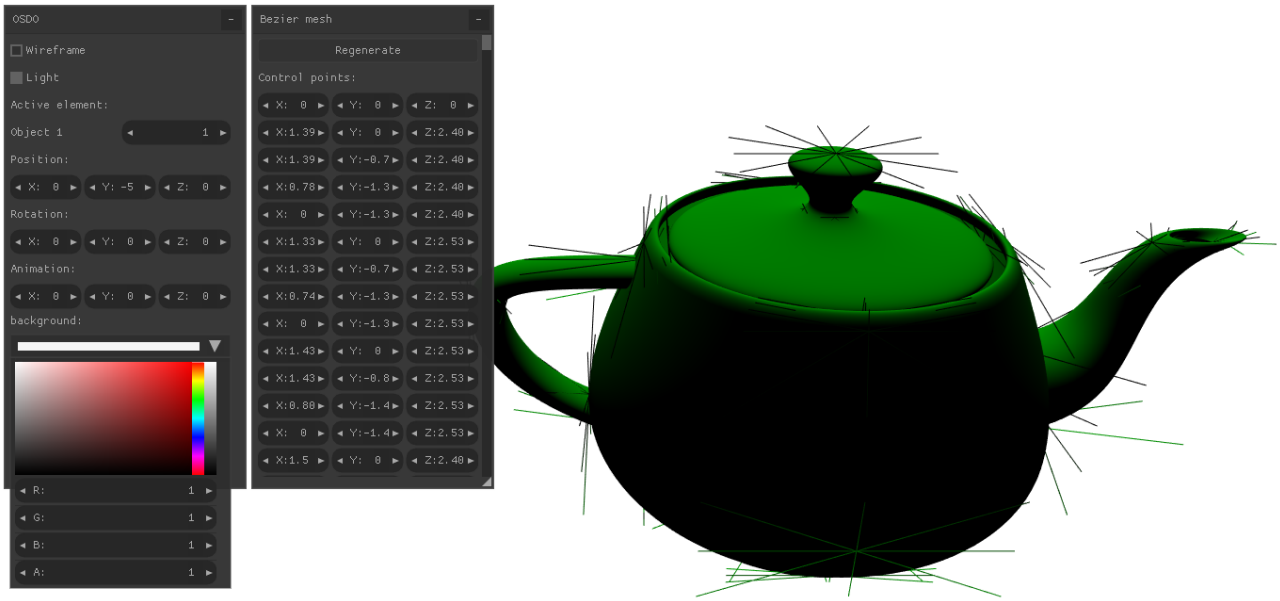

Рис. 3. Приклад інтерфейсу

На мал 3 ми бачимо інтерфейс програми, перша панель складається з таких елементів:

- перехід в режим каркасу (англ. Wireframe),
- перехід у режим світла (англ. Light),
- переходу між об’єктами, тобто вибираємо активний об’єкт над яким будуть відбуватися операції,
- позиція об’єкта (англ. Position), що задається вектором у просторі,
- кут обертання (англ. Rotation) у радіанах навколо відповідної осі глобальної системи координат
- швидкість обертання, або анімація (англ. Animation), навколо відповідної осі,

- колір фону (англ. Background).

Наступна панель складається з кнопки перегенерування об’єкту, списку контрольних точок та їх координат, далі іде задавання індексів контрольних точок у поверхні.

На малюнках 4, 5, 6 ми бачимо стандартну модель чайнику в різних режимах. Ця модель чайнику стандатрний приклад складної гладкої фігури і називається чайник із Юти (англ. The Utah Teapot) або чайник Н’юелла.

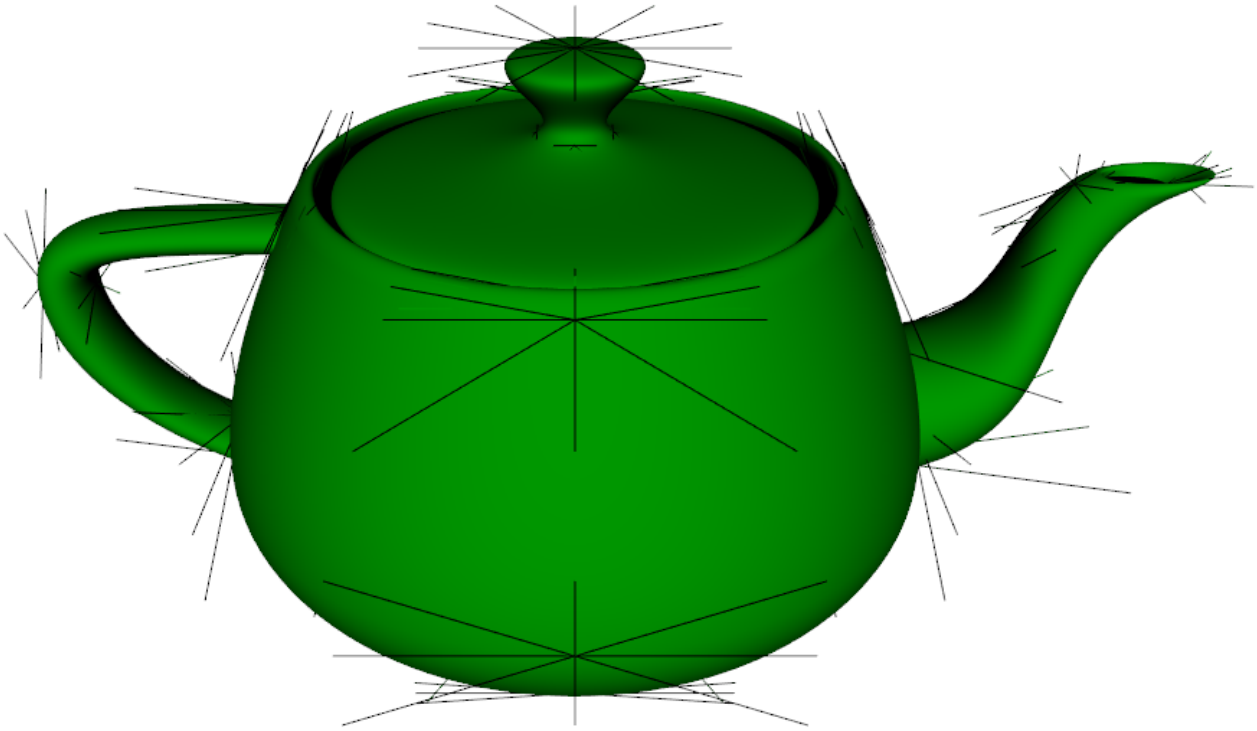

Рис. 4. Чайник

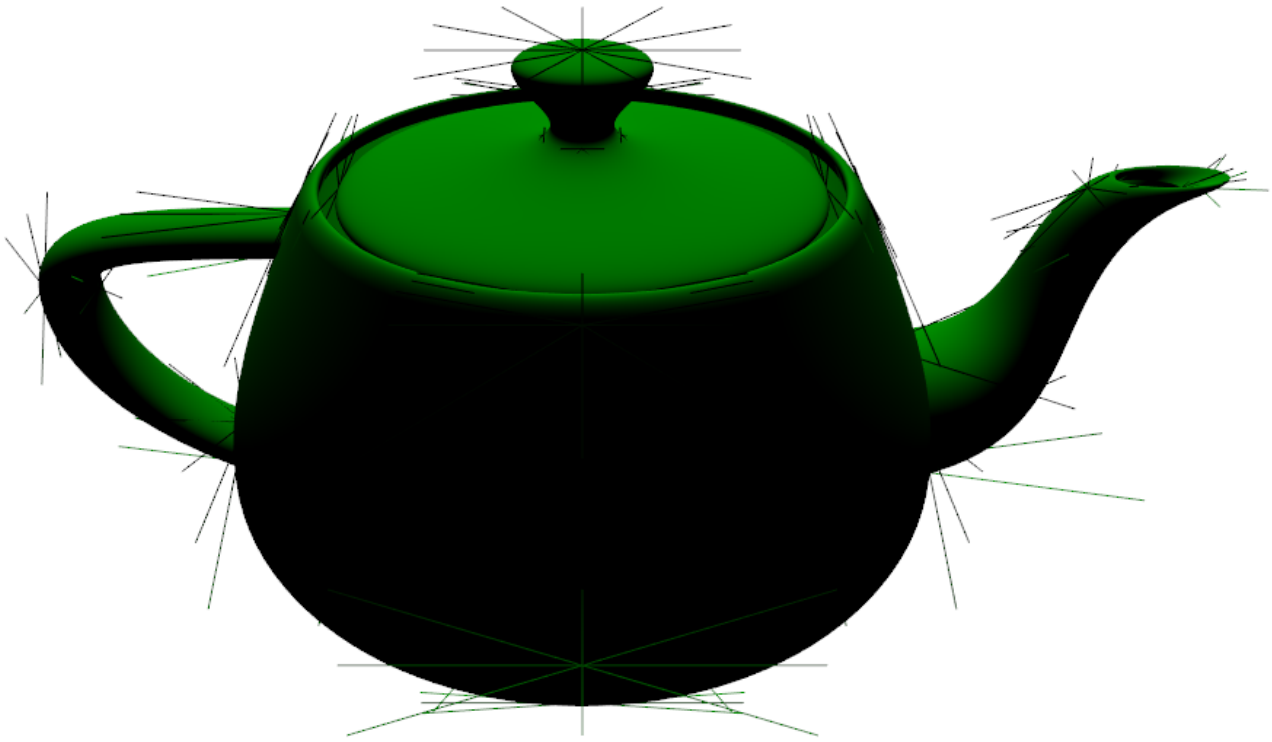

Рис. 5. Чайник із світлом зверху

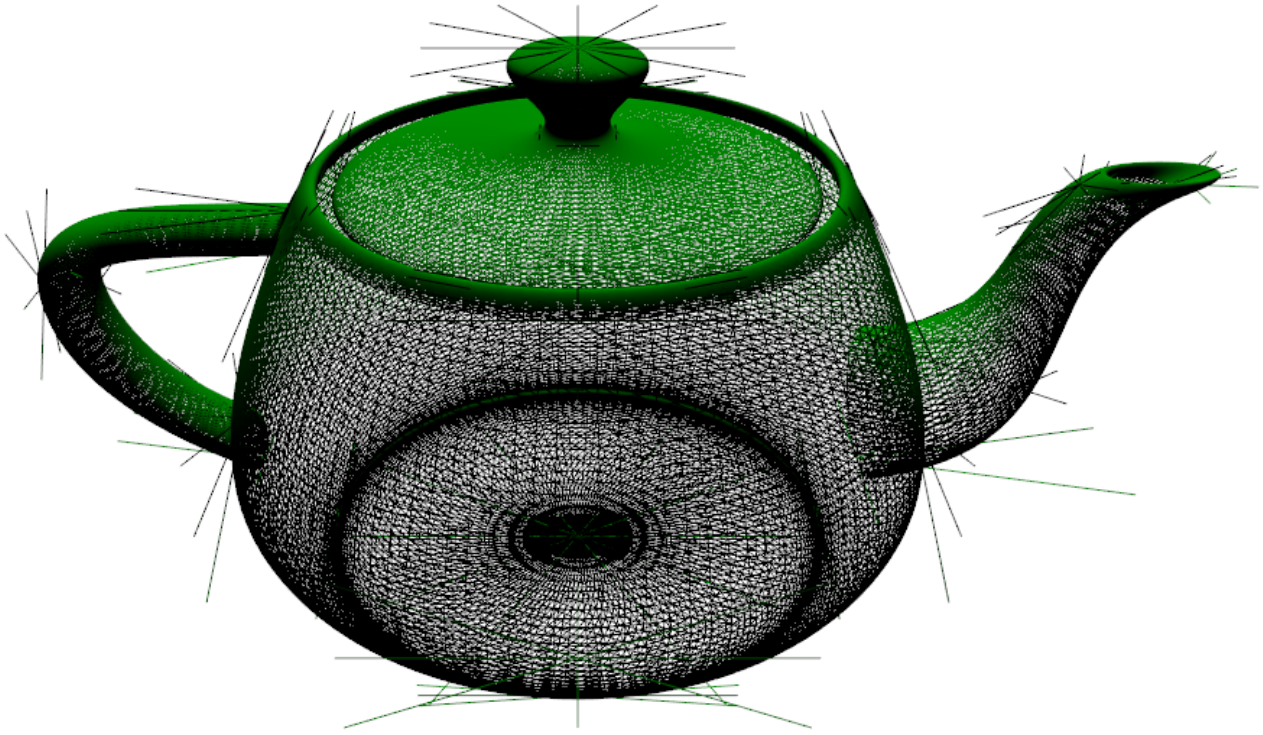

Рис. 6. Карас чайнику

ВИСНОВКИ

У ході курсової роботи були отримані такі результати:

- досліджено математичні моделі динамічних гладких об’єктів та вибрано оптимальний спосіб для розв’язання поставленої задачі,
- розроблене програмне забезпечення для відображення динамічних об’єктів,
- частково розроблене програмне забезпечення для створення або редагування динамічних об’єктів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

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.

