Оглавление

Введение

Рендеринг — это процесс создания изображения или последовательности из изображений на основе двухмерных или трёхмерных данных. Данный процесс происходит с использованием компьютерных программ и зачастую сопровождается сложными и комплексными техническими вычислениями, которые ложатся на вычислительные мощности компьютера или на отдельные его комплектующие части.

Процесс рендеринга присутствует в разных сферах профессиональной деятельности: киноиндустрия, видеоблогинг, игровая индустрия, телеиндустрия и т. д.. Зачастую, рендер является последним или предпоследним этапом в работе над проектом, после чего работа считается завершенной или же нуждается в небольшой постобработке. Также стоит отметить, что нередко рендером называют не сам процесс рендеринга, а скорее уже завершенный этап данного процесса или его итоговый результат[1].

Если мы говорим о задаче рендеринга изображения, то чаще всего имеем в виду рендеринг в трехмерной графике, так как это задача является самой востребованной в этой сфере. Это связано с тем, что как такового трехмерного измерения в компьютерной графике не существует и работа ведется с двухмерным изображением - с экраном.

В качестве такого изображения для данной работы была взята анимация развевающегося на ветру флага. Цель работы заключается в моделировании и визуализации данного изображения в трёхмерном пространстве.

1. Аналитический раздел

В данном разделе производится анализ и сравнение методов и алгоритмов предметной области, необходимых для выполнения поставленной задачи.

1.1 Постановка технического задания

Необходимо разработать программу моделирования анимации развевающегося на ветру флага на флагштоке методом ключевых кадров. Для каждого промежуточного кадра программа должна рассчитывать текущее положение точек поверхности флага. Должен быть задан набор законов управления движением при переходе между двумя соседними ключевыми кадрами. Исследовать возможность учёта освещённости, оптических свойств поверхностей. Источником света является солнце.

В данной задаче можно выделить следующие подзадачи:

- реализация процедуры создания объектов сцены;
- анализ, выбор и реализация алгоритма удаления невидимых линий;
- анализ, выбор и реализация алгоритма закраски объекта;
- реализация камеры и источника света;
- реализация анимации при помощи метода ключевых кадров на основе законов движения;
- реализация пользовательского интерфейса.

1.2 Формализация объектов синтезируемой сцены

Сцена включает в себя следующие объекты: флагшток, закреплённый на нём флаг, платформа, на котором стоит флагшток с флагом.

Флагшток представляет из себя тонкий длинный цилиндр с наконечником в виде полусферы. Определяется координатами центра нижнего основания, диаметром оснований и высотой. Радиус полусферы наконечника флагштока равен диаметру основания флагштока.

Флаг представляет из себя прямоугольное полотнище с отношением ширины к длине 2:3. Ширина флага равна 1/6 части высоты флагштока. Определяется координатами точки крепления верхней части флага, шириной и длиной. Для придания гибкости флагу полотнище разбивается на несколько треугольников.

Платформа определяется прямоугольным параллелепипедом с квадратным основанием. В центре имеется отверстие в форме цилиндра, радиус которого равен радиусу цилиндра. Таким образом центр основания флагштока совпадает с центром нижней грани платформы.

1.3 Законы управления движением точек объектов

Для решения задачи расчёта нового положения точек поверхности флага следует определить вектор скорости точки в определённый момент времени. Для этого необходимо проанализировать физические факторы, определяющие движение точки.

Если распределить массу флага M по всем точкам его поверхности, то можно говорить о силах, действующих на точку поверхности. Чтобы сдвинуть точки, в которых флаг крепится к флагштоку, необходимо приложить бесконечно большую силу. Будем считать, что точки, в которых флаг крепится к флагштоку, невесомыми, чтобы не учитывать воздействие сил на данные точки (их масса $m = 0$). Рассмотрим силы, воздействующие на остальные точки флага.

Так как точка имеет массу, то на неё воздействует сила гравитации $\vec{F}_{grav}$, направленная к земле. Вектор этой силы определён одинаковым образом для каждой точки:

$$\vec{F}_{grav} = m\vec{g}, \tag{1.1}$$

где m - масса точки, $\vec{g}$ - вектор ускорения свободного падения.

Точки поверхности между собой связаны, потому на каждую из них действуют силы упругости ткани $\vec{F}_{spring}$. Силы упругости стремятся вернуть точки в исходное состояние. Так как точки поверхности связаны между собой рёбрами, то эти рёбра могут выступать в качестве пружин. Каждая такая пружина имеет свою жёсткость и длину в состоянии покоя. Вектор силы воздействия пружины на точки рассчитывается по закону Гука

$$\vec{F}_{spring} = k\vec{x}, \tag{1.2}$$

где k - коэффициент жёсткости пружины, $\vec{x}$ - вектор изменения длины пружины, направленный вдоль пружины.

Направление силы определяется вектором $\vec{x}$, то есть тем, растянута или сжата пружина: если $\vec{x}$ направлен от точки, то пружина растянута, если к точке - сжата.

Сила ветра $\vec{F}_{wind}$ прикладывается ко всем граням ткани флага, поэтому необходимо вначале определить вектор направления силы, создаваемой ветром в каждой точке. Для этого находим сначала векторы нормали $\vec{n}$ к каждой грани, после чего находим угол между вектором ветра $\vec{w}$ и вектором нормали $\vec{n}$ (обозначим как α) из их скалярного произведения

$$\alpha = \frac{x_w x_n + y_w y_n + z_w z_n}{|\vec{w}||\vec{n}|}, \tag{1.3}$$

где $(x_w, y_w, z_w), (x_n, y_n, z_n)$ - координаты векторов $\vec{w}$ и $\vec{n}$ соответственно, $|\vec{w}|, |\vec{n}|$ - их длины.

Угол α определяет направление вектора силы воздействия ветра на грань. Так как каждая точка являются частью одновременно нескольких граней (кроме угловых точек, каждая из них принадлежит одной грани), то результирующий вектор силы воздействия ветра $\vec{F}_w$ для данной точки равен сумме векторов воздействия на грани.

Замедляется движение точек поверхности под воздействием сил трения. Без трения движение может никогда не прекратиться. Вектор силы трения $\vec{F}_{fric}$ определяется следующим образом:

$$\vec{F}_{fric} = -\mu\vec{F}, \tag{1.4}$$

где μ - коэффициент трения, $\vec{F}$ - результирующая сила, действующая на точку.

Как видно из уравнения 1.4, сила трения направлена в обратную от результирующей силы сторону, чем и замедляет движение.

После определения всех составляющих сил определяется вектор результирующей силы $\vec{F}$ как сумма составных

$$\vec{F} = \vec{F}_{grav} + \vec{F}_{spring} + \vec{F}_{fric}. \tag{1.5}$$

По второму закону Ньютона определяется ускорение точки

$$\vec{a} = \frac{\vec{F}}{m}. \tag{1.6}$$

Для того чтобы сократить число вычислений и чтобы не считать каждую миллисекунду новое значение ускорения $\vec{a}$, воспользуемся интегрированием для вычисления ускорения $\vec{a}$, достигнутого за заданный промежуток времени t

$$\vec{a} = t\frac{\vec{F}}{m}. \tag{1.7}$$

По найденному ускорению определяется новая скорость движения точки $\vec{v}$

$$\vec{v} = \vec{v}_0 + \vec{a}t, \tag{1.8}$$

где $\vec{v}_0$ - предыдущее значение скорости. Исходя из полученного вектора скорости определяются новые координаты (x, y, z) заданной точки:

$$(x, y, z) = (x_0 + v_x, y_0 + v_y, z_0 + v_z), \tag{1.9}$$

где (x_0, y_0, z_0) - координаты предыдущего положения точки, (v_x, v_y, v_z) - координаты вектора скорости $\vec{v}$.

