Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и

информатики»

(СибГУТИ)

02.03.02 Фундаментальная информатика и информационные технологии

Профиль: Системное программное обеспечение

(очная форма обучения)

ОТЧЕТ ПО производственной ПРАКТИКЕ

на кафедре вычислительных систем

(наименование структурного подразделения СибГУТИ)

Изучение растеризации, реализация и описание проекции перспективы, разработка скелетной анимации 2D моделей

Выполнил:

студент института ИВТ

гр. ИС-142 / Куликов Д.А./

«13» июля 2024г. (подпись)

Проверил:

Руководитель от СибГУТИ / Перышкова Е.Н./

«13» июля 2024г. (подпись)

Новосибирск 2024

Оглавление

[1. Задание на практику 4](#_Toc171007980)

[2. Введение 5](#_Toc171007981)

[3. Введение в OpenGL, GLFW и GLEW 6](#_Toc171007982)

[3.1 OpenGL 6](#_Toc171007983)

[3.2 GLFW 8](#_Toc171007984)

[3.3 GLEW 9](#_Toc171007985)

[4. Проекция перспективы 10](#_Toc171007986)

[5. Структура классов 12](#_Toc171007987)

[5.1 Шаблонные математические классы 12](#_Toc171007988)

[5.2 ObjectTransform 12](#_Toc171007989)

[5.3 Шаблонные геометрические объекты 12](#_Toc171007990)

[5.4 Базовые классы рендера 16](#_Toc171007991)

[5.5 Actor, Pawn и Character 18](#_Toc171007992)

[5.6 Скелеты и анимации 19](#_Toc171007993)

[5.7 Actor интерфейс 27](#_Toc171007994)

[5.8 Pawn интерфейс 29](#_Toc171007995)

[5.9 Создание сцены 29](#_Toc171007996)

[5.10 Многопоточность, классы потоков и функции обратного вызова 30](#_Toc171007997)

[5.11 GameManager 33](#_Toc171007998)

[6. Сборка проекта 36](#_Toc171007999)

[7. Функция main 36](#_Toc171008000)

[8. Оптимизация 36](#_Toc171008001)

[9. Тестирование 37](#_Toc171008002)

[10. Применение 38](#_Toc171008003)

[11. Заключение 40](#_Toc171008004)

[Список используемой литературы 41](#_Toc171008005)

[Приложения 42](#_Toc171008006)

**План-график проведения** производственной **практики** Вид практики

Фамилия Имя Отчество студента

института Информатика и вычислительная техника , 3 курса,

гр. ИС-142

Направление: 02.03.02 Фундаментальная информатика и информационные технологии

Профиль: Системное программное обеспечение

Место прохождения практики кафедра вычислительных систем

Объем практики: **360/10** часов/ЗЕ

Вид практики  ***производственная***

Тип практики ***Технологическая (проектно-технологическая) практика***

Срок практики с «29» января 2024 г. по «27» мая 2024 г.;

с «17» июня 2024 г. по «13» июля 2024 г.

Содержание практики\*:

|  |  |
| --- | --- |
| Наименование видов деятельности | Дата (начало – окончание) |
| 1. Общее ознакомление со структурным подразделением предприятия, вводный инструктаж по технике безопасности | 29.01.2024 - 09.02.2024 |
| 2. Выдача задания на практику, деление студентов на группы (если необходимо), определение конкретной индивидуальной темы, формирование плана работ | 09.02.2024 - 16.02.2024 |
| 3. Работа с библиотечными фондами структурного подразделения или предприятия, сбор и анализ материалов по теме практики | 16.02.2024 – 15.03.2024 |
| 4. Выполнение работ в соответствии с составленным планом:  – Изучение процесса растеризации и применение его при рендеринге  – Разработка базовых геометрических фигур  – Создание классов необходимых для создания модели персонажи (Actor, Bone и другие)  – Разработка скелетной анимации и класса Motion  – Добавление многопоточности и оптимизация рендера  – Тестирование | 15.03.2024 – 27.03.2024  27.03.2024 -26.04.2024  26.04.2024 -28.05.2024  28.05.2024 -29.06.2024  29.06.2024 -3.07.2024  3.07.2024 -4.07.2024 |
| 5. Анализ полученных результатов и произведенной работы  Составление отчета по практике, защита отчета | 11.07.2024 – 13.07.2024 |

\*В соответствии с программой практики

Руководитель от СибГУТИ / Перышкова Е.Н./

«29» \_\_\_\_01\_\_\_\_ 2024г. (подпись)

# 1. Задание на практику

Основными целями практики являются:

* Знакомство с технологиями компьютерной графики
* Изучить процесс растеризации с использование API OpenGL
* Развитие навыков проектирования архитектуры проекта
* Реализация проекции перспективы при помощи матриц преобразования
* Разработка скелетной анимации для 2D моделей

Для выполнения работы нужно блоки кода на самостоятельные модули, тщательно продумать иерархию классов. Модульность проекта и принципы ООП значительно упростят разработку и поддержку программы.

# 2. Введение

Сначала дадим определения основным терминам, используемым в работе:

* **Рендеринг** – процесс генерации изображения из модели с помощью компьютерной программы.
* **Растеризация** – процесс преобразования геометрических данных (векторов) в пиксели для отображения на экране.
* **Матрица преобразования** – математическая конструкция, используемая для изменения координат объектов в пространстве (перемещение, вращение, масштабирование).
* **Проекция перспективы** – метод отображения трёхмерных объектов на двухмерную плоскость с учётом перспективы, что делает удалённые объекты меньше по сравнению с ближними.
* **Скелетная анимация** – метод анимации, при котором используется набор костей (rig), управляемых иерархией для деформации модели.
* **Кость (rig)** – часть скелетной анимации, представляющая собой абстрактную ось или линию, вдоль которой осуществляется деформация модели.

В ходе работы будет описана реализация проекции перспективы и скелетной анимации, основанные на матрице преобразования. Работа реализована на языке C++. В процессе разработки подкрепятся полученные во время учёбы знания и опыт.

# 3. Введение в OpenGL, GLFW и GLEW

## 3.1 OpenGL

OpenGL (Open Graphics Library) — это спецификация, определяющая стандарт кроссплатформенного интерфейса для написания приложений, использующих 2D и 3D графику. Она предоставляет широкий набор функций для создания графических приложений, от игр до научных симуляций. В основе OpenGL лежат примитивы, такие как точки, линии и полигоны, из которых строятся более сложные графические объекты.

Команды OpenGL претерпевают одинаковый порядок обработки, проходя через последовательность стадий, называемых конвейером обработки OpenGL.

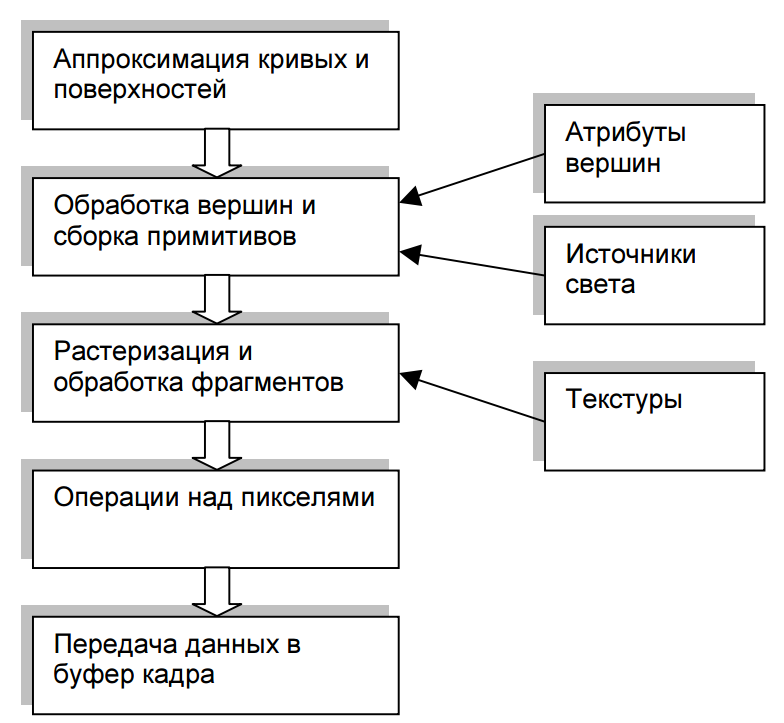


Рис. 1 Конвейер обработки OpenGL

**Аппроксимация кривых и поверхностей**

При отображении векторные геометрические примитивы должны быть представлены в виде (иногда достаточно большого) набора вершин. Однако многие полезные и часто используемые кривые и поверхности могут быть описаны гораздо меньшим количеством параметров. На этапе аппроксимации кривых и поверхностей по простому описанию геометрического примитива формируется набор вершин. При этом можно определять количество вершин и тем самым управлять качеством и производительностью визуализации.

Результатом прохождения этого этапа является сцена, представленная только в виде простейших геометрических примитивов - вершин.

**Обработка вершин и сборка примитивов**

На этом этапе над каждой вершиной выполняются операции необходимые для ее последующего отображения: выполняются преобразования системы координат, проецирование координат на двумерную поверхность экрана, определение цвета вершины с учётом свойств материала и освещение, наложение текстур и т.п. Также в зависимости от режима осуществляется сборка примитивов в линии и многоугольники. На этом шаге выполняется отсечение невидимых линий и вершин. Они могут просто не попадать в порт и глубину вывода, а могут закрываться другими примитивами.

Результатом этого этапа является полное описание геометрических примитивов и их свойств, размещённых в пространстве в соответствии с установленной системой координат, и готовое к этапу растеризации.

**Растеризация и обработка фрагментов (дискретизация)**

В процессе растеризации из векторных и растровых данных создаются фрагменты. Каждый фрагмент соответствует пикселу буфера кадра. На этом этапе формируются заполненные цветом многоугольники, пунктирные линии заданной ширины, точки заданного размера, применяется сглаживание (antialiasing).

**Операции над пикселями**

Перед тем как полученные значения цвета каждого пиксела будут действительно размещены в буфере кадра каждый фрагмент может быть дополнительно обработан. Именно на этом этапе происходит наложение текстур из памяти. Затем выполняются операции вырезания, определения прозрачности, наложения шаблонов и масок, логические операции и наложение изображений, моделирование тумана. Только после этого обработанные фрагменты помещаются в активный буфер кадра, пикселам которого присваиваются значения окончательно отображаемого цвета.

**Передача данных в буфер кадра**

Буфер кадра - это область памяти, которая представляет собой пиксельный массив, отображаемый на экране. Библиотека OpenGL может работать сразу с несколькими буферами кадров (пока один буфер отображается другие уже формируются). Это позволяет добиться эффективного формирования и визуализации анимационных кадров. Пиксельное изображение из буфера кадра может быть считано в системную память, обработано как обычный массив, упаковано, сохранено и снова использовано.

