

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _____ «Информатика и системы управления»

КАФЕДРА _____ «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ
НА ТЕМУ:
«Здесь пишем тему»

Студент _____
(Группа) _____
(Подпись, дата) _____
(И.О. Фамилия) _____

Руководитель _____
(Подпись, дата) _____
(И.О. Фамилия) _____

2025 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 Теоретические основы морфинга	6
1.1 Понятие морфинга и его место в компьютерной графике	6
1.2 Историческое развитие морфинга	6
1.3 Математические основы морфинга	7
1.4 Алгоритмы морфинга	8
1.5 Применение морфинга	8
1.6 Методы морфинга трёхмерных объектов	9
1.7 Особенности морфинга куба и сферы	10
1.8 Оптимизация вычислений	10
1.9 Примеры использования морфинга в реальных проектах	11
1.10 Будущее морфинга	11
2 Описание реализации программы	13
2.1 Выбор алгоритма	13
2.2 Использование алгоритма	13
2.2.1 Генерация вершин сферы	13
2.2.2 Адаптация вершин куба	13
2.2.3 Вычисление нормалей	14
2.2.4 Вычисление текстурных координат	14
2.2.5 Передача данных в вершинный шейдер	14
2.2.6 Отображение и взаимодействие	15
2.3 Обоснование выбранных технологий	15
2.4 Общая структура проекта	15
2.5 Форматы представления данных	16
2.6 Структуры данных для внутреннего представления	16
2.7 Оценка теоретической сложности алгоритмов	16
2.8 Генерация объектов	16
2.8.1 Класс CUBE	17
2.8.2 Класс SPHERE	17
2.9 Реализация морфинга	18

2.10 Работа с событиями	18
2.11 Руководство администратора	21
2.11.1 Процедура инсталляции и деинсталляции	21
2.11.2 Параметры запуска из командной строки	22
2.11.3 Требования к аппаратному и системному программному обеспечению	22
2.12 Руководство пользователя	22
2.12.1 Описание графического интерфейса	22
2.12.2 Перечень сообщений об ошибках	22
 3 Экспериментальная часть	 23
3.1 Демонстрация работы программы	23
3.2 Визуальная оценка качества	24
3.3 Оценка производительности	24
3.4 Анализ результатов	26
3.5 Обсуждение возможных улучшений	26
 ЗАКЛЮЧЕНИЕ	 28
 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	 30
 ПРИЛОЖЕНИЕ А	 31

ВВЕДЕНИЕ

Современная компьютерная графика активно используется в различных сферах, включая разработку игр, создание анимации, визуализацию научных данных и моделирование. В основе многих графических технологий лежат преобразования объектов, которые позволяют менять их положение, форму и размеры. Однако простых операций, таких как поворот, масштабирование и перемещение, недостаточно для создания сложных визуальных эффектов. Здесь на помощь приходят более сложные методы, например, морфинг — плавное преобразование одной формы в другую.

Данная работа посвящена изучению и реализации морфинга трёхмерных объектов, на примере перехода между кубом и сферой, сделанных на лабораторных по компьютерной графике. Выбор темы обусловлен желанием изучить алгоритмы морфинга и их применение для анимации. Это исследование дополняет базовые знания, полученные в ходе изучения курса компьютерной графики, и позволяет расширить практический опыт работы с такими инструментами, как OpenGL и Python.

Целью данной работы является изучение и реализация алгоритма морфинга 3D-объектов с использованием современных технологий.

Для достижения цели поставлены следующие задачи:

1. Изучить теоретические основы морфинга и его применение.
2. Реализовать базовые формы (куб и сферу) с использованием треугольников (`GL_TRIANGLE_STRIP`), что обеспечит удобство отображения и высокую производительность.
3. Разработать алгоритм плавного перехода между объектами с использованием шейдеров.
4. Обеспечить взаимодействие с пользователем, позволяя изменять параметры морфинга в реальном времени.
5. Провести экспериментальное исследование работы программы, оценив её производительность и визуальное качество.

Работа организована следующим образом: в первой главе рассматриваются теоретические основы морфинга, включая его области применения и методы. Вторая глава посвящена описанию архитектуры программы и реализации алгоритма.

В третьей главе представлены результаты тестирования, а также анализ качества и производительности программы. Заключение содержит выводы и направления дальнейших исследований.

1 Теоретические основы морфинга

1.1 Понятие морфинга и его место в компьютерной графике

Морфинг (от англ. morphing — «превращение») представляет собой процесс плавного перехода одной формы объекта в другую. Этот метод широко используется в компьютерной графике для создания визуальных эффектов в анимации, фильмах, играх и других мультимедийных приложениях. Морфинг включает интерполяцию геометрических и визуальных характеристик объектов, таких как вершины, текстуры и освещение, что позволяет добиться реалистичного изменения формы [1].

В источнике [2] сказано, что одним из первых примеров использования морфинга в кино стал фильм "Вилли Вонка и шоколадная фабрика"(1971), где морфинг был использован для создания спецэффектов. С тех пор технология морфинга значительно эволюционировала, и сегодня она используется в самых разных областях, от медицины до видеоигр.

В основе морфинга лежат два ключевых этапа:

- определение соответствия между вершинами начального и конечного объектов;
- вычисление промежуточных состояний объектов через интерполяцию.

1.2 Историческое развитие морфинга

История морфинга начинается с ранних экспериментов в области компьютерной графики в 1970-х и 1980-х годах. В это время исследователи начали разрабатывать алгоритмы для плавного перехода одной формы в другую. Одним из первых значительных достижений в этой области стала работа Томаса Бера и Скотта Уэйверли, которые представили метод морфинга, основанный на интерполяции ключевых точек [3].

В 1990-х годах морфинг начал активно использоваться в кинематографе. Фильмы, такие как "Терминатор 2: Судный день"(1991) и "Маска"(1994), продемонстрировали возможности морфинга для создания спецэффектов. В этих филь-

макс морфинг использовался для создания эффекта превращения одного персонажа в другого, что стало возможным благодаря развитию компьютерных технологий и улучшению алгоритмов интерполяции.

Морфинг также находит применение в области научной визуализации, где он используется для отображения динамических процессов, таких как рост кристаллов или изменения в биологических структурах. В медицине морфинг позволяет визуализировать изменения в анатомических структурах, например, рост опухолей или изменения в костной ткани [4]. В архитектуре морфинг используется для моделирования изменений в зданиях и сооружениях, что позволяет архитекторам визуализировать изменения в форме зданий и изучать их влияние на окружающую среду.

1.3 Математические основы морфинга

Морфинг основан на математических методах интерполяции, которые позволяют плавно изменять форму объекта. Как сказано в [5], основные методы интерполяции включают:

- **Линейную интерполяцию:** простейший метод, при котором промежуточные точки вычисляются как среднее значение между начальной и конечной точками. Формула линейной интерполяции выглядит следующим образом: $P(t) = (1 - t) * P_0 + t * P_1$, где P_0 и P_1 — начальная и конечная точки, а t — параметр интерполяции, изменяющийся от 0 до 1.
- **Кубическую интерполяцию:** более сложный метод, который учитывает не только начальные и конечные точки, но и их производные, что позволяет добиться более плавного перехода. Формула кубической интерполяции выглядит следующим образом: $P(t) = a*t^3 + b*t^2 + c*t + d$, где a, b, c, d — коэффициенты, определяемые на основе начальных и конечных точек и их производных.
- **Сплайновую интерполяцию:** метод, использующий сплайны для создания гладких кривых, которые проходят через заданные точки.

Кроме того, в [6] сказано, что существуют и другие методы интерполяции, такие как бикубическая интерполяция и интерполяция с использованием радиальных базисных функций. Эти методы позволяют добиться еще более плавных и реалистичных переходов между формами объектов.

1.4 Алгоритмы морфинга

Алгоритмы морфинга можно разделить на несколько категорий в зависимости от используемых методов интерполяции и подходов к реализации. Основные категории, согласно [7], включают:

- **Геометрический морфинг:** этот метод фокусируется на интерполяции геометрических характеристик объектов, таких как вершины и грани. Геометрический морфинг широко используется в компьютерной графике для создания анимаций и спецэффектов.
- **Текстурный морфинг:** этот метод фокусируется на интерполяции текстурных характеристик объектов, таких как цвет и текстура. Текстурный морфинг используется для создания реалистичных переходов между текстурами объектов.
- **Гибридный морфинг:** этот метод комбинирует геометрический и текстурный морфинг для создания более сложных и реалистичных переходов. Гибридный морфинг используется в кинематографе и видеоиграх для создания спецэффектов и анимаций.

1.5 Применение морфинга

Морфинг нашёл своё применение в различных областях:

- **Кинематограф:** создание спецэффектов, например, превращение одного персонажа в другого. Примером может служить фильм "Терминатор 2: Судный день"(1991), где морфинг был использован для создания эффекта превращения робота Т-1000.
- **Научная визуализация:** плавное отображение изменения данных, таких как модели молекул или результаты симуляций. Морфинг позволяет визуализировать динамические процессы, такие как рост кристаллов или изменения в биологических структурах.
- **Игровая индустрия:** анимация персонажей, трансформация объектов и переходы между формами. В играх морфинг используется для создания реалистичных анимаций, таких как превращение персонажа в другого или изменение формы объектов.

- **Дизайн:** моделирование прототипов и изучение переходов между концепциями. Морфинг позволяет дизайнерам визуализировать изменения в форме объектов и изучать их влияние на конечный продукт.
- **Медицина:** визуализация изменений в анатомических структурах, например, рост опухолей или изменения в костной ткани. Морфинг позволяет врачам наблюдать за динамикой изменений в организме пациента и планировать лечение.
- **Архитектура:** моделирование изменений в зданиях и сооружениях, например, визуализация строительных проектов. Морфинг позволяет архитекторам визуализировать изменения в форме зданий и изучать их влияние на окружающую среду.
- **Образование:** использование морфинга для создания наглядных учебных материалов, таких как анимации и симуляции, которые помогают студентам лучше понять сложные концепции.
- **Реклама:** создание эффектных визуальных эффектов для рекламных роликов и презентаций, что позволяет привлечь внимание зрителей и сделать рекламу более запоминающейся.
- **Искусственный интеллект:** использование морфинга для создания реалистичных анимаций в виртуальных ассистентах и роботах, что позволяет улучшить взаимодействие с пользователями.