Таким образом, итоговая система уравнений, определяющая движение незакреплённых точек поверхности полотна флага, имеет следующий вид:

$$\begin{cases} \vec{F}_{grav} = m\vec{g} \\ \vec{F}_{spring} = k\vec{x} \\ \vec{F}_{fric} = -\mu\vec{F} \\ \vec{F} = \vec{F}_{grav} + \vec{F}_{spring} + \vec{F}_{fric} \\ \vec{a} = t\dfrac{\vec{F}}{m} \\ \vec{v} = \vec{v}_0 + \vec{a}t \\ (x, y, z) = (x_0 + v_x, y_0 + v_y, z_0 + v_z) \end{cases}. \tag{1.10}$$

1.4 Визуализация

1.4.1 Алгоритм Брезенхема

Алгоритм Брезенхема построения отрезков выбирает оптимальные растровые координаты для представления отрезка, другими словами, данный алгоритм позволяет получить приближение идеальной прямой точками растровой сетки.

Основная идея алгоритма базируется на расстоянии между действительным положением отрезка и ближайшими координатами сетки. Это расстояние называют ошибкой[2].

В данном алгоритме проверяется принадлежность проекции к оси x или y. На какую ось проекция будет больше, на ту ось и смещается пиксель. По другой оси смещение на один пиксель проискходить лишь в том случае, когда линия отклонилось от оси более чем на половину пикселя.

Преимущество:

- высокая скорость растрирования вектора.

Недостатки:

- при маленьком разрещении экрана наблюдается эффект ступенчатости;
- округление значений: действительные величины преобразуются в ближайшие целые числа.

1.4.2 Удаление невидимых линий

Удаление невидимых линий и поверхностей не самая простая задача в компьютерной графике: на данный момент не существует такого алгоритма, который работал бы быстро и с высокой детализацией результата. Под детализацией понимается: просчет теней, прозрачности, фактуры, отражения и преломления света.

Алгоритмы удаления невидимых линий и поверхностей делятся на два вида:

- алгоритмы, работающие в пространстве изображения;
- алгоритмы, работающие в объектном пространстве.

В алгоритмах первого вида находятся ближайшие точки сцены к наблюдателю и отображаются. Такой подход к удалению невидимых линий и поверхностей не требует высокой точности вычислений, поэтому сложности таких алгоритмов не превышает $O(c \cdot n)$, где c - число пикселей, n - число объектов [3]. В алгоритмах данного вида точность ограничивается разрешающей способностью экрана.

В алгоритмах второго вида все действия происходят в физической системе координат, в которой описаны данные объекты. Сложность таких алгоритмов в среднем для n объектов $O(n^2)$[3]. Однако, медленная скорость работы обусловлена точными результатами. Качество, полученных изображений, не будет зависеть от разрешений экрана.

В рамках курсового проекта необходимо выбрать такой алгоритм, который бы быстро удалял задние линии и поверхности, поэтому выбор осуществлялся между следующими алгоритмами:

- алгоритм художника;
- алгоритм Z-буфера;
- алгоритм A-буфера.

Рассмотрим каждый из данных алгоритмов, сравним достоинства и недостатки и определим, какой из алгоритмов лучше подходит для решения нашей задачи.

Алгоритм художника

Алгоритм художника является простейшим вариантом решения задачи об удалении невидимых линий и граней в компьютерной графике.

Название «алгоритм художника» был использован в связи с тем, что его реализация схожа с техникой, которой пользуются многие живописцы: сначала рисуют дальние части сцены, затем ближние части. Так и в алгоритме художника: многоугольники сортируются по координате z по возрастанию, в начале расположены те грани, которые расположены ближе всего к наблюдателю. После того, как грани отсортированы по расстоянию от наблюдателя, их начинают отрисовывать на сцене, начиная с самых дальних.

Главным недостатком данного алгоритма является неправильное вычислений задних граней при определенных сценариях: когда один многоугольник является относительно одной фигуры неэкранируемым, а относительно другой является экранируемым. Данную проблему можно наблюдать на рис. 1.1.

Также этот алгоритм работает медленно, из-за отрисовки ненужных задних граней, которые по итогу будут закрашены ближней гранью к наблюдателю. Также если задних граней будет много, то данный алгоритм будет весьма трудозатратным.

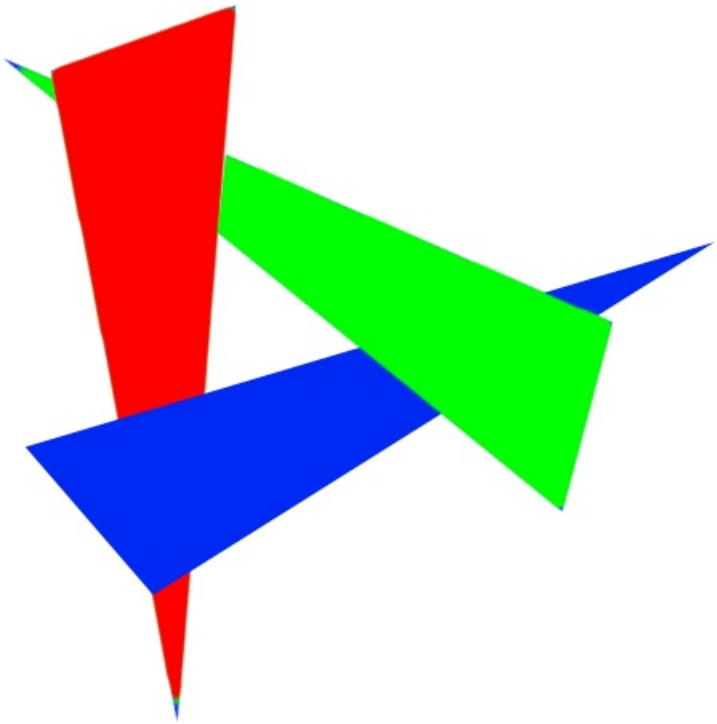

Рис. 1.1: Ошибочный вариант расположения фигур для алгоритма художника

Алгоритм Z-буфера

Алгоритм Z-буфера заключается в том, что помимо буфера кадра (матрицы, в которой содержится информация о цвете для каждого пикселя в пространстве изображения) используется также так называемый Z-буфер, в котором хранится координата z для каждого пикселя, т. е. его глубина.

Во время работы алгоритма, значение Z-буфера или глубины сравнивается со значением, который нужно занести в Z-буфер. Если сравнение показывает, что новое значение расположено ближе к наблюдателю по оси z, то новый пиксель заносится в буфер кадра, а в Z-буфер заносится новое значение глубины. Можно сказать, что алгоритм выполняет поиск наибольшего значения функции $z(x, y)$ по x и y. Визуализацию работы алгоритма можно посмотреть на рис. 1.2.

Данный алгоритм является быстродейственным, так как в нем не выполняются лишние операции, в отличие от алгоритма художника.

Существенными недостатками являются:

- для хранения буферов требуется большой объём памяти[4];
- пиксели в буфер кадра заносятся в произвольном порядке, поэтому появляются трудности с реализацией эффектов прозрачности или просвечивания.

Вторую проблему можно решить, если заранее известно, из каких материалов состоит объект.