Команды OpenGL всегда обрабатываются в том порядке, в котором они поступают, хотя могут происходить задержки перед тем, как проявится эффект от их выполнения.

Как правило, разные типы примитивов имеют различную скорость визуализации на разных платформах. Для увеличения производительности предпочтительнее использовать примитивы, требующие меньшее количество информации для передачи на сервер, такие как GL\_TRIANGLE\_STRIP, GL\_QUAD\_STRIP, GL\_TRIAGLE\_FAN.

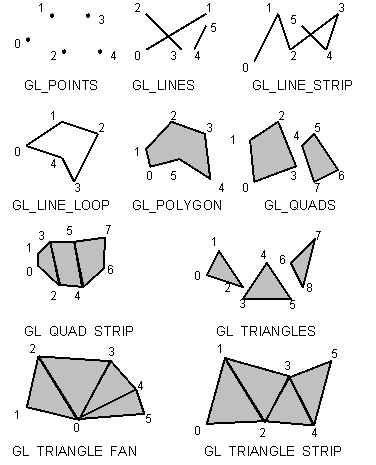


Рис. 2 Примитивы OpenGL

## 3.2 GLFW

Библиотека Open Source для кроссплатформенного создания и открытия окон, создания OpenGL контекста и управления вводом.

Интерфейс GLFW:

* glfwInit - инициализирует библиотеку GLFW. Прежде чем можно будет использовать большинство функций GLFW, необходимо инициализировать GLFW, а перед завершением работы приложения следует завершить GLFW, чтобы освободить все ресурсы, выделенные во время или после инициализации. Если эта функция завершается ошибкой, она вызывает glfwTerminate перед возвратом. Если это удастся, вам следует вызвать glfwTerminate перед завершением работы приложения.
* glfwCreateWindow - эта функция создает окно и связанный с ним контекст OpenGL или OpenGL ES. Большинство параметров, управляющих тем, как должно создаваться окно и его контекст, задаются с помощью подсказок для окна. В аргументах указывается размер окна и его название.
* glfwMakeContextCurrent - делает контекст указанного окна текущим для вызывающего потока. Эта функция делает контекст OpenGL или OpenGL ES указанного окна текущим для вызывающего потока. Контекст должен быть текущим только в одном потоке одновременно, и каждый поток может одновременно иметь только один текущий контекст.
* glfwSetKeyCallback - устанавливает обратный вызов клавиш. Эта функция устанавливает обратный вызов клавиш в указанном окне, который вызывается при нажатии, повторении или отпускании клавиши. Функции клавиш работают с физическими клавишами, а маркеры клавиш, не зависящие от раскладки, называются в соответствии с их значениями в стандартной раскладке клавиатуры США.
* glfwSetWindowUserPointer - устанавливает пользовательский указатель на указанное окно. Эта функция устанавливает пользовательский указатель на указанное окно.
* glfwSetInputMode - устанавливает параметр режима ввода для указанного окна.
* glfwSetWindowShouldClose - устанавливает флаг закрытия указанного окна.

## 3.3 GLEW

Кроссплатформенная библиотека на C/C++, которая упрощает запрос и загрузку расширений OpenGL. GLEW обеспечивает эффективные run-time механизмы для определения того, какие OpenGL расширения поддерживаются на целевой платформе. Все расширения OpenGL размещаются в одном заголовочном файле, который автоматически генерируется из официального списка расширений.

Для работы с OpenGL используется кроссплатформенная библиотека GLEW. Интерфейс GLEW:

* glewInit – инициализация. Функции glEnable и glDisable позволяют включить или отключить возможности OpenGL. В программе включён тест глубины, чтобы объекты на заднем плане не перекрывали объекты, находящиеся ближе к камере.
* glClearColor - определяет четкие значения для цветовых буферов

Выше упомянутые функции нужны для инициализации библиотек и устанавливание им параметров. Остальные функции будут объяснены позже.

# 4. Проекция перспективы

Проекция перспективы принимает 10 параметров:

1. размер объекта
2. поворот объекта
3. положение объекта
4. положение камеры
5. Target – точка, на которую направлена камера.
6. Up – вектор, указывающий направление "вверх" для камеры. Этот вектор необходим для определения ориентации камеры в пространстве и предотвращения ее переворачивания.
7. FOV (Field of View) – угол обзора по вертикали в градусах. Определяет, насколько "широким" будет угол обзора камеры. Большие значения приводят к эффекту "рыбьего глаза", а меньшие - к эффекту "телескопа".
8. Ширина и высота области просмотра в пикселях. Используется для определения соотношения сторон экрана.
9. zNear – расстояние до ближней плоскости отсечения. Объекты ближе этого расстояния не будут отображаться.
10. zFar – расстояние до дальней плоскости отсечения. Объекты дальше этого расстояния не будут отображаться.

Все эти данные приводятся к матрицам размера 4 \* 4 и перемножаются.

Матриц InitScaleTransform используется для сохранения размера:

|  |
| --- |
| **void** InitScaleTransform(T ScaleX, T ScaleY, T ScaleZ) {  m[0][0] = ScaleX; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0f;  m[1][0] = 0.0f; m[1][1] = ScaleY; m[1][2] = 0.0f; m[1][3] = 0.0f;  m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = ScaleZ; m[2][3] = 0.0f;  m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;  } |

Матрица InitRotateTransform хранит поворот:

|  |
| --- |
| **void** InitRotateTransform(T RotateX, T RotateY, T RotateZ) {  Matrix4f rx, ry, rz;  **const** T x = ToRadian(RotateX);  **const** T y = ToRadian(RotateY);  **const** T z = ToRadian(RotateZ);  rx.m[0][0] = 1.0f; rx.m[0][1] = 0.0f ; rx.m[0][2] = 0.0f ; rx.m[0][3] = 0.0f;  rx.m[1][0] = 0.0f; rx.m[1][1] = cosf(x); rx.m[1][2] = -sinf(x); rx.m[1][3] = 0.0f;  rx.m[2][0] = 0.0f; rx.m[2][1] = sinf(x); rx.m[2][2] = cosf(x) ; rx.m[2][3] = 0.0f;  rx.m[3][0] = 0.0f; rx.m[3][1] = 0.0f ; rx.m[3][2] = 0.0f ; rx.m[3][3] = 1.0f;  ry.m[0][0] = cosf(y); ry.m[0][1] = 0.0f; ry.m[0][2] = -sinf(y); ry.m[0][3] = 0.0f;  ry.m[1][0] = 0.0f ; ry.m[1][1] = 1.0f; ry.m[1][2] = 0.0f ; ry.m[1][3] = 0.0f;  ry.m[2][0] = sinf(y); ry.m[2][1] = 0.0f; ry.m[2][2] = cosf(y) ; ry.m[2][3] = 0.0f;  ry.m[3][0] = 0.0f ; ry.m[3][1] = 0.0f; ry.m[3][2] = 0.0f ; ry.m[3][3] = 1.0f;  rz.m[0][0] = cosf(z); rz.m[0][1] = -sinf(z); rz.m[0][2] = 0.0f; rz.m[0][3] = 0.0f;  rz.m[1][0] = sinf(z); rz.m[1][1] = cosf(z) ; rz.m[1][2] = 0.0f; rz.m[1][3] = 0.0f;  rz.m[2][0] = 0.0f ; rz.m[2][1] = 0.0f ; rz.m[2][2] = 1.0f; rz.m[2][3] = 0.0f;  rz.m[3][0] = 0.0f ; rz.m[3][1] = 0.0f ; rz.m[3][2] = 0.0f; rz.m[3][3] = 1.0f;  \***this** = rz \* ry \* rx;  } |

InitTranslationTransform – положение в пространстве:

|  |
| --- |
| **void** InitTranslationTransform(T x, T y, T z) {  m[0][0] = 1.0f; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = x;  m[1][0] = 0.0f; m[1][1] = 1.0f; m[1][2] = 0.0f; m[1][3] = y;  m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = 1.0f; m[2][3] = z;  m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;  } |

InitCameraTransform – настройки камеры: Target, Up, FOV, ширина и высота области просмотра, zNear и zFar:

|  |
| --- |
| **void** InitCameraTransform(**const** Vector3<T>& Target, **const** Vector3<T>& Up) {  Vector3<T> N = Target;  N.Normalize();  Vector3<T> U = Up;  U.Normalize();  U = U.Cross(N);  Vector3<T> V = N.Cross(U);    m[0][0] = U.x; m[0][1] = U.y; m[0][2] = U.z; m[0][3] = 0.0f;  m[1][0] = V.x; m[1][1] = V.y; m[1][2] = V.z; m[1][3] = 0.0f;  m[2][0] = N.x; m[2][1] = N.y; m[2][2] = N.z; m[2][3] = 0.0f;  m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;  }  **void** InitPersProjTransform(T FOV, T Width, T Height, T zNear, T zFar) {  **const** T ar = Width / Height;  **const** T zRange = zNear - zFar;  **const** T tanHalfFOV = tanf(ToRadian(FOV / 2.0f));  m[0][0] = 1.0f/(tanHalfFOV \* ar); m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0;  m[1][0] = 0.0f; m[1][1] = 1.0f/tanHalfFOV; m[1][2] = 0.0f; m[1][3] = 0.0;  m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = (-zNear -zFar)/zRange ; m[2][3] = 2.0f \* zFar\*zNear/zRange;  m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 1.0f; m[3][3] = 0.0;  } |

# 5. Структура классов

## 5.1 Шаблонные математические классы

Matrix4f – матрица 4x4, произвольного типа. Интерфейс класса выполняет операции над матрицей преобразования для достижения проекции перспективы с указанными параметрами.

|  |
| --- |
| **template**<**typename** T>  **class** Matrix4f  {  **public**:  T m[4][4];  ...  } |

Vector3 – вектор из трёх элементов произвольного типа. Имеет базовый интерфейс необходимый для вектора:

* Арифметика с Vector3 и скалярами
* Cross - возвращает вектор, который перпендикулярен к обоим исходным векторам
* VDot - возвращает скалярное значение, которое является мерой проекции одного вектора на другой
* Normalize - возвращает вектор той же направленности, но с длиной, равной единице

|  |
| --- |
| **template**<**typename** T>  **class** Vector3  {  **public**:  T x;  T y;  T z;  ...  } |

5.2 ObjectTransform  
ObjectTransform – используется для управления положением объектов в 3D пространстве.

В интерфейс класса входят методы для изменения значения полей класса, получения направления по указанной оси, смещение по осям и движение в указанном направлении.

|  |
| --- |
| **class** objectTransform  {  **public**:  Vector3<GLfloat> WorldPos;  Vector3<GLfloat> Rotate;  Vector3<GLfloat> Scale;  ...  }; |

## 5.3 Шаблонные геометрические объекты

Информация о геометрии и буферах фигуры хранятся в структуре GeometryInfo. Для экономии памяти каждый класс имеет статический экземпляр этой структуры. Экземпляры фигур получают доступ к их геометрии при помощи метода GetGeometry. Перед использованием фигур их нужно инициализировать, вызвав статическую функцию initializeGeometry.