1.6 Методы морфинга трёхмерных объектов

Существует несколько подходов к реализации морфинга в трёхмерной графике:

1. **Линейный морфинг.** Этот метод предполагает линейную интерполяцию позиций вершин начального и конечного объектов. Преимущество метода в его простоте, однако он может создавать визуально некорректные результаты, если структуры объектов сильно различаются.
2. **Нелинейный морфинг.** В данном подходе интерполяция может быть модифицирована функциями, учитывающими форму объектов. Это позволяет создать более реалистичные переходы, но увеличивает сложность реализации. Одним из примеров нелинейного морфинга является использование кубической интерполяции.

3. **Морфинг с использованием шейдеров.** Этот метод позволяет выполнять вычисления на графическом процессоре (GPU), что значительно ускоряет обработку и позволяет добиться плавности анимации даже при сложных преобразованиях.
4. **Морфинг с использованием физических моделей.** Этот метод использует физические законы для моделирования переходов между объектами, что позволяет создать более реалистичные и естественные анимации. Пример использования физических моделей для морфинга включает в себя моделирование деформаций объектов под воздействием внешних сил. Это позволяет создать более реалистичные анимации, такие как превращение одного объекта в другой под воздействием гравитации или других физических сил.

1.7 Особенности морфинга куба и сферы

Куб и сфера представляют собой объекты с разной структурой и количеством вершин. Куб состоит из шести граней, каждая из которых образована четырьмя вершинами. Сфера, напротив, имеет гладкую поверхность, которая аппроксимируется большим количеством треугольников.

В данной работе для удобства отображения и повышения производительности оба объекта описываются набором треугольников, упорядоченных с использованием `GL_TRIANGLE_STRIP`. Такой подход позволяет упростить обработку данных и повысить производительность за счёт эффективного использования графического процессора.

Для реализации морфинга между кубом и сферой важно обеспечить одинаковое количество вершин и корректное их соответствие. Это достигается путём равномерного разбиения поверхностей объектов и использования алгоритмов для поиска ближайших точек.

1.8 Оптимизация вычислений

Для улучшения производительности морфинга можно использовать различные методы оптимизации:

- **Использование GPU:** выполнение вычислений на графическом процессоре позволяет значительно ускорить обработку данных.
- **Оптимизация геометрии:** уменьшение количества вершин и использование более эффективных структур данных для хранения геометрии объектов.
- **Параллельные вычисления:** использование многопоточности и параллельных вычислений для ускорения обработки данных.
- **Алгоритмы сжатия:** использование алгоритмов сжатия для уменьшения объема данных, что позволяет ускорить передачу и обработку данных.
- **Кэширование:** использование кэширования для хранения промежуточных результатов, что позволяет избежать повторных вычислений и ускорить процесс морфинга.
- **Оптимизация шейдеров:** использование оптимизированных шейдеров для ускорения вычислений на GPU. Это включает в себя использование более эффективных алгоритмов и структур данных в шейдерах.

1.9 Примеры использования морфинга в реальных проектах

Морфинг находит широкое применение в реальных проектах, таких как создание спецэффектов в кино, анимация персонажей в видеоиграх и визуализация данных в научных исследованиях. Например, в фильме "Аватар"(2009) морфинг был использован для создания реалистичных анимаций персонажей и окружающей среды, что показано в [8]. В видеоигре "The Legend of Zelda: Breath of the Wild"(2017) морфинг был использован для создания плавных переходов между различными формами объектов и персонажей.

1.10 Будущее морфинга

С развитием технологий морфинг продолжает эволюционировать и находить новые области применения. В будущем можно ожидать улучшения качества и реалистичности морфинга за счет использования более сложных математических

моделей и алгоритмов. Также можно ожидать развития технологий виртуальной и дополненной реальности, где морфинг будет играть ключевую роль в создании реалистичных и интерактивных визуальных эффектов.

2 Описание реализации программы

2.1 Выбор алгоритма

Для реализации морфинга между кубом и сферой был выбран алгоритм линейной интерполяции. Этот алгоритм позволяет плавно переходить от одной формы к другой, используя параметр `morphFactor`, который изменяется от 0 до 1. Реализация морфинга осуществляется в вершинном шейдере, что позволяет эффективно использовать возможности GPU для параллельных вычислений. В вершинном шейдере выполняется интерполяция позиций вершин между начальной (куб) и конечной (сфера) формами. Это обеспечивает высокую производительность и плавность анимации, так как все вычисления выполняются на GPU, что значительно снижает нагрузку на центральный процессор (CPU).

2.2 Использование алгоритма

Алгоритм морфинга между кубом и сферой включает несколько ключевых этапов, начиная с генерации вершин и заканчивая их интерполяцией в вершинном шейдере. Подробное описание использования алгоритма представлено ниже.

2.2.1 Генерация вершин сферы

Первым шагом является генерация вершин сферы. Поверхность сферы аппроксимируется сеткой широт и долгот, что обеспечивает гладкость поверхности. Количество вершин сферы определяется параметрами `lat_segments` (количество сегментов широты) и `lon_segments` (количество сегментов долготы).

2.2.2 Адаптация вершин куба

Исходя из количества полученных вершин сферы, необходимо адаптировать количество вершин куба для обеспечения совместимости с морфингом. Этот процесс включает несколько шагов:

1. **Разбиение граней куба на сетку точек.** Каждая грань куба разбивается на равномерную сетку точек. Количество точек на каждой грани определяется параметром `divisions`, который указывает количество отрезков,

на которые делится ребро куба. Общее количество вершин в кубе после этого шага можно посчитать по формуле: $number\ of\ vertices = 2 * (divisions + 1)^2 - 2 * (divisions + 1)$

2. **Удаление полос точек.** Если количество вершин после первого шага превышает необходимое количество, удаляются одни или несколько полос точек на каждой грани. Точки удаляются полосами для корректной отрисовки треугольников в дальнейшем. Параметр `divisions` подбирается таким образом, чтобы минимизировать разницу между необходимым количеством точек и количеством точек после удаления полос.
3. **Добавление точек.** Если после удаления полос точек все еще недостаточно, добавляются дополнительные точки. Точки добавляются равномерно в грани. Если в грани нужно добавить три или более точек, то точки добавляются дублированием различных треугольников. Если осталось добавить одну или две точки, то дублируется последняя точка в грани. Это сделано для правильного поиска нормалей в дальнейшем и соответственно корректного отображения куба с текстурой.

2.2.3 Вычисление нормалей

После генерации вершин куба и сферы вычисляются нормали для каждой точки. Нормали для сферы вычисляются просто как нормализованные радиус-векторы для точек сферы. Нормали для куба вычисляются на основе его геометрии.

2.2.4 Вычисление текстурных координат

Далее вычисляются текстурные координаты для куба и сферы. Эти координаты также будут меняться с помощью морфинга.

2.2.5 Передача данных в вершинный шейдер

Все найденные данные (вершины, нормали, текстурные координаты) передаются в вершинный шейдер с использованием библиотеки OpenGL. В вершинном шейдере выполняется интерполяция этих данных между начальной (куб) и конечной (сфера) формами. Переменная для интерполяции `morphFactor` вычисляется в

программе с учетом течения времени и изменяется в зависимости от направления преобразования.

2.2.6 Отображение и взаимодействие

С помощью библиотеки OpenGL происходит отображение полученных фигур. Пользователь может взаимодействовать с программой в реальном времени, управляя различными параметрами морфинга и отображения объектов с помощью клавиатуры и мыши.

2.3 Обоснование выбранных технологий

Для реализации комплекса программ были выбраны следующие языки программирования, технологии и сторонние библиотеки:

- **Python:** Язык программирования, обеспечивающий высокую производительность и удобство разработки.
- **OpenGL:** Библиотека для работы с 3D-графикой, позволяющая использовать возможности GPU для рендеринга.
- **GLFW:** Библиотека для создания окон и обработки событий.
- **NumPy:** Библиотека для работы с массивами данных, обеспечивающая высокую производительность вычислений.

2.4 Общая структура проекта

Архитектура проекта разделена на несколько модулей, каждый из которых отвечает за свою функциональность:

- `main.py` — основной модуль программы, содержащий логику запуска приложения, настройки шейдеров и обработки морфинга.
- `callbacks.py` — обработка событий, таких как нажатия клавиш, движение мыши и прокрутка колеса.
- `utils.py` — вспомогательные функции для работы с матрицами, текстурами и шейдерами.
- `shape.py` — генерация трёхмерных объектов (куб и сфера).
- `window_manager.py` — управление окном и настройками OpenGL.

- `fragment_shader.frag` — фрагментный шейдер.
- `vertex_shader.vert` — вершинный шейдер.

Подобная структура обеспечивает модульность и удобство разработки, позволяя независимо модифицировать и тестировать отдельные компоненты программы.

2.5 Форматы представления данных

Входные данные представляют собой количество сегментов широты и количество сегментов долготы. Выходные данные включают в себя промежуточные и окончательные результаты морфинга, которые отображаются на экране.

2.6 Структуры данных для внутреннего представления

Для внутреннего представления данных используются массивы вершин и нормалей, которые хранятся в формате `numpy.array`. Это позволяет эффективно работать с данными и передавать их на GPU для рендеринга.

2.7 Оценка теоретической сложности алгоритмов

Теоретическая сложность реализуемых алгоритмов оценивается следующим образом:

Генерация вершин куба и сферы: $O(n)$, где n — количество вершин. Морфинг: $O(1)$ для каждой вершины, так как интерполяция выполняется за константное время.

2.8 Генерация объектов

Основой работы программы является генерация трёхмерных объектов: куба и сферы. Оба объекта строятся с использованием треугольников (`GL_TRIANGLE_STRIP`), что упрощает их рендеринг и повышает производительность.

2.8.1 Класс CUBE

Куб представлен классом CUBE, который создаёт его вершины и нормали. Для обеспечения совместимости с морфингом количество точек на поверхности куба адаптируется к числу вершин сферы. За это отвечает метод `_subdivide_face`, показанный в листинге 3, который разбивает каждую грань куба на равномерную сетку точек в нужном порядке. В качестве входного параметра метод принимает `total_expected_points`, которое указывает общее ожидаемое количество точек на всём кубе. Упрощенный вариант класса представлен в листинге 1.