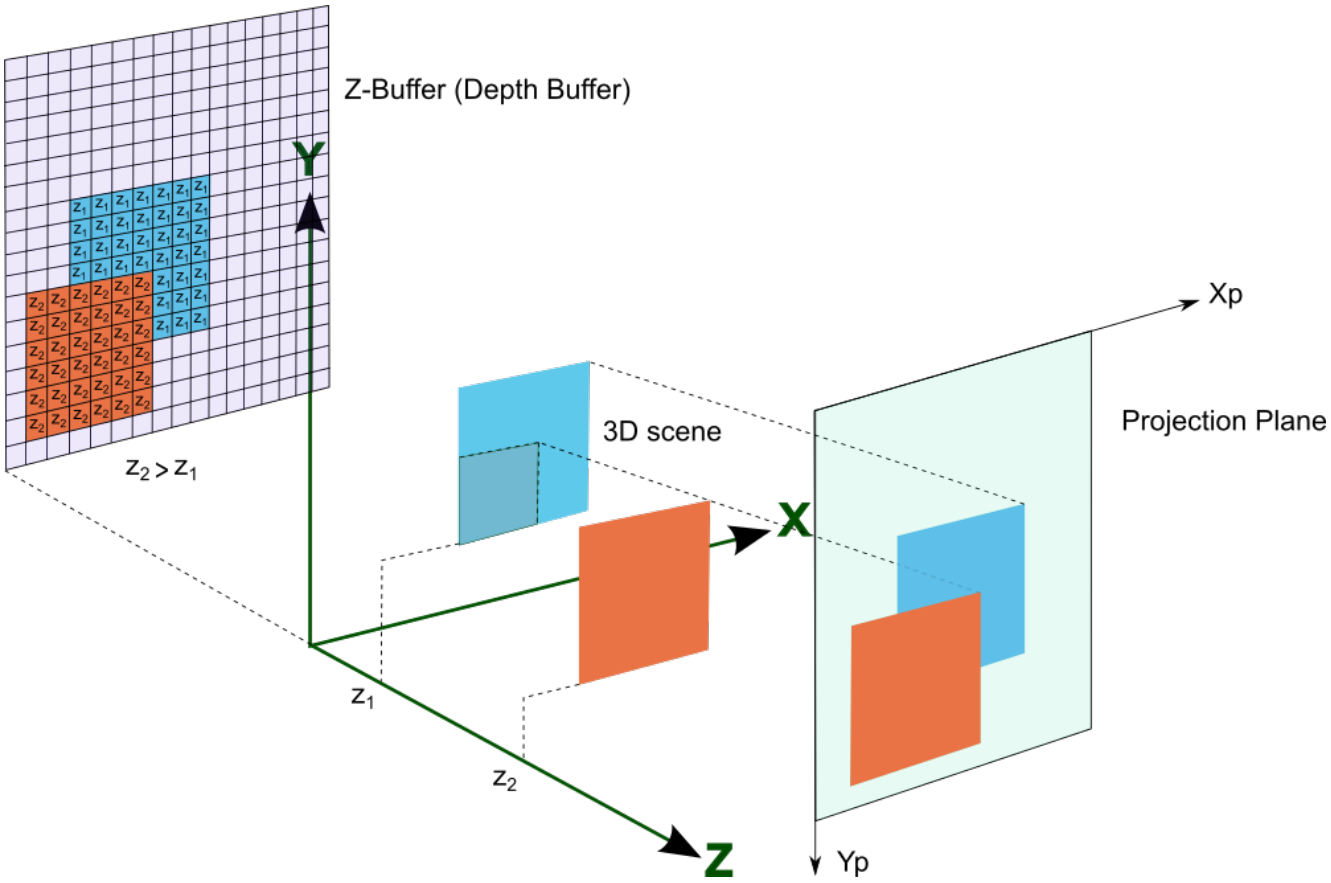

Рис. 1.2: Работа алгоритма z-буфер

Алгоритм A-буфера

Данный алгоритм называют модификацией Z-буфера. Алгоритм A-буфера специально предназначен для осуществления эффекта усреднения по области, прозрачности и наложения полигонов. Вместо Z-буфера используется буфер накопления, который помимо координаты z хранит ещё данные о поверхности (к примеру, прозрачность). Пример работы можно увидеть на рис. 1.3.

Выбор алгоритма для удаления задних линий

Алгоритмы, работающие в трехмерном пространстве, и алгоритм художника не подходят из-за высокой трудозатратности, т. к. число обрабатываемых точек модели полотна флага велико.

Для поставленной задачи будет использоваться алгоритм Z-буфера: флаг не состоит из прозрачных материалов, поэтому алгоритм A-буфера будет выполнять лишние действия.

1.4.3 Метод тонирования Гуро

Метод тонирования Гуро основан на интерполяции интенсивности, данный подход к закраске объекта позволяет устранить дискретность изменения интенсивности.

Процесс закраски Гуро можно разделить на четыре этапа:

1. вычисление нормалей ко всем полигонам;

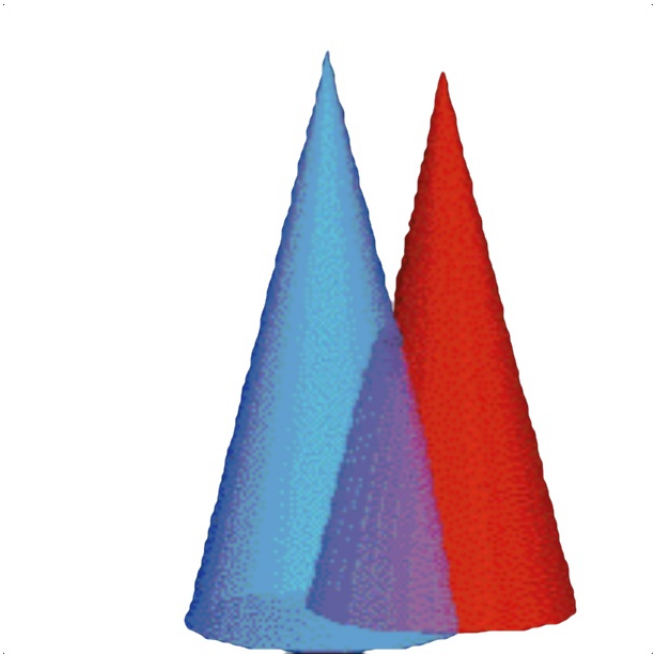

Рис. 1.3: Работа алгоритма a-буфер

2. определение нормали в вершинах путем усреднения нормалей по всем полигональным граням;

3. вычисление значений интенсивности в вершинах, используя нормали в вершинах и применяя произвольный метод закраски;

4. закрашивание многоугольника путем линейной интерполяции значений интенсивности в вершинах (сначала вдоль каждого ребра, а затем между ребрами).