Переменные типа GLuint используются для хранения идентификаторов объектов OpenGL, таких как буферы, текстуры и шейдеры. Эти идентификаторы позволяют ссылаться на соответствующие объекты для выполнения различных операций с ними.

Переменные, имеющие название g\*Location, связанны с соответствующими переменными в шейдере. Принцип их работы будет описан позже.

|  |
| --- |
| **struct** GeometryInfo {  GLuint VAO = 0;  GLuint VBO = 0;  GLuint EBO = 0;  GLint numVertices = 0;  GLint numIndices = 0;  }; |

Основным классом является Sprite (квадрат), который наследуют другие фигуры.

Конструктор Sprite компилирует шейдеры и загружает текстуры. Если указанные шейдеры и текстуры уже были созданы, то класс использует их.

|  |
| --- |
| **class** Sprite  {  **public**:  Sprite(**const** std::string &\_name, **const** **char** \*FS, **const** **char** \*VS, **const** **char** \*texturePath);  **static** **void** initializeGeometry();  **virtual** **struct** GeometryInfo \*GetGeometry();  std::string name;  Vector3<GLfloat> Scale;  Vector3<GLfloat> color = Vector3<GLfloat>(1.0, 1.0, 1.0);  GLuint texture = 0;  GLuint shader = 0;    GLuint gWorldLocation;  GLuint gColorLocation;  GLuint gTextureSamplerLocation;  **protected**:  GLuint loadShader(**const** **char** \*shaderPath, GLuint type);  **void** loadTextures(**const** **char** \*texturePath);  **void** compileShaders(**const** **char** \*FS, **const** **char** \*VS);  **inline** **static** **struct** GeometryInfo geometryInfo = {0, 0, 0, 0, 0};  **private**:  **inline** **static** std::map<std::string, std::array<GLuint, 4>> shadersMap;  **inline** **static** std::map<std::string, GLuint> texturesMap;  }; |

Общие экземпляры шейдеров, текстур и буферов сокращает число обращений к OpenGL, потому что придётся реже менять шейдеры, текстуры и буферы. Стоит отметить, что сокращение обращений к OpenGL – один из самых лучших методов оптимизации рендеринга.

|  |
| --- |
| Sprite::Sprite(**const** std::string &\_name, **const** **char** \*FS, **const** **char** \*VS, **const** **char** \*texturePath)  : name(\_name)  {  **auto** sdr = shadersMap.find(std::string(FS) + std::string(VS));  **if** (sdr != shadersMap.end()) {  shader = sdr->second[0];  gWorldLocation = sdr->second[1];  gColorLocation = sdr->second[2];  gTextureSamplerLocation = sdr->second[3];  } **else** {  compileShaders(FS, VS);  shadersMap[std::string(FS) + std::string(VS)] = std::array<GLuint, 4>{shader, gWorldLocation, gColorLocation, gTextureSamplerLocation};  }  **auto** txr = texturesMap.find(std::string(texturePath));  **if** (txr != texturesMap.end()) {  texture = txr->second;  } **else** {  loadTextures(texturePath);  texturesMap[std::string(texturePath)] = texture;  }  } |

Шейдеры представляют из себя отдельную программу и создаются при помощи функции glCreateProgram. К этой программе прикрепляются фрагментный и вершинный шейдеры функцией glAttachShader.

Этапы работы метода compileShaders:

1. Создает объект шейдерной программы, если он еще не создан.
2. Загружает и компилирует фрагментный и вершинный шейдеры.
3. Прикрепляет шейдеры к программе.
4. Линкует шейдерную программу.
5. Проверяет результат линковки и выводит лог ошибок при необходимости.
6. Получает расположения униформов gWorld и gColor.

Для выполнения 6-го этапа метода выполняется следующий код:

|  |
| --- |
| gWorldLocation = glGetUniformLocation(shader, "gWorld");  gColorLocation = glGetUniformLocation(shader, "gColor");  gTextureSamplerLocation = glGetUniformLocation(shader, "textureSampler"); |

g\*Location переменные связываются с соответствующими переменными в шейдере.

Этапы работы метода loadTextures:

1. Загружает изображение с помощью stbi\_load.
2. Изменяет размер изображения для использования в OpenGL. Для оптимизации текстура должна иметь разрешение кратное степени двойки.
3. Генерирует и связывает текстуру.
4. Загружает данные изображения в текстурный объект OpenGL.
5. Настраивает параметры текстуры (повторение и фильтрация).
6. Получает расположение семплера текстуры из шейдера.
7. Освобождает память, занятую исходным и измененным изображением.
8. Генерирует мип-карты.
9. Настраивает масштаб текстуры.
10. Проверяет ошибки OpenGL.

|  |
| --- |
| **void** Sprite::loadTextures(**const** **char** \*texturePath)  {  ...  std::string path = std::string("assets/") + texturePath;  **unsigned** **char** \*img = stbi\_load(path.c\_str(), &x, &y, &n, 0);  **int** new\_x = 1 << (**int**)std::ceil(std::log2(x));  **int** new\_y = 1 << (**int**)std::ceil(std::log2(y));  **unsigned** **char** \*resized\_img = (**unsigned** **char**\*)malloc(new\_x \* new\_y \* n);  **if** (resized\_img == **nullptr**) {  std::cerr << "Failed to allocate memory for resized texture" << std::endl;  stbi\_image\_free(img);  **return**;  }  stbir\_resize\_uint8(img, x, y, 0, resized\_img, new\_x, new\_y, 0, n);  glGenTextures(1, &texture);  glBindTexture(GL\_TEXTURE\_2D, texture);  ...  stbi\_image\_free(img);  free(resized\_img);  glGenerateMipmap(GL\_TEXTURE\_2D);  glBindTexture(GL\_TEXTURE\_2D, 0);  ...  } |

Для загрузки текстур и исправления их разрешения используются библиотеки stb\_image и stb\_image\_resize.

Инициализация геометрии осуществляется методом initializeGeometry. Сначала создаются точки, в которых находятся вершины. Затем вершины объединяются в треугольники. Важно, вершины должны быть соединены так, чтобы при рендере как можно чаще использовались вершины, которые использовались предыдущем этапе рендеринга.

|  |
| --- |
| **void** Sprite::initializeGeometry()  {  std::vector<GLfloat> vertices = {  -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,  -1.0f, 1.0f, 0.0f, 0.0f, 1.0f,  1.0f, 1.0f, 0.0f, 1.0f, 1.0f,  1.0f, -1.0f, 0.0f, 1.0f, 0.0f  };  std::vector<GLuint> indices = {  0, 1, 2,  0, 2, 3  };  geometryInfo.numVertices = vertices.size();  geometryInfo.numIndices = indices.size();  glGenVertexArrays(1, &geometryInfo.VAO);  glBindVertexArray(geometryInfo.VAO);  glGenBuffers(1, &geometryInfo.VBO);  glBindBuffer(GL\_ARRAY\_BUFFER, geometryInfo.VBO);  glBufferData(GL\_ARRAY\_BUFFER, vertices.size() \* **sizeof**(GLfloat), vertices.data(), GL\_STATIC\_DRAW);  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 5 \* **sizeof**(GLfloat), (GLvoid\*)0);  glEnableVertexAttribArray(0);  glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 5 \* **sizeof**(GLfloat), (GLvoid\*)(3 \* **sizeof**(GLfloat)));  glEnableVertexAttribArray(1);  glGenBuffers(1, &geometryInfo.EBO);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, geometryInfo.EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* **sizeof**(GLuint), indices.data(), GL\_STATIC\_DRAW);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glBindVertexArray(0);  } |

Сфера реализована как шаблонный класс, для которого указывается уровень его детализации. Сферы разной детализации имеют разную геометрию.

|  |
| --- |
| **template**<std::size\_t N>  **class** sphere : **public** Sprite  {  ...  } |

Фигуры состоящие не из граней, а из линий имеют дополнительные поля для определения ширины рёбер.

|  |
| --- |
| **class** line : **public** Sprite  {  **public**:  line(**const** std::string &\_name, **const** objectTransform &\_trans, **const** Vector3<GLfloat> \_color);  line(**const** std::string &\_name, **const** Vector3<GLfloat> \_color);  ...  GLfloat lineWidth;  objectTransform trans;  ...  }; |

## 5.4 Базовые классы рендера

Camera – класс камеры хранит важные параметры камеры и перспективной проекции для вычисления матрицы преобразования.

Интерфейс класса:

* Установление параметров класса
* Получение положения камеры с учётом положения владельца камеры

|  |
| --- |
| **class** Camera  {  **public**:  **struct** {  Vector3<GLfloat> WorldPos;  Vector3<GLfloat> Target;  Vector3<GLfloat> Up;  } Params;  **struct** {  GLfloat FOV;  GLfloat Width;  GLfloat Height;  GLfloat zNear;  GLfloat zFar;  } PersProj;  …  **const** objectTransform\* OwnerTransformPtr = **nullptr**;  } |

Pipeline – класс отвечающий за конвейер, который вычисляет матрицу преобразования, для реализации проекции перспективы для указанного объекта с учётом параметров камеры.

|  |
| --- |
| **class** Pipeline  {  **public**:  Pipeline() {}  Matrix4f<GLfloat> GetTransform(**const** objectTransform& transform) **const**;  Camera \*camera;  }; |

Render – класс отвечающий за рендер, работу с буферами OpenGL и взаимодействие с шейдерами.

|  |
| --- |
| **class** Render  {  **public**:  Render(Pipeline &\_pipeline);  Render(Camera \*\_camera);    **void** SetCamera(Camera \*\_camera);  **void** clearRender();  **void** PushGeometry(**struct** GeometryInfo \*geometry);  **void** drawObject(Matrix4f<GLfloat>& matrix, Sprite \*sprite);  Pipeline pipeline;  GLuint shader = 0;  GLuint texture = 0;  }; |

Интерфейс класса:

* clearRender – отвязывает все буферы от OpenGL.
* PushGeometry – загружает геометрию объектов. Геометрия обновляется редко, поэтому этот метод отделён от метода drawObject
* drawObject – рендерит объекты, обновляет шейдеры и текстуры, если они отличаются от тех, которые уже загружены.

|  |
| --- |
| **void** Render::clearRender() {  glUseProgram(0);  glBindTexture(GL\_TEXTURE\_2D, 0);  glBindVertexArray(0);  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);  }    **void** Render::PushGeometry(**struct** GeometryInfo \*geometry)  {  glBindVertexArray(geometry->VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, geometry->VBO);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, geometry->EBO);  }  **void** Render::drawObject(Matrix4f<GLfloat>& matrix, Sprite \*sprite)  {  ...  **if** (shader != sprite->shader) {  glUseProgram(sprite->shader);  shader = sprite->shader;  }  **if** (texture != sprite->texture) {  glActiveTexture(GL\_TEXTURE0);  glBindTexture(GL\_TEXTURE\_2D, sprite->texture);  texture = sprite->texture;  glUniform1i(sprite->gTextureSamplerLocation, 0);  }  **if** (sprite->gColorLocation != 0xFFFFFFFF) glUniform3f(sprite->gColorLocation, sprite->color.x, sprite->color.y, sprite->color.z);  glUniformMatrix4fv(sprite->gWorldLocation, 1, GL\_TRUE, &matrix);  **if** (sprite->GetGeometry()->EBO != 0) {  glDrawElements(GL\_TRIANGLES, sprite->GetGeometry()->numIndices, GL\_UNSIGNED\_INT, 0);  } **else** {  glDrawArrays(GL\_LINE\_STRIP, 0, sprite->GetGeometry()->numVertices);  }  } |

Данные передаются в шейдер при помощи функций glUniformMatrix4fv, glUniform3f и переменных g\*Location, которые обеспечивают связь с шейдером.