ДОДАТОК А. ЛІСТИНГ ПРОГРАМИ

Файл ./src/scene.h

```c
#ifndef SCENE_H
#define SCENE_H

#include "osdo.h"

#include "object.h"

typedef struct Scene {
    UT_array *objects;

    // active element
    size_t active;

    // states
    int wireframe;
    bool light;
    bool dirLightOn;
    bool pointLight1On;
    bool pointLight2On;
    bool spotLightOn;
} Scene;

void scene_init(Scene *scene, UT_array *objects);

void scene_del(Scene *scene);

#endif // SCENE_H
```

Файл ./src/bijective.c

```c
#include "bijective.h"

void bijective_get_position(Bijective bijective, vec4 **position) {
    bijective.type->get_position(
                bijective.bijective, position);
}

void bijective_get_mat4(Bijective bijective, mat4 matrix) {
    bijective.type->get_mat4(
                bijective.bijective, matrix);
}

void bijective_translate(Bijective bijective, vec3 distances, float
↪  delta_time) {
    bijective.type->translate(
                bijective.bijective, distances, delta_time);
}

void bijective_rotate(
        Bijective bijective, enum coord_enum coord, float delta_time) {
    bijective.type->rotate(
                bijective.bijective, coord, delta_time);
}

void bijective_rotate_all(Bijective bijective, vec3 angles) {
    bijective.type->rotate_all(
                bijective.bijective, angles);
}

void bijective_get_animation(Bijective bijective, vec3 **animation) {
    bijective.type->get_animation(
                bijective.bijective, animation);
}

void bijective_set_animation(Bijective bijective, vec3 angles, float
↪  delta_time) {
    bijective.type->set_animation(
                bijective.bijective, angles, delta_time);
}
```

Файл ./src/object.c

```c
#include <stdio.h>

#include "object.h"
#include "conf.h"
#include "app.h"
#include "shader.h"

void object_init(Object *object, Model *model, Shader *shader) {
    *object = OBJECT(model, shader);
}

void object_init_empty(void *object) {
    *((Object*)object) = OBJECT_EMPTY;
}

void object_draw(Object *object, mat4 mat4buf, GLdouble delta_time) {
    // render the loaded model
    object_animate(object, (GLfloat)delta_time);
    object_get_mat4(object, mat4buf);
    shader_set_mat4(object->shader, "model", mat4buf);
    model_draw(object->model);
}

void object_get_position_bijective(Object *object, vec4 **position) {
    *position = &object->position;
}

void object_get_mat4(Object *object, mat4 dest) {
    glm_translate_make(dest, object->position);
    glm_mat4_mul(dest, object->transform, dest);
}

void object_translate(Object* object, vec3 distances) {
    glm_vec3_add(object->position, distances, object->position);
}

void object_translate_bijective(Object *object, vec3 distances, float
↪  delta_time) {
    vec3 new_distances = GLM_VEC3_ZERO_INIT;
    glm_vec3_muladds(distances, OBJECT_MOVE_SPEED * delta_time,
                     new_distances);
    object_translate(object, new_distances);
}

void object_rotate(Object* object, float angle, enum coord_enum coord) {
    mat4 matrix = GLM_MAT4_IDENTITY_INIT;
    switch (coord) {
    case X: glm_rotate_x(matrix, angle, matrix); break;
    case Y: glm_rotate_y(matrix, angle, matrix); break;
    case Z: glm_rotate_z(matrix, angle, matrix); break;
    }
    glm_mat4_mul(matrix, object->transform, object->transform);
}

void object_rotate_all(Object *object, vec3 angles) {
    object_rotate(object, angles[0], X);
    object_rotate(object, angles[1], Y);
    object_rotate(object, angles[2], Z);
}

void object_rotate_bijective(Object *object, enum coord_enum coord,
                             float delta_time) {
    object_rotate(object, delta_time * OBJECT_ROTATE_SPEED, coord);
}

void object_rotate_all_bijective(Object *object, vec3 angles) {
    object_rotate_all(object, angles);
}

void object_animate(Object* object, float step) {
    vec3 animation = GLM_VEC3_ZERO_INIT;
    glm_vec3_muladds(object->animation, step, animation);
    object_rotate_all(object, animation);
}

void object_get_animation(Object* object, vec3 **animation) {
    *animation = &object->animation;
}

void object_set_animation(Object* object, vec3 angles, float delta_time) {
    vec3 animation = GLM_VEC3_ZERO_INIT;
    glm_vec3_muladds(angles, delta_time, animation);
    glm_vec3_add(object->animation, animation,
                 object->animation);
}

void object_scale(Object *object, vec3 scale) {
    glm_scale(object->transform, scale);
}
```

Файл ./src/camera.h

```c
#ifndef CAMERA_H
#define CAMERA_H

#include "osdo.h"

#include "bijective.h"

#define CAMERA_DIRECTION_INIT {0.0f, 0.0f, -1.0f, 0.0f}
#define CAMERA_DIRECTION ((vec4)CAMERA_DIRECTION_INIT)

typedef struct Camera {
    mat4 rotation;
    vec4 position;
    vec3 animation;
} Camera;
```

```
16
17  void camera_get_direction(Camera* camera, vec4 dest);
18  void camera_get_mat4(Camera *camera, mat4 dest);
19  void camera_get_rotation_mat4(Camera *camera, mat4 dest);
20  void camera_get_rotation_inv_mat4(Camera *camera, mat4 dest);
21  void camera_get_position_bijective(
22          Camera *camera, vec4 **position);
23
24  void camera_translate(Camera* camera, vec3 distances);
25  void camera_translate_bijective(
26          Camera *camera, vec3 distances, float delta_time);
27
28  void camera_rotate(Camera* camera, float angle, enum coord_enum coord);
29  void camera_rotate_all(Camera* camera, vec3 angles);
30  void camera_rotate_bijective(
31          Camera *camera, enum coord_enum coord, float delta_time);
32  void camera_rotate_all_bijective(Camera *camera, vec3 angles);
33
34  void camera_get_animation(Camera *camera, vec3 **animation);
35  void camera_set_animation(Camera *camera, vec3 angles, float delta_time);
36
37  static const BijectiveType camera_bijective = {
38      camera_get_position_bijective,
39      camera_get_mat4,
40      camera_translate_bijective,
41      camera_rotate_bijective,
42      camera_rotate_all_bijective,
43      camera_get_animation,
44      camera_set_animation
45  };
46
47  #define CAMERA_INIT {GLM_MAT4_IDENTITY_INIT, GLM_VEC4_BLACK_INIT, \
48      GLM_VEC3_ZERO_INIT}
49  #define CAMERA ((Camera)CAMERA_INIT)
50
51  #define CAMERAINIT_EMPTY CAMERA_INIT(NULL, NULL)
52
53  #endif // CAMERA_H
```

Файл ./src/model.h

```
1   #ifndef MODEL_H
2   #define MODEL_H
3
4   #include "osdo.h"
5
6   typedef union __attribute__((transparent_union)) model_t {
7      struct Beziator *beziator;
8      struct Mesh *mesh;
9   } model_t;
10
11  typedef void (*draw_cb)(model_t model);
12  typedef bool (*generate_cb)(model_t model);
13  typedef void (*free_cb)(model_t model);
14
15  typedef struct ModelType {
16      draw_cb draw;
17      generate_cb generate;
18      free_cb free;
19  } ModelType;
20
21  typedef struct Model {
22      char name[64];
23      model_t model;
24      const ModelType *type;
25      UT_hash_handle hh;
26  } Model;
27
28  static const UT_icd model_icd = {
29      sizeof(Model), NULL, NULL, NULL
30  };
31
32  void model_init(Model* model, const char *name,
33                  model_t model_child, const ModelType *type);
34
35  Model *model_create(const char *name, model_t model,
36                      const ModelType *type);
37
38  void model_draw(Model* model);
39
40  void model_generate(Model *model);
41
42  void model_free(Model *model);
43
44  #endif // MODEL_H
```

Файл ./src/bijective.h

```
1   #ifndef BIJECTIVE_H
2   #define BIJECTIVE_H
3
4   #include "osdo.h"
5
6   typedef union __attribute__((transparent_union)) bijective_t {
7      struct Object *object;
8      struct Camera *camera;
9   } bijective_t;
10
11  typedef void (*get_position_cb)(
12          bijective_t bijective, vec4 **position);
13  typedef void (*get_mat4_cb)(
14          bijective_t bijective, mat4 matrix);
15  typedef void (*translate_cb)(
16          bijective_t bijective, vec3 distances, float delta_time);
17  typedef void (*rotate_cb)(
18          bijective_t bijective, enum coord_enum coord, float delta_time);
19  typedef void (*rotate_all_cb)(
20          bijective_t bijective, vec3 angles);
21  typedef void (*get_animation_cb)(
22          bijective_t bijective, vec3 **animation);
23  typedef void (*set_animation_cb)(
24          bijective_t bijective, vec3 angles, float delta_time);
25
26  typedef struct BijectiveType {
27      get_position_cb get_position;
28      get_mat4_cb get_mat4;
29      translate_cb translate;
30      rotate_cb rotate;
31      rotate_all_cb rotate_all;
32      get_animation_cb get_animation;
33      set_animation_cb set_animation;
34  } BijectiveType;
35
36  typedef struct Bijective {
37      bijective_t bijective;
38      const BijectiveType *type;
39  } Bijective;
40
41  void bijective_get_position(
42          Bijective bijective, vec4 **position);
43
44  void bijective_get_mat4(
45          Bijective bijective, mat4 matrix);
46
47  void bijective_translate(
48          Bijective bijective, vec3 distances, float delta_time);
49
50  void bijective_rotate(
51          Bijective bijective, enum coord_enum coord, float delta_time);
52
53  void bijective_rotate_all(
54          Bijective bijective, vec3 angles);
55
56  void bijective_get_animation(
57          Bijective bijective, vec3 **animation);
58
59  void bijective_set_animation(
60          Bijective bijective, vec3 angles, float delta_time);
61
62  #endif // BIJECTIVE_H
```

Файл ./src/main.c

```
1   #include "app.h"
2
3   int main() {
4       int status;
5       App app;
6       status = app_init(&app);
7       if (status) return status;
8       return app_loop(&app);
9   }
```