Листинг 1: Код генерации куба

```
1 class CUBE(Shape):
2     def __init__(self, total_expected_points):
3         super().__init__()
4         self.vertices = np.array([...]) # координаты 8 вершин
5         self.faces = [...] # описание граней
6         self._subdivide_face(total_expected_points) # разбиение куба на
7             # нужное количество вершин
8         self._update_normals() # обновление нормалей
9
10    def _custom_shuffle(self, arr, a, b):
11        # Метод для переставления точек в нужном порядке на отрезке [a,
12            # b]
13        # Вызывается только в _subdivide_face
14
15        return shuffled_arr
16
17    def _update_normals(self):
18        # Метод для обновления нормалей
```

2.8.2 Класс SPHERE

Сфера создаётся классом SPHERE, в котором поверхность аппроксимируется сеткой широт и долгот. Использование триангуляции для генерации вершин обеспечивает гладкость поверхности. Пример кода представлен в листинге 2.

Листинг 2: Код генерации сферы

```
1 class SPHERE(Shape):
2     def __init__(self, lat_segments, lon_segments, radius=0.5):
3         super().__init__()
4         self.lat_segments = lat_segments
5         self.lon_segments = lon_segments
6         self.radius = radius
7         self.generate_sphere() # Генерация точек сферы с учетом порядка
    → для однозначного отображения с помощью GL_TRIANGLE_STRIP
```

2.9 Реализация морфинга

Морфинг реализован с использованием шейдеров, которые обрабатывают данные о вершинах объектов на графическом процессоре. Логика морфинга основывается на интерполяции позиций и нормалей вершин между начальной (куб) и конечной (сфера) формами.

На уровне программы интерполяция контролируется переменной morphFactor, изменяющейся от 0 (куб) до 1 (сфера). Код обработки переменной морфинга приведён в 4.

На GPU выполняются вычисления по смешиванию данных вершин, как показано в листинге 5.

2.10 Работа с событиями

Программа поддерживает взаимодействие с пользователем. Основные события включают:

- Нажатие клавиши M для включения/выключения морфинга.
- Нажатие клавиши F для переключение между отображением каркаса и заполненной модели.
- Нажатие клавиши T для включения/выключения текстуры.
- Нажатие \uparrow / \downarrow для увеличения/уменьшения объекта.
- Управление скоростью морфинга клавишами + и -.
- Нажатие клавиши Space для сброса настроек.
- Нажатие клавиши Escape для закрытия окна.

Листинг 3: Код метода `_subdivide_face` класса CUBE

```
1 def _subdivide_face(self, total_expected_points):
2     # Разбиение каждой грани на одинаковое количество точек
3     points = []
4     for k, face in enumerate(self.faces):
5         # v1, v2, v3, v4 - вершины грани
6         for i in range(divisions):
7             for j in range(divisions + 1):
8                 p1 = v1 + (v2 - v1) * (j / divisions)
9                 p2 = v4 + (v3 - v4) * (j / divisions)
10                point1 = p1 + (p2 - p1) * (i / divisions)
11                point2 = p1 + (p2 - p1) * ((i + 1) / divisions)
12
13                # Добавляем поочерёдно точки двух полос для получения
14                # треугольников
15                points.append(point1)
16                points.append(point2)
17
18    # Если точек больше, чем нужно, то удаляем одинаковое количество
19    # линий с каждой грани
20    if len(points) > total_expected_points:
21        ...
22
23    # Добавляем недостающее количество точек
24    missing_points = total_expected_points - len(points)    #
25    # Количество нехватавших точек
26    num_faces = 6
27    face_size = len(points) // num_faces    # Массив точек всегда можно
28    # разделить на равные 6 частей (грани)
29    for face_index in range(num_faces - 1, -1, -1):
30        end = (face_index + 1) * face_size if face_index < num_faces
31        else len(points)
32        # Вычисляем, сколько точек добавить в текущую грань
33        points_to_add = missing_points // num_faces
34        if face_index < missing_points % num_faces:
35            points_to_add += 1    # Распределяем остаток равномерно
36
37        i = end - 3
38        while points_to_add >= 3:    # Добавляем повторением
39            # треугольников
40            point = points[i:i + 3]
41            for j in range(2, -1, -1):
42                points.insert(i, point[j])
43            points_to_add -= 3
44            i -= 3
45        end += 3
46
47    self.vertices = np.array(points, dtype=np.float32)
```

Листинг 4: Алгоритм управления фактором морфинга

```

1 def morph_factor(t, last_time, state):
2     EPSILON = 1e-6 # Погрешность для проверки границ
3     PAUSE_TIME = 500 # Время задержки на границах в миллисекундах
4
5     time = glutGet(GLUT_ELAPSED_TIME)
6     if state.get('morph') == True:
7         current_time = time # Текущее время
8         # Проверка: задерживаем t на границах
9         if t >= 1.0 - EPSILON:
10             t = 1.0 # Для правильного отображения
11             if current_time - last_time > PAUSE_TIME:
12                 if state.get('direction') > 0:
13                     state['direction'] *= -1 # Меняем направление движения
14                     last_time = current_time # Обновляем время последней паузы
15                     t += 0.001 * state.get('direction')
16             elif t <= 0.0 + EPSILON:
17                 t = 0.0 # Для правильного отображения
18                 if current_time - last_time > PAUSE_TIME:
19                     if state.get('direction') < 0:
20                         state['direction'] *= -1 # Меняем направление движения
21                         last_time = current_time # Обновляем время последней паузы
22                         t += 0.001 * state.get('direction')
23             else:
24                 # t движется непрерывно внутри диапазона (0, 1)
25                 t += 0.001 * state.get('direction')
26                 last_time = current_time
27
28     return t, last_time

```

Листинг 5: Реализация морфинга в вершинном шейдере

```

1 uniform float morphFactor;
2 in vec3 cubePosition;
3 in vec3 spherePosition;
4
5 void main() {
6     vec3 morphedPosition = mix(cubePosition, spherePosition, morphFactor);
7     gl_Position = projection * view * model * vec4(morphedPosition, 1.0);
8 }

```

— Прокрутка колеса мыши для изменения масштаба объекта.

— Перемещение мыши для вращения объекта.

Логика обработки событий реализована в модуле callbacks.py, пример кода приведён в листингах 6 и 7.

Листинг 6: Обработка событий клавиатуры

```
1 def key_callback(window, key, scancode, action, mods, state):
2     if action == glfw.PRESS:
3         if key == glfw.KEY_M:
4             state['morph'] = not state['morph']
```

Листинг 7: Обработка событий мыши

```
1 def mouse_button_callback(window, button, action, mods, state):
2     if button == glfw.MOUSE_BUTTON_LEFT:
3         if action == glfw.PRESS:
4             state['mouse_pressed'] = True
5         elif action == glfw.RELEASE:
6             state['mouse_pressed'] = False
7
8 def cursor_position_callback(window, xpos, ypos, state):
9     """
10     Обработка движения курсора.
11     """
12     if state.get('mouse_pressed', False):
13         dx = xpos - state['last_mouse_pos'][0]
14         dy = ypos - state['last_mouse_pos'][1]
15         state['rotation_angle_x'] += dy * 0.2 # Скорость вращения
16         state['rotation_angle_y'] -= dx * 0.2
17     state['last_mouse_pos'] = [xpos, ypos]
18
19 def scroll_callback(window, xoffset, yoffset, state):
20     """
21     Обработка прокрутки колеса.
22     """
23     if xoffset > 0:
24         state['size'] -= yoffset / 10
25     else:
26         state['size'] += yoffset / 10
```

2.11 Руководство администратора

2.11.1 Процедура инсталляции и деинсталляции

1. Инсталляция:

- Установите Python и необходимые библиотеки: `pip install numpy glfw PyOpenGL`

- Скачайте исходный код программы и разместите его в удобной директории.

2. Деинсталляция:

- Удалите директорию с исходным кодом программы.
- При необходимости удалите установленные библиотеки: pip uninstall numpy glfw PyOpenGL

2.11.2 Параметры запуска из командной строки

Программа запускается командой: python main.py -s

2.11.3 Требования к аппаратному и системному программному обеспечению

- Операционная система: Windows, macOS, Linux.
- Графический процессор с поддержкой OpenGL 3.3.
- Python 3.x.

2.12 Руководство пользователя

2.12.1 Описание графического интерфейса

Программа предоставляет окно с рендерингом 3D-объектов. Пользователь может взаимодействовать с объектами с помощью клавиатуры и мыши.

2.12.2 Перечень сообщений об ошибках

- **Ошибка инициализации GLFW:** Программа не может инициализировать GLFW.
- **Ошибка создания окна:** Программа не может создать окно.
- **Ошибка нехватки вершин:** $lat_segments * (lon_segments + 1)$ должно быть больше либо равно 16 для корректного отображения фигур.

3 Экспериментальная часть

3.1 Демонстрация работы программы

Разработанная программа успешно реализует морфинг между кубом и сферой с использованием трёхмерных объектов, описанных треугольниками (GL_TRIANGLE_STRIP). На рисунке 1 показаны промежуточные этапы морфинга, включая начальную форму (куб), конечную форму (сфера) и несколько состояний между ними.