## 5.5 Actor, Pawn и Character

Эти классы используется для создания объектов на сцене. Pawn наследует Actor, а Character – Pawn.

* Actor – базовый класс, определяет базовый интерфейс объекта. Подходит для объектов декора.
* Pawn – класс с более расширенными возможностями, реализованы методы для перемещения.
* Character – класс для разграничения обычных объектов и управляемых юнитов. Ничем не отличается от Pawn, но может пригодиться в более сложны проектах, например, в сетевых играх.

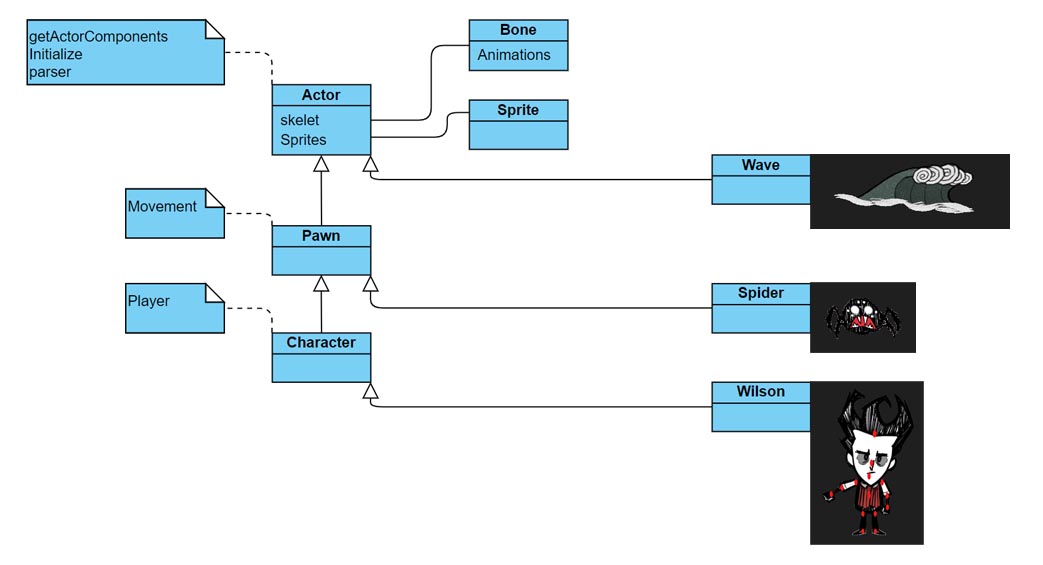


Рис. 3 Диаграмма классов

Основная функция класса Actor в создании из отдельных компонентов целостную модель объекта. Модели состоят только из спрайтов.

|  |
| --- |
| **class** Actor  {  **public**:  Actor(**const** std::string &path, **const** size\_t SkeletSize);  ...  **protected**:  **int** state = STATE::STAND;  **inline** **static** **float** time = 0.0;  Motion::FunType \*motionFunPtr = **nullptr**;  Motion \*motionPtr = **nullptr**;  Vector3<GLfloat> direction = 0;  objectTransform trans;  AnimationInfo animationInfo;  **inline** **static** size\_t skeletSize = 0;  **inline** **static** Bone skelet;  **inline** **static** std::string name = "NoName";  **inline** **static** std::map<std::string, Sprite> Sprites;  **inline** **static** Motion motion;  }; |

## 5.6 Скелеты и анимации

Скелет и анимации создаются в xml файле. Пример записи скелета персонажа:

|  |
| --- |
| <skelet name="Wilson skelet">  <bone name="body">  <bone name="head">  <bone name="face" />  <bone name="hair\_front" />  </bone>  <bone name="shoulder\_left">  <bone name="arm\_left">  <bone name="hand\_left" />  </bone>  </bone>  <bone name="shoulder\_right">  <bone name="arm\_right">  <bone name="hand\_right" />  </bone>  </bone>  <bone name="leg\_left">  <bone name="foot\_left" />  </bone>  <bone name="leg\_right">  <bone name="foot\_right" />  </bone>  </bone>  </skelet> |

Структура скелета хранится в файле: «название объекта»\models\skelet\ skelet.xml

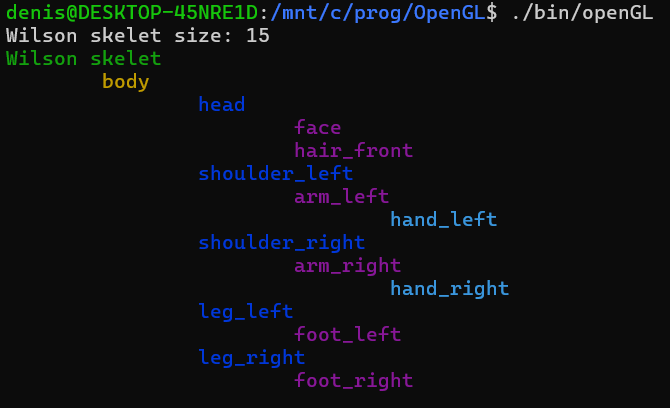


Рис. 4 Структура скелета персонажа

Скелет хранится в классе Bone. Файлы xml считываются при помощи pigixml.

|  |
| --- |
| **class** Bone  {  **public**:  Bone(**const** std::string &\_name, Bone \*\_parent) : name(\_name), parent(\_parent) {};  Bone(**const** std::string &\_name) : name(\_name) {};  Bone() {};  ~Bone();  **void** printBones(**int** lvl);  size\_t addChildBone(pugi::xml\_node &node, std::string \_name, Bone& \_parent);  size\_t parseBone(pugi::xml\_node &node, Bone& \_parent);  size\_t createSkelet(**const** std::string &\_path, **const** std::string &\_name);  std::string name = "None";  Bone \*parent = **nullptr**;  std::vector<Bone> children;  std::map<std::string, Animation> Animations;  }; |

|  |
| --- |
| size\_t Bone::addChildBone(pugi::xml\_node &node, std::string \_name, Bone &\_parent)  {  \_parent.children.emplace\_back(Bone(\_name, &\_parent));  **return** \_parent.children[\_parent.children.size() - 1].parseBone(node, \_parent.children[\_parent.children.size() - 1]);  };  size\_t Bone::parseBone(pugi::xml\_node &node, Bone& \_parent)  {  size\_t size = 0;  **for** (pugi::xml\_node boneNode = node.child(Bone::BONE); boneNode; boneNode = boneNode.next\_sibling(Bone::BONE)) {  std::string boneName = boneNode.attribute(Bone::NAME).as\_string();  size += \_parent.addChildBone(boneNode, boneName, \_parent);  size++;  }  **return** size;  }  size\_t Bone::createSkelet(**const** std::string &\_path, **const** std::string &\_name)  {  std::string full\_path = "assets/entities/" + \_path + "/models/skelet/" + \_name + ".xml";  pugi::xml\_document doc;  pugi::xml\_node node;  pugi::xml\_parse\_result parse\_result = doc.load\_file(full\_path.c\_str());  **if** (!parse\_result) {  std::cout << "Error: file not found (" << full\_path << ")" << std::endl;  **return** 0;  }  node = doc.child("skelet");  name = node.attribute(Bone::NAME).as\_string();  **return** parseBone(node, \***this**) + 1;  } |

Пример записи начального положения компонентов в анимации:

|  |
| --- |
| <animation name="stand" x="0.0" y="-1.5" z="-6.0" tangent="0.0" radius="0.0" flip="0.0" rotate="0.0" width="1.0" height="1.0">  <body sprite="body" x="0.0" y="0.0" z="0.0" tangent="0.0" radius="0.0" flip="0.0" rotate="0.0" width="1.0" height="1.0">  <head sprite="head" x="0.0" y="1.0" z="0.05" tangent="0.0" radius="-1.0" flip="0.0" rotate="0.0" width="1.5" height="1.5">  <face sprite="face" x="-0.03" y="-0.1" z="0.01" tangent="0.0" radius="0.0" flip="0.0" rotate="0.0" width="0.7" height="0.7"/>  <hair\_front sprite="hair\_front" x="0.0" y="0.9" z="-0.01" tangent="0.0" radius="0.0" flip="0.0" rotate="0.0" width="1.45" height="1.45"/>  </head>    <shoulder\_left sprite="shoulder" x="0.8" y="0.7" z="0.02" tangent="-0.05" radius="0.45" flip="25" rotate="0" width="0.35" height="0.35">  <arm\_left sprite="arm" x="0.0" y="-0.8" z="-0.025" tangent="-0.05" radius="1.0" flip="-20.0" rotate="0" width="0.4" height="0.4">  <hand\_left sprite="hand" x="-0.0" y="-1.0" z="0.015" tangent="-0.0" radius="1.5" flip="-30.0" rotate="0" width="2.5" height="2.5" />  </arm\_left>  </shoulder\_left>  <shoulder\_right sprite="shoulder" x="-0.8" y="0.7" z="0.02" tangent="-0.05" radius="0.45" flip="-25.0" rotate="0" width="0.35" height="0.35" >  <arm\_right sprite="arm" x="-0.0" y="-0.8" z="-0.025" tangent="0.05" radius="1.0" flip="20.0" rotate="0" width="0.4" height="0.4" >  <hand\_right sprite="hand" x="0.0" y="-1.0" z="0.015" tangent="-0.0" radius="1.5" flip="30.0" rotate="0" width="2.5" height="2.5" mirrorX="true" />  </arm\_right>  </shoulder\_right>    <leg\_left sprite="leg" x="0.4" y="-1.0" z="-0.02" tangent="0.0" radius="0.5" flip="0.0" rotate="0" width="0.25" height="0.25">  <foot\_left sprite="foot" x="0.1" y="-1.0" z="-0.01" tangent="-1.0" radius="0.0" flip="0.0" rotate="0" width="2.0" height="2.0" />  </leg\_left>  <leg\_right sprite="leg" x="-0.4" y="-1.0" z="-0.02" tangent="0.0" radius="0.5" flip="0.0" rotate="0" width="0.25" height="0.25" mirrorX="true">  <foot\_right sprite="foot" x="-0.1" y="-1.0" z="-0.01" tangent="1.0" radius="0.0" flip="0.0" rotate="0" width="2.0" height="2.0" />  </leg\_right>  </body>  </animation> |

В файле: «название объекта»\models \animations\ stand.xml хранится начальное положение всех компонентов модели при проигрывании анимации «stand».