Файл ./src/beziator.c

```
1   #include "beziator.h"
2   #include "conf.h"
3
4   #define BEZIATOR_PATH RES_DIR"/%s.odom"
5   #define BEZIER_TANGENT_INIT {\
6   {  0,  0,  0,  0},\
7   { -3,  9, -9,  3},\
8   {  6,-12,  6,  0},\
9   { -3,  3,  0,  0},}
10  #define DETALIZATION 32
11
12  bool beziator_init(
13          Beziator *beziator, const char *name, Shader *editmode) {
14      mesh_init(&beziator->mesh);
15      mesh_init(&beziator->frame);
16      mesh_init(&beziator->normals);
17      beziator->editmode = editmode;
18      const size_t path_len = strlen(BEZIATOR_PATH);
19      const size_t len = strlen(BEZIATOR_PATH);
20      char *path = calloc(len + path_len, sizeof(char));
21      snprintf(path, len + path_len, BEZIATOR_PATH, name);
22      FILE *file = fopen(path, "r");
23      if (file == NULL) {
24          printf("ERROR: failed to open file %s\n", path);
25          free(path);
26          return false;
```

```
    }
    free(path);

    fscanf(file, "%lu%lu", &beziator->points_size, &beziator->surfaces_size);
    beziator->points = (vec4*)calloc(beziator->points_size, sizeof(vec4));
    if (beziator->points == NULL) {
        fclose(file);
        printf("Failed to allocate memory.\n");
        return false;
    }
    beziator->surfaces = (surface_t*)calloc(beziator->surfaces_size,
    ↪  sizeof(surface_t));
    if (beziator->points == NULL) {
        fclose(file);
        free(beziator->points);
        printf("Failed to allocate memory.\n");
        return false;
    }
    vec4 *point;
    surface_t *surface;
    for (size_t i = 0; i < beziator->points_size; i++) {
        point = &beziator->points[i];
        glm_vec4_copy(GLM_VEC4_BLACK, *point);
        fscanf(file, "%f%f%f", (*point), (*point) + 1, (*point) + 2);
    }
    int j, k;
    size_t tmp;
    for (size_t i = 0; i < beziator->surfaces_size; i++) {
        surface = &beziator->surfaces[i];
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++) {
                fscanf(file, "%lu", &tmp);
                (*surface)[j][k] = beziator->points + tmp;
            }
    }
    fclose(file);
    return true;
}

Beziator *beziator_create(const char *name, Shader *editmode) {
    Beziator *beziator = calloc(1, sizeof(Beziator));
    beziator_init(beziator, name, editmode);
    return beziator;
}

void beziator_del(Beziator *beziator) {
    if (beziator->points) {
        free(beziator->points);
        beziator->points = NULL;
    }
    if (beziator->surfaces) {
        free(beziator->surfaces);
        beziator->surfaces = NULL;
    }
    mesh_del(&beziator->mesh);
    mesh_del(&beziator->frame);
    mesh_del(&beziator->normals);
}

void beziator_free(Beziator *beziator) {
    beziator_del(beziator);
    free(beziator);
}

void beziator_draw(Beziator *beziator) {
    //shader_set_vec3(beziator->editmode, "min_coord", beziator->min_coord);
    //shader_set_vec3(beziator->editmode, "max_coord", beziator->max_coord);
    //mesh_draw_mode(&beziator->frame, GL_POINTS);
    mesh_draw_mode(&beziator->frame, GL_LINES);
    shader_set_float(beziator->editmode, "alpha", 0.5f);
    mesh_draw_mode(&beziator->mesh, GL_TRIANGLES);
    //mesh_draw_mode(&beziator->normals, GL_LINES);
    shader_set_float(beziator->editmode, "alpha", 1);
}


void bezier_curve(float a, mat4 points, vec4 dest) {
    mat4 matrix;
    glm_vec4_cubic(a, dest);
    glm_mat4_mul(points, GLM_BEZIER_MAT, matrix);
    glm_mat4_mulv(matrix, dest, dest);
}

void bezier_curve_tangent(float a, mat4 points, vec4 dest) {
    mat4 matrix;
    glm_vec4_cubic(a, dest);
    glm_mat4_mul(points, (mat4)BEZIER_TANGENT_INIT, matrix);
    glm_mat4_mulv(matrix, dest, dest);
}

void bezier_surface(
        float u, float v, surface_t points, vec4 dest, vec4 normal) {
    mat4 m, res1, res2, res3;

    for (int i = 0; i < 4; i++) {
        glm_vec4_copy(*(points[0][i]), m[0]);
        glm_vec4_copy(*(points[1][i]), m[1]);
        glm_vec4_copy(*(points[2][i]), m[2]);
        glm_vec4_copy(*(points[3][i]), m[3]);
        bezier_curve(u, m, res1[i]);

        glm_vec4_copy(*(points[i][0]), m[0]);
        glm_vec4_copy(*(points[i][1]), m[1]);
        glm_vec4_copy(*(points[i][2]), m[2]);
        glm_vec4_copy(*(points[i][3]), m[3]);
        bezier_curve(v, m, res2[i]);
    }

    bezier_curve(v, res1, dest);
    bezier_curve_tangent(v, res1, res3[1]);
    bezier_curve_tangent(u, res2, res3[3]);

    glm_cross(res3[1], res3[3], normal);
}

bool beziator_generate(Beziator *beziator) {
    static const int controls_lines[][2] = {
        {0, 0}, {0, 1}, {0, 0}, {1, 1}, {0, 0}, {1, 0},
        {0, 3}, {0, 2}, {0, 3}, {1, 2}, {0, 3}, {1, 3},
        {3, 0}, {2, 0}, {3, 0}, {2, 1}, {3, 0}, {3, 1},
        {3, 3}, {3, 2}, {3, 3}, {2, 2}, {3, 3}, {2, 3},
    };
    static const int ctrls_size = sizeof(controls_lines) / sizeof(int[2]);
    /* // Old variant of config, I leave it for several commits
    static const uint8_t ALL_SQUARE_LINES[][4] = {
        {1, 0, 0, 0}, {0, 0, 0, 1}, {0, 1, 1, 1}, {1, 1, 1, 0},
        {1, 1, 0, 0}, {0, 0, 1, 1}, {1, 0, 0, 1}, {0, 1, 1, 0}
    };*/
    static const uint8_t SQUARE_TYPES[][10][2] = {
        {{0, 0}, {0, 1}, {1, 1}, {0, 0}, {8, 8},
         {1, 1}, {1, 0}, {0, 0}, {1, 1}, {9, 9}},
        {{0, 0}, {0, 1}, {1, 1}, {1, 0}, {0, 0}, {9, 9}},
        {{1, 0}, {0, 0}, {0, 1}, {1, 1}, {8, 8},
         {0, 1}, {1, 1}, {1, 0}, {0, 1}, {9, 9}},
        /* // And again old variant of config
        {1, 2, 4, 1, 8, 0, 5, 3, 0, 9},
        {0, 1, 2, 3, 0, 9},
        {0, 1, 5, 0, 8, 2, 3, 5, 2, 9},*/
    };
    static const uint8_t BEZIER_SQUARE_TYPES[3][3] = {
        {0, 1, 2}, {1, 1, 1}, {2, 1, 0}
    };

    size_t j, k, index;
    float x, u, v;
    vec4 *point, vertex, normal;
    surface_t *surface;
    GLuint verts = 0, verts2 = 0, verts3 = 0;
    const int *c;
    mat4 m4b;
    uint8_t si, sj;
    const uint8_t (*st)[2];

    Mesh *mesh = &beziator->mesh, *mesh_skel = &beziator->frame,
            *mesh_normals = &beziator->normals;
    const size_t d = DETALIZATION;
    x = 1.f / (d - 1);

    const unsigned long size =
            6 * 9 * d * d * beziator->surfaces_size;
    const GLsizei sizei = (GLsizei)size;
    Vertex *V = (Vertex*)calloc(size, sizeof(Vertex));
    GLuint *E = (GLuint*)calloc(size, sizeof(GLuint));
    Vertex *V2 = (Vertex*)calloc(size, sizeof(Vertex));
    GLuint *E2 = (GLuint*)calloc(size, sizeof(GLuint));
    Vertex *V3 = (Vertex*)calloc(beziator->points_size, sizeof(Vertex));
    GLuint *E3 = (GLuint*)calloc(beziator->points_size * 4, sizeof(GLuint));

    // Creator frame vertices
    for (size_t i = 0; i < beziator->points_size; i++) {
        point = &beziator->points[i];
        glm_vec3_copy(*point, V2[i].position);
        V2[i].color[1] = 255;
        V2[i].color[3] = 255;
    }

    for (size_t i = 0; i < beziator->surfaces_size; i++) {
        // Creator frame lines
        for (j = 0; j < ctrls_size; j++) {
            c = controls_lines[j];
            E2[verts2++] =
                    (unsigned)(beziator->surfaces[i][c[0]][c[1]] -
                    ↪  beziator->points);
        }

        // Create vertices
        for (j = 0; j < d; j++) {
            for (k = 0; k < d; k++) {
                u = (float)j*x; v = (float)k*x;
                index = i * d * d + j * d + k;
                bezier_surface(u, v, beziator->surfaces[i], vertex, normal);
                glm_normalize(normal);
                glm_vec3_copy(vertex, V[index].position);
                glm_vec3_copy(normal, V[index].normal);
                V[verts].color[1] = 255;
                /*glm_vec3_copy(vertex, V3[verts3].position);
                E3[verts3] = verts3;
                verts3++;
                glm_vec3_add(normal, vertex, V3[verts3].position);
                E3[verts3] = verts3;
                verts3++;*/
            }
        }

        // Create triangles
```

```
        for (j = 0; j < d - 1; j++)
            for (k = 0; k < d - 1; k++) {
                E[verts++] = (unsigned)(i * d * d + (j + 1) * d + k);
                E[verts++] = (unsigned)(i * d * d + j * d + k);
                E[verts++] = (unsigned)(i * d * d + j * d + k + 1);

                E[verts++] = (unsigned)(i * d * d + j * d + k + 1);
                E[verts++] = (unsigned)(i * d * d + (j + 1) * d + k + 1);
                E[verts++] = (unsigned)(i * d * d + (j + 1) * d + k);
            }

        surface = &(beziator->surfaces[i]);
        for (si = 0; si < 3; si++) {
            for (sj = 0; sj < 3; sj++) {
                st = SQUARE_TYPES[BEZIER_SQUARE_TYPES[si][sj]];
                while (st[2][0] != 9) {
                    if (st[2][0] == 8) {
                        st += 3;
                    }
                    index = (size_t)((*surface)[si+st[1][0]][sj+st[1][1]]
                    ↪  beziator->points);
                    glm_vec3_sub(*((*surface)[si+st[1][0]][sj+st[1][1]]),
                           *((*surface)[si+st[0][0]][sj+st[0][1]]), m4b[0]);
                    glm_vec3_sub(*((*surface)[si+st[1][0]][sj+st[1][1]]),
                           *((*surface)[si+st[2][0]][sj+st[2][1]]), m4b[1]);
                    glm_vec3_cross(m4b[0], m4b[1], m4b[2]);
                    glm_vec3_add(V2[index].normal, m4b[2], V2[index].normal);
                    st++;
                }
            }
        }
    }
    /*
    // Example drawing of normals for frame
    for (size_t i = 0; i < beziator->points_size; i++) {
        glm_normalize(V2[i].normal);
        glm_vec3_add(V2[i].position, V2[i].normal,
                     V2[i+beziator->points_size].position);
        E2[verts2++] = (unsigned)i;
        E2[verts2++] = (unsigned)(i+beziator->points_size);
    }*/
    mesh_update(mesh, sizei, sizei, V, E);
    mesh_update(mesh_skel, sizei, sizei, V2, E2);
    mesh_update(mesh_normals, sizei, sizei, V3, E3);
    return true;
}
```

Файл ./src/nkglfw.h

```
#ifndef NKGLFW_H
#define NKGLFW_H

#include "osdo.h"
#include "conf.h"

#include "shader.h"
#include "nuklear.h"
#include "window.h"

#define NK_GLFW_TEXT_MAX 256

typedef struct NkGlfw {
    Shader *shader;
    Mesh mesh;
    struct nk_context context;
    struct nk_font_atlas atlas;
    struct nk_buffer cmds;
    struct nk_draw_null_texture null;
    struct nk_vec2 scroll;
    struct nk_convert_config config;
    GLuint font_tex;
    Window *window;
    mat4 ortho;
} NkGlfw;

void nk_glfw_init(NkGlfw* nkglfw, Window *window, Shader *shader);
void nk_glfw_del(NkGlfw* nkglfw);

void nk_glfw_font_stash_begin(NkGlfw* nkglfw,
                              struct nk_font_atlas **atlas);
void nk_glfw_font_stash_end(NkGlfw* nkglfw);

void nk_glfw_begin_input(NkGlfw* nkglfw);
void nk_glfw_end_input(NkGlfw* nkglfw);
void nk_glfw_render(NkGlfw* nkglfw);

void nk_gflw_scroll_callback(NkGlfw* nkglfw, double xoff, double yoff);
void nk_glfw_mouse_callback(NkGlfw *nkglfw, vec2 pos, vec2 offset);
void nk_glfw_char_callback(NkGlfw* nkglfw, unsigned int codepoint);
void nk_glfw_mouse_button_callback(
        NkGlfw* nkglfw, enum BUTTONS button, bool pressed);
void nk_glfw_key_callback(
        NkGlfw* nkglfw, enum KEYS key, bool pressed);

#endif // NKGLFW_H
```

Файл ./src/shader.h

```
#ifndef SHADER_H
#define SHADER_H

#include "osdo.h"

typedef struct Shader {
    char name[64];
    GLuint shader;
    UT_hash_handle hh;
} Shader;

bool shader_compile(const char* vertexCode, const char* fragmentCode,
                    Shader *shader);

// constructor generates the shader on the fly
bool shader_init(Shader *shader, const char* name);
Shader *shader_create(const char* name);

void shader_del(Shader *shader);
void shader_free(Shader *shader);

// activate the shader
void shader_use(Shader *shader);

// utility uniform functions
void shader_set_bool (Shader *shader, const char* name, bool value);
void shader_set_int  (Shader *shader, const char* name, int value);
void shader_set_float(Shader *shader, const char* name, float value);
void shader_set_vec2 (Shader *shader, const char* name, vec2 value);
void shader_set_vec2f(Shader *shader, const char* name,
                      float x, float y);
void shader_set_vec3 (Shader *shader, const char* name, vec3 value);
void shader_set_vec3f(Shader *shader, const char* name,
                      float x, float y, float z);
void shader_set_vec4 (Shader *shader, const char* name, vec4 value);
void shader_set_vec4f(Shader *shader, const char* name,
                      float x, float y, float z, float w);
void shader_set_mat2 (Shader *shader, const char* name, mat2 mat);
void shader_set_mat3 (Shader *shader, const char* name, mat3 mat);
void shader_set_mat4 (Shader *shader, const char* name, mat4 mat);

#endif // SHADER_H
```

Файл ./src/app.h

```
#ifndef APP_H
#define APP_H

#include "osdo.h"
#include "conf.h"

#include "shader.h"
#include "mesh.h"
#include "scene.h"
#include "camera.h"
#include "nkglfw.h"
#include "model.h"
#include "window.h"

enum TRANSFORMATIONS {
    ROTATE    = 0,
    TRANSLATE = 1,
    ANIMATE   = 2,
};

typedef struct App {
    Model *models;
    Shader *shaders;
    Scene scene;
    UT_array *objects;
    Camera camera;
    NkGlfw nkglfw;
    Window window;
    bool interactive_mode;
    int trans[3][3];

    // buffered data for loop
    mat4 mat4buf, projection, last_camera;
    vec4 vec4buf;
} App;

int app_init(App *app);

void app_del(App *app);

int app_loop(App *app);

bool app_load_shader(App *app, const char *name);

void app_scroll(Window* window, GLdouble xoffset, GLdouble yoffset);
void app_mouse(Window* window, vec2 pos, vec2 offset);
void app_char_callback(Window* window, unsigned int codepoint);
void app_mouse_button_callback(
        Window *window, enum BUTTONS button, bool pressed);
void app_key(Window* window, enum KEYS key, bool pressed);

// process all input
// -----------------
void app_process_input(App *app);

#endif // APP_H
```

Файл ./src/window.h

```
#ifndef WINDOW_H
#define WINDOW_H

#include "osdo.h"

struct GLFWwindow;

enum KEYS {
    KEY_DELETE,
    KEY_ENTER,
    KEY_TAB,
    KEY_BACKSPACE,
    KEY_UP,
    KEY_DOWN,
    KEY_LEFT,
    KEY_RIGHT,
    KEY_HOME,
    KEY_END,
    KEY_PAGE_UP,
    KEY_PAGE_DOWN,
    KEY_LEFT_SHIFT,
    KEY_RIGHT_SHIFT,
    KEY_LEFT_CONTROL,
    KEY_RIGHT_CONTROL,
    KEY_1, KEY_2, KEY_3, KEY_4, KEY_5, KEY_6, KEY_7, KEY_8, KEY_9, KEY_0,
    KEY_Q, KEY_W, KEY_E, KEY_R, KEY_T, KEY_Y, KEY_U, KEY_I, KEY_O, KEY_P,
    KEY_A, KEY_S, KEY_D, KEY_F, KEY_G, KEY_H, KEY_J, KEY_K, KEY_L,
    KEY_Z, KEY_X, KEY_C, KEY_V, KEY_B, KEY_N, KEY_M,
};

enum BUTTONS {
    MOUSE_BUTTON_LEFT,
    MOUSE_BUTTON_MIDDLE,
    MOUSE_BUTTON_RIGHT,
    MOUSE_BUTTON_DOUBLE,
};

struct Window;

typedef void (*scroll_cb_t)(
        struct Window *window, GLdouble xoffset, GLdouble yoffset);
typedef void (*mouse_motion_cb_t)(
        struct Window *window, vec2 pos, vec2 offset);
typedef void (*char_cb_t)(
        struct Window *window, unsigned int codepoint);
typedef void (*mouse_button_cb_t)(
        struct Window *window, enum BUTTONS button, bool pressed);
typedef void (*key_cb_t)(
        struct Window *window, enum KEYS key, bool pressed);

typedef struct Window {
    struct GLFWwindow *window;
    void *user_pointer;

    // screen size
    int size[2], display[2];
    vec2 scale, cursor;
    bool mouse_capute;
    double last_click_time, current_time, last_time, delta_time;

    scroll_cb_t scroll_cb;
    mouse_motion_cb_t mouse_motion_cb;
    char_cb_t char_cb;
    mouse_button_cb_t mouse_button_cb;
    key_cb_t key_cb;
} Window;

int window_init(Window *window);
void window_del(Window *window);

bool window_alive(Window *window);
bool window_pre_loop(Window *window);
void window_post_loop(Window *window);

void window_set_user_pointer(Window *window, void *pointer);
void *window_get_user_pointer(Window *window);

float window_get_resolution(Window *window);

double window_get_delta_time(Window *window);

const char * window_get_clipboard(Window *window);
void window_set_clipboard(Window *window, const char * str);

bool window_is_mouse_caputed(Window *window);
void window_grab_mouse(Window *window, bool grab);

bool window_is_key_pressed(Window *window, enum KEYS key);
bool window_is_mouse_pressed(Window *window, enum BUTTONS key);

void window_get_cursor(Window *window, vec2 dest);
void window_set_cursor(Window *window, vec2 coords);

int *window_get_size(Window *window);
int *window_get_display(Window *window);
float *window_get_scale(Window *window);

void window_set_scroll_cb(Window *window, scroll_cb_t callback);
void window_set_mouse_motion_cb(
        Window *window, mouse_motion_cb_t callback);
void window_set_char_cb(Window *window, char_cb_t callback);
void window_set_mouse_button_cb(
        Window *window, mouse_button_cb_t callback);
void window_set_key_cb(Window *window, key_cb_t callback);

void window_resize_cb(
        struct GLFWwindow* window, GLint width, GLint height);
void window_scroll_cb(
        struct GLFWwindow* window, GLdouble xoffset, GLdouble yoffset);
void window_mouse_motion_cb(
        struct GLFWwindow* window, double xpos, double ypos);
void window_char_cb(struct GLFWwindow* window, unsigned int codepoint);
void window_mouse_button_cb(
        struct GLFWwindow *window, int button, int action, int mods);
void window_key_cb(struct GLFWwindow* window, int key, int scancode,
                   int action, int mods);

#endif // WINDOW_H
```