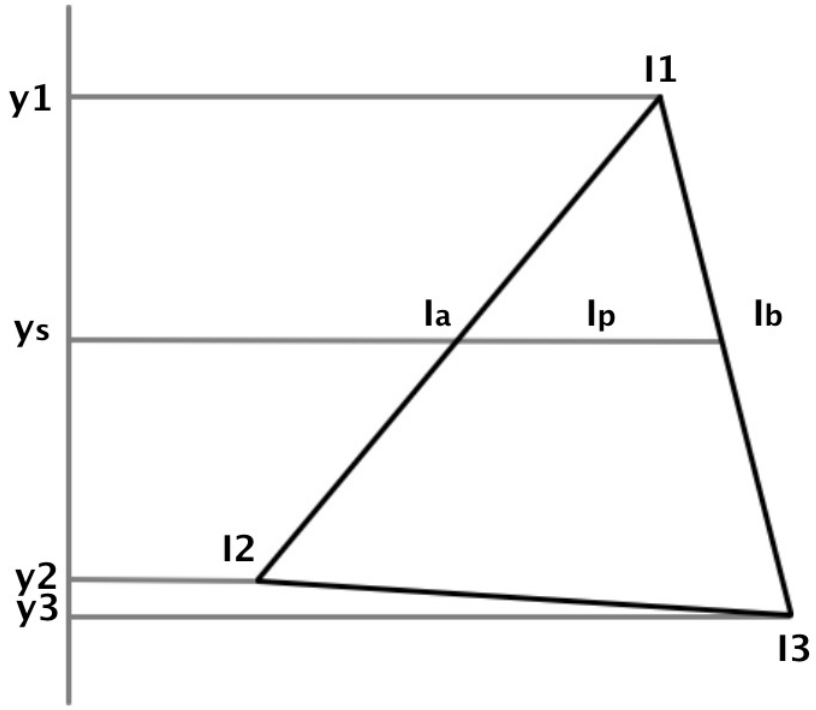

Рис. 1.4: Интерполяция интенсивностей

Интерполяция интенсивностей работает следующим образом: для всех ребер запоминается начальная интенсивность, изменение интенсивности при каждом шаге по координате y. Затем, заполнение видимого интервала производится путем интерполяции между значениями интенсивности на ребрах, ограничивающих интервал (рис. 1.4).

Ниже приведены три формулы вычисления интенсивности цвета для каждой границы строки от I_a до I_b, приведенной на рис. 1.4.

$$I_a = I_1\frac{y_s - y_2}{y_1 - y_2} + I_2\frac{y_1 - y_s}{y_1 - y_2} \tag{1.11}$$

$$I_b = I_1\frac{y_s - y_3}{y_1 - y_3} + I_3\frac{y_1 - y_s}{y_1 - y_3} \tag{1.12}$$

$$I_p = I_a\frac{x_b - x_p}{x_b - x_a} + I_b\frac{x_p - x_a}{x_b - x_a} \tag{1.13}$$

1.5 Метод ключевых кадров

Метод ключевых кадров является базовой анимацией во многих прикладных программах, и для многих программистов, незнакомых с анимированием объектов данный метод является интуитивно понятным. Основная идея использования ключевых кадров заключается в создании ключей анимации для начального и конечного положения объекта, при этом состояние объектов в промежуточных стадиях просчитывает компьютер.

Термины, использующиеся в данном методе:

- ключевой кадр - кадр с ключем анимации;
- ключ - маркер со значениями анимируемых объектов в данный момент;
- автоключ - процедура для отслеживания изменения параметров анимируемго объекта, может автоматически создавать ключевой кадр.

Наглядный пример работы метода ключевых кадров можно увидеть на рис. 1.5[7].

Рис. 1.5: Работа метода ключевых кадров на примере маятника

Недостатками данного метода являются

- большая трудоемкость создания и вычисления;
- большой объем памяти, необходимый для хранения промежуточных состояний объекта.

1.6 Вывод

В данном разделе были рассмотрены алгоритм рисования линий Брезенхема, алгоритмы удаления невидимых линий художника, Z-буфера и A-буфера, метод тонировки Гуро и метод ключевых кадров. В результате сравнения в качестве алгоритма удаления невидимых линий для решения поставленной задачи был выбран алгоритм Z-буфера. Для решении задачи изменения положения точек используется система уравнений 1.10, с помощью которой можно формализовать алгоритм решения.

2. Конструкторский раздел

В данном разделе будут рассмотрены типы и структуры данных, необходимые для решения поставленных задач. Будут формализованы алгоритм изменения положения точек полотна флага и алгоритм Z-буфера, а также будут описаны процесс рендера в целом и работа с камерой.

2.1 Алгоритм изменения положения точек

Формально последовательность действий в данном алгоритме можно описать следующим образом:

1. Определить вектора силы воздействия ветра.
   (a) Для каждой грани:
      i. Определить вектор нормали.
      ii. Определить угол между вектором нормали и вектором ветра.
      iii. Задать вектор воздействия ветра на грань (значение вектор ветра, направление в соответствии с углом).
      iv. Для каждой точки грани прибавить к результирующему векторы силы полученный вектор воздействия ветра.
2. Определить вектора сил упругости и трения пружины.
   (a) Для каждой «пружины»:
      i. Определить текущую длину пружину.
      ii. Определить вектор силы трения как вектор, пропорциональный относительной скорости двух точек.
      iii. Для каждой из двух вершин:
         A. Определить вектор силы упругости.
         B. Прибавить к вектору силы трения.
         C. Прибавить к результирующему вектору силы суммарный вектор.
3. Для каждой точки:
   (a) Определить вектор силы гравитации.
   (b) Прибавить вектор силы гравитации к результирующему.
   (c) Определить вектор силы трения.
   (d) Прибавить данный вектор к результирующему.
   (e) Найти ускорение точки путём интегрирования.
   (f) Найти новую скорость.
   (g) Изменить координаты точки в соответствии с найденной скоростью.

Схемы данного алгоритма представлены на рис. 2.1, рис. 2.2, рис. 2.3.