Параметры кости:

* x/y/z – смещение якоря (anchorPoint) по оси
* tangent – смещение по оси X спрайта от якоря
* radius – смещение по оси Y спрайта от якоря
* flip – поворот относительно якоря
* rotate – наклон спрайта
* width – ширина
* height – высота
* mirrorX – отражение спрайта по оси X

Рассмотрим влияние этих параметров на примере:

Рис. 5 а) указана только ширина, высота и глубина кости. б) добавлено смещение по x и y. в) изменено значение tangent и radius (якорь/красная точка сместилась с центра спрайта). г) добавлено значение flip.

Эти параметры находятся в поле класса Bone std::map<std::string, Animation> Animations, а точнее в экземплярах класса Animation. Для всех анимаций в каждой кости создаётся экземплярах класса Animation. Также этот класс хранит длительность анимаций.

|  |
| --- |
| **class** Animation  {  **public**:  Animation(Sprite \*\_sprite, **const** Vector3<GLfloat> &\_spriteScale, **const** objectTransform &\_transform)  : sprite(\_sprite), spriteScale(\_spriteScale), transform(\_transform) {};  Animation() {};  **static** **void** PushDuration(**const** std::string &actorName, **const** std::string &animationName, **const** **float** \_duration)  {  duration[actorName][animationName] = \_duration;  }  **static** **float** GetDuration(**const** std::string &actorName, **const** std::string &animationName)  {  **auto** actor = duration.find(actorName);  **if** (actor != duration.end()) {  **auto** it = actor->second.find(animationName);  **if** (it != actor->second.end()) {  **return** it->second;  }  }    **return** 0.0;  }  Sprite \*sprite = **nullptr**;  Vector3<GLfloat> spriteScale;  Vector3<GLfloat> anchorPoint;  objectTransform transform;  **static** **inline** std::map<std::string, std::map<std::string, **float**>> duration;  }; |

Текущее состояние компонентов и анимации хранится в классе AnimationInfo:

|  |
| --- |
| **class** AnimationInfo  {  **public**:  AnimationInfo(std::size\_t SkeletSize)  : globalFlip(SkeletSize), transforms(SkeletSize), components(SkeletSize), animations(SkeletSize)  {}  std::vector<GLfloat> globalFlip;  std::vector<Motion::bone\_attribute> transforms;  std::vector<Component> components;  std::vector<Animation> animations;  **float** AnimationTimeStart = 0.0;  std::string animation;  }; |

Класс Component – удобная структура для хранения спрайта и его положения в пространстве.

|  |
| --- |
| **class** Component  {  **public**:  Component(objectTransform &\_transform, Sprite \*\_sprite) : transform(\_transform), sprite(\_sprite) {};  Component() : Component(objectTransform(), **nullptr**) {}  objectTransform transform;  Sprite \*sprite = **nullptr**;  }; |

Трансформация компонентов модели при проигрывании анимации хранится в классе Motion в виде лямбда функций.

|  |
| --- |
| **class** Motion  {  **public**:  Motion(std::vector<std::pair<std::string, **float**>> \*\_float, std::vector<std::pair<std::string, **int**>> \*\_int)  {  **if** (\_float != **nullptr**) {  **for** (**auto** &it : \*\_float)  uniform\_float.insert(it);  }  **if** (\_int != **nullptr**) {  **for** (**auto** &it : \*\_int)  uniform\_int.insert(it);  }  };  Motion(std::vector<std::pair<std::string, **float**>> \*\_float) : Motion(\_float, **nullptr**) {};  Motion(std::vector<std::pair<std::string, **int**>> \*\_int) : Motion(**nullptr**, \_int) {};  Motion() {};  ~Motion() {};  **typedef** std::function<**void**()> FunType;  **struct** bone\_attribute  {  **float** offset[2];  **float** flip;  **float** scale[2];  };  ...  bone\_attribute \*transformations = **nullptr**;  Bone \*skelet = **nullptr**;  std::unordered\_map<std::string, std::pair<**float**, FunType>> function;  std::unordered\_map<std::string, **float**> uniform\_float = { { "time", 0.0 }, { "duration", 0.0 }};  std::unordered\_map<std::string, **int**> uniform\_int;  }; |

Принцип работы анимаций реализован как в шейдерах. Анимации имеют uniform float/int переменные, которые используются в анимациях. Uniform переменные создаются при инициализации класса, который является наследником Actor. Uniform переменные «time» и «duration» создаются по умолчанию.

Пример создания анимации:

|  |
| --- |
| **void** Unit::SetMotion()  {  **auto**\* vec\_float = **new** std::vector<std::pair<std::string, **float**>>;  **auto**\* vec\_int = **new** std::vector<std::pair<std::string, **int**>>;  vec\_float->push\_back(std::pair<std::string, **float**>("flip", 0.0));  vec\_int->push\_back(std::pair<std::string, **int**>("HP", max\_hp));  motion = Motion(vec\_float, vec\_int);  motion.PushSkelet(&Unit::skelet);  Motion::FunType kill = [&]() {};  std::pair<**float**, Motion::FunType> \_kill = {1.0, kill};  motion.PushMotion("kill", \_kill);  Motion::FunType stand = [&]() {  **static** **int** size = skeletSize;  Motion::bone\_attribute \*T = motion.transformations;  std::fill(**reinterpret\_cast**<**float**\*>(&T[0]), **reinterpret\_cast**<**float**\*>(&T[size]), **static\_cast**<**float**>(0.0));    **float** flip = \*motion.FindUniformFloat("flip");  **int** hp = \*motion.FindUniformInt("HP");  **static** size\_t body = motion.FindBone("body");  **static** size\_t HP = motion.FindBone("HP");  **static** size\_t scale = motion.FindBone("scale");  T[body].flip = flip;  **if** (hp >= max\_hp) {  T[scale].scale[0] = -1.0f;  } **else** {  T[HP].scale[0] = **static\_cast**<GLfloat>(hp) / **static\_cast**<GLfloat>(max\_hp) - 1;  T[HP].offset[0] = 1 - **static\_cast**<GLfloat>(hp) / **static\_cast**<GLfloat>(max\_hp);  }  };  std::pair<**float**, Motion::FunType> \_stand = {1.0, stand};  motion.PushMotion("stand", \_stand);  } |

Метод SetMotion класса Unit является статическим и вызывается при инициализации класса Unit. В это примере создаются uniform переменные «flip» и «HP» и две анимации «kill» и «stand». В анимации «kill» персонаж ничего не делает. А анимацию «stand» рассмотрим подробнее.

В начале анимации «stand» обнуляются все трансформации. Далее лямбда функция получает значения uniform переменных при помощи методов FindUniformFloat и FindUniformInt. Значение «flip» используется для поворота персонажа вслед за мышью, а «HP» для отображения очков здоровья. Далее для упрощения написания анимации, индексы нужных костей находятся при помощи метода FindBone. Далее описывается анимация. Анимации записываются в вектор при помощи PushMotion с указанием их длительности и названия.

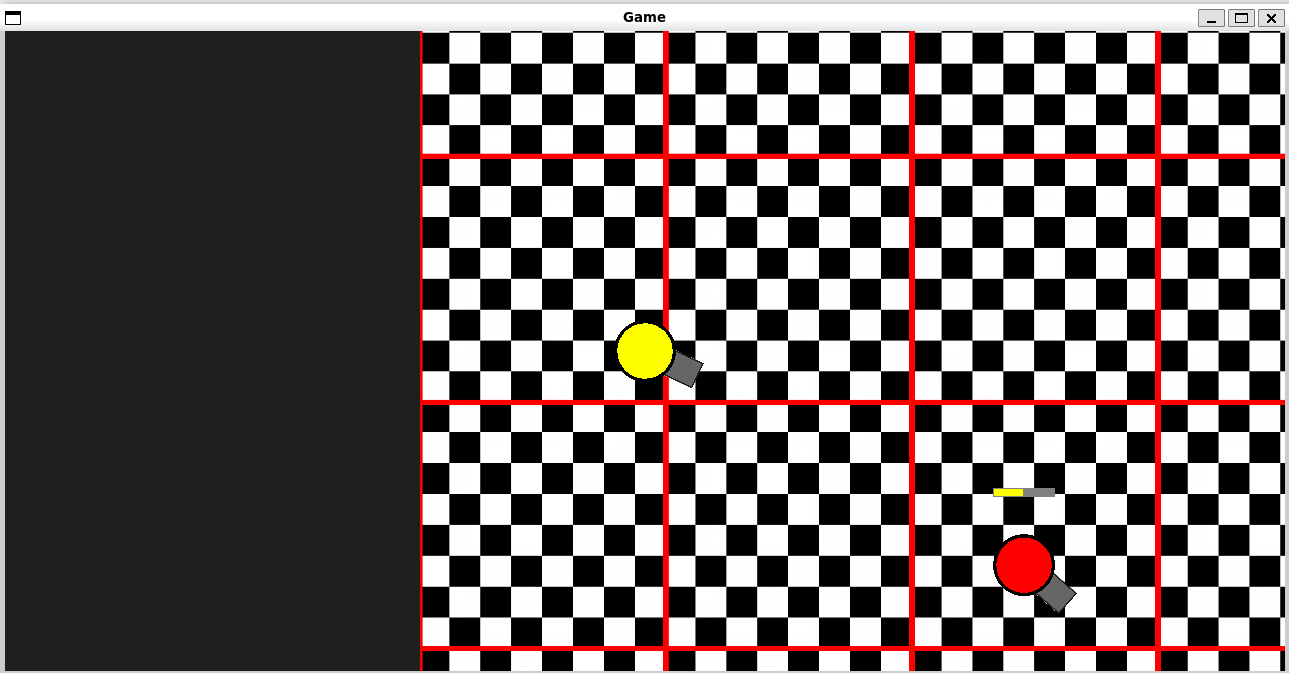


Рис. 6 Демонстрация анимации класса Unit

## 5.7 Actor интерфейс

При помощи функции loadActor можно установить начальное положение объекта при помощи xml файла:

|  |
| --- |
| **bool** Actor::loadActor(**const** std::string &path)  {  std::string full\_path = std::string("assets/entities/") + path + "/actor.xml";  pugi::xml\_document doc;  pugi::xml\_node word;  pugi::xml\_parse\_result parse\_result = doc.load\_file(full\_path.c\_str());  **if** (!parse\_result) {  std::cout << "Error " << GetName() << ".loadActor: file not found (" << full\_path << ")" << std::endl;  **return** **false**;  }  Vector3<GLfloat> v;  v.x = std::stof(doc.child("character").child("objectTransform").child("worldPos").attribute("x").value());  v.y = std::stof(doc.child("character").child("objectTransform").child("worldPos").attribute("y").value());  v.z = std::stof(doc.child("character").child("objectTransform").child("worldPos").attribute("z").value());  trans.SetWorldPos(v.x, v.y, v.z);  v.x = std::stof(doc.child("character").child("objectTransform").child("rotate").attribute("deg").value());  trans.SetRotate(0.0, 0.0, v.x);  v.x = std::stof(doc.child("character").child("objectTransform").child("scale").attribute("x").value());  v.y = std::stof(doc.child("character").child("objectTransform").child("scale").attribute("y").value());  trans.SetScale(v.x, v.y, 0.0);  **return** **true**;  } |