Файл ./src/nuklear.c

```
#include "conf.h"
#define NK_IMPLEMENTATION
#include <nuklear.h>
```

Файл ./src/shader.c

```
#include <stdio.h>
#include <string.h>

#include "shader.h"
#include "conf.h"

#define VERTEX_PATH RES_DIR"/%s.vs"
#define FRAGMENT_PATH RES_DIR"/%s.fs"

char * readFromFile(const char *path) {
    char* data;
    size_t size;
    FILE *file = fopen(path, "r");
    if (file == NULL) {
        printf("ERROR: failed to open file %s\n", path);
        return NULL;
    }
    fseek(file, 0L, SEEK_END);
    size = (size_t)ftell(file);
    fseek(file, 0L, SEEK_SET);
    data = (char*) malloc(size + 1);
    fread(data, 1, size, file);
    data[size] = 0;
    fclose(file);
    return data;
}

// utility function for checking shader compilation/linking errors.
bool check_shader(GLuint shader, const int type) {
    GLint status = 0, size = 0;
    GLchar *log;
    GLuint status_type = GL_COMPILE_STATUS;
    void (*gl_get)(GLuint, GLuint, GLint*) = glGetShaderiv;

    if (type == 0) {
        gl_get = glGetProgramiv;
        status_type = GL_LINK_STATUS;
    }

    gl_get(shader, status_type, &status);
    if (status == GL_FALSE) {
        gl_get(shader, GL_INFO_LOG_LENGTH, &size);
        log = (GLchar*)malloc((unsigned long)size);
        if (log == NULL) {
            printf("Got some error, but cant allocate memory to read it.\n");
            return false;
        }
        glGetShaderInfoLog(shader, size, &size, log);
        puts(log);
        fflush(stdout);
        free(log);
        return false;
    }
    return true;
}

bool shader_compile(const char* vertexCode, const char* fragmentCode,
                    Shader *shader) {
    // 2. compile shaders
    GLuint vertex, fragment;

    // vertex shader
    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vertexCode, NULL);
    glCompileShader(vertex);
    if (!check_shader(vertex, 1)) {
        printf("Failed to compile vertex shader.\n%s", vertexCode);
        return false;
    }

    // fragment Shader
```

```
    fragment = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment, 1, &fragmentCode, NULL);
    glCompileShader(fragment);
    if (!check_shader(fragment, 1)) {
        printf("Failed to compile fragment shader.\n%s", fragmentCode);
        glDeleteShader(vertex);
        return false;
    }

    // shader Program
    shader->shader = glCreateProgram();
    glAttachShader(shader->shader, vertex);
    glAttachShader(shader->shader, fragment);
    glLinkProgram(shader->shader);
    if (!check_shader(shader->shader, 0)) {
        printf("Failed to attach shaders.");
        glDeleteShader(vertex);
        glDeleteShader(fragment);
        return false;
    }

    // delete the shaders as they're linked into our program now and no
    ↪ longer necessery
    glDeleteShader(vertex);
    glDeleteShader(fragment);
    return true;
}

bool shader_init(Shader *shader, const char* name) {
    // 1. retrieve the vertex/fragment source code from filePath
    const size_t path_len = strlen(VERTEX_PATH);
    const size_t len = strlen(name);
    strcpy(shader->name, name);
    char *vertex_path = calloc(len + path_len, sizeof(char)),
        *fragment_path = calloc(len + path_len, sizeof(char));
    snprintf(vertex_path, len + path_len, VERTEX_PATH, name);
    snprintf(fragment_path, len + path_len, FRAGMENT_PATH, name);
    GLchar* vertex = readFromFile(vertex_path);
    if (vertex == NULL) {
        printf("ERROR: failed to read from vertex shader file %s.\n",
               vertex_path);
        free(vertex_path);
        free(fragment_path);
        return false;
    }

    GLchar* fragment = readFromFile(fragment_path);
    if (fragment == NULL) {
        printf("ERROR: failed to read from fragment shader file %s.\n",
               fragment_path);
        free(vertex_path);
        free(fragment_path);
        free(vertex);
        return false;
    }

    if (!shader_compile(vertex, fragment, shader))
        return false;

    free(vertex);
    free(fragment);
    free(vertex_path);
    free(fragment_path);

    return true;
}

Shader *shader_create(const char *name) {
    Shader *shader = calloc(1, sizeof(Shader));
    if (shader && !shader_init(shader, name)) {
        free(shader);
        return NULL;
    }
    return shader;
}

void shader_del(Shader *shader) {
    glDeleteProgram(shader->shader);
}

void shader_free(Shader *shader) {
    shader_del(shader);
    free(shader);
}

void shader_use(Shader *shader) {
    glUseProgram(shader->shader);
}

void shader_set_bool(Shader *shader, const char* name, bool value) {
    glUniform1i(glGetUniformLocation(shader->shader, name), (int)value);
}

void shader_set_int(Shader *shader, const char* name, int value) {
    glUniform1i(glGetUniformLocation(shader->shader, name), value);
}

void shader_set_float(Shader *shader, const char* name, float value) {
    glUniform1f(glGetUniformLocation(shader->shader, name), value);
}

void shader_set_vec2(Shader *shader, const char* name, vec2 value) {
    glUniform2fv(glGetUniformLocation(shader->shader, name),
                 1, &value[0]);
}

void shader_set_vec2f(Shader *shader, const char* name,
                      float x, float y) {
    glUniform2f(glGetUniformLocation(shader->shader, name), x, y);
}

void shader_set_vec3(Shader *shader, const char* name, vec3 value) {
    glUniform3fv(glGetUniformLocation(shader->shader, name),
                 1, &value[0]);
}

void shader_set_vec3f(Shader *shader, const char* name,
                      float x, float y, float z) {
    glUniform3f(glGetUniformLocation(shader->shader, name), x, y, z);
}

void shader_set_vec4(Shader *shader, const char* name, vec4 value) {
    glUniform4fv(glGetUniformLocation(shader->shader, name),
                 1, &value[0]);
}

void shader_set_vec4f(Shader *shader, const char* name,
                      float x, float y, float z, float w) {
    glUniform4f(glGetUniformLocation(shader->shader, name), x, y, z, w);
}

void shader_set_mat2(Shader *shader, const char* name, mat2 mat) {
    glUniformMatrix2fv(glGetUniformLocation(shader->shader, name),
                       1, GL_FALSE, &mat[0][0]);
}

void shader_set_mat3(Shader *shader, const char* name, mat3 mat) {
    glUniformMatrix3fv(glGetUniformLocation(shader->shader, name),
                       1, GL_FALSE, &mat[0][0]);
}

void shader_set_mat4(Shader *shader, const char* name, mat4 mat) {
    glUniformMatrix4fv(glGetUniformLocation(shader->shader, name),
                       1, GL_FALSE, &mat[0][0]);
}
```