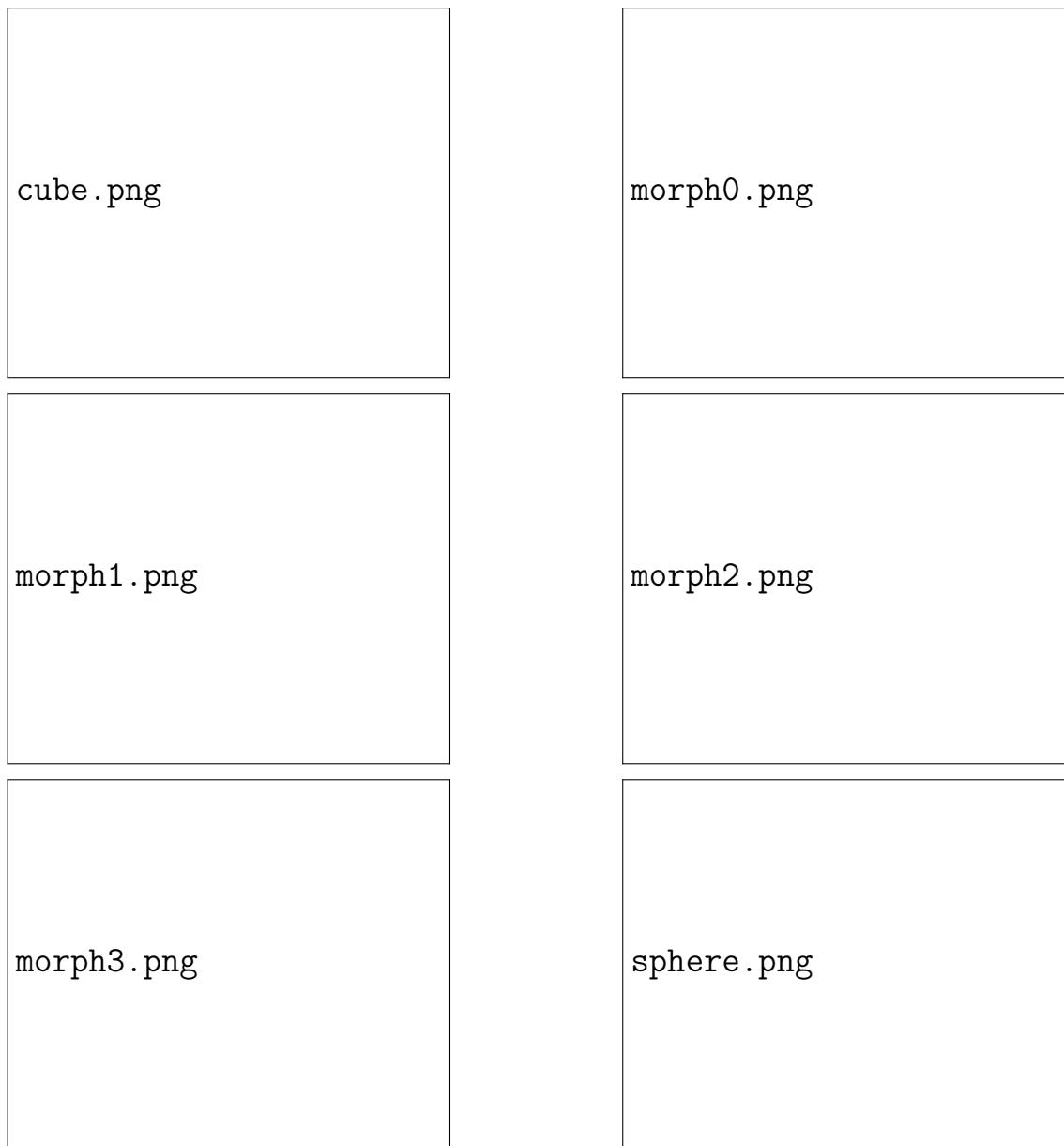


Рисунок 1 — Демонстрация работы программы: промежуточные этапы морфинга.

Пользователь может взаимодействовать с программой в реальном времени, управляя следующими параметрами:

- включение и отключение морфинга (нажатие клавиши M);
- вращение объекта с помощью движения мыши;
- изменение масштаба прокруткой колеса мыши;
- переключение между отображением каркаса и заполненной модели (нажатие клавиши F);
- управление скоростью морфинга клавишами + и -.

3.2 Визуальная оценка качества

Для оценки визуального качества морфинга были проведены тесты с различным количеством сегментов сферы. Результаты показали, что при увеличении числа сегментов улучшается плавность переходов, однако возрастают требования к производительности при создании вершин куба и сферы. Оптимальным было выбрано значение 200 полос широты и 249 сегментов долготы, что обеспечивает баланс между качеством изображения и скоростью работы программы.

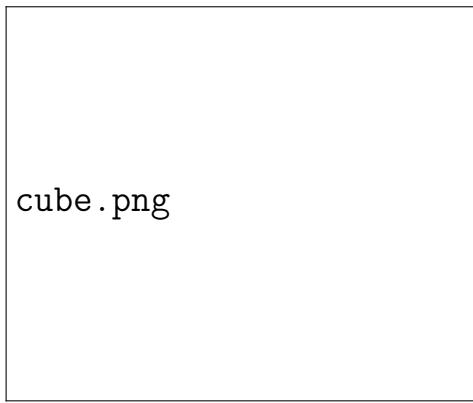
На рисунке 2 приведено сравнение объектов с различным уровнем детализации.

3.3 Оценка производительности

Производительность программы была протестирована на нескольких конфигурациях компьютеров, включая системы с различными характеристиками графических процессоров. Важными метриками производительности стали:

- частота кадров (FPS);
- задержка при изменении формы;
- загрузка GPU при разных уровнях детализации объектов.

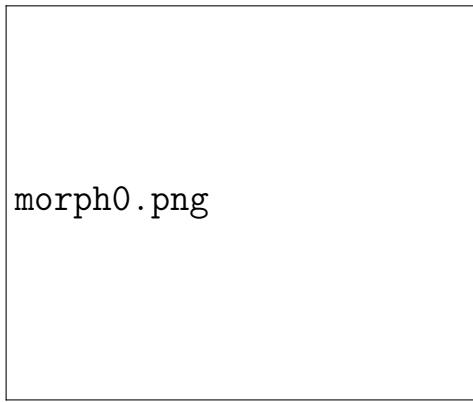
На рисунке 3 показана зависимость частоты кадров от количества вершин в моделях. Видно, что даже при значительном увеличении числа вершин (до 100,000) программа демонстрирует стабильную производительность благодаря оптимизированной обработке на GPU.



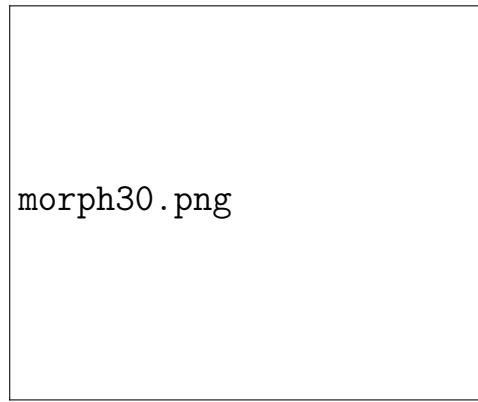
cube.png



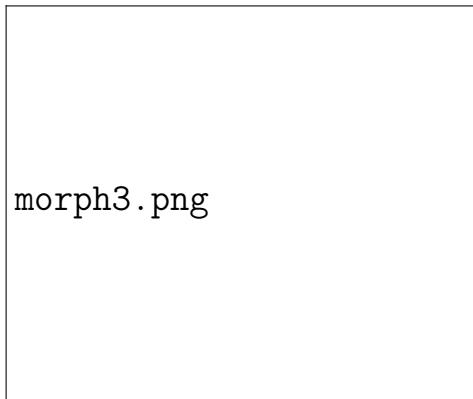
cube30.png



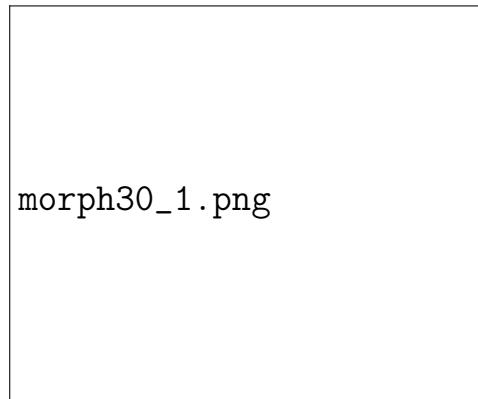
morph0.png



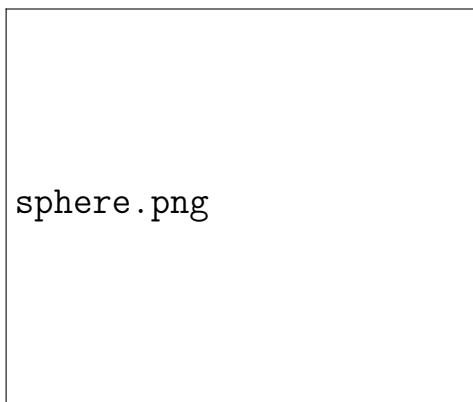
morph30.png



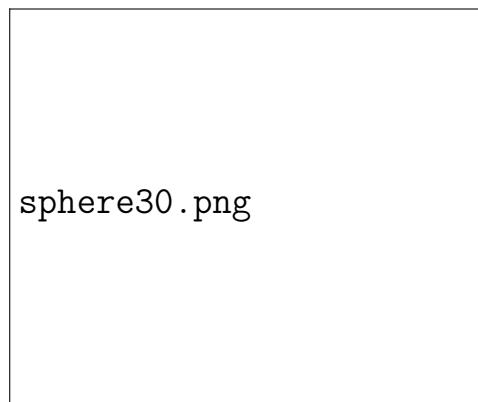
morph3.png



morph30_1.png



sphere.png

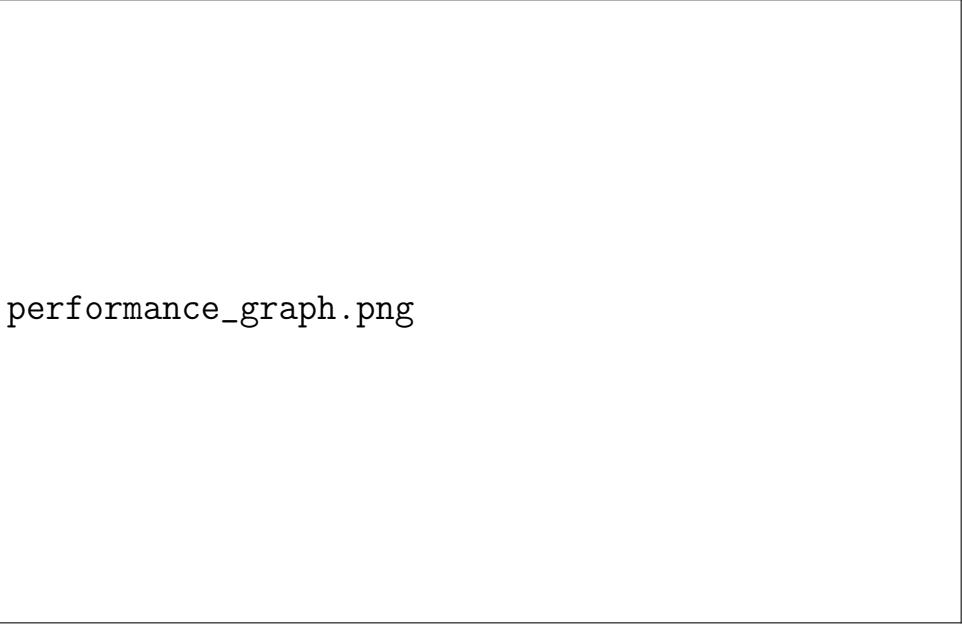


sphere30.png

100000 вершин

1800 вершин

Рисунок 2 — Сравнение уровня детализации объектов при разных параметрах.



performance_graph.png

Рисунок 3 — График зависимости частоты кадров от числа вершин.