Статический метод Initialize загружает скелет модели, спрайты и анимации:

|  |
| --- |
| **template**<**typename** Derived>  **static** **void** Initialize(std::string path, std::string \_name, std::vector<std::string> \_animations)  {  Derived::name = \_name;  loadSkelet<Derived>(path);  loadSprites<Derived>(path);  Derived::SetMotion();  **for** (**const** **auto** &it : Derived::motion.function)  loadAnimation<Derived>(path, it.first);  } |

Метод updateAnimation заменяет анимацию на другую. Метод проходит по всему скелету и меняет анимацию у всех костей:

|  |
| --- |
| **void** Actor::updateAnimationRecursive(Bone &\_parent, **const** std::string &animationName, size\_t &n)  {  n++;  **for** (**auto**& it : \_parent.children) {  **if** (it.Animations.find(animationName) != it.Animations.end()) {  animationInfo.animations[n] = it.Animations[animationName];  animationInfo.components[n].sprite = animationInfo.animations[n].sprite;  } **else** {  std::cout << "Error: " << \_\_FUNCTION\_\_ << " " << animationName << " sprite not found" << std::endl;  }  updateAnimationRecursive(it, animationName, n);  }  }  **void** Actor::updateAnimation(**const** std::string &animationName)  {  **if** (animationInfo.animation == animationName) **return**;  **if** (GetSkelet()->Animations.find(animationName) != GetSkelet()->Animations.end()) {  std::pair<**float**, Motion::FunType> \*motionPairPtr = &motionPtr->function[animationName];  motionPtr->PushDuration(motionPairPtr->first);  motionFunPtr = &motionPairPtr->second;  animationInfo.animation = animationName;  animationInfo.animations[0] = GetSkelet()->Animations[animationName];  }  size\_t n = 0;  updateAnimationRecursive(\*GetSkelet(), animationName, n);  } |

Вычисления анимации происходят в методе getActorComponents.

|  |
| --- |
| std::vector<Component\*> Actor::getActorComponents()  {  **if** (animationInfo.components.size() == 0) {  std::vector<Component\*> ActorComponents;  **return** ActorComponents;  }    size\_t n = 0;  animationInfo.animations[0].transform = animationInfo.components[0].transform = trans; // Updating the skelet position  GLfloat duration = Animation::GetDuration(\*GetName(), &animationInfo.animation[0]);  motionPtr->PushTime( std::fmod((GetTime() - animationInfo.AnimationTimeStart) / 1e9, \*motionPtr->FindUniformFloat("duration")) );  motionPtr->PushTransform(&animationInfo.transforms[0]);  (\*motionFunPtr)();  **return** getActorComponents(\*GetSkelet(), n, duration);  } |

В Motion передаётся время, которое нужно использовать при вычислении анимации. После вычислений анимации нужно рекурсивно обновить положение костей. При этом кости должны учитывать трансформацию родительских костей. При перемещении/повороте должны переместиться/повернуться и дочерние кости.

## 5.8 Pawn интерфейс

В класс Pawn добавлены методы для перемещения объекта.

|  |
| --- |
| **class** Pawn : **public** Actor  {  **public**:  Pawn(**const** std::string &path, **const** size\_t SkeletSize);  **void** Teleport(**const** Vector3<GLfloat> newPosition);  **void** Move(**const** Vector3<GLfloat> offset);  **void** Move(**const** GLfloat distance, **const** Vector3<GLfloat> direction);  **void** MoveForward();  **void** MoveForward(**const** GLfloat distance);    **template**<**typename** T>  **void** MoveTowards(T\* targetPawn, GLfloat speed) {  Vector3<GLfloat> direction = targetPawn->GetTransform()->WorldPos - **this**->trans.WorldPos;  Move(speed, direction);  }  **void** Rotate(**const** Vector3<GLfloat> \_rotate);  **void** AddRotate(**const** Vector3<GLfloat> \_rotate);  **void** SetScale(**const** Vector3<GLfloat> \_scale);  **void** MultiplyScale(**const** Vector3<GLfloat> \_scale);  GLfloat GetSpeed();  **protected**:  GLfloat speed = 0.3;  }; |

## 5.9 Создание сцены

Класс сцены для хранения объектов:

|  |
| --- |
| **class** Scene  {  **public**:  Scene() {};  ~Scene() {  **for** (**auto**& it : actors) **delete** it;  };  **void** pushActor(Actor\* actor) { actors.push\_back(actor); };  std::vector<Actor\*> actors;  }; |

Перед созданием экземпяров классов Wilson, spider, wave и grass нужно вызвать статический метод для их инциализации. После создания экземпляров нужно указать какую анимацию им нужно проигрывать. В завершении классу GameManager передаётся персонаж, которым может управлять пользователь, также этот персонаж становится владельцем камеры.

|  |
| --- |
| Scene \*createScene()  {  Wilson \*character = **nullptr**;  Spider \*spider[SPIDER\_NUM] = {**nullptr**};  Wave \*wave[WAVE\_SUM] = {**nullptr**};  Grass \*grass = **nullptr**;  **auto** \*scene = **new** Scene();  Wilson::Initialize();  Spider::Initialize();  Wave::Initialize();  Grass::Initialize();  character = **new** Wilson();  character->updateAnimation("stand");  scene->pushActor(character);  ...  GameManager::PushPlayer(character);  GameManager::render->pipeline.camera->OwnerTransformPtr = character->GetTransform();  **return** scene;  } |

## 5.10 Многопоточность, классы потоков и функции обратного вызова

Вычисления для рендера кадра происходят в callback функции (функции обратного вызова). Когда вычисления и рендеринг кадра заканчивается, callback функция вызывается заново, если не было команды об завершении программы.

Для сложных сцен в однопоточном режиме callback функции приходилось бы выполнять очень много вычислений. Поэтому лучше разделить вычисления на несколько потоков, у которых будут callback функции.

Программа запускает 3 основных потока:

1. Обрабатывает событий на сцене и передаёт второму потоку объекты, для которых нужно вычислить анимацию и проекцию перспективы
2. Вычисляет анимацию и проекцию перспективы, вычисленные значения передаёт третьему потоку для рендера.
3. Управляет классом рендера и рендерит объекты

Для такой многопоточной реализации понадобится много дополнительных структур данных, мьютексов и атомарных переменных. Для группировки и модульности многопоточного режима были созданы 3 класса.

В потоках запускается функция job, в которой до завершения работы будет вызваться callback. Коммуникация потоков использует топологию "однонаправленное кольцо".



Рис. 7 Визуализация коммуникации между потоками

Важно отметить, OpenGL может работать только в одном потоке, который его инициализировал. Поэтому метод job класса RenderThread нужно вызвать именно родительским потоком.

|  |
| --- |
| **class** SceneThread {  **public**:  SceneThread();  **void** start();  **void** setScene(**const** Scene\* \_scene);  **void** job();  **private**:  **void** callback();  **const** Scene\* scene = **nullptr**;  ComponentsThread componentsThread;  std::atomic<**bool**> endTick;  std::mutex mutex;  }; |

SceneThread реализован, как основной класс, в котором хранятся другие классы. Для запуска потоков реализован метод start. Метод запускает в новых потоках методы job классов SceneThread и ComponentsThread, и затем в основном потоке job класса RenderThread.

|  |
| --- |
| **void** SceneThread::start() {  std::thread(&SceneThread::job, **this**).detach();  std::thread(&ComponentsThread::job, &components Thread).detach();  components Thread.renderThread.job();  } |

SceneThread выполняет вычисления на сцене (перемещение, проверка входит ли объект в зону видимости камеры) и передаёт ComponentsThread объеты, которые нужно вывести на экран. В завершении обработки кадра класс ждёт завершении работы RenderThread, чтобы начать обработку следующего кадра.

|  |
| --- |
| **void** SceneThread::callback() {  ...  **for** (**auto**& it : scene->actors) {  **if** (\*it->GetName() == "Wilson") {  (**reinterpret\_cast**<Pawn\*>(it))->MoveForward();  it->updateAnimation(it-> GetAnimation(GameManager::Time.GetCurrentTime()));  } **else** **if** (\*it->GetName() == "Spider") {  (**reinterpret\_cast**<Pawn\*>(it))->MoveTowards(scene->actors[1], 0.008);  }  componentsThread.pushActor(it);  }  componentsThread.setEnd();  **while** (!endTick) {  **if** (GameManager::IsEnd) **return**;  }  endTick = **false**;  } |

ComponentsThread вычисляет анимацию объектов, а затем матрицу преобразования для каждого компонента модели. В RenderThread передаётся спрайт и его матрица преобразования.

|  |
| --- |
| **void** ComponentsThread::callback() {  **if** (actors.empty()) {  **if** (endTick) {  endTick = **false**;  renderThread.setEnd();  }  } **else** {  mutex.lock();  Actor\* actor = actors.top();  actors.pop();  mutex.unlock();  **for** (**auto**& it : actor->getActorComponents()) {  renderThread.pushSprite(  std::pair<Matrix4f<GLfloat>, Sprite \*>(  GameManager::render->pipeline.GetTransform(it->transform), it->sprite  )  );  }  }  } |

RenderThread отрисовывает объекты и в завершении своей работы сообщает об конце кадра классу SceneThread.

|  |
| --- |
| **void** RenderThread::callback() {  **if** (sprites.empty()) {  **if** (endTicks == **false**) { std::this\_thread::sleep\_for(std::chrono::milliseconds(THREADS\_SLEEP\_TIME\_MS \* THREAD\_RENDER));  **return**;  }  endTicks = **false**;  \*sceneEndTickPtr = **true**;  swapBuffer();  } **else** {  mutex.lock();  std::pair<Matrix4f<GLfloat>, Sprite\*> sprite = sprites.top();  sprites.pop();  mutex.unlock();  GameManager::render->drawObject(sprite.first, sprite.second);  }  } |

Для предотвращения «гонки данных» используются мьютексы в callback и других методах:

|  |
| --- |
| **void** ComponentsThread::pushActor(Actor\* actor) {  std::lock\_guard<std::mutex> lock(mutex);  actors.push(actor);  } |
| **void** RenderThread::pushSprite(**const** std::pair<Matrix4f<GLfloat>, Sprite\*>& componentSprite) {  std::lock\_guard<std::mutex> lock(mutex);  sprites.push(componentSprite);  } |

Для сообщения другим потокам об завершении своей работы используется атомарная переменная std::atomic<bool> endTicks.