Файл ./src/app.c

```
#include "float.h"
#include "app.h"
#include "conf.h"
#include "shader.h"
#include "beziator.h"

int app_init(App *app) {
    app->models = NULL;
    app->shaders = NULL;
    app->interactive_mode = false;
    window_init(&app->window);
    utarray_new(app->objects, &object_icd);
    app->camera = CAMERA;
    memset(app->trans, 0, sizeof(app->trans));

    window_set_user_pointer(&app->window, app);
    window_set_scroll_cb(&app->window, app_scroll);
    window_set_mouse_motion_cb(&app->window, app_mouse);
    window_set_char_cb(&app->window, app_char_callback);
    window_set_mouse_button_cb(&app->window, app_mouse_button_callback);
    window_set_key_cb(&app->window, app_key);

    // build and compile our shader zprogram
    // ------------------------------------
    if (!app_load_shader(app, "simple") ||
            !app_load_shader(app, "textured") ||
            !app_load_shader(app, "lighting") ||
            !app_load_shader(app, "nuklear") ||
            !app_load_shader(app, "editmode"))
        return -1;

    Shader *shader;
    HASH_FIND_STR(app->shaders, "editmode", shader);
    if (shader == NULL)
        return -1;

    {
        model_t model_ch;
        Model *model;
        Object object;

        model_ch.beziator = beziator_create("test", shader);
        model = model_create("teapot", model_ch, &beziator_type);
        beziator_generate(model_ch.beziator);
        object_init(&object, model, shader);
        object_translate(&object, (vec3){0, -5, 0});
        object_scale(&object, (vec3){4, 4, 4});
        object_rotate(&object, -M_PI_F / 3.f, X);
        HASH_ADD_STR(app->models, name, model);
        utarray_push_back(app->objects, &object);

        /*model_ch.mesh = mesh_create();
        mesh_cube_update(model_ch.mesh);
        model = model_create("cube", model_ch, &mesh_type);
```

```
        object_init(&object, model, shader);
        HASH_ADD_STR(app->models, name, model);
        utarray_push_back(app->objects, &object);*/
    }

    scene_init(&app->scene, app->objects);
    camera_translate(&app->camera, BASIS0POS);

    HASH_FIND_STR(app->shaders, "nuklear", shader);
    nk_glfw_init(&app->nkglfw, &app->window, shader);

    return 0;
}

void app_del(App *app) {
    nk_glfw_del(&app->nkglfw);
    scene_del(&app->scene);
    utarray_free(app->objects);

    {
        Model *i, *tmp;
        HASH_ITER(hh, app->models, i, tmp)
            model_free(i);
    }

    {
        Shader *i, *tmp;
        HASH_ITER(hh, app->shaders, i, tmp)
            shader_free(i);
    }
}

int app_loop(App *app) {
    Scene* scene = &app->scene;
    Shader *sh, *sh2;
    HASH_FIND_STR(app->shaders, "editmode", sh);
    HASH_FIND_STR(app->shaders, "simple", sh2);
    struct nk_context *ctx = &app->nkglfw.context;
    struct nk_colorf bg;
    bg.r = 0.8f; bg.g = 0.9f; bg.b = 0.8f; bg.a = 1.0f;
    char text[128];
    vec4 *position, direction;
    vec3 rotation, *animation;
    Bijective bijective;
    int light = false;

    // render loop
    // -----------
    while(window_alive(&app->window)) {
        nk_glfw_begin_input(&app->nkglfw);
        window_pre_loop(&app->window);
        nk_glfw_end_input(&app->nkglfw);

        if (scene->active)
            snprintf(text, 128, "Object %zu", scene->active);
        else snprintf(text, 128, "Camera");

        if (scene->active) {
            bijective.bijective.object = (void*)utarray_eltptr(
                        scene->objects, (unsigned)scene->active - 1);
            bijective.type = &object_bijective;
        } else {
            bijective.bijective.camera = &app->camera;
            bijective.type = &camera_bijective;
        }
        bijective_get_position(bijective, &position);
        bijective_get_animation(bijective, &animation);
        glm_vec3_copy(GLM_VEC3_ZERO, rotation);

        /* GUI */
        if (nk_begin(ctx, "OSDO", nk_rect(4, 4, 256, 512),
            NK_WINDOW_BORDER|NK_WINDOW_MOVABLE|NK_WINDOW_SCALABLE|
            NK_WINDOW_MINIMIZABLE|NK_WINDOW_TITLE)) {
            nk_layout_row_dynamic(ctx, 25, 1);
            nk_checkbox_label(ctx, "Wireframe",
                              &app->scene.wireframe);
            nk_checkbox_label(ctx, "Light",
                              &light);
            nk_label(ctx, "Active element:", NK_TEXT_LEFT);
            nk_layout_row_dynamic(ctx, 25, 2);
            nk_label(ctx, text, NK_TEXT_LEFT);
            scene->active = (size_t)(nk_propertyi(
                                ctx, "#", 0, (int)scene->active,
                                (int)utarray_len(scene->objects), 1, 0.1f));
            nk_layout_row_dynamic(ctx, 25, 1);
            nk_label(ctx, "Position:", NK_TEXT_LEFT);
            nk_layout_row_dynamic(ctx, 25, 3);
            nk_property_float(ctx, "#X:", -FLT_MAX, *position,
                              FLT_MAX, 0.1f, 0.1f);
            nk_property_float(ctx, "#Y:", -FLT_MAX, *position + 1,
                              FLT_MAX, 0.1f, 0.1f);
            nk_property_float(ctx, "#Z:", -FLT_MAX, *position + 2,
                              FLT_MAX, 0.1f, 0.1f);

            nk_layout_row_dynamic(ctx, 25, 1);
            nk_label(ctx, "Rotation:", NK_TEXT_LEFT);
            nk_layout_row_dynamic(ctx, 25, 3);
            nk_property_float(ctx, "#X:", -FLT_MAX, rotation,
                              FLT_MAX, 0.1f, 0.1f);
            nk_property_float(ctx, "#Y:", -FLT_MAX, rotation + 1,
                              FLT_MAX, 0.1f, 0.1f);
            nk_property_float(ctx, "#Z:", -FLT_MAX, rotation + 2,
                              FLT_MAX, 0.1f, 0.1f);

            nk_label(ctx, "Animation:", NK_TEXT_LEFT);
            nk_layout_row_dynamic(ctx, 25, 3);
            nk_property_float(ctx, "#X:", -FLT_MAX, *animation,
                              FLT_MAX, 0.1f, 0.1f);
            nk_property_float(ctx, "#Y:", -FLT_MAX, *animation + 1,
                              FLT_MAX, 0.1f, 0.1f);
            nk_property_float(ctx, "#Z:", -FLT_MAX, *animation + 2,
                              FLT_MAX, 0.1f, 0.1f);

            nk_layout_row_dynamic(ctx, 20, 1);
            nk_label(ctx, "background:", NK_TEXT_LEFT);
            nk_layout_row_dynamic(ctx, 25, 1);
            if (nk_combo_begin_color(ctx, nk_rgb_cf(bg),
            ↪  nk_vec2(nk_widget_width(ctx),400))) {
                nk_layout_row_dynamic(ctx, 120, 1);
                bg = nk_color_picker(ctx, bg, NK_RGBA);
                nk_layout_row_dynamic(ctx, 25, 1);
                bg.r = nk_propertyf(ctx, "#R:", 0, bg.r, 1.0f, 0.01f,0.005f);
                bg.g = nk_propertyf(ctx, "#G:", 0, bg.g, 1.0f, 0.01f,0.005f);
                bg.b = nk_propertyf(ctx, "#B:", 0, bg.b, 1.0f, 0.01f,0.005f);
                bg.a = nk_propertyf(ctx, "#A:", 0, bg.a, 1.0f, 0.01f,0.005f);
                nk_combo_end(ctx);
            }
        }
        nk_end(ctx);

        if (scene->active) {
            Beziator *beziator = app->models->model.beziator;
            if (nk_begin(ctx, "Bezier mesh", nk_rect(267, 4, 256, 512),
                       NK_WINDOW_BORDER|NK_WINDOW_MOVABLE|NK_WINDOW_SCALABLE|
                         NK_WINDOW_MINIMIZABLE|NK_WINDOW_TITLE)) {
                nk_layout_row_dynamic(ctx, 25, 1);
                if (nk_button_label(ctx, "Regenerate"))
                    beziator_generate(beziator);
                nk_layout_row_dynamic(ctx, 25, 1);
                nk_label(ctx, "Control points:", NK_TEXT_LEFT);
                for (size_t i = 0; i < beziator->points_size; i++) {
                    nk_layout_row_dynamic(ctx, 25, 3);
                    nk_property_float(ctx, "#X:", -FLT_MAX,
                    ↪  beziator->points[i],
                                      FLT_MAX, 0.1f, 0.1f);
                    nk_property_float(ctx, "#Y:", -FLT_MAX,
                    ↪  beziator->points[i] + 1,
                                      FLT_MAX, 0.1f, 0.1f);
                    nk_property_float(ctx, "#Z:", -FLT_MAX,
                    ↪  beziator->points[i] + 2,
                                      FLT_MAX, 0.1f, 0.1f);
                }
                nk_layout_row_dynamic(ctx, 25, 1);
                nk_label(ctx, "Surfaces:", NK_TEXT_LEFT);
                for (size_t i = 0; i < beziator->surfaces_size; i++) {
                    for (size_t j = 0; j < 4; j++) {
                        nk_layout_row_dynamic(ctx, 25, 4);
                        for (size_t k = 0; k < 4; k++) {
                            nk_property_int(ctx, "#", 0,
                            ↪  (int*)(beziator->surfaces[i][j] + k),
                                            100, 1, 1);
                        }
                    }
                    nk_layout_row_dynamic(ctx, 25, 1);
                    nk_label(ctx, "---", NK_TEXT_LEFT);
                }
            }
            nk_end(ctx);
        }

        // render
        // ------
        glClearColor(bg.r, bg.g, bg.b, bg.a);
        glClearDepth(1.0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // input
        // -----
        app_process_input(app);

        // configure global opengl state
        // -----------------------------
        glEnable(GL_DEPTH_TEST);

        shader_use(sh);

        // pass projection matrix to shader
        glm_perspective(45.f * (GLfloat) M_RAD,
                        window_get_resolution(&app->window),
                        0.01f, 100.0f, app->projection);
        shader_set_mat4(sh, "projection", app->projection);
        shader_set_float(sh, "materialShininess", 32.0f);
        shader_set_vec3(sh, "objectColor", (vec3){0,1,0});

        // directional light
        camera_get_direction(&app->camera, direction);
        if (!light)
            shader_set_vec3(sh, "dirLight.direction", (vec3){0,-1,0});
        else
            shader_set_vec3(sh, "dirLight.direction", direction);
        shader_set_vec3f(sh, "dirLight.ambient", 0.0f, 0.0f, 0.0f);
        shader_set_vec3f(sh, "dirLight.diffuse", 0.6f, 0.6f, 0.6f);
        shader_set_vec3f(sh, "dirLight.specular", 0.f, 0.f, 0.f);

        // camera/view transformation
        camera_get_mat4((void*)&app->camera, app->last_camera);
```

```
            shader_set_vec3(sh, "viewPos", app->camera.position);
            shader_set_vec3f(sh, "objectColor", 0.4f, 0.8f, 0.4f);
            shader_set_float(sh, "materialShininess", 32.0f);
            shader_set_mat4(sh, "camera", app->last_camera);
            shader_set_vec2(sh, "vp", (vec2){(float)app->window.size[0],
            ↪  (float)app->window.size[1]});

            shader_set_float(sh, "alpha", 1.0);

            bijective_rotate_all(bijective, rotation);

            // Culling unneded back faced
            //glEnable(GL_CULL_FACE);
            //glCullFace(GL_BACK);
            //glFrontFace(GL_CCW);

            glEnable(GL_DEPTH_TEST);

            //glEnable(GL_BLEND);
            //glBlendEquation(GL_FUNC_ADD);
            //glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

            //glEnable(GL_ALPHA_TEST);
            //glAlphaFunc(GL_GREATER, 0.0f);

            glPointSize(10);
            if (!scene->wireframe)
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

            // render the loaded models
            for_each_utarr (Object, i, scene->objects){
                object_draw(i, app->mat4buf,
                            window_get_delta_time(&app->window));
            }

            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
            nk_glfw_render(&app->nkglfw);
            window_post_loop(&app->window);
        }
        return 0;
    }

    bool app_load_shader(App *app, const char *name) {
        Shader *shader = shader_create(name);
        if (shader == NULL) {
            printf("Failed to compile shaders.\n");
            return false;
        }
        HASH_ADD_STR(app->shaders, name, shader);
        return true;
    }

    void app_scroll(Window* window, GLdouble xoffset, GLdouble yoffset) {
        App *app = window_get_user_pointer(window);
        nk_glfw_scroll_callback(&app->nkglfw, xoffset, yoffset);
    }

    void app_mouse(Window* window, vec2 pos, vec2 offset) {
        App *app = window_get_user_pointer(window);

        if (app->interactive_mode) {
            vec2 offset_sens = GLM_VEC2_ZERO_INIT;
            glm_vec2_muladds(offset, -SENSITIVITY, offset_sens);
            //offset_sens[0] *= -1;

            camera_rotate(&app->camera, offset_sens[0], Y);
            camera_rotate(&app->camera, offset_sens[1], X);
        }
        nk_glfw_mouse_callback(&app->nkglfw, pos, offset);
    }

    void app_char_callback(Window* window, unsigned int codepoint) {
        App *app = window_get_user_pointer(window);
        nk_glfw_char_callback(&app->nkglfw, codepoint);
    }

    void app_mouse_button_callback(
            Window *window, UNUSED enum BUTTONS button,
            UNUSED bool pressed) {
        App *app = window_get_user_pointer(window);
        /*Object *object = (void*)utarray_eltptr(app->objects, 0);
        Model *model = object->model;
        int *size = window_get_size(&app->window);
        int *cursor = window_get_cursor(&app->window);
        vec4 cp;
        mat4 m;
        object_get_mat4(object, m);
        glm_mat4_mul(app->last_camera, m, m);
        glm_mat4_mul(app->projection, m, m);
        for (size_t i = 0; i < model->points_size; i++) {
            glm_mat4_mulv(m, model->points[i], cp);
            printf("%f %f %f\n", cp[0] / cp[2], cp[1] / cp[2], cp[2]);
        }
        printf("Tap: %f %f\n", (float)cursor[0] * 2.0 / (float)size[0] - 1.0,
            (float)cursor[1] * 2.0 / (float)size[1] - 1.0);*/
        nk_glfw_mouse_button_callback(&app->nkglfw, button, pressed);
    }

    // glfw: when the keyboard was used, this callback is called
    // -------------------------------------------------------------------
    void app_key(Window* window, enum KEYS key, bool pressed) {
        App *app = window_get_user_pointer(window);
        Scene *scene = &app->scene;
        int t = pressed ? 1 : -1;
        if (pressed) {
            switch (key) {
            case KEY_TAB:
                scene->active++;
                break;
            case KEY_Z:
                scene->wireframe = !scene->wireframe;
                break;
            case KEY_X:
                scene->light = (scene->light)? 0 : 1;
                break;
            case KEY_1:
                scene->dirLightOn = (scene->dirLightOn)? 0 : 1;
                break;
            case KEY_2:
                scene->pointLight1On = (scene->pointLight1On)? 0 : 1;
                break;
            case KEY_3:
                scene->pointLight2On = (scene->pointLight2On)? 0 : 1;
                break;
            case KEY_4:
                scene->spotLightOn = (scene->spotLightOn)? 0 : 1;
                break;
            case KEY_0:
                scene->dirLightOn = 0;
                scene->pointLight1On = 0;
                scene->pointLight2On = 0;
                scene->spotLightOn = 0;
                break;
            case KEY_M:
                scene_del(scene);
                scene_init(scene, app->objects);
                break;
            case KEY_B:
            {
                app->interactive_mode = !app->interactive_mode;
                if (!app->interactive_mode)
                    window_grab_mouse(window, false);
                else
                    window_grab_mouse(window, true);
            }
                break;
            default:
                break;
            }
        }
        switch (key) {
        case KEY_Q: app->trans[TRANSLATE][Y] -= t; break;
        case KEY_A: app->trans[TRANSLATE][X] -= t; break;
        case KEY_W: app->trans[TRANSLATE][Z] -= t; break;
        case KEY_S: app->trans[TRANSLATE][Z] += t; break;
        case KEY_E: app->trans[TRANSLATE][Y] += t; break;
        case KEY_D: app->trans[TRANSLATE][X] += t; break;
        case KEY_U: app->trans[ROTATE   ][Z] += t; break;
        case KEY_J: app->trans[ROTATE   ][Y] -= t; break;
        case KEY_I: app->trans[ROTATE   ][X] -= t; break;
        case KEY_K: app->trans[ROTATE   ][X] += t; break;
        case KEY_O: app->trans[ROTATE   ][Z] -= t; break;
        case KEY_L: app->trans[ROTATE   ][Y] += t; break;
        case KEY_R: app->trans[ANIMATE  ][Z] += t; break;
        case KEY_F: app->trans[ANIMATE  ][Y] -= t; break;
        case KEY_T: app->trans[ANIMATE  ][X] -= t; break;
        case KEY_G: app->trans[ANIMATE  ][X] += t; break;
        case KEY_Y: app->trans[ANIMATE  ][Z] -= t; break;
        case KEY_H: app->trans[ANIMATE  ][Y] += t; break;
        default:
            break;

        }
        if (scene->active > utarray_len(scene->objects)) scene->active = 0;
        nk_glfw_key_callback(&app->nkglfw, key, pressed);
    }

    static mat3 m3i = GLM_MAT3_IDENTITY_INIT;

    void app_process_input(App *app) {
        Window *window = &app->window;
        Scene *scene = &app->scene;
        Bijective bijective;
        float t;
        GLfloat delta_time = (GLfloat)window_get_delta_time(window);
        if (scene->active) {
            bijective.bijective.object = (void*)utarray_eltptr(
                        scene->objects, (unsigned)scene->active - 1);
            bijective.type = &object_bijective;
        } else {
            bijective.bijective.camera = &app->camera;
            bijective.type = &camera_bijective;
        }

        if (window_is_key_pressed(window, KEY_LEFT_CONTROL))
            delta_time *= 10;

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (app->trans[i][j]) {
                    t = (float)app->trans[i][j] * delta_time;
                    switch (i) {
                    case TRANSLATE:
                        bijective_translate(bijective, m3i[j], t);
                        break;
```

```
                case ROTATE:
                    bijective_rotate(bijective, (enum coord_enum)j, t);
                    break;
                case ANIMATE:
                    bijective_set_animation(bijective, m3i[j], t);
                    break;
                }
            }
        }
    }
}
```

Файл ./src/mesh.h

```
#ifndef MESH_H
#define MESH_H

#include "osdo.h"
#include "model.h"

typedef struct Vertex {
    vec3 position;
    vec3 normal;
    unsigned char color[4];
    vec2 uv;
} Vertex;

typedef struct Mesh {
    GLsizei vertices_size, indices_size;
    Vertex *vertices;
    GLuint *indices;
    GLuint vao, vbo, ebo;
} Mesh;

void mesh_init(Mesh* mesh);
void mesh_cube_update(Mesh* mesh);
Mesh *mesh_create(void);

void mesh_del(Mesh* mesh);
void mesh_free(Mesh* mesh);

void mesh_update(
        Mesh* mesh, GLsizei vertices_size, GLsizei indices_size,
        Vertex *vertices, GLuint *indices);
void mesh_clear(Mesh* mesh);

void mesh_draw(Mesh *mesh);
void mesh_draw_mode(Mesh *mesh, GLenum mode);

static const ModelType mesh_type = {
    mesh_draw, NULL, mesh_free,
};

#endif
```