3.4 Анализ результатов

Результаты экспериментов показали, что разработанная программа успешно реализует морфинг между кубом и сферой, обеспечивая плавные и реалистичные переходы. Визуальная оценка качества морфинга показала, что при увеличении числа сегментов сферы улучшается плавность переходов, однако возрастают требования к производительности. Оптимальное значение в 200 полос широты и 249 сегментов долготы было выбрано также для более быстрого запуска программы.

Производительность программы была протестирована на нескольких конфигурациях компьютеров, и результаты показали, что программа демонстрирует стабильную производительность даже при значительном увеличении числа вершин благодаря оптимизированной обработке на GPU.

3.5 Обсуждение возможных улучшений

Несмотря на успешную реализацию морфинга, существуют возможности для улучшения программы. Одним из таких улучшений может быть использование более сложных математических моделей для интерполяции, таких как бикубическая интерполяция или интерполяция с использованием радиальных базисных

функций. Это позволит добиться еще более плавных и реалистичных переходов между формами объектов.

Другим возможным улучшением может быть использование адаптивных сегментов для динамического изменения разрешения геометрии объектов в зависимости от сложности сцены. Это позволит оптимизировать вычисления и улучшить производительность программы при работе с объектами различной сложности.

Также можно рассмотреть возможность использования параллельных вычислений и многопоточности для ускорения обработки данных. Это позволит улучшить производительность программы и обеспечить более плавную анимацию даже при сложных преобразованиях.

ЗАКЛЮЧЕНИЕ

В ходе данной работы был изучен и реализован алгоритм морфинга трёхмерных объектов на примере перехода от куба к сфере. Проект позволил углубиться в основы работы с 3D-графикой и изучить применение современных инструментов, таких как OpenGL, для создания анимационных эффектов.

Основные результаты работы:

- Изучены теоретические основы морфинга, включая его применение в различных областях компьютерной графики.
- Реализованы трёхмерные объекты (куб и сфера) с использованием треугольников (GL_TRIANGLE_STRIP) для повышения производительности рендеринга.
- Разработан алгоритм морфинга, обеспечивающий плавный переход между объектами, и реализован на уровне GPU с помощью шейдеров.
- Обеспечено интерактивное управление параметрами морфинга, включая изменение скорости, вращение и масштабирование объектов.
- Проведены эксперименты, продемонстрировавшие стабильную производительность программы и высокое визуальное качество при оптимальном количестве сегментов.

Разработанная программа имеет несколько ограничений, которые могут быть устранены в дальнейшем:

1. Добавление поддержки более сложных форм для морфинга, например, моделей, импортированных из внешних файлов.
2. Реализация нелинейного морфинга для более реалистичных преобразований.
3. Использование текстур высокого разрешения для улучшения визуального восприятия.
4. Оптимизация кода для работы на устройствах с ограниченными ресурсами.

Работа над этим проектом позволила не только закрепить теоретические знания в области компьютерной графики, но и получить практические навыки в программировании 3D-анимации. Предложенный подход может быть использован

в дальнейшем для создания более сложных визуальных эффектов и интерактивных приложений.

Таким образом, поставленные цели и задачи были выполнены, что подтверждает успешность проведённого исследования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Foley J. D., Hughes J. F., Dam A. van. Computer Graphics: Principles and Practice. — Boston : Addison-Wesley Professional, 2013.
2. Goulekas K. E. Visual Effects in a Digital World: A Comprehensive Glossary of over 7000 Visual Effects Terms. — San Francisco : Morgan Kaufmann, 2001.
3. Ryan D. History of Computer Graphics: DLR Associates Series. — Scotts Valley, CA : CreateSpace Independent Publishing Platform, 2011.
4. Haque H., Hassaniem A.-E., Nakajima M. Generation of Missing Medical Slices Using Morphing Technology // IEICE Transactions on Information & Systems. — 2000. — Июль. — Т. E83—D, № 8. — С. 1400—1407.
5. Vince J. Mathematics for Computer Graphics. — Fifth Edition. — Breinton, UK : Springer-Verlag London Ltd., 2017.
6. Гонсалес Р. и Вудс Р. Цифровая обработка изображений. — 3-е издание. — Москва : Williams Publishing House, 2012.
7. Mukundan R. Advanced Methods in Computer Graphics: With examples in OpenGL. — London, Dordrecht, Heidelberg, New York : Springer London Ltd., 2012.
8. Bergan R. The Film Book: A Complete Guide to the World of Film. — New York, New York : DK Publishing, 2011.

ПРИЛОЖЕНИЕ А

Листинг 8: Основной код программы

```
1 //MAIN.py
2 import time
3 import argparse
4 import glm
5 import glfw
6 from OpenGL.GL.shaders import compileProgram, compileShader
7 from utils import *
8 from window_manager import init_window, terminate
9 from shape import SPHERE, CUBE
10 from callbacks import key_callback, mouse_button_callback,
11     → cursor_position_callback, scroll_callback
12
13 WIDTH = 1000
14 HEIGHT = 800
15
16 def display(window, shader_program, vao, state, n, t, last_time):
17     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
18     glUseProgram(shader_program)
19     glBindVertexArray(vao)
20
21     if state.get('tex_enable', True):
22         glUniform1i(glGetUniformLocation(shader_program, "tex_enable"),
23             → 1)
24     else:
25         glUniform1i(glGetUniformLocation(shader_program, "tex_enable"),
26             → 0)
27
28     v_x = np.array([1.0, 0.0, 0.0]) # Ось X
29     v_y = np.array([0.0, 1.0, 0.0]) # Ось Y
30
31     rotation_x = rotation_matrix(v_x,
32         → np.radians(state['rotation_angle_x']))
33     rotation_y = rotation_matrix(v_y,
34         → np.radians(state['rotation_angle_y']))
35
36     model = rotation_x @ rotation_y
37     model = model @ np.array([[state.get('size'), 0, 0, 0],
38                             [0, state.get('size'), 0, 0],
39                             [0, 0, state.get('size'), 0],
40                             [0, 0, 0, 1]])
41     glUniformMatrix4fv(glGetUniformLocation(shader_program, "model"), 1,
42         → GL_FALSE, model)
```

```

37     morph_factor_location = glGetUniformLocation(shader_program,
38         "morphFactor")
39     glUniform1f(morph_factor_location, t)
40     glDrawArrays(GL_TRIANGLE_STRIP, 0, n)
41
42     if state.get('fill'): # switching between wireframe and solid-state
43         ↵ model display
44         glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)
45     else:
46         glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
47
48     # Swap the front and back buffers
49     glfw.swap_buffers(window)
50
51     # Poll for and process events
52     glfw.poll_events()
53
54
55 def initialize_window_and_state(args):
56     window = init_window(WIDTH, HEIGHT, "Morphing")
57     state = {
58         'rotation_angle_x': 0.0,
59         'rotation_angle_y': 0.0,
60         'size': 1.2,
61         'mouse_pressed': False,
62         'last_mouse_pos': (0.0, 0.0),
63         'tex_enable': True,
64         'morph': False,
65         'fill': True,
66         'direction': 1.5
67     }
68     return window, state
69
70
71
72 def setup_callbacks(window, state):
73     # Назначаем обработчики событий
74     glfw.set_key_callback(window, lambda w, k, s, a, m: key_callback(w,
75         ↵ k, s, a, m, state))
76     glfw.set_scroll_callback(window, lambda w, x, y: scroll_callback(w,
77         ↵ x, y, state))
78     glfw.set_mouse_button_callback(window, lambda w, b, a, m:
79         ↵ mouse_button_callback(w, b, a, m, state))
80     glfw.set_cursor_pos_callback(window, lambda w, x, y:
81         ↵ cursor_position_callback(w, x, y, state))
82
83
84
85 def load_and_compile_shaders():
86     # Загрузка кода шейдеров
87     vertex_shader_code = load_shader_code("shaders/vertex_shader.vert")
88     fragment_shader_code =
89         ↵ load_shader_code("shaders/fragment_shader.frag")

```

```

80      # Компиляция и связывание шейдеров
81      shader_program = compileProgram(
82          compileShader(vertex_shader_code, GL_VERTEX_SHADER),
83          compileShader(fragment_shader_code, GL_FRAGMENT_SHADER)
84      )
85      glUseProgram(shader_program)
86      return shader_program
87
88
89  def setup_vertex_buffers(sphere, cube, shader_program, lat_segments,
90      ↪ lon_segments):
91      texture_sphere = generate_texture_coords_sphere(lat_segments,
92          ↪ lon_segments)
93      texture_cube = generate_texture_coords_cube(cube.vertices)
94      vao = glGenVertexArrays(1)
95      glBindVertexArray(vao)
96
97      vbo_cube = glGenBuffers(1)
98      glBindBuffer(GL_ARRAY_BUFFER, vbo_cube)
99      glBufferData(GL_ARRAY_BUFFER, cube.vertices.nbytes, cube.vertices,
100          ↪ GL_STATIC_DRAW)
101
102      vbo_sphere = glGenBuffers(1)
103      glBindBuffer(GL_ARRAY_BUFFER, vbo_sphere)
104      glBufferData(GL_ARRAY_BUFFER, sphere.vertices.nbytes,
105          ↪ sphere.vertices, GL_STATIC_DRAW)
106
107      vbo_normals = glGenBuffers(1)
108      glBindBuffer(GL_ARRAY_BUFFER, vbo_normals)
109      glBufferData(GL_ARRAY_BUFFER, cube.normals.nbytes, cube.normals,
110          ↪ GL_STATIC_DRAW)
111
112      vbo_texture_cube = glGenBuffers(1)
113      glBindBuffer(GL_ARRAY_BUFFER, vbo_texture_cube)
114      glBufferData(GL_ARRAY_BUFFER, texture_cube.nbytes, texture_cube,
115          ↪ GL_STATIC_DRAW)
116
117      vbo_texture_sphere = glGenBuffers(1)
118      glBindBuffer(GL_ARRAY_BUFFER, vbo_texture_sphere)
119      glBufferData(GL_ARRAY_BUFFER, texture_sphere.nbytes, texture_sphere,
120          ↪ GL_STATIC_DRAW)
121
122      # Получение ссылки на атрибут позиции вершин в шейдере
123      position_cube = glGetAttribLocation(shader_program, "cubePosition")
124      glBindBuffer(GL_ARRAY_BUFFER, vbo_cube)
125      glVertexAttribPointer(position_cube, 3, GL_FLOAT, GL_FALSE, 0, None)
126      glEnableVertexAttribArray(position_cube)
127
128      normal_cube = glGetAttribLocation(shader_program, "cubeNormal")
129      glBindBuffer(GL_ARRAY_BUFFER, vbo_normals)
130      glVertexAttribPointer(normal_cube, 3, GL_FLOAT, GL_FALSE, 0, None)
131      glEnableVertexAttribArray(normal_cube)