## 5.11 GameManager

Для объединения всех модулей используется класс GameManager, который хранит ссылки на все важные объекты проекта. Через него любой модуль может обратиться к нужному ресурсу. В роль класса входит инициализация проекта и объектов.

|  |
| --- |
| **class** GameManager  {  **public**:  GameManager ();  ~GameManager ();  **static** **void** PushCamera(Camera \*\_camera);  **static** **void** PushPlayer(Character \*\_player);  **static** Camera \*createCamera();  **static** **void** KeyboardCB(GLFWwindow\* window, **int** key, **int** scancode, **int** action, **int** mods);  **static** **void** InitializeObjects();  **static** **void** InitializeGLFW(**int** \_width, **int** \_height);  **struct** CallbackData {  Camera\* camera = **nullptr**;  Character\* player = **nullptr**;  };  **inline** **static** GameTime Time;  **inline** **static** SceneThread\* threads = **nullptr**;  **inline** **static** Render\* render = **nullptr**;  **inline** **static** CallbackData callbackData = {**nullptr**, **nullptr**};  **inline** **static** GLFWwindow\* window = **nullptr**;  **inline** **static** **int** width = 640;  **inline** **static** **int** height = 480;  **inline** **static** **bool** IsEnd = **false**;  }; |

Инициализация объектов:

|  |
| --- |
| **void** GameManager::InitializeObjects()  {  Sprite::initializeGeometry();  } |

Инициализация проекта:

|  |
| --- |
| **void** GameManager::InitializeGLFW(**int** \_width, **int** \_height)  {  **if** (!glfwInit()) {  std::cerr << "Error: " << "Failed to initialize GLFW" << std::endl;  exit(EXIT\_FAILURE);  }  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);  glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);  width = \_width;  height = \_height;  window = glfwCreateWindow(width, height, "Game", NULL, NULL);  **if** (!window) {  glfwTerminate();  std::cerr << "Error: " << "Failed to create GLFW window" << std::endl;  exit(EXIT\_FAILURE);  }  glfwMakeContextCurrent(window);  glfwSetKeyCallback(window, GameManager::KeyboardCB);  glfwSetWindowUserPointer(window, &callbackData);  glfwSetInputMode(window, GLFW\_STICKY\_KEYS, GL\_TRUE);  GLenum err = glewInit();  **if** (err != GLEW\_OK) {  std::cerr << "Error: " << "Failed to initialize glew" << std::endl;  exit(EXIT\_FAILURE);  }  glEnable(GL\_DEPTH\_TEST);  glClearColor(0.12f, 0.12f, 0.12f, 0.0f);  render = **new** Render(callbackData.camera);  **if** (render == **nullptr**) {  std::cerr << "Error: " << "Failed to allocate memory to the renderer" << std::endl;  exit(EXIT\_FAILURE);  }  threads = **new** SceneThread();  **if** (threads == **nullptr**) {  std::cerr << "Error: " << "Failed to allocate memory to the threads" << std::endl;  exit(EXIT\_FAILURE);  }  } |

Функция для обработки нажатия клавиш:

|  |
| --- |
| **void** GameManager::KeyboardCB(GLFWwindow\* window, **int** key, **int** scancode, **int** action, **int** mods)  {  **static** **bool** keys[GLFW\_KEY\_LAST] = {**false**};  **const** GLfloat speed\_rotation = -0.125;  Character\* player = callbackData.player;  Camera\* camera = callbackData.camera;  **if** (action == GLFW\_PRESS) {  keys[key] = **true**;  } **else** **if** (action == GLFW\_RELEASE) {  keys[key] = **false**;  }  **if** (keys[GLFW\_KEY\_F]) {  glfwSetWindowShouldClose(window, GLFW\_TRUE);  GameManager::IsEnd = **true**;  }  player->SetDirection(Vector3<GLfloat>(  keys[GLFW\_KEY\_A] \* 1.0 - keys[GLFW\_KEY\_D] \* 1.0,  0,  keys[GLFW\_KEY\_S] \* 1.0 - keys[GLFW\_KEY\_W] \* 1.0  ));  **if** (keys[GLFW\_KEY\_Q]) {  camera->Params.Target.x -= speed\_rotation;  }  } |

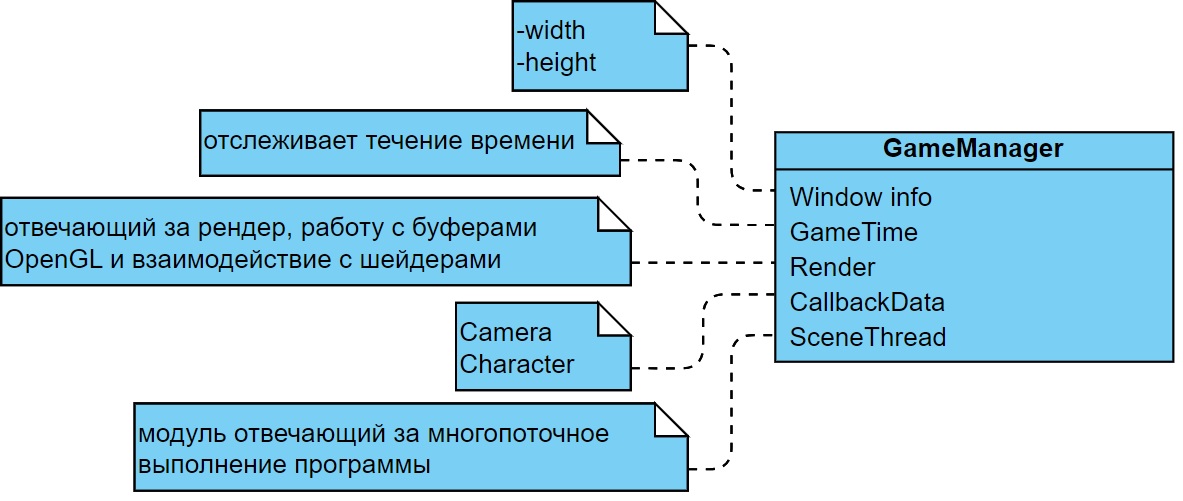


Рис. 8 Диаграмма класса GameManager

# 6. Сборка проекта

Проект собирается при помощи Cmake, который сильно упростил разработку проекта. Благодаря FetchContent было удобно интегрировать в программу код .

OpenGL и GLFW – кроссплатформенные библиотеки, поэтому программу можно собрать как для Linux, так и для Windows.

# 7. Функция main

В функции main происходит инициализация библиотек, окна и создание сцены. Затем запускается многопоточная обработка сцены и вывод её на экран. В завершении программы освобождается память, уничтожается окно и glfw завершает свою работу.

|  |
| --- |
| #include <game/gameManager.hpp>  **int** main(**int** argc, **char**\*\* argv)  {  **const** **int** width = 1600, height = 960;  GameManager::InitializeGLFW(width, height);  GameManager::InitializeObjects();  std::unique\_ptr<Scene> scene(createScene());  GameManager::threads->setScene(scene.get());  GameManager::threads->start();  glfwDestroyWindow(GameManager::window);  glfwTerminate();  **return** 0;  } |

# 8. Оптимизация

1. Благодаря созданию буферов, оптимизации рендера значительно сократилось число обращений к OpenGL
2. Оптимизация текстур. Разрешение текстур приводится к значению кратному степени двойки. Влияние на fps не значительное, но благодаря даже незначительной оптимизации программа сможет использовать трудоёмкие технологии для исправления артефактов графики и улучшения картинки.
3. многопоточность. Благодаря многопоточности программа повысила свою производительность и расширила предел своих возможностей.

В будущем можно добавить функции рендеринга, использующие инстансинг.

Инстансинг – технология, позволяющая выводить множество объектов, используя один вызов функции отрисовки, что избавляет от лишнего обмена между CPU и GPU.

# 9. Тестирование

При помощи Valgrind был проведён анализ утечек памяти. В результате было потеряно 6,319 Кбайт, что не критично. Все утечки происходят в сторонних библиотеках, на которые я повлиять на прямую не имею возможности.

|  |
| --- |
| ==488== LEAK SUMMARY:  ==488== definitely lost: 6,319 bytes in 258 blocks  ==488== indirectly lost: 56 bytes in 1 blocks  ==488== possibly lost: 0 bytes in 0 blocks  ==488== still reachable: 261,336 bytes in 2,642 blocks  ==488== suppressed: 24 bytes in 1 blocks  ==488== Reachable blocks (those to which a pointer was found) are not shown.  ==488== To see them, rerun with: --leak-check=full --show-leak-kinds=all  ==488==  ==488== For lists of detected and suppressed errors, rerun with: -s  ==488== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0) |

Нагрузочное тестирование показало хороший результат. При отрисовке 20 000 компонентов fps в среднем сохранялось выше 60. Следовательно, можно использовать трудоёмкие технологии для повышения качества изображения, например, сглаживание.



Рис. 9 Нагрузочное тестирование

# 10. Применение

Разработанный рендер может работать только в real-time режиме, поэтому он больше подходит для игр, интерактивных программ, визуализации небольших баз данных.