Файл ./src/mesh.c

```
#include <string.h>
#include "mesh.h"
#include "conf.h"

void mesh_init(Mesh *mesh) {
    mesh->vertices_size = 0;
    mesh->indices_size = 0;
    mesh->vertices = NULL;
    mesh->indices = NULL;

    // create buffers/arrays
    glGenVertexArrays(1, &mesh->vao);
    glGenBuffers(1, &mesh->vbo);
    glGenBuffers(1, &mesh->ebo);
    glBindVertexArray(mesh->vao);
    glBindBuffer(GL_ARRAY_BUFFER, mesh->vbo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh->ebo);

    // set the vertex attribute pointers
    // vertex Positions
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (void*)offsetof(Vertex, position));

    // vertex normals
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (void*)offsetof(Vertex, normal));

    // vertex color
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 4, GL_UNSIGNED_BYTE, GL_TRUE, sizeof(Vertex),
                          (void*)offsetof(Vertex, color));

    // vertex uv
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (void*)offsetof(Vertex, uv));

    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}

void mesh_update(Mesh* mesh, GLsizei vertices_size, GLsizei indices_size,
↪  Vertex *vertices, GLuint *indices) {
    if (vertices != NULL && indices != NULL) {
        glBindVertexArray(mesh->vao);
        // load data into vertex buffers
        glBindBuffer(GL_ARRAY_BUFFER, mesh->vbo);
        glBufferData(GL_ARRAY_BUFFER,
                     (size_t)vertices_size * sizeof(Vertex),
                     vertices, GL_DYNAMIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh->ebo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                     (size_t)indices_size * sizeof(GLuint),
                     indices, GL_DYNAMIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
        glBindVertexArray(0);
    }

    mesh_clear(mesh);
    mesh->vertices_size = vertices_size;
    mesh->indices_size = indices_size;
    mesh->vertices = vertices;
    mesh->indices = indices;
}

void mesh_clear(Mesh* mesh) {
    if (mesh->vertices) {
        free(mesh->vertices);
        mesh->vertices = NULL;
    }
    if (mesh->indices) {
        free(mesh->indices);
        mesh->indices = NULL;
    }
    mesh->vertices_size = 0;
    mesh->indices_size = 0;
}


void mesh_cube_update(Mesh* mesh) {
    Vertex *V = (Vertex*)malloc(sizeof(EXAMPLE_CUBE_VERTEX));
    GLuint *E = (GLuint*)malloc(sizeof(EXAMPLE_CUBE_INDICIES));
    memcpy(V, EXAMPLE_CUBE_VERTEX, sizeof(EXAMPLE_CUBE_VERTEX));
    memcpy(E, EXAMPLE_CUBE_INDICIES, sizeof(EXAMPLE_CUBE_INDICIES));
    mesh_update(mesh, sizeof(EXAMPLE_CUBE_VERTEX)/sizeof(Vertex),
                sizeof(EXAMPLE_CUBE_INDICIES)/sizeof(GLuint), V, E);
}

Mesh *mesh_create(void) {
    Mesh *mesh = calloc(1, sizeof(Mesh));
    mesh_init(mesh);
    return mesh;
}

void mesh_draw_mode(Mesh *mesh, GLenum mode) {
    glBindVertexArray(mesh->vao);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh->ebo);
    glBindBuffer(GL_ARRAY_BUFFER, mesh->vbo);
    glDrawElements(mode, mesh->indices_size,
                   GL_UNSIGNED_INT, 0);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}

void mesh_draw(Mesh *mesh) {
    mesh_draw_mode(mesh, GL_TRIANGLES);
}

void mesh_del(Mesh *mesh) {
    glDeleteVertexArrays(1, &mesh->vao);
    glDeleteBuffers(1, &mesh->vbo);
    glDeleteBuffers(1, &mesh->ebo);
    mesh_clear(mesh);
}

void mesh_free(Mesh* mesh) {
    mesh_del(mesh);
    free(mesh);
}
```

Файл ./src/window.c

```
#include "window.h"
#include "conf.h"
#include <GLFW/glfw3.h>

typedef struct key_map_t {
    enum KEYS key;
    int glfw_key;
} key_map_t;
```

```c

typedef struct mouse_map_t {
    enum BUTTONS btn;
    int glfw_btn;
} mouse_map_t;

static const key_map_t key_list[] = {
    {KEY_DELETE,        GLFW_KEY_DELETE},
    {KEY_ENTER,         GLFW_KEY_ENTER},
    {KEY_TAB,           GLFW_KEY_TAB},
    {KEY_BACKSPACE,     GLFW_KEY_BACKSPACE},
    {KEY_UP,            GLFW_KEY_UP},
    {KEY_DOWN,          GLFW_KEY_DOWN},
    {KEY_LEFT,          GLFW_KEY_LEFT},
    {KEY_RIGHT,         GLFW_KEY_RIGHT},
    {KEY_HOME,          GLFW_KEY_HOME},
    {KEY_END,           GLFW_KEY_END},
    {KEY_PAGE_UP,       GLFW_KEY_PAGE_UP},
    {KEY_PAGE_DOWN,     GLFW_KEY_PAGE_DOWN},
    {KEY_LEFT_SHIFT,    GLFW_KEY_LEFT_SHIFT},
    {KEY_RIGHT_SHIFT,   GLFW_KEY_RIGHT_SHIFT},
    {KEY_LEFT_CONTROL,  GLFW_KEY_LEFT_CONTROL},
    {KEY_RIGHT_CONTROL, GLFW_KEY_RIGHT_CONTROL},

    {KEY_1, GLFW_KEY_1}, {KEY_2, GLFW_KEY_2}, {KEY_3, GLFW_KEY_3},
    {KEY_4, GLFW_KEY_4}, {KEY_5, GLFW_KEY_5}, {KEY_6, GLFW_KEY_6},
    {KEY_7, GLFW_KEY_7}, {KEY_8, GLFW_KEY_8}, {KEY_9, GLFW_KEY_9},
    {KEY_0, GLFW_KEY_0},

    {KEY_Q, GLFW_KEY_Q}, {KEY_W, GLFW_KEY_W}, {KEY_E, GLFW_KEY_E},
    {KEY_R, GLFW_KEY_R}, {KEY_T, GLFW_KEY_T}, {KEY_Y, GLFW_KEY_Y},
    {KEY_U, GLFW_KEY_U}, {KEY_I, GLFW_KEY_I}, {KEY_O, GLFW_KEY_O},
    {KEY_P, GLFW_KEY_P},

    {KEY_A, GLFW_KEY_A}, {KEY_S, GLFW_KEY_S}, {KEY_D, GLFW_KEY_D},
    {KEY_F, GLFW_KEY_F}, {KEY_G, GLFW_KEY_G}, {KEY_H, GLFW_KEY_H},
    {KEY_J, GLFW_KEY_J}, {KEY_K, GLFW_KEY_K}, {KEY_L, GLFW_KEY_L},

    {KEY_Z, GLFW_KEY_Z}, {KEY_X, GLFW_KEY_X}, {KEY_C, GLFW_KEY_C},
    {KEY_V, GLFW_KEY_V}, {KEY_B, GLFW_KEY_B}, {KEY_N, GLFW_KEY_N},
    {KEY_M, GLFW_KEY_M},
};

static const mouse_map_t mouse_list[] = {
    {MOUSE_BUTTON_LEFT, GLFW_MOUSE_BUTTON_LEFT},
    {MOUSE_BUTTON_MIDDLE, GLFW_MOUSE_BUTTON_MIDDLE},
    {MOUSE_BUTTON_RIGHT, GLFW_MOUSE_BUTTON_RIGHT},
};

static bool map_inited = false;
static int key_glfw_map[60];
static enum KEYS glfw_key_map[1024];
static int btn_glfw_map[60];
static enum BUTTONS glfw_btn_map[1024];

void map_init(void) {
    if (!map_inited) {
        map_inited = true;
        const void* end;
        end = key_list + sizeof(key_list) / sizeof(key_map_t);
        for (const key_map_t *i = key_list; i != end; i++) {
            key_glfw_map[i->key] = i->glfw_key;
            glfw_key_map[i->glfw_key] = i->key;
        }
        end = mouse_list + sizeof(mouse_list) / sizeof(mouse_map_t);
        for (const mouse_map_t *i = mouse_list; i != end; i++) {
            btn_glfw_map[i->btn] = i->glfw_btn;
            glfw_btn_map[i->glfw_btn] = i->btn;
        }
    }
}

void error_callback(int e, const char *d) {
    printf("Error %d: %s\n", e, d);
}

int window_init(Window *win) {
    // glfw: initialize and configure
    // ------------------------------
    map_init();
    glfwSetErrorCallback(error_callback);
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glEnable(GL_MULTISAMPLE);
    glfwWindowHint(GLFW_SAMPLES, 8);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "osdo",
                                          NULL, NULL);
    if (window == NULL) {
        printf("Failed to create GLFW window\n");
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);
    glfwSetWindowUserPointer(window, win);
    glfwSetFramebufferSizeCallback(window, window_resize_cb);
    glfwSetScrollCallback(window, window_scroll_cb);
    glfwSetCharCallback(window, window_char_cb);
    glfwSetMouseButtonCallback(window, window_mouse_button_cb);
    glfwSetCursorPosCallback(window, window_mouse_motion_cb);
    glfwSetKeyCallback(window, window_key_cb);

    // load glew
    // ---------
    glewExperimental = GL_TRUE;
    if (glewInit() != GLEW_OK) {
        printf("Failed to initialize GLEW\n");
        return -1;
    }

    win->size[0] = SCR_WIDTH;
    win->size[1] = SCR_HEIGHT;
    win->mouse_capute = false;
    win->window = window;
    win->current_time = glfwGetTime();
    win->last_time = win->current_time;
    win->delta_time = 0;
    win->scroll_cb = NULL;
    win->mouse_motion_cb = NULL;
    win->char_cb = NULL;
    win->mouse_button_cb = NULL;
    win->key_cb = NULL;
    return 0;
}

void window_del(UNUSED Window *window) {
    glfwTerminate();
}

bool window_alive(Window *window) {
    return !glfwWindowShouldClose(window->window);
}

bool window_pre_loop(Window *window) {
    if (glfwWindowShouldClose(window->window)) return false;
    glfwMakeContextCurrent(window->window);
    glfwPollEvents();
    //glfwWaitEvents();
    glfwGetWindowSize(window->window, window->size, window->size + 1);
    glfwGetFramebufferSize(
                window->window, window->display, window->display + 1);
    window->scale[0] = (float)window->display[0] / (float)window->size[0];
    window->scale[1] = (float)window->display[1] / (float)window->size[1];
    window->current_time = glfwGetTime();
    window->delta_time = window->current_time - window->last_time;
    window->last_time = window->current_time;
    return true;
}

void window_post_loop(Window *window) {
    glfwSwapBuffers(window->window);
}

void window_set_user_pointer(Window *window, void *pointer) {
    window->user_pointer = pointer;
}

void *window_get_user_pointer(Window *window) {
    return window->user_pointer;
}

float window_get_resolution(Window *window) {
    return (float)window->size[0] / (float)window->size[1];
}

double window_get_delta_time(Window *window) {
    return window->delta_time;
}

const char * window_get_clipboard(Window *window) {
     return glfwGetClipboardString(window->window);
}

void window_set_clipboard(Window *window, const char * str) {
    glfwSetClipboardString(window->window, str);
}

bool window_is_mouse_caputed(Window *window) {
    return window->mouse_capute;
}

void window_grab_mouse(Window *window, bool grab) {
    window->mouse_capute = grab;
    glfwSetInputMode(window->window, GLFW_CURSOR,
                     grab ? GLFW_CURSOR_DISABLED : GLFW_CURSOR_NORMAL);

}

bool window_is_key_pressed(Window *window, enum KEYS key) {
    return glfwGetKey(window->window, key_glfw_map[key]) == GLFW_PRESS;
}

bool window_is_mouse_pressed(Window *window, enum BUTTONS key) {
    return glfwGetMouseButton(
                window->window, btn_glfw_map[key]) == GLFW_PRESS;
}
```

```

void window_get_cursor(Window *window, vec2 dest) {
    glm_vec2_copy(window->cursor, dest);
}

void window_set_cursor(Window *window, vec2 coords) {
    glfwSetCursorPos(window->window, (double)coords[0], (double)coords[1]);
    glm_vec2_copy(coords, window->cursor);
}

int *window_get_size(Window *window) {
    return window->size;
}

int *window_get_display(Window *window) {
    return window->display;
}

float *window_get_scale(Window *window) {
    return window->scale;
}

void window_set_scroll_cb(Window *window, scroll_cb_t callback) {
    window->scroll_cb = callback;
}

void window_set_mouse_motion_cb(
        Window *window, mouse_motion_cb_t callback) {
    window->mouse_motion_cb = callback;
}

void window_set_char_cb(Window *window, char_cb_t callback) {
    window->char_cb = callback;
}
void window_set_mouse_button_cb(
        Window *window, mouse_button_cb_t callback) {
    window->mouse_button_cb = callback;
}

void window_set_key_cb(Window *window, key_cb_t callback) {
    window->key_cb = callback;
}

void window_resize_cb(
        struct GLFWwindow* window, GLint width, GLint height) {
    Window *win = glfwGetWindowUserPointer(window);
    glfwMakeContextCurrent(window);
    glViewport(0, 0, width, height);
    win->size[0] = width; win->size[1] = height;
}

void window_scroll_cb(
        struct GLFWwindow* window, GLdouble xoffset, GLdouble yoffset) {
    Window *win = glfwGetWindowUserPointer(window);
    win->scroll_cb(win, xoffset, yoffset);
}

void window_mouse_motion_cb(
        struct GLFWwindow* window, double xpos, double ypos) {
    Window *win = glfwGetWindowUserPointer(window);
    vec2 pos = {(float)xpos, (float)ypos}, offset;
    glm_vec2_sub(pos, win->cursor, offset);
    glm_vec2_copy(pos, win->cursor);
    win->mouse_motion_cb(win, pos, offset);
}

void window_char_cb(GLFWwindow* window, unsigned int codepoint) {
    Window *win = glfwGetWindowUserPointer(window);
    win->char_cb(win, codepoint);
}

void window_mouse_button_cb(
        GLFWwindow *window, int button, int action, UNUSED int mods) {
    Window *win = glfwGetWindowUserPointer(window);
    enum BUTTONS btn = glfw_btn_map[button];
    bool pressed = action == GLFW_PRESS;
    if (button == GLFW_MOUSE_BUTTON_LEFT && pressed) {
        double dt = glfwGetTime() - win->last_click_time;
        if (dt > NK_GLFW_DOUBLE_CLICK_LO && dt < NK_GLFW_DOUBLE_CLICK_HI) {
            btn = MOUSE_BUTTON_DOUBLE;
        }
        win->last_click_time = glfwGetTime();
    }
    if (pressed || action == GLFW_RELEASE)
        win->mouse_button_cb(win, btn, pressed);
}

void window_key_cb(GLFWwindow* window, int key, UNUSED int scancode,
                   int action, UNUSED int mods) {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    Window *win = glfwGetWindowUserPointer(window);
    bool pressed = action == GLFW_PRESS;
    if (pressed || action == GLFW_RELEASE)
        win->key_cb(win, glfw_key_map[key], pressed);
}
```