```

```

125     position_sphere = glGetAttribLocation(shader_program,
126         ↪ "spherePosition")
127     glBindBuffer(GL_ARRAY_BUFFER, vbo_sphere)
128     glVertexAttribPointer(position_sphere, 3, GL_FLOAT, GL_FALSE, 0,
129         ↪ None)
130     glEnableVertexAttribArray(position_sphere)

131     texture_cube_coord = glGetAttribLocation(shader_program,
132         ↪ "textureCube")
133     glBindBuffer(GL_ARRAY_BUFFER, vbo_texture_cube)
134     glVertexAttribPointer(texture_cube_coord, 2, GL_FLOAT, GL_FALSE, 0,
135         ↪ None)
136     glEnableVertexAttribArray(texture_cube_coord)

137     texture_sphere_coord = glGetAttribLocation(shader_program,
138         ↪ "textureSphere")
139     glBindBuffer(GL_ARRAY_BUFFER, vbo_texture_sphere)
140     glVertexAttribPointer(texture_sphere_coord, 2, GL_FLOAT, GL_FALSE, 0,
141         ↪ None)
142     glEnableVertexAttribArray(texture_sphere_coord)

143     glBindVertexArray(0)

144
145 def setup_matrices(shader_program):
146     # Установка матрицы проекции
147     projection_location = glGetUniformLocation(shader_program,
148         ↪ "projection")
149     projection = glm.perspective(glm.radians(45.0), WIDTH / HEIGHT, 0.1,
150         ↪ 100.0)
151     glUniformMatrix4fv(projection_location, 1, GL_FALSE,
152         ↪ glm.value_ptr(projection))

153     camera_position = np.array([1, 1, -2]) # Камера отодвинута назад
154     camera_target = np.array([0, 0, 0]) # Центр фигуры
155     up = np.array([0, 1, 0]) # Направление "вверх"
156     view = look_at(camera_position, camera_target, up)
157     view_location = glGetUniformLocation(shader_program, "view")
158     glUniformMatrix4fv(view_location, 1, GL_FALSE, view)

159     # Активируем текстурный блок и привязываем текстуру
160     texture_id = make_texture('textures/texture.bmp')
161     glActiveTexture(GL_TEXTURE0)
162     glBindTexture(GL_TEXTURE_2D, texture_id)
163     glUniform1i(glGetUniformLocation(shader_program, "textureSampler"),
164         ↪ 0) # Текстурный блок 0

165
166 def main_loop(window, shader_program, vao, state, n, fps):
167     last_time = glutGet(GLUT_ELAPSED_TIME)

```

```

167     t = 0.0
168     while not glfw.window_should_close(window):
169         start_time = time.time()
170         t, last_time = morph_factor(t, last_time, state)
171         display(window, shader_program, vao, state, n, t, last_time)
172         if (time.time() - start_time) != 0:
173             fps.append(1 / (time.time() - start_time))
174
175
176     def cleanup(shader_program, vbo_cube, vbo_normals, vbo_sphere):
177         # Освобождение ресурсов
178         glDeleteBuffers(1, [vbo_cube])
179         glDeleteBuffers(1, [vbo_normals])
180         glDeleteBuffers(1, [vbo_sphere])
181         glDeleteProgram(shader_program)
182         terminate()
183
184
185     def main():
186         parser = argparse.ArgumentParser(description='Морфинг')
187         parser.add_argument('--lat_segments', type=int, default=200,
188             ↴ help='Количество сегментов по широте')
189         parser.add_argument('--lon_segments', type=int, default=249,
190             ↴ help='Количество сегментов по долготе')
191         args = parser.parse_args()
192         lat_segments, lon_segments = args.lat_segments, args.lon_segments
193
194         window, state = initialize_window_and_state(args)
195
196         setup_callbacks(window, state)
197
198         shader_program = load_and_compile_shaders()
199
200         sphere = SPHERE(lat_segments, lon_segments)
201         n = len(sphere.vertices)
202         cube = CUBE(n)
203
204         vao, vbo_cube, vbo_normals, vbo_sphere = setup_vertex_buffers(sphere,
205             ↴ cube, shader_program, lat_segments, lon_segments)
206
207         setup_matrices(shader_program)
208
209         setLight(shader_program)
210         fps = []
211         main_loop(window, shader_program, vao, state, n, fps)
212
213
214         print(f"FPS: {sum(fps) / len(fps)}")
215         cleanup(shader_program, vbo_cube, vbo_normals, vbo_sphere)
216
217
218     if __name__ == "__main__":
219         main()

```

```

216 //SHAPE.py
217 class SPHERE(Shape):
218     def __init__(self, lat_segments=5, lon_segments=9, radius=0.5):
219         super().__init__()
220         self.lat_segments = lat_segments
221         self.lon_segments = lon_segments
222         self.radius = radius
223
224         self.generate_sphere()
225
226     def generate_sphere(self):
227         vertices = []
228         for i in range(self.lat_segments):
229             for j in range(self.lon_segments + 1): # Добавляем +1 для
230                 # замыкания по долготе
231                 # Точка текущей полосы
232                 phi1 = np.pi * i / self.lat_segments
233                 theta = 2 * np.pi * j / self.lon_segments
234                 x1 = self.radius * np.sin(phi1) * np.cos(theta)
235                 y1 = self.radius * np.sin(phi1) * np.sin(theta)
236                 z1 = self.radius * np.cos(phi1)
237
238                 # Точка следующей полосы
239                 phi2 = np.pi * (i + 1) / self.lat_segments
240                 x2 = self.radius * np.sin(phi2) * np.cos(theta)
241                 y2 = self.radius * np.sin(phi2) * np.sin(theta)
242                 z2 = self.radius * np.cos(phi2)
243
244                 # Добавляем вершины поочередно
245                 vertices.append((x1, y1, z1))
246                 vertices.append((x2, y2, z2))
247
248         self.vertices = np.asarray(vertices, dtype=np.float32)
249
250
251     class CUBE(Shape):
252         def __init__(self, total_expected_points):
253             # Координаты вершин куба (центрированного в (0,0,0) с длиной
254             # ребра 0.6)
255             super().__init__()
256             self.vertices = np.array([
257                 [-0.3, -0.3, 0.3], # Вершина 0
258                 [0.3, -0.3, 0.3], # Вершина 1
259                 [0.3, 0.3, 0.3], # Вершина 2
260                 [-0.3, 0.3, 0.3], # Вершина 3
261                 [-0.3, -0.3, -0.3], # Вершина 4
262                 [0.3, -0.3, -0.3], # Вершина 5
263                 [0.3, 0.3, -0.3], # Вершина 6
264                 [-0.3, 0.3, -0.3] # Вершина 7
265             ])

```

```

264     # Границы куба (по индексу вершин)
265     self.faces = [
266         [1, 2, 3, 0],    # Передняя
267         [3, 2, 6, 7],    # Верхняя грань
268         [6, 5, 1, 2],    # Правая грань
269         [5, 4, 0, 1],    # Нижняя грань
270         [0, 4, 7, 3],    # Левая грань
271         [7, 4, 5, 6],    # Задняя грань
272     ]
273     self.normals = None
274
275     self._subdivide_face(total_expected_points)
276     self._update_normals()
277
278     def _subdivide_face(self, total_expected_points):
279         """
280             Разбиваем все грани куба с 8 вершинами на равномерные сетку.
281             :param total_expected_points: необходимое количество точек на
282                 всем кубе
283             :return: список всех точек
284         """
285         if total_expected_points < 16:
286             raise "Недостатка вершин"
287
288         divisions = 0
289         count_point_in_face = (2 * (divisions + 1) ** 2 - 2 * (divisions
290             + 1))
291         c_p_i_f_2 = count_point_in_face
292         while count_point_in_face * 6 < total_expected_points:
293             c_p_i_f_2 = count_point_in_face
294             divisions += 1
295             count_point_in_face = (2 * (divisions + 1) ** 2 - 2 *
296             (divisions + 1))
297
298         points_to_add_1 = count_point_in_face
299         while points_to_add_1 * 6 > total_expected_points:
300             points_to_add_1 -= (divisions + 1) * 2
301
302         points_to_add_1 = total_expected_points - points_to_add_1 * 6
303         points_to_add_2 = total_expected_points - c_p_i_f_2 * 6
304
305         if points_to_add_2 < points_to_add_1:
306             count_point_in_face = c_p_i_f_2
307             divisions -= 1
308
309             points = []
310             for k, face in enumerate(self.faces):    # Для каждой грани куба
311                 v1 = self.vertices[face[0]]
312                 v2 = self.vertices[face[1]]
313                 v3 = self.vertices[face[2]]
314                 v4 = self.vertices[face[3]]
315
316                 for i in range(divisions):