Рис. 2.1: Схема алгоритма вычисления нового положения точек

Рис. 2.2: Схема алгоритма вычисления векторов сил воздействия ветра

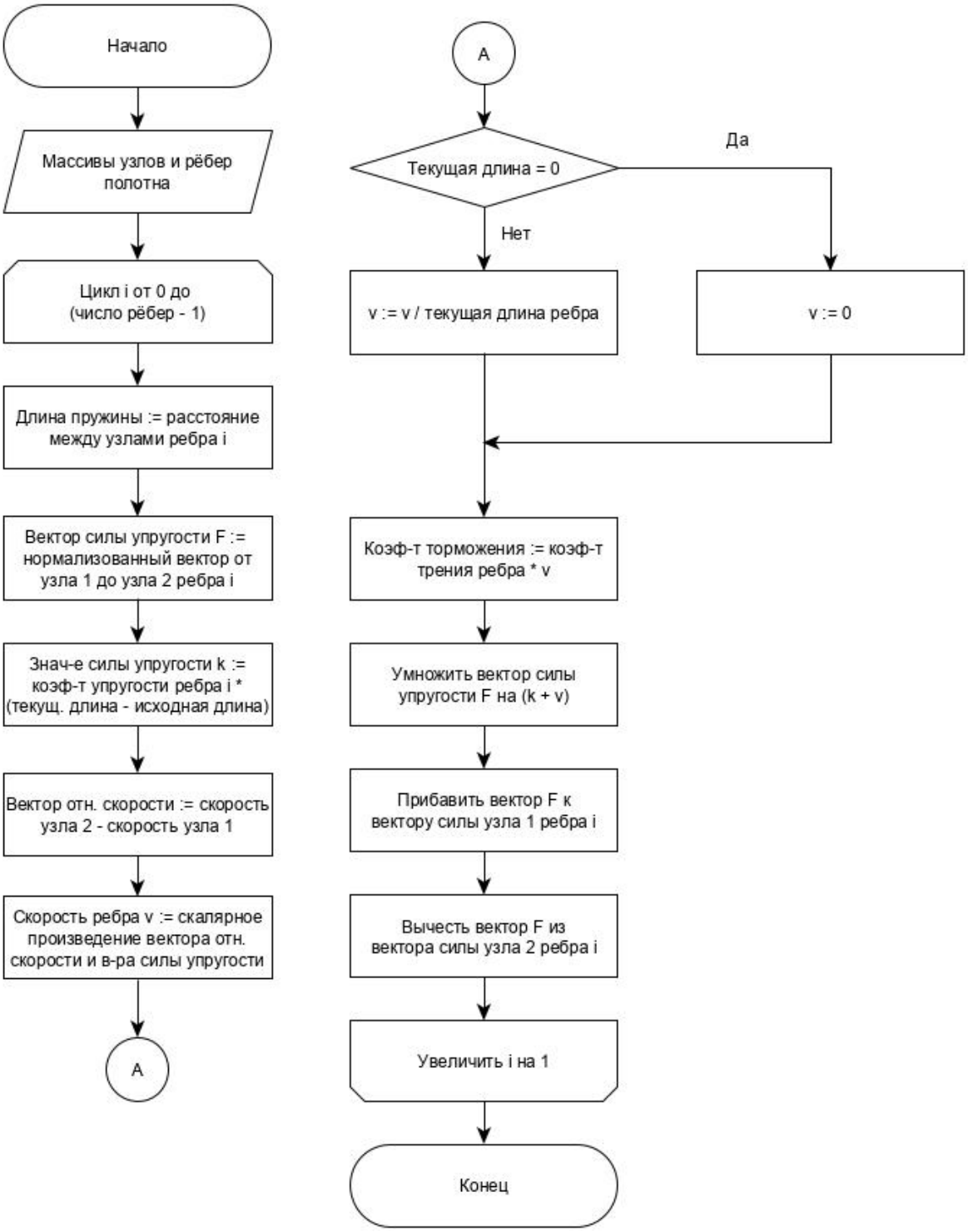

Рис. 2.3: Схема алгоритма вычисления векторов сил упругости и трения рёбер

2.2 Трёхмерные аффинные преобразования координат

В процессе работы программе возникает необходимость преобразования объектов сцены - их сдвига, масштабирования и поворота. Для трёхмерного пространства любо аффинное преобразование может быть представлено последовательностью простейших операций.

Ниже приведены уравнения преобразований координат (x, y, z) точки:

- сдвиг на (dx, dy, dz):

$$\begin{cases} x_{new} = x + dx \\ y_{new} = y + dy \\ z_{new} = z + dz \end{cases} \tag{2.1}$$

- масштабирование с коэффициентами k_x, k_y, k_z относительно точки C

$$\begin{cases} x_{new} = x_C + k_x(x - x_C) \\ y_{new} = y_C + k_y(y - y_C) \\ z_{new} = z_C + k_z(z - z_C) \end{cases} \tag{2.2}$$

- поворот на угол ϕ относительно
  - оси X

$$\begin{cases} x_{new} = x \\ y_{new} = y \cdot cos\phi - z \cdot sin\phi \\ z_{new} = y \cdot sin\phi + z \cdot cos\phi \end{cases} \tag{2.3}$$

  - оси Y

$$\begin{cases} x_{new} = z \cdot sin\phi + x \cdot cos\phi \\ y_{new} = y \\ z_{new} = z \cdot cos\phi - x \cdot cos\phi \end{cases} \tag{2.4}$$

  - оси Z

$$\begin{cases} x_{new} = x \cdot cos\phi - y \cdot sin\phi \\ y_{new} = x \cdot sin\phi + y \cdot cos\phi \\ z_{new} = z \end{cases} \tag{2.5}$$

2.3 Процесс отрисовки изображения

В качестве алгоритма визуализации объекта был выбран алгоритм Z-буфера. Формальное описание алгоритма Z-буфера:

1. инициализировать и заполнить буфер кадра фоновым значением цвета;
2. инициализировать и заполнить Z-буфер максимальным значением z;
3. преобразовать каждый многоугольник в растровую форму в произвольном порядке;
4. для каждого пикселя в многоугольнике вычислить его глубину $z(x, y)$ или расстояние по оси z;
5. сравнить глубину $z(x, y)$ со значением в Z-буфере в позиции $[x, y]$;
6. если $z(x, y) >$ Z-буфер(x, y), то записать атрибут (цвет, интенсивность и т. п.) в буфер кадра и в Z-буфер записать $z(x, y)$, иначе никаких действий не производить.

На рис. 2.4 представлен полный процесс отрисовки изображения.

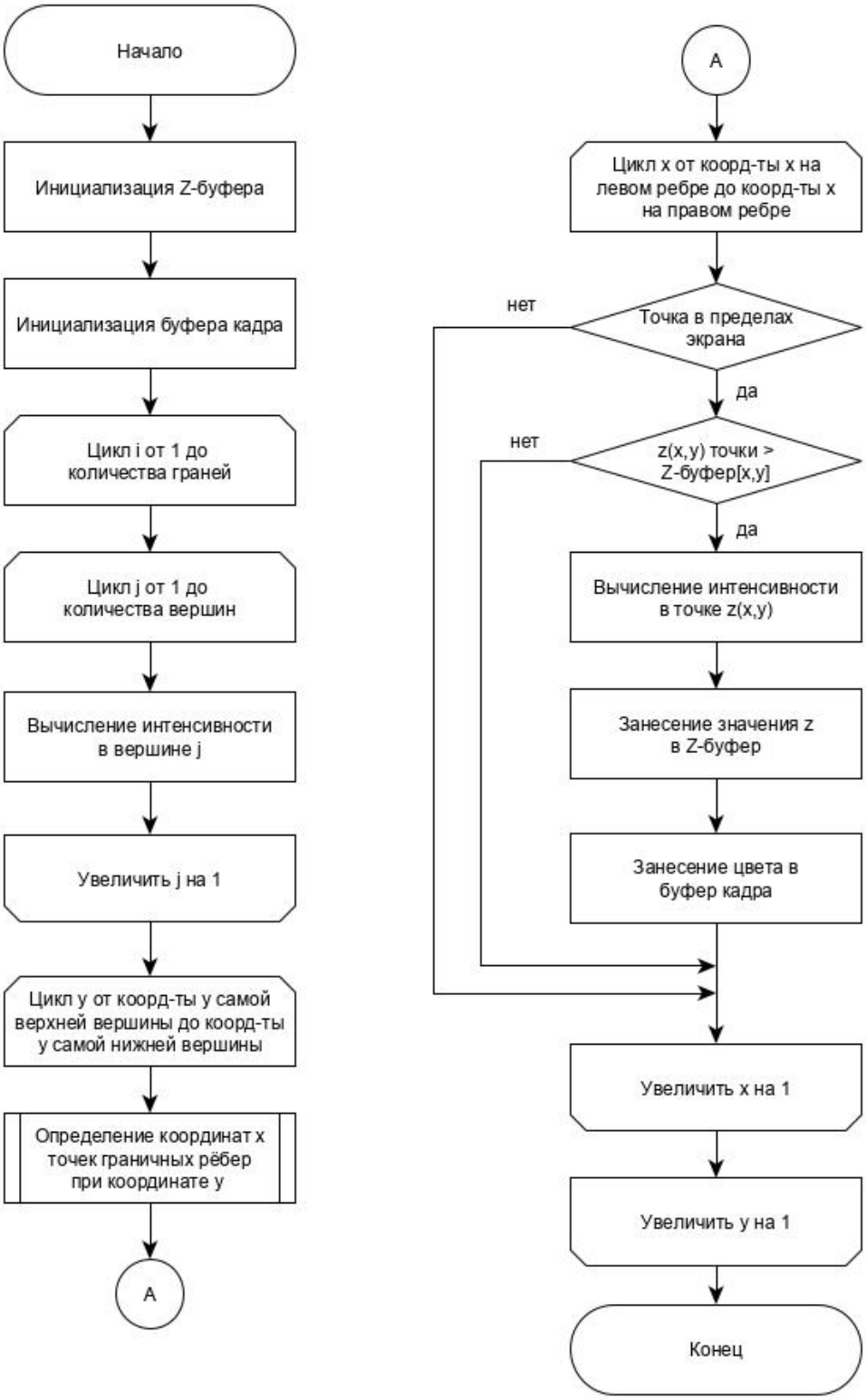

Рис. 2.4: Схема работы алгоритма Z-буфера

2.4 Анимация объектов

Для моделирования развевающегося флага был выбран метод анимации по ключевым кадрам. В данном методе происходит изменение положения различных частей объекта от начального состояния до конечного. Плавность анимации осуществляется за счет просчитывания промежуточных состояний объекта. В данном случае своё положение меняют

точки полотна флага в соответствии с системой уравнений 1.10. Каждый раз объект переходит из одного промежуточного состояния в другое и так, пока не достигнет конечного.