Файл ./src/scene.c

```

#include "scene.h"
#include "conf.h"

#include "app.h"

void scene_init(Scene *scene, UT_array *objects) {
    *scene = (Scene){
        NULL, 0, 1, 0, 1, 1, 1, 1
    };
    utarray_new(scene->objects, &object_icd);
    if (utarray_len(objects))
        utarray_inserta(scene->objects, objects, 0);
}

void scene_del(Scene *scene) {
    utarray_free(scene->objects);
}
```

Файл ./src/model.c

```

#include "model.h"

void model_init(Model *model, const char *name, model_t model_child, const
↪  ModelType *type) {
    strcpy(model->name, name);
    model->model = model_child;
    model->type = type;
}

Model *model_create(const char *name, model_t model, const ModelType *type) {
    Model *m = calloc(1, sizeof(Model));
    model_init(m, name, model, type);
    return m;
}

void model_draw(Model *model) {
    model->type->draw(model->model);
}

void model_generate(Model *model) {
    model->type->generate(model->model);
}

void model_free(Model *model) {
    model->type->free(model->model);
    free(model);
}
```

Файл ./src/object.h

```

#ifndef OBJECT_H
#define OBJECT_H

#include "osdo.h"

#include "bijective.h"
#include "model.h"
#include "shader.h"

typedef struct Object {
    mat4 transform;
    vec4 position;
    vec3 animation;
    Model *model;
    Shader *shader;
} Object;

void object_init(Object *object, Model *model, Shader *shader);
void object_init_empty(void *object);
void object_draw(Object *object, mat4 mat4buf, GLdouble delta_time);

void object_get_position_bijective(
        Object* object, vec4 **position);
void object_get_mat4(Object* object, mat4 dest);

void object_translate(Object* object, vec3 distances);
void object_translate_bijective(
        Object* object, vec3 distances, float delta_time);

void object_rotate(Object* object, float angle, enum coord_enum coord);
void object_rotate_all(Object* object, vec3 angles);
void object_rotate_bijective(
        Object* object, enum coord_enum coord, float delta_time);
void object_rotate_all_bijective(Object* object, vec3 angles);

void object_get_position(Object* object, vec4 dest);

void object_animate(Object* object, float step);
void object_get_animation(Object *object, vec3 **animation);
void object_set_animation(Object *object, vec3 angles, float delta_time);

void object_scale(Object *object, vec3 scale);

static const BijectiveType object_bijective = {
    object_get_position_bijective,
    object_get_mat4,
    object_translate_bijective,
```

```
    object_rotate_bijective,
    object_rotate_all_bijective,
    object_get_animation,
    object_set_animation
};

#define OBJECT_INIT(model, shader) {\
    GLM_MAT4_IDENTITY_INIT,\
    GLM_VEC4_BLACK_INIT,\
    GLM_VEC3_ZERO_INIT, model, shader}
#define OBJECT(model, shader) ((Object)OBJECT_INIT(model, shader))

#define OBJECT_INIT_EMPTY OBJECT_INIT(NULL, NULL)
#define OBJECT_EMPTY OBJECT(NULL, NULL)

static const UT_icd object_icd = {
    sizeof(Object), object_init_empty, NULL, NULL
};

#endif // OBJECT_H
```

Файл ./src/beziator.h

```
#ifndef BEZIATOR_H
#define BEZIATOR_H

#include "osdo.h"

#include "shader.h"
#include "mesh.h"
#include "model.h"

typedef vec4 *surface_t[4][4];

typedef struct Beziator {
    size_t points_size, surfaces_size;
    vec4 *points;
    surface_t *surfaces;
    Mesh mesh, frame, normals;
    Shader *editmode;
    Model model;
} Beziator;

bool beziator_init(
        Beziator *beziator, const char *name, Shader *editmode);
Beziator *beziator_create(const char *name, Shader *editmode);

void beziator_del(Beziator *beziator);
void beziator_free(Beziator *beziator);

void beziator_draw(Beziator *beziator);

bool beziator_generate(Beziator *beziator);

void beziator_save(Beziator *beziator);

static const ModelType beziator_type = {
    beziator_draw, beziator_generate, beziator_free,
};

#endif // BEZIATOR_H
```

Файл ./src/osdo.h

```
#ifndef OSDO_H
#define OSDO_H

#include <cglm/cglm.h>
#include <GL/glew.h>
#include <GL/gl.h>
#include <uthash.h>
#include <utarray.h>

#define for_each_utarr(type, item, list) \
    for(type *item = (type*)(void*)utarray_front(list); item != NULL; \
        item = (type*)(void*)utarray_next(list, item))

enum coord_enum {X = 0, Y = 1, Z = 2};

#endif // OSDO_H
```

Файл ./src/camera.c

```
#include "camera.h"
#include "conf.h"

void camera_get_direction(Camera* camera, vec4 dest) {
    mat4 matrix;
    camera_get_rotation_inv_mat4(camera, matrix);
    glm_mat4_mulv(matrix, CAMERA_DIRECTION, dest);
}

void camera_get_mat4(Camera *camera, mat4 dest) {
    camera_get_rotation_mat4(camera, dest);
    glm_translate(dest, camera->position);
}

void camera_get_rotation_mat4(Camera *camera, mat4 dest) {
    glm_mat4_copy(camera->rotation, dest);
    glm_mat4_inv(dest, dest);
}

void camera_get_rotation_inv_mat4(Camera *camera, mat4 dest) {
    glm_mat4_copy(camera->rotation, dest);
}

void camera_get_position_bijective(Camera *camera, vec4 **position) {
    *position = &camera->position;
}

void camera_translate(Camera *camera, vec3 distances) {
    glm_vec3_add(camera->position, distances, camera->position);
}

void camera_translate_bijective(
        Camera *camera, vec3 distances, float delta_time) {
    vec3 new_distances = GLM_VEC3_ZERO_INIT;
    mat4 rotation;
    camera_get_rotation_inv_mat4(camera, rotation);

    glm_vec3_muladds(distances, -OBJECT_MOVE_SPEED * delta_time,
                     new_distances);
    glm_vec3_rotate_m4(rotation, new_distances, new_distances);
    camera_translate(camera, new_distances);
}

void camera_rotate(Camera *camera, float angle, enum coord_enum coord) {
    switch (coord) {
    case X: glm_rotate_x(camera->rotation, angle, camera->rotation); break;
    case Y: glm_rotate_y(camera->rotation, angle, camera->rotation); break;
    case Z: glm_rotate_z(camera->rotation, angle, camera->rotation); break;
    }
    /*if (glm_vec3_dot(camera->rotation[2], GLM_XUP) > 0.1f) {
        glm_cross(camera->rotation[2], GLM_XUP, camera->rotation[1]);
        glm_cross(camera->rotation[1], camera->rotation[2],
↪   camera->rotation[0]);
        glm_normalize(camera->rotation[0]);
        glm_normalize(camera->rotation[1]);
    }*/
}

void camera_rotate_all(Camera *camera, vec3 angles) {
    camera_rotate(camera, angles[0], X);
    camera_rotate(camera, angles[1], Y);
    camera_rotate(camera, angles[2], Z);
}

void camera_rotate_bijective(
        Camera *camera, enum coord_enum coord, float delta_time) {
    camera_rotate(camera, -OBJECT_ROTATE_SPEED * delta_time, coord);
}

void camera_get_animation(Camera *camera, vec3 **animation) {
    *animation = &camera->animation;
}

void camera_set_animation(
        Camera *camera, vec3 angles, float delta_time) {
    vec3 animation;
    glm_vec3_muladds(angles, delta_time, animation);
    glm_vec3_add(camera->animation, animation,
                 camera->animation);
}

void camera_rotate_all_bijective(Camera *camera, vec3 angles) {
    camera_rotate_all(camera, angles);
}
```