Примеры:

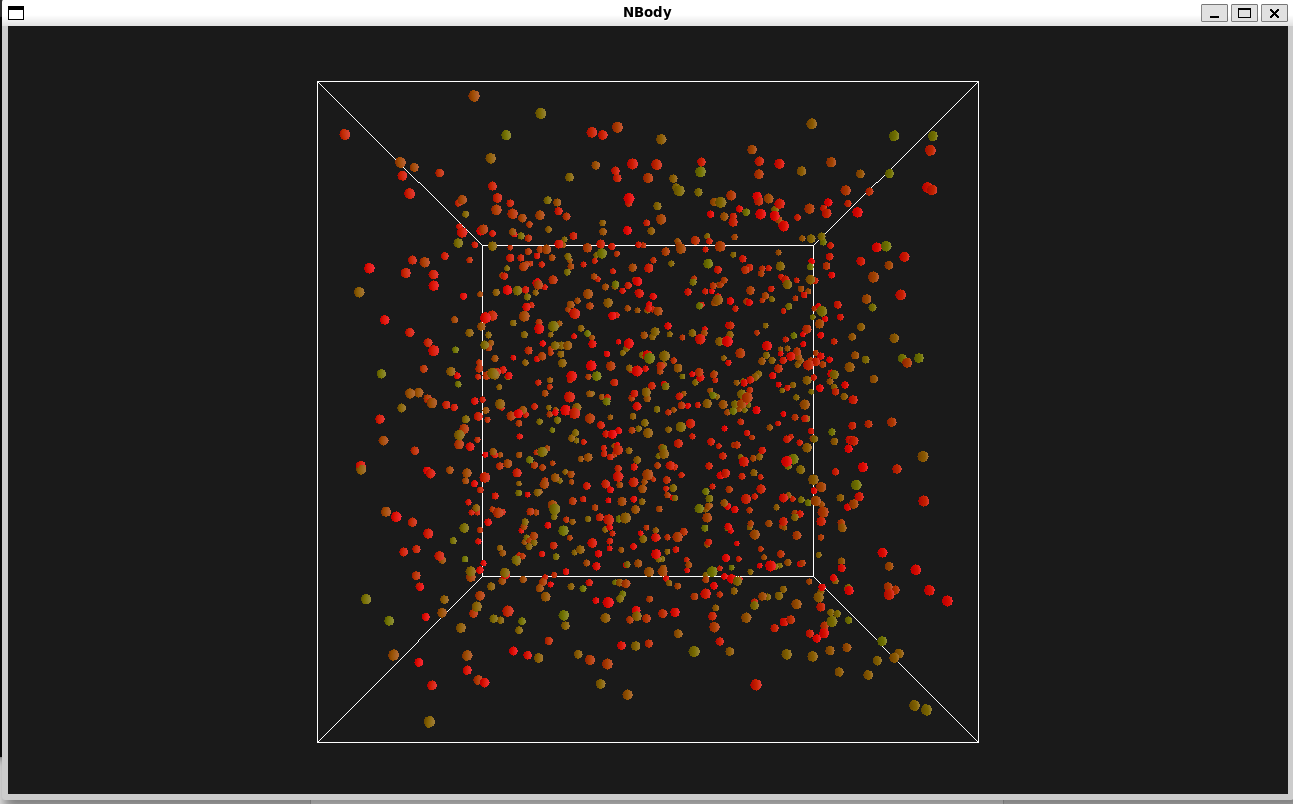


Рис. 10 Визуализация моделирования молекулярной динамики. N-body задача

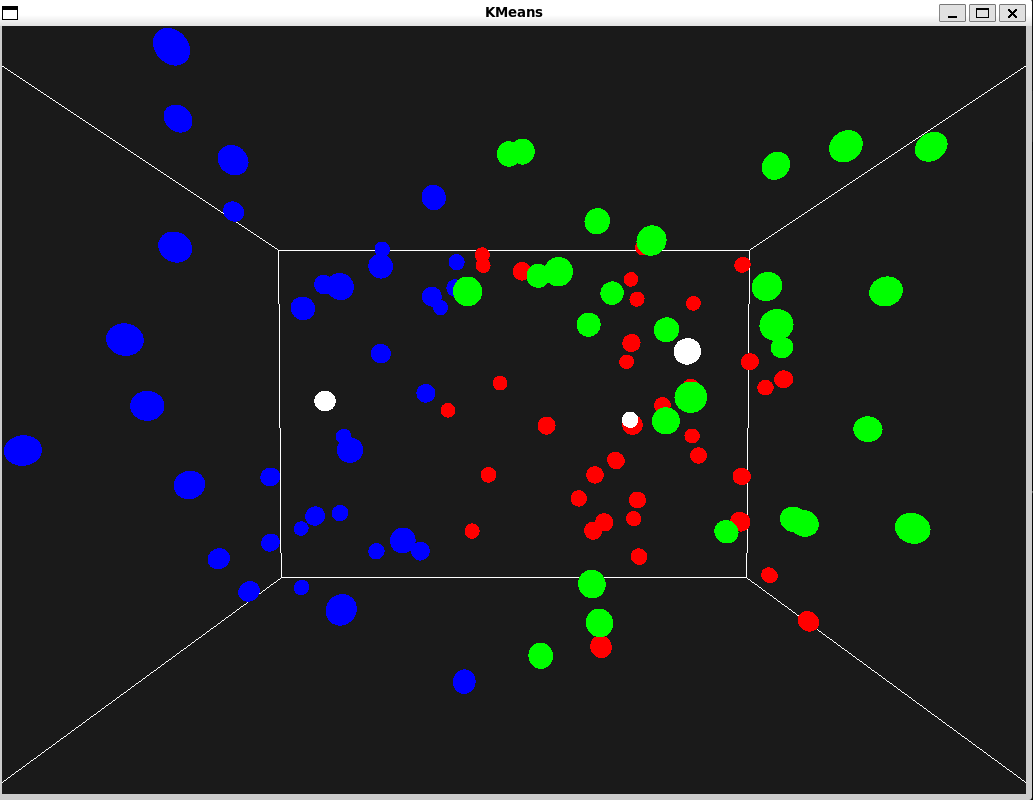


Рис. 11 Визуализация 3D кластеризации. KMeans задача (белые точки – центроиды)

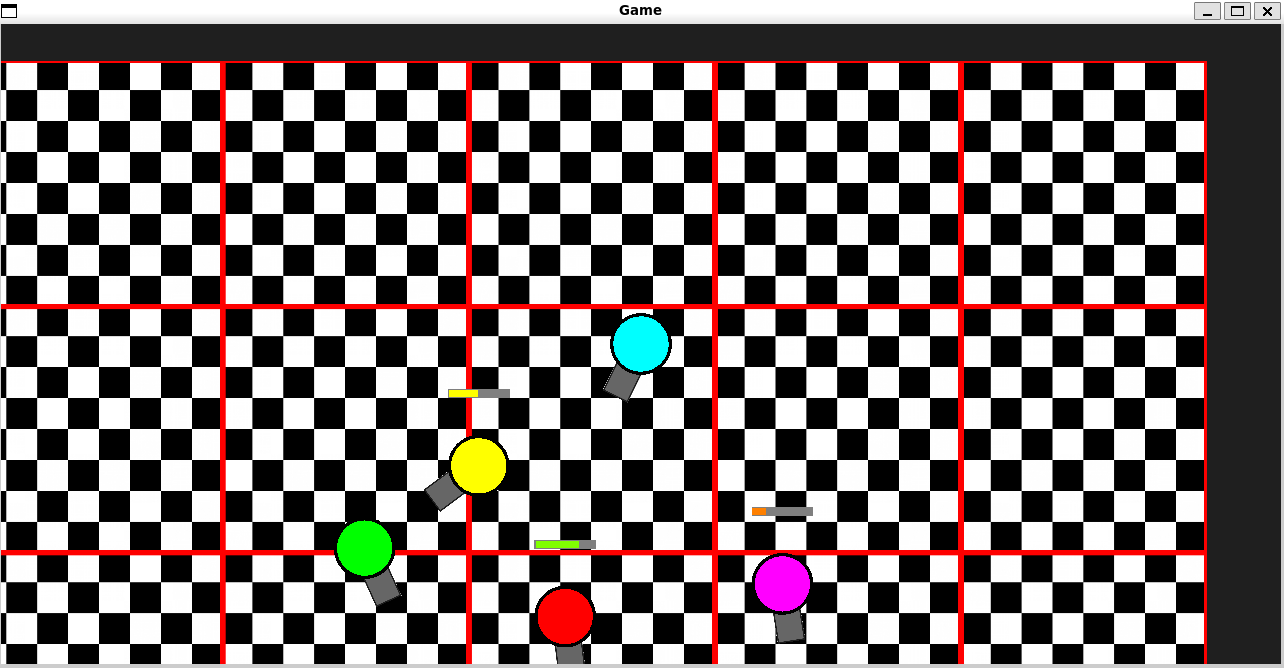


Рис. 12 Рендеринг игр



Рис. 13 Рендеринг интерактивных сцен

Примеры демонстрируют работоспособность программы, её возможности и эффективность.

# 11. Заключение

В результате была разработан простейший рендер, подходящий для разных задач. Благодаря модульности программу легко поддерживать и изменять. Для повышения эффективности было принято много мер оптимизации и проведены тестирования.

В завершении работы над проектом я:

* Ознакомился с принципом работы графики и подкрепил их опытом
* Развил навыки планирования архитектуры проекта
* Подкрепил навыки разработки многопоточных программ
* Получилось создать свою первую кроссплатформенную оконную программу

# Список используемой литературы

1.. М. Г. Курносов, Д. М. Берлизов. «Алгоритмы и структуры обработки информации» – Новосибирск: Параллель, 2019. – 227 с

2.Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. / М.: Издательский дом «Вильямс», 2011.— 1296 с.

3.THE ART OF MOLECULAR DYNAMICS SIMULATION Second Edition D. C. RAPAPORT. – Cambridge University Press 1995, Dennis Rapaport 2004 / <http://www.r-5.org/files/books/computers/algo-list/fuzzy-logic/modeling/Dennis_Rapaport-The_Art_of_Molecular_Dynamics_Simulation-EN.pdf>

4. Уроки по OpenGL с сайта OGLDev <https://triplepointfive.github.io/ogltutor/>

# Приложения

Репозиторий github с кодом программы: <https://github.com/Denis-Kulikov/OpenGL/tree/game>

Видео демонстрация работы программы:

* кластеризация: <https://disk.yandex.ru/i/0uNDMwgsZgeXAw>
* интерактивная сцена: <https://disk.yandex.ru/i/CxsUrtxPRDLTww>

**Отзыв о работе студента**

|  |
| --- |
|  |
| (ФИО студента) |
|  |

Уровень освоения компетенций

|  |  |
| --- | --- |
|  | |
| (ФИО студента) | |
|  | |
| Компетенции | | Уровень сформированности  компетенций |
| *ПК-6 - Способность собирать, обрабатывать и интерпретировать экспериментальные данные, необходимые для проектной и производственно- технологической деятельности; способность к разработке новых алгоритмических, методических и технологических решений в конкретной сфере профессиональной деятельности.* | |  |
| *ПК-7 - Способность к анализу требований и разработке вариантов реализации информационной системы; способность к оценке качества, надежности и эффективности информационной системы в конкретной профессиональной сфере.* | |  |

отметка о зачете с оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель практики от СибГУТИ:

Заведующий кафедрой ВС Перышкова Евгения Николаевна

Должность руководителя подпись ФИО руководителя

"13" июля 2024 г.