2.5 Используемые типы и структуры данных

Для того, что было возможно задать объекты, необходимо ввести следующие структуры:

- **вершина**: содержит координаты x, y, z;
- **грань (треугольник)**: содержит поля
  - массив из индексов трёх вершин;
  - цвет;
- **модель**: содержит поля
  - список «вершин»;
  - список «граней» («треугольников»).

Для работы алгоритма изменения положения точек полотна флага используются различные физические параметры. В связи с этим требуется ввести дополнительные структуры для точек полотна:

- **узел**: содержит поля
  - указатель на структуру «вершина», к которому привязан узел;
  - индекс данной вершины;
  - начальное положение узла - структура «вершина»;
  - значение массы узла и обратное значение массы (для упрощения вычислений);
  - значение коэффициента трения в узле;
  - математический вектор силы F;
  - математический вектор силы v;
- **ребро** (соединяет узлы): содержит поля
  - индексы «вершин» двух «узлов», которые «ребро» связывает;
  - длина ребра в состоянии покоя (начальная длина);
  - текущая длина ребра;
  - значение коэффициента упругости ребра;
  - значение коэффициента трения на ребре;
- **флаг**: содержит поля
  - структура «модель», описывающая текущее состояние полотна флага;
  - список «узлов» в текущем положении;
  - список «рёбер» в текущем положении;
  - структура «модель», описывающая начальное состояние полотна флага;

  - список «узлов» в начальном положении;
  - список «рёбер» в начальном положении.

Для хранения объектов сцены в программе используется список структур «модель». Все значения координат, физических параметров и прочих полей, требуемых при вычислении, описываются вещественным типом *double* для получения более точных результатов при вычислении положения точек. Все индексы описываются целым типом *int*.

При рендере моделей значения координат округляются до целого для занесения в буфер кадра, так как пиксели неделимы. Также для выполнения математических действий, связанных с использованием математического вектора (сложения физических сил, вычисления интенсивности в точке), вводится соответствующая структура, которая

- содержит в качестве полей свои координаты x, y, z;
- предусматривает операции
  - определения длины вектора;
  - сложения и вычитания векторов;
  - умножения вектора на число;
  - скалярного произведения векторов;
  - векторного произведения векторов;
  - нормализации вектора.

2.6 Вывод

В данном разделе было дано формальное описание алгоритмам нахождения нового положения точек полотна флага, отрисовки изображения (алгоритм Z-буфера), были приведены математические формулы, описывающие аффинные преобразования координат точек. Также были перечислены и описаны основные типы и структуры данных, использующиеся в проекте.

3. Технологический раздел

В данном разделе будут рассмотрены требования к разрабатываемому программному обеспечиванию, средства, использованные в процессе разработки для реализации поставленных задач, а также представлены листинги кода программы. Будет описаны основные функции интерфейса программы и примеры работы программы (результат рендера).

3.1 Требования к программному обеспечению

Программное обеспечение должно реализовывать поставленную на курсовой проект задачу. В программном модуле должен присутствовать интерфейс для взаимодействия с объектом. Программа не должна аварийно завершаться, сильно нагружать процессор, а также не должна задействовать большой объем оперативной памяти.

3.2 Средства реализации

Для реализации программы был использован язык программирования C++ (стандарт GNU++11[8]). C++ имеет достаточный функционал в стандартных библиотеках, поддерживает многопоточность выполнения программы и является высокопроизводительным языком, что важно при реализации задача высокой нагрузки, таких как рендеринг.

Проект выполнен в среде разработки QtCreator с использованием сопутствующей графической оболочки QtGraphics[9]. Данная оболочка предоставляет удобный функционал отрисовки графических объектов по пикселям, а также возможность создания удобного интерфейса для взаимодействия с пользователем. Платформа включает в себя архитектуру распространения событий, которая позволяет использовать возможности взаимодействия элементов на сцене.

3.3 Сборка и запуск проекта

Для сборки проекта используется компилятор g++ и система сборки qmake, поставляемая Qt[9]. Система сборки qmake автоматически генерирует файлы сборки make, основываясь на информации в файлах проекта. Для корректной сборки приложения необходимы стандартные библиотеки языка C++ (включая библиотеки пространства имён std) и библиотеки QtGraphics.

3.4 Реализация структур и алгоритмов

На основе схем, приведённых в конструкторском разделе, в соответствии с указанными требованиями к реализации было разработано программное обеспечение, содержащее реализации выбранных алгоритмов. В данном подразделе приведены листинги 3.1-3.10 реализаций структур данных и алгоритма.

Листинг 3.1: Класс «вершина» Vertex

```
class Vertex
{
public:
 Vertex() : _x(0), _y(0), _z(0) {}
 Vertex(double x, double y, double z) : _x(x), _y(y), _z(z) {}
 Vertex(const Vertex &vertex);
 Vertex(Vertex &&vertex);

 Vertex& operator=(const Vertex &vertex);
 Vertex& operator=(Vertex &&vertex);

 double x();
 double y();
 double z();

 void set_x(double x);
 void set_y(double y);
 void set_z(double z);
 void set(double x, double y, double z);

 void shift(double dx, double dy, double dz);
 void scale(double kx, double ky, double kz, Vertex pc = Vertex(0, 0, 0));
 void rotate_x(double angle, Vertex pc = Vertex(0, 0, 0));
 void rotate_y(double angle, Vertex pc = Vertex(0, 0, 0));
 void rotate_z(double angle, Vertex pc = Vertex(0, 0, 0));

 void shift(std::vector<double> d);
 void scale(std::vector<double> k, Vertex pc = Vertex(0, 0, 0));
 void rotate(std::vector<double> a, Vertex pc = Vertex(0, 0, 0));

private:
 double _x, _y, _z;
};

double distance(Vertex &p1, Vertex &p2);
```

Листинг 3.2: Класс «грань» Trinangle

```
class Triangle
{
public:
 Triangle(size_t i1, size_t i2, size_t i3, QColor c=QColor(255, 0, 0));
 Triangle(const Triangle &tr);
 Triangle(Triangle &&tr);

 Triangle& operator=(const Triangle &tr);
 Triangle& operator=(Triangle &&tr);

 size_t get_vertex(size_t index);
 void set_vertex(size_t index, size_t value);

 std::vector<Vertex> get_vertices(std::vector<Vertex>& vertices);
 std::vector<MathVector> get_normals(std::vector<MathVector>& normals);
```

```
16
17  QColor get_color() const;
18  void set_color(QColor c);
19
20 private:
21  size_t vertex_indexes[3];
22  QColor colour;
23 };
```