Файл ./src/nkglfw.c

```
#include "nkglfw.h"
#include "window.h"

static const struct nk_draw_vertex_layout_element vertex_layout[] = {
    {NK_VERTEX_POSITION, NK_FORMAT_FLOAT, NK_OFFSETOF(Vertex, position)},
    {NK_VERTEX_COLOR, NK_FORMAT_R8G8B8A8, NK_OFFSETOF(Vertex, color)},
    {NK_VERTEX_TEXCOORD, NK_FORMAT_FLOAT, NK_OFFSETOF(Vertex, uv)},
    {NK_VERTEX_LAYOUT_END}
};

void nk_glfw_clipboard_paste(nk_handle usr, struct nk_text_edit *edit)
{
    Window* window = usr.ptr;
    const char *text = window_get_clipboard(window);
    if (text) nk_textedit_paste(edit, text, nk_strlen(text));
    (void)usr;
}

void nk_glfw_clipboard_copy(nk_handle usr, const char *text, int len)
{
    char *str = 0;
```

```
    if (!len) return;
    str = (char*)malloc((size_t)len+1);
    if (!str) return;
    memcpy(str, text, (size_t)len);
    str[len] = '\0';
    Window* window = usr.ptr;
    window_set_clipboard(window, str);
    free(str);
}

void nk_glfw_init(NkGlfw* nkglfw, Window *window, Shader *shader) {
    nk_init_default(&nkglfw->context, 0);
    nkglfw->context.clip.copy = nk_glfw_clipboard_copy;
    nkglfw->context.clip.paste = nk_glfw_clipboard_paste;
    nkglfw->context.clip.userdata = nk_handle_ptr(window);

    nk_buffer_init_default(&nkglfw->cmds);
    mesh_init(&nkglfw->mesh);
    nkglfw->shader = shader;
    nkglfw->window = window;
    struct nk_font_atlas *atlas;
    nk_glfw_font_stash_begin(nkglfw, &atlas);
    nk_glfw_font_stash_end(nkglfw);

    /* fill convert configuration */
    struct nk_convert_config *config = &nkglfw->config;
    memset(config, 0, sizeof(*config));
    config->vertex_layout = vertex_layout;
    config->vertex_size = sizeof(Vertex);
    config->vertex_alignment = NK_ALIGNOF(Vertex);
    config->null = nkglfw->null;
    config->circle_segment_count = 22;
    config->curve_segment_count = 22;
    config->arc_segment_count = 22;
    config->global_alpha = .95f;
    config->shape_AA = NK_ANTI_ALIASING_ON;
    config->line_AA = NK_ANTI_ALIASING_ON;

    glm_mat4_copy((mat4){
        {2.0f, 0.0f, 0.0f, 0.0f},
        {0.0f,-2.0f, 0.0f, 0.0f},
        {0.0f, 0.0f,-1.0f, 0.0f},
        {-1.0f,1.0f, 0.0f, 1.0f},
    }, nkglfw->ortho);
}

void nk_glfw_del(NkGlfw* nkglfw) {
    glDeleteTextures(1, &nkglfw->font_tex);
    nk_buffer_free(&nkglfw->cmds);
    mesh_del(&nkglfw->mesh);
}

void nk_glfw_font_stash_begin(NkGlfw* nkglfw,
                              struct nk_font_atlas **atlas) {
    nk_font_atlas_init_default(&nkglfw->atlas);
    nk_font_atlas_begin(&nkglfw->atlas);
    *atlas = &nkglfw->atlas;
}

void nk_glfw3_device_upload_atlas(NkGlfw* nkglfw, const void *image,
                                  int width, int height) {
    glGenTextures(1, &nkglfw->font_tex);
    glBindTexture(GL_TEXTURE_2D, nkglfw->font_tex);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                 (GLsizei)width, (GLsizei)height, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, image);
}

void nk_glfw_font_stash_end(NkGlfw* nkglfw) {
    const void *image;
    int w, h;
    image = nk_font_atlas_bake(&nkglfw->atlas, &w, &h, NK_FONT_ATLAS_RGBA32);
    nk_glfw3_device_upload_atlas(nkglfw, image, w, h);
    nk_font_atlas_end(&nkglfw->atlas, nk_handle_id((int)nkglfw->font_tex),
                      &nkglfw->null);
    if (nkglfw->atlas.default_font)
        nk_style_set_font(&nkglfw->context,
                          &nkglfw->atlas.default_font->handle);
}

void nk_glfw_begin_input(NkGlfw* nkglfw) {
    nk_input_begin(&nkglfw->context);
}

void nk_glfw_end_input(NkGlfw* nkglfw) {
    nk_input_end(&nkglfw->context);
}

void nk_glfw_render(NkGlfw* nkglfw) {
    int *size = window_get_size(nkglfw->window),
          *display = window_get_display(nkglfw->window);
    float *scale = window_get_scale(nkglfw->window);
    nkglfw->ortho[0][0] = 2.0f/(GLfloat)size[0];
    nkglfw->ortho[1][1] = -2.0f/(GLfloat)size[1];

    if (nkglfw->context.input.mouse.grab)
        window_grab_mouse(nkglfw->window, true);
    if (nkglfw->context.input.mouse.ungrab)
        window_grab_mouse(nkglfw->window, false);

    /* setup global state */
    glEnable(GL_BLEND);
    glBlendEquation(GL_FUNC_ADD);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDisable(GL_CULL_FACE);
    glDisable(GL_DEPTH_TEST);
    glEnable(GL_SCISSOR_TEST);
    glActiveTexture(GL_TEXTURE0);

    /* setup program */
    shader_use(nkglfw->shader);
    shader_set_int(nkglfw->shader, "frag_texture", 0);
    shader_set_mat4(nkglfw->shader, "projection", nkglfw->ortho);
    glViewport(0, 0, display[0], display[1]);
    {
        /* convert from command queue into draw list
         * and draw to screen */
        const struct nk_draw_command *cmd;
        const nk_draw_index *offset = NULL;

        /* allocate vertex and element buffer */
        glBindVertexArray(nkglfw->mesh.vao);
        glBindBuffer(GL_ARRAY_BUFFER, nkglfw->mesh.vbo);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, nkglfw->mesh.ebo);

        glBufferData(GL_ARRAY_BUFFER, MAX_VERTEX_BUFFER, NULL,
                     GL_STREAM_DRAW);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, MAX_ELEMENT_BUFFER, NULL,
                     GL_STREAM_DRAW);
        {
            /* load draw vertices & elements directly into vertex
             * + element buffer */
            struct nk_buffer vbuf, ebuf;
            void *vertices, *elements;
            vertices = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
            elements = glMapBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY);
            /* setup buffers to load vertices and elements */
            nk_buffer_init_fixed(
                        &vbuf, vertices, (size_t)MAX_VERTEX_BUFFER);
            nk_buffer_init_fixed(
                        &ebuf, elements, (size_t)MAX_ELEMENT_BUFFER);
            nk_convert(&nkglfw->context, &nkglfw->cmds,
                       &vbuf, &ebuf, &nkglfw->config);
            glUnmapBuffer(GL_ARRAY_BUFFER);
            glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
        }

        /* iterate over and execute each draw command */
        nk_draw_foreach(cmd, &nkglfw->context, &nkglfw->cmds)
        {
            if (!cmd->elem_count) continue;
            glBindTexture(GL_TEXTURE_2D, (GLuint)cmd->texture.id);
            glScissor(
                (GLint)(cmd->clip_rect.x * scale[0]),
                ((size[1] - (GLint)(cmd->clip_rect.y +
                                    cmd->clip_rect.h)) * (GLint)scale[1]),
                (GLint)(cmd->clip_rect.w * scale[0]),
                (GLint)(cmd->clip_rect.h * scale[1]));
            glDrawElements(GL_TRIANGLES, (GLsizei)cmd->elem_count,
                           GL_UNSIGNED_SHORT, offset);
            offset += cmd->elem_count;
        }
        nk_clear(&nkglfw->context);
        nk_buffer_clear(&nkglfw->cmds);
    }

    /* default OpenGL state */
    glUseProgram(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
    glDisable(GL_BLEND);
    glDisable(GL_SCISSOR_TEST);
}

void nk_gflw_scroll_callback(NkGlfw* nkglfw, double xoff, double yoff) {
    nk_input_scroll(&nkglfw->context, nk_vec2((float)xoff, (float)yoff));
}

void nk_glfw_mouse_callback(
        NkGlfw* nkglfw, vec2 pos, UNUSED vec2 offset) {
    struct nk_context *ctx = &nkglfw->context;
    nk_input_motion(ctx, (int)pos[0], (int)pos[1]);
}

void nk_glfw_char_callback(NkGlfw* nkglfw, unsigned int codepoint) {
    nk_input_unicode(&nkglfw->context, codepoint);
}

void nk_glfw_mouse_button_callback(
        NkGlfw* nkglfw, enum BUTTONS button, bool pressed) {
    vec2 cursor;
    window_get_cursor(nkglfw->window, cursor);
    int c[2] = {(int)cursor[0], (int)cursor[1]};
    struct nk_context *ctx = &nkglfw->context;
    switch (button) {
    case MOUSE_BUTTON_LEFT:
        nk_input_button(ctx, NK_BUTTON_LEFT, c[0], c[1],
                pressed);
        break;
    case MOUSE_BUTTON_MIDDLE:
        nk_input_button(ctx, NK_BUTTON_MIDDLE, c[0], c[1],
                pressed);
```

```
            break;
        case MOUSE_BUTTON_RIGHT:
            nk_input_button(ctx, NK_BUTTON_RIGHT, c[0], c[1],
                    pressed);
            break;
        case MOUSE_BUTTON_DOUBLE:
            nk_input_button(ctx, NK_BUTTON_DOUBLE, c[0], c[1],
                    pressed);
            break;
        }
    }

    void nk_glfw_key_callback(
            NkGlfw* nkglfw, enum KEYS key, bool pressed) {
        struct nk_context *ctx = &nkglfw->context;
        enum nk_keys k[16], inv[16];
        Window *win = nkglfw->window;
        int n = 0, m = 0;
        const bool ctrl =
                window_is_key_pressed(win, KEY_LEFT_CONTROL) ||
                window_is_key_pressed(win, KEY_RIGHT_CONTROL);
        switch (key) {
        case KEY_DELETE:      k[n++] = NK_KEY_DEL;              break;
        case KEY_ENTER:       k[n++] = NK_KEY_ENTER;            break;
        case KEY_TAB:         k[n++] = NK_KEY_TAB;              break;
        case KEY_BACKSPACE:   k[n++] = NK_KEY_BACKSPACE;        break;
        case KEY_UP:          k[n++] = NK_KEY_UP;               break;
        case KEY_DOWN:        k[n++] = NK_KEY_DOWN;             break;
        case KEY_PAGE_DOWN:   k[n++] = NK_KEY_SCROLL_DOWN;      break;
        case KEY_PAGE_UP:     k[n++] = NK_KEY_SCROLL_UP;        break;
        case KEY_C:           k[n++] = NK_KEY_COPY;             break;
        case KEY_V: if (ctrl) k[n++] = NK_KEY_PASTE;            break;
        case KEY_X: if (ctrl) k[n++] = NK_KEY_CUT;              break;
        case KEY_Z: if (ctrl) k[n++] = NK_KEY_TEXT_UNDO;        break;
        case KEY_R: if (ctrl) k[n++] = NK_KEY_TEXT_REDO;        break;
        case KEY_B: if (ctrl) k[n++] = NK_KEY_TEXT_LINE_START; break;
        case KEY_E: if (ctrl) k[n++] = NK_KEY_TEXT_LINE_END;   break;
        case KEY_LEFT:
            if (ctrl) k[n++] = NK_KEY_TEXT_WORD_LEFT;
            else k[n++] = NK_KEY_LEFT;
            break;
        case KEY_RIGHT:
            if (ctrl) k[n++] = NK_KEY_TEXT_WORD_RIGHT;
            else k[n++] = NK_KEY_RIGHT;
            break;
        case KEY_HOME:
            k[n++] = NK_KEY_TEXT_START; k[n++] = NK_KEY_SCROLL_START;
            break;
        case KEY_END:
            k[n++] = NK_KEY_TEXT_END;   k[n++] = NK_KEY_SCROLL_END;
            break;
        case KEY_LEFT_CONTROL:
        case KEY_RIGHT_CONTROL:
            if (window_is_key_pressed(win, KEY_V))
                k[n++] = NK_KEY_PASTE;
            if (window_is_key_pressed(win, KEY_X))
                k[n++] = NK_KEY_CUT;
            if (window_is_key_pressed(win, KEY_Z))
                k[n++] = NK_KEY_TEXT_UNDO;
            if (window_is_key_pressed(win, KEY_R))
                k[n++] = NK_KEY_TEXT_REDO;
            if (window_is_key_pressed(win, KEY_B))
                k[n++] = NK_KEY_TEXT_LINE_START;
            if (window_is_key_pressed(win, KEY_E))
                k[n++] = NK_KEY_TEXT_LINE_END;
            if (window_is_key_pressed(win, KEY_LEFT)) {
                k[n++] = NK_KEY_TEXT_WORD_LEFT;
                inv[m++] = NK_KEY_LEFT;
            } else {
                inv[m++] = NK_KEY_TEXT_WORD_LEFT;
                k[n++] = NK_KEY_LEFT;
            }
            if (window_is_key_pressed(win, KEY_RIGHT)) {
                k[n++] = NK_KEY_TEXT_WORD_RIGHT;
                inv[m++] = NK_KEY_RIGHT;
            } else {
                inv[m++] = NK_KEY_TEXT_WORD_RIGHT;
                k[n++] = NK_KEY_RIGHT;
            }
            break;
        default:
            break;
        }
        for (int i = 0; i < n; i++)
            if (k[i] != NK_KEY_NONE)
                nk_input_key(ctx, k[i], pressed);
        for (int i = 0; i < m; i++)
            if (inv[i] != NK_KEY_NONE)
                nk_input_key(ctx, inv[i], !pressed);
    }
```

Файл ./src/conf.h

```
#ifndef CONF_H
#define CONF_H

#include "osdo.h"
#include "mesh.h"

#define M_PI 3.14159265358979323846
#define M_RAD M_PI / 180
#define M_PI_F 3.14159265358979323846f
#define M_RAD_F M_PI_F / 180

#define RES_DIR "../share/osdo"

#ifdef __GNUC__
#define UNUSED __attribute__ ((unused))
#else
#define UNUSED
#endif
#define NK_INCLUDE_FIXED_TYPES
#define NK_INCLUDE_STANDARD_IO
#define NK_INCLUDE_STANDARD_VARARGS
#define NK_INCLUDE_DEFAULT_ALLOCATOR
#define NK_INCLUDE_VERTEX_BUFFER_OUTPUT
#define NK_INCLUDE_FONT_BAKING
#define NK_INCLUDE_DEFAULT_FONT
//#define NK_KEYSTATE_BASED_INPUT

#define MAX_VERTEX_BUFFER 512 * 1024
#define MAX_ELEMENT_BUFFER 128 * 1024

#define NK_GLFW_DOUBLE_CLICK_LO 0.02
#define NK_GLFW_DOUBLE_CLICK_HI 0.2

#define BASIS0POS ((vec3){ 0.0f, 0.0f,-32.0f})
#define BASIS1POS ((vec3){-8.0f, 0.0f,  0.0f})
#define BASIS2POS ((vec3){ 8.0f, 0.0f,  0.0f})

#define BASIS1ROT ((vec3){ 0.0f, 0.0f,  0.2f})
#define BASIS2ROT ((vec3){ 0.0f, 0.0f, -0.2f})

#define WINDOWS_NUM 2
#define MESHES_NUM 3

static const unsigned int SCR_WIDTH = 1366;
static const unsigned int SCR_HEIGHT = 700;

static const float OBJECT_MOVE_SPEED   =  5.0f;
static const float OBJECT_ROTATE_SPEED =  1.0f;
static const float OBJECT_ANIMATE_SPEED = 1.0f;

static const float SENSITIVITY = 0.01f;

static vec3 UNUSED LAMP_POSITIONS[] = {
    {5.0f, 0.0f, 5.0f},
    {-1.0f, 0.0f, 1.0f}
};

static const Vertex EXAMPLE_CUBE_VERTEX[] = {
    {{-1.,  1., -1.}, { 0.,  1.,  0.}, {  0, 255,   0, 255}, {0., 0.}},
    {{ 1.,  1.,  1.}, { 0.,  1.,  0.}, {255, 255, 255, 255}, {0., 0.}},
    {{ 1.,  1., -1.}, { 0.,  1.,  0.}, {255, 255,   0, 255}, {0., 0.}},

    {{ 1.,  1.,  1.}, { 0.,  0.,  1.}, {255, 255, 255, 255}, {0., 0.}},
    {{-1., -1.,  1.}, { 0.,  0.,  1.}, {  0,   0, 255, 255}, {0., 0.}},
    {{ 1., -1.,  1.}, { 0.,  0.,  1.}, {255,   0, 255, 255}, {0., 0.}},

    {{-1.,  1.,  1.}, {-1.,  0.,  0.}, {  0, 255, 255, 255}, {0., 0.}},
    {{-1., -1., -1.}, {-1.,  0.,  0.}, {  0,   0,   0, 255}, {0., 0.}},
    {{-1., -1.,  1.}, {-1.,  0.,  0.}, {  0,   0, 255, 255}, {0., 0.}},

    {{ 1., -1., -1.}, { 0., -1.,  0.}, {255,   0,   0, 255}, {0., 0.}},
    {{-1., -1.,  1.}, { 0., -1.,  0.}, {  0,   0, 255, 255}, {0., 0.}},
    {{-1., -1., -1.}, { 0., -1.,  0.}, {  0,   0,   0, 255}, {0., 0.}},

    {{ 1.,  1., -1.}, { 1.,  0.,  0.}, {255, 255,   0, 255}, {0., 0.}},
    {{ 1., -1.,  1.}, { 1.,  0.,  0.}, {255,   0, 255, 255}, {0., 0.}},
    {{ 1., -1., -1.}, { 1.,  0.,  0.}, {255,   0,   0, 255}, {0., 0.}},

    {{-1.,  1., -1.}, { 0.,  0., -1.}, {  0, 255,   0, 255}, {0., 0.}},
    {{ 1., -1., -1.}, { 0.,  0., -1.}, {255,   0,   0, 255}, {0., 0.}},
    {{-1., -1., -1.}, { 0.,  0., -1.}, {  0,   0,   0, 255}, {0., 0.}},

    {{-1.,  1., -1.}, { 0.,  1.,  0.}, {  0, 255,   0, 255}, {0., 0.}},
    {{-1.,  1.,  1.}, { 0.,  1.,  0.}, {  0, 255, 255, 255}, {0., 0.}},
    {{ 1.,  1.,  1.}, { 0.,  1.,  0.}, {255, 255, 255, 255}, {0., 0.}},

    {{ 1.,  1.,  1.}, { 0.,  0.,  1.}, {255, 255, 255, 255}, {0., 0.}},
    {{-1.,  1.,  1.}, { 0.,  0.,  1.}, {  0, 255, 255, 255}, {0., 0.}},
    {{-1., -1.,  1.}, { 0.,  0.,  1.}, {  0,   0, 255, 255}, {0., 0.}},

    {{-1.,  1.,  1.}, {-1.,  0.,  0.}, {  0, 255, 255, 255}, {0., 0.}},
    {{-1.,  1., -1.}, {-1.,  0.,  0.}, {  0, 255,   0, 255}, {0., 0.}},
    {{-1., -1., -1.}, {-1.,  0.,  0.}, {  0,   0,   0, 255}, {0., 0.}},

    {{ 1., -1., -1.}, { 0., -1.,  0.}, {255,   0,   0, 255}, {0., 0.}},
    {{ 1., -1.,  1.}, { 0., -1.,  0.}, {255,   0, 255, 255}, {0., 0.}},
    {{-1., -1.,  1.}, { 0., -1.,  0.}, {  0,   0, 255, 255}, {0., 0.}},

    {{ 1.,  1., -1.}, { 1.,  0.,  0.}, {255, 255,   0, 255}, {0., 0.}},
    {{ 1.,  1.,  1.}, { 1.,  0.,  0.}, {255, 255, 255, 255}, {0., 0.}},
    {{ 1., -1.,  1.}, { 1.,  0.,  0.}, {255,   0, 255, 255}, {0., 0.}},

    {{-1.,  1., -1.}, { 0.,  0., -1.}, {  0, 255,   0, 255}, {0., 0.}},
    {{ 1.,  1., -1.}, { 0.,  0., -1.}, {255, 255,   0, 255}, {0., 0.}},
    {{ 1., -1., -1.}, { 0.,  0., -1.}, {255,   0,   0, 255}, {0., 0.}},
};

static const GLuint EXAMPLE_CUBE_INDICIES[] = {
     0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
```

```
110         12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
111         24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
112     };
113
114     #endif // CONF_H
```