```

```

314         for j in range(divisions + 1):
315             p1 = v1 + (v2 - v1) * (j / divisions)
316             p2 = v4 + (v3 - v4) * (j / divisions)
317             point1 = p1 + (p2 - p1) * (i / divisions)
318             point2 = p1 + (p2 - p1) * ((i + 1) / divisions)
319
320             # Добавляем поочерёдно точки двух полос
321             points.append(point1)
322             points.append(point2)
323
324     # Если точек больше, чем нужно
325     if len(points) > total_expected_points:
326         # Избыточное количество точек
327         extra_points = len(points) - total_expected_points
328         face_row_points = (divisions + 1) * 2
329         faces_to_process = 1
330
331     # Определяем, сколько точек нужно удалить для каждой грани
332     while face_row_points * faces_to_process * 6 < extra_points:
333         faces_to_process += 1
334
335     points_to_remove = face_row_points * faces_to_process * 6
336     removed_points = points_to_remove
337
338     last_face_index = 5
339     first_strip = True
340     removed_count = 0
341     start_index = 0
342     end_index = 0
343     remove_alternate = False # Управляет удалением через одну
344     ↪ или подряд. True - через одну. False - подряд
345     for i in range(len(points) - 1, -1, -1):
346         current_face_index = i // count_point_in_face
347
348         # Пропуск следующих точек
349         if remove_alternate:
350             remove_alternate = False
351             continue
352
353         if count_point_in_face - i % count_point_in_face > 1:
354             if first_strip:
355                 first_strip = False
356                 start_index = i
357
358             # Удаляем точки
359             if points_to_remove - removed_points <
360             ↪ face_row_points * faces_to_process:
361                 points.pop(i)
362                 removed_points -= 1
363                 end_index = i
364                 removed_count += 1
365
366             remove_alternate = True

```

```

365             # Управляем состоянием пропуска
366             if (divisions + 1) <= removed_count <= (divisions
367                 + 1) * 2 * faces_to_process - (divisions +
368                 1):
369                 remove_alternate = False
370
371             # Переход к следующей грани
372             if last_face_index != current_face_index or i < 1:
373                 face_row_points += (divisions + 1) * 2
374                 last_face_index = current_face_index
375                 removed_count = 0
376                 first_strip = True
377                 remove_alternate = False # Возвращаемся к удалению
378                     → через одну
379
380             # Шафлим оставшиеся точки для корректного порядка
381             points = self._custom_shuffle(points, end_index - 1,
382                                         start_index -
383                                         → (divisions + 1) * 2
384                                         → * faces_to_process)
385
386             missing_points = total_expected_points - len(points) # Кол-во
387                 → нехватавших точек
388             num_faces = 6
389             face_size = len(points) // num_faces # Массив точек всегда можно
390                 → разделить на равные 6 частей (грани)
391             for face_index in range(num_faces - 1, -1, -1):
392                 end = (face_index + 1) * face_size if face_index < num_faces
393                     → - 1 else len(points)
394                 # Вычисляем, сколько точек добавить в текущую грань
395                 points_to_add = missing_points // num_faces
396                 if face_index < missing_points % num_faces:
397                     points_to_add += 1 # Распределяем остаток равномерно
398
399                 i = end - 3
400                 while points_to_add >= 3: # Добавляем повторением
401                     → треугольников
402                     point = points[i:i + 3]
403                     for j in range(2, -1, -1):
404                         points.insert(i, point[j])
405                     points_to_add -= 3
406                     i -= 3
407                     end += 3
408
409                 while points_to_add != 0: # Добавляем по одной в конец, если
410                     → нужно
411                     point = points[end - 1]
412                     points.insert(end - 1, point)
413                     points_to_add -= 1
414
415             print("Final number of points:", len(points))
416
417             self.vertices = np.array(points, dtype=np.float32)

```

```

408 def _custom_shuffle(self, arr, a, b):
409     # Получаем подмассив элементов с индексами от a до b
410     # (включительно)
411     subarray = arr[a:b + 1]
412
413     # Разделяем подмассив на две половины
414     mid = (len(subarray) + 1) // 2 # Это будет середина массива
415     first_half = subarray[:mid] # Первая половина
416     second_half = subarray[mid:] # Вторая половина
417
418     # Чертежаем элементы из первой и второй половины
419     shuffled_subarray = []
420     for i in range(len(first_half)):
421         shuffled_subarray.append(first_half[i])
422         if i < len(second_half):
423             shuffled_subarray.append(second_half[i])
424
425     # Создаем новый массив с переставленными элементами
426     shuffled_arr = arr[:a] + shuffled_subarray + arr[b + 1:]
427
428     return shuffled_arr
429
430 def _update_normals(self):
431     self.normals = np.zeros_like(self.vertices)
432     n = len(self.vertices)
433     num_faces = 6
434     missing_points = n % num_faces # Сколько точек было добавлено в
435     # грани в subdivided_face
436     points_in_faces = []
437     for face_index in range(num_faces): # Рассчитываем количество
438     # точек в каждой грани
439     points = n // num_faces
440     if face_index < missing_points % num_faces:
441         points += 1
442
443     points += points_in_faces[face_index - 1] if face_index != 0
444     # else 0
445     points_in_faces.append(points)
446
447     no_start_tr = set()
448     for i in points_in_faces: # При поиске нормалей НЕ использовать
449     # эти точки как начало треугольника, так как конец тогда
450     # окажется на другой грани
451     no_start_tr.add(i - 1)
452     no_start_tr.add(i - 2)
453
454     # Проходим по треугольникам
455     for i in range(0, len(self.vertices) - 2, 1):
456         if i in no_start_tr:
457             continue
458
459         i1, i2, i3 = i, i + 1, i + 2

```

```

454     # Получаем вершины треугольника
455     v1 = self.vertices[i1]
456     v2 = self.vertices[i2]
457     v3 = self.vertices[i3]
458
459     # Вычисляем векторы сторон
460     edge1 = v2 - v1
461     edge2 = v3 - v1
462
463     # Нормаль треугольника - векторное произведение
464     normal = np.cross(edge1, edge2)
465
466     if np.linalg.norm(normal) != 0: # Тут разные точки
467         # Присваиваем нормаль каждой вершине треугольника
468         self.normals[i1] += normal
469         self.normals[i2] += normal
470         self.normals[i3] += normal
471
472         self.normals[i1] = self.normals[i1] /
473             np.linalg.norm(self.normals[i1])
474         self.normals[i2] = self.normals[i2] /
475             np.linalg.norm(self.normals[i2])
476         self.normals[i3] = self.normals[i3] /
477             np.linalg.norm(self.normals[i3])
478     else: # Тут несколько одинаковых точек
479         self.normals[i2] = self.normals[i1]
480         self.normals[i3] = self.normals[i1]
481
482     # Нормализуем все нормали
483     self.normals = self.normals / np.linalg.norm(self.normals,
484         axis=1)[:, np.newaxis]
485
486     # Ориентируем нормали наружу (проверяем их ориентацию
487     # относительно внешнего вектора)
488     # Для этого используем точку внутри объекта (например, центр) и
489     # смотрим на угол с нормалью
490     center = np.mean(self.vertices, axis=0) # Центральная точка
491     # объекта
492
493     for i in range(len(self.vertices)):
494         # Направление от центра к вершине
495         direction_to_center = self.vertices[i] - center
496         # Скалярное произведение с нормалью
497         if np.dot(self.normals[i], direction_to_center) < 0:
498             # Если нормаль направлена внутрь (отрицательное скалярное
499             # произведение), инвертируем её
500             self.normals[i] = -self.normals[i]
501
502
503 //WINDOW_MANAGER.py
504 def init_window(width=800, height=600, title="OpenGL Window"):
505     if not glfw.init():
506         raise Exception("GLFW could not be initialized")

```

```

499     window = glfw.create_window(width, height, title, None, None)
500     if not window:
501         glfw.terminate()
502         raise Exception("GLFW window could not be created")
503
504
505     # Установка контекста OpenGL для окна
506     glfw.make_context_current(window)
507
508     glViewport(0, 0, width, height)
509     glEnable(GL_DEPTH_TEST)
510
511     glClearColor(0.2, 0.3, 0.3, 1.0)
512
513     return window
514
515
516 def terminate():
517     glfw.terminate()
518
519
520 //UTILS.py
521 def rotation_matrix(i, f):
522     i = np.asarray(i)
523     assert i.size == 3, "i must be a 3d vector"
524     # Normalize i
525     i /= np.linalg.norm(i)
526
527     c, s = np.cos(f), np.sin(f)
528     a = 1 - c
529
530     # Build the rotation matrix
531     R = np.array([[i[0] ** 2 * a + c, i[0] * i[1] * a - i[2] * s, i[0] *
532                   ↳ i[2] * a + i[1] * s, 0],
533                   [i[0] * i[1] * a + i[2] * s, i[1] ** 2 * a + c, i[1] *
534                   ↳ i[2] * a - i[0] * s, 0],
535                   [i[0] * i[2] * a - i[1] * s, i[1] * i[2] * a + i[0] *
536                   ↳ s, i[2] ** 2 * a + c, 0],
537                   [0, 0, 0, 1]], dtype=np.float32)
538
539     return R
540
541
542 def load_shader_code(file_path):
543     """
544     Загружает код шейдера из файла.
545     :param file_path: Путь к файлу шейдера.
546     :return: Код шейдера в виде строки.
547     """
548
549     with open(file_path, 'r') as file:
550         return file.read()

```

```

547 def look_at(eye, target, up):
548     f = (target - eye)
549     f = f / np.linalg.norm(f)
550
551     u = up / np.linalg.norm(up)
552     s = np.cross(f, u)
553     u = np.cross(s, f)
554
555     m = np.eye(4, dtype=np.float32)
556     m[0, :3] = s
557     m[1, :3] = u
558     m[2, :3] = -f
559     m[:3, 3] = -np.dot(m[:3, :3], eye)
560
561     return m.T
562
563
564 def make_texture(filename):
565     """
566     Загружает текстуру из файла и возвращает идентификатор текстуры
567     ← OpenGL.
568
569     :param filename: Путь к изображению.
570     :return: Идентификатор текстуры OpenGL.
571     """
572
573     # Открываем изображение и преобразуем в массив пиктру
574     img = Image.open(filename)
575     img_data = np.array(list(img.getdata()), dtype=np.uint8)
576
577     # Генерация и привязка текстуры
578     texture_id = glGenTextures(1)
579     glBindTexture(GL_TEXTURE_2D, texture_id)
580
581     # Загрузка данных текстуры
582     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, img.size[0], img.size[1], 0,
583                  GL_RGB, GL_UNSIGNED_BYTE, img_data)
584
585     # Настройка параметров фильтрации и повторения текстуры
586     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
587     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
588     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
589     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
590
591     return texture_id
592
593
594 def generate_texture_coords_sphere(lat_segments=5, lon_segments=9):
595     """
596     Генерация текстурных координат для сферической поверхности,
597     соответствующей порядку вершин из GL_TRIANGLE_STRIP.
598
599     :param lat_segments: Количество полос широты.
600     """