Листинг 3.3: Класс «модель» Model

```
1  class Model : public VisibleObject
2  {
3  public:
4   Model() {}
5   Model(const Model &model);
6   Model(Model &&model);
7   ~Model() = default;
8
9   Model& operator=(const Model &model);
10  Model& operator=(Model &&model);
11
12  std::vector<Vertex>& get_vertices();
13  std::vector<Triangle>& get_triangles();
14
15  void shift(double dx, double dy, double dz);
16  void scale(double kx, double ky, double kz, Vertex pc=Vertex(0, 0, 0));
17  void rotate(double ax, double ay, double az, Vertex pc=Vertex(0, 0, 0));
18
19  void shift(std::vector<double> d);
20  void scale(std::vector<double> k, Vertex pc = Vertex(0, 0, 0));
21  void rotate(std::vector<double> a, Vertex pc = Vertex(0, 0, 0));
22
23  std::vector<MathVector> get_normals();
24
25 private:
26  std::vector<Vertex> vertices;
27  std::vector<Triangle> triangles;
28 };
```

Листинг 3.4: Класс «узел» Node

```
1  class Node
2  {
3  public:
4   Node(const Vertex& v, size_t index, double mass, double friction);
5   ~Node() = default;
6
7   Vertex start_pos;
8   double mass;
9   double inv_mass;
10  double friction;
11  MathVector f;
12  MathVector v;
```

```
    size_t vertex_index;
    std::shared_ptr<Vertex> vertex;
};
```

Листинг 3.5: Класс «ребро» Edge

```
class Edge
{
public:
    Edge(const Node &node1, const Node &node2, double spring_rate,
    double friction);
    ~Edge() = default;

    size_t v1;
    size_t v2;
    double len_rest;
    double len_cur;
    double spring_rate;
    double friction;
};
```

Листинг 3.6: Класс «флаг» Flag

```
class Flag
{
public:
    Flag(Vertex topleft, double w, double h, QColor colour);
    ~Flag() = default;

    std::shared_ptr<Model> get_model();

    void set_mass(double mass);
    void set_friction(double friction);
    void set_spring(double spring_rate);

    void reset();
    void update(MathVector wind_vector, int msec);
    void update_thread(MathVector wind_vector, int msec);

private:
    Model model;
    std::vector<Node> nodes;
    std::vector<Edge> edges;

    Model start_model;
    std::vector<Node> start_nodes;
    std::vector<Edge> start_edges;

    void update_nodes_init(size_t start, size_t end);
    void update_wind_tr(MathVector& wv, size_t start, size_t end,
    std::vector<Vertex>& v, std::vector<Triangle>& tr);
    void update_edges(size_t start, size_t end, std::vector<Vertex>& v);
    void update_nodes(int msec, size_t start, size_t end,
    std::vector<Vertex>& v);
```

```
};
```

Листинг 3.7: Функции аффинных преобразований

```
void Vertex::shift(double dx, double dy, double dz)
{
 _x += dx;
 _y += dy;
 _z += dz;
}

void Vertex::scale(double kx, double ky, double kz, Vertex pc)
{
 double px = pc.x(), py = pc.y(), pz = pc.y();
 _x = px + (_x - px) * kx;
 _y = py + (_y - py) * ky;
 _z = pz + (_z - pz) * kz;
}

void Vertex::rotate_x(double angle, Vertex pc)
{
 double py = pc.y(), pz = pc.z();
 double dy = _y - py, dz = _z - pz;
 _y = py + dy * cos(angle) - dz * sin(angle);
 _z = pz + dy * sin(angle) + dz * cos(angle);
}

void Vertex::rotate_y(double angle, Vertex pc)
{
 double pz = pc.z(), px = pc.x();
 double dz = _z - pz, dx = _x - px;
 _z = pz + dz * cos(angle) - dx * sin(angle);
 _x = px + dz * sin(angle) + dx * cos(angle);
}

void Vertex::rotate_z(double angle, Vertex pc)
{
 double px = pc.x(), py = pc.y();
 double dx = _x - px, dy = _y - py;
 _x = px + dx * cos(angle) - dy * sin(angle);
 _y = py - dx * sin(angle) + dy * cos(angle);
}
```

Листинг 3.8: Функция отрисовки грани

```
void ModelDrawer::draw_triangle(std::vector<Vertex> v,
                std::vector<MathVector> n,
                MathVector l, QColor colour, bool outline)
{
 if (v[0].y() == v[1].y() && v[0].y() == v[2].y())
   return;

 double intens[3];
 for (int i = 0; i < 3; i++)
```

```
10    intens[i] = abs(n[i] & l);
11
12  if (v[0].y() > v[1].y())
13  {
14    std::swap(v[0], v[1]);
15    std::swap(intens[0], intens[1]);
16  }
17
18  if (v[0].y() > v[2].y())
19  {
20    std::swap(v[0], v[2]);
21    std::swap(intens[0], intens[2]);
22  }
23
24  if (v[1].y() > v[2].y())
25  {
26    std::swap(v[1], v[2]);
27    std::swap(intens[1], intens[2]);
28  }
29
30  Vertex v_zero(0, 0, 0);
31  MathVector vec[] = { MathVector(v_zero, v[0]),
32  MathVector(v_zero, v[1]),
33  MathVector(v_zero, v[2]) };
34
35  double dy_0_2 = v[2].y() - v[0].y();
36  double dy_0_1 = v[1].y() - v[0].y();
37  double dy_1_2 = v[2].y() - v[1].y();
38
39  double ia_0 = intens[0] / dy_0_1;
40  double ia_1 = intens[1] / dy_0_1;
41  double ib_0 = intens[0] / dy_0_2;
42  double ib_2 = intens[2] / dy_0_2;
43
44  for (int i = 0; i < dy_0_2; i++)
45  {
46    bool second_half = (i > dy_0_1) || (v[1].y() == v[0].y());
47    double seg_height = second_half ? dy_1_2 : dy_0_1;
48
49    double alpha = i / dy_0_2;
50    double beta = 1.0;
51    if (second_half)
52      beta = (i - dy_0_1) / seg_height;
53    else
54      beta = i / seg_height;
55
56    MathVector A = vec[0] + alpha * (vec[2] - vec[0]);
57    MathVector B;
58    if (second_half)
59      B = vec[1] + beta * (vec[2] - vec[1]);
60    else
61      B = vec[0] + beta * (vec[1] - vec[0]);
62
```

```
63    if (A.x() > B.x())
64      std::swap(A, B);
65
66    double dx = B.x() - A.x();
67    double ddx = (dx == 0) ? 0.0 : 1 / dx;
68    double ia = (dx == 0) ? 0.0 : (ia_0 * (v[1].y() - A.y()) +
69    ia_1 * (A.y() - v[0].y())) / dx;
70    double ib = (dx == 0) ? 0.0 : (ib_0 * (v[2].y() - B.y()) +
71    ib_2 * (B.y() - v[0].y())) / dx;
72
73    for (int x = 0; x <= dx; x++)
74    {
75      MathVector P = A + x * ddx * (B - A);
76      double intens_p = ia * (B.x() - P.x()) + ib * (P.x() - A.x());
77
78      int idx = round(P.x());
79      int idy = round(P.y());
80
81      if (idx >= 0 && idx < IMG_SIZE && idy >= 0 && idy < IMG_SIZE &&
82        z_buf[idx][idy] < P.z())
83      {
84        z_buf[idx][idy] = int(round(P.z()));
85
86        int r = round(colour.red() * abs(intens_p));
87        int g = round(colour.green() * abs(intens_p));
88        int b = round(colour.blue() * abs(intens_p));
89
90        if (r > 255)
91          r = 255;
92        if (g > 255)
93          g = 255;
94        if (b > 255)
95          b = 255;
96
97        c_buf[idx][idy] = QColor(r, g, b);
98      }
99    }
100 }
101
102 if (outline)
103 {
104   colour = colour.darker(120);
105   draw_edge(v[0], v[1], colour);
106   draw_edge(v[1], v[2], colour);
107   draw_edge(v[0], v[2], colour);
108 }
109 }
```