```

```

598 """:param lon_segments: Количество сегментов долготы.
599 :return: Массив текстурных координат ( $N \times 2$ ).
600 """
601
602     tex_coords = []
603     for i in range(lat_segments):
604         for j in range(lon_segments + 1): # +1 для замыкания по долготе
605             u = j / lon_segments
606             v1 = i / lat_segments # Верхняя широта
607             tex_coords.append([u, v1]) # Верхняя точка
608
609             v2 = (i + 1) / lat_segments # Нижняя широта
610             tex_coords.append([u, v2]) # Нижняя точка
611
612     return np.asarray(tex_coords, dtype=np.float32)
613
614
615 def generate_texture_coords_cube(vertices):
616     n = len(vertices)
617     num_faces = 6
618     missing_points = n % num_faces # Сколько точек было добавлено в
619     # грань в subdivide_face
620     points_in_faces = []
621     for face_index in range(num_faces): # Рассчитываем количество точек
622     # в каждой грани
623         points = n // num_faces
624         if face_index < missing_points % num_faces:
625             points += 1
626
627         points += points_in_faces[face_index - 1] if face_index != 0 else
628         0
629         points_in_faces.append(points)
630
631     points_in_faces.append(0)
632     tex_coords = []
633
634     # Генерация текстурных координат для каждой грани
635     for i in range(num_faces):
636         x = abs(vertices[points_in_faces[i] - 1][0] -
637         vertices[points_in_faces[i - 1]][0])
638         y = abs(vertices[points_in_faces[i] - 1][1] -
639         vertices[points_in_faces[i - 1]][1])
640         z = abs(vertices[points_in_faces[i] - 1][2] -
641         vertices[points_in_faces[i - 1]][2])
642         a = 0
643         b = 1
644
645         if x == 0:
646             x = z
647             a = 2
648         if y == 0:
649             y = z
650             b = 2

```

```

645     for j in range(points_in_faces[i - 1], points_in_faces[i]):
646         u = abs(vertices[j][a] - vertices[points_in_faces[i] - 1][a])
647         ↪ / x
648         v = abs(vertices[j][b] - vertices[points_in_faces[i] - 1][b])
649         ↪ / y
650
651     tex_coords.append([u, v])
652
653
654 def setLight(shader_program):
655     light_position = [5, 5, 5, 1]
656
657     light_position_location = glGetUniformLocation(shader_program,
658         "lightPosition")
659     light_color_location = glGetUniformLocation(shader_program,
660         "lightColor")
661     ambient_color_location = glGetUniformLocation(shader_program,
662         "ambientColor")
663     diffuse_color_location = glGetUniformLocation(shader_program,
664         "diffuseColor")
665     specular_color_location = glGetUniformLocation(shader_program,
666         "specularColor")
667     shininess_location = glGetUniformLocation(shader_program,
668         "shininess")
669
670
671
672 def morph_factor(t, last_time, state):
673     EPSILON = 1e-6 # Погрешность для проверки границ
674     PAUSE_TIME = 500 # Время задержки на границах в миллисекундах
675
676     time = glutGet(GLUT_ELAPSED_TIME)
677     if state.get('morph') == True:
678         current_time = time # Текущее время
679         # Проверка: задерживаем t на границах
680         if t >= 1.0 - EPSILON:
681             t = 1.0
682             if current_time - last_time > PAUSE_TIME:
683                 if state.get('direction') > 0:
684                     state['direction'] *= -1 # Меняем направление
685                     ↪ движения
686                     last_time = current_time # Обновляем время последней
687                     ↪ паузы
688                     t += 0.001 * state.get('direction')

```

```

687     elif t <= 0.0 + EPSILON:
688         t = 0.0
689         if current_time - last_time > PAUSE_TIME:
690             if state.get('direction') < 0:
691                 state['direction'] *= -1 # Меняем направление
692                     → движения
693                 last_time = current_time # Обновляем время последней
694                     → паузы
695                 t += 0.001 * state.get('direction')
696             else:
697                 # t движется непрерывно внутри диапазона (0, 1)
698                 t += 0.001 * state.get('direction')
699                 last_time = current_time
700
701
702 //CALLBACKS.py
703 def key_callback(window, key, scancode, action, mods, state):
704     """
705     Обработка нажатий клавиш.
706     :param window: окно GLFW
707     :param key: клавиша
708     :param scancode: аппаратный код клавиши
709     :param action: действие (нажата, отпущена, удерживается)
710     :param mods: модификаторы (Shift, Ctrl)
711     :param state: объект состояния программы
712     """
713
714     if action == glfw.PRESS:
715         if key == glfw.KEY_SPACE:
716             state['rotation_angle_x'] = 0.0
717             state['rotation_angle_y'] = 0.0
718             state['size'] = 1.2
719             state['direction'] = 1.5
720         elif key == glfw.KEY_T:
721             state['tex_enable'] = not state['tex_enable']
722         elif key == glfw.KEY_M:
723             state['morph'] = not state['morph']
724         elif key == glfw.KEY_F:
725             state['fill'] = not state['fill']
726         elif key == glfw.KEY_ESCAPE:
727             glfw.set_window_should_close(window, True)
728         elif key == glfw.KEY_EQUAL:
729             if state['direction'] > 0:
730                 state['direction'] += 0.2
731             else:
732                 state['direction'] -= 0.2
733         elif key == glfw.KEY_MINUS:
734             if state['direction'] > 0:
735                 state['direction'] -= 0.2
736             else:
737                 state['direction'] += 0.2
738         elif key == glfw.KEY_UP:

```

```

738         state['size'] += 0.1
739     elif key == glfw.KEY_DOWN:
740         state['size'] -= 0.1
741
742
743 def mouse_button_callback(window, button, action, mods, state):
744     if button == glfw.MOUSE_BUTTON_LEFT:
745         if action == glfw.PRESS:
746             state['mouse_pressed'] = True
747         elif action == glfw.RELEASE:
748             state['mouse_pressed'] = False
749
750
751 def cursor_position_callback(window, xpos, ypos, state):
752     """
753     Обработка движения курсора.
754     :param window: окно GLFW
755     :param xpos: координата X курсора
756     :param ypos: координата Y курсора
757     :param state: объект состояния программы
758     """
759
760     if state.get('mouse_pressed', False):
761         dx = xpos - state['last_mouse_pos'][0]
762         dy = ypos - state['last_mouse_pos'][1]
763         state['rotation_angle_x'] += dy * 0.2 # Скорость вращения
764         state['rotation_angle_y'] -= dx * 0.2
765     state['last_mouse_pos'] = [xpos, ypos]
766
767
768 def scroll_callback(window, xoffset, yoffset, state):
769     """
770     Обработка прокрутки.
771     :param window: окно GLFW
772     :param xoffset: смещение по горизонтали
773     :param yoffset: смещение по вертикали
774     :param state: объект состояния программы
775     """
776
777     if xoffset > 0:
778         state['size'] -= yoffset / 10
779     else:
780         state['size'] += yoffset / 10

```

Листинг 9: Код шейдеров

```
779 //fragment_shader.frag
780
781 #version 330 core
782
783 in vec3 position;
784 in vec3 normals_sphere;    // flat in vec3 normals;
785 flat in vec3 normals_cube;
786 in vec2 fragTextureCoord; // Координаты текстуры
787
788 out vec4 fragColor;
789
790 uniform float morphFactor;
791 uniform vec3 lightPosition;
792 uniform vec3 lightColor;
793 uniform vec3 ambientColor;
794 uniform vec3 diffuseColor;
795 uniform vec3 specularColor;
796 uniform float shininess;
797 uniform sampler2D textureSampler; // Сэмплер текстуры
798 uniform int tex_enable;
799
800 void main()
801 {
802     vec3 normal = morphFactor == 0.0 ? normalize(normals_cube) :
803         normalize(normals_sphere);
804
805     // Рассчитываем вектор направления света
806     vec3 lightDirection = normalize(lightPosition - position);
807
808     // Рассчитываем фоновое освещение
809     vec3 ambient = ambientColor * lightColor;
810
811     // Рассчитываем диффузное освещение
812     float diffuseFactor = max(dot(normal, lightDirection), 0.0);
813     vec3 diffuse = diffuseColor * lightColor * diffuseFactor;
814
815     // Рассчитываем зеркальное отражение
816     vec3 viewDirection = normalize(-position);
817     vec3 reflectDirection = reflect(-lightDirection, normal);
818     float specularFactor = pow(max(dot(viewDirection, reflectDirection),
819         0.0), shininess);
820     vec3 specular = specularColor * lightColor * specularFactor;
821
822
823     vec4 textureColor = texture(textureSampler, fragTextureCoord); // 
824         // Получение цвета из текстуры
825
826     // Общий цвет с учетом всех эффектов освещения
827     vec3 finalColor = ambient + diffuse + specular;
```

```

825 // Присваиваем цвет фрагменту
826 fragColor = vec4(finalColor, 1.0) * mix(vec4(1.0), textureColor,
827     ↵ tex_enable);
828 //fragColor = vec4(normal * 0.5 + 0.5, 1.0);
829 //fragColor = textureColor;
830 }
831
832 //vertex_shader.frag
833
834 #version 330 core
835 layout(location = 0) in vec3 cubePosition;
836 layout(location = 1) in vec3 cubeNormal;      // Нормали для куба
837 layout(location = 2) in vec3 spherePosition;   // Позиции вершин сферы
838 layout(location = 3) in vec2 textureCube;
839 layout(location = 4) in vec2 textureSphere;
840
841 uniform float morphFactor;
842
843 out vec3 position;
844 out vec3 normals_cube;
845 out vec3 normals_sphere;
846 out vec2 fragTextureCoord;
847
848 uniform mat4 model;
849 uniform mat4 projection;
850 uniform mat4 view;
851
852 void main() {
853     fragTextureCoord = mix(textureCube, textureSphere, morphFactor);
854
855     vec3 morphedPosition = mix(cubePosition, spherePosition,
856         ↵ morphFactor);
857     position = vec3(view * model * vec4(morphedPosition, 1.0));
858
859     vec3 morphedNormal = mix(cubeNormal, normalize(spherePosition),
860         ↵ morphFactor);
861     normals_cube = mat3(transpose(inverse(view * model))) * cubeNormal;
862     normals_sphere = mat3(transpose(inverse(view * model))) *
863         ↵ morphedNormal;
864
865     gl_Position = projection * view * model * vec4(morphedPosition, 1.0);
866 }

```