Листинг 3.9: Функция отрисовки сцены

```
void ModelDrawer::draw_scene(std::vector<Model> models, Camera& cam)
{
3  QImage &img = canvas->get_image();
4  QColor bg_color(180, 240, 240);
```

```
5  img.fill(bg_color);
6
7  if (models.size() == 0)
8    return;
9
10 for (int i = 0; i < IMG_SIZE; i++)
11 {
12   for (int j = 0; j < IMG_SIZE; j++)
13   {
14     z_buf[i][j] = -INT_MAX;
15     c_buf[i][j] = bg_color;
16   }
17 }
18
19 Vertex light_pos = light;
20
21 light_pos.shift(cam.get_shift_params());
22 light_pos.scale(cam.get_scale_params(), cam.get_center());
23 light_pos.rotate(cam.get_rotate_params(), cam.get_center());
24
25 light_pos.shift(canv_shift);
26 light_pos.rotate(canv_rotate, canv_center);
27
28 std::vector<double>& params = cam.get_rotate_params();
29
30 Vertex v0(0, 0, 0);
31
32 v0.shift(cam.get_shift_params());
33 v0.scale(cam.get_scale_params(), cam.get_center());
34 v0.rotate(cam.get_rotate_params(), cam.get_center());
35
36 v0.shift(canv_shift);
37 v0.rotate(canv_rotate, canv_center);
38
39 MathVector l(v0, light_pos);
40 l.normalize();
41
42 for (auto model : models)
43 {
44   std::vector<Vertex>& vertices = model.get_vertices();
45   std::vector<Triangle>& triangles = model.get_triangles();
46
47   for (size_t i = 0; i < vertices.size(); i++)
48   {
49     vertices[i].shift(canv_shift);
50     vertices[i].rotate(canv_rotate, canv_center);
51   }
52
53   std::vector<MathVector> normals = model.get_normals();
54
55   for (auto tr : triangles)
56   {
57     std::vector<Vertex> v = tr.get_vertices(vertices);
```

```
58      std::vector<MathVector> n = tr.get_normals(normals);
59      draw_triangle(v, n, l, tr.get_color());
60    }
61  }
62
63  size_t di = IMG_SIZE / 4, i = 0;
64
65  std::vector<std::thread> threads;
66  for (size_t k = 0; k < 4; k++, i += di)
67    threads.push_back(std::thread(ModelDrawer::fill_buffer,
68                  this, std::ref(img), i, i + di));
69
70  for (size_t k = 0; k < 4; k++)
71    threads[k].join();
72
73  canvas->repaint();
74 }
```

Листинг 3.10: Функция обновления положения точки флага

```
1  void Flag::update_thread(MathVector wv, int msec)
2  {
3  std::vector<Vertex>& v = model.get_vertices();
4  std::vector<Triangle>& tr = model.get_triangles();
5
6  size_t dn = nodes.size() / 4;
7  std::vector<std::thread> threads_init;
8  for (size_t k = 0, i = 0; k < 4; k++, i += dn)
9    threads_init.push_back(std::thread(Flag::update_nodes_init,
10                 this, i,
11                 (k == 3 ? nodes.size() : i + dn)));
12 for (size_t k = 0; k < 4; k++)
13   threads_init[k].join();
14
15 size_t dt = tr.size() / 4;
16 std::vector<std::thread> threads_tr;
17 for (size_t k = 0, i = 0; k < 4; k++, i += dt)
18   threads_tr.push_back(std::thread(Flag::update_wind_tr,
19                    this, std::ref(wv), i,
20                    (k == 3 ? tr.size() : i + dt),
21                    std::ref(v), std::ref(tr)));
22 for (size_t k = 0; k < 4; k++)
23   threads_tr[k].join();
24
25 size_t de = edges.size() / 4;
26 std::vector<std::thread> threads_edges;
27 for (size_t k = 0, i = 0; k < 4; k++, i += de)
28   threads_edges.push_back(std::thread(Flag::update_edges,
29                       this, i,
30                       (k == 3 ? edges.size() : i + de),
31                       std::ref(v)));
32 for (size_t k = 0; k < 4; k++)
33   threads_edges[k].join();
34
```

```
35 std::vector<std::thread> threads_nodes;
36 for (size_t k = 0, i = 0; k < 4; k++, i += dn)
37   threads_nodes.push_back(std::thread(Flag::update_nodes,
38                         this, msec, i,
39                         (k == 3 ? nodes.size() : i + dn),
40                         std::ref(v)));
41 for (size_t k = 0; k < 4; k++)
42   threads_nodes[k].join();
43 }
```

3.5 Интерфейс программы

3.6 Примеры работы

3.7 Вывод

В данном разделе были рассмотрены требования к разрабатываемому программному обеспечению, средства реализации, использованные в процессе разработки, были приведены структуры данных, листинг кода к каждой из написанной структур данных. Было подробно расписано взаимодействие с интерфейсом и за что каждая часть интерфейса отвечает.

4. Экспериментальный раздел

В данном разделе будет приведено экспериментальное исследование временных затрат разработанного программного обеспечения.

4.1 Исследование скорости работы алгоритма

4.2 Вывод

Заключение

В ходе выполнения курсового проекта было изучено и реализовано моделирование развевающегося на ветру флага. Была определена и описана предметная область работы, рассмотрены существующие алгоритмы для удаления невидимых линий и поверхностей, из которых был выбран наиболее подходящий алгоритм для решения поставленной задачи. Был спроектирован пользовательский интерфейс и было реализовано требуемое программное обеспечение.

В аналитическом разделе был произведен анализ предметной области, алгоритмов, необходимых для решения поставленной задачи. В конструкторском разделе были подробно описаны методы и алгоритмы, было дано описание принципов работы камеры наблюдателя. В технологическом разделе были предъявлены требования к программному обеспечению, написаны средства реализации. Также были перечислены и описаны основные типы и структуры данных, использующиеся в проекте. Приведены листинги структур данных, реализации некоторых функций. Приведен пользовательский интерфейс с подробным описанием отдельных блоков. В экспериментальном разделе было проведено исследование скорости работы программы с двумя разными способами хранения данных.

Литература

[1] Виталий Якин. Рендер изображения [Электронный ресурс] - Режим доступа: https://render.ru/ru/yakin/post/11353, свободный - (17.10.2019)

[2] Роджерс Д. Алгоритмические основы машинной графики: Пер. с англ. – М.:Мир, 1989 - 512 с. ISBN 5-03-000476-9.

[3] Белорусский государственный университет. Машинная графика, Computer Graphics [Электронный ресурс] - Режим доступа: https://bsu.by/Cache/Page/353573.pdf, свободный - (20.11.2019)

[4] Куров А.В. Курс лекций по машинной графике. - М., 2019

[5] Боресков А. В. Программирование компьютерной графики. Современный OpenGL. – М.: ДМК Пресс, 2019. – 372 с. ISBN 978-5-97060-779-4

[6] Вольф Д. OpenGL 4. Язык шейдеров. Книга рецептов / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2015. – 368 с. ISBN 978-5-97060-255-3

[7] Акимов С.В. Введение в анимацию [Электронный ресурс] - Режим доступа: http://structuralist.narod.ru/it/flash/animation.htm, свободный - (Дата обращения: 20.11.2019 г.)

[8] Документация C++ [Электронный ресурс]. - Режим доступа: https://cppreference.com/, свободный. (Дата обращения: 20.11.2019 г.)

[9] Документация Qt [Электронный ресурс]. - Режим доступа: https://doc.qt.io/, свободный. (Дата обращения: 20.11.2019 